TECHNISCHE
UNIVERSITÄT
DARMSTADT

An alternative OpenMP-Backend for `Polly`

Bachelorthesis by

Michael Halkenhäuser

March 2019

Examiner:
Prof. Dr.-Ing. Andreas Koch

Supervisor:
Lukas Sommer, M.Sc.

Technische Universität Darmstadt
Department of Computer Science
Embedded Systems and Applications Group (ESA)

An alternative OpenMP-Backend for `Polly`
Ein alternatives OpenMP-Backend für `Polly`
Bachelorthesis by Michael Halkenhäuser
Submitted on 05.03.2019
Examiner: Prof. Dr.-Ing. Andreas Koch
Supervisor: Lukas Sommer, M.Sc.

Thesis Statement

I herewith formally declare that I, Michael Halkenhäuser, have written the submitted thesis independently. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. This thesis has not been handed in or published before in the same or similar form. I am aware, that in case of an attempt at deception based on plagiarism (§ 38 Abs. 2 APB), the thesis would be graded with 5,0 and counted as one failed examination attempt. The thesis may only be repeated once. In the submitted thesis the written copies and the electronic version for archiving are identical in content.

Darmstadt, 5. March 2019

(Michael Halkenhäuser)

Acknowledgements

Heartfelt thanks to my family and friends, who supported me throughout the past few years and encouraged me to press on. This work would probably not have happened without these people. I am particularly grateful for the motivational assistance provided by my better half, and all who assisted in shaping the writing style of this document.

Regarding my academic background I would like to thank my supervisor Lukas Sommer, who patiently supported me, and Prof. Dr. Andreas Koch who initially triggered my interest in compiler technologies – especially polyhedral compilation.

Abstract

Multicore architectures have found their way into many areas of application by now. While this allows for the execution of several tasks in parallel, software still has to be adapted for the specific architectures to utilize the available resources effectively. Thus, the development of code that may be run in parallel is oftentimes left to human experts, who are faced with the challenge of supporting different systems and their peculiarities.

While there are standardized means to realize multithreaded software more easily, like for example OpenMP, it still remains a tedious and time-consuming task. Additionally, a programmer may introduce severe errors rather quickly, if the software is not carefully engineered. Fortunately, automatic tools exist which are based on a specific mathematical representation known as the polyhedral model. On the one hand, such representations may only describe certain code structures, since they are based on linear expressions. On the other hand, this allows to exactly define and test what may be parallelized, due to correct analysis results, as for example dependency analyses. Furthermore, powerful program transformations can be defined in a very abstract manner, using methods from linear algebra.

One of these tools is Polly, which may automatically generate parallelized code without any manual preparation. Polly is a subproject of the LLVM compiler framework and operates solely on a low-level intermediate representation. This brings several advantages since this representation is language independent and can be deployed on multiple platforms.

However, the generation of multithreaded code is currently limited to a specific OpenMP runtime environment. In this work we will therefore present an extension to the existing infrastructure, which enables the use of an additional implementation and therefore expands Polly's field of application.

Kurzfassung

Mittlerweile verfügen moderne Prozessorarchitekturen verschiedenster Anwedungsbereiche über eine Mehrzahl unabhängiger Ausführungseinheiten. Zwar erlaubt dies die parallele Ausführung mehrerer Aufgaben, erfordert aber im Gegenzug die Anpassung der Software auf die eingesetze Hardware-Architektur, um eine effektive Nutzung der Ressourcen zu gewährleisten. Daher wird die Entwicklung parallel laufender Programme meist Experten überlassen, welche sich mit der Aufgabe konfrontiert sehen verschiedene Systeme und ihre Eigenheiten zu unterstützen.

Obwohl es standardisierte Programmierschnittstellen gibt, welche die Realisierung paralleler Software erleichtern, wie beispielweise OpenMP, bleibt dies eine mühsame und langwierige Angelegenheit. Zusätzlich kann ein Programmierer sehr schnell schwerwiegende Fehler einführen, wenn die Software nicht sorgfältig entwickelt wird. Glücklicherweise existieren automatische Werkzeuge, die auf einer speziellen mathematischen Darstellung, dem sogenannten polyhedralen Modell, basieren.

Einerseits kann diese Darstellungsform nur bestimmte Code Strukturen beschreiben, da sie auf linearen Ausdrücken basiert. Andererseits erlaubt ebendiese Einschränkung eine exakte Definition und Evaluation von Programmteilen die parallel ausgeführt werden können. Derartige Informationen liefert beispielsweise eine rigorose, im polyhedralen Modell formulierte Abhängigkeitsanalyse. Außerdem können mithilfe der linearen Algebra ausdrucksstarke Programmtransformationen sehr abstrakt definiert werden.

Eines dieser Werkzeuge ist Polly, welches gänzlich ohne manuelle Anpassungen der Quelldateien parallelen Code erzeugen kann. Polly ist ein Teilprojekt des LLVM Compiler Frameworks und operiert ausschließlich auf einer hardwarenahen Zwischendarstellung. Dies bietet gleichzeitig den Vorteil der Sprachunabhängigkeit und die Möglichkeit den resultierenden Code auf mehreren Platformen einzusetzen.

Jedoch ist die derzeitige Generierung von parallelem Code auf eine bestimmte OpenMP Laufzeitumgebung beschränkt. In dieser Arbeit stellen wir deshalb eine Erweiterung der bereits bestehenden Infrastruktur vor, die es ermöglicht eine weitere Implementierung zu nutzen und so das Anwendungsgebiet von Polly erweitert.

Contents

1 Introduction

With the widespread availability of multi-core central processing units (CPUs) comes a natural demand for parallelized software. As such, exploiting possibilities for parallel execution of programs has become important not only in High Performance Computing (HPC) but also on mobile devices. In addition to the use of multiple cores at the same time there are so-called Single Instruction Multiple Data (SIMD) instructions. They allow single cores to perform one specific operation on multiple data values at once, while the data size can vary from single bytes to several machine words. Using those different kinds of parallelism is crucial when it comes to an efficient use of the available compute power and achieving a reasonable workload.

Current compilers are capable of many sophisticated analyses and optimization passes. Nevertheless, annotating source code in order to achieve parallel execution, e.g. by means of OpenMP pragmas, is often still left to human experts. The situation is complicated even further when taking into account that the involved instruction sets are platform dependent in most cases, e.g. AVX2 on Intel/AMD and NEON on ARM architectures. This results in the necessity of optimizing the considered program for each target platform individually.

Furthermore, it should be noted that neither exposing nor exploiting parallelism is trivial. In general it is important to keep in mind that not every part of a program can be parallelized; for example as a result of instructions which depend on values that have to be computed first. Assuming that in theory it might be possible to transform the program and eliminate the dependencies which prevent parallel execution, it may prove to be difficult and costly. In addition to these changes, the programmer will now at least have to (manually) tell the compiler which code sections should run in parallel.

However, if those modifications are not done correctly, the program semantics will most likely have changed, ultimately leading to errors during runtime. But especially testing and debugging code that is run in parallel is a tedious task and might consume a large amount of development time.

Consequently, it is highly desirable to offload the process of optimizing a program for parallel execution to an automatic tool, which will transform the given program correctly. And while tools exist that are able to perform such optimizations, they tend to operate on specific program structures only.

A promising approach to realize such an automatic tool, is using a mathematical representation known as the *polyhedral model* [1][10]. Program parts which can be expressed in the polyhedral model are amenable to analyses and methods from the branch of linear algebra. Thus, *only structures which can be represented as linear transformations*[1] can be taken into account. In exchange for a rather narrow range of application, this allows mathematical reasoning about the correctness of performed modifications and even traversing solution spaces to discover different optimizations [12][13]. Moreover, it is possible to identify dependencies between specific program instructions with relative ease – enabling the exposition of parallelism at coarse- (OpenMP) and fine-grained level (SIMD) [3]. With this information it is possible to realize automatic parallelization for the regarded code region. But many tools only support a small set of programming languages or do not allow influencing the optimization process. This limits the portability of the generated code.

In the course of this work we will concentrate on the tool *Polly*[2] [5]. The fact that Polly performs all of its transformations on a language- and target-independent representation makes it particularly interesting. As a subproject of the Low Level Virtual Machine (LLVM) compiler infrastructure it uses the corresponding LLVM Intermediate Representation (LLVM-IR). Therefore it is possible to support software projects where multiple programming languages are involved and generate optimized code for different platforms without changes to the source files. This implies the realization of thread-level parallelism by using OpenMP API library calls [14], instead of pragmas. While OpenMP is merely a standard, there are multiple implementations available as for example from GNU, Intel and LLVM.

At the time of writing Polly utilizes the GNU OpenMP library to generate code that is optimized for parallel execution on multiple CPU cores. And while this might not be a drawback for most developers, it would not allow projects without GNU library support to take advantage of the automatic parallelization. One of these projects is the Nymble system [7] used by the Embedded Systems and Applications (ESA) workgroup at TU Darmstadt.

Hence, we decided to extend on the work of Raghesh A [14] to enable support of the LLVM OpenMP implementation for Polly, by providing an alternative backend.

[1] This is a simplified statement – for specific limitations see Section 2.1.1

[2] Which is a combination of the words **Poly**hedral and **LL**VM

This thesis is organized as follows. Chapter 2 will provide general knowledge about polyhedral representations and how transformations may be realized in this mathematical model before we move on to OpenMP, LLVM and Polly in particular. After Polly has been introduced, Chapter 3 will present how parallelized code is automatically generated by its GNU OpenMP-backend. Afterwards our alternative backend will be presented in Chapter 4, which implements the same functionalities and utilizes the LLVM OpenMP library to provide thread-level parallelism. Chapter 5 presents the experimental setup and evaluates the performance of our alternative backend in various scenarios. Among other things we will compare our results to the existing implementation. Finally, we conclude in Chapter 6.

2 Background

2.1 Polyhedral Model

While the theory on which the polyhedral model and therefore polyhedral compilation is based upon dates back into the 1960s [8], tools utilizing this theory were first developed around 1990. Since then the development continued and has received a notable boost, particularly with regard to the launch of the first consumer dual core CPUs in 2005 and the integration of polyhedral techniques in popular compilers such as the GNU Compiler Collection (GCC) [11]. The following sections will provide further information on when and how specific source code structures can be transformed into a polyhedral representation.

2.1.1 From Source Code to Mathematical Representation

First we will define those *specific source code structures*, which are henceforth called Static Control Parts (SCoPs). An abstract definition would be that SCoPs are the parts of a program which can be represented using the polyhedral model. A precise definition is provided in section 5.1 of [5].

Among other things a SCoP is defined as part of a program where control-flow is realized using for-loops and if-conditions only. Each loop utilizes a single integer induction variable which is constantly incremented from a lower bound to an upper bound. Those bounds may be integer constants or affine expressions utilizing variables which are not manipulated inside the corresponding SCoP. If-conditions may only consist of the comparison of two affine expressions, meaning they can be expressed as a linear equation. There exists only a single allowed type of statement inside a SCoP: an expression may be assigned to an array-element (addressed by an affine expression). Multiple instances of such statements are allowed but the assigned expression is further restricted as follows: operators or functions used have to be side effect free and operands are limited to variables, parameters or other array elements.

While this might suggest that SCoPs are a scarce phenomenon, Figure 2.1 provides a valid example. Additionally there are efforts to enlarge the scope of polyhedral optimizations [2]. One possibility is for example to use sequences of compiler passes to transform the code into a so-called *canonical* form, which is compliant to the aforementioned restrictions.

```
1 for (i = 0; i <= n; i++) {
2   s[i] = 0;                          // Statement R
3   for (j = 0; j <= n; j++)
4     s[i] = s[i] + a[i][j] * x[j];    // Statement S
5 }
```

Figure 2.1: `matvect` Kernel

The code fragment above is a prominent example of a SCoP in C++ (taken from [12]). It realizes a simple matrix-vector multiplication and we will use it to demonstrate some basic steps. At first, we will take a look at the two statements R and S. Since both are writing to the same array element they may not be executed in parallel. To begin with the transformation we formulate inequalities, which can be extracted from each conditional statement. This will yield a set (of e.g. integer tuples) representing the iterations of each statement. These sets are called the *iteration domains* $\mathcal{D}_R : \{i_R \mid 0 \leq i_R \leq n\}$ and $\mathcal{D}_S : \{i_S, j_S \mid 0 \leq i_S \leq n \wedge 0 \leq j_S \leq n\}$ of their respective statement. Both these sets can again be formulated as matrices, which express the exact same inequalities:

$$\mathcal{D}_R : \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \end{bmatrix} \cdot \begin{pmatrix} i_R \\ n \\ 1 \end{pmatrix} \geq \vec{0} \qquad \mathcal{D}_S : \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 1 & 0 \end{bmatrix} \cdot \begin{pmatrix} i_S \\ j_S \\ n \\ 1 \end{pmatrix} \geq \vec{0}$$

Figure 2.2: `matvect` Iteration Domains

The number of different induction variables, one (i_R) for statement R and two (i_S, j_S) for S, can be interpreted as the dimension of the corresponding iteration space. While the parameter n however, will only change its size in this example. These spaces consist of (integer) points, called *iteration vectors*, representing each single iteration of a statement.

The general, mathematical description of the resulting solid is a *polyhedron*, hence the name polyhedral model (or *polytope*[1] model).

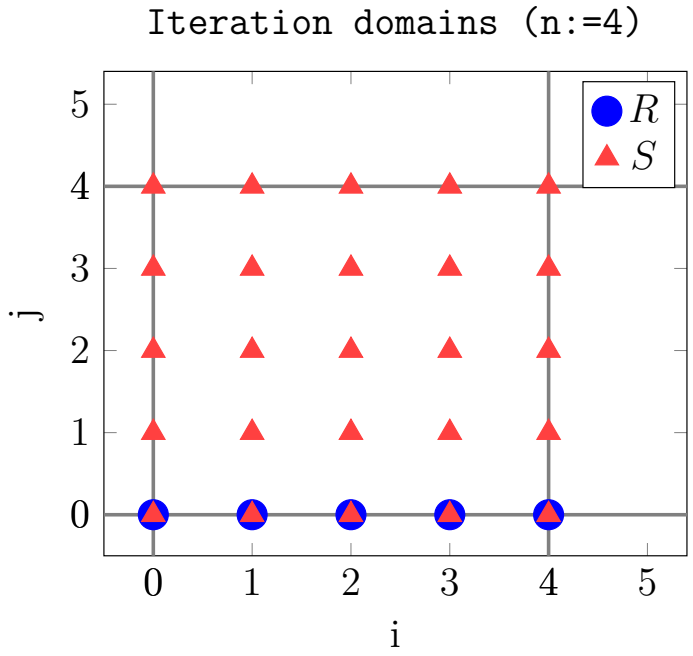

Figure 2.3: `matvect` Iteration Domains $\mathcal{D}_R$ and $\mathcal{D}_S$ for $n = 4$

Figure 2.3 shows a stacked plot of both iteration domains for $n = 4$ and we can easily identify the iterations in which concurrent writes to the array would occur. Also note the gray lines depicting the polytope bounds that emerge from the inequality relations. However, since polytopes can be of arbitrary dimension, it is reasonable to identify these bounds as hyperplanes. While all of these visual observations might be *correct*, we need a mathematical expression which is able to identify the conflicting statement instances, i.e. the intersection of all iteration domains. Therefore we need to expand and adapt each of our inequalities from Figure 2.2 to cope with every induction variable involved.

[1] A bounded (finite) polyhedron is called a *polytope*.

$$\mathcal{D}_R : \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{pmatrix} i_R \\ i_S \\ j_S \\ n \\ 1 \end{pmatrix} \geq \vec{0} \qquad \mathcal{D}_S : \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 \end{bmatrix} \cdot \begin{pmatrix} i_R \\ i_S \\ j_S \\ n \\ 1 \end{pmatrix} \geq \vec{0}$$

Figure 2.4: `matvect` Iteration Domains (expanded)

In addition to the inequalities in Figure 2.4, we need to check for an equality of the shared induction variables because instances of R and S conflict (with regard to the parallel writing of `s`) if and only if $i_R = i_S$ holds[2]. Now we are able to define the *dependency domain* $\mathcal{D}_{R,S}$, which consists of all points formed by the intersection of $\mathcal{D}_R$, $\mathcal{D}_S$ and satisfy $i_R = i_S$:

$$\mathcal{D}_{R,S} : \begin{bmatrix} 1 & -1 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 \end{bmatrix} \cdot \begin{pmatrix} i_R \\ i_S \\ j_S \\ n \\ 1 \end{pmatrix} \geq \vec{0}$$

Figure 2.5: `matvect` Dependency Domain

If this space proves to be empty, i.e. no feasible solution vector(s) exist that satisfy the inequality, the statements can safely be run in parallel without changes to the execution order.

Otherwise we need to preserve the sequence of the respective iteration instances, or this would most likely change the program's semantics. An equivalent expression, where the *schedule* $\mathcal{S}(\vec{x})$ provides the logical (multidimensional) execution time of a given iteration vector, would be:

$$\mathcal{S}_R(\vec{x}_R) < \mathcal{S}_S(\vec{x}_S) \ \forall \vec{x}_R, \vec{x}_S \in \mathcal{D}_{R,S} \tag{2.1}$$

[2]For notational reasons we will express this single formula by two equivalent inequalities.

However, a non-empty dependency domain does not necessarily imply the absence of a valid, parallel schedule. In this thesis we will omit the one-dimensional case (see e.g. [12]) but rather concentrate on multidimensional execution dates [13]. For now we will only define a sequential schedule (Figure 2.6), which will trivially comply to (Equation (2.1)) and therefore be valid. In Section 2.1.2 we will take a closer look at parallelizing the statements.

$$\mathcal{S}_R = \{R[i] \to [i, 0]\}$$
$$\mathcal{S}_S = \{S[i, j] \to [i, 1, j]\}$$

Figure 2.6: `matvect` Schedule

The last construct we are about to introduce are *access functions*. They are mapping a given iteration vector to a list of memory accesses and enable for instance the detection of sequential reads. For our `matvect` example the access functions will look like this:

$$\begin{aligned}\mathcal{A}_R &= \{R[i] \to s[i]\} \\ \mathcal{A}_S &= \{S[i, j] \to s[i], \\ &\qquad S[i, j] \to A[i][j], \\ &\qquad S[i, j] \to x[j]\}\end{aligned}$$

Figure 2.7: `matvect` Access Functions

Combining all of these definitions we are able to represent a given SCoP[3]. Furthermore we can identify possible conflicts, reorder the execution of statements and as such efficiently describe valid program transformations using the polyhedral model.

[3]See Section 2.2.2 on how SCoPs are detected

2.1.2 Optimizations in Theory

While SCoPs can be optimized using different approaches, with changing the execution order and the sequence or pattern of memory accesses being the most popular ones, we will concentrate on scheduling-based optimizations. This is due to the fact that Polly relies on scheduling transformations [5][6]. The schedule represents a program's temporal sequence of statement executions, consequently this approach realizes optimizations solely by permuting these logical execution dates. One possible goal could be to resolve dependencies, that may block SIMD instructions or thread-level parallelization.

In this section we will demonstrate three interesting optimization examples of polyhedral transformations from section 6 of [5]. We will also adapt to the notation used there and in [12], with "$\circ$" being the application of a transformation "$\mathcal{T}$" and "θ" representing a function which takes a point in time and returns an actual execution date.

Loop Fission

Loop nests may incur dependencies and as such hide or block possibilities for parallel execution. Scattering the contained statements over multiple loops or loop nests might resolve these dependencies; this procedure is called *loop fission*.

Assuming a program structure like in our `matvect` example (Figure 2.1):

```
1  for (i = 0; i <= K; i++) {
2    R(i);
3    for (j = 0; j <= L; j++)
4      S(i, j);
5  }
```

Figure 2.8: Loop Fission Example

We have seen (Figure 2.3) that these nested loops carry a dependency which prevents them from being executable in parallel. Now we are about to split them into two distinct loops which consist of one statement each.

For this purpose we will make use of our multidimensional time specification and augment the existing schedule with an additional dimension. This additional *coordinate* is set to 0 for statement R and 1 for S, so they still comply to Equation (2.1).

$$
\begin{aligned}
\mathcal{D}_R &= \{i_R \mid 0 \leq i_R \leq n\} \\
\mathcal{D}_S &= \{i_S, j_S \mid 0 \leq i_S \leq K \ \wedge \ 0 \leq j_S \leq L\} \\
\mathcal{S}_R &= \{R[i] \to \theta[i, 0, 0]\} \\
\mathcal{S}_S &= \{S[i, j] \to \theta[i, 1, j]\} \\
\mathcal{T}_{Fission_R} &= \{\theta[t_0, t_1, t_2] \ \to \ \theta[0, t_0, t_1, t_2]\} \\
\mathcal{T}_{Fission_S} &= \{\theta[t_0, t_1, t_2] \ \to \ \theta[1, t_0, t_1, t_2]\} \\
\mathcal{S}'_R &= \mathcal{S}_R \circ \mathcal{T}_{Fission_R} \\
&= \{R[i] \to \theta[0, i, 0, 0]\} \\
\mathcal{S}'_S &= \mathcal{S}_S \circ \mathcal{T}_{Fission_S} \\
&= \{S[i, j] \to \theta[1, i, 1, j]\}
\end{aligned}
$$

Note that we had to extend the dimensionality of $\mathcal{S}_R$, to allow a coherent definition of the transformation and a matching dimensionality of the resulting schedules. However, this does not change the schedules' information.

Using the modified schedules $\mathcal{S}'$ will allow the parallel execution of both statements, since they will not interleave anymore:

```
1  for (i = 0; i <= K; i++)        // Loop #1
2      R(i);
3
4  for (i = 0; i <= K; i++) {      // Loop #2
5    for (j = 0; j <= L; j++)
6      S(i, j);
7  }
```

Figure 2.9: Loop Fission Example after Transformation

Loop Strip Mining

Strip mining apportions the regarded loop into *strips* of a fixed stride, introducing an additional loop level in the process. This is especially useful in preparation of so-called *trivially SIMDizable loops* (see e.g. section 7.3.3 of [5]).

Now we concentrate on the loop nest of Figure 2.9:

```
1  for (i = 0; i <= K; i++) {
2    for (j = 0; j <= L; j++)
3      S(i, j);
4  }
```

While it might not provide advantages at first, organizing a loop in chunks of a maximum size is the basis of several optimizations. In this example $\mathcal{T}_{Stripmine_S}$ transforms the outer loop into two nested loops, with a maximum chunk size of four.

$$\begin{aligned}
\mathcal{D}_S &= \{i_S, j_S \mid 0 \leq i_S \leq K \ \wedge \ 0 \leq j_S \leq L\} \\
\mathcal{S}_S &= \{S[i,j] \to \theta[i,j]\} \\
\mathcal{T}_{Stripmine_S} &= \{\theta[t_0,t_1] \ \to \ \theta[t,t_0,t_1] : t \bmod 4 = 0 \wedge t \leq t_0 < t_0 + 4\} \\
\mathcal{S}'_S &= \mathcal{S}_S \circ \mathcal{T}_{Stripmine_S} \\
&= \{S[i,j] \to \theta[t,i,j] : t \bmod 4 = 0 \wedge t \leq t_0 < t_0 + 4\}
\end{aligned}$$

The corresponding code, after applying $\mathcal{S}'_S$:

```
1  for (ii = 0; ii <= K; i+=4)                  // Added loop level
2    for (i = ii; i <= min(ii+4, K); i++) {
3      for (j = 0; j <= L; j++)
4        S(i, j);
5    }
```

Figure 2.10: Loop Strip Mining Example after Transformation

Loop Interchange

Basically, this transformation is swapping the loop header positions of the considered loops. This transformation might not seem very helpful at first glance, but it can prove to be beneficial or necessary to permute loop headers, e.g. as part of an optimization.

Therefore we will define a schedule transformation $\mathcal{T}_{Interchange_S}$, which exchanges the time coordinates of the corresponding loops and then apply it to the code in Figure 2.10.

$$\begin{aligned}
\mathcal{D}_S &= \{i_S, j_S \mid 0 \leq i_S \leq K \ \wedge \ 0 \leq j_S \leq L\} \\
\mathcal{S}_S &= \{S[i,j] \to \theta[t,i,j] : t \bmod 4 = 0 \wedge t \leq t_0 < t_0 + 4\} \\
\mathcal{T}_{Interchange_S} &= \{\theta[t_0,t_1,t_2] \ \to \ \theta[t_0,t_2,t_1]\} \\
\mathcal{S}'_S &= \mathcal{S}_S \circ \mathcal{T}_{Interchange_S} \\
&= \{S[i,j] \to \theta[t,j,i] : t \bmod 4 = 0 \wedge t \leq t_0 < t_0 + 4\}
\end{aligned}$$

Using the modified schedule $\mathcal{S}'_S$ will yield code with an altered loop order:

```
1  for (ii = 0; ii <= K; i+=4)
2    for (j = 0; j <= L; j++) {                // Has been "moved up"
3      for (i = ii; i <= min(ii+4, K); i++)    // Has been "moved down"
4        S(i, j);
5  }
```

Figure 2.11: Loop Interchange Example after Transformation

With these examples we have seen how the `matvect` kernel could be parallelized and how transformations can be combined, by simple sequential application. Together, the last two transformations form the so-called *Unroll-And-Jam*, which we can now express as $\mathcal{T}_{UnrollAndJam} = \mathcal{T}_{Stripmine} \circ \mathcal{T}_{Interchange}$. This rather simple sequence of schedule modifications can expose possibilities for SIMD instructions, if the statements may run in parallel, and demonstrates the expressiveness of the polyhedral model.

After covering these theoretical fundamentals, we should note that it has taken several decades to finally apply polyhedral transformations in practical scenarios. The following section will present such an implementation and provide an overview on how respective program transformations can be realized.

2.2 Polly

Polly[4] [6][5] is a polyhedral optimization suite, embedded in the LLVM compiler infrastructure. As such it is based upon the LLVM-IR and has access to LLVM's wide range of analysis and transformation passes, which are continuously maintained and extended. Its purpose is the improvement of data-locality and enabling parallelism in provided programs. The workflow of Polly (Figure 2.12) can be divided into three abstract phases (or sections): *Detection* (frontend), *Transformation* (middle) and *Reconversion* (backend).

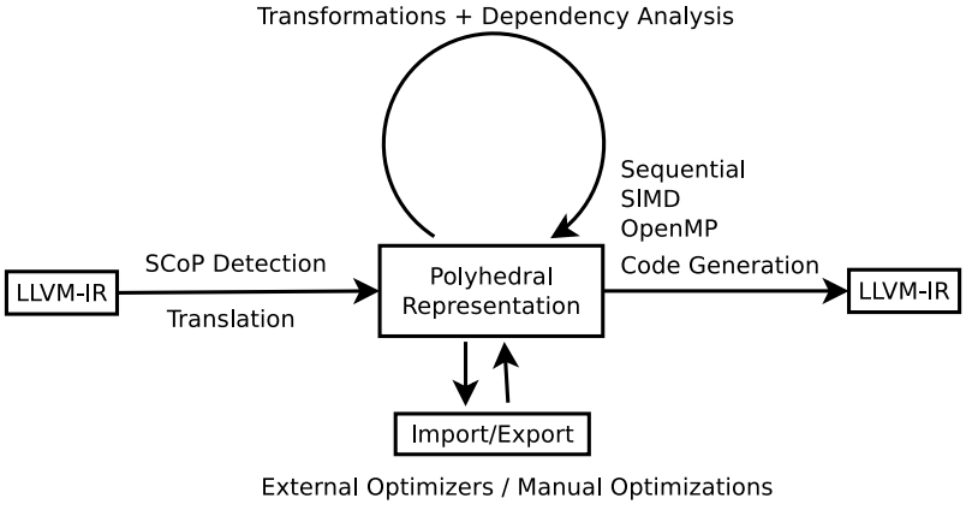

Figure 2.12: Polly's Architecture [5]

The frontend performs a set of transformations provided by LLVM to convert the given code into a specific form and expose further information, like e.g. induction variables and their values. Among other things[5], these transformations will simplify certain loop structures and ensure that loop induction variables feature an incremental stride of one with an initial value of zero. This *canonicalization pass* ensures that the following SCoP detection is able to recognize as much SCoPs as possible and allows for a rather simple implementation. Afterwards the detected SCoPs are translated into a polyhedral representation.

[4] `http://polly.llvm.org/`
[5] See section 2.4 of [5] for further information

Polly's middle part will then perform dependency analyses and apply polyhedral optimizations by itself (e.g. tiling, prevectorization) or even utilize external optimizers, like the Polyhedral Compiler Collection (PoCC)[6]. The latter can be achieved since it is possible to export and (re)import program data, which may also be used to perform manual modifications.

As soon as all transformations are done, the backend will convert the provided polyhedral representation into a generic Abstract Syntax Tree (AST) and regenerate the corresponding LLVM-IR instructions. This reconversion into LLVM-IR is of particular interest, since it decides which loops may be parallelized using OpenMP or SIMD instructions.

We will now take a closer look at single aspects, to get a more precise understanding of how Polly achieves its goals.

2.2.1 LLVM

LLVM[7] [9] is a compiler framework and provides a broad range of different compiler toolchains. Its major design concern was to provide optimization at different stages of a program's lifetime in a transparent, modular manner. A main feature is the LLVM-IR, which is a platform independent low-level program representation in Static Single Assignment (SSA) form, commonly used by the provided transformation passes and subprojects (like e.g. clang or Polly). Since it can be exported at any compilation stage and in human-readable form, it is easy to transfer the representation to external tools or even perform manual modifications, if desired.

Apart from that, LLVM provides several front- and backends. This allows projects, implemented in multiple programming languages, to be deployed on different platforms. Hence, Polly is also basically platform independent and benefits from the already implemented and maintained toolchain.

2.2.2 Practical Detection and Representation

Besides the theoretical definition of a SCoP in Section 2.1.1, we need a more tangible approach to detect SCoPs in LLVM-IR. In section 5.2 of [5] such a definition is described as a Control Flow Graph (CFG) subgraph with a particular form. These *simple regions*[8] can be identified by LLVM passes and interpreted as a function call, since they may only feature exactly one edge that enters or exits this region, respectively. Moreover, execution of this region can only affect its own control flow. Thus, a simple region can be replaced by an optimized version without incisive changes to the CFG.

[6] `https://sourceforge.net/projects/pocc/`

[7] `http://llvm.org/`

[8] Further region definitions can be found in section 2.3.3 of [5].

In addition to this structural constraint, Polly checks if the control-flow is potentially malformed or contains instructions which cannot be represented. Therefore only (un-)conditional branches are allowed and e.g. function returns or switch statements are denied.

Furthermore, it is necessary to statically know the number of loop iterations. LLVM provides loop analyses like scalar evolution (`scev`) which exposes this information via expressions that describe value patterns of loop induction variables. Among other things Polly needs to check whether this information is available *and* can be expressed as an affine expression[9]. This expression may only consist of integer parameters and constants which are not changed inside the SCoP.

If the control flow complies to the constraints, it is important to make sure that only side effect free functions and operators are used or that all possible side effects are known and can be represented. While LLVM is able to provide information on possible side effects of instructions and functions, exception handling is never allowed because it cannot be described using the polyhedral model.

Now that SCoPs expressed in LLVM-IR can be identified, we will take a glimpse into the data structures and tools used to represent them (for further information cf. section 5.3 of [5]). Those structures are primarily based on the integer set library (isl) [16], which provides the means to store and manipulate integer sets, as we have already encountered them in the form of e.g. iteration domains (Section 2.1.1).

In general a SCoP is then divided into its *context*, which represents constraints and relations of involved parameters as an integer set and a list of the included *statements*. Each statement on the other hand forms a quadruple, consisting of a name and the three known mathematical definitions: *iteration domain*, *schedule* and *access function*. Only the access differs from its theoretical counterpart in which it is further annotated with the *kind* of access, i.e. *read*, *write* or *may write*.

With the isl at hand, it is also possible to perform dataflow analyses and apply polyhedral transformations as seen in Section 2.1.2. But since the code transformations are solely applied to the derived polyhedral representations, we still need to convert them back into LLVM-IR. Hence, we now need to regenerate imperative code, which is the purpose of Polly's backend.

[9] Affine expressions are a **subset** of scalar evolution expressions

2.2.3 Code Reconversion

Since LLVM cannot operate on the isl data structures, but only on its IR, we need to (iteratively) regenerate a corresponding LLVM-IR. The first step is to replace each SCoP by an enumeration of all statement instances, described by the transformed schedule. To represent these loop structures *generic ASTs* are used. At the time of writing this reconversion step is also performed by isl (replacing CLooG), as it enables the conversion of Polly's internal data structures into such an AST by now. Even at this stage, the resulting code can still be very different. The reason for this is: while the execution order is defined, it is left to the code generation how to realize this order. An example SCoP, translated into two different ASTs (taken from [5]) illustrates this circumstance in Figure 2.13 and Figure 2.14, respectively. While both express a semantically equivalent program, the first one is optimized for minimal code size and the second one for a minimum number of branches.

```
1 for (i = 0; i <= N; i++) {
2   if (i <= 10)
3     B[0] += 1;
4
5
6   A[i] = i;
7 }
```

Figure 2.13: AST Example – min. Code Size

```
1 for (i = 0; i <= min(10, N); i++) {
2   B[0] += 1;
3   A[i] = i;
4 }
5 for (i = 11; i <= N; i++) {
6   A[i] = i;
7 }
```

Figure 2.14: AST Example – min. Branching

Additionally, now that structures are coarsely defined (e.g. by control flow that cannot be realized otherwise), Polly calculates the statement dependencies, similar to the dependency domains. The gathered information will be used later on to decide whether parallel execution, SIMD instructions, etc. may be used, or not.

By default Polly then performs a sequential LLVM-IR code generation. Therefore all abstract constructs introduced by the generic AST will be replaced by their LLVM-IR analog in a straightforward manner. The other two options are OpenMP and vectorized code generation, which do not exclude each other.

Vectorized code emerges from *trivially* vectorizable loops, e.g. as seen in Section 2.1.2 (*Loop Strip Mining / Unroll-And-Jam*). Such loops have to be created by choosing the right sequence of polyhedral transformations in the middle part of Polly. Afterwards loop statements will be replaced by LLVM's *internal* vector instructions. On final code generation by the machine backends of LLVM (e.g. ARM or x86-64) these internal instructions will eventually be translated to platform dependent SIMD instructions.

Before we take a closer look at OpenMP code generation in Chapter 3, we will focus on OpenMP itself in the following section.

2.3 OpenMP

Back in the 1980s when the first Symmetric Multi-Processors (SMPs) were produced in larger quantities, the first platform dependent multi-processing instructions were implemented to allow parallel computing. While this was a huge step, programs exploiting these instructions were oftentimes not portable among different processors. Additionally, the already existent Message Passing Interface (MPI) standard did not[10] support so-called *shared memory*, which allows multiple programs to access the same memory locations simultaneously. This in turn may reduce redundancy and enable inter-program communication at the same time.

With these shortcomings in mind the Open Multi-Processing (OpenMP) standard was born in the late 1990s [4] and is still being improved and expanded. In the meantime OpenMP has become very popular and supports the programming languages Fortran, C and C++ as well as a wider range of processing units like e.g. Digital Signal Processors (DSPs).

In principle OpenMP[11] is a specification for compiler directives, so-called *pragmas*, library routines and environment variables, which are used to control OpenMP e.g. at compile-time. Altogether, this enables programmers to write portable, parallelized code at a reasonable level of abstraction.

Our well-known `matvect` kernel for example, can be parallelized simply by adding such a *pragma* (Figure 2.15) right before the statement, which shall be run in parallel. Naturally, such a change will only lead to a semantically equivalent program if the code carries no dependencies between the iterations. Ignoring these dependencies may lead to errors during runtime. Assumed that we provide a pragma which realizes our *intended* semantics and parallel execution, OpenMP will redirect the loop iterations to the requested worker threads. Therefore, the loop body is wrapped into a so-called *outlined* function[12] and each worker is instructed to perform its share of work by executing this function the appropriate number of times.

It would exceed the scope of this work to cover every possible detail. Hence, we will concentrate on what this particular code will do and have a look at different scheduling strategies since it will be of interest in Chapter 5.

[10] MPI-3 (2015) provides this feature

[11] `https://www.openmp.org/`

[12] Which is basically equivalent to the loop itself

```
1  #include <omp.h>
2  /// Dynamically parallelize with chunk size one on four threads ...
3  #pragma omp parallel for shared(a, s, x) private(i, j) \
4                           schedule ( dynamic, 1 ) num_threads(4)
5  for (i = 0; i <= n; i++) {
6    s[i] = 0;
7    for (j = 0; j <= n; j++)
8      s[i] = s[i] + a[i][j] * x[j];
9  }
```

Figure 2.15: `matvect` Kernel (OpenMP)

Apart from the *pragma*, we need to include the corresponding header file and begin with the keyword *omp*. While it might seem obvious that this code is meant to be executed in *parallel*, this keyword is mandatory and if omitted, the following code will end up as a sequential loop. Since we want to perform our computations using the given variables, it is crucial to allow *shared* access to them, especially for `s`, which will hold our resulting vector. But not everything needs to or even should be shared among the worker threads – each thread ought to maintain a *private* set of control variables. And lastly, before we focus on schedules, *num_threads* will request four workers (e.g. CPU-cores) among which the work will be shared.

schedule will affect *how* the work is distributed, this in turn can have a quite large impact on the performance of a program. There are many different strategies, which can basically be divided into three *kinds*:

`static`: The amount of work is evenly distributed among the threads.
Assuming four workers, a loop of 100 iterations will be split into four *chunks* of 25 iterations each. This strategy is especially profitable if iterations take up (nearly) equal amounts of time, such that all threads complete at about the same moment. Additionally, since the whole task is distributed right from the start, there are no further requests, which may take up time.

`dynamic`: At the beginning and on each request, each thread will receive a fixed amount of work, known as *chunk size*. After completion a thread will request the next chunk until there is no more work. Assuming for example 100 iterations, a chunk size of five and four workers. At first glance each worker will request five chunks, but not every computation may be equally costly and threads might not start at the same time. Therefore this strategy provides benefits when calculations turn out to vary in processing time.

`guided`: While the previous strategies have their downsides, guided tries to strike a balance between static and dynamic scheduling. Initially workers receive rather big chunks. But their size will be reduced, down to a minimum, which may be provided by the programmer. This approach potentially reduces the number of requests and counters load imbalances at the same time.

OpenMP might be a standardized specification, but there are multiple ways to realize this standard. In this thesis we will encounter the LLVM and the GNU implementations, which provide different Application Programming Interfaces (APIs). These allow direct access to OpenMP functionalities without the need for pragmas and are therefore especially suited for automatic parallelization and code generation of Polly's backend.

Later on, we will evaluate the performance impact of the above-mentioned settings and different OpenMP implementations in Section 5.2.1 and Section 5.2.2 respectively.

3 Polly's GNU OpenMP-Backend

The current OpenMP-backend of Polly uses the GNU OpenMP library *libgomp*[1] and thereby provides the means to automatically generate parallelized code. Performance-wise, the resulting code is equivalent to code that has been manually annotated (as discussed in section 4.2 and 4.5 of [14]). This is because the information (pragma) provided by a programmer will be mapped to the corresponding library calls by the compiler. Hence, there is no particular downside in using the automatic generation except for the fact that everything that can be run in parallel, will be parallelized – which *might*[2] not always be beneficial.

Information about the loops is gathered by LLVM and in particular isl after the generic AST has been generated. Therefore the results of LLVM's `scev` analysis (see Section 2.2.2) are used by isl to construct the iteration domain(s) and derive the dependency domain. The latter is tested for *emptiness* (cf. Section 2.1.1), indicating whether the considered SCoP may be safely run in parallel. If this requirement is violated, the backend is not used at all and sequential code is generated for this particular SCoP – otherwise Polly will use isl to further investigate the corresponding AST. In the course of this inspection, the loop is segmented into parts like its variable initialization and body, which is also reflected in Figure 3.1. Afterwards, these segments are used to finally retrieve the variables' upper and lower bounds, as well as their increment. This is done by inspecting each dimension of the SCoP's polyhedron because either dimension maps to a particular induction variable as seen in Section 2.1.1.

But it would presumably generate a large amount of overhead to introduce OpenMP calls for each and every one of those loops, which is why Polly will only introduce them for the outermost loop(s) of a SCoP, by calling the corresponding *Parallel Loop Generator* [14]. We will now have a look at the current implementation of this loop generator.

[1] Documentation: `http://gcc.gnu.org/onlinedocs/libgomp/`

[2] See our evaluations in the three sub-sections of Section 5.2

When Polly detects a loop as parallel and OpenMP parallelization is enabled, the LLVM-IR generation of the considered SCoP is left to this backend. All instructions contained by the SCoP will then be put into a *subfunction* (or: *outlined* function, cf. Section 2.3) and as with the use of OpenMP pragmas, this subfunction has to be executed by the worker threads. Though, it should be noted that while it might seem that each loop iteration in the subfunction matches one of the original loop, this is generally not the case in the final code[3] because of optimizations (like *vectorization*, *unrolling* or *tiling*). Realizing all these transformations entails a series of steps to be taken regarding the OpenMP library. While a detailed description can be found in section 4.5 of [14], we will provide a high-level abstract of the necessary actions:

- Since the original program code might not utilize OpenMP at all, the required functions need to be declared and linked against the library.
- While manually annotated programs provide a certain amount of information, the backend has to determine some parameter values or purport them by itself. This includes, for example, the loop boundaries and shared variables. However, the majority is available beforehand, by the above-mentioned analyses of isl, while the used variables are collected during the actual LLVM-IR code generation.
- Because the subfunction has to be controlled by the library functions (from the outside *and* within), the control flow has to be adapted accordingly.
- Lastly, the actual library calls have to be placed in the prepared locations using the respective parameters.

In Figure 3.1 we present the general structure of a transformed SCoP, resulting from the aforementioned steps. After a team of threads has been ordered from the GNU OpenMP runtime by calling `GOMP_parallel_loop_runtime_start`, each thread will execute the same subfunction. The subfunction itself is built around the original (sequential) SCoP statements which are located in the CFG subgraph *Loop body*. After initializing several variables (e.g. pointers, loop boundaries, structures), `GOMP_loop_runtime_next` will request the upper and lower bound of the next chunk. If there is another chunk, the execution continues – otherwise, since there are no blocking dependencies[4] between the iterations, it is safe to release a thread once all work is distributed even if not all threads are finished.

[3] For example: During tests and evaluations of the resulting code, oftentimes (multiples of) 32 *original* iterations were condensed into a single subfunction call – see also: Section 5.2.1

[4] If there were such dependencies, the considered SCoP would not have been forwarded to the parallel loop generator in the first place.

This allows the runtime to reassign the now freed thread to another task, making more efficient use of the available resources. When all threads are finished and have called `GOMP_loop_end_nowait`, the OpenMP section is finally terminated by `GOMP_parallel_end`.

Irrespective of the actual program, the GNU backend will only use these four API functions[5], in exactly this manner. Therefore, this backend is limited to dynamic scheduling strategies only, with a preset chunksize provided by the OpenMP runtime. But utilizing the compiler switch `polly-num-threads` allows the user to set the number of threads to use. If omitted, a system-specific default value will be used (see e.g. the environment variable `OMP_NUM_THREADS`), which is *usually* the number of cores.

As already mentioned in our introduction (Chapter 1), it might be problematic to be bound to the *libgomp* library and even prevent the use of this backend. Apart from this issue there are no further customization options regarding the optimization, so users are limited to the preset scheduling and chunksize. While this might not impose difficulties of any kind, it surely leaves opportunities untapped as we will see in Chapter 5. In the following chapter we will present our alternative OpenMP-backend, which is largely based on this implementation.

[5]For implementation details see e.g. `https://www.cs.rice.edu/~la5/doc/ligbomp-doc/`

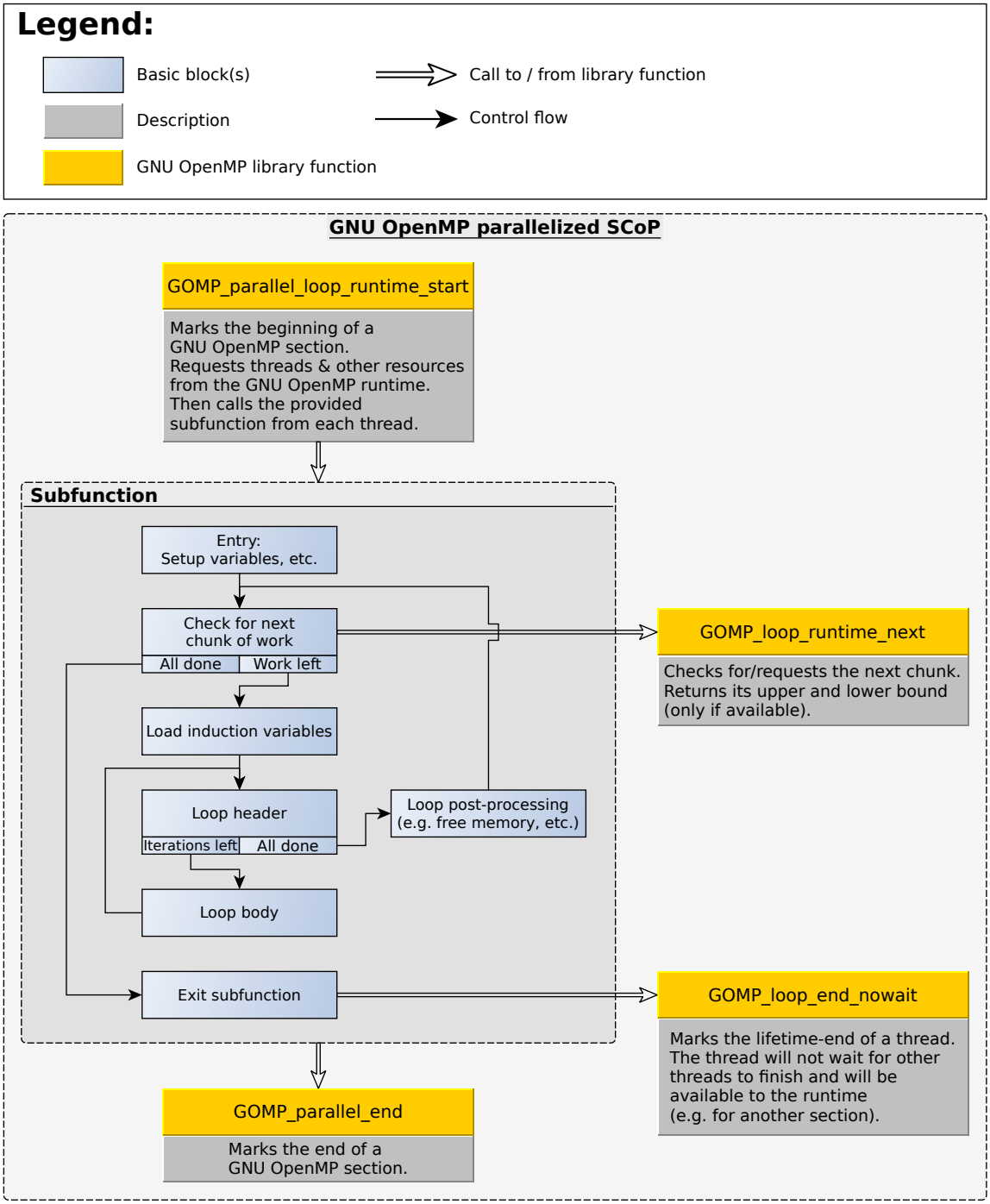

Figure 3.1: GNU OpenMP-Backend – Call and CFG Illustration

4 An alternative LLVM OpenMP-Backend

As discussed in the beginning of this thesis, the ESA work group at TU Darmstadt maintains a compiler system which is based on the LLVM-IR [7]. In addition, only LLVM's OpenMP implementation is supported, which prevents the Nymble-OMP extension [15] to benefit from Polly's automatic parallelization feature.

Therefore we decided to extend the implementation of Polly's OpenMP-backend not only for the sake of an additionally supported library but also with further customization options in mind. These will allow users to adjust the OpenMP execution to match their specific problem characteristics more precisely. *Release* version 4.0 of Polly forms the basis of this alternative backend.

While the numerous API differences would not allow for a completely straightforward port of the existing backend from *libgomp* to the LLVM OpenMP library (*libomp*), there were many possibilities for reusing existing infrastructure. Hence, we decided to implement our backend as a class, which is derived from the original *Loop Generator* (GNU backend). This allows us to use specific functions directly, like the building or preparation of loops and the handover of shared variables between subfunctions. Therefore the resulting structures, positioning of calls, etc. bear a strong resemblance to the LLVM-IR that is generated by the GNU backend.

However, the LLVM API demands a slightly more fine-grained control, which results in an increased number of library calls that are used to prepare the runtime. An example would be that `GOMP_parallel_loop_runtime_start` takes *the number of threads to request* and *a pointer to the subfunction* as two of its parameters. The subfunction is then automatically called by the requested threads – in LLVM OpenMP this would generally require three distinct library calls. At first, the global id of the parent thread has to be determined using `__kmpc_global_thread_num`, then the required number of threads has to be requested for the parent thread with `__kmpc_push_num_threads`. And finally the subfunction / *outlined* is called (utilizing the requested threads) by `__kmpc_fork_call`. We will now present a very simplified example (Figure 4.1) to clarify the general principle.

```
; Declarations of all used external functions
; e.g. "@__kmpc_fork_call" -- omitted!

define i32 @main() local_unnamed_addr {
  GLOBAL_ID = call i32 @__kmpc_global_thread_num( ... )
  call void @__kmpc_push_num_threads(i32 GLOBAL_ID, i32 NUM_THREADS)
  call void @__kmpc_fork_call(@outlined)
  ret i32 0
}

define internal void @outlined( ... ) {
  setup: ; Entry block: Variable Initializations
    ; This init-call will e.g. set lower and upper bound (LB, UB)
    call void @__kmpc_dispatch_init_8(i32 DYN_SCHEDULING_TYPE)
    %workToDo = call i32 @__kmpc_dispatch_next_8(LB, UB )
    br i1 %workToDo, label "loopBody", label "exit"

  checkNext:
    %moreWorkToDo = call i32 @__kmpc_dispatch_next_8(LB, UB)
    br i1 %moreWorkToDo, label "loopBody", label "exit"

  ; The induction variable setup (basic blocks) is cut-out
  ; and would take place before the "loopBody"
  loopBody:
    ; This would be a basic block structure that performs the
    ; calculations until the induction variable reaches the
    ; specified (adjusted) UB -- this adjustment is necessary
    ; e.g. because of different comparisons like "<" and "<="
    br label "checkNext"

  exit:
    ret void ; This will return to @main
}
```

Figure 4.1: LLVM-IR Example (OpenMP)

The signatures of functions and the content of basic blocks or even the program in Figure 4.1 were reduced to the bare minimum. If we were using pragmas, Figure 4.2 could be used to produce a similar output. So, we are going to see an OpenMP parallelized code that is dynamically scheduled with a chunk size of one and is executed using four threads. Each of these threads will execute `@outlined`, beginning with the thread-specific setup using `__kmpc_dispatch_init` and fetching the first chunk of work (which may be empty) with a call to `__kmpc_dispatch_next`. If there is no work, the returned value will be zero and therefore lead to the exit of the subfunction. Otherwise (return value is one) the thread has also received the upper and lower bounds of their respective work chunk and may proceed with the loading of induction variables (omitted in Figure 4.1). Afterwards, the actual calculation takes place and will be repeated until the upper bound is reached.

```
1 #pragma omp parallel for shared( commonly used variables ) \
2                          private( induction variables go here ) \
3                          schedule (dynamic, 1) num_threads(4)
```

Figure 4.2: Sample OpenMP Pragma

Possible program structures (depending on the scheduling type) are visualized by Figure 4.3 among other, smaller differences (compared to Figure 3.1) on a more abstract level. On the basic block level, there is only one minor change: if there is no work in the first place, for example when other threads already processed it, the control flow will directly branch to the exit block. This behavior was adopted from code that has been manually annotated and compiled by LLVM (like in Figure 4.1).

In principle, calls to the library functions are in the exact same locations. Functions with *dispatch* in their name[1] are tied to the execution of dynamically scheduled loops (this also includes e.g. *guided* scheduling). `init` will prepare the runtime and `next` will provide the bounds of the next chunk of work (along with its stride). While their GNU counterparts need to be terminated with a call to `GOMP_loop_end_nowait`, this is done implicitly[2] by the runtime.

On the other hand, there are the *static* functions, which will perform the corresponding actions for statically scheduled loops. The main difference is that in this scenario, threads need to call `__kmpc_static_fini` upon completion.

[1] A general note on naming: Many functions have a suffix of 4, 4u, 8, or 8u that indicates the width (in bytes) and type (signed or unsigned) of parameters

[2] There is an equivalence (`__kmpc_dispatch_fini`), but calling it will create an explicit *barrier*, resulting in the thread waiting for other threads to finish

Since Polly is only active when the `O3` switch is provided, and uses the GNU backend by default, we implemented several switches for our purpose. The most important one `polly-omp-flavor` allows to actually use our backend – if set to 1 our loop generator will be used, 0 (default) will use the GNU loop generator.

And as with the GNU backend, it is possible to provide the number of threads to be used by the OpenMP runtime via `polly-num-threads-llvm`. Consequently, a user can also choose which scheduling strategy shall be used and the backend will perform the correct placement of the according library functions. At the time of writing we accept the following schedules:

`static` schedules

- static_chunked (33)
- static (34)
- static_greedy (40)
- static_balanced (41)
- static_steal (44)
- static_balanced_chunked (45)

`dynamic` schedules

- dynamic_chunked (35)
- guided_chunked (36)
- trapezoidal (39)
- guided_iterative_chunked (42)
- guided_analytical_chunked (43)

The integer values originate from the corresponding enumeration of the library source code, and have to be provided using the switch `polly-llvm-scheduling`. Furthermore, the chunk size may be chosen via `polly-llvm-chunksize` to control the work distribution at an even lower level. This allows, for example to adjust the optimization to a specific program. Assuming that it has widely different processing requirements per iteration, *dynamic* scheduling should be chosen. But maybe the number of iterations is resulting in diminishing returns because of the equally high amount of (`next`) library calls – this would be an opportunity to increase the chunk size. However, sometimes *static* scheduling is suitable anyway, since it reduces the number of (explicit) library calls to a fixed amount (oftentimes this amount is around twice the number of threads → calls to `init` and `fini`).

With all these possible settings at hand, we will now evaluate the potential benefits and generally compare the libraries performance-wise in the following chapter.

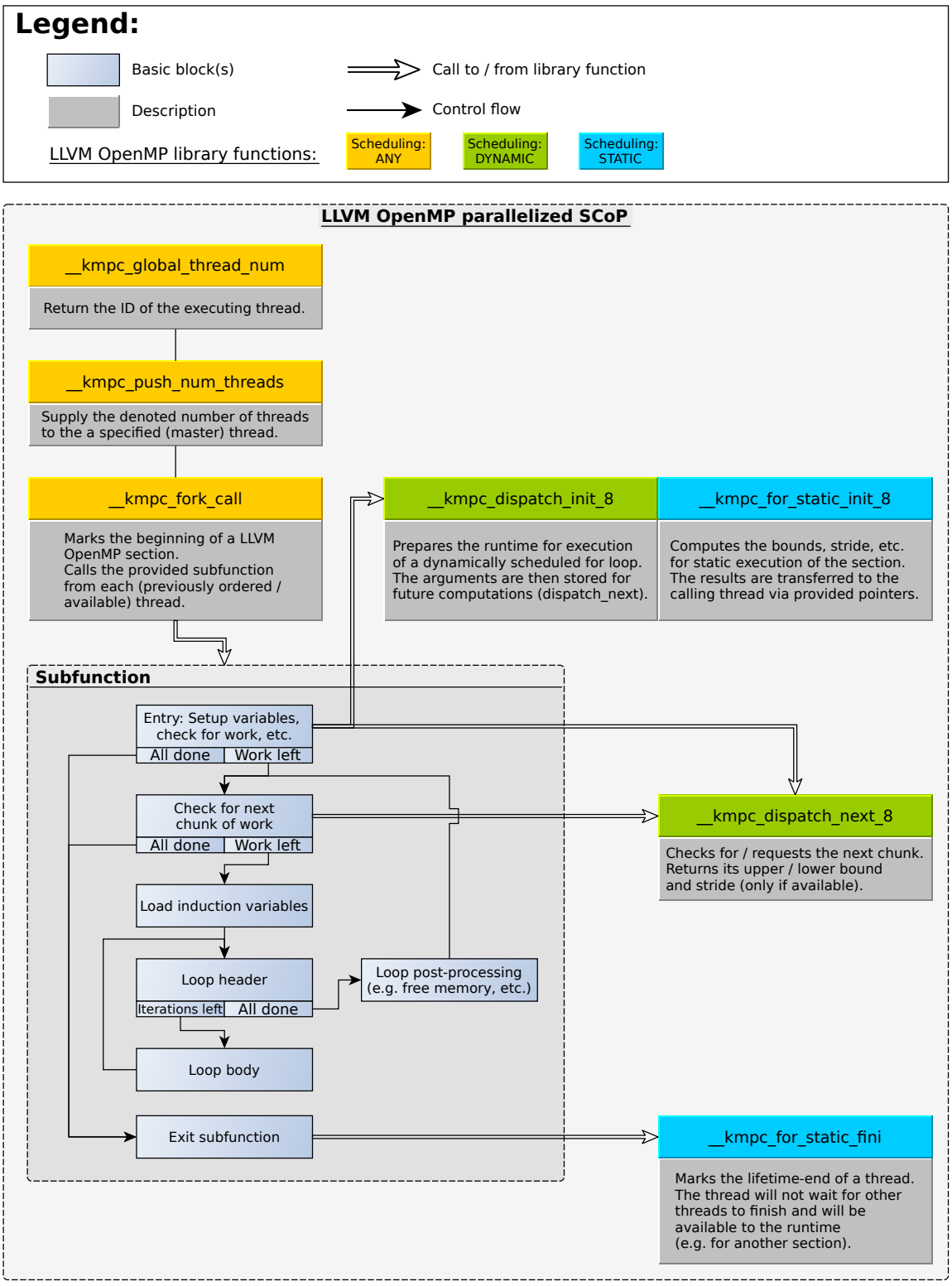

Figure 4.3: LLVM OpenMP-Backend – Call and CFG Illustration

5 Experiments

Since one of the central ideas regarding automatic parallelization is the efficient use of computing resources, we need to conduct several repeatable experiments. We decided to primarily compete against the GNU loop generator, because both backends are strongly tied together. This will also allow us to draw conclusions regarding both OpenMP implementations. Additionally, we include a set of comparisons to LLVM's C-language compiler project *clang-4.0*[1], which will be used throughout the process.

In the following section we will present the experimental methodology and then move on to the actual results.

5.1 Experimental Setup

After initial tests with matrix or vector multiplications (similar to the `matvect` example) we decided to utilize *PolyBench*[2]. This benchmark suite is designed for testing polyhedral techniques and features several key properties:

- Each measured computation is made up of SCoPs.
- All 30 sample computations are extracted from *real-world* algorithms, coming from various domains.
- Parameterized dataset sizes offer different testing scenarios.
- Dumping of results, which allows for regression testing.
- Accurate timing and setup before measured execution (e.g. cache flushing).

At first we performed mostly regression testing via the `POLYBENCH_DUMP_ARRAYS` compile option. While this will increase the overall runtime it does not affect the provided results of `POLYBENCH_TIME`. When dataset sizes are provided, like *small* and *large*, the programs were compiled using the respective option (e.g. `SMALL_DATASET`).

[1] `http://clang.llvm.org/`
[2] `https://sourceforge.net/projects/polybench/`

PolyBench may provide 30 different computation kernels, but not every SCoP is parallelizable by the backend. Therefore we profiled every program using *valgrind/callgrind*[3] and reduced the set of benchmark candidates accordingly, if no OpenMP library calls were issued, to 20 remaining programs.

However, there are still two noteworthy candidates in our selection of 20 benchmarks (see Table 5.1), namely *lu* and *trmm*. When vectorization (we use *strip mining*) was not actively **disabled**, neither the GNU nor our LLVM backend were able to compile *lu* correctly, due to errors during compile-time. If some results could not be collected because of this issue, we marked the respective x-axis label in light red. Though, using only vectorization and no OpenMP parallelization did not cause any problems. *trmm* on the other hand was oftentimes *off-scale* regarding the other results. So, we decided to exclude this program from the graphical evaluation, to avoid widely different scalings between similar setups as far as possible.

All of the following experiments were conducted on a Linux platform (kernel version 4.16), equipped with an AMD Ryzen 5 1600X[4] and 32 GiB of RAM. The utilized source code revision of PolyBench was `r108` (from 2018-Feb-08). Each *large* result was obtained by running the programs 60 times, then omitting the five best/worst results and calculating the arithmetic average of the remaining 50 runtimes. Results for *small* datasets were calculated the same way, but feature twice the amount of runs (i.e. 100 considered results out of 120).

We will now be closing this section with the description of a *result*. Because all investigations aim at comparing different settings against each other, *relative speedups* are provided. These are calculated according to the following formula, where `baseline` and `competitor` represent the corresponding average runtimes:

$$\texttt{speedup(baseline, competitor)} = \frac{\texttt{baseline}}{\texttt{competitor}}$$

Naturally, specific settings will be described alongside the presented results, for example if a vectorized variant has been competing against a non-vectorized one. Moreover, to allow for a slightly easier reading of all plots, we added a red horizontal line at $speedup = 1$. So, an improvement or a decrease in performance (relative to the baseline) can be found above or beneath this line, respectively.

[3] `http://www.valgrind.org/`

[4] Hexa-Core CPU with twelve hardware threads. For more information see e.g. `http://www.cpu-world.com/CPUs/Zen/AMD-Ryzen51600X.html`

Table 5.1: PolyBench: Selected Benchmark Programs

Selected PolyBench Benchmarks			
Category		Name	Short Description
Linear-Algebra			
1.	Kernel	`2mm`	2 Matrix Multiplications ($\alpha \cdot A \times B \times C + \beta \cdot D$)
2.	Kernel	`3mm`	3 Matrix Multiplications ($(A \times B) \times (C \times D)$)
3.	Kernel	`atax`	Matrix Transpose and Vector Multiplication
4.	Kernel	`doitgen`	Multi-resolution Analysis Kernel (MADNESS)
5.	Kernel	`mvt`	Matrix Vector Product and Transpose
6.	BLAS	`gemm`	Matrix-multiply ($C = \alpha \cdot A \times B + beta \cdot C$)
7.	BLAS	`gemver`	Vector Multiplication and Matrix Addition
8.	BLAS	`gesummv`	Scalar, Vector and Matrix Multiplication
9.	BLAS	`syr2k`	Symmetric rank-2k Update
10.	BLAS	`syrk`	Symmetric rank-k Update
11.	BLAS	`trmm`	Triangular Matrix-Multiply
12.	Solver	`cholesky`	Cholesky Decomposition
13.	Solver	`lu`	LU decomposition
14.	Solver	`ludcmp`	LU decomposition and Forward Substitution
Stencil			
15.		`adi`	Alternating Direction Implicit solver
16.		`fdtd-2d`	Heat Equation over 3D Data Domain
17.		`head-3d`	Heat Equation over 3D Data Domain
Datamining			
18.		`correlation`	Correlation Computation
19.		`covariance`	Covariance Computation
Medley			
20.		`deriche`	Edge Detection Filter

5.2 Experimental Results

We will now have a step-by-step look at the different experimental results[5]. In the beginning the LLVM backend is only compared to itself – with different settings to isolate their respective impact on runtime performance. Afterwards the evaluation of both backends will be discussed, followed by our last comparison between the most important program transformations.

5.2.1 Comparisons within the LLVM Backend

Chunk sizes

At first we want to investigate the impact of different chunk sizes on performance, for large (Figure 5.1) and small (Figure 5.2) datasets. While chunk size has no impact on the investigated *static* scheduling variant, it can be used to fine-tune both *dynamic* kinds. When used in combination with *dynamic chunked*, every requested chunk will consist of as many iterations as the chunk size – until the number of leftover work is smaller where simply the remainder is returned. In conjunction with *guided chunked* it depicts the minimum chunk size, so except for the last chunk, no chunk may be smaller than this number. Variation of the chunk size might prove useful in settings where processed iterations have a (widely) different processing time. Here, *smaller* chunk sizes may reduce idle times that result from the unbalanced thread workloads.

Since it is the default value, we chose a chunk size of *1* as our baseline. The general picture of both variants is pretty similar: oftentimes it is not beneficial to increase the chunk size. However, we can see that the picture shifts irregularly if the dataset is smaller: seven of 20 programs may be accelerated by larger chunk sizes but overall the performance decreases even more. An exception is *trmm* which iteratively gains positive speedups, as the chunk size increases, of around 1.3× to 4.1× for the small dataset. (ca. 1.3× to 3.0× for the large dataset).

Possibly, the problem size will be too small and a large fraction of the runtime is being spent on managing the threads, leading to a rather continuous performance loss. In the course of other evaluations we encountered the phenomenon that a large amount of originally single iterations gets accumulated, e.g. by a factor of 32. In a very small setting where the total number of iterations lies within the magnitude of this factor it might be the case that only one or two iterations have to be processed. Because of this they are possibly assigned to a single thread, resulting in all other threads being idle. Nevertheless it costs some time to request and maintain those threads, leading to an (evenly) increased runtime.

[5] We may provide the actual data, on request

We may conclude that a chunk size of one is usually a *good* choice. Though there are cases, where a higher chunk size (considerably) improves the performance. That confirms our assumption that introducing additional ways to influence the optimization process might prove useful.

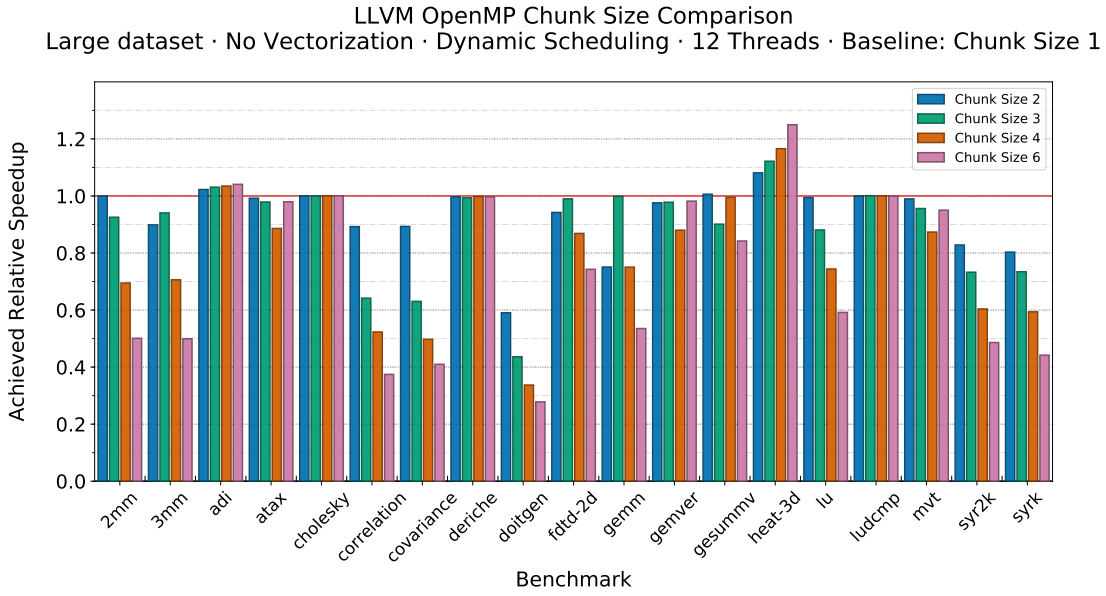

Figure 5.1: LLVM OpenMP-Backend – Chunk Sizes – Large Dataset

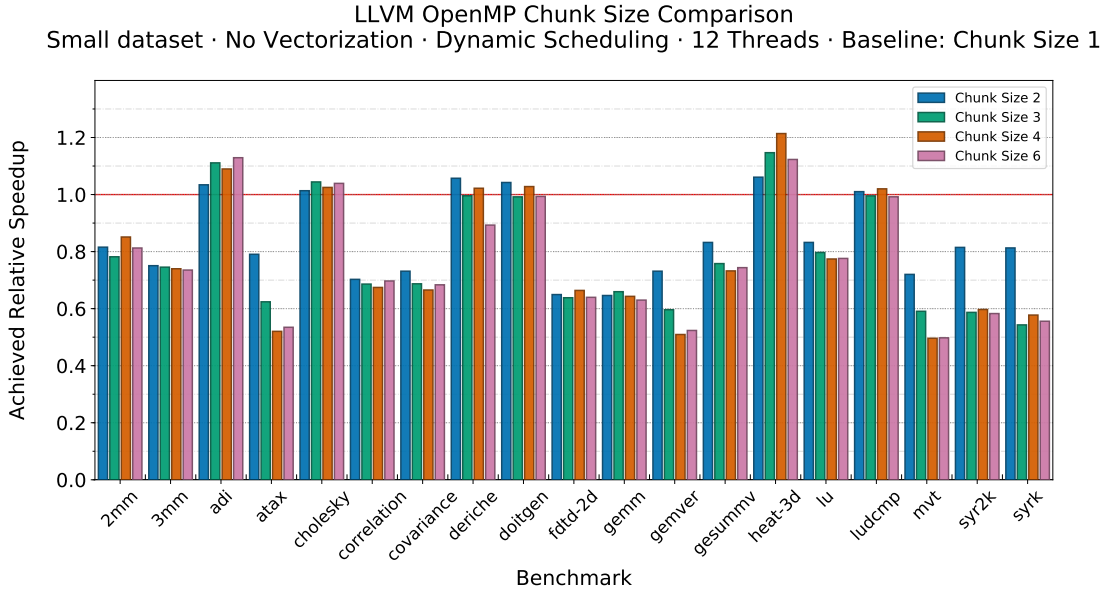

Figure 5.2: LLVM OpenMP-Backend – Chunk Sizes – Small Dataset

Vectorization

Since vectorization may have a huge impact on a program's performance, e.g. because of the reduced number of executed instructions, it is especially important to support this kind of transformation. We want to demonstrate the possible gains compared to non-vectorized code, which therefore represents the baseline of the following graph (Figure 5.3). Large and small results are normalized using their respective baseline. So, the next results may be rather unsurprising: Certainly, the majority of benchmarks benefits from vectorization.

Nevertheless, we can see that especially linear-algebra kernels (*2mm*, *3mm* and *doitgen*) gain quite large performance boosts. Though, overall it is not that clear, even when reducing the dataset size it may be the case that the relative speedup will decrease. A fifth of the regarded programs will not gain any performance at all, in specific scenarios.

It might seem that *adi* loses performance, but the *loss* is only around 2%. *trmm*'s performance is on-par with the non-vectorized version for the large dataset and also *loses* around five percent with a small dataset. Both discrepancies correspond to total execution time differences that are in the range of 10^{-5} seconds – so, they can be explained by measurement errors. Note: As mentioned earlier, we had to exclude *lu* from this particular evaluation, since it could not be compiled with vectorization.

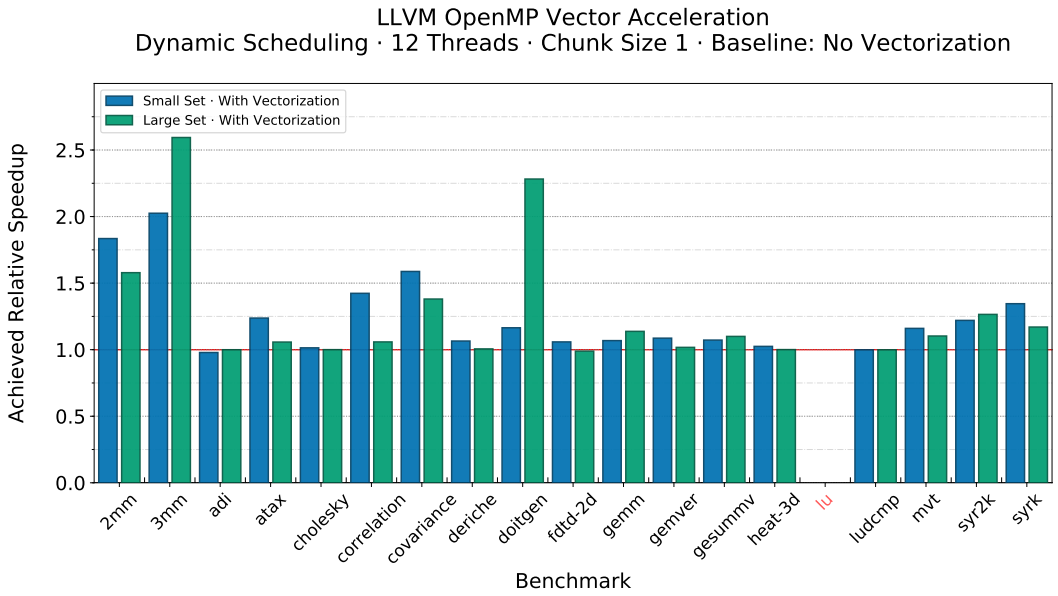

Figure 5.3: LLVM OpenMP-Backend – Vectorization

Scheduling Strategies

Our next parameter of interest is the scheduling strategy (Section 2.3). Since, depending on the problem, it might be beneficial to utilize a particular kind of scheduling to avoid idle/waiting threads or a large amount of library calls to request the next chunk. While we evaluated the three scheduling variants *static* and *guided chunked* against *dynamic chunked* with three different thread counts (four, eight and twelve), we will only present one graph with a thread count of twelve (Figure 5.4). We decided to do this because all plots were very similar and therefore offered no additional insights. As the GNU backend uses *dynamic* scheduling, the respective small/large results act as the baseline to compare against the LLVM implementation.

The first thing to note are the low differences between the baseline and *guided* results, which means that both strategies behave quite similar in these settings. This is due to the fact that *guided* will eventually change to pure *dynamic* scheduling once the amount of remaining work hits a certain threshold. Certainly, it may sometimes outperform *static* and *dynamic* scheduling, but only by small margins.

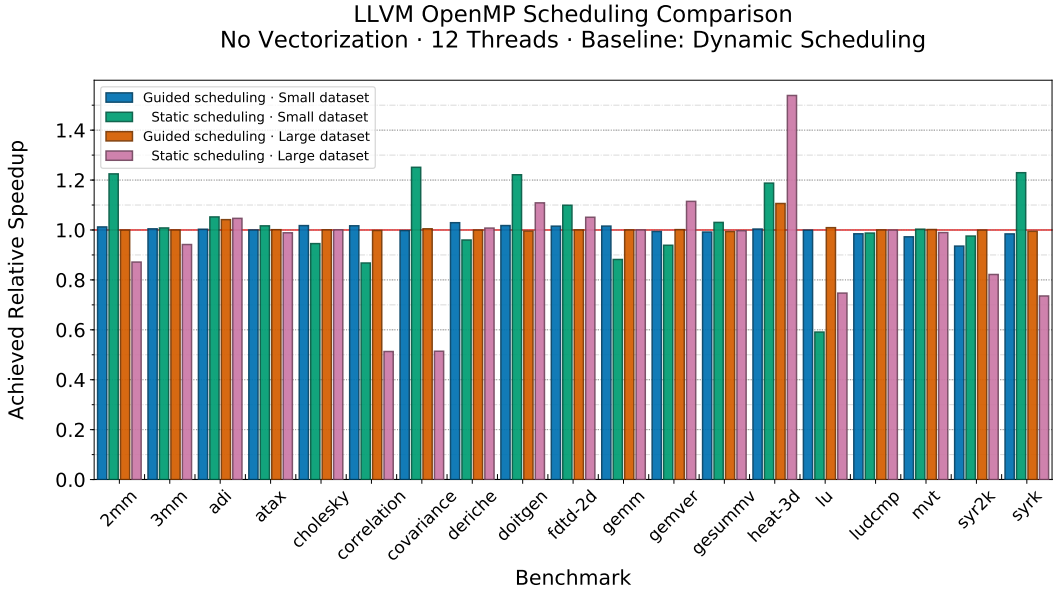

Figure 5.4: LLVM OpenMP-Backend – Scheduling Strategies

Another thing to note would be that static is quite problem-dependent, with rather high numbers on both ends: loss and gain. In *trmm*, static scheduling even reaches positive speedups of 3.9 and 9.3 (small / large) which is also a huge improvement over guided scheduling with 1.5 and 4.9 (small / large). This clearly demonstrates that implementing switches for different scheduling strategies was a favorable decision.

Generally, we can deduce that any scheduling strategy may have its advantages and can be used to fine-tune the performance of certain applications.

Thread Counts

Finally, we will investigate how the number of threads correlates to performance gains – this is of particular interest since the overall performance rating of modern CPUs tends to scale with their (increasing) number of available cores. Therefore we chose four threads as our baseline, to see if performance scales with the number of threads used. Only *dynamic* (Figure 5.5) and *static* (Figure 5.6) strategies are evaluated, since the *guided* scheduling achieved very similar results to the *dynamic* strategy (see also the former observations on scheduling).

According to Amdahl's law[6] we could gain speedups of 2.0 to 3.0 for eight and twelve threads on our normalized scale. But only if the corresponding program would be **fully** parallelizable. Naturally, there are sequential program parts and a finite amount of resources that is shared among the threads (e.g. memory access or CPU cache), so we do not reach comparable numbers. Overall we can see, that more threads will usually result in shorter runtimes, while the most significant speedups can be seen for the large datasets. A possible explanation would be that the setup-cost for a thread has to be compensated. But there are also results that contradict this picture: *adi* and *doitgen* will lose performance when executed with a higher number of threads.

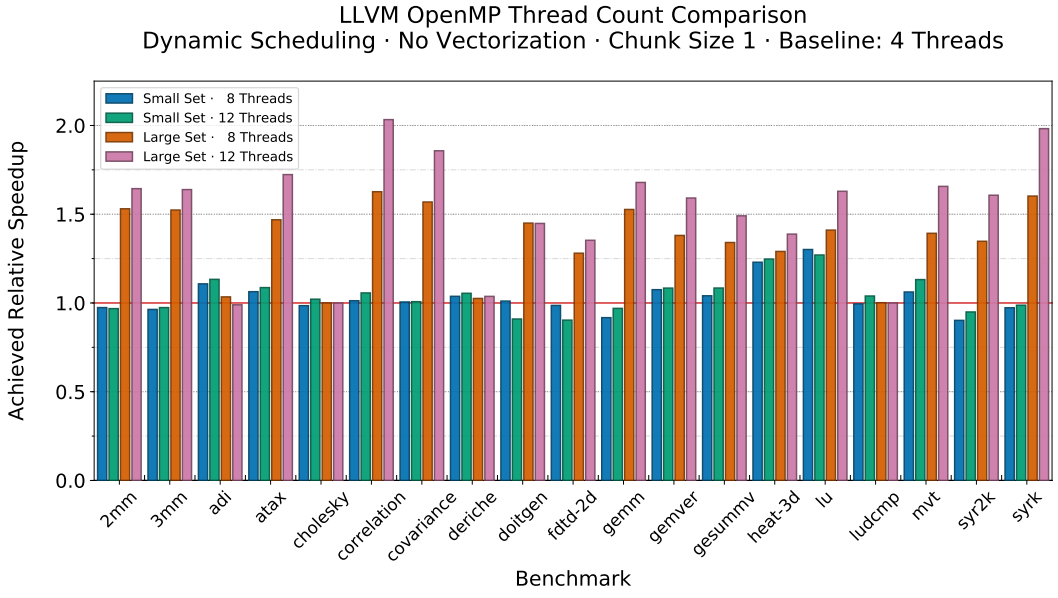

Figure 5.5: LLVM OpenMP-Backend – Thread Counts – Dynamic Scheduling

[6]Speedup $= \left((1-f_p)+\frac{f_p}{\text{threads}}\right)^{-1}$, where f_p is the fraction of the program that can be parallelized

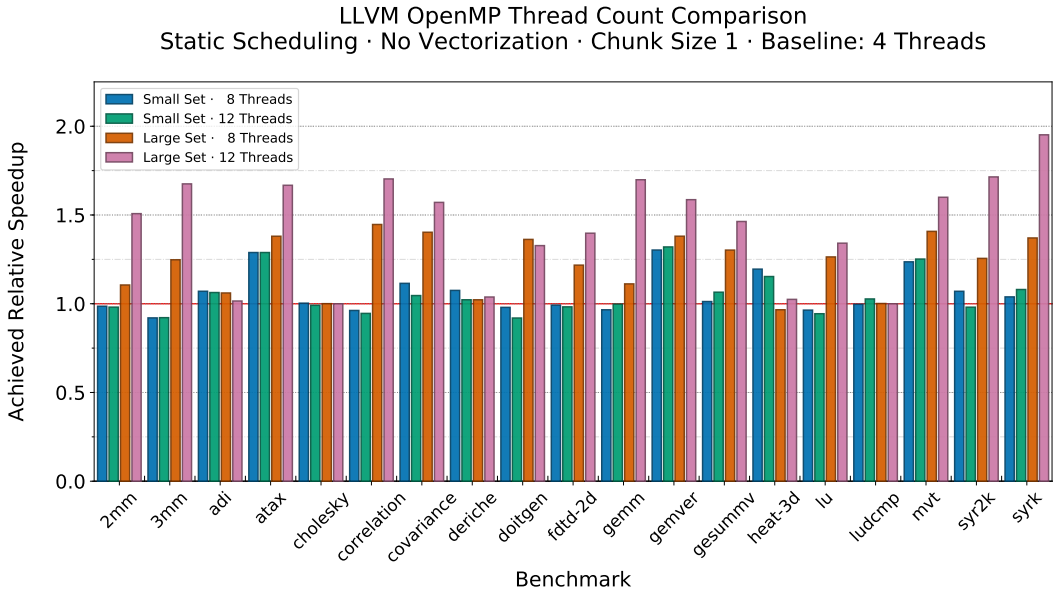

Figure 5.6: LLVM OpenMP-Backend – Thread Counts – Static Scheduling

After some investigation with valgrind and taking the average total execution times into account we found two possible cases:

- We already know the first one, which explains most slower cases for small dataset sizes: very small runtimes that get expanded by thread setups. Especially because of threads that will never work on a chunk.

- But that would not explain why *adi* or *doitgen* will lose performance in the large cases. We found that these programs (also e.g. *fdtd-2d*, *heat-3d* and *lu*) tend to call the optimized subfunction *more often*. For example: *doitgen* (small) calls its subfunction 500 times and even 21000 times per thread with the large dataset – while other programs tend to call theirs only a single time. This in turn creates a new OpenMP section every time, with the corresponding number of threads, which is basically an extreme version of the aforementioned case. Naturally, such an amount of library calls and thread setups will impact the performance regardless of the employed schedule strategy.

Again *trmm* tends to be rather unique: its runtime is doubled in the static case for twelve threads, while eight threads perform just as good as four. This is the *first* case: we have a very small execution time and a small number of iterations. While it might seem that *trmm* gradually gains performance in the dynamic case, we find that these total execution times have increased about tenfold, when compared to the static runtimes (large: 1.15s versus small: 0.12s). That confirms our previous observations about the significance of choosing the right scheduling strategy.

Interim Conclusion

Our results have shown that providing switches to users of the backend was a pretty advantageous decision, since it enables them to adapt the resulting code to meet the requirements of their particular program. In addition, these customizable settings clearly distinguish our backend from the current implementation. Therefore, we will evaluate the ability to compete against the GNU OpenMP implementation in the next section.

Beforehand, we will summarize our observations so far: Many settings we presented tend to provide their best result when adjusted for the specific problem. Nevertheless, we want to designate parameters which *might* be used as default values for a broad range of applications:

- Chunk size: 1 – This setting works pretty well for the vast majority of considered algorithms.
- Scheduling: dynamic – Achieves *good* results on average and constitutes the default of Polly's GNU OpenMP-backend.
- Threads: *12* (i.e. the systems' *respective maximum*) – Especially useful for large datasets and OpenMP runtimes would also use all available threads by default.
- Vectorization: on – Usually, this only offers advantages; still we will not use vectorization for the next section, as SIMD instructions might influence the results too much.

5.2.2 Comparisons between the GNU and LLVM Backends

In this section we will investigate the performance of our backend, relative to the current implementation. Since we produce the same structures as the GNU backend when dynamic scheduling is used, the following comparisons are also *possibly* transferable onto the two different OpenMP implementations.

We will investigate the relative runtime performance of the two backends, by varying the number of threads and dataset size. Additionally, we decided to include the results of Polly without vectorization as a reference – so we can assess if parallelization is profitable at all. Both runtime implementations are mature and widely adopted, therefore we do not expect big discrepancies. All six graphs use the corresponding GNU average runtime as its baseline.

GNU/LLVM – Large Datasets

First, we will take the large dataset size into consideration, where we can confirm that OpenMP parallelization is profitable in nearly every case, when focusing on Polly's results. The LLVM backend on the other hand is slower with every scheduling strategy in twelve out of 20 cases for four threads (Figure 5.7). *trmm*'s speedup values range from around 1.0 for dynamic scheduling to 8.7 and 10.6 for guided and static – exceeding the 4× speedup achieved by Polly.

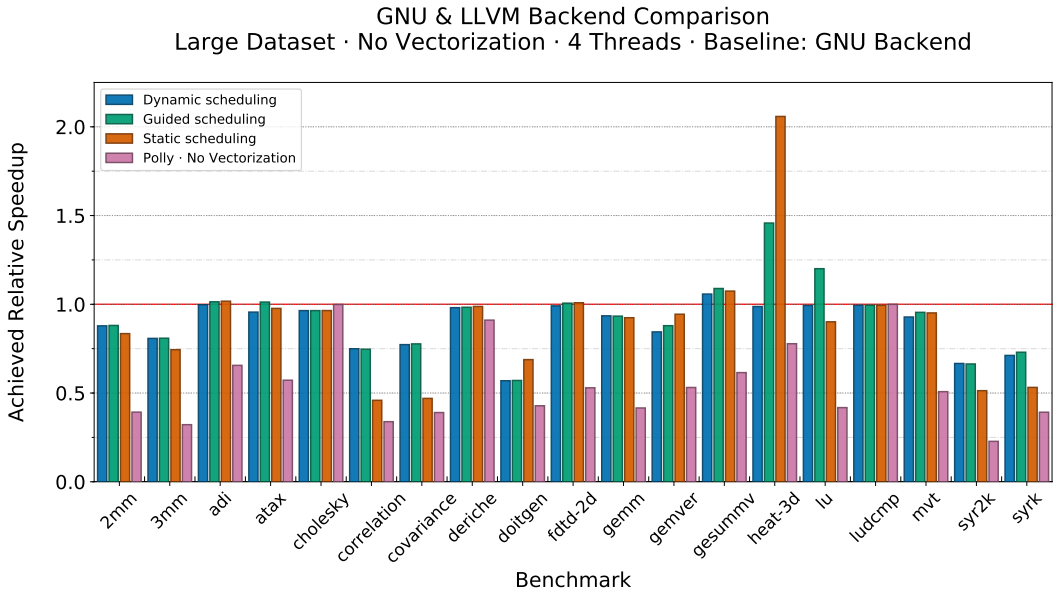

Figure 5.7: GNU and LLVM OpenMP-Backends – Large Dataset – 4 Threads

However, the programs that were compiled with our backend gain performance once the number of threads is increased (Figure 5.8), which reduces this number to nine out of 20. This is especially important, because the overall decreased speedups of Polly indicate that the baseline generally achieved higher performance than in the case of four threads. In fact, only two benchmarks (*heat-3d* and *lu*) cannot keep up with the relative performance gained by using more threads in combination with the GNU backend. As a result, those two benchmarks achieve lower speedup values, compared to the baseline.

Regarding *trmm*'s execution times, the dynamically scheduled variants are now nearly two times faster (compared to four threads), which also explains the largely reduced speedup values. These are 1.1, 6.1 and 6.8 for dynamic, guided and static scheduling variants.

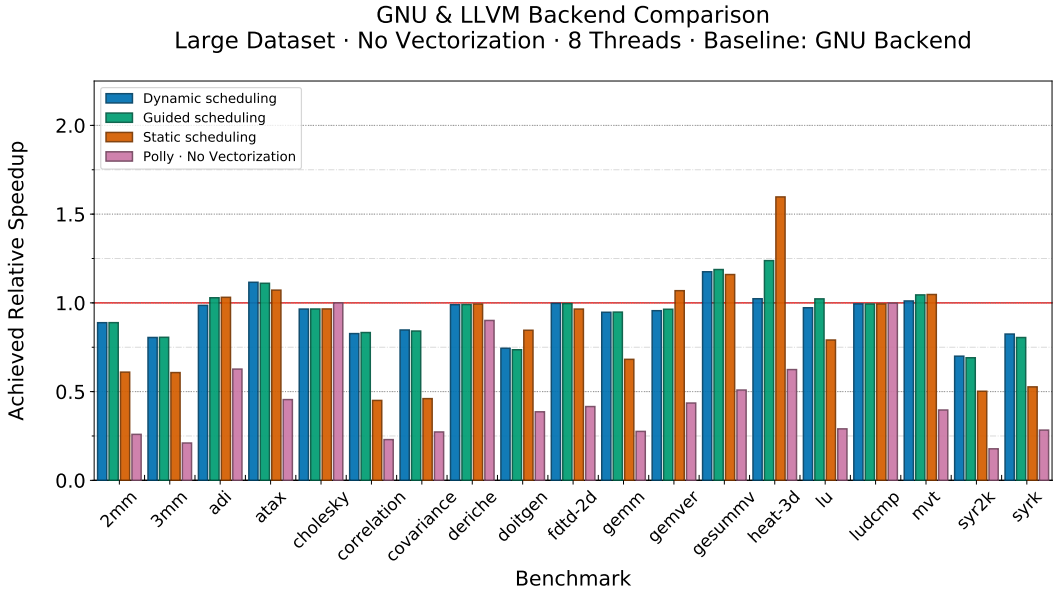

Figure 5.8: GNU and LLVM OpenMP-Backends – Large Dataset – 8 Threads

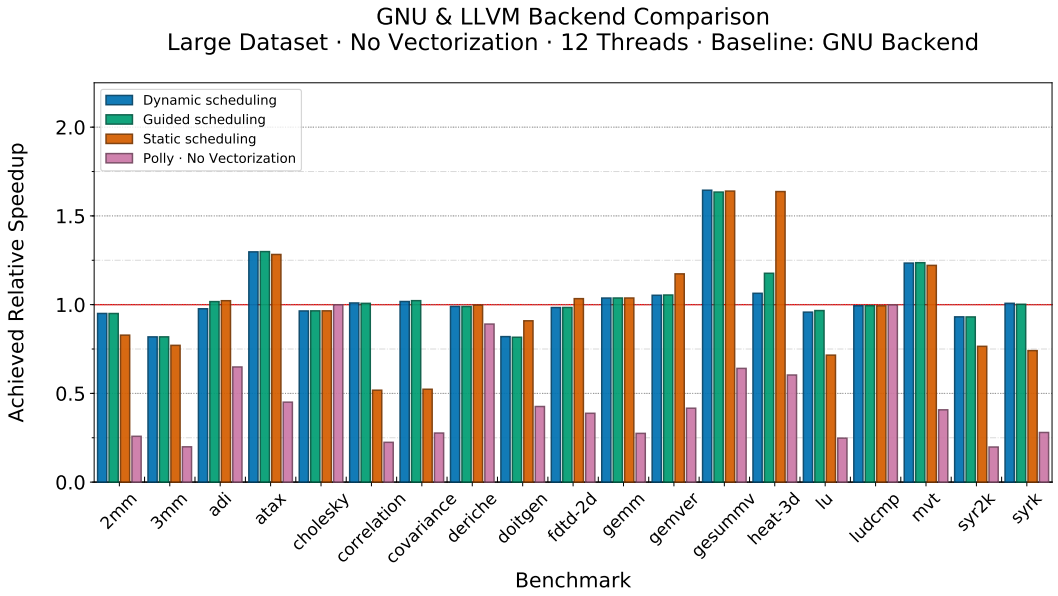

Figure 5.9: GNU and LLVM OpenMP-Backends – Large Dataset – 12 Threads

Once we shift from eight to twelve threads (Figure 5.9) the performance gain becomes quite noticeable and only six cases are left where our backend is slower with every kind of scheduling. But only three cases remain where the LLVM backend is considerably (i.e. more than 5%) slower : *3mm*, *doitgen* and *syr2k*. Then again, our backend exceeds the baseline in five other cases with speedups of 1.2 to 1.6. One of them is *trmm*, where we even achieve speedups of 4.9 (guided) to 9.3 (static). On the other hand, dynamic scheduling does not provide any benefit at all (speedup of ca. 1) for *trmm* and is even outperformed by Polly, which scores a 2× speedup.

These results suggest, that the maximum number of threads should be used to achieve comparable results with our backend. Moreover, we can see that once twelve threads are used, the results become generally quite comparable and we can even surpass the performance of the existing backend in specific cases. In particular, this is the case when changing the scheduling strategy may provide large improvements (*heat-3d* and *trmm*).

GNU/LLVM – Small Datasets

Now we move on to the small dataset size, where we use a quite large scale on the y-axis. While the case of four threads (Figure 5.10) might not seem interesting, we should remember that Polly was slower in the previous comparison (Figure 5.7). Note that Polly's runtimes are **not** dependent on the thread count.

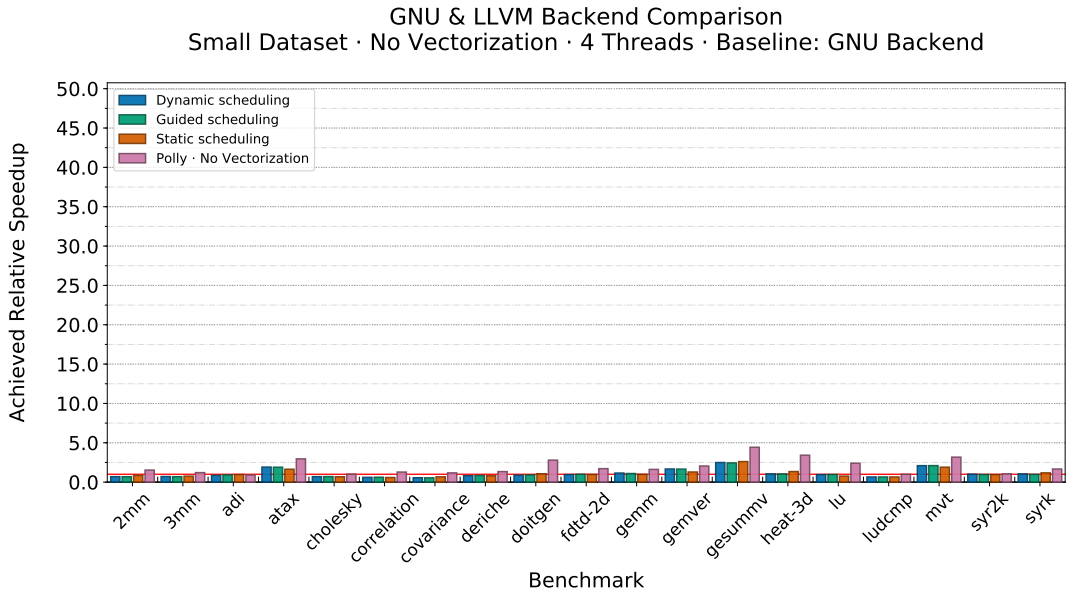

Figure 5.10: GNU and LLVM OpenMP-Backends – Small Dataset – 4 Threads

Polly will oftentimes achieve speedups beyond 1× and the relative speedup increases once the number threads is *raised* to eight (Figure 5.11).

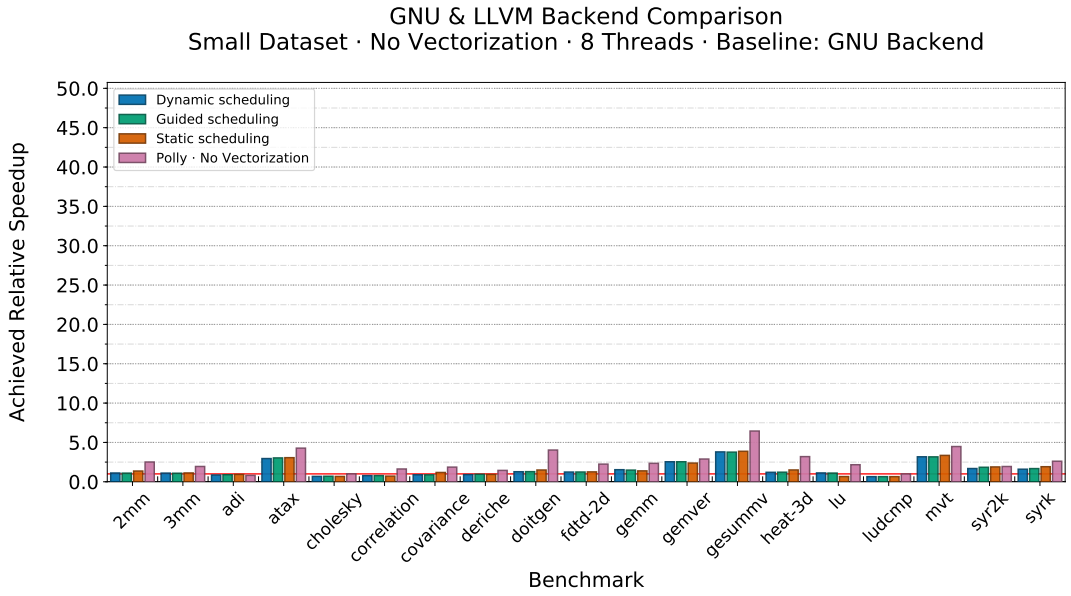

Figure 5.11: GNU and LLVM OpenMP-Backends – Small Dataset – 8 Threads

Once twelve threads (Figure 5.12) are specified, speedup values reach questionable magnitudes for the relative speedups of Polly, with peaks that lie beyond 30×. Interestingly, *trmm* remains quite calm (Polly variant peaks at around 13×) and there are several benchmarks where no peculiarities occur.

The reason for the increased speedup values is that the runtimes of many programs that were compiled using the GNU backend rise under certain circumstances, which explains the high speedups. For example *mvt*'s execution time grows from 76μs to 108μs to 981μs (four, eight, and twelve threads) – as opposed to our backend: 36μs, 34μs and 32μs.

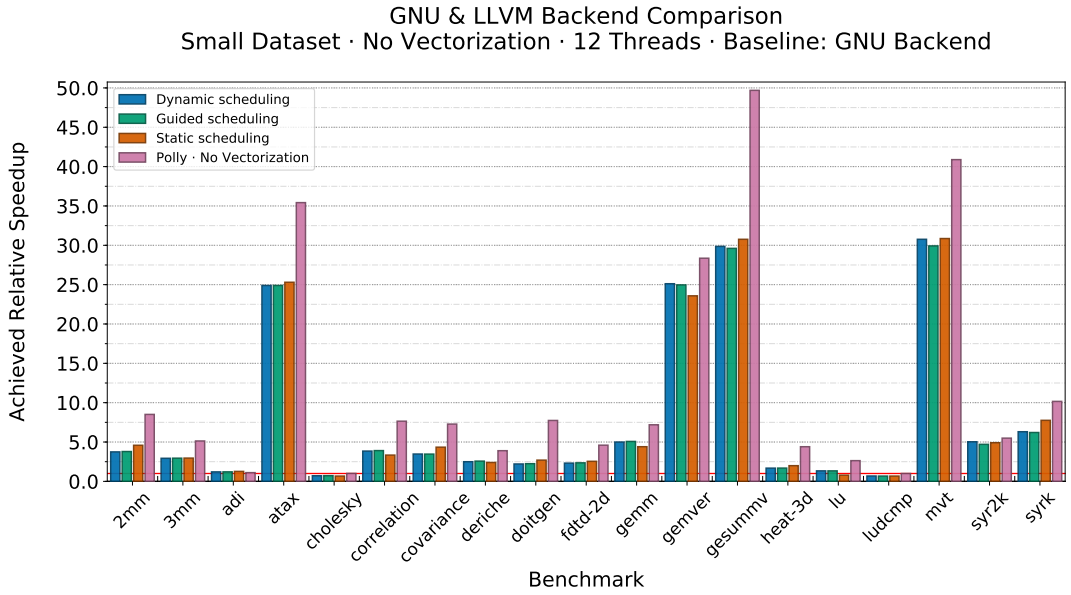

Figure 5.12: GNU and LLVM OpenMP-Backends – Small Dataset – 12 Threads

We can only speculate that the setup cost of threads might be quite high, when using the GNU OpenMP runtime. Especially when looking at the last plot, we can draw the conclusion that it is not beneficial to parallelize in an obligatory way, since Polly is faster in nearly every scenario. This emphasizes the importance of the right choice of applied transformations.

Generally it *might* be beneficial to use all available threads, but distributing the information (bounds, stride, variables) and collecting results after the calculation takes time. In contrast to the increasing runtimes of the GNU backend, the results of our backend stay roughly the same, (as demonstrated by the *mvt* example). But still, Polly without OpenMP parallelization excels in this small dataset setting, and demonstrates that thread-level parallelism is even unfavorable under specific circumstances.

5.2.3 General Evaluation of Transformations

Eventually, we will compare the different polyhedral program transformations against the *usual* `O3` optimization level. Therefore, we set `clang-4.0 -O3` as our baseline and take Polly with and without vectorization, as well as both OpenMP-backends *with* vectorization into account. Note that *lu* has only two valid results, because the other variations could not be compiled. Since we anticipate high speedups, which should increase with dataset size, we will investigate large and small datasets (absolute results can be found in Table 5.2).

We will therefore begin with the latter (Figure 5.13) and recognize that our last conclusion holds. Very small datasets may lose performance if they are parallelized – scheduling optimizations and vectorization on the other hand achieve positive speedups in nearly half of the benchmarks. In most favorable settings (*2mm*, *3mm*, *gesummv*) these speedups will even reach around 2.5 and more, while our backend will also reach rather small speedups of 1.5 for *3mm* and *gesummv*.

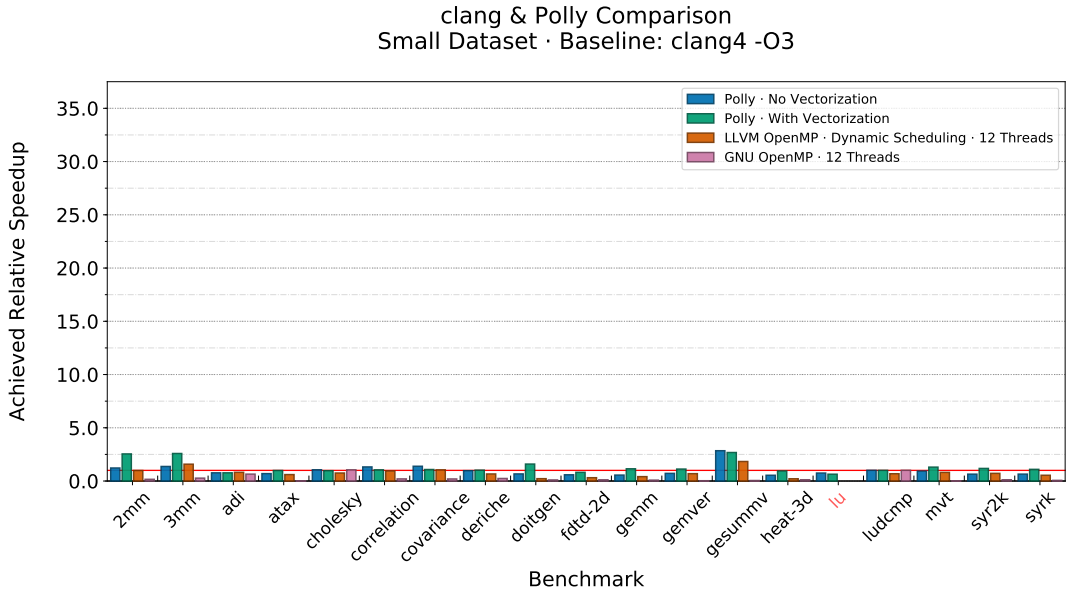

Figure 5.13: Comparison of different Transformations – Small Dataset

Lastly, the large dataset is evaluated (Figure 5.14), where we can see that the changed scenario shifts the overall picture completely. Only on rare occasions (*cholesky* and *trmm*) parallelization cannot improve on vectorization. *trmm* does not gain important speedups with the OpenMP-backends, and only around 2× with both Polly variants, when compared to clang. As we saw in Section 5.2.2, this is because *trmm* is not suited for dynamically scheduled parallelization.

The noteworthy speedups of the OpenMP parallelized code start at around 2× and get as high as 30×, imposing a significant improvement. However, we should also note that eight out of 20 programs did not improve by much and even lost performance (like *cholesky*) with every optimization variant. We investigated the special case of *cholesky*, and found that the timed code is not contained by any of the created OpenMP sections. Only the part of the program that handles initialization was executed in multiple threads, therefore no (measured) speedup may be gained.

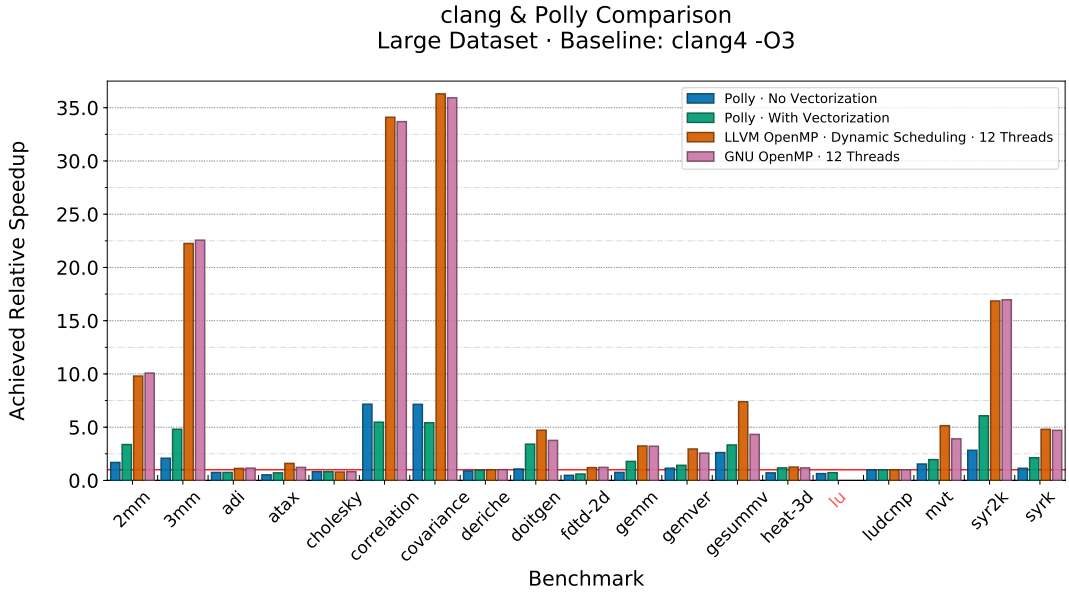

Figure 5.14: Comparison of different Transformations – Large Dataset

In this evaluation, our alternative backend is able to achieve similar speedups as Polly's current OpenMP loop generator when compared to the results of `clang -O3`, which is very promising. Therefore we want to conclude that our implementation was overall successful, and is a viable alternative for people who want (or have) to use LLVM's OpenMP libraries.

Table 5.2: Benchmark Results of `clang -O3` Comparison – absolute Runtimes

Average Benchmark Results of `clang -O3` Comparison – absolute Runtimes (in Seconds)

	`clang -O3`		`Polly No Vectorization`		`Polly With Strip Mine`		`LLVM OpenMP-Backend Dynamic Scheduling 12 Threads + Strip Mine`		`GNU OpenMP-Backend 12 Threads + Strip Mine`	
Dataset Size:	Small	Large	Small	Large	Small	Large	Small	Large	Small	Large
`2mm`	0.000206	1.543840	0.000168	0.915503	0.000081	0.458185	0.000208	0.157546	0.001275	0.153197
`3mm`	0.000379	3.006193	0.000277	1.442250	0.000147	0.625597	0.000240	0.135105	0.001401	0.133240
`adi`	0.002033	8.033012	0.002643	10.739964	0.002642	10.735671	0.002487	7.135621	0.003139	6.970367
`atax`	0.000018	0.004651	0.000026	0.008863	0.000018	0.006528	0.000030	0.002911	0.001014	0.003800
`cholesky`	0.000207	1.327227	0.000196	1.629158	0.000213	1.626948	0.000275	1.687250	0.000197	1.626862
`correlation`	0.000263	3.841846	0.000199	0.536753	0.000250	0.702498	0.000277	0.112657	0.001317	0.114025
`covariance`	0.000264	3.846521	0.000191	0.538435	0.000244	0.710497	0.000252	0.105963	0.001357	0.107059
`deriche`	0.000288	0.209365	0.000296	0.234263	0.000282	0.217160	0.000434	0.209536	0.001203	0.205780
`doitgen`	0.000245	0.422131	0.000368	0.393311	0.000154	0.123976	0.001107	0.089565	0.002405	0.112328
`fdtd-2d`	0.000247	2.252745	0.000423	4.699240	0.000303	3.758315	0.000793	1.873327	0.002159	1.838115
`gemm`	0.000101	0.571594	0.000182	0.759435	0.000088	0.320403	0.000245	0.176783	0.001204	0.178219
`gemver`	0.000031	0.017351	0.000043	0.015118	0.000028	0.012238	0.000045	0.005878	0.001052	0.006763
`gesummv`	0.000041	0.008380	0.000014	0.003202	0.000015	0.002517	0.000022	0.001134	0.000822	0.001939
`heat-3d`	0.000461	2.186447	0.000853	3.079125	0.000489	1.858315	0.002184	1.745304	0.003827	1.857358
`lu`	0.000390	3.407569	0.000521	5.319351	0.000609	4.675026	nan	nan	nan	nan
`ludcmp`	0.000353	3.254195	0.000350	3.258124	0.000352	3.254887	0.000515	3.276681	0.000350	3.254667
`mvt`	0.000022	0.013578	0.000024	0.008821	0.000017	0.006939	0.000028	0.002640	0.000980	0.003477
`syr2k`	0.000142	2.512329	0.000220	0.888285	0.000120	0.413876	0.000197	0.149084	0.001180	0.148117
`syrk`	0.000073	0.465934	0.000112	0.409302	0.000066	0.218891	0.000135	0.097115	0.001163	0.099028
`trmm`	0.000093	1.263382	0.000097	0.560358	0.000096	0.558911	0.000661	1.145054	0.001349	1.153156

6 Conclusion

Polyhedral modeling allows to analyze certain code structures and provides the means to define generic transformation functions, that enable thread- and instruction level parallelism. These transformations, paired with automatic generation of OpenMP parallelized code, offer a convenient way to efficiently deploy programs for multi-threaded architectures.

As part of the LLVM project, Polly is especially interesting. One of the most significant features is that it solely operates on LLVM's language *and* platform independent IR. Polly is offering an open-source polyhedral infrastructure, which is constantly extended and maintained by an active community – this definitely expands the impact of polyhedral compilation. Additionally, developments in the LLVM project itself provide a vast amount of different optimization and analysis passes, further supporting the future of Polly.

In this thesis we presented an extension to Polly's existing OpenMP-backend, which allows users to utilize an alternative library implementation and therefore widens the area of application. Furthermore, based on 20 selected examples of PolyBench (Table 5.1), we investigated a broad range of different scenarios that illustrate advantages and disadvantages of certain *setting–optimization* combinations. Our experiments have shown that the implementation is overall competitive to the current loop generator and provides additional possibilities to fine-tune the resulting code to a specific problem.

These are provided a user via compiler switches, and enable the customization of *thread count*, *chunk size* and *scheduling strategy*. A *thread count* switch is also offered by the existing backend, and already accelerates programs significantly. Though, our evaluation has shown that choosing a *scheduling strategy* that matches the characteristics of the considered program will increase the performance even further (by ranges of 10% to 50%). Additionally, if dynamic scheduling kinds are used, variation of the *chunk size* can be utilized to influence the employed scheduling's behavior more precisely. However, caution is advised as this adjustment can easily lead to seriously increased runtimes (even greater than 3×). Because of that, this refinement should only be used in some particular cases, where it may yield additional, noteworthy performance increases of 5% to 20%.

We are confident that this work would be a reasonable addition to Polly and are therefore planning to get in touch with the original authors to discuss further steps.

Certainly, there is still room for improvement: As we demonstrated in our evaluation, it might be interesting to implement a cost estimation for Polly's loop generation (as already shortly discussed in [14]). Depending on the result, the creation of OpenMP code may be blocked and could therefore avoid considerable overhead that is introduced in specific scenarios. As we have seen in Section 5.2.2 small problem sizes present this difficulty, where the usage of OpenMP's runtime library might increase the execution time significantly. Or the structure of the problem might not be suited, because the generated subfunction is called very often, which imposes a large amount of library calls, that lead to thread creations. It should be investigated if the latter scenario might be transformed, such that this recognized optimization opportunity is not completely lost, e.g. when threads could provably be reused. Another (further step) would be to automatically determine values for the various parameters like *scheduling kind* and *chunk size*, in a way that they are *suited* for the SCoP in consideration.

But we have seen that even in favorable settings, Polly might miss some optimization opportunities. The *cholesky* benchmark for example is not improved by any transformation, although it was designed (with regard to PolyBench) to feature a SCoP. Because of this, we think that investigating this phenomenon is particularly interesting. Unfortunately, at this time we can only speculate whether this might be resolved e.g. by extending the canonicalization pass or even has to be tackled by improving the detection of SCoPs in general. A combination of both approaches might provide synergies as an extended canonicalization can simplify the detection of certain (new) structures.

List of Figures

List of Tables

List of Acronyms

AMD	Advanced Micro Devices
API	Application Programming Interface
ARM	Advanced RISC Machine
AST	Abstract Syntax Tree
AVX2	Advanced Vector eXtensions 2
CFG	Control Flow Graph
CLooG	Clunky Loop Generator
CPU	central processing unit
DSP	Digital Signal Processor
ESA	Embedded Systems and Applications
GCC	GNU Compiler Collection
GNU	GNU's Not Unix!
HPC	High Performance Computing
IR	Intermediate Representation
isl	integer set library
LLVM	Low Level Virtual Machine
LLVM-IR	LLVM Intermediate Representation
MPI	Message Passing Interface
OpenMP	Open Multi-Processing

PoCC Polyhedral Compiler Collection

RAM Random Access Memory

SCoP Static Control Part

SIMD Single Instruction Multiple Data

SMP Symmetric Multi-Processor

SSA Static Single Assignment

Bibliography

[1] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam. “Putting polyhedral loop transformations to work”. In: *International Workshop on Languages and Compilers for Parallel Computing*. Springer. 2003, pp. 209–225.

[2] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. “The polyhedral model is more widely applicable than you think”. In: *International Conference on Compiler Construction*. Springer. 2010, pp. 283–303.

[3] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. “A practical automatic polyhedral parallelizer and locality optimizer”. In: *Acm Sigplan Notices*. Vol. 43. 6. ACM. 2008, pp. 101–113.

[4] L. Dagum and R. Menon. “OpenMP: an industry standard API for shared-memory programming”. In: *IEEE computational science and engineering* 5.1 (1998), pp. 46–55.

[5] T. Grosser. “Enabling polyhedral optimizations in LLVM”. MA thesis. 2011.

[6] T. Grosser, A. Groesslinger, and C. Lengauer. “Polly—performing polyhedral optimizations on a low-level intermediate representation”. In: *Parallel Processing Letters* 22.04 (2012), p. 1250010.

[7] J. Huthmann, B. Liebig, J. Oppermann, and A. Koch. “Hardware/software co-compilation with the nymble system”. In: *Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2013 8th International Workshop on*. IEEE. 2013, pp. 1–8.

[8] R. M. Karp, R. E. Miller, and S. Winograd. “The organization of computations for uniform recurrence equations”. In: *Journal of the ACM (JACM)* 14.3 (1967), pp. 563–590.

[9] C. Lattner and V. 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*. IEEE Computer Society. 2004, p. 75.

[10] C. Lengauer. “Loop parallelization in the polytope model”. In: *International Conference on Concurrency Theory*. Springer. 1993, pp. 398–416.

[11] S. Pop, A. Cohen, C. Bastoul, S. Girbal, G.-A. Silber, and N. Vasilache. “GRAPHITE: Polyhedral analyses and optimizations for GCC”. In: *Proceedings of the 2006 GCC Developers Summit*. Citeseer. 2006, pp. 179–197.

[12] L.-N. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache. “Iterative optimization in the polyhedral model: Part I, one-dimensional time”. In: *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society. 2007, pp. 144–156.

[13] L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos. “Iterative optimization in the polyhedral model: Part II, multidimensional time”. In: *ACM SIGPLAN Notices*. Vol. 43. 6. ACM. 2008, pp. 90–100.

[14] A Raghesh. “A Framework for Automatic OpenMP Code Generation”. In: *M. Tech thesis, Indian Institute of Technology, Madras, India* (2011).

[15] L. Sommer, J. Oppermann, J. Hofmann, and A. Koch. “Synthesis of interleaved multithreaded accelerators from OpenMP loops”. In: *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. Dec. 2017, pp. 1–7.

[16] S. Verdoolaege. “isl: An integer set library for the polyhedral model”. In: *International Congress on Mathematical Software*. Springer. 2010, pp. 299–302.