PHILIPP SCHAAD

MULTI-DEVICE GPU-CODE GENERATION WITH LLVM

MULTI-DEVICE GPU-CODE GENERATION WITH LLVM

PHILIPP SCHAAD

D-INFK
ETH Zürich

April-September 2017

SUPERVISORS:
Prof. Dr. Torsten Hoefler
Dr. Tobias Grosser

LOCATION:
Zürich

TIME FRAME:
April-September 2017

ABSTRACT

General purpose GPU computing is becoming increasingly more relevant in terms of leveraging massively parallel processing power, but writing code for the very heterogeneous space of GPU architectures is often cumbersome. We propose a solution that enables the automated generation of device independent GPGPU code in the compiler infrastructure project LLVM. There exist a number of different tools for generating GPGPU code from a set of given source languages but they are typically limited to one GPU architecture or can only work with few source languages. We offer a way to utilize the modular power of LLVM and standards like OpenCL to escape such limitations by extending LLVM's polyhedral optimizer with an OpenCL runtime and SPIR code generation, allowing execution of GPGPU code on any compatible device. By opening new doors for executing the same code on multiple different platforms, this allows us to build performance models that tell us where our code can be run most efficiently, and potentially enables the execution of GPGPU code on multiple different devices in parallel. Additionally, transporting code from one architecture to a different one does not require the code to be rewritten, thus greatly reducing the time investments in an architectural change.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

AST Abstract Syntax Tree

CFG Control Flow Graph

GPGPU General-Purpose GPU

ISA Instruction Set Architecture

isl integer set library

LLVM-IR `LLVM` Intermediate Representation

NUMA Non Unified Memory Architecture

PST Program Structure Tree

PTX Parallel Thread Execution

ROCm Radeon Open Compute

SCoP Static Control Part

SESE Single Entry Single Exit

SIMD Single Instruction Multiple Data

SPIR Standard Portable Intermediate Representation

Part I

INTRODUCTION

1 INTRODUCTION

In the past decades, the performance and speed of single-core processors has been steadily increasing. However, Moore's Law [25] and Dennard-Scaling [11] have started to fail, and it has become increasingly difficult and more expensive to pack more performance into a single processing unit [10]. Thus, in order to reach better performance, the goal has become to elevate the importance of parallel computation, utilizing multi-core CPUs and multi-processor systems to distribute the computational effort between different processing units. More recently, we have even started to harness the power of GPUs for this purpose, since they have been designed with parallelism in mind, which gives them a clear edge over CPUs in terms of parallel information processing [29].

However, to date a lot of the code being written is not particularly well optimized to utilize that new dimension of performance, instead heavily relying on sequential computation and containing loop nests that could be parallelized more effectively. In addition to that, writing good parallel code or adapting older code to this paradigm is often coupled with higher development costs, and the resulting code is generally more complex, making further development more difficult and expensive. There exist a number of tools and solutions, that allow the optimization of sequential program code to parallel or hybrid code using polyhedral techniques [31]. One of those tools is called `Polly` [18, 30], an extension to `LLVM`, working on `LLVM`'s intermediate language.

`LLVM` [20, 21] is a compiler infrastructure project that encompasses a wide collection of modular technologies, including the aforementioned language and platform-independent `LLVM` Intermediate Representation (LLVM-IR), reusable optimizers and analysis tools, in addition to a number of language specific front-ends and target code generators. The modular design allows it to be used for a broad range of different programming languages, making it relatively easy to implement and add new front- and back-ends, while still using `LLVM`'s optimizer. There already exist a number of front-ends for most of the well known and heavily used programming languages, and a lot of the most used target architectures - like x86(-64), ARM, PowerPC, MIPS, and even some graphics accelerators and C - already have code generation back-ends. This makes it a perfect parent project for `Polly`, especially in terms of re-usability.

`Polly`'s code generation has recently been extended to generate General-Purpose GPU (GPGPU) code [16], allowing generated parallel

code to benefit from the increased parallel processing capabilities of GPUs. However, the GPU world is incredibly heterogeneous with the mobile GPU market alone bringing a huge number of different architectures to the table, making it difficult to write or generate code for multiple devices [34]. `Polly`'s GPU code generation is also hindered by that diversity and thus currently only has the ability of generating Parallel Thread Execution (PTX) kernel code, which is NVIDIA's assembly-like intermediate language that can only be executed on supporting NVIDIA GPUs.

The goal of this work thus, is to extend `Polly` with the capability of generating GPGPU code for most of the major GPU architectures. We provide `Polly` with an interface to the OpenCL [28, 39] runtime library in addition to the current CUDA [26] interface, which enables the execution of SPIR/SPIR-V [38] kernel code (an intermediate language introduced in 2011 to develop device-independent binaries with OpenCL), allowing compilation to a vastly broader range of devices in the heterogeneous GPU spectrum.

1.1 CONTRIBUTIONS

We first provide an introduction to the background relevant for our work in Section 1.2. Chapter 2 forms an architectural description of our implementations:

- In Section 2.2 we introduce our OpenCL runtime library interface
- We discuss SPIR code generation in Section 2.3
- AMD support gets introduced in Section 2.4

In Chapter 3 we will evaluate the performance of our solution, provide some pointers to future and related work in Chapter 4 and Chapter 5, and finally offer our conclusions in Chapter 6.

1.2 BACKGROUND

In this section we will introduce a few important notions that reappear throughout our work and give additional background to some of the concepts touched upon in the Introduction.

1.2.1 *GPU Computing*

The GPU has been designed to be strong in its original field of application: working with and rendering 3-D graphics. An object in the 3-D world coordinate system is made up of a set of geometric primitives like triangles. That set of primitives is fed into the GPU, where each vertex of every primitive has to be transformed into the screen

space and shaded, meaning its interaction with light has to be computed. Since 3-D objects are often complex and the worlds we want to model and render are sometimes large, containing many objects, the number of vertices passed to the GPU can be enormous. However, they can all be computed independently of each other. The vertices are now assembled into triangles and mapped to their 2-D pixel locations where they form so called fragments. Each pixel's color information gets computed from the fragments interacting with it, possibly together with textures. Again, the number of pixels on a screen is large, but each pixel's color can be computed independently. Because of this parallel nature of graphics processing, the GPU's architecture has been designed to be a collection of smaller parallel processors.

Originally, GPUs have been implemented as fixed-function processors only able to excel in graphics rendering, but not much else. Over the past few years however, the architecture has gradually evolved to the point where fixed-function operations being run on every vertex and fragment have been replaced with functions provided by the user. This allows for an increased flexibility in the type of work with which the GPU can be tasked.

At the beginning of this development, customized GPGPU computations still had to be given to the GPU via graphics APIs, meaning programs had to be structured according to the graphics pipeline. This made GPGPU computations still very cumbersome and difficult to use for the average programmer. Emerging programming environments like NVIDIA's CUDA [26] later addressed that problem by providing a more direct interaction with the programmable parts of the GPU and easier to understand syntax. Through such environments, GPGPU programming no longer is a tool used by only few researchers, but has been made available to the common programmer, requiring little to no knowledge about the graphics pipeline [29].

1.2.2 *OpenCL and SPIR(-V)*

OpenCL [28] is an industry standard that aims to provide task-parallel and data-parallel computing on a variety of different microprocessors, including graphics accelerators, CPUs and even FPGAs, while trying to hide heterogeneity and keeping the source code platform independent. The existing software environments mentioned in Section 1.2.1 have generally been limited to a single type of architecture, and the required source code did not run on any other microprocessor families (for example CUDA, which only runs on NVIDIA hardware). OpenCL thus provides a number of core functionalities that are supported by all devices with an OpenCL implementation. The list of OpenCL supporting hardware is continuously growing but major vendors like Intel, AMD, NVIDIA, ARM and Qualcomm

(non exhaustive list) [27] all support OpenCL through their driver implementations [39].

OpenCL's core contributions are a common language for accelerator programming, standardized APIs, and lots of hardware abstractions, allowing easier development of accelerated task- or data-parallel applications. The standard OpenCL environment consists of a host CPU and any number of attached heterogeneous OpenCL devices, like GPUs, that typically have a completely different machine instruction set than the host CPU.

Run-time compilation is the driving force behind OpenCL's compatibility with different instruction sets. Device vendors ship OpenCL implementations in their drivers, providing run-time kernel compilers into their own instruction set. While this allows OpenCL programs to run natively on any target hardware, even if the developer does not have access to the platform, it does not guarantee that a particular program is optimized for peak performance on all architectures. However, even if a device vendor decides to make large changes to their instruction sets, drivers or supporting libraries, this remains hidden from the programmers perspective, and OpenCL will automatically utilize the latest features on the target device. OpenCL programs can still be pre-compiled off-line and shipped as binaries, at the cost of compatibility with heterogeneous hardware.

SPIR is a mapping from OpenCL C code to LLVM-IR, that aims to provide a portable interchange format for partially compiled OpenCL C programs [35, 41]. The goal is to provide the means of distributing device independent binaries rather than kernel source code in its corresponding OpenCL C format. Many device vendors support the necessary extension in their drivers to read and execute OpenCL kernels in their SPIR format. This means that one can easily generate a valid OpenCL kernel binary from within `LLVM`, without having to commit to a specific architecture.

1.2.3 *Presburger Sets and Relations*

Any program expression e is affine if it is one of the following:

- An integer constant (c)
- A variable (n)
- The negation of an affine expression ($-e$)
- The addition or subtraction of two affine expressions ($e_1 \pm e_2$)
- The multiplication of an integer constant with an affine expression ($c * e$)

If instead it is the floor division of an affine expression by an (integer) constant ($\lfloor \frac{e}{c} \rfloor$) or that division's remainder ($e \mod c$), we call it quasi-affine.

Building on that, a Presburger expression (or Presburger formula) [16] is an expression e_p made up of:

- A boolean constant ($\top$, $\bot$)
- A boolean negation, conjunction or disjunction ($\neg e_p$, $e_{p1} \wedge e_{p2}$, $e_{p1} \vee e_{p2}$)
- A quantifier expression ($\forall x : e_p$, $\exists x : e_p$)
- A comparison between (quasi-)affine expressions ($e_1 \diamond e_2$, $\diamond \in \{<, \leqslant, \geqslant, >, \neq, =\}$)

An n-dimensional Presburger set S is a subset of $\mathbb{Z}^n$ where the elements of S are described by a Presburger formula [16]. The sets

$$S_1 = \{(x_0, x_1) \mid p_0 = (50 \leqslant x_0 \leqslant 100) \wedge p_1 = (0 \leqslant x_1 \leqslant x_0)\}$$

and

$$S_2 = \{(x_2, x_3) \mid p_2 = (0 < x_2 \leqslant 20) \wedge p_3 = (x_2 \leqslant x_3 < 50)\}$$

are two examples of valid Presburger sets in $\mathbb{Z}^2$ since p_0, p_1, p_2 and p_3 are Presburger formulas. An empty two-dimensional Presburger set can be expressed with $S_e = \{(x_4, x_5) \mid \bot\}$, and a universal n-dimensional Presburger set gets described with $S_n = \{(y_0, ..., y_n)\}$.

We can obtain Presburger sets with elements from different name spaces by correctly annotating them with the corresponding name space. For example a set with elements from name spaces A and B could have the form $S_{AB} = \{[A, (x_0, x_1)] \mid 0 \leqslant x_0 < x_1; [B, (x_2)] \mid 50 < x_2 < 70\}$. Such sets are aptly called *named Presburger sets*.

Finally, a Presburger relation r is a binary relation of the form $r = \{(x_0, x_1) \mapsto (f_0) \mid p_0\}$, forming a subspace of $\mathbb{Z}^n \times \mathbb{Z}^m$ which is constrained by a Presburger formula p_0. Presburger sets and relations are closed under normal set operations such as union, intersection, or subtraction and allow the projection onto subspaces [16].

For a more detailed and formal explanation on Presburger formulas, sets and relations, see Verdoolaege [44].

1.2.4 *Polyhedron Model*

Code optimization via polyhedral techniques mainly relies on the transformation of so called Static Control Parts (SCoPs) in the source code, though it is possible to apply the polyhedral model to more general programs [8]. SCoPs are loop nests formally defined as the maximum number of consecutive program statements, where all (upper and lower) loop bounds and conditionals are Presburger expressions containing only surrounding loop iterators, numerical constants, and/or structure parameters (constants whose values are not known at compile-time, but remain fixed once assigned) [6, 13]. Note

Listing 1: Loop nest that can be transformed using the polyhedron model

```
1   for (int i = 1; i < n; i++) {
2     for (int j = 1; j < i + m; j++) {
3 A:    M[i, j] = M[i-1, j] + M[i, j-1];
4     }
5 B:  M[i, i+m+1] = M[i-1, i+m] + M[i, i+m];
6   }
```

that those requirements have to be semantically fulfilled, not necessarily syntactically. This implies that any control flow structure is a valid SCoP if it can be written as a set of loops and conditionals containing only Presburger expressions. Listing 1 shows a slightly modified example of a basic SCoP taken from Feautrier [14].

The polyhedron model is an abstract graph representation of such SCoPs in which it is easier to reason about possible parallel executions. Each program statement S inside a SCoP gets assigned an n-dimensional ($\mathbb{Z}^n$) polyhedron D_S, called its iteration domain, where n is the number of loops surrounding S inside the SCoP (also called the depth $d(S)$). E.g., for the SCoP in Listing 1 we get depths of $d(A) = 2$ and $d(B) = 1$ for statements A and B respectively. D_S is made up of n-dimensional vectors representing every loop iteration in which statement S executes. The elements of said vectors contain the current values of all surrounding loop iterators in that iteration. Each of the vectors forms a point in our $\mathbb{Z}^n$ graph representation.

For statement A and B in Listing 1 the resulting iteration domains thus are

$$D_A = \{(i,j) \mid 1 \leqslant i \leqslant n, 1 \leqslant j \leqslant i+m\}$$

and

$$D_B = \{(i) \mid 1 \leqslant i \leqslant n\}$$

respectively. We can obtain the iteration domain D_{scop} for our entire SCoP by unifying the iteration domains of all statements inside the SCoP. In the case of our example, the resulting domain is $D_{scop} = D_A \cup D_B = \{(i,j) \mid 1 \leqslant i \leqslant n, 1 \leqslant j \leqslant i+m\} \cup \{(i) \mid 1 \leqslant i \leqslant n\}$.

The main optimization steps of polyhedral optimizers are coordinate transformations on the source iteration domain of a SCoP. The basic procedure is to transform all the relevant SCoPs of a program into their respective polyhedron representation, then performing transformations on the SCoP's iteration domain, before using the resulting target domain to generate new code. A correct transformation must not violate any of the original data dependencies, meaning that if two or more operations access the same memory location and include at least one write operation, their order must be preserved. This dependency between statements can be expressed as a relation on the

iteration domain, mapping dependent operations to its source. The optimization step now is extremely flexible, because the goal is simply to find the best transformation matrix with the goal of maximizing a desired objective function, like exploiting parallelism as much as possible. Note that more recent approaches like `Polly` do not in fact modify the actual iteration domain itself anymore, but instead impose a scheduling relation on the domain, which determines the new execution order.

1.2.5 *Polly*

`Polly` mostly follows the pattern described in Section 1.2.4, but detects semantically valid SCoPs in the program source completely automatically. Since the input to `Polly` is a program in LLVM-IR, the semantic validity can be easily checked thanks to a number of analysis passes offered by `LLVM`, meaning that the original source code does not have to adhere to a specific syntactic structure. `Polly` uses a region-based SCoP detection, where the program's Control Flow Graph (CFG) is used to identify the relevant program parts. More specifically, it relies on a modified version of the Program Structure Tree (PST) introduced by Johnson [19].

The nodes inside a PST represent so called Single Entry Single Exit (SESE) regions in the program source. Such a region is a section of the CFG, which is only connected to the rest of the graph by a single entry edge and a single exit edge, meaning it can be represented by a function call. Such a function call can thus easily be replaced by a call to an optimized version, without affecting the control flow. `Polly` searches for the maximal regions inside the CFG of a function, to detect its largest possible SCoPs.

After finding the valid SCoPs, they get transformed into their polyhedral representation. `Polly` has native support for $\mathbb{Z}$-polyhedra and polyhedron operations, thanks to the integer set library (isl) developed by Verdoolaege [42]. As mentioned in Section 1.2.4, `Polly` does not change the domain of program statements. Instead, it applies all transformations to the scheduling relation imposed on the domains. This is done in part because this way it is easy to obtain the composition of two transformations, by just taking the composition of their two scheduling relations [18]. The resulting polyhedral representation then gets used to get back a generic Abstract Syntax Tree (AST) [17], which gets translated back into LLVM-IR.

The addition of GPU code generation with `Polly-ACC` by Grosser et al. [16] brings a lightweight CUDA runtime interface to `Polly` and allows the extracted and optimized SCoPs to be translated into GPU specific compute kernels. It automatically decides whether or not to perform accelerator mapping, and does not require any specific

coding styles to be enforced. However, to date, `Polly` can only target NVIDIA GPUs because the kernel code generation is limited to the NVPTX `LLVM` back-end, which produces PTX executable only by NVIDIA CUDA accelerators.

Part II

MULTI-DEVICE GPU-CODE GENERATION IN LLVM

2 ARCHITECTURE

2.1 OVERVIEW

We first provide an overview (see Figure 1) of Polly's accelerator mapping and our contributions (9b, 9c, and 14) to it. Since Polly is part of LLVM, we can use a number of different front-end compilers 2a - 2d to translate a large variety of programming languages 1a - 1d into LLVM-IR 3, where we then perform our polyhedral optimizations and accelerator mapping.

The resulting LLVM-IR first gets run through a series of canonicalization passes 4 with the goal of simplifying control flow, translating memory to registers, and optionally performing inlining to reduce the level of abstraction and make the code easier to work with.

The two following steps have been roughly outlined in Section 1.2.5. Polly automatically detects valid SCoPs in the canonicalized LLVM-IR and extracts them as parts to be optimized 5. Once the relevant SCoPs have been extracted, Polly transforms them into their polyhedral representation and continues to optimize the scheduling relation describing them 6.

Polly now decides on which parts of the schedule tree would benefit from being run on an accelerator and marks them accordingly 7. Those parts then get mapped to the accelerator using a recent version of PPCG [45]. Together with that, Polly introduces the necessary data transfers between global and shared memory.

2.1.1 *GPGPU Code Generation*

The schedule tree containing the correct device mapping now gets translated back into a generic AST [17], which in turn gets used to regenerate LLVM-IR. This is accomplished by first translating the AST's outer layers into code that should be run on the host device 9a, before parts are reached, that are meant to be run on accelerators.

In steps 9b through 9d we now insert a few device specific runtime API calls into the host code to correctly launch the kernels. Together with that, the kernel IR is extended with special calls and primitives according to which device they were mapped to. Such calls include memory accesses, inter-thread synchronization, and thread-identification/-numbering. The resulting kernel code gets loaded into separate modules which can now be either directly embedded back in to the host IR 12, or get passed through the relevant LLVM tar-

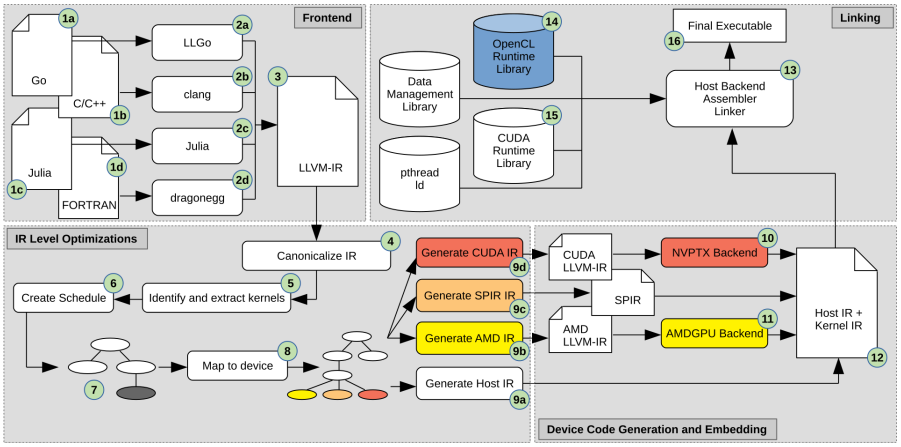

Figure 1: An overview of `Polly`'s multi device GPGPU code generation (Modified overview from Grosser et al. [16])

get code generation back-end (10 and 11) to retrieve device specific IR/Assembly before being merged with the host code.

The resulting combined IR module can now finally be run through the host's target back-end, the assembler and the linker 13. In this last step we link our program against a data management library (also referred to as our GPU runtime library), which has the duty of optimizing and managing data transfers between accelerators and host. This library is also responsible for providing and handling calls to the GPGPU compute APIs OpenCL and CUDA and their respective runtime libraries 14/15. Our completed multi-device executable finally gets emitted after the linking step 14.

For a more detailed explanation and a closer look at the individual steps up to this point, see Grosser et al. [16].

2.1.2 *CUDA Code Generation*

We will now take a closer look at what `Polly` has been capable of doing up to now in terms of GPU device specific target code generation, namely the creation of CUDA PTX assembly. This procedure can be split into two parts: CUDA specific IR generation 9d and translation to PTX assembly 10.

In the first step, we insert the aforementioned device specific annotations, primitives and API calls into the kernel IR. In this case we're talking about the CUDA API. While there are tons of CUDA specific API calls, annotations, and primitives, only a couple of them are relevant for our `Polly` kernels. Listing 2 shows a silly toy example of CUDA specific LLVM-IR that does nothing other than add the current Block- and Grid-IDs[1] together and return, storing the result in the

1 Block- and Grid-IDs are used to determine the GPU thread's indexing number where the kernel is currently being executed on

Listing 2: A toy example of CUDA-ready LLVM-IR

```
1  target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-" \
2                      "i32:32:32-i64:64:64-i128:128:128-" \
3                      "f32:32:32-f64:64:64-v16:16:16-v32:32:32-" \
4                      "v64:64:64-v128:128:128-n16:32:64"
5  target triple = "nvptx64-nvidia-cuda"
6
7  define ptx_kernel void @kernel(i32* %res) {
8  entry:
9    %0 = call i32 @llvm.nvvm.read.ptx.sreg.ctaid.x()
10   %1 = call i32 @llvm.nvvm.read.ptx.sreg.tid.x()
11   %2 = add i32 %0, %1
12   store i32 %2, i32* %res
13   ret void
14 }
15
16 declare i32 @llvm.nvvm.read.ptx.sreg.ctaid.x()
17 declare i32 @llvm.nvvm.read.ptx.sreg.tid.x()
18 !nvvm.annotations = !{!0}
19 !0 = !{void (i32*)* @kernel, !"maxntidx", i32 32, !"maxntidy",
20      i32 1, !"maxntidz", i32 1}
```

return parameter. It perfectly shows all of the small additions and changes to the IR to make it CUDA compatible.

First, we need to specify the `target datalayout`, which indicates data-type size, offset, and alignment properties to the target code generation back-end (NVPTX in our case). Next is the `target triple`. This tells LLVM that we want to use the NVPTX back-end to generate CUDA compliant code for an NVIDIA graphics card. Together with that, the kernel function gets annotated with `ptx_kernel`, which dictates the calling convention used to call the function. The module itself also gets some named metadata (`nvvm.annotations`) to tell the CUDA runtime what the maximum number of GPU threads for the execution should be. Finally, the correct calls for GPU thread-indexing and -synchronization are inserted (lines `9` and `10` in Listing 2 show examples of CUDA thread indexing calls).

In the second step we pass the CUDA compliant LLVM-IR obtained this way to the NVPTX target code generation back-end. NVPTX takes on the job of translating the IR to valid PTX assembly code, which then gets embedded back in the host program. When we execute our binary, this PTX assembly code gets passed along to the GPU runtime library, where CUDA launches it on an accelerator.

It is obvious that up to this point the full potential in the modularity and re-usability of `LLVM`'s code generation back-end architecture has not yet been exploited. By only generating PTX with NVPTX we effectively are bound by two hefty limitations. For one, PTX assembler

code can only be run on NVIDIA devices, and `Polly`'s GPU runtime library needs to have a CUDA API present on the target system.

2.2 OPENCL RUNTIME

In our first contribution we thus eliminate the second limitation by extending `Polly`'s data management library with a lightweight interface to the OpenCL runtime API (14) in addition to the already present CUDA interface. Having access to the OpenCL runtime library has two big implications.

For one, we are no longer limited to systems running CUDA. `Polly`'s GPGPU code generation now only requires at least one of the two compute libraries to be installed. This addition is designed in such a way that the data management library's API remains identical to the host program, independent of which runtime API is being used. The user can thus freely choose between the two runtimes - if they are both available - using the flag `-polly-gpu-runtime=libcudart/libopencl`. Different initialization calls to the data management library are inserted into the host code according to the user's selection, but the remaining interactions with the library remain unchanged. This minimal invasive approach keeps complexity in the host program low and modularity as well as flexibility high.

In addition to that, the use of OpenCL as a GPU compute library builds an important foundation for us to run GPGPU code on any number of different accelerator architectures. As mentioned in Section 1.2.2, OpenCL's aim is to provide platform independent GPGPU computing for a huge variety of devices, including GPUs, CPUs, and FPGAs. Having direct access to the OpenCL runtime through our data management library thus makes `Polly`'s GPU code generation future proof, allowing the later addition of pretty much any architecture by simply adding a new IR code generation step (9b) - (9d).

On NVIDIA devices, the PTX kernel code generated that way can be executed via OpenCL by treating it as a pre-compiled kernel binary (OpenCL's intermediate representation for kernels compiled online on NVIDIA devices is also PTX), just like it would be with CUDA. We just need to adapt two minor things when generating the code:

- The `target triple` needs to be changed to `"nvptx64-nvidia-nvcl"` to indicate use of OpenCL instead of CUDA
- Kernel arguments that are pointers to GPU memory need to be annotated with address space indicators, typically `addrspace(1)`, which stands for global memory

Since the second change gets ignored by CUDA, we now always annotate address space indicators, even when compiling for CUDA. The `target triple` changes based on the runtime flag chosen at compile time.

2.3 SPIR CODE GENERATION

Our second contribution aims to reduce the restriction the use of PTX assembly imposes on us, by providing `Polly` with SPIR code generation. As discussed in Section 1.2.2, SPIR is a mapping from OpenCL C into LLVM-IR, and thus it provides the same cross-device re-usability as OpenCL. In theory, this allows us to generate GPU code for pretty much any platform or device with a driver supporting OpenCL and the `cl_khr_spir` extension[2]. In practice, it is not that simple.

According to its standard, SPIR is based off of LLVM-IR release version `3`, with SPIR `1.2` and `2.0` being based on IR versions `3.2` and `3.4` respectively [35, 41]. This is problematic, since there currently is no target code generation back-end for SPIR in `LLVM`. The only way of obtaining valid SPIR `1.2` or `2.0` code is through standalone compilers or tools like Khronos Group's SPIR generator [36]. We can also not directly use our kernels' LLVM-IR, even though SPIR is at its core just LLVM-IR. This is because our extensions to `Polly`'s accelerator code generation are found in the latest release and development versions of `LLVM/Polly`, which are `5.0` and up. With the evolution from `LLVM 3.0` to version `4.0`, the IR has been subject to changes as well. Thus, our kernel IR is not directly compliant to the SPIR standard, and as a result cannot be read by most standard OpenCL driver implementations.

However, for this work we mainly focused our efforts on getting SPIR code generation to work for Intel devices, where there are ways of getting around that problem. Since Intel's standard OpenCL driver does not have the capability of handling the SPIR code produced via our code generation method (expects standard flavor based on LLVM-IR `3.2`), our implementation utilizes a different driver on Intel Platforms. Beignet is an open source implementation of the OpenCL specification for Intel Platforms [7]. The driver allows for the creation of OpenCL kernel programs from LLVM-IR files in valid SPIR format, which is exactly what we need. Beignet's driver back-end uses `LLVM 4.0`, which has two implications.

For one, `LLVM 4.0` needs to be installed on the system. Our added features however affect the most recent versions of `LLVM`, and thus, two instances need to be available (version `4.0` installed for Beignet, and a more recent copy containing our contributions).

2 `cl_khr_spir` is a platform extension that adds support to create OpenCL programs from SPIR binaries [40]

Listing 3: Listing 2's toy example, but in SPIR-ready LLVM-IR

```
target datalayout = "e-p:32:32:32-i1:8:8-i8:8:8-i16:16:16-" \
                    "i32:32:32-i64:64:64-i128:128:128-" \
                    "f32:32:32-f64:64:64-v16:16:16-v24:32:32-" \
                    "v32:32:32-v48:64:64-v64:64:64-v96:128:128-"\
                    "v128:128:128-v192:256:256-v256:256:256-" \
                    "v512:512:512-v1024:1024:1024"
target triple = "spir-unknown-unknown"

define spir_kernel void @kernel(i32 addrspace(1)* %res)
  !kernel_arg_addr_space !0 !kernel_arg_name !1
  !kernel_arg_access_qual !1 !kernel_arg_type !1
  !kernel_arg_type_qual !1 !kernel_arg_base_type !1 {
entry:
  %0 = call i32 @__gen_ocl_get_group_id0()
  %1 = call i32 @__gen_ocl_get_local_id0()
  %2 = add i32 %0, %1
  store i32 %2, i32* %res
  ret void
}

declare spir_func i32 @__gen_ocl_get_group_id0()
declare spir_func i32 @__gen_ocl_get_local_id0()
!0 = !{i32 1}
!1 = !{!""}
```

As a second implication, the LLVM-IR passed to Beignet when creating an OpenCL kernel program needs to adhere to the specifications as of version `4.0`. Since the `IR` has not been subject to any major changes from version `4.0` onwards, we can directly use our generated code (which is based on the most recent version of LLVM-IR) without any major changes, other than making it compliant to Beignet's SPIR flavor.

To obtain valid Beignet-SPIR code we follow a similar work-flow as described in Section 2.1.2. Since our resulting code is modified LLVM-IR, we can however directly embed our SPIR specific IR back into our host program, without having to run it through an additional target code generation back-end (9c). Listing 3 shows the same toy example used in Section 2.1.2 to explain CUDA specific IR, but this time it is compliant to Beignet's SPIR flavor.

The kernel function gets annotated with `spir_kernel`, indicating the use of SPIR calling conventions. Our `target triple` and `target datalayout` get set according to the SPIR `1.2` specification [35], and all thread-indexing and -synchronization calls get assigned their SPIR specific names (here they also get marked with the calling convention `spir_func`, which was not required for CUDA/PTX). Because

this code will be run by our OpenCL runtime interface, GPU memory pointer arguments also need to carry the correct address space indicators (`addrspace(1)` in our toy example).

Finally, we have to take care of the metadata. SPIR does not require the maximum number of GPU threads to be marked and since NVIDIA's `nvvm` compiler is not involved, the annotation `nvvm.annotations` can be left out. Instead, SPIR requires the kernel functions to be annotated with information about their arguments. Not all of those details are directly at our finger tips when generating the compute kernels, since they were not needed for CUDA and have not been made readily available so far. However, Beignet luckily handles that information in a sloppy way. Our kernel's SPIR code is deemed valid as long as every type of information Beignet looks for is present (even if it is empty) with the exception of the argument address space qualifier. This means that we only have to set the correct address spaces for each kernel argument in `kernel_arg_addr_space` (In our toy example this is only one argument with global address space, so we set `kernel_arg_addr_space` to `!{i32 1}`). The rest of the metadata fields can be set to an empty string since their value is not important for the validity and runtime correctness of our code in Beignet.

The resulting SPIR specific kernel IR now gets directly embedded back into the host program, where it then gets passed to the OpenCL runtime for execution. Here Beignet makes things a little more complicated by not accepting the kernel in OpenCL's standard function `clCreateProgramWithBinary`. Instead, Beignet provides a new API call named `clCreateProgramWithLLVMIntel`, which only accepts SPIR kernels from a file input. Our GPU runtime library (data management library) thus automatically detects if Beignet is available on the system during execution, and writes our kernel to a temporary file, passing it to Beignet's special function.

It is important to note that Beignet needs to be at the active development version. The latest release does not yet have support for `LLVM` version `4.0`, and with that will not be able to read our LLVM-IR. Since it is a development build, things like the way the OpenCL program creation are handled may be subject to change before we get an official Beignet release supporting our SPIR code. However, major changes should probably not be expected, and our code generation can be very easily adapted to minor changes.

Listing 4: Listing 2's toy example, but in AMDGPU-ready LLVM-IR

```
1  target datalayout = "e-p:32:32-p1:64:64-p2:64:64-p3:32:32-" \
2                      "p4:64:64-p5:32:32-i64:64-v16:16-v24:32-" \
3                      "v32:32-v48:64-v96:128-v192:256-v256:256-" \
4                      "v512:512-v1024:1024-v2048:2048-n32:64"
5  target triple = "amdgcn-amd-amdhsa-opencl"
6
7  define amdgpu_kernel void @kernel(i32 addrspace(1)* %res) {
8  entry:
9    %0 = call i32 @llvm.amdgcn.workitem.id.x()
10   %1 = call i32 @llvm.amdgcn.workgroup.id.x()
11   %2 = add i32 %0, %1
12   store i32 %2, i32* %res
13   ret void
14 }
15
16 declare i32 @llvm.amdgcn.workitem.id.x()
17 declare i32 @llvm.amdgcn.workgroup.id.x()
```

2.4 AMD CODE GENERATION

With our third contribution we try to further minimize the restrictions under which we are kept by the current code generation method(s), by providing `Polly` with target code generation for AMD GPUs. Luckily for this we can fall back on one of the many `LLVM` back-ends already implemented in the project, just like the aforementioned NVIDIA NVPTX for CUDA PTX. The AMDGPU back-end provides AMD Instruction Set Architecture (ISA) [1] code generation for AMD GPUs, starting with the R600 family up until the current GCN families [2]. This implies that we can follow pretty much the same steps (with one addition which we discuss later) as for NVIDIA code generation, simply replacing the CUDA and NVPTX specific IR elements with their AMD and AMDGPU specific counterparts.

Listing 4 shows our familiar toy example in AMDGPU back-end ready LLVM-IR after our transformations. As with SPIR, we first adapt our `target datalayout` and `target triple` to values specified by the AMDGPU code generation back-end. The target triple here indicates that the produced ISA will later be used as an OpenCL kernel, and should thus follow some specific guidelines. We also change our kernel function's calling convention to `amdgpu_kernel`, mark OpenCL memory object pointer arguments with the global address space qualifier (`addrspace(1)`), and insert our AMD specific calls for thread-indexing and -synchronization. The two examples for OpenCL's local and global ID are seen on lines `9` and `10` in our toy examples, with their declarations on lines `16` and `17`. Our AMD kernel IR does not

require any additional metadata to be added, so we can skip that step and drop the NVIDIA specific `nvvm` annotations as well.

After running our newly transformed kernel IR through the AMDGPU target code generation back-end, we obtain an ELF relocatable object file. This file standard can not directly be executed by OpenCL, but must first be linked into an ELF shared object file. We can use `LLVM`'s own linker `LLD`[3] for that. Unfortunately, `LLD` can not be called from within `LLVM/Polly`'s source code at this time, since it is not available as a standard pass linked into `LLVM`. Thus, we have to implement a workaround, by piping the ELF relocatable object file output from the AMDGPU back-end into a temporary file. We then call `LLD` as an external program using the C standard library function `system`, which passes a command string to the host environment, telling the command processor to pass our temporary file through `LLD`. It then blocks and returns when our `LLD` pass has completed. We can then read back the now linked ELF shared object file and embed it back in our host program code, where it then gets passed to the OpenCL runtime interface in our data management library upon execution. OpenCL can now create a working kernel program from it, just like it would with NVIDIA PTX binaries generated by the NVPTX back-end.

While this approach does not work for the standard AMD GPU drivers, Radeon Open Compute (ROCm) takes care of this. ROCm is a complete Linux kernel-driver and runtime stack that offers support for our AMDGPU generated binaries. All AMD GPUs from the Fiji family and up are supported [33]. However, ROCm's general hardware support is currently still very limited, making it available only on the newer platforms with compatible host CPUs and the correct operating systems. In addition to that, ROCm is fairly young (2016) and still under heavy development, so there are a number of bugs and things to change. With that being said, this is a very future proof and robust solution though. The hardware support is bound to grow in the future, and our implementation is resistant to changes, since AMDGPU back-end generated code should always be executable on the ROCm stack.

3 LLD - The LLVM Linker: https://lld.llvm.org/

3 EVALUATION

3.1 OPENCL RUNTIME

We begin our evaluation by looking at the difference in performance when switching from the pre-existing CUDA runtime interface to the newly provided OpenCL runtime. Since systems equipped with NVIDIA GPUs can run both CUDA and OpenCL, we can get a fair, direct comparison using the same hardware for both cases. Our test system for this part of the evaluation runs Ubuntu 16.04.3 LTS (with Linux kernel 4.8.0-59) and consists of an Intel Core i7-3930K processor (6 cores, 12 hyper-threads @ 4.5 GHz - Overclocked), 16 gigabytes of RAM and a NVIDIA GTX 680 GPU. CUDA tools is at version 8.0.44 with NVIDIA driver 375.66, and OpenCL version 1.2 running. For a more detailed listing of the CPU and GPU specifications, see Table 1.

To evaluate the differences in performance between using the OpenCL runtime over the CUDA version, we used 28 applications[1] from the polybench 3.2 and 4.2.1 beta test suite [32]. Figure 2 shows the performance gain when using `Polly` to optimize the code and generate GPGPU code for the OpenCL runtime, compared to only using clang's `-O3` optimization flag. In many cases, like the codes of `2mm`, `correlation`, `covariance`, `gemm`, `symm`, `syr2k`, and `syrk`, this leads to massive performance gains when using polybench's large data set. We achieve a 16 fold runtime improvement for `covariance` while `2mm`, `symm`, and `syr2k` each give us 10 times better performance.

1 Since our contributions are added to an active development branch of `Polly`, there have been a number of bugs surfacing during our evaluation. This has led to only 28 of the 30 polybench applications being compilable.

CPU	**i7-3930K**	**GPU**	**GTX 680**
Architecture	SandyBridge-E	Architecture	Kepler
Cores	6	Shader Blocks	8
Hyperthreads	12	Cores per block	192
		Total cores	1,536
Boost clock	4.5 GHz (OC)	Boost clock	1,085 MHz
Performance (f)	307 Gflops	Performance (f)	3,250 Gflops
Memory	16 GB (DDR3)	Memory	2 GB (GDDR5)

Table 1: Hardware specifications for NVIDIA platform

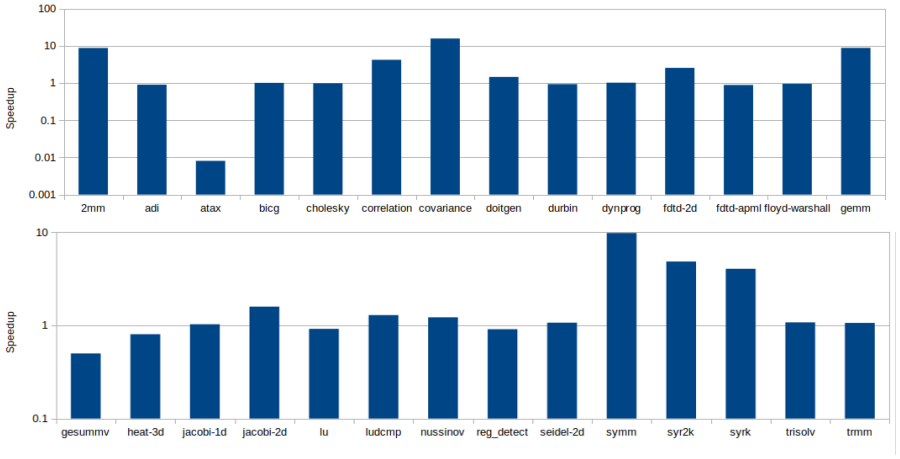

Figure 2: Speedup when using GPGPU generation for NVIDIA with OpenCL over standard compilation without `Polly`

The remaining applications' performance gain is less noticeable at this problem size, but could potentially be much larger when scaling up the data size. It is also interesting to note that the program `atax` suffered from the added polyhedral optimization and GPGPU code generation. On average we get 2.8x better performance when optimizing with Polly and accelerator mapping.

Figure 3 shows the direct performance comparison between the two runtimes in the form of speedup when using OpenCL over CUDA. We can see that 8(/28) programs performed slightly better, and 2(/28) applications ran a little slower. However, the differences are rather marginal, and one can thus say that the performance between the two runtimes is pretty much equivalent. With that we can conclude that the data management library can be directed to use either one of the runtime interfaces on NVIDIA systems without suffering a performance penalty. This also leads us to believe that any performance differences observed in the future on alternative device architectures

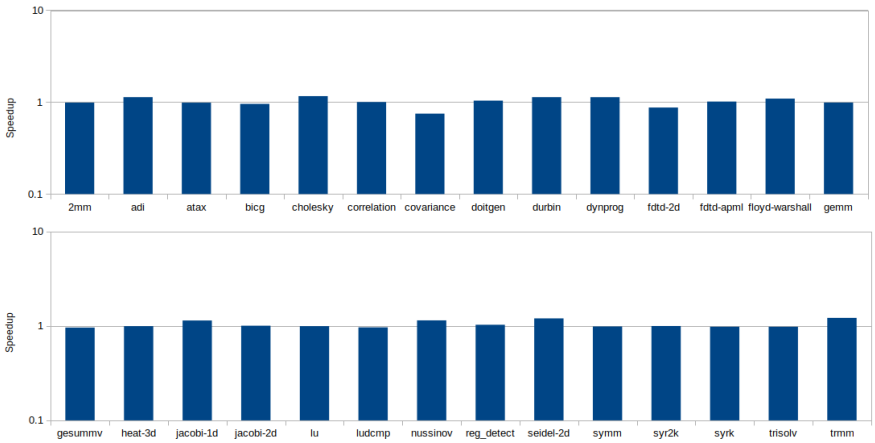

Figure 3: Speedup of using the OpenCL runtime versus the CUDA runtime

(with corresponding code generation provided) in the same performance category can be linked to their architectural differences and not to the use of a different runtime interface.

3.2 SPIR CODE GENERATION ON INTEL

We will now investigate the performance of our Intel SPIR code generation, compared to using `Polly`'s default optimization without accelerator mapping. Our test bench for this is a mobile system that includes an Intel integrated GPU (Intel HD Graphics (IHDG) 5500). It holds an Intel Core i7-5500U processor (2 cores, 4 hyper-threads @ 3.0 GHz Turbo) and 8 gigabytes of RAM. Similarly to our NVIDIA test system we're running Ubuntu 16.04.3 LTS (with Linux kernel 4.4.0-93) and OpenCL version 1.2, as provided by Beignet 1.4. The specific Beignet development version is `git-4933bf9`. Details on the CPU and GPU specifications can be seen in Table 2.

We use a collection of 24[2] polybench 3.2 [32] kernels to evaluate our runtime performance. We tried compiling all the programs with polybench's extra large dataset, but 8 of them (`atax`, `bicg`, `doitgen`, `durbin`, `fdtd-apml`, `gesummv`, `gramschmidt`, and `jacobi-2d`) immediately ran out of memory upon execution, since the Intel platform used does not have enough RAM for those applications with that problem size. The size for those special cases was thus changed to smaller datasets, which worked flawlessly. The result from those mentioned cases is displayed in Figure 4, where we plot the speedup from using GPGPU code generation and running the generated SPIR code with our OpenCL runtime interface via the Beignet driver, compared to standard `-polly` optimization and only clang's `-O3`. For `atax`, `bicg`, `durbin`, `fdtd-apml`, and `gesummv` we notice a very hefty slowdown compared to their non-accelerator-mapped (just `-polly`) counterparts. Overall the programs perform mostly equivalent to when we skip polyhedral optimization entirely (except `atax`, `fdtd-apml`, and `gesummv`). We can conclude that this is with a high probability due to

2 For the same reason mentioned in Section 3.1

CPU	**i7-5500U**	**GPU**	**IHDG 5500**
Architecture	Broadwell-U	Architecture	Broadwell GT2
Cores	2	Cores	192
Hyperthreads	4		
Boost clock	3.0 GHz	Boost clock	950 MHz
Performance (f)	192 Gflops	Performance (f)	364.8 Gflops
Memory	8 GB (DDR3)	Memory	3 GB (DDR3)

Table 2: Hardware specifications for Intel platform

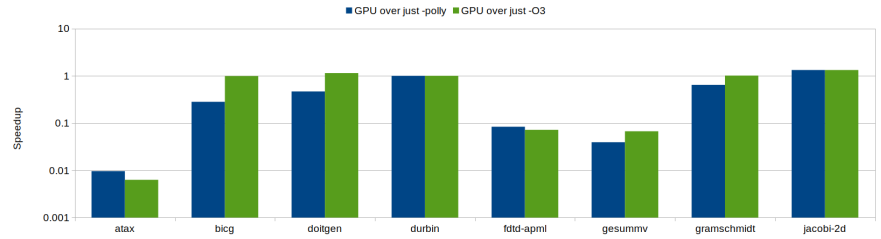

Figure 4: Speedup when using GPGPU generation for Intel over standard `Polly` optimizations and only `-O3` - Smaller datasets

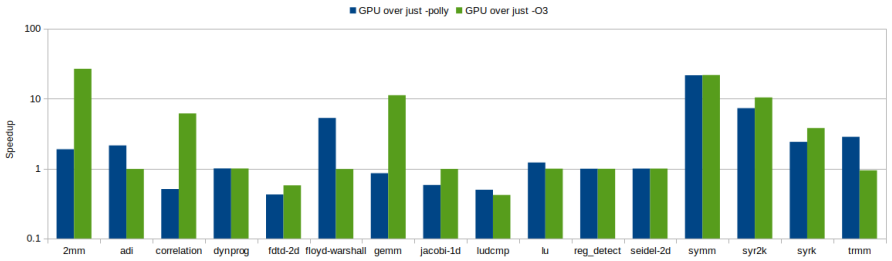

Figure 5: Speedup when using GPGPU generation for Intel over standard `Polly` optimizations and only `-O3` - Extra large dataset

the additional overhead caused by GPGPU code execution, which is negatively noticeable with this relatively small problem size. If there were more RAM available, those programs would most likely perform better once the problem size starts to increase past a certain threshold under which accelerator mapping is not worth it and does not yield any performance gains.

That hypothesis gets backed up when we look at the results from executing our extra large dataset kernels, the speedup of which is plotted in Figure 5. Here, it is visible that only 4 cases (`correlation`, `fdtd-2d`, `jacobi-1d`, and `ludcmp`) experienced a noticeable slowdown when compared to the non-accelerator-mapped versions (`correlation` still outperforms its counterpart without polyhedral optimizations by a factor of 8). For 7 cases the GPU mapping has led to significant improvements in runtime, namely for `2mm`, `adi`, `floyd-warshall`, `symm`, `syr2k`, `syrk`, and `trmm`. In some cases (`symm`) we experience an up to 22x speedup compared to CPU-only optimizations using `Polly`. The remaining kernels performed similarly within the margin of error compared to their non-GPU versions. Overall, including the less than optimal smaller test cases, our average speedup over optimizations using only the `-polly` flag is a factor of 2.3, with a 4x average speedup compared to clang's `-O3` optimizations. When timing those programs using smaller datasets, those improvements drop significantly. This implies that scaled up versions of those programs would probably show even bigger improvements in runtime performance.

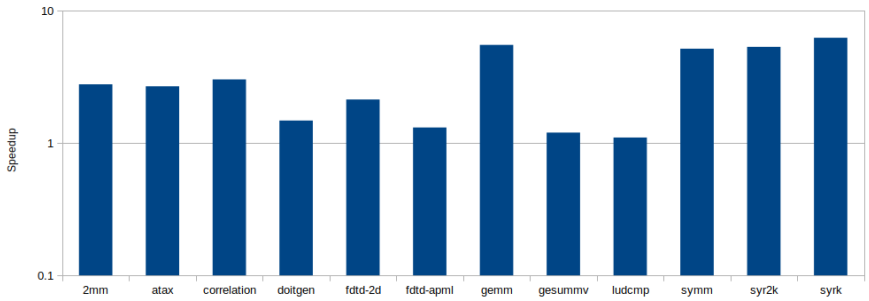

Figure 6: Speedup simulation of exploiting unified memory on Intel platform

While those results are already really good, the Intel platform has another trick up its sleeve. Intel's integrated graphics in Intel CPUs are built according to an unified memory architecture. This stands in contrast to the so far more common Non Unified Memory Architecture (NUMA), where the GPU and host CPU have their own device specific RAM. The implication for systems built on the NUMA model has been that in order for kernels to be executed on an accelerator efficiently, the necessary data first needed to be transferred into GPU specific memory. After the kernel's completion, the result data needs to be copied back into host memory. Both of those transitions obviously cost time, and skipping that step could lead to massive overall performance improvements, especially if the computational effort is rather low in comparison to the amount of data transferred between GPU and host memory.

Our implementation is currently not capable of exploiting unified memory on Intel devices, but nonetheless we wanted to show just how big of an improvement one could achieve by utilizing the same RAM between GPU and host CPU. For this we picked out the 12 polybench kernels where `Polly` decided GPU mapping was worth it, and we disabled the data copying between host and accelerator. This lets us simulate the kernel runtime when `Polly`'s data management library is aware of unified memory architectures and knows how to exploit them. Figure 6 shows the performance improvements of said scenario in the form of speedup over the corresponding NUMA versions. We can see that the runtime gets improved noticeably, leading to an on average 3.2x speedup.

Figure 7 shows the percentage of time spent just copying data between host and accelerator. This obviously stands in stark correlation to Figure 6, and we can see that for some cases (like `gemm`, `symm`, `syr2k`, and `syrk`) up to almost 85% of the execution time gets spent with data transfers. On average the time spent on computation is only about 45%, which means that especially large problems would benefit massively from being able to skip data transfers entirely and instead utilize a zero-copy behavior offered by unified memory architectures

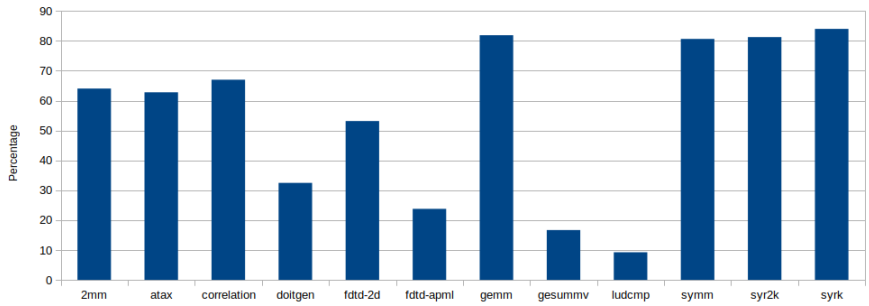

Figure 7: Percentage of kernel runtime spent copying data on Intel platform

like Intel integrated graphics. See Appendix A for some more performance analyses and overviews.

3.3 AMD CODE GENERATION

Our runtime evaluation for AMD devices unfortunately did not turn out the way we anticipated. This is not because we obtained bad results, but rather because we were unable to gather any significant data in the first place. As mentioned in Section 2.4, ROCm's hardware support is still very limited. This led to two of our initial test systems equipped with AMD Fiji (R9 Nano) GPUs to be ruled out shortly after beginning the testing, since parts of ROCm's stack were non-functional. We finally managed to hack together a system that fits all of the ROCm requirements and managed to get test samples running.

Unfortunately, this was not the end of our problems. When starting to run our tests, we quickly noticed that some of them would initially run correctly, but at some point the GPU would fail and cause every program following after that to hang. We immediately got in touch with ROCm developers and tried to resolve this issue. Due to the time limit set for our work, we have however been unable to reach a solution up to this point. The current suspicion is that this is a problem in the base AMDGPU driver inside the kernel. The responsible developers have been made aware of this and are looking into it. The

CPU	**i7-7700**	**GPU**	**R9 Nano**
Architecture	Kaby Lake	Architecture	Fiji
Cores	4	Cores	4,096
Hyperthreads	8		
Boost clock	4.2 GHz	Boost clock	1,000 MHz
Performance (f)	537 Gflops	Performance (f)	8,192 Gflops
Memory	32 GB (DDR4)	Memory	4 GB (HBM)

Table 3: Hardware specifications for AMD platform

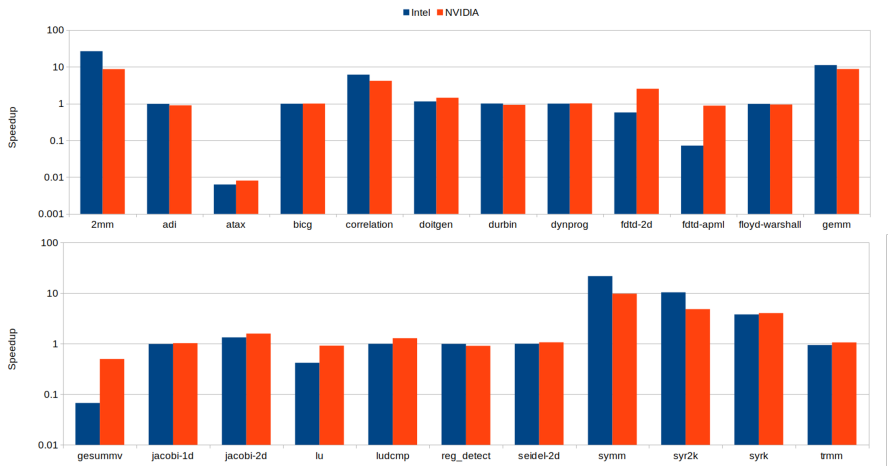

Figure 8: Comparison of the speedup obtained via mapping to accelerator versus clang's `-O3` optimization on the Intel and NVIDIA platform

relevant and necessary evaluation and testing will be conducted as future work, once this issue has been resolved, but can sadly not be included in this report.

It is highly unlikely that our code generation has any impact on this behavior, since everything our code can do is harmless and should be handled by OpenCL. Invalid memory accesses, should they ever occur, would be reported and blocked by the OpenCL runtime, and thus are unlikely to be the cause for the GPU's failure.

The fact that we did not get any evaluation data is especially sad when we take a look at the hardware specifications for our AMD system in Table 3 and compare that to the specifications for our NVIDIA system in Table 1. When considering the already great improvements in runtime from generating GPGPU code for said NVIDIA GPU, one can extrapolate that the performance gain on our AMD system would have been even more impressive. To date, we unfortunately cannot back this up.

3.4 SUMMARY

Each of the platforms targeted by our work (NVIDIA, AMD, Intel) have their own strengths and weaknesses. Performance gains can be made on every one of them, with different implications. NVIDIA GPUs might have more raw power, leading to better performance on compute-heavy kernels, whereas Intel integrated graphics can make use of unified memory, which (in the future) can show a bigger impact on particularly memory-heavy compute kernels.

Figure 8 shows a direct speedup comparison between the 23 polybench 3.2 kernels that were able to be compiled and executed on both our NVIDIA and Intel test systems. It is clearly visible that for most

cases we can observe a very similar trend in terms of how much performance we gained by mapping the kernels to the corresponding systems' accelerator. However, there are slight differences here and there, which might indicate what program types might run more efficiently on what platform. It is important to point out that the speedup shown for the Intel platform here are *including* the memory transfer between host CPU and GPU.

We would like to point out that while we are using very (relatively speaking) simple programs from the polybench test suite [32], our provided solution is very reusable and generally applicable. Any sort of program containing SCoPs can be optimized by `Polly` and mapped to an accelerator device, as long as it belongs to one of the architectures discussed here. This makes it a very powerful tool, as some (particularly poorly optimized) applications might greatly benefit from the massively parallel Single Instruction Multiple Data (SIMD) architecture of GPUs.

4 FUTURE WORK

This work builds a solid foundation for a number of interesting future projects. We would like to propose a few ideas and plans, and point out a few currently ongoing projects related to our work.

4.1 SPIR-V

With our contributions, we tried to address a large number of platforms by providing SPIR code generation, but sadly the approach we had to take is not entirely satisfactory, and will only address Intel devices. We are thus still relatively limited. However, there is currently an ongoing discussion [47] about integrating a SPIR-V target code generation back-end natively into `LLVM`.

SPIR-V is Khronos group's successor standard to SPIR, with a few additional goals. It should easily map to other intermediate languages, and it should be low-level enough to require a reverse-engineering step to reconstruct source code [37], which allows for some form kernel code protection. In addition to that, SPIR-V is standard in the OpenCL 2.1 specification, meaning that every device driver supporting OpenCL 2.1 automatically allows for the creation of kernel programs from SPIR-V binaries with the newly provided API function `clCreateProgramWithIL`.

While not a lot of device drivers support the OpenCL 2.1 standard at this point, the number of supporting architectures is bound to grow in the near future. With many devices allowing kernel creation from SPIR-V, the integration of a corresponding target code generation back-end into `LLVM` in theory implies that the addition of GPU code generation for all supporting architectures should be trivial. We can simply change the back-end specific points mentioned in Section 2.1.2 and Section 2.4 to address the SPIR-V back-end instead of NVPTX or AMDGPU respectively, and add a call to `clCreateProgramWithIL` to our GPU runtime library. This would theoretically eventually open up GPU code generation to all OpenCL (2.1) supporting device architectures.

4.2 PERFORMANCE MODELS

With the capability of compiling the same source code for a larger amount of different GPU architectures and devices, we have the option of creating performance models for our programs based on run-

time evaluations. Such performance models could show potential correlations between our problem type or specific patterns in the source code, and the associated execution time on a particular device architecture. Based on such correlations one could decide on what architecture best to target for a specific program when trying to optimize a particular parameter (like execution time).

4.3 CONCURRENT HETEROGENEITY

It would be interesting to have the capability of executing kernel code on multiple different accelerators concurrently. If a system for example contains an integrated Intel GPU, plus an additional dedicated graphics card (say NVIDIA), it would be great to have the option of harnessing their combined power at once, by splitting the kernel computation up and distributing it amongst the devices. Our contributions lay an important foundation for that, since we can now generate code for both of the devices and embed both architecture specific code modules back into our host binary.

Additionally, the aforementioned performance models could help with the decision on which available compute devices to best utilize, and whether or not parallel execution on multiple accelerators makes sense. This could be done at runtime, allowing for truly portable binaries, or at compile time to shrink binary size but confine portability. Currently we are only able to generate GPU code for one specific device architecture at a time, but for this it would maybe make sense to provide such larger, more portable binaries, which contain multiple different accelerator codes of the same kerenls for different architectures.

4.4 ARCHITECTURE SPECIFIC OPTIMIZATIONS

The original code generation methods used in `Polly` have been designed with CUDA in mind. However, now that we are generating code for different architectures, we are potentially missing out on performance boosts by not exploiting certain device architecture specific strengths and weaknesses. The next step in optimizing our multi-device code generation would be to add some optimizations tailored to the selected architecture(s).

4.5 EXPLOITING UNIFIED MEMORY

In Section 3.2 we have observed the positive impact of unified memory systems on the runtime of GPU-mapped programs with and Intel integrated graphics chip. However, Intel is not the only platform offering an alternative to the traditional NUMA model. AMD and Intel both

offer chips featuring an unified memory architecture with their AMD APUs and Intel integrated graphics respectively. For both of these platforms it would thus be very interesting and massively beneficial to extend `Polly`'s GPU data management library with the capability of detecting said unified memory scenario and exploiting it.

Thankfully, this is not a very hard thing to do. In fact, it only involves telling OpenCL to use host memory when typically allocating device RAM, and then supplying it with a valid host memory pointer. This can easily be done in `Polly` with a bit of minor refactoring. Since this was not possible anymore in our work due to timely restrictions, we will leave this as feature work. Implementing such a zero-copy behavior can lead to huge speedups, particularly for kernels that spend almost 85% of their runtime just copying data to and from GPU memory, as observed in Section 3.2. For more information on Intel's memory model and how to easily exploit it for OpenCL, see Lake [15].

5 RELATED WORK

There are many solutions of mapping code to the heterogeneous accelerator space. The most obvious ones are those previously discussed here, namely OpenCL [39] and CUDA [26]. Using one of those two approaches gives the necessary flexibility and almost direct control over the GPU. However they drastically increase the complexity of GPGPU programs by introducing separate compute kernels and various boilerplate code. There are a number of approaches that try to simplify this procedure by providing directive statements. Such approaches like OpenARC [23], OpenMPC [22], OpenACC [46], OpenMP for accelerator [9], and HMPP [12] reduce the complexity by quite a lot by providing pragma based statements to direct the generation and handling of accelerator code. `Polly` contrasts this by eliminating the need for annotations, instead automatically detecting SCoPs in the program source code and deciding on a beneficial mapping strategy [16].

However, `Polly` is not the only project trying to detect potential parallelism in the source code automatically, and then mapping it to GPU code. Other polyhedral approaches include Baskaran's work [5] which can be found in the R-Stream compiler [24] in its improved form. The most refined solution so far is PPCG [43, 45], but it still relies on preprocessed code to be effective. `Polly` and its GPGPU code generation are the first solution that bring advanced GPU mapping techniques to a large set of programs by not enforcing specific coding styles and by automatically choosing when to perform GPU mapping [16]. Other approaches not based on polyhedral optimization techniques include Par4All [3], which uses an abstract interpretation based approach, and a proposal made by Baghdadi et al. [4], which has proven to be too costly in practice though.

6 CONCLUSION

Our contributions provide an easy way of compiling GPGPU code retrieved from `Polly` for not just NVIDIA accelerators, but also AMD and Intel devices. While our AMD-ready binaries are at the moment still relatively poorly evaluated and the ROCm stack brings its own set of problems and limitations, the chosen approach should in the future be a rather robust and stable solution. In addition to that, access to the AMD platform can potentially bring runtime improvements for certain kernels, since the architecture might suit the required computations better. Plus, while ROCm's support for AMD APUs with unified memory architectures is currently still limited, the developers are at work with extending said compatibility, which in the future adds capabilities of utilizing unified memory for AMD compute kernels.

Our SPIR code generation is sadly limited to the Intel platform and relies on an alternative driver (Beignet) to work. But having access to an array of Intel accelerator devices is very interesting, especially when trying to leverage unified memory in the future. As seen in Section 3.2, even on a relatively low-memory and low-power Intel mobile system, there are a number of computational problems that can greatly benefit from such an accelerator mapping. The successful generation of SPIR compliant kernel IR also serves as a proof of concept for the potential extension of SPIR code generation in the future, showing that it is a viable option for delivering device independent binaries to supporting drivers and architectures.

Finally, our OpenCL runtime interface in `Polly`'s data management library builds an important foundation for lots of future additions to the accelerator code generation and creation of truly multi-device binaries. Using the newly provided OpenCL runtime interface over the previous CUDA API does not impact our program runtime, nor does it slow down compilation. This adds an immense amount of flexibility and is the basis on which SPIR-V code generation mentioned in Section 4.1 will or would be built.

In conclusion, while the current state of `Polly`'s multi-device GPU code generation is not yet entirely satisfactory, we can already show great success in adding an additional device architecture (Intel) to the repertoire and delivering runtime improvements on it. A third architecture (AMD) remains yet to be fully evaluated, but the provided code generation shows that targets with a back-end integrated into `LLVM` are relatively easy to add in the future. `Polly` is now one step closer to having multi-device accelerator mapping and has had its doors opened for lots of interesting ideas and extensions.

Part III

APPENDIX

APPENDIX

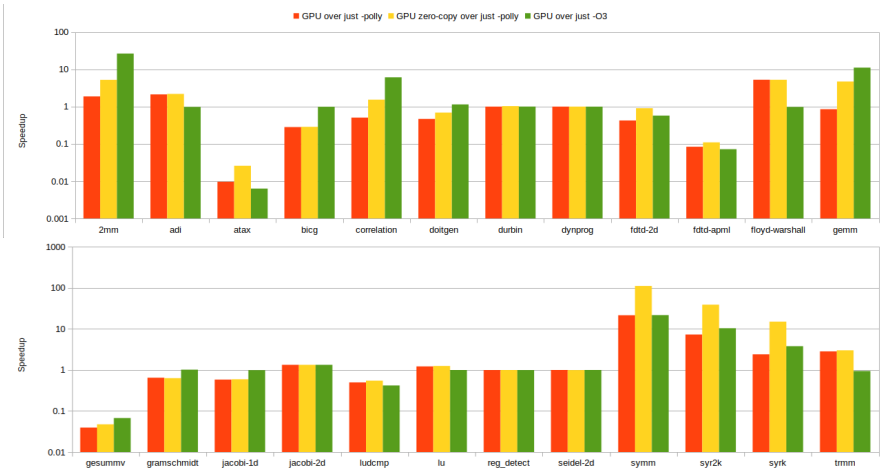

Figure 9: Performance gain of GPU mapping with and without zero-copy behavior on Intel platform, compared to CPU-only optimization (`-polly`) and clang's `-O3`

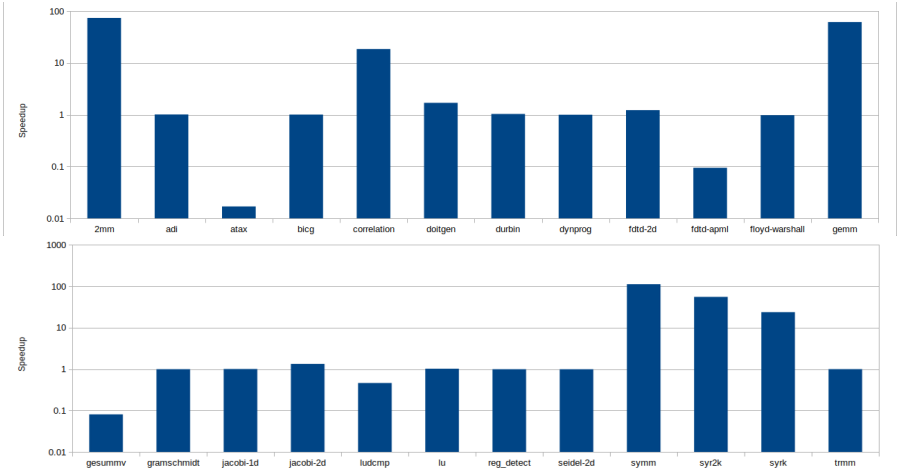

Figure 10: Speedup of GPU mapping with zero-copy behavior over clang's `-O3` on Intel platform

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

MULTI-DEVICE GPU-CODE GENERATION WITH LLVM

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
SCHAAD	PHILIPP

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
ZÜRICH, 11.09.2017	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.

BIBLIOGRAPHY

[1] *AMD Graphics Core Next Architecture, Generation 3. Reference Guide*. 2017. URL: http://developer.amd.com/wordpress/media/2013/12/AMD_GCN3_Instruction_Set_Architecture_rev1.1.pdf (visited on 09/10/2017).

[2] *AMDGPU Backend*. 2017. URL: https://llvm.org/docs/AMDGPUUsage.html#introduction (visited on 09/10/2017).

[3] Mehdi Amini, Béatrice Creusillet, Stéphanie Even, Ronan Keryell, Onig Goubier, Serge Guelton, Janice Onanian McMahon, François-Xavier Pasquier, Grégoire Péan, and Pierre Villalon. "Par4all: From convex array regions to heterogeneous computing." In: *IMPACT 2012: Second International Workshop on Polyhedral Compilation Techniques HiPEAC 2012*. 2012.

[4] Soufiane Baghdadi, Armin Größlinger, and Albert Cohen. "Putting automatic polyhedral compilation for GPGPU to work." In: *Proceedings of the 15th Workshop on Compilers for Parallel Computers (CPC'10)*. 2010.

[5] Muthu Baskaran, Jj Ramanujam, and P Sadayappan. "Automatic C-to-CUDA code generation for affine programs." In: *Compiler Construction*. Springer. 2010, pp. 244–263.

[6] Cédric Bastoul. "Code Generation in the Polyhedral Model Is Easier Than You Think." In: *PACT'04 IEEE International Conference on Parallel Architecture and Compilation Techniques* (Sept. 2004), pp. 7–16.

[7] *Beignet*. 2017. URL: https://www.freedesktop.org/wiki/Software/Beignet/ (visited on 09/10/2017).

[8] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. "The Polyhedral Model Is More Widely Applicable Than You Think." In: *ETAPS International Conference on Compiler Construction* (Mar. 2010), pp. 283–303.

[9] James Beyer, Eric Stotzer, Alistair Hart, and Bronis de Supinski. "OpenMP for accelerators." In: *OpenMP in the Petascale Era* (2011), pp. 108–121.

[10] Shekhar Borkar and Andrew A. Chien. "The Future of Microprocessors." In: *Communications of the ACM* 54.5 (May 2011).

[11] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. "Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions." In: *IEEE Journal of Solid-State Circuits* 9.5 (Oct. 1974).

[12] Romain Dolbeau, Stéphane Bihan, and François Bodin. "HMPP: A hybrid multi-core parallel programming environment." In: *Workshop on general purpose processing on graphics processing units (GPGPU 2007)*. Vol. 28. 2007.

[13] Paul Feautrier. "Dataflow Analysis of Array and Scalar References." In: *International Journal of Parallel Programming* 20.1 (Feb. 1991), pp. 23–53.

[14] Paul Feautrier and Christian Lengauer. "Polyhedron Model." In: *Encyclopedia of Parallel Computing* (2011), pp. 1581–1592.

[15] *Getting the Most from OpenCL 1.2. How to Increase Performance by Minimizing Buffer Copies on Intel Processor Graphics*. 2014. URL: https://software.intel.com/en-us/articles/getting-the-most-from-opencl-12-how-to-increase-performance-by-minimizing-buffer-copies-on-intel-processor-graphics (visited on 07/02/2017).

[16] Tobias Grosser and Torsten Hoefler. "Polly-ACC: Transparent compilation to heterogeneous hardware." In: *Proceedings of the 30th International Conference on Supercomputing (ICS'16)*. June 2016.

[17] Tobias Grosser, Sven Verdoolaege, and Albert Cohen. "Polyhedral AST generation is more than scanning polyhedra." In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 37.4 (2015), p. 12.

[18] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Grösslinger, and Louis-Noël Pouchet. "Polly - Polyhedral optimization in LLVM." In: *IMPACT* (2011).

[19] Richard Johnson, David Pearson, and Keshav Pingali. "The Program Structure Tree: Computing Control Regions in Linear Time." In: *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*. Vol. 29. 6. ACM. 1994, pp. 171–185.

[20] *LLVM. The LLVM Compiler Infrastructure*. 2017. URL: http://llvm.org (visited on 07/02/2017).

[21] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation." In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. CGO '04. IEEE Computer Society, 2004, pp. 75–.

[22] Seyong Lee and Rudolf Eigenmann. "OpenMPC: Extended OpenMP programming and tuning for GPUs." In: *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society. 2010, pp. 1–11.

[23] Seyong Lee and Jeffrey S Vetter. "OpenARC: open accelerator research compiler for directive-based, efficient heterogeneous computing." In: *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. ACM. 2014, pp. 115–120.

[24] Allen Leung, Nicolas Vasilache, Benoît Meister, Muthu Baskaran, David Wohlford, Cédric Bastoul, and Richard Lethin. "A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction." In: *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. ACM. 2010, pp. 51–61.

[25] G. E. Moore. "Cramming more components onto integrated circuits." In: *Electronics* 38.8 (Apr. 1965).

[26] *NVIDIA Accelerated Computing. CUDA Zone*. 2017. URL: https://developer.nvidia.com/cuda-zone (visited on 07/20/2017).

[27] *OpenCL Conformant Products. List of Products Conforming to the OpenCL standard*. 2017. URL: https://www.khronos.org/conformance/adopters/conformant-products#opencl (visited on 08/11/2017).

[28] *OpenCL. The open standard for parallel programming of heterogeneous systems*. 2017. URL: https://www.khronos.org/opencl/ (visited on 09/10/2017).

[29] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Philips. "GPU Computing." In: *Proceedings of the IEEE* 96.5 (May 2008).

[30] *Polly. LLVM Framework for High-Level Loop and Data-Locality Optimizations*. 2017. URL: http://polly.llvm.org (visited on 07/02/2017).

[31] *Polyhedral Software. Tools and libraries to translate a polyhedral representation into source code*. 2017. URL: http://www.polyhedral.info/software.html (visited on 09/10/2017).

[32] Louis-Noël Pouchet. *Polybench/C. the Polyhedral Benchmark suite*. 2017. URL: web.cse.ohio-state.edu/~pouchet.2/software/polybench/ (visited on 09/10/2017).

[33] *ROCm, a New Era in Open GPU Computing. Platform for GPU-Enabled HPC and Ultrascale Computing*. 2017. URL: https://rocm.github.io/install.html (visited on 09/10/2017).

[34] Jon Peddie Research. *Mobile Devices and the GPUs Inside*. Report. Jon Peddie Research, Oct. 2013.

[35] *SPIR 1.2 Specification for OpenCL*. 2013. URL: https://www.khronos.org/files/opencl-spir-12-provisional.pdf (visited on 09/09/2017).

[36] *SPIR Generator*. 2017. URL: https://github.com/KhronosGroup/SPIR (visited on 09/10/2017).

[37] *SPIR-V Specification*. 2017. URL: https://www.khronos.org/registry/spir-v/specs/1.0/SPIRV.pdf (visited on 09/10/2017).

[38] *SPIR(V). The first open standard intermediate language for parallel compute and graphics*. 2017. URL: https://www.khronos.org/spir (visited on 07/02/2017).

[39] John E. Stone, David Gohara, and Guochun Shi. "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems." In: *Computing in Science and Engineering* 12.3 (May 2010), pp. 66–73.

[40] *The OpenCL Extension Specification*. 2016. URL: https://www.khronos.org/registry/OpenCL/specs/opencl-2.0-extensions.pdf (visited on 09/10/2017).

[41] *The SPIR Specification*. 2014. URL: https://www.khronos.org/registry/SPIR/specs/spir_spec-2.0.pdf (visited on 09/10/2017).

[42] Sven Verdoolaege. "isl: An Integer Set Library for the Polyhedral Model." In: *Mathematical Software - ICMS 2010*. Lecture Notes in Computer Science 6327 (2010), pp. 299–302.

[43] Sven Verdoolaege. "PENCIL support in pet and PPCG." PhD thesis. INRIA Paris-Rocquencourt; INRIA, 2015.

[44] Sven Verdoolaege. *Presburger formulas and polyhedral compilation*. 2016. URL: http://polyhedral.info/2016/01/26/tutorial.html (visited on 07/12/2017).

[45] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. "Polyhedral parallel code generation for CUDA." In: *ACM Trans. Archit. Code Optim.* 9.4 (Jan. 2013), 54:1–54:23. ISSN: 1544-3566. DOI: 10.1145/2400682.2400713.

[46] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. "OpenACC—first experiences with real-world applications." In: *Euro-Par 2012 Parallel Processing* (2012), pp. 859–870.

[47] Nicholas Wilson. *[llvm-dev] [SPIR-V] SPIR-V in LLVM*. 2017. URL: http://lists.llvm.org/pipermail/llvm-dev/2017-May/112538.html (visited on 09/09/2017).