POSTER: Automated Code Acceleration Targeting Heterogeneous OpenCL Devices

Heinrich Riebler Gavin Vaz Tobias Kenter Christian Plessl
Department of Computer Science, Paderborn University, 33098 Paderborn, Germany
{firstname.lastname}@uni-paderborn.de

Abstract

Accelerators can offer exceptional performance advantages. However, programmers need to spend considerable efforts on acceleration, without knowing how sustainable the employed programming models, languages and tools are. To tackle this challenge, we propose and demonstrate a new runtime system called HTrOP that is able to automatically generate and execute OpenCL code from sequential CPU code. HTrOP transforms suitable data-parallel loops into independent OpenCL-typical work-items and handles concrete calls to these devices through a mix of library components and application-specific OpenCL host code. Computational hotspots are identified and can be offloaded to different resources (CPU, GPGPU and Xeon Phi). We demonstrate the potential of HTrOP on a broad set of applications and are able to improve the performance by 4.3× on average.

CCS Concepts • **Computing methodologies** → *Parallel programming languages*; • **Computer systems organization** → *Heterogeneous (hybrid) systems*;

Keywords Transparent Acceleration; Runtime System; Multi-Accelerator; OpenCL; LLVM

ACM Reference Format:
Heinrich Riebler Gavin Vaz Tobias Kenter Christian Plessl. 2018. POSTER: Automated Code Acceleration Targeting Heterogeneous OpenCL Devices. In *PPoPP '18: PPoPP '18: 23nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, February 24–28, 2018, Vienna, Austria.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3178487.3178534

1 Introduction

OpenCL provides an open standard interface for parallel computing using task- and data-based parallelism, which can be executed across different devices. This means that by generating OpenCL kernel code (once), one can target multiple accelerators. However, OpenCL poses not only the challenge of extracting hotspots into kernels and optimizing them for the target accelerator architecture, it also involves many tedious adjustments to the remaining host code. Given these challenges, there is a considerable gap between the architectural potential of highly heterogeneous multi-accelerator architectures and their actual adoption and utilization that we aim to overcome with HTrOP.

Our approach builds upon and integrates results from different open-source projects: We consider LLVM bitcode as the input format to HTrOP, on which all optimization, transformation and acceleration steps are performed. The detection of data-parallel loops is based on LLVM's Polly project [2]. Polly uses an abstract mathematical description to detect and model static control flow regions (so-called SCoPs). And finally, we use LLVM's Axtor backend [3] to translate LLVM bitcode into OpenCL kernel code.

Related work has researched SCoP-based hotspot detection and acceleration, but with other programming models and fewer and different devices in the backend. In our own previous work [1], we used OpenMP and vectorization to offload hotspots from a low-power client to a remote server with an Intel Xeon PHI accelerator. With Polly-ACC, Grosser et al. [2] target Nvidia GPUs using CUDA calls from the host CPU and a PTX backend.

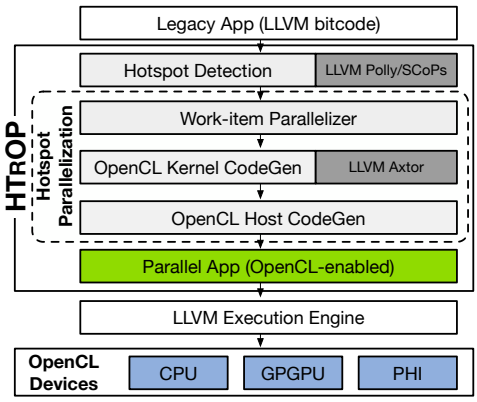

Figure 1. Architecture of the runtime system.

2 Overview and Approach

Figure 1 gives an overview of our approach. Our tool flow receives the legacy application in LLVM bitcode and detects the computational hotspots as SCoPs. These get parallelized and offloaded using three subsequent optimization passes. In

PPoPP '18, February 24–28, 2018, Vienna, Austria

ACM ISBN 978-1-4503-4982-6/18/02.
https://doi.org/10.1145/3178487.3178534

the first transformation step, the *Work-item Parallelizer* uses the dependence analysis information (from Polly) to determine how a loop can be transformed to expose parallelism suitable for OpenCL. Listing 1 shows a simple 2D convolution example in pseudo code. The outer two `for` loops (line 2 and 3) iterate over the entire input `in`. The inner two `for` loops (line 5-6) perform the convolution for each entry. The dependence info reveals that the innermost loops are data dependent. Hence, only the outer two loops are parallized.

```
1 || heavyConv2D(int *in, int *out, int rows, int cols) {
2 ||   for (int r = 0; r < rows; r++) {
  ||     ↳ AFTER Work-item Parallelizer: int r = get_global_id(0);
3 ||     for (int c = 0; c < cols; c++) {
  ||       ↳ AFTER Work-item Parallelizer: int c = get_global_id(1);
4 ||       int sum = 0;
5 ||       for (int i = 0; i < 5; i++) {
6 ||         for (int j = 0; j < 5; j++) {
7 ||           // ...
8 ||           sum += in[r + i][c + j] * COEFFS[i][j];
9 ||       // ...
10 ||      out[r][c] = sum;
```

Listing 1. Nested loops performing a 2D convolution. The two highlighted lines show the modifications performed by the *Work-item Parallelizer* to expose parallelism.

The following steps are performed to expose work-item parallelism in each loop that has no dependencies:

1. Determine the loop induction variable.
2. Remove the loop control flow.
3. Replace the induction variable with a call to the `get_global_id` OpenCL API call.

The induction variable of a loop represents the variable that is incremented/decremented for each iteration (e.g. `r` and `c` in Listing 1). The induction variable can be obtained from the loop header. Once it is found, we find the corresponding compare instruction that checks the loop exit condition. The compare and branch instructions associated with the loop control flow are removed. This effectively removes the loop structure, with all the code previously inside the loop being executed exactly once. The final step is to replace the induction variable with a call to the `get_global_id` OpenCL API call. The lines without line numbers in Listing 1 replace Lines 2 and 3 after the *Work-item Parallelizer* is done.

In the second optimization pass, this modified LLVM bitcode is fed into the Axtor-based *OpenCL Kernel CodeGen* to produce corresponding OpenCL kernel code. Since the legacy application does not originally support OpenCL, the *OpenCL Host CodeGen* updates the application to support all the devices along with the corresponding OpenCL host code to invoke the kernel. We have implemented a wrapper library that creates and exposes device handles for all appropriate OpenCL devices of our evaluation platform to the global scope of the application.

The result is an OpenCL-enabled parallel application that is executed through the LLVM Execution Engine and can offload hotspots to the appropriate OpenCL device.

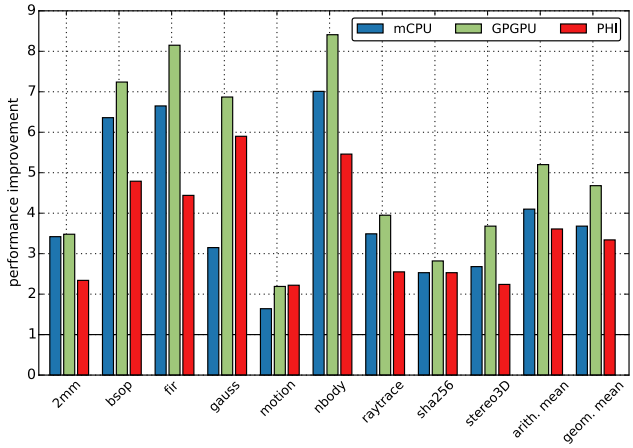

Figure 2. Performance (speedup) of our runtime system (including all overheads) compared to the normalized CPU baseline (= 1).

3 Evaluation

The evaluation was performed on a multi-accelerator platform with 32GB of main memory, consisting of 2 Intel Xeon E5-2609 v2 multithreading CPUs (`mCPU`) each with 4 physical cores, a Nvidia Tesla K20c `GPGPU` and an Intel Xeon `PHI` 31S1P – both connected via PCIe supporting OpenCL v1.2.

We use a set of benchmark applications extracted from scientific computing, financial, signal- and image processing, and security domains. The baseline is singlethreaded CPU code compiled with `gcc` v4.8.2 using the highest optimization level `-O3`. The performance evaluation in Figure 2 reveals speedups for all measured applications with considerable differences between applications and with visible, but small differences among the target devices. We see an average speedup of 4.1x, 5.2x and 3.6x for the `mCPU`, `GPGPU` and `PHI`, respectively, and an overall average speedup of 4.3x.

OpenCL turned out to be an effective vehicle for targeting multiple architectures, allowing us to generate the mechanical parts of the host code and to use the same parallelism pattern for the transformation of computationally intensive regions of the application into accelerator code.

Acknowledgment

This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing" (SFB 901) and the European Union Seventh Framework Programme under grant agreement no. 610996 (SAVE).

References

[1] Marvin Damschen, Heinrich Riebler, Gavin Vaz, and Christian Plessl. 2015. Transparent offloading of computational hotspots from binary code to Xeon Phi. In *Proc. Design, Automation and Test in Europe Conf. (DATE)*. EDA Consortium, 1078–1083.

[2] Tobias Grosser and Torsten Hoefler. 2016. Polly-ACC Transparent compilation to heterogeneous hardware. In *Proceedings of the 2016 International Conference on Supercomputing*. ACM, 1.

[3] Simon Moll. 2011. Decompilation of LLVM IR. *Master's thesis* (2011).