Équipe ICPS
Université
de Strasbourg

Université de Strasbourg

msii
ÉCOLE DOCTORALE

École Doctorale Mathématiques, Sciences de l'Information et de l'Ingénieur
Laboratoire des sciences de l'Ingénieur, de l'Informatique et de l'Imagerie

THÈSE présentée par :

Harenome Razanajato

soutenue le : **24 septembre 2020**

pour obtenir le grade de : **Docteur de l'Université de Strasbourg**
Discipline / Spécialité : **Informatique**

Polyhedral Code Generation: Reducing Overhead and Increasing Parallelism

THÈSE dirigée par :

M. **BASTOUL Cédric**	Professeur à l'Université de Strasbourg, France
M. **LOECHNER Vincent**	Maître de Conférences à l'Université de Strasbourg, France

RAPPORTEURS :

M. **IRIGOIN François**	Directeur de recherche à MINES ParisTech, France
M. **ALIAS Christophe**	Chargé de recherche, Inria Grenoble Rhône Alpes, France

Examinateurs :

Mme **EISENBEIS Christine**	Directrice de recherche, Inria Saclay Île-de-France, France
M. **MEISTER Benoît**	Fellow and Managing Engineer, Reservoir Labs, États-unis

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Computer science is the study of computation and information [30]. In the current information era, the most critical issue is speed: increasingly huge amounts of data need to be processed in the shortest possible time [1].

[1] Other concerns are power consumption and cost. Usually, decreasing compute time also reduces these two variables.

Reaching this goal with modern automatic computers is a two fold challenge: designing effective algorithms and translating these algorithms into programs that utilize the full potential of the target hardware. Unfortunately, scientific problems become more and more complex and technology keeps evolving: scientists specialize and become experts in very narrow fields. Being able to solve problems from a very specific category and to keep track of the latest hardware capabilities at the same time is hard.

Here programming languages and compilers come into play. Programming languages provide developers with an abstraction layer while compilers are in charge of effectively transforming the abstraction into a program that makes the best use of the available hardware. Ideally, this lets programmers focus on how to solve a problem and not on how to use the hardware.

Transfering the responsibility of reaching high performance to the compiler implies that good compilers can not merely translate high level languages into machine code: compilers must be able to automatically optimize the programs.

The spectrum of possible automatic optimizations is vast. Loop nests are known to be among the main culprits for long compute times. In consequence, great effort has been done to optimize loops [2]. A popular way

to optimize loops is to use the polyhedral model [35]. This thesis deals with polyhedral code generation and how to improve the generated code by reducing control overhead (see Chapter 4), reducing synchronization overhead (see Chapter 5) and increasing parallelism with pipelined multithreading (see Chapter 6).

1.1 Programming Languages And Compilers

A computer left to its own devices will do nothing. It requires instructions. Modern computers are electronic machines that do not understand natural speech. Rumor has it (see Figure 1.1) that some expert programmers can rely on bugs to control their computers.

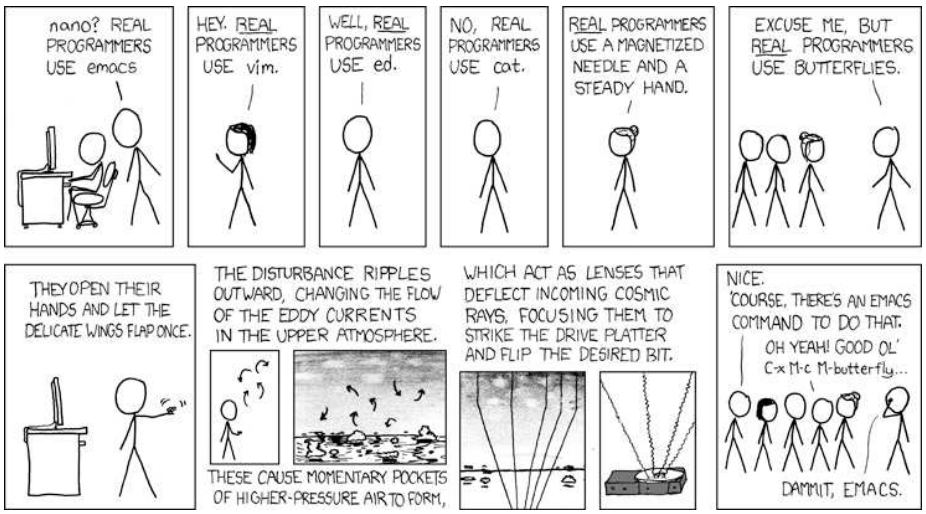


Figure 1.1: **Real Programmers** [72]. "Real programmers set the universal constants at the start such that the universe evolves to contain the disk with the data they want." (image credit: Randal Munroe, xkcd comics).

A more realistic way to address a computer is to use the same language as the computer. They are built in a way that certain strings of binary digits have meaning. However, writing binary strings is rather impractical. An equivalent human-readable way to write programs for a computer is to write assembly code: each word is associated with a valid binary string. A simple program can then be used to convert the assembly code into a binary. For instance, Listing 1.1 shows how to write an "Hello, world." in `x86` assembly.

Most developers do not write assembly code. It allows very fine control over what parts of the computer are used which may be necessary to reach optimal performance in very specific situations. However, in the general case

```
.global _start
.text
message:
  .ascii  "Hello, world.\n"
_start:
  # C equivalent: 'write(1, message, 14);'
  mov     $1, %rax        # write: syscall 1
  mov     $1, %rdi        # stdout: 1
  mov     $message, %rsi # address of string to output
  mov     $14, %rdx       # number of bytes
  syscall

  # C equivalent: 'exit(0);'
  mov     $60, %rax
  mov     $60, %rdi
  syscall
```

Listing 1.1 – **"Hello, World." In X86 Assembly.** Targeting another architecture will require, at least, modifications of this code. To write this kind of code, the developer is also required to know how the target architecture works (in this example, syscalls numbers, what registers are available).

it imposes a strong constraint: extended knowledge of the inner works of the target machine are required. Another caveat of assembly code is that it is architecture dependent. Each processor family supports a different assembly language. A program may have to run on multiple architectures: different end users may own different kinds of computer (i.e. with different architectures). Furthermore, it is common to find multiple microprocessor architectures within a single computer. As a programmer, maintaining an assembly code base can quickly become a hassle if multiple architectures must be supported.

This is the main reason why high level programming languages exist. A programming language is a restricted set of keywords, grammar and syntax rules that can be used to describe programs. A compiler will then translate the program for the target hardware. The advantage is that it is the responsibility of the compiler to support new architectures while the computer scientist can focus on implementing efficient algorithms. Moreover, high level programming languages provide abstractions so that the developer does not

need to know about the hardware.

For instance, Listing 1.2 is an example of "Hello, world." written in C11. As opposed to the code given in Listing 1.1, this program can be used on multiple architectures without modification: it just needs to be translated with the appropriate compiler. Also, notice how the main part is shorter: writing "Hello, world." roughly requires 10 lines (lines 3-4 and 7-15) in Listing 1.1 while it only amounts to 3 lines (lines 4-6) in Listing 1.2.

```
1 #include <stdlib.h>
2 #include <unistd.h>
3 int main(int argc, char* argv[argc + 1]) {
4   const char* const hello = "Hello, world.\n";
5   write(1, hello, 14);
6   exit(0);
7 }
```

Listing 1.2 – **"Hello, World." In C11 (On A POSIX-Compliant Operating System).** The program does not need to be modified to target different architectures: the developper just needs to use a compiler that supports the architecture (assuming — for this example — that a POSIX-compliant and the C standard library will be available).

Compilers make life easier for developers. Early developers had to both design effective algorithms and then find out how the best way to map these algorithms onto hardware. Compilers take care of the translation so that modern programmers can focus on the algorithms. More can be done: compilers can also automatically optimize code.

1.2 Automatic Optimizations

It is commonly accepted that literal translation from a spoken language to another is rarely adequate. Conveying the initial message sometimes requires to select the appropriate idiom. The same principle also applies to compilers.

Listing 1.3 a simple loop. This is the code most developers would write to accumulate the product of two arrays in another array. This code sequentially multiplies the elements of the arrays and updates the result in the des-

tination array. On modern parallel hardware, the performance of a sequential execution of this code would be subpar. Indeed, this loop is *embarassingly parallel*. For instance, it can divided into small parts that can be executed concurrently. Each of these parts can also be vectorized (some processors can apply the some operation to multiple operands at the same time).

```
1 for (size_t i = 0; i < N; ++i)
2   A[i] += B[i] * C[i];
```

Listing 1.3 – **An *Embarassingly Parallel* Loop.**

Experienced developers could tune the code from Listing 1.3 to achieve better performance. There are many ways to do so (inlining assembly code, using intrinsics, parallelizing with pthreads or OpenMP, etc.). However, just as writing assembly code ties the program to a specific architecture, modifying the code for performance could also restrict the set of computers that may benefit from these adjustments.

Moreover, these optimizations could be done automatically during compilation. We argued that compilers make life easier for programmers by leveraging the need to care about the hardware. They can also leverage many optimizations. Not only may the compiler be a better judge on the appropriate way to optimize the code for the target hardware, this also requires less effort and time from the developer.

Automatically optimizing the kind of loop given in Listing 1.3 can, for example, be done using tools based on the polyhedral model [35]. The polyhedral model[2] is an abstraction of loops as unions of polyhedra. Various transformations can be applied to the polyhedra depending on the target optimization. Then, a code that scans the integer points of these polyhedra is generated. This polyhedral model takes root in work on parallelization [51, 57], parametric integer programming [32] and code generation algorithms [7]. It eventually found its way into compiler infrastructures or automatic optimizers such as GCC/Graphite [82], LLVM/Polly [47, 45], R-Stream [69], Pluto [19, 16] or PPCG [106]. Our interest in this manuscript is the generation of optimized code by polyhedral techniques.

[2] also known as the polytope model

1.3 Outline and Contributions

This thesis focuses on improving code generation in polyhedral compilers for resulting programs to achieve shorter execution times. Code generation in the polyhedral model has been considered until now as mature. We show that there is still significant room for improvement and propose several techniques to reduce the execution time of the generated code.

First of all, Chapter 2 introduces the scientific notions required to understand the remainder of this manuscript. It presents the polyhedral model and gives an overview of fork-join parallelization and its implementation in the OpenMP framework. Chapter 3 discusses related work.

Chapter 4 presents our first contribution: how to reduce control overhead in the generated code. It demonstrates that the internal representation used in polyhedral compilers influences the end results and impacts runtime control overhead. We propose to further refine Bastoul's [11] extension of Quilleré, Rajopadhye and Wilde [85] algorithm. Our technique uses Loechner and Wilde's [65] chamber decomposition of parameterized polyhedra to split polyhedra during code generation in order to reduce the control overhead of the generated code. Part of the work outlined in this chapter is also presented in the following paper:

> Harenome Razanajato, Vincent Loechner and Cédric Bastoul. "Splitting Polyhedra to Generate More Efficient Code". In: *IMPACT 2017, 7th International Workshop on Polyhedral Compilation Techniques*. Stockholm, Sweden, Jan. 2017. URL: https://hal.inria.fr/hal-01505764

Chapter 5 tackles the problem of reducing the number of synchronizations in parallel code generated by a polyhedral compiler. State-of-the-art automatic parallelizers generate code where parallel loops are enclosed in distinct single parallel regions. We argue that generating separate parallel regions is subpar for it incurs thread team management overhead and superfluous synchronizations. This chapter explains how to generate a joint parallel region and how to take advantage of this unified region to remove unnecessary synchronizations. The polyhedral model is used to analyze dependencies and determine where synchronization barriers are needed and where they can be omitted. This work is also presented in a paper:

Harenome Razanajato, Cédric Bastoul and Vincent Loechner. "Lifting Barriers Using Parallel Polyhedral Regions". In: *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*. IEEE. 2017, pp. 338–347. DOI: 10.1109/HiPC.2017.00046

Chapter 6 discusses our third contribution: *pipelined multithreading* of sequential loop nests. It shows how to identify groups of sequential loops generated by an automatic parallelizer that could benefit from pipelining. Although individual loops may be sequential, interlacing their iterations introduces pipelined parallelism. This work was first introduced in:

Harenome Razanajato, Cédric Bastoul and Vincent Loechner. "Pipelined Multithreading Generation in a Polyhedral Compiler". In: *IMPACT 2020, in conjunction with HiPEAC 2020*. Bologna, Italy, Jan. 2020. URL: https://hal.inria.fr/hal-02456521

Finally, Chapter 7 concludes this thesis and discusses perspectives on future work.

Chapter 2

Scientific Background

This thesis discusses code generation and how to reduce control overhead, how to reduce synchronizations and how to increase parallelism during the code generation phase of polyhedral compilers. The remainder of this manuscript uses concepts from the polyhedral model and notions of parallelization. This chapter introduces the scientific background and notations that will be used throughout the thesis. Section 2.1 introduces the polyhedral model, an intermediate representation and framework that allows compilers to perform code transformations and optimizations. Section 2.2 presents parallel execution paradigms and, more specifically, focuses on the fork-join model and its implementation in the OpenMP framework.

2.1 Polyhedral Model

A great amount of the total compute time of a program is spent in loop nests. An experienced developer may be able to optimize a given set of loop nests. In order to alleviate this burden to experts and to give inexperienced programmers access to these optimizations, extensive work has been done to enable automatic optimizations of loop nests within compilers [2].

The polyhedral model (or polytope model) is used in computer science to analyze, transform and optimize *Static Control Parts* (also known as SCoPs) of programs, or affine loop nests [35]. A SCoP is a set of a loop nests where loop bounds and conditionals are affine expressions of outer loop indices or

constant parameters. A visual overview of the use of the polyhedral model in compilers is shown in Figure 2.1. Roughly, there are three stages:

1. Raising the original code in the geometrical view as a set of polyhedra associated to each statement. Polyhedra may represent iteration domains, iteration ordering, data accesses, data dependences, etc.
2. Performing some geometrical transformations in this view while ensuring the code is correct (i.e. the semantics of the original code is preserved).
3. Lowering back the set of polyhedra to generated code.

2.1.1 Polyhedra

[3] or *d-dimensional* polyhedron

DEFINITION 1 (Implicit representation). A d-polyhedron[3] $\mathcal{P}$ is a subspace of $\mathbb{Q}^d$ that can be formally described by a system of inequalities and equalities:

$$\mathcal{P} = \left\{ x \in \mathbb{Q}^d \;\middle|\; Ax \geq a \right\} = \left\{ x \in \mathbb{Q}^d \;\middle|\; Bx = b, Cx \geq c \right\} \tag{2.1}$$

where $A \in \mathbb{Z}^{n \times d}$ and $a \in \mathbb{Z}^n$. As the system $Ax \geq a$ may contain *implicit equalities*, it is sometimes written as $Bx = b, Cx \geq c$ where $B \in \mathbb{Z}^{m_1 \times d}$, $C \in \mathbb{Z}^{m_2 \times d}$, $b \in \mathbb{Z}^{m_1}$ and $c \in \mathbb{Z}^{m_2}$ to explicitly express the equalities and inequalities such that $Ax \geq a \Leftrightarrow Bx = b, Cx \geq c$.

DEFINITION 2 (Explicit representation). A d-polyhedron $\mathcal{P}$ can be written as the composition of a linear combination of lines, a positive linear combination of rays and a convex combination of vertices:

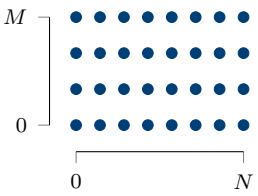
$$\mathcal{P} = \left\{ x \in \mathbb{Q}^d \;\middle|\; x = L\lambda + R\mu + V\nu, \forall \lambda, \forall \mu \geq 0, \forall \nu \geq 0, \sum \nu = 1 \right\} \tag{2.2}$$

where $L \in \mathbb{Q}^{n_1 \times d}$ represents the lines, $\lambda \in \mathbb{Q}^{n_1}$, $R \in \mathbb{Q}^{n_2 \times d}$ represents the rays, $\mu \in \mathbb{Q}^{n_2}$, $V \in \mathbb{Q}^{n_3 \times d}$ represents the vertices and $\nu \in \mathbb{Q}^{n_3}$. This representation is also known as the Minkowski representation [70, 97].

The representations given in Definition 1 and Definition 2 are equivalent. Computing one of the representation from the other can be done either using the simplex method [68] or the Chernikova algorithm [37, 60].

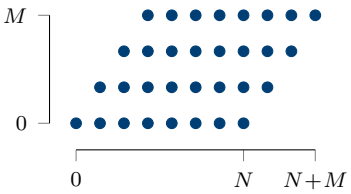
```
1  for (i = 0; i <= N; ++i)
2    for (j = 0; j <= M; ++j)
3      S(i, j);
```

(a) **Input *Static Control Part*.** Loop bounds are affine expressions of outer loop iterators or constant parameters. (See Definition 11)



$$\mathcal{D}_{in}\begin{pmatrix}N\\M\end{pmatrix} = \left\{ \begin{pmatrix}i\\j\end{pmatrix} \in \mathbb{Q}^2 \;\middle|\; \begin{array}{l} 0 \leq i \leq N, \\ 0 \leq j \leq M \end{array} \right\}$$

(b) **Mathematical Representation.** Polyhedron $\mathcal{D}_{in}$ represents the *iteration domain* (see Definition 14) of statement S.



$$\mathcal{D}_{out}\begin{pmatrix}N\\M\end{pmatrix} = \left\{ \begin{pmatrix}c1\\c2\end{pmatrix} \in \mathbb{Q}^2 \;\middle|\; \begin{array}{l} 0 \leq c1 \\ c1 \leq N + M \\ 0 \leq c2 \\ c1 - N \leq c2 \\ c2 \leq M \\ c2 \leq c1 \end{array} \right\}$$

(c) **Mathematical Transformation.** For instance, skewing can be applied to $\mathcal{D}_{in}$. This results in $\mathcal{D}_{out}$.

```
1  for (c1 = 0; c1 <= N+M; c1++)
2    for (c2 = max(0, c1-N); c2<= min(M, c1); c2++)
3      S(c1 - c2, c2);
```

(d) **Output Code.** A new code is generated. It scans the integer points of polyhedron $\mathcal{D}_{out}$.

1 Raising

2 Transformation

3 Code Generation

Figure 2.1 – **Polyhedral Model Overview.** First, an input SCoP is analyzed and represented as a union of polyhedra. Next, various mathematical transformations may be applied depending on the optimization goals. Lastly, a new code that scans the resulting union of polyhedra is generated.

DEFINITION 3 (Homogeneous representation). The *homogeneous form* $\hat{\mathcal{P}}$ of a *d-polyhedron* $\mathcal{P}$ is the result of the application of the transformation $x \to \begin{pmatrix} \xi x \\ \xi \end{pmatrix}$ to $\mathcal{P}$:

$$\hat{\mathcal{P}} = \left\{ \begin{pmatrix} \xi x \\ \xi \end{pmatrix} \in \mathbb{Q}^{d+1} \;\middle|\; \hat{B} \begin{pmatrix} \xi x \\ \xi \end{pmatrix} = 0, \hat{C} \begin{pmatrix} \xi x \\ \xi \end{pmatrix} \geq 0 \right\} \tag{2.3}$$

where $\hat{B} = [B \mid -b]$ and $\hat{C} = \left[\begin{array}{c|c} C & -c \\ \hline 0 \dots 0 & 1 \end{array}\right]$.

The original polyhedron $\mathcal{P}$ is the result of the intersection of $\hat{\mathcal{P}}$ with the $\xi = 1$ hyperplane:

$$\left[\begin{array}{c|c} B & -b \end{array}\right] \begin{pmatrix} x \\ 1 \end{pmatrix} = 0, \left[\begin{array}{c|c} A & -a \\ \hline 0 \dots 0 & 1 \end{array}\right] \begin{pmatrix} x \\ 1 \end{pmatrix} \geq 0 \Leftrightarrow Bx = b, Cx \geq c$$

This transformation can also be applied to the explicit representation and results in the homogeneous form of the Minkowski representation as a cone, transforming the vertices into rays:

$$\hat{\mathcal{P}} = \left\{ \begin{pmatrix} \xi x \\ \xi \end{pmatrix} \in \mathbb{Q}^{d+1} \;\middle|\; \begin{pmatrix} \xi \\ \xi \end{pmatrix} = \hat{L}\lambda' + \hat{R}\mu, \forall\lambda, \forall\mu \geq 0, \right\} \tag{2.4}$$

where $\hat{L} = \left[\begin{array}{c} L \\ \hline 0 \dots 0 \end{array}\right]$ and $\hat{R} = \left[\begin{array}{c|c} R & V \\ \hline 0 \dots 0 & 1 \dots 1 \end{array}\right]$

EXAMPLE 1. Consider the 3-polyhedron $\mathcal{P}_0$[4]:

$$\mathcal{P}_0 = \left\{ \begin{pmatrix} x \\ y \\ z \end{pmatrix} \in \mathbb{Q}^3 \;\middle|\; \begin{array}{l} 0 \leq x \leq 20, \\ 0 \leq y \leq 7, \\ 10 \leq z \leq 10, \end{array} \right\} \tag{2.5}$$

This polyhedron, as illustrated in Figure 2.2, is a rectangle on the $z = 10$ plane. Hence, its implicit representation can be expressed both with implicit

[4] Note that we introduce here an equivalent, *human-friendly*, notation. We will privilege this representation as often as possible.

equalities as in Equation (2.6) or with explicit equalities as in Equation (2.7):

$$\mathcal{P}_0 = \left\{ \begin{pmatrix} x \\ y \\ z \end{pmatrix} \in \mathbb{Q}^3 \;\middle|\; \begin{pmatrix} 1 & 0 & 0 \\ -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & -1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} \geq \begin{pmatrix} 0 \\ -20 \\ 0 \\ -7 \\ 10 \\ -10 \end{pmatrix} \right\} \tag{2.6}$$ implicit equalities

$$= \left\{ \begin{pmatrix} x \\ y \\ z \end{pmatrix} \in \mathbb{Q}^3 \;\middle|\; \begin{array}{l} \begin{pmatrix} 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = 10, \\ \begin{pmatrix} 1 & 0 & 0 \\ -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} \geq \begin{pmatrix} 0 \\ 20 \\ 0 \\ 7 \end{pmatrix} \end{array} \right\} \tag{2.7}$$ explicit equalities

The corresponding homogeneous representation (with implicit equalities) using a single constraints matrix is:

$$\mathcal{P}_0 = \left\{ \begin{pmatrix} x \\ y \\ z \end{pmatrix} \in \mathbb{Q}^3 \;\middle|\; \begin{pmatrix} 1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 20 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 7 \\ 0 & 0 & 1 & -10 \\ 0 & 0 & -1 & 10 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \geq 0 \right\} \tag{2.8}$$

2.1.2 Incidence Matrix

DEFINITION 4 (Saturation). A line or ray r of the explicit homogeneous form of a d-polyhedron $\hat{\mathcal{P}}$ saturates a constraint c ($c \in \hat{B}$ or $c \in \hat{C}$) from the implicit homogeneous form of $\hat{\mathcal{P}}$ if $c \cdot r = 0$.

When a ray saturates a constraint, it means it lies on the facet of the polyhedron limited by this constraint.

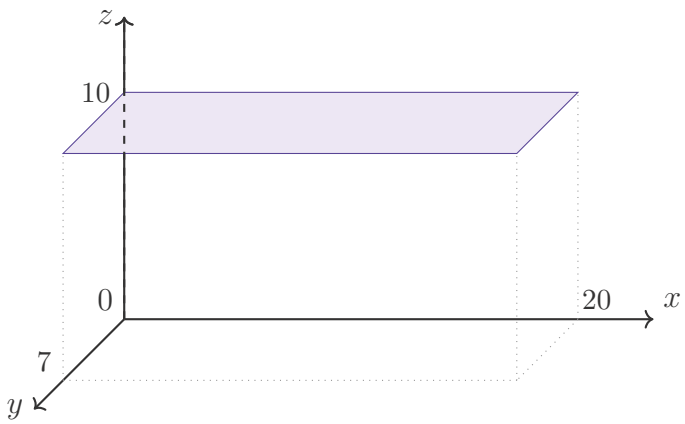


Figure 2.2 – **3D Representation Of Polyhedron $\mathcal{P}_0$.** It is defined as a 3-polyhedron but the resulting polyhedron is a 2D rectangle on the $z = 10$ plane.

DEFINITION 5 (Incidence matrix). The incidence matrix S is a boolean matrix where rows correspond to constraints (lines from $\hat{A}$ and $\hat{B}$ from Equation (2.3)) and columns correspond to lines and rays (lines from $\hat{L}$ and $\hat{R}$ from Equation (2.4)). Each element $S_{i,j} \in S$ is the boolean value which indicates whether a line or ray r_j saturates as constraint c_i:

$$S_{i,j} = \begin{cases} 1 \text{ if } c_i \cdot r_j = 0, \\ 0 \text{ otherwise.} \end{cases} \tag{2.9}$$

The incidence matrix tells which intersection of constraints (as equalities) generate a ray, and which rays lie on the facet corresponding to a constraint.

2.1.3 Faces And k-faces

DEFINITION 6 (Supporting hyperplane). An hyperplane of dimension $d-1$ which intersects the hull of a d-polyhedron $\mathcal{D}$ without intersecting its relative interior is a *supporting hyperplane* of $\mathcal{D}$.

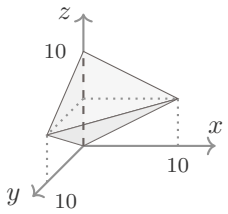
DEFINITION 7 (Face). A face of a polyhedron $\mathcal{D}$ is the intersection of $\mathcal{D}$ and a supporting hyperplane of $\mathcal{D}$.

DEFINITION 8 (k-face). A face of a polyhedron $\mathcal{D}$ is called a k-face if it is a k-polyhedron.

More specifically, the 0-faces of a d-polyhedron are its *vertices* and the $(d-1)$-faces its *facets*.

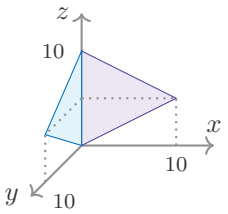
EXAMPLE 2. Consider the 3-polyhedron $\mathcal{P}_k$ (a tetrahedron) as described in Equation (2.10):

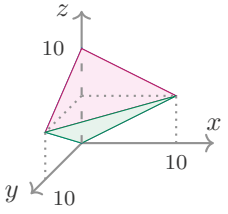
$$\mathcal{P}_k = \left\{ \begin{pmatrix} x \\ y \\ z \end{pmatrix} \in \mathbb{Q}^3 \;\middle|\; \begin{array}{l} 0 \leq x \\ 0 \leq y \\ x + y \leq 10 \\ \dfrac{x+y}{2} \leq z \\ z \leq 10 - \dfrac{x+y}{2} \end{array} \right\} \tag{2.10}$$



This 3-polyhedron has four 2-faces (its *facets*), six 1-faces (its *edges*) and four 0-faces (its *vertices*). The four *facets* (f_1 to f_4) of $\mathcal{P}_k$ can be described as:

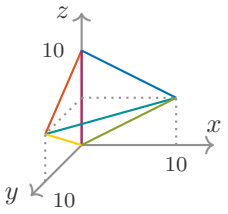
$$f_1 = \left\{ (x,y,z) \in \mathbb{Q}^3 \;\middle|\; 0 \leq x \leq 10, y = 0, \frac{x}{2} \leq z \leq 10 - \frac{x}{2} \right\}$$
$$f_2 = \left\{ (x,y,z) \in \mathbb{Q}^3 \;\middle|\; 0 \leq y \leq 10, x = 0, \frac{y}{2} \leq z \leq 10 - \frac{y}{2} \right\}$$
$$f_3 = \left\{ (x,y,z) \in \mathbb{Q}^3 \;\middle|\; x + y = 10, 5 \leq z \leq 10 \right\}$$
$$f_4 = \left\{ (x,y,z) \in \mathbb{Q}^3 \;\middle|\; x + y = 10, 0 \leq z \leq 5 \right\}$$



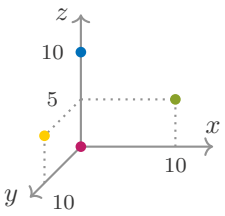


The six *edges* (e_1 to e_6) can be expressed as follows:

$$e_1 = \left\{ (x,y,z) \in \mathbb{Q}^3 \;\middle|\; z = 10 - \frac{y}{2}, 0 \leq y \leq 10, x = 0 \right\}$$
$$e_2 = \left\{ (x,y,z) \in \mathbb{Q}^3 \;\middle|\; x = 0, y = 0, 0 \leq z \leq 10 \right\}$$
$$e_3 = \left\{ (x,y,z) \in \mathbb{Q}^3 \;\middle|\; z = 10 - \frac{x}{2}, 0 \leq x \leq 10, y = 0 \right\}$$
$$e_4 = \left\{ (x,y,z) \in \mathbb{Q}^3 \;\middle|\; z = 5, x + y = 10 \right\}$$
$$e_5 = \left\{ (x,y,z) \in \mathbb{Q}^3 \;\middle|\; z = \frac{y}{2}, 0 \leq y \leq 10, x = 0 \right\}$$
$$e_6 = \left\{ (x,y,z) \in \mathbb{Q}^3 \;\middle|\; z = \frac{x}{2}, 0 \leq x \leq 10, y = 0 \right\}$$



Finally, the four *vertices* (v_1 to v_4) are:



$$v_1 = \left\{(x, y, z) \in \mathbb{Q}^3 \;\middle|\; x = 0, y = 0, z = 10\right\}$$
$$v_2 = \left\{(x, y, z) \in \mathbb{Q}^3 \;\middle|\; x = 0, y = 10, z = 5\right\}$$
$$v_3 = \left\{(x, y, z) \in \mathbb{Q}^3 \;\middle|\; x = 10, y = 0, z = 5\right\}$$
$$v_4 = \left\{(x, y, z) \in \mathbb{Q}^3 \;\middle|\; x = 0, y = 0, z = 0\right\}$$

2.1.4 Parametric Polyhedra

DEFINITION 9 (Parametric polyhedron). A parametric *d-polyhedron* $\mathcal{D}(p)$ is a polyhedron where the constant part linearly depends on m parameters:

$$\mathcal{D}(p) = \left\{x \in \mathbb{Q}^d \;\middle|\; Ax \geq A'p + a\right\} \tag{2.11}$$

where $A \in \mathbb{Z}^{n\times d}$, $A' \in \mathbb{Z}^{n\times m}$, $p \in \mathbb{Z}^m$ and $a \in \mathbb{Z}^n$.

DEFINITION 10 (Combined polyhedron). A parametric *d-polyhedron* $\mathcal{D}(p)$ with m parameters can be represented as a non-parametric polyhedron, or *combined polyhedron*, $\mathcal{D}'$ in the *combined* space $\mathbb{Q}^{d+m}$:

$$\mathcal{D}' = \left\{ \begin{pmatrix} x \\ p \end{pmatrix} \in \mathbb{Q}^{d+m} \;\middle|\; \left[\; A \;\middle|\; -A' \;\right] \begin{pmatrix} x \\ p \end{pmatrix} \geq a \right\} \tag{2.12}$$

The intersection of the combined polyhedron with a hyperplane cutting the parameter space (with a given value of p) is equivalent to the instanciation of the parameter p.

2.1.5 Static Control Part

DEFINITION 11 (Static Control Part). A *Static Control Part* or SCoP is a set of program statements where loop bounds, conditionals and data accesses are affine expressions of outer loop indices or constant parameters.

EXAMPLE 3. Listing 2.1 provides an example of a SCoP. Statement `S1` is enclosed in two nested loops over iterators `i` and `j`. All loop bounds and array accesses are affine forms of outer loop indices or constant parameters (`N` and `M`).

```
1 for (i = 0; i < N; ++i)
2   for (j = 0; j < M; ++j)
3     S1: A[j] = A[j] + B[i];
```

Listing 2.1 – **Simple SCoP Example**

2.1.6 Iteration Domain

DEFINITION 12 (Statement instance). Statements within loop nests may be executed more than once. A *statement instance* is a given execution of a statement. It corresponds to some given values of the enclosing loop iterators.

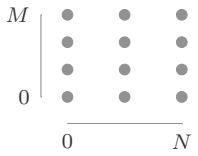
DEFINITION 13 (Iteration vector). A given *statement instance* is uniquely identified by its *iteration vector*: it consists of the iterator values of the surrounding loops.

EXAMPLE 4. Statement `S1` from Listing 2.1 is located in the body of a loop nest of dimension 2. Hence, *iteration vectors* of statement `S1` have two components. There are $N \times M$ distinct instances of statement `S1` from Listing 2.1. Assuming that $N \geq 1$ and $M \geq 2$, $\begin{pmatrix} 0 \\ 0 \end{pmatrix}$, $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$ or $\begin{pmatrix} 0 \\ M-1 \end{pmatrix}$ are valid *iteration vectors* for statement `S1` whereas $\begin{pmatrix} 0 \\ M+2 \end{pmatrix}$ is not a possible *statement instance* of `S1`.

DEFINITION 14 (Iteration domain). The *iteration domain* $\mathcal{D}_S(\vec{p})$ of a statement S is the set of its possible iteration vectors. It may depend on fixed yet unknown values or *parameters*. This set can be represented as the integer points of a parametric polyhedron:

$$\mathcal{D}_S(\vec{p}) = \left\{ \vec{\imath}_S \;\middle|\; D_S \begin{pmatrix} \vec{\imath}_S \\ \vec{p} \\ 1 \end{pmatrix} \geq \vec{0} \right\} \tag{2.13}$$

where $\vec{p}$ is the vector of parameters, $\vec{\imath}_S \in \mathbb{Z}^{dim(\vec{\imath}_S)}$ stands for an iteration vector of statement S, and $D_S \in \mathbb{Z}^{m_{D_S} \times (dim(\vec{\imath}_S)+dim(\vec{p})+1)}$ — where m_{D_S} is the number of constraints — is an integer matrix that encodes the constraints.



EXAMPLE 5. The iteration domain $\mathcal{D}_{S1}$ for statement S1 from Listing 2.1 is:

$$\mathcal{D}_{S1}\begin{pmatrix} N \\ M \end{pmatrix} = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \middle| \begin{matrix} 0 \leq i < N \\ 0 \leq j < M \end{matrix} \right\} \tag{2.14}$$

Here, the iteration vector $\vec{\imath}_{S1}$ is composed of the two loop indices (i, j) and the vector of parameters $\vec{p}$ of the two unknown program variables (N, M).

2.1.7 Relation

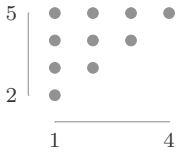
DEFINITION 15. A relation $R : E \to F$ is a subset $G \subseteq E \times F$. $x \in E$ is said to be related with $y \in F$ if $(x, y) \in G$. It is commonly noted [5] $x\ R\ y$ or $R\ x\ y$.

[5] the postfix notation $x\ y\ R$ can also be used but is seldom seen

EXAMPLE 6. Let R be the relation between two sets $E = \{1, 2, 3, 4\}$ and $F = \{2, 3, 4, 5\}$ such that $x\ R\ y\ (x \in E, y \in F)$ if $x < y$:

$$R = \left\{ \begin{pmatrix} x \\ y \end{pmatrix} \in E \times F \middle| x < y, x \in E, y \in F \right\} \tag{2.15}$$

This relation can be expressed as a polyhedron where each integer point $\begin{pmatrix} x \\ y \end{pmatrix}$ of the polyhedron corresponds to a relation $x\ R\ y$ of $R : E \to F$:



$$R = \left\{ x \to y \middle| \begin{pmatrix} 1 & 0 & -1 \\ -1 & 0 & 4 \\ 0 & 1 & -2 \\ 0 & -1 & 5 \\ -1 & 1 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \geq 0 \right\} \tag{2.16}$$

2.1.8 Dependence Relation

DEFINITION 16 (Dependence relation). Dependences between statement instances of a source statement *S* and a target statement *T* can be represented as a relation between *iteration vectors*. Each integer point in the polyhedron associated with the relation signifies that there is a dependency between the

corresponding input and output *iteration vectors*. Such a polyhedron can be defined by the following relation:

$$\delta_{S,T}(\vec{p}) = \left\{ \vec{\imath_S} \rightarrow \vec{\imath_T} \;\middle|\; R_{S,T} \begin{pmatrix} \vec{\imath_S} \\ \vec{\imath_T} \\ \vec{p} \\ 1 \end{pmatrix} \geq \vec{0} \right\} \tag{2.17}$$

DEFINITION 17 (Dependence graph). Dependencies between *statement instances* can be represented with directed graphs: each vertex is associated with a program statement and labeled with its *iteration domain* while edges symbolize an existing dependence relation between two program statements and are labelled with the corresponding dependence relation.

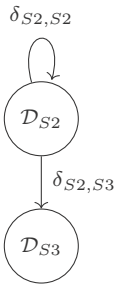
EXAMPLE 7. For the SCoP example in Listing 2.1 there is a dependency due to the consecutive accesses to array `A`, from iteration (i, j) to iteration (i', j'), when $i < i'$ and $j = j'$. The dependence relation can be expressed as:

$$\delta_{S1,S1}(\vec{p}) = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \rightarrow \begin{pmatrix} i' \\ j' \end{pmatrix} \;\middle|\; \begin{array}{l} i < i' \\ j = j' \end{array} \right\} \tag{2.18}$$

EXAMPLE 8. Consider the SCoP given in Listing 2.2. There are two dependence relations: a dependency between instances of `S2` and a dependency between instances of `S2` and `S3`.

$$\delta_{S2,S2}(\vec{p}) = \left\{ \left(i\right) \rightarrow \left(i'\right) \;\middle|\; i = i' - 1 \right\} \tag{2.19}$$

$$\delta_{S2,S3}(\vec{p}) = \left\{ \left(i\right) \rightarrow \left(i'\right) \;\middle|\; i = i' \right\} \tag{2.20}$$




```
1 for (i = 1; i < N; ++i)
2   S2: A[i] = A[i - 1] + B[i];

3 for (i = 1; i < N; ++i)
4   S3: C[i] = A[i] * D[i];
```

Listing 2.2 – **Simple SCoP**

2.1.9 Schedule Relation

DEFINITION 18 (Schedule relation). Schedule relations determine the temporal ordering between statement instances. To do so, each instance of a statement is associated with a logical date $\vec{t_S}$.

$$\theta_S(\vec{p}) \;=\; \left\{ \vec{i_S} \;\rightarrow\; \vec{t_S} \;\middle|\; T_S \begin{pmatrix} \vec{i_S} \\ \vec{t_S} \\ \vec{p} \\ 1 \end{pmatrix} \;\geq\; \vec{0} \right\} \tag{2.21}$$

Schedule relations are used to reorder statement instances. For example, parallelizing SCoPs amounts to determining a new scheduling where multiple statement instances have the same logical date (or where some logical date dimensions are identified as parallel) while ensuring that all dependencies are preserved [19, 34]. Ensuring dependencies are preserved means ensuring that all vectors of the scheduled (transformed by the scheduling relation) dependence relations are lexicographically positive vectors.

2.1.10 Code Generation

The last step of the polyhedral compilation scheme is code generation. Once a scheduling has been decided, the corresponding code can be generated. A polyhedral code generation algorithm shall produce a code that scans each point of the polyhedra in the partial order specified by the new scheduling [54, 85]. The most recent refinements to the code generation problem can be found in CLooG [11] (further discussed in Section 4.2 of Chapter 4), CodeGen+ [23] and isl [46].

2.2 Parallel Execution Paradigms

One way to decrease the compute time of a given executed code is to execute instructions concurrently. Two things are needed for concurrent execution: hardware capable of parallel execution and software that can use the parallel features of this hardware. Many architectures, models and programming

paradigms have been proposed to tackle this challenge.

2.2.1 Hardware Capabilities Overview

Multiple levels of parallelism can be found in modern architectures:

Within Processing Units

Vectorization allows to execute one instruction on multiple data at once. The most common form found in CPUs are SIMD (Single Instruction, Multiple Data) vectors. Instruction Pipelining attempts to reduce processor idleness by dividing instructions to be executed into multiple stages. Out-of-order execution allows a processor to reorder some instructions depending on data and execution units availability. *Very Long Instruction Word* (VLIW) instructions encode multiple operations to be run in parallel.

Symmetric Multiprocessing

A single processor can feature multiple physical cores (separate processing units) and motherboards can host several processors. The physical cores in a CPU do not necessarily need to be identical. For instance ARM big.LITTLE CPUs consist of multiple cores from two different architectures (the idea is to switch between powerful or energy-saving cores depending on the work load). CPUs can also integrate logical cores that share the same hardware.

Accelerators

Dedicated specialized units can also be used to offload computations: Graphic Processing Units (GPUs), Digital Signal Processors (DSPs), Field-Programmable Gate-Arrays (FPGAs), Application-Specific Integrated Circuits (ASICs) or other accelerators. These specialized units can also feature inner parallelism as described previously.

Distributed computing

Parallelism can also be achieved by using multiple computers and distributing work over a network. In computer clusters, nodes will execute the

same task in parallel. whereas grid computers will be given different tasks. The most powerful known supercomputers are collections of thousands of powerful interconnected computers.

The work presented in this thesis will mostly focus on shared memory multithreaded environments usually supported by multi core and multi CPU architectures.

2.2.2 Parallel software development

Depending on the target level of parallelism, several ways to implement parallelism are at the disposal of the programmer.

The most low level features of a processing unit usually require to write assembly code. Hardware manufacturers may also provide libraries to access specific features of the processors from high level languages. For instance, Intel provides intrinsics [50] for SSE, AVX, MMX (and more) instructions and Arm provides intrinsics [8] for NEON instructions.

Various libraries and frameworks allow developers to introduce parallelism in their programs and can be combined to take advantage of multiple forms of parallelism at once. For example, multithreading can be achieved with POSIX pthreads [48] or C11 threads [1, 49]. CUDA [74] or OpenCL [73] provide facilities for offloading to specialized units. OpenMP [26, 79] and OpenACC [109, 77] can support (depending on the implementation) both multithreading and offloading via compiler directives. MPI [40, 107] allows to distribute computations both on a single computer or on multiple computers over the network.

Some languages such as Cilk, Go, or X10 are specifically designed for parallel computing (either implicitly or explicitly).

At a higher level, programs can also be used to schedule tasks on a computer (for instance, GNU parallel, xargs, xjobs) or on multiple networked computers (for example slurm, pssh).

Without loss of generality, we mainly focus on OpenMP parallel programming in this thesis.

2.2.3 Shared Memory Fork-Join: The OpenMP Framework

OpenMP [26, 79] is an *Application Programming Interface* (API) for shared memory parallel programming. The specification of OpenMP is overviewed by the OpenMP Architecture Review Board consortium while the actual implementation is left to third parties. Hence, several competing implementations exist (for instance, gcc/libgomp, clang/libomp, icc/libiomp) and various independent extensions have been proposed (and sometimes adapted in subsequent versions of the specification).

The programmer controls the execution of an OpenMP program by using compiler directives, calls to functions of the OpenMP library or environment variables. The library routines offer fine control over the execution flow of the program. However restricting oneself to compiler directives results in portable code that can still be compiled without support for OpenMP with minimal effort[6], as compilers are authorized to ignore unrecognized compiler directives. In C or C++ programs, these compiler directives (for OpenMP) start with `#pragma omp`.

[6] this could be achieved with preprocessor as well but it is more involving

Fork-join model

DEFINITION 19 (fork-join model). The *fork-join* model is a parallel design pattern where sequential execution of program is occasionally interrupted with parallel sections or regions. At the start of a parallel region, the execution *forks* into multiple threads that may be executed in parallel. The threads eventually *join* at the end of the parallel region. The program then resumes sequential execution.

Parallel Regions

Parallel regions in OpenMP are delimited with the `parallel` construct. When the execution flow of a program encounters this construct, a thread team is created. By default, all threads in the thread team will execute the code within the region. A barrier is implied at the end of the region: sequential execution resumes only once all threads have reached the end. Some aspects of the parallel region can be customized using clauses. Here follow a few examples of such clauses. The number of threads can be changed using the

`num_threads` clause. The `shared`, `private` or `firstprivate` clauses set the visibility and behaviour of variables from outer scopes.

EXAMPLE 9. Consider Listing 2.3. The parallel region (lines 3-6) contains a single instruction which may be executed multiple times: each thread in the thread team will execute the whole code enclosed in the parallel region. Hence `printf("Hello, World.\n")` (line 5) will be executed as many times as the number of threads in the thread team.

```
1 #include <stdio.h>
2 int main(int argc, char* argv[argc + 1]) {
3   #pragma omp parallel
4   {
5     printf("Hello, World.\n");
6   }
7   return 0;
8 }
```

Listing 2.3 – **Hello, World With OpenMP.** The corresponding program will print as many "`Hello, World.`" as the number of threads.

Worksharing Constructs

Various *worksharing* constructs are provided to control how the threads in a thread team execute some code in parallel. All threads will indiscriminately execute the code within a parallel region until a worksharing construct is encountered. At this point the execution will differ in the threads in a way that depends on the worksharing construct. A barrier is also implied at the end of all worksharing constructs, it may be lifted using the `nowait` construct.

Two of those *worksharing* constructs are particularly relevant to this thesis: the `single` and `for` constructs. The `single` construct enforces the execution of the enclosed code by only one thread while the `for` construct distributes parts of a loop to threads. We will also discuss the `task` construct which is used to create explicit tasks that may be executed right away or at a later point in time.

The `for` construct

The `for` construct is used in conjunction with `for` loops to distribute iterations. In C programs, the corresponding directive is `#pragma omp for`.

The loop is divided into *chunks* that are distributed to threads. If not specified, the chunk size choice is left to the underlying OpenMP implementation. Each thread will execute one *chunk* at a time (in the case where there are more *chunks* than threads) but independent threads may run in parallel. The iterations within a *chunk* are executed sequentially.

EXAMPLE 10. Consider Listing 2.4. A loop that prints successive integers (lines 6-7) is distributed with the loop construct (line 5). Assuming the thread team contains more than one thread, the iterations of the loop will be divided in chunks that will be executed in parallel: the numbers will most likely not be printed in the original total order. However, total order *within* chunks will still be preserved because the contents of a *chunk* are executed sequentially.

For instance, if the loop is divided in 4 *chunks* of 4 iterations, the sequence $0, 1, 2, 3$ will always be printed in this order but other numbers may be printed before, in between or after those numbers.

```
1  #include <stdio.h>
2  int main(int argc, char* argv[argc + 1]) {
3    #pragma omp parallel
4    {
5      #pragma omp for
6      for (size_t i = 0; i < 16; ++i)
7        printf("%zu\n", i);
8    }
9    return 0;
10 }
```

Listing 2.4 – **Loop Iterations Distribution With The For Construct.** The numbers will not be printed in order because *chunks* will be executed in parallel.

If the sole content of a parallel region is a unique for construct, the `for` and `parallel` constructs may be combined as `#pragma omp parallel for`. This is how current polyhedral automatic parallelizers parallelize loops.

Multiple clauses can be used to tune the behaviour of the for construct. Here follows a brief description of some of the clauses that will be used in the remainder of this thesis.

The `schedule(<policy>, <size>)` clause can be used to control chunk distribution and chunk sizes. The `size` parameter may be omitted. The scheduling policy determines how the chunks are distributed among threads. In particular, with the `static` policy, chunks sizes are approximately identical (except the very last chunk which may be smaller) and the chunks are distributed in a round robin fashion to the threads: it is possible to know in advance which thread will execute which chunk. If the chunk size is not specified, at most one chunk is distributed to each thread.

The `nowait` clause allows to omit the implicit barrier at the end of the worksharing construct. It is the responsibility of the developer to ensure that the semantics of the program is preserved. In the case of loops executed with the `static` policy, the specification gives conditions under which compliant implementations will guarantee safe use of the `nowait` clause on a parallel loop succeeded by another parallel loop:

- both iteration domains are of the same size
- both loops have the same chunk size (be it explicitly specified or the default value)
- both loops are in the same parallel region
- neither loops are associated with a SIMD construct[7]

[7] another OpenMP construct which we will not discuss

The `ordered` clause can be used along the `ordered` construct to execute *parts* of a parallel loop sequentially. The clause is added to the `for` construct to indicate the presence of a sequential body and the `ordered` construct is used in the loop body to delimit the sequential part. Only one `ordered` construct may appear in a loop body. Code outside the `ordered` construct may still be executed in parallel.

Example 11. Consider Listing 2.5. The parallel loop (lines 5-10) contains an ordered construct (line 8-9). The code outside of this construct (line 7) will be executed in parallel, while all iterations of the ordered construct will remain sequential.

```
1  #include <stdio.h>
2  int main(int argc, char* argv[argc + 1]) {
3    #pragma omp parallel
4    {
5      #pragma omp for ordered
6      for (size_t i = 0; i < 8; ++i) {
7        printf("parallel: %zu\n", i);
8        #pragma omp ordered
9        printf("ordered: %zu\n", i);
10     }
11   }
12   return 0;
13 }
```

Listing 2.5 – **Ordered Loop.** The numbers preceded with the "`parallel`" will be printed in parallel whereas the numbers preceded with the "`ordered`" text will be printed in sequence.

Note that this construct was referred to as *loop* construct in prior versions (up to 4.5 [78]). But version 5 [79] introduced another *loop* construct and renamed the previous one as the *worksharing-loop* construct. To avoid confusion, we have referred to the *worksharing-loop* construct as the `for` construct.

The Single Construct

The `single` construct is used to specify that only one thread shall execute the enclosed code. The choice of the thread that will execute the code is implementation defined. Some clauses may be used to further tune the construct. In particular, the `nowait` clause may be used to omit an implicit barrier.

EXAMPLE 12. Consider Listing 2.6. The code in the `single` construct (lines 5-6) will be executed once by one of the threads in the thread team. The loop iterations of the parallel loop (lines 7-9) will be distributed among the threads. Note that because of the implicit barrier at the end of worksharing constructs, all threads will wait for the thread that executes the single construct before proceeding with the parallel loop.

```
1  #include <stdio.h>
2  int main(int argc, char* argv[argc + 1]) {
3    #pragma omp parallel
4    {
5      #pragma omp single
6      printf("single thread.\n");

7      #pragma omp for
8      for (i = 0; i < 8; i++)
9        printf("%zu: loop iteration\n", i);
10   }
11   return 0;
12 }
```

Listing 2.6 – **Example Of The Single Construct.** "`single thread.`" will be printed once by a single thread. Other threads will wait at this point because of the implicit barrier ah the end of the `single` construct. "`*: loop iteration`" will be printed multiple times in parallel (hence the numbers will most likely not be in ascending order).

Task Construct

The `task` construct defines explicit tasks that may be executed at a later point in time by any thread of the thread team: encountering a task construct only creates the task. Hence the task may or may not be executed right away. Threads may be assigned existing tasks at multiple points in a program. In particular at implicit or explicit barriers or at `taskwait` points.

EXAMPLE 13. Consider Listing 2.7. Because of the `single` construct (line 5), a single thread will create the tasks while other threads may proceed with the remainder (thanks to the `nowait` clause). Upon reaching the `#pragma omp taskwait`, threads will be assigned and execute existing tasks. Note that in this example, `taskwait` is superfluous because it is right before the end of the parallel region: an implicit barrier is present at the end of the parallel region.

Synchronizations

In parallel contexts, synchronizations may be necessary to guarantee the correctness of the program. By default, a barrier is implied at the end of a work-

```c
#include <stdio.h>
int main(int argc, char* argv[argc + 1]) {
  #pragma omp parallel
  {
    #pragma omp single nowait
    for (size_t i = 0; i < 8; ++i) {
      #pragma omp task
      printf("task: %zu\n", i);
    }
    #pragma omp taskwait
  }
  return 0;
}
```

Listing 2.7 – **Example Of The Task Construct.** One of the threads will create tasks. Threads will start executing existing tasks upon reaching the `taskwait` point.

sharing construct: threads will wait until all threads have reached the end of the construct.

Barriers can also be placed anywhere in a parallel region using the `barrier` construct or removed using the `nowait` clause (see previous examples). It is the responsibility of the programmer to ensure that using the `nowait` clause on a given construct does not endanger the correctness of the program (data-races, required execution order).

The low level runtime library also provides explicit locks and corresponding routines. The locks must be declared as `omp_lock_t` and manipulated using the classical `omp_init_lock()`, `omp_destroy_lock()`, `omp_set_lock()` and `omp_unset_lock()`

Chapter 3

Related Work

Several historical and seminal publications have already been referenced in Chapters 1 and 2. Section 3.1 covers polyhedral code generation. Section 3.2 presents automatic parallelization. Section 3.3 overviews pipelined parallelism.

3.1 Code Generation And Affine Control Overhead

Minimizing control overhead in generated code is a critical issue for polyhedral code generation algorithms. Two alternatives to reach this goal have been studied: either generating inefficient code then trying to remove its control overhead, or generating directly efficient code.

Code generation in the polyhedral model started with the seminal work by Ancourt and Irigoin [7]. It relies on the Fourier-Motzkin [41, 27] pair-wise elimination technique and a generates a significant amount of redundant control: Fourier-Motkin's variable elimination may greatly increase the number of inequalities. Le Fur [59] improved the technique using the simplex method [97] to remove redundant constraints.

Kelly et al. [54] showed how to scan several polyhedra when different mappings may be used for each statements. Their method first generates perfectly nested loops without duplication and then partly eliminates redundant conditionals. It is implemented in the Omega library [53]. This library relies on the Omega test [84] an extension of the Fourier-Motzkin

technique that can determine whether a dependence exists between two array references. Chen refined Kelly et al.'s method in CodeGen+ [23]. Loop overhead is removed by propagating up constraints identified as introducing overhead and duplicating code. Guards in `if` statements are simplified by comparing the constraints in a given guard condition with successives nodes and building `if-then-else` trees.

Wetzel, Lengauer and Griebl [108, 44] deal with arbitrary affine schedules by generating separate program parts and then merging them at the cost of high control overhead or longer code.

The other family of code generation algorithms stems from Le Verge [61]'s algorithm which relies on the dual representation of polyhedra to avoid redundant constraints. Parts of this work were included in the Polylib [66]. This work was extended by Quilleré, Rajopadhye and Wilde [85]'s algorithm. They proposed a recursive algorithm to scan unions of polyhedra by projecting the polyhedra onto the outermost dimensions and separating these projections into disjoint polyhedra. This method can directly generate code without guards inside the loops (except some conditions that include modulos) at the expense of potentially high code length. Bastoul [11] proposed changes to this algorithm to reduce code generation time and limit code explosion: it avoids unnecessary polyhedra separation calculations (for instance, input polyhedra may be identical or already disjoint) and merges back *point* polyhedra into *host* polyhedra. Vasilache et al. [102] further improved the algorithm with additional control overhead removal techniques. Internal guards are be removed with `if` conditional hoisting, a pass depth-first traversal of the AST that focuses only on conditionals at the current depth to separates the domains. Modulo conditions are removed using a combination of loop unrolling and strip-mining. The CLooG [11, 9] tool incorporates the methods that were introduced in this paragraph.

Grosser et al. [46] added several new extensions such as *shifted stride detection* to remove modulo conditions, *components* which avoid polyhedra separation or an *isolation* mechanism that allows users to specify a part of the space which should be processed separately to, e.g., isolate a vectorizable loop or separate partial/full tiles. The isl [104] library was initially developed to be used in CLooG (in place of PolyLib) and hence includes its code gener-

ation techniques (or equivalent methods) as well as the algorithms proposed by Grosser et al. [46].

Renganarayanan et al. [94]'s approach is based on finding the *inset* polyhedron, i.e. the polyhedron that contains full tiles origins, to distinguish full and partial tiles in and remove unnecessary loop bounds. This method targeted one-level parametric tiling method and they further extended it to multi-level parametric tiles [55, 95].

Specifying tiling information to let the code generator extract full tiles is also possible in Reservoir Labs's R-Stream Compiler [13].

Some work has been done towards control reduction in FPGA codes. For instance, Zuo et al. [113] prefer to avoid polyhedra separation and propose to simplify control using several loop bound tuning methods. Alias and Plesco [5] use semantic factorization of affine expressions to reduce control in FPGA pipelines. These methods focus on polyhedral code generation in high-level synthesis contexts.

Techniques based on Quilleré et al.'s algorithm may be strongly impacted by polyhedral splitting (see Chapter 4). It was ignored because code generation tools let the underlying polyhedral libraries choose how to split (or not to split) polyhedra regardless of the code generation problem. Isolation and full tile extraction are a first step towards considering the problem.

3.2 Synchronization Reduction

A significant part of the overall compute time of a program can be imparted to loops. Hence, the optimizing compilation community produced extensive work on loop parallelization. Lamport [57] proposed a method to find parallelism for multiprocessors. It modeled loops as iteration spaces and cutting used hyperplanes to find a schedule.

Allen and Kennedy [6] propose a parallelization algorithm that computes strongly connected components of the dependence graph to decide about convenient loop distribution and extract parallel loops. Wolf and Lam's perfectly nested loop parallelization algorithm uses a unified representation of a subset of loop transformations, known as unimodular transformations, to extract parallelism [110]. The first algorithm for a general solution to inner-

most parallelism extraction computes affine transformations and was proposed by Feautrier [33, 34]. Lim and Lam [62, 63] extended Feautrier's work to extract outermost parallel loops. Lossing et al. [67] proposed a tool for automatic generation of distributed code for task parallelization. It features multiple optmization passes including a pass for communications minimization. Bondhugula et al. [19, 16] developed the PLUTO scheduler, an automatic parallelizer and data-locality optimizer. These techniques found their way into high-level compilers such as Pluto [19], R-Stream [69] or TRACO [15], and also in low-level compilers such as GCC [99], LLVM [45] or IBM XL [18].

Most of these techniques generate parallel loops with an implicit barrier synchronization at the end of each parallel loop, such as the `omp parallel for` construct from OpenMP [79]. Few algorithms are designed to generate synchronizations e.g., Allen-Kennedy[6] and Lim-Lam [63].

Our technique (see Chapter 5) for synchronization reduction via barrier lifting does not compete but complements these techniques: we do not propose a new scheduler but a post processing phase. It takes as input an optimized generated code (or its internal polyhedral representation) and further optimizes it by building a wider parallel region to minimize runtime overhead and removing spurious synchronizations.

Synchronization barrier placement and optimization has been the subject of past works. Aiken and Gay [3] target SPMD (*Simple Program, Multiple Data*) programs. They proposed a method to check the correctness of program's synchronization pattern and language features to make synchronizations more explicit and easier to check. O'Boyle and Stöhr [76] proposed a graph-based approach to determine the smallest number of required synchronizations for a given program. Darte and Schreiber [28] proposed an algorithm in linear time for barrier minimization at all levels of loop nests. Cytron et al. [25] suggested to mix fork-join and SPMD programming. Their approach takes as input a fork-join program and converts select parts into SPMD programs. Tseng [100] further explored the idea of combining concepts from fork-join and SPMD programming models to reduce synchronization overhead. Sequential parts are executed by the master thread while parallel regions are built to be treated as SPMD programs. Barriers are eliminated using communication analysis. Feautrier, Violard and Ketterlin [36]

computes affine dates to remove clocks, which express explicit synchronizations, from X10 programs. Zhao et al. also proposed a technique which reduces task creation overhead [112] in the context of task-parallel programs.

All these publications strived for reducing synchronization overhead. Our work goes in the same direction in a fork-join context.

3.3 Pipelined Multithreading

Extensive work has been done on software pipelining. The initial idea behind software pipelining comes from Patel and Davidson [80]. They proposed to delay computations to avoid collisions by inserting *non compute segments*. Rau and Glaeser [90] initially introduced *Modulo Scheduling* [52]. They use the notion of *Minimum Initiation Interval* to ensure that a schedule is correct. Their method targeted specialized hardware (the Polycyclic Architecture) designed with software pipelining in mind. Software pipelining was improved by Lam [56] with new heuristics and *modulo variable expansion* which increases throughput by using multiple registers (in different iterations) for a variable in a loop. This method did not require specialized hardware and was applicable to VLIW architectures. Ning and Gao [75] proposed a new software pipelining method that takes into account both instruction scheduling and register allocation instead of considering them as separate passes. Rau [89, 88] proposed an iterative modulo scheduling algorithm to find near-optimal solutions. Feautrier [31] showed that resource constraints could be expressed as linear equalities that could be used, for instance, with affine scheduling algorithms [33, 34]. It was extended by Fimmel and Müller [39] who combine, instead of decoupling them, the optimization process and the search for an optimum initiation interval.

There has also been a lot of research focusing on low-level pipelined code generation [29, 4, 71], usually based on the concept of process network [98, 105]. The main concerns of these papers are the characterization of stream types and sizes and the efficient placement of tasks on a fixed sized static hardware. These issues are not relevant in a multithreaded general purpose environment like OpenMP where the number of threads is virtually infinite and the streams limited to the memory size.

Previous work introduced pipelined execution and other forms of parallelims using OpenMP. Gonzàlez et al. [43, 42] proposed an extension that allowed to specify explicit point-to-point synchronizations. A similar behaviour can be attained in recent versions of OpenMP with `task` constructs and `depend` clauses. Baudisch et al. [14] translate synchronous programs into pipelines using OpenMP sections. Each pipeline stage is enclosed in an infinite loop which is placed in an OpenMP section. Variables read and written by multiple stages are transferred using pipeline variables wich are read and written in FIFO buffers. Sbîrlea et al. [96] parallelize doaccross loops using the OpenMP `ordered` construct and clause. This transformation requires specialized input: the programs are expressed in a Data-Flow Graph Language and extended information on dependencies must be provided. Chatarasi et al. [22]'s framework handles OpenMP `tasks` and `ordered` constructs. Input programs are already annotated OpenMP programs and hence explicitly parallel. The framework can check the correctness of the input or optimize it. Pop et al. [81] proposed OpenStream as an extension of OpenMP. It can generate code which can exploit pipeline parallelism but it requires the programmer to first annotate and expose parallelism in the input code. Raman et al. [87] identify strongly connected components in the program dependence graph to construct pipelines and considers blocking to avoid false sharing. Liu et al. [64] extend the method to create pipelines in OpenMP using tasks. However, neither of these works target automatic parallelization to pipeline SCoPs. Nonetheless, all these works inspired new features in parallel languages allowing fine control over thread parallelism and synchronization which we extensively use in this thesis.

Our method for pipelined multithreading (see Chapter 6) leaves room for software pipelining when sequential loops still contain more than one instruction. As opposed to aforementioned work, which expects annotated or even explicitly parallel programs, our approach can target simple annotation-free SCoPs: it expects as input a classic optimized polyhedral AST, which may have been produced from a simple SCoP by an automatic optimizer.

Chapter 4

Reducing Affine Loop Nests Control Overhead

Code generation in polyhedral compilation frameworks corresponds to the construction of a code that scans the integer points of unions of polyhedra [35, 10]. Those polyhedra are typically the result of a smart and complex optimization process which makes sure the integer points are scanned in a satisfactory order. As many scanning codes are possible for a given set of unions of polyhedra, it is extremely important to generate an efficient scanning code where the control overhead does not impair the optimizations enabled by the polyhedral compiler.

One of the first methods to generate code that scans unions of polyhedra is to generate a code that scans the convex hull of the input union where each statement is enclosed in a guard which checks whether a given iteration of that statement belongs to the iteration domain [7]. This approach results in small code sizes at the expense of poor runtime performance because of the heavy control overhead. The state-of-the-art code generation algorithm (also known as QRW algorithm) was introduced by Quilleré, Rajopadhye and Wilde [85] and extended by various subsequent works to further improve the quality of the generated code [12, 11, 102, 46]. Although it avoids the generation of loop guards implied by the use of convex hulls, complex loop bounds may remain in the final generated code and the corresponding program may incur high control overhead.

In this chapter, we propose a new extension of this algorithm to further reduce the control overhead in loop bounds by splitting polyhedra. The idea is to split polyhedra during the code generation phase into simple polyhedra using the chamber decomposition of parametric polyhedra [65]. This chamber decomposition allows to compute the validity domains along each scanned dimension. We will use these validity domains to split complex polyhedra into unions of simple polyhedra.

This chapter is organized as follows. Section 4.1 demonstrates on a motivating example the relevance of our proposal. To understand how polyhedra splitting is performed, the extended QRW [11] code generation algorithm is recalled in Section 4.2 while Section 4.3 presents the chamber decomposition of parametric polyhedra and how it is computed. Polyhedra splitting relying on chamber decomposition is eventually explained in Section 4.4. Experimental results are detailed in Section 4.5. The limits of our technique and possible solutions are discussed in Section 4.6 before Section 4.7 concludes this chapter.

4.1 Motivation

In the polyhedral compilation context, a given polyhedron may be represented by many equivalent unions of polyhedra as long as they cover the same set of integer points. Code generation tools heavily manipulate unions of polyhedra without taking into account this fact and merely settle for the representation choice of the underlying polyhedral library. However, using different unions of polyhedra — even if mathematically equivalent — may lead to generating very different outputs. For instance, a given polyhedron may be translated into a code which requires high control overhead while an alternative union of polyhedra would correspond to a longer yet lighter — control-wise — code.

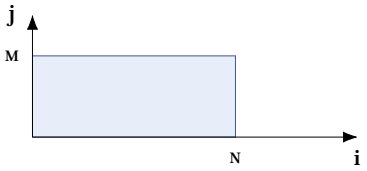
Consider the iteration domain $\mathcal{P}_0$:

$$\mathcal{P}_0(N,M) = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \in \mathbb{Q}^2 \;\middle|\; \begin{matrix} 0 \leq i \leq N, \\ 0 \leq j \leq M \end{matrix} \right\} \tag{4.1}$$

This iteration domain corresponds to the simple loop nest shown in Figure 4.1 which executes statement `S1` depending on two parameters N and M (with the context $N > M$).

```
1 for (int i=0 ; i<=N ; i++)
2   for (int j=0 ; j<=M ; j++)
3     S1(i,j);
```

(a) **Loop Nest That Scans $\mathcal{P}_0$.**



(b) **2D Representation Of $\mathcal{P}_0$.**

Figure 4.1 – **Simple Loop Nest Defined By Equation 4.1.** Each iterator is independent and has a single lower and a single upper bound: the loop nest scans a $N \times M$ rectangle.

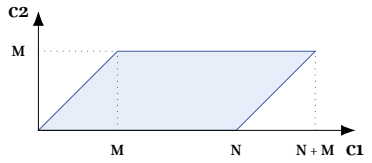
Imagine that improving locality of data accesses performed by statement `S1` or allowing parallel execution of the inner loop may be achieved by skewing this loop nest. The corresponding scheduling function which would be given to a polyhedral code generator along the polyhedron $\mathcal{P}_0$ could be:

$$\begin{pmatrix} i \\ j \end{pmatrix} \mapsto \begin{pmatrix} c1 \\ c2 \end{pmatrix} = \begin{pmatrix} i+j \\ j \end{pmatrix} \tag{4.2}$$

Applying the aforementioned transformation to polyhedron $\mathcal{P}_0$ would result in the polyhedron $\mathcal{P}_1$ as depicted in Figure 4.2 and as formally described in Equation 4.3.

$$\mathcal{P}_1(N,M) = \left\{ \begin{pmatrix} c1 \\ c2 \end{pmatrix} \in \mathbb{Q}^2 \;\middle|\; \begin{array}{l} 0 \leq c1, \\ c1 \leq N+M, \\ 0 \leq c2, \\ c1 - N \leq c2, \\ c2 \leq M, \\ c2 \leq c1 \end{array} \right\} \tag{4.3}$$

Notice the presence of `max()` and `min()`, which stem from the fact that iterator `c2` has two lower bounds constraints ($0 \leq c2$ and $c1 - N \leq c2$) and two upper bounds constraints ($c2 \leq M$ and $c2 \leq c1$). These additional operations introduce control overhead which could be avoided by consider-



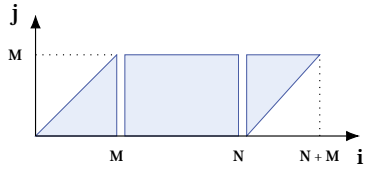
(a) Skewed Iteration Domain

```
1 for (c1=0;c1<=N+M;c1++)
2   for
  ↪   (c2=max(0,c1-N);c2<=min(M,c1);c2++)
3     S1((c1-c2),c2);
```

(b) Generated Code for Polyhedron $\mathcal{P}_1$

Figure 4.2 – **Skewed Loop Nest Represented by Equation 4.3.** The transformation from Equation 4.2 has been applied to the polyhedron $\mathcal{P}_0$, effectively transforming the input rectangle into a parallelogram. Iterator `c2` has complex loop bounds: scanning this parallelogram requires `min()` and `max()` calculations.

ing a "better" equivalent union of polyhedra for the skewed iteration domain. Our proposal is to transform polyhedra to this "better" representation before the code generation phase. The key idea of our processing is to split the outer loop in several parts, so that each inner loop has a single lower and a single upper bound constraint, as presented in Figure 4.3.



(a) Split Iteration Domain

```
1 for (c1=0;c1<=M;c1++)
2   for (c2=0;c2<=c1;c2++)
3     S1((c1-c2),c2);
4 for (c1=M+1;c1<=N-1;c1++)
5   for (c2=0;c2<=M;c2++)
6     S1((c1-c2),c2);
7 for (c1=N;c1<=N+M;c1++)
8   for (c2=c1-N;c2<=M;c2++)
9     S1((c1-c2),c2);
```

(b) New Generated Code

Figure 4.3 – **Split Alternative To The Skewed Loop Nest From Figure 4.2.** The parallelogram has been split into two triangles and one rectangle. The loops are much simpler since each iterator has only one lower and only one upper bound constraint: scanning such shapes can be done without resorting to costly `max()` or `min()` calculations at the price of a longer code.

To achieve our splitting into a form that leads to simpler generated loop bounds, we rely on the notion of *chambers* [65] which is further explained in Section 4.3. These *chambers* are used to split the polyhedron $\mathcal{P}_1$ into the

three domains shown in Figure 4.3a.

The vanilla program (the skewed loop nest from Figure 4.2b) and the split program (the split loop nest from Figure 4.3b) were compiled with parameter values $N = 10,000,000$ and $M = 1,024$ and executed on an Intel Xeon CPU E52650v3 at 2.30GHz. Several compilers and compilation options were tested. Statement S1 was defined as the assignment of an incrementing counter to an arbitrary array[1]:

```
1 A[c1%10][c2] += counter++;
```

Table 4.1 presents execution times of various versions produced with gcc 5.4. The speedup column is the speedup (or slowdown) of the split loops over the vanilla loops. With options `-O3 -march=native`, gcc achieves vectorization on both the vanilla and the split loops. However, extra runtime tests had to be inserted in the vanilla version whereas the split loops do not require any test, resulting in better performance (1.32x).

Table 4.1 – gcc Execution Times and Speedup

gcc options	vanilla	split	speedup
-O0	5.72s	5.32s	1.07
-O1	1.33s	0.95s	1.40
-O2	1.44s	0.95s	1.52
-O3 -march=native	0.29s	0.22s	1.32

The execution times of the outputs of icc 17.0 are given in Table 4.2 It seems that splitting the loops prevents some icc optimizations with option `-O0` as the split version is more than 2 times slower than the vanilla version. However, with options `-O2` and `-O3 -march=native`, icc vectorizes all loop nests in both versions. Some of the loops were either protected with runtime tests or peeled. With `-O3 -march=native`, the speedup reaches 1.67x on this example.

Table 4.3 shows the execution times of the programs when compiled using clang 3.8. It could not vectorize the vanilla loops whereas the split loops were vectorized and lead to a very important speedup of 4.63x with options `-O3 -march=native`.

[1]We used a modulo 10 in the first array index to avoid allocating a huge array.

Table 4.2 – icc Execution Times and Speedup

icc options	vanilla	split	speedup
-O0	12.33s	11.95s	1.03
-O1	1.41s	3.38s	0.42
-O2	0.36s	0.25s	1.41
-O3 -march=native	0.27s	0.16s	1.67

Table 4.3 – clang Execution Times and Speedup.

clang options	vanilla	split	speedup
-O0	7.30s	6.16s	1.18
-O1	1.02s	0.95s	1.08
-O2	1.02s	0.36s	2.85
-O3 -march=native	1.20s	0.26s	4.63

This first very simple example exhibits wide variations of performance depending on the compiler, the compilation options and the polyhedra splitting technique. Lower control overhead may lead to significant performance benefits and may also aid compilers in enabling further optimizations thanks to a simpler generated code. Significant performance benefits may come from a lower control overhead. Our technique to generate a code with simpler control is explained in Section 4.4. Beforehand, as this technique builds on existing work, the next Section details polyhedral code generation and chamber decomposition.

4.2 Extended QRW Code Generation

The state-of-the-art Quilleré, Rajopadhye and Wilde [85]'s (QRW) code generation algorithm generates loops that scan the integer points of unions of polyhedra by recursively processing each polyhedra dimension. At each dimension, a disjunct union of polyhedra is computed. The scanning code is successively generated for each resulting subset. The output code has lower runtime complexity than a convex-hull approach — since runtime verifications of iteration domain membership are unnecessary — at the expense of a larger code size. On the other hand, some tests and multiple loop bound

constraints remain and lead to *minimum* or *maximum* calculations. This is caused by the coarse granularity splitting at each level of the recursion.

Algorithm 1, as implemented in CLooG, describes Bastoul [11]'s extension of the QRW code generation algorithm. It introduces various improvements to avoid code explosion, notably by reducing the complexity of the splitting, the number of different scanned subsets and the size of the generated code, without degrading performance.

It is a recursive algorithm along the dimensions, from outermost ($d = 1$) to innermost ($d = n$). It takes as input a polyhedron list, a context and the current recursion depth. For a given loop level d, Step 1 is to intersect the input polyhedra with the context. The polyhedra are then projected onto the d outermost dimensions at Step 2. Step 3 and Step 4 then separate the polyhedra into an ordered list of disjoint polyhedra. This list of subdomains is scanned at Step 5 to generate the code for the current level d: for each of these subdomains, the lower bound and the stride are computed at Step 5(a) from the inner polyhedra that will be scanned in the subdomain. Step 5(b) merges inner polyhedra (if possible) to keep the code compact. The intersection of the input context and the loops bounds of the current loop is used as the context in recursive iterations of the algorithm to generate loops of inner dimensions at Step 5(c). Step 6 and Step 7 reduce code size by removing dead code and merging *point* polyhedra that may have been separated from their neighboring higher dimensional *host* polyhedra.

EXAMPLE 14. Figure 4.4 gives an example of code generation. The input polyhedron list contains two polyhedra $\mathcal{S}_1$ and $\mathcal{S}_2$ for statements S_1 and S_2. Step 1 of the algorithm is shown in Figure 4.4a where the polyhedra are intersected with the context[2]. Figure 4.4b encompasses Step 2 to Step 5(a) of the algorithm: the polyhedra are projected on the first dimension, separated into disjoint polyhedra and the loop bounds and strides are computed. Figure 4.4c illustrates the recursion into inner dimensions as per Steps 5(b) and 5(c). This example is left unchanged by Step 6 and Step 7.

[2]Note that — though not shown in the Figure for the sake of simplicity — the polyhedra should live in the same space and the algorithm actually uses two parametric polyhedra $\mathcal{S}_1(\vec{p})$ and $\mathcal{S}_2(\vec{p})$ with the same parameter and iterator space.

Algorithm 1: Extended QRW Code Generation

input : $T :=$ polyhedron list $\left[\mathcal{T}_{S_1}, \ldots, \mathcal{T}_{S_n}\right]$
$\mathcal{C} :=$: context (constraints on the parameters and the outer loop bounds)
$d :=$: dimension
output: $L :=$: ordered list of loops $\left[\mathcal{L}_1, \ldots, \mathcal{L}_m\right]$, a loop
$\mathcal{L}_i = (\mathcal{D}_i, s_i, \mathcal{B}_i)$ is a 3-tuple where $\mathcal{D}_i$ is the polyhedron to be scanned in this loop, s_i the stride of this loop and $\mathcal{B}_i$ is the body of the loop represented as another ordered list of loops

Function `CodeGeneration(`$T, \mathcal{C}, d$`)`

Step 1 `/* Intersect the polyhedra with the context */`
$T_{\mathcal{C}} \leftarrow \left[\mathcal{T}_{\mathcal{C}_i} \,\middle|\, \mathcal{T}_{\mathcal{C}_i} = \mathcal{T}_{S_i} \cap \mathcal{C}, \mathcal{T}_{S_i} \in T\right]$;

Step 2 $P \leftarrow$ list of the projections of $\mathcal{T}_{\mathcal{C}_i} \in T_{\mathcal{C}}$ onto the d outer dimensions;

Step 3 $D \leftarrow$ separate P into a list of disjoint polyhedra;

Step 4 `/* Build` L `from the polyhedra of` D `in lexicographical order */`
$L \leftarrow \left[(\mathcal{D}_i, \varnothing, \varnothing) \,\middle|\, \mathcal{D}_i \in D \wedge \mathcal{D}_j < \mathcal{D}_k \,\forall\, j < k\right]$;

Step 5 **foreach** $(\mathcal{D}, s, \mathcal{B}) = \mathcal{L} \in L$ **do**

5(a) compute the stride and lower bound for the current loop $\mathcal{L}$ at level d, from the inner dimensions of the polyhedra $\mathcal{T}_{S_p}$, ..., $\mathcal{T}_{S_q}$ touched by this loop and store the stride in s;

5(b) $T' \leftarrow$ new list of polyhedra created from $\mathcal{T}_{S_p}$, ..., $\mathcal{T}_{S_q}$ by merging adjacent polyhedra scanning the same set of statements in this loop subdomain ;

5(c) $\mathcal{B} =$ `CodeGeneration(`$T', \mathcal{C} \cap \mathcal{D}, d+1$`)` ;

end

Step 6 **foreach** $(\mathcal{D}, s, \mathcal{B}) \in L$ **do**

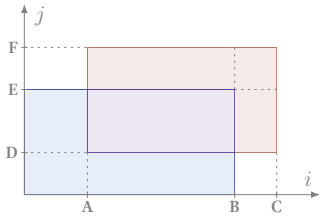
`/* Remove dead code and empty polyhedra */`
Apply steps 2 to 4 to $\mathcal{B}$;

end

Step 7 `/* Reduce code size */`
merge *point* polyhedra to adjacent *host* polyhedra in L;

Step 8 **return** L;

end

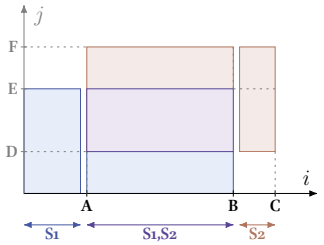


$$Context : \left\{ \begin{array}{l} 0 \leq A < B < C \\ 0 \leq D < E < F \end{array} \right\}$$

$$\mathcal{S}_1(B,E) = \left\{ (i,j) \in \mathbb{Z}^2 \;\middle|\; \begin{array}{l} 0 \leq i \leq B \\ 0 \leq j \leq E \end{array} \right\}$$

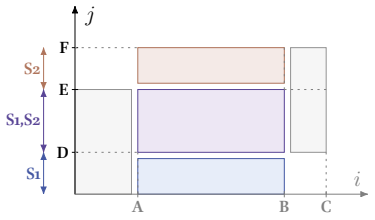
$$\mathcal{S}_2(A,C,D,F) = \left\{ (i,j) \in \mathbb{Z}^2 \;\middle|\; \begin{array}{l} A \leq i \leq C \\ D \leq j \leq F \end{array} \right\}$$

(a) **Step 1: Intersection Of The Polyhedra With The Context.** The iteration domain for $\mathcal{S}_1$ is represented in blue while the iteration domain for $\mathcal{S}_2$ is represented in red except for the part where the iteration domains overlap drawn in purple.




```
1  for (i = 0; i <= A-1; ++i)
2    /* S1 */
3  for (i = A; i <= B; ++i) {
4    /* S1 and S2 */
5  }
6  for (i = B+1; i <= C; ++i)
7    /* S2 */
```

(b) **Step 2 To Step 5(a): Projection And Separation On The First Dimension** Domains $0 \leq i < A$ and $B < i \leq C$ correspond to domains where either $\mathcal{S}_1$ or $\mathcal{S}_2$ is the sole statement whereas $\mathcal{S}_1$ and $\mathcal{S}_2$ overlap somewhere in the $A \leq i \leq B$ domain.




```
1  for (i = 0; i <= A-1; ++i)
2    for (j = 0; j <= E; ++j)
3      S1(i, j);
4  for (i = A; i <= B; ++i) {
5    for (j = 0; j <= D-1; ++j)
6      S1(i, j);
7    for (j = D; j <= E; ++j) {
8      S1(i, j);
9      S2(i, j);
10   }
11   for (j = E+1; j <= F; ++j)
12     S2(i, j);
13 }
14 for (i = B+1; i <= C; ++i)
15   for (j = D; j <= F; ++j)
16     S2(i, j);
```

(c) **Step 5(b) To Step 5(c): Recursion On The Second Dimension** The iteration domain is further separated for $A \leq i \leq B$ to distinguish the parts where $\mathcal{S}_1$ and $\mathcal{S}_2$ overlap or are isolated.

Figure 4.4 – **Simple Execution Example Of The Extended QRW Algorithm 1.** Note that Step 6 and Step 7 have no effect on this particular example.

4.3 Chamber Decomposition

The chambers of a parametric polyhedron define constraints on the parameters so that in each chamber there exists a single affine expression of the vertices of the parametric polyhedron, or, equivalently, so that the parametric polyhedron has strictly (not piecewise) affine bounds.

Loechner and Wilde [65] propose an algorithm for computing the chambers of a parametric polyhedron $\mathcal{P}$. The idea is to consider a parametric polyhedron of dimension d with m parameters as the combined space $\mathcal{P}'$ of the iterators and parameters and then to find the m-faces of this combined space. These m-faces are then used to compute the parametric vertices and corresponding validity domains which are the chambers of the parametric polyhedron.

Chamber decomposition of a d-polyhedron parametric with m parameters consists of three steps :

1. Finding the m-faces of the combined polyhedron (see Subsection 4.3.1).
2. Computing the list of (parametric vertex, validity domain) couples (see Subsection 4.3.2).
3. Separating the validity domains into disjunct domains (see Subsection 4.3.3).

We will derive this concept to our purpose in Section 4.4 by computing the chambers of a slightly modified version of the inner polyhedra: outer loop indices must be temporarily viewed as additional parameters.

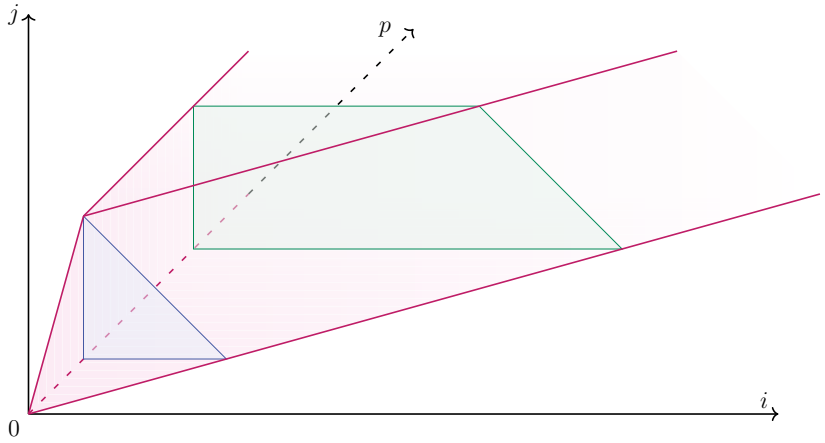
As a running example along this section, we will use the parametric 2-polyhedron $\mathcal{D}_0(p)$ illustrated in Figure 4.5. It is defined by Equation 4.4 while a 3D representation of the combined polyhedron $\mathcal{D}'_0$ is depicted in Figure 4.5b. Table 4.4 contains the corresponding 1-faces, parametric vertices and validity domains.

$$\mathcal{D}_0(p) = \left\{ (i,j) \in \mathbb{Q}^2 \;\middle|\; \begin{array}{l} i \geq 0 \\ i \leq p - j \\ j \geq 0 \\ j \leq 10 \end{array} \right\} = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \;\middle|\; \begin{pmatrix} 1 & 0 & 0 & 0 \\ -1 & -1 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 10 \end{pmatrix} \begin{pmatrix} i \\ j \\ p \\ 1 \end{pmatrix} \geq \vec{0} \right\} \tag{4.4}$$

$$\mathcal{R}_0 = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 10 & 10 \end{pmatrix} \tag{4.5}$$

$$\mathcal{I}_0 = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \end{pmatrix} \tag{4.6}$$

(a) **Polyhedron $\mathcal{D}_0(p)$, Corresponding Rays $\mathcal{R}_0$ And Incidence Matrix $\mathcal{I}_0$.**



(b) **3D Representation Of Combined Polyhedron $\mathcal{D}'_0$.** The combined polyhedron $\mathcal{D}'_0$ is drawn in purple. The thick purple lines are its 1-faces. Instances of $\mathcal{D}_0$ are triangles for $p \leq 10$ and quadrilaterals for parameter $p > 10$. For example, $\mathcal{D}_0(p = 10)$ is drawn in blue and has three vertices: $\begin{pmatrix} 0 \\ 0 \end{pmatrix}$, $\begin{pmatrix} p \\ 0 \end{pmatrix}$ and $\begin{pmatrix} 0 \\ p \end{pmatrix}$. $\mathcal{D}_0(p = 30)$ is drawn in green and has four vertices: $\begin{pmatrix} 0 \\ 0 \end{pmatrix}$, $\begin{pmatrix} p \\ 0 \end{pmatrix}$, $\begin{pmatrix} 0 \\ 10 \end{pmatrix}$ and $\begin{pmatrix} p - 10 \\ 10 \end{pmatrix}$.

Figure 4.5 – **Running Example Polyhedron $\mathcal{D}_0(p)$ For Section 4.3** Section 4.3.1 will explain how to find the 1-faces of the combined polyhedron $\mathcal{D}'_0$ and Section 4.3.2 will detail how to use these 1-faces to compute the validity domains and parametric vertices of polyhedron $\mathcal{D}_0(p)$.

4.3.1 Finding The k-faces Of A Polyhedron

A k-polyhedron $\mathcal{F}$ is a k-face of a d-polyhedron $\mathcal{P}$ if and only if:

- it saturates at least $(d - k)$ equalities and inequalities, including all equalities
- it contains $k + 1$ independent lines, rays and vertices

Algorithm 2 describes how to find the k-faces of a d-polyhedron $\mathcal{P}$ using its incidence matrix. The set of k-faces is progressively constructed by considering all $(d - k)$ combinations of constraints from the input polyhedron. If a given set satisfies the two conditions stated above, it is a k-face. The algorithm proposes to remember what constraints have already been inspected in order to avoid duplicates.

Algorithm 2: Finding k-faces Of A d-polyhedron

input : $P :=$ d-polyhedron
$I :=$ incidence matrix of P
$k :=$ k-face dimension
output: $L :=$ list of k-faces

```
1  L ← ∅;
2  C ← all (d − k) combinations of constraints of P;
3  foreach C_i ∈ C do
4  |   S ← I ∩ C_i;
5  |   n ← number of values equal to 1 in S;
6  |   if n ≥ k + 1 and S contains at least one vertex then
7  |   |   if No other already considered constraint saturates S then
8  |   |   |   L ← L ∪ C_i;
9  |   |   end
10 |   end
11 end
```

In chamber decomposition, this algorithm is used to find the m-faces (m is the number of parameters) of a given parametric polyhedron.

For example, $\mathcal{D}'_0$ is a 3-polyhedron. It is the combined polyhedron for parametric polyhedron $\mathcal{D}_0(p)$.

When applied to $\mathcal{D}'_0$, this algorithm finds the 1-faces represented as thick purple lines in Figure 4.5b.

4.3.2 Finding The Vertices And Validity Domains

Once the m-faces are found, the corresponding vertices and validity domains can be computed as detailed in Algorithm 3 using the bases of the m-faces. The base $\mathcal{B}$ is the lineality space of an m-face $\mathcal{F}$ and is built by taking $m+1$ independent columns from the rays of $\mathcal{F}$.

The parametric vertex is the matrix product $\mathcal{B}_D \cdot \mathcal{B}_P^{-1}$ (where $\mathcal{B}_P$ and $\mathcal{B}_D$ are restrictions of the k-face's base $\mathcal{B}$ to the parameters or the iterators) while the validity domain is the projection of $\mathcal{F}$ on the parameter space.

Algorithm 3: Finding Vertices And Validity Domains

input : $\mathcal{F} :=$ list of m-faces of a given polyhedron
output: $\mathcal{L} :=$ list of (vertice, validity domain) couples

$L \leftarrow \varnothing$;
foreach $\mathcal{F}_i \in \mathcal{F}$ **do**
  $\mathcal{B} \leftarrow$ base of $\mathcal{F}_i$;
  $\mathcal{B}_P \leftarrow m+1$ last lines of $\mathcal{B}$;
  $\mathcal{B}_D \leftarrow n$ first lines and last line of $\mathcal{B}$;
  if $\mathcal{B}_P$ *can be inverted* **then**
    $\mathcal{T} \leftarrow \mathcal{B}_D \cdot \mathcal{B}_P^{-1}$;
    $\mathcal{V} \leftarrow$ projection of $\mathcal{F}$ on parameter space;
    $\mathcal{L} \leftarrow \mathcal{L} \cup (\mathcal{T}, \mathcal{V})$;
  end
end

4.3.3 Disjunct Validity Domains

The validity domains found with the method described in Section 4.3.2 may overlap while it is desirable to find a partition $\pi(\mathcal{D}')$ of the combined polyhedron. One way to construct this partition from the list of validity domains is to progressively remove overlapping parts from previous domains as a new domain is considered.

Algorithm 4 details this method. The algorithm iterates on the domains from the input list. Each domain is intersected with the domains in the output list. If the domains overlap, they are separated in up to three parts: the intersection and the remainders from both domains. The separated parts are

kept in the output list while the remainder from the currently considered domain will be checked against the next domains in the output list. Once the whole output list has been checked, the remainder of the currently considered domain, if it is not empty, is added to the output list.

Algorithm 4: Disjunct Validity Domains

input : $L :=$ list of (domain, parametric vertice) couples
$m :=$ parameter number
output: $P :=$ list of (domain, list of parametric vertices) couples

```
1  P ← ∅;
2  foreach (D, V) ∈ L do
3      P' ← ∅;
4      remainder ← D;
5      foreach (D', V') ∈ P do
6          if dim(D ∩ D') = m then
7              intersection ← (D' ∩ remainder, V' ∪ {V});
8              difference ← (D'\ remainder, V');
9              remainder ← (D\ remainder, {V});
10             P' ← P' ∪ {intersection} ∪ {difference};
11             if dim(remainder) = m then
12                 P' = P' ∪ {remainder};
13             end
14         else
15             P' ← P' ∪ {D};
16         end
17     end
18     P ← P';
19 end
```

Table 4.4 presents the vertices and validity domains found when applying Algorithm 3 and Algorithm 4 to $\mathcal{D}$.

4.4 Splitting polyhedra

Our goal is to reduce control overhead in generated code by splitting polyhedra at Step 3 of the QRW Algorithm 1 using the chamber decomposition of parametric polyhedra. These parametric polyhedra are versions of the parametric polyhedra to be scanned where, at a given dimension, outer iterators

Table 4.4 – 1-**faces And Corresponding Bases, Parameterized Vertices And Validity Domains For** $\mathcal{D}(p)$**.**

1-face	base	vertex	domain
$\mathcal{F}_1 = \left\{ (i,j,p) \in \mathbb{Q}^3 \middle\| \begin{array}{l} i = 0 \\ j = p \\ 0 \le p \le 10 \end{array} \right\}$	$\mathcal{B}_1 = \begin{pmatrix} 0 & 0 \\ 0 & 10 \\ 0 & 10 \\ 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 0 \\ p \end{pmatrix}$	$0 \le p \le 10$
$\mathcal{F}_2 = \left\{ (i,j,p) \in \mathbb{Q}^3 \middle\| \begin{array}{l} i = 0 \\ j = 10 \\ p \ge 10 \end{array} \right\}$	$\mathcal{B}_2 = \begin{pmatrix} 0 & 0 \\ 0 & 10 \\ 1 & 10 \\ 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 0 \\ 10 \end{pmatrix}$	$p \ge 10$
$\mathcal{F}_3 = \left\{ (i,j,p) \in \mathbb{Q}^3 \middle\| \begin{array}{l} i = p - 10 \\ j = 10 \\ p \ge 10 \end{array} \right\}$	$\mathcal{B}_3 = \begin{pmatrix} 1 & 0 \\ 0 & 10 \\ 1 & 10 \\ 0 & 1 \end{pmatrix}$	$\begin{pmatrix} p - 10 \\ 10 \end{pmatrix}$	$p \ge 10$
$\mathcal{F}_4 = \left\{ (i,j,p) \in \mathbb{Q}^3 \middle\| \begin{array}{l} i = 0 \\ j = 0 \\ p \ge 0 \end{array} \right\}$	$\mathcal{B}_4 = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 0 \\ 0 \end{pmatrix}$	$p \ge 0$
$\mathcal{F}_5 = \left\{ (i,j,p) \in \mathbb{Q}^3 \middle\| \begin{array}{l} i = p \\ j = 0 \\ p \ge 0 \end{array} \right\}$	$\mathcal{B}_5 = \begin{pmatrix} 1 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}$	$\begin{pmatrix} p \\ 0 \end{pmatrix}$	$p \ge 0$

are considered as parameters. In order to find out how to split a domain, we need to compute its subdomains in which each inner scanned polyhedron is *regular*: its bounds are strictly (not piecewise-) affine functions of the parameters and outer loop indices. At a given loop level, the parametric polyhedron is built from the domain to be scanned, by taking as parameters the program parameters, the outer loop indices and the current loop index. This parametric polyhedron is exactly the inner polyhedron that will be scanned by the inner loops. By computing its chambers, we get the set of domains in the parameters space in which the inner polyhedron is *regular*.

Algorithm 5 explains how to split polyhedra at a given loop level d. This should occur after Step 3 of Algorithm 1. First, the input polyhedron and the context are modified to consider the d outer iterators as parameters. The chamber decomposition of the modified polyhedron is then computed. The transformation of iterators into parameters is then reverted on the chambers so that they are defined over the same space as the original polyhedron.

Algorithm 5: Polyhedra Splitting

input : $\mathcal{D} :=$ input polyhedron
$\mathcal{K} :=$ context
$d :=$ current level
output: $L :=$ list of polyhedra

Step 1 $\mathcal{D}' \leftarrow$ new polyhedron from $\mathcal{D}$ where outer iterators up to dimension d are considered as parameters;
$\mathcal{K}' \leftarrow$ extended context with the constraints from $\mathcal{K}$ and the constraints on the outer iterators from $\mathcal{D}$;

Step 2 $C \leftarrow$ compute the chambers on $\mathcal{D}'$ and $\mathcal{K}'$;

Step 3 $R \leftarrow$ revert temporary parameters as iterators in all $C_i \in C$;

Step 4 $P \leftarrow$ intersect the reverted chambers with the original domain;

Finally, the original polyhedron is split: it is intersected with the reverted chambers.

EXAMPLE 15. Let us compute the chambers for polyhedron $\mathcal{P}_1$ from Section 4.1. The polyhedron $\mathcal{P}_{c1}$ corresponds to Step 1 of the algorithm (iterator $c1$ is now considered as a parameter):

$$\mathcal{P}_{c1}(N, M, c1) = \left\{ (c2) \in \mathbb{Z} \;\middle|\; \begin{array}{l} 0 \leq c2 \\ c1 - N \leq c2 \\ c2 \leq c1 \\ c2 \leq M \end{array} \right\} \tag{4.7}$$

The chambers C_1 to C_3 are then computed at Step 2:

$$\mathcal{C}_1 = \{(N, M, c1) \in \mathbb{Q}^3 |\, 0 \leq c1 \leq M\} \tag{4.8}$$

$$\mathcal{C}_2 = \{(N, M, c1) \in \mathbb{Q}^3 |\, M + 1 \leq c1 \leq N - 1\} \tag{4.9}$$

$$\mathcal{C}_3 = \{(N, M, c1) \in \mathbb{Q}^3 |\, N \leq c1 \leq N + M\} \tag{4.10}$$

Step 3 converts the chambers into the appropriate polyhedra R'_1 to R'_3:

$$\mathcal{R}'_1(N,M) = \{(c1,c2) \in \mathbb{Q}^2 |\, 0 \leq c1 \leq M\} \quad (4.11)$$

$$\mathcal{R}'_2(N,M) = \{(c1,c2) \in \mathbb{Q}^2 |\, M+1 \leq c1 \leq N-1\} \quad (4.12)$$

$$\mathcal{R}'_3(N,M) = \{(c1,c2) \in \mathbb{Q}^2 |\, N \leq c1 \leq N+M\} \quad (4.13)$$

Finally, they are intersected with the original polyhedron $\mathcal{P}_0$ at Step 4 to get the polyhedra $\mathcal{P}_{1,0}$ to $\mathcal{P}_{1,2}$ as shown in Figure 4.6:

$$\mathcal{P}_{1,0}(N,M) = \left\{ (c1,c2) \in \mathbb{Z}^2 \,\middle|\, \begin{array}{l} 0 \leq c1 \leq M \\ 0 \leq c2 \leq c1 \end{array} \right\} \quad (4.14)$$

$$\mathcal{P}_{1,1}(N,M) = \left\{ (c1,c2) \in \mathbb{Z}^2 \,\middle|\, \begin{array}{l} M+1 \leq c1 \leq N-1 \\ 0 \leq c2 \leq M \end{array} \right\} \quad (4.15)$$

$$\mathcal{P}_{1,2}(N,M) = \left\{ (c1,c2) \in \mathbb{Z}^2 \,\middle|\, \begin{array}{l} N \leq c1 \leq N+M \\ c1-N \leq c2 \leq M \end{array} \right\} \quad (4.16)$$

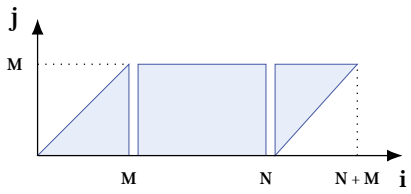


Figure 4.6 – **2D Representation of Polyhedra $\mathcal{P}_{1,0}$, $\mathcal{P}_{1,1}$ and $\mathcal{P}_{1,2}$**

EXAMPLE 16. Let us consider a more complex example: imagine that domain $\mathcal{P}_1$ is tiled in 8x8 square tiles. We will show that our technique is also effective to split loops taking out partial tiles from the full ones. Our algorithm would split those loops in the following way. The original iteration domain is:

$$\mathcal{D}(N,M) = \left\{ \begin{pmatrix} t1 \\ t2 \\ c1 \\ c2 \end{pmatrix} \in \mathbb{Z}^4 \,\middle|\, \begin{array}{l} 0 \leq c1 \leq N+M, \\ 0 \leq c2 \leq c1, \\ c1-N \leq c2 \leq M, \\ 8*t1 \leq c1 < 8*t1+8, \\ 8*t2 \leq c2 < 8*t2+8 \end{array} \right\}$$

The first splitting on index $t1$ is done by computing the parametric polyhedron $\mathcal{D}'(N, M, t1) \in \mathbb{Z}^3$ having the same constraints: $t1$ was taken out of the dimensions of this polyhedron and considered as a parameter. There are six chambers in this parametric polyhedron, cutting the domain into six pieces, depicted in Figure 4.7 as horizontal arrowed lines below the c1 axis.

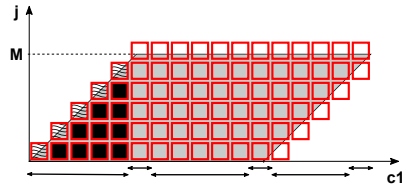


Figure 4.7 – Tiled Version of Polyhedron $\mathcal{P}_1$

The first chamber is: $\mathcal{C}_1 = \{(t1) \in \mathbb{Z} \mid 0 \leq t1, 8 * t1 \leq M - 7\}$. Recursing in the algorithm, we generate the $t2$ dimension: the following parametric polyhedron will be scanned in this first chamber:

$$\mathcal{D}(N, M, t1, t2) = \left\{ \begin{pmatrix} c1 \\ c2 \end{pmatrix} \in \mathbb{Z}^2 \;\middle|\; \begin{array}{l} \text{(same constraints, and)} \\ 0 \leq t1, \\ 8 * t1 \leq M - 7 \end{array} \right\}$$

Computing the chambers of this parametric polyhedron, we obtain as expected two of them, the lower part containing complete tiles (darkened in the figure) and the partial tiles on the upper edge of this triangle (dashed in the figure):

$$\begin{aligned} \mathcal{C}_1 &= \{(N, M, t1, t2) \in \mathbb{Z}^3 \mid 0 \leq t2, 8 * t2 \leq 8 * t1 - 7\} \\ \mathcal{C}_2 &= \{(N, M, t1, t2) \in \mathbb{Z}^3 \mid 8 * t1 - 7 < 8 * t2 \leq t1\} \end{aligned}$$

The same process repeats for the five other chambers of the $t1$ dimension. At the end of the algorithm, we obtain 30 different tile shapes being scanned by 30 different loops bodies. The original CLooG version (see Listing 4.1) is 9 lines long, hence it increases in the split version to 205 lines but removes `min()` or `max()` calculations (see Appendix A.1.1 in the Appendix).

```
for (t1=0;t1<=floord(N+M,8);t1++) {
  for (t2=max(0,ceild(8*t1-N-7,8));
  ↪  t2<=min(floord(M,8),t1);t2++) {
    for (c1=8*t1; c1<=min(min(N+M,8*t1+7),8*t2+N+7);c1++) {
      for (c2=max(8*t2,c1-N); c2<=min(min(M,c1),8*t2+7);c2++) {
        S1((c1-c2),c2);
      }
    }
  }
}
```

Listing 4.1 – Extended QRW Generated Code

4.5 Experimental Results

We chose to implement our splitting algorithm in CLooG/PolyLib 0.18.4, since the PolyLib embeds the required chambers decomposition computation. The benchmarks were compiled with `gcc` version 6.2.1, `icc` version 17.0.0 and `clang` version 3.8.1, in two flavors: without optimizations (`-O0`) and with optimizations (`-O3 -march =native`). The target architecture is a 2.40GHz Intel Xeon E5-2620v3, running linux 4.8. All the programs were run sequentially on a single core. All measurements on CLooG's test suite are an average of 3 runs using the `time` command. The PolyBench measurements were made using the provided script.

4.5.1 CLooG's test suite

We ran a first set of benchmarks on the many test files that are shipped with the CLooG distribution. Notice that most of them do *not* use tiling. The code generated by our version differs from the mainline CLooG for 46 of them. As can be seen in Figure 4.8, the resulting code size is usually bigger, with a geometric mean (*geomean*) growth of 2.5x, and a maximum of 64.5x for a corner test case for CLooG. The slowdown of the code generation itself varies from 1x to 100x, with a geomean of 4.5x.

The execution time measurements were performed on those programs, by incrementing a volatile counter in all statements. A summary of the res-

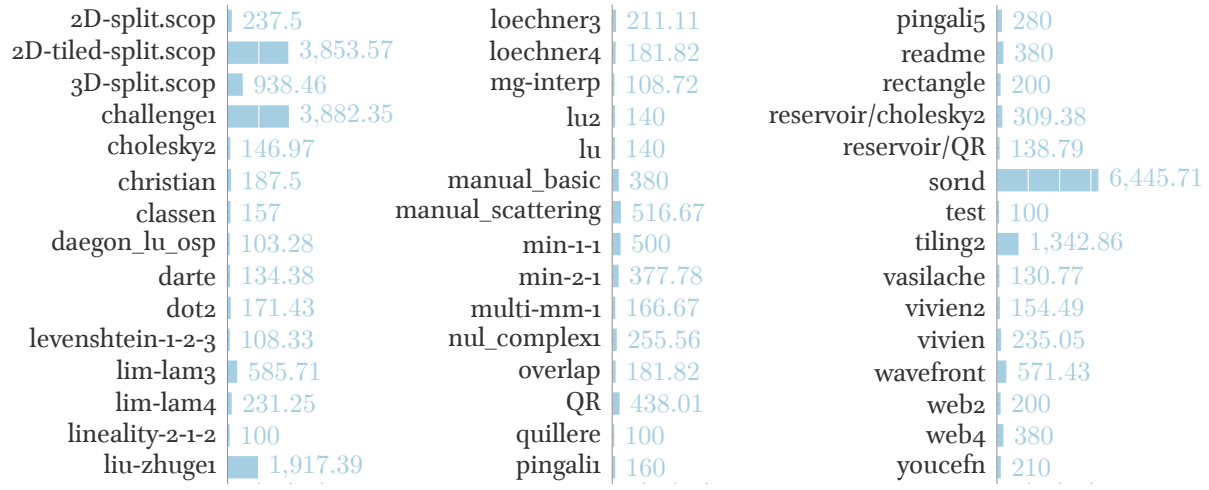


Figure 4.8: **Output Code Size Ratios In % On CLooG's Test Suite**

ults is presented in Table 4.5. The first three columns represent the benchmarks that were improved by our splitting method; the three following ones, the benchmarks that were degraded. In each of these groups of three columns, we show the percentage of improved versus degraded benchmarks, the maximum speedup/slowdown, and the geomean speedup. The last column shows the total geomean speedup.

Table 4.5 – **Overview Of The Speedups For The CLooG Examples.**

	speedup > 1			speedup < 1			geomean
compiler	%	max	geomean	%	min	geomean	speedup
`gcc -O0`	59	1.34	1.07	41	0.81	0.96	1.02
`icc -O0`	66.7	1.21	1.03	33.3	0.97	0.99	1.01
`clang -O0`	82.5	1.96	1.19	17.5	0.5	0.9	1.12
`gcc -O3`	51.2	1.07	1.03	48.8	0.9	0.97	1.00
`icc -O3`	61.5	1.11	1.02	38.5	0.89	0.98	1.00
`clang -O3`	63.9	1.17	1.09	36.1	0.66	0.95	1.03

Full results with options `-O0` and `-O3 -march=native` are given in Figure 4.9 and Figure 4.10.

Overall, there are more improvements than degradations. The geomean of speedups for the improved benchmarks is about 10 percent greater than

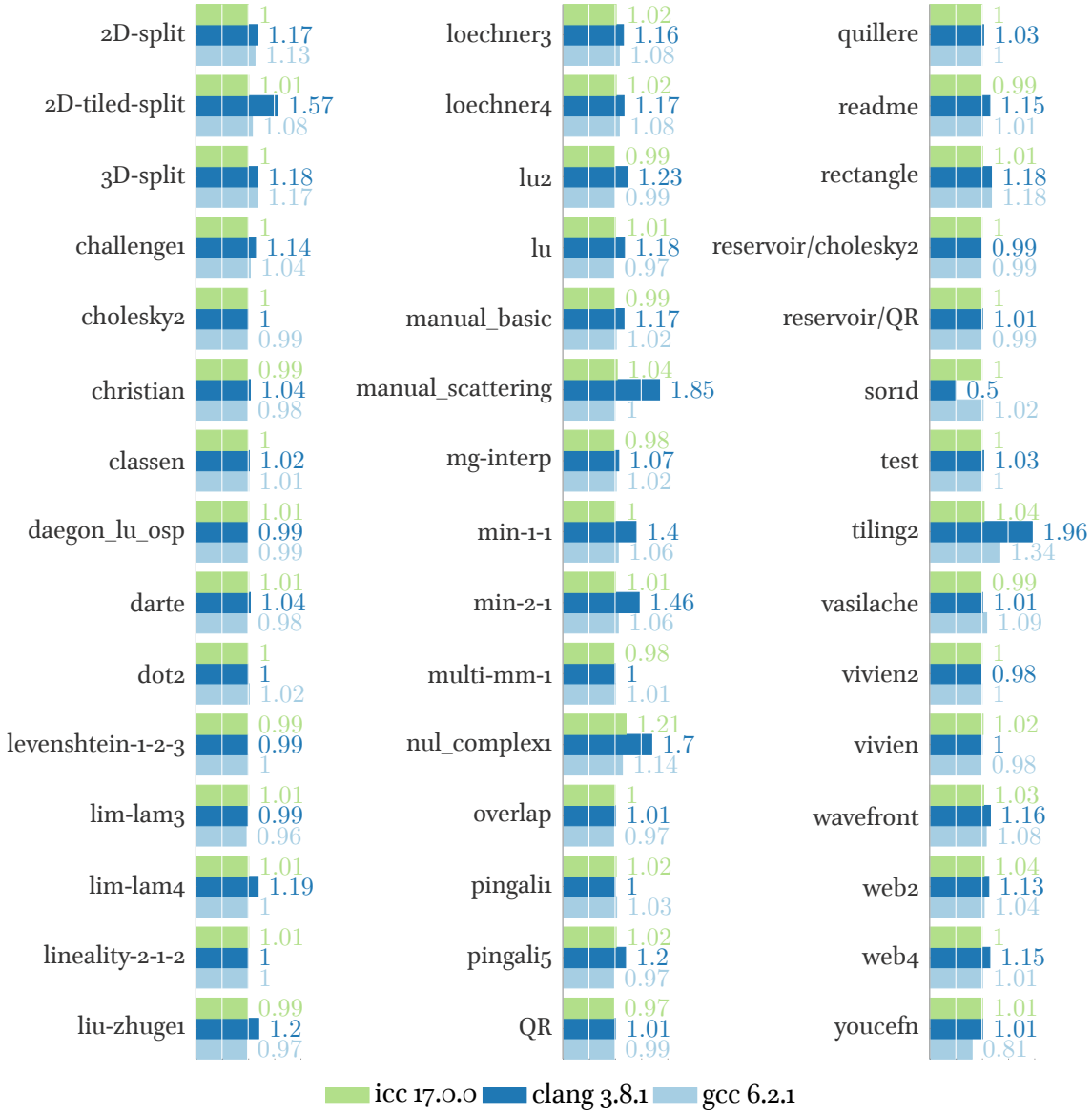


Figure 4.9: **Speedups On CLooG's Test Suite With `-O0`.**

the one of degraded benchmarks. The difference is fading when the compiler optimizations become more aggressive: with `-O0` all the control is transferred in the generated code, while it is simplified by the compiler with `-O3`. However, extremal values show that the choice of polyhedral representation is important: from half to twice the performance, depending on the way the polyhedra are split. Performance and code size growth do not seem to correlate.

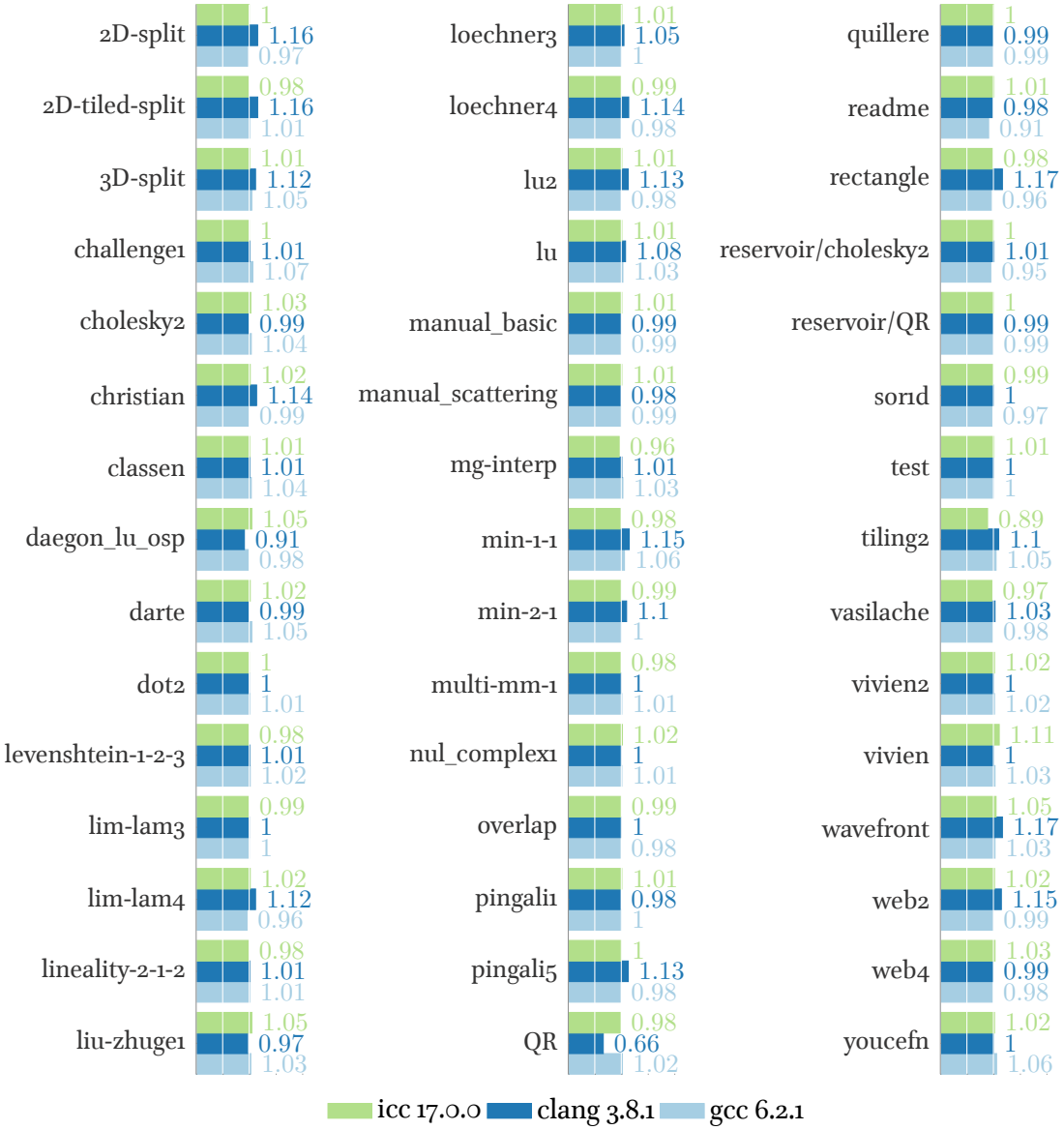


Figure 4.10: **Speedups On CLooG's Test Suite With `-O3`.**

4.5.2 PolyBench

We ran a test on the linear-algebra and stencil examples from PolyBench 4.1 test suite. We used PLuto 0.11.4 with tiling (`--tile`), and plugged our version of CLooG to generate the output code. The slowdown of the code generation by CLooG varies from 2x to 50x. The synthetic results are given in Table 4.6 while complete results are shown in Figure 4.11 and Figure 4.12.

Table 4.6 – Overview of the speedups for the Polybench

	speedup > 1			speedup < 1			geomean
compiler	%	max	geomean	%	min	geomean	speedup
gcc -O0	80.00	1.13	1.08	20.00	0.93	0.96	1.05
icc -O0	70.00	1.47	1.11	30.00	0.96	0.98	1.07
clang -O0	77.78	1.37	1.15	22.22	0.99	0.99	1.09
gcc -O3	72.73	1.98	1.17	27.27	0.94	0.96	1.11
icc -O3	18.18	1.07	1.04	81.82	0.30	0.79	0.84
clang -O3	66.67	1.25	1.17	33.33	0.88	0.92	1.05

Those results confirm our previous observations. With gcc and clang, the split versions are faster on average, up to 1.98x, with an average of 1.11x on all these benchmarks for gcc `-O3`. However, it seems that icc `-O3` performs much better on the original versions than on the transformed ones: the geomean speedup is 0.84x in this case. This is probably due to icc's linear algebra and stencil pattern recognition algorithms performing very aggressive optimizations that were ineffective for our transformed versions.

4.6 Perspectives on Splitting Fitness Decision

We proposed a new method to perform a smart splitting to disjoint polyhedra, to generate code without tests or complex bounds in loops, with the objective of reducing control overhead and facilitating vectorization for the compiler. Experimental results show that although it may significantly improve the quality of the generated code, loop splitting is not always beneficial. It elongates code generation times and the resulting code may not necessarily run faster than the unsplit version.

We have investigated multiple traits of both input and output loop nests: we tried to determine how the number of dimensions, the number of constraints, the iteration domain sizes, the number of splits, the output code size, the nesting depth of the splits impact the resulting performance. We were not able to find a deterministic relation between these characteristics and the final performance. Because control overhead is most significant in

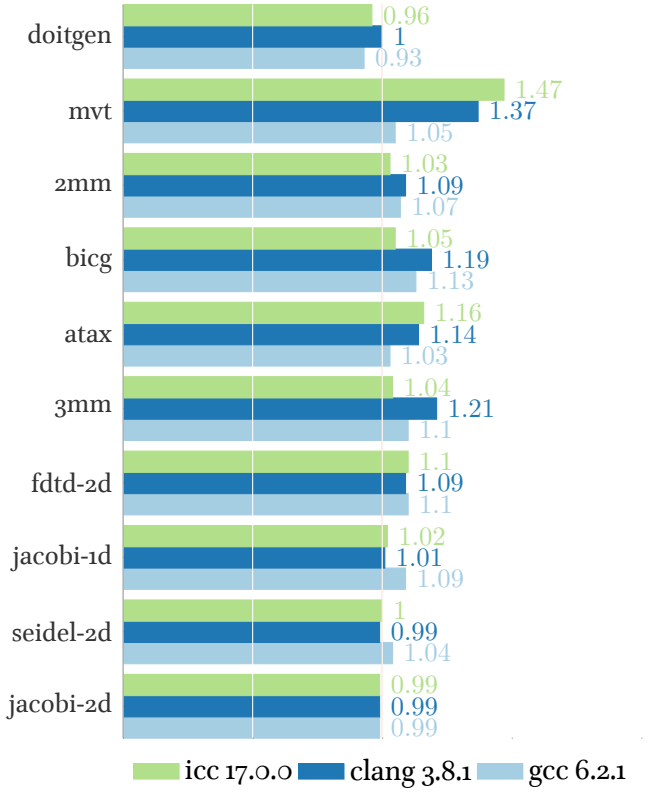


Figure 4.11 – Speedups on the PolyBench Test Suite with `-O0`

innermost loops, another educated guess is to start splitting a loop nest at the innermost depth and then progressively split loops on the outer dimensions until a threshold number of splits is reached. In the current implementation, we manually limited the total number of split domains that are generated to an arbitrary fixed maximum.

The method described in Section 4.4 is general and attempts to split all polyhedra at all loop levels. The initial observation on the motivation example from Section 4.1 was that *minimum* and *maximum* calculations increased control overhead. A naive way to identify such upper and lower bounds is to inspect the constraints of a given iteration domain: if a bound on an iterator involves multiple constraints, the corresponding code may require a minimum or maximum calculation. For instance, in polyhedron $\mathcal{P}_0$, we can identify two constraints for the upper bound of iterator $c2$: $c2 \leq c1$ and $c2 \leq M$. It is actually possible to split $\mathcal{P}_0$ with as little information as these two constraints: subtracting these two constraints builds the new con-

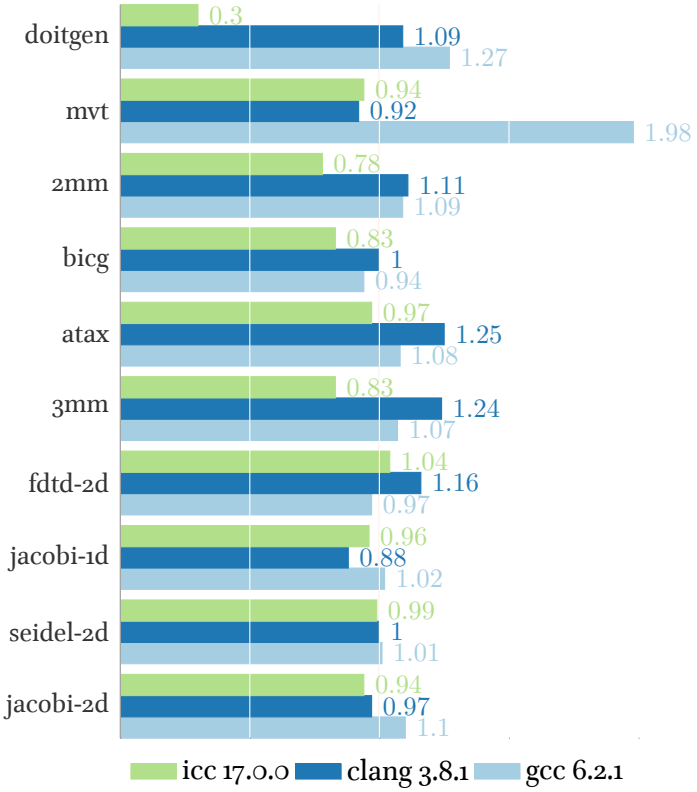


Figure 4.12 – Speedups on the PolyBench Test Suite with `-O3 -march=native`

straint $0 \leq c1 - M \Leftrightarrow c1 \geq M$ and inverting this constraint gives $c1 \leq M$. Adding either of these constraints to $\mathcal{P}_0$ creates two new polyhedra:

$$\mathcal{P}_{1-alt-0}(N, M) = \left\{ \begin{pmatrix} c1 \\ c2 \end{pmatrix} \in \mathbb{Q}^2 \middle| \begin{array}{l} 0 \leq c1, \\ c1 \leq N + M, \\ 0 \leq c2, \\ c1 - N \leq c2, \\ c2 \leq M, \\ c2 \leq c1 \\ c1 \geq M \end{array} \right\}$$

$$\mathcal{P}_{1-alt-1}(N,M) = \left\{ \begin{pmatrix} c1 \\ c2 \end{pmatrix} \in \mathbb{Q}^2 \;\middle|\; \begin{array}{l} 0 \leq c1, \\ c1 \leq N + M, \\ 0 \leq c2, \\ c1 - N \leq c2, \\ c2 \leq M, \\ c2 \leq c1 \\ c1 \leq M \end{array} \right\}$$

Code explosion is one of the caveats of our general splitting technique. Because the method exposed above is restricted, it could serve as a basis to a more conservative — albeit less general — technique which could perhaps limit the output size of the code. We attempted to mitigate the drawbacks of our splitting algorithm by trying to find how to limit its application domains. Future work could perhaps build upon this restricted method and gradually expand its use cases until a given threshold is reached.

4.7 Conclusion

Code generation tools heavily rely on polyhedral operations, usually provided by general-purpose polyhedral libraries that are not specifically designed for code generation. Unions of polyhedra have many possible representations and we have shown that the generated code can significantly vary depending on the way the polyhedra are modeled. Our initial intent was to reduce control overhead. These differences can definitely aid a compiler's subsequent optimization passes or, on the contrary, impair its capabilities. A polyhedral code generator can and should attempt to facilitate the work of a compiler. However, to what extent? This remains an open question.

Chapter 5

Reducing Synchronization Overhead

State-of-the-art polyhedral parallelizers generate code where loops identified as parallel are annotated so that each parallel loop is enclosed in a parallel region. This implies that threads may be created and destroyed at each parallel loop entry and exit. Synchronization barriers will also be required at loop exit before thread destruction. This synchronization overhead can be averted. Indeed, an expert parallel programmer would avoid creating/destroying parallel regions whenever possible and permit threads to run asynchronously for as long as possible. We will see in this chapter how to achieve this automatically.

We propose to improve the generated parallel code by taking advantage of several features of the OpenMP framework that are — so far — unused by polyhedral parallelizers in order to remove implicit barriers and thread restarts which are known to introduce significant overhead [20, 21]. Our method takes as input an already optimized SCoP and embeds the whole piece of code of interest into a single parallel region. This allows finer control over synchronizations as unnecessary implicit barriers may be lifted while explicit barriers may be placed where required. This is done automatically thanks to the polyhedral model dependence analysis. We will exploit the following OpenMP [79] constructs and clauses: `parallel`, `for`, `single`, `nowait`, and `barrier`.

Our proposal is introduced in Section 5.1 with an example to show that it can reduce the execution times of resulting parallel codes. Section 5.2 covers the general method for annotating an optimized code in order to create a unique parallel region. Barrier lifting or explicit placement within a parallel region is then explained in Section 5.3. Section 5.4 discusses the adjustments to be made to generate code for a unique parallel region where barriers have been lifted or explicitly placed. We show that our proposal improves the performance of generated code on the PolyBench benchmark suite in Section 5.5. Finally, Section 5.6 concludes this chapter.

5.1 Motivating Example

When used as an automatic parallelizer, the state-of-the-art polyhedral compiler Pluto [19] generates multi-threaded code where parallel loops are annotated with OpenMP directives. Each parallel loop is preceded by an OpenMP `omp parallel for` directive. This method yields satisfactory results on simple cases, *e.g.*, if there is one single outer parallel loop. However, superfluous synchronizations and overhead may be induced in more complex cases: threads are started and stopped (even if modern OpenMP implementations do not create and destroy thread teams for each parallel region) for each `parallel for` construct and there is an implicit synchronization barrier at the end of parallel loops.

Let us consider the main computation of the *atax* program from the PolyBench suite [83]. The original code is given in Listing 5.1. It is a linear algebra kernel that consists of a matrix transposition and a vector multiplication.

Listing 5.2 presents the parallelized and optimized code produced by Pluto version 0.11.4 (with option `--parallel`. Array `tmp` is first initialized in a parallel loops (lines 1–3) and its content is computed in another parallel loop (lines 4–9). Array `y` is initialized in a parallel loop (lines 10–12). Two nested loops then compute the content of array `y` (lines 13-19). Due to data-dependencies, Pluto skewed the loop nest to parallelize the inner loop (line 16).

We propose to reduce overhead by embedding this whole code into a single parallel region and removing unnecessary implicit synchronizations

```
1 for (i = 0; i < N; i++)
2   y[i] = 0;
3 for (i = 0; i < M; i++) {
4   tmp[i] = 0.0;
5   for (j = 0; j < N; j++)
6     tmp[i] += A[i][j] * x[j];
7   for (j = 0; j < N; j++)
8     y[j] += A[i][j] * tmp[i];
9 }
```

Listing 5.1 – **Atax: Original Code** A linear algebra kernel which computes $y = A^T(Ax)$.

as shown in Listing 5.3. The `omp parallel` directive encloses the whole kernel (lines 1–2 and 22). The threads are started and stopped once, at region entry and exit. The `schedule(static)` and `nowait` clauses are then used on the first two parallel loops. The dependencies between the first and second loop (lines 3–11) allow the safe use of the `nowait` clause on the first loop since the same threads will access the same array elements of array `tmp`. There are no dependencies from the first or the second loop to the third parallel loop (lines 3–14): the `nowait` clause can be used on the second parallel loop (line 7). How to ensure these clauses can be used and how to annotate loops is detailed in Section 5.3. On the other hand, the successive writes to array `y` prevent the asynchronous execution of the last two loop nests: the third parallel loop (line 12) is not annotated with the `nowait` clause. In the same vein, no `nowait` clause is added to the last parallel loop (line 117) because of successive iterations of the outer loop (line 16).

The speedups of those versions of the code, depending on the dataset size, are given in Table 5.1. The baseline (1x) is the execution time of the

Table 5.1 – **Performance On The *Atax* Benchmark**

Dataset	PLUTO version	Reduced synchronizations version
medium	0.46x	1.31x
large	1.24x	1.94x
extralarge	1.27x	2.19x

```
#pragma omp parallel for
for (int t2 = 0; t2 <= M - 1; t2++)
  tmp[t2] = 0.0;
if (N > = 1) {
  #pragma omp parallel for
  for (int t2 = 0; t2 <= M - 1; t2++)
    for (int t3 = 0; t3 <= N - 1; t3++)
      tmp[t2] += A[t2][t3] * x[t3]
}
#pragma omp parallel for
for (int t2 = 0; t2 <= N - 1; t2++)
  y[t2] = 0;
if ((M >= 1) && (N >= 1)) {
  for (int t2 = 0; t2 <= N + M - 2; t2++) {
    #pragma omp parallel for
    for (int t3 = max(0, t2 - M + 1); t3 <= min(t2, N -
    ↪  1); t3++)
      y[t3] += A[t2 - t3][t3] * tmp[t2 - t3];
  }
}
```

Listing 5.2 – **Atax: Parallelized And Optimized Code Using PLUTO Version 0.11.4** Each parallel loop is annotated with an `omp parallel for` directive. In the last loop nest, the inner loop is parallelized.

sequential version. The experimental platform is a 6-core Intel processor, as described later in Section 5.5. Our version is on average 1.97x faster than the PLUTO version. This is mainly due to the OpenMP thread re-creation that is avoided inside the last parallel loop of this code.

Next section introduces the general method for parallel region generation.

5.2 Parallel Region Generation

Current code generation methods in polyhedral compilers create a distinct OpenMP parallel region for each parallel loop. The consequences are that:

1. the start of a loop coincides with the parallel region start: a thread team is created at the start of the loop,

```
1  #pragma omp parallel
2  {
3    #pragma omp for schedule(static) nowait
4    for (int t2 = 0; t2 <= M - 1; t2++)
5      tmp[t2] = 0.0;
6    if (N >= 1) {
7      #pragma omp for schedule(static) nowait
8      for (int t2 = 0; t2 <= M - 1; t2++)
9        for (int t3 = 0; t3 <= N - 1; t3++)
10         tmp[t2] += A[t2][t3] * x[t3];
11   }
12   #pragma omp for
13   for (int t2 = 0; t2 <= N-1; t2++)
14     y[t2] = 0;
15   if ((M >= 1) && (N >= 1)) {
16     for (int t2 = 0; t2 <= N + M - 2; t2++) {
17       #pragma omp for
18       for (int t3 = max(0, t2 - M + 1); t3 <= min(t2,
         ↪  N-1); t3++)
19         y[t3] += A[t2 - t3][t3] * tmp[t2 - t3];
20     }
21   }
22 }
```

Listing 5.3 – **Atax: Further Optimized Code Where Some Synchronizations Have Been Lifted.** As opposed to Listing 5.2, a unique parallel region encompasses the kernel. The loops are annotated with `omp for` directives (instead of `omp parallel for`) and `nowait` directives have been added to the first two parallel loops (lines 3 and 7).

2. the thread team must *always* synchronize at the end of a loop because it is also the end of the surrounding parallel region,

3. the end of a loop coincides with the parallel region end: the thread team may be destroyed at the end of the loop.

Modern OpenMP implementations attempt to be clever in regard to this matter: thread teams may be simply started or stopped instead of being completely created or destroyed. Nonetheless, consequent control overhead still remains [20, 21].

We propose to refine current code generation methods by generating a

single parallel region, when it is profitable: it would be futile for a kernel that contains a single outer parallel loop. However, gathering multiple parallel loops or nested inner parallel loops within a unique outer parallel region will eliminate the overhead of multiple parallel regions. Thus, our technique should be applied only on loop nests containing inner parallel loops or when factorizing multiple parallel loops into a single parallel region is possible.

The general algorithm for parallel region and annotation is described in Alg. 6. The input of the algorithm is an automatically parallelized (using the polyhedral techniques) Abstract Syntax Tree (AST), encoded as a list of AST nodes. The `annotate` function is a function that annotates a given AST node with the text passed as a second argument.

Step 1 tackles both the annotation of parallel loops and sequential portions of the code. Sequential parts must also be annotated because the input AST may contain statements which were not identified as potentially parallel. Within a parallel region, all threads will execute the statements encountered by the execution flow unless instructed otherwise. Thus, statements not identified as parallel must be protected with the OpenMP `single` construct to ensure they are executed only once. Parallel loops are annotated with the `for` construct (instead of `parallel for`). The algorithm recurses in inner nodes of the AST unless the current node is a parallel loop: in this case the annotation process can stop at level of the parallel loop. Because Step 1 indiscriminately may annotate more nodes than necessary with the `single` construct (for example if a block contains multiple non parallel statements: the statements will be annotated and the block will also be annotated). Step 2 removes nested inner `single` constructs in order to keep only the outermost annotations.

Another reason to embed kernels in wide parallel regions is to get rid of superfluous synchronization barriers. This is performed at Step 3 which is further explained in Section 5.3.

Step 4 merges consecutive nodes annotated with the `single` construct that require a barrier: multiple barriers are not created in that case. On the other hand, successive `single` constructs without barriers should not be merged: different threads may then execute the distinct `single` constructs in parallel. This step also justifies why the `single` construct is preferred over

Algorithm 6: Parallel Regions Annotation. This algorithm takes as input an *Abstract Syntax Tree* and annotates its nodes with work-sharing constructs or synchronization clauses. Where appropriate, `nowait` annotations indicate that no thread synchronization is required between two given nodes. `annotate_nowait` is described in Algorithm 7

input : $T :=$ Abstract Syntax Tree
output: $T' :=$ Annotated AST

Function `region_annotation(`T`)`

```
Step 1  foreach node n ∈ root(T) do
            if n is a loop identified as parallel then
                annotate(n, "#pragma omp for");
            else if n has a body then
                region_annotation(body(n));
            end
            if neither n nor body(n) contain a parallel loop then
                annotate(n, "#pragma omp single");
            end
        end
Step 2  remove inner nested #pragma omp single constructs;
Step 3  annotate_nowait(T, T, ∅);
Step 4  /* Clean up successive synchronized #pragma omp single
           constructs                                          */
        T ← merge consecutive #pragma omp single constructs that
         require a barrier;
Step 5  /* Create the parallel region                         */
        T' ← new block with T as body;
        annotate(T', "#pragma omp parallel");
        return T';
    end
```

the `master` construct[1].

Eventually[2], the unique parallel region is created with the `omp parallel directive`. As it should surround the whole kernel, Step 5 creates a new AST

[1] Programmers often confuse both constructs as they instruct that the enclosed code must be executed by only one thread. However the `master` construct behaves very differently as it enforces the execution by the *master* (or first) thread and has no implicit barrier.

[2] Note that Step 5 could also be placed at the very beginning of Algorithm 6.

node which surrounds the input AST and annotates the new node with the directive.

Listing 5.4 presents a short comparison of the output of this algorithm with the output of a classic automatic parallelizer. Listing 5.4a would be the output of a classic automatic polyhedral parallelizer whereas Listing 5.4b corresponds to the output of our algorithm for the same input AST. First, Algorithm 6 places the whole kernel in a single parallel region (lines 1–2 and 17). Assuming both the loops over statements S1 and S2 require a barrier, the two loops are grouped into a unique `single` construct. Assuming the dependencies allow it, the first parallel loop would be annotated with the `nowait` clause. Finally, the last parallel loop is left untouched.

```
1  for (int i=0; i<N; i++)
2    S1(i);

3  for (int i=0; i<N; i++)
4    S2(i);

5  #pragma omp parallel for
6  for (int i=0; i<N; i++)
7    S3(i);

8  for (int i=0; i<M; i++)
9    #pragma omp parallel for
10   for (int j=0; j<N; j++)
11     S4(i, j);
```

(a) **Classic Output.**

```
1  #pragma omp parallel
2  {
3    #pragma omp single
4    {
5      for (int i=0; i<N; i++)
6        S1(i);
7      for (int i=0; i<N; i++)
8        S2(i);
9    }

10   #pragma omp for nowait
11   for (int i=0; i<N; i++)
12     S3(i);

13   for (int i=0; i<M; i++)
14     #pragma omp for
15     for (int j=0; j<N; j++)
16       S4(i, j);
17 }
```

(b) **Our Output.**

Listing 5.4 – **Simple Comparison Example Of Algorithm 6.** Listing 5.4a would be generated after an automatic polyhedral parallelizer optimized some code. Two parallel loops are annotated with `omp parallel for` constructs. Listing 5.4b corresponds to the output of our algorithm for the same optimized AST.

5.3 Barrier Lifting

Enclosing code in a single parallel region raises the opportunity to alleviate synchronization overhead. Indeed, implicit barriers are placed at the end of OpenMP worksharing constructs. These barriers may be lifted using the `nowait` clause.

The OpenMP specification states that under certain circumstances, the `nowait` clause may be safely used on a `for` construct which precedes another `for` construct if the latter loop's statement instances depend only on the same logical iteration of the former loop. If such dependences exist, the specification imposes several restrictions:

1. the sizes of both *iteration domains* are equal,
2. the *chunk size* is either the same for both loops or not specified,
3. both loops are bound to the same parallel region,
4. none of the loops is associated with a `SIMD` construct.

Subsection 5.3.1 will explain how to check the dependences and how to ensure the conditions are met.

Safe use of the `nowait` clause in this fashion also requires the scheduling policy of both loops to be `static`. Enforcing this policy ensures that a given thread is assigned the same logical iterations (or the same *chunks*) for both loops.

Current known implementations of OpenMP default to the `static` policy. However, not only is it not required by the specification, it may be modified by surrounding code. Hence, in our case, the scheduling policy should be explicitly specified in the generated code. Note that, for readability, code examples in this chapter may omit the scheduling policy.

Special care must be paid to worksharing constructs enclosed in loops. If the worksharing construct of interest is the last of the outer loop, it may *precede* the next worksharing construct in the generated code as well as the first worksharing construct in the outer loop. In this case, determining whether the `nowait` clause can be used requires to analyze multiple dependencies.

The minimal requirement is the possibility to use the `nowait` clause between constructs within the loop. If the next worksharing construct is not compatible with the `nowait` clause, a `barrier` construct must be used right before the latter construct instead of right after the former loop. Listing 5.5 presents such an example: the `nowait` clause can safely be used for successive iterations of the loop on `i` for statement `S1` (lines 3-6) but the very last parallel loop over `S1` (`i = N - 1`) must be completed before the parallel loop over `S2` can be executed. A barrier is placed *after* the enclosing loop (line 7) as a synchronization to ensure the parallel loop on statement `S2` can be entered. Correct barrier placement is addressed in Subsection 5.3.2.

```
1  #pragma omp parallel
2  {
3    for (int i = 0; i < N; i++)
4      #pragma omp for nowait
5      for (int j = 0; j < M; j++)
6        S1: A[j] = A[j] + B[i];
7    #pragma omp barrier
8    #pragma omp for
9    for (int i = 0; i < M; i++)
10     for (int j = 0; j < i; j++)
11       S2: C[i] = C[i] * A[j];
12 }
```

Listing 5.5 – **SCoP Example Where The Use Of The `nowait` Clause Requires A Barrier.**

Subsection 5.3.1 explains how to ensure the `nowait` clause can be used. Subsection 5.3.2 details how to annotate an already optimized AST to encode parallel regions, single constructs and barrier lifting. Finally, Subsection 5.4 discusses the changes required at the code generation phase.

5.3.1 Determining the validity of the `nowait` clause

Implicit barriers are placed at the end of worksharing constructs. Omitting these barriers may be done under certain circumstances.

The most simple situation is the case of two or more successive worksharing constructs without any dependencies. It is valid to annotate such

worksharing constructs (except the last one) with the `nowait` clause. Annotating the last construct still requires further analysis of the remainder of the code.

The OpenMP specification details four conditions for the safe use of the safe use of `nowait` (as long as the schedule type is set to `static`) when there are dependencies between subsequent `for` constructs.

One of the requirements is for the *iteration domain* sizes of the loops to be equal. This can often be easily verified by comparing the loop bounds and strides of the loops. If needed, it can be further ascertained using the polyhedral model: determining that the sizes of two iteration domains coincide is possible by comparing their Ehrhart polynomials [24].

The fulfillment of the other conditions (identical *chunk sizes*, the loops binding to the same parallel region and the absence of `SIMD` constructs) is trivial: our code generation algorithm aims to produce a single parallel region and thus enforces by design that the worksharing constructs bind to the same parallel region. In the same vein, identical *chunk* sizes and the absence of `SIMD` constructs can be enforced by design during the code generation phase.

If the aforementioned conditions are met, compliant OpenMP implementations assign the same logical iterations to the same threads. Hence, the last step is to ensure that the only existing dependencies lie between identical logical iterations. This dependence analysis is presented hereunder.

Let S and T be two statements such that T depends on S. Using the notation introduced in Definition 16, the dependencies $\delta_{S,T}$ between S and T are expressed as follows:

$$\delta_{S,T}(\vec{p}) \;=\; \left\{ \vec{u} \;\rightarrow\; \vec{v} \;\middle|\; R_{S,T} \begin{pmatrix} \vec{u} \\ \vec{v} \\ \vec{p} \\ 1 \end{pmatrix} \;\geq\; \vec{0} \right\} \tag{5.1}$$

Assuming u_i is the parallel dimension for S and v_j is the parallel dimension for T, the polyhedron $\mathcal{L}_{S,T}(\vec{p})$ that links the same logical iterations of

S and T can be expressed as:

$$\mathcal{L}_{S,T}(\vec{p}) \;=\; \{\vec{u} \;\rightarrow\; \vec{v} \mid u_i = v_j\} \tag{5.2}$$

The dependencies in different logical iterations of S and T can be described using the two previous equations as:

$$\delta_{S,T}(\vec{p}) \setminus \mathcal{L}_{S,T}(\vec{p}) \tag{5.3}$$

Let $\delta'_{S,T}$ and $\mathcal{L}'_{S,T}$ be the projections of these two equations onto the space combined space of dimensions $u_i + v_i$. Then, $\delta'_{S,T} \setminus \mathcal{L}_{S,T}$ is a simpler way to express all dependencies of some form over dimensions u_i and v_j between S and T. It immediately follows that the necessary condition for the validity of the `nowait` clause on statement S is:

$$\delta'_{S,T}(\vec{p}) \setminus \mathcal{L}'_{S,T}(\vec{(p)}) \equiv \emptyset \tag{5.4}$$

EXAMPLE 17. For instance, consider the two statements $S1$ and $S2$ in Listing 5.6.

```
1  #pragma omp parallel
2  {
3    #pragma omp for nowait
4    for (int i = 0; i < N; ++i)
5      for (int j = 0; j < M; ++j)
6        A[i][j] = S1(B[i][j]);

7    #pragma omp for
8    for (int a = 0; a < N; ++a)
9      for (int b = 0; b < M; ++b)
10       C[a] = S2(A[a][b]);
11 }
```

Listing 5.6 – **Example Of A Loop Nest Compatible With The `nowait` Clause.**

Statement `S2` depends on the same logical iteration of `S1` (assuming the loops are normalized):

$$\delta_{S1,S2} = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \rightarrow \begin{pmatrix} a \\ b \end{pmatrix} \;\middle|\; \begin{matrix} i = a, \\ j = b \end{matrix} \right\} \tag{5.5}$$

$$\mathcal{L}_{S1,S2} = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \rightarrow \begin{pmatrix} a \\ b \end{pmatrix} \;\middle|\; i = a \right\} \tag{5.6}$$

The dependencies $\delta_{S1,S2}$ link the same logical iterations of $S1$ and $S2$. Here are the projections of the two polyhedra:

$$\delta'_{S1,S2} = \left\{ \begin{pmatrix} i \end{pmatrix} \rightarrow \begin{pmatrix} a \end{pmatrix} \;\middle|\; i = a \right\} \tag{5.7}$$

$$\mathcal{L}'_{S1,S2} = \left\{ \begin{pmatrix} i \end{pmatrix} \rightarrow \begin{pmatrix} a \end{pmatrix} \;\middle|\; i = a \right\} \tag{5.8}$$

Thus, $\delta'_{S1,S2}(\vec{p}) \setminus \mathcal{L}'_{S1,S2}(\vec{p})$ is empty and the first parallel loop (line 3) can be annotated with the `nowait` clause.

EXAMPLE 18. The dependencies between statements $S3$ and $S4$ in Listing 5.7 prevent the safe use of the `nowait` clause on the first parallel loop:

$$\delta_{S3,S4} = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \rightarrow \begin{pmatrix} k \end{pmatrix} \;\middle|\; \begin{array}{l} i \geq k-1, \\ i \leq k+1 \end{array} \right\} \tag{5.9}$$

$$\mathcal{L}_{S3,S4} = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \rightarrow \begin{pmatrix} k \end{pmatrix} \;\middle|\; i = a \right\} \tag{5.10}$$

$$\delta'_{S3,S4} = \left\{ \begin{pmatrix} i \end{pmatrix} \rightarrow \begin{pmatrix} k \end{pmatrix} \;\middle|\; \begin{array}{l} i \geq k-1, \\ i \leq k+1 \end{array} \right\} \tag{5.11}$$

$$\mathcal{L}'_{S3,S4} = \left\{ \begin{pmatrix} i \end{pmatrix} \rightarrow \begin{pmatrix} k \end{pmatrix} \;\middle|\; i = k \right\} \tag{5.12}$$

Indeed, a given statement instance of the second parallel loop (lines 7-9) depends on three distinct statement instances of the first loop. Here, $\delta_{S3,S4}(\vec{p}) \setminus \mathcal{L}_{S3,S4}(\vec{p})$ is nonempty:

$$\delta_{S3,S4}(\vec{p}) \setminus \mathcal{L}_{S3,S4}(\vec{p}) = \{(i) \rightarrow (k) \mid i = k-1\} \cup \{(i) \rightarrow (k) \mid i = k+1\} \tag{5.13}$$

and thus the first parallel loop can not be annotated with the `nowait` clause.

```
1  #pragma omp parallel
2  {
3    #pragma omp for
4    for (int i = 0; i < N; ++i)
5      for (int j = 0; j < M; ++j)
6        S3(A[i], B[i][j]);

7    #pragma omp for
8    for (int k = 0; k < N; ++k)
9      S4(C[k], A[k-1], A[k], A[k+1]);
10 }
```

Listing 5.7 – **`nowait` Incompatible Code**

5.3.2 `nowait` Annotation

Once an optimized AST has been annotated in order to generate a single parallel region, implicit barriers may be lifted and explicit barriers may be placed. Step 3 of Algorithm 6 calls `annotate_nowait` which is described in Algorithm 7.

The algorithm will annotate each node of the AST to indicate whether a barrier is needed. The key idea is to progressively build *nowait-groups*. Two successive nodes may belong to the same nowait group if no barrier is required between the two nodes. If a barrier is required between two nodes, the first node is annotated and a new *nowait-group* is built (lines 6-7, 19-20 and 25-26).

At first, the algorithm assumes a given node does not require a barrier (line 2). If the node is a worksharing construct, it is checked against the current *nowait-group*. Either the node is added to the current group (line 9) or a new group is started (line 7). If the current node is not a work sharing construct, the algorithm recurses into its body, if any (line 12-13). If the node is a loop, the algorithm also checks the dependencies between its first and last *nowait-group* (if there is more than one): because of the loop, the last *nowait-group* is also a predecessor of the first *nowait-group*. The node is checked against the current *nowait-group* which is adjusted accordingly (lines 23-36). Finally, the algorithm proceeds with the next node (line 37).

Algorithm 7: `nowait` Annotation. This recursive algorithm annotates nodes where barriers are required. The *nowait-group* is expanded or replaced before recursing on inner or following nodes.

```
   input : n := Current AST node
           R := Root of the AST
           G := Current nowait group
   output: n := Annotated AST node
1  Function annotate_nowait(n, R, G)
2      n.nowait ← true;
3      if n is a worksharing construct then
4          if check_nowait_safety(G, n) then
5              G ← G ∪ {n};
6          else
7              pred ← last_predecessor(R, n);
8              pred.nowait ← false;
9              G ← {n};
10         end
11     else
12         b ← body(n);
13         b ← annotate_nowait(b, R, G);
14         last_node ← last(b);
15         first_group ← first_group(b);
16         if n is a loop then
17             last_group ← last_group(b);
18             if ¬ check_nowait_safety(last_group, first_group) then
19                 last_node.nowait ← false;
20                 G ← ∅;
21             end
22         end
23         if ¬ check_nowait_safety(G, first_group) then
24             pred ← last_predecessor(R, n);
25             pred.nowait ← false;
26             G ← ∅;
27         end
28         if last_node.nowait then
29             G ← ∅;
30         else if first_group ≠ last_group then
31             G ← last_group;
32         else
33             G ← G ∪ {n};
34         end
35     end
36     n.next ← annotate_nowait(next(n), R, G);
37     return n;
38 end
```

`check_nowait_safety()` ensures the dependencies allow barrier lifting as per Subsection 5.3.1. `last_predecessor()` returns the preceding node.

`last()` returns the last node from a node list. `first_group()` and `last_group()` respectively return the first and last *nowait-group* from a node list.

5.4 Pretty Printing

The pretty printing phase is akin to current techniques, with a few minute modifications. First, the whole SCoP must be enclosed in a block annotated with the `#pragma omp parallel` construct. Parallel loops will be annotated with `#pragma omp for` instead of `#pragma omp parallel for`. Any sequential block or loop must be annotated with `#pragma omp single` constructs. If a given AST node is marked as requiring a barrier: the `#pragma omp barrier` directive must be printed right after the generated code for this node. If this node corresponds to a worksharing construct, no additional code is needed as worksharing constructs have implicit barriers. On the other hand, if the AST node does not require a barrier, no additional code is needed unless the node corresponds to a worksharing construct: worksharing constructs must be annotated with the `nowait` clause. Moreover, the `schedule(static)` clause should be added if the worksharing construct is a `for` loop,

5.5 Experimental Results

We evaluated our approach with benchmarks taken from the PolyBench [83] suite. We used Pluto version 0.11.4 on all benchmarks of the suite with two sets of options: the first one with automatic parallelization (`--parallel`), and the second one with parallelization and tiling (`--tile --parallel`). We applied our method on a selection of 34 benchmarks out of 50 (2×25). Those selected benchmarks were chosen because they contained at least one internal parallel loop. 14 out the 34 benchmarks are tiled versions. Table 5.2 describes the main characteristics of each of these benchmarks: the number of main loop nests that they contain, the number of parallel loops embedded into outer sequential loops, the number of single regions, and the number of nowait clauses that are introduced by our method.

Table 5.2 – **Benchmarks Main Characteristics**

benchmark	#main loop nests	#inner paral. loops	#single	#nowait
adi	2	2	0	0
adi-tile	2	2	0	0
atax	4	1	0	2
bicg	4	1	0	2
cholesky	2	2	2	1
cholesky-tile	2	2	0	0
correlation	9	2	0	1
covariance	7	1	0	2
doitgen	3	3	0	1
doitgen-tile	3	3	0	2
fdtd-2d	6	4	3	2
fdtd-2d-tile	1	1	0	0
floyd-warshall	1	1	0	0
floyd-warshall-tile	1	1	0	0
gemver	3	1	0	0
gramschmidt	8	3	1	1
gramschmidt-tile	5	2	1	1
heat-3d	11	4	7	5
heat-3d-tile	1	1	0	0
jacobi-1d	5	4	7	5
jacobi-1d-tile	1	1	0	0
jacobi-2d	10	4	7	4
jacobi-2d-tile	1	1	0	0
lu	2	1	1	0
lu-tile	2	1	0	0
nussinov	2	1	0	0
nussinov-tile	1	1	0	0
reg_detect	16	3	4	2
reg_detect-tile	1	1	0	0
seidel-2d	1	1	0	0
seidel-2d-tile	1	1	0	0
trisolv	2	1	1	1
trisolv-tile	2	1	0	0
trmm	2	1	0	0

We conducted our experiments on three platforms:

1. An Intel Xeon E5-2620v3 @ 2.40GHz (6 cores, 12 threads), running the Linux 4.11.5 kernel. Intel Turbo Boost and Hyperthreading were dynamically disabled during the execution of the benchmarks. In order to further reduce the variance of the measurements, Linux FIFO scheduling was enabled via the PolyBench's macro `POLYBENCH_LINUX_FIFO_SCHEDULER`. The compiler is `gcc 7.1.1` using options `-fopenmp`, `-O3` and `-march=native`.

2. The second platform is a dual socket Intel Xeon E5-2650v3 @ 2.30GHz (2*10 cores, 40 total threads), running Linux 4.4.0. The compiler is `gcc 5.4.0`, using options `-O3 -march=native -fopenmp`. No particular environment variable was set on this platform to get stable measurements.

3. The last platform is the same computer, but using this time the `icc` compiler version `17.0.0` and options `-O3 -march=native -qopenmp`. We had to put the environment variable `OMP_NUM_THREADS` to 20 in order not to use hyperthreading to get more stable measurements. Still, the variance was much higher on this configuration than on the previous ones, so the results on this platform are less reliable: the variance exceeds 5% in about half of the measurements that we made.

The environment variable `OMP_PROC_BIND` was set to `true` on all platforms. The PolyBench scripts that we used perform all time measurements as the average of 3 median measurements out of 5 runs. Notice that we report some speedups that are larger than the number of available cores; this is not a surprise as PLUTO is not only a parallelizer but also a data locality optimizer and vectorizer.

Figure 5.1 presents the acceleration of our version compared to the PLUTO version on the 6-cores first platform for the small, medium, and large datasets of the PolyBench suite. We can notice from this figure that our method improves many of these benchmarks: the acceleration is most often greater than 1x. The geometric mean acceleration for all datasets is given on the bottom line. On *smaller* datasets, the benefit of our method is often *greater*: in

many cases the ratio between threads creation and synchronizations time towards computation time is higher when computing small datasets. The overall mean acceleration on all dataset sizes available in PolyBench (including mini and extralarge, not shown in the figure) is 1.36x.

However, we noticed that in some of these benchmarks, Pluto did not improve the performance over the sequential version of the code. In order to support our conclusions, we checked that our method improves both the efficient parallelized codes and the inefficient ones.

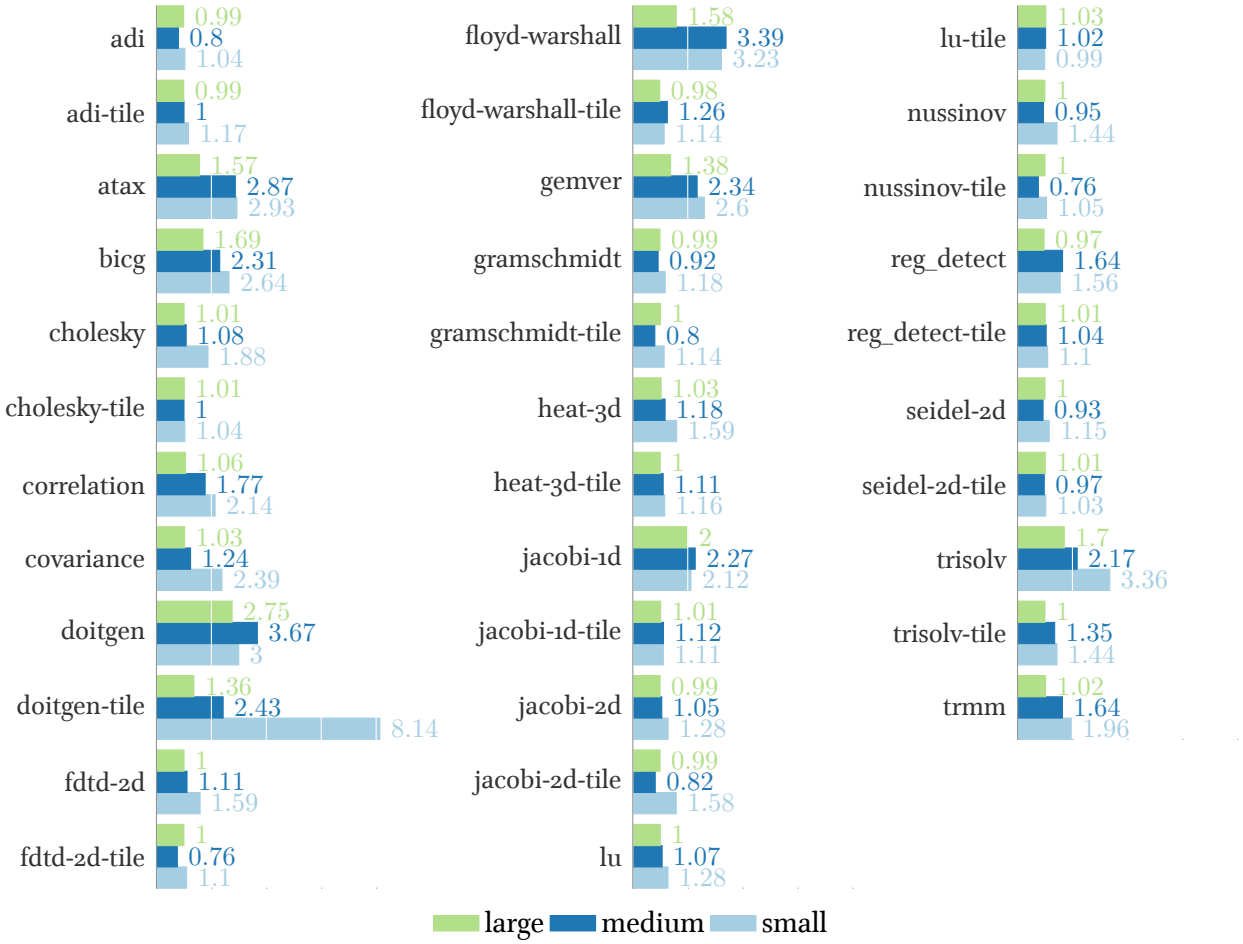


Figure 5.1: **Speedup Over Pluto, Platform 1.**

Figure 5.2 presents both the acceleration of the PLUTO version and the one of our version, over the sequential version on the standard dataset. The geometric mean of our version speedup over PLUTO, when PLUTO performs worse than 1x is 1.95x, and when PLUTO performs better than 1x it is 1.09x. So our method improves more the poor performing PLUTO codes, which is not a surprise: the synchronization over computation time ratio is usually higher on those codes. Nevertheless, we checked that our method improves the performance of most of the PLUTO parallel codes, whether PLUTO performs well or not.

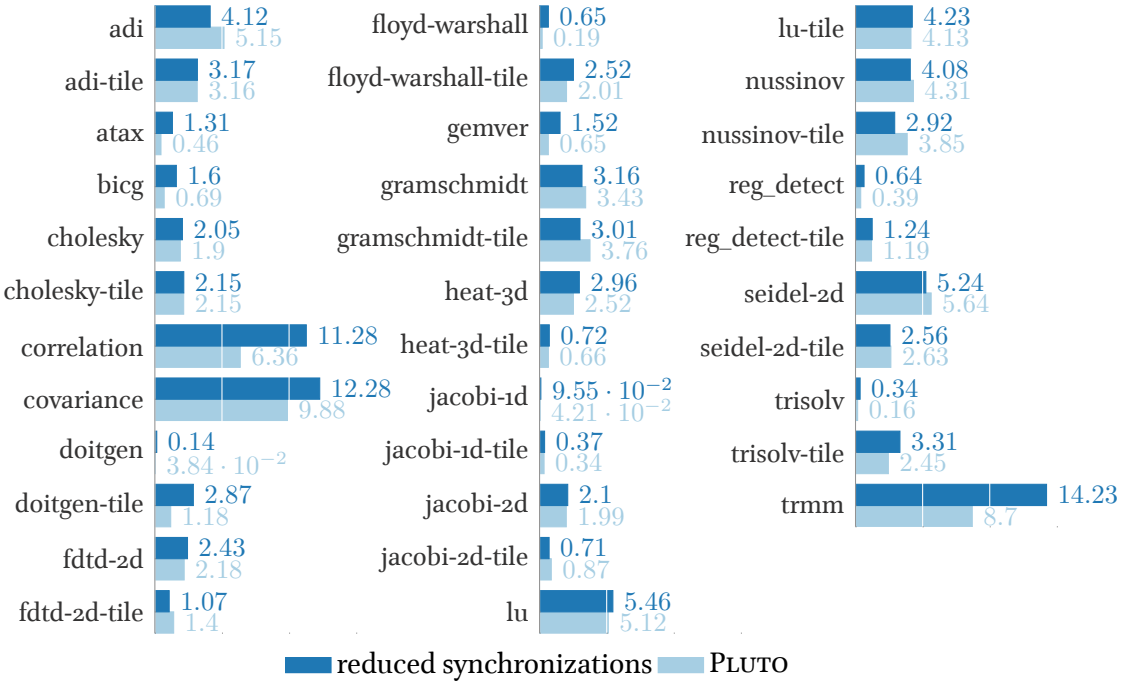


Figure 5.2: **Speedup Over The Sequential Version (Platform 1, Medium).**

We also ran those benchmarks on 40 threads in the second configuration (2x 10-cores hyperthreaded), to get the results presented in Fig. 5.3. The geometric mean of those accelerations is 1.52x on the large dataset, and 1.39x on the extralarge dataset.

Finally, we ran the benchmarks on the 20 threads third configuration, using the icc compiler and the Intel OpenMP runtime. Each benchmark acceleration of our version over the PLUTO version is given in Fig. 5.4 for the large and extralarge datasets. Some benchmarks, marked with *, are not reported

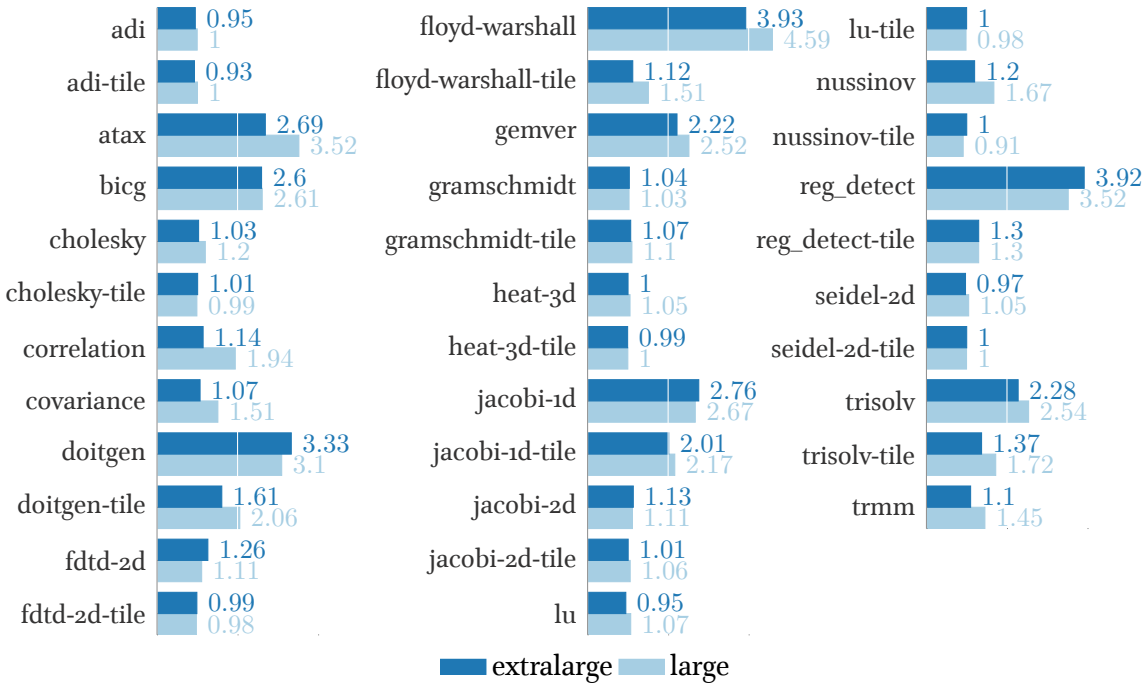


Figure 5.3: **Speedup Over Pluto, Platform 2.**

since there is a numerical divergence between the different versions, probably due to the icc vectorizer: the vector floating point unit does not have the same precision as the main one. The average acceleration of our version is respectively 1.14x and 1.11x for the large and extralarge datasets. The overall acceleration is a bit lower than the previous ones, most probably due to the more efficient OpenMP runtime. But those measurements are less reliable, as said before, since the variance in time measurements often exceeds the PolyBench default limit (5%).

5.6 Conclusion

In this chapter, we explained how to generate wide *parallel regions* rather than separate *parallel loops*. Our approach brings many advantages to most high-level optimizing compilers that solely rely on parallel `for` constructs. First, the overhead of computation threads start/stop is minimized. Second, it allows to remove unnecessary and costly synchronizations. Lastly, it may increase data locality between loops at thread level. Our method is not com-

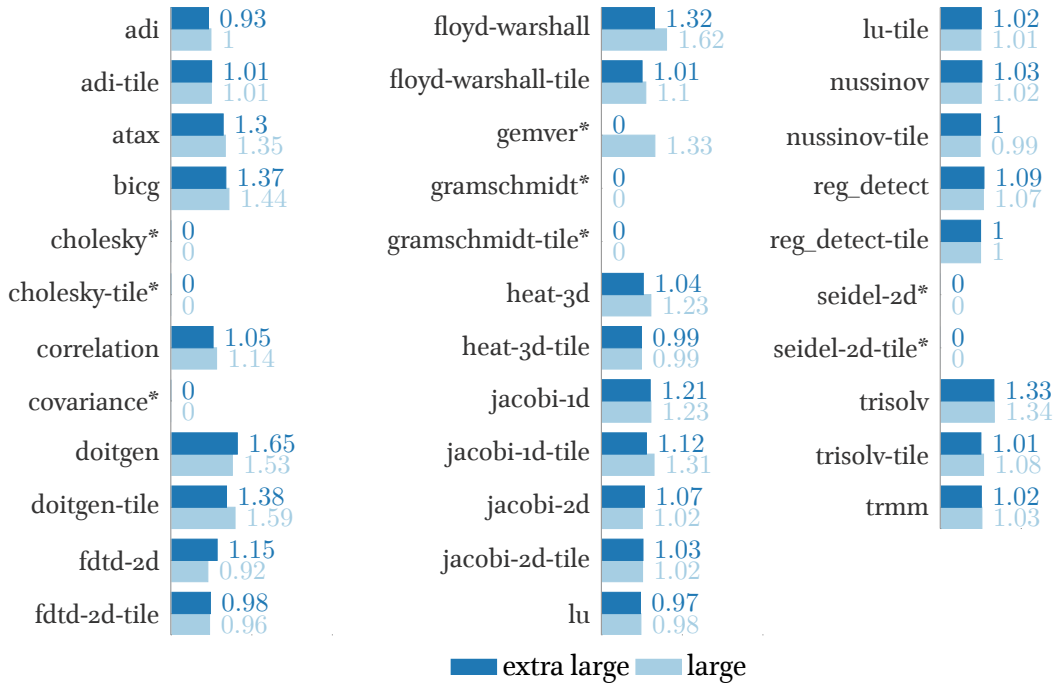


Figure 5.4: **Speedup Over PLUTO, Platform 3.**

peting but is *complementary* to existing parallelization frameworks: its input is an already optimized code and its output is an even more efficient optimization. It exploits the polyhedral representation of programs and a barrier lifting phase to factorize parallel loops into wider and deeper loop regions where superfluous synchronization barriers have been removed. We conducted a wide experimental study showing that our approach is nearly always beneficial and brings a significant gain over the state-of-the-art PLUTO compiler, from 1.14 to 1.63 speedup in average, depending on the dataset size. High-level polyhedral compilers should now always target parallel regions rather than collections of independent parallel loops.

Chapter 6

Exploiting Pipelined Multithreading

6.1 Introduction

Pipelines are essential constructions for exploiting parallelism at any level, from processor hardware to high-level software. This chapter explains how we can automatically generate a pipelined parallel code in a multihtreading environment such as OpenMP.

There has been extensive research on pipelines mainly focusing on hardware generation, low level implementation [80, 90] or VHDL generation [29, 4, 71, 114] for instance, to target field-programmable devices. In most of these works the input programs are already in pipelined tasks format, and their objective is to map the tasks efficiently on the available hardware. The two main addressed difficulties are to characterize stream types and stream sizes between tasks and to allocate the streams and the tasks on the fixed-size hardware.

However, automatically extracting tasks amenable to a pipeline execution from general sequential programs and implementing the pipeline itself remains a challenge.

We address the automatic generation of high-level pipelined multithreading codes. The two main issues that we address are:

1. how to automatically identify pipelines within a polyhedral compiler

2. how to generate pipelined code using the OpenMP high level multithreading API

The concept of pipelined multithreading is introduced in Section 6.2 on a short example. Section 6.3 describes the method for pipelined code generation in a polyhedral compiler while Section 6.4 refines the technique with explicit synchronizations. Experimental results are given in Section 6.5. Future work is discussed in Section 6.6 and Section 6.7 concludes this chapter.

6.2 Motivation

Listing 6.1 shows an example of an obviously pipelineable — yet ignored by current polyhedral optimizers — SCoP. This program is a succession of loops with loop-carried dependencies. Hence, an automatic polyhedral parallelizer will leave this code untouched because no *parallel* loop can be identified as exhibited by the red arrows in the visual representation of the dependencies in Figure 6.1. However, it can also be observed that dependencies between loops do not require a given loop to be completely executed before another loop can start. This is where pipelined parallelism can be extracted: the red and green arrows in the dependency graph show that an iteration of a given loop only has two direct predecessors: the preceding iteration within the loop and the same logical iteration in the preceding loop. Our goal is to transform Listing 6.1 into Listing 6.2 to take advantage of pipelined multithreading with OpenMP without violating the execution order imposed by the dependencies.

Roughly, the following steps can transform the original code from Listing 6.1 into the pipelined multithreaded parallel code from Listing 6.2. First, a traditional polyhedral scheduling is applied — for example, using the high-level compiler PLUTO. Then, strongly connected components are identified in the flow graph thanks to the precision of the polyhedral dependence analysis and can be used to detect potential pipelines. The corresponding code can be generated by enclosing the pipeline stages in a parallel region and

```
1 for (int i = 1; i < N; ++i)
2   S1: A[i] = f(A[i], A[i-1]);
3 for (int i = 1; i < N; ++i)
4   S2: B[i] = f(A[i], B[i-1]);
5 /* ... */
6 for (int i = 1; i < N; ++i)
7   S6: F[i] = f(E[i], F[i-1]);
```

Listing 6.1 – **Original Pipelineable Example.** Loop-carried dependences prevent polyhedral automatic parallelizers from identifying parallelism.

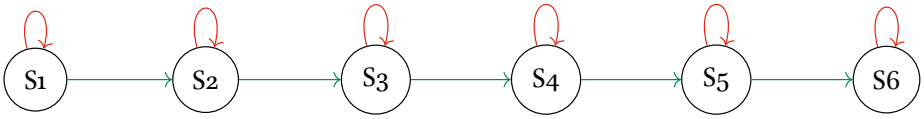


Figure 6.1 – **Visual Representation Of The Dependency Graph For The Code From Listing 6.1.** Red arrows represent loop carried dependencies. These dependencies prevent a polyhedral optimizer from identifying any loop as parallel. Green arrows symbolize dependencies between loops.

using the OpenMP clauses `ordered` and `nowait` to allow pipelined execution while preserving the required order. Loop bodies must be enclosed in a `ordered` construct: it is necessary to specify what part (the whole loop body in our case) of a `ordered` loop is not parallel.

Using this method on the original code produces the code shown in Listing 6.2. For each loop, each thread will execute a block of successive iterations. The `static` scheduling policy (not shown in the code excerpt) ensures that each thread will always be assigned the same range of iterations. For instance, the first thread will be assigned iterations 1 to N/number of threads for each loop[8]. The `nowait` clause specifies that a thread can proceed right away with the next loop while the `ordered` clause ensures that iteration blocks within a loop are executed in the correct order. Figure 6.2 further shows the execution flows for Listing 6.1 and Listing 6.2: the sequential code will completely execute a given loop (following the red arrows) before entering the next loop (progressing from one line to the other). In the parallel code, each thread will be assigned a column and will follow the green arrows. In the parallel execution, the red arrows represent the synchronizations that enforce

[8] Note that the actual range is unspecified, what matters is that this range will be the same for all loops

```
#pragma omp parallel
{
  #pragma omp for ordered nowait
  for (int i = 1; i < N; ++i)
    #pragma omp ordered
    S1: A[i] = f(A[i], A[i-1]);
  #pragma omp for ordered nowait
  for (int i = 1; i < N; ++i)
    #pragma omp ordered
    S2: B[i] = f(A[i], B[i-1]);
  /* ... */
  #pragma omp for ordered nowait
  for (int i = 1; i < N; ++i)
    #pragma omp ordered
    S6: F[i] = f(E[i], F[i-1]);
}
```

Listing 6.2 – **Pipelined OpenMP Target Program.** The use of `ordered` ensures correct iterations ordering within a loop while `nowait` allows early entry into the following loop.

relative ordering of the iterations of a given loop. The number of columns is lower or equal to the number of threads.

This process enables the parallelization of a code which would remain sequential with state-of-the-art polyhedral automatic parallelization techniques. Next section details this method.

6.3 Pipelined Multithreading Generation

Pipelined multithreading interlaces sequential loops and thus targets sequential loops. Hence, our approach takes as input an already scheduled SCoP and targets loops that are still sequential. The steps are:

1. performing loop distribution on sequential loops,
2. annotating groups of consecutive sequential loops with the `ordered` and `nowait` clauses,
3. if necessary, performing loop fusion between parallel and ordered sequential loops

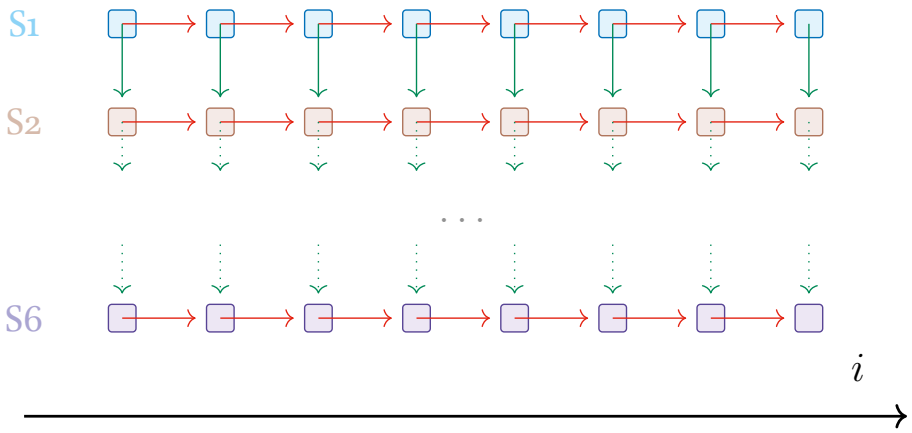


Figure 6.2 – **Visual representation of the execution order.** Loop carried dependencies are drawn in red. Dependencies between loops are represented in green.

With the code from Listing 6.1, the single thread will execute one loop after the other. In this example, each line will be completely executed before the next line starts.

With the code from Listing 6.2, each loop will be divided into *chunks* (in this example, 8 *chunks*). Each thread will execute the *chunks* on a column, ordered as indicated by the green arrows. The `nowait` clause allows a thread to immediately enter the next loop once its current *chunk* has been completed. The dependencies symbolized by red arrows are not violated thanks to the `ordered` clause: a *chunk* will not be executed unless its preceding *chunk* has already been executed.

Note that our method also requires the scheduling policy of `for` loop iterations to be set to static. Moreover, the choice of the chunk sizes should not be specified[9]: we want the number of chunks to be lower or equal to the number of threads. Specifying a wrong size may result in a number of chunks greater than the number of threads.

[9] As per the OpenMP specification [79], each thread will be assigned *at most* one chunk.

6.3.1 Sequential loop distribution

Interlacing loop iterations using the `ordered` and `nowait` requires separate loops. Input loops may contain multiple statements. To maximize the potential for interlacing, our strategy is to perform loop distribution on such loops. However, loop distribution can not be performed on each and every

```
for (int i = 2; i < N; ++i) {
  S01: a[i] = h[i-1] + R[i];
  S02: b[i] = a[i-1] + a[i];
  S03: c[i] = b[i-1] + b[i];
  S04: d[i] = c[i-1] + c[i];
  S05: e[i] = d[i-2] + d[i-1];
  S06: f[i] = e[i-2] + e[i-1];
  S07: g[i] = f[i] + X[i];
  S08: h[i] = g[i] + Y[i];
  S09: u[i] = v[i-1] + d[i];
  S10: v[i] = u[i] + Z[i];
}
```

(a) Van Dongen: original code

```
for (int i = 2; i < N; ++i) {
  S01: a[i] = h[i-1] + R[i];
  S02: b[i] = a[i-1] + a[i];
  S03: c[i] = b[i-1] + b[i];
  S04: d[i] = c[i-1] + c[i];
  S05: e[i] = d[i-2] + d[i-1];
  S06: f[i] = e[i-2] + e[i-1];
  S07: g[i] = f[i] + X[i];
  S08: h[i] = g[i] + Y[i];
}
for (int i = 2; i < N; ++i) {
  S09: u[i] = v[i - 1] + d[i];
  S10: v[i] = u[i] + Z[i];
}
```

(b) Loop distribution performed on Van Dongen

Listing 6.3 – **Van Dongen: Code After Safe Loop Distribution.** The original loop can be separated into two loops because the dependency graph contains two strongly connected components: statements `S01` to `S08` and statements `S09` and `S10`.

statement of such loops because it may change the scheduling in an illegal way. Loop distribution may only be applied to groups of statements when there are no backward loop-carried data dependencies, which we assess using polyhedral data dependence analysis.

Since we take as input a SCoP which should have already been optimized for locality, we do not want to reorder the statements from the input loop (apart from the reordering caused by the loop distribution): relative ordering within strongly connected components should remain unscathed. Determining how to perform loop distribution for our pipelining method amounts to computing strongly connected components from the dependence graph: each strongly connected component may belong to a standalone loop.

For example, in Listing 6.3, statements `S1` to `S8` are grouped in the same loop while statements `S9` and `S10` are placed in another loop. There is a loop-carried dependence between statements `S1` and `S8`: `S1`, `S8` and all statements in between belong to the same loop. In the same vein, `S9` and `S10` must be placed together but can be separated from statements `S1` to `S8` since there is

no loop-carried dependence between the two groups.

6.3.2 Relaxed conditions on the `nowait` clause

According to the OpenMP specification [79], the `nowait` clause may be safely used on a `for` construct which is followed by another `for` construct if the following conditions are met:

1. both iteration domain sizes are equal
2. *chunk sizes* are equal or not specified
3. both `for` loops are bound to the same parallel region
4. the loops are not associated with a SIMD construct

Chapter 5 explains how to generate code using the `nowait` clause for consecutive parallel `for` loops. This method can only be applied when the dependencies between loops link the same logical iterations. For example, assuming each block in Figure 6.2 represents exactly one iteration, only dependences represented as green arrows allow to safely use the `nowait` clause on consecutive parallel `for` loops.

Using the `ordered` clause allows us to relax the restrictions. Once a given thread executes an iteration i_n of a loop, all previous iterations i_m $(m < n)$ have been executed. In this case, the allowed dependencies between loops include more than dependencies between identical logical iterations. In other words, on Figure 6.2, using the `ordered` clause ensures that for any given block, all blocks in the intersection of previous lines and previous columns have been executed.

Hence, Equation (5.2) can be extended to:

$$\mathcal{L}_{S1,S2}(\vec{p}) \;=\; \{\vec{u} \;\rightarrow\; \vec{v} \mid u_i \leq v_j\} \tag{6.1}$$

The remainder of the analysis is unchanged as explained in Chapter 5: determining whether the nowait clause can be used to create multithreaded pipelines amounts to verifying that:

$$\delta'_{S1,S2} \setminus \mathcal{P}'_{S1,S2} \equiv \emptyset$$

6.3.3 Annotations

Annotating sequential loops

To annotate sequential loops with the `ordered` and `nowait` clauses, the method from **??** can be used with the following modifications:

1. all sequential loops are annotated with `#pragma omp for ordered`,
2. the bodies of `ordered` loops are enclosed in `#pragma omp ordered` regions,
3. the validity of the `nowait` clause is determined as described in Section 6.3.2.

The motivating example of Listing 6.2 gives an example of such a generated loop nest.

Extension to parallel loops and cleanup

Section 5.3.1 and Section 6.3.2 explain how to add the `nowait` clause on loops that precede another loop of the same kind: either a parallel loop preceding another parallel loop or a sequential loop preceding another sequential loop. Determining whether a parallel loop preceding a sequential loop — or vice versa — can be annotated with the `nowait` clause amounts to selecting the appropriate set of requirements: if the considered loop is a parallel loop, the stricter conditions on the use of the `nowait` clause apply, otherwise, the relaxed conditions are sufficient.

Finally, any `ordered` loop without a `nowait` clause that is neither preceded by another `ordered` loop nor precedes another `ordered` loop should be reverted to an annotation-free loop enclosed in a `#pragma omp single` region.

6.4 Explicitly synchronized pipelines

Section 6.3 introduced multithreaded pipelines with the `ordered` and `nowait` clauses. However, this method could be more flexible. For instance, the

chunk sizes should be chosen so that each thread is assigned, at most, one *chunk*. Otherwise, threads will wait for their next chunk (because of the `ordered` clause) and will not start the next loop right away. Moreover, in the general case, the `ordered` construct and clause may be used to order only some statements within a parallel loop. Thus, an OpenMP implementation may place synchronizations for each instance of an ordered construct. In our case, because the loops we target are completely sequential, we only need synchronizations at the start and at the end of a chunk.

An alternative to generating code with `#pragma omp ordered` and `nowait` annotations is to block loops over a given block size, fuse the resulting loops over the blocking dimension, distribute the iterations, with a `chunk_size` of 1, over this dimension and manually synchronize threads using OpenMP locks as shown in the example in Listing 6.4.

The pipelining then occurs as follows:

1. for each thread, each stage of the pipeline is associated with a lock (thus, `n` $\times$ `m` locks are required for `n` threads and m stages),

2. for each stage `i`, thread `t` attempts to own `lock[t%n][i]` at stage entry and releases `lock[(t+1)%n][i]` at stage exit (hence, each thread will wait for its predecessor to complete a given stage of the pipeline),

3. except for the first thread, all locks are locked at the beginning of the parallel region.

Construction

Additional code must be generated at the start and the end of the `parallel` region to allocate, initialize, destroy and free the locks. In particular, it is imperative to set all locks — apart from the locks assigned to the first thread — before the multithreaded pipeline starts.

Because the code generation already splits the loops into iteration chunks, the enclosing loop must be annotated with `schedule(static, 1)` to correctly distribute its iterations. We want threads to execute only one iteration of this loop at a time: otherwise pipelining opportunities may be lost as a given thread may execute two or more subsequent blocks of a given stage. We

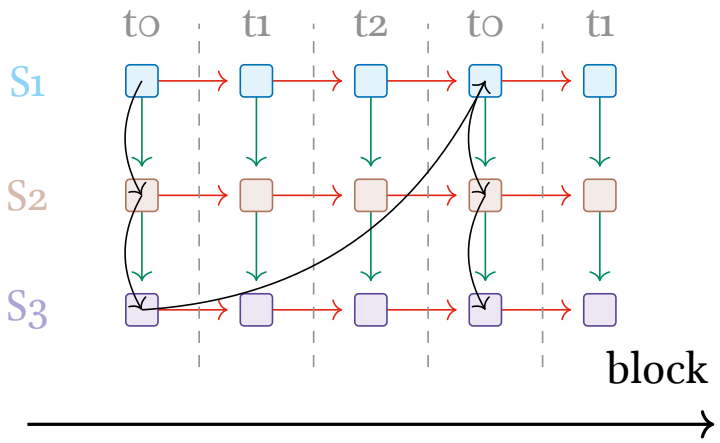


Figure 6.3 – **Stage-blocking execution order example for a 3-stage pipeline.** This graph is an example of a 3-stage pipeline akin to the code presented in Listing 6.4. Loop carried dependencies are drawn in red. Dependencies between loops are represented in green.
The `schedule(static, 1)` clause ensures each thread will execute one *block* at a time. The dependencies symbolized by green arrows are always respected because the iterations within a loop iteration are sequential. The locks ensure that the dependencies represented by red arrows are not violated.
The execution order for thread $t0$ is depicted by the black arrows. It will first execute all the stages of a given block. Thread $t1$ will progressively be allowed to start its own pipeline. At the end of its current pipeline, thread $t0$ will proceed with another *block* but it will only start once thread $t2$ as completed its first stage (in the same vein, $t2$ will not start its first stage until $t1$ has completed its own first stage).

also want the scheduling policy to be static so that iterations are distributed in a round robin fashion. Dynamic (or similar methods such as guided) distributions of iterations would break our locking mechanism where threads lock and unlock their preceding or succeeding threads in a round robin fashion.

In the example in Listing 6.4, The `block_size` value approaches the *chunk* size which would be chosen by most known OpenMP implementations for the code example shown in Listing 6.2.

```
omp_lock_t** locks;
#pragma omp parallel
{
  const size_t num_threads = (size_t)
  ↪  omp_get_num_threads();
  const size_t block_size = (N / num_threads) + 1;
  const size_t block_count = ((N + block_size - 1) /
  ↪  block_size);
  /* Omitted: code to allocate, initialize and set the
  ↪  locks. */
  #pragma omp for schedule(static, 1)
  for (size_t block = 0; block < block_count; ++block)
  ↪  {
    const size_t start = 1 + block * block_size;
    const size_t end = MIN(start + block_size, N);
    const size_t self = block % num_threads;
    const size_t next = (block + 1) % num_threads;

    omp_set_lock(&locks[self][0]);
    for (size_t i = start; i < end; ++i) {
      A[i] = f(A[i], A[i-1]);
    }
    omp_unset_lock(&locks[next][0]);
    /* Omitted: other stages of the pipeline */
    omp_set_lock(&locks[self][5]);
    for (size_t i = start; i < end; ++i) {
      F[i] = f(E[i], F[i-1]);
    }
    omp_unset_lock(&locks[next][5]);
  }
  /* Omitted: code to destroy and free locks */
}
```

Listing 6.4 – **Teaser: Explicitly synchronized multithreaded pipeline.** At the start, all locks are set except for thread 0. Thread 0 will proceed and progressively set its locks and unlock stages for thread 1. Hence, on its next iteration over the block loop, it thread 0 will wait for the last thread to complete its stages. In the same vein, each stage entry for a thread requires stage exit from the previous thread.

Comparison With Ordered Loops

We presented two multithreading pipelining methods. Both techniques require the scheduling policy to be set to `static`. However, explicitly synchronized pipelines are superior as they provide greater control over various aspects of the code.

First, explicit pipelines allow to precisely control the amount of synchronizations. Second, they permit a number of blocks greater than the number of threads: the `for` construct enforces that threads complete all their assigned chunks before they leave the loop. An example of this execution flow can be viewed in Figure 6.3. With the `ordered` and `nowait` clauses, thread $t0$ would have to execute both *chunks* for `S1` before it could start executing its *chunks* for `S2` (and it would have to wait for threads $t1$ and $t2$ to complete their first chunk). With explicitly synchronized pipelines, thread $t0$ can follow the black arrows in the figure.

The third reason why explicitly synchronized pipelines should be favored is that they allow further refinements which are discussed in Section 6.6: greater control over synchronizations permit *chunk* starts and endings (and thus sizes) to be adjusted on a stage basis.

6.5 Experimental Results and Discussion

Our experiments were conducted on an Intel Xeon E5-2620v3 @ 2.40GHz (6 cores, 12 threads) running Linux 5.5.4. The benchmarks were compiled with options `-O3 -march=native -fopenmp` using gcc 9.2.1, clang 9.0.1 and icc 19.1.0.166. Linux FIFO scheduling was enabled and process priority was set to 75 for measurement stability.

Benchmarks `van_dongen` (Listing 6.5) and `wdf` (Listing 6.6) come from Fimmel and Müller [39] who derive these examples from Van Dongen et al. [101] and Fettweis [38]. The `mix` benchmark (Listing 6.7) is a code example which contains both parallel loops and sequential loops. The last benchmark (Listing 6.8) is based on our teaser example where all statements execute a 1 nanosecond nanosleep and then return the sum of the two operands. The aim is to simulate long sequential computation.

```
for (int i = 2; i < N; ++i) {
  S01: a[i] = h[i-1] + R[i];
  S02: b[i] = a[i-1] + a[i];
  S03: c[i] = b[i-1] + b[i];
  S04: d[i] = c[i-1] + c[i];
  S05: e[i] = d[i-2] + d[i-1];
  S06: f[i] = e[i-2] + e[i-1];
  S07: g[i] = f[i] + X[i];
  S08: h[i] = g[i] + Y[i];
  S09: u[i] = v[i-1] + d[i];
  S10: v[i] = u[i] + Z[i];
}
```

Listing 6.5 – Van Dongen: original code

```
for (int i = 1; i < N; ++i) {
  S1: a[i] = X[i] + e[i - 1];
  S2: b[i] = a[i] - g[i - 1];
  S3: c[i] = b[i] + e[i];
  S4: d[i] = gamma1 * b[i];
  S5: e[i] = d[i] + e[i - 1];
  S6: f[i] = gamma2 * b[i];
  S7: g[i] = f[i] + g[i - 1];
  S8: Y[i] = c[i] - g[i];
}
```

Listing 6.6 – WDF: original code

We used Pluto [19] version 0.11.4 and `autoPar` from the Rose [86] Compiler version 0.9.12.0 on our code samples and can confirm these tools do not expose parallelism on our benchmarks (apart from the parallel loop in `mix`). Nonetheless, statements are reorganized in `wdf` and `van_dongen` to improve data locality.

We compared four versions of each benchmark: the output code of Pluto, the code annotated with our `ordered+nowait` approach and the code with explicit locks.

A tempting alternative to our approach is to use OpenMP `task` constructs. Indeed, the `task` construct (in conjunction with the `depend` annotation) allows finer control over execution order. We manually wrote the fourth version of our benchmarks where tasks are created for each loop body after we

```
for (int i = 1; i < N; ++i)
  for (int j = 0; j < M; ++j)
    for (int k = 0; k < M; ++k)
      S1: C[i] += B[j] + A[k] + 1.;
for (int i = 1; i < N; ++i) {
  S2: D[i] = D[i - 1] * C[i];
  S3: E[i] = E[i - 1] * D[i];
}
```

Listing 6.7 – Mix: original code

```
for (int i = 1; i < N; ++i)
  S1: A[i] = f(A[i], A[i-1]);
for (int i = 1; i < N; ++i)
  S2: B[i] = f(A[i], B[i-1]);
for (int i = 1; i < N; ++i)
  S3: C[i] = f(B[i], C[i-1]);
for (int i = 1; i < N; ++i)
  S4: D[i] = f(C[i], D[i-1]);
for (int i = 1; i < N; ++i)
  S5: E[i] = f(D[i], E[i-1]);
for (int i = 1; i < N; ++i)
  S6: F[i] = f(E[i], F[i-1]);
```

Listing 6.8 – Teaser: original code

applied loop distribution as described in Section 6.3.1.

Figure 6.4 presents results observed on the code compiled with gcc with options `-O3 -march=native -fopenmp`.

Both our approach and the task versions allow our `teaser+nanosleep` example to significantly outperform the sequential version. Moreover, it can be noticed that `task` creation competes with our approach when the number of tasks remains contained and compute times are high. However, the introduced overhead can tremendously impede the resulting program as the number of tasks grows larger and the compute times of the tasks is small. This can clearly be seen on `teaser`, `wdf` and `vandongen`. The overhead is due to the growing number of tasks to create and the dependencies which must be checked at runtime.

Explicit locks outperform the `ordered+nowait` versions and even allow

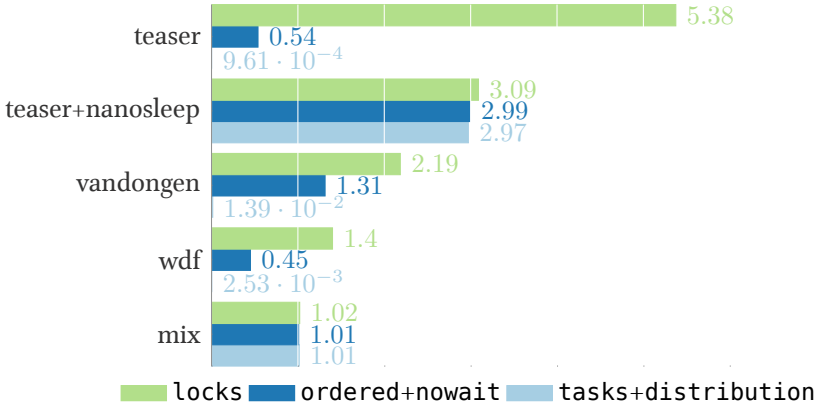


Figure 6.4 – **Pipelined Multithreading: Speedups Or Slowdowns Over Pluto Version (Code Compiled With gcc 9.2.1).** $N = 100,000$ for `teaser`, `wdf` and `van_dongen`, $N = 2,000$ and $M = 2,000$ for `mix`

our teaser example to exhibit speedups without the call to `nanosleep`. The results show the speedups for block sizes similar to the default chunk size for non blocked versions. We have observed even greater speedups with different block sizes (for instance a speedup of 9.6 with a block size of 2048 on our teaser code). The values were guessed based on the input code and cache size of our test environment. Further work is required to determine how to find optimal block sizes for a given pipeline and environment.

The measurements were also conducted over code compiled with `clang` and `libomp` and the results are presented in Figure 6.5. The same observations can be made: stage blocking and explicit locks generally outperform the tasks and `ordered+nowait` versions although the locking mechanism seems less efficient.

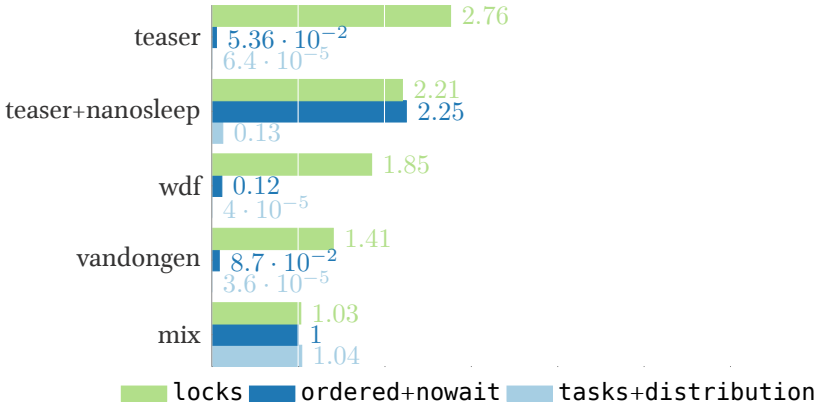


Figure 6.5 – **Speedups Or Slowdowns Over PLUTO Version (clang 9.0.1).** $N = 100,000$ for `teaser`, `wdf` and `van_dongen`. $N = 2000$ and $M = 2000$ for `mix`.

Figure 6.6 presents experimental results on the code compiled with icc 19.1.0.166. Synchronization is less efficient with icc/libiomp as can be observed on the `teaser+nanosleep` example. Great speedups are observed on the `mix` example because icc is able to heavily optimize the parallel loop and significantly decrease its compute time. Overall, those results show that the explicit locks version outperforms all the other versions on those pipelined loops whatever the compiler and execution environment.

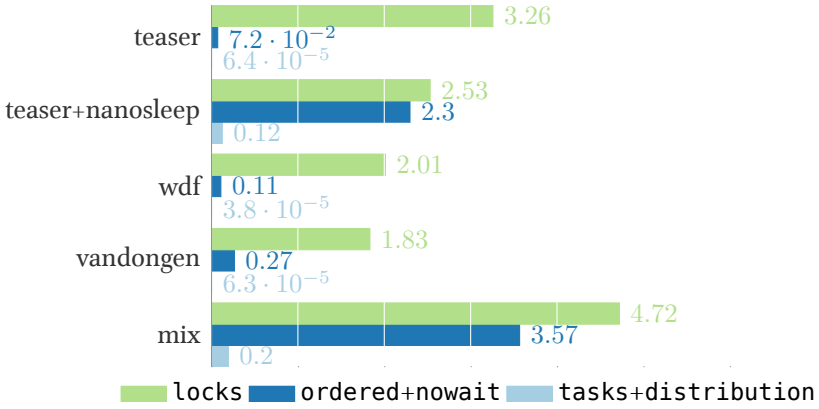


Figure 6.6 – **Speedups Or Slowdowns Over PLUTO Version (icc 19.1.0.166).** $N = 100,000$ for `teaser`, `wdf` and `van_dongen`. $N = 2000$ and $M = 2000$ for `mix`.

6.6 Future Work: Skewed Pipelines

We introduced explicitly synchronized pipelines in Section 6.4. Such pipelines offer greater control over the distributions of the iterations. In particular, they make it possible to control the placement of the *chunks*: different stages of the pipeline do not necessarily need to be of equal size or to be aligned (iteration domain wise). The techniques previously presented in this chapter targeted sequential loops. We show in this section how to pipeline parallel loops as well, using this greater control over *chunk* placement and sizes.

Further extending Equation (6.1), multithreaded pipelines could be created for loops if the dependencies between the loops on the parallel dimen-

sions are of the form:

$$\mathcal{L}_{S1,S2}(\vec{p}) \;=\; \{\vec{u} \;\rightarrow\; \vec{v}_{S2} \mid u_i - a \leq v_j \leq u_i + b\} \tag{6.2}$$

Let us consider `jacobi-1d` from the PolyBench suite as shown in Listing 6.9. A given iteration i of loop `S2` depends on previous iterations $i, i-1$ and $i+1$ of loop `S1`. In the same vein, (because the two loops are enclosed in a loop), a given iteration i of loop `S1` depends on previous iterations $i, i-1$ and $i+1$ of loop `S2`. PLUTO [19] version 0.11.4 with option `--parallel` will skew and fuse the two loops (see Appendix A.2.1).

```
1 for (t = 0; t < _PB_TSTEPS; t++) {
2   for (i = 1; i < _PB_N - 1; i++) {
3     S1: B[i] = 0.33333 * (A[i-1] + A[i] + A[i + 1]);
4   }
5   for (i = 1; i < _PB_N - 1; i++) {
6     S2: A[i] = 0.33333 * (B[i-1] + B[i] + B[i + 1]);
7   }
8 }
```

Listing 6.9 – **jacobi-1d: Original Code.** This kernel is a stencil computation over 1D data. This is a simplified version which computes the average of three points.

This code is actually eligible for multithreaded pipelining as shown in Listing 6.10. The method is similar to the method described in Section 6.4. However, the main difference is that each stage potentially starts and ends at a different index (the code example stores these indices in an array (lines 9-15) instead of inlining them in the loop bounds for legibility): each stage is one iteration shorter and starts one iteration earlier except the first stage. The execution order is close to skewed loops but the array accesses remain as is. This code features another useful characteristic for pipeline extraction: the surrounding loop over `t`. Partially unrolling this loop to taste makes it possible to expose as many stages as needed (within the boundaries set by the actual value of TSTEPS and the block sizes).

With options `--partlbtile --parallel`, PLUTO [19] can perform diamond tiling [17]. This tiling technique is especially efficient tiling for stencil computations. We compared three versions: the "skewed pipelines" code

```
for (size_t steps = 0; steps < TSTEPS; t++) {
  /* Omitted: code to set locks... */
  #pragma omp for schedule(static, 1)
  for (size_t block = 0; block <= block_count; ++block) {
    /* Compute self and next thread indexes. */
    const size_t self = block % num_threads;
    const size_t next = (block + 1) % num_threads;
    const int distance = 1;

    int start[stages], end[stages];
    for (size_t i = 0; i < stages; ++i) {
      const int skew = i + distance - 1;
      const int skew_start = offset + block * block_size - skew;
      start[i] = MAX(skew_start, offset);
      end[i] = MIN(skew_start + block_size, N - 1);
    }

    size_t stage = 0;
    omp_set_lock(&locks[self][stage]);
    for (int i = start[stage]; i < end[stage]; i++)
      B[i] = 0.33333 * (A[i-1] + A[i] + A[i + 1]);
    omp_unset_lock(&locks[next][stage]);
    stage++;

    omp_set_lock(&locks[self][stage]);
    for (int i = start[stage]; i < end[stage]; i++)
      A[i] = 0.33333 * (B[i-1] + B[i] + B[i + 1]);
    omp_unset_lock(&locks[next][stage]);
    stage++;
    /* Omitted: subsequent stages... */
  }
  /* Omitted: code to reset locks... */
}
```

Listing 6.10 – **jacobi-1d: Skewed Pipelines.**

(with 12 stages), the "parallel" version (the output of Pluto with `--parallel`) and the "diamond". We set `TSTEPS` to 1200 and the size to `EXTRALARGE` and compiled the three versions with with gcc 9.3.0, icc 19.1.0.166 and clang 9.0.1 using options `-O3 -march=native -fopenmp`. Table 6.1 presents execution times of the three versions.

Table 6.1 – **Execution Times For `jacobi-1d`**

flavour	gcc	icc	clang
parallel	0.02396233	0.09838166	0.12271866
skewed pipeline	0.00246000	0.01128500	0.00947800
diamond	0.00271200	0.00593033	0.00595900

With all three compilers the simple parallel version is the less favorable while the skewed pipelines are significantly faster. With gcc, the skewed pipelines are on par with the diamond tiles. Although diamond tiling increases the amount of concurrent starts, the compiler failed to vectorize some of the numerous loops in the diamond tiled version. All loops in the skewed pipelines were vectorized. The execution times are less encouraging on the versions compiled with icc and clang. As seen previously, both locking mechanisms with `clang/libomp` and `icc/libiomp` seem less efficient than in `gcc/libgomp`. Moreover, `icc` failed to vectorize the skewed pipeline stages.

The other multithreaded pipelining methods presented in this chapter target loops that are left as sequential after the scheduling phase and can be applied during the code generation phase. On the other hand, skewed pipelines generation target parallel loops and also require the scheduling phase to be modified: we do not want the scheduling phase to heavily transform the parallel loops which may benefit from skewed pipelines. This section introduced the idea of skewed pipelines, further work is needed to determine whether other stencil computations may benefit from this technique and how to modify scheduling algorithms to specifically target such pipelines.

6.7 Conclusion

In order to preserve the semantics of a program, some loops must remain sequential. However, this restriction does not necessarily forbid all forms of parallelism. We present a new compiler technique which is able to detect, to extract and to implement pipelined multithreading in some sequences of sequential loops. The specific dependence analysis for the parallel execu-

tion of interlaced iterations is enabled by polyhedral techniques, while the pipelined execution is implemented using a specific stage-blocking transformation and a code generation building on either `ordered` and `nowait` OpenMP clauses, or explicit locks. With our method, we are able to expose pipelined multithreading in programs which current state-of-the-art automatic polyhedral parallelizers fail to parallelize.

Our study shows that programs with multiple consecutive sequential loops after loop distribution and with long enough sequential iterations can benefit from pipelined multithreading. Future work should investigate how to find optimal chunk sizes for multithreaded pipelines. We present the general idea for skewed multithreaded pipelines. This new transformation may compete with tiling or diamond tiling and requires further refinements and investigations.

Chapter 7

Conclusion And Perspectives

High performance computing requires extensive knowledge of the problem domain, software engineering and the target hardware architecture. It is increasingly difficult for single individuals to master all of these trades at once. Automatic optimizers can alleviate this burden on programmers. Programmers can focus on problem solving and let the optimizing compilers select an adequate way to use the target hardware. Automatic loop nest optimization can be performed using tools based on the polyhedral model: various aspects of the loops (iteration domains, dependences, etc.) can be represented as unions of polyhedra. These polyhedra can be manipulated via algebraic transformations. Once polyhedra have been modified to reach a given optimization goal, a code that scans the integer points of these polyhedra is generated. In this thesis, we demonstrate that code generation in polyhedral compilers, — despite being considered as efficient — still has room for improvement and we propose new solutions towards generating efficient codes.

7.1 Contributions

7.1.1 Reducing Affine Control Overhead

Latest advances on polyhedral compilation focus on, e.g., tiling strategies [17], optimizing tensor codes [103, 111] and targeting accelerators [29, 4, 58, 106]

leave the impression that code generation algorithms in polyhedral compilers are considered to be fairly satisfactory. These algorithms produce a code that scans the integer points of unions of polyhedra [11]. Unions of polyhedra can be represented in multiple equivalent ways. Code generation tools use general-purpose libraries that do not necessarily focus on code generation and do not consider efficiency of the generated code.

We showed in Chapter 4 that the internal representation actually influences the performance of the end result. Generating code from unions of polyhedra may introduce control overhead in loop bounds such as minimum and maximum calculations. We demonstrate that these calculations can be avoided by properly splitting polyhedra.

We proposed to refine Bastoul's extension [11] (extended QRW) of the Quilleré, Rajopadhye and Wilde's [85] (QRW) algorithm for code generation: polyhedra splitting occurs during the separation phase of extended QRW algorithm.

Our method relies on the "chamber decomposition" of parametric polyhedra [65]. During the separation phase of the extended QRW algorithm, we temporarily convert outer iterators of the considered polyhedra into parameters. We then compute the chambers of these tuned polyhedra. Finally, these chambers are then intersected with the original polyhedra.

We implemented our algorithm in CLooG [11], a code generator that implements the extended QRW algorithm. We applied our code generation technique on CLooG's test suite and on programs from the PolyBench [83] benchmark suite. We compared the output of our method to the output of an unaltered version of CLooG. Experimental results show that our method is beneficial on average.

7.1.2 Reducing Synchronization Overhead

Classic automatic parallelizers such as PLUTO [19] generate code where parallel loops are isolated in distinct parallel regions using `omp parallel for` OpenMP annotations. We argue in Chapter 5 that separate parallel regions are not the best way to generate parallel code. Control overhead is induced by thread management at entry and exit of parallel regions and many super-

fluous synchronizations are performed.

We show how to generate wide parallel regions from an optimized polyhedral AST instead of sparse parallel regions. We proposed to enclose the whole optimized SCoP in a unique parallel region, to embed sequential code in `single` constructs and to annotate parallel loops with the `for` construct. Placing all worksharing constructs in a common parallel region also gives the opportunity to lift barriers: by default, worksharing constructs end with implicit barriers which may not always be required.

Our approach for synchronization reduction relies on dependence analysis through the polyhedral model. It allows us to identify sequences of worksharing constructs that may be executed without interlaced synchronizations. Our algorithm recurses on the input optimized AST and progressively builds groups of synchronization free worksharing constructs. Each worksharing construct is tested (for inclusion) against the current group and either expands the group or is placed in a new group. At the end of the algorithm, implicit synchronizations are removed within the groups and explicit synchronizations (if no implicit synchronization is present at this point) are placed at the end of each group.

We compared the code produced by our method to the output of PLUTO. Our best performing versions were up to 4x faster than the code whereas the least favorable exhibited no significant difference over the output of PLUTO. Based on these observations, this method should always be applied. Automatic parallelizers should target unique parallel regions and stop generating isolated parallel regions.

7.1.3 Pipelined multithreading

Automatic parallelizers such as PLUTO [19], PPCG [106] or ROSE [86] are fairly proficient in parallelizing a wide class SCoPs by introducing loop-level parallelism. However, they do not generate multi-loop pipeline parallelism.

We show in Chapter 6 that these compilers could not automatically parallelize loop nests where individual loops had to remain sequential but could be interlaced into pipelined multithreading. We presented a method for identifying, extracting and implementing such pipelines. Our method tar-

gets successive sequential loops after an automatic parallelization pass. We identify strongly connected components in the dependence graph of consecutive sequential loops and perform loop fission of sequential loops in order to maximize the number of pipeline stages: distinct strongly connected components belong to distinct loops. Pipelined multithreading validity is ensured by extending the dependence analysis introduced in Chapter 5: the method can be slightly relaxed for sequential loops. We then proposed two alternative code generation methods to expose pipelined multithreading: using `ordered` constructs or using explicit locks.

We carried out experiments on a limited set of cases. We compared the output of our algorithm to the output of Pluto. The output of Pluto was not parallel but the tool still performed data-locality optimizations. We showed that, for pipelined multithreading, explicit locks should be favoured over `ordered` constructs. Our versions with locks either performed better or were equivalent to the output of Pluto. Our study shows that programs including sequential loops could benefit from pipelined multithreading.

7.2 Perspectives

7.2.1 Affine Control Overhead Reduction

Chapter 4 introduced polyhedra splitting to reduce control overhead. Although experimental results show an overall benefit, performance is subpar on a few cases. As the transformation should not always be applied, a decision method is needed. Future work may attempt to identify what characteristics make a loop nest eligible to polyhedra splitting.

Another possible direction would be to gradually expand the set of target loop nests instead of limiting the splitting of polyhedra. The method we proposed indiscriminately splits all polyhedra up to a given threshold. Faced with the code explosion aspect of the algorithm, we eventually attempted to restrict the method. A starting point in the opposite direction was given in Section 4.6. We show how in a very simple case, a polyhedron can be split into several polyhedra via a naive inspection of the constraints. Hence, future work may achieve better results by progressively increasing the scope of

the transformation.

7.2.2 Transformed Pipelined Multithreading

We present pipelined multithreading in Chapter 6. This technique targets groups of successive loops. In Section 6.6, we explained how this method could be adapted to target parallel loops and used it on a example stencil kernel. A comparison with a version of this kernel where diamond tiling [17] is applied shows promising results. Further work should investigate how to systematically apply this transformation to parallel loops and whether it could complement or compete with other parallelization schemes.

7.2.3 Further use of OpenMP

Classic automatic parallelizers optimize loop nests by parallelizing loop nests with `#pragma omp parallel for`. We have shown in this thesis that the output programs could be further improved by using more features of OpenMP. The possibilities offered by OpenMP have yet to be used to their full extent by automatic parallelizers. Work [96, 22, 81] has been done in that direction but these methods require initial annotations to expose parallelism whereas other optimizers such as Pluto target loop nests without annotations.

For instance, our pipelined multithreading method is limited by the number of stages: the number of threads dedicated to a pipeline should not be greater than the number of stages. Adjusting the number of threads could potentially free some threads for other independent tasks. This could be addressed with the `teams` construct (which can also be used on the host since version 5.0 of OpenMP [79]) or with nested parallelism, if supported by the implementation. In the same vein, the `section` construct can be used to execute multiple independent sequential parts in parallel.

In this thesis, we explored some of the features provided by OpenMP. These features, and many other, are currently used only by expert programmers. Automatic optimizers should exploit them in the best way to reach the level of performance that a skilled parallel programmer can achieve.

Bibliography

[1] ISO/IEC JTC 1/SC 22. *Programming languages — C. ISO/IEC 9899:2018*. June 2018. URL: https://www.iso.org/standard/74528.html (cit. on p. 22).

[2] Alfred V Aho, Monica Lam, Ravi Sethi and Jeffrey D Ullman. *Compilers, principles, techniques, and tools*. 2nd ed. Pearson Education Limited, 2014. ISBN: 978-1-292-02434-9 (cit. on pp. 1, 9).

[3] Alexander Aiken and David Gay. "Barrier Inference". In: *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Diego, California, USA, 1998, pp. 342–354. DOI: 10.1145/268946.268974 (cit. on p. 34).

[4] Christophe Alias, Bogdan Pasca and Alexandru Plesco. "Automatic generation of FPGA-specific pipelined accelerators". In: *International Symposium on Applied Reconfigurable Computing*. Springer. 2011, pp. 53–66. DOI: 10.1007/978-3-642-19475-7_7 (cit. on pp. 35, 85, 107).

[5] Christophe Alias and Alexandru Plesco. "Optimizing affine control with semantic factorizations". In: *ACM Transactions on Architecture and Code Optimization* (*TACO*) 14.4 (2017), pp. 1–22. DOI: 10.1145/3162017 (cit. on p. 33).

[6] J. Allen and K. Kennedy. "Automatic translation of FORTRAN programs to vector form". In: *ACM Transactions on Programming Languages and Systems* 9.4 (Oct. 1987), pp. 491–542. DOI: 10.1145/29873.29875 (cit. on pp. 33, 34).

[7] Corinne Ancourt and François Irigoin. "Scanning polyhedra with DO loops". In: *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. PPOPP '91* (Williamsburg, Virginia, United States). Vol. 26. SIGPLAN Notices 7. Association for Computing Machinery. Apr. 1991, pp. 39–50. ISBN: 0897913906. DOI: 10.1145/109626.109631. URL: https://hal-mines-paristech.archives-ouvertes.fr/hal-00752774 (cit. on pp. 5, 31, 37).

[8] Arm. *Neon Intrinsics*. URL: https://developer.arm.com/architectures/instruction-sets/simd-isas/neon/intrinsics (cit. on p. 22).

[9] Cédric Bastoul. *CLooG: Chunky Loop Generator*. URL: http://cloog.org/ (cit. on p. 32).

[10] Cédric Bastoul. "Code Generation". In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Springer, 2011, pp. 310–318. ISBN: 978-0-387-09765-7. DOI: 10.1007/978-0-387-09766-4_67 (cit. on p. 37).

[11] Cédric Bastoul. "Code Generation in the Polyhedral Model Is Easier Than You Think". In: *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques. PACT 2004* (Antibes Juan-les-Pins, France). IEEE. Oct. 2004, pp. 7–16. ISBN: 0-7695-2229-7. DOI: 10.1109/PACT.2004.1342537. URL: https://hal.archives-ouvertes.fr/hal-00017260 (cit. on pp. 6, 20, 32, 37, 38, 43, 108).

[12] Cédric Bastoul. "Efficient Code Generation for Automatic Parallelization and Optimization". In: *Proceedings of the Second International Symposium on Parallel and Distributed Computing. ISPDC '03* (Ljubljana, Slovenia). IEEE Computer Society. Oct. 2003, pp. 23–30. ISBN: 0769520693. DOI: 10.1109/ISPDC.2003.1267639. URL: https://dl.acm.org/doi/abs/10.5555/1899290.1899294 (cit. on p. 37).

[13] Cédric Bastoul, Nicolas Vasilache, Allen Leung, Benoît Meister, David Wohlford and Richard Lethin. "Extended Static Control Programs as a Programming Model for Accelerators, A Case Study: Targetting ClearSpeed CSX700 With the R-Stream Compiler". In: *Workshop on Programming Models for Emerging Architectures. PMEA '09* (Raleigh, North Caro-

lina). Sept. 2009, pp. 45–52. URL: https://www.reservoir.com/publication/extended-static-control-programs-programming-model-accelerators-case-study-targeting-clear-speed-csx700-r-stream-compiler/ (cit. on p. 33).

[14] Daniel Baudisch, Jens Brandt and Klaus Schneider. "Multithreaded Code from Synchronous Programs: Generating Software Pipelines for OpenMP." In: *MBMV*. 2010, pp. 11–20 (cit. on p. 36).

[15] Wlodzimierz Bielecki, Marek Palkowski and Tomasz Klimek. "Free Scheduling for Statement Instances of Parameterized Arbitrarily Nested Affine Loops". In: *Parallel Comput.* 38.9 (Sept. 2012), pp. 518–532. DOI: 10.1016/j.parco.2012.06.001 (cit. on p. 34).

[16] Uday Bondhugula, Aravind Acharya and Albert Cohen. "The Pluto+ Algorithm: A Practical Approach for Parallelization and Locality Optimization of Affine Loop Nests". In: *ACM Transactions on Programming Languages and Systems. TOPLAS* 38.3 (Apr. 2016). Ed. by Andrew C. Myers. ISSN: 0164-0925. DOI: 10.1145/2896389. URL: https://dl.acm.org/doi/abs/10.1145/2896389 (cit. on pp. 5, 34).

[17] Uday Bondhugula, Vinayaka Bandishti and Irshad Pananilath. "Diamond tiling: Tiling techniques to maximize parallelism for stencil computations". In: *IEEE Transactions on Parallel and Distributed Systems* 28.5 (2016), pp. 1285–1298. DOI: 10.1109/TPDS.2016.2615094 (cit. on pp. 102, 107, 111).

[18] Uday Bondhugula, Oktay Günlük, Sanjeeb Dash and Lakshminarayanan Renganarayanan. "A model for fusion and code motion in an automatic parallelizing compiler". In: *Proceedings of the 19th International Conference on Parallel Architecture and Compilation Techniques. PACT '10* (Vienna, Austria). ACM SIGARCH et al. Association for Computing Machinery, Sept. 2010, pp. 343–352. ISBN: 978-1-4503-0178-7. DOI: 10.1145/1854273.1854317. URL: https://dl.acm.org/doi/abs/10.1145/1854273.1854317 (cit. on p. 34).

[19] Uday Bondhugula, Albert Hartono, J. Ramanujam and P. Sadayappan. "A practical automatic polyhedral parallelizer and locality op-

timizer". In: *SIGPLAN Notices* 43.6 (2008), pp. 101–113. ISSN: 0362-1340. DOI: 10.1145/1379022.1375595 (cit. on pp. 5, 20, 34, 64, 97, 102, 108, 109).

[20] J Mark Bull. "Measuring synchronisation and scheduling overheads in OpenMP". In: *Proceedings of First European Workshop on OpenMP*. Vol. 8. 1999, p. 49 (cit. on pp. 63, 67).

[21] Mark Bull, Fiona Reid and Nix Mc Donnell. *EPCC OpenMP microbenchmark suite, version 3.1*. 2015. URL: https://www.epcc.ed.ac.uk (cit. on pp. 63, 67).

[22] Prasanth Chatarasi, Jun Shirako and Vivek Sarkar. "Polyhedral optimizations of explicitly parallel programs". In: *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE. 2015, pp. 213–226. DOI: 10.1109/PACT.2015.44 (cit. on pp. 36, 111).

[23] Chun Chen. "Polyhedra Scanning Revisited". In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '12* (Beijing, China). Vol. 47. ACM SIGPLAN Notices 6. Association for Computing Machinery. June 2012, pp. 499–508. ISBN: 9781450312059. DOI: 10.1145/2345156.2254123. URL: https://dl.acm.org/doi/10.1145/2345156.2254123 (cit. on pp. 20, 32).

[24] Philippe Clauss. "Counting Solutions to Linear and Nonlinear Constraints Through Ehrhart Polynomials: Applications to Analyze and Transform Scientific Programs". In: *Proceedings of the 10th International Conference on Supercomputing*. Philadelphia, Pennsylvania, USA, 1996, pp. 278–285. DOI: 10.1145/237578.237617 (cit. on p. 73).

[25] Ron Cytron, Jim Lipkis and Edith Schonberg. "A Compiler-assisted Approach to SPMD Execution". In: *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*. Supercomputing '90. New York, New York, USA, 1990, pp. 398–406. DOI: 10.1109/SUPERC.1990.130047 (cit. on p. 34).

[26] Leonardo Dagum and Ramesh Menon. "OpenMP: an industry standard API for shared-memory programming". In: *IEEE Computational Science and Engineering* 5.1 (1998). ISSN: 1070-9924. DOI: 10.1109/99.660313 (cit. on pp. 22, 23).

[27] George B Dantzig. *Fourier-Motzkin elimination and its dual*. Tech. rep. STANFORD UNIV CA DEPT OF OPERATIONS RESEARCH, 1972 (cit. on p. 31).

[28] Alain Darte and Robert Schreiber. "A Linear-time Algorithm for Optimal Barrier Placement". In: *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '05. Chicago, IL, USA, 2005, pp. 26–35. DOI: 10.1145/1065944.1065949 (cit. on p. 34).

[29] Steven Derrien, Sanjay Rajopadhye, Patrice Quinton and Tanguy Risset. "High-level synthesis of loops using the polyhedral model". In: *High-level synthesis*. Springer, 2008, pp. 215–230. DOI: 10.1007/978-1-4020-8588-8_12 (cit. on pp. 35, 85, 107).

[30] Edsger W Dijkstra. "On a cultural gap". In: *The Mathematical Intelligencer* 8.1 (1986), pp. 48–52. DOI: 10.1007/BF03023921 (cit. on p. 1).

[31] Paul Feautrier. "Fine-grain scheduling under resource constraints". In: *International Workshop on Languages and Compilers for Parallel Computing*. Springer. 1994, pp. 1–15. DOI: 10.1007/BFb0025867 (cit. on p. 35).

[32] Paul Feautrier. "Parametric Integer Programming". In: *RAIRO Recherche opérationnelle* 22.3 (1988), pp. 243–268 (cit. on p. 5).

[33] Paul Feautrier. "Some efficient solutions to the affine scheduling problem. I. One-dimensional time". In: *International journal of parallel programming* 21.5 (1992), pp. 313–347. DOI: 10.1007/BF01407835 (cit. on pp. 34, 35).

[34] Paul Feautrier. "Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time". In: *International journal of parallel programming* 21.6 (1992), pp. 389–420. DOI: 10.1007/BF01379404 (cit. on pp. 20, 34, 35).

[35] Paul Feautrier and Christian Lengauer. "Polyhedron Model". In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Springer, 2011, pp. 1581–1592. ISBN: 978-0-387-09765-7. DOI: 10.1007/978-0-387-09766-4_502 (cit. on pp. 2, 5, 9, 37).

[36] Paul Feautrier, Éric Violard and Alain Ketterlin. "Improving the performance of x10 programs by clock removal". In: *International Conference on Compiler Construction*. Springer. 2014, pp. 113–132. DOI: 10.1007/978-3-642-54807-9_7 (cit. on p. 34).

[37] Felipe Fernandez and Patrice Quinton. *Extension of Chernikova's algorithm for solving general mixed linear programming problems*. Research Report RR-0943. INRIA, 1988. URL: https://hal.inria.fr/inria-00075615 (cit. on p. 10).

[38] Alfred Fettweis. "Wave digital filters: Theory and practice". In: *Proceedings of the IEEE* 74.2 (1986), pp. 270–327. DOI: 10.1109/PROC.1986.13458 (cit. on p. 96).

[39] Dirk Fimmel and Jan Müller. "Optimal software pipelining under resource constraints". In: *International Journal of Foundations of Computer Science* 12.06 (2001), pp. 697–718. DOI: 10.1142/S0129054101000825 (cit. on pp. 35, 96).

[40] The MPI Forum. "MPI: A Message Passing Interface". In: *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*. Supercomputing '93. Portland, Oregon, USA: Association for Computing Machinery, 1993, pp. 878–883. ISBN: 0818643404. DOI: 10.1145/169627.169855. URL: https://doi.org/10.1145/169627.169855 (cit. on p. 22).

[41] J.B.J Fourier. "Analyse des travaux de l'Académie Royale des Sciences pendant l'année 1824". In: *Partie mathématique* (1827) (cit. on p. 31).

[42] Marc Gonzalez, Eduard Ayguadé, Xavier Martorell and Jesus Labarta. "Exploiting pipelined executions in OpenMP". In: *2003 International Conference on Parallel Processing, 2003. Proceedings.* IEEE. 2003, pp. 153–160. DOI: 10.1109/ICPP.2003.1240576 (cit. on p. 36).

[43] Marc Gonzàlez, Eduard Ayguadé, Xavier Martorell and Jesús Labarta. "Complex pipelined executions in OpenMP parallel applications". In: *International Conference on Parallel Processing, 2001.* IEEE. 2001, pp. 295–302. DOI: 10.1109/ICPP.2001.952074 (cit. on p. 36).

[44] Martin Griebl, Christian Lengauer and Sabine Wetzel. "Code generation in the polytope model". In: *Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques* (*Cat. No. 98EX192*). IEEE. 1998, pp. 106–111. DOI: 10.1109/PACT.1998.727179 (cit. on p. 32).

[45] Tobias Grosser, Armin Größlinger and Christian Lengauer. "Polly – Performing polyhedral optimizations on a low-level intermediate representation". In: *Parallel Processing Letters* 22.04 (2012). DOI: 10.1142/S0129626412500107 (cit. on pp. 5, 34).

[46] Tobias Grosser, Sven Verdoolaege and Albert Cohen. "Polyhedral AST Generation Is More Than Scanning Polyhedra". In: *ACM Transactions on Programming Languages and Systems. TOPLAS* 37.4 (July 2015). Ed. by Andrew C. Myers. ISSN: 0164-0925. DOI: 10.1145/2743016. URL: https://dl.acm.org/doi/10.1145/2743016 (cit. on pp. 20, 32, 33, 37).

[47] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger and Louis-Noël Pouchet. "Polly-Polyhedral optimization in LLVM". In: *Proceedings of the First International Workshop on Polyhedral Compilation Techniques* (*IMPACT*). Vol. 2011. 2011, p. 1 (cit. on p. 5).

[48] Austin Group. *POSIX.1c, Threads extensions*. 1995. URL: https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/pthread.h.html (cit. on p. 22).

[49] Jens Gustedt. *Modern C*. Manning Publications, 2019 (cit. on p. 22).

[50] Intel. *Intel Intrinsics Guide*. online. URL: https://software.intel.com/sites/landingpage/IntrinsicsGuide (cit. on p. 22).

[51] Richard M Karp, Raymond E Miller and Shmuel Winograd. "The organization of computations for uniform recurrence equations". In: *Journal of the ACM* (*JACM*) 14.3 (1967), pp. 563–590. DOI: 10.1145/321406.321418 (cit. on p. 5).

[52] Arun Kejariwal and Alexandru Nicolau. "Modulo Scheduling and Loop Pipelining". In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Springer, 2011, pp. 1158–1173. ISBN: 978-0-387-09765-7. DOI: 10.1007/978-0-387-09766-4_65 (cit. on p. 35).

[53] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman and D. Wonnacott. *The Omega Library*. Tech. rep. University of Maryland, Nov. 1996 (cit. on p. 31).

[54] Wayne Kelly, William Pugh and Evan Rosser. "Code generation for multiple mappings". In: *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation. Frontiers '95* (McLean, VA, USA). IEEE. Feb. 1995, pp. 332–341. ISBN: 0-8186-6965-9. DOI: 10.1109/FMPC.1995.380437. URL: https://ieeexplore.ieee.org/abstract/document/380437 (cit. on pp. 20, 31).

[55] DaeGon Kim, Lakshminarayanan Renganarayanan, Dave Rostron, Sanjay Rajopadhye and Michelle Mills Strout. "Multi-level tiling: M for the price of one". In: *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. 2007, pp. 1–12. DOI: 10.1145/1362622.1362691 (cit. on p. 33).

[56] Monica Lam. "Software Pipelining: An Effective Scheduling Technique for VLIW Machines". In: *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*. PLDI '88. Atlanta, Georgia, USA: Association for Computing Machinery, 1988, pp. 318–328. ISBN: 0897912691. DOI: 10.1145/53990.54022 (cit. on p. 35).

[57] Leslie Lamport. "The parallel execution of do loops". In: *Communications of the ACM* 17.2 (1974), pp. 83–93. DOI: 10.1145/360827.360844 (cit. on pp. 5, 33).

[58] Chris Lattner, Jacques Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache and Oleksandr Zinenko. "MLIR: A Compiler Infrastructure for the End of Moore's Law". In: *arXiv preprint arXiv:2002.11054* (2020) (cit. on p. 107).

[59] Marc Le Fur. "Scanning parameterized polyhedron using Fourier-Motzkin elimination". In: *Concurrency and Computation: Practice and Experience. CCPE* 8.6 (July 1996). Ed. by Geoffrey C., David W. Walker, Jinjun Chen and Jeyan Thiyagalingam, pp. 445–460. ISSN: 1532-0634. DOI: 10.1002/(SICI)1096-9128(199607)8:6<445::AID-CPE253>3.0.CO;2-G. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/%28SICI%291096-9128%28199607%298%3A6%3C445%3A%3AAID-CPE253%3E3.0.CO%3B2-G (cit. on p. 31).

[60] Hervé Le Verge. *A Note on Chernikova's algorithm*. Research Report RR-1662. INRIA, 1992. URL: https://hal.inria.fr/inria-00074895 (cit. on p. 10).

[61] Hervé Le Verge, Vincent Van Dongen and Doran K. Wilde. *Loop nest synthesis using the polyhedral library*. Research Report RR-2288. INRIA, 1994. URL: https://hal.inria.fr/inria-00074384 (cit. on p. 32).

[62] A. Lim. "Improving Parallelism and Data Locality with Affine Partitioning". PhD thesis. Stanford University, 2001 (cit. on p. 34).

[63] Amy W. Lim and Monica S. Lam. "Maximizing Parallelism and Minimizing Synchronization with Affine Transforms". In: *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Paris, France, 1997, pp. 201–214. DOI: 10.1145/263699.263719 (cit. on p. 34).

[64] Xiaoxian Liu, Rongcai Zhao, Lin Han and Peng Liu. "An Automatic Parallel-Stage Decoupled Software Pipelining Parallelization Algorithm Based on OpenMP". In: *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE. 2013, pp. 1825–1831. DOI: 10.1109/TrustCom.2013.227 (cit. on p. 36).

[65] V. Loechner and D. K. Wilde. "Parameterized Polyhedra and their Vertices". In: *International Journal of Parallel Programming* 25.6 (Dec. 1997), pp. 525–549. DOI: 10.1023/A:1025117523902 (cit. on pp. 6, 38, 40, 46, 108).

[66] Vincent Loechner. *PolyLib: A library for manipulating parameterized polyhedra*. 1999. URL: http://icps.u-strasbg.fr/polylib/ (cit. on p. 32).

[67] Nelson Lossing, Corinne Ancourt and Francois Irigoin. "Automatic Code Generation of Distributed Parallel Tasks". In: *2016 IEEE Intl Conference on Computational Science and Engineering (CSE) and IEEE Intl Conference on Embedded and Ubiquitous Computing (EUC) and 15th Intl Symposium on Distributed Computing and Applications for Business Engineering (DCABES)*. IEEE. 2016, pp. 234–241. DOI: 10.1109/CSE-EUC-DCABES.2016.190 (cit. on p. 34).

[68] TH Matheiss and David S Rubin. "A survey and comparison of methods for finding all vertices of convex polyhedral sets". In: *Mathematics of operations research* 5.2 (1980), pp. 167–185. DOI: 10.1287/moor.5.2.167 (cit. on p. 10).

[69] Benoît Meister, Nicolas Vasilache, David Wohlford, Muthu Manikandan Baskaran, Allen Leung and Richard Lethin. "R-Stream Compiler". In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Springer, 2011, pp. 1756–1765. ISBN: 978-0-387-09765-7. DOI: 10.1007/978-0-387-09766-4 (cit. on pp. 5, 34).

[70] H Minkowski. "Geometrie der Strahlen (Erste Lieferung)". In: *Teubner, Leipzig (reprinted: Chelsea, New York, 1953)* (1896) (cit. on p. 10).

[71] Antoine Morvan, Steven Derrien and Patrice Quinton. "Polyhedral bubble insertion: A method to improve nested loop pipelining for high-level synthesis". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32.3 (2013), pp. 339–352. DOI: 10.1109/TCAD.2012.2228270 (cit. on pp. 35, 85).

[72] Randall Munroe. "Real Programmers". In: *xkcd*. Feb. 2008. URL: https://xkcd.com/378/ (cit. on p. 2).

[73] Aaftab Munshi. "The opencl specification". In: *2009 IEEE Hot Chips 21 Symposium (HCS)*. IEEE. 2009, pp. 1–314. DOI: 10.1109/HOTCHIPS.2009.7478342 (cit. on p. 22).

[74] John Nickolls, Ian Buck, Michael Garland and Kevin Skadron. "Scalable parallel programming with CUDA". In: *Queue* 6.2 (2008), pp. 40–53. DOI: 10.1145/1365490.1365500 (cit. on p. 22).

[75] Qi Ning and Guang R Gao. "A novel framework of register allocation for software pipelining". In: *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1993, pp. 29–42. DOI: 10.1145/158511.158519 (cit. on p. 35).

[76] Michael O'Boyle and Elena Stohr. "Compile time barrier synchronization minimization". In: *IEEE Transactions on Parallel and Distributed Systems* 13.6 (2002), pp. 529–543. DOI: 10.1109/TPDS.2002.1011394 (cit. on p. 34).

[77] OpenACC Working Group. *The OpenACC Application Programming Interface version 2.7*. Nov. 2018. URL: https://www.openacc.org/sites/default/files/inline-files/OpenACC.2.7.pdf (cit. on p. 22).

[78] OpenMP Architecture Review Board. *OpenMP Application Program Interface version 4.5*. Nov. 2015. URL: https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf (cit. on p. 27).

[79] OpenMP Architecture Review Board. *OpenMP Application Program Interface version 5.0*. Nov. 2018. URL: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf (cit. on pp. 22, 23, 27, 34, 63, 89, 91, 111).

[80] Janak H Patel and Edward S Davidson. "Improving the throughput of a pipeline by insertion of delays". In: *Proceedings of the 3rd annual symposium on Computer architecture*. 1976, pp. 159–164. DOI: 10.1145/800110.803575 (cit. on pp. 35, 85).

[81] Antoniu Pop and Albert Cohen. "OpenStream: Expressiveness and data-flow compilation of OpenMP streaming programs". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 9.4 (2013), p. 53. DOI: 10.1145/2400682.2400712 (cit. on pp. 36, 111).

[82] Sebastian Pop, Albert Cohen, Cédric Bastoul, Sylvain Girbal, Georges-André Silber and Nicolas Vasilache. "GRAPHITE: Polyhedral analyses and optimizations for GCC". In: *Proceedings of the 2006 GCC Developers Summit*. Citeseer. 2006, p. 2006 (cit. on p. 5).

[83] Louis-Noël Pouchet. *PolyBench/C 4.1: The polyhedral benchmark suite*. 2015. URL: http://polybench.sourceforge.net (cit. on pp. 64, 78, 108).

[84] William Pugh. "The Omega test: a fast and practical integer programming algorithm for dependence analysis". In: *Supercomputing'91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*. IEEE. 1991, pp. 4–13. DOI: 10.1145/125826.125848 (cit. on p. 31).

[85] Fabien Quilleré, Sanjay Rajopadhye and Doran Wilde. "Generation of Efficient Nested Loops from Polyhedra". In: *International Journal of Parallel Programming* 28.5 (2000), pp. 469–498. ISSN: 1573-7640. DOI: 10.1023/A:1007554627716. URL: http://dx.doi.org/10.1023/A:1007554627716 (cit. on pp. 6, 20, 32, 37, 42, 108).

[86] Dan Quinlan and Chunhua Liao. "The ROSE source-to-source compiler infrastructure". In: *Cetus users and compiler infrastructure workshop, in conjunction with PACT*. Vol. 2011. Citeseer. 2011, p. 1 (cit. on pp. 97, 109).

[87] Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew J Bridges and David I August. "Parallel-stage decoupled software pipelining". In: *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*. ACM. 2008, pp. 114–123. DOI: 10.1145/1356058.1356074 (cit. on p. 36).

[88] B Ramakrishna Rau. "Iterative modulo scheduling". In: *International Journal of Parallel Programming* 24.1 (1996), pp. 3–64. DOI: 10.1007/BF03356742 (cit. on p. 35).

[89] B Ramakrishna Rau. "Iterative modulo scheduling: An algorithm for software pipelining loops". In: *Proceedings of MICRO-27. The 27th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE. 1994, pp. 63–74. DOI: 10.1145/192724.192731 (cit. on p. 35).

[90] B Ramakrishna Rau and Christopher D Glaeser. "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing". In: *ACM SIGMICRO Newsletter* 12.4 (1981), pp. 183–198. DOI: 10.1145/1014192.802449 (cit. on pp. 35, 85).

[91] Harenome Razanajato, Cédric Bastoul and Vincent Loechner. "Lifting Barriers Using Parallel Polyhedral Regions". In: *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*. IEEE. 2017, pp. 338–347. DOI: 10.1109/HiPC.2017.00046 (cit. on pp. 7, 140).

[92] Harenome Razanajato, Cédric Bastoul and Vincent Loechner. "Pipelined Multithreading Generation in a Polyhedral Compiler". In: *IMPACT 2020, in conjunction with HiPEAC 2020*. Bologna, Italy, Jan. 2020. URL: https://hal.inria.fr/hal-02456521 (cit. on pp. 7, 140).

[93] Harenome Razanajato, Vincent Loechner and Cédric Bastoul. "Splitting Polyhedra to Generate More Efficient Code". In: *IMPACT 2017, 7th International Workshop on Polyhedral Compilation Techniques*. Stockholm, Sweden, Jan. 2017. URL: https://hal.inria.fr/hal-01505764 (cit. on pp. 6, 140).

[94] Lakshminarayanan Renganarayanan, DaeGon Kim, Sanjay Rajopadhye and Michelle Mills Strout. "Parameterized tiled loops for free". In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2007, pp. 405–414. DOI: 10.1145/1250734.1250780 (cit. on p. 33).

[95] Lakshminarayanan Renganarayanan, Daegon Kim, Michelle Mills Strout and Sanjay Rajopadhye. "Parameterized Loop Tiling". In: *ACM Trans. Program. Lang. Syst.* 34.1 (May 2012). ISSN: 0164-0925. DOI: 10.1145/2160910.2160912. URL: https://doi.org/10.1145/2160910.2160912 (cit. on p. 33).

[96] Alina Sbîrlea, Jun Shirako, Louis-Noël Pouchet and Vivek Sarkar. "Polyhedral optimizations for a data-flow graph language". In: *Languages and Compilers for Parallel Computing*. Springer. 2015, pp. 57–72. DOI: 10.1007/978-3-319-29778-1_4 (cit. on pp. 36, 111).

[97] Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, 1998. ISBN: 978-0-471-98232-6 (cit. on pp. 10, 31).

[98] Todor Stefanov, Claudiu Zissulescu, Alexandru Turjan, Bart Kienhuis and Ed Deprettere. "System design using Kahn process networks: the Compaan/Laura approach". In: *Proceedings of the conference on Design, automation and test in Europe-Volume 1*. IEEE Computer Society. 2004, p. 10340. DOI: 10.1109/DATE.2004.1268870 (cit. on p. 35).

[99] Konrad Trifunovic, Albert Cohen, David Edelsohn, Feng Li, Tobias Grosser, Harsha Jagasia, Razya Ladelsky, Sebastian Pop, Jan Sjödin and Ramakrishna Upadrasta. "GRAPHITE Two Years After: First Lessons Learned From Real-World Polyhedral Compilation". In: *GCC Research Opportunities Workshop (GROW'10)*. Pisa, Italy, Jan. 2010. URL: https://hal.inria.fr/inria-00551516 (cit. on p. 34).

[100] Chau-Wen Tseng. "Compiler Optimizations for Eliminating Barrier Synchronization". In: *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPOPP '95. Santa Barbara, California, USA, 1995, pp. 144–155. DOI: 10.1145/209937.209952 (cit. on p. 34).

[101] Vincent H Van Dongen, Guang R Gao and Qi Ning. "A polynomial time method for optimal software pipelining". In: *Parallel Processing: CONPAR 92—VAPP V*. Springer, 1992, pp. 613–624. DOI: 10.1007/3-540-55895-0_462 (cit. on p. 96).

[102] Nicolas Vasilache, Cédric Bastoul and Albert Cohen. "Polyhedral Code Generation in the Real World". In: *Proceedings of the International Conference on Compiler Construction (ETAPS CC'06)*. LNCS 3923. Vienna, Austria, Mar. 2006, pp. 185–201. DOI: 10.1007/11688839_16 (cit. on pp. 32, 37).

[103] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams and Albert Cohen. "Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions". In: *arXiv preprint arXiv:1802.04730* (2018) (cit. on p. 107).

[104] Sven Verdoolaege. "isl: An integer set library for the polyhedral model". In: *International Congress on Mathematical Software*. Springer. 2010, pp. 299–302. DOI: 10.1007/978-3-642-15582-6_49 (cit. on p. 32).

[105] Sven Verdoolaege. "Polyhedral process networks". In: *Handbook of Signal Processing Systems*. Springer, 2013, pp. 1335–1375. DOI: 10.1007/978-1-4614-6859-2_41 (cit. on p. 35).

[106] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, Jose Ignacio Gomez, Christian Tenllado and Francky Catthoor. "Polyhedral parallel code generation for CUDA". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 9.4 (2013), pp. 1–23. DOI: 10.1145/2400682.2400713 (cit. on pp. 5, 107, 109).

[107] David W Walker and Jack J Dongarra. "MPI: a standard message passing interface". In: *Supercomputer* 12 (1996), pp. 56–68 (cit. on p. 22).

[108] Sabine Wetzel, Christian Lengauer and Martin Griebl. "Automatic code generation in the polytope model". In: *Master's thesis, Fakultät für Mathematik und Informatik, Universität Passau* (1995) (cit. on p. 32).

[109] Sandra Wienke, Paul Springer, Christian Terboven and Dieter an Mey. "OpenACC—first experiences with real-world applications". In: *European Conference on Parallel Processing*. Springer. 2012, pp. 859–870. DOI: 10.1007/978-3-642-32820-6_85 (cit. on p. 22).

[110] M. E. Wolf and M. S. Lam. "A Loop Transformation Theory and an Algorithm to Maximize Parallelism". In: *IEEE Trans. Parallel Distrib. Syst.* 2.4 (Oct. 1991), pp. 452–471. DOI: 10.1109/71.97902 (cit. on p. 33).

[111] Tim Zerrell and Jeremy Bruestle. "Stripe: Tensor compilation via the nested polyhedral model". In: *arXiv preprint arXiv:1903.06498* (2019) (cit. on p. 107).

[112] Jisheng Zhao, Jun Shirako, V. Krishna Nandivada and Vivek Sarkar. "Reducing Task Creation and Termination Overhead in Explicitly Parallel Programs". In: *Proceedings of the 19th International Conference on Parallel Architecture and Compilation Techniques. PACT '10* (Vienna, Austria). ACM SIGARCH et al. Association for Computing Machinery, Sept. 2010, pp. 169–180. ISBN: 978-1-4503-0178-7. DOI: 10.1145/1854273.1854298. URL: https://dl.acm.org/doi/abs/10.1145/1854273.1854298 (cit. on p. 35).

[113] Wei Zuo, Peng Li, Deming Chen, Louis-Noël Pouchet, Shunan Zhong and Jason Cong. "Improving polyhedral code generation for high-level synthesis". In: *2013 International Conference on Hardware/Software Codesign and System Synthesis* (*CODES+ ISSS*). IEEE. 2013, pp. 1–10. DOI: 10.1109/CODES-ISSS.2013.6659002 (cit. on p. 33).

[114] Wei Zuo, Yun Liang, Peng Li, Kyle Rupnow, Deming Chen and Jason Cong. "Improving high level synthesis optimization opportunity through polyhedral transformations". In: *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*. ACM. 2013, pp. 9–18. DOI: 10.1145/2435264.2435271 (cit. on p. 85).

Appendix A

Code Excerpts

A.1 Reducing Affine Loop Nests Control Overhead

A.1.1 Split Tiled Motivation

```
/* Generated from o2.cloog by CLooG 0.18.4-8bde68f gmp bits in
   0.42s. */
for (t1=0;t1<=floord(M-7,8);t1++) {
  for (t2=0;t2<=t1-1;t2++) {
    for (c1=8*t1;c1<=8*t1+7;c1++) {
      for (c2=8*t2;c2<=8*t2+7;c2++) {
        S1(t1,t2,(c1-c2),c2);
      }
    }
  }
  for (t2=t1;t2<=floord(8*t1+7,8);t2++) {
    for (c1=8*t2;c1<=8*t1+7;c1++) {
      for (c2=8*t2;c2<=c1;c2++) {
        S1(t1,t2,(c1-c2),c2);
      }
    }
  }
}
```

```
18  if (M%8 <= 6) {
19    t1 = floord(M,8);
20    for (t2=ceild(4*t1-3,4);t2<=0;t2++) {
21      for (c1=8*t1;c1<=8*t2+7;c1++) {
22        for (c2=0;c2<=c1;c2++) {
23          S1(t1,t2,(c1-c2),c2);
24        }
25      }
26      for (c1=8*t2+8;c1<=8*t1+7;c1++) {
27        for (c2=0;c2<=8*t2+7;c2++) {
28          S1(t1,t2,(c1-c2),c2);
29        }
30      }
31    }
32    for (t2=0;t2<=t1-1;t2++) {
33      for (c1=8*t1;c1<=8*t1+7;c1++) {
34        for (c2=8*t2;c2<=8*t2+7;c2++) {
35          S1(t1,t2,(c1-c2),c2);
36        }
37      }
38    }
39    for (t2=max(1,ceild(4*t1-3,4));t2<=floord(M-8,8);t2++) {
40      for (c1=8*t1;c1<=8*t2+6;c1++) {
41        for (c2=8*t2;c2<=c1;c2++) {
42          S1(t1,t2,(c1-c2),c2);
43        }
44      }
45      for (c1=8*t2+7;c1<=8*t1+7;c1++) {
46        for (c2=8*t2;c2<=8*t2+7;c2++) {
47          S1(t1,t2,(c1-c2),c2);
48        }
49      }
50    }
51    for (t2=max(1,ceild(M-7,8));t2<=t1;t2++) {
```

```
52      for (c1=8*t1;c1<=M-1;c1++) {
53        for (c2=8*t2;c2<=c1;c2++) {
54          S1(t1,t2,(c1-c2),c2);
55        }
56      }
57      for (c1=M;c1<=8*t1+7;c1++) {
58        for (c2=8*t2;c2<=M;c2++) {
59          S1(t1,t2,(c1-c2),c2);
60        }
61      }
62    }
63  }
64  for (t1=ceild(M+1,8);t1<=floord(N-7,8);t1++) {
65    for (t2=0;t2<=floord(M-8,8);t2++) {
66      for (c1=8*t1;c1<=8*t1+7;c1++) {
67        for (c2=8*t2;c2<=8*t2+7;c2++) {
68          S1(t1,t2,(c1-c2),c2);
69        }
70      }
71    }
72    t2 = floord(M,8);
73    for (c1=8*t1;c1<=8*t1+7;c1++) {
74      for (c2=8*t2;c2<=M;c2++) {
75        S1(t1,t2,(c1-c2),c2);
76      }
77    }
78  }
79  if ((N+7)%8 <= 5) {
80    t1 = floord(N-1,8);
81    for (t2=0;t2<=floord(8*t1-N,8);t2++) {
82      for (c1=8*t1;c1<=N-1;c1++) {
83        for (c2=0;c2<=8*t2+7;c2++) {
84          S1(t1,t2,(c1-c2),c2);
85        }
```

```
86      }
87      for (c1=N;c1<=8*t2+N+7;c1++) {
88        for (c2=c1-N;c2<=8*t2+7;c2++) {
89          S1(t1,t2,(c1-c2),c2);
90        }
91      }
92    }
93    for (t2=0;t2<=min(floord(M-7,8),floord(8*t1-N+6,8));t2++) {
94      for (c1=8*t1;c1<=8*t2+N;c1++) {
95        for (c2=8*t2;c2<=8*t2+7;c2++) {
96          S1(t1,t2,(c1-c2),c2);
97        }
98      }
99      for (c1=8*t2+N+1;c1<=8*t1+7;c1++) {
100       for (c2=c1-N;c2<=8*t2+7;c2++) {
101         S1(t1,t2,(c1-c2),c2);
102       }
103     }
104   }
105   for (t2=ceild(M-6,8);t2<=floord(8*t1-N+6,8);t2++) {
106     for (c1=8*t1;c1<=8*t2+N;c1++) {
107       for (c2=8*t2;c2<=M;c2++) {
108         S1(t1,t2,(c1-c2),c2);
109       }
110     }
111     for (c1=8*t2+N+1;c1<=8*t1+7;c1++) {
112       for (c2=c1-N;c2<=M;c2++) {
113         S1(t1,t2,(c1-c2),c2);
114       }
115     }
116   }
117   for (t2=ceild(8*t1-N+7,8);t2<=floord(M-8,8);t2++) {
118     for (c1=8*t1;c1<=8*t1+7;c1++) {
119       for (c2=8*t2;c2<=8*t2+7;c2++) {
```

```
120           S1(t1,t2,(c1-c2),c2);
121         }
122       }
123     }
124     for
      ↪   (t2=max(ceild(M-7,8),ceild(8*t1-N+7,8));t2<=floord(M,8);t2++)
      ↪   {
125       for (c1=8*t1;c1<=8*t1+7;c1++) {
126         for (c2=8*t2;c2<=M;c2++) {
127           S1(t1,t2,(c1-c2),c2);
128         }
129       }
130     }
131   }
132   for (t1=ceild(N,8);t1<=floord(N+M-8,8);t1++) {
133     if ((7*N+7)%8 <= 6) {
134       t2 = floord(8*t1-N-1,8);
135       for (c1=8*t1;c1<=8*t2+N+7;c1++) {
136         for (c2=c1-N;c2<=8*t2+7;c2++) {
137           S1(t1,t2,(c1-c2),c2);
138         }
139       }
140     }
141     for
      ↪   (t2=ceild(8*t1-N,8);t2<=min(floord(M-7,8),floord(8*t1-N+6,8));t2++)
      ↪   {
142       for (c1=8*t1;c1<=8*t2+N;c1++) {
143         for (c2=8*t2;c2<=8*t2+7;c2++) {
144           S1(t1,t2,(c1-c2),c2);
145         }
146       }
147       for (c1=8*t2+N+1;c1<=8*t1+7;c1++) {
148         for (c2=c1-N;c2<=8*t2+7;c2++) {
149           S1(t1,t2,(c1-c2),c2);
```

```
150         }
151       }
152     }
153     for (t2=ceild(M-6,8);t2<=floord(8*t1-N+6,8);t2++) {
154       for (c1=8*t1;c1<=8*t2+N;c1++) {
155         for (c2=8*t2;c2<=M;c2++) {
156           S1(t1,t2,(c1-c2),c2);
157         }
158       }
159       for (c1=8*t2+N+1;c1<=8*t1+7;c1++) {
160         for (c2=c1-N;c2<=M;c2++) {
161           S1(t1,t2,(c1-c2),c2);
162         }
163       }
164     }
165     for (t2=ceild(8*t1-N+7,8);t2<=floord(M-8,8);t2++) {
166       for (c1=8*t1;c1<=8*t1+7;c1++) {
167         for (c2=8*t2;c2<=8*t2+7;c2++) {
168           S1(t1,t2,(c1-c2),c2);
169         }
170       }
171     }
172     for
      ↪  (t2=max(ceild(M-7,8),ceild(8*t1-N+7,8));t2<=floord(M,8);t2++)
      ↪  {
173       for (c1=8*t1;c1<=8*t1+7;c1++) {
174         for (c2=8*t2;c2<=M;c2++) {
175           S1(t1,t2,(c1-c2),c2);
176         }
177       }
178     }
179   }
180   t1 = floord(N+M,8);
181   for (t2=ceild(8*t1-N-7,8);t2<=floord(M-8,8);t2++) {
```

```
182   for (c1=8*t1;c1<=8*t2+N+7;c1++) {
183     for (c2=c1-N;c2<=8*t2+7;c2++) {
184       S1(t1,t2,(c1-c2),c2);
185     }
186   }
187 }
188 for (t2=ceild(M-7,8);t2<=floord(8*t1-N,8);t2++) {
189   for (c1=8*t1;c1<=N+M;c1++) {
190     for (c2=c1-N;c2<=M;c2++) {
191       S1(t1,t2,(c1-c2),c2);
192     }
193   }
194 }
195 for (t2=ceild(8*t1-N+1,8);t2<=floord(M,8);t2++) {
196   for (c1=8*t1;c1<=8*t2+N;c1++) {
197     for (c2=8*t2;c2<=M;c2++) {
198       S1(t1,t2,(c1-c2),c2);
199     }
200   }
201   for (c1=8*t2+N+1;c1<=N+M;c1++) {
202     for (c2=c1-N;c2<=M;c2++) {
203       S1(t1,t2,(c1-c2),c2);
204     }
205   }
206 }
```

A.2 Pipelined Multithreading

A.2.1 Jacobi-1d: Pluto Parallel

```
1 int t, i;
2 int t1, t2, t3;
3 int lb, ub, lbp, ubp, lb2, ub2;
4 register int lbv, ubv;
```

```
/* Start of CLooG code */
if ((_PB_N >= 3) && (_PB_TSTEPS >= 1)) {
  B[1] = 0.33333 * (A[1 -1] + A[1] + A[1 + 1]);;
  for (t1=2;t1<=min(_PB_N-2,3*_PB_TSTEPS-2);t1++) {
    if ((2*t1+1)%3 == 0) {
      B[1] = 0.33333 * (A[1 -1] + A[1] + A[1 + 1]);;
    }
    lbp=ceild(2*t1+2,3);
    ubp=t1;
#pragma omp parallel for private(lbv,ubv,t3)
    for (t2=lbp;t2<=ubp;t2++) {
      B[(-2*t1+3*t2)] = 0.33333 * (A[(-2*t1+3*t2)-1] +
      ↪  A[(-2*t1+3*t2)] + A[(-2*t1+3*t2) + 1]);;
      A[(-2*t1+3*t2-1)] = 0.33333 * (B[(-2*t1+3*t2-1)-1] +
      ↪  B[(-2*t1+3*t2-1)] + B[(-2*t1+3*t2-1) + 1]);;
    }
  }
  if (_PB_N == 3) {
    for (t1=2;t1<=3*_PB_TSTEPS-2;t1++) {
      if ((2*t1+1)%3 == 0) {
        B[1] = 0.33333 * (A[1 -1] + A[1] + A[1 + 1]);;
      }
      if ((2*t1+2)%3 == 0) {
        A[1] = 0.33333 * (B[1 -1] + B[1] + B[1 + 1]);;
      }
    }
  }
  for (t1=3*_PB_TSTEPS-1;t1<=_PB_N-2;t1++) {
    lbp=t1-_PB_TSTEPS+1;
    ubp=t1;
#pragma omp parallel for private(lbv,ubv,t3)
    for (t2=lbp;t2<=ubp;t2++) {
      B[(-2*t1+3*t2)] = 0.33333 * (A[(-2*t1+3*t2)-1] +
      ↪  A[(-2*t1+3*t2)] + A[(-2*t1+3*t2) + 1]);;
```

```
36        A[(-2*t1+3*t2-1)] = 0.33333 * (B[(-2*t1+3*t2-1)-1] +
          ↪  B[(-2*t1+3*t2-1)] + B[(-2*t1+3*t2-1) + 1]);;
37      }
38    }
39    if (_PB_N >= 4) {
40      for (t1=_PB_N-1;t1<=3*_PB_TSTEPS-2;t1++) {
41        if ((2*t1+1)%3 == 0) {
42          B[1] = 0.33333 * (A[1 -1] + A[1] + A[1 + 1]);;
43        }
44        lbp=ceild(2*t1+2,3);
45        ubp=floord(2*t1+_PB_N-2,3);
46  #pragma omp parallel for private(lbv,ubv,t3)
47        for (t2=lbp;t2<=ubp;t2++) {
48          B[(-2*t1+3*t2)] = 0.33333 * (A[(-2*t1+3*t2)-1] +
            ↪  A[(-2*t1+3*t2)] + A[(-2*t1+3*t2) + 1]);;
49          A[(-2*t1+3*t2-1)] = 0.33333 * (B[(-2*t1+3*t2-1)-1] +
            ↪  B[(-2*t1+3*t2-1)] + B[(-2*t1+3*t2-1) + 1]);;
50        }
51        if ((2*t1+_PB_N+2)%3 == 0) {
52          A[(_PB_N-2)] = 0.33333 * (B[(_PB_N-2)-1] + B[(_PB_N-2)]
            ↪  + B[(_PB_N-2) + 1]);;
53        }
54      }
55    }
56    for
      ↪  (t1=max(_PB_N-1,3*_PB_TSTEPS-1);t1<=3*_PB_TSTEPS+_PB_N-5;t1++)
      ↪  {
57      lbp=t1-_PB_TSTEPS+1;
58      ubp=floord(2*t1+_PB_N-2,3);
59  #pragma omp parallel for private(lbv,ubv,t3)
60      for (t2=lbp;t2<=ubp;t2++) {
61        B[(-2*t1+3*t2)] = 0.33333 * (A[(-2*t1+3*t2)-1] +
          ↪  A[(-2*t1+3*t2)] + A[(-2*t1+3*t2) + 1]);;
```

```
62        A[(-2*t1+3*t2-1)] = 0.33333 * (B[(-2*t1+3*t2-1)-1] +
          ↪  B[(-2*t1+3*t2-1)] + B[(-2*t1+3*t2-1) + 1]);;
63      }
64      if ((2*t1+_PB_N+2)%3 == 0) {
65        A[(_PB_N-2)] = 0.33333 * (B[(_PB_N-2)-1] + B[(_PB_N-2)] +
          ↪  B[(_PB_N-2) + 1]);;
66      }
67    }
68    A[(_PB_N-2)] = 0.33333 * (B[(_PB_N-2)-1] + B[(_PB_N-2)] +
      ↪  B[(_PB_N-2) + 1]);;
69  }
```

Annexe B

Résumé en français

B.1 Introduction

En raison des difficultés technologiques limitant l'augmentation des performances des unités de calcul séquentiel, l'informatique s'est progressivement tournée vers toujours plus de parallélisme. Plusieurs types de parallélisme ont vu le jour : pipelines d'éxécution, instructions vectorielles, threads (fils d'exécution) ou processus multiples sur processeurs multi-coeurs, architectures multi-processeurs à mémoire distribuée et accélérateurs de calcul. Exploiter au mieux ces types de systèmes nécessite actuellement une connaissance approfondie de plusieurs domaines (techniques d'optimisation, architecture, parallélisme) et est par conséquent l'apanage d'une poignée d'experts. Pour y pallier, il existe des outils d'optimisation et de parallélisation automatiques.

Certains de ces outils reposent sur le modèle polyédrique. Il s'agit d'un modèle mathématique permettant de représenter et manipuler sous forme de polyèdres des nids de boucles affines. Son utilisation en compilation du modèle polyédrique peut se décomposer en trois phases : modélisation, transformation et génération de code. Dans un premier temps, le code source est analysé afin d'en extraire une représentation mathématique sous forme de polyèdres. Ces polyèdres peuvent ensuite être manipulés et transformés en utilisant les techniques de l'algèbre linéaire selon l'optimisation ciblée. Enfin, la génération de code consiste à produire un code qui parcourt ces polyèdres.

Une grande partie des travaux actuels dans le domaine du modèle polyédrique porte sur les transformations à appliquer et les optimisations ainsi permises. À l'inverse, la génération de code est délaissée car la littérature correspondante offre des algorithmes considérés comment étant suffisamment performants. Nous montrons dans cette thèse non seulement qu'il subsiste une nette marge d'amélioration vis-à-vis de l'état de l'art mais que, par ailleurs, les défis à relever par la phase de génération de code mériteraient d'être pris en considération lors de la phase de transformation et d'optimisation.

Nous expliquons en Section B.2 comment le coût du contrôle dû au parcours de plusieurs polyèdres dans le code généré par un compilateur polyédrique peut être diminué [93]. Dans le cas de codes parallèles, les synchronisations peuvent limiter drastiquement la performance d'un programme. Nous donnons en Section B.3 la méthode permettant de détecter et éliminer les barrières de synchronisation superflues [91]. Enfin, la Section B.4 aborde le *pipelined multithreading* [92], une transformation qui introduit du parallélisme sur une classe de programmes jusqu'à présent ignorée par les paralléliseurs polyédriques.

B.2 Réduction du coût du contrôle

L'étape de génération de code dans un compilateur polyédrique prend en entrée une union de polyèdres et produit un code qui parcourt les points entiers de cette union. De nombreux travaux ont été menés afin d'améliorer le code produit. Néanmoins, le coût du contrôle reste élevé dans certains cas.

Il est possible de représenter un même domaine par plusieurs unions distinctes — mais équivalentes — de polyèdres. Les outils d'optimisation automatique n'y prêtent pas attention et laissent le choix de l'union retenue aux bibliothèques de manipulation de polyèdres sous-jacentes. Le parcours de certains polyèdres requiert du contrôle supplémentaire tel que des calculs de minimums ou maximums tandis qu'une union de polyèdres plus simple mais équivalente pourrait ne pas nécessiter de tels calculs.

La décomposition en chambres d'un polyèdre paramétré permet de calculer les domaines de validité et les sommets de ce polyèdre. Nous propo-

sons d'utiliser ce procédé afin de découper les domaines d'itération lors de la génération de code polyédrique. L'algorithme de génération de code étant un algorithme récursif sur les dimensions des polyèdres, à chaque étape, nous considérons temporairement les dimensions extérieures comme des paramètres afin de rechercher sa décomposition en chambres. Les chambres ainsi calculées sont alors utilisées pour découper le polyèdre de départ.

Les résultats expérimentaux montrent sur un grand nombre d'exemples la pertinence de cette approche : selon les conditions, jusqu'à 98% d'accélération et jusqu'à 10% en moyenne. Cependant, la baisse de performance observée (jusqu'à 70%) sur certains cas extrêmes suggère que le découpage de domaines ne peut être systématique. La nature des paramètres permettant de prendre cette décision restent un problème ouvert.

B.3 Réduction des synchronisations

Les paralléliseurs polyédriques automatiques génèrent du code annoté à l'aide d'instructions de parallélisation. L'outil considéré comme l'état de l'art se contente d'ajouter des annotations autour des boucles identifiées comme étant parallèles et ne tire pas parti de l'ensemble des instructions de parallélisation disponibles. Ce choix d'annotations isole chaque boucle parallèle dans une région parallèle distincte. En conséquence, des threads doivent être créés au début de chaque boucle parallèle et chaque boucle parallèle est terminée par une barrière de synchronisation.

Nous proposons d'utiliser des structures de contrôle de parallélisation disponibles afin de créer une unique région parallèle pour l'ensemble du code généré. Le désavantage de cette transformation est que les parties séquentielles doivent désormais être explicitement exclues du parallélisme. Mais l'avantage et le but de cette transformation est d'éliminer les synchronisations superflues (ce qui était impossible lorsque les régions parallèles étaient séparées). Le modèle polyédrique nous permet d'analyser les dépendances entre les différentes boucles parallèles. Cette analyse nous permet d'identifier les zones du code ne nécessitant pas de synchronisation.

D'après les expériences, et comme prévu, l'impact de cette transformation est plus élevé sur des programmes où le temps de synchronisation consti-

tue une part non négligeable du temps total d'exécution. À l'inverse, les effets sont de moins en moins marqués lorsque les temps de calculs augmentent et progressivement masquent les temps de synchronisation. Néanmoins, notre méthode n'introduit jamais de ralentissement et il est donc raisonnable de toujours l'appliquer en complément de la phase de parallélisation automatique.

B.4 *Pipelines* multi-fils d'exécutions

Les outils de parallélisation automatique polyédrique ciblent essentiellement des boucles parallélisables de manière isolée. Une boucle est annotée avec des instructions de parallélisation lorsque l'exécution en parallèle de toutes les itérations de cette boucle est valide. De même, en ce qui concerne le parallélisme, les transformations appliquées à une ou plusieurs boucles vise à obtenir des boucles dont les itérations sont indépendantes.

À l'inverse, une boucle n'est pas parallélisée dès lors que certaines de ses itérations doivent être exécutées séquentiellement. Cependant, une autre forme de parallélisme peut parfois être extrait : le parallélisme de *pipeline*. Nous proposons de cibler les boucles restées séquentielles après la passe de parallélisation automatique.

Notre technique de pipelining s'applique sur des groupes de boucles séquentielles successives. La première étape est, après l'analyse du graphe de dépendances, d'identifier les composantes fortement connexes afin de les séparer en boucles distinctes. Une autre analyse permet ensuite de determiner si des boucles successives peuvent être exécutées en pipeline. Si tel est le cas, nous utilisons des annotations de parallélisation pour indiquer que chaque boucle doit être découpée en blocks (*chunks*), que ces *chunks* doivent être exécutés dans l'ordre et que chaque fil d'exécution peut passer à la boucle suivante dès lors que son *chunk* est terminé.

Cette transformation peut ensuite être raffinée en utilisant des synchronisations explicites : nous découpons les boucles en *chunks* de notre choix et les pipelines progressent lorsque chaque fil d'exécution effectue une synchronisation à la fin d'un chunk.

Nos résultats expérimentaux montrent que cette méthode est d'autant

plus bénéfique que le nombre de boucles susceptibles d'être affectées augmente. À défaut d'être bénéfique dans des cas plus simples, cette transformation n'est pas négative non plus. Nous n'avons pas observé de contre-indication à une application systématique de notre technique.

B.5 Conclusion

Au cours de cette thèse nous avons montré que la génération de code polyédrique efficace est une tâche ardue. Nous avons présenté des méthodes permettant de réduire le coût du contrôle, d'éliminer les synchronisations superflues et d'introduire plus de parallélisme. Bien que les résultats soient encourageants, notre technique de réduction du coût de contrôle offre des performances mitigées : de futurs travaux devront déterminer dans quelle mesure appliquer le découpage de domaines. La réduction des synchronisations et l'introduction du *pipelined multithreading* sont deux opérations qui peuvent s'appliquer *après* une phase de génération de code classique. Il n'y a aucune contre indication à l'application de ces passes. Il s'agit donc en l'état de méthodes complémentaires aux techniques d'optimisation polyédrique existantes. Néanmoins, nos observations montrent que ces méthodes seraient plus efficaces si ces possiblités étaient prises en compte lors des phases précédentes d'optimisation. De futures améliorations des techniques d'ordonnancement polyédrique pourraient explicitement cibler des boucles sans synchronisation ou des pipelines de boucles.

Génération de code polyédrique efficace : vers plus de parallélisme et moins de contrôle

Harenome RAZANAJATO

Abstract

This thesis proposes new extensions to the code generation phase in polyhedral compilers. The main focus of recent work on polyhedral compilation focus is the optimizations leveraged by polyhedral transformations while state-of-the-art code generation algorithms are considered satisfactory. We show that state-of-the-art polyhedral code generation can still be further improved.

We explain how splitting polyhedra can reduce the control overhead introduced by polyhedra scanning in the code generated by a polyhedral compiler. Synchronizations in parallel code can drastically impede a program's performance. We propose a method to detect and lift unnecessary synchronization barriers. Finally, we introduce *pipelined multithreading*, a transformation that introduces parallelism in a class of programs that was, until now, ignored by polyhedral parallelizers.

Résumé

Cette thèse propose de nouvelles extensions à la phase de génération de code dans les compilateurs polyédriques. Une grande partie des travaux actuels dans le domaine du modèle polyédrique porte sur les transformations à appliquer et les optimisations ainsi permises. À l'inverse, la génération de code est délaissée car la littérature correspondante offre des algorithmes considérés comment étant suffisamment performants. Nous montrons dans cette thèse qu'il subsiste une nette marge d'amélioration vis-à-vis de l'état de l'art.

Nous expliquons comment le coût du contrôle dû au parcours de plusieurs polyèdres dans le code généré par un compilateur polyédrique peut être diminué. Dans le cas de codes parallèles, les synchronisations peuvent limiter drastiquement la performance d'un programme. Nous donnons la méthode permettant de détecter et éliminer les barrières de synchronisation superflues. Enfin, nous proposons le *pipelined multithreading*, une transformation qui introduit du parallélisme sur une classe de programmes jusqu'à présent ignorée par les paralléliseurs polyédriques.

Équipe ICPS
Université de Strasbourg