12-16-2022 11:30 AM

# Advances in the Automatic Detection of Optimization Opportunities in Computer Programs

Delaram Talaashrafi, *The University of Western Ontario*

Supervisor: Moreno Maza, Marc, *The University of Western Ontario*
A thesis submitted in partial fulfillment of the requirements for the Doctor of Philosophy degree in Computer Science

## Recommended Citation

Talaashrafi, Delaram, "Advances in the Automatic Detection of Optimization Opportunities in Computer Programs" (2022). *Electronic Thesis and Dissertation Repository*. 9054.
https://ir.lib.uwo.ca/etd/9054

# Abstract

Massively parallel and heterogeneous systems together with their APIs have been used for various applications. To achieve high-performance software, the programmer should develop optimized algorithms to maximize the system's resource utilization. However, designing such algorithms is challenging and time-consuming. Therefore, *optimizing compilers* are developed to take part in the programmer's optimization burden.

Developing effective optimizing compilers is an active area of research. Specifically, because loop nests are usually the hot spots in a program, their optimization has been the main subject of many optimization algorithms.

This thesis aims to improve the *scope* and *applicability* of performance optimization algorithms used in the compiler optimization phase. In the first two chapters, we focus on the parts of the programs with for-loop nests. We take advantage of the polyhedral model and the scalar evolution to develop algorithms that can automatically discover new optimization opportunities in computer programs. Our functions operate at the *intermediate representation* level and are implemented as part of the LLVM infrastructure. In the final chapter, we improve the performance of the Fourier-Motzkin elimination method, which is an underlying algorithm in the polyhedral theory.

**Keywords:** compiler optimization, LLVM, polyhedral model, pipelining, scalar evolution, OpenMP, target offloading, polyhedral theory, Fourier-Motzkin elimination, bit complexity

# Summary

Massively parallel and heterogeneous systems have been used for various applications. To achieve high-performance software, the programmer should develop optimized algorithms in order to maximize the system's resource utilization. However, designing such algorithms is challenging and time-consuming. Therefore, optimizing compilers are developed to take part in the programmer's optimization burden. Developing effective optimizing compilers is an active area of research. Specifically, because loop nests are usually the hot spots in a program, their optimization has been the main subject of many optimization algorithms. This thesis aims to improve the scope and applicability of performance optimization algorithms used in the compiler optimization phase.

# Co-Authorship Statement

- **Chapter 3** is published in [1]. It is joint work with Johannes Doerfert and Marc Moreno Maza. The thesis author's contributions include designing the underlying algorithm for pipeline pattern detection, implementing the algorithm in LLVM/Polly framework, and collecting experimental results.
- **Chapter 4** is published in [2]. It is joint work with Marc Moreno Maza and Johannes Doerfert. The thesis author's contributions include designing the underlying algorithm for finding memory locations and prefetching them, implementing the algorithm as an Inter-procedural pass in the LLVM framework, and collecting experimental results.
- **Chapter 5** is published in [3]. It is joint work with Rui-Juan Jing and Marc Moreno Maza. The thesis author's contributions include improving and modifying the algorithm proposed by Balas to efficiently remove redundant inequalities generated in the process of Fourier-Motzkin elimination, implementing the algorithm in the C programming language, and collecting experimental results.

# Acknowledgement

Firstly, I would like to sincerely thank my supervisor Professor Marc Moreno Maza for giving me the opportunity to find and follow the subject that I am truly interested in. Moreover, I want to thank my examiners, Dr. Taylor Brysiewicz, Dr. Ettore Tiotto, Dr. Mike Domaratzki, and Dr. Hanan Lutfiyya for taking the time out of their busy schedules to read my thesis, and also for their insightful questions and comments.

I would like to thank my co-author and LLVM and OpenMP mentor Johannes Doerfert for the many things I have learned from our joint works and the different ideas we discussed. Our collaboration has had an incredible impact on my professional path.

Last but not least, I would like to thank my family for their infinite love, support, and constant encouragement toward taking significant steps and moving forward. Specifically, I would like to thank my *Mom* for being my role model in kindness and elegance; I would like to thank my *Dad* for teaching me the value of hard work toward high-quality results; and I would like to thank my *Sister*, whose artistic mind has always been a delight.

*Dedicated to my beloved Family.*

# Contents

# List of Figures

# List of Tables

Chapter 1

# Introduction

Developing high-performance programs has always been an essential aspect of computer science. However, in the past decade, it has become even more crucial. As a result of Moore's law slowing down [4], programmers cannot rely on the increasing computational power of one processor to improve the program's performance. On the other hand, various applications need to process an increasing amount of data in limited time frames, which cannot be achieved by the maximum possible performance of single-threaded systems. To tackle this issue, *massively parallel and heterogeneous systems*, such as multi-core CPUs and GPUs, have emerged. These systems have different architectures, and utilizing their capabilities requires various programming models. As a result, different Application Programming Interfaces (APIs) have been developed to facilitate the programming of these systems. For instance, OpenMP [5, 6], Massage Passing Interface (MPI) [7], and Compute Unified Device Architecture (CUDA) [8] to name a few.

The parallel and heterogeneous systems, along with their programming APIs, increase the potential computational power. However, for achieving high-performance software, the programmer should develop optimized algorithms to maximize the system's resource utilization. Designing such algorithms can be very challenging and time-consuming because the programmer should explicitly specify parallelism while considering hardware properties, such as its execution model and memory hierarchy. Therefore, *optimizing compilers* have been developed to take part in the programmer's optimization burden. In modern optimizing compilers, the program first goes through the compiler's front-end. The output of the front-end is the program in its machine-independent, intermediate representation (IR). After that, the middle-end applies optimization to the IR. Finally, the back-end takes the IR, applies some machine-dependent optimizations, and generates machine-dependent code.

A modern compiler infrastructure, designed for developing optimizing compilers is LLVM [9]. It is an open-source and industry standard compiler infrastructure, developed in `C++`. It translates source programs to its own intermediate representation (LLVM-IR), which is independent from the source language and the target architecture. LLVM also provides libraries and tools for manipulating and optimizing LLVM-IR. It also provides tools for translating the IR to target-dependent machine code. Moreover, LLVM provides procedures for utilizing and also developing Link Time Optimization (LTO) and Profile Guided Optimization (PGO). In summary, we can say that LLVM provides a framework and an infrastructure to easily integrate different optimization functions into the compilation pipeline.

Developing optimizing compilers is an active area of research. The main goal of research and development in this field is to equip different stages of the compilation pipeline with various functions that can automatically transform the program to a more optimized version, considering various criteria. The focus of this thesis is on the methods developed in the middle-end of the pipeline.

Researchers have taken different approaches to developing optimizing compilers, by using various methods to detect optimization opportunities on different levels of abstraction. A promising method for designing performance optimization algorithms is to consider the intermediate representation of the program. Then, describe parts of the program with appropriate mathematical objects. Having this representation, the program transformation can be done by manipulating the underlying object. For instance, graphs and trees are used to represent the control flow and the domination relation between different basic blocks in the program, respectively.

Loop nests are usually the hot spots in a program, and their optimization has been the main subject of many optimization algorithms. Different models have been proposed to model their attributes and facilitate optimizing them. The *polyhedral model* and *scalar evolution* are two effective methods for this goal, which we use in this thesis.

The *polyhedral model* [10, 11] has proved to be adequate for describing and optimizing for-loop nests in a program. In this technique, iterations of a for-loop nest are modeled as integer points inside a polyhedron. By using algorithms in the polyhedral theory, the methods based on the polyhedral model apply transformations to the programs in order to get more optimized versions. For instance, they can improve the program's performance by optimizing memory usage and parallelizing the loops in for-loop nests, when it is possible.

Another optimization technique related to the for-loop nests is the *scalar evolution* [12] method. This method tries to compute the values of the scalars used in a for-loop nest as functions of the loop nest's induction variables. Then, these functions can be used for various optimizations.

This thesis aims to improve the *scope* and *applicability* of performance optimization algorithms used in the compiler optimization phase. In Chapters 3 and 4, we focus on the parts of the programs with for-loop nests. We take advantage of the polyhedral model and the scalar evolution to develop algorithms that can automatically discover new optimization opportunities in the programs. Our functions operate at the Intermediate Representation level and are implemented as part of the LLVM infrastructure. In Chapter 5, we improve the performance of the Fourier-Motzkin elimination method, which is an underlying algorithm in the polyhedral theory.

## 1.1 Contributions

In this section, we provide a short summary and a list of our objectives for each research project presented in this thesis.

### 1.1.1 A Pipeline Detection Technique in Polly

In Chapter 3, we introduce a polyhedral-model-based analysis and scheduling algorithm that exposes and utilizes cross-loop parallelization through tasking. This work exploits pipeline patterns between iterations in different loop nests, and it is well suited to handle imbalanced iterations.

The current polyhedral-model-based approaches can optimize a program by loop tiling, loop parallelizing, and software pipelining [13, 14, 15]. Also, some works have been done to detect pipeline pattern in the program [16, 17], but they have not been very effective for arbitrary array access functions. In this project, we design a transformation algorithm for detecting the pipeline parallelism between iteration blocks of different loop nests in the program. We implement a prototype of the algorithm in the LLVM/Polly [18] pipeline in a way that it can be applied to the programs automatically, and without programmer's intervention. We also design experiments to test the effectiveness of our prototype. For future works, we plan to increase the applicability of the method by improving the code generation phase, and make the pipeline detection to be compatible with other optimization pattern detection techniques.

### 1.1.2 Automatic OpenMP-Aware Utilization of Fast GPU Memory

In Chapter 4, we develop an inter-procedural LLVM transformation to improve the performance of OpenMP target regions by optimizing memory transactions. This transformation pass effectively prefetches some of the read-only input data to the *fast* shared memory via compile time code injection. Especially if there is reuse, accesses to shared memory far outpace global memory accesses. Consequently, our method can significantly improve performance if the right data is placed in shared memory.

Different methods have been developed for optimizing OpenMP program offloads to GPUs [19]. In this project, we design an algorithm to map the locations in the GPU's global memory to the locations in the shared memory, as well as some heuristics to avoid bank conflict and improve space efficiency. We leverage the infrastructure developed in [19] to implement our algorithm as a part of the OpenMP optimization passes, inside LLVM. For future works, we plan to add support for more general kernels, and develop a method to find the array whose prefteching improves the performance the most.

### 1.1.3 Complexity Estimates of Fourier-Motzkin Elimination

In Chapter 5, we propose an efficient method for removing all redundant inequalities generated by Fourier-Motzkin elimination. This method is based on an improved version of Balas' work and can also be used to remove all redundant inequalities in the input system. Moreover, our method only uses arithmetic operations on matrices and avoids resorting to linear programming techniques. Algebraic complexity estimates and experimental results show that our method outperforms alternative approaches, in particular those based on linear programming and the simplex algorithm.

Fourier-Motzkin elimination is a fundamental operation in the polyhedral theory and computation. It is an exponential algorithm [20], however its performance can be improved by removing redundant inequalities that are generated in the process of the algorithm. Researchers have followed different approaches to detect and remove these redundancies. Some methods are based on linear programming and can remove *all* redundancies [21], some other methods use only matrix operations, but cannot detect all redundancies [22, 23, 24]. In this project, we design and implement a method based on the algorithm proposed by Balas [25] that uses linear algebra algorithms to detect *all* redundant inequalities. We compare the performance of our algorithm with linear programming-based methods by comparing the running times of some experimental linear inequality systems. We also compare the exact bit-complexity of the algorithms. For future works, we plan to develop a heuristic method to use the properties of the input inequality system and choose the best projection method based on that.

# Chapter 2

# Background

In this chapter, we provide some background information used in the rest of this thesis. First, after explaining some basic concepts of the polyhedral theory in Section 2.1, we go through the details of the polyhedral model in Section 2.2. Then, in Section 2.3, we explain the LLVM compiler infrastructure, followed by detailed explanation of Polly, a sub-project of LLVM for applying polyhedral transformations, and scalar evolution, a program analyzing method in LLVM, in Sections 2.3.3 and 2.3.4, respectively. Finally, we conclude this chapter by explaining some related features of OpenMP and its compilation and optimization process in Section 2.4.

## 2.1 Polyhedral theory

In this section, we review some basic concepts of the polyhedral theory. We go through definitions and related theorems of the polyhedral cone, polyhedron, and $\mathbb{Z}$-polyhedron. These concepts are expanded to more advanced topics in Section 5.1.1.

**Definition 2.1.1.** A set of vectors $P \subseteq \mathbb{Q}^n$ is called a *convex polyhedron* if $P = \{\vec{x} \mid A\vec{x} \leq \vec{b}\}$, for a matrix $A \in \mathbb{Q}^{m \times n}$ and a vector $\vec{b} \in \mathbb{Q}^m$. Moreover, the polyhedron $P$ is called a *polytope* if $P$ is bounded. From now on, we always use the notation $P = \{\vec{x} \mid A\vec{x} \leq \vec{b}\}$ to represent a polyhedron in $\mathbb{Q}^n$. We call the system of linear inequalities $\{A\vec{x} \leq \vec{b}\}$ a *representation* of $P$.

**Example** A *cube* is a convex polyhedron in 3 dimensions.

**Definition 2.1.2.** We define a *d-dimensional lattice* to be a subset of $\mathbb{Z}^d$ that is defined as a integer linear combination of linearly independent integer vectors. With this definition, we define *$\mathbb{Z}$-polyhedron* (*$\mathbb{Z}$-polytope*) to be the intersection of a rational polyhedron (polytope) with a lattice defined in the same space. In other words, it is the subset of a polyhedron, containing all integer points inside the polyhedron.

**Example** Figure 2.1 shows a bounded polyhedron (polytope). The integer points inside it (the $\mathbb{Z}$-polytope) are also illustrated.

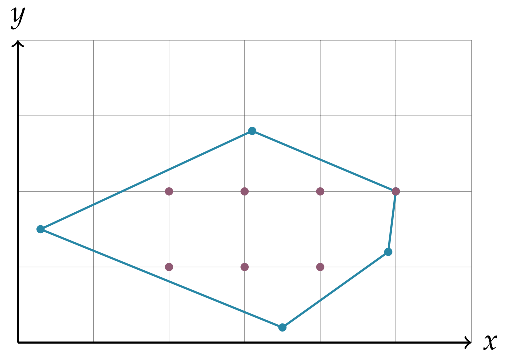


Figure 2.1: Example of a bounded polyhedron (polytope) and $\mathbb{Z}$-polytope

## 2.2 Polyhedral compilation

The *polyhedral model* [26, 27, 28, 29, 30] is a mathematical abstraction to describe for-loop nests execution in a program. The model provides an abstract representation of the program. This representation of for-loop nests enables us to analyze them and automatically optimize them via different transformations. Here by transformation, we mean a wide variety of program modifications that we can apply to the program for different kinds of optimization, including:

1- loop tiling (blocking) for more efficient cache usage [31];

2- loop parallelizing [32];

3- including vector instructions [33];

4- partitioning the code for heterogeneous execution [34].

In the rest of this section, we explain in more detail how we can represent and optimize a for-loop nest with the polyhedral model. Then, we describe the mathematical foundation of this modeling and the way it is implemented in the ISL library.

### 2.2.1 Polyhedral representation and transformation

The polyhedral model can handle a program's *Static Control Parts (SCoP)* [10]. They are the parts of the program for which we can determine the control flow and memory accesses at *compile time*, and all the involved expressions are *affine* in the loop induction variables and program parameters. Figure 2.2 shows an example of a SCoP.

More precisely, the polyhedral model can optimize parts of the program with the following constraints:

1- structured control flow, which includes for-loop nests with a lower-bound, an upper-bound, and an induction variable which increases or decreases by a constant number in each iteration, and conditional statements,

```
for(i=0; i<2*N+5; i++){
  for(j=0; j<=i; j++)
    A[i+j] = A[i];
}
```

Figure 2.2: An example of a SCoP.

2- the loop bounds, conditions, and memory access functions should be affine expressions in the induction variables and program parameters, and

3- side effect free (e.g., no use of pointers).

These conditions describe the constraints on the parts of a program supported by the original version of the polyhedral model. Further research articles, for example [30][35], have proposed different methods to lift some of these constraints.

After finding program parts the polyhedral model can potentially optimize, we should see how the model describes, analyzes, and transforms them. The polyhedral model uses four main components to represent a SCoP: 1) iteration domain, 2) access relations, 3) schedule, and 4) dependency relations. In the rest of this part, we explain each of these components in more details.

An essential aspect of the polyhedral model is that it represents programs and manipulates them based on the *statement instances*. Outside of any loops, statement instances are similar to the statements. Inside a for-loop nest, a statement instance is a statement with fixed values of its surrounding loops' induction variables. For example, in Figure 2.2, `A[i + j] = A[i]` is a statement. We can get the lexicographically smallest instance of it by setting `i = 0`, `j = 0`. Therefore, the first instance of this statement that executes is `A[0] = A[0]`.

Instead of listing all the points, we represent them by the constraints on their values. These constraints come from the initial values of the induction variables of the loops surrounding the statement and loops' conditions. As a result of our assumption that the loop bounds are affine expressions, these constraints form a system of linear inequalities. For instance, in Figure 2.2, we can describe all statement instances with $\{(i, j) \in \mathbb{Z}^2 \mid 0 \leq i < 2N+5,\ 0 \leq j < i\}$. These inequality systems form a *parametric polyhedron*, and are called *iteration domains*. The statement instances are the integer points in the iteration domain polyhedrons.

Furthermore, each element of the iteration domain (each statement instance) accesses some array elements. These accesses can either be *read* or *write*. The index of these elements are called the *access relations*. For example, in Figure 2.2, each element `[i, j]` of the iteration domain, reads the location `i` of the array `A`, and writes in the location `i + j` of the same array.

Note that the iteration domain only represents statement instances and does not provide any information on their execution order. To specify this order, the polyhedral model uses the notion of *schedule*s. A schedule is a relation that maps each statement instance to an integer vector. Then, the lexicographical order of these vectors determines the execution order of instances.

The polyhedral model can optimize programs by changing the schedule assigned to each statement. However, we cannot assign *any* schedule to a loop nest. In fact, the new schedule should not change the semantics of the loops. We call them *valid schedules*. This happens because of the *dependencies* in the program. If there is an instance of an statement that uses the result of the execution of another instance, then the instance that produces data must execute before the instance that uses that data. All the valid schedules keep the original order between dependent instances.

### 2.2.2 Mathematical foundations of the polyhedral model in the ISL library

As explained in Section 2.2.1, the polyhedral model is based on manipulating integer points of parametric polyhedrons.

Various libraries such as ISL [36, 37, 38], Polylib [39], Piplib [40], and Omegalib [41] implement the polyhedral model's underlying mathematical operations. For the purpose of this thesis, we use the ISL library.

The ISL library uses em set and *map* data structures to represent parametric $\mathbb{Z}-$polyhedrons. More accurately, the ISL library represents the integer points of a parametric polyhedron (parametric $\mathbb{Z}-$polyhedron) as a family of *sets* of integer tuples. Equation (2.1) shows ISL's representation of a parametric two-dimensional $\mathbb{Z}-$polytope, where the values of `i` and `j` are bounded between `0` and the parameter `n`.

```
Set("[n]->{[i,j] :  0 < i <= n and 0 < j <= n }")
```
(2.1)

Different mathematical operations including intersection and union are also defined for the set object in the ISL library.

In abstract algebra, A *map* or *binary relation* is a subset of the Cartesian product of two sets. Different operations are defined on maps in the ISL library. Let $M$ be a map. The *inverse map* of $M$, denoted by $M^{-1}$, is the set of the pairs $(\vec{j}, \vec{i})$ such that $(\vec{i}, \vec{j}) \in M$. The domain (resp. range) of $M$ denoted by $\mathsf{Dom}(M)$ (resp. $\mathsf{Range}(M)$) is the set of all $\vec{i}$ (resp. $\vec{j}$) such that $(\vec{i}, \vec{j}) \in M$.

We denote by $\mathsf{lexmax}(M)$ (resp. $\mathsf{lexmin}(M)$) the subset of $M$ consisting of all pairs $(\vec{i}, \vec{j})$, where $\vec{i} \in \mathsf{Dom}(M)$ and $\vec{j}$ is the lexicographically largest (resp. smallest) $\vec{k} \in \mathsf{Range}(M)$ such that $(\vec{i}, \vec{k}) \in M$.

The *composition* of two maps $M_1$ and $M_2$ is denoted by $M_1(M_2)$. It is the set of all pairs $(\vec{i}, \vec{j})$, such that there exists a vector $\vec{k}$, where $(\vec{i}, \vec{k}) \in M_2$ and $(\vec{k}, \vec{j}) \in M_1$.

One construction of maps, important to the polyhedron model, is as follows. Given two subsets $S_1$ and $S_2$ of the same totally ordered set $S$, we denote by $\mathsf{lexleset}(\mathsf{S}_1, \mathsf{S}_2)$ the set of all $(\vec{i}, \vec{j}) \in S_1 \times S_2$ where $\vec{i}$ is lexicographically less or equal to $\vec{j}$.

Returning to the ISL library, and similarly to sets, maps can also be defined with parameters. In this case, fixed values of those parameters give rise to a unique map in the sense of abstract algebra. For example, Equation (2.2) is the representation of a map in ISL, with `n` as parameter.

```
Map("[n]->{[i,j] -> [2+i, -1+j] :  0 < i <= n and 0 < j <= n}")     (2.2)
```

Note that ISL stores and uses sets and maps by storing the constraints on the values of each coordinate, where the constraints are computed by algorithms in (integer) polyhedral geometry.

Schedules are represented in the form of a tree, called *schedule tree* in the ISL library. Nodes in a schedule tree have different types for representing different execution orders. The most important ones that we are using in this article are: domain node, band node, sequence node, mark node, and expansion node. More detailed information on the tree representation of the schedules can be found in [37, 38].

## 2.3 LLVM

LLVM [42, 43] is a compiler infrastructure written in `C++`. It is an open source, industry standard, and evolving project. The updated version of LLVM project mono-repository can be accessed from `https://github.com/llvm/llvm-project`. LLVM is an umbrella project and many different projects are included in there. Some of these projects are:

- LLVM-core: the main part of LLVM project.
- `Clang` [44, 45]: `Clang` is a `C` and `C++` front-end of LLVM.
- `Flang` [46]: `Flang` is a `Fortran` front-end of LLVM.
- `MLIR` [47, 48] : `MLIR` is another compiler infrastructure with multi-level intermediate representations, designed to address the challenges that LLVM cannot solve.
- Polly [18]: Polly is for applying polyhedral transformations to the program.
- Many other LLVM-based tools such as debugger, linker, ... .

We will explain some of these projects in more details in the rest of this thesis.

### 2.3.1 LLVM intermediate representation

LLVM Intermediate Representation (LLVM-IR) [49] is a Static Single Assignment (SSA) [50] representation of a program designed to be able to represent all high-level languages. It is a low-level programming language with RISC-like instruction set, and it is independent from the source language and the target architecture.

It is used through the LLVM framework, and its powerful representation allows the infrastructure to implement efficient compiler analysis and transformations. Figure 2.3 shows a sample of a function written in LLVM-IR.

```
define i32 @bar() {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 5, i32* %1, align 4
  store i32 7, i32* %2, align 4
  store i32 8, i32* %3, align 4
  %5 = load i32, i32* %1, align 4
  %6 = load i32, i32* %3, align 4
  %7 = add nsw i32 %5, %6
  store i32 %7, i32* %4, align 4
  %8 = load i32, i32* %4, align 4
  ret i32 %8
}
```

Figure 2.3: A sample of LLVM-IR.

### 2.3.2 LLVM compilation and optimization pipeline

In this part, we explain what happens when a `C` or `C++` file enters the LLVM pipeline and the different stages the file goes through.

In the first step, `Clang` processes the file and performs the compiler's lexical analysis and parsing stages . The output of this stage is the corresponding Abstract Syntax Tree (AST) [51] of the program. In the next step, `Clang` generates LLVM-IR from the AST generated in the previous step. The `opt` tool can optimize the LLVM-IR generated at this stage. After that, the `llc` tool generates LLVM Machine IR (MIR), optimizes it, and finally generates machine code.

The *LLVM pass framework* provides a systematic way to analyze programs in LLVM-IR and also to apply different transformations on the program. This framework supports two kinds of passes:

1- *Analysis passes*, which go through the IR and gather some information, but does not modify the program. Alias Analysis [52] and Loop Info [53] passes are examples of analysis passes in LLVM.

2- *Transformation passes*, usually use the information provided by one or more of the analysis passes and modify the IR for different purposes. Function in-lining and loop unrolling are examples of transformation passes in LLVM.

The code in Figure 2.4 illustrates the same function after optimization.

```
define i32 @bar() {
  ret i32 13
}
```

Figure 2.4: Optimized version of the program in Figure 2.3.

Our focus in this thesis is on optimizing the program in the LLVM-IR level. The `opt` tool gets a program in LLVM-IR, applies different passes on it, and generates another LLVM-IR program.

### 2.3.3 Polly

Many tools have been developed to analyze and optimize programs using the polyhedral model. Polly [18] is one of the projects in the LLVM infrastructure for applying polyhedral transformations on the LLVM-IR programs.

In the first step, Polly analyzes a LLVM-IR program and finds its polyhedral representation. To this end, it first detects SCoP parts of the program. Then, it finds the polyhedral description of the SCoPs in terms of the iteration domains, schedule, and memory accesses. Figure 2.5 shows the result of this analysis for the program in Figure 2.2.

In Polly, transforming a SCoP is equivalent to changing its schedule. Therefore, in the next step Polly uses the ISL library optimizer and composes new schedules for the statements in the SCoPs. The algorithm used in the ISL library optimizer is based on the Pluto algorithm. There is also the option for importing new schedules to the Polly's pipeline. Moreover, we can implement new transformation and scheduling algorithms as part of Polly to improve its optimization capabilities.

The final step is to transform the polyhedral representation back to the LLMV-IR. After optimizing the program by changing the schedule, Polly creates a new schedule tree object based on the paper [37], and uses that to create an updated AST. Then. Polly uses the new AST to generate LLVM-IR code. If required, Polly can detect parallel loops. In this case, it uses OpenMP as its back-end for exploiting the detected parallelism. In the code generation phase, it outlines [1] the parallel loops and call the outline function through OpenMP runtime calls.

### 2.3.4 Scalar evolution

*Scalar evolution* [54, 55, 12] is one of the fundamental analysis passes of LLVM. The main purpose of the scalar evolution (`scev`) pass is to study the changes of a scalar over iterations

[1]Outlining is a method to extract a region or a sequence of instructions of the code, place it in a function, and replace the region with a call to the created function with appropriate input arguments.

```
Function: main
Region: %for.cond4.preheader---%for.end14
Max Loop Depth:  2
Invariant Accesses: {
}
Context:
[N] -> {  : -2147483648 <= N <= 2147483647 }
Assumed Context:
[N] -> {  :  }
Invalid Context:
[N] -> {  : N <= -1073741825 or N >= 1073741822 }
Defined Behavior Context:
[N] -> {  : -1073741824 <= N <= 1073741821 }
p0: %call
Arrays {
    i32 A[*]; // Element size 4
}
Arrays (Bounds as pw_affs) {
    i32 A[*]; // Element size 4
}
Alias Groups (0):
    n/a
Statements {
  Stmt1
        Domain :=
            [N] -> { Stmt1[i0, i1] : i0 <= 4 + 2N and 0 <= i1 <= i0;
                     Stmt1[0, 0] : N <= -3 };
        Schedule :=
            [N] -> { Stmt1[i0, i1] -> [i0, i1] : i0 <= 4 + 2N;
                     Stmt1[0, 0] -> [0, 0] : N <= -3 };
        ReadAccess := [Reduction Type: NONE] [Scalar: 0]
            [N] -> { Stmt1[i0, i1] -> A[i0] };
        MustWriteAccess :=  [Reduction Type: NONE] [Scalar: 0]
            [N] -> { Stmt1[i0, i1] -> A[i0 + i1] };
}
```

Figure 2.5: Representation of SCoP of the code in Figure 2.2 in Polly.

of a loop. This pass is based on the mathematic concept of *chains of recurrences* [56, 57]. In fact, it is an implementation of this method to be specifically used for analyzing for-loops. In this part, we first briefly explain the mathematical framework of scalar evolution. After that, we explain the scalar evolution implementation in LLVM.

### 2.3.5 Chain of recurrences

We begin by defining the concept of *basic recurrence (BR)*. This concept can be used to represent functions using recurrences.

Having a constant value $\alpha_0$, a function $f_1 : \mathbb{N} \rightarrow \mathbb{N}$, and an operator $\odot$ which is an addition or a multiplication, we represent a basic recurrence $f$ as:

$$f = \{\alpha_0, \odot, f_1\}.$$

If the operator ($\odot$) is an addition, we define the basic recurrence as a function over the natural numbers as:

$$f = \{\alpha_0, \odot, f_1\}(i) = \alpha_0 + \sum_{j=0}^{i-1} f_1(j).$$

If the operator ($\odot$) is a multiplication, we define it as a function over the natural numbers as:

$$f = \{\alpha_0, \odot, f_1\}(i) = \alpha_0 \prod_{j=0}^{i-1} f_1(j).$$

An important point is that $f$ can be defined recursively by the $f(i) = f(i-1) \odot f_1(i)$ relation.

We can apply the concept of basic recursions to analyze scalars value changes in iterations of a loop. For instance, consider the loop in Figure 2.6.

```
int a = M;
for(int i=0; i<n; i++){
   a = a + k;
}
```

Figure 2.6: An example of a loop for showing basic recursion used for analyzing scalar changes.

We define function $f_a : \mathbb{N} \rightarrow \mathbb{N}$ to map each iteration number to the value of scalar a in that iteration. We can define $f_a$ as:

$$f_a = \begin{cases} M & i = 0 \\ f_a(i-1) + k & i > 0 \end{cases}$$

Additionally, we can represent it as basic recurrence tuple with $f_a(i) = \{M, +, k\}$.

We can extend the definition of basic recurrences and define the concept of *chain of recurrences*. Having $k$ constants $\alpha_0, \cdots, \alpha_{k-1}$, a function $f : \mathbb{N} \rightarrow \mathbb{N}$, and $k$ operators $\odot_0, \cdots, \odot_{k-1}$ that are either addition or multiplication, we represent the chain of recurrence $F$ as:

$$F = \{\alpha_0, \odot_0, \cdots, \odot_{k-1}, f\}.$$

It is defined recursively as:

$$F(i) = \{\alpha_0, \odot_0, \{\alpha_1, \odot_1, \cdots, \odot_{k-1}, f\}\}(i).$$

For example, consider the chain of recurrence $F = \{k_0, +, \{k_1, +, k_2\}\}$. We can consider $f_1$ as a BR represented as $f_1 = \{k_1, +, k_2\}$. Then, we can represent $F$ as a BR $\{k_0, +, f_1\}$. The BR $f_1(i)$ can be defined as:

$$f_1(i) = \begin{cases} k_1 & i = 0 \\ f_1(i-1) + k_2 & i > 0 \end{cases}$$

Using $f_1(i)$, we represent $F$ as:

$$F(i) = \begin{cases} k_0 & i = 0 \\ F(i-1) + f_1(i-1) & i > 0 \end{cases}$$

### 2.3.6 Scalar evolution pass in LLVM

As mentioned before, `scev` pass in LLVM uses a simplified implementation of chains of recurrences. The goal of this pass is to analyze the scalar values and expressions in loops and generate the `scev` representation (in the format of $\{a, \odot, b\}$) of them. Having expressions in this format helps other passes to apply different optimizations.

To generate `scev` representation of an expression, `scev` pass goes through the LLVM-IR instructions. It begins with generating `scev` representation of each variable in the expression. If the variable is loop invariant, the variable itself can be used as its `scev` representation. On the other hand, if the variable changes in the considered loop and its value is specified by a `phi` node, `scev` forms a CR for that.

After having the `scev` representation of all individual variables in the expression, `scev` uses algebraic *rewrite rules* to combine different CRs and to create the final CR of the expression. Table 2.1 shows these rewrite rules.

| Expression | Rewrite |
|---|---|
| $C + \{a, +, b\}$ | $\{C + a, +, b\}$ |
| $C * \{a, +, b\}$ | $\{C * a, +, C * b\}$ |
| $\{a, +, b\} + \{d, +, e\}$ | $\{a + d, +, b + e\}$ |
| $\{a, +, b\} * \{d, +, e\}$ | $\{a * d, +, a * e + b * d + b * e, +, 2 * b * e\}$ |

Table 2.1: Rewrite rules for `scev` expressions.

## 2.4 OpenMP

OpenMP is a programming API, designed for shared memory parallelization in `C`, `C++`, and `Fortran`. Programmers can specify and control parallelization using various *compiler directives*. OpenMP also consists of *runtime functions* and some *environment variables*.

The OpenMP execution model for parallelism targeting multi-core CPUs is the *fork-join model*. An OpenMP program starts in the serial mode and a single thread, the *master thread*, starts running the program. To write a parallel program with OpenMP, the programmer should specify the *parallel region* using the `pragma omp parallel` pragma. Once the master thread encounters the parallel region, the runtime system launches some additional threads, *worker threads*. Then, the master thread and worker threads run the parallel region with respect to the pattern and the constraints specified by the programmer, using different directives. An implicit barrier exists at the end of each parallel region and synchronization between threads happens at that point. After that point, the master thread continues the execution of the rest of the program.

OpenMP has support for two different kinds of memory: *private* and *shared*. Each thread has access to its own private memory. The programmer can define variable `x` in the private memory of each thread by adding `private(x)` to the parallel region pragma. Rather than the private memory space, all threads have access to the shared memory space. When a variable is defined in the shared space, all threads have access to a similar memory location for that variable.

As mentioned, a programmer can direct the API to parallelize a region with different methods using pragmas. The complete and growing list of these directives is accessible from the OpenMP manual. We explain the `task` and `target` directives that we employ in this thesis in detail in Sections 3.3 and 4.2.1, respectively. In the rest of this section, we present more information about the compilation and optimization of OpenMP programs in LLVM.

### 2.4.1 OpenMP compilation in LLVM/Clang

In order to compile OpenMP programs, the front-end (e.g., `Clang` for `C` and `C++`) first parses the OpenMP pragmas and applies semantic analysis to them. After that, for generating LLVM-IR code, `Clang` first outlines the code regions specified with the pragmas. Then, it passes the addresses of the outlined functions with their required inputs to appropriate library runtime calls and replaces the parallel regions with the runtime function calls. For instance, consider the code in Figure 2.7. The for loop (lines 2 and 3 of Figure 2.8) is in

```
1 #pragma omp parallel
2 for(int i=1; i<N; i++)
3    B[i] = A[i]+A[i-1];
```

Figure 2.7: Example of a OpenMP program.

the `parallel` pragma region. As explained, in the first step, `Clang` outlines this region to a function. Figure 2.8 shows the `C` pseudo-code of this function. Note that this operation takes place in the IR level, but for more clear representation, we show it in `C`.

```c
void omp_par_outline(int tid, int *N, int **A, int **B){
   for(int i=0; i<(*N); i++)
      (*B)[i] = (*A)[i]+(*A)[i-1];
   return;
}
```

Figure 2.8: Pseudo-code in `C` of how Figure 2.7 is compiled.

In the next step, the compiler replaces the call to the outlined function with a call to an OpenMP library runtime call for parallelism management. Figure 2.9 illustrates this call.

```c
omp_rt_parallel(0, &omp_par_outline, &N, &A, &B)
```

Figure 2.9: Example of a OpenMP runtime call in a program.

Note that in Figure 2.7, the for loop is only in a parallel region. This means the runtime system spawns some threads, and *all* of them will execute *all* iterations of the for loop. This execution model usually is not what one expects when using OpenMP, and other pragmas exist besides `parallel` pragma for exposing different parallelism patterns.

For each pragma, the compiler inserts different runtime calls. Moreover, the compiler may need to insert code in the outlined function to handle the prescribed parallelism correctly.

For example, consider the code in Figure 2.10. In this example, lines 2 and 3 are in the `parallel for` pragma. In this case, the iterations of the for loop should be *divided* between

```c
1 #pragma omp parallel for
2 for(int i=1; i<N; i++)
3    B[i] = A[i]+A[i-1];
```

Figure 2.10: Another example of a OpenMP program.

threads. In other words, each thread is responsible for running its specific *chunk* of the parallel for loop. To handle this case, the compiler first inserts another runtime call in the outlined function to compute the lower bound and the upper bound of the chunk that each thread should run. Then, for each thread, the loop in the outlined function should iterate from the lower bound to the upper bound computed for that thread.

The `Clang` compiler follows the same general strategy of outlining and runtime call insertion in the code for all of the OpenMP directives, although the details might be different. Another component in the LLVM's OpenMP compilation pipeline is `OpenMPIRBuilder` [58], that is designed to unify the IR code generation of `C/C++` and `Fortran` front-ends.

LLVM includes different OpenMP runtime systems (e.g., `libomp.so` for running OpenMP on the host, and `libomptarget.so` for OpenMP offloading). The runtime functions are responsible for managing parallelization, including:

- they spawn the right number of teams of threads at the beginning of the parallel region,
- run the specified region in the desired pattern, considering the limitations and conditions defined by pragmas and clauses,
- setup barriers to synchronize and join the threads.

As a result of this abstraction, the program in the LLVM-IR level is not aware of any parallel execution. Although this design has its benefits and provides flexibility, it introduces some challenges for optimization. We explain these challenges in more details in the rest of this section.

### OpenMP-aware optimizations

As explained before, different transformation passes are developed as part of the LLVM project to optimize programs. However, we cannot directly apply them to the programs in the existence of OpenMP parallel regions. The reason is that the early outlining approach taken by `Clang` (explained in Section 2.4.1) prevents the compiler from applying optimizations that require moving between the boundaries of the serial and parallel regions. The paper [59] discusses optimization complications and methodologies to overcome these challenges in parallel programs.

To handle parallel program optimization challenges, `OpenMPOpt`[19] pass is developed. It is an Inter Procedural Optimization (IPO) pass in LLVM. It runs in the `O1,O2` and `O3` optimization pipelines, and it is specifically designed for optimizing OpenMP programs. The main goal of this pass is to apply high level, OpenMP-aware optimizations to the program. The pass is made aware of what each OpenMP runtime calls do. It uses this information to apply effective optimization.

In addition to optimizing OpenMP programs, the `OpenMPOpt` is specially equipped to optimize device code when there exists offloading regions in the program.

# Chapter 3

# A Pipeline Detection Technique in Polly

The polyhedral model has repeatedly shown how it facilitates various loop transformations, including loop parallelization, loop tiling, and software pipelining. However, parallelism is almost exclusively exploited on a per-loop basis without much work on detecting cross-loop parallelization opportunities. While many problems can be scheduled such that loop dimensions are dependence-free, the resulting loop parallelism does not necessarily maximize concurrent execution, especially not for unbalanced problems.

In this work, we introduce a polyhedral-model-based analysis and scheduling algorithm that exposes and utilizes cross-loop parallelization through tasking. This work exploits pipeline patterns between iterations in different loop nests, and it is well suited to handle imbalanced iterations.

Our LLVM/Polly-based prototype performs schedule modifications and code generation targeting a minimal, language agnostic tasking layer. We present results using an implementation of this API with the OpenMP `task` construct. For different computation patterns, we achieved speed-ups of up to 3.5× on a quad-core processor while LLVM/Polly alone fails to exploit the parallelism.

## 3.1 Overview

The polyhedral model [10, 11] has proved to be very effective for optimizing loop nests by using different methods such as loop tiling, loop parallelizing, and software pipelining [13, 14, 15]. Almost all these methods optimize for-loop nests on a per-loop basis. However, another opportunity for optimization might exist in the program, which one can exploit by considering cross-loop parallelization; executing iteration blocks of different loop nests in parallel when it does not violate any dependence relations. There has been some efforts to consider this parallelization opportunity. The method proposed in [16] generates pipelined multi-thread code by interleaving iterations of some loops. Authors in [17] propose an algorithm for detecting pipeline opportunities between iteration blocks of two for-loop nests. Also, [60] uses cross-loop data reuse for cache optimizations. However, we are not aware of any fully-automatic, LLVM-based method for detecting and exploiting parallelization opportunities between iterations of different for-loop nests through tasking.

The main objective of this project is to detect the cross-loop task parallelism in a program. We exploit this opportunity by detecting pipeline pattern between iteration blocks of different for-loop nests; [1] we call it *cross-loop pipeline pattern*.

Detecting cross-loop pipelining provides a building block towards exploiting the natural data-flow parallelism. However, the existing loop optimization methods based on the polyhedral model have a limited ability to extract cross-loop pipeline patterns.

We assume the program consists of consecutive for-loop nests. We also assume that an iteration of a loop nest may depend on the previous iterations of the same loop nest, as well as iterations of the loop nests before it. Detecting cross-loop task parallelism is particularly important and effective for programs where (1) compute-intensive functions are called inside for-loop nests, or (2) no optimization opportunities for individual for-loop nests exist.

For instance, consider the program in Figure 3.1, where `A` and `B` are two initialized $N \times N$ matrices. Polly[18], LLVM-based framework for applying polyhedral transformations,

```
for(i=0; i<N-1; i++)
  for(j=0; j<N-1; j++)
  S:  A[i][j]=f(A[i][j], A[i][j+1], A[i+1][j+1]);

for(i=0; i<N/2-1; i++)
  for(j=0; j<N/2-1; j++)
  R:  B[i][j]=g(A[i][2*j], B[i][j+1], B[i+1][j+1],
                B[i][j]);
```

Figure 3.1: Example with cross-loop pipeline.

detects first level tiling, but it cannot detect parallelism for any of the for-loops in the program. However, there is a parallelization opportunity between iteration blocks of `S` and iteration blocks of `R`. For instance, consider the first two iterations of the second for-loop nest for computing `B[0][0]` and `B[0][1]`. The only element of the matrix `A` we need for the first iteration is `A[0][0]`, and for the second iteration is `A[0][2]`. Therefore, when `A[0][0]` is computed (after finishing the first iteration of the first loop nest), we can compute `B[0][0]` and the following elements of the matrix `A` in parallel. With the same method, when `A[0][2]` is computed, we can compute `B[0][1]` and the following elements of the matrix `A` in parallel. Note that the iterations of each statement run in their sequential order.

[1]Not be confused with software pipelining and DOACROSS loops, where a pipeline pattern exists between different iterations of the *same* loop nest.

Figures 3.2 and 3.3 illustrate this idea. The upper part, 3.2, shows the sequential execution of the iterations of the statements S and R, where iterations of R begin after all iterations of S are finished. The lower part, 3.3, shows the execution of the program after exploiting cross-loop task parallelization. In this case, `thread_0` runs the iteration blocks of S, and `thread_1` runs the iteration blocks of R. `Thread_1` can start running an iteration of R right after `thread_0` finishes the iteration block of S that it depends on.

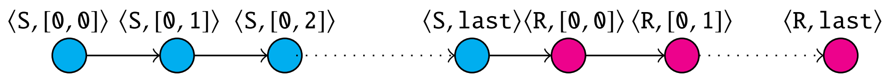


Figure 3.2: Sequential execution. R starts after iterations of S are finished.

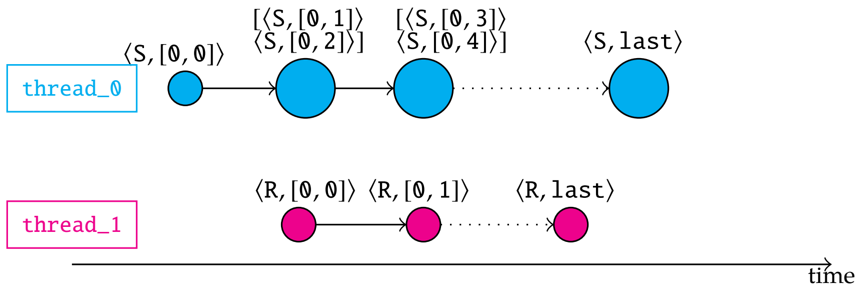


Figure 3.3: Pipeline execution. Iterations of R are overlapped with iterations of S, and R is not part of the critical path anymore.

In this work, we detect and exploit the cross-loop pipeline pattern using the polyhedral model. Our method is implemented as part of LLVM/Polly[18] and operates at compile time on the LLVM-IR [9]. The main idea is to block iteration domains such that finishing each block provides the requirements for running not only the next block in the same iteration domain, but also blocks in other iteration domains. After computing dependence relations between newly generated blocks, we construct an OpenMP `task` for each block to exploit the detected cross-loop task parallelism.

Some parts of the background information required in this chapter are explained in Chapter 2, and the rest of this chapter is organized as follows: in Sections 3.2 and 3.3, we explain the pipeline parallelism and tasking in OpenMP, respectively. These information provide the foundation of our research. Then, we explain some related works and comparing them to our research in Section 3.4. After that, we explain our transformation algorithm in Section 3.5, and we go through the details of the scheduling algorithm and code generation in Section 4.3. We conclude this chapter with reporting on our experimental results in Section 3.7 and the future plans for continuing this project in Section 3.8.

## 3.2 Pipeline parallelism

Pipeline parallelism is a well-known technique [61, 62, 63, 64] for parallelizing different applications. The main idea of pipelining is to divide a process into multiple *stages*. Then, if there exist enough hardware resources, stages of different processes can *potentially* run concurrently. Whether they actually run concurrently or not depends on the dependency relations between different stages of one process, and stages of different process. In other words, pipelining can be used when a sequence of data items has to go through a sequence of stages, and the input of each stage is the output of its previous stage. Concurrency happens when a stage $i$ can start operating on a data item $d$ after stage $i-1$ has finished processing $d$, but not the whole sequence of data items.

## 3.3 Tasking in OpenMP

Since version 3.0, OpenMP supports task parallelism, using the `omp task` pragma. To increase the applicability of task parallelism, OpenMP 4.0 introduces the `depend` clause. Let `M` be a shared memory location. OpenMP uses the `depend(in:M)`, `depend(out:M)`, and `depend(inout:M)` clauses to specify whether the considered task reads, writes, or both reads and writes `M`. The runtime system uses this information to manage dependencies between tasks, and decide whether a task can execute, or should wait for other tasks to finish.

For example, consider the code in Figure 3.4 from [65]. In this example, there are three

```
int x = 1;
#pragma omp parallel
#pragma omp single
{
   #pragma omp task shared(x) depend(out: x)
      x = 2;
   #pragma omp task shared(x) depend(in: x)
      printf("%d ", x+1);
   #pragma omp task shared(x) depend(in: x)
      printf("%d ", x+2);
}
```

Figure 3.4: Example of OpenMP tasks.

tasks: the first one assigns `2` to the variable `x`, the second one prints the value of `x + 1`, and the last one prints the value of `x + 2`. As a result of the dependencies to `x`, the first task should finish before the second and the third tasks, but the second and third ones can run in parallel. Therefore, the output can be both `3 4` and `4 3`.

## 3.4 Related works

Various lines of research have been investigated on automatically detecting pipeline-related patterns. In this section, we go through some of the papers that are related to our work.

In [66] authors discuss the exploitation of pipeline parallelism, including use-cases, and the critical challenge of managing dependencies between the source region and the target region. We try to address this challenge in the case of pipelining iterations of for-loop nests. Also, the software pipelining technique for pipelining iterations of a single loop is discussed in [67].

The method proposed in [16] follows the same objective as our work, that is, to use the polyhedral model and exploit pipeline parallelism opportunities between loop nests so as to optimize (part of) programs that conventional polyhedral optimizers cannot optimize. However, there are differences between the two approaches. Our prototype operates at the IR level of a non-optimized program, whereas the prototype in [16] operates at the source level of programs already optimized by Pluto and parallelized by the OpenMP API. Also, it can detect and exploit the pipeline pattern only if (1) the considered loop nests have identical iteration domains and chunk sizes, (2) are not associated with SIMD constructs, and (3) are in the same parallel region. Moreover, each iteration of the target loop nest should depend on the same or the previous iterations of its source loop nest. With these considerations, the prototype in [16] can use the clauses `ordered` and `nowait` of OpenMP to exploit the pipeline pattern. In our work, by using the general transformation algorithm described in Section 3.5 , and by taking advantage of the OpenMP constructs `task` and `depend`, we can detect and exploit pipeline patterns in loop nests of sequential programs with arbitrary memory accesses. Also, our transformation algorithm for task detection is independent of the OpenMP tasking layer.

The method explained in [17] detects pipeline parallelism to make machine learning models' executions more efficient on the so-called *computational memory accelerators* considered in that paper. We provide more details on the algorithm of this paper in Section 3.5.

The objectives of the authors in [68] are similar to ours: they aim at exploiting parallelization between different loop nests. However, the method of [68] and the output are different. The authors discuss a method based on linear regression for detecting pipeline patterns in pairs of consecutive loop nests, using run-time information.

The Pluto [13] algorithm supports a method for detecting software pipelining in the form of DOACROSS loops applied on a program tiled by the Pluto algorithm. [69] explains a polyhedral-model-based method for designing a compiler-runtime system to exploit task parallelism in distributed and shared memory architectures. It uses Pluto's tiling and parallelization for task detection, and the runtime system coordinates the dependencies between tasks.

The polyhedral model is used in [70] to exploit DOACROSS loops using OpenMP. Contrary to our approach, their input program is in data-flow graph language. The work explained in [71] optimizes programs using OpenMP tasks, where program annotations explicitly specify the tasks. Also, [72] introduces OpenStream as a data-flow extension of OpenMP. It can exploit pipeline and data-flow parallelism on an annotated program.

A well-studied subject closely related to our work is automatically extracting the data-flow graph from the program to run it on data-flow architectures. For example, [73] develops an algorithm for automatically extracting data-flow threads from programs. In another work, [74] develops an LLVM-based prototype to find the data-flow graph between LLVM-IR instructions of a program.

## 3.5 Transformation Algorithm

In this section, we explain our algorithm for detecting the cross-loop pipeline pattern in a program. We explain each step in detail and conclude the section with performance analysis of the algorithm.

The algorithm proposed in [17] provides the foundation for our transformation algorithm. The algorithm of [17] first considers two for-loop nests, called source and target, where iterations of the source loop nest write to a shared array, and iterations of the target loop nest read from that same shared array. Then, this algorithm finds a relation that maps the index of each write in the shared array to the maximum iteration of the target that can safely execute. Finally, using the specifications of the so-called *computational memory accelerator* studied in that paper, this map coordinates different pipeline stages between iteration blocks of the two considered for-loop nests.

The *pipeline map* computed by our algorithm (Section 3.5.1) considers iteration blocks of the source and the target for-loop nests for coordinating different stages of the pipeline. Moreover, contrary to the method in [17], our algorithm does not stop after finding the pipeline relations between pairs of for-loop nests. By computing *pipeline blocking maps of iteration domains* (Section 3.5.2), we extend the algorithm of [17] to detect the pipeline pattern between all dependent loop nests in the program. In addition, we compute *pipeline dependency maps* (Section 3.5.3) to determine dependence relations between tasks at compile time and make them suitable for generating task-parallel OpenMP program in the next phase.

### 3.5.1 Pipeline Map

Consider two statements S and T with respective iteration domains $\mathcal{I}$ and $\mathcal{J}$. Also, assume the iterations of S write in a set of memory locations $\mathcal{M}$, and the iterations of T read from $\mathcal{M}$. We define the *pipeline map* between S and T to be the relation $\mathcal{T}_{\mathrm{S,T}}(\mathcal{I} \rightarrow \mathcal{J})$, where $(\vec{i}, \vec{j}) \in \mathcal{T}_{\mathrm{S,T}}$ if and only if:

1- after running all iterations of S up to $\vec{i}$, we can safely run all iterations of T up to $\vec{j}$,

2- $\vec{i}$ is the smallest (lexicographically) vector and $\vec{j}$ is the largest (lexicographically) vector with Property (1).

This map is called the pipeline map; because for every pair $(\vec{i}, \vec{j})$ in $\mathcal{T}_{\mathrm{S,T}}$, we can run iterations of T up to $\vec{j}$ and iterations of S after $\vec{i}$, in parallel. Repeating this pattern creates a pipeline among iteration blocks of the loop nests.

To compute the pipeline map, we take a similar approach as the one used in [17]. Let $Wr(\mathcal{I} \to \mathcal{M})$ be the *write relation* of S, that is, the set of the pairs $(\vec{i}, m) \in \mathcal{I} \times \mathcal{M}$ such that location $m$ is written at iteration $\vec{i}$. Similarly, let $Rd(\mathcal{J} \to \mathcal{M})$ be the *read relation* of T, that is, the set of the pairs $(\vec{j}, m) \in \mathcal{J} \times \mathcal{M}$ such that location $m$ is read at iteration $\vec{j}$. Also, assume that there is no over-write, that is, $Wr$ is injective.

Using $Wr$ and $Rd$, we define $\mathcal{P}(\mathcal{J} \to \mathcal{I})$, as the composition of $Wr^{-1}$ by $Rd$ to relate the two iteration domains:

$$\mathcal{P} = Wr^{-1}(Rd).$$

Note that $(\vec{j}, \vec{i}) \in \mathcal{P}$ means that iteration $\vec{j}$ reads the same memory location that iteration $\vec{i}$ writes.

Then, we find the domain of $\mathcal{P}$, $\mathcal{D}_{\mathcal{P}}$. By mapping each member of $\mathcal{D}_{\mathcal{P}}$ to all other members of the same iteration domain ($\mathcal{J}$) that are lexicographically less than or equal to it, we get the map $\mathcal{D}'(\mathcal{J} \to \mathcal{J})$.

After that, we find the relation $\mathcal{H}(\mathcal{J} \to \mathcal{I})$ defined by:

$$\mathcal{H} = \mathsf{lexmax}(\mathcal{P}(\mathcal{D}')).$$

Relation $\mathcal{H}$ maps each read iteration $\vec{j}$ of the target statement to the lexicographically largest write iteration $\vec{i}$ of the source statement that $\vec{j}$ and its previous iterations depend on.

The final step to get the pipeline map is to find $\mathcal{H}^{-1}(\mathcal{I} \to \mathcal{J})$, and deduce the pipeline map $\mathcal{T}_{\mathtt{S},\mathtt{T}}$ as:

$$\mathcal{T}_{\mathtt{S},\mathtt{T}} = \mathsf{lexmax}(\mathcal{H}^{-1}). \tag{3.1}$$

Having $(\vec{i}, \vec{j}) \in \mathcal{H}^{-1}$ means that after running iteration $\vec{i}$ of the source statement, we can run iteration $\vec{j}$ of the target, and every iteration before that. Note that if $(\vec{i}, \vec{j}) \in \mathcal{H}^{-1}$, then $(\vec{i}, \vec{j_1}) \in \mathcal{H}^{-1}$, for all $\vec{j_1}$ that are lexicographically less than $\vec{j}$. As a result, one iteration of the source statement may be mapped to multiple iterations of the target statement. Therefore, we use the operation lexmax in 3.1 to get the maximum one.

As an example, consider Figure 3.1 with `N=20`. The pipeline map between statements S and R is:

$$\begin{aligned}\{\mathtt{S}[i_0, i_1] \to \mathtt{R}[o_0, o_1] : \exists (e_0 = \lfloor (i1)/2 \rfloor : \\ o_0 = i_0 \land 2e_0 = i_1 \land 2o_1 \geq i_1 \land 2o_1 \leq 1 + i_1 \\ \land\, i_0 \geq 0 \land i_0 \leq 8 \land i_1 \geq 0 \land i_1 \leq 16)\}.\end{aligned}$$

This should be read as the set of all $((i_0, i_1), (o_0, o_1))$ where $(i_0, i_1)$ is an iteration of the source loop and $(o_0, o_1)$ is an iteration of the target loop such that

$$o_0 = i_0 \land 2e_0 = i_1 \land 2o_1 \geq i_1 \land 2o_1 \leq 1 + i_1 \land i_0 \geq 0 \land i_0 \leq 8 \land i_1 \geq 0 \land i_1 \leq 16,$$

where $e_0 = \lfloor (i1)/2 \rfloor$.

In the next step, we use the pipeline maps to partition the iteration domain of each statement to get the iteration blocks that are in pipeline relation. For a statement S and a pipeline map

$\mathcal{T}$, if S is the source (resp. target) statement, we first partition its iteration domain, $\mathcal{I}$, such that each element of $\mathsf{Dom}(\mathcal{T})$ (resp. $\mathsf{Range}(\mathcal{T})$) is the lexicographically largest member of its part. Then, by mapping each member of each part to the lexicographically largest member of that part, we get a *source blocking map* $\mathcal{V}_{\mathtt{S}}(\mathcal{I} \to \mathcal{I})$ (resp. a *target blocking map* $\mathcal{Y}_{\mathtt{S}}(\mathcal{I} \to \mathcal{I})$).

To compute these maps, let $\mathcal{B} = \mathsf{Dom}(\mathcal{T})$ if S is the source in the pipeline map $\mathcal{T}$ (resp. $\mathcal{B} = \mathsf{Range}(\mathcal{T})$ if S is the target in the pipeline map $\mathcal{T}$). We compute $\mathcal{B}'$ as:

$$\mathcal{B}' = \mathsf{lexleset}(\mathcal{I}, \mathcal{B}).$$

Operation lexleset between two sets $S_1$ and $S_2$, maps each element of $S_1$ to all other elements in $S_2$ that are less than or equal to the considered element.

After computing the map $\mathcal{B}'$, the source blocking map, $\mathcal{V}_{\mathtt{S}}(\mathcal{I} \to \mathcal{I})$ (resp. the target blocking map $\mathcal{Y}_{\mathtt{S}}(\mathcal{I} \to \mathcal{I})$) is as:

$$\mathsf{lexmin}(\mathcal{B}'_{\mathcal{T}}). \tag{3.2}$$

If there are no iterations of T depending on the final iterations of S, then those last iterations of S do not appear in $\mathcal{T}_{\mathtt{S},\mathtt{T}}$; therefore, they do not appear in the source blocking map. To handle this case, we add a block consisting of all remaining iterations by mapping them to the lexicographically maximum iteration of the iteration domain.

Continuing with the example of the Figure 3.1, one part of the source blocking map is:

$$\exists (e_0 = \lfloor (o_1)/2 \rfloor : o_0 = i_0 \wedge 2e_0 = o_1 \wedge i_0 \geq 0 \wedge i_0 \leq 8 \wedge$$
$$i_1 \geq 0 \wedge i_1 \leq 16 \wedge o_1 \geq i_1 \wedge o_1 \leq 1 + i_1).$$

Therefore, some elements of the map are:

$$\{\mathtt{S}[1,1] \to \mathtt{S}[1,2], \mathtt{S}[1,2] \to S[1,2],$$
$$\mathtt{S}[1,3] \to \mathtt{S}[1,4], \mathtt{S}[1,4] \to \mathtt{S}[1,4]\}.$$

This means that iterations $[1, 1]$ and $[1, 2]$ are in one block, and $[1, 3]$ and $[1, 4]$ are in another block.

### 3.5.2 Pipeline Blocking Maps of Iteration Domains

It is important to note that for each statement, there are *several* pipeline maps. As a result, there are several source and target blocking maps. For instance, consider Figure 3.5, which adds a for-loop nest to Figure 3.1. There are two source blocking maps for the statement S; one for the pipeline map between S and R, and one for the pipeline map between S and U. For the statement R, there is one target blocking map for the pipeline map between S and R, and one source blocking map for the pipeline map between R and U. For the statement U, there are two target blocking maps; for the pipeline maps between S and U, and between R and U.

```
for(i=0; i<N-1; i++)
  for(j=0; j<N-1; j++)
  S:  A[i][j]=f(A[i][j], A[i][j+1], A[i+1][j+1]);

for(i=0; i<N/2-1; i++)
  for(j=0; j<N/2-1; j++)
  R:  B[i][j]=g(A[i][2*j], B[i][j+1], B[i+1][j+1],
                B[i][j]);

for(i=0; i<N/2-1; i++)
  for(j=0; j<N/2-1; j++)
  U:  C[i][j]=h(A[2*i][2*j], B[i][j], C[i][j+1],
                C[i+1][j+1], C[i][j]);
```

Figure 3.5: Example with 3 loop nests.

However, we need to have a single *pipeline blocking map* of iteration domain per statement, where each pipeline block can be considered an *atomic task*. Therefore, for each statement, we should integrate all its source and target blocking maps such that we can establish a pipeline relation between all blocks of all statements. We also need to choose these blocks to maximize the number of blocks of different loops that can execute in parallel to get the best possible performance at the end. To satisfy both conditions, we *minimize* the size of the blocks as much as possible and construct the *optimal blocks* from all blocking maps associated with each statement. In fact, for each statement, we compute the lexmin of the union of all source and target pipeline blocking maps:

$$\mathcal{E}_{\mathsf{S}} = \mathsf{lexmin}((\bigcup_{j}(\mathcal{V}_{\mathsf{S}}^{j}) \cup (\bigcup_{i}(\mathcal{Y}_{\mathsf{S}}^{i}))). \tag{3.3}$$

In this equation, $\mathcal{Y}_{\mathsf{S}}^{i}$ (resp. $\mathcal{V}_{\mathsf{S}}^{j}$) goes over the target (resp. source) blocking maps of S with respect to the pipeline maps between S and other statements that S depends on (resp. statements that depend on S).

From this point on, for two vectors $\vec{i}$ and $\vec{j}$ in the iteration domain of S, if $\mathcal{E}_{\mathsf{S}}(\vec{i}) = \mathcal{E}_{\mathsf{S}}(\vec{j}) = \vec{\ell}$, we say that $\vec{i}$ and $\vec{j}$ are in the same block, and we call this block $\vec{\ell}$. With this definition, we can say that Equation 3.3 assigns to each iteration the smallest block that it belongs to, among all source and target blocking maps.

To illustrate the effectiveness of choosing optimal blocks for correctness and efficiency, consider Figure 3.7. In this example, statements $\mathsf{S}_1$ and $\mathsf{S}_2$ are sources of the statement $\mathsf{S}_3$, and $\mathsf{S}_3$ is the source for statement $\mathsf{S}_4$. Figure 3.6 shows the dependencies and the pipeline maps associated with each of them.

We want to find the first pipeline block of $\mathsf{S}_3$ after $\vec{j_0}$. In other words, we are looking for the lexicographical maximum vector of the first block after $\vec{j_0}$.

After finishing the execution of $\mathsf{S}_1$ up to iteration $\vec{i_1}$, the dependencies to $\mathsf{S}_1$ are satisfied

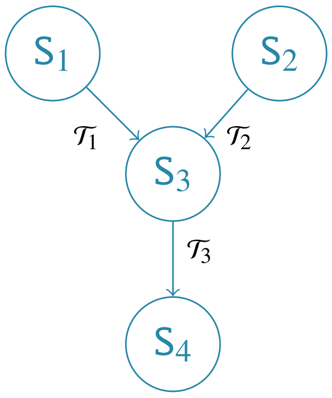


Figure 3.6: Dependency relations graph between statements and the corresponding pipeline maps.

for iterations of $S_3$ up to $\vec{j}_1$. The same holds for iterations of $S_2$ up to $\vec{i}_2$ and iterations of $S_3$ up to $\vec{j}_2$. On the other hand, after finishing iterations of $S_3$ up to $\vec{j}_3$, we can run iterations of $S_4$ up to $\vec{i}_3$. As a result, after finishing iterations of $S_1$ up to $\vec{i}_1$, and iterations of $S_2$ up to $\vec{i}_2$, we can safely run $S_3$ up to iteration $\vec{j}_2$. Therefore, considering any vector between $\vec{j}_0$ and $\vec{j}_2$ maintains the correct execution of $S_3$. However, by choosing $\vec{j}_3$, the optimal block computed by Equation 3.3, we maximize the number of blocks of different statements that can run in parallel, because $S_4$ can also start running right after $\vec{j}_3$ is finished.

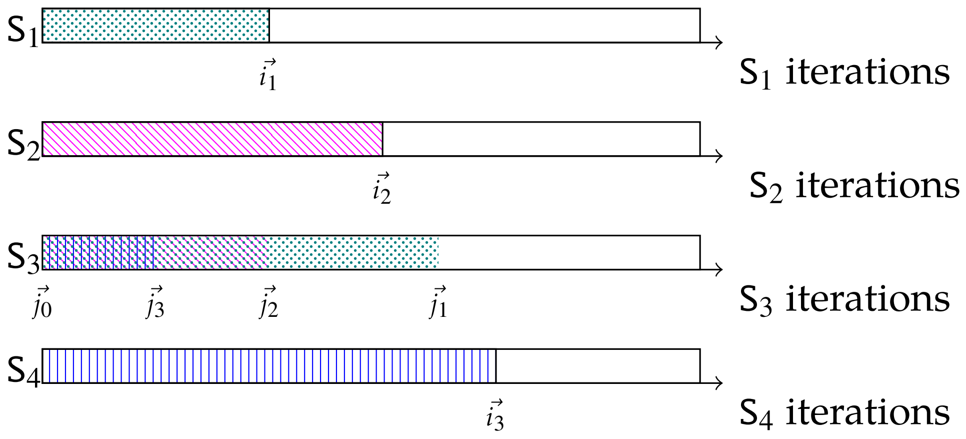


Figure 3.7: Choosing $\vec{j}_3$ as the first pipeline block after $\vec{j}_0$ maintains correctness and maximizes the number of blocks of different statements that can run in parallel.

### 3.5.3 Pipeline Dependency Relations

Up to this point, we have found the pipeline blocking maps of the iteration domain of each statement. These blocks of iterations are considered as the tasks (pipeline stages). However, to have a correct task-parallel program, we also need to compute the dependence relations between different tasks so that they can be used to coordinate OpenMP tasks. Therefore, after computing pipeline blocking maps of all statements, in the next step, we find the requirements of each block. By requirements of a block, we mean the blocks of its source statements it needs to execute safely. This part explains how to find *pipeline dependency relations*, which are maps between each block and its requirements. These maps will be used as in-dependencies (`depend(in:)`) of the OpenMP tasks we create in the next phase.

To find pipeline dependency relations of a statement S, consider a specific pipeline map $\mathcal{T}_i$ and its corresponding target blocking map, $\mathcal{Y}_i$. First, for every block of S, that is, for each element of $\mathsf{Range}(\mathcal{E}_\mathsf{S})$, we compute the block of $\mathcal{Y}_i$ that it belongs to. Then, we can get the last required block using $\mathcal{T}_i$. Considering all target blocking maps of S, we get an array of maps showing the requirements of the blocks of S. We show this array with $Q_\mathsf{S}$, and each index of it with $Q^i_\mathsf{S}$. Equation 3.4 shows the computation of each map $Q^i_\mathsf{S}$.

$$Q^i_\mathsf{S} = \mathcal{T}_i^{-1}(\mathcal{Y}_i(\mathsf{Range}(\mathcal{E}_\mathsf{S}))). \tag{3.4}$$

In Equation 3.4, $\mathcal{T}_i$ goes over all pipeline maps that S is considered as their target statement, and $\mathcal{Y}_i$ is the target blocking map corresponds to $\mathcal{T}_i$.

Furthermore, running each block of a statement S provides the requirements for some blocks of the statements that are dependent on S. This only depends on the last executed iteration of S. We can get this relation from the identity map of $\mathsf{Range}(\mathcal{E}_\mathsf{S})$, and we call it $Q'_\mathsf{S}$. This maps will be used as the out-dependency (`depend(out:)`) of the OpenMP tasks we create in the next phase.

The final algorithm for finding the cross-loop pipeline relation of a SCoP is summarized in Algorithm 1.

### 3.5.4 Algorithm Correctness and Efficiency

In this part, we show that the algorithm presented in this section is correct (the transformation does not change the semantics of the program). Also, we show that in the general case of the algorithm, the best performance we can get from cross-loop pipelining is constrained by the most time-consuming loop nest. We also explain that with our transformation algorithm, we can automatically get the ideal speedup (ignoring the overhead related to task creation).

#### Correctness

Considering minimal blocks as computed in Section 3.5.2 and the dependencies computed in 3.5.3 ensures the correctness of the transformation. Consider a statement that depends on multiple other statements. By considering optimal blocks as tasks with dependencies described in 3.5.3, we know that dependencies to all other statements are satisfied when the task is running. Moreover, by considering optimal blocks, we can run every task as

**Algorithm 1:** Pipeline detection algorithm

```
Input  : SCoP in its polyhedral representation
Output: SCoP with pipeline information
1  for all statement pairs S and T of the SCoP do
2      if T depends on S then
3          $\mathcal{T}_{S,T}$ = pipeline map(S, T);
4          $\mathcal{V}_{S,T}$ = source blocking map(S, $\mathcal{T}_{S,T}$);
5          $\mathcal{Y}_{T,S}$ = target blocking map(T, $\mathcal{T}_{S,T}$);
6          $\mathcal{E}_S = \mathcal{E}_S \cup \mathcal{V}_{S,T}$;
7          $\mathcal{E}_T = \mathcal{E}_T \cup \mathcal{Y}_{T,S}$;
8  for all statements S of the SCoP do
9      $\mathcal{E}_S$ = lexmin($\mathcal{E}_S$);
10     $Q'_S$ = identity map(Range($\mathcal{E}_S$));
11 for all pipeline maps $\mathcal{T}_{S,T}$ do
12     $Q_T$ = append($\mathcal{T}_{S,T}^{-1}$($\mathcal{Y}_{T,S}$(Range($\mathcal{E}_T$))), $Q_T$);
13 for all statements S of the SCoP do
14     add $\mathcal{E}_S$, $Q_S$, $Q'_S$ to the SCoP;
15 return SCoP;
```

soon as its dependencies are stratified. This increases the potential number of tasks that can run concurrently.

### Efficiency

Assume that the input program consists of N for-loop nests $L_1, \cdots, L_N$. We want to compare the total running time of the pipelined execution and the sequential execution.

We have to run all iterations of all loops, and since we do not consider any other form of parallelism in the general case, we do not reduce the running time of individual for-loop nests. The performance improvement comes from the places that we can *overlap* the execution of iteration blocks of *different* for-loop nests. Therefore, *the performance of the pipelined program is limited to the loop nest with the maximum running time*, $L_{max}$, and we have the following formula for the running time of the pipelined program:

$$\text{time}(L_{max}) \leq \text{time(pipeline)} \leq \text{time(sequential)} \tag{3.5}$$

The lower bound happens when the first loop nest has the maximum running time and the execution of all other for-loop nests can be covered by that. The average case is when the $i^{th}$ loop nest has the maximum running time. Also, we usually cannot assume that the running time of all loop nests after the maximum loop can be covered. For instance, consider Figure 3.8.

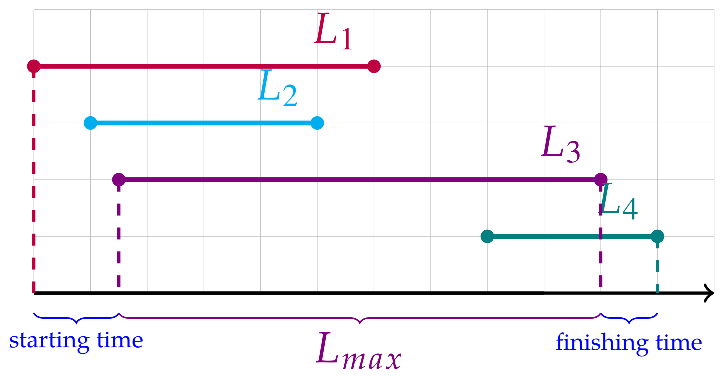


Figure 3.8: Average case performance of pipelined program, where the third loop has the maximum running time.

As a result, we can compute the total running time of the pipelined program with Equation 3.6.

$$\text{time}(\text{pipeline}) = \text{starting time} + \text{time}(L_{max}) + \text{finishing time} \tag{3.6}$$

In Equation 3.6, starting time is the duration between the beginning of the program and beginning of the $L_{max}$, and finishing time is the duration between the termination of $L_{max}$ and the termination of the program.

With the assumption that we have enough hardware resources, because each tasks starts running as soon as its requirements are satisfied, we minimize the starting and finishing times, and we get the maximum possible overlap between the iteration blocks of different for-loop nests.

## 3.6 Implementation

We implemented a prototype [2] of the algorithm explained in Section 3.5 as a part of Polly [18] and use the ISL library [36] for polyhedral computation. We modify Polly passes in the analysis, transformation, and code generation phases to add support for the pipeline pattern detection and code generation. For exploiting the detected parallelism, we use OpenMP `task` constructs.

### 3.6.1 Analysis

In the analysis passes, we extend the definition of the SCoP to include information needed for pipelining. We use the iteration domains and memory access relations and compute the maps $\mathcal{E}_S$, $Q_S$, and $Q'_S$ for every statement in the SCoP, by using Algorithm 1.

[2] https://github.com/dtalaashrafi/polly-pipeline

### 3.6.2 Transformation and Scheduling

In this step, we use the pipeline information of the SCoP to find the new schedule tree. For each statement `S`, we transform its schedule to separate the loops iterating over the blocks determined by $\mathcal{E}_S$, from the ones iterating inside each block. The reason is that each block is an atomic task, with its dependencies computed in $\mathcal{Q}_S$ array of maps.

We define the *pipeline loop* to be the inner-most loop that iterates over blocks. The critical property of a pipeline loop is that its body iterates inside the blocks. Therefore, each of its iterations is a single task. To summarize, our goal is to construct a new schedule tree that:

1- blocks iteration domains,

2- finds pipeline loops, and

3- attaches dependency information to each task.

To begin with, we want the pipeline dependency relations to be defined as functions of the induction variables of the loops iterating over blocks. Therefore, for each statement `S`, we construct a `pw_multi_aff_list` from the maps in $\mathcal{Q}_S$ and a `pw_multi_aff` from the map $\mathcal{Q}'_S$. After that, we create a mark node from them to add to the schedule tree.

To construct the final schedule, we begin by creating two separate schedule trees: one for iterating over blocks and one for iterating inside each block. Then, we *expand* the first schedule tree with the second one.

Let $\mathcal{D}_{\mathcal{E}_S}$ and $\mathcal{R}_{\mathcal{E}_S}$ be the domain and the range of $\mathcal{E}_S$, respectively. We first create a schedule domain node from $\mathcal{R}_{\mathcal{E}_S}$. Then, we get the partial schedule of the identity map of $R_{\mathcal{E}_S}$ and add the corresponding band node to the created domain node. This schedule tree iterates over the blocks in lexicographical order. The next step is to construct the expansion schedule tree for iterating inside the blocks. This time, we repeat the same process as above, and we use $D_{\mathcal{E}_S}$ (instead of $R_{\mathcal{E}_S}$) for creating the domain node and the partial schedule. At this step, we add the mark node containing pipeline dependency information. Note that this mark node is located before the band node iterating inside the block, and it can be used for finding the pipeline loop. To complete the expansion process, we need to provide the *contraction function* for mapping domain elements of the original schedule and domain elements of the expansion schedule. For this purpose, we use the map $\mathcal{E}_S$, as it defines this relation by definition.

To summarize, Algorithm 2 is our final scheduling method.

### 3.6.3 AST Generation

In the AST generation phase, we use the schedule tree to create the AST. Specifically, we use the mark nodes in the schedule tree to annotate the AST. Figure 3.9 shows parts of the AST of the transformed program of Figure 3.5. In Figure 3.9, there exists a for-loop nest corresponding to each loop nest in the original program. The comments in lines 3, 11, and

**Algorithm 2:** Computing schedule tree

```
Input   : Pipeline information of statements in SCoP
Output: Updated schedule tree
1  for all statements S in SCoP do
2      $\mathcal{D}_{\mathcal{E}_S}$ = Domain($\mathcal{E}_S$), $\mathcal{R}_{\mathcal{E}_S}$ = Range($\mathcal{E}_S$);
3      $ps_1$ = partial schedule(identity map($\mathcal{R}_{\mathcal{E}_S}$));
4      $ps_2$ = partial schedule(identity map($\mathcal{D}_{\mathcal{E}_S}$));
5      $m$ = mark node( $Q_S, Q'_S$ );
6      $node_1$ = domain node($\mathcal{R}_{\mathcal{E}_S}$);
7      $sch_1$ = insert partial schedule($ps_1, node_1$);
8      $node_2$ = domain node($\mathcal{D}_{\mathcal{E}_S}$);
9      $sch_2$ = insert partial schedule($ps_2, node_2$);
10     $sch_2$ = insert mark node($m, node_2$);
11     $contraction$ = union_pw_multi_aff($\mathcal{E}_S$));
12     $sch_S$ = expand($sch_1, sch_2, contraction$);
13 $sch$ = sequence($sch_{\forall S \in SCoP}$);
14 return sch
```

16 are representatives of the AST annotations. They show that the for loops in lines 2, 10, and 15 are the pipeline loop in their loop nest. They also contain the pipeline dependency information for the block that follows them.

### 3.6.4 Code Generation

The main idea for the code generation phase is to extract the tasks, which are the bodies of pipeline loops, to function calls. Then by passing the extracted function along with its dependency information to a high-level function implemented in a framework capable of task parallelism, we can utilize the detected parallelism. In this prototype, we can generate code for programs with for-loop nests of depth at most two, with only one task annotation per loop nest. However, considering loops in the general case is feasible, and it is a matter of further developing our code generation function.

To get the pipeline dependency information, we use the annotations of the AST and convert them to the format needed by the framework we are using. In this work, we use OpenMP `task` constructs with `depend` clauses. Remember that the annotation assigns a `pw_multi_aff_list` and a `pw_multi_aff` to each task. Using OpenMP terms, each member of the `pw_multi_aff_list` is an in-dependency of the task, and the `pw_multi_aff` is its out-dependency. To find pipeline dependency information of each task, we compute unique integer values from each in-dependencies and the out-dependency. Each piece is a vector that we convert to an integer. We multiply each dimension to a large enough integer and add them all to get a single integer. To distinguish between each pw_multi_affs, we pair an index with the integer we got in the previous step.

```
for(c0=0; c0<N; c0+=1)
  for(c1=0; c1<N; c1+=1) {
    // task
    ...
    S(c0,c1)
    ...
}
if (N>=2) {
  for (c0=0; c0<N/2; c0+=1)
    for (c1=0; c1<N/2; c1+=1) {
      // task
      R(c0, c1);
    }
  for (c0=0; c0<N/2; c0+=1)
    for (c1=0; c1<N/2; c1+=1) {
      // task
      U(c0, c1);
      }
}
```

Figure 3.9: Example of the AST of a pipelined program.

In the final step, we extract all loop nests that we want to pipeline in another function. This function is called in `omp parallel` and `omp single` pragmas to launch and initialize the tasks.

### 3.6.5 OpenMP Tasks

In the final step, we design a high-level OpenMP function for exploiting the detected task parallelism.

Each task is defined as a function pointer with its input arguments integrated into a structure. We use the in-dependencies and out-dependencies of the tasks as computed in Section 3.6.4. We also need the size of the input argument and the total number of statements that a task depends on them. Figure 3.10 shows the signature of this high-level function.

```
void CreateTask(void (*f) (void *), void *input,
               int outDepend, int outIdx,
               int *inDepend, int *inIdx,
               int inputSize, int dependNum)
```

Figure 3.10: Signature of the function for creating tasks.

To coordinate between tasks, we define a global integer pointer `dependArr` and initialize it with NULL. We treat this pointer as a linearized two-dimensional array, where each column corresponds for each statement, and each row is for a specific iteration block of that statement. We also define a variable, `writeNum` to keep the number of loop-nests in the program that are sources of other loop-nests. With these assumptions, each task writes in the location `[writeNumber*outDepend+outIdx]` and reads from the locations `[writeNumber*inDepend[i]+inIdx[i]]` of `dependArr`. Also, based on Equation 3.1 and the definition of pipeline maps, for maintaining the correctness of the program, blocks of the same for-loop nest should run in order. To add this in-dependency, we use the fact that all tasks created from iterations of the same for-loop nest have the same function pointer. Therefore, we keep track of the number of tasks created from each loop nest in a global array, `funcCount`, and use the function pointer of each task to coordinate different blocks of that task. The code in Figure 3.11 illustrates the creation of a task in the general case.

```
void *taskInput = malloc(inputSize);
memcpy(taskInput, input, inputSize);
int *self = (int *) f;
#pragma omp task
depend(out:dependArr[writeNum*outDepend+outIdx])
depend(iterator(i=0:dependNum),in:dependArr
                 [writeNum*inDepend[i]+inIdx[i]])
depend(in:self[funcCount[outIdx]-1])
depend(out:self[funcCount[outIdx]])
{
    f(taskInput);
    free(taskInput);
}
```

Figure 3.11: Function for creating tasks in OpenMP.

## 3.7 Evaluation

We explained in Section 3.1 that cross-loop pipelining is especially important for the programs that compute-intensive functions are called in for-loop nests. For the evaluation of our algorithm and prototype, we use two benchmark sets, where programs are compiled using the `Clang` compiler with the `O3` option, and all tests run on an x86_64 Intel quad-core processor with two threads per core, clocks at 2900.000 MHz.

In the first benchmark set, we want to show the improvements that cross-loop pipelining can make to the programs it is designed for; programs consist of a sequence of compute-intensive serial for-loop nests. For this benchmark, we simulate compute-intensive kernels by using the `next_prime` function of the GMP library [75]. The basic data structure, `gmp_data`, is an array of `mpz` (data structure for multi-precision integer in the GMP library), and it has `SIZE` elements. All loop nests have depth two, and the $i^{th}$ loop nest of the program updates elements of the matrix `A`$_i$ by calling a function that adds its input arguments element-wise and finds the `num`$_i^{th}$ prime number after that (with `next_prime` function). The `A`$_i$s are two dimensional `N` × `N` matrices of `gmp_data`. Kernels are designed such that Polly cannot parallelize the loops (loops are sequential) and the running time of the version optimized with Polly is comparable with the sequential version. Table 3.1 shows the properties of our experimental data. The *Specification* column shows the number of loop nests and the values of $num_i$s. The *Memory access* column shows the read access of each statement from the arrays written in the previous loop nests (lower and upper bounds of the loops are set accordingly). In this column, `Si` is the statement in the $i^{th}$ loop nest. The access patterns of the test `P3` is similar to the patterns in the factorization algorithm.

Recall that blocks of one for-loop nest should run sequentially. Therefore, for a program with $n$ loop nests, there can be at most $n$ tasks running in parallel. Figure 3.12 shows the pipelined program's speed-up compared with the sequential program for different values of `N` and `SIZE`. From Table 3.1 and Figure 3.12, we can see that cross-loop pipelining always gains speed-up; however the amount of it depends on the loops access patterns.

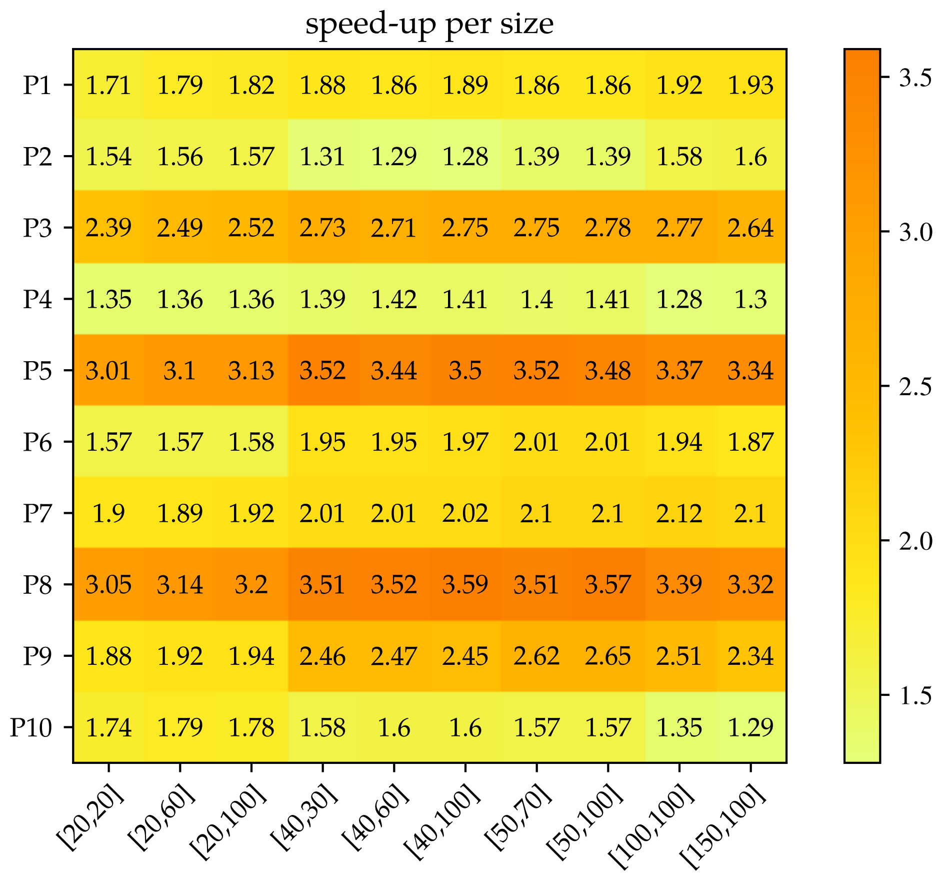


Figure 3.12: Speed-up of different test cases, considering different values for `N` and `SIZE`, comparing sequential version and pipelined version.

In the second benchmark set, we use different variants of a sequence of matrix multiplication, the `2mm` and `3mm` benchmarks of Polybench [76] followed by the similar kernel `4mm`. To be able to generate code and also to make it a more suitable application of our framework, we consider matrix multiplication as consecutive vector-matrix multiplications. Our goal in this benchmark is to illustrate the advantages and disadvantages of cross-loop pipelining compared to Polly (Pluto's scheduling algorithm). For `n=2,3,4`, the `nmm` and `nmmt` kernels are `n` consecutive matrix multiplications; in `nmmt` kernels, the second matrix is transposed beforehand. Similarly, the `ngmm` and `ngmmt` kernels are generalized matrix multiplication, wherein the loop nest, each element of the result matrix (e.g., `C[i][j]`) is multiplied by the addition of the element of the result matrix (`C`) in the same column of the next row (`C[i+1][j]`) and the element in the same row of the previous column (`C[i][j-1]`). Figure 3.13 shows the speed-ups of the programs generated by applying cross-loop pipelining (`pipeline`), Polly running with all available threads (`polly8`), and Polly running with `n` threads (`n` is the number of loop nests) (`polly`), with respect to the sequential version.

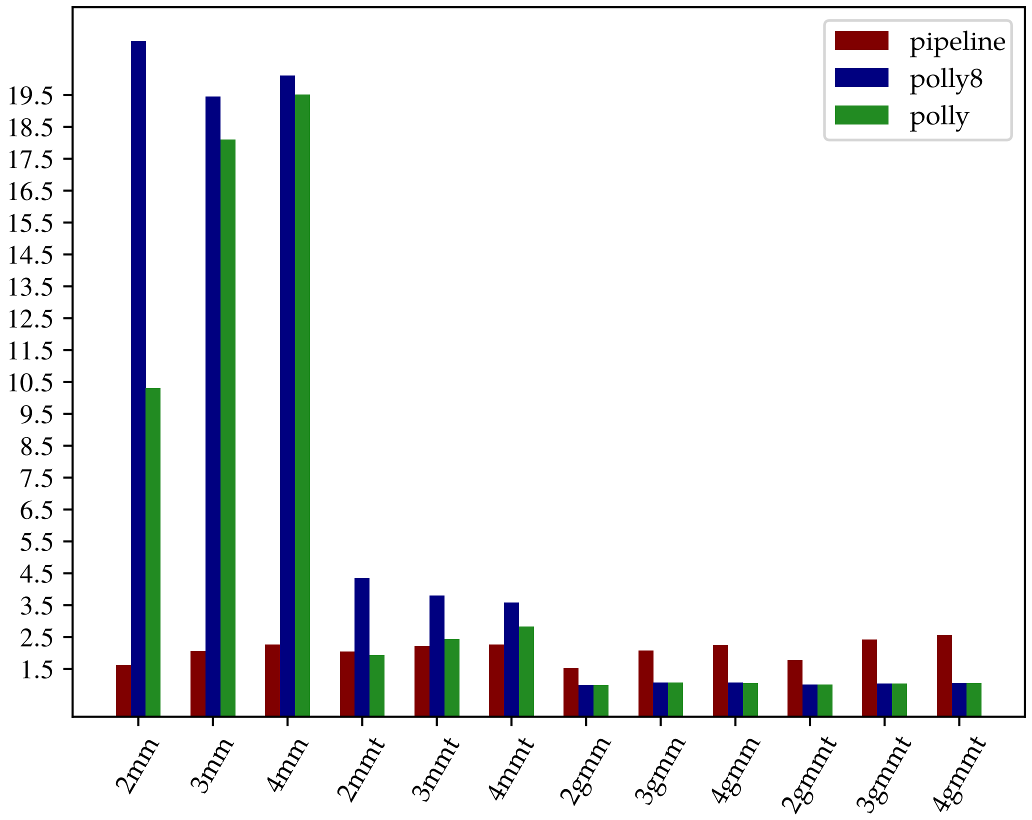


Figure 3.13: Comparing speed-up gains of Polly running by all available threads, Polly running by `n` threads (`n` is the number of loop nests), and cross-loop pipelining for variants of generalized matrix multiplication.

As Figure 3.13 shows, the speed-up we gain by using Polly is more than the gain by applying cross-loop pipelining in the `nmm` and `nmmt`. For these kernels, Polly can optimize locality by tiling, and it also parallelizes all loop nests. However, in the `gnmm` and `gnmmt` kernels, Polly cannot detect any optimization, but by using cross-loop pipelining, we can gain speed-up.

## 3.8 Conclusion and Future Works

In this work, we developed a polyhedral model-based algorithm for detecting cross-loop task parallelism. We implemented our algorithm as part of LLVM/Polly. With this prototype, we can detect parallelization opportunities that conventional polyhedral optimizers cannot detect. We exploit the detected parallelism using OpenMP task constructs. We tested our prototype on kernels with compute-intensive function calls inside for-loop nests and reported the speed-ups considering different sizes and different memory access patterns. We also considered kernels with variants of a sequence of generalized matrix multiplications and compared the speed-ups of cross-loop pipelining and Polly.

We plan to generalize our code generation phase to generate code for loops with arbitrary depth and also arbitrary number of tasks per loop. After this generalization, we can experiment with more complicated algorithms.

Also, as mentioned in Section 3.5, we assume that the write functions are injective; we want to study the possibilities of extending the transformation algorithm to relax this assumption. Moreover, we would like to extend both the transformation algorithm and the prototype to work correctly with the algorithms that detect DOACROSS parallelism in loops.

An essential factor in the performance of the final program is the granularity of the tasks. An interesting idea would be to develop an algorithm to choose a good task granularity when there are multiple choices.

In the current version, the tasking layer is independent of creating and scheduling the task. Therefore, we expect to be able to change the tasking layer from the OpenMP `task` to other platforms with minimal changes. For future works, we would like to experiment with this idea and have results in both performance improvements of different tasking platforms and how easy it is to use our method and make it compatible with other platforms.

In the current version of this work, when using the cross-loop tasking, we do not take advantage of other parallelization opportunities. We would like to know the effect of the cross-loop pipelining on the other patterns and study the results of possible combinations of this method with other optimization techniques on the performance improvements.

| Name | Specifications | Memory access |
|---|---|---|
| P1 | 2 for-loop<br>$\mathtt{num}_{1,2} = 1$ | $\mathtt{S2} \leftarrow \mathtt{A}_1\mathtt{[i][j]}$ |
| P2 | 2 for-loop<br>$\mathtt{num}_1 = 2$<br>$\mathtt{num}_2 = 6$ | $\mathtt{S2} \leftarrow \mathtt{A}_1\mathtt{[2*i][2*j]}$ |
| P3 | 3 for-loop<br>$\mathtt{num}_{1,2,3} = 1$ | $\mathtt{S2,S3} \leftarrow \mathtt{A}_1\mathtt{[i][j]}$<br>$\mathtt{S3} \leftarrow \mathtt{A}_2\mathtt{[i][j]}$ |
| P4 | 3 for-loop<br>$\mathtt{num}_{1,2} = 2$<br>$\mathtt{num}_3 = 8$ | $\mathtt{S2} \leftarrow \mathtt{A}_1\mathtt{[i+j][j]}$<br>$\mathtt{S3} \leftarrow \mathtt{A}_1\mathtt{[2*i+j][2*j]}$<br>$\mathtt{S3} \leftarrow \mathtt{A}_2\mathtt{[2*i][2*j]}$ |
| P5 | 4 for-loop<br>$\mathtt{num}_{1,2,3,4} = 1$ | $\mathtt{S2,S3,S4} \leftarrow \mathtt{A}_1\mathtt{[i][j]}$<br>$\mathtt{S3,S4} \leftarrow \mathtt{A}_2\mathtt{[i][j]}$<br>$\mathtt{S4} \leftarrow \mathtt{A}_3\mathtt{[i][j]}$ |
| P6 | 4 for-loop<br>$\mathtt{num}_1 = 1$<br>$\mathtt{num}_2 = 8$<br>$\mathtt{num}_{3,4} = 32$ | $\mathtt{S2,S3,S4} \leftarrow \mathtt{A}_1\mathtt{[i+j][j]}$<br>$\mathtt{S3,S4} \leftarrow \mathtt{A}_2\mathtt{[i][j]}$<br>$\mathtt{S4} \leftarrow \mathtt{A}_3\mathtt{[i][j]}$ |
| P7 | 4 for-loop<br>$\mathtt{num}_1 = 1$<br>$\mathtt{num}_{2,3,4} = 8$ | $\mathtt{S2,S3} \leftarrow \mathtt{A}_1\mathtt{[2*i][2*j]}$<br>$\mathtt{S3} \leftarrow \mathtt{A}_2\mathtt{[2*i][2*j]}$<br>$\mathtt{S4} \leftarrow \mathtt{A}_1\mathtt{[i][j]}$<br>$\mathtt{S4} \leftarrow \mathtt{A}_2\mathtt{[i][j]}$ |
| P8 | 4 for-loop<br>$\mathtt{num}_{1,2,3,4} = 1$ | $\mathtt{S2,S3} \leftarrow \mathtt{A}_1\mathtt{[i][j]}$<br>$\mathtt{S4} \leftarrow \mathtt{A}_2\mathtt{[i][j]}$ |
| P9 | 4 for-loop<br>$\mathtt{num}_{1,2,3,4} = 1$ | $\mathtt{S2,S4} \leftarrow \mathtt{A}_1\mathtt{[i][2*j]}$<br>$\mathtt{S3} \leftarrow \mathtt{A}_1\mathtt{[i][j]}$<br>$\mathtt{S3} \leftarrow \mathtt{A}_2\mathtt{[i][2*j]}$<br>$\mathtt{S4} \leftarrow \mathtt{A}_2\mathtt{[i][j]}$ |
| P10 | 4 for-loop<br>$\mathtt{num}_1 = 1$<br>$\mathtt{num}_{2,3,4} = 2$ | $\mathtt{S2} \leftarrow \mathtt{A}_1\mathtt{[i+j][j]}$<br>$\mathtt{S3} \leftarrow \mathtt{A}_1\mathtt{[i][j]}$<br>$\mathtt{S4} \leftarrow \mathtt{A}_2\mathtt{[i][j]}$ |

Table 3.1: Properties of the experimental data. The *Specification* column shows the number of loop nests and values of $num_i$s. The *Memory access* column shows the read accesses of each statement.

# Chapter 4

# Towards Automatic OpenMP-Aware Utilization of Fast GPU Memory

OpenMP has supported target offloading since version 4.0, and LLVM/Clang supports its compilation and optimization. There have been several optimizing transformations in LLVM aiming to improve the performance of the offloaded region, especially for targeting GPUs. Although using the memory efficiently is essential for high performance on a GPU, there has not been much work done to automatically optimize memory transactions inside the target region at compile time.

In this work, we develop an inter-procedural LLVM transformation to improve the performance of OpenMP target regions by optimizing memory transactions. This transformation pass effectively prefetches some of the read-only input data to the *fast* shared memory via compile time code injection. Especially if there is reuse, accesses to shared memory far outpace global memory accesses. Consequently, our method can significantly improve performance if the right data is placed in shared memory.

## 4.1 Overview

On modern GPUs, the global memory is off-chip with high access latency. Therefore, using the global memory efficiently and reducing the number of transactions to/from it is essential to maximize a GPU's computation capability utilization. An alternative to global memory is *shared memory* which is limited on-chip and fast memory space. The shared memory is allocated for each block (or team in OpenMP terminology), and it can be used for optimizing a program in different ways, including *prefetching*. For prefetching, a programmer first utilizes the threads in a team to copy (most often read-only) global memory content into a shared memory buffer. Then, all accesses to these locations in the global memory are replaced with accesses to the prefetched data in the faster shared memory. This method is especially beneficial if the data is reused multiple times as each access is sped up while the initial copy costs are fixed.

In this work, we develop a compiler optimization technique to improve the performance of OpenMP programs containing device offloading regions by *automatically prefetching parts of the required data to the shared memory through code injected at compile time*. In the current version of OpenMP, runtime functions and directives exist to explicitly allocate and use memory in the shared space [6]. Also, the `OpenMPOpt` pass [19], developed as a part of the LLVM framework [42], implements different OpenMP-aware optimization techniques that utilize shared memory. These have proven to effectively improve the performance of a program's target regions. We leverage the `OpenMPOpt` pass infrastructure and the LLVM/OpenMP GPU runtime functions for allocating (dynamic) shared memory for our own optimization. By identifying suitable candidate memory regions and prefetching them into the shared memory buffer automatically, we can improve the program's performance as each original load from the global memory is now significantly faster served from shared memory instead.

The rest of this chapter is organized as follows: We first introduce background information related to this work in Section 4.2. Then, we detail our method and implementation in Section 4.3. Further optimization techniques are described in Section 4.4. Our experimental results are reported and discussed in Sections 4.5 and 4.6, before we conclude the chapter in Section 4.7.

## 4.2 Background

In this section, we briefly explain some of the topics related to OpenMP target offloading and GPU programming models relevant to our work. In the context of this part of the work, and without loss of generality, we assume our target device is an NVIDIA GPU.

### 4.2.1 OpenMP target offloading support in LLVM/Clang

OpenMP has supported device offloading since version 4.0 [77], using the `target` directive. Compiling and optimizing OpenMP programs with target offload regions has been supported by LLVM/Clang since version 11 [78]. The primary approach for compiling programs with OpenMP constructs is outlining [79]. We explained the process in details in Section 2.4.1.

#### LLVM/OpenMP GPU execution model

When there is a target region in a an OpenMP program the host (CPU) is responsible for managing *kernels*, the functions to be executed on the device (GPU). In other words, the host schedules and coordinates kernels, allocates memory, and also transfers data from the host memory to the device memory.

The `teams` directive creates a league of teams. Each team begins running a single thread, *main thread*. The `parallel` directive makes the main thread to spawn some *worker threads*. Moreover, by using the `distribute for` directives, we can distribute iterations of a loop between teams and threads in each team.

In the compilation process of OpenMP programs with offloading regions, we can map each team to an *streaming multiprocessor(SM)* of the GPU (similar to the blocks in CUDA), and each thread in the team to hardware threads inside the SMs.

### OpenMP target offloading compilation in LLVM/Clang

The compiler takes some extra stages to handle device offloading. This process has two passes:

1- The first pass is a regular compilation of the host part of the program. Also, the compiler finds the offloading regions and replaces them with calls to the host runtime library functions with appropriate arguments (consisting of the kernel function identifier and its arguments). Moreover, a fallback code of the offloaded region is generated for the cases where the target does not exist or offloading fails at runtime. In this case, the offloaded region will run in the host.

2- The second pass uses the information on the offloaded regions gathered in the first pass and generates *target dependent* code for the device. It generates kernel functions for each target region, device functions, some necessary global variables, and also target dependent metadata. Also, the OpenMP runtime library is linked to the program.

In other words, the compiler first generates two separate modules, the *host module* for regions of the program running on the host and the *device module* for the offloaded region to run on the device. Therefore, the compilation flow of OpenMP programs with target offloading regions is different from the programs with other constructs. For more details on the complete compilation flow of an OpenMP program with target offloading and GPU runtime, refer to [80, 81, 82, 83].

In this work, we only need to manipulate the device module. Moreover, we focus on the *single program multiple data* (SPMD) execution mode and do not consider the generic mode [19].

To compile an OpenMP program with the `omp target` pragma, the Clang compiler outlines the target region to a kernel function and uses OpenMP runtime functions to call the kernel from the host module. Parallel loops in an OpenMP program are handled similarly. Clang first outlines the body of the parallel loop, the *parallel region*, to a function. We call this outlined function the *parallel region function*. Then, the compiler replaces the `parallel for` pragma with the call to an OpenMP runtime function, e.g., `kmpc_parallel`. This function takes a pointer to the parallel region function and all variables needed for the execution as inputs. The work-sharing loop logic is explicitly generated through more runtime calls placed inside the parallel region function. Distribution of iterations is based on the thread Id initialized by the parallel region.

As explained in Section 4.1, we only consider kernels that expose two levels of parallelism on the outer-most loop level. To handle these kernels, Clang first distributes the iterations of the "distribute component" between teams based on the number of threads in each team. Then, it parallelizes the execution of all iterations in each chunk assigned to a team by utilizing the team's threads.

```
void target_region(/* inputs */)
{
  kmpc_distribute_init(/* loop bounds */);
  // begining of the distribute region
  foreach chunk { //<- distribute loop
    // parallel region (par_region) with enclosed
    // workshare (=for) loop executed by all threads
    kmpc_parallel(par_region, ...);
  }
  // end of the distribute region
}
```

Figure 4.1: Compiler view of the kernels we consider in this paper.

Figure 4.1 shows the high-level structure of the kernel created for a target region with a single `distribute parallel for`. The compiler first inserts a call to the runtime function `kmpc_distribute_init` in the kernel to determine the lower and upper bounds of the chunks assigned to the team. After that, the compiler inserts a for loop in the kernel to iterate over the chunks. We call this loop the *distribute loop*. The compiler then inserts a call to the `kmpc_parallel` function in the distribute loop to distribute the iterations between threads of the teams. The inputs to this runtime call are the pointer to the parallel region function and its inputs, and the chunk of the work-sharing loop specified by each iteration of the distribute loop. For easier reference, we call the region of the kernel after the call to the `kmpc_distribute_static_init` function the *distribute region*.

As explained in details in 2.4.1, `OpenMPOpt` is an inter-procedural optimization (IPO) pass in LLVM, which is implemented for optimizing OpenMP GPU execution. This pass is enabled by default since LLVM 11 when compiling with `02` and `03` options. It first runs on the module and later it runs on the call graph of the program. It uses domain knowledge about OpenMP runtime calls to better optimize the LLVM-IR of the program.

### 4.2.2 CUDA Memory hierarchy

While executing, GPU threads can have access to different memory spaces. All threads across all teams have access to the *global memory*. Each team of threads has access to the *shared memory*. Finally, all threads have private *local memory*.

In this work, our focus is on the shared memory. Shared memory is an on-chip memory;

therefore, it is faster than global memory. There are two kinds of shared memory: static and dynamic. Static shared memory is used when the required size of the shared memory is known at compile time, and dynamic shared memory is used when this size is unknown at compile time.

A challenge while using the GPU's shared memory is to avoid *bank conflict*. The shared memory is managed in modules of equal size or memory banks. Different memory banks can be accessed simultaneously. However, multiple threads cannot access different locations in the same bank in parallel. Therefore, having multiple threads accessing the same memory bank causes the bank conflict problem, and the accesses will be serialized. More detailed information can be found in [84, 85].

In OpenMP, `#pragma omp allocate(X)allocator(omp_pteam_mem_alloc)` is for allocating static shared memory, where `X` should be replaced by the variable's name in the shared memory.

With LLVM, the function `llvm_omp_target_dynamic_shared_alloc` returns a pointer to the beginning of the dynamic shared memory. This dynamic shared space is "allocated" with the `LIBOMPTARGET_SHARED_MEMORY_SIZE` environment variable.

## 4.3 Implementation

This section explains the method we use and our implementation [1] details. We first explain the problem we are solving in more detail in Section 4.3.1. After that, Sections 4.3.2, 4.3.3, and 4.3.4 explain our method and implementation phases for prefetching and retrieving the memory locations correctly and efficiently. Finally, in Section 4.3.5, we discuss an application of our method to load input arrays of small sizes to the static shared memory.

### 4.3.1 Problem statement

As stated in Section 4.1, our goal is to improve the performance of the target region by prefetching some required data to the shared memory by code generation at compile time. However, the shared memory space is limited, and we cannot prefetch all the input data.

Considering the arrays accessed in the parallel region, we can categorize their access relations in two cases: (1) they are a function of the work-sharing loop's induction variable, and (2) they do not depend on this variable. The former case is more challenging for prefetching because different teams access different locations of the input array. We propose a solution for prefetching an input array in the first case under some conditions; however, the method and the implementation can be extended to relax these limitations and also to handle the second case.

[1] https://github.com/dtalaashrafi/kernel-mem-opt

This work considers one of the input arrays to the kernel that is read in the parallel region in a loop for prefetching. In other words, we consider the first input two dimensional array which its rows or its columns are read in the kernel. That means the read access instruction is surrounded by a loop-nest of the depth of at least two, where the outer-most loop is the work-sharing loop. Also, the array's access relation is a function of the work-sharing loop's induction variable and the induction variable of one of the inner loops. We call that inner loop the *access loop*. Note that the access loop can be nested in other loops, but their induction variables cannot be present in the access function. Figure 4.2 shows the work-sharing loop, the access loop, and an eligible read access for prefetching (v1). We

```
#pragma omp target teams map(to:v1[0:N*M])
{
   #pragma omp distribute parallel for
      for (int i=0; i<N; i++)  //->work-sharing loop
         for(int j=0; j<N; j++)
            for(int k=0; k<M; k++)  //->access loop
               sum += v1[i*M+k] * 3;
               //        /\->eligible access for prefetching
}
```

Figure 4.2: Example of the supported read access.

also assume there are no conditional branches in the target region, and the total number of available threads (number of teams multiplied by the number of threads per team) is equal to the number of iterations of the work-sharing loop. Furthermore, we assume the amount of shared memory usage per team does not exceed the shared space allocated for the program.

We perform shared memory prefetching in the distribute region before the distribute loop (before entering the parallel region). This way, we can prefetch those locations needed in each team to the shared memory before the threads begin computation. After that, in the parallel region, we access those locations from the shared memory instead of the global memory. We implement this procedure as a part of `OpenMPOpt` pass of LLVM, and if activated, it executes as a part of the `O3` compiler optimization passes.

### 4.3.2 Finding memory locations to prefetch

The first step is to find the memory locations that we want to prefetch for each team. In fact, our goal in this phase is to find all the global memory locations of the considered array that are read by the threads of each team in the input program. To find these memory locations, we need to have (1) the chunks of the work-sharing loop assigned to each team and (2) the memory locations accessed in each iteration of the chunks. Then, we get all the accessed locations by iterating over all locations in each iteration assigned to the team.

As we explained in Section 4.1, the compiler inserts a call to a runtime function [2] in the kernel to get the lower (`team_LB`) and upper (`team_UB`) bounds of the chunks of the work-

[2]The function is: `kmpc_distribute_init`

sharing loop assigned to each team. Therefore, we can access their values after this function call in the kernel (after line 3 in Figure 4.1).

For each team, we want to find the locations accessed in all iterations `i` of the work-sharing loop, where `i` is between `team_LB` and `team_UB`. For the kernels we consider, we can find these memory locations by having the integer values of the first location of the array that is accessed in iteration `i` ($\texttt{Base}_\texttt{i}$), the distance between two consecutive accesses in iteration `i` ($\texttt{Step}_\texttt{i}$), and the number of accessed locations by iteration `i` ($\texttt{Number}_\texttt{i}$). Having these numbers, we can find all the accessed memory locations in iteration `i` by computing:

$$(\texttt{Base}_\texttt{i} + \texttt{k} \times \texttt{Step}_\texttt{i}), \texttt{0} \leq \texttt{k} < \texttt{Number}_\texttt{i}. \tag{4.1}$$

The thread that executes iteration `i` of the work-sharing loop reads $\texttt{Number}_\texttt{i}$ locations of the array, starting from $\texttt{Base}_\texttt{i}$, and the difference between two indexes it accesses consecutively is $\texttt{Step}_\texttt{i}$. For example, consider the pseudo-code in Figure 4.3.

```
//distribute parallel loop
for(i=0; i<6; i++)
   for(j=0; j<3; j++)
      ... = A[i][j];
```

Figure 4.3: Example pseudo-code.

Figure 4.4 illustrates the memory locations accessed by threads in the teams, and also values of $\texttt{Base}_\texttt{i}$. We assume that the kernel is launched with three teams, each of them with two threads. Same colors shows the accesses in each team, and different shades are used to show accesses of various threads in the same team.

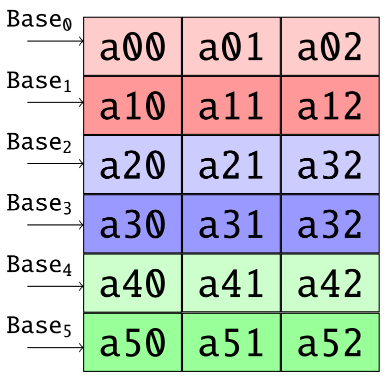


Figure 4.4: Illustration of locations accessed by each team.

We apply the LLVM scalar evolution (`scev`) [12] analysis pass to the parallel region function to get these values. Using the result of `scev` analysis, we can get the required information as objects of the LLVM `Value` class [86]. To be more precise, we begin by getting the result of scalar evolution of the array's access relation (using the `getSCEV` function) in the parallel region function and casting it to SCEVAddRecExpr [87]. After that, we call the getStart and getStepRecurrence functions of the result of the previous step. The outputs of these functions are objects of the SCEV class, and we call them BaseSCEV and StepSCEV, respectively. In the next step, we use an object of the SCEVExpander class [88] to expand code for BaseSCEV and StepSCEV.

The code expanded for the BaseSCEV is an instruction that computes the $\texttt{Base}_\texttt{i}$ values corresponding to each iteration i of the work-sharing loop. Because of the kernel's structure, the value of $\texttt{Base}_\texttt{i}$ is a function of i and the input parameters of the kernel. If the expanded instruction has any operands resulting from other instructions, we get those instructions and store them in a stack. We repeat this operation on the newly added instructions to the stack until we reach an instruction that all its operands are either integer numbers, the variable i, or the input parameters. Then, the instructions in the stack make the instruction sequence that, given the iteration number i, computes the corresponding value of $\texttt{Base}_\texttt{i}$.

The code expanded for the StepSCEV is an object of the LLVM Value class. It is the value of $\texttt{Step}_\texttt{i}$ and can be inserted into the program as an integer number.

Moreover, we can get the access loop from the SCEVAddRecExpr object we got in the first step. The value of $\texttt{Number}_\texttt{i}$ is the number of iterations of the access loop, and we can get it as an object of the LLVM Value class using the bounds of the loop. Then, it can be inserted into the program as an integer value or as a kernel input parameter. Note that the values of $\texttt{Step}_\texttt{i}$ and $\texttt{Number}_\texttt{i}$ are equal for all iterations and we can drop the i subscript.

For example, in the kernel of Figure 4.2, the code expanded for $\texttt{Base}_\texttt{i}$ is (M*i), the Step is the integer 1, and the Number is the kernel input parameter M.

Up to this point, we have generated code for computing $\texttt{Base}_\texttt{i}$, Step, and Number. Based on Equation 4.1, to have access to all the locations we want to prefetch, we need to generate code for iterating over the values of $\texttt{Base}_\texttt{i}$ corresponding to the iterations assigned to each team. Therefore, we create a loop iterating from team_LB to team_UB, and insert the instruction sequence for computing the $\texttt{Base}_\texttt{i}$ value in the body of this loop. We call this loop the *base computing loop*. We insert the base computing loop in the distribute region before the distribute loop. In the following steps, we will explain how to use the generated $\texttt{Base}_\texttt{i}$ values to prefetch their corresponding memory locations.

Notice that the function parameters in the instruction sequence for computing the $\texttt{Base}_\texttt{i}$ value and in the instruction for Number are the ones in the parallel region function. Therefore, to keep the program's semantic correctness, the next step is to replace the parameters with their correspondences in the distribute region. Also, the work-sharing loop's iteration number variable (for example, variable i in the instruction for computing Base for Figure 4.2) should be replaced with the induction variable of the base computing loop to compute $\texttt{Base}_\texttt{i}$ for the team's iteration chunk.

### 4.3.3 Loading data to the shared memory

After generating code for iterating over values of $\texttt{Base}_\texttt{i}$ for `i` iterating over the team's chunk, and also for `Step` and `Number`, the next step is to prefetch their corresponding locations from the global memory to consecutive locations in the shared memory. For this purpose, we develop a high-level function called `copy_to_shared_mem`. This function prefetches memory locations accessed in *one* iteration of the work-sharing loop. As a result, to prefetch all the locations of the considered array read by each team, we should call this function in the base computing loop. This function gets the lower bound of the team's chunk (`team_LB`), the iteration number of the work-sharing loop (the induction variable of the base computing loop `i`), a pointer to the array we want to prefetch (`V`), the value of $\texttt{Base}_\texttt{i}$, and the values of `Step` and `Number` as its inputs. Therefore, after inserting the instruction sequence for computing $\texttt{Base}_\texttt{i}$ in the base computing loop, we have all the input values and we can insert a call to the `copy_to_shared_mem` function. Figure 4.5 shows the call to this function in the transformed kernel.

```
void kernel(/* inputs */)
{
  kmpc_distribute(/* loop bounds */);
  for(int i=team_LB; i<team_UB; i++) //->base computing loop
    //instruction sequence for computing Base_i
    copy_to_shared_mem(team_LB, i, V, Base_i, Step, Number);
  // distribute region
}
```

Figure 4.5: Call to the `copy_to_shared_mem` function in the transformed kernel.

The first global memory location accessed by each team is $\texttt{V[Base}_\texttt{i}\texttt{]}$, for `i =team_LB`, and it should be stored in location `0` of the shared buffer. The global memory locations accessed in iteration `i` are $\texttt{V[(Base}_\texttt{i} + \texttt{k} \times \texttt{Step)]}$, for `k` between `0` and `Number`. The `copy_to_shared_mem` function stores these locations in consecutive indexes of the shared buffer beginning from $\texttt{S}_\texttt{i}$ to $\texttt{S}_\texttt{i} + \texttt{Number}$, where $\texttt{S}_\texttt{i}$ is the starting storing location computed for iteration `i`. To avoid over-writing already prefetched data, $\texttt{S}_\texttt{i}$ is equal to the total number of the locations prefetched in the previous iterations of the base computing loop, and it is equal to $\texttt{(i-team\_LB)}\times\texttt{Number}$. With these relations, each team requires $\texttt{(team\_UB-team\_LB)}\times\texttt{Number} \times \texttt{B}$ bits of the shared space, where `B` is the size of the prefetched array's type. Continuing on the pseudo-code in Figure 4.3, Figure 4.6 illustrates the locations of each team in the shared buffer.

Moreover, we want to prefetch data in the shared memory in parallel to get better performance. The function is called in the target region outside the parallel region. Therefore, we distribute the work between threads based on their ids. Figure 4.7 shows the `copy_to_shared_mem` function for prefetching an integer array into the shared memory.

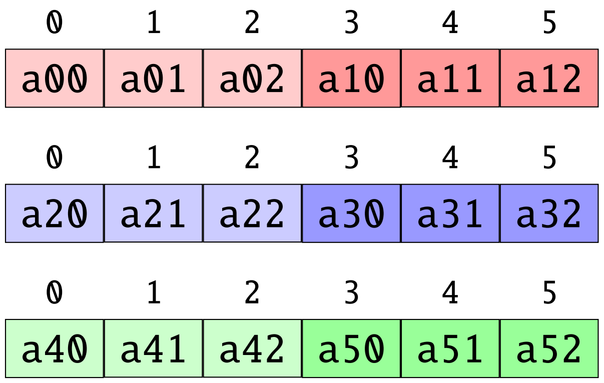


Figure 4.6: Memory locations prefeched.

```
1  void copy_to_shared_mem(int team_Lb, int i, int *V,
2                          int Base_i, int Step, int Number)
3  {
4      int Tid = get_thread_id();
5      int *DataBuf = (int*) get_dynamic_shared();
6      int bufOff = (i-teamLb)*Number+Tid;
7      int stride = omp_get_max_threads();
8      for(int k=0; k<num; k+=stride)
9          DataBuf[bufOff+k] = V[Base_i+Tid*Step+k*Step];
10 }
```

Figure 4.7: The implementation of `copy_to_shared_mem` function.

### 4.3.4 Retrieving data from the shared memory

The final step is to generate code for replacing accesses to the global memory with their corresponding accesses in the shared memory in the parallel region function.

Based on the explanation in Section 4.3.3 and line 9 of Figure 4.7, the index $(\texttt{Base}_\texttt{i} + \texttt{k} \times \texttt{Step})$ of the considered array in the global memory is stored in index $(\texttt{i}-\texttt{team\_LB}) \times \texttt{Number} + \texttt{k}$ of the shared buffer. In these relations `k` iterates from `0` to `Number` and it is the induction variable of the access loop.

To access the shared memory locations, we first generate code for getting the pointer to the dynamic shared memory in the parallel region function. Then, we use the values of `team_LB` and induction variable of the access loop in the the parallel region function and generate code for the access relation to the shared memory. Finally, we replace the access to the global memory with the (newly generated) access to the shared memory.

### 4.3.5 Static shared memory prefetching

We can take advantage of the method explained in the previous parts to prefetch input arrays of small sizes to the static shared memory. In this case, we prefetch the whole considered arrays into the shared memory for all teams. We first select input arrays that fit in the shared memory. Then, we use the approach of inserting a high-level copy function in the distribute region and replacing read instructions from the global memory with read instructions from the shared memory in the parallel region function to prefetch them.

## 4.4 Optimization

In this section we explain two optimization methods that can be applied at compile time for some special cases to improve the performance and applicability of the method explained in Section 4.3 in these cases.

### 4.4.1 Space optimization

The method explained in Section 4.3 stores every read location in the team to the shared memory. Although this method works correctly for all the cases, its time and space usage is not efficient if some iterations in the chunks assigned to each team read the same locations. In other words, if $\texttt{Base}_\texttt{i}$ is equal for some values of `i` ranging from `team_LB` to `team_UB`, the corresponding iterations of the base computing loop prefetch redundant data.

We extend the method of Section 4.3 to more efficiently handle the special case that *all* [3] iterations of the team's chunk read the exact same locations. For this purpose, we add an option that can be set at compile time to `different`, or to `same`. The `different` option is the default one and works as explained in Section 4.3. The `same` option can be used when it is known in advance that all iterations assigned to each team read the exact same locations. In this case, the base computing loop is unnecessary, and we can load all the required data for a team by finding $\texttt{Base}_\texttt{i}$ for `i=team_LB` and call the `copy_to_shared_mem` function only once. In this case, the global memory locations are prefetched in the shared buffer from index `0` to `Number-1` and we can retrieve them in the parallel region by the induction variable of the access loop.

### 4.4.2 Reducing bank conflict

We explained in Section 4.2.2 that bank conflict might happen when a kernel uses GPU's shared memory. In the kernels we consider, the number of accesses with bank conflict depends on the value of `Number`. In the worst case, where `Number` is a multiple of 32 almost all of the accesses have bank conflict and we get slowdown by prefetching. The reason is that each iteration of the base computing loop (calls to the `copy_to_shared_mem` function) starts storing data to the bank `0` of the shared memory. This causes all threads requesting from the same memory bank when retrieving data.

[3] It is better to use the default option for cases where *some* (but not all) of the team's chunk iterations read the same locations. The reason is that avoiding prefetching redundant data in these cases complicates the `copy_to_shared_mem` function in different ways (e.g., adds conditional branches to it) that degrades the performance.

To solve this problem, we use the *padding technique*. More specifically, if the value of `Number` is a multiple of 32, we store one invalid data in the shared memory by altering the `copy_to_shared_mem` function. We can apply this modification by changing line 4 of Figure 4.7 to `bufOff=(i-teamLb)×(Number+1)+Tid`. Also, to ignore the invalid location, we add `1` to the `Number` when retrieving data in the parallel region function.

## 4.5 Evaluation

In this section, we evaluate the method explained in Section 4.3 and the optimization techniques proposed in Section 4.4. We evaluate our method in terms of running time improvements. We compile all the programs with the Clang compiler, along with `-O3` and `-openmp-opt-inline-device` options. We run the experiments on an NVIDIA `GeForce MX150` GPU of `sm=6.0`. For the shared memory experiments, we allocate enough space of dynamic shared memory for the kernel. We do this by setting the value of the environment variable `LIBOMPTARGET_SHARED_MEMORY_SIZE` to the appropriate number. This number varies based on the size of the experiment.

For evaluation, we use the rectangular matrix multiplication kernel, and consider multiplying matrices with different number of rows and columns. We compare the running times of the kernel (reported by NVIDIA profiler, `nvprof`) with and without prefetching and report the speedups based on the multiplier's size. Figures 4.8 and 4.9 show the speedup we get when multiplying two matrices by prefetching *rows* of the multiplier, and Figures 4.9 and 4.11 show the speedup we get when we multiply transposes of two matrices and prefetch *columns* of the multiplier.

In both of these sets of experiments, the outermost loop is considered the work-sharing loop with `distribute parallel for` pragma, the number of threads we use is 32, and the number of teams is the number of iterations of the work-sharing loop (the number of rows in the first case, and the number of columns in the second case) divided by 32. Also, locations read by each thread in each team are different and we cannot apply the space optimization from Section 4.4.1 on these kernels (compile them with the default (`different`) option).

To examine the effect of bank conflict and padding method's effectiveness explained in Section 4.4.2, Figures 4.8 and 4.10 show the speedup we get without applying the padding method, and Figures 4.9 and 4.11 shows the speedup when we apply the padding method when prefetching data.

To test the space optimization of Section 4.4.1, we again consider the matrix multiplication kernel and we add the `collapse(2)` construct to the outermost loop. We set the number of teams and the number of threads per team equal to the number of rows of the multiplier. For these examples, the locations used by all threads in a team are similar and we compile them with the `same` option. Figure 4.12 shows the speedups we get by prefetching. In the final evaluation, we consider the XSBench [89] with small size. For different grid types (`Nuclide, Unionized, Hash`) the input data structure has three small size arrays that we

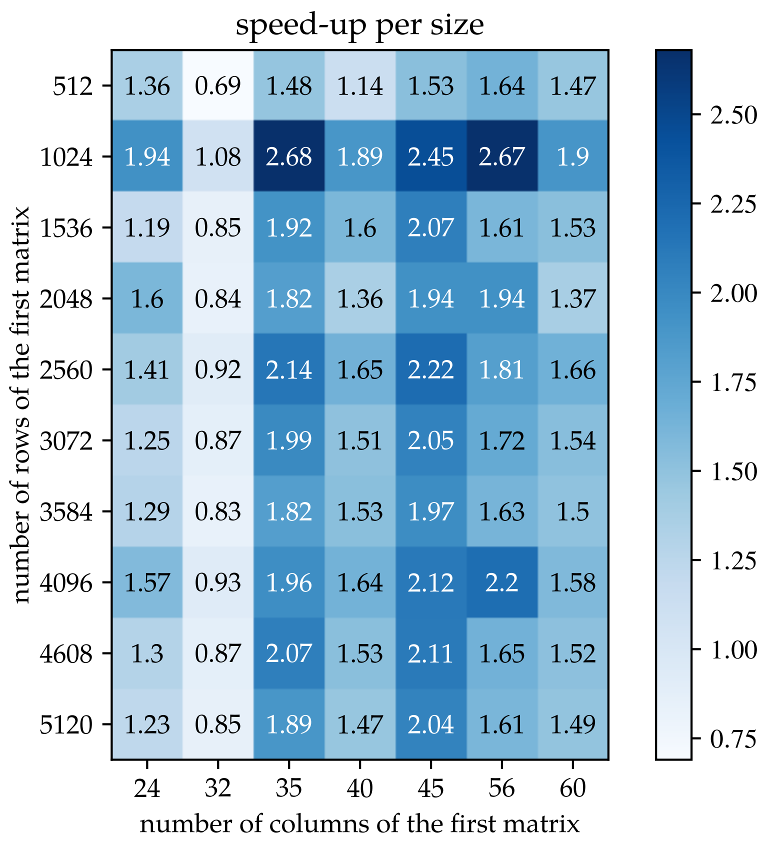


Figure 4.8: Prefetching speedup of the matrix multiplication by prefetching without padding.

can prefetch to the static shared memory, as explained in Section 4.3.5.

Figure 4.1 shows the speedup and also the total number of global memory load requests (ld_req) with and without prefetching, reported by `nvprof`. The maximum speedup is 5 percent, and the number of load requests from the global memory decreased by prefetching.

## 4.6 Analysis of the experiments

In the experiments represented in Figures 4.8, 4.9, 4.10, and 4.11, there are reuses of the multiplier's rows and columns in each team of threads, respectively. As a result, by applying the prefetching technique, we reduce the number of global memory accesses, which improves the performance of the kernels in most cases. However, as shown in 4.8 and 4.10, the kernels get slowdown when the number of rows in Figure 4.8 and the number of columns in Figure 4.10 is 32, because of shared memory bank conflict. We can improve the performance in these cases by applying the padding method, as explained in Section 4.4.2. The effectiveness of the padding method is shown in Figures 4.9 and 4.11.

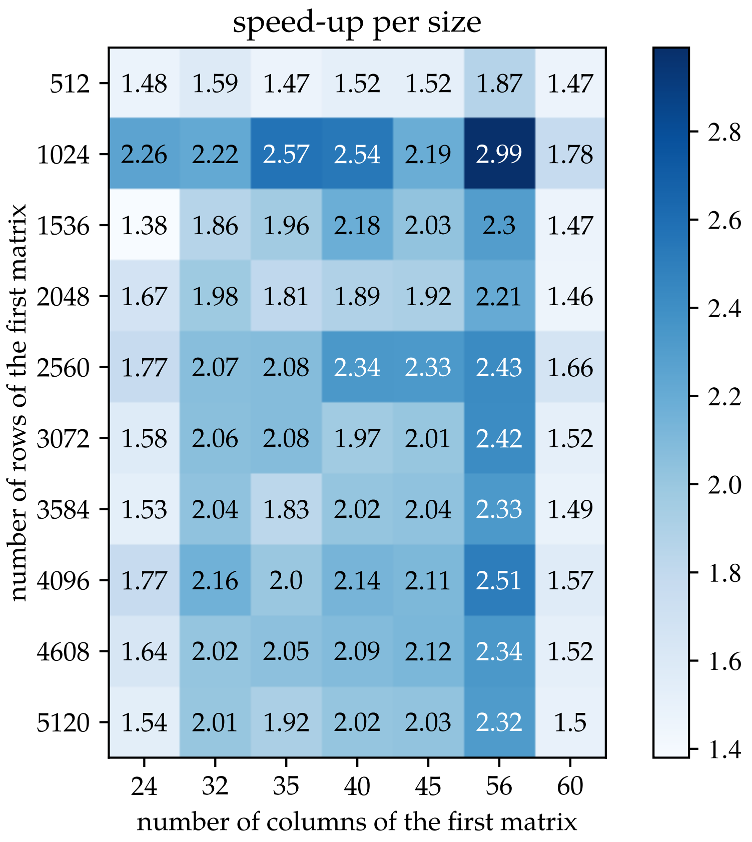


Figure 4.9: Prefetching speedup of the matrix multiplication by prefetching with padding.

Similarly, in the experiment represented in Figure 4.12, there are reuses of the same row of the multiplier in each team of threads. By prefetching and applying the space optimization explained in Section 4.4.1 the kernels gain speedup in all cases.

In our experiment with XSBench, represented in Figure 4.1, accessing the prefetched arrays is not time-consuming compared to the other steps of the algorithms. Although prefetching works as expected and reduces the number of load requests from the global memory, the kernels do not gain significant speedup.

## 4.7 Conclusion and future works

In this work, we used the infrastructure of the `OpenMPOpt` pass to develop an LLVM pass to optimize offloaded regions of OpenMP when targeting GPUs by prefetching data to the shared memory. The method can be applied on the kernels with some properties (ref. Section 4.3.1) and we show that it improves the performance of these kernels. We also propose solutions for more efficient use of shared space and for avoiding bank conflicts.

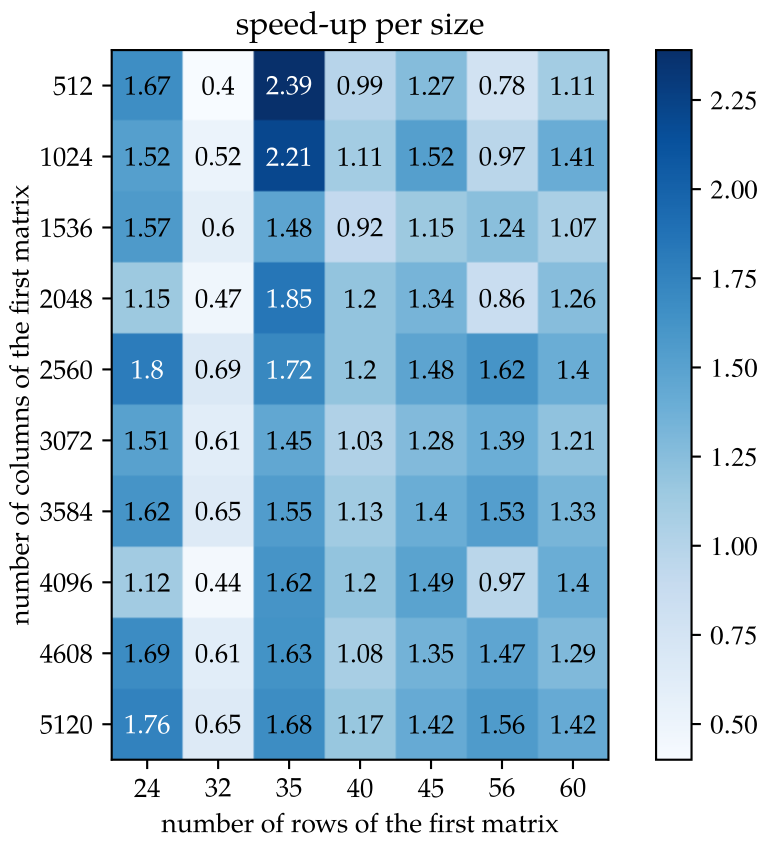


Figure 4.10: Prefetching speedup of the matrix multiplication by prefetching without padding.

For future works, we plan to improve the applicability of the method by supporting more general kernels and by relaxing the limitations explained before. For instance, we want to improve our process to handle functions that use other OpenMP constructs. Moreover, in the current version, we only prefetch one of the read-only arrays. An interesting idea is to improve the algorithm to prefetch more than one array or choose the best one for prefetching.

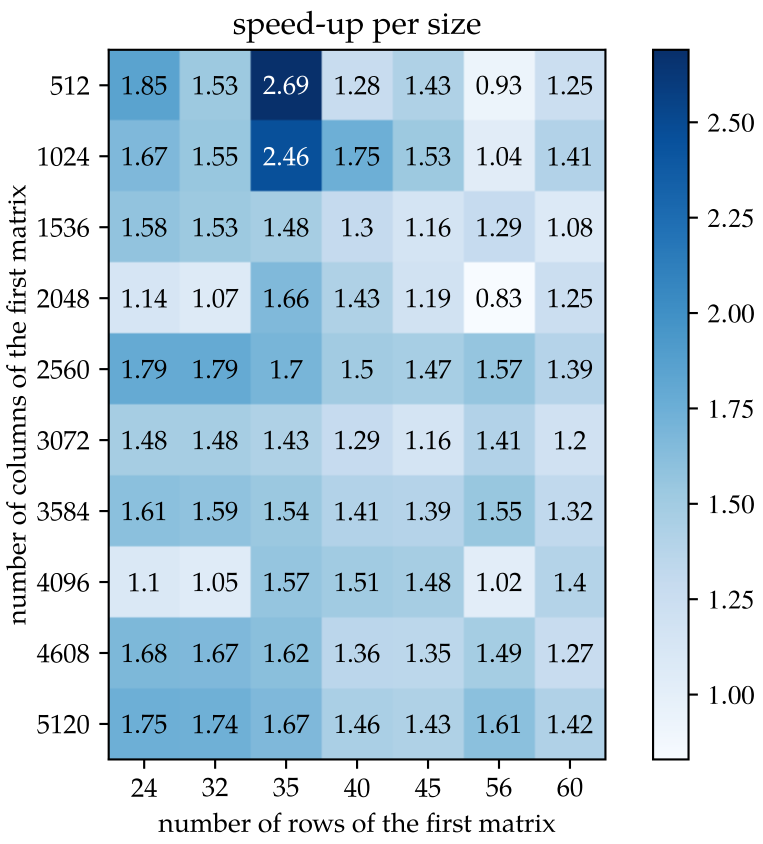
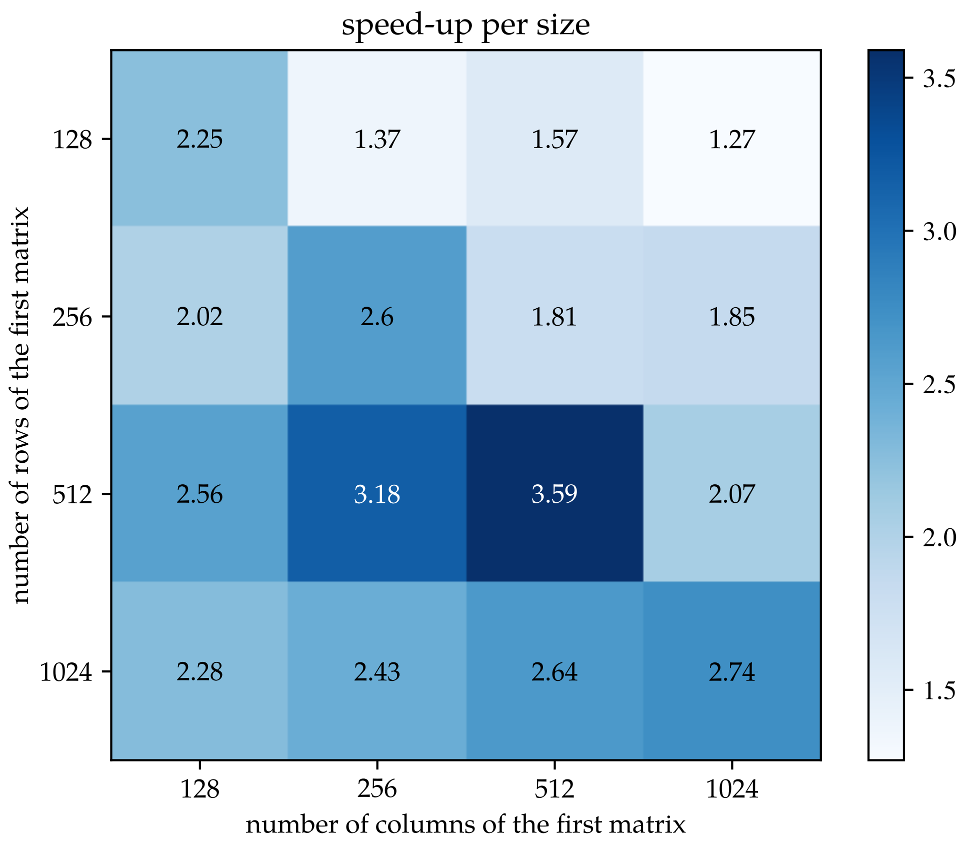


Figure 4.11: Prefetching speedup of the matrix transpose multiplication by applying prefetching with padding.

| grid type | speedup | ld_req | ld_req with prefetching |
|---|---|---|---|
| `Nuclide` | 1.05 | 1 885 268 743 | 1 754 280 336 |
| `Unionized` | 1.03 | 1 066 897 410 | 935 422 655 |
| `Hash` | 1.05 | 1 331 110 977 | 1 199 607 982 |

Table 4.1: Prefetching speedup and comparing number of global memory load requests in XSBench.



| | 128 | 256 | 512 | 1024 |
|---|---|---|---|---|
| 128 | 2.25 | 1.37 | 1.57 | 1.27 |
| 256 | 2.02 | 2.6 | 1.81 | 1.85 |
| 512 | 2.56 | 3.18 | 3.59 | 2.07 |
| 1024 | 2.28 | 2.43 | 2.64 | 2.74 |

Figure 4.12: Prefetching speedup of matrix multiplication with collapsed loops.

# Chapter 5

# Complexity Estimates for Fourier-Motzkin Elimination

In this work, we propose an efficient method for removing all redundant inequalities generated by Fourier-Motzkin elimination. This method is based on an improved version of Balas' work and can also be used to remove all redundant inequalities in the input system. Moreover, our method only uses arithmetic operations on matrices and avoids resorting to linear programming techniques. Algebraic complexity estimates and experimental results show that our method outperforms alternative approaches, in particular those based on linear programming and the simplex algorithm.

## 5.1 Introduction

Polyhedral sets play an important role in computational sciences. For instance, they are used to model, analyze, transform and schedule for-loops of computer programs; we refer to the articles [90, 91, 92, 93, 94, 95, 96]. Of prime importance are the following operations on polyhedral sets: conversion between H-representation and V-representation (performed, for instance, by the double description method); and projection, as performed by Fourier-Motzkin elimination.

Fourier-Motzkin elimination is an algorithmic tool for projecting a polyhedral set onto a linear subspace. It was proposed independently by Joseph Fourier and Theodore Motzkin, respectively in 1827 and 1936. See the paper [97] of George Danzing and Section 12.2 of the book [98] of Alexander Schrijver, for a presentation of Fourier-Motzkin elimination. As an example, for eliminating variable $t_1$ from the inequality system 5.1, one step of the Fourier-Motzkin elimination algorithm finds a positive linear combination of the inequality $a_1$ with the inequality $a_2$ and the inequality $a_3$, such that the coefficient of $t_1$ is zero in the result inequality. The output is the inequality system 5.2.

$$A = \begin{cases} a_1 : 3t_1 - 2t_2 + t_3 \leq 7 \\ a_2 : -2t_1 + 2t_2 - t_3 \leq 12 \\ a_3 : -4t_1 + t_2 + 3t_3 \leq 15 \end{cases} \tag{5.1}$$

$$A' = \begin{cases} 2t_2 - t_3 \leq 50 \\ -5t_2 - 13t_3 \leq 73 \end{cases} \tag{5.2}$$

The original version of this algorithm produces large amounts of redundant inequalities and has a double exponential algebraic complexity. Removing all these redundancies is equivalent to giving the so-called *minimal representation* of the projection of a polyhedron. Leonid Khachiyan explained in [99] how linear programming (LP) could be used to remove all redundant inequalities, thereby reducing the cost of Fourier-Motzkin elimination to a number of machine word operations singly exponential in the dimension of the ambient space. However, Khachiyan did not state a more precise running time estimate taking into account the characteristics of the polyhedron being projected, such as the number of its facets.

As we shall prove in this work, rather than using linear programming one may use only matrix arithmetic, increasing the theoretical and practical efficiency of Fourier-Motzkin elimination while still producing an irredundant representation of the projected polyhedron.

Other algorithms for projecting polyhedral sets remove some (but not all) redundant inequalities with the help of extreme rays: see the work of David A. Kohler [22]. As observed by Jean-Louis Imbert in [23], the method he proposed in that paper and that of Sergei N. Chernikov in [24] are equivalent. On the topic of finding extreme rays of a polyhedral set in H-representation, see Natalja V. Chernikova [100], Hervé Le Verge [101] and Komei Fukuda [21]. These methods are very effective in practice, but none of them can remove all redundant inequalities generated by Fourier-Motzkin elimination.

Fourier-Motzkin elimination is well suited for projecting a polyhedron, described by its facets (given by linear inequalities), onto different sub-spaces. Our work is about projecting polyhedral sets to lower dimensions, eliminating one variable after another, thanks to the Fourier-Motzkin elimination algorithm as described in Schrijver's book [98]. In fact, our goal is to find the minimal representations of all of the successive projections of a given polyhedron (in H-representation, thus given by linear inequalities), by eliminating variables one after another, using the Fourier-Motzkin elimination algorithm.

### 5.1.1 Polyhedral cones and polyhedral sets

We explained some basic concepts and definitions related to the polyhedral sets in Section 2.1. In this subsection, we explain more advanced subjects and theorems associated with the polyhedral theory, that are required for presenting the ideas in this work.

In this Chapter, we use bold letters, e.g. $\mathbf{v}$, to denote vectors and we use capital letters, e.g. $A$, to denote matrices. Also, we assume that vectors are column vectors. For row vectors, we use the transposition notation, as in $A^t$ for the transposition of a matrix $A$. For a matrix $A$ and an integer $k$, $A_k$ is the row of index $k$ in $A$. Also, if $K$ is a set of integers, $A_K$ denotes the sub-matrix of $A$ with row indices in $K$.

We begin this section with the fundamental theorem of linear inequalities.

**Theorem 5.1.1 ([98])** *Let $\mathbf{a}_1, \cdots, \mathbf{a}_m$ be a set of linearly independent vectors in $\mathbb{Q}^n$. Also, let $\mathbf{b}$ be a vector in $\mathbb{Q}^n$. Then, exactly one of the following holds:*

$(i)$ *the vector $\mathbf{b}$ is a non-negative linear combination of $\mathbf{a}_1, \ldots, \mathbf{a}_m$. In other words, there exist positive numbers $y_1, \ldots, y_m$ such that we have $\mathbf{b} = \sum_{i=1}^{m} y_i \mathbf{a}_i$, or,*

$(ii)$ *there exists a vector $\mathbf{d} \in \mathbb{Q}^n$, such that both $\mathbf{d}^t\mathbf{b} < 0$ and $\mathbf{d}^t\mathbf{a}_i \geq 0$ hold for all $1 \leq i \leq m$.*

**Definition 5.1.1.** A subset of points $C \subseteq \mathbb{Q}^n$ is called a *cone* if for each $\mathbf{x} \in C$ and each real number $\lambda \geq 0$ we have $\lambda\mathbf{x} \in C$. A cone $C \subseteq \mathbb{Q}^n$ is called *convex* if for all $\mathbf{x}, \mathbf{y} \in C$, we have $\mathbf{x} + \mathbf{y} \in C$. If $C \subseteq \mathbb{Q}^n$ is a convex cone, then its elements are called the *rays* of $C$. For two rays $\mathbf{r}$ and $\mathbf{r}'$ of $C$, we write $\mathbf{r}' \simeq \mathbf{r}$ whenever there exists $\lambda \geq 0$ such that we have $\mathbf{r}' = \lambda\mathbf{r}$.

**Definition 5.1.2.** A subset $H \subseteq \mathbb{Q}^n$ is called a *hyperplane* if $H = \{\mathbf{x} \in \mathbb{Q}^n \mid \mathbf{a}^t\mathbf{x} = 0\}$ for some non-zero vector $\mathbf{a} \in \mathbb{Q}^n$.

**Definition 5.1.3.** A *half-space* is a set of the form $\{x \in \mathbb{Q}^n \mid \mathbf{a}^t x \leq 0\}$ for a some vector $\mathbf{a} \in \mathbb{Q}^n$.

**Definition 5.1.4.** A cone $C \subseteq \mathbb{Q}^n$ is a *polyhedral cone* if it is the intersection of finitely many half-spaces, that is, $C = \{x \in \mathbb{Q}^n \mid Ax \leq \mathbf{0}\}$ for some matrix $A \in \mathbb{Q}^{m\times n}$.

**Definition 5.1.5.** Let $\{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$ be a set of vectors in $\mathbb{Q}^n$. The *cone generated* by $\{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$, denoted by $\mathsf{Cone}(\mathbf{x}_1, \cdots, \mathbf{x}_m)$, is the smallest convex cone containing those vectors. In other words, we have $\mathsf{Cone}(\mathbf{x}_1, \ldots, \mathbf{x}_m) = \{\lambda_1\mathbf{x}_1 + \cdots + \lambda_m\mathbf{x}_m \mid \lambda_1 \geq 0, \ldots, \lambda_m \geq 0\}$. A cone obtained in this way is called a *finitely generated cone*. With the following lemma, which is a consequence of the fundamental Theorem of linear inequalities, we can say that the two concepts of polyhedral cones and finitely generated cones are equivalent, see [98].

**Theorem 5.1.2 (Minkowski-Weyl theorem)** *A convex cone is polyhedral if and only if it is finitely generated.*

**Definition 5.1.6.** For two subsets $P$ and $Q$ of $\mathbb{Q}^n$, their *Minkowski sum*, denoted by $P + Q$, is the subset of $\mathbb{Q}^n$ defined as $\{p + q \mid (p, q) \in P \times Q\}$.

The following lemma, which is another consequence of the fundamental theorem of linear inequalities, helps us to determine the relation between polytopes and polyhedra. The proof can be found in [98]

**Lemma 5.1.3 (Decomposition theorem for convex polyhedra)** *A subset $P$ of $\mathbb{Q}^n$ is a convex polyhedron if and only if it can be written as the Minkowski sum of a finitely generated cone and a polytope.*

Another consequence of the fundamental theorem of inequalities, is the famous Farkas lemma. This lemma has different variants. Here we only mention a variant from [98].

**Lemma 5.1.4 (Farkas' lemma)** *Let $A \in \mathbb{Q}^{m\times n}$ be a matrix and $\mathbf{b} \in \mathbb{Q}^m$ be a vector. Then, there exists a vector $\mathbf{t} \in \mathbb{Q}^n$, $\mathbf{t} \geq \mathbf{0}$ satisfying $A\mathbf{t} = \mathbf{b}$ if and only if $\mathbf{y}^t\mathbf{b} \geq 0$ holds for each vector $\mathbf{y} \in \mathbb{Q}^m$ such that we have $\mathbf{y}^t A \geq 0$.*

A consequence of Farkas' lemma is the following criterion for testing whether an inequality $\mathbf{c}^t\mathbf{x} \leq c_0$ is *redundant* w.r.t. a polyhedron representation $A\mathbf{x} \leq \mathbf{b}$, that is, whether $\mathbf{c}^t\mathbf{x} \leq c_0$ is implied by $A\mathbf{x} \leq \mathbf{b}$.

**Lemma 5.1.5 (Redundancy test criterion)** *Let $\mathbf{c} \in \mathbb{Q}^n$, $c_0 \in \mathbb{Q}$, $A \in \mathbb{Q}^{m\times n}$ and $\mathbf{b} \in \mathbb{Q}^m$. Then, the inequality $\mathbf{c}^t\mathbf{x} \leq c_0$ is redundant w.r.t. the system of inequalities $A\mathbf{x} \leq \mathbf{b}$ if and only if there exists a vector $\mathbf{t} \geq \mathbf{0}$ and a number $\lambda \geq 0$ satisfying $\mathbf{c}^t = \mathbf{t}^t A$ and $c_0 = \mathbf{t}^t\mathbf{b} + \lambda$.*

**Definition 5.1.7.** An inequality $\mathbf{a}^t\mathbf{x} \leq b$ (with $\mathbf{a} \in \mathbb{Q}^n$ and $b \in \mathbb{Q}$) is an implicit equation of the inequality system $A\mathbf{x} \leq \mathbf{b}$ if $\mathbf{a}^t\mathbf{x} = b$ holds for all $\mathbf{x}$ satisfiying the inequality system.

**Definition 5.1.8.** A representation of a polyhedron is *minimal* if no inequality of that representation is implied by the other inequalities of that representation.

**Definition 5.1.9.** The *characteristic cone* of $P$ is the polyhedral cone denoted by $\mathsf{CharCone}(P)$ and defined by $\mathsf{CharCone}(P) := \{\mathbf{y} \in \mathbb{Q}^n \mid \mathbf{x} + \mathbf{y} \in P, \forall \mathbf{x} \in P\} = \{\mathbf{y} \mid A\mathbf{y} \leq \mathbf{0}\}$.

**Definition 5.1.10.** The *linearity space* of the polyhedron $P$ is the linear space denoted by $\mathsf{LinearSpace}(P)$ and defined as $\mathsf{CharCone}(P) \cap -\mathsf{CharCone}(P) = \{\mathbf{y} \mid A\mathbf{y} = \mathbf{0}\}$, where $-\mathsf{CharCone}(P)$ is the set of the $-\mathbf{y}$ for $\mathbf{y} \in \mathsf{CharCone}(P)$. The polyhedron $P$ is *pointed* if its linearity space is $\{\mathbf{0}\}$.

**Lemma 5.1.6** *The polyhedron $P$ is pointed if and only if the matrix $A$ is full column rank.*

**Definition 5.1.11.** The *dimension* of the polyhedron $P$, denoted by $\dim(P)$, is $n - r$, where $n$ is dimension[1] of the ambient space (that is, $\mathbb{Q}^n$) and $r$ is the maximum number of implicit equations defined by linearly independent vectors. We say that $P$ is *full-dimensional* whenever $\dim(P) = n$ holds. In another words, $P$ is full-dimensional if and only if it does not have any implicit equations.

**Definition 5.1.12.** A subset $F$ of the polyhedron $P$ is called a *face* of $P$ if $F$ equals $\{\mathbf{x} \in P \mid A_{\text{sub}}\mathbf{x} = \mathbf{b}_{\text{sub}}\}$ for a sub-matrix $A_{\text{sub}}$ of $A$ and a sub-vector $\mathbf{b}_{\text{sub}}$ of $\mathbf{b}$.

**Remark 5.1.7** *It is obvious that every face of a polyhedron is also a polyhedron. Moreover, the intersection of two faces $F_1$ and $F_2$ of $P$ is another face $F$, which is either $F_1$, or $F_2$, or a face with a dimension less than $\min(\dim(F_1), \dim(F_2))$. Note that $P$ and the empty set are faces of $P$.*

**Definition 5.1.13.** A face of $P$, distinct from $P$ and of maximal dimension is called a *facet* of $P$.

[1]Of course, this notion of dimension coincides with the topological one, that is, the maximum dimension of a ball contained in $P$.

**Remark 5.1.8** *It follows from the previous remark that $P$ has at least one facet and that the dimension of any facet of $P$ is equal to $\dim(P) - 1$. When $P$ is full-dimensional, there is a one-to-one correspondence between the inequalities in a minimal representation of $P$ and the facets of $P$. From this latter observation, we deduce that the minimal representation of a full dimensional polyhedron is unique up to multiplying each of the defining inequalities by a positive constant.*

**Definition 5.1.14.** A non-empty face that does not contain any other face of a polyhedron is called a *minimal face* of that polyhedron. Specifically, if the polyhedron $P$ is pointed, each minimal face of $P$ is just a point and is called an *extreme point* or *vertex* of $P$.

**Definition 5.1.15.** Let $C$ be a cone such that $\dim(\mathsf{LinearSpace}(C)) = t$. Then, a face of $C$ of dimension $t + 1$ is called a minimal proper face of $C$. In the special case of a pointed cone, that is, whenever $t = 0$ holds, the dimension of a minimal proper face is 1 and such a face is called an *extreme ray* . We call an *extreme ray* of the polyhedron $P$ any extreme ray of its characteristic cone $\mathsf{CharCone}(P)$. We say that two extreme rays $\mathbf{r}$ and $\mathbf{r}'$ of the polyhedron $P$ are *equivalent*, and denote it by $\mathbf{r} \simeq \mathbf{r}'$, if one is a positive multiple of the other. When we consider the set of all extreme rays of the polyhedron $P$ (or the polyhedral cone $C$) we will only consider one ray from each equivalence class.

**Lemma 5.1.9 (Generating a cone from its extreme rays)** *A pointed cone $C$ can be generated by its extreme rays, that is, we have $C = \{\mathbf{x} \in \mathbb{Q}^n \mid (\exists \mathbf{c} \geq \mathbf{0})\ \mathbf{x} = R\mathbf{c}\}$, where the columns of $R$ are the extreme rays of $C$.*

**Remark 5.1.10** *From the previous definitions and lemmas, we derive the following observations:*

1. *the number of extreme rays of each cone is finite,*
2. *the set of all extreme rays is unique up to multiplication by a scalar, and,*
3. *all members of a cone are positive linear combination of extreme rays.*

We denote by $\mathsf{ExtremeRays}(C)$ the set of extreme rays of the cone $C$. Recall that all cones considered here are polyhedral.

The following, see [102, 103], is helpful in the analysis of algorithms manipulating extreme rays of cones and polyhedra.

**Lemma 5.1.11 (Maximum number of extreme rays)** *Let $E(C)$ be the number of extreme rays of a polyhedral cone $C \in \mathbb{Q}^n$ with $m$ facets. Then, we have:*

$$E(C) \leq \binom{m - \lfloor \frac{n+1}{2} \rfloor}{m-1} + \binom{m - \lfloor \frac{n+2}{2} \rfloor}{m-n} \leq m^{\lfloor \frac{n}{2} \rfloor}. \tag{5.3}$$

From Remark 5.1.10, it appears that extreme rays are important characteristics of polyhedral cones. Therefore, two algorithms have been developed in [21] to check whether a member of a cone is an extreme ray or not. For explaining these algorithms, we need the following definition.

**Definition 5.1.16.** For a cone $C = \{\mathbf{x} \in \mathbb{Q}^n \mid A\mathbf{x} \leq \mathbf{0}\}$ and $\mathbf{t} \in C$, we define the *zero set* $\zeta_A(\mathbf{t})$ as the set of row indices $i$ such that $A_i\mathbf{t} = 0$, where $A_i$ is the $i$-th row of $A$. For simplicity, we use $\zeta(\mathbf{t})$ instead of $\zeta_A(\mathbf{t})$ when there is no ambiguity. Consider a cone $C = \{\mathbf{x} \in \mathbb{Q}^n \mid A'\mathbf{x} = \mathbf{0},\ A''\mathbf{x} \leq \mathbf{0}\}$ where $A'$ and $A''$ are two matrices such that the system $A''\mathbf{x} \leq \mathbf{0}$ has no implicit equations. The proofs of the following lemmas are straightforward and can be found in [21] and [103].

**Lemma 5.1.12 (Algebraic test for extreme rays)** *Let* $\mathbf{r} \in C$*. Then, the ray* $\mathbf{r}$ *is an extreme ray of* $C$ *if and only if we have* $\operatorname{rank}\left(\begin{bmatrix} A' \\ A''_{\zeta(r)} \end{bmatrix}\right) = n - 1$.

**Lemma 5.1.13 (Combinatorial test for extreme rays)** *Let* $\mathbf{r} \in C$*. Then, the ray* $\mathbf{r}$ *is an extreme ray of* $C$ *if and only if for any ray* $\mathbf{r}'$ *of* $C$ *such that* $\zeta(\mathbf{r}) \subseteq \zeta(\mathbf{r}')$ *holds we have* $\mathbf{r}' \simeq \mathbf{r}$.

**Definition 5.1.17.** For the given polyhedral cone $C \subseteq \mathbb{Q}^n$, the *polar cone* induced by $C$ is denoted $C^*$ and given by:

$$C^* = \{\mathbf{y} \in \mathbb{Q}^n \mid \mathbf{y}^t\mathbf{x} \leq \mathbf{0}, \forall \mathbf{x} \in C\}.$$

The following lemma shows an important property of the polar cone of a polyhedral cone. The proof can be found in [98].

**Lemma 5.1.14 (Polarity property)** *For a given cone* $C \in \mathbb{Q}^n$*, there is a one-to-one correspondence between the faces of* $C$ *of dimension* $k$ *and the faces of* $C^*$ *of dimension* $n - k$*. In particular, there is a one-to-one correspondence between the facets of* $C$ *and the extreme rays of* $C^*$*.*

Each polyhedron $P$ can be embedded in a higher-dimensional cone, called the homogenized cone associated with $P$.

**Definition 5.1.18.** The *homogenized cone* of the polyhedron $P = \{\mathbf{x} \in \mathbb{Q}^n \mid A\mathbf{x} \leq \mathbf{b}\}$ is denoted by $\mathsf{HomCone}(P)$ and defined by:

$$\mathsf{HomCone}(P) = \{(\mathbf{x}, x_{\text{last}}) \in \mathbb{Q}^{n+1} \mid C[\mathbf{x}^t, x_{\text{last}}]^t \leq 0\},$$

where

$$C = \begin{bmatrix} A & -\mathbf{b} \\ \mathbf{0}^t & -1 \end{bmatrix}$$

is an $(m + 1) \times (n + 1)$-matrix, if $A$ is an $(m \times n)$-matrix.

**Lemma 5.1.15 (H-representation correspondence)** *An inequality* $A_i\mathbf{x} \leq b_i$ *is redundant in* $P$ *if and only if the corresponding inequality* $A_i\mathbf{x} - b_i x_{\text{last}} \leq 0$ *is redundant in* $\mathsf{HomCone}(P)$.

**Theorem 5.1.16 (Extreme rays of the homogenized cone)** *Every extreme ray of the homogenized cone* $\mathsf{HomCone}(P)$ *associated with the polyhedron* $P$ *is either of the form* $(\mathbf{x}, 0)$ *where* $\mathbf{x}$ *is an extreme ray of* $P$*, or* $(\mathbf{x}, 1)$ *where* $\mathbf{x}$ *is an extreme point of* $P$*.*

## 5.1.2 Polyhedral computations

In this section, we review two of the most important algorithms for polyhedral computations: the double description algorithm (DD for short) and the Fourier-Motzkin elimination algorithm (FME for short).

A polyhedral cone $C$ can be represented either as an intersection of finitely many half-spaces (thus using the so-called *H-representation* of $C$) or as by its extreme rays (thus using the so-called *V-representation* of $C$); the DD algorithm produces one representation from the other. We shall explain the version of the DD algorithm which takes as input the H-representation of $C$ and returns as output the V-representation of $C$.

The FME algorithm performs a standard projection of a polyhedral set to lower dimension subspace. In algebraic terms, this algorithm takes as input a polyhedron $P$ given by a system of linear inequalities (thus an H-representation of $P$) in $n$ variables $x_1 < x_2 < \cdots < x_n$ and computes the H-representation of the projection of $P$ on $x_1 < \cdots < x_k$ for some $1 \leq k < n$.

**The double description method**

We know from 5.1.2 that any polyhedral cone $C = \{\mathbf{x} \in \mathbb{Q}^n \ | A\mathbf{x} \leq \mathbf{0}\}$ can be generated by finitely many vectors, say $\{\mathbf{x}_1, \ldots, \mathbf{x}_q\} \in \mathbb{Q}^n$. Moreover, from 5.1.9 we know that if $C$ is pointed, then it can be generated by its extreme rays, that is, $C = \mathsf{Cone}(R)$ where $R = [\mathbf{x}_1, \ldots, \mathbf{x}_q]$. Therefore, we have two possible representations for the pointed polyhedral cone $C$:

**H-representation:** as the intersection of finitely many half-spaces, or equivalently, with a system of linear inequalities $A\mathbf{x} \leq \mathbf{0}$;

**V-representation:** as a linear combination of finitely many vectors, namely $\mathsf{Cone}(R)$, where $R$ is a matrix, the columns of which are the extreme rays of $C$.

We say that the pair $(A, R)$ is a *Double Description Pair* or simply a *DD pair* of $C$. We call $A$ a *representation matrix* of $C$ and $R$ a *generating matrix* of $C$. We call $R$ (resp. $A$) a *minimal generating (resp. representing) matrix* when no proper sub-matrix of $R$ (resp. $A$) is generating (resp. representing) $C$.

It is important to notice that, for some queries in polyhedral computations, the output can be calculated in polynomial time using one representation (either a representation matrix or a generating matrix) while it would require exponential time using the other representation.

For example, we can compute in polynomial time the intersection of two cones when they are in H-representation but the same problem would be harder to solve when the same cones are in V-representation. Therefore, it is important to have a procedure to convert between these two representations, which is the focus of the articles [100] and [103].

We will explain this procedure, which is known as the *double description method* as well as *Chernikova's algorithm*. This algorithm takes a cone in H-representation as input and returns a V-representation of the same cone as output. In other words, this procedure finds the extreme rays of a polyhedral cone, given by its representation matrix. It has been proven that this procedure runs in single exponential time. To the best of our knowledge, the most practically efficient variant of this procedure has been proposed by Fukuda in [21] and is implemented in the CDD library. We shall explain his approach here and analyze its algebraic complexity. Before presenting Fukuda's algorithm, we need a few more definitions and results. In this section, we assume that the input cone $C$ is pointed.

The *double description method* works in an incremental manner. Denoting by $H_1, \ldots, H_m$ the half-spaces corresponding to the inequalities of the H-representation of $C$, we have $C = H_1 \cap \cdots \cap H_m$. Let $1 < i \leq m$ and assume that we have computed the extreme rays of the cone $C^{i-1} := H_1 \cap \cdots \cap H_{i-1}$. Then the $i$-th iteration of the DD method deduces the extreme rays of $C^i$ from those of $C^{i-1}$ and $H_i$.

Assume that the half-spaces $H_1, \ldots, H_m$ are numbered such that $H_i$ is given by $A_i\mathbf{x} \leq 0$, where $A_i$ is the $i$-th row of the representing matrix $A$. We consider the following partition of $\mathbb{Q}^n$:

$H_i^+ = \{\mathbf{x} \in \mathbb{Q}^n \mid A_i\mathbf{x} > 0\}$, $H_i^0 = \{\mathbf{x} \in \mathbb{Q}^n \mid A_i\mathbf{x} = 0\}$ and $H_i^- = \{\mathbf{x} \in \mathbb{Q}^n \mid A_i\mathbf{x} < 0\}$.

Assume that we have found the DD-pair $(A^{i-1}, R^{i-1})$ of $C^{i-1}$. Let $J$ be the set of the column indices of $R^{i-1}$. We use the above partition $\{H_i^+, H_i^0, H_i^-\}$ to partition $J$ as follows:

$J_i^+ = \{j \in J \mid \mathbf{r}_j \in H^+\}$, $J_i^0 = \{j \in J \mid \mathbf{r}_j \in H^0\}$ and $J_i^- = \{j \in J \mid \mathbf{r}_j \in H^-\}$,
where $\{\mathbf{r}_j \mid j \in J\}$ is the set of the columns of $R^{i-1}$, hence the set of the extreme rays of $C^{i-1}$.

For future reference, let us denote by $\mathsf{partition}(\mathsf{J}, \mathsf{A_i})$ the function which returns $J^+, J^0, J^-$ as defined above. The proof can be found in [21].

**Lemma 5.1.17 (Double description method)** *Let $J' := J^+ \cup J^0 \cup (J^+ \times J^-)$. Let $R^i$ be the $(n \times |J'|)$-matrix consisting of*

- *the columns of $R^{i-1}$ with index in $J^+ \cup J^0$, followed by*
- *the vectors $\mathbf{r'}_{(j,j')}$ for $(j, j') \in (J^+ \times J^-)$, where*

$$\mathbf{r'}_{(j,j')} = (A_i\mathbf{r}_j)\mathbf{r}_{j'} - (A_i\mathbf{r}_{j'})\mathbf{r}_j,$$

*Then, the pair $(A^i, R^i)$ is a DD pair of $C^i$.*

The most efficient way to start the incremental process is to choose the largest sub-matrix of $A$ with linearly independent rows; we call this matrix $A^0$. Indeed, denoting by $C^0$ the cone with $A^0$ as representation matrix, the matrix $A^0$ is invertible and its inverse gives the extreme rays of $C^0$, that is:

$$\mathsf{ExtremeRays}(C^0) = (A^0)^{-1}.$$

Therefore, the first DD-pair that the above incremental step should take as input is $(A^0, (A^0)^{-1})$.

The next key point towards a practically efficient DD method is to observe that most of the vectors $\mathbf{r'}_{(j,j')}$ in Lemma 5.1.17 are redundant. Indeed, Lemma 5.1.17 leads to a construction of a generating matrix of $C$ (in fact, this would be Algorithm 4 where Lines 13 and 16 are suppressed) producing a double exponential number of rays (w.r.t. the ambient dimension $n$) whereas Lemma 5.1.11 guarantees that the number of extreme rays of a polyhedral cone is singly exponential in its ambient dimension. To deal with this issue of redundancy, we need the notion of *adjacent* extreme rays.

**Definition 5.1.19. Adjacent extreme rays** Two distinct extreme rays $\mathbf{r}$ and $\mathbf{r}'$ of the polyhedral cone $C$ are called *adjacent* if they span a 2-dimensional face of $C$. [2]

The following lemma shows how we can test whether two extreme rays are adjacent or not. The proof can be found in [21].

**Lemma 5.1.18 (Adjacency test)** *Let $\mathbf{r}$ and $\mathbf{r}'$ be two distinct rays of $C$. Then, the following statements are equivalent:*

1. *$\mathbf{r}$ and $\mathbf{r}'$ are adjacent extreme rays,*
2. *$\mathbf{r}$ and $\mathbf{r}'$ are extreme rays and $\operatorname{rank}(A_{\zeta(\mathbf{r})\cap\zeta(\mathbf{r}')}) = n-2$,*
3. *if $\mathbf{r}''$ is a ray of $C$ with $\zeta(\mathbf{r}) \cap \zeta(\mathbf{r}') \subseteq \zeta(\mathbf{r}'')$, then $\mathbf{r}''$ is a positive multiple of either $\mathbf{r}$ or $\mathbf{r}'$.*

*It should be noted that the second statement is related to algebraic test for extreme rays while the third one is related to the combinatorial test.*

Based on Lemma 5.1.18, we have Algorithm 3 for testing whether two extreme rays are adjacent or not.

**Algorithm 3:** Adjacency Test algorithm.

```
Input  : $(A, \mathbf{r}, \mathbf{r}')$, where $A \in \mathbb{Q}^{m\times n}$ is the representation matrix of cone $C$, $\mathbf{r}$ and $\mathbf{r}'$ are two
         extreme rays of $C$.
Output: true if $\mathbf{r}$ and $\mathbf{r}'$ are adjacent, false otherwise
$\mathbf{s} := A\mathbf{r}$, $\mathbf{s}' := A\mathbf{r}'$;
let $\zeta(\mathbf{r})$ and $\zeta(\mathbf{r}')$ be set of indices of zeros in $\mathbf{s}$ and $\mathbf{s}'$ respectively;
$\zeta := \zeta(\mathbf{r}) \cap \zeta(\mathbf{r}')$;
if rank$(A_\zeta) = n-2$;
 then
    return true;
else
    return false;
```

The following lemma explains how to obtain $(A^i, R^i)$ from $(A^{i-1}, R^{i-1})$, where $A^{i-1}$ (resp. $A^i$) is the sub-matrix of $A$ consisting of its first $i-1$ (resp. $i$) rows. The double description method is a direct application of this lemma, see [21] for details.

**Lemma 5.1.19** *As above, let $(A^{i-1}, R^{i-1})$ be a DD-pair and denote by $J$ be the set of indices of the columns of $R^{i-1}$. Assume that $\operatorname{rank}(A^{i-1}) = n$ holds. Let $J' := J^- \cup J^0 \cup \operatorname{Adj}$, where $\operatorname{Adj}$ is the set of the pairs $(j, j') \in J^+ \times J^-$ such that $\mathbf{r}_j$, and $\mathbf{r}_{j'}$ are adjacent as extreme rays of $C^{i-1}$, the cone with $A^{i-1}$ as representing matrix. Let $R^i$ be the $(n \times |J'|)$-matrix consisting of*

- *the columns of $R^{i-1}$ with index in $J^- \cup J^0$, followed by*

[2]We do not use the minimal face, as it used in the main reference because it makes confusion.

- *the vectors $\mathbf{r'}_{(j,j')}$ for $(j, j') \in (J^+ \times J^-)$, where*

$$\mathbf{r'}_{(j,j')} = (A_i\mathbf{r}_j)\mathbf{r}_{j'} - (A_i\mathbf{r'}_j)\mathbf{r}_j,$$

*Then, the pair $(A^i, R^i)$ is a DD pair of $C^i$. Furthermore, if $R^{i-1}$ is a minimal generating matrix for the representation matrix $A^{i-1}$, then $R^i$ is also a minimal generating matrix for the representation matrix $A^i$.*

Using Lemmas 5.1.18 and 5.1.19, we can obtain Algorithm 4 for computing the extreme rays of a cone. Note that in this algorithm, $A^i$ shows the representation matrix in step $i$

**Algorithm 4:** DDmethod

```
Input  : a matrix $A \in \mathbb{Q}^{m\times n}$ defining the H-representation of a pointed cone $C$
Output: a matrix $R$ defining the V-representation of $C$
1  let $K$ be the set of indices of $A$'s independent rows;
2  $A^0 := A_K$;
3  $R^0 := (A^0)^{-1}$;
4  let $J$ be set of column indices of $R^0$;
5  while $K \neq \{1, \cdots, m\}$ do
6      select a $A$-row index $i \notin K$;
7      $J^+,\ J^0,\ J^- :=$ partition$(J, A_i)$;
8      add vectors with indices in $J^+$ and $J^0$ as columns to $R^i$;
9      for $p \in J^+$ do
10         for $n \in J^-$ do
11             if AdjacencyTest$(A^{i-1}, \mathbf{r}_p, \mathbf{r}_n)$ = true then
12                 $\mathbf{r}_{\text{new}} := (A_i\mathbf{r}_p)\mathbf{r}_n - (A_i\mathbf{r}_n)\mathbf{r}_p$;
13                 add $\mathbf{r}_{\text{new}}$ as columns to $R^i$;
14     let $J$ be set of indices in $R^i$;
15 return $R = R^m$
```

**Fourier-Motzkin elimination**

**Definition 5.1.20. Projection of a polyhedron** Let $A \in \mathbb{Q}^{m\times p}$ and $B \in \mathbb{Q}^{m\times q}$ be matrices. Let $\mathbf{c} \in \mathbb{Q}^m$ be a vector. Consider the polyhedron $P \subseteq \mathbb{Q}^{p+q}$ defined by $P = \{(\mathbf{u}, \mathbf{x}) \in \mathbb{Q}^{p+q} \mid A\mathbf{u} + B\mathbf{x} \leq \mathbf{c}\}$. We denote by $\mathsf{proj}(P; \mathbf{x})$ the *projection of $P$ on* $\mathbf{x}$, that is, the subset of $\mathbb{Q}^q$ defined by

$$\mathsf{proj}(P; \mathbf{x}) = \{\mathbf{x} \in \mathbb{Q}^q \mid \exists\, \mathbf{u} \in \mathbb{Q}^p,\ (\mathbf{u}, \mathbf{x}) \in P\}.$$

Fourier-Motzkin elimination (FME for short) is an algorithm computing the projection $\mathsf{proj}(P; \mathbf{x})$ of the polyhedron of $P$ by successively eliminating the $\mathbf{u}$-variables from the inequality system $A\mathbf{u} + B\mathbf{x} \leq \mathbf{c}$. This process shows that $\mathsf{proj}(P; \mathbf{x})$ is also a polyhedron.

**Definition 5.1.21. Inequality combination** Let $\ell_1, \ell_2$ be two inequalities: $a_1x_1 + \cdots + a_nx_n \leq$

$d_1$ and $b_1x_1+\cdots+b_nx_n \leq d_2$. Let $1 \leq i \leq n$ such that the coefficients $a_i$ and $b_i$ of $x_i$ in $\ell_1$ and $\ell_2$ are respectively positive and negative. The *combination* of $\ell_1$ and $\ell_2$ w.r.t. $x_i$, denoted by $\mathsf{Combine}(\ell_1, \ell_2, x_i)$, is:

$$-b_i(a_1x_1 + \cdots + a_nx_n) + a_i(b_1x_1 + \cdots + b_nx_n) \leq -b_id_1 + a_id_2.$$

Theorem 5.1.20 shows how to compute $\mathsf{proj}(P; \mathbf{x})$ when $\mathbf{u}$ consists of a single variable $x_i$. When $\mathbf{u}$ consists of several variables, FME obtains the projection $\mathsf{proj}(P; \mathbf{x})$ by repeated applications of Theorem 5.1.20.

**Theorem 5.1.20 (Fourier-Motzkin theorem [22])** *Let $A \in \mathbb{Q}^{m\times n}$ be a matrix and let $\mathbf{b} \in \mathbb{Q}^m$ be a vector. Consider the polyhedron $P = \{\mathbf{x} \in \mathbb{Q}^n \mid A\mathbf{x} \leq \mathbf{b}\}$. Let $S$ be the set of inequalities defined by $A\mathbf{x} \leq \mathbf{b}$. Also, let $1 \leq i \leq n$. We partition $S$ according to the sign of the coefficient of $x_i$: $S^+ = \{\ell \in S \mid \mathrm{coeff}(\ell, x_i) > 0\}$, $S^- = \{\ell \in S \mid \mathrm{coeff}(\ell, x_i) < 0\}$ and $S^0 = \{\ell \in S \mid \mathrm{coeff}(\ell, x_i) = 0\}$. We construct the following system of linear inequalities:*

$$S' = \{\mathsf{Combine}(s_p, s_n, x_i) \mid (s_p, s_n) \in S^+ \times S^-\} \cup S^0.$$

*Then, $S'$ is a representation of* $\mathsf{proj}(P; \mathbf{x} \setminus \{x_i\})$.

With the notations of Theorem 5.1.20, assume that each of $S^+$ and $S^-$ counts $\frac{m}{2}$ inequalities. Then, the set $S'$ counts $(\frac{m}{2})^2$ inequalities. After eliminating $p$ variables, the projection would be given by $O((\frac{m}{2})^{2^p})$ inequalities. Thus, FME is *double exponential* in $p$.

On the other hand, from [104] and [105], we know that the maximum number of facets of the projection on $\mathbb{Q}^{n-p}$ of a polyhedron in $\mathbb{Q}^n$ with $m$ facets is $O(m^{\lfloor n/2 \rfloor})$. Hence, it can be concluded that most of the generated inequalities by FME are *redundant*. Eliminating these redundancies is the main subject of the subsequent sections.

### 5.1.3 Cost model

We use the notion of *height* of an algebraic number as defined by Michel Waldschmidt in Chapter 3 of [106]. In particular, for any rational number $\frac{a}{b}$, thus with $b \neq 0$, we define the *height* of $\frac{a}{b}$, denoted as $\mathsf{height}(\frac{a}{b})$, as $\log \max(|a|, |b|)$.

For a given matrix $A \in \mathbb{Q}^{m\times n}$, let $\|A\|$ denote the infinity norm of $A$, that is, the maximum absolute value of a coefficient in $A$. We define the height of $A$, denoted by $\mathsf{height}(A) := \mathsf{height}(\|A\|)$, as the maximal height of a coefficient in $A$. For the rest of this section, our main reference is the PhD thesis of Arne Storjohann [107]. Let $k$ be a non-negative integer. We denote by $\mathcal{M}(k)$ an upper bound for the number of bit operations required for performing any of the basic operations (addition, multiplication, and division with remainder) on input $a, b \in \mathbb{Z}$ with $|a|, |b| < 2^k$. Using the multiplication algorithm of Arnold Schönhage and Volker Strassen [108] one can choose $\mathcal{M}(k) \in O(k \log k \log \log k)$.

We also need complexity estimates for some matrix operations. For positive integers $a, b, c$, let us denote by $\mathcal{MM}(a, b, c)$ an upper bound for the number of arithmetic operations (on the coefficients) required for multiplying an $(a \times b)$-matrix by an $(b \times c)$-matrix. In the case of square matrices of order $n$, we simply write $\mathcal{MM}(n)$ instead of $\mathcal{MM}(n, n, n)$. We denote by $\theta$ the exponent of linear algebra, that is, the smallest real positive number such that $\mathcal{MM}(n) \in O(n^{\theta})$.

In the following, we give complexity estimates in terms of $\mathcal{M}(k) \in O(k \log k \log\log k)$ and $\mathcal{B}(k) = \mathcal{M}(k) \log k \in O(k(\log k)^2 \log\log k)$. We replace every term of the form $(\log k)^p (\log\log k)^q (\log\log\log k)^r$, (where $p, q, r$ are positive real numbers) with $O(k^{\epsilon})$ where $\epsilon$ is a (positive) infinitesimal. Furthermore, in the complexity estimates of algorithms operating on matrices and vectors over $\mathbb{Z}$, we use a parameter $\beta$, which is a bound on the magnitude of the integers occurring during the algorithm. Our complexity estimates are measured in terms of *machine word operations*. Let $A \in \mathbb{Z}^{m\times n}$ and $B \in \mathbb{Z}^{n\times p}$. Then, the product of $A$ by $B$ can be computed within $O(\mathcal{MM}(m, n, p)(\log \beta) + (mn + np + mp)\mathcal{B}(\log \beta))$ word operations, where $\beta = n\, \|A\|\, \|B\|$ and $\|A\|$ (resp. $\|B\|$) denotes the maximum absolute value of a coefficient in $A$ (resp. $B$). Neglecting logarithmic factors, this estimate becomes $O(\max(m, n, p)^{\theta} \max(h_A, h_b))$ where $h_A = \mathsf{height}(A)$ and $h_B = \mathsf{height}(B)$. For a matrix $A \in \mathbb{Z}^{m\times n}$, a cost estimate of Gauss-Jordan transform is $O(nmr^{\theta-2}(\log \beta) + nm(\log r)\mathcal{B}(\log \beta))$ word operations, where $r$ is the rank of the input matrix $A$ and $\beta = (\sqrt{r}\|A\|)^r$. Let $h$ be the height of $A$, for a matrix $A \in \mathbb{Z}^{m\times n}$, with height $h$, the rank of $A$ is computed within $O(mn^{\theta+\epsilon}h^{1+\epsilon})$ word operations, and the inverse of $A$ (when this matrix is invertible over $\mathbb{Q}$ and $m = n$) is computed within $O(m^{\theta+1+\epsilon}h^{1+\epsilon})$ word operations. Let $A \in \mathbb{Z}^{n\times n}$ be an integer matrix, which is invertible over $\mathbb{Q}$. Then, the absolute value of any coefficient in $A^{-1}$ (inverse of $A$) can be bounded up to $(\sqrt{n-1}\|A\|^{(n-1)})$.

## 5.2 Revisiting Balas' method

As recalled, FME produces a representation of the projection of a polyhedron by eliminating one variable at a time. However, this procedure generates lots of redundant inequalities limiting its use in practice to polyhedral sets with a handful of variables only. In this section, we propose an efficient algorithm which generates the minimal representation of a full-dimensional pointed polyhedron, as well as its projections. Throughout this section, we use $Q$ to denote a full-dimensional pointed polyhedron in $\mathbb{Q}^n$, where

$$Q = \{(\mathbf{u}, \mathbf{x}) \in \mathbb{Q}^p \times \mathbb{Q}^q \mid A\mathbf{u} + B\mathbf{x} \le \mathbf{c}\}, \tag{5.4}$$

with $A \in \mathbb{Q}^{m\times p}$, $B \in \mathbb{Q}^{m\times q}$ and $\mathbf{c} \in \mathbb{Q}^m$. Thus, $Q$ has no implicit equations in its representation and the coefficient matrix $[A, B]$ has full column rank. Our goal in this section is to compute the minimal representation of the projection $\mathsf{proj}(Q; \mathbf{x})$ given by

$$\mathsf{proj}(Q; \mathbf{x}) := \{\mathbf{x} \mid \exists \mathbf{u}, s.t. (\mathbf{u}, \mathbf{x}) \in Q\}. \tag{5.5}$$

We call the cone

$$C := \{\mathbf{y} \in \mathbb{Q}^m \mid \mathbf{y}^t A = 0 \text{ and } \mathbf{y} \ge \mathbf{0}\} \tag{5.6}$$

the *projection cone* of $Q$ w.r.t.$\mathbf{u}$. When there is no ambiguity, we simply call $C$ the projection cone of $Q$. Using the following so-called *projection lemma*, we can compute a representation for the projection $\mathsf{proj}(Q; \mathbf{x})$:

**Lemma 5.2.1 ([24])** *The projection* $\mathsf{proj}(Q;\mathbf{x})$ *of the polyhedron* $Q$ *can be represented by*

$$S := \{\mathbf{y}^t B\mathbf{x} \leq \mathbf{y}^t \mathbf{c}, \forall \mathbf{y} \in \mathsf{ExtremeRays(C)}\},$$

*where* $C$ *is the projection cone of* $Q$ *defined by Equation (5.6).*

Lemma 5.2.1 provides the main idea of the block elimination method. However, the representation produced in this way may have redundant inequalities. The following example from [109] shows this point.

**Example** Let $P$ be the polyhedron represented by

$$P := \begin{cases} 12x_1 + x_2 - 3x_3 + x_4 \leq 1 \\ -36x_1 - 2x_2 + 18x_3 - 11x_4 \leq -2 \\ -18x_1 - x_2 + 9x_3 - 7x_4 \leq -1 \\ 45x_1 + 4x_2 - 18x_3 + 13x_4 \leq 4 \\ x_1 \geq 0 \\ x_2 \geq 0. \end{cases} \tag{5.7}$$

The projection cone of $P$ w.r.t. $\mathbf{u} := \{x_1, x_2\}$ is

$$C := \begin{cases} 12y_1 - 36y_2 - 18y_3 + 45y_4 = 0, \\ y_1 - 2y_2 - y_3 + 4y_4 = 0, \\ y_1 \geq 0, y_2 \geq 0, y_3 \geq 0, y_4 \geq 0. \end{cases} \tag{5.8}$$

The extreme rays of the cone $C$ are:

$(0,0,5,2,0,3), (3,0,2,0,0,1), (0,0,0,1,45,4), (1,0,0,0,12,1), (0,5,0,4,0,6), (3,1,0,0,0,1).$

These extreme rays generate a representation of $\mathsf{proj}(P;\{x_3,x_4\})$:

$$\begin{cases} 3x_3 - 3x_4 \leq 1, \quad 9x_3 - 11x_4 \leq 1, \quad 6x_3 - x_4 \leq 2, \\ -3x_3 + x_4 \leq 1, \; -18x_3 + 13x_4 \leq 4, \quad 9x_3 - 8x_4 \leq 1. \end{cases} \tag{5.9}$$

One can check that, in the above system of linear inequalities, the inequality $3x_3 - 3x_4 \leq 1$ is redundant.

In [25], Balas observed that if the matrix $B$ is invertible, then we can find a cone such that its extreme rays are in one-to-one correspondence with the facets of the projection of the polyhedron (the proof of this fact is similar to the proof of our Theorem 5.2.3). Using this fact, Balas developed an algorithm to find all redundant inequalities for all cases, including the cases where $B$ is singular.

In this section, we will explain Balas' algorithm [3] in detail. To achieve this, we lift the polyhedron $Q$ to a space in higher dimension by constructing the following objects.

[3] It should be noted that, although we are using his idea, we have found a flaw in Balas' paper. In fact, the last inequality in representation of $W^0$ is written as equality that paper.

**Construction of $B_0$.** Assume that the first $q$ rows of $B$, denoted as $B_1$, are independent. Denote the last $m-q$ rows of $B$ as $B_2$. Add $m-q$ columns, $\mathbf{e}_{q+1}, \ldots, \mathbf{e}_m$, to $B$, where $\mathbf{e}_i$ is the $i$-th vector in the canonical basis of $\mathbb{Q}^m$, thus with 1 in the $i$-th position and 0's anywhere else. The matrix $B_0$ has the following form:

$$B_0 = \begin{bmatrix} B_1 & \mathbf{0} \\ B_2 & I_{m-q} \end{bmatrix}.$$

To maintain consistency in the notation, let $A_0 = A$ and $\mathbf{c}_0 = \mathbf{c}$.

**Construction of $Q^0$.** We define:

$$Q^0 := \{(\mathbf{u}, \mathbf{x}') \in \mathbb{Q}^p \times \mathbb{Q}^m \mid A_0\mathbf{u} + B_0\mathbf{x}' \le \mathbf{c}_0\ ,\ x_{q+1} = \cdots = x_m = 0\}.$$

From now on, we use $\mathbf{x}'$ to represent the vector $\mathbf{x} \in \mathbb{Q}^q$, augmented with $m-q$ variables $(x_{q+1}, \ldots, x_m)$. Since the extra variables $(x_{q+1}, \ldots, x_m)$ are assigned to zero, we note that $\mathsf{proj}(Q; \mathbf{x})$ and $\mathsf{proj}(Q^0; \mathbf{x}')$ are "isomorphic" by means of the bijection $\Phi$:

$$\Phi : \begin{array}{c} \mathsf{proj}(Q; \mathbf{x}) \to \mathsf{proj}(Q^0; \mathbf{x}') \\ (x_1, \ldots, x_q) \mapsto (x_1, \ldots, x_q, 0, \ldots, 0) \end{array}$$

In the following, we will treat $\mathsf{proj}(Q; \mathbf{x})$ and $\mathsf{proj}(Q^0; \mathbf{x}')$ as the same polyhedron when there is no ambiguity.

**Construction of $W^0$.** Define $W^0$ to be the set of all $(\mathbf{v}, \mathbf{w}, v_0) \in \mathbb{Q}^q \times \mathbb{Q}^{m-q} \times \mathbb{Q}$ satisfying

$$\begin{aligned} W^0 = \{(\mathbf{v}, \mathbf{w}, v_0) \mid [\mathbf{v}^t, \mathbf{w}^t]B_0^{-1}A_0 = 0, [\mathbf{v}^t, \mathbf{w}^t]B_0^{-1} \ge 0, \\ -[\mathbf{v}^t, \mathbf{w}^t]B_0^{-1}\mathbf{c}_0 + v_0 \ge 0\}. \end{aligned} \tag{5.10}$$

Similar to the discussion in Balas' work, the extreme rays of the cone $\mathsf{proj}(W^0; \{\mathbf{v}, v_0\})$ are used to construct the minimal representation of the projection $\mathsf{proj}(Q; \mathbf{x})$. To prove this relation, we need a preliminary observation.

**Lemma 5.2.2** *The operations "computing the characteristic cone" and "computing projections" commute. To be precise, we have:*

$$\mathsf{CharCone}(\mathsf{proj}(Q; \mathbf{x})) = \mathsf{proj}(\mathsf{CharCone}(Q); \mathbf{x}).$$

**Proof** By the definition of the characteristic cone, we have $\mathsf{CharCone}(Q) = \{(\mathbf{u}, \mathbf{x}) \mid A\mathbf{u} + B\mathbf{x} \le \mathbf{0}\}$, whose representation has the same left-hand side as the one of $Q$. The lemma is valid if we can show that the representation of $\mathsf{proj}(\mathsf{CharCone}(Q); \mathbf{x})$ has the same left-hand side as $\mathsf{proj}(Q; \mathbf{x})$. This is obvious with the Fourier-Motzkin elimination procedure.

Theorem 5.2.3 shows that extreme rays of the cone $\overline{\mathsf{proj}(W^0; \{\mathbf{v}, v_0\})}$, which is defined as

$$\overline{\mathsf{proj}(W^0; \{\mathbf{v}, v_0\})} := \{(\mathbf{v}, -v_0) \mid (\mathbf{v}, v_0) \in \mathsf{proj}(W^0; \{\mathbf{v}, v_0\})\},$$

are in one-to-one correspondence with the facets of the homogenized cone of $\mathsf{proj}(Q; \mathbf{x})$. As a result its extreme rays can be used to find the minimal representation of $\mathsf{HomCone}(\mathsf{proj}(Q; \mathbf{x}))$.

**Theorem 5.2.3** *The polar cone of* $\mathsf{HomCone}(\mathsf{proj}(Q;\mathbf{x}))$ *is equal to* $\overline{\mathsf{proj}(W^0;\{\mathbf{v},v_0\})}$.

**Proof** By definition, the polar cone $(\mathsf{HomCone}(\mathsf{proj}(Q;\mathbf{x}))^*$ is equal to

$$\{(\mathbf{y},y_0)\mid[\mathbf{y}^t,y_0][\mathbf{x}^t,x_{\text{last}}]^t\le 0,\forall\,(\mathbf{x},x_{\text{last}})\in\mathsf{HomCone}(\mathsf{proj}(Q;\mathbf{x}))\}.$$

This claim follows immediately from: $(\mathsf{HomCone}(\mathsf{proj}(Q;\mathbf{x}))^*=\overline{\mathsf{proj}(W^0;\{\mathbf{v},v_0\})}$. We prove this latter equality in two steps.

($\supseteq$) For any $(\overline{\mathbf{v}},-\overline{v}_0)\in\overline{\mathsf{proj}(W^0;\{\mathbf{v},v_0\})}$, we need to show that $[\overline{\mathbf{v}}^t,-\overline{v}_0][\mathbf{x}^t,x_{\text{last}}]^t\le 0$ holds when $(\mathbf{x},x_{\text{last}})\in\mathsf{HomCone}(\mathsf{proj}(Q;\mathbf{x}))$. Remember that $Q$ is pointed. As a result, $\mathsf{HomCone}(\mathsf{proj}(Q;\mathbf{x}))$ is also pointed. Therefore, we only need to verify the desired property for the extreme rays of $\mathsf{HomCone}(\mathsf{proj}(Q;\mathbf{x}))$, which either have the form $(\mathbf{s},1)$ or $(\mathbf{s},0)$ (Theorem 5.1.16). Before continuing, we should notice that since $(\overline{\mathbf{v}},\overline{v}_0)\in\mathsf{proj}(W^0;\{\mathbf{v},v_0\})$, there exists $\overline{\mathbf{w}}$ such that $[\overline{\mathbf{v}}^t,\overline{\mathbf{w}}^t,\overline{v}_0]\in W^0$. Cases 1 and 2 below conclude that $(\overline{\mathbf{v}},-\overline{v}_0)\in\mathsf{HomCone}(\mathsf{proj}(Q;\mathbf{x}))^*$ holds.

Case 1: For the form $(\mathbf{s},1)$, we have $\mathbf{s}\in\mathsf{proj}(Q;\mathbf{x})$. Indeed, $\mathbf{s}$ is an extreme point of $\mathsf{proj}(Q;\mathbf{x})$. Hence, there exists $\overline{\mathbf{u}}\in\mathbb{Q}^p$, such that we have $A\overline{\mathbf{u}}+B\mathbf{s}\le\mathbf{c}$. By construction of $Q^0$, we have $A_0\overline{\mathbf{u}}+B_0\mathbf{s}'\le\mathbf{c}_0$, where $\mathbf{s}'=[\mathbf{s}^t,s_{q+1},\ldots,s_m]^t$ with $s_{q+1}=\cdots=s_m=0$. Therefore, we have: $[\overline{\mathbf{v}}^t,\overline{\mathbf{w}}^t]B_0^{-1}A_0\overline{\mathbf{u}}+[\overline{\mathbf{v}}^t,\overline{\mathbf{w}}^t]B_0^{-1}B_0\mathbf{s}'\le[\overline{\mathbf{v}}^t,\overline{\mathbf{w}}^t]B_0^{-1}\mathbf{c}_0$. This leads us to $\overline{\mathbf{v}}^t\mathbf{s}=[\overline{\mathbf{v}}^t,\overline{\mathbf{w}}^t]\mathbf{s}'\le[\overline{\mathbf{v}}^t,\overline{\mathbf{w}}^t]B_0^{-1}\mathbf{c}_0\le\overline{v}_0$. Therefore, we have $[\overline{\mathbf{v}}^t,-\overline{v}_0][\mathbf{s}^t,x_{\text{last}}]^t\le 0$, as desired.

Case 2: For the form $(\mathbf{s},0)$, we have $\mathbf{s}\in\mathsf{CharCone}(\mathsf{proj}(Q;\mathbf{x}))=\mathsf{proj}(\mathsf{CharCone}(Q);\mathbf{x})$. Thus, there exists $\overline{\mathbf{u}}\in\mathbb{Q}^p$ such that $A\overline{\mathbf{u}}+B\mathbf{s}\le\mathbf{0}$. Similarly to Case 1, we have $[\overline{\mathbf{v}}^t,\overline{\mathbf{w}}^t]B_0^{-1}A_0\overline{\mathbf{u}}+[\overline{\mathbf{v}}^t,\overline{\mathbf{w}}^t]B_0^{-1}B_0\mathbf{s}'\le[\overline{\mathbf{v}}^t,\overline{\mathbf{w}}^t]B_0^{-1}\mathbf{0}$. Therefore, we have $\overline{\mathbf{v}}^t\mathbf{s}=[\overline{\mathbf{v}}^t,\overline{\mathbf{w}}^t]\mathbf{s}'\le[\overline{\mathbf{v}}^t,\overline{\mathbf{w}}^t]B_0^{-1}\mathbf{0}=0$, and thus, we have $[\overline{\mathbf{v}}^t,-\overline{v}_0][\mathbf{s}^t,x_{\text{last}}]^t\le 0$, as desired.

($\subseteq$) For any $(\overline{\mathbf{y}},\overline{y}_0)\in\mathsf{HomCone}(\mathsf{proj}(Q;\mathbf{x}))^*$, we have $[\overline{\mathbf{y}}^t,\overline{y}_0][\mathbf{x}^t,x_{\text{last}}]^t\le 0$ for all $(\mathbf{x},x_{\text{last}})\in\mathsf{HomCone}(\mathsf{proj}(Q;\mathbf{x}))$. For any $\overline{\mathbf{x}}\in\mathsf{proj}(Q;\mathbf{x})$, we have $\overline{\mathbf{y}}^t\overline{\mathbf{x}}\le-\overline{y}_0$. The reason is that $(\overline{\mathbf{x}},1)\in\mathsf{HomCone}(\mathsf{proj}(Q;\mathbf{x}))$. Therefore, we have $\overline{\mathbf{y}}^t\mathbf{x}\le-\overline{y}_0$, for all $\mathbf{x}\in\mathsf{proj}(Q;\mathbf{x})$, which makes the inequality $\overline{\mathbf{y}}^t\mathbf{x}\le-\overline{y}_0$ redundant in the system $\{A\mathbf{u}+B\mathbf{x}\le\mathbf{c}\}$. By Farkas' Lemma (see Lemma 5.1.5), there exists $\mathbf{p}\ge\mathbf{0},\mathbf{p}\in\mathbb{Q}^m$ and $\lambda\ge 0$ such that $\mathbf{p}^tA=\mathbf{0}$, $\overline{\mathbf{y}}=\mathbf{p}^tB$, $\overline{y}_0=\mathbf{p}^t\mathbf{c}+\lambda$. Remember that $A_0=A$, $B_0=[B,B']$, $\mathbf{c}_0=\mathbf{c}$. Here $B'$ is the last $m-q$ columns of $B_0$ consisting of $\mathbf{e}_{q+1},\ldots,\mathbf{e}_m$. Let $\overline{\mathbf{w}}=\mathbf{p}^tB'$. We then have

$$\{\mathbf{p}^tA_0=\mathbf{0},\ [\overline{\mathbf{y}}^t,\overline{\mathbf{w}}^t]=\mathbf{p}^tB_0,-\overline{y}_0\ge\mathbf{p}^t\mathbf{c}_0,\mathbf{p}\ge\mathbf{0}\},$$

which is equivalent to

$$\begin{aligned}\{&\mathbf{p}^t=[\overline{\mathbf{y}}^t,\overline{\mathbf{w}}^t]B_0^{-1},[\overline{\mathbf{y}}^t,\overline{\mathbf{w}}^t]B_0^{-1}A_0=\mathbf{0},\\ &-\overline{y}_0\ge[\overline{\mathbf{y}}^t,\overline{\mathbf{w}}^t]B_0^{-1}\mathbf{c}_0,[\overline{\mathbf{y}}^t,\overline{\mathbf{w}}^t]B_0^{-1}\ge\mathbf{0}\}\end{aligned}$$

Therefore, $(\overline{\mathbf{y}},\overline{\mathbf{w}},-\overline{y}_0)\in W^0$, and $(\overline{\mathbf{y}},-\overline{y}_0)\in\mathsf{proj}(W^0;\{\mathbf{v},v_0\})$. From this, we deduce that $(\overline{\mathbf{y}},\overline{y}_0)\in\overline{\mathsf{proj}(W^0;\{\mathbf{v},v_0\})}$ holds.

**Theorem 5.2.4** *The minimal representation of* $\mathsf{proj}(Q; \mathbf{x})$ *is given exactly by*

$$\{\mathbf{v}^t\mathbf{x} \le v_0 \mid (\mathbf{v}, v_0) \in \mathsf{ExtremeRays}(\mathsf{proj}(W^0; (\mathbf{v}, v_0))) \setminus \{(\mathbf{0}, 1)\}\}.$$

**Proof** By Theorem 5.2.3, we know that the minimal representation of the homogenized cone $\mathsf{HomCone}(\mathsf{proj}(Q; \mathbf{x}))$ is given exactly by:

$$\{\mathbf{v}\mathbf{x} - v_0 x_{\text{last}} \le 0 \mid (\mathbf{v}, v_0) \in \mathsf{ExtremeRays}(\mathsf{proj}(W^0; (\mathbf{v}, v_0)))\}.$$

Using Lemma 5.1.15, any minimal representation of $\mathsf{HomCone}(\mathsf{proj}(Q; \mathbf{x}))$ has at most one more inequality than any minimal representation of $\mathsf{proj}(Q; \mathbf{x})$. This extra inequality is $x_{\text{last}} \ge 0$ and, in this case, $\mathsf{proj}(W^0; (\mathbf{v}, v_0))$ will have the extreme ray $(\mathbf{0}, 1)$, which can be detected easily. Therefore, the minimal representation of $\mathsf{proj}(Q; \mathbf{x})$ is given by $\{\mathbf{v}^t\mathbf{x} \le v_0 \mid (\mathbf{v}, v_0) \in \mathsf{ExtremeRays}(\mathsf{proj}(W^0; (\mathbf{v}, v_0))) \setminus \{(\mathbf{0}, 1)\}\}$.

For simplicity, we call the cone $\mathsf{proj}(W^0; \{\mathbf{v}, v_0\})$ the *redundancy test cone* of $Q$ w.r.t. $\mathbf{u}$ and denote it by $\mathcal{P}_{\mathbf{u}}(Q)$. When $\mathbf{u}$ is empty, we define $\mathcal{P}(Q) := \mathcal{P}_{\mathbf{u}}(Q)$ and we call it the *initial redundancy test cone*. If there is no ambiguity, we use only $\mathcal{P}_{\mathbf{u}}$ and $\mathcal{P}$ to denote the redundancy test cone and the initial redundancy test cone, respectively.

It should be noted that $\mathcal{P}(Q)$ can be used to detect redundant inequalities in the input system, as it is shown in Algorithm 7.

## 5.3 Minimal representation of the projected polyhedron

In this section, we present our algorithm for removing all the redundant inequalities generated during Fourier-Motzkin elimination. Our algorithm detects and eliminates redundant inequalities, right after their generation, using the redundancy test cone introduced in Section 5.2. Intuitively, we need to construct the cone $W^0$ and obtain a representation of the redundancy test cone, $\mathcal{P}_{\mathbf{u}}(Q)$, where $\mathbf{u}$ is the vector of eliminated variables, each time we eliminate a variable during FME. This method is time consuming because it requires to compute the projection of $W^0$ onto $\{\mathbf{v}, v_0\}$ space at each step. However, as we prove in 5.3.2, we only need to compute the initial redundancy test cone, using Algorithm 5, and the redundancy test cones, used in the subsequent variable eliminations, can be found incrementally without any extra cost. After generating the redundancy test cone, the algorithm uses Algorithm 6, and keeps the newly generated inequality only if it is an extreme ray of the redundancy test cone.

Note that a byproduct of this process is a *minimal projected representation* of the input system, according to the specified variable ordering. This representation is useful for finding solutions of linear inequality systems. The notion of projected representation was introduced in [110, 105] and will be reviewed later in this chapter.

For convenience, we rewrite the input polyhedron $Q$ defined in 5.4 as: $Q = \{\mathbf{y} \in \mathbb{Q}^n \mid \mathbf{A}\mathbf{y} \le \mathbf{c}\}$, where $\mathbf{A} = [A, B] \in \mathbb{Q}^{m \times n}$, $n = p + q$ and $\mathbf{y} = [\mathbf{u}^t, \mathbf{x}^t]^t \in \mathbb{Q}^n$. We assume the first $n$ rows of $\mathbf{A}$ are linearly independent.

**Algorithm 5:** Generate initial redundancy test cone

```
Input  : $S = \{\mathbf{A}\mathbf{y} \le \mathbf{c}\}$, a representation of the input polyhedron $Q$
Output: $\mathcal{P}$, a representation of the initial redundancy test cone
1 Construct $\mathbf{A}_0$ in the same way we constructed $B_0$, that is, $\mathbf{A}_0 := [\mathbf{A}, \mathbf{A}']$, where
    $\mathbf{A}' = [\mathbf{e}_{n+1}, \ldots, \mathbf{e}_m]$ with $\mathbf{e}_i$ being the $i$-th vector of the canonical basis of $\mathbb{Q}^m$;
2 Let $W := \{(\mathbf{v}, \mathbf{w}, v_0) \in \mathbb{Q}^n \times \mathbb{Q}^{m-n} \times \mathbb{Q} \mid -[\mathbf{v}^t, \mathbf{w}^t]\mathbf{A}_0^{-1}\mathbf{c} + v_0 \ge 0, [\mathbf{v}^t, \mathbf{w}^t]\mathbf{A}_0^{-1} \ge \mathbf{0}\}$;
3 $\mathcal{P} = \mathsf{projW}; \{\mathbf{v}, \mathsf{v}_0\}$;
4 return $\mathcal{P}$;
```

**Remark 5.3.1** *There are two important points about the Algorithm 5. First, we only need a representation of the initial redundancy test cone. This representation does not need to be minimal. Therefore, calling Algorithm 5 in Algorithm 7 (which computes a minimal projected representation of a polyhedron) does not lead to a recursive call to Algorithm 7. Second, to compute the projection* $\mathsf{proj}(W; \{\mathbf{v}, v_0\})$*, we need to eliminate $m-n$ variables from $m+1$ inequalities. The block elimination method is applied to achieve this. As it is shown in 5.2.1, the block elimination method requires to compute the extreme rays of the projection cone (denoted by $C$), which contains $m+1$ inequalities and $m+1$ variables. However, considering the structural properties of the coefficient matrix of the representation of $C$, we can see that computing the extreme rays of $C$ is equivalent to computing the extreme rays of another simpler cone, which still has $m+1$ inequalities but only $n+1$ variables.*

**Lemma 5.3.2** *A representation of the redundancy test cone $\mathcal{P}_{\mathbf{u}}(Q)$ can be obtained from $\mathcal{P}(Q)$ by setting coefficients of the corresponding $p$ eliminated variables to $0$ in the representation of $\mathcal{P}(Q)$.*

**Proof** To distinguish from the construction of $\mathcal{P}(Q)$, we rename the variables $\mathbf{v}, \mathbf{w}, v_0$ as $\mathbf{v}_{\mathbf{u}}, \mathbf{w}_{\mathbf{u}}, v_{\mathbf{u}}$, when constructing $W^0$ and computing the test cone $\mathcal{P}_{\mathbf{u}}(Q)$.

That is, we have $\mathcal{P}_{\mathbf{u}}(Q) = \mathsf{proj}(W^0; \{\mathbf{v}_{\mathbf{u}}, v_{\mathbf{u}}\})$, where $W^0$ is the set of all $(\mathbf{v}_{\mathbf{u}}, \mathbf{w}_{\mathbf{u}}, v_{\mathbf{u}}) \in \mathbb{Q}^q \times \mathbb{Q}^{m-q} \times \mathbb{Q}$ satisfying $\{(\mathbf{v}_{\mathbf{u}}, \mathbf{w}_{\mathbf{u}}, v_{\mathbf{u}}) \mid [\mathbf{v}_{\mathbf{u}}^t, \mathbf{w}_{\mathbf{u}}^t]B_0^{-1}A = \mathbf{0}, -[\mathbf{v}_{\mathbf{u}}^t, \mathbf{w}_{\mathbf{u}}^t]B_0^{-1}\mathbf{c} + v_{\mathbf{u}} \ge 0, [\mathbf{v}_{\mathbf{u}}^t, \mathbf{w}_{\mathbf{u}}^t]B_0^{-1} \ge \mathbf{0}\}$,

while we have $\mathcal{P}(Q) = \mathsf{proj}(W; \{\mathbf{v}, v_0\})$ where $W$ is the set of all $(\mathbf{v}, \mathbf{w}, v_0) \in \mathbb{Q}^n \times \mathbb{Q}^{m-n} \times \mathbb{Q}$ satisfying $\{(\mathbf{v}, \mathbf{w}, v_0) \mid -[\mathbf{v}^t, \mathbf{w}^t]\mathbf{A}_0^{-1}\mathbf{c} + v_0 \ge 0, [\mathbf{v}^t, \mathbf{w}^t]\mathbf{A}_0^{-1} \ge \mathbf{0}\}$.

By Step 1 of Algorithm 5, $[\mathbf{v}^t, \mathbf{w}^t]\mathbf{A}_0^{-1}\mathbf{A} = \mathbf{v}^t$ holds for all $(\mathbf{v}, \mathbf{w}, v_0) \in W$. We can rewrite $\mathbf{v}$ as $\mathbf{v}^t = [\mathbf{v}_1^t, \mathbf{v}_2^t]$, where $\mathbf{v}_1$ and $\mathbf{v}_2$ are the first $p$ and last $n-p$ variables of $\mathbf{v}$. Then, we have $[\mathbf{v}^t, \mathbf{w}^t]\mathbf{A}_0^{-1}A = \mathbf{v}_1^t$ and $[\mathbf{v}^t, \mathbf{w}^t]\mathbf{A}_0^{-1}B = \mathbf{v}_2^t$. Similarly, we have $[\mathbf{v}_{\mathbf{u}}^t, \mathbf{w}_{\mathbf{u}}^t]B_0^{-1}A = \mathbf{0}$ and $[\mathbf{v}_{\mathbf{u}}^t, \mathbf{w}_{\mathbf{u}}^t]B_0^{-1}B = \mathbf{v}_{\mathbf{u}}^t$ for all $(\mathbf{v}_{\mathbf{u}}, \mathbf{w}_{\mathbf{u}}, v_{\mathbf{u}}) \in W^0$. This lemma holds if we can show $\mathcal{P}_{\mathbf{u}} = \mathcal{P}|_{\mathbf{v}_1 = \mathbf{0}}$. We prove this in two steps:

($\subseteq$) For any $(\overline{\mathbf{v}}_{\mathbf{u}}, \overline{v}_{\mathbf{u}}) \in \mathcal{P}_{\mathbf{u}}(Q)$, there exists $\overline{\mathbf{w}}_{\mathbf{u}} \in \mathbb{Q}^{m-q}$, such that $(\overline{\mathbf{v}}_{\mathbf{u}}, \overline{\mathbf{w}}_{\mathbf{u}}, \overline{v}_{\mathbf{u}}) \in W^0$. Let $[\overline{\mathbf{v}}^t, \overline{\mathbf{w}}^t] := [\overline{\mathbf{v}}_{\mathbf{u}}^t, \overline{\mathbf{w}}_{\mathbf{u}}^t]B_0^{-1}\mathbf{A}_0$, where $\overline{\mathbf{v}}^t = [\overline{\mathbf{v}}_1^t, \overline{\mathbf{v}}_2^t]$ ($\overline{\mathbf{v}}_1 \in \mathbb{Q}^p, \overline{\mathbf{v}}_2 \in \mathbb{Q}^{n-p}$, and $\overline{\mathbf{w}} \in \mathbb{Q}^{m-n}$). Then, because $(\overline{\mathbf{v}}_{\mathbf{u}}, \overline{\mathbf{w}}_{\mathbf{u}}, \overline{v}_{\mathbf{u}}) \in W^0$, we have $\overline{\mathbf{v}}_1^t = [\overline{\mathbf{v}}_{\mathbf{u}}^t, \overline{\mathbf{w}}_{\mathbf{u}}^t]B_0^{-1}A = \mathbf{0}$ and $\overline{\mathbf{v}}_2^t = [\overline{\mathbf{v}}_{\mathbf{u}}^t, \overline{\mathbf{w}}_{\mathbf{u}}^t]B_0^{-1}B = \overline{\mathbf{v}}_{\mathbf{u}}$. Let $\overline{v}_0 = \overline{v}_{\mathbf{u}}$, it is easy to verify that $(\overline{\mathbf{v}}, \overline{\mathbf{w}}, \overline{v}_0) \in W$. Therefore, $(\mathbf{0}, \overline{\mathbf{v}}_{\mathbf{u}}, \overline{v}_{\mathbf{u}}) = (\overline{\mathbf{v}}, \overline{v}_0) \in \mathcal{P}(Q)$.

($\supseteq$) For any $(\mathbf{0}, \overline{\mathbf{v}}_2, \overline{v}_0) \in \mathcal{P}(Q)$, there exists $\overline{\mathbf{w}} \in \mathbb{Q}^{m-n}$, such that $(\mathbf{0}, \overline{\mathbf{v}}_2, \overline{\mathbf{w}}, \overline{v}_0) \in W$. Let

$[\overline{\mathbf{v}}_{\mathbf{u}}^t, \overline{\mathbf{w}}_{\mathbf{u}}^t] := [\mathbf{0}, \overline{\mathbf{v}}_2^t, \overline{\mathbf{w}}^t]\mathbf{A}_0^{-1}B_0$. We have $\overline{\mathbf{v}}_{\mathbf{u}} = [\mathbf{0}, \overline{\mathbf{v}}_2^t, \overline{\mathbf{w}}^t]\mathbf{A}_0^{-1}B = \overline{\mathbf{v}}_2$. Let $\overline{v}_{\mathbf{u}} = \overline{v}_0$, it is easy to verify that $(\overline{\mathbf{v}}_{\mathbf{u}}, \overline{\mathbf{w}}_{\mathbf{u}}, \overline{v}_{\mathbf{u}}) \in W^0$. Therefore, $(\overline{\mathbf{v}}_2, \overline{v}_0) = (\overline{\mathbf{v}}_{\mathbf{u}}, \overline{v}_{\mathbf{u}}) \in \mathcal{P}_{\mathbf{u}}(Q)$.

Consider again the polyhedron $Q = \{\mathbf{y} \in \mathbb{Q}^n \mid \mathbf{A}\mathbf{y} \le \mathbf{c}\}$, where $\mathbf{A} = [A, B] \in \mathbb{Q}^{m\times n}$, $n = p + q$ and $\mathbf{y} = [\mathbf{u}^t, \mathbf{x}^t]^t \in \mathbb{Q}^n$. Fix a variable ordering, say $y_1 > \cdots > y_n$, For $1 \le i \le n$, we denote by $\mathbf{A}^{(y_i)}$ the inequalities in the representation $\mathbf{A}\mathbf{y} \le \mathbf{c}$ of $Q$ whose largest variable is $y_i$. We denote by $\mathsf{ProjRep}(Q; y_1 > \cdots > y_n)$ the linear system $\mathbf{A}^{(y_1)}$ if $n = 1$ and the conjunction of $\mathbf{A}^{(y_1)}$ and $\mathsf{ProjRep}(\mathsf{proj}(Q; \mathbf{y}_2); y_2 > \cdots > y_n)$ otherwise, where $\mathbf{y}_2 = (y_2, \ldots, y_n)$. Of course, $\mathsf{ProjRep}(Q; y_1 > \cdots > y_n)$ depends on the representation which is used of $Q$.

**Definition 5.3.1. Projected representation** For the polyhedron $Q \subseteq \mathbb{Q}^n$, we call *projected representation* of $Q$ *w.r.t.* the variable order $y_1 > \cdots > y_n$ any linear system of the form $\mathsf{ProjRep}(Q; y_1 > \cdots > y_n)$. We say that such a linear system $P$ is a *minimal projected representation* of $Q$ if, for all $1 \le k \le n$, every inequality of $P$, with $y_k$ as largest variable, is not redundant among all the inequalities of $P$ with variables among $y_k, \ldots, y_n$.

Algorithm 7 generates a *minimal projected representation* of a polyhedron, w.r.t. an specific variable ordering.

**Algorithm 6:** Extreme ray test

```
Input : $(\mathcal{P}, \ell)$, where $\mathcal{P} := \{(\mathbf{v}, v_0) \in \mathbb{Q}^n \times \mathbb{Q} \mid M[\mathbf{v}^t, v_0]^t \le \mathbf{0}\}$ with $M \in \mathbb{Q}^{m\times(n+1)}$,
        $\ell : \mathbf{a}^t\mathbf{y} \le c$ with $\mathbf{a} \in \mathbb{Q}^n$ and $c \in \mathbb{Q}$
Output: true if $[\mathbf{a}^t, c]^t$ is an extreme ray of $\mathcal{P}$, false otherwise
1 Let $M$ be the coefficient matrix of $\mathcal{P}$;
2 Let $\mathbf{s} := M[\mathbf{a}^t, c]^t$;
3 Let $\zeta(\mathbf{s})$ be the index set of the zero coefficients of $\mathbf{s}$;
4 if rank$(M_{\zeta(\mathbf{s})}) = n$ then
5     return true;
6 else
7     return false;
```

## 5.4 Complexity estimates

In this section, we analyze the computational complexity of Algorithm 7, which computes a minimal projected representation of a given polyhedron. This computation is equivalent to eliminating all variables, one after another, in Fourier-Motzkin elimination. We prove that using our algorithm, finding a minimal projected representation of a polyhedron is singly exponential in the dimension $n$ of the ambient space. The most consuming procedure in Algorithm 7 is finding the initial redundancy test cone. This operation requires another polyhedron projection in higher dimension. As it is shown in Remark 5.3.1, we can use block elimination method to perform this task efficiently. This requires the computation of the extreme rays of the projection cone. The double description method is an efficient way

**Algorithm 7:** Minimal Projected Representation of $Q$

```
Input  : $S = \{\mathbf{Ay} \le \mathbf{c}\}$: a representation of the input polyhedron $Q$
Output: A minimal projected representation of $Q$
1  Generate the initial redundancy test cone $\mathcal{P}$ by Algorithm 5 $S_0 := \{\ \}$;
2  for $i$ from 1 to $m$ do
3      Let $f$ be the result of applying 6 with the inputs $\mathcal{P}$ and $\mathbf{A}_i\mathbf{y} \le \mathbf{c}_i$;
4      if $f = true$ then
5          $S_0 := S_0 \cup \{\mathbf{A}_i\mathbf{y} \le \mathbf{c}_i\}$;
6  $\mathcal{P} := \mathcal{P}|_{v_1=0}$;
7  for $i$ from 0 to $n-1$ do
8      $S_{i+1} := \{\ \}$;
9      for $\ell_{\text{pos}} \in S_i$ with positive coefficient of $y_{i+1}$ do
10         for $\ell_{\text{neg}} \in S_i$ with negative coefficient of $y_{i+1}$ do
11             $\ell_{\text{new}} := \mathsf{Combine}(\ell_{\text{pos}}, \ell_{\text{neg}}, y_{i+1})$;
12             Let $f$ be the result of applying 6 with the inputs $\mathcal{P}$ and $\ell_{\text{new}}$;
13             if $f = true$ then
14                 $S_{i+1} := S_{i+1} \cup \{\ell_{\text{new}}\}$;
15     for $\ell \in S_i$ with zero coefficient of $y_{i+1}$ do
16         Let $f$ be the result of applying 6 with the inputs $\mathcal{P}$ and $\ell$ ;
17         if $f = true$ then
18             $S_{i+1} := S_{i+1} \cup \{\ell\}$;
19     $\mathcal{P} := \mathcal{P}|_{v_{i+1}=0}$;
20 return $S_0 \cup S_1 \cup \cdots \cup S_n$;
```

to solve this problem. We begin this section by computing the bit complexity of the double description algorithm.

**Lemma 5.4.1 (Coefficient bound of extreme rays)** *Let $S = \{\mathbf{x} \in \mathbb{Q}^n \mid A\mathbf{x} \le \mathbf{0}\}$ be a minimal representation of a cone $C \subseteq \mathbb{Q}^n$, where $A \in \mathbb{Q}^{m\times n}$. Then, the absolute value of a coefficient in any extreme ray of C is bounded over by $(n-1)^n \|A\|^{2(n-1)}$.*

**Proof** From the properties of extreme rays, see Section 5.1.1, we know that when $\mathbf{r}$ is an extreme ray, there exists a sub-matrix $A' \in \mathbb{Q}^{(n-1)\times n}$ of $A$, such that $A'\mathbf{r} = 0$. This means that $\mathbf{r}$ is in the null-space of $A'$. Thus, the claim follows by proposition 6.6 of [107].

**Lemma 5.4.2** *Let $S = \{\mathbf{x} \in \mathbb{Q}^n \mid A\mathbf{x} \le \mathbf{0}\}$ be the minimal representation of a cone $C \subseteq \mathbb{Q}^n$, where $A \in \mathbb{Q}^{m\times n}$. The double description method, as specified in Algorithm 4, requires $O(m^{n+2} n^{\theta+\epsilon} h^{1+\epsilon})$ bit operations, where $h$ is the height of the matrix $A$.*

**Proof** The cost of Algorithm 4 during the processing of the first $n$ inequalities (Line 4) is negligible (in comparison to the subsequent computations) since it is equivalent to find the inverse of an $n \times n$ matrix. Therefore, to analyze the complexity of the DD method, we focus on the while-loop located at Line 5. After adding $t$ inequalities, with $n \le t \le m$, the first step is to partition the extreme rays at the $t-1$-iteration, with respect to the newly added inequality (Line 8). Note that we have at most $(t-1)^{\lfloor \frac{n}{2} \rfloor}$ extreme rays whose coefficients can be bounded over by $(n-1)^n \|A\|^{2(n-1)}$ (Lemma 5.4.1) at the $t-1$-iteration. Hence, this step needs at most $C_1 := (t-1)^{\lfloor \frac{n}{2} \rfloor} \times n \times \mathcal{M}(\log((n-1)^n \|A\|^{2(n-1)})) \le O(t^{\lfloor \frac{n}{2} \rfloor} n^{2+\epsilon} h^{1+\epsilon})$ bit operations. After partitioning the vectors, the next step is to check adjacency for each pair of vectors (Line 11). The cost of this step is equivalent to computing the rank of a sub-matrix $A' \in \mathbb{Q}^{(t-1)\times n}$ of $A$. This should be done for $\frac{t^n}{4}$ pairs of vectors. This step needs at most $C_2 := \frac{t^n}{4} \times O((t-1)n^{\theta+\epsilon}h^{1+\epsilon}) \le O(t^{n+1}n^{\theta+\epsilon}h^{1+\epsilon})$ bit operations. We know there are at most $t^{\lfloor \frac{n}{2} \rfloor}$ pairs of adjacent extreme rays. The next step is to combine every pair of adjacent vectors in order to obtain a new extreme ray (Line 12). This step consists of $n$ multiplications in $\mathbb{Q}$ of coefficients with absolute value bounded over by $(n-1)^n \|A\|^{2(n-1)}$ (Lemma 5.4.1) and this should be done for at most $t^{\lfloor \frac{n}{2} \rfloor}$ vectors. Therefore, the bit complexity of this step, is no more than $C_3 := t^{\lfloor \frac{n}{2} \rfloor} \times n \times \mathcal{M}(\log((n-1)^n \|A\|^{2(n-1)})) \le O(t^{\lfloor \frac{n}{2} \rfloor} n^{2+\epsilon} h^{1+\epsilon})$. Finally, the complexity of iteration $t$ of the while loop is $C := C_1 + C_2 + C_3$. The claim follows after simplifying $m \times C$.

**Lemma 5.4.3 (Complexity of constructing the initial redundancy test cone)** *Let $h$ be the maximum height of $A$ and $\mathbf{c}$ in the input system, then generating the initial redundancy test cone (Algorithm 5) requires at most $O(m^{n+3+\epsilon}(n+1)^{\theta+\epsilon}h^{1+\epsilon})$ bit operations. Moreover,* $\mathsf{proj}(W; \{\mathbf{v}, v_0\})$ *can be represented by $O(m^{\lfloor \frac{n+1}{2} \rfloor})$ inequalities, each with a height bound of $O(m^\epsilon n^{2+\epsilon} h)$.*

**Proof** We analyze Algorithm 5 step by step.

**Step 1: construction of $A_0$ from $A$.** The cost of this step can be neglected. However, it should be noticed that the matrix $A_0$ has a special structure. Without loss of generality, we can assume that the first $n$ rows of $A$ are linearly independent. The matrix $A_0$ has the following structure $A_0 = \begin{pmatrix} A_1 & \mathbf{0} \\ A_2 & I_{m-n} \end{pmatrix}$, where $A_1$ is a full rank matrix in $\mathbb{Q}^{n\times n}$ and $A_2 \in \mathbb{Q}^{(m-n)\times n}$.

**Step 2: construction of the cone $W$.** Using the structure of the matrix $A_0$, its inverse can be expressed as $A_0^{-1} = \begin{pmatrix} A_1^{-1} & \mathbf{0} \\ -A_2A_1^{-1} & I_{m-n} \end{pmatrix}$. Also, from 5.1.3 we have $\|A_1^{-1}\| \le (\sqrt{n-1}\|A_1\|)^{n-1}$. Therefore, $\|A_0^{-1}\| \le n^{\frac{n+1}{2}} \|A\|^n$, and $\|A_0^{-1}\mathbf{c}\| \le n^{\frac{n+3}{2}} \|A\|^n \|\mathbf{c}\| + (m-n)\|\mathbf{c}\|$. That is, $\mathsf{height}(A_0^{-1}) \in O(n^{1+\epsilon}h)$ and $\mathsf{height}(A_0^{-1}\mathbf{c}) \in O(m^\epsilon + n^{1+\epsilon}h)$. As a result, height of coefficients of $W$ can be bounded over by $O(m^\epsilon + n^{1+\epsilon}h)$.

To estimate the bit complexity, we need the following consecutive steps:

- Computing $A_0^{-1}$, which requires
$$\begin{aligned} & O(n^{\theta+1+\epsilon}h^{1+\epsilon}) + O((m-n)n^2\mathcal{M}(\max(\mathsf{height}(A_2), \mathsf{height}(A_1^{-1})))) \\ \le & O(mn^{\theta+1+\epsilon}h^{1+\epsilon}) \text{ bit operations;} \end{aligned}$$
- Constructing $W := \{(\mathbf{v}, \mathbf{w}, v_0) \mid -[\mathbf{v}^t, \mathbf{w}^t]\mathbf{A}_0^{-1}\mathbf{c} + v_0 \ge 0, [\mathbf{v}^t, \mathbf{w}^t]\mathbf{A}_0^{-1} \ge \mathbf{0}\}$ requires at most

$$\begin{aligned} C_1 :=& O(m^{1+\epsilon} n^{\theta+1+\epsilon} h^{1+\epsilon}) + O(mn\mathcal{M}(\mathsf{height}(A_0^{-1}, \mathbf{c}))) \\ &+O((m-n)h) \leq O(m^{1+\epsilon} n^{\theta+\epsilon+1} h^{1+\epsilon}) \text{ bit operations.} \end{aligned}$$

**Step 3: projecting $W$ and finding the initial redundancy test cone.** Following Lemma 5.2.1, we obtain a representation of $\mathsf{proj}(W; \{\mathbf{v}, v_0\})$ through finding extreme rays of the corresponding projection cone.

Let $E = (-A_2 A_1^{-1})^t \in \mathbb{Q}^{n\times(m-n)}$ and $\mathbf{g}^t$ be the last $m-n$ elements of $(A_0^{-1}\mathbf{c})^t$. Then, the projection cone can be represented by:

$$C = \{\mathbf{y} \in \mathbb{Q}^{m+1} \mid \mathbf{y}^t \begin{pmatrix} E \\ \mathbf{g}^t \\ I_{m-n} \end{pmatrix} = \mathbf{0}, \mathbf{y} \geq \mathbf{0}\}.$$

Note that $y_{n+2}, \ldots, y_{m+1}$ can be solved from the system of equations in the representation of $C$. We substitute them in the inequalities and obtain a representation of the cone $C'$, given by:

$$C' = \{\mathbf{y}' \in \mathbb{Q}^{n+1} \mid \mathbf{y}'^t \begin{pmatrix} E \\ \mathbf{g}^t \end{pmatrix} \leq \mathbf{0}, \mathbf{y}' \geq \mathbf{0}\}$$

In order to find the extreme rays of the cone $C$, we can find the extreme rays of the cone $C'$ and then back-substitute them into the equations to find the extreme rays of $C$. Applying the double description algorithm to $C'$, we can obtain all extreme rays of $C'$, and subsequently, the extreme rays of $C$. The cost estimate of this step is bounded over by the complexity of the double description algorithm with $C'$ as input. This operation requires at most $C_2 := O(m^{n+3}(n+1)^{\theta+\epsilon} \max(\mathsf{height}(E, \mathbf{g}^t))^{1+\epsilon}) \leq O(m^{n+3+\epsilon}(n+1)^{\theta+\epsilon} h^{1+\epsilon})$ bit operations. The overall complexity of the algorithm can be bounded over by: $C_1 + C_2 \leq O(m^{n+3+\epsilon}(n+1)^{\theta+\epsilon} h^{1+\epsilon})$. Also, by 5.4.1 and 5.4.2, we know that the cone $C$ has at most $O(m^{\lfloor \frac{n+1}{2} \rfloor})$ distinct extreme rays, each with height no more than $O(m^{\epsilon} n^{2+\epsilon} h)$. That is, $\mathsf{proj}(W^0; \{\mathbf{v}, v_0\})$ can be represented by at most $O(m^{\lfloor \frac{n+1}{2} \rfloor})$ inequalities, each with a height bound of $O(m^{\epsilon} n^{2+\epsilon} h)$.

**Lemma 5.4.4** *Algorithm 6 runs within $O(m^{\frac{n}{2}} n^{\theta+\epsilon} h^{1+\epsilon})$ bit operations.*

**Proof** The first step is to multiply the matrix $M$ and the vector $(\mathbf{t}, t_0)$. Let $d_M$ and $c_M$ be the number of rows and columns of $M$, respectively. Thus, $M \in \mathbb{Q}^{d_M \times c_M}$. We know that $M$ is the coefficient matrix of $\mathsf{proj}(W^0, \{\mathbf{v}, v_0\})$. Therefore, after eliminating $p$ variables $c_M = q+1$, where $q = n - p$ and $d_M \leq m^{\frac{n}{2}}$. Also, we have $\mathsf{height}(M) \in O(m^{\epsilon} n^{2+\epsilon} h)$. With these specifications, the multiplication step and the rank computation step need $O(m^{\frac{n}{2}} n^{2+\epsilon} h^{1+\epsilon})$ and $O(m^{\frac{n}{2}}(q+1)^{\theta+\epsilon} h^{1+\epsilon})$ bit operations, respectively. The claim follows after simplification.

Using Algorithms 5 and 6, we can find the minimal projected representation of a polyhedron in singly exponential time w.r.t. the number of variables $n$.

**Theorem 5.4.5** *Algorithm 7 is correct. Moreover, a minimal projected representation of $Q$ can be produced within $O(m^{\frac{5n}{2}} n^{\theta+1+\epsilon} h^{1+\epsilon})$ bit operations.*

**Proof** The correctness of the algorithm follows from Theorem 5.2.4 and Lemma 5.3.2.

By [23, 22], we know that after eliminating $p$ variables, the projection of the polyhedron has at most $m^{p+1}$ facets. For eliminating the next variable, there will be at most $(\frac{m^{p+1}}{2})^2$ pairs of inequalities to be considered and each of the pairs generate a new inequality which should be checked for redundancy. Therefore, the overall complexity of the algorithm is:

$$O(m^{n+3+\epsilon}(n+1)^{\theta+\epsilon}h^{1+\epsilon}) + \sum_{p=0}^{n} m^{2p+2} O(m^{\frac{n}{2}} n^{\theta+\epsilon} h^{1+\epsilon}) = O(m^{\frac{5n}{2}} n^{\theta+1+\epsilon} h^{1+\epsilon}).$$

## 5.5 Experimentation

In this section we report on our software implementation of the algorithms presented in the previous sections. Our implementation as well as our test cases are part of the `BPAS` library, available at `www.bpaslib.org/`. In our experiments, we report on serial and parallel implementation of the Minimal Projected Representation (MPR) algorithm, in terms of effectiveness for removing redundant inequalities and also in comparing with the `Project` command of the `PolyhedralSets` package of Maple 2017 and the famous `CDD` library (version 2018), we have been able to solve our test cases more efficiently. We believe that this is the result of using a more effective algorithm and an efficient implementation in `C`.

As test cases we use 16 consistent linear inequality systems. The first 9 test cases, (t1 to t9) are linear inequality systems that are randomly generated. The systems S24 and S35 are 24-simplex and 35-simplex polytopes. The systems C56 and C510 are cyclic polytopes in dimension five with six and ten vertices. The system C68 is a cyclic polytope in dimension six with eight vertices, C1011 is cyclic polytope in dimension ten with eleven vertices, and, Cro6 is the cross polytope in 6 dimension [111]. The `test` column of Table 5.1 shows these systems along with the number of variables and the number of inequalities for each of them.

We implemented the MPR algorithm with two different approaches: one iterative following closely Algorithm 7, and the other reorganization that algorithm by means of a divide and conquer scheme. In both implementations, we use a dense representation for the linear inequalities. In the first approach, we use *unrolled linked lists* to encode linear inequality systems. Indeed, using this data structure, we are able to store an array of inequalities in each node of a linked list and we can improve data locality. However, we use simple linked lists in the divide and conquer version to save time on dividing and joining lists. Although both these approaches have shown quite similar and promising results in terms of running time, we anticipate to get better results if we combine unrolled linked lists with the divide and conquer scheme while using a varying threshold for recursion as the algorithm goes on.

Columns `MPR-itr` and `MPR-rec` of Table 5.1 give the running time (in milliseconds) of these implementations on a configuration with Intel-i7-7700T (4 cores, 8 threads, clocking at 3.8 GHz). Also, columns `CDD`, `Maple`, and `Maple-MPR` are corresponding to running times of the Fourier algorithm in the `CDD` library, which uses LP for redundancy elimination, the

| Test (var,ineq) | MPR-itr | MPR-rec | CDD | Maple | Maple-MPR |
|---|---|---|---|---|---|
| S24 (24,25) | 46 | 41 | 411 | 6485 | 3040 |
| S35 (35,36) | 205 | 177 | 2169 | 57992 | 9840 |
| Cro6 (6,64) | 28 | 29 | 329 | 246750 | 8610 |
| C56 (5,6) | 1 | 1 | 13 | 825 | 140 |
| C68 (6,16) | 4 | 4 | 866 | 20154 | 650 |
| C1011 (10,11) | 95 | 92 | >1h | >1h | >1h |
| C510 (5,42) | 23 | 22 | 7674 | 6173 | 6070 |
| T1 (5,10) | 7 | 7 | 142 | 7974 | 1400 |
| T2 (10,12) | 109 | 112 | 122245 | 3321217 | 13330 |
| T3 (7,10) | 26 | 26 | 8207 | 117021 | 2900 |
| T4 (10,12) | 368 | 370 | 1177807 | >1h | 26650 |
| T5 (5,11) | 7 | 7 | 75 | 8229 | 1650 |
| T6 (10,20) | 26591 | 26156 | >1h | >1h | >1h |
| T7 (9,19) | 162628 | 158569 | >1h | >1h | >1h |
| T8 (8,19) | 21411 | 20915 | >1h | >1h | >1h |
| T9 (6,18) | 1281 | 1263 | 77372 | >1h | 267920 |

Table 5.1: Running time (in milliseconds) table for a set of examples, varying in the number of variables and inequalities, collected on a system with Intel-i7-7700T 4-core processor, clocking at 3.8 GHz.

function `PolyhedralSets:-Project` of Maple, and, an implementation of our algorithm in the Maple programming language, on the same system, respectively.

Using the divide and conquer scheme, we have been able to parallelize our program with Cilk [112]. We call this algorithm Parallel Minimal Projected Representation (PMPR).

Table 5.2 presents the running time (in milliseconds) and speedup of the multi-core version of the algorithm. The columns `PMPR-1`, `PMPR-4`, `PMPR-8`, and `PMPR-12` demonstrate the running time of the multi-core program on a system with Intel-Xeon-X5650 (12 cores, 24 threads, clocking at 2.6GHz), using 1, 4, 8, and 12 Cilk workers, respectively. The numbers in brackets show the speedup we gain using multi-threading.

## 5.6 Related works and concluding remarks

As we previously discussed, removing redundant inequalities during the execution of Fourier-Motzkin elimination is the central issue towards efficiency. Different algorithms have been developed to solve this problem. They also have been implemented in the various software libraries, including but not limited to: `CDD`[113], `VPL`[114], `PPL`[115], and `Polymake`[116] In this section, we briefly review *some* of these works.

In [24], Chernikov proposed a redundancy test with little added work, which greatly improves the practical efficiency of Fourier-Motzkin elimination. Kohler proposed a method in [22] which only uses matrix arithmetic operations to test the redundancy of inequalities.

| Test | PMPR-1 | PMPR-4 | | PMPR-8 | | PMPR-12 | |
|---|---|---|---|---|---|---|---|
| S24 | 67 | 71 | (0.9 **x**) | 73 | (0.9 **x**) | 83 | (0.8 **x**) |
| S35 | 291 | 308 | (0.9 **x**) | 310 | (0.9 **x**) | 375 | (0.7 **x**) |
| Cro6 | 54 | 45 | (1.2 **x**) | 36 | (1.5 **x**) | 34 | (1.5 **x**) |
| C56 | 2 | 3 | (0.6 **x**) | 3 | (0.6 **x**) | 12 | (0.1 **x**) |
| C68 | 8 | 7 | (1.1 **x**) | 7 | (1.1 **x**) | 19 | (0.4 **x**) |
| C1011 | 176 | 62 | (2.8 **x**) | 47 | (3.7 **x**) | 53 | (3.3 **x**) |
| C510 | 38 | 33 | (1.1 **x**) | 34 | (1.1 **x**) | 40 | (0.9 **x**) |
| T1 | 13 | 8 | (1.6 **x**) | 9 | (1.4 **x**) | 17 | (0.7 **x**) |
| T2 | 205 | 67 | (3.0 **x**) | 55 | (3.7 **x**) | 57 | (3.5 **x**) |
| T3 | 48 | 20 | (2.4 **x**) | 18 | (2.6 **x**) | 20 | (2.4 **x**) |
| T4 | 685 | 207 | (3.3 **x**) | 141 | (4.8 **x**) | 126 | (5.4 **x**) |
| T5 | 14 | 9 | (1.5 **x**) | 10 | (1.3 **x**) | 11 | (1.2 **x**) |
| T6 | 44262 | 12995 | (3.4 **x**) | 6785 | (6.5 **x**) | 5163 | (8.5 **x**) |
| T7 | 282721 | 78176 | (3.6 **x**) | 48048 | (5.8 **x**) | 35901 | (7.8 **x**) |
| T8 | 41067 | 10669 | (3.8 **x**) | 5689 | (7.2 **x**) | 4471 | (9.1 **x**) |
| T9 | 2407 | 742 | (3.2 **x**) | 491 | (4.8 **x**) | 448 | (5.3 **x**) |

Table 5.2: Running time (in milliseconds) table for our set of examples, with different number of Cilk workers, collected on a system Intel-Xeon-X5650 and 12 CPU cores, clocking at 2.6GHz.

As observed by Imbert in his work [23], the method he proposed in his paper as well as those of Chernikov and Kohler are essentially equivalent. Even though these works are effective in practice, none of them can remove all redundant inequalities generated by Fourier-Motzkin elimination.

Besides Fourier-Motzkin elimination, block elimination is another algorithmic tool to project polyhedra on a lower dimensional subspace. This method relies on the extreme rays of the so-called projection cone. Although there exist efficient methods to enumerate the extreme rays of this projection cone, like the *double description method* [21] (also known as Chernikova's algorithm [100, 101]), this method can not remove all the redundant inequalities.

In [25], Balas shows that if certain *inconvertibility conditions* are satisfied, then the extreme rays of the redundancy test cone exactly defines a minimal representation of the projection of a polyhedron. As Balas mentioned in his paper, this method can be extended to any polyhedron.

A drawback of Balas' work is the necessity of enumerating the extreme rays of the redundancy test cone (so as to produce a minimal representation of the projection $\mathsf{proj}(Q; \mathbf{x})$) which is time consuming. Our algorithm tests the redundancy of the inequality $\mathbf{ax} \le c$ by checking whether $(\mathbf{a}, c)$ is an extreme ray of the redundancy test cone or not.

Another related topic to our work is the concept of subsumption cone, as defined in [109]. Consider the polyhedron $Q$ given in Equation (5.4), define $T := \{(\lambda, \alpha, \beta) \mid \lambda^t \mathbf{A} = \alpha^t, \lambda^t \mathbf{c} \le$

$\beta, \lambda \geq \mathbf{0}\}$, where $\lambda$ and $\alpha$ are vectors of dimension $m$ and $n$ respectively, and $\beta$ is a variable. The *subsumption cone* of $Q$ is obtained by eliminating $\lambda$ in $T$, that is, $\mathsf{proj}(T; \{\alpha, \beta\})$. We proved that considering a full-dimensional, pointed polyhedron, where the first $n$ rows of the coefficient matrix are linearly independent, the initial redundancy test cone and the subsumption cone are equivalent.

Based on the improved version of Balas' methods, we obtain an algorithm to remove all the redundant inequalities produced by Fourier-Motzkin elimination. Even though this algorithm still has exponential complexity, which is expected, it is very effective in practice, as we have shown before.

The projection of polyhedra is a useful tool to solve problem instances in parametric linear programming, which plays an important role in the analysis, transformation and scheduling of for-loops of computer programs, see for instance [117, 118, 119].

Given a V-representation of a polyhedron $P$, one can obtain the V-representation of any projection of $P$[4]. The double description method turns the V-representation of the projection to its H-representation. Most existing software libraries dealing with polyhedral sets store a polyhedron with these two representations, like the *Parma Polyhedra Library (PPL)*. In this case, it is convenient to compute the projection using the block elimination method. When we are only given the H-representation, the first thing is to compute the V-representation, which is equivalent to the procedure of computing the initial test cone in our method. When we need to perform successive projections, it is well-known that Fourier-Motzkin elimination performs better than repeated applications of the double description method.

Recently, the verified polyhedron library (VPL) takes advantage of parametric linear programming to project a polyhedron. Like PPL, VPL may not beat Fourier-Motzkin elimination when we need to perform successive projections. In VPL, the authors rely on *raytracing* to remove redundant inequalities. This is an efficient way of removing redundancies, but this cannot remove them all, thus Linear Programming (LP) is still needed. As pointed out in [120], ray tracing is effective when there are not many redundancies; unfortunately, Fourier-Motzkin elimination typically generates lots of redundancies.

Another modern library dealing with polyhedral sets computation is the Normaliz library. In this library, Fourier-Motzkin Elimination is used for conversion between different descriptions of polyhedral sets. This is a different strategy than the one of our paper. As discussed in the introduction, we are motivated here by performing successive projections as required in the analysis, scheduling and transformation of for loop nests of computer programs.

As explained before, projecting a polyhedral set using the FME method is a well-studied subject, and the method proposed in this chapter is one of many methods for solving this problem. Specifically, many algorithms are suggested to make the Simplex-based methods more efficient. Although our experiments show that our method outperforms some other

[4]for example, $P$ is generated by $\{(1,2,3,4)^t, (2,3,4,5)^t, (2,3,7,9)^t\}$, the projection of $P$ onto the last two coordinates is generated by $\{(3,4)^t, (4,5)^t, (7,9)^t\}$

methods, we believe it may not be the case for all inequality systems. The method used for the projection should be chosen based on the properties of the input system (e.g., sparsity). For future works, we plan to experiment with a broader range of inequality systems and projection methods and find a heuristic to choose the best projection method based on the input inequality system. Also, we want to extend the proposed method to relax the mild assumptions that we make on the input system.

# Chapter 6

# Conclusion

In this thesis, we suggested techniques to improve the scope and applicability of compiler optimization methods.

We first went through our motivation and objectives in Chapter 1. Then we provided the necessary background information on the polyhedral theory, LLVM compiler infrastructure, some compiler optimization techniques, and program parallelization via OpenMP in Chapter 2.

In Chapter 3, we developed and implemented a method for detecting pipeline patterns between iteration blocks of different for-loop nests in a program. This method uses the polyhedral model techniques to block the iteration domains in a way that each block can be considered as an atomic task. It then finds the dependency relations between the tasks. For implementation, we extended and modified analysis, transformation, scheduling, and code generation passes in LLVM/Polly. After that, we rely on the `task` directive and the `depend` clause of OpenMP for exploiting the detecting parallelism. With this improvement, Polly can detect parallelism in programs that its conventional methods cannot. Also, the new methods provides a way to find the dataflow between different loop nests.

In Chapter 4, we developed a compiler technique to improve the performance of OpenMP programs with offloading regions by automatically prefetching data to the GPU's shared memory. In this project, we use scalar evolution to find the memory locations accessed in each team of threads, and we prefetch them into consecutive locations of the shared memory. In case there are multiple reads from the same locations, the method improves the program's performance by reducing the number of accesses to the high-latency global memory. We also extended our approach to handling bank conflict and extra share memory usage. It is implemented as part of the `OpenMPOpt` pass in LLVM. With this new pass, OpenMP programs with offloading regions can automatically take advantage of the GPU's memory hierarchy.

In Chapter 5, we developed and analyzed an algorithm to detect and remove redundant inequalities generated in the process of the Fourier-Motzkin elimination, a fundamental algorithm in the polyhedral theory. Our method is based on the research of Egon Balas.

It only uses matrix operations and can detect redundant inequalities right after their generation. We then analyzed the bit complexity of the method. This improved algorithm increases the performance of the Fourier-Motzkin elimination process, and as a result, it improves the performance of other algorithms that require Fourier-Motzkin elimination.

There are many ways to improve the applicability of the methods we developed for future works. Some ideas are listed below for each project:

- For the pipeline detection method explained in Chapter 3, we can improve the detection algorithm and code generation phases to optimize more general programs by relaxing some assumptions we made.
- For the memory prefetching method developed in Chapter 4, we can improve the inter-procedural optimization pass to be able to work correctly with different OpenMP pragmas.
- For the improved version of the Fourier-Motzkin elimination algorithm developed in Chapter 5, we can improve the applicability by extending the algorithm to relax our assumptions on the input system.

Moreover, we believe that the effectiveness of the methods we developed could be improved in many ways. Some ideas are listed below for each project:

- The method in Chapter 3 can be more effective if it considers other optimization methods. Also, it can be more effective if it can choose a good task granularity and reduce task generation overhead.
- The method in Chapter 4 can be more effective if it can find the most advantageous array to prefetch by considering the number of reads from the same location and the coalescence of the accesses.
- The method in Chapter 4 can be more effective if it can decide on the optimized algorithm for removing redundant inequalities.

# Bibliography

[1] Delaram Talaashrafi, Johannes Doerfert, and Marc Moreno Maza. A Pipeline Pattern Detection Technique in Polly. 2022.

[2] Delaram Talaashrafi, Marc Moreno Maza, and Johannes Doerfert. Towards Automatic OpenMP-Aware Utilization of Fast GPU Memory. In *International Workshop on OpenMP*, pages 67–80. Springer, 2022.

[3] Rui-Juan Jing, Marc Moreno-Maza, and Delaram Talaashrafi. Complexity estimates for Fourier-Motzkin elimination. In *International Workshop on Computer Algebra in Scientific Computing*, pages 282–306. Springer, 2020.

[4] Charles E Leiserson, Neil C Thompson, Joel S Emer, Bradley C Kuszmaul, Butler W Lampson, Daniel Sanchez, and Tao B Schardl. There's plenty of room at the Top: What will drive computer performance after Moore's law? *Science*, 368(6495):eaam9744, 2020.

[5] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[6] OpenMP application programming interface version 5.0. https://www.openmp.org/spec-html/5.0/openmp.html.

[7] David W Walker and Jack J Dongarra. MPI: a standard message passing interface. *Supercomputer*, 12:56–68, 1996.

[8] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA: Is CUDA the parallel programming model that application developers have been waiting for? *Queue*, 6(2):40–53, 2008.

[9] Chris Arthur Lattner. *LLVM: An infrastructure for multi-stage optimization*. PhD thesis, University of Illinois at Urbana-Champaign, 2002.

[10] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

[11] Tomofumi Yuki, Paul Feautrier, Sanjay Rajopadhye, and Vijay Saraswat. Array dataflow analysis for polyhedral x10 programs. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 23–34, 2013.

[12] SCEV Class Reference. https://llvm.org/doxygen/classllvm_1_1SCEV.html.

[13] Uday Bondhugula, A Hartono, J Ramanujam, and P Sadayappan. Pluto: A practical and fully automatic polyhedral program optimization system. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008)*. Citeseer, 2008.

[14] Aravind Acharya and Uday Bondhugula. Pluto+: Near-complete modeling of affine transformations for parallelism and locality. *ACM SIGPLAN Notices*, 50(8):54–64, 2015.

[15] Cédric Bastoul, Albert Cohen, Sylvain Girbal, Saurabh Sharma, and Olivier Temam. Putting polyhedral loop transformations to work. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 209–225. Springer, 2003.

[16] Harenome Razanajato, Cédric Bastoul, and Vincent Loechner. Pipelined Multithreading Generation in a Polyhedral Compiler. In *IMPACT 2020, in conjunction with HiPEAC 2020*, 2020.

[17] Kornilios Kourtis, Martino Dazzi, Nikolas Ioannou, Tobias Grosser, Abu Sebastian, and Evangelos Eleftheriou. Compiling Neural Networks for a Computational Memory Accelerator. 2020.

[18] 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)*, volume 2011, page 1, 2011.

[19] Joseph Huber, Melanie Cornelius, Giorgis Georgakoudis, Shilei Tian, Jose M Monsalve Diaz, Kuter Dinel, Barbara Chapman, and Johannes Doerfert. Efficient execution of OpenMP on GPUs. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 41–52. IEEE, 2022.

[20] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, New York, NY, USA, 2 edition, 2003.

[21] Komei Fukuda and Alain Prodon. Double description method revisited. In *Combinatorics and computer science*, pages 91–111. Springer, 1996.

[22] David A. Kohler. Projections of convex polyhedral sets. Technical report, California Univ. at Berkeley, Operations Research Center, 1967.

[23] Jean-Louis Imbert. Fourier's elimination: Which to choose? In *PPCP*, pages 117–129, 1993.

[24] Sergei Nikolaevich Chernikov. Contraction of systems of linear inequalities. *Doklady Akademii Nauk SSSR*, 131(3):518–521, 1960.

[25] Egon Balas. Projection with a minimal system of inequalities. *Computational Optimization and Applications*, 10(2):189–193, 1998.

[26] Paul Feautrier. Some efficient solutions to the affine scheduling problem. I. One-dimensional time. *International journal of parallel programming*, 21(5):313–347, 1992.

[27] Paul Feautrier. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *International journal of parallel programming*, 21(6):389–420, 1992.

[28] David Padua. *Encyclopedia of parallel computing*. Springer Science & Business Media, 2011.

[29] Cédric Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.*, pages 7–16. IEEE, 2004.

[30] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you think. In *International Conference on Compiler Construction*, pages 283–303. Springer, 2010.

[31] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, Jagannathan Ramanujam, Atanas Rountev, and Ponnuswamy Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *International Conference on Compiler Construction*, pages 132–146. Springer, 2008.

[32] Paul Feautrier. Automatic parallelization in the polytope model. *The Data Parallel Programming Model*, pages 79–103, 1996.

[33] Konrad Trifunovic, Dorit Nuzman, Albert Cohen, Ayal Zaks, and Ira Rosen. Polyhedral-model guided loop-nest auto-vectorization. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 327–337. IEEE, 2009.

[34] Tobias Grosser and Torsten Hoefler. Polly-ACC transparent compilation to heterogeneous hardware. In *Proceedings of the 2016 International Conference on Supercomputing*, pages 1–13, 2016.

[35] Aravind Sukumaran-Rajam and Philippe Clauss. The polyhedral model of nonlinear loops. *ACM Transactions on Architecture and Code Optimization (TACO)*, 12(4):1–27, 2015.

[36] Sven Verdoolaege. ISL: An integer set library for the polyhedral model. In *International Congress on Mathematical Software*, pages 299–302. Springer, 2010.

[37] Sven Verdoolaege, Serge Guelton, Tobias Grosser, and Albert Cohen. Schedule trees. In *Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques (IMPACT 2014), Vienna, Austria*, 2014. http://impact.gforge.inria.fr/impact2014/papers/impact2014-verdoolaege.pdf.

[38] Sven Verdoolaege. Integer set library: Manual. *Tech. Rep.*, 2016.

[39] Vincent Loechner. Polylib: A library for manipulating parameterized polyhedra, 1999.

[40] Paul Feautrier. Parametric integer programming. *RAIRO-Operations Research*, 22(3):243–268, 1988.

[41] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and David Wonnacott. The omega library interface guide, 1995.

[42] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[43] LLVM. `https://llvm.org/`.

[44] Clang Compiler. `https://clang.llvm.org/`.

[45] Chris Lattner. LLVM and Clang: Next generation compiler technology. In *The BSD conference*, volume 5, pages 1–20, 2008.

[46] Flang. `https://flang.llvm.org/`.

[47] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14. IEEE, 2021.

[48] Multi-Level Intermediate Representation. `https://mlir.llvm.org/`.

[49] LLVM-IR Language Reference. `https://llvm.org/docs/LangRef.html`.

[50] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[51] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques, & tools*. Pearson Education India, 2007.

[52] AAResults Concept Class Reference. `https://llvm.org/doxygen/classllvm_1_1AAResults_1_1Concept.html`.

[53] LoopInfo Class Reference. `https://llvm.org/doxygen/classllvm_1_1LoopInfo.html`.

[54] Robert A van Engelen. Efficient symbolic analysis for optimizing compilers. In *International Conference on Compiler Construction*, pages 118–132. Springer, 2001.

[55] Javed Absar. Scalar evolution demystified. In *European LLVM Developers Meeting*, 2018.

[56] Olaf Bachmann, Paul S Wang, and Eugene V Zima. Chains of recurrences—a method to expedite the evaluation of closed-form functions. In *Proceedings of the international symposium on Symbolic and algebraic computation*, pages 242–249, 1994.

[57] Eugene V Zima. Simplification and optimization transformations of chains of recurrences. In *Proceedings of the 1995 international symposium on Symbolic and algebraic computation*, pages 42–50, 1995.

[58] OpenMPIRBuilder Class Reference. `https://llvm.org/doxygen/classllvm_1_1OpenMPIRBuilder.html`.

[59] Johannes Doerfert and Hal Finkel. Compiler optimizations for parallel programs. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 112–119. Springer, 2018.

[60] Keith Cooper, Ken Kennedy, and Nathaniel McIntosh. Cross-loop reuse analysis and its application to cache optimizations. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 1–19. Springer, 1996.

[61] Michael I Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *ACM SIGPLAN Notices*, 41(11):151–162, 2006.

[62] Guy E Blelloch and Margaret Reid-Miller. Pipelining with futures. *Theory of Computing Systems*, 32(3):213–239, 1999.

[63] Antoniu Pop and Albert Cohen. A stream-computing extension to OpenMP. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, pages 5–14, 2011.

[64] I-Ting Angelina Lee, Charles E Leiserson, Tao B Schardl, Zhunping Zhang, and Jim Sukha. On-the-fly pipeline parallelism. *ACM Transactions on Parallel Computing (TOPC)*, 2(3):1–42, 2015.

[65] OpenMP Application Programming Interface Examples. `https://www.openmp.org/wp-content/uploads/openmp-examples-5.0.0.pdf`.

[66] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, et al. The tao of parallelism in algorithms. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 12–25, 2011.

[67] 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*, pages 318–328, 1988.

[68] Zia Ul Huda, Rohit Atre, Ali Jannesari, and Felix Wolf. Automatic parallel pattern detection in the algorithm structure design space. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 43–52. IEEE, 2016.

[69] Roshan Dathathri, Ravi Teja Mullapudi, and Uday Bondhugula. Compiling affine loop nests for a dynamic scheduling runtime on shared and distributed memory. *ACM Transactions on Parallel Computing (TOPC)*, 3(2):1–28, 2016.

[70] 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*, pages 57–72. Springer, 2015.

[71] Prasanth Chatarasi, Jun Shirako, and Vivek Sarkar. Polyhedral optimizations of explicitly parallel programs. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 213–226. IEEE, 2015.

[72] Antoniu Pop and Albert Cohen. OpenStream: Expressiveness and data-flow compilation of OpenMP streaming programs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):1–25, 2013.

[73] Feng Li, Antoniu Pop, and Albert Cohen. Automatic extraction of coarse-grained data-flow threads from imperative programs. *IEEE Micro*, 32(4):19–31, 2012.

[74] Manideepa Mukherjee, Alexander Fell, and Apala Guha. DFGenTool: A dataflow graph generation tool for coarse grain reconfigurable architectures. In *2017 30th International Conference on VLSI Design and 2017 16th International Conference on Embedded Systems (VLSID)*, pages 67–72. IEEE, 2017.

[75] Torbjörn Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 5.0.5 edition, 2012. http://gmplib.org/.

[76] Louis-Noël Pouchet et al. Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/pouchet/software/polybench*, 437:1–1, 2012.

[77] OpenMP application programming interface version 4.0. https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf.

[78] LLVM version 11. https://releases.llvm.org/download.html#11.0.0.

[79] ABA Bataev, Andrey Bokhanko, and James Cownie. Towards OpenMP support in LLVM. In *2013 European LLVM Conference*, 2013.

[80] Akihiro Hayashi, Jun Shirako, Etorre Tiotto, Robert Ho, and Vivek Sarkar. Performance evaluation of OpenMP's target construct on GPUs-exploring compiler optimisations. *International Journal of High Performance Computing and Networking*, 13(1):54–69, 2019.

[81] Shilei Tian, Jon Chesterfield, Johannes Doerfert, and Barbara Chapman. Experience Report: Writing A Portable GPU Runtime with OpenMP 5.1. In *International Workshop on OpenMP*, pages 159–169. Springer, 2021.

[82] Carlo Bertolli, Samuel F Antao, Gheorghe-Teodor Bercea, Arpith C Jacob, Alexandre E Eichenberger, Tong Chen, Zehra Sura, Hyojin Sung, Georgios Rokos, David Appelhans, et al. Integrating GPU support for OpenMP offloading directives into

Clang. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, pages 1–11, 2015.

[83] Samuel F Antao, Alexey Bataev, Arpith C Jacob, Gheorghe-Teodor Bercea, Alexandre E Eichenberger, Georgios Rokos, Matt Martineau, Tian Jin, Guray Ozen, Zehra Sura, et al. Offloading support for OpenMP in Clang and LLVM. In *2016 Third Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, pages 1–11. IEEE, 2016.

[84] CUDA programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide.

[85] Using Shared Memory in CUDA C/C++. https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/.

[86] Value Class Reference. https://llvm.org/doxygen/classllvm_1_1Value.html.

[87] SCEVAddRecExpr Class Reference. https://llvm.org/doxygen/classllvm_1_1SCEVAddRecExpr.html.

[88] SCEVExpander Class Reference. https://llvm.org/doxygen/classllvm_1_1SCEVExpander.html.

[89] John R. Tramm, Andrew R. Siegel, Tanzima Islam, and Martin Schulz. XSBench - The Development and Verification of A Performance Abstraction for Monte Carlo Reactor Analysis. *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)*, 2014.

[90] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20, 1991.

[91] Paul Feautrier. Automatic parallelization in the polytope model. In *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*, pages 79–103, London, UK, UK, 1996. Springer-Verlag. http://dl.acm.org/citation.cfm?id=647429.723579.

[92] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, and L. Pouchet. Polly - Polyhedral optimization in LLVM. In *First International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*, Chamonix, France, April 2011.

[93] M. Benabderrahmane, L. Pouchet, A. Cohen, and C. Bastoul. The polyhedral model is more widely applicable than you think. In *Proceedings of the 19th joint European conference on Theory and Practice of Software, international conference on Compiler Construction*, CC'10/ETAPS'10, pages 283–303, Berlin, Heidelberg, 2010. Springer-Verlag.

[94] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. *SIGPLAN Not.*, 43(6):101–113, June 2008.

[95] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compila-*

*tion Techniques*, PACT '04, pages 7–16, Washington, DC, USA, 2004. IEEE Computer Society.

[96] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor. Polyhedral parallel code generation for CUDA. *TACO*, 9(4):54, 2013.

[97] George B Dantzig. Fourier-Motzkin elimination and its dual. Technical report, STANFORD UNIV CA DEPT OF OPERATIONS RESEARCH, 1972.

[98] Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, Inc., New York, NY, USA, 1986.

[99] Leonid Khachiyan. Fourier-Motzkin Elimination Method. In Christodoulos A. Floudas and Panos M. Pardalos, editors, *Encyclopedia of Optimization, Second Edition*, pages 1074–1077. Springer, 2009.

[100] Natal'ja V. Chernikova. Algorithm for finding a general formula for the non-negative solutions of a system of linear inequalities. *Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki*, 5(2):334–337, 1965.

[101] Hervé Le Verge. *A note on Chernikova's algorithm*. PhD thesis, INRIA, 1992.

[102] Peter McMullen. The maximum numbers of faces of a convex polytope. *Mathematika*, 17(2):179–184, 1970.

[103] Marco Terzer. *Large scale methods to enumerate extreme rays and elementary modes*. PhD thesis, ETH Zurich, 2009.

[104] David Monniaux. Quantifier elimination by lazy model enumeration. In *International Conference on Computer Aided Verification*, pages 585–599. Springer, 2010.

[105] Rui-Juan Jing and Marc Moreno Maza. Computing the integer points of a polyhedron, II: complexity estimates. In *Computer Algebra in Scientific Computing - 19th International Workshop, CASC 2017, Beijing, China, September 18-22, 2017, Proceedings*, pages 242–256, 2017.

[106] Michel Waldschmidt. *Diophantine approximation on linear algebraic groups*. Springer Verlag, 2000.

[107] Arne Storjohann. *Algorithms for matrix canonical forms*. PhD thesis, Swiss Federal Institute of Technology Zurich, 2000.

[108] Arnold Schönhage and Volker Strassen. Schnelle multiplikation großer zahlen. *Computing*, 7(3-4):281–292, 1971.

[109] Tien Huynh, Catherine Lassez, and Jean-Louis Lassez. Practical issues on the projection of polyhedral sets. *Annals of mathematics and artificial intelligence*, 6(4):295–315, 1992.

[110] Rui-Juan Jing and Marc Moreno Maza. Computing the integer points of a polyhedron, I: algorithm. In *Computer Algebra in Scientific Computing - 19th International Workshop, CASC 2017, Beijing, China, September 18-22, 2017, Proceedings*, pages 225–241, 2017.

[111] Martin Henk, Jürgen Richter-Gebert, and Günter M Ziegler. 16 basic properties of convex polytopes. *Handbook of discrete and computational geometry*, pages 255–382, 2004.

[112] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, August 1995.

[113] Komei Fukuda. The CDD and CDDplus homepage. https://www.inf.ethz.ch/personal/fukudak/cdd_home/.

[114] Sylvain Boulmé, Alexandre Marechaly, David Monniaux, Michaël Périn, and Hang Yu. The verified polyhedron library: an overview. In *2018 20th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pages 9–17. IEEE, 2018.

[115] Roberto Bagnara, Patricia M Hill, and Enea Zaffanella. The parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1-2):3–21, 2008.

[116] Ewgenij Gawrilow and Michael Joswig. Polymake: a framework for analyzing convex polytopes. In *Polytopes—combinatorics and computation*, pages 43–73. Springer, 2000.

[117] Colin Jones. Polyhedral tools for control. Technical report, University of Cambridge, 2005.

[118] Francesco Borrelli, Alberto Bemporad, and Manfred Morari. Geometric algorithm for multiparametric linear programming. *Journal of optimization theory and applications*, 118(3):515–540, 2003.

[119] Colin N. Jones, Eric C. Kerrigan, and Jan M. Maciejowski. On polyhedral projection and parametric programming. *Journal of Optimization Theory and Applications*, 138(2):207–220, 2008.

[120] Alexandre Maréchal and Michaël Périn. Efficient elimination of redundancies in polyhedra by raytracing. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 367–385. Springer, 2017.

# Curriculum Vitae

| | |
|---|---|
| **Name:** | Delaram Talaashrafi |
| **Education:** | |
| 2019 - 2022 | Western University<br>London, Ontario, Canada<br>Ph.D. in Computer Science |
| 2017 - 2018 | Western University<br>London, Ontario, Canada<br>M.Sc. in Computer Science |
| 2012 - 2017 | Isfahan University of Technology<br>B.Sc. in Computer Engineering |