Transparent Acceleration for Heterogeneous Platforms With Compilation to OpenCL

HEINRICH RIEBLER, GAVIN VAZ, TOBIAS KENTER, and CHRISTIAN PLESSL,
Paderborn Center for Parallel Computing (PC²), Paderborn University

Multi-accelerator platforms combine CPUs and different accelerator architectures within a single compute node. Such systems are capable of processing parallel workloads very efficiently while being more energy efficient than regular systems consisting of CPUs only. However, the architectures of such systems are diverse, forcing developers to port applications to each accelerator using different programming languages, models, tools, and compilers. Developers not only require domain-specific knowledge but also need to understand the low-level accelerator details, leading to an increase in the design effort and costs.

To tackle this challenge, we propose a compilation approach and a practical realization called *HTR_{OP}* that is completely transparent to the user. *HTR_{OP}* is able to automatically analyze a sequential CPU application, detect computational hotspots, and generate parallel OpenCL host and kernel code. The potential of *HTR_{OP}* is demonstrated by offloading hotspots to different OpenCL-enabled resources (currently the CPU, the general-purpose GPU, and the manycore Intel Xeon Phi) for a broad set of benchmark applications. We present an in-depth evaluation of our approach in terms of performance gains and energy savings, taking into account all static and dynamic overheads. We are able to achieve speedups and energy savings of up to two orders of magnitude, if an application has sufficient computational intensity, when compared to a natively compiled application.

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

Additional Key Words and Phrases: Transparent acceleration, runtime system, multi-accelerator, OpenCL

ACM Reference format:

Heinrich Riebler, Gavin Vaz, Tobias Kenter, and Christian Plessl. 2019. Transparent Acceleration for Heterogeneous Platforms With Compilation to OpenCL. *ACM Trans. Archit. Code Optim.* 16, 2, Article 14 (April 2019), 26 pages.
<https://doi.org/10.1145/3319423>

This article is an extension of a conference paper/poster at PPoPP'18 [49]. Prior work only outlined the tool flow and the parallelization mechanism. Evaluation is entirely new, including energy measurements and trends over several data points per application.

This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre “On-The-Fly Computing” (SFB 901).

Authors' addresses: H. Riebler, G. Vaz, T. Kenter, and C. Plessl, Department of Computer Science, Paderborn University, Warburger Str. 100, 33098 Paderborn, Germany; emails: {heinrich.riebler, gavin.vaz, tobias.kenter, christian.plessl}@uni-paderborn.de.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

1544-3566/2019/04-ART14

<https://doi.org/10.1145/3319423>

1 INTRODUCTION

Performance and energy challenges drive the ongoing trend to use accelerators. The most prominent type of discrete accelerators today are general-purpose GPUs (GPGPUs). Manycore architectures like Xeon Phi have been introduced as discrete accelerators or stand-alone computing products. Conventional multicore CPUs also mainly scale up by adding more cores, which can be regarded as accelerators when they are used for the parallel execution of performance-critical parts of a specific application. Further architectures like field programmable gate arrays (FPGAs) are starting to receive scientific and commercial attention. Such accelerators and parallel architectures are now prevalent in many computing domains, from personal devices to HPC and cloud computing. For many concrete workloads, regular CPU systems combined with accelerators can reach higher levels of performance while being more energy efficient [1]. Cloud providers profit from economics of scale by deploying hundreds of thousands or millions of identical machines. Thus, in the long run, beneficial accelerators might just show up in every such machine. Given the complementary advantages of different accelerator architectures for many applications [2], this could potentially even lead to multiple different accelerators per machine. First incarnations of such innovative multi-accelerator architectures [55, 60] have already been explored.

However, tools and programmers struggle to keep up with this trend toward heterogeneous systems. Pragma-guided or fully automated parallelizing compilers rely on information about the target architecture to perform their optimizations and often support just a single target platform. In addition, when moving computation (*offloading*) to a discrete accelerator, only computational hotspots that can overcome call and data transfer overheads should be considered. Thus, whenever static code analysis cannot unequivocally identify these hotspots, benchmarking or specific developer knowledge needs to be incorporated. Among the programming models that expose many platform details to the developer, from a functional perspective, OpenCL [58] is the most portable one, supported by multi- and manycore CPUs, GPUs, and recently also by FPGAs [32, 56]. However, OpenCL poses not only the challenge of extracting hotspots into kernels and optimizing them for the target accelerator architecture but 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.

In this work, we propose a compilation and runtime approach to bridge this gap. We present HTrOP, a user-transparent end-to-end approach and practical realization for bringing sequential legacy programs to parallel execution on different accelerator targets via OpenCL. First prototypical results of HTrOP have been introduced in our previous work [49]. Partially building upon and integrating results from different open-source projects (LLVM [38], Polly [20], and Axtor [43]), HTrOP is also publicly available as open source under the MIT license on GitHub [50]. Besides the practical tool integration efforts, our contributions include the following:

- (1) Automatic transformation of suitable data-parallel loops into independent OpenCL-typical work items that are executed in parallel.
- (2) A two-layered approach of identifying hotspots at compile time and refining offloading decisions at runtime based on parameters such as input sizes and availability of accelerators.
- (3) Infrastructure for offloading to and migrating between accelerators while minimizing data transfer overheads by reusing data through application-specific generated code parts.
- (4) A thorough evaluation of performance gains and energy savings with different accelerator targets, taking into account one-time and recurring overheads introduced by our approach. The evaluation includes a comparison to handwritten pragma-based OpenACC code for multicore CPUs and GPUs.

Table 1. Parallelization Approaches, Required Developer Activity, and Target Architectures

	OpenMP	OpenACC	OpenCL	Others	Ours
Parallelization	pragmas	pragmas	kernel + host code	automatic	automatic
Multicore CPU	supported	supported	supported	Polly [20] (OpenMP)	demonstrated
Xeon Phi	supported	—	supported	—	demonstrated
NVidia GPU	experimental	supported	supported	Polly-ACC [21] (NVPTX)	demonstrated
Other GPUs	experimental	supported	supported	—	possible
FPGAs	research	research	supported	—	research

Overall, our results demonstrate the feasibility and the effectiveness of our approach by improving the overall performance and energy efficiency for a wide range of application domains and different accelerator technologies. HTROP can achieve comparable performance to handwritten OpenACC code while being fully automated. We believe that in the domain of parallelization and offloading tools, HTROP is unique in its combination of transparency and flexible support for different target architectures. Table 1 summarizes this with a comparison to the most relevant alternative approaches. Section 2 provides a more detailed discussion on related tools and research. Our evaluation also confirms an interesting observation that may represent a general trend for acceleration in heterogeneous systems: Any observed performance improvements largely go hand-in-hand with similar energy savings despite very different power profiles of the target accelerators. A particular quality of our OpenCL-based approach is that it treats additional CPU cores beyond the first one essentially like any discrete accelerator, which can greatly facilitate resource allocation and utilization in a dynamic, heterogeneous cloud context.

The remainder of this work is structured as follows. In Section 2, we present background information for LLVM and related work in the area of accelerator programming and transparent acceleration. Section 3 gives an overview of the architecture and tool flow of HTROP. In Section 4, we describe its components and the actual implementation. In Section 5, we measure the performance and energy efficiency for a set of benchmark applications, give in-depth details of the overheads, and discuss the generality and limitations. Finally, we outline future work and conclude in Section 6.

2 BACKGROUND AND RELATED WORK

In this section, we give background information and outline related work and concepts in the area of accelerator programming, the LLVM compiler infrastructure, and transparent acceleration.

2.1 Accelerator Programming

Architectures with at least one accelerator attached to the system are becoming omnipresent. However, the implementation of heterogeneous applications has many difficulties that limit their widespread adoption. The developer has to identify the regions that benefit from execution on an accelerator. This analysis is done by offline profiling tools to determine hotspots. Then, the hotspot has to be manually adapted for the particular accelerator or programming model. This is a time-consuming and error-prone process, as each model has its own APIs and levels of abstraction, whereas each accelerator has a different architecture. OpenCL [58] provides an open standard interface for parallel computing using task- and data-based parallelism, which can be executed across different heterogeneous devices consisting of CPUs, GPUs, Intel Xeon Phis, and, more recently, FPGAs. OpenCL specifies unified APIs to control the device, transfer and handle the data, and execute the hotspots (OpenCL kernels). OpenCL is not only highly relevant in the

HPC context but particularly interesting for mobile devices and embedded platforms. SYCL [22] is an abstraction layer and open standard that builds on top of OpenCL to enable single-source code for heterogeneous devices. The time-intensive work of writing accelerator code needs only to be carried out once, and a compiler (e.g., the ComputeCpp [7]) generates the executables for different devices. This reduces the potential of errors and the effort porting the code for each accelerator. However, the initial host and device codes need to be manually written in C++. CUDA [44] is another parallel computing model with a mature but proprietary ecosystem created for Nvidia GPUs. Similar to OpenCL, a developer has precise control of the accelerator, which implies that they have to write most of the accelerator code from scratch. However, developers can use pragma- or task-based APIs such as OpenMP [9], OpenACC [30], Cilk [52], or TBB [36] to annotate regions (with directives or additional functions) that should be accelerated. Instead of writing accelerator code from scratch, the compiler takes care of the thread and data management, freeing the user from low-level details. Pragmas can be ignored by non-conforming compilers or runtime environments. Thus, when code with pragmas meets the right environment, it is executed with acceleration and otherwise without. With our approach, any code that meets the execution environment parameters can be accelerated.

2.2 Compilation and Analysis Frameworks

LLVM is a compiler infrastructure [38] and plays a vital role in our approach of automated acceleration. A wide range of programming languages are supported by LLVM. Frontends (clang for C/C++, flang for Fortran, etc.) translate source files from their programming-specific description into language-independent intermediate representation (IR) or *LLVM bitcode*. We consider LLVM bitcode as the input format to HTROP, on which all optimization, transformation, and acceleration steps are performed. LLVM bitcode can be considered as a binary representation, because it is possible to disassemble and translate x86 instructions to equivalent LLVM instructions [13, 48].

Within LLVM, HTROP uses the Execution Engine [26], which is a library for running LLVM bitcode. To identify the hotspots of the input application, our approach uses the LLVM-based tool Polly [20]. Polly is a polyhedral loop optimizer that uses an abstract mathematical description to detect and model static control flow regions (so-called SCoPs [4, 16]) for (nested) loops of an application. The control flow of a SCoP has no BREAKs or GOTOs, and the execution count of a loop must be computable by an affine function. However, this does not mean that the execution count must be known at compile time, but rather that it does not change during the execution. The loops detected by Polly can have cross-iteration dependencies, if they do not violate the preceding restrictions. Our approach only parallelizes loops without cross-iteration dependencies, which is a subset of the loops detected and parallelized by Polly. In the case of cross-iteration dependencies, the code generation would still be correct but not parallel. For each detected hotspot, HTROP can query Polly's SCoP info [46] to get detailed data dependency analysis and memory access pattern of the corresponding loop. Polly has the ability to use OpenMP pragmas to emit thread-parallel code for the detected SCoPs. And last, HTROP relies on the LLVM project called *Axtor* [43]. Axtor is a backend for LLVM, which can translate LLVM bitcode into OpenCL kernel code. Therefore, it restructures the control flow without the need for GOTO statements (or similar constructs). This is an essential requirement to emit strict control flow structured OpenCL code. Unfortunately, the initial efforts were not continued, and the source code has been removed from the LLVM tree due to lack of maintenance. Magni et al. [41] revived and adapted Axtor to LLVM 3.5 for their thread-coarsening approach. We used this version and ported it to LLVM 3.8.0.

2.3 Transparent Acceleration

Techniques from runtime resource management [51] allow a scheduler to change the resource allocation of a hotspot at runtime and even to interrupt the execution on one device and migrate it to another. However, to achieve this, a deep manual reconstruction of the applications is required and the hotspots must be known *a priori*. In contrast, the process of transparent acceleration tries to improve application characteristics by automatically dispatching or offloading computations to another resource—without any developer effort or intervention.

Several prior works from the embedded systems and HPC domain have tackled the problem of (semi-) automatically splitting threads of a single data-parallel kernel across multiple, heterogeneous devices. Tool flows from the different domains typically share the same tasks on the software side but tackle other types of accelerators and have often a stronger focus on either energy efficiency or performance. Verdoolaege et al. [63] developed a technique to parallelize loops using Kahn process networks (KPNs), a predecessor of the polyhedral model used in our work with the help of Polly. KPNs fit very well on MPSoCs (multi-processor systems-on-chip), because they can be scheduled independently and are well suited for quantitative performance analysis. All loops must be expressible by a static affine function, which is only a subset of all loops in real-world applications, similar to the restriction of the polyhedral model used in Polly. Ceng et al. [5] present a parallelization tool flow for MPSoCs that translates the application into a control data-flow graph, which is gradually clustered into independent tasks. In contrast to our work, this approach is only semi-automatic, because user feedback is required to control the level of granularity of the clusters. Yang et al. [65] present an energy-aware two-phase scheduling scheme of tasks for a MPSoCs running two processors at distinct voltage levels. Different scheduling options are predefined at compile time and dynamically selected at runtime to save energy while meeting the timing constraints. We follow a similar two-phase approach. At compile time, we augment the application with a runtime decision that enables offloading, and at runtime, we dynamically select the best strategy based on the different criteria. Taylor et al. [59] use machine learning techniques to help select the optimal computing device for OpenCL programs on embedded heterogeneous systems (big.LITTLE with Mali GPU). The predictive model is trained offline with features extracted from LLVM bitcode. A similar approach can be combined with our framework for smarter device selection, which is currently configurable, but uses static thresholds (see Section 4.4.1). However, successfully training the optimal device selection and using the information for a real run depends on the accuracy of the training to the real execution, and the training needs to be repeated for each new device. Therefore, Kaleem et al. [33] tackle the same problem with online profiling, which is agnostic to the application and the platform but adds a runtime overhead.

Kim et al. [35] present an OpenCL framework that basically virtualizes multiple GPUs in a single node as one compute device. Data-parallel applications are written for one GPU (with OpenCL), and the framework automatically distributes the work across all GPUs. The memory footprints must follow a regular pattern and are detected by sampling traces at runtime on the CPU. Similar to our work, the framework is also transparent to the user. However, it requires manual effort to write a full OpenCL application in the first place and only tackles NVidia GPUs. Sabne et al. [54] study the portability (functional and performance) of OpenACC programs across different accelerators (NVidia + AMD GPUs, Intel Xeon Phi). Similar to HTROP, they propose a single input program (with existing OpenACC pragmas) to tackle multiple accelerators. In addition, they apply further optimizations to exploit architecture-specific features (instructions, memory, etc.). Spafford et al. [57] developed Maestro, a runtime library on top of OpenCL for automatic data transfer and decomposition of independent tasks across multiple OpenCL devices. Instead of having one task queue per device, they propose a single high-level task queue. One strong focus of the work is

Fig. 1. Overview of the architecture and the tool flow.

dedicated to data transfers—for example, they try to overlap communication and computation to hide offloading latencies. With Maestro, the programming effort for the developer is simplified, but in contrast to our work, the approach is not transparent to the user. Diamos et al. [12] present Ocelot, a compilation framework that can map CUDA binaries (in PTX format) onto multicore platforms. Similar to HTROP, the input is translated into LLVM IR, and then the analysis, optimizations, and transformations are performed with the help of LLVM. In contrast, Ocelot does not utilize OpenCL but uses built-in LLVM backends to generate GPU/CPU code. The CPUs are targeted by PTX emulation. In the embedded systems domain, similar work flows have been examined. Damschen et al. [11] describe an LLVM-based client-server architecture called *BAAR* where hotspots are also detected as SCoPs and offloaded at runtime to an Intel Xeon Phi. In the envisioned architecture, the client machine only consists of a low-power CPU, and the server has the Phi attached to it (loosely coupled). To accelerate a SCoP, the client issues a remote procedure call (RPC) with the LLVM bitcode and sends the function arguments via MPI [19] to the server. The server generates code and executes the request. To utilize the important vector processing units and manycore processors of the Phi, the code is vectorized using the loop- and superword-level parallelism (SLP) provided by LLVM [24]. The sequential code is parallelized with OpenMP to generate sufficient computing threads. However, this approach has a number of drawbacks: Targeting OpenMP instead of OpenCL limits this approach to a smaller number of heterogeneous devices, the server can easily become a bottleneck for multiple clients, and the RPC calls can cause additional data transfer overheads. In our work, the execution environment is tightly coupled to the accelerators, and the code is generated by a centralized service. Grosser and Hoefler [21] developed Polly-ACC, a heterogeneous compiler to generate hybrid executables (CPU + GPU) prior to execution. An elaborate data management library is linked against the final executable to ensure that the data is available when and where needed. Similar to BAAR, the hotspot is identified as a SCoP. Polly-ACC uses the scheduling algorithm from Pluto [3] and the mapping strategy from PPCG [62] to partition the application. To generate GPU code, Polly-ACC leverages LLVM’s NVPTX [29] backend for Nvidia GPUs. Kalms et al. [34] reuse Polly-ACC but call the OpenCL code generation feature of PPCG to generate host and kernel code. However, neither OpenCL nor the heterogeneity of a system is described in their work. The main focus of the work is an LLVM IR to C code emitter (similar to the C backend used in BAAR), which decouples Polly and PPCG to emit intermediate results.

3 OVERVIEW AND APPROACH

In this section, we give an overview of the architecture of HTROP and introduce the corresponding tool flow. In our approach, we aim to automatically and transparently accelerate single-threaded applications (legacy code) by detecting and offloading the hotspots to accelerators. OpenCL provides standard APIs that can be used across all supported devices. This means that by generating OpenCL kernel code (once), we can target multiple accelerators. This reduces the code generation effort required for different accelerators and makes it a good candidate for our approach. Figure 1 shows the architecture of the HTROP. HTROP accepts the platform-independent LLVM bitcode ① of the legacy application. The bitcode is parsed into an in-memory representation and analyzed to detect application hotspot candidates (as SCoPs) ②. A code generation decision (CGD) ③ is made

depending on whether the detected hotspot candidate has enough potential to be offloaded with benefits (depending on the number of instructions, the instruction mix, etc.). Once this decision is made, a message is sent to the Parallelization Engine ④ asking it to generate OpenCL kernel code. The Parallelization Engine first transforms the hotspot into a form where parallelism is exposed ⑤, then makes use of the LLVM OpenCL backend Axtor to generate the OpenCL kernel code ⑥ and sends the OpenCL kernel code back to HTROP ⑦. Once the OpenCL kernel code is available, the application is updated by adding OpenCL device handles and OpenCL wrapper functions to it. Additionally, all calls to the original hotspot are replaced with calls to a control function that performs the runtime offloading decision ⑧. This runtime control function decides where the hotspot needs to be offloaded to or if it even needs to be offloaded at all. This decision depends on a number of factors, such as loop execution count, data sizes, available accelerators, and overhead amortization. As the penultimate step, OpenCL host code is generated ⑨. This involves updating the wrapper functions with OpenCL API calls that create buffers on the device, transfer the data to and from the device, map the kernel arguments, and invoke the kernel. The final outcome is the accelerated OpenCL-enabled application ⑩. In case the CGD decides not to accelerate the code, ④ through ⑨ are skipped and the code is directly compiled like a regular application, which reduces compilation overheads. HTROP is designed in a modular fashion, allowing us to switch out components for future versions. The modular approach allows us to change the hotspot detection logic ② or code generation mechanism ⑥ with reasonable effort.

4 TOOL FLOW AND REALIZATION

In this section, we take an in-depth look at the inner workings of HTROP. The description will follow the flow of Figure 1 with a focus on the actual realization and implementation decisions.

4.1 Hotspot Detection Module

Legacy applications can be translated into LLVM bitcode ① using different frontends (see Section 2.2) resulting in differently structured inputs for HTROP. Programming languages allow the developer to program the same task in multiple ways. This results in a combinatorial explosion of patterns, and recognizing all patterns is infeasible. Hence, HTROP canonicalizes the input with the help of built-in LLVM passes [23] to normalize the structure, to expose parallelism, and to improve hotspot detection. Additionally, LLVM code optimization passes such as memory to register promotion, loop simplification, normalization of induction variables, loop invariant code motion, constant propagation, combining redundant instructions, and dead code elimination are run. At a high level, loops are good candidates for hotspots. Thus, the first step after canonicalization is to identify them. HTROP uses LLVM's Polly project (see Section 2.2) in the Hotspot Detection Module ② for this task. Polly provides fine-grain information about possible hotspot regions, their data access patterns, and data dependencies—in our terminology, a SCoP is a hotspot candidate.

4.2 Code Generation Decision

After executing the Hotspot Detection Module, HTROP scores ③ each detected hotspot candidate to decide if the SCoP contains enough computation to benefit from offloading at all. Code for the candidate is only generated if its score is high enough. The score is computed as a weighted summation of the floating-point/integer operations execution counts. Different kinds of floating-point/integer operations have different, configurable weights:

$$\text{score_hotspot_candidate}(\text{SCoP } S) = \sum_{instr \in S} \text{weight}(instr) \cdot \text{iterationCount_loopNest}(S)$$

The loop iterations are computed with the help of LLVM's ScalarEvolution and LoopInfo passes. If the loop iteration count for a loop level cannot be determined at compile time, it is set to 1. Hotspot candidates whose score exceed a user-defined threshold γ (by default, γ is set to 0) are considered as hotspots and extracted into a separate function. The default threshold of zero accepts all SCoPs that are executed at least once, whereas a higher threshold can be used to exclude unwanted candidates (e.g., initialization routines). For the evaluated benchmark applications, a minimum score of 16 is required to include them all. More details on the decision process can be found in our previous work [10, 11, 61]. After the CGD, HTROP requests the Parallelization Engine to generate the OpenCL kernel ④.

4.3 Generating Parallel Kernel Code

Now that the hotspot is identified, the corresponding OpenCL kernel code can be generated with the help of the Parallelization Engine ⑤ and ⑥. HTROP sends the Parallelization Engine a code generation message. With this information, the Parallelization Engine first tries to directly serve the request from a cache. By serving all instances of HTROP, the Parallelization Engine can enhance reuse of already processed requests. If the request is not cached, it requests further information regarding the corresponding hotspot function (the LLVM bytecode, analysis information, etc.). Once the Parallelization Engine receives this information, it generates the OpenCL kernel code (*.cl-file) and sends it back to HTROP ⑦.

The code generation module ⑥ currently uses LLVM's Axtor backend (see Section 2.2) to generate OpenCL kernel code. Axtor is capable of generating OpenCL C kernel code from LLVM bytecode; however, this code is *not* automatically parallelized. This implies that the kernel code would be executed sequentially (single threaded) on the accelerator. Therefore, HTROP first parallelizes the LLVM bytecode ⑤ before handing it to Axtor. Conceptually, our approach exposes parallelism in OpenCL by splitting data-parallel loops into independent *work items* that can be processed in any order by multiple compute units (CU). Each CU is made up of many processing elements that execute the same OpenCL kernel code but work on a different work item. The `get_global_id()` OpenCL API call returns the unique global work-item ID that the kernel needs to work on. In a sequential program, work is performed one by one in a loop, with the loop iterator specifying which work item is currently being executed. To be able to transform a sequential loop into a parallelized OpenCL kernel, HTROP needs to replace the loop structure with a `get_global_id()` OpenCL API call. This step builds upon information obtained from Polly in the form of dependence analysis that either allows or prevents the transformation of a loop or loop level in the case of nested loops.

Listing 1 shows a simplified 2D 5×5 convolution in pseudocode. The outer two loops (lines 3 and 4) iterate over the 2D input `in`. The inner two loops (lines 7 and 8) perform a 5×5 convolution for each entry. The DependenceInfo from Polly (Hotspot Detection Module) tells HTROP that the innermost loops are data dependent. Hence, only the outer two loops are parallelized. This is done by performing the following steps on the LLVM bytecode for each loop that has no dependencies:

- (1) Determine the loop induction variable.
- (2) Remove the control flow for the loop.
- (3) Replace the induction variable with a call to the `get_global_id()` OpenCL API call.
- (4) Information about the loop iteration space will be used in the host code to generate the appropriate work items.

The induction variable of a loop represents the variable that is incremented/decremented for each loop iteration. In our example in Listing 1, `r` and `c` are the induction variables of the outer loops. Since the LLVM bytecode is canonicalized (see Section 4.1), the loop induction variable is represented as a PHINode instruction in the loop header. The PHINode belongs to every loop control


```

1 || conv2D(int *in, int *out, int rows, int cols)
2 || // Iterate through 2D input.
3 || for (int r = 0; r < rows - 4; r++) {
4 ||     for (int c = 0; c < cols - 4; c++) {
5 ||         // Convolve over 5x5 filter.
6 ||         int sum = 0;
7 ||         for (int i = 0; i < 5; i++) {
8 ||             for (int j = 0; j < 5; j++) {
9 ||                 // ...
10 ||                 sum += in[r+i][c+j] * COEFFS[i][j];
11 ||                 // ...
12 ||                 out[r][c] = sum;

```

Listing 1. Nested loop performing a 2D 5×5 convolution.

```

1 || conv2D(int *in, int *out, int rows, int cols)
2 || // Iterate through 2D input.
3 || int r = get_global_id(0);
4 || int c = get_global_id(1);
5 || // Convolve over 5x5 filter.
6 || int sum = 0;
7 || for (int i = 0; i < 5; i++) {
8 ||     for (int j = 0; j < 5; j++) {
9 ||         // ...
10 ||         sum += in[r+i][c+j] * COEFFS[i][j];
11 ||         // ...
12 ||         out[r][c] = sum;

```

Listing 2. Same convolution with implicit work-item parallelism.

flow and is used to select a value depending on the predecessor block. Once the induction variable is found, HTROP looks for the corresponding ICmpInst 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. Listing 2 shows the transformed pseudocode after all the steps are completed (loops in lines 3 and 4 are replaced). The modified LLVM bytecode is fed into the Axtor backend to produce corresponding OpenCL kernel code. Afterward, the code is written to a *.cl-file and sent back. Please note that this is the fundamental requirement to be able to distribute work across an OpenCL device. This kernel code is currently identical for all offloading devices that are evaluated in this work. As later discussed in Section 5.3, one could also apply accelerator-specific guidance and other optimizations (loop unrolling, tiling, etc.) to generate more efficient code or exploit custom units (e.g., vector data types) or take advantage of memory reuse.

4.4 Integrating Host Code and Runtime Into Application

The generated parallel OpenCL kernel code represents only the raw computation on a compute device. Corresponding OpenCL host code is required to set up the device, compile the kernel code, handle data buffers, and enqueue the kernel. Additionally, control code that dynamically selects the most suitable accelerator (based on runtime parameters) is also required. In this section, we describe how HTROP automatically generates the OpenCL host code and integrates the runtime decision mechanism into the application, to dynamically select between devices.

The following steps outline the tasks performed by HTROP to add acceleration support and dynamic runtime decisions to a legacy application ① by altering the control flow ⑧ and generating OpenCL host code ⑨:

- (1) Create control and wrapper functions to enable dynamic runtime decisions.
- (2) Create OpenCL device handles and command queues to access the target devices.
- (3) Create required data buffers to enable transfers of data to and from the devices.
- (4) Set up the kernel arguments and enqueue the kernel for execution.
- (5) Transfer data from the device memory to the host memory and release all the buffers.

Please note that all of these steps are automatically added by HTROP at an LLVM instruction level using the IRBuilder [27]. For simplicity, the methods are described at a higher abstraction level.

4.4.1 Control and Wrapper Function. To simplify the addition of the described OpenCL host code for different hotspots and accelerators, HTROP first constructs wrapper functions for every

(hotspot $\times$ device) pair. The OpenCL wrapper functions encapsulate the code for data allocation, data transfers, kernel calls, and synchronization. Each wrapper is created by inserting an empty function with the same parameters and return type as the original sequential hotspot function. After the OpenCL wrapper functions are created, the application needs to be updated with calls to the newly created wrappers. Instead of directly replacing all calls to the sequential hotspot with calls to a device-specific OpenCL wrapper function, HTROP introduces an interjacent control function. The control function is able to select between the original sequential hotspot or the OpenCL wrapper functions based on the runtime offloading decision.

Runtime offloading decisions. The control function with the runtime decision is inserted in step ⑧ in the overall tool flow by replacing all calls to the hotspot with calls to the corresponding control function. This enables our approach to switch between accelerators or to run the original sequential function. Conceptually, the inserted code block for offloading decisions complements the earlier CGD ③ as a second filter that can make use of additional information available at runtime (e.g., data sizes or accelerator availability). HTROP selects the most-promising accelerator by using a dynamic runtime decision model with configurable weights. This is similar to the scoring method for hotspot candidates in Section 4.2. The runtime decision considers the following parameters, which are evaluated in Section 5.2.5:

- *Size of input and output data:* Small data sizes cannot amortize the offloading overheads (data transfers and kernel compilation) to loosely coupled accelerators. Hence, small sizes favor the sequential function, whereas large sizes favor the accelerators.
- *Availability and reuse:* Unavailable devices get a penalty. Reusing the same accelerator for successive hotspots is preferred, because the OpenCL kernel code for this device is already compiled and data can be left on the device to avoid data transfers. Our runtime keeps track of the used buffers on each device. The mechanism is described in Section 4.4.3.

4.4.2 OpenCL Device Handles. The legacy application does not natively support OpenCL. This means that it lacks any OpenCL API headers, has no handles to OpenCL devices, and is not linked against an OpenCL library. Before adding the OpenCL host code, HTROP needs to add OpenCL support to the legacy application. We have implemented an OpenCL wrapper library that adds common OpenCL API headers and creates OpenCL device handles and command queues for all the OpenCL devices of our evaluation platform (described in Section 5). This step is only performed once, as the device handles and command queues are reused. We execute the OpenCL-enabled parallel application ⑩ through the LLVM Execution Engine (see Section 2.2) that is linked against the OpenCL library. Hence, the inserted OpenCL API calls can be dynamically resolved.

Augmenting the legacy application with general OpenCL support (device handles, headers, etc.) only adds minor overheads to the application execution time. However, *using* the OpenCL runtime adds considerable overheads that need to be amortized. Data transfers to and from the devices are obvious ones, but the compilation of the OpenCL kernels by the device drivers also has a strong impact (see evaluation of runtime overheads in Section 5.2.3). To minimize the OpenCL compilation overheads, HTROP performs two optimizations. First, HTROP waits for the runtime decision to select a target accelerator and only then compiles the kernel for that specific accelerator (call-by-need or *lazy* compilation) instead of compiling the OpenCL kernel for all accelerators at the start of the application. If the runtime decision decides to execute the application sequentially (small input size or busy accelerators), the OpenCL kernel compilation, if it has not already been done, is skipped and the application is executed on the CPU with minimal overheads introduced by HTROP. Second, in the case of applications with multiple hotspots (with multiple runtime decisions), HTROP tries to hide the OpenCL compilation overhead by executing the application and

```

1 || Function: conv2D
2 || // ...
3 || Invariant Accesses: {
4 ||   // (A) Read access to 25 elements of COEFFS array.
5 ||   ReadAccess := [rows, cols] -> { Stmt[r, c] -> COEFFS[0][0] }
6 ||   ReadAccess := [rows, cols] -> { Stmt[r, c] -> COEFFS[0][1] }
7 ||   // ...
8 ||   ReadAccess := [rows, cols] -> { Stmt[r, c] -> COEFFS[4][4] }
9 || }
10 || Statements {
11 ||   // (B) Boundary conditions for the induction variables.
12 ||   Domain := [rows, cols] -> { Stmt[r, c] :
13 ||     r >= 0 and r <= rows - 5 and c >= 0 and c <= cols - 5 }
14 ||   // (C) Fine-grained information on the access locations.
15 ||   ReadAccess := [rows, cols] -> { Stmt[r, c] -> in[r, c] }
16 ||   ReadAccess := [rows, cols] -> { Stmt[r, c] -> in[r, c + 1] }
17 ||   // ...
18 ||   ReadAccess := [rows, cols] -> { Stmt[r, c] -> in[r + 4, c + 4] }
19 ||   MustWriteAccess := [rows, cols] -> { Stmt[r, c] -> out[r, c] }
20 || }

```

Listing 3. Simplified SCoP information for the 2D 5×5 convolution example.

simultaneously compiling the OpenCL kernels for different accelerators in separate threads that are then available for later use.

4.4.3 Data Buffers and Optimizations. The OpenCL kernel works on data that needs to be present on the device. Before the data can be transferred, HTRoP needs to allocate corresponding buffers of the right data type, size, and buffer type (OpenCL memory flag). A naive approach would be to allocate and transfer all the data from the host to the device and back. Instead, by transferring only the data that is required by the computing kernel and transferring only data back that is required by the host, the data transfer times can be optimized. However, determining which portion of the data needs to be present is a challenging task. To tackle this problem, our approach uses the fine-grain information about the data access locations in the Hotspot Detection Module provided by Polly (see Section 4.1). Listing 3 shows the simplified information (as a SCoP) that we receive for the 2D 5×5 convolution example in Listing 1.

By analyzing the information, HTRoP can either directly determine which data is required by the kernel (e.g., the conv2D kernel reads 25 elements from `COEFFS`, see (A) in lines 5–8) or how the required data can be determined at runtime (e.g., the required size for `in` and `out` depend on the values of `rows` and `cols`, see (B) and (C) in lines 12–19). To resolve the runtime dependent values, our approach models them similar to Scalar Evolution Expressions in LLVM [28]. Scalar Evolution is used to precisely describe the expected runtime value for evolving (affine) induction variables in loops. The value is described by a mathematical expression that defines an equation that represents the actual runtime value that the variable will hold during the application execution. In our example from Listing 1, we get a lower and upper bound for the induction variables `r` and `c` in lines 12 and 13 that serve as the access indices for the runtime dependent arrays `in` and `out`. From these bounds, HTRoP computes the minimum and maximum *access locations* and hence the required buffer size. The minimum access location for array `in` is in line 15 and straightforward to derive, because variables `r` and `c` are bound to be greater or equal to 0. The same holds for array `out`, which is bound in line 19. When we substitute the induction variables `r` and `s` with the minimum bounds, we get the following:

$$\min : \begin{cases} \text{in}[\mathbf{r}][\mathbf{c}] \rightarrow \text{in}[0][0], \\ \text{out}[\mathbf{r}][\mathbf{c}] \rightarrow \text{out}[0][0] \end{cases}$$

The maximum access location for array `in` is in line 18 and in line 19 for array `out`. They are slightly more complicated to derive because they depend on the runtime values of `rows` and `cols` reduced by the size of the 5×5 convolution matrix at this border. When we substitute the induction variables `r` and `s` with their maximum bounds ($r \leq \text{rows}-5$ and $c \leq \text{cols}-5$), we get the following:

$$\text{max} : \begin{cases} \text{in}[r+4][c+4] \rightarrow \text{in}[(\text{rows}-5)+4][(\text{cols}-5)+4] \rightarrow \text{in}[\text{rows}-1][\text{cols}-1], \\ \text{out}[r][c] \rightarrow \text{out}[\text{rows}-5][\text{cols}-5] \end{cases}$$

Based on the access locations, HTROP computes the required portion of data that needs to be allocated and transferred to/from the device.

$$\text{size} : \begin{cases} \text{in}[\text{rows}-1+1][\text{cols}-1+1] \rightarrow \text{in}[\text{rows}][\text{cols}], \\ \text{out}[\text{rows}-5+1][\text{cols}-5+1] \rightarrow \text{out}[\text{rows}-4][\text{cols}-4] \end{cases}$$

For the first dimension of array `in`, a size of `rows - 1 + 1 = rows` is required (`rows - 1` comes from the maximum location and the `+1` is required to allocate one element for the memory access at index 0, the minimum). Vice versa, the second dimension of array `in` requires a size of `cols - 1 + 1 = cols`. And applying the same rules for array `out`, the sizes are `[rows - 5 + 1 = rows - 4][cols - 5 + 1 = cols - 4]`. This leads to an optimization because HTROP can now allocate a smaller buffer and save data transfer overheads from the device. Please note that we currently only apply this optimization for *maximum* access location to allocate a smaller buffer size. The same technique can be applied for the minimum access location (if *min* is greater than 0). Then all existing access locations in the kernel need to be automatically aligned to the new offset (basically a new addressing scheme).

The data type is inferred from static code analysis while the data size is obtained from the previous access location analysis. Additionally, our approach also specifies if the buffer will be used for read-only, write-only, or read-write operations to introduce another optimization to reduce the data transfer overheads. In our example, we can infer that array `in` has only read memory accesses (lines 15–18) and array `out` has only write memory accesses (line 19). Hence, HTROP transfers array `in` to the read-only device buffer, then creates array `out` in the write-only device buffer and transfers only array `out` back to the host. Once the data size, data type, and buffer type are known, the buffers are created on the device by calling the `clCreateBuffer` API. The `clEnqueueWriteBuffer` API call is then used to transfer data from host memory to the buffer on the OpenCL device.

For applications with multiple hotspots or multiple calls to hotspots, further data optimizations are applied. HTROP is able to skip data transfers when the data is already present on the device due to a previous execution and will not be modified by others. To achieve this, every OpenCL device created by HTROP contains a data structure that stores the buffers that are currently present on the device. Once the data is transferred to a device or created there, unnecessary data transfers are skipped by leaving the data on the device for as long as possible. This optimization is especially important for applications with multiple calls to hotspots that use the same buffers between successive hotspot calls. To reuse buffers between different hotspot calls, their size needs to be sufficiently large and of the appropriate buffer type. HTROP analyzes the required buffer size and type for all the hotspot calls and chooses the largest size and appropriate buffer type that is valid for all the hotspot calls. Appropriate buffer type means that if, for example, a hotspot call uses a buffer as read-only and a successive hotspot call could reuse the buffer but needs to write to it, we directly allocate the buffer as read-write. Data is only transferred back to the host if the runtime decision selects a different accelerator or the sequential version. To maintain data consistency, all

Fig. 2. The legacy applications are either compiled with a native compiler to form the sequential x86 baseline or with HTROP to form the parallel OpenCL-enabled application. The compiled baseline and parallel applications are executed on the hardware platform. For the baseline, the computation is sequential on the CPU, whereas the parallel application is transparently offloaded to all available devices (CPU, GPGPU, and PHI).

the buffers that were modified on the device are transferred back to the host. Afterward, only the buffers required by the next hotspot call are transferred to the new target accelerator.

5 EVALUATION

In this section, we evaluate our approach. First, we present the hardware platform, the benchmark applications, and the test setup. Figure 2 gives an overview of the comparisons that are carried out. We analyze HTROP from two perspectives: the left part depicts the compilation comparison (details in Section 5.1), whereas the right part focuses on the execution aspects (details in Section 5.2). The execution is measured in terms of execution time and energy consumption to show the performance improvement and energy savings for each individual application and device. To give additional valuable insights of the impact of HTROP, we present results on the entire application- and hotspot level.

Multi-accelerator OpenCL platform. The evaluation is performed on a multi-accelerator platform with off-the-shelf hardware components. The platform consists of two Intel Xeon E5-2609 v2 multithreading CPUs (CPU in short), each with four physical cores (without hyperthreading) and 32GB of main memory. The platform features two additional accelerators with distinct architectures to offload the computation—a Nvidia Tesla K20c GPGPU and an Intel Xeon Phi 31S1P—both connected via PCI Express. The CPU, GPGPU, and Phi support OpenCL in version 1.2. The device drivers report the number of CUs as (CPU, GPGPU, PHI) = (8, 13, 224). Each CU is capable of working concurrently. The number of CUs needs to be evaluated carefully; it is not a clear indicator of the overall parallel capability of the device. In fact, each CU is able to handle the execution of multiple threads at the same time. For example, the CUs of our GPGPU device can process up to 2,048 threads (under optimal conditions: no thread-divergence and maximal occupancy), whereas, the CUs of the Phi can execute just 4 threads, but each of them can execute different instructions (i.e., they support thread divergence). We do not explicitly specify the work-group size but instead let the OpenCL device drivers automatically select the size for each device. Determining an optimal work-group size can lead to better performance, as shown for example by Cummins et al. [8]. State-of-the-art methods apply machine learning-based auto-tuning and source transformation to learn the significant features of the underlying hardware, the kernel implementations, and the transferred data. Integrating such methods into HTROP is an interesting area to further improve the results. We use the Ampehre framework [40] to accurately measure the power of all computing resources. The measurements are sampling based, which introduces small overheads when enabled.

Table 2. Applications With the Name and Description

Application	LoC Hotspot	Description
2mm [47]	18	2D matrix multiplications
bsop [66]	40	Black-Scholes option pricing for European options
fir [45]	16	Finite impulse response signal processing
enhance [14]	133	Chain of convolutions to enhance image quality
heat2D [47]	25	Simulation of heat transfer in 2D
nbody [31]	20	N-body particle simulation (kernel)
motion [50]	82	Motion detection; application contains five hotspots
raytrace [64]	68	Calculate rays from point of light (kernel)
sha256 [15]	37	Cryptographic hash function with 256-bit digest
stereo3D [61]	23	3D stereo matching

Lines of code (LoCs) are related to the hotspot size.

Benchmark applications. We use a set of benchmark applications for evaluating our approach. The applications are extracted from a broad set of domains (scientific computing, signal- and image processing, security, etc.). The benchmark applications are computationally intensive. For large input sizes, the execution time is dominated by the hotspot functions. All applications besides motion consist of one hotspot, whereas motion has five different hotspots that are also called multiple times. This application is a good candidate to evaluate the runtime decisions and data transfer optimizations (see Section 5.2.5). The source code for motion is available in our GitHub repository, and the remaining benchmarks can be found at their references (Table 2). We applied only minor modifications to some applications to allow the utilized version of Polly to identify all hotspots as SCoPs. Some hotspots contained irregular border handling for corner cases, which were resolved into conditional BREAK statements that could not be modeled with affine functions and were therefore not originally detected as SCoPs. The authors of Polly [21] follow a similar approach. But instead of applying minor modifications to the benchmark application, they analyze why the application is not detected as a SCoP and extend Polly to overcome this specific limitation. Extending Polly is outside of the scope of this work, but our approach will profit from such general improvements to Polly. In Table 2, all applications are listed by name, lines of code (LoCs) of the main hotspot, and a small description. Since our main goal is to enable legacy applications to use heterogeneous resources without any effort to the user, the benchmark applications are single-threaded CPU codes written in C/C++.

5.1 Application Compilation Comparison

In this section, we evaluate the different compilation tasks (see Section 3, ①–⑩) performed by HTROP and compare them to a native compiler (clang v3.8.0 with highest optimization level -O3). The application compilation is a static one-time effort that is typically not performance critical and depends on the application's size and its structure but not its input data. For the sake of clarity, we grouped tasks into the following categories:

- (a) *Init LLVM*: Seq. application ① is loaded into an LLVM module and the environment is set up.
- (b) *Preparation*: Optimizations are run to expose parallelism and improve the SCoP detection.
- (c) *Analysis*: Hotspots are detected and analyzed. Tasks (b) and (c) correspond to ②.
- (d) *CodeGen Decision*: Hotspots are evaluated and a codegen/no-codegen decision ③ is taken.

Fig. 3. Application compilation comparison. The hatched bars on the left represent the baseline compilation time with clang, whereas the right stacked bars represent the compilation duration of tasks performed by HTROP. The left y-axis shows the compilation time (seconds), and the right y-axis shows the application size (LLVM bitcode) in kB.

- (e) *Parallel Kernel CodeGen*: Hotspots are transformed to expose work-item parallelism and the OpenCL backend is called to translate LLVM bitcode into OpenCL C kernel code. This task corresponds to (4) through (7).
- (f) *Runtime and Host Code Integration*: The runtime control functions along with the corresponding wrappers and OpenCL host code are generated in (8) and (9).

Figure 3 represents the compilation comparison between the clang baseline and our approach. Overall, the compilation through our approach is 4.4× slower on average. The geometric mean drops to 3.6×, because three applications (enhance, stereo3D, and motion) take noticeably more time to compile than the others. enhance has the largest kernel size (see Table 2), stereo3D has a complicated hotspot structure, and motion has multiple hotspots. Phase (a) is proportional to the application size. Phases (b) and (c) take the biggest fraction of the compilation, followed by phase (e). The remaining phases (d) and (f) are extremely small. The compilation time for enhance is the largest and is dominated by the preparation phase (b). This phase uses built-in LLVM canonicalization and optimization passes to improve hotspot detection (see Section 4.1). The preparation phase works at an application level, and its runtime is dependent on the application size, as seen in the figure. In absolute times, clang compiles all applications on average in 0.7 seconds, whereas HTROP requires 3.5 seconds (2.3 seconds in terms of geom. mean). This shows that the HTROP approach incurs compilation overheads. However, these overheads seem tolerable since compilation is usually only performed once and the majority of the applications (7 of 10) still take less than 3 seconds to compile.

5.2 Performance and Energy Consumption

Following the overview in Figure 2, the next step after application compilation is the execution of the sequential baseline versus the OpenCL-enabled parallel application. Thus, this section presents the main outcome of this work. Even though the focus of this article is on the overall method and tool chain, its ultimate goal is to improve single-threaded legacy applications in terms of performance gains and energy savings. This is achieved by enabling the applications to offload the computation to different accelerators that need not be known at compile time. To highlight the different aspects of our approach, we do not just focus on the raw performance; we also complement the

findings with fine-grain energy measurements. We present our measuring method in Section 5.2.1 and the results on both the whole application level in Section 5.2.2 and the hotspot level in Section 5.2.3. Then we analyze the runtime overheads induced by online compilation of kernels and offloading them in Section 5.2.3 (similar to the compile time comparison in Section 5.1). Finally, we demonstrate the potential of HTROP's runtime decisions and data optimization in Section 5.2.5 and evaluate the hotspot energy across different devices in Section 5.2.6.

5.2.1 Measuring Method. Our evaluation platform supports three OpenCL-enabled offloading devices: CPU, GPGPU, and PHI. In a production environment, the runtime offloading decision can be used to offload the computation to the best-suited device. However, as we want to evaluate all application variances on all offloading devices, the runtime decision is disabled and the offloading device is selected prior execution with the help of a debug parameter in the runtime offloading decision module (see Section 4.4.1 and ⑧ in Figure 1). All measurements are executed multiple times (25 times for performance measurements and 15 times for energy measurements). There are fewer repetitions for the energy measurements, because the energy measuring framework needs to wait for the system to stabilize to an idle state after each execution, which requires a lot of time. The caching mechanism of HTROP, which reuses already generated code, is turned off. The kernel compilation is executed for each individual run. For the energy measurements, we measure the energy consumed by the CPU, GPGPU, and PHI. The CPU energy consumption is an aggregation of the energy consumed by both CPU packages as well as both DRAM memory banks. To reduce the interference by the system OS, we use thread pinning for the single-threaded baseline applications and set the CPU governor to performance to prevent effects from dynamic frequency scaling. The multi-accelerator platform has turbo boost and hyperthreading disabled. Even with this provisions, there were still some outliers. After the data was acquired, we automatically removed outliers with the interquartile range (IQR) [37] method (factor $1.5 \cdot \text{IQR}$). Over all measurement points, the method marked and removed on average about 3 of 25 runs for performance measurements and about 2 of 15 runs for energy measurement. After eliminating the outliers, the average over all remaining runs was taken. To ensure that the code generated by our approach is correct, we use Google Test [17] to validate the outputs of our approach against the original CPU code. Since accelerators may induce small rounding errors for floating-point computations, we tolerate values within a small threshold.

5.2.2 Application-Level Evaluation. Figure 4 shows the overall speedup (a) and energy-efficiency factor (b) across all evaluated applications and offloading devices. As the application execution time and energy consumption mainly depend on the input parameters, we do not select any single arbitrary input size but use many different input variants to show the overall behavior including break-even-points to contrast the areas where offloading is un-/profitable. Figure 4 lets us observe that the baseline energy consumption is driven by the execution time, with larger execution times resulting in higher energy consumption. The speedup potential/efficiency factor is governed by the reference execution time, which needs to be sufficiently large to allow amortization of the overheads of our approach. Hence, all points below the horizontal line ($\text{speedup} < 1$ or $\text{efficiency factor} < 1$) reflect applications/inputs, where computation offloading leads to an overall slowdown/energy increase of the applications. This is especially the case for very small execution times below 1 second. We also see that the break-even point ($\text{speedup} > 1$ or $\text{efficiency factor} > 1$) is different for different accelerators. For smaller execution times, the CPU is the first to reach the break-even point because the runtime overheads (e.g., data transfers, see Section 5.2.3) amortize quicker. However, the overall speedups achieved on the CPU mostly saturate at around 8 $\times$, because this is the maximum number of cores on the evaluation platform. For some applications (e.g., stereo3D and fir), we see higher speedups, where parallelization likely yields to positive

Fig. 4. Overall speedup (a) and energy-efficiency factor (b) across all applications and offloading devices. In (a), the x-axis represents the time required to execute the baseline application for a particular input size, whereas the y-axis represents the speedup (baseline execution time divided by the accelerator execution time). In (b), the x-axis represents the energy consumption of the whole system for the baseline application for a particular input size, whereas the y-axis represents the energy-efficiency factor (energy consumption of the system when executing the baseline divided by the energy consumption of the system when the computation is offloaded).

superscalar/memory reuse effects. The GPGPU and then the PHI require larger execution times to reach the break-even-point. If the application has sufficient computation, speedups and energy savings of up to two orders of magnitude can be achieved on the GPGPU and PHI.

5.2.3 Hotspot-Level Evaluation. Figure 4 showed the overall trends when executing the legacy applications versus offloading the computation with HTROP. In Figures 5, 6, and 7, we take an in-depth look at the performance and energy efficiency on the hotspot level compared to the application level and explicitly plot the input parameters on the x-axis for a representative subset of the benchmark applications (fir, raytrace, and bsop). We additionally evaluate our automated approach against a pragma-based OpenACC production compiler. We manually added OpenACC directives and pragmas to the three applications. The programmer-provided hints specify the regions of code for acceleration and the mapping and sizes for memory allocation and transfer. We use the NVidia/PGI compiler (version 18.4-0) to generate executables for the CPU and the GPU, denoted as OpenACC-CPU and OpenACC-GPU, respectively. The hotspot numbers (left plots, (a)) include kernel execution *and* data transfer overheads. The application numbers (right plots, (b)) are the same data points that are already shown in Figure 4. The figures demonstrate how the size of the input data effects the performance and energy efficiency of the hotspot and also that of the complete application. Please note that in Figures 5, 6, and 7, the light blue areas denote baseline executions times of less than 1 second. For these areas, our sampling-based measuring framework gives noisy results.

Comparing hotspot and application speedups, we see that the hotspot speedups quickly saturate for increasing input sizes. However, for the corresponding application speedups, we initially see a slowdown for smaller input sizes and later see a speedup for larger input sizes. Amdahl's law gives us the maximum speedup S of a parallel application as follows:

$$S \leq \frac{1}{f + (1 - f)/N},$$

(a) Hotspot including data transfer.

(b) Whole application.

Fig. 5. Hotspot- and application-level evaluation for fir on various input sizes. Left plots (a) show the performance and energy efficiency for the hotspot including data transfer overheads, whereas the right plots (b) show the performance and energy efficiency for the whole application.

(a) Hotspot including data transfer.

(b) Whole application.

Fig. 6. Hotspot- and application-level evaluation for raytrace on various input sizes.

(a) Hotspot including data transfer.

(b) Whole application.

Fig. 7. Hotspot- and application-level evaluation for bsop on various input sizes.

where f is the sequential fraction of the application and N the number of processors. As N tends to infinity, S tends to $1/f$. This means that the smaller the sequential fraction is, the larger are the achievable speedups. The benchmark applications have sequential initialization and finalize phases that contribute to f . However, using OpenCL for offloading, additional sequential overheads are introduced. These include the compilation time required by the OpenCL device drivers along with the time required for data transfers to and from the device. This results in a noticeable increase in the sequential fraction f of the applications, which limits the overall speedup potential. For smaller input sizes, f dominates the equation and cannot be amortized resulting in application

slowdowns despite seeing speedups for the corresponding hotspots. As the input sizes increase, the parallel fraction $1 - f$ of the application amortize the sequential fraction and we start seeing speedups. A detailed breakdown of the runtime overheads are presented in Section 5.2.3.

Looking at our approach for all three benchmarks, we see that for the hotspots alone (Figures 5(a), 6(a), and 7(a)), the GPGPU delivers the highest speedup followed by the PHI and the CPU. This is true for all input sizes. The application speedup, however, does not follow the same trend. In Figures 5(b) and 6(b), we see that initially, for small input sizes, the CPU delivers the highest speedup, and as the input size increases, the GPGPU and PHI achieve better speedups as compared to the CPU. This behavior can also be attributed to the runtime overheads (especially the OpenCL kernel compilation times), which are different for each device.

We can also observe that as the input size increases, the CPU's application speedup begins to saturate. This saturation point is close to the corresponding CPU hotspot speedup. Since the hotspot speedups of the GPGPU and PHI are higher than the CPU, they continue to climb, overtaking the CPU speedup until they also eventually reach their saturation point.

Looking at the overall trends between performance and energy, we see that the energy-efficiency factor closely follows the same trend as that of the speedup, which indicates that energy is mostly saved through reducing the execution time. However, we can see that the energy-efficiency factors are slightly lower compared to their corresponding speedups. In Figure 5(a) (hotspot-level analysis for fir), on the GPGPU, the hotspot performance constantly increases with higher input sizes. However, the energy improvement peaks at the input size 2^{11} and then slightly decreases. One reason is the impact of the energy-measuring framework, which needs to sample the power consumption on a regular basis to get fine-grain values. In this instance, the execution time up to input size 2^{11} is too small, resulting in only a few samples.

When comparing our approach to handwritten OpenACC code for the hotspot execution, we see that the performance of the OpenACC-CPU remains nearly constant over all input sizes. For fir and raytrace, offloading to the GPGPU with OpenACC performs slightly better than our approach, whereas the saturation point is the same for large input sizes. It is interesting to note that for the largest bsop input size, OpenACC-CPU is marginally better than HTROP-CPU, whereas the OpenACC-GPU has around $3\times$ the performance of the HTROP-GPGPU. Looking at the whole application, OpenACC performs better than HTROP. This is because HTROP needs to additionally amortize the OpenCL compilation overheads, whereas the PGI compiler already generates offloading code at compile time. Nevertheless, we see that for larger input sizes, HTROP's performance is comparable to handwritten OpenACC code while being fully automated.

5.2.4 Runtime Overheads. Similar to the compile time overheads in Section 5.1, we give a detailed breakdown of individual contributors to the runtime overheads. The compile time overheads are one time, whereas the runtime overheads are recurring for each execution of the application and/or each call to offload the computations. To evaluate our approach for different accelerators and for different input sizes, the runtime decision is configured to select a specific accelerator before execution. Figure 8 depicts the runtime phases (stacked) of the enhance application for various input sizes. As mentioned in the previous section, all runtime overheads contribute to the sequential execution fraction f of an application and thus limit the overall speedup potential that can be achieved. Hence, it is important to measure them. The initialization (a) and finalization (f) phases that create and destroy the data structures are extremely small/not visible. The total hotspot offloading time is decomposed into data to the accelerator (c), kernel execution (d), and data from the accelerator (e). The baseline application does not use OpenCL and hence cannot be further decomposed. Phases (b), (c), and (e) are only required for our approach to offload the computation. Hence, we consider them as overheads that contribute to f , which need to be amortized to achieve

Fig. 8. Different phases during the execution of enhance for various input sizes. The y-axis shows the execution time, and each group on the x-axis consists of the execution of enhance with the baseline or with HTROP.

Fig. 9. Concurrent execution of different applications compiled with HTROP.

speedups. We see that as the data transfers scale roughly with the kernel execution time, they do not prevent the hotspot speedups (see Figure 7(a)) for small input sizes.

The OpenCL compilation time (b) depends on the size of the kernel to compile and the device to compile for, but not the input data to the application. Therefore, the compilation time differs between devices but is mostly constant between the data points. This is because device-specific drivers compile the OpenCL kernel C code into device-specific executable code at runtime. The OpenCL compilation of the CPU and the GPGPU takes around 0.84 seconds. However, for the PHI, compilation takes 5.59 seconds, which is 7× more. This observation gives an additional explanation for Figure 4: The PHI suffers from slowdowns/energy inefficiencies for small executions because of this significant kernel code compilation.

5.2.5 Concurrent Execution With Multiple Applications and Hotspots. After measuring the impact of HTROP for applications in terms of performance improvement, energy consumption, and dynamic runtime overheads, we present a concurrent execution of the enhance, heat2D, and motion applications compiled separately with HTROP that demonstrates the potential of the runtime decisions and data optimization (Figure 9). The three available OpenCL devices (CPU, GPGPU, and PHI) are depicted at the top of the figure, whereas the lower part of the figure shows a breakdown of the execution trace for the three applications. The figure shows the real application trace when executed concurrently on the evaluation platform. All runs were repeated 25 times and

processed with the methods described in Section 5.2.1. In the presented scenario, enhance and heat2D start executing immediately, whereas motion is only ready after 5 seconds. The delayed start of motion helps to highlight and visualize the core features of our approach.

At (A), the runtime decision for enhance evaluates the input image size ($2,560 \times 1,440$ pixels) to be too small for acceleration. Therefore, the sequential CPU version is selected and no accelerator code is compiled. For this input size of heat2D at (B), the GPGPU has the highest score and is selected by the runtime decision. Only the GPGPU OpenCL kernel code is compiled, as the application has only one call to the hotspot and more accelerators cannot be used within this run. To understand the execution trace of motion, details about the applications functionality are important. The application receives two input images i_0 , i_1 and performs motion detection on them. It consists of five different hotspots $k_0 - k_4$. Hotspots k_0 , k_1 , and k_2 pre-process the images, whereas k_3 and k_4 compute and highlight the motion differences. Respectively, k_0 , k_1 , and k_2 are first applied to image i_0 , then to the second image i_1 separately. After that, k_3 and k_4 are applied to the pre-processed versions of the images. This sums up to eight calls to the hotspots and to eight corresponding runtime decisions. At (C), motion is ready and the first runtime decision evaluates the GPGPU to be the best resource, followed by the PHI and the CPU. However, as the GPGPU is not available, the PHI is selected and the OpenCL kernel compilation for the PHI is started. motion has multiple hotspots with multiple calls to these hotspots. To help reduce the OpenCL compilation overheads for further runtime decisions, the OpenCL kernel compilation for the GPGPU and CPU are started in separate threads (D). Once the compilation for the PHI is complete, image i_0 is transferred to the PHI and kernel k_0 is executed. At (E), the runtime decision evaluates the GPGPU to be the best resource, but as it is still not available, the PHI is selected. Kernel k_1 works only on the output of k_0 , which is already present on the PHI, and hence no data transfers are required. After the execution of k_1 , the runtime decision at (F) detects the availability of the GPGPU and selects it as the best candidate. The data migration functions are triggered, and all modified buffers from the PHI are transferred back to the host. The buffer required by k_2 (output of k_1) is transferred to the GPGPU and k_2 is executed. At (G), the kernel k_0 requires image i_1 and is transferred to the GPGPU. Since the pre-processed version of the first image i_0 is already present on the GPGPU (required by k_3), the computation stays on the GPGPU until all modified buffers are transferred back at the end. In (H), we can see that the same kernels k_0 and k_1 perform better on the GPGPU than on the PHI. Although for k_0 the difference is only 5% faster, k_1 is 3 $\times$ faster. Overall, if the same scenario is executed concurrently with the legacy baseline on the CPU, the end-to-end execution time is dominated by heat2D with 98.2 seconds. Consequently, the execution through HTrop (dominated by motion with 17.6 seconds) leads to a 5.6 $\times$ improvement in end-to-end execution time, even though motion was migrated during the execution from the PHI to the GPGPU. A study of different scenarios and measurements of the affinity of kernels for various accelerators is presented by Löscher et al. [39].

5.2.6 Hotspot Energy Consumption Across Devices. At the beginning of this section, we saw as a general trend that improved hotspot performance results in a corresponding energy saving. We now look at the average power consumption of each resource when they are active or idle to draw further insights from this general trend. We selected the relatively long-running heat2D and nbody applications with a large input size. Table 3 gives a breakdown of the average power consumed by each device for different offloading scenarios. Please note that the CPU is always executing OS processes in the background and handles the communication to/from the accelerator when the computation is offloaded to the GPGPU or PHI, which affects its idle power consumption. Additionally, the energy measurement framework does not report the accurate idle power consumption for the PHI. Whenever the PHI is queried, it automatically transitions from its idle to the active power state to service the query and hence reports a biased/overestimated value.

Table 3. Execution of Two Applications Through Different Scenarios (With/Without HTROP)

Data Point	Execution Through	Aerage Power in Watts			Total Time (Seconds)	Total Energy (Watt-Hours)	Performance Speedup	Energy Efficiency
heat2D input 2 ¹⁴	Baseline	37.54	26.71	96.56	227.14	10.15	1.00	1.00
	HTROP CPU	59.05	26.73	99.03	124.16	6.37	1.83	1.59
	HTROP GPGPU	38.64	116.11	98.59	40.75	2.87	5.57	3.54
	HTROP PHI	21.10	26.83	150.38	15.14	0.83	15.00	12.17
nbody input 2 ¹⁷	Baseline	37.89	26.74	96.36	130.85	5.85	1.00	1.00
	HTROP CPU	71.25	26.77	98.33	12.26	0.67	10.67	8.75
	HTROP GPGPU	38.65	115.51	97.73	4.53	0.32	28.89	18.46
	HTROP PHI	19.15	26.69	150.14	5.03	0.27	26.02	21.38

The dark grey cells show the average power consumption when the device is selected to offload the computation (active).

Looking at the average power consumption, we see that the HTROP CPU consumes 59.05W for executing heat2D while consuming around 12W more (71.25W on average) executing nbody. However, this does not signify that the CPU is more efficient as the accelerator for heat2D than for nbody. On the contrary, when looking at the performance and energy efficiency for both applications on this device, we see that nbody yields a 10.67× and 8.75× improvement in performance and energy efficiency, whereas heat2D only yields 1.83× and 1.59×. Thus, the higher performance for nbody along with only moderately higher power consumption allows for higher efficiency than for heat2D.

The same general observation holds for the GPGPU and PHI in comparison with the CPU. When well utilized, as in the test cases of Table 3, both use higher active power and show higher differences between idle (baseline) and active power compared to the CPU. Looking at the data for the nbody, we see an increase in the power consumption of around 33W, 89W, and 54W for the CPU, GPGPU, and PHI respectively, when they are active. This seems to be an increase in power consumption of 88%, 332%, and 56% for the CPU, GPGPU, and PHI. However, the idle power consumption of the overall system is quite high at around 160W, and when compared to this, we only see an increase of 22%, 56%, and 22% in overall power consumption for the CPU, GPGPU, and PHI, respectively. Since this increase in power consumption goes along with speedups of at least one order of magnitude, the overall efficiency of the system is significantly improved when using the accelerators. Overall, given these very distinct power profiles of the different compute devices, it is interesting to see how energy and performance still follow the same speedup and amortization trends (see Figure 4).

5.3 Discussion and Future Work

The success of our approach depends on some important factors that are discussed in this section.

First, as shown in the evaluation, the phases performed by HTROP introduce small but visible overheads to the application compilation and runtime. For the examined applications, we outlined clear break-even points, where applications offloaded to an accelerator were able to profit from parallelization, amortizing these overheads. However, this might not always be the case: If the computation is not sufficiently large or the hotspot is not a SCoP, the amortization is challenging. We try to minimize these overheads by adding early feasibility checks ③ at compile time (which exits the parallelization efforts if the hotspot shows not enough potential) and at runtime with offloading decisions ⑧ (that try to not invoke the kernel code compilation and accelerator offloading for input sizes below the break-even point).

Second, the reach of our approach is limited by what we can identify as a hotspot in our detection module. The use of Polly provides a good set of candidates along with supporting analysis functions. However, it is limited to loops that are classified as SCoPs, essentially affine loops that fit into the polyhedral model. Based on this model, Polly features powerful polyhedral transformations, such as loop tiling and loop fusion, and optimizations that we do not fully exploit here. However, this retains us the option to switch to other, non-polyhedral detection modules. Alternatives, for example, based on manual online and offline profiling [42], or on static partitioning based on application features [18], have been demonstrated but are currently not integrated into LLVM.

Last, we do not focus on generating fine-tuned highly efficient OpenCL compute kernels but rely to a great extent on the results produced by Axtor. Alternative OpenCL code generators do exist. With the help of LLVM's C backend [25], one can emit C code from LLVM bytecode. The result can be used by source-to-source compilers to translate it into OpenCL code (e.g., PPCG [62]). Similar to the hotspot detection module, our code generation module is implemented as a flexible module enabling us to replace the Axtor backend with another one with reasonable effort. Currently, the OpenCL kernel code is identical for all offloading devices. This is indicated in the evaluation, where the difference in performance between compute devices is relatively moderate for some applications. Given that the devices have different architectures, memory systems, available compute units, supported instructions, and so forth, our approach will benefit from accelerator-specific code optimizations and transformations (loop unrolling, tiling, etc.). Our approach could benefit from related work [53, 54] on performance-portability of OpenCL code and automated optimization compilers as presented by Chen et al. [6] for deep learning.

Although this article examines our tool flow on our multi-accelerator platform, we argue that the results can also be extended to other systems and domains. With its modular design, we can change components in HTRoP (hotspot detection, code generation, etc.) with reasonable effort. Furthermore, OpenCL is not only limited to the HPC domain but is also supported by mobile and embedded devices like MPSoCs, where energy consumption is also highly important. As a first step of our ongoing work, we were able to port HTRoP to an embedded board (Odroid) equipped with big.LITTLE CPUs and a Mali GPU.

6 CONCLUSION

In this article, we introduced our automatic and transparent compilation system called *HTRoP* to accelerate single-threaded CPU applications into OpenCL-enabled parallel applications that are offloaded to different heterogeneous accelerators (multithreaded CPU, GPGPU, and Xeon Phi). After highlighting related work and projects we build upon, we presented an overview of the design of HTRoP and described key features of the realization. HTRoP operates as a two-layered approach, identifying hotspots at compile time and offloading the computation at runtime based on a decision mechanism incorporating the input sizes or availability of accelerators. An application compiled with HTRoP is augmented with an infrastructure that enables migrating the execution between accelerators while minimizing data transfers by reusing data. We demonstrated that HTRoP can improve the application performance and reduce the energy consumption—with no user intervention. To this end, we analyzed our tool chain on a broad set of benchmark applications from diverse domains such as scientific computing, security, and signal- and image processing. The evaluation of these applications showed that energy savings mostly follow the performance gains. If the application has sufficient computational intensity, speedups and energy savings of up to two orders of magnitude can be achieved when compared to a natively compiled clang application with the highest optimization level (-O3). Our evaluation also shows that HTRoP can achieve comparable performance to handwritten OpenACC code while being fully automated. 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. The ability to target multiple different accelerators even for legacy applications, as well as the offloading decisions at runtime, can be enabling factors toward further optimizing performance and efficiency of an entire workload mix on a heterogeneous platform.

ACKNOWLEDGMENTS

Calculations leading to the results presented here were performed on resources provided by the Paderborn Center for Parallel Computing.

REFERENCES

- [1] B. Betkaoui, D. B. Thomas, and W. Luk. 2010. Comparing performance and energy efficiency of FPGAs and GPUs for high productivity computing. In *Proceedings of the International Conference on Field-Programmable Technology (ICFPT'10)*. 94–101.
- [2] Alecio Pedro Delazari Binotto, Dionisio Doering, Thorsten Stetzelberger, Patrick McVittie, Sergio Zimmermann, and Carlos Eduardo Pereira. 2013. A CPU, GPU, FPGA system for x-ray image processing using high-speed scientific cameras. In *Proceedings of the International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'13)*. IEEE, Los Alamitos, CA, 113–119.
- [3] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. 2008. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *Proceedings of the International Conference on Compiler Construction (CC'08)*.
- [4] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. *ACM SIGPLAN Notices* 43, 6 (June 2008), 101–113.
- [5] Jianjiang Ceng, Jerónimo Castrillón, Weihua Sheng, Hanno Scharwächter, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, et al. 2008. MAPS: An integrated framework for MPSoC application parallelization. In *Proceedings of the Design Automation Conference*. ACM, New York, NY, 754–759.
- [6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, et al. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. 578–594.
- [7] Codeplay. 2018. ComputeCpp. Retrieved March 26, 2019 from <https://www.codeplay.com/products/computesuite/compute/cpp>.
- [8] Chris Cummins, Pavlos Petoumenos, Michel Steuwer, and Hugh Leather. 2015. Autotuning OpenCL workgroup size for stencil patterns. arXiv:1511.02490.
- [9] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: An industry standard API for shared-memory programming. *IEEE Computational Science and Engineering* 5, 1 (1998), 46–55.
- [10] Marvin Damschen and Christian Plessl. 2015. Easy-to-use on-the-fly binary program acceleration on many-cores. In *Proceedings of the International Workshop on Adaptive Self-Tuning Computing Systems*.
- [11] Marvin Damschen, Heinrich Riebler, Gavin Vaz, and Christian Plessl. 2015. Transparent offloading of computational hotspots from binary code to Xeon Phi. In *Proceedings of the Design, Automation, and Test in Europe Conference (DATE'15)*. 1078–1083.
- [12] Gregory Frederick Diamos, Andrew Robert Kerr, Sudhakar Yalamanchili, and Nathan Clark. 2010. Ocelot: A dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'10)*. ACM, New York, NY, 353–364.
- [13] Artem Dinaburg and Andrew Ruef. 2014. McSema: Static translation of x86 instructions to LLVM. In *Proceedings of the ReCon Conference*.
- [14] Gianluca C. Durelli and Marco D. Santambrogio. 2016. Autonomic thread scaling library for QoS management. *ACM SIGBED Review* 13, 1 (March 2016), 41–47.
- [15] Oliver Gay. 2014. C++ SHA256 Function. Retrieved March 26, 2019 from <http://www.zedwood.com/article/cpp-sha256-function>.
- [16] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. 2006. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming* 34, 3 (June 2006), 261–317.
- [17] Google. 2018. Google Test. Retrieved March 26, 2019 from <https://github.com/google/googletest>.

- [18] Dominik Grewe and Michael F. P. O’Boyle. 2011. A static task partitioning approach for heterogeneous systems using OpenCL. In *Proceedings of the International Conference on Compiler Construction (CC’11)*. 286–305.
- [19] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. 1996. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing* 22, 6 (1996), 789–828.
- [20] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. 2012. Polly-performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters* 22, 4 (2012), 1250010.
- [21] Tobias Grosser and Torsten Hoefler. 2016. Polly-ACC transparent compilation to heterogeneous hardware. In *Proceedings of the International Conference on Supercomputing*. ACM, New York, NY.
- [22] Khronos Group. 2018. C++ Single-Source Heterogeneous Programming for OpenCL. Retrieved March 26, 2019 from <https://www.khronos.org/sycl>.
- [23] LLVM Dev. Group. 2018. LLVM’s Analysis and Transform Passes. Retrieved March 26, 2019 from <https://llvm.org/docs/Passes.html>.
- [24] LLVM Dev. Group. 2018. Auto-Vectorization in LLVM. Retrieved March 26, 2019 from <https://llvm.org/docs/Vectorizers.html>.
- [25] LLVM Dev. Group. 2018. C Backend. Retrieved March 26, 2019 from <https://github.com/JuliaComputing/llvm-cbe>.
- [26] LLVM Dev. Group. 2018. `llvm::ExecutionEngine` Class Reference. Retrieved March 26, 2019 from http://llvm.org/doxygen/classllvm_1_1ExecutionEngine.html.
- [27] LLVM Dev. Group. 2018. `llvm::IRBuilder<T, Inserter>` Class Template Reference. Retrieved March 26, 2019 from http://llvm.org/doxygen/classllvm_1_1IRBuilder.html.
- [28] LLVM Dev. Group. 2018. `ScalarEvolution.cpp`. Retrieved March 26, 2019 from http://llvm.org/doxygen/ScalarEvolution_8cpp_source.html.
- [29] LLVM Dev. Group. 2018. User Guide for NVPTX Back-End. Retrieved March 26, 2019 from <https://llvm.org/docs/NVPTXUsage.html>.
- [30] OpenACC Working Group. 2011. The OpenACC Application Programming Interface. Retrieved March 26, 2019 from <https://www.openacc.org/sites/default/files/inline-files/OpenAcc.2.7.pdf>.
- [31] Mark Harris. 2014. Mini-Nbody: A Simple N-Body Code. Retrieved March 26, 2019 from <https://github.com/harrism/mini-nbody>.
- [32] Xilinx Inc. 2015. SDAccel Development Environment. Retrieved March 26, 2019 from <https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html>.
- [33] Rashid Kaleem, Rajkishore Barik, Tatiana Shpeisman, Chunling Hu, Brian T. Lewis, and Keshav Pingali. 2014. Adaptive heterogeneous scheduling for integrated GPUs. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT’14)*. IEEE, Los Alamitos, CA, 151–162.
- [34] Lester Kalms, Tim Hebbeler, and Diana Göhringer. 2018. Automatic OpenCL code generation from LLVM-IR using polyhedral optimization. In *Proceedings of the Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*. ACM, New York, NY, 45–50.
- [35] Jungwon Kim, Honggyu Kim, Joo Hwan Lee, and Jaejin Lee. 2011. Achieving a single compute device image in OpenCL for multiple GPUs. *ACM SIGPLAN Notices* 46, 8 (Feb. 2011), 277–288.
- [36] W. Kim and M. Voss. 2011. Multicore desktop programming with Intel threading building blocks. *IEEE Software* 28, 1 (Jan. 2011), 23–31.
- [37] Stephen Kokoska and Daniel Zwillinger. 1999. *CRC Standard Probability and Statistics Tables and Formulae*. CRC Press, Boca Raton, FL.
- [38] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*. 75–86.
- [39] Achim Lösch, Tobias Beisel, Tobias Kenter, Christian Plessl, and Marco Platzner. 2016. Performance-centric scheduling with task migration for a heterogeneous compute node in the data center. In *Proceedings of the Conference on Design, Automation, and Test in Europe*. 912–917.
- [40] Achim Lösch, Alex Wiens, and Marco Platzner. 2018. Amphele: An open source measurement framework for heterogeneous compute nodes. In *Proceedings of the International Conference on Architecture of Computing Systems*. 73–84.
- [41] Alberto Magni, Christophe Dubach, and Michael O’Boyle. 2014. Automatic optimization of thread-coarsening for graphics processors. In *Proceedings of the International Conference on Parallel Architectures and Compilation (PACT’14)*. ACM, New York, NY, 455–466.
- [42] Toan X. Mai and Jongeun Lee. 2014. Efficient software-based runtime binary translation for coarse-grained reconfigurable architectures. In *Proceedings of the Parallel and Distributed Processing Symposium Workshops (IPDPSW’14)*. IEEE, Los Alamitos, CA, 132–140.
- [43] Simon Moll. 2011. *Decompilation of LLVM IR*. Master’s Thesis. Saarland University.

- [44] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable parallel programming with CUDA. In *Proceedings of the ACM SIGGRAPH Conference*. ACM, New York, NY, 16.
- [45] Sebastian Nilsson. 2017. FIR Filter Arduino Library. Retrieved March 26, 2019 from <https://github.com/sebnil/FIR-filter-Arduino-Library>.
- [46] Polly. 2018. polly::SCoP Class Reference. Retrieved March 26, 2019 from https://polly.llvm.org/doxygen/classpolly_1_1Scop.html.
- [47] Louis-Noël Pouchet. 2018. PolyBench Benchmark Suite. Retrieved March 26, 2019 from <http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/>.
- [48] Dagger Project. 2018. Dagger Decompile Framework Based on LLVM. Retrieved March 26, 2019 from <http://dagger.repzret.org>.
- [49] Heinrich Riebler, Gavin Vaz, Tobias Kenter, and Christian Plessl. 2018. Automated code acceleration targeting heterogeneous OpenCL devices. *ACM SIGPLAN Notices* 53, 1 (Jan. 2018), 417–418.
- [50] Heinrich Riebler, Gavin Vaz, Tobias Kenter, and Christian Plessl. 2018. HTROP. Retrieved March 26, 2019 from <https://github.com/pc2/htrop>.
- [51] Sean Rul, Hans Vandierendonck, Joris D’Haene, and Koen De Bosschere. 2010. An experimental study on performance portability of OpenCL kernels. In *Proceedings of the Symposium on Application Accelerators in High Performance Computing (SAAHPC’10)*.
- [52] Amit Sabne, Putt Sakdhnagool, Seyong Lee, and Jeffrey S. Vetter. 2015. Understanding portability of a high-level programming model on contemporary heterogeneous architectures. *IEEE Micro* 35, 4 (2015), 48–58.
- [53] Michael Showerman, Jeremy Enos, Avneesh Pant, Volodymyr Kindratenko, Craig Steffen, Robert Pennington, Wenmei Hwu, et al. 2009. QP: A heterogeneous multi-accelerator cluster. In *Proceedings of the International Conference on High-Performance Clustered Computing*.
- [54] Deshanand Singh. 2011. *Implementing FPGA Design With the OpenCL Standard*. White Paper. Altera.
- [55] Kyle Spafford, Jeremy Meredith, and Jeffrey Vetter. 2010. Maestro: Data orchestration and tuning for OpenCL devices. In *Euro-Par - Parallel Processing*. Springer, 275–286.
- [56] John E. Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering* 12, 3 (2010), 66–73.
- [57] Ben Taylor, Vicent Sanz Marco, and Zheng Wang. 2017. Adaptive optimization for OpenCL programs on embedded heterogeneous systems. *ACM SIGPLAN Notices* 52, 5 (May 2017), 11–20.
- [58] Kuen Hung Tsoi and Wayne Luk. 2010. Axel: A heterogeneous cluster with FPGAs and GPUs. In *Proceedings of the International Symposium on Field Programmable Gate Arrays*. ACM, New York, NY, 115–124.
- [59] G. Vaz, H. Riebler, T. Kenter, and C. Plessl. 2014. Deferring accelerator offloading decisions to application runtime. In *Proceedings of the International Conference on ReConfigurable Computing and FPGAs (ReConFig’14)*. 1–8.
- [60] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, Jose Ignacio Gomez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimization* 9, 4 (2013), 54.
- [61] Sven Verdoolaege, Hristo Nikolov, and Todor Stefanov. 2007. PN: A tool for improved derivation of process networks. *EURASIP Journal on Embedded Systems* 2007, 1 (2007), 19.
- [62] Trent Willis. 2014. Advanced Ray-Tracer. Retrieved March 26, 2019 from <https://github.com/trentwillis/ray-tracer>.
- [63] Peng Yang, Chun Wong, Paul Marchal, Francky Catthoor, Dirk Desmet, Diederik Verkest, and Rudy Lauwereins. 2001. Energy-aware runtime scheduling for embedded-multiprocessor SOCs. *IEEE Design & Test of Computers* 18, 5 (2001), 46–58.
- [64] NVIDIA Developer Zone. 2018. NVIDIA OpenCL SDK Code Samples. Retrieved March 26, 2019 from <https://developer.nvidia.com/opencl>.

Received August 2018; revised January 2019; accepted February 2019