DISSERTATION

DETECTION OF LINEAR ALGEBRA OPERATIONS IN POLYHEDRAL PROGRAMS

Submitted by

Guillaume IOOSS

Department of Computer Science, Colorado State University

Laboratoire de l'Informatique du Parallélisme, École Normale Supérieure de Lyon

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Fall 2016

Doctoral Committee:

Advisors: Sanjay Rajopadhye, Christophe Alias, Alain Darte

Philippe Clauss

Sriram Sankaranarayanan

Stephan Thomassé

Jennifer Mueller

Hamid Chitsaz

ABSTRACT

DETECTION OF LINEAR ALGEBRA OPERATIONS
IN POLYHEDRAL PROGRAMS

Writing a code which uses an architecture at its full capability has become an increasingly difficult problem over the last years. For some key operations, a dedicated accelerator or a finely tuned implementation exists and delivers the best performance. Thus, when compiling a code, identifying these operations and issuing calls to their high-performance implementation is attractive. In this dissertation, we focus on the problem of detection of these operations. We propose a framework which detects linear algebra subcomputations within a polyhedral program. The main idea of this framework is to partition the computation in order to isolate different subcomputations in a regular manner, then we consider each portion of the computation and try to recognize it as a combination of linear algebra operations.

We perform the partitioning of the computation by using a program transformation called *monoparametric tiling*. This transformation partitions the computation into blocks, whose shape is some homothetic scaling of a fixed-size partitioning. We show that the tiled program remains polyhedral while allowing a limited amount of parametrization: a single size parameter. This is an improvement compared to the previous work on tiling, that forced us to choose between these two properties.

Then, in order to recognize computations, we introduce a template recognition algorithm. This template recognition algorithm is built on a state-of-the-art program equivalence algorithm. We also propose several extensions in order to manage some semantic properties.

Finally, we combine these two previous contributions into a framework which detects linear algebra subcomputations. A part of this framework is a library of template, based on the BLAS specification. We demonstrate our framework on several applications.

RÉSUMÉ

RECONNAISSANCE D'OPÉRATIONS D'ALGÈBRE LINÉAIRE
DANS UN PROGRAMME POLYÉDRIQUE

Durant ces dernières années, Il est de plus en plus compliqué d'écrire du code qui utilise une architecture au mieux de ses capacités. Certaines opérations clefs ont soit un accélérateur dédié, ou admettent une implémentation finement optimisée qui délivre les meilleures performances. Ainsi, il est intéressant d'identifier ces opérations pendant la compilation d'un programme, et de faire appel à une implémentation optimisée.

Nous nous intéressons dans cette thèse au problème de détection de ces opérations. Nous proposons un procédé qui détecte des sous-calculs correspondant à des opérations d'algèbre linéaire à l'intérieur de programmes polyédriques. L'idée principale de ce procédé est de découper le programme en sous-calculs isolés, et essayer de reconnaître chaque sous-calculs comme une combinaison d'opérateurs d'algèbre linéaire.

Le découpage du calcul est effectué en utilisant une transformation de programme appelée tuilage monoparamétrique. Cette transformation partitionne le calcul en tuiles dont la forme est un agrandissement paramétrique d'une tuile de taille constante. Nous montrons que le programme tuilé reste polyédrique tout en permettant une paramétrisation limitée des tailles de tuile. Les travaux précédents sur le tuilage nous forçaient à choisir l'une de ces deux propriétés.

Ensuite, afin d'identifier les opérateurs, nous introduisons un algorithme de reconnaissance de template, qui est une extension d'un algorithme d'équivalence de programme. Nous proposons plusieurs extensions afin de tenir compte des propriétés sémantiques communément rencontrées en algèbre linéaire.

Enfin, nous combinons les deux contributions précédentes en un procédé qui détecte les sous-calculs correspondant à des opérateurs d'algèbre linéaire. Une de ses composantes est une

librairie de template, inspirée de la spécification BLAS. Nous démontrons l'efficacité de notre procédé sur plusieurs applications.

ACKNOWLEDGEMENTS

First of all, I would like to thanks the members of my committee, Philippe Clauss, Sriram Sankaranarayanan, Hamid Chitsaz, Stéphan Thomassé and Jennifer Mueller, for accepting to review my work and providing helpful feedback during the proposal defense exam and the writing of the dissertation. I was particularly impressed by the depth of comprehension their questions shown, even on topics outside of their usual domain. New ideas and trails to investigate were almost literally bouncing from everywhere during the defense. On a side note, I will also try to avoid any more typos on page 12 in the future.

I would also like to thank my two PhD advisors, Sanjay Rajopadhye and Christophe Alias. Both of them complement each other nicely both scientifically and their work methodologies. Also, managing a PhD, in cotutelle, with long distances involved is very technical to manage, for example when trying to explain a complicated notion by videoconference when only a notepad and a camera are available, or when trying to appease the requirements of both administrations involved. I firmly believe that I could have not done this work without the involvement of both of them, and that they both did a wonderful job.

My PhD years were split between two locations: ENS Lyon in France and the Computer Science department in the Colorado State University. I would like to thanks the members of the team Compsys in Lyon: Alexandre, fellow coworker who is always eager to discuss and try out new crazy ideas, Laure, Paul, Alain and Fabrice. I would also like to thanks the members of the team Mélange in Fort Collins: Tomofumi, Yun, Waruna, Nirmal, Revathy, Swetha, Daniel, Louis, Yohann, and many others for all the time spent discussing ideas and sharing problems over our numerous meetings and coffee breaks.

A huge thanks to the members of both administrations for their infinite patience. It was sometimes complicated to complete some administrative procedures from across the ocean, or when some documents were due on one side before being available, because of deadlines on the

other side. Évelyne, our team administrative assistant at Lyon, and the third cycle office at the ENS are living proof that the administration can be nice, understanding and competent. I would like to thanks Daniel Hirschkoff and David Coeurjolly for letting me supervise their lab session of their class, during my years in Lyon, and provide me some valuable teaching experience.

I would like to thanks all the people who supported me personally and morally, Agathe, Damien, Margeaux, Arthur, Jonathan, Elie, Alice, Etienne, Robin, Thomas, Nimé, Apeiron, Solenn, Elvire, Imryss, Elro, Camille, Rev, Xavier, maki, Etienne, Sto, Jonas, Ophélia, Mikael, Mickael, Laetitia, Benjamin, Alexandre, Alexandre, Benoit, Elodie, and many many others. A large thanks to the people who though about the concept of the foyer, where you could get some fresh air and cold fruit juice during the summer, whereas our office was facing south without air conditioning.

I would like to thanks my family for always being supportive, wherever I was. I understand that it is hard for a non-scientist to have an idea of what I am doing, but I still hope to find a way to explain what I am doing in more details than just a vague "mathematics applied on computers to make them work better".

Finally, on the Colorado side, this work was partially supported by NSF grants CCF-0917319 and CNS-1240991, AFOSR grant FA9550-13-1-0064, and DoE grant DE-SC0014495.

TABLE OF CONTENTS

Abstract ii

Résumé iii

Acknowledgements v

1 Introduction 1

1.1 Architecture evolution and high-performance libraries 1

1.2 Using high-performance libraries automatically 3

1.3 Contributions 4

1.4 Outline of the dissertation 5

2 Background 8

2.1 The polyhedral model 8

2.2 Program representation 11

2.3 Program transformation 19

2.4 Program equivalence and template recognition 22

3 Monoparametric Partitioning 29

3.1 Hyperrectangular Monoparametric partitioning 31

3.1.1 Monoparametric partitioning of polyhedra 31

3.1.2 Monoparametric partitioning of affine functions 41

3.2 Hyperrectangular monoparametric partitioning program transformation 48

3.2.1 Monoparametric partitioning program transformation 48

3.2.2 Derivation of the partitioning 50

3.2.3 Experimental validation . . . 56
3.3 General monoparametric partitioning . . . 61
3.3.1 General monoparametric partitioning of polyhedra . . . 62
3.3.2 General monoparametric partitioning of affine functions . . . 65
3.3.3 General monoparametric partitioning program transformation . . . 69
3.4 Discussion . . . 71

4 From Partitioning to Tiling **76**
4.1 Hierarchical programs . . . 77
4.2 Monoparametric tiling without reduction . . . 80
4.2.1 Example - Smith Waterman . . . 81
4.2.2 Preprocessing - Preparing for the outlining . . . 85
4.2.3 Tile group . . . 89
4.2.4 Monoparametric Tiling with outlining without reduction . . . 92
4.3 Monoparametric tiling with reduction . . . 97
4.3.1 Monoparametric partitioning with reductions . . . 98
4.3.2 Tile groups and reduction . . . 101
4.3.3 Monoparametric Tiling with reductions . . . 108
4.4 Experimental Validation . . . 113

5 Template Recognition **119**
5.1 Barthou's equivalence algorithm . . . 120
5.2 Adapting the equivalence algorithm into a template algorithm . . . 120
5.3 Examples . . . 125
5.4 Managing semantic properties . . . 136
5.5 Experimental validation . . . 141
5.6 Discussion . . . 142

6 Recognizing subcomputations **147**
6.1 Template library . . . 147
6.2 Linear algebra operation recognition framework . . . 154

6.3 Applications . . . 159
6.3.1 Dense Linear algebra applications . . . 160
6.3.2 Applications outside of dense linear algebra . . . 165
6.4 Discussion . . . 171

7 Related Work **173**
7.1 Tiling transformation and code generation . . . 173
7.2 Program equivalence and template recognition . . . 176
7.2.1 Program equivalence . . . 176
7.2.2 Template recognition . . . 178
7.3 Dense linear algebra algorithm derivation . . . 180

8 Conclusion **183**
8.1 Conclusion . . . 183
8.2 Future directions . . . 184
8.2.1 Monoparametric tiling transformation . . . 185
8.2.2 Template recognition algorithm . . . 186
8.2.3 Template recognition framework . . . 188

A Résumé du travail de thèse **189**

Chapter 1

Introduction

Writing a code which uses an architecture at its full capability has become an increasingly difficult problem over the last years. For some key operations, a dedicated accelerator or a finely tuned implementation exists and delivers the best performance. Thus, when compiling a program, identifying these operations and issuing calls to their high-performance implementation is attractive. In this dissertation, we focus on the problem of detecting these operations. We propose a framework which recognizes dense linear algebra operations as the subcomputations of a program.

1.1 Architecture evolution and high-performance libraries

Moore's law [55, 72] predicted that the number of transistors on chip has been doubling every 18 months. At the same time, because of Dennard scaling [20], the dynamic power consumed by a Central Processing Unit had remained constant, thereby directly translating the density increase into a performance gain. Thus, the processing power of a chip was doubling every year and a half, without having to change the architecture. However, about ten years ago, Dennard scaling ended, because the leakage power became a significant portion of the consumed power and could no longer be ignored. Thus, power has become a critical issue in the design of an

architecture and manufacturers reacted by increasing the complexity of their circuits, such as introducing multicore architectures.

Architectures have become more and more hierarchical, especially their memories. For example, the number of levels of cache have increased in CPU, due to the memory wall: having small private memory which can prefetch data, thus can be accessed quickly and efficiently in term of energy, and can communicate with higher level of memories which are shared, is attractive. Also, because of the size of the main memory, several layers of memories, with increasing capacity is interesting. Another example of hierarchy is the number of logical level present when implementing on a GPU (grid, thread block, warp and thread).

Architectures have also become more and more heterogeneous. Accelerators tend to migrate on chip, such as the Floating Point Unit in the past, because of the number of transistors available on a CPU increases. We can probably expect the same to happen for Graphics Process Unit (which is a Single Instruction Multiple Data (SIMD) architecture and can manage efficiently coalesced memory accesses and vectorized code), and for other accelerators. Because we will end up in the near future with more transistors that can be powered at once (because of thermal issues) [21, 74], we will probably have some parts of a chip which implements specific operations and can be powered-on when needed [17, 51, 59].

Therefore, architectures have become and will become much more complex, making their exploitation at their full capabilities extremely challenging. For several core computations, their most efficient implementations are hand-written and can be found in high-performance libraries (such as BLAS [46] or LAPACK [6] for the dense linear algebra domain). These implementations were carefully tuned, either by hand, or through dedicated generator, such that a functional equivalent code generated through a general-purpose compiler does not reach the same level of performance [88].

Hence, we have a set of highly efficient operations which are hard-coded inside a dedicated accelerator or admit an highly efficient implementation, and whose performance is not reachable by a generated code. Now, let us see if we can improve automatically the performance of a given code by using these implementation.

1.2 Using high-performance libraries automatically

In many applications, we can find portions of their code which correspond to an operation from a high-performance library. In this case, this portion of code can be substituted by a call to the highly-tuned implementation, instead of relying on a compiler. Several experiments [3, 52] show that this substitution is beneficial for the performance of a code. However, such opportunity might be missed, either because the operation was not identified (for example, because it was not exposed in the computation), or the existence of a corresponding tuned implementation was unknown.

The next step would be to make the compiler generate the library call automatically, but we first need to detect the occurrences of such operation in a program. We focus on this problem in this dissertation, in the context of polyhedral program, and for dense linear algebra operations. We emphasis the fact that even if the program is not a linear algebra computation, it can still contain several linear algebra subcomputations.

To the best of our knowledge, this problem was only partially solved. For example, Menon and Pingali [52] focus on detecting instances of matrix multiplication and matrix vector multiplication in a Matlab code. Alias [3] can detect a larger class of operations, but these operations are forced to have at most one occurrence of an input in its computation (which preclude computations such as TRSM [46], i.e., $C = L^{-1}.B$ where L is a lower triangular matrix). We overcome these limitations in our work.

In order to solve this problem, we have to face several underlying challenges. First, if we want to replace parts of a computation by a function call, we should avoid overlaps between recognized subcomputations. The alternative implies introducing some extra work. Then, because we detect linear algebra operations, we have to manage the common semantic properties found in linear algebra. In particular, many linear algebra operations involve a summation over a parametric number of terms (e.g., $C_{i,j} = \sum_k A_{i,k} \times B_{k,j}$ for matrix multiplication). Hence the associativity and commutativity property of the sum operation has to be considered during the recognition process. Finally, if the number of linear algebra operations we aim to recognize

is important, we have to be careful about the scalability of our recognition process. This dissertation addresses all of these issues.

1.3 Contributions

Our strategy toward the recognition of dense linear algebra computation is the following. In order to avoid overlappings between the recognized subcomputation, we partition the computation into blocks beforehand, and we consider each block independently. We focus on linear algebra operations which manipulate matrices and vectors, thus the data used by such operations should be contained in rectangle regions of data (corresponding to its input matrices or vectors). Thereby, if we partition the data of a program into rectangular blocks and isolate the computation according to which block of data is used, we should be able to recognize some of them as a composition of different linear algebra operations. Our main contributions are the following.

Monoparametric tiling transformation In order to split the computation according to the block of data touched, we use a tiling transformation. Usually, this tiling transformation is a fixed-size tiling (the size of a tile is a constant, but cannot be changed after compilation), or a parametric tiling (the size of a tile is a program parameter, but the transformed program is not polyhedral, which means that we cannot apply any polyhedral analysis after this transformation). We introduce a novel kind of tiling called *monoparametric tiling*, in which the tile sizes are multiples of a single parameter. We prove that the monoparametric tiling transformation is polyhedral, while allowing limited parametrization. This is an improvement compared to the previous works on tiling, which forced us to choose between these two properties. We also present how to obtain a structured program through this transformation, in which we have a finite non-parametric number of subprograms, and each subprogram correspond to the computation of a tile.

Template recognition algorithm We extend a program equivalence algorithm introduced by Barthou [8] into a template recognition algorithm. This means that instead of comparing two programs in order to prove that the computations they perform are identical, we compare a program with a template (i.e., a pattern of computation) in order to prove that the program fits the pattern. We also show how to manage some semantic properties commonly found in linear algebra computations.

Framework to recognize linear algebra subcomputations We use the two previous contributions to build a framework which detects linear algebra subcomputations. More precisely, we first apply the monoparametric tiling transformation to obtain a list of subprograms. Then, we consider each subprogram independently and try to recognize it as a combination of linear algebra *template* (i.e., program pattern). These template come from a template library, inspired by the BLAS specification.

1.4 Outline of the dissertation

This dissertation describes the different elements needed by our strategy, in order to recognize dense linear algebra subcomputations. The details of our core contributions are found from Chapter 3 to Chapter 6, and can be divided into two parts. The first part consists of Chapter 3 and 4, describes the monoparametric tiling transformation, and should be read in that order. The second part consists of Chapter 5 and 6, describes the template recognition algorithm and its application to find linear algebra subcomputation, and can be read independently. These two parts are fairly independent, thus can be read independently.

Chapter 2 This chapter describes the preliminary notion we will need in the rest of the document. We start by introducing the polyhedral model, then we describe the program representation and some program transformations which will be used in the rest of this document. Then, we summarize a state-of-the-art program equivalence algorithm, which will be adapted in Chapter 5.

The next two chapters describe how we divide the computation according to the data touched. This is done through a new transformation called monoparametric tiling. This transformation is introduced in two parts: the first part of this transformation (called *monoparametric partitioning* and covered in Chapter 3) is just a reindexing transformation, which replaces the original indices of a program into those used for tiling. The second part of this transformation (covered in Chapter 4) distributes the computation into different subprograms.

Chapter 3 This chapter focuses on the first part of the monoparametric tiling transformation. The first half of this chapter discusses about the case where the tile shape is a multidimensional rectangle (i.e., hyperrectangular). We first show how to transform polyhedra and affine functions, before showing how to transform a full polyhedral program. In particular, we present an algorithm which derives the missing tile shapes (called *ratio* for the hyperrectangular case) while still obtaining a polyhedral program. Then, we generalize this work to any polyhedral tile shape.

Chapter 4 This chapter presents the monoparametric tiling transformation, which is built on top of the monoparametric partitioning transformation. The monoparametric tiling split the computation of a program into a finite number of separated subprograms (called *subsystems*) which communicate through a *main system*. The main intuition is that each subsystem might correspond to a combination of linear algebra operations. We describe this transformation when the original program does not contain any reduction, then we consider the general case.

The next two chapters focus on the problem of template recognition, i.e., recognizing specific pattern of computation in a program.

Chapter 5 This chapter describes our template recognition algorithm, as an extension of the program equivalence algorithm described in Chapter 2. We adapt it in order to manage semantic properties commonly found in linear algebra.

Chapter 6 This chapter presents our framework to detect linear algebra subcomputations. We first describe the library of templates, based on BLAS [46], and the various optimizations performed to this library. Then, we combine the previously introduced pieces into a single framework: we consider each subsystem generated by the monoparametric tiling transformation independently, and apply recursively our template recognition algorithm, using our template library. We evaluate our framework (in term of compile time and efficiency) on linear algebra and bioinformatic applications.

Chapters 7 and 8 This dissertation ends with a review of the related work about how the tiling transformation is managed in a compiler, program equivalence, template recognition and the existing frameworks linked to dense linear algebra derivation, before concluding our work.

Chapter 2

Background

2.1 The polyhedral model

Program analysis is the automatic study of programs in order to extract properties about its behavior. Such properties might be used in various ways, such as gaining a better understanding of the program (by checking its correctness, or its robustness, or some other safety properties). They can also be used to modify a program, for example to improve its performance, to reduce the resources spent during the execution or to adapt its computation to another model of execution.

Two kinds of program analysis exist: *static* and *dynamic*. Static analysis study the program during the compilation phase, thus before its execution. At this point, the execution trace of a program (i.e., the list of states a program goes through during its execution) cannot be determined precisely, because it might depend on the input provided to the program right before executing it. However, because we are at compile time, the benefit of an analysis can counterbalance its possibly large amount of time taken to perform such analysis.

In the case of dynamic program analysis, the program is analyzed during its execution. The analysis has an immediate access to the trace of execution, which provides them with more information than at static time and allows them to react to certain events. However, because

the program is running at the same time, such analysis is limited in term of resources (typically the additional time and memory taken), which limits its complexity and might prevent certain aggressive modification of the program (such that changing non-locally the order of execution of the instructions of a program). In the rest of this document, we will consider static analysis.

Another issue is that some problems of obtaining certain properties (such as deciding the termination of a program, branch prediction or checking the equivalence between two of them) are unfortunately undecidable. Thus, we have a choice between the precision of the analysis (relying on approximation instead of exact informations) and the expressiveness of the class of program considered. The former choice is made in polyhedral compilation, for which the class of program is restricted to *affine* computation.

Affine computation An *affine expression* of a set of indices $i_1, i_2, \ldots$ is a expression of the form $(a_1.i_1 + a_2.i_2 + \cdots + b)$, where the a_i and b are scalar. An *affine computation* is a sequence of operations which can be described by a combination of:

- Loop nests (`for (i) ...`) whose boundaries are affine expressions of the surrounding loop indices
- Assignment statements (`S: A[`$u(\vec{i})$`] = f(`B_1`[`$v_1(\vec{i})$`], ..., `B_k`[`$v_k(\vec{i})$`])`), whose array accesses functions ($u(\vec{i})$, $v_1(\vec{i}), \ldots v_k(\vec{i})$) are affine expressions of the surrounding loop indices.
- Sequence of statements (`S1; S2`)
- Branching conditions on the indices (`if (`$u(\vec{i} \leq v(\vec{i})$`) then S1; else S2;`), whose condition is an affine constraints on the surrounding loop indices

In addition to the surrounding loop indices, the affine expressions can also use *program parameters*, i.e. symbolic variables which are constant during the program execution, and whose value is passed by the user. Typically, a program parameter can be the size of an array.

For example, the following program corresponds to an affine computation (which is a matrix multiplication between two square matrices), where N is a parameter:

```
for (int i=0; i<N; i++)
    for (int j=0; j<N; j++)
        for (int k=0; k<N; k++)
S(i,j,k):    C[i,j] = C[i,j] + A[i,k] * B[k,j];
```

Indirect array accesses (such as `A[B[i]]`) or non-affine conditions (such as `for (int i=0; i<N; i++) { for (int j=0; j<sqrt(i); j++) S; }`), are not allowed inside affine computations.

Polyhedral model In order to represent such computation, we rely on a mathematical model called the *polyhedral model*. Two mathematical objects are used to represent aspects of such a computation: *polyhedra* (a set of integer points satisfying affine constraints) and *affine functions*. As an example of utilization of such objects, we use a polyhedron (called the *iteration domain*) to represent the set of instances of a statement inside a loop nest (a point of this set corresponds to one execution of the loop body). An affine function (called *dependence function*) can be used to represent the producer-consumer relationship between two statement instances.

This model allows us to summarize precisely the trace of execution of a program, whose size is parametric and generally huge, through a finite non-parametric number of mathematical objects. It allows many analyses (called *polyhedral analysis*) to derive many useful informations about the program (such as loops which can be parallelized, or a better order of execution of the statements of a loop).

Polyhedra and affine function have several stability properties, which ensures that we keep having union of polyhedra and affine functions while we manipulate them. Union of polyhedra are stable by intersection, union, difference and preimage by an affine function. The image of an union of polyhedra is still an union of polyhedra if the affine function is *unimodular* (i.e., its determinant is 1 or -1). In general, taking the image of a polyhedron (such as $\{i|0 \leq i < N\}$) by a non-unimodular affine function (such as $(i \mapsto 2i)$) might not give an union of polyhedra ($2\mathbb{Z} \cap \{i|0 \leq i < 2N\}$, because of the holes introduced). Other mathematical objects (such as Presburger sets) can be used to represent such results.

For example, if we consider the matrix multiplication program presented above, the iteration domain of this loop is the polyhedron $\{i, j, k \mid 0 \leq i < N \ \wedge \ 0 \leq j < N \ \wedge \ 0 \leq k < N\}$. We can represent the dependences between the different instances of `S` by the following affine function: $(i, j, k \mapsto i, j, k-1)$ (which means that the statement instance `S(i,j,k)` depends on the statement instance `S(i,j,k-1)`). Notice that this dependence exists only of the instances of $S(i, j, k)$ for which $k > 0$.

Matricial representation Later in the document (cf Chapter 3), we will use the matricial representation of affine functions and polyhedra. Mathematically, an affine function can be represented by a matrix A and a vector $\vec{c}$: $f : (\vec{i} \mapsto A\vec{i} + \vec{c})$. If the program has parameters, we differentiate them from the indices, and use an additional matrix: $f : (\vec{i} \mapsto A\vec{i} + B\vec{p} + \vec{c})$.

Likewise, a polyhedron can be represented by two matrices (Q,R) and a vector ($\vec{q}$): $\mathcal{P} = \{\vec{i} \mid Q\vec{i} + R\vec{p} + \vec{q} \geq \vec{0}\}$. This representation is enough to express equalities and strict inequalities: equalities $Q_k\vec{i} + R_k\vec{p} + q_k = 0$ can be represented as the conjunction of the two inequalities $Q_k\vec{i} + R_k\vec{p} + q_k \geq 0$ and $Q_k\vec{i} + R_k\vec{p} + q_k \leq 0$, and strict inequalities $Q_k\vec{i} + R_k\vec{p} + q_k > 0$ can be replaced by $Q_k\vec{i} + R_k\vec{p} + (q_k - 1) \geq 0$

For example, given a triangle $\mathcal{T} = \{i, j \mid i \geq 0 \ \wedge \ j \geq 0 \ \wedge \ N - i - j \geq 0\}$ where N is a parameter, its matricial representation is:

$$\mathcal{T} = \left\{ i, j \mid \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -1 & -1 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} .(N) + \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \right\}$$

2.2 Program representation

Several ways of representing an affine computation have been introduced in the literature. In this subsection, we will present two of them (*Affine Control Loop* and *System of Affine Recurrence Equations*). Then, we will present another program representation as a middle ground, which

will be used in the rest of this document. Finally, we will enrich our program representation by introducing *reductions*.

Affine Control Loop One of the most commonly used polyhedral program representation is called *Affine Control Loop* (ACL). An informal definition was introduced in the previous section, when introducing the notion of affine computation. A formal definition is the following:

Definition 2.1. An ACL is a program $P(\vec{p}, ())$ of the form:

$$\begin{aligned} P(\vec{p}, \vec{i}) = \quad & \texttt{for (k = } lb(\vec{p}, \vec{i})\texttt{; k < } ub(\vec{p}, \vec{i})\texttt{; k++)}\{P(\vec{p}, (\vec{i}, k))\texttt{;}\} \\ || \quad & P1(\vec{p}, \vec{i})\texttt{;}\ P2(\vec{p}, \vec{i}) \\ || \quad & \texttt{if (}Cond(\vec{p}, \vec{i})\texttt{) then } P1(\vec{p}, \vec{i})\texttt{; else } P2(\vec{p}, \vec{i})\texttt{;} \\ || \quad & A[u(\vec{p}, \vec{i})] := f(\dots, B_k[v_k(\vec{p}, \vec{i})], \dots) \end{aligned}$$

where:

- $\vec{p}$ are the program parameters
- $lb(\vec{p}, \vec{i})$ and $ub(\vec{p}, \vec{i})$ are affine expressions of the program parameter and the surrounding loop indices
- $Cond(\vec{p}, \vec{i})$ is an affine constraint on the program parameter and the surrounding loop indices
- $u(\vec{p}, \vec{i})$ and the $v_k(\vec{p}, \vec{i})$ are affine expressions
- f is an arbitrary function
- A and the B_k are arrays

An example of ACL has been given in the previous subsection, corresponding to a matrix multiplication `C = A * B`.

We notice that, in addition to describing a computation, a ACL provides an order of execution of the statements (called *schedule*, and given by the `for` loops) and a memory allocation (in

this example, all the elements computed by `S(i,j,k)` are stored in the same location `C[i,j]`, independent of k and erasing the previous value when computed). The original schedule and memory allocation of an ACL might interfere with and complicate some polyhedral analysis. For example, when we list the set of dependences of a program, we need to deal with *dataflow dependences* (the result of a statement is used by another statement), *output dependences* (the result of a statement is stored at the same location of the result of another statement) and *anti-dependences* (the result of a statement must be used before it is overwritten). The last two types of dependences are related to the memory allocation and the schedule which are provided, and are not inherent to the computation itself.

System of Affine Recurrence Equations Another commonly used polyhedral program representation is called *System of Affine Recurrence Equation* (SARE) [37, 63, 64]. The idea is to represent the computation itself by a list of affine equations, without any information about the schedule or the memory allocation. Therefore, only the dataflow dependences remain. The formal definition is the following:

Definition 2.2. A SARE is a list of equations of the form

$$\texttt{Var}[\vec{i}] \texttt{ = } \begin{cases} \dots \\ \vec{i} \in \mathcal{D}_k : \texttt{Expr}_k \\ \dots \end{cases}$$

where the $\mathcal{D}_k$ are disjoint, and where:

- `Var` is a *variable*, is defined over a polyhedral domain $\mathcal{D}$ and is either an input, an output or a local variable
- `Expr` is an *expression*, and can be either:
  - A variable `Var[f(`$\vec{i}$`)]` where `f` is an affine function
  - A constant `Const`,
  - An affine function of the indices `f(`$\vec{i}$`)`

– An operation `Op(Expr`$_1$`, ..., Expr`$_k$`)` of arity k (i.e., the operation has k arguments)

Moreover, we assume that *Expr* depends strictly on all of its arguments (i.e., the value of each of the argument impacts the value of *Expr*).

For example, the SARE corresponding to a matrix multiplication computation is the following:

$$\begin{aligned} \texttt{C[i,j]} &= \texttt{Temp[i,j,N-1];} \\ \texttt{Temp[i,j,k]} &= \begin{cases} \texttt{Temp[i,j,k-1] + A[i,k] * B[k,j];} & \texttt{if k>0} \\ \texttt{A[i,0] * B[0,j];} & \texttt{if k=0} \end{cases} \end{aligned}$$

where `C` is an output variable defined over $\{i, j \mid 0 \leq i, j < N\}$, `Temp` is a local variable defined over $\{i, j, k \mid 0 \leq i, j, k < N\}$ and `A` and `B` are input variables defined over $\{i, j \mid 0 \leq i, j < N\}$.

The *Alpha* language [27] is an extension of this representation which allows more kinds of expression on the right-hand side of an equation.

Compared to an ACL, the SARE program representation does not have implicit schedule or memory allocation. An ACL can be transformed into a SARE, using in particular an analysis called *Array Dataflow Analysis* [22, 24]. The opposite is true only if the SARE is computable [37, 70] (i.e. the SARE admits a schedule), and code generation algorithms [5, 10, 61, 62, 84] can be used to do the translation.

Polyhedral Reduced Dependence Graph One of the first steps performed by a polyhedral compiler (such as Pluto [15]) consists on building the *Polyhedral Reduced Dependence Graph* (PRDG). Its definition is the following.

Definition 2.3. A *PRDG* is a graph such that:

- Each node correspond to a statement (resp. variable) of the program, and is labeled by its iteration domain (resp. domain of the variable).
- Each edge between two nodes correspond to a dependence between two statements `S1` and `S2` (resp. variables). The source of the edge `S1` depends on the destination of the edge `S2`.

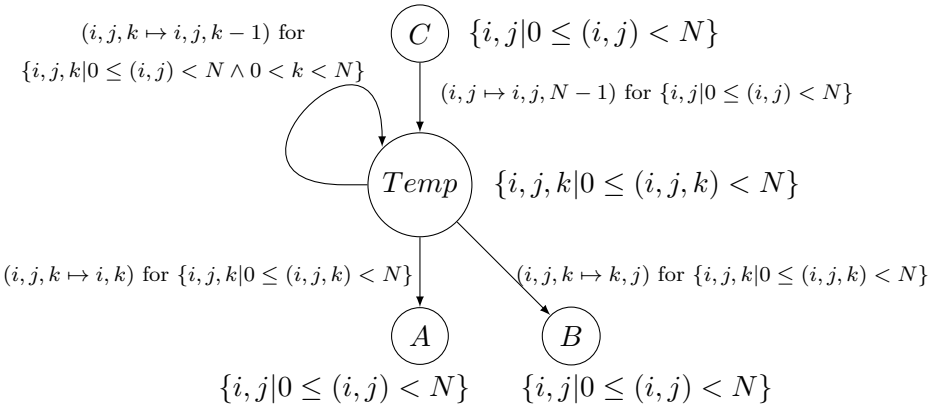


FIGURE 2.1: Polyhedral Reduced Dependence Graph of a matrix multiplication

[1] It is labeled by a *dependence polyhedron* $\{\vec{i_1}, \vec{i_2} | \dots\}$ which specifies which instances $\vec{i_1}$ of `S1` depends on which instances $\vec{i_2}$ of `S2`.

It is possible to replace the dependence polyhedron of each edges by the dependence function $(\vec{i_1} \mapsto \vec{i_2})$ and a polyhedron specifying which instances of $\vec{i_1}$ is concerned.

This graph subsumes all the information about the dependences of a program, thus is a useful intermediate representation before searching at a new schedule function [25] or legal tiling hyperplanes [15]. For example, the PRDG of our matrix multiplication example is described in Figure 2.1.

If we start from a ACL, we have to use the Array Dataflow Analysis [24] to figure out which statements depend on which other statements, and the set of instances which are involved in this dependence. This information is not exposed in a ACL if we have multiple statements writing on the same memory location. In the case of a SARE, these informations are already exposed.

[1] In the literature, we also find this definition where the edges are of the opposite direction. In that case, the dependences are said to be *dataflow*. However, in the rest of this document, we will not consider the dataflow direction, but the *true dependence* direction

Chosen Representation As a middle-ground between the previously introduced, the program representation we will use in the rest of this document will be close to the notion of PRDG:

Definition 2.4 (Program Representation)**.** A polyhedral program can be abstracted as a set of operations, each of which is described as follows:

$$\vec{i} \in \mathcal{D} : S[\vec{i}] = Expr(S_1[u_1(\vec{i})], \ldots, S_d[u_d(\vec{i})])$$

where $\vec{i}$ is the iteration vector for the statement S, $\vec{D}$ is a subset of the domain for statement S, the expression $Expr$ depends strictly on its d arguments and each argument is a result of a statement, and for $k = 0 \ldots d$, u_k is a dependence function. For every variable S, the associated domain $\mathcal{D}$ must be disjoint.

The expression $Expr$ can be either:

- A variable: $S[u(\vec{i})]$
- An operation: $op(Expr_1, \ldots, Expr_k)$ where k is the arity of the operation. When the arity is 0, this expression is a constant.
- An affine expression of the indices $f(\vec{i})$

This representation can be seen as a PRDG in which the dependence edges originating from the same operation are regrouped into hyperedges. Moreover, these hyperedges are labeled by the operation performed $Expr$. The program inputs are represented as special "dummy statements" which are sink nodes in the PRDG. A similar program representation was used by Saouter [70].

For example, the matrix multiplication computation can be expressed as:
$(\forall i, j,\ \ 0 \leq i, j < N)$ `C[i,j] = Temp[i,j,N-1];`
$(\forall i, j, k,\ \ 0 \leq i, j, k < N)$ `Temp[i,j,k] = Temp[i,j,k-1] + A[i,k] * B[k,j];`
$(\forall i, j, k,\ \ 0 = k \leq i, j < N)$ `Temp[i,j,k] = A[i,0] * B[0,j];`

The corresponding graphical representation is shown in Figure 2.2.

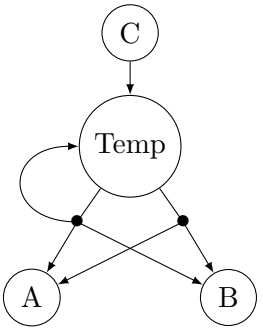


FIGURE 2.2: Graphical representation of our program representation for the matrix multiplication. We have in total 3 hyperedges, corresponding to the 3 equations needed to describe the computation

As an abuse of notation and to save space, we will often write $(\forall 0 \leq i, j < N)$ instead of $(\forall i, j$ such that $0 \leq i < N \wedge 0 \leq j < N)$ in the rest of this document.

Reductions *Reductions* and *scans* (also called *prefix computation*) are very powerful programming and computational abstractions. They can be specified as the application of associative (and often commutative) operators to (collections of) values, producing (collections of) values. Redon and Feautrier showed [65, 66, 71, 90] that for ACLs, after obtaining flow dependences as piece-wise affine functions, reductions and scans also also be detected. We extend our program representation to include reductions [47].

Formally, a reduction is the successive application of an *associative and commutative* binary operator over a set of expressions. Intuitively, a reduction is a (potentially parametric) accumulation, where the operator allows us to perform this accumulation in any order we want. In many other formalisms, only the associativity property of the reduction operator is required, however, in our case, we ask for both associativity and commutativity. This last property is needed in order to reorder the way the accumulation is performed. It is also needed in the case of accumulation over multi-dimensional expressions, for which the accumulation order is not automatically defined (contrary to the unidimensional case).

For example, a matrix multiplication can be written by using a reduction, instead of accumulating the values of $A[i,k] * B[k,j]$ in a predefined order:

$$C[i,j] \quad = \quad \sum_{k=0}^{k<N} A[i,k] * B[k,j];$$

In general, the value of a reduction at the point $\vec{i}$ is $\Big(\sum_{\pi(\vec{k})=\vec{i}} SExpr[\vec{k}] \Big)$, where $SExpr$ is an expression, and π is typically a many-to-one affine function, called the *projection function*. In the example above, we sum over the index k, thus $\pi : (i,j,k \mapsto i,j)$, and the result of the reduction is a two-dimensional variable whose indices $(i,j) \in Image(\pi)$.

All the reduction considered in this document will have a projection function which admits an integer right-inverse [48] (i.e., there exists a function π' such that $\pi \circ \pi' = Id$). For example, if we consider $(i,k \mapsto i)$, a possible integer right-inverse is $(i \mapsto i,0)$. This property is needed so that some analyses stay within the polyhedral model. In our previous example, the image of any polyhedron through the affine function $(i,k \mapsto i)$ is still a polyhedron. However, if we have a projection function $(i,j \mapsto 2i)$, this function does not admit an integer right-inverse, and the domain on which the reduction is defined is $\{i | i \text{ is even }\}$, which is not a polyhedron.

In the context of our program representation, we represent reduction as a special equation:

$$\vec{i} \in \mathcal{D}_r : S[\vec{i}] = \bigoplus_{\substack{\vec{j} \in \mathcal{D} \\ \vec{i} = \pi(\vec{j})}} Expr(S_1[f_1(\vec{j})], \ldots, S_d[f_d(\vec{j})])$$

where $\oplus$ is an associative and communicative binary operator, π is a projection function, $\mathcal{D}_r$ is the domain of the reduction statement and $\mathcal{D}$ is the domain of the reduction body.

Moreover, we allow, as a convenience, equations of the following form, which uses reductions are subexpressions:

$$\vec{i} \in \mathcal{D} : S[\vec{i}] = Expr\left(\begin{array}{l} \ldots S_k[u_k(\vec{i})], \ldots, \bigoplus_{\substack{\vec{j} \in \mathcal{D} \\ \vec{i} = \pi(\vec{j})}} ExprRed\left(\ldots, S'_k[f_k(\vec{j})], \ldots\right), \ldots \end{array}\right)$$

We can force reductions to be the top node of the right side of an equation by introducing a temporary variable for every internal reductions. Thus, allowing such equation do not modify the expressiveness of our program representation.

2.3 Program transformation

In this section, we introduce two program transformations: the *Change of Basis* transformation and the *tiling* transformation. Both transformation restructure the domains of the variables of the program, while preserving its semantics.

Change of Basis transformation The *Change of Basis* (CoB) transformation changes the domain of a variable using a unimodular function (i.e., an affine function whose determinant is 1 or -1). This function is a one-to-one mapping from the old iteration space to the new one (the unimodularity of this function being here to ensure that the new iteration space is still a polyhedron). The transformation adjusts the dependence functions of the rest of the program to ensure that exactly the same data are used: the operations remain strictly the same, but the indexing of one domain is changed.

For example, let us consider the following equation, which is a part of a bigger program:

$$(\forall (i,t), 0 < i < N \wedge 1 \leq t < T)\ temp[i,t] = temp[i-1,t-1] + temp[i,t-1] + temp[i+1,t-1];$$

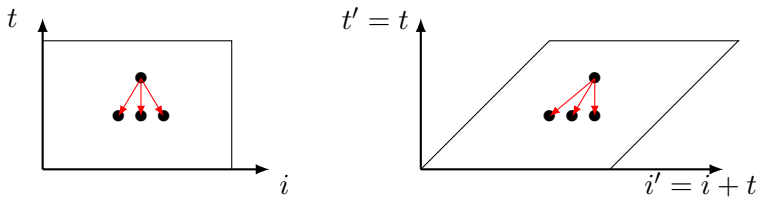


FIGURE 2.3: Change of Basis transformation, using the unimodular function $(i, t \mapsto t + i, t)$ applied on a variable with a Jacobi-like pattern of dependences

We want to apply a change of basis transformation on the variable *temp*, using the function $(i, t \mapsto i, t + i)$, which is unimodular. Figure 2.3 shows the effect of this transformation on the domain of *temp* and its dependences. The equation becomes, after transformation:

$$(\forall (i', t'), t' < i' < N + t' \wedge 1 \leq t' < T)$$
$$temp[i', t'] = temp[i' - 2, t' - 1] + temp[i' - 1, t' - 1] + temp[i', t' - 1];$$

The transformation used in this example is an instance of *time skewing* [85, 86] transformation, which is a CoB in which the time dimension (t) of a stencil computation (regular computation with only uniform dependences) is added to other space dimensions (i). It is often used to make all the dependences of a program go into the same directions (in the example toward the bottom and the left), which is a crucial property required by the tiling transformation.

Tiling transformation Tiling [35, 87] is an important program transformation which groups the instances of a loop into sets (called *tiles*), such that each tile is atomic. Figure 2.4 shows an example of tiling for a stencil computation, with 3×3 square tiles.

Because each tile is executed atomically, we cannot have cyclic dependences between two tiles (i.e., some operations from a tile depending on data produced by another tile, and vice versa). In our example in Figure 2.4, because all dependences between tiles are going toward the left or the bottom, there is no cyclic dependences between tiles. Therefore, this tiling transformation is legal. A CoB transformation can be used as a preprocessing step, in order to respect the legality condition of a tiling transformation.

$$
\begin{aligned}
(\forall i = j = 0)\ & A[i,j] = 1 \\
(\forall j = 0 < i)\ & A[i,j] = A[i-1,j] \\
(\forall i = 0 < j)\ & A[i,j] = A[i,j-1] \\
(\forall 0 < i,j)\ & A[i,j] = A[i-1,j] + A[i,j-1]
\end{aligned}
$$

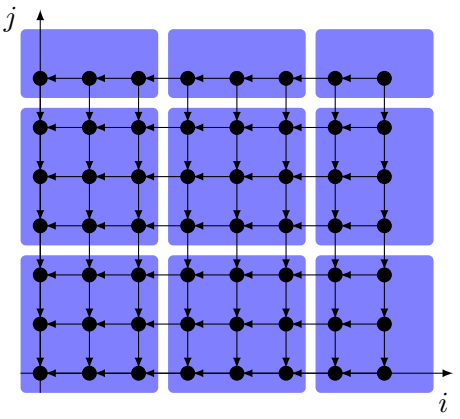


FIGURE 2.4: Example of tiling transformation, with square 3×3 tiles

We can see the tiling transformation as a two-part transformation: a first part is a reindexation of the domain tiled by introducing new dimensions. The second part is a modification of the schedule by using these newly introduced dimensions to ensure the atomicity of the tiles. We call the first part *partitioning*. It introduces new dimensions to identify a tile and a point inside a tile, doubling the number of dimensions if our tiles partition along all dimensions. This part of the transformation is always legal, and does not change the schedule (i.e., the same operations are executed at the same moment, except that the indices are not the same, like a change of basis).

The tiles of a tiling can have different shapes, and can be either of constant size (for example, a rectangle tile of sizes 16×32), or of parametric size (for example, a rectangular tile of sizes $b_1 \times b_2$). The wildest used tile shape is a hyper-parallelepiped, defined through their hyperplanes. However other shapes have been studied, such as trapezoid (with redundant computation [45]) or hexagonal [28, 67]. This transformation is useful to improve the locality of a program and to create coarse-grain parallelism opportunities.

If we have constant tile sizes, this transformation stays in the polyhedral model [35]. For example, in the hyper-rectangular case, we can substitute each of the original indices i by affine expressions of the form $t_1.i_b + i_l$, where:

- t_1 is a constant and is a tile size

- i_b is a *blocked index*, and corresponds to the tile number along the dimension of i
- i_l is a *local index*, and corresponds to local position inside the tile. Also, because of the shape of the tile, we have $0 \leq i_l < t_1$

Therefore, all domains and functions remain affine after the tiling transformation. In the example described in Figure 2.4, we have $i = 3.i_b + i_l$ and $j = 3.j_b + j_l$, where (i_b, j_b) is the tile number and (i_l, j_l) is the local position inside the tile.

In the case of parametric tile size, this transformation is **no longer polyhedral**. Indeed, if we consider the hyper-rectangular case with tiles of size $b_1, \ldots, b_d$, we have to substitute the original indices by a **quadratic expression** of the form $b_k.i_b + i_l$ where b_k is a parameter of the program. Thus, the resulting domains and functions are quadratic in general, and no longer polyhedral.

A variant of the tiling transformation, called *data tiling*, was introduced by Kodukula et al. [42]. Where the classical tiling transformation tiles the iteration space of a program, the data tiling transformation tiles the data space, and distribute the operations among *data shackle*, according to which block of data is touched.

2.4 Program equivalence and template recognition

Notion of equivalence There exist several notions of program equivalence. One of them is called *Herbrand equivalence* [3]. Assuming that we have a correspondence between the inputs and outputs of two programs, they are equivalent if and only if the computations performed by both programs are identical. This equivalence is purely structural: the same intermediate values are computed in both programs and will be used by the same operations to compute the same output, even if these operations might be organized differently. The problem of deciding the Herbrand equivalence between two SAREs is undecidable [8].

However, Herbrand equivalence does not consider any semantic properties. For example, if we compare two programs, one computing $(a + b) + c$ and the other one $a + (b + c)$, they

will not be Herbrand-equivalent, because the operations performed are different. Likewise, a program computing $(a + b)$ will not be equivalent to a program computing $(b + a)$. We will consider the Herbrand equivalence modulo associativity and commutativity properties later in this document.

Barthou's equivalence semi-algorithm Barthou et al [8] introduced an equivalence semi-algorithm for SARE, checking Herbrand equivalence. Because we will use their algorithm as a foundation of one of our algorithm in Chapter 5, we explain it in the following.

Let us consider two systems of affine recurrence equations without reductions. We want to decide equivalence without considering any semantic property (i.e., Herbrand equivalence). A semi-algorithm was proposed by Barthou et al. [8] and is based on the notion of equivalence automaton. First of all, let us introduce the notion of *Memory State Automaton* (MSA, also called *Presburger automaton*).

Definition 2.5. A Memory State Automaton (MSA) is a finite automaton where:

- Every state p is associated with an integer vector $\vec{v_p}$ of some dimension n_p,
- p_0 is the initial state,
- Every transition from p to q is associated with a firing relation $F_{p,q} \in \mathbb{Z}^{n_p} \times \mathbb{Z}^{n_q}$,
- A transition from $(p, \vec{v_p})$ to $(q, \vec{v_q})$ $((p, \vec{v_p}) \rightarrow (q, \vec{v_q}))$ can only happen if $(\vec{v_p}, \vec{v_q}) \in F_{p,q}$

We say that a state p is *accessible* iff it exists a finite path from the initial state p_0 to p for some initial vector. The *accessibility relation* of a state p is the set of pairs $(\vec{v_0}, \vec{v_p})$, such that it exists a finite path which starts from p_0 with the value of its vector being $\vec{v_0}$, and which ends up on the state p while the value of the associated vector is $\vec{v_p}$. Mathematically, we can express this relation by using a transitive closure:

$$\mathcal{R}_p = \{(\vec{v_0}, \vec{v_p}) \mid (p_0, \vec{v_0}) \rightarrow^* (p, \vec{v_p})\}$$

Equivalence automaton Barthou's algorithm is based on the notion of *equivalence automaton*. Let us consider an equivalence problem, i.e. two SAREs and a mapping between their inputs which indicate their corresponding inputs. We use the convention that expressions, operators and indices of the second SARE are "primed" (e.g., X', E'_1). The equivalence automaton is an MSA defined (and built) as follows:

- **States:** A state is labeled by an equality $e(\vec{i}) = e'(\vec{i'})$ and is associated with the vector $(\vec{i}, \vec{i'})$, where e and e' are expressions.
- **Initial state:** The initial state of the automaton is $O[\vec{i_0}] = O'[\vec{i'_0}]$, where O and O' are the outputs currently compared.
- **Final state:** There are two kinds of final states: the *success states* and the *failure states*. The *failure states* are:
  - $f(\dots) = f'(\dots)$ where f and f' are different operators,
  - $I_k[\vec{i}] = f'(\dots)$ or $f(\dots) = I'_k[\vec{i'}]$ where f and f' are operators,
  - $I_k[\vec{i}] = I'_{k'}[\vec{i'}]$ where I_k and $I'_{k'}$ are non-corresponding inputs.

  On the other side, the *accept states* are:
  - $f() = f'()$ (i.e., two identical constants)
  - $I_k = I'_{k'}$ where I_k and $I'_{k'}$ are corresponding inputs.
- **Transitions:** We have 3 types of transitions (*rules*) in the equivalence MSA: *Decompose*, *Compute* and *Generalize*, as described in Fig 2.5. The *Decompose* rule deals with operators and simply says that two expressions using the same operator are Herbrand-equivalent iff their arguments are Herbrand-equivalent. The *Compute* rule allows us to "unroll" a definition and creates a state per equations defining the unrolled variable. Note that given a value $(\vec{i}, \vec{i'})$ associated with the source state, because the branch conditions are disjoint, there is only one path which can be taken afterward.

  These two rules allow us to unroll both computations while comparing the occurring operations, starting from the outputs of both programs. However, because of recursions,

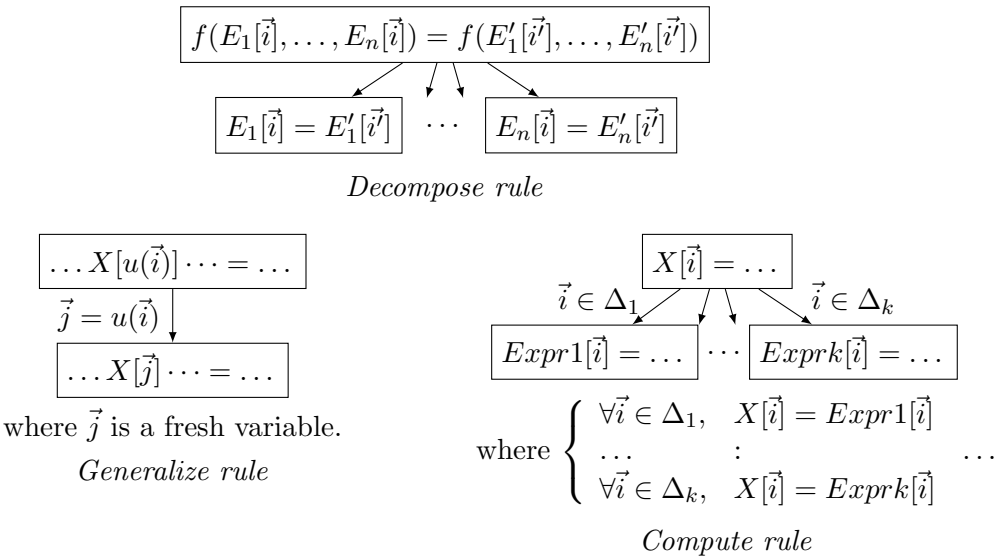


FIGURE 2.5: Construction rules for the equivalence automaton. The *Decompose rule* allows us to simplify an equality if the same operator is present on both side. The *Compute rule* unrolls a definition and create a state per case. The *Generalization rule* remove dependence functions and allow us to create cycles in the automaton.

simply unrolling both programs leads to a trace of parametric size, which is not manageable in practice. Therefore, the *Generalize* rule is used to deal with such parametric recursions. It replaces an affine expression by a fresh index. By doing so, we might end up on a state which is already built previously in the automaton. In that case, instead of creating a new state, we just add an edge going back to the previously constructed state, creating a loop in the equivalence automaton.

Deciding equivalence using the equivalence automaton Intuitively, if a state $Expr(\vec{i}) = Expr(\vec{i'})$ can be reached for a given $(\vec{i}, \vec{i'})$, then these two expressions must be equivalent in order for the two SAREs to be equivalent. Thus, the equivalence problem between the two considered SAREs can be decided by studying the accessibility sets of the success and failure states.

Theorem 2.6 (from [8])**.** *Two SAREs are equivalent iff, in their equivalence MSA:*

- *No failure state is accessible from the initial state. Indeed, a failure state corresponds to the comparison of two expressions which are obviously not equivalent.*

- *The accessibility relation of each success state is included in the identity relation. This means that, if we compare the same element of the outputs (i.e., if the vector associated with the initial state is of the form $(\vec{i_0}, \vec{i_0})$), then when we end up on a reachable success state, the compared elements must be the same (i.e., if $I[\vec{i}] = I[\vec{i'}]$ is the success state, then we must have $\vec{i} = \vec{i'}$).*

This algorithm only checks Herbrand equivalence, semantic properties like associativity/commutativity of operators are not taken into account. For instance, if we try to compare the SAREs $O = I_1 + I_2$ and $O' = I'_2 + I'_1$, the equivalent automaton will have a *decompose* rule which will generate two failure states with respective labels $(I_1 = I'_2)$ and $(I_2 = I'_1)$.

This algorithm is a reduction of the problem of program equivalence toward the problem of reachability set computation in a Presburger automaton. Both problems are undecidable in general. Hence, this equivalence algorithm is a semi-algorithm, i.e., in some situations, we cannot conclude if two programs are equivalent or not. This happens when the reachability sets are overapproximated.

Example 2.1. *As an example, let us compare the following program with itself:*

$$\begin{aligned} O &= A[N] \\ (\forall i = 0)\ A[i] &= I[0] \\ (\forall 0 < i \leq N)\ A[i] &= f(I[i], A[i-1]) \end{aligned}$$

where O is the output of the program, I the input and f is an arbitrary operation. The equivalence automaton is given in Fig 2.6.

We can notice that the automaton has a cycle: it corresponds to the comparison between the recursions of both programs. We can notice that, for every state of the automaton, we have $i = i'$ (indeed, for each transition we are modifying i, we are also modifying i' in the same way). Thus, because the reachability set of the failure states are respectively $\{i, i' \mid i = 0 \wedge i' > 0\}$ and $\{i, i' \mid i > 0 \wedge i' = 0\}$, then they are both empty. Moreover, the equalities that need to be satisfied

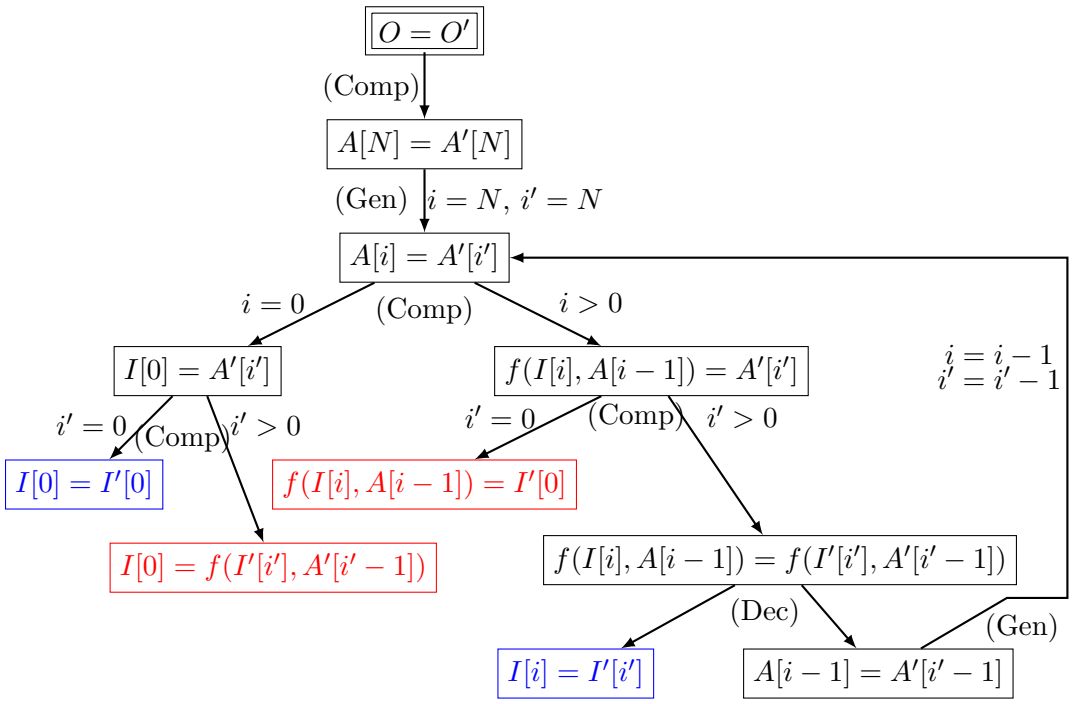


FIGURE 2.6: Equivalence automaton of Example 2.1. Success states are in blue and failure states in red. The initial state is the one inside the double-boxed rectangle

when reaching a success state are respectively $0 = 0$ *(trivially satisfied) and* $i = i'$ *(satisfied). Thus, according to Thm 2.6 these two programs are equivalent.*

Template Intuitively, a template is a program with unknown parts, which can be operations or inputs of a program. We usually try to match this template to another program or template using a template matching algorithm. Such algorithm will answer if the program and the template match, and give possible values to the unknown part of the template.

Several variants of the definition of a template exist. For example, in [3], the considered template must be *linear*, i.e. its inputs must only occur once in its expression, and the unknown part can be operations or inputs of the template.

In our case, we will assume that the unknown parts of a template correspond only to its inputs (which might correspond to a bigger computation on the side of the program). Also, we will not assume that our templates are linear, thus an input might appear several times in the template.

For example, the following template correspond to a $L^{-1}.B$ operation (where L is a lower triangular matrix, this operation is called `TRSM` in the BLAS library [46]), where L and B are inputs to the template and might correspond to a more complicated computation in a program:

$$\begin{aligned} (\forall 0 = i \leq j < N) \quad Out[i,j] = \quad & A[0,j]/L[0,0] \\ (\forall 0 < i < N \wedge 0 \leq j < N) \quad Out[i,j] = \quad & \Big(A[i,j] - \sum_{k<i} L[i,k] * Out[k,j]\Big)/L[i,i] \end{aligned}$$

where L and A are the inputs of the template and might correspond to more complicated expression in another program/template. Because the template input L occurs at multiple places, this template is not linear.

Chapter 3

Monoparametric Partitioning

In the next two chapters, we present how to divide a polyhedral computation into smaller blocks of computation, before considering each block separately and trying to recognize them as a combination of linear algebra operations (cf Chapter 5). We use a tiling transformation to distribute of the computation, which is composed of two parts: the first part is a reindexing of the domains of the program (called *partitioning*), which allows us to identify which operation belongs to which tiles. The second part gathers all the operations affiliate to a tile, and regroup them at the same place. This chapter focuses on the first part of the transformation, and Chapter 4 will focus on the later part.

Monoparametric tiling and partitioning The tiling transformation admits several variants, as presented in Section 2.3. When the tile sizes are constants (e.g., 16×16 for rectangular tiles), this transformation is called *fixed-size tiling*. When the tile sizes are parameters of the program (e.g., $b_1 \times b_2$ for rectangular tiles), this transformation is called *parametric tiling*. If we transform a polyhedral program using a fixed-size tiling transformation, we obtain another polyhedral program. However, if we use instead a parametric tiling transformation, the resulting program is no longer polyhedral.

In our context, because we want to recognize linear algebra operations inside the produced block of operations, and because the algorithm we will use require a polyhedral program as an input,

we cannot use parametric tiling to distribute the operations. It is possible to use fixed-size tiling to produce such code, but we lose in flexibility: indeed, because the tile sizes are fixed, if we want to change them, we have to reapply the tiling transformation once more.

In the next two chapters, we show that we can do better than fixed-size tiling, by using a *monoparametric tiling* transformation. A monoparametric tiling is a parametric tiling, in which the tile sizes are multiples of the **same** parameter (e.g., $b \times 2b$ for rectangular tiles). Under such a condition, this transformation produces a polyhedral program, thereby allowing a small amount of parametrization.

In this chapter, we focus on the first part of the monoparametric tiling transformation, called *monoparametric partitioning*. This part of the transformation is just a reindexing transformation, which replaces the original indices of a program into those used for tiling. The semantics of the program remains unchanged and no block of computation becomes atomic, thus legality conditions are not relevant.

Plan of the chapter In the first two sections of this chapter, we restrict ourselves to the case where the tile shapes are multi-dimensional rectangles (i.e., *hyperrectangles*). In Section 3.1, we prove the basic closure properties of the monoparametric partitioning transformation on polyhedra and affine functions. In the case of affine functions, depending on the tile shape chosen for the input and output spaces of the function, we obtain either a piecewise quasi-affine function, or a piecewise function with integer division and modulo conditions. Since the former class is preferable, we isolate the necessary and sufficient condition such that the obtained function is a piecewise quasi-affine function.

These two closure properties are the main building blocks in order to apply the monoparametric partitioning transformation to a polyhedral program, as presented in Section 3.2. Because of the condition about the monoparametric partitioning of affine functions, we have to be careful about the dependence functions of a program and the tile shape (a.k.a., in the rectangular case, the *ratio of the tile*) used for variables. In order to alleviate the need to specify a ratio for each variable of a program, we present an algorithm which derives automatically the missing ratio

of a program by finding the minimal values of a ratio for each variable, while avoiding modulo conditions in the resulting program.

In Section 3.3, we consider general tile shapes. First, we show that we can still define the monoparametric partitioning transformation for a general polyhedral shape, and prove that the closure properties on polyhedra and affine functions are still valid. The application of these closure properties to a polyhedral program is similar to their application for the hyperrectangular case. We extend our ratio derivation algorithm so that it manages any arbitrary tile shape, while not introducing modulo conditions.

Finally, we conclude this chapter in Section 3.4 with some additional remarks about the monoparametric partitioning transformation.

3.1 Hyperrectangular Monoparametric partitioning

In this section, we focus on the two main mathematical objects in our program representation: polyhedra and affine functions. We show that applying a monoparametric partitioning transformation to these objects gives us, respectively, a union of polyhedra, and a piecewise quasi-affine function. These operations will be applied to tile a complete program in Section 3.2. In Section 3.3 we extend these two properties to any general shape and to complete programs.

3.1.1 Monoparametric partitioning of polyhedra

Monoparametric partitioning Let us first define what is the monoparametric partitioning transformation in the hyperrectangular case.

Given a n-dimensional space $\mathbb{Z}^n$, let us introduce a *block size parameter* b (also called *tile size parameter*) and a diagonal matrix D of size n called *ratio of a tile*, whose coefficients are strictly positive and used to specify the "shape" of the tile. These informations define a hyperrectangular tiling of the space, the tile size being $b.D.\vec{1}$, where $\vec{1}$ is a n-dimensional vector with only 1 elements.

The monoparametric partitioning transformation $\mathcal{T}_{b,D}$ maps an index point $\vec{i} \in \mathbb{Z}^n$ in the original space to a point $(\vec{i_b}, \vec{i_l}) \in \mathbb{Z}^{2n}$ in the tiled space, such that $\vec{i_b}$ is the number of the tile in which $\vec{i}$ belongs, and $\vec{i_l}$ is the local coordinate of $\vec{i}$ inside its tile. $\vec{i_b}$ is called the *block indices* and $\vec{i_l}$ the *local indices* (c.f. Figure 3.1). This transformation is similar to a "strip mining" transformation [50].

Formally, we define the monoparametric partitioning transformation as the following:

Definition 3.1. Given the block size parameter b and a diagonal matrix D of ratio of a tile, the hyperrectangular monoparametric partitioning transformation associated to this tiling is:

$$\mathcal{T}_{b,D} = \begin{cases} \mathbb{Z}^n & \mapsto & \mathbb{Z}^{2n} \\ \vec{i} & \mapsto & (\vec{i_b}, \vec{i_l}) = (\left\lfloor \frac{\vec{i}}{b.D.\vec{1}} \right\rfloor, \vec{i} \bmod (b.D.\vec{1})) \end{cases}$$

where we have extended the division, modulo and floor operation elementwise to vectors.

The inverse of a monoparametric partitioning, $\mathcal{T}^{-1}_{b,\vec{D}}$ is:

$$\mathcal{T}^{-1}_{b,D}(\vec{i_b}, \vec{i_l}) = b.D.\vec{i_b} + \vec{i_l}$$

where the product and sum are elementwise.

Monoparametric partitioning applied to a polyhedron Let us consider a polyhedron $\mathcal{D} = \left\{ \vec{i} \mid \dots \right\} \subset \mathbb{Z}^n$ and a monoparametric partitioning transformation $\mathcal{T}_{b,D}$. We want to compute the image of $\mathcal{D}$ by this monoparametric partitioning transformation ($\Delta = \mathcal{T}_{b,D}(\mathcal{D})$). In order to do that, we have to translate the constraints of $\mathcal{D}$, which works on the original indices $\vec{i}$, into constraints of Δ, working on the block and local indices $(\vec{i_b}, \vec{i_l})$. We also assume that all parameters $\vec{p}$ can be decomposed into the *block parameters* $\vec{p_b}$ and the *local parameters* $\vec{p_l}$, where $\vec{p} = b.\vec{p_b} + \vec{p_l}$.

Starting from the constraints of $\mathcal{D}$, by eliminating the old indices $\vec{i}$ and parameters $\vec{p}$, it is possible to obtain a disjunction of integral affine constraints on the block and local indices and parameters, expressing Δ as a finite union of polyhedra.

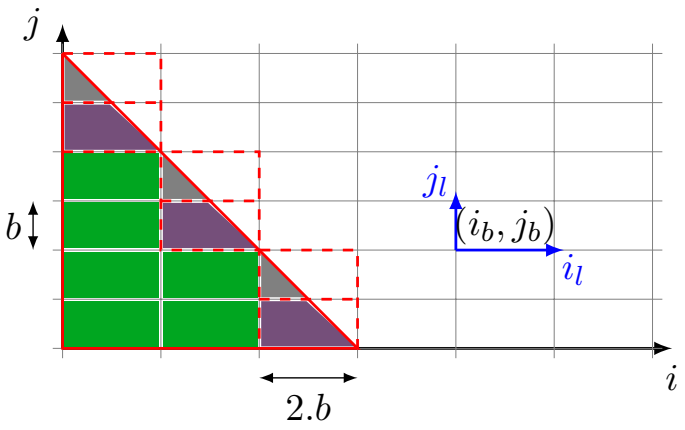


FIGURE 3.1: A 2 dimensional monoparametric partitioning. The tiles are rectangles of ratio 2×1, and the domain is $\mathcal{D} = \{i, j \mid 0 \leq i, j \wedge i + j < N\}$. Each tile is uniquely identify by the block indices (i_b, j_b). A point inside a tile is identify by the local indices (i_l, j_l). When partitioning $\mathcal{D}$, we observe 3 kinds of tiles: the full ones (in green), the triangle ones (in gray) and the trapezoid ones (in purple). The shape of each kind of tiles and their placement can both be expressed as polyhedral sets.

For example, if we tile a triangle $\mathcal{D} = \{i, j \mid 0 \leq i, j \wedge i+j < N\}$ with square tiles (assuming that the block size parameter divides N), we have two tile shapes: the tiles along the diagonal are triangles, all internal tiles are full squares. Therefore, Δ is the union of two polyhedra: a one-dimensional collection of triangles corresponding to the diagonal tiles, and a two dimensional, triangular collection of squares corresponding to the interior tiles.

More interestingly, if the same triangle $\mathcal{D} = \{i, j \mid 0 \leq i, j \wedge i + j < N\}$ is tiled with $2b \times b$ rectangles as in Fig. 3.1, we get three sets of shapes (if b divides N). In the "tall-skinny" triangular region • $\{i_b, j_b \mid 0 \leq i_b, j_b \wedge 2i_b + j_b + 3 \leq N_b\}$, where $N_b = N/b$, we have full rectangles, specified by $\{i_l, j_l \mid 0 \leq i_l < 2b \wedge 0 \leq j_l < b\}$. Along the line segment • $\{i_b, j_b \mid 2i_b + j_b + 2 = N_b\}$, we have trapezoidal tiles whose shape is $\{i_l, j_l \mid 0 \leq i_l, j_l \wedge i_l + j_l < 2b \wedge j_l < b\}$. And finally, along the line segment • $\{i_b, j_b \mid 2i_b + j_b + 1 = N_b\}$, we have triangular tiles $\{i_l, j_l \mid 0 \leq i_l, j_l \wedge i_l + j_l < b\}$.

Notice how each collection itself is a disjoint polyhedron, and its constraints involve *only the block indices*. Also notice how the constraints defining each shape involve *only the local indices*, and the size parameter, b. This is not a coincidence, and the following theorem shows that Δ is *separable* in this sense.

Mathematically, the corresponding theorem is the following:

Theorem 3.2. *The image of a polyhedron* $\mathcal{D} = \{\vec{i} \mid Q.\vec{i} + Q^{(p)}.\vec{p} + \vec{q} \geq \vec{0}\}$ *by a monoparametric partitioning transformation is:*

$$\Delta = \bigcap_{c=1}^{m} \left[\biguplus_{k_c^{\min} < k_c \leq k_c^{\max}} \left\{ \vec{i_b}, \vec{i_l} \;\middle|\; \begin{array}{c} Q_c.D.\vec{i_b} + Q_c^{(p)}.\vec{p_b} + k_c = 0 \\ b.k_c \leq Q_c.\vec{i_l} + Q_c^{(p)}.\vec{p_l} + q_c \\ \vec{0} \leq \vec{i_l} < b.D.\vec{1} \end{array} \right\} \right.$$
$$\left. \uplus \left\{ \vec{i_b}, \vec{i_l} \;\middle|\; \begin{array}{c} Q_c.D.\vec{i_b} + Q_c^{(p)}.\vec{p_b} + k_c^{min} \geq 0 \\ \vec{0} \leq \vec{i_l} < b.D.\vec{1} \end{array} \right\} \right]$$

where $\vec{k}$ *enumerates the possible values of* $\left\lfloor \frac{Q.\vec{i_l} + Q^{(p)}.\vec{p_l} + \vec{q}}{b} \right\rfloor \in [|\vec{k}^{\min}; \vec{k}^{\max}|]$.

where $[|\vec{a}, \vec{b}|]$ is the set of integral points in the rectangle whose corners are $\vec{a}$ and $\vec{b}$.

Proof. Let us derive the constraints of Δ from the constraints of $\mathcal{D}$:

$$Q.\vec{i} + Q^{(p)}.\vec{p} + \vec{q} \geq \vec{0} \tag{3.1}$$

$\mathcal{D}$ is the intersection of m half planes, each one of them defined by a single constraint $Q_c.\vec{i} + Q_c^{(p)}.\vec{p} + q_c \geq 0$, for $1 \leq c \leq m$, and we consider each constraint independently. Let us use the definitions of $\vec{i_b}$, $\vec{i_l}$, $\vec{p_b}$ and $\vec{p_l}$ to eliminate $\vec{i}$ and $\vec{p}$.

$$b.Q_c.D.\vec{i_b} + Q_c.\vec{i_l} + b.Q_c^{(p)}.\vec{p_b} + Q_c^{(p)}.\vec{p_l} + q_c \geq 0 \tag{3.2}$$

Notice that these constraints are no longer linear, because of the $b.\vec{i_b}$ and $b.\vec{p_b}$ terms. To eliminate them, we divide each constraint by $b > 0$ to obtain:

$$Q_c.D.\vec{i_b} + Q_c^{(p)}.\vec{p_b} + \frac{Q_c.\vec{i_l} + Q_c^{(p)}.\vec{p_l} + q_c}{b} \geq 0$$

In general, this fraction is a rational vector. Thus, to define integer points, we take the floor of each constraint (which is valid because $a \geq 0 \Leftrightarrow \lfloor a \rfloor \geq 0$ and $\lfloor n + a \rfloor = n + \lfloor a \rfloor$ for $n \in \mathbb{Z}$):

$$Q_c.D.\vec{i_b} + Q_c^{(p)}.\vec{p_b} + \left\lfloor \frac{Q_c.\vec{i_l} + Q_c^{(p)}.\vec{p_l} + q_c}{b} \right\rfloor \geq 0 \tag{3.3}$$

Let use define $k_c(\vec{i_l}) = \left\lfloor \frac{Q_c.\vec{i_l}+Q_c^{(p)}.\vec{p_l}+q_c}{b} \right\rfloor$. Now $k_c(\vec{i_l})$ can only take a *constant non parametric* number of values. Indeed, $\vec{i_l}$ belongs to a rectangle: $0 \leq \vec{i_l} < D.b.\vec{1}$. Thus the maximum will be reached on a vertex of the rectangle, i.e., when all the coordinates of $\vec{i_l}$ are either 0 or $d.(b-1)$ (depending on the sign of its coefficient). Let us define QD_c^+ the vector of non-negative coefficients of $Q_c.D$, $Q_c^{(p)+}$ the vector of non-negative coefficients of $Q^{(p)}$, and note that $||.||_1$ denotes the L1-norm. We have:

$$k_c^{\max} = \max_{\vec{i_l}} \left\lfloor \frac{Q_c.\vec{i_l} + Q_c^{(p)}.\vec{p_l} + q_c}{b} \right\rfloor$$

Also: $Q_c.\vec{i_l} = \sum_j Q_{c,j}.\vec{i_l}(j)$. According to the remark above, the sum is maximized for $\vec{i_l}(j) = d_j.(b-1)$ if $Q_{c,j}.d_j(b-1) > 0$ and $\vec{i_l}(j) = 0$ otherwise. Hence: $\max_{\vec{i_l}} Q_c.\vec{i_l} = \sum_j \{Q_{c,j}.d_j(b-1) \mid Q_{c,j}.d_j > 0\}$, which is exactly $(b-1)||QD_c^+||_1$. Therefore:

$$\begin{aligned}
k_c^{\max} &= \left\lfloor \tfrac{||QD_c^+||_1.(b-1)+Q_c^{(p)}.\vec{p_l}+q_c}{b} \right\rfloor \\
&= ||QD_c^+||_1 + \left\lfloor \tfrac{Q_c^{(p)}.\vec{p_l}-||QD_c^+||_1+q_c}{b} \right\rfloor \\
&\leq ||QD_c^+||_1 + \left\lfloor \tfrac{||Q_c^{(p)+}||_1.(b-1)-||QD_c^+||_1+q_c}{b} \right\rfloor \\
&\leq ||QD_c^+||_1 + ||Q_c^{(p)+}||_1 + \left\lfloor \tfrac{q_c-||Q_c^{(p)+}||_1-||QD_c^+||_1}{b} \right\rfloor
\end{aligned}$$

Thus, we have a constant upper-bound on all $k_c(\vec{i_l})$. Likewise, we can show that we have a constant lower-bound on $k_c(\vec{i_l})$, therefore, $k_c(\vec{i_l})$ can only take a constant number of values. Thus, we create one polyhedron per value of $k_c(\vec{i_l})$.

Let us build the polyhedron obtained for a value of $k_c(\vec{i_l})$ in $[|k_c^{\min}; k_c^{\max}|]$. Eqn (3.3) becomes:

$$Q_c.D.\vec{i_b} + Q_c^{(p)}.\vec{p_b} + k_c(\vec{i_l}) \geq 0 \tag{3.4}$$

$k_c(\vec{i_l})$ is the quotient of the integer division in (3.3). Then there exists r_c such that $0 \leq r_c < b$ and $Q_c.\vec{i_l} + Q_c^{(p)}.\vec{p_l} + q_c = b.k_c(\vec{i_l}) + r_c$. Hence:

$$b.k_c(\vec{i_l}) \leq Q_c.\vec{i_l} + Q_c^{(p)}.\vec{p_l} + q_c < b.(k_c(\vec{i_l}) + 1) \tag{3.5}$$

Also, the constraints (3.4) and (3.5) are affine, since $k_c(\vec{i_l})$ is a constant, and all we need to do is to ensure that $\vec{i_l}$ belongs to the tile, by adding the constraint $\vec{0} \leq \vec{i_l} < b.D.\vec{1}$, and we get the desired polyhedron.

To summarize, the c^{th} constraint of (3.1) has the same set of integer solutions as the union of polyhedra obtained for each value of $k_c(\vec{i_l})$:

$$\biguplus_{k_c} \left\{ \vec{i_b}, \vec{i_l} \;\middle|\; \begin{array}{c} Q_c.D.\vec{i_b} + Q_c^{(p)}.\vec{p_b} + k_c \geq 0 \\ b.k_c \leq Q_c.\vec{i_l} + Q_c^{(p)}.\vec{p_l} + q_c < b.(k_c + 1) \\ \vec{0} \leq \vec{i_l} < b.D.\vec{1} \end{array} \right\}$$

where k_c enumerates all possible values of $\left\lfloor \frac{Q_c.\vec{i_l} + Q_c^{(p)}.\vec{p_l} + q_c}{b} \right\rfloor$ in the interval $[|k_c^{\min}; k_c^{\max}|]$.

Now, all we need to do is to intersect these unions for each constraint $c \in [|1; m|]$ to obtain the partitioning. Actually, it is possible to improve the result, as described below.

First, let us study the pattern of the constraints of the polyhedra of the union. Let us call $(Block_{k_c})$ the constraint on the block indices and $(Local_{k_c})$ the constraints on the local indices. We notice some properties among these constraints (Figure 3.2):

- Each k_c covers a different stripe of a tile (whose equations is given by $(Local_{k_c})$). The union of all these stripes, for $k_c^{\min} \leq k_c \leq k_c^{\max}$ forms a partition of the whole tile (by definition of $k_c^{\min}$ and $k_c^{\max}$).

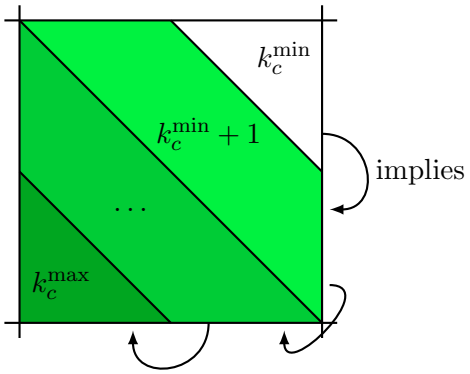


FIGURE 3.2: Stripe coverage of a tile. Given a constraint, we have obtain a disjoint union of polyhedra , each polyhedra covering a stripe of a given tile. These polyhedra are shown in different shades of green, and ranging from k_c^{min} to k_c^{max}). By examining the constraints on the block indices, we deduce that given a tile, if the stripe k_c occurs in this tile, then all the stripes$k'_c > k_c$ also occurs in this tile. Thus, we merge all of these stripes to obtain a single polyhedra per tile.

- If a tile $\vec{i_b}$ satisfies the constraint $(Block_{k_c})$ for a given k_c, then the same tile also satisfies $(Block_{k'_c})$ for every $k'_c > k_c$ (because $a \geq 0 \Rightarrow a+1 \geq 0$). In other words, if the k_cth stripe in a tile is non-empty, the tile will have all the k'_c stripes, for every $k'_c > k_c$.

Thus, if a block $\vec{i_b}$ satisfies $(Block_{k_c^{\min}})$, then it satisfy all the $(Block_{k_c})$ for $k_c \geq k_c^{\min}$ and the whole rectangular tile is covered by the union of polyhedra Δ

Also, if a block $\vec{i_b}$ satisfies exactly $(Block_{k_c})$ (i.e., if $Q_c.D.\vec{i_b} + Q_c^{(p)}.\vec{p_b} + k_c = 0$), then it does not satisfy the $(Block_{k'_c})$ for $k'_c < k_c$ and we do not have the stripes below k_c. Therefore, only the local indices i_l which satisfy $(b.k_c \leq Q_c.\vec{i_l} + Q_c^{(p)}.\vec{p_l} + q_c)$ are covered by the union of polyhedra Δ.

Using these observations, we separate the tiles into two categories: those which satisfy $(Block_{k_c^{\min}})$ (corresponding to a full tile), and those which satisfy exactly a $(Block_{k_c})$ where $k_c^{\min} < k_c$ (corresponding to a portion of the tile).

Mathematically, by splitting all of the polyhedra of the union according to the constraints $Q_c.D\vec{i_b} + Q_c^{(p)}.\vec{p_b} + k_c = 0$, $k_c^{min} < k_c \leq k_c^{max}$, then pasting them together, we obtain the

following improved expression:

$$\biguplus_{k_c^{\min}<k_c\leq k_c^{\max}} \left\{ \vec{i_b}, \vec{i_l} \middle| \begin{array}{c} Q_c.D.\vec{i_b} + Q_c^{(p)}.\vec{p_b} + k_c = 0 \\ b.k_c \leq Q_c.\vec{i_l} + Q_c^{(p)}.\vec{p_l} + q_c \\ \vec{0} \leq \vec{i_l} < b.D.\vec{1} \end{array} \right\}$$
$$\uplus \left\{ \vec{i_b}, \vec{i_l} \middle| \begin{array}{c} Q_c.D.\vec{i_b} + Q_c^{(p)}.\vec{p_b} + k_c^{min} \geq 0 \\ \vec{0} \leq \vec{i_l} < b.D.\vec{1} \end{array} \right\}$$

Thus, by intersecting all of these unions for each constraint, we obtain the expression of Δ. By distributing the intersection of the union of polyhedra, we obtain a union of disjoint polyhedra. After eliminating the empty polyhedra, the number of obtained disjoint polyhedra is the number of different tile shapes of the partitioned version of $\mathcal{D}$. □

Example 3.1. *Let us consider the following parameterized triangle:*

$$\mathcal{D} = \{i, j \mid N - 1 - i - j \geq 0 \wedge i \geq 0 \wedge j \geq 0\}$$

We consider monoparametric tiles of size $b \times b$. *Let us introduce* $\begin{pmatrix} i \\ j \end{pmatrix} = b. \begin{pmatrix} i_b \\ j_b \end{pmatrix} + \begin{pmatrix} i_l \\ j_l \end{pmatrix}$ *and, to simplify the presentation, let us assume that the parameter* N *is a multiple of the size parameter* b: $N = N_b.b$. *Then, the first inequality becomes:*

$$\begin{aligned} N - 1 - i - j \geq 0 &\Leftrightarrow N_b.b - 1 - b.i_b - i_l - b.j_b - j_l \geq 0 \\ &\Leftrightarrow N_b - i_b - j_b + \left\lfloor \tfrac{-i_l - j_l - 1}{b} \right\rfloor \geq 0 \end{aligned}$$

Let us study the values of $k_1(i_l, j_l) = \left\lfloor \frac{-i_l - j_l - 1}{b} \right\rfloor$. *Because of the sign of the numerator coefficients, the maximum is* -1 *(*$i_l = j_l = 0$*) and the minimum is* -2 *(*$i_l = j_l = b - 1$*). After*

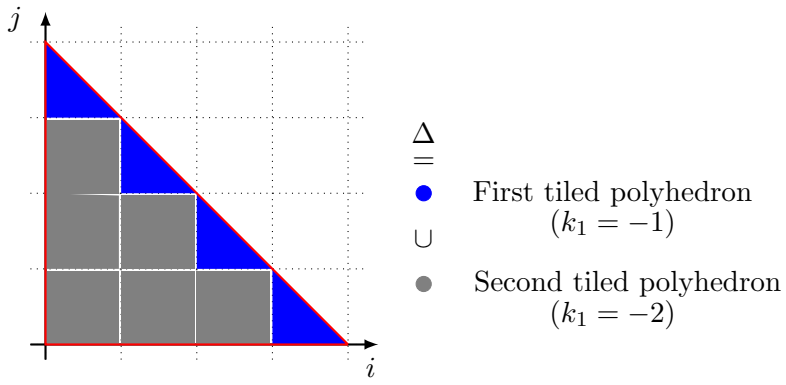


FIGURE 3.3: Obtained union of tiled polyhedra Δ for Example 3.1. The original polyhedron is a triangle, and we have assume that the tile sizes divide its sizes. We have two polyhedra in Δ: one corresponding to the full tiles, and another for the diagonal lower-triangular tiles

analyzing the two other inequalities, we obtain:

$$\Delta = \left\{ i_b, j_b, i_l, j_l \mid \begin{array}{c} N_b - i_b - j_b - 1 = 0 \\ i_b, j_b \geq 0 \\ 0 \leq i_l, j_l < b \\ -b \leq -i_l - j_l - 1 \end{array} \right\} \biguplus \left\{ i_b, j_b, i_l, j_l \mid \begin{array}{c} N_b - i_b - j_b - 2 \geq 0 \\ i_b, j_b \geq 0 \\ 0 \leq i_l, j_l < b \end{array} \right\}$$

This union of polyhedra is shown in Figure 3.3.

Example 3.2. *Let us consider the following polyhedron:* $\mathcal{D} = \{i, j \mid i + j \leq N - 1 \ \wedge \ j \leq M \ \wedge \ 0 \leq i, j\}$ *with tiles of size* $b \times b$. *Let us define* $N = N_b.b + N_l$ *and* $M = M_b.b + M_l$ *the block and local parameters, where* $0 \leq M_l < b$ *and* $0 \leq N_l < b$. *By going through the same steps as in the proof, we obtain:*

$$\left\{ \begin{array}{l} N - 1 - i - j \geq 0 \\ M - j \geq 0 \\ i \geq 0 \\ j \geq 0 \end{array} \right. \rightsquigarrow \left\{ \begin{array}{l} N_b - i_b - j_b + k_1 \geq 0 \\ M_b - j_b + k_2 \geq 0 \\ i_b + k_3 \geq 0 \\ j_b + k_4 \geq 0 \end{array} \right.$$

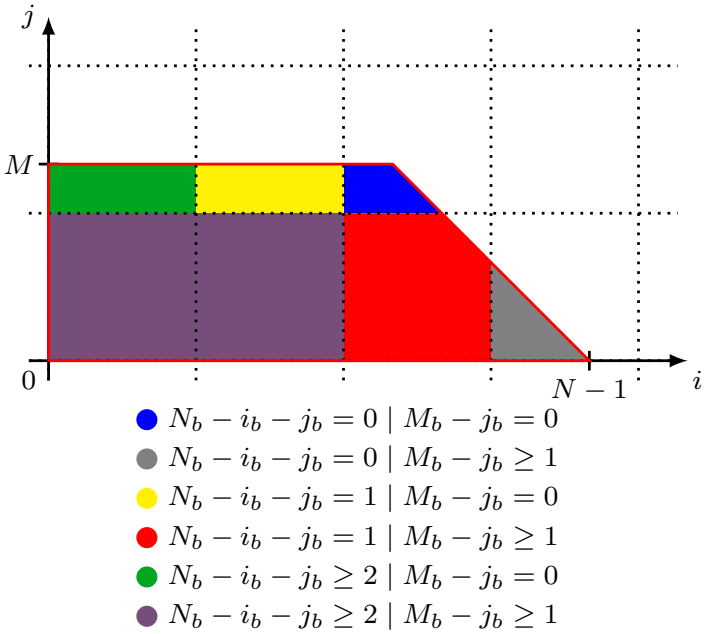


FIGURE 3.4: Obtained union of polyhedra for Example 3.2, for square tile sizes of size $b \times b$. We have in total 6 polyhedra contributing to the union. Among those 6, two of them have the same shape in the figure, but, if increasing the value of M, their shapes become different. i_b and j_b are the block indices along the i and j dimensions respectively. N_b and M_b are the integer division of the parameters N and M by the block size b

where

$$\begin{cases} k_1 = \left\lfloor \frac{N_l - i_l - j_l - 1}{b} \right\rfloor = -2, -1 \text{ or } 0 \\ k_2 = \left\lfloor \frac{M_l - j_l}{b} \right\rfloor = -1 \text{ or } 0 \\ k_3 = \left\lfloor \frac{i_l}{b} \right\rfloor = 0 \\ k_4 = \left\lfloor \frac{j_l}{b} \right\rfloor = 0 \end{cases}$$

We obtain a union of 6 polyhedra, one for each possible value of (k_1, k_2, k_3, k_4) *which are shown in Figure 3.4. We notice that two of these polyhedra (yellow and green) have the same shapes: this is because, in the situation illustrated by the figure,* $M_l \leq N_l$*, but, in general, these two polyhedra do not have the same shape. Moreover, when* $M_l \geq N_l$*, the blue and gray polyhedra will have the same shape.*

3.1.2 Monoparametric partitioning of affine functions

Monoparametric partitioning applied to an affine function Let us consider an affine function $f : (\vec{i} \mapsto Q.\vec{i}+Q^{(p)}.\vec{p}+\vec{q})$. Let us consider two monoparametric partitioning transformation $\mathcal{T}_{b,D}$ and ${\mathcal{T}'}_{b,D'}$, sharing the same block size parameter b, such that the first one corresponds to a partitioning of the input space of f, and the second one corresponds to a partitioning of the output space of f.

Given a element $\vec{i}$ of the input space, we have a first set of block and local indices for the input space $((\vec{i_b}, \vec{i_l}) = \mathcal{T}_{b,D}(\vec{i}))$. Likewise, given a element $\vec{i'}$ of the output space, we have another set of block and local indices for the output space $((\vec{i'_b}, \vec{i'_l}) = \mathcal{T}_{b,D'}(\vec{i'}))$. We also introduce the block and local parameters in the same manner than for the polyhedron case: $\vec{p} = b.\vec{p_b} + \vec{p_l}$ where $\vec{0} \leq \vec{p_l} < b.\vec{1}$.

We want to replace the original input and output indices of f by their block and local counterparts. Mathematically, this means that we want to compute $\phi = {\mathcal{T}'}_{b,D'} \circ f \circ {\mathcal{T}^{-\infty}}_{b,D}$. If f was a n to n' dimensional function, then ϕ is a $2n$ to $2n'$ dimensional function.

Like in the previous subsection, by starting with the definition of f, we derive the value of ϕ:

Theorem 3.3. *Given two monoparametric partitioning transformation ($\mathcal{T}_{b,D}$ and ${\mathcal{T}'}_{b,D'}$) and any affine function ($f(\vec{i}) = Q.\vec{i}+Q^{(p)}.\vec{p}+\vec{q}$), the composition $\phi = {\mathcal{T}'}_{b,D'} \circ f \circ \mathcal{T}^{-1}_{b,D}$ is a piecewise quasi-affine function, whose branches are:*

$$\phi(\vec{i_b}, \vec{i_l}) = \begin{pmatrix} D'^{-1}.Q.D.\vec{i_b} + D'^{-1}.Q^{(p)}.\vec{p_b} + \vec{k} \\ Q.\vec{i_l} + Q^{(p)}.\vec{p_l} + \vec{q} - b.D'.\vec{k} \end{pmatrix}$$
$$if \quad b.\vec{k} \leq D'^{-1}.Q.\vec{i_l} + D'^{-1}.Q^{(p)}.\vec{p_l} + D'^{-1}.\vec{q} < b.(\vec{k} + \vec{1})$$

for each $\vec{k} \in [|\vec{k}^{\min}; \vec{k}^{\max}|]$, and assuming that $(D'^{-1}.Q.D)$ and $(D'^{-1}.Q^{(p)})$ are integer matrices.

We will show later that the condition on $(D'^{-1}.Q.D)$ and $(D'^{-1}.Q^{(p)})$ is a necessary and sufficient condition to have only affine conditions in the piecewise quasi-affine function ϕ. If these

hypothesis are not respected, then we might end up with modulo conditions in the branches of ϕ.

Proof. Let us start from the definition of f: $\vec{i'} = Q.\vec{i} + Q^{(p)}.\vec{p} + \vec{q}$. With similar arguments as at the beginning of the proof of Theorem 3.2, we get rid of $\vec{i_l'}$ to obtain:

$$\vec{i_b'} = \left\lfloor D'^{-1}.Q.D.\vec{i_b} + D'^{-1}.Q^{(p)}.\vec{p_b} + \frac{D'^{-1}.(Q.\vec{i_l} + Q^{(p)}.\vec{p_l} + \vec{q})}{b} \right\rfloor$$

In general, if we do not have the additional hypothesis, we have no guarantee that $(D'^{-1}.Q.D.\vec{i_b})$ and $(D'^{-1}.Q^{(p)}.\vec{p_b})$ are integral vectors. In order to draw these terms outside the floor operator, we have assumed that $(D'^{-1}.Q.D)$ and $(D'^{-1}.Q^{(p)})$ are integer matrices. Using this hypothesis, we obtain:

$$\vec{i_b'} = D'^{-1}.Q.D.\vec{i_b} + D'^{-1}.Q^{(p)}.\vec{p_b} + \left\lfloor \frac{D'^{-1}.(Q.\vec{i_l} + Q^{(p)}.\vec{p_l} + \vec{q})}{b} \right\rfloor$$

By defining $\vec{k}(\vec{i_l}) = \left\lfloor \frac{D'^{-1}.(Q.\vec{i_l}+Q^{(p)}.\vec{p_l}+\vec{q})}{b} \right\rfloor$ and by conducting the same kind of analysis as previously, we manage to bound $\vec{k}(\vec{i_l})$ between $\vec{k}^{\min}$ and $\vec{k}^{\max}$. Finally, we obtain a piecewise expression of $\vec{i_b'}$, in which each branch corresponds to one value of $\vec{k}(i_l)$:

$$\begin{aligned} \vec{i_b'} &= D'^{-1}.Q.D.\vec{i_b} + D'^{-1}.Q^{(p)}.\vec{p_b} + \vec{k} \\ &\text{if} \quad b.\vec{k} \leq D'^{-1}.Q.\vec{i_l} + D'^{-1}.Q^{(p)}.\vec{p_l} + D'^{-1}.\vec{q} < b.(\vec{k} + \vec{1}) \end{aligned}$$

for each $\vec{k} \in [|\vec{k}^{\min}; \vec{k}^{\max}|]$.

We easily compute $\vec{i_l'}$ for each obtained branch by using the definition of $\vec{i_b'}$, to obtain the expression of ϕ as a piecewise quasi-affine function. Indeed, for a given branch:

$$\begin{aligned} \vec{i_l'} &= \vec{i'} - b.D'.\vec{i_b'} = Q.\vec{i} + Q^{(p)}.\vec{p} + \vec{q} - b.D'.(D'^{-1}.Q.D.\vec{i_b} + D'^{-1}.Q^{(p)}.\vec{p_b} + \vec{k}) \\ &= Q.(b.D.\vec{i_b} + \vec{i_l}) + Q^{(p)}.(b.\vec{p_b} + \vec{p_l}) + \vec{q} - b.Q.D.\vec{i_b} - b.Q^{(p)}.\vec{p_b} - b.D'.\vec{k} \\ &= Q.\vec{i_l} + Q^{(p)}.\vec{p_l} + \vec{q} - b.D'.\vec{k} \end{aligned}$$

□

$$\phi(i_b, j_b, i_l, j_l) = \begin{cases} (4i_b, M - 2j_b - 1, i_b + j_b, & 2i_l, b - j_l - 1, i_l + j_l) \\ & \text{if} \quad 0 \le i_l < b \wedge 0 \le j_l < b \wedge 0 \le i_l + j_l < 2b \\ (4i_b + 1, M - 2j_b - 1, i_b + j_b, & 2i_l - b, b - j_l - 1, i_l + j_l) \\ & \text{if} \quad b \le i_l < 2b \wedge 0 \le j_l < b \wedge 0 \le i_l + j_l < 2b \\ (4i_b, M - 2j_b - 2, i_b + j_b, & 2i_l, 2b - j_l - 1, i_l + j_l) \\ & \text{if} \quad 0 \le i_l < b \wedge b \le j_l < 2b \wedge 0 \le i_l + j_l < 2b \\ (4i_b, M - 2j_b - 2, i_b + j_b + 1, & 2i_l, 2b - j_l - 1, i_l + j_l - 2b) \\ & \text{if} \quad 0 \le i_l < b \wedge b \le j_l < 2b \wedge 2b \le i_l + j_l < 4b \\ (4i_b + 1, M - 2j_b - 1, i_b + j_b + 1, & 2i_l - b, b - j_l - 1, i_l + j_l - 2b) \\ & \text{if} \quad b \le i_l < 2b \wedge 0 \le j_l < b \wedge 2b \le i_l + j_l < 4b \\ (4i_b + 1, M - 2j_b - 2, i_b + j_b + 1, & 2i_l - b, 2b - j_l - 1, i_l + j_l - 2b) \\ & \text{if} \quad b \le i_l < 2b \wedge b \le j_l < 2b \wedge 2b \le i_l + j_l < 4b \end{cases}$$

FIGURE 3.5: Example 3.3 - obtained piecewise quasi-affine function after applying the partitioning transformation to $(i, j \mapsto 2i, N - j - 1, i + j)$, for a $2b \times 2b$ rectangular tiling on the inputs and a $b \times b$ rectangular tiling on the outputs. Each branch corresponds to a different value of the function, thus cannot be merged

Compared to the decomposition we obtained for polyhedra, we do not merge the branches according to their conditions to have a single branch per tile. Indeed, the value of the piecewise quasi-affine function is different for each branch, thus we cannot merge them.

Example 3.3. *Let us consider the affine function* $f : (i, j \mapsto 2i, N - j - 1, i + j)$.

Let us introduce $\begin{pmatrix} i \\ j \end{pmatrix} = b. \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix} . \begin{pmatrix} i_b \\ j_b \end{pmatrix} + \begin{pmatrix} i_l \\ j_l \end{pmatrix}$ *where* $0 \le i_l, j_l < 2b$ *and* $\begin{pmatrix} i' \\ j' \end{pmatrix} = b. \begin{pmatrix} i'_b \\ j'_b \end{pmatrix} + \begin{pmatrix} i'_l \\ j'_l \end{pmatrix}$ *where* $0 \le i'_l, j'_l < b$. *We assume that the parameter* N *is divisible by* b, *and we introduce* $N = N_b.b$. *We check that* $(D'^{-1}.Q.D)$ *and* $(D'^{-1}.Q^{(p)})$ *are both integral, thus we will have purely affine constraints.*

After performing the operations described previously, we obtain an expression of $\vec{i'_b}$*:*

$$\begin{bmatrix} i'_b \\ j'_b \\ k'_b \end{bmatrix} = \begin{pmatrix} 4 & 0 \\ 0 & -2 \\ 1 & 1 \end{pmatrix} \begin{bmatrix} i_b \\ j_b \end{bmatrix} + \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \cdot \begin{bmatrix} M \end{bmatrix} + \begin{bmatrix} k_1 \\ k_2 \\ k_3 \end{bmatrix}$$

where $k_1 = \left\lfloor \frac{2i_l}{b} \right\rfloor$, $k_2 = \left\lfloor \frac{-j_l - 1}{b} \right\rfloor$ *and* $k_3 = \left\lfloor \frac{i_l + j_l}{2b} \right\rfloor$. *Thus,* $0 \le k_1 \le 1$, $-2 \le k_2 \le -1$ *and* $0 \le k_3 \le 1$.

Two out of the resulting eight branches have unsatisfiable conditions. Therefore, after pruning them out, we obtain the expression of ϕ described in Figure 3.5.

Example 3.4. *Let us consider the affine function $f : (i, j \mapsto 2N + 2i + 4j - 1)$, with a block size of $b \times b$ for the input indices, and $2b$ for the output indices. The conditions are verified, and we obtain after derivation:*

$$i'_b = N_b + i_b + 2j_b + k_1 \quad \textit{where} \quad k_1 = \left\lfloor \frac{2.N_l + 2.i_l + 4.j_l - 1}{2b} \right\rfloor$$

The final result is:

$$(i'_b, i'_l) = \begin{cases} (N_b + i_b + 2.j_b - 1, N_l + i_l + 2.j_l + b) & \textit{if} \quad 2.N_l + 2.i_l + 4.j_l - 1 < 0 \\ (N_b + i_b + 2.j_b, N_l + i_l + 2.j_l) & \textit{if} \quad 0 \leq 2.N_l + 2.i_l + 4.j_l - 1 < 2b \\ (N_b + i_b + 2.j_b + 1, N_l + i_l + 2.j_l - b) & \textit{if} \quad 2b \leq 2.N_l + 2.i_l + 4.j_l - 1 < 4b \\ (N_b + i_b + 2.j_b + 2, N_l + i_l + 2.j_l - 2b) & \textit{if} \quad 4b \leq 2.N_l + 2.i_l + 4.j_l - 1 < 6b \\ (N_b + i_b + 2.j_b + 3, N_l + i_l + 2.j_l - 3b) & \textit{if} \quad 6b \leq 2.N_l + 2.i_l + 4.j_l - 1 \end{cases}$$

In Theorem 3.3, we have introduced a condition on two products of matrices to have only affine conditions in ϕ. Let us see what happens when this condition is not satisfied.

Example 3.5. *Let us consider the identity function $(i \mapsto i)$ where $D = (2)$ and $D' = (6)$. Because $Q = (1)$ and $Q^{(p)} = (0)$, $D'^{-1}.Q.D = \left(\frac{1}{3}\right)$ and $D'^{-1}.Q^{(p)} = (0)$, the conditions are not satisfied. In particular, given a point (i_b, i_l) in the input domain of this function, we need to know the result of the integer division of i_b by 3 to know in which block we end up, i.e., you need to know the value of i_b mod 3 to compute the new local index (as shown in Figure 3.6).*

Necessary and sufficient condition to avoid modulo constraints Now, let us show that the condition in Theorem 3.3 is a necessary and sufficient condition to have only affine conditions in ϕ, and when it is not respected, conditions containing modulo appear in ϕ.

Theorem 3.4. *Given two monoparametric partitioning transformations ($\mathcal{T}_b$ and $\mathcal{T}'_b$) and any affine function ($f(\vec{i}) = Q.\vec{i} + Q^{(p)}.\vec{p} + \vec{q}$). Assuming that the ratio associated with $\mathcal{T}_b$ is D and the*

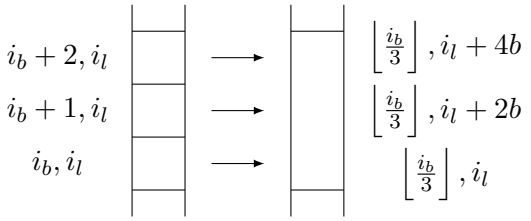


FIGURE 3.6: Example 3.5: graphical representation of the tiles on both sides of a partitionned identity function, for a $2b$ tiling on the inputs and a $6b$ tiling on the outputs. Notice that, in order to retrieve the number of the tile on the output space from (i_b, i_l), we need to perform an integer division.

ratio associated with ${\mathcal{T}'}_b$ *is* D'*, the composition* ${\mathcal{T}'}_b \circ f \circ \mathcal{T}_b^{-1}$ *has only purely affine constraints iff* $(D'^{-1}.Q.D)$ *and* $(D'^{-1}.Q^{(p)})$ *are integral matrices.*

Proof. In the proof of Theorem 3.3, we had obtained the following equality:

$$\vec{i_b'} = \left\lfloor D'^{-1}.Q.D.\vec{i_b} + D'^{-1}.Q^{(p)}.\vec{p_b} + \frac{D'^{-1}.(Q.\vec{i_l} + Q^{(p)}.\vec{p_l} + \vec{q})}{b} \right\rfloor$$

Let us consider the c-th dimension, $0 \leq c < |\vec{i_b'}|$:

$$i'_{bl,c} = \left\lfloor \frac{Q_c.D.\vec{i_b}}{D'_{c,c}} + \frac{Q_c^{(p)}.\vec{p_b}}{D'_{c,c}} + \frac{Q_c.\vec{i_l} + Q_c^{(p)}.\vec{p_l} + q_c}{D'_{c,c}.b} \right\rfloor$$

If the fraction $\frac{Q_c.D}{D'_{c,c}}$ is non-integral, it can affect the value of k_c (which was previously only a function of the local indices and parameters). This means that, depending the value of $\vec{i_b}$ and its modulo with respect to $D'_{c,c}$, the value of k_c is shifted and the cuts are different. Thus, we need to distinguish the different values of i_b modulo $D'_{c,c}$, and this is a non-affine constraint. Likewise, if the fraction $\frac{Q_c^{(p)}.\vec{p_b}}{D'_{c,c}}$ is non-integer, $\vec{p_b}$ affects the value of k_c and we have non-affine constraints on $\vec{p_b}$.

Therefore, we just have shown that if the condition is not satisfied, then we have modulo constraints. Theorem 3.3 has already shown that if the condition is satisfied, we do not have modulo constraints. Therefore, this condition is a necessary and sufficient condition. □

Example 3.5 shows what happens in practice when the condition is not satisfied.

Derivation when the condition is not satisfied If the necessary and sufficient condition is not satisfied, we can still finish the computation of ϕ and obtain a piecewise quasi-affine function with modulo conditions, as shown by the following theorem:

Theorem 3.5. *Given two monoparametric partitioning transformation ($\mathcal{T}_{b,D}$ and $\mathcal{T'}_{b,D'}$) and any affine function ($f(\vec{i}) = Q.\vec{i} + Q^{(p)}.\vec{p} + \vec{q}$), if ($D'^{-1}.Q.D$) or ($D'^{-1}.Q^{(p)}$) is not an integer matrix, the composition $\phi = \mathcal{T'}_{b,D'} \circ f \circ \mathcal{T}_{b,D}^{-1}$ is a piecewise quasi-affine function with modulo conditions in its branches.*

Proof. We consider the integer divisions of $\vec{i_b}$ by the diagonal elements of D': $\vec{i_b} = \vec{i_b}^{(div),l}.D'_{l,l} + \vec{i_b}^{(mod),l}$ where $\vec{i_b}^{(div),l}$ is the quotient and $\vec{i_b}^{(mod),l}$ is the rest of the integer division (thus, $\vec{0} \leq \vec{i_b}^{(mod),l} < D'_{l,l}.\vec{1}$). Likewise, we consider the integer divisions of $\vec{p_b}$ by the diagonal elements of D': $\vec{p_b} = \vec{p_b}^{(div),l}.D'_{l,l} + \vec{p_b}^{(mod),l}$ where $\vec{0} \leq \vec{p_b}^{(mod),l} < D'_{l,l}.\vec{1}$.

In the beginning of the derivation of Theorem 3.3, before using the conditions on the matrices ($D'^{-1}.Q.D$) and ($D'^{-1}.Q^{(p)}$), we have obtained the following equality:

$$\vec{i_b}' = \left\lfloor D'^{-1}.Q.D.\vec{i_b} + D'^{-1}.Q^{(p)}.\vec{p_b} + \frac{D'^{-1}.(Q.\vec{i_l} + Q^{(p)}.\vec{p_l} + \vec{q})}{b} \right\rfloor$$

By using the quotient and rest of the integer divisions we have introduced at the beginning of this proof, we obtain the following equality in our derivation:

$$\vec{i_{bl}}' = Q_l.D.\vec{i_b}^{(div),l} + Q_l^{(p)}.\vec{p_b}^{(div),l} + \left\lfloor \frac{Q_l.D.\vec{i_b}^{(mod),l} + Q_l^{(p)}.\vec{p_b}^{(mod),l}}{D'_{l,l}.b} + \frac{Q_l.\vec{i_l} + Q_l^{(p)}.\vec{p_l} + q_l}{D'_{l,l}.b} \right\rfloor$$

Let us define $k_l(\vec{i_b}^{(mod),l}, \vec{p_b}^{(mod),l}) = \left\lfloor \frac{Q_l.D.\vec{i_b}^{(mod),l} + Q_l^{(p)}.\vec{p_b}^{(mod),l}}{D'_{l,l}.b} + \frac{Q_l.\vec{i_l} + Q_l^{(p)}.\vec{p_l} + q_l}{D'_{l,l}.b} \right\rfloor$. Because $\vec{i_b}^{(mod),l}$ and $\vec{p_b}^{(mod),l}$ can only take a finite number of values, we do one analysis of k_l for each of their values.

The number of branches resulting from the analysis of the l-th dimension correspond to the number of values the triplet $(\vec{i_b}^{(mod),l}, \vec{p_b}^{(mod),l}, k_l)$ can take. The total number of branches of the piecewise quasi-affine function is the product of the number of branches for each dimension. Thus, the number of branches might be large, but an expression for ϕ can be computed. $\square$

Even if we manage to get an expression of ϕ when the condition is not satisfied, the number of branches is considerable, and it means going introducing modulo conditions.

Example 3.6. *Let us consider* $f : (i, j \mapsto i, j)$ *where the input indices are tiled as* $\begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} i_b \\ j_b \end{pmatrix}.b + \begin{pmatrix} i_l \\ j_l \end{pmatrix}$ *and the output indices are tiled as* $\begin{pmatrix} i' \\ j' \end{pmatrix} = \begin{pmatrix} 2i'_b \\ 3j'_b \end{pmatrix}.b + \begin{pmatrix} i'_l \\ j'_l \end{pmatrix}$. *Let us consider the first output dimension:*

$$\begin{aligned} i' = i \quad &\Leftrightarrow \quad 2.i'_b.b + i'_l = i_b.b + i_l \\ &\Rightarrow \quad i'_b = \left\lfloor \tfrac{i_b}{2} + \tfrac{i_l}{2b} \right\rfloor = i^{(1)}_{bb} + \left\lfloor \tfrac{i^{(1)}_{bl}}{2} + \tfrac{i_l}{2b} \right\rfloor \end{aligned}$$

where $i_b = 2.i^{(1)}_{bb} + i^{(1)}_{bl}$ *and* $0 \leq i^{(1)}_{bl} \leq 1$. *Likewise, we have:*

$$j'_b = j^{(2)}_{bb} + \left\lfloor \frac{j^{(2)}_{bl}}{3} + \frac{j_l}{3b} \right\rfloor$$

where $j_b = 3.j^{(2)}_{bb} + j^{(2)}_{bl}$ *and* $0 \leq j^{(2)}_{bl} \leq 2$. *Finally, we build the pieces of* ϕ *by enumerating all the possible values of* $i^{(1)}_{bl}$ *and* $j^{(2)}_{bl}$. *For example, for* $i^{(1)}_{bl} = j^{(2)}_{bl} = 0$*:*

$$\vec{k}(i_l, j_l) = \begin{pmatrix} \left\lfloor \frac{i_l}{2b} \right\rfloor & \left\lfloor \frac{j_l}{3b} \right\rfloor \end{pmatrix}^T$$

We only have one possible value for $k_1(i_l, j_l)$ and $k_2(i_l, j_l)$ (which is 0 in both cases), thus we will only have one branch in ϕ corresponding to these values. The full expression of ϕ is:

$$\phi : \begin{bmatrix} i_b \\ j_b \\ i_l \\ j_l \end{bmatrix} \mapsto \begin{cases} (i_b/2, j_b/3, & i_l, j_l)^T & \text{if } i_b \equiv 0 \bmod 2 \wedge j_b \equiv 0 \bmod 3 \\ (i_b/2, (j_b-1)/3, & i_l, j_l+b)^T & \text{if } i_b \equiv 0 \bmod 2 \wedge j_b \equiv 1 \bmod 3 \\ (i_b/2, (j_b-2)/3, & i_l, j_l+2b)^T & \text{if } i_b \equiv 0 \bmod 2 \wedge j_b \equiv 2 \bmod 3 \\ ((i_b-1)/2, j_b/3, & i_l+b, j_l)^T & \text{if } i_b \equiv 1 \bmod 2 \wedge j_b \equiv 0 \bmod 3 \\ ((i_b-1)/2, (j_b-1)/3, & i_l+b, j_l+b)^T & \text{if } i_b \equiv 1 \bmod 2 \wedge j_b \equiv 1 \bmod 3 \\ ((i_b-1)/2, (j_b-2)/3, & i_l+b, j_l+2b)^T & \text{if } i_b \equiv 1 \bmod 2 \wedge j_b \equiv 2 \bmod 3 \end{cases}$$

3.2 Hyperrectangular monoparametric partitioning program transformation

In the previous section, we showed how to apply the monoparametric partitioning transformation to a polyhedron and an affine function, the two main mathematical objects in any polyhedral program representation. In this section, we show how to apply this transformation to a complete polyhedral program. Then, we show how to choose a ratio for the local variables which do not already have a ratio assigned, which does not introduce modulo conditions in our transformed program.

3.2.1 Monoparametric partitioning program transformation

Let us consider an equation coming from a polyhedral program. This equation has one of the following two forms:

$$\begin{aligned} (\forall \vec{i} \in \mathcal{D}) : S[\vec{i}] &= Expr(S_1[u_1(\vec{i})], \ldots, S_d[u_d(\vec{i})]) \\ (\forall \vec{i} \in \mathcal{D}_r) : S[\vec{i}] &= \bigoplus_{\substack{\vec{j} \in \mathcal{D} \\ \vec{i} = \pi(\vec{j})}} Expr(S_1[f_1(\vec{j})], \ldots, S_d[f_d(\vec{j})]) \end{aligned}$$

To apply the monoparametric partitioning transformation to this program, we have to replace all the polyhedra and affine functions of this program by their monoparametric partitioned alter-egos. The number of dimensions of all domains is doubled, and, because the polyhedra and affine functions remain the same (but are expressed in a different basis) the operations performed by the program are not changed.

However, this substitution introduces piecewise quasi-affine functions in the middle of the program, which is not allowed. Thus, a post-processing step (called *normalization*) is required. Given an equation, the normalization step gathers the conditions of the branches of the piecewise quasi-affine functions from this program and compute their intersections. At this point, we obtain a list of equations, in which each element correspond to a specific combination of the branches of the piecewise quasi-affine functions of this equation. We finish by eliminating the combinations which are not satisfiable.

Thus, the normalization step flattens all the branches of the piecewise quasi-affine functions, and prune the empty branches. If we try to distinguish these two steps, the normalization does not scale. For example, in our *Jacobi1D* example, if we consider the last equation, we have a summation between 3 variables. After flattening them and before pruning the empty branches, we have a total of $4 \times 2 \times 4 = 32$ branches before pruning. This number explodes when considering stencils of higher-orders. For example, if we consider a *Jacobi2D* example, we have a summation between 9 variables, corresponding to a total of $4^8 * 2 = 2^{17}$ different combination, thus different branches before pruning. Therefore, the pruning must occur during the gathering of the branches.

Example 3.7. *Let us consider the following program, corresponding to a Jacobi1D computation:*

$$
\begin{aligned}
(\forall 0 \leq i < N) : Out[i] &= Temp[T-1, i] \\
(\forall 0 \leq i < N \wedge t = 0) : Temp[t, i] &= I[i] \\
(\forall i = 0 \wedge 0 < t < T) : Temp[t, i] &= Temp[t-1, i] \\
(\forall i = N-1 \wedge 0 < t < T) : Temp[t, i] &= Temp[t-1, i] \\
(\forall 0 < i < N-1 \wedge 0 < t < T) : Temp[t, i] &= (Temp[t-1, i-1] + \\
& \quad Temp[t-1, i] + Temp[t-1, i+1])/3
\end{aligned}
$$

where Out is an output variable and I an input, both defined over $\{i | 0 \leq i < N\}$*. For simplicity, we assume that the parameters* N *and* T *are multiples of the tile size parameter* b *(*$N = N_b.b$ *and* $T = T_b.b$*).*

We want to apply a monoparametric partitioning transformation such that the variable Temp is tiled with square tiles of size $b \times b$*, and the variables Out and I are tiles with tiles of size* b*. In order to do this, we consider each domain and dependence functions of this program and apply the monoparametric partitioning transformation on them. Finally, we substitute them with their monoparametric partitioned alter ego and to obtain the program described in Figure 3.7, before applying the normalization post-processing step.*

3.2.2 Derivation of the partitioning

While applying the monoparametric partitioning transformation, we might have different partitionings (a.k.a., ratio, in the case of rectangular tiles) interacting within an expression. For example, if we choose a ratio of 1×2 for a variable T, what ratio should we pick for a variable whose equation uses T, say $S[i, j, k] = g(\dots T[i, k+j] \dots)$, and how to adapt this expression to make the (potentially different) tiling compatible?

If we assume that the ratio of all variables were chosen beforehand, we just have to check for their compatibility, i.e., we have to check that partitioning the dependence functions do not introduce non-polyhedral modulo constraints (cf Theorem 3.4). This means that we have to check, for any dependence function $(\vec{i} \mapsto Q.\vec{i} + Q^{(p)}.\vec{p} + \vec{q})$ and ratio D and D', that $(D'^{-1}.Q.D)$ and $(D'^{-1}.Q^{(p)})$ are integral.

$$
\begin{aligned}
&\forall \left\{ \begin{array}{l} 0 \le i_b < N_b \\ 0 \le i_l < b \end{array} \right. : Out[i_b, i_l] = Temp[T_b, i_b, b-1, i_l] \\
&\forall \left\{ \begin{array}{l} 0 \le i_b < N_b \\ 0 \le i_l < b \\ t_b = t_l = 0 \end{array} \right. : Temp[t_b, i_b, t_l, i_l] = I[i_b, i_l] \\
&\forall \left\{ \begin{array}{l} i_b = i_l = 0 \\ \left(\begin{array}{c} 0 < t_b \wedge 0 \le t_l < b \\ \vee \\ t_b = 0 \wedge 0 < t_l < b \end{array} \right) \end{array} \right. : Temp[t_b, i_b, t_l, i_l] = Temp \left[\begin{array}{l} t_l = 0 : (t_b - 1, i_b, b-1, i_l) \\ 0 < t_l : (t_b, i_b, t_l - 1, i_l) \end{array} \right] \\
&\forall \left\{ \begin{array}{l} i_b = N_b - 1 \wedge i_l = b - 1 \\ \left(\begin{array}{c} 0 < t_b \wedge 0 \le t_l < b \\ \vee \\ t_b = 0 \wedge 0 < t_l < b \end{array} \right) \end{array} \right. : Temp[t_b, i_b, t_l, i_l] = Temp \left[\begin{array}{l} t_l = 0 : (t_b - 1, i_b, b-1, i_l) \\ 0 < t_l : (t_b, i_b, t_l - 1, i_l) \end{array} \right] \\
&\forall \left\{ \begin{array}{l} \left(\begin{array}{c} i_b = 0 \wedge 0 < i_l < b \\ \vee \\ i_b = N_b - 1 \wedge 0 \le i_l < b - 1 \\ \vee \\ 0 < i_b < N_b - 1 \wedge 0 < i_l < b \end{array} \right) \\ \left(\begin{array}{c} 0 < t_b \wedge 0 \le t_l < b \\ \vee \\ t_b = 0 \wedge 0 < t_l < b \end{array} \right) \end{array} \right. : Temp[t_b, i_b, t_l, i_l] = 1/3 \times \Bigg(\\
&\qquad Temp \left[\begin{array}{l} t_l = 0 \wedge i_l = 0 : (t_b - 1, i_b - 1, b-1, b-1) \\ t_l = 0 \wedge i_l > 0 : (t_b - 1, i_b, b-1, i_l - 1) \\ 0 < t_l \wedge i_l = 0 : (t_b, i_b - 1, t_l - 1, b-1) \\ 0 < t_l \wedge i_l > 0 : (t_b, i_b, t_l - 1, i_l - 1) \end{array} \right] + Temp \left[\begin{array}{l} t_l = 0 : (t_b - 1, i_b, b-1, i_l) \\ 0 < t_l : (t_b, i_b, t_l - 1, i_l) \end{array} \right] \\
&\qquad\quad + Temp \left[\begin{array}{l} t_l = 0 \wedge i_l = b-1 : (t_b - 1, i_b + 1, b-1, 0) \\ t_l = 0 \wedge i_l < b-1 : (t_b - 1, i_b, b-1, i_l + 1) \\ 0 < t_l \wedge i_l = b-1 : (t_b, i_b + 1, t_l - 1, 0) \\ 0 < t_l \wedge i_l < b-1 : (t_b, i_b, t_l - 1, i_l + 1) \end{array} \right] \Bigg)
\end{aligned}
$$

FIGURE 3.7: Jacobi1D computation, after substituting every polyhedron and affine function by its monoparametric partitioned equivalent, and before the normalization step. In order to save space, we allow union of polyhedra in the domain of the equations (instead of having one equation per polyhedron).

In a more general situation, we assume that the ratio of some variables were chosen beforehand (either by the user or by the compiler), but not all ratios were decided. In order to apply the monoparametric partitioning transformation, we need to find ratio for all the remaining variables, such that no modulo constraints are introduced in their equations.

We assume that for any cycle in the PRDG of our program, at least one variable was given a ratio. For example, if a variable S depends on itself then its ratio must be specified. Or if a variable S depends on T, which depends on S, at least one of these variables must have their ratio specified. This condition avoids recursive divisibility equation when we derive the missing

ratio of the program. About the order of derivation of the missing ratio of the program, under this condition, it is always possible to find a variable for which all the variables it uses have already been given a ratio. Therefore, by considering successively such variables, we derive a ratio for all the variables of the program.

We always pick the smallest ratios possible for an expression: indeed, let us assume that we have derived D_{T_k} for a variable T_k and let us consider an equation $S[\vec{i}] = g(\dots T_k[f_k(\vec{i})] \dots)$ in which the ratio of S is determined. According to the conditions of Theorem 3.4, we have to make sure that $(D_{T_k}^{-1}.Q.D_S)$ is integral. By taking the lowest ratio possible for T_k (i.e., the lowest values for D_{T_k}), we minimize the risk that this condition is not satisfied, thus the risk that the algorithm does not manage to avoid modulo constraints.

Ratio derivation algorithm Let us consider an equation in which all the used variables have a ratio. Two situations might arise, depending on the nature of the equation:

- **If the equation is not a reduction:** $S[\vec{i}] = g(T_1[f_1(\vec{i})], \dots, T_d[f_d(\vec{i})])$. Assuming that each dependence function f_k are of the form $f_k : (\vec{i} \mapsto Q_k.\vec{i} + Q_k^{(p)}.\vec{p} + \vec{q})$, the constraints that must be satisfied by the ratio of S are:

$$\begin{cases} (\forall 1 \leq k \leq d)\ D_{T_k}^{-1}.Q_k.D_S \text{ is integer} \\ (\forall 1 \leq k \leq d)\ D_{T_k}^{-1}.Q_k^{(p)} \text{ is integer} \end{cases}$$

  This means:

$$\begin{cases} (\forall 1 \leq k \leq d)\ (\forall i,j)\ (D_{T_k})_i \text{ divides } (Q_k)_{i,j}.(D_S)_j \\ (\forall 1 \leq k \leq d)\ (\forall i,j)\ (D_{T_k})_i \text{ divides } (Q_k^{(p)})_{i,j} \end{cases}$$

  The last condition (concerning the parameters) does not impact the ratios of S. Moreover, if this condition is not satisfied, then we must have modulo constraints on the parameters when partitioning this dependence expression. Let us now study the first condition to find the smallest ratio of S possible. We factorize $(D_{T_k})_i$ as a product of prime numbers. Because of the first condition, these prime numbers must be present either inside $(Q_k)_{i,j}$ or $(D_S)_j$ (which is the unknown). If some of them are already inside $(Q_k)_{i,j}$, they do not

need to be in $(D_S)_j$. Thus, let us introduce $(\delta_k)_{i,j}$, the product of prime factors of $(D_{T_k})_i$ which are not inside $(Q_k)_{i,j}$:

$$(\delta_k)_{i,j} = (D_{T_k})_i / gcd((D_{T_k})_i, (Q_k)_{i,j})$$

The conditions become $(\forall k)(\forall i\ j), (\delta_k)_{i,j}$ divides $(D_S)_j$. Thus, the smallest ratio we can take for S are:

$$(D_S)_j = lcm_{k,i}((\delta_k)_{i,j})$$

- **If the equation is a reduction:**

$$S[\vec{i}] = \bigoplus_{\substack{\vec{i} = \pi(\vec{j}) \\ \vec{j} \in \mathcal{D}}} g(T_0[f_0(\vec{j})], \ldots, T_d[f_d(\vec{j})]).$$

We consider two ratios for this equation: one corresponding to the subexpression of the reduction body, and one corresponding to the reduction itself. In order to determine the minimal ratios for the reduction body D_{SExpr}, we simply use the method described in the case of a normal equation. Then, all that remains is to partition the projection function $\pi : (\vec{j} \mapsto Q.\vec{j} + Q^{(p)}.\vec{p} + \vec{q})$. The conditions to avoid modulo constraints when partitioning π are:

$$\begin{cases} (\forall i, j)\ (D_S)_i \text{ divides } Q_{i,j}.(D_{SExpr})_j \\ (\forall i, j)\ (D_S)_i \text{ divides } Q^{(p)}_{i,j} \end{cases}$$

We notice that the divisibility constraints are in the opposite direction than what we had in the previous case: instead of having to find a value of $(D_S)_i$ which is divisible by another value, we have to find a value of $(D_S)_i$ which divides another value. Thus, we could just take $(D_S)_i = 1$, which is the smallest ratio possible.

However, after simplification, we might obtain a projection function which does not admit an integer right inverse [48]. For example, if we consider a ratio $D_{SExpr} = \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix}$ and a projection function $\pi : (i, j \mapsto i)$, then we obtain a piecewise quasi-affine function with

two branches:

$$(i_b, j_b, i_l, j_l) \mapsto \begin{cases} (2.i_b, i_l) & \text{when} \quad 0 \leq i_l < b \\ (2.i_b + 1, i_l - b) & \text{when} \quad b \leq i_l < 2b \end{cases}$$

Because a projection function must be a non-piecewise quasi-affine function, we need to split the reduction into two separate reductions, whose projection function correspond to a single branch of the piecewise quasi-affine function. However, the projection functions of the reductions produces by each branch do not admit an integer right inverse (it admits a rational right inverse, with a division by 2), thus each obtained reduction is defined over non-polyhedral domains (with the modulo conditions being respectively "i'_b even" and "i'_b odd").

To avoid this situation, we apply a preprocessing step to the program to make the projection *canonic*, i.e., of the form $(\vec{x}, \vec{y} \mapsto \vec{x})$. Then, we just keep the ratio of SExpr for the dimensions which are not projected. Under these circumstances, the partitioned projection function will have only one branch, of the form $(\vec{x_b}, \vec{y_b}, \vec{x_l}, \vec{y_l} \mapsto \vec{x_b}, \vec{x_l})$.

We call *valid ratios of variables* a set of ratios which do not introduce modulo conditions when we use them for a monoparametric partitioning transformation. A set of ratios which are always valid is $(1 \times 1 \times \cdots \times 1)$ for every variable, corresponding to square shapes. Thus, for any program, there always exist valid ratios of their variable.

Example 3.8. *Let us consider a matrix multiplication computation, where the ratios of A are 2×2, the ratios of B are 2×1 and the ratios of C are 2×1:*

$$(\forall 0 \leq i, j < N)\ C[i, j] = \sum_{0 \leq k < N} A[i, k] * B[k, j]$$

After examining the subexpression of the reduction, we find $2 \times 1 \times 2$ as the minimal ratio. The reduction projects the k dimension, thus the smallest ratio of the right side of the equation of C is 2×1. This ratio is exactly the same as C, thus the algorithm succeeds. A graphical representation of the result of this derivation is shown in Figure 3.8.

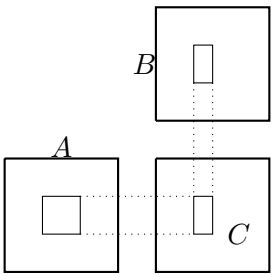


FIGURE 3.8: Example 3.8 - Chosen ratios for a matrix multiplication computation. The chosen ratio are: 2×2 for A, 2×1 for B and 1×2 for C.

Example 3.9. *Let us consider the following program, in which A has a ratio of 2, B has a ratio of 3 and Out a ratio of 1:*

$$(\forall 0 \leq i < N)\ Temp[i] = A[i+1] + B[3i]$$
$$(\forall 0 \leq i < N)\ Out[i] = Temp[i]$$

Because the ratio of Temp is not decided yet, we cannot consider first the equation of Out, and have to start with the equation of Temp. The contribution of A in this equation ($A[i+1]$) forces the ratio of Temp to be a multiple of 2. The contribution of B in this equation ($B[3i]$) forces this ratio to be a multiple of $3/3 = 1$. Therefore, the minimal ratio of Temp is 2.

Then, we consider the equation of Out. The ratio of Temp is 2 and the ratio of Out is 1. Because $1/2$ is not an integer, we are forced to introduce a modulo constraint if we partition this program, and the algorithm fails. If we had picked a multiple of 2 as a the ratio for Out, the algorithm would have succeed.

Set of possible ratios Let us show that if our algorithm does not manage to find valid ratios, then such ratios do not exist.

Theorem 3.6. *The set of valid ratios for a variable are the multiples of a single minimal ratio, which is the one found by our algorithm.*

Therefore, given a set of pre-specified ratios, if our algorithm fails to complete this specification, then no valid ratios exist.

Proof. At every step of our algorithm, the ratio we pick for each variable is always the smallest ratio which avoids modulo constraints. The key observation is that all the constraints on the ratios we consider are divisibility constraints. Thus, if we consider the prime number decomposition of the ratio we find, our algorithm discards the divisors which can be eliminated (because of the dependence functions) and only keeps the divisors which cannot be removed. Therefore, all the ratios that our algorithm find are the product of the divisors which cannot be eliminated, and hence, are the smallest valid ratios.

If our algorithm fails, then there exists an equation such that the ratio of the right side does not divide the ratio of the variable of the equation. This means that there is at least a divisor of the ratio of the right side which does not divide the ratio of the variable of the equation. Because our algorithm only keeps all the divisors which cannot be eliminated, this means that there is not valid ratio. □

We notice that a valid ratio for any variable must be a multiple of the minimal ratio we find. If our compiler framework can manage modulo constraints inside our program, we are not forced to find a *valid ratio* of the program. However, as shown in Theorem 3.5, because the number of branches are usually much larger when modulos are introduced, we might still want to avoid modulo conditions whenever possible.

3.2.3 Experimental validation

In this subsection, we present our implementation of the rectangular monoparametric partitioning, and report our experiment with this transformation.

Implementation The rectangular monoparametric partitioning transformation has been implemented in Java, using the *AlphaZ* compiler framework [89]. A C++ standalone version of this transformation for polyhedra and affine functions (manipulated through their matrix representation) is available online[1]. We use the fact that the block and local indices are separated

[1] `http://compsys-tools.ens-lyon.fr/cart/index.html`

the constraints to manipulate them separately (i.e., we manipulate cross-product of polyhedra, the first one being on the block indices and the second one on the local indices), in order to reduce the cost of the polyhedral operations performed on them.

We have implemented several options to the monoparametric partitioning transformation, in order to reduce the size of the transformed program:

- We can specify if the parameters of the program must be multiple of the block size parameter (i.e., if N is a parameter, we can force that $N = N_b.b$ and the local parameter is $N_l = 0$). This option allows us to remove a lot of corner cases. For example, if we have a two-dimensional square polyhedra $\{i, j \mid 0 \leq (i, j) < N\}$, if we do not assume that N is divisible by the block size parameter b, we obtain a union of 4 polyhedra: one for the full tiles, one for the last column of tiles, one for the last row of tiles and one for the top-right tile. If the block size parameter divides N, we only obtain a union of a single polyhedra (corresponding to the full tile).

- We can specify a minimal value for the block size parameter b. This is especially useful for uniform dependence functions. For example, if we have an equation of the form $A[i] = B[i-2]$, if the ratio of A and B are both 1, the dependence function $(i \mapsto i-2)$ access the previous tile of the variable B for $b \geq 2$. However, if $b = 1$, this dependence jumps a tile. Hence, when we partition this affine function, we need a special branch of the resulting piecewise quasi-affine function to treat this special case. Imposing that $b \geq 2$ remove such branch.

- We can specify a minimal value for the block parameters (such as N_b, where N is a parameter). For example, if we consider a Jacobi1D computation (cf Example 3.7 Page 49), we have a rectangular domain with a special computation at the bottom row ($t = 0$) and at the two extremal columns ($i = 0$ and $i = N - 1$). Assuming that the block size parameter b divides the program parameter $N = N_b.b$, when $N_b = 1$ we have a single tile spawning over the length of the domain. To avoid such extreme case, we can force $N_b \geq 2$.

Experiment on the scalability of the monoparametric partitioning transformation We want to study the scalability of our implementation of the rectangular monoparametric partitioning transformation. This means that we want to check that the time performed by our transformation in a compiler is reasonable. In addition, we want to study the scalability of an arbitrary polyhedral analysis on the transformed program (which is larger than the original program).

As our set of benchmark, we use Polybench/Alpha[2] benchmarks, an hand-written *Alpha* implementation of the *Polybench 4.0* benchmark suite. We run our experiment on a machine with an Intel Xeon E5-1650 CPU with 12 cores running at 1.6 GHz (max speed at 3.8GHz), and 31GB of memory.

We run the following experiment for each kernel:

- After parsing the program, we apply the rectangular monoparametric partitioning transformation. Because the partitioning transformation is the reindexing part of a tiling, we do not have any legality condition to respect. Thus, we select by default a rectangular tiling of ratio 1^d where d is the number of dimensions of a variable. We assume that the program parameters (N_b) are multiple of the block size parameter (b) and we impose a minimal value for both of them.
- We apply a polyhedral analysis after the monoparametric partitioning transformation, which computes the *context domain* of each node of the AST of our program. The context domain of an expression is the set of indices on which the expression value is needed to compute the output of a program. This analysis performs a tree traversal of the AST of the program, and regularly performs polyhedral operations (such as image and preimage) at certain nodes of the AST. Thus, our choice of using this analysis in order to investigate the scalability of polyhedral analysis after the partitioning transformation.

Figure 3.9 reports the time taken by each phase for all the kernel of Polybench/Alpha, and the number of node of the AST of the program after the partitioning transformation.

[2] `http://www.cs.colostate.edu/AlphaZsvn/Development/trunk/mde/edu.csu.melange.alphaz.polybench/polybench-alpha-4.0/`

Time taken (ms)	correlation	covariance	gemm	gemver	gesummv	symm	syr2k	syrk	trmm	2mm	3mm
Parsing	121	69	62	83	50	118	83	54	43	93	112
Partitioning	300	157	151	178	93	282	439	119	82	308	482
Context Domain	1147	504	163	230	162	1257	685	153	207	319	451
Num AST Nodes	110	66	21	47	29	136	36	21	25	34	39
Num Equations	10	6	2	5	3	14	3	2	3	4	6

Time taken (ms)	atax	bicg	doitgen	mvt	cholesky	durbin	gramschmidt	lu	ludcmp	trisolv	deriche
Parsing	51	51	54	55	389	121	147	106	179	74	468
Partitioning	112	113	187	159	369	266	398	284	472	139	1213
Context Domain	153	153	185	201	1197	2182	1867	1208	2672	203	2843
Num AST Nodes	25	25	13	29	113	315	123	138	216	39	659
Num Equations	4	4	2	4	15	34	20	20	30	5	40

Time taken (ms)	floyd-warshall	nussinov	adi	fdtd-2d	jacobi-1d	jacobi-2d	seidel-2d	heat-3d
Parsing	220	122	546	331	139	134	183	278
Partitioning	390	380	2393	1048	678	628	550	3275
Context Domain	335	6845	2m 32s	1m 52s	2913	58s	1m 28s	37m 13s
Num AST Nodes	27	537	11931	4194	334	2836	4684	50170
Num Equations	4	57	570	495	38	194	210	1242

FIGURE 3.9: Time taken by the hyperrectangular monoparametric partitioning transformation inside the compiler, number of nodes of the AST of the program after the partitioning transformation and number of equations of the partitioned program. All the considered stencil computations (adi to heat-3d) have an order of 1.

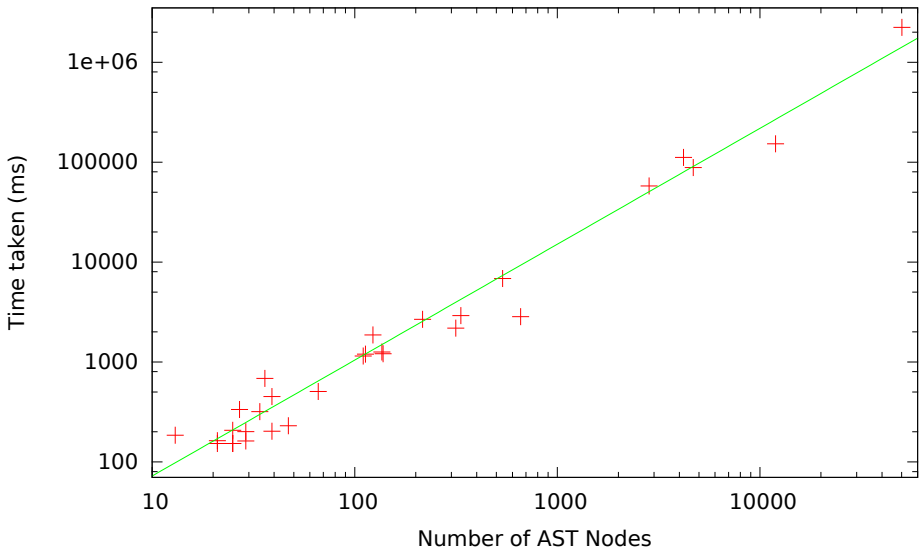


FIGURE 3.10: Time taken by the context domain polyhedral analysis against the number of AST Nodes of the program after the monoparametric partitioning transformation, plotted in a log-log scale. This plot shows that the size of the program after the transformation is linked to the time taken by the following polyhedral analysis.

The time taken by the transformation itself remains reasonable (no more than about 2 seconds for heat-3d). However, the time taken by the following polyhedral analysis (i.e., the context domain calculation) is not for the stencils kernels (the kernels from adi to heat-3d, the later taking up to about 37 minutes).

Indeed, these stencil computations have equations with several uniform dependences (of the form $(\vec{i} \mapsto \vec{i} + \vec{c})$ where $\vec{c}$ is a vector of constants). When we partition these dependences independently, each dependence becomes a piecewise quasi-affine function before the normalization post-processing step, each branch of this piecewise function corresponding to a different tile accessed. After the normalization step, we still have a lot of branches which cannot be eliminated (because of an empty domain) or merged (because each computation is unique).

In order to figure out why the context domain calculation takes so much time for some kernels, we have plotted in Figure 3.10 the time taken by the context domain analysis, against the number of nodes the AST of the program has after the monoparametric partitioning transformation. This plot shows a correlation between the time taken and the size of the AST, thus the main

reason the following polyhedral analysis takes so much time is because of the size of the program afterward.

Hence, building explicitly the entire program after the partitioning transformation is expensive for the potential polyhedral operations afterward. Notice that if we used a fixed-size partitioning instead of a monoparametric partitioning, the same issue happens. However, in our situation, we need to keep all of this information for later, in order to recognize instances of linear algebra operations. In Chapter 4, we will see how to distribute these computation into submodules of computation (called *subsystems*), which are much smaller than the whole program and can be considered independently.

3.3 General monoparametric partitioning

In Section 3.1 and Section 3.2, we have only considered *hyperrectangular* monoparametric partitioning, i.e., hyperrectangular shapes for the partitions. We now show that this theory can be extended to any polyhedral tile shape (hexagonal [28], diamond [7], etc).

First of all, let us describe what a general monoparametric partitioning is. Let us start from a general fixed size partitioning. We need 3 objects to describe it:

- A non-parametric bounded convex polyhedron $\mathcal{P}$
- A non-parametric integer lattice $\mathcal{L}$ of the tile origins (which admits a basis L) and,
- A function $\mathcal{T}$ which decomposes any point $\vec{i}$ in the following way:

$$\mathcal{T}(\vec{i}) = (\vec{i_b}, \vec{i_l}) \Leftrightarrow \vec{i} = L.\vec{i_b} + \vec{i_l} \quad \text{where } (L.\vec{i_b}) \in \mathcal{L} \text{ and } \vec{i_l} \in \mathcal{P}$$

Notice that if the decomposition is not unique, then we have overlapping tiles. If the decomposition is unique, this partitioning defines a partition of the space. Some partitionings do not have an integral lattice of tile origins (such as diamond partitioning with non-unimodular

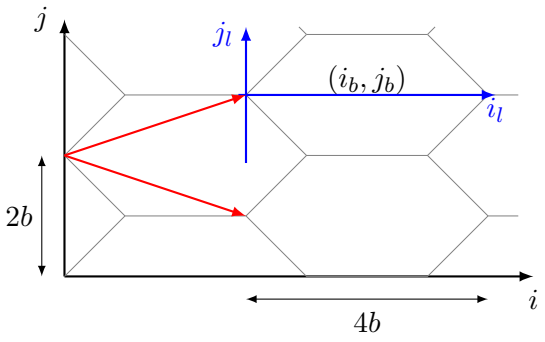


FIGURE 3.11: Example of hexagonal monoparametric blocking for a 2D space. (i_b, j_b) are the block indices, which identify a tile, (i_l, j_l) are the local indices, which identify the position of a point inside a tile. The tile shape is an hexagon with 45° slopes and of size $4b \times 2b$, and can be viewed as the homothetic scaling of a 4×2 hexagon. The red arrows correspond to a basis of the lattice of tile origins.

hyperplanes). We do not consider partitioning with overlapped tiles or with non-integral tile origins in this document.

A *homothetic transformation* $a \times \mathcal{D}$, where a is a constant and $\mathcal{D}$ is a set, is the set $a \times \mathcal{D} = \{\vec{z} \mid (\vec{z}/a) \in \mathcal{D}\}$.

Definition 3.7. A *general monoparametric partitioning* is a partitioning whose tile shape is the homothetic scaling of a fixed size partitioning, by a factor of b: $\mathcal{P}_b = b \times \mathcal{P}$. The new lattice of tile origins is $\mathcal{L}_b = b \times \mathcal{L}$ and we obtain the new partitioning function $\mathcal{T}_b$ from $\mathcal{T}$.

3.3.1 General monoparametric partitioning of polyhedra

Let us consider a n-dimensional polyhedron $\mathcal{D} = \{\vec{i} \mid Q.\vec{i} + Q^{(p)}.\vec{p} + \vec{q} \geq \vec{0}\}$ where $\vec{p}$ are the program parameters. As in Section 3.1, we want to replace $\vec{i}$ by the *block indices* $\vec{i_b}$ and the *local indices* $\vec{i_l}$, such that $\vec{i} = \mathcal{T}_b(\vec{i_b}, \vec{i_l})$ (cf Figure 3.11). We still assume that all parameters $\vec{p}$ can be decomposed into block and local parameters. Let us show that the derivation of Theorem 3.2, for a hyperrectangular monoparametric partitioning, can be adapted to a general monoparametric partitioning.

Let us consider the c-th constraint of $\mathcal{D}$: $Q_c.\vec{i} + Q_c^{(p)}.\vec{p} + q_c \geq 0$. We substitute $\vec{i}$ by $b.L.\vec{i_b} + \vec{i_l}$ where $\vec{i_l} \in \mathcal{P}_b$. By doing exactly the same operations as in the proof of Theorem 3.2, we obtain the following expression:

$$Q_c.L.\vec{i_b} + Q_c^{(p)}.\vec{p_b} + \left\lfloor \frac{Q_c.\vec{i_l} + Q_c^{(p)}.\vec{p_l} + q_c}{b} \right\rfloor \geq \vec{0}$$

We define $k_c(\vec{i_l}) = \left\lfloor \frac{Q_c.\vec{i_l}+Q_c^{(p)}.\vec{p_l}+q_c}{b} \right\rfloor$. Because $\vec{i_l} \in \mathcal{P}_b$ and $\mathcal{P}_b = b \times \mathcal{P}$ where $\mathcal{P}$ is bounded, $k_c(\vec{i_l})$ only takes a finite number of values. Because the shape of the tile is more complex than a rectangle, we cannot simply look at the sign of the coefficient to find the extremal values of $k_c(\vec{i_l})$. Because k_c is an affine function and because $\vec{i_l}$ belongs to $\mathcal{P}_b$, we use linear programming solvers (such as PIP [23]) to find the extremal values of $k_c(\vec{i_l})$. The rest of the proof caries on exactly in the same way as for Theorem 3.2.

Therefore, we obtain a union of polyhedron having the same properties as the rectangular case, for a general form of tiles:

Theorem 3.8. *The image of a polyhedron $\mathcal{D} = \{\vec{i} \mid Q.\vec{i}+Q^{(p)}.\vec{p}+\vec{q} \geq \vec{0}\}$ by a general monoparametric partitioning transformation is:*

$$\Delta = \bigcap_{c=1}^{m} \left[\biguplus_{k_c^{\min} < k_c \leq k_c^{\max}} \left\{ \vec{i_b}, \vec{i_l} \;\middle|\; \begin{array}{c} Q_c.L.\vec{i_b} + Q_c^{(p)}.\vec{p_b} + k_c = 0 \\ b.k_c \leq Q_c.\vec{i_l} + Q_c^{(p)}.\vec{p_l} + q_c \\ \vec{i_l} \in \mathcal{P}_b \end{array} \right\} \right.$$
$$\left. \uplus \left\{ \vec{i_b}, \vec{i_l} \;\middle|\; \begin{array}{c} Q_c.L.\vec{i_b} + Q_c^{(p)}.\vec{p_b} + k_c^{min} \geq 0 \\ \vec{i_l} \in \mathcal{P}_b \end{array} \right\} \right]$$

where $\vec{k}$ enumerates the possible values of $\left\lfloor \frac{Q.\vec{i_l}+Q^{(p)}.\vec{p_l}+\vec{q}}{b} \right\rfloor$.

After distributing the intersection across the unions and eliminating the empty polyhedral, we obtain as many polyhedra as the number of different tile shapes of the partitioned version of $\mathcal{D}$ (which is, at most, the number of different values of $\vec{k}$).

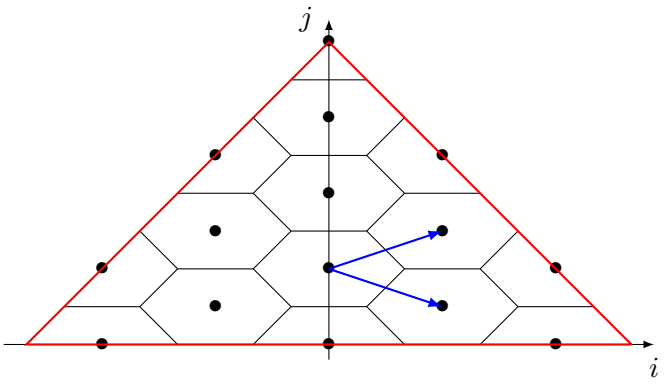


FIGURE 3.12: Polyhedron and tiling of Example 3.10. The dots correspond to the tile origins of the tiles contributing to the polyhedron. The blue arrows show the basis of the lattice of tile origins.

Example 3.10. *Let us consider the following polyhedron:* $\{i, j \mid j - i \leq N \wedge i + j \leq N \wedge 0 < j\}$ *and the following partitioning:*

- $\mathcal{P}_b = \{i, j \mid -b < j \leq b \ \wedge \ -2b < i + j \leq 2b \ \wedge \ -2b < j - i \leq 2b\}$
- $L_b = L.b.\mathbb{Z}^2$ *where* $L = \begin{bmatrix} 3 & 3 \\ 1 & -1 \end{bmatrix}$

For simplicity, we assume that $N = 6.b.N_b + 2b$*, where* N_b *is a positive integer. A graphical representation of the polyhedron and of the tiling is shown in Figure 3.12.*

Let us start with the first constraint of the polyhedron.

$$\begin{aligned} j - i \leq N \quad &\Leftrightarrow \quad 0 \leq 6.b.N_b + 2.b + b.(3.i_b + 3.j_b) + i_l - b.(i_b - j_b) - j_l \\ &\Leftrightarrow \quad 0 \leq 6.N_b + 2 + 2.i_b + 4.j_b + \left\lfloor \tfrac{i_l - j_l}{b} \right\rfloor \end{aligned}$$

where $-2b \leq i_l - j_l < 2b$*. Therefore,* $k_1 = \left\lfloor \frac{i_l - j_l}{b} \right\rfloor \in [|-2, 1|]$*. For* $k_1 = -1$ *and* 1*, the equality constraint* $6.N_b + 2.i_b + 4.j_b + 2 + k_1 = 0$ *is not satisfied (because of the parity of its terms), thus the corresponding polyhedra are empty.*

Let us examine the second constraint of the polyhedron.

$$\begin{aligned} i+j \leq N &\Leftrightarrow 0 \leq 6.b.N_b + 2.b - b.(3.i_b + 3.j_b) - i_l - L.b.(i_b - j_b) - j_l \\ &\Leftrightarrow 0 \leq 6.N_b + 2 - 4.i_b - 4.j_b + \left\lfloor \frac{-i_l - j_l}{b} \right\rfloor \end{aligned}$$

where $-2b \leq -i_l - j_l < 2b$. *Therefore* $k_2 = \left\lfloor \frac{-i_l - j_l}{b} \right\rfloor \in [|-2, 1|]$. *For the same reason as the previous constraint,* $k_2 = -1$ *and* 1 *lead to empty polyhedra.*

Let us examine the third constraint of the polyhedron.

$$\begin{aligned} 0 \leq j - 1 &\Leftrightarrow 0 \leq b.(i_b - j_b) + j_l - 1 \\ &\Leftrightarrow 0 \leq i_b - j_b + \left\lfloor \frac{j_l - 1}{b} \right\rfloor \end{aligned}$$

where $-b \leq j_l - 1 < b$. *Therefore* $k_3 = \left\lfloor \frac{j_l - 1}{b} \right\rfloor \in [|-1, 0|]$

Therefore, we obtain a union of $2 \times 2 \times 2 = 8$ *polyhedra, which are the result of the following intersections:*

$$\begin{bmatrix} \{i_b, j_b, i_l, j_l | 0 \leq 6.N_b + 2.i_b + 4.j_b \wedge (i_l, j_l) \in \mathcal{T}_b\} \\ \uplus \{i_b, j_b, i_l, j_l | 0 = 6.N_b + 2.i_b + 4.j_b + 2 \wedge (i_l, j_l) \in \mathcal{T}_b \wedge 0 \leq i_l - j_l\} \end{bmatrix}$$
$$\cap \begin{bmatrix} \{i_b, j_b, i_l, j_l | 0 \leq 6.N_b - 4.i_b - 4.j_b \wedge (i_l, j_l) \in \mathcal{T}_b\} \\ \uplus \{i_b, j_b, i_l, j_l | 0 = 6.N_b - 4.i_b - 4.j_b + 2 \wedge (i_l, j_l) \in \mathcal{T}_b \wedge 0 \leq -i_l - j_l\} \end{bmatrix}$$
$$\cap \begin{bmatrix} \{i_b, j_b, i_l, j_l | 0 \leq i_b - j_b - 1 \wedge (i_l, j_l) \in \mathcal{T}_b\} \\ \uplus \{i_b, j_b, i_l, j_l | 0 = i_b - j_b \wedge (i_l, j_l) \in \mathcal{T}_b \wedge 0 \leq j_l - 1\} \end{bmatrix}$$

3.3.2 General monoparametric partitioning of affine functions

Let us consider an affine function $f : (\vec{i} \mapsto Q.\vec{i} + Q^{(p)}.\vec{p} + \vec{q})$ and two partitionings: one for the input indices and one for the output indices (denoted with primes). Note that the "tile shapes" in the input and output dimensions, $\mathcal{P}_b$ and $\mathcal{P}'_b$ might be different. Let us show how to adapt the derivation of Theorem 3.3 to these general partitionings.

Theorem 3.9. *Given two general monoparametric partitioning transformations ($\mathcal{T}_b$ and ${\mathcal{T}'}_b$) and any affine function ($f(\vec{i}) = Q.\vec{i} + Q^{(p)}.\vec{p} + \vec{q}$), the composition ($ {\mathcal{T}'}_b \circ f \circ \mathcal{T}_b^{-1}$) is a piecewise quasi-affine function, whose branches are of the form:*

$$\phi(\vec{i_b}, \vec{i_l}) = \begin{pmatrix} L'^{-1}.Q.D.\vec{i_b} + L'^{-1}.Q^{(p)}.\vec{p_b} + \vec{k} - \vec{k'} \\ Q.\vec{i_l} + Q^{(p)}.\vec{p_l} + \vec{q} + b.L'(\vec{k'} - \vec{k}) \end{pmatrix}$$

$$if \quad \begin{cases} b.\vec{k} \leq L'^{-1}.Q.\vec{i_l} + L'^{-1}.Q^{(p)}.\vec{p_l} + L'^{-1}.\vec{q} < b.(\vec{k} + \vec{1}) \\ Q.\vec{i_l} + Q^{(p)}.\vec{p_l} + \vec{q} + b.L'(\vec{k'} - \vec{k}) \in \mathcal{P}'_b \\ \vec{i_l} \in \mathcal{P}_b \end{cases}$$

for each $\vec{k} \in [|\vec{k}^{\min}; \vec{k}^{\max}|]$, for each $\vec{k'} \in [|\vec{k'}^{\min}; \vec{k'}^{\max}|]$, where L, L' are bases of the lattices of tile origins of respectively $\mathcal{T}$ and $\mathcal{T}'$, and assuming that $(L'^{-1}.Q.L)$ and $(L'^{-1}.Q^{(p)})$ are integer matrices.

Proof. Starting from the definition of f, we perform the same manipulation as in the proof of Theorem 3.3 to obtain:

$$\vec{i_b}' + \left\lfloor \frac{L'^{-1}.\vec{i_l}'}{b} \right\rfloor = \left\lfloor L'^{-1}.Q.L.\vec{i_b} + L'^{-1}.Q^{(p)}.\vec{p_b} + \frac{L'^{-1}.(Q.\vec{i_l} + Q^{(p)}.\vec{p_l} + \vec{q})}{b} \right\rfloor$$

However, in the case of a non-rectangular partitioning, $\vec{k'}(\vec{i_l}') = \left\lfloor \frac{L'^{-1}.\vec{i_l}'}{b} \right\rfloor$ is not always equal to 0, so we cannot eliminate it, as in the rectangular case [3]. Because $\vec{i_l}' \in \mathcal{P}'_b$, $\vec{k'}(\vec{i_l}')$ only takes a finite number of values, whose extremal values can be determined through a linear programming solver.

For each value $\vec{k'}$ of $\vec{k'}(\vec{i_l}')$, we perform the same analysis as in Theorem 3.3. The new necessary and sufficient condition to avoid modulo conditions is that the matrices $L'^{-1}.Q.L$ and $L'^{-1}.Q^{(p)}$ are integral. After defining $\vec{k}(\vec{i_l}) = \left\lfloor \frac{L'^{-1}.(Q.\vec{i_l} + Q^{(p)}.\vec{p_l} + \vec{q})}{b} \right\rfloor$, we obtain a piecewise quasi-affine function in which each branch corresponds to a different value of $\vec{k}(\vec{i_l})$.

[3] Intuitively, $\vec{k'} = \vec{0}$ means that $\vec{i_l}'$ belongs to the parallelepiped $\{L'.\vec{z} | \vec{0} \leq \vec{z} < \vec{1}\}$. For a hyperrectangular tile, this is always the case, but for the hexagonal tile shown in Figure 3.12, it only corresponds to the portion of the hexagon between the two red arrows

Finally, we gather the constraints for each branch. From the definition of $\vec{k}$, we obtain the following constraint:

$$b.\vec{k} \leq L'^{-1}.Q.\vec{i_l} + L'^{-1}.Q^{(p)}.\vec{p_l} + L'^{-1}.\vec{q} < b.(\vec{k} + \vec{1})$$

From the definition of $\vec{k'}$ and after substituting $\vec{i_l}'$ by its value, we obtain the following constraint

$$b.\vec{k'} \leq L'^{-1}.(Q.\vec{i_l} + Q^{(p)}.\vec{p_l} + \vec{q} - b.L'.\vec{k}) < b.(\vec{k'} + \vec{1})$$

However, after simplification, we obtain exactly (and surprisingly) the same constraint as we got from the definition of $\vec{k}$. The two remaining constraints are $\vec{i_l} \in \mathcal{P}_b$ and $\vec{i'_l} \in \mathcal{P}'_b$ (in which $\vec{i'_l}$ can be substituted by its value).

Finally, we regroup all the branches derived for every $\vec{k'}$ to form the partitioned piecewise quasi-affine function corresponding to f. □

Note that the condition to avoid modulo constraints is that $L'^{-1}.Q.L$ and $L'^{-1}.Q^{(p)}$ are integral. This condition depends only on the lattice of the tile origins and the coefficient matrix of the polyhedron, but is independent of the shape of a tile considered.

Example 3.11. *Let us consider the identity affine function $(i, j \mapsto i, j)$, and let us consider the two following partitionings:*

- *For the input space, we choose an hexagonal tiling:*
  - $\mathcal{T}_b = \{i, j \mid -b < j \leq b \ \wedge \ -2b < i + j \leq 2b \ \wedge \ -2b < j - i \leq 2b\}$
  - $L_b = L.b.\mathbb{Z}^2$ *where* $L = \begin{bmatrix} 3 & 3 \\ 1 & -1 \end{bmatrix}$
- *For the output space, we choose a rectangular tiling, with the same lattice:*
  - $\mathcal{T}'_b = \{i, j \mid 0 \leq i < 3b \ \wedge \ 0 \leq j < 2b\}$
  - $L'_b = L'.b.\mathbb{Z}^2$ *where* $L' = \begin{bmatrix} 3 & 3 \\ 1 & -1 \end{bmatrix}$

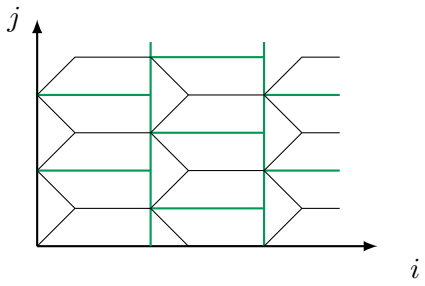


FIGURE 3.13: Overlapping of the rectangular (in green) and the hexagonal tiles introduced in Example 3.11

An overlapping of these two tilings is shown in Figure 3.13.

The derivation goes as follow:

$$\begin{aligned}
& \begin{bmatrix} i' \\ j' \end{bmatrix} = \begin{bmatrix} i \\ j \end{bmatrix} \\
\Leftrightarrow \quad & L'.b. \begin{bmatrix} i'_b \\ j'_b \end{bmatrix} + \begin{bmatrix} i'_l \\ j'_l \end{bmatrix} = L.b. \begin{bmatrix} i_b \\ j_b \end{bmatrix} + \begin{bmatrix} i_l \\ j_l \end{bmatrix} \\
\Leftrightarrow \quad & \begin{bmatrix} i'_b \\ j'_b \end{bmatrix} + L'^{-1}.\tfrac{1}{b}. \begin{bmatrix} i'_l \\ j'_l \end{bmatrix} = \begin{bmatrix} i_b \\ j_b \end{bmatrix} + L'^{-1}.\tfrac{1}{b}. \begin{bmatrix} i_l \\ j_l \end{bmatrix}
\end{aligned}$$

Because $L'^{-1} = \frac{1}{6}. \begin{bmatrix} 1 & 3 \\ 1 & -3 \end{bmatrix}$*, then the constraints become:*

$$\begin{cases} i'_b + \frac{i'_l + 3.j'_l}{6b} = i_b + \frac{i_l + 3.j_l}{6b} \\ j'_b + \frac{i'_l - 3.j'_l}{6b} = j_b + \frac{i_l - 3.j_l}{6b} \end{cases}$$

After taking the floor of these constraints:

$$\begin{cases} i'_b + \left\lfloor \frac{i'_l + 3.j'_l}{6b} \right\rfloor = i_b + \left\lfloor \frac{i_l + 3.j_l}{6b} \right\rfloor \\ j'_b + \left\lfloor \frac{i'_l - 3.j'_l}{6b} \right\rfloor = j_b + \left\lfloor \frac{i_l - 3.j_l}{6b} \right\rfloor \end{cases}$$

We define $k'_1 = \left\lfloor \frac{i'_l+3.j'_l}{6b} \right\rfloor$, $k_1 = \left\lfloor \frac{i_l+3.j_l}{6b} \right\rfloor$, $k'_2 = \left\lfloor \frac{i'_l-3.j'_l}{6b} \right\rfloor$ *and* $k_2 = \left\lfloor \frac{i_l-3.j_l}{6b} \right\rfloor$. *After analysis of the extremal values of these quantities, we obtain:*

- $k_1 \in [|-1;0|]$ *and* $k_2 \in [|-1;0|]$
- $k'_1 \in [|0;1|]$ *and* $k'_2 \in [|-1;0|]$

Therefore, we obtain a piecewise quasi-affine function with 16 branches (one for each value of (k_1, k'_1, k_2, k'_2)*). Each branch has the following form:*

$$\Big(i_b + k_1 - k'_1, j_b + k_2 - k'_2, \quad i_l + 3b(k'_1 + k'_2 - k_1 - k_2), j_l + b(k'_1 + k_2 - k_1 - k'_2)\Big)$$
$$\text{when } 0 \leq i_l + 3b(k'_1 + k'_2 - k_1 - k_2) < 3b \ \wedge \ 0 \leq j_l + b(k'_1 + k_2 - k_1 - k'_2) < 2b$$
$$k_1.b \leq i_l + 3j_l < (k_1+1).b \ \wedge \ k_2.b \leq i_l - 3j_l < (k_2+1).b$$
$$-b < j_l \leq b \ \wedge \ -2b < i_l + j_l \leq 2b \ \wedge \ -2b < j_l - i_l \leq 2b$$

3.3.3 General monoparametric partitioning program transformation

In the previous subsections, we have extended the closure properties for the polyhedron and affine function, we can apply in a similar way the general monoparametric partitioning transformation to a complete polyhedral program. In this subsection, we show how to extend the compatibility algorithm to manage general tiles.

In Section 3.2, we manipulated rectangular tile sizes $D.b$, which correspond to a special case of the lattice of tile origins, were the basis is canonic. In the general case, we manipulate lattice bases, whose vectors are the columns of an invertible matrix L. The constraints to avoid modulo conditions are of the form "the matrix $L'^{-1}.Q.L$ is integral", which is the same as saying that the input lattice $Q.L.\mathbb{Z}^n$ is a subset of the output lattice $L'.\mathbb{Z}^m$. For similar reasons as in the hyperrectangular case, we want to select the lattice of minimal basis, i.e., to minimize the size of the considered tiles.

The same derivation algorithm can be adapted to affine lattices:

- For a normal edge in the PRDG, corresponding to $\langle S, \vec{i} \rangle = g(\langle T_1, f_1(\dots, \langle T_k, f_k(\vec{i}) \dots \rangle)$, and assuming f_k is of the form $f_k : (\vec{i} \mapsto Q_k.\vec{i} + Q_k^{(p)}.\vec{p} + \vec{q_k})$, the constraints that must be satisfied by the lattice of tile origins of S are:

$$\begin{cases} (\forall 1 \leq k \leq d)\ L_{T_k}^{-1}.Q_k.L_S \text{ is integer} \\ (\forall 1 \leq k \leq d)\ L_{T_k}^{-1}.Q_k^{(p)} \text{ is integer} \end{cases}$$

  Once again, we drop the second constraint. The first constraint means that all the lattices $L_{T_k}^{-1}Q_k.L_S.\mathbb{Z}^n$ are subsets of the lattice $\mathbb{Z}^m$. Therefore, the lattices $Q_k.L_S.\mathbb{Z}^n$ are subsets of the lattices $L_{T_k}.\mathbb{Z}^m$, for $1 \leq k \leq d$. Let us define the affine functions $u_k : (\vec{z} \mapsto Q_k.\vec{z})$. The lattices $u_k(L_S.\mathbb{Z}^n)$ are subsets of the lattices $L_{T_k}.\mathbb{Z}^m$, $1 \leq k \leq d$. Because $[u(A) \subset B \Rightarrow A \subset u^{-1}(B)]$, this constraints means that the lattice $L_S.\mathbb{Z}^n$ is a subset of all the preimages of the lattice $L_{T_k}.\mathbb{Z}^m$ by the affine function u_k. Therefore, we have:

$$L_S.\mathbb{Z}^n \subset \bigcap_{1 \leq k \leq d} u_k^{-1}(L_{T_k}.\mathbb{Z}^m)$$

  We compute the right affine lattice, then take any of its bases as the value of L_S. There is no constraint on the tile shape for S, thus we select any one we want.

- If the statement is a reduction:

$$\langle S, \vec{i} \rangle = \bigoplus_{\substack{\vec{i} = \pi(\vec{j}) \\ \vec{j} \in \mathcal{D}}} g(\langle T_0, f_0(\vec{j}) \rangle, \dots, \langle T_d, f_d(\vec{j}) \rangle).$$

  For similar reasons as stated in Section 3.2, we assume that the projection function is canonic. When we tile the projection function, if the result is a piecewise quasi-affine function, the affine function inside a branch might not admit an integer right inverse. One method, to be sure that the resulting projection function will be correct, is to force the tile shape of the subexpression of the reduction to be hyperrectangular. Indeed, for such a form, projecting along a canonic dimension is trivial, but this forces the tile shape of statement S to be a rectangle. A more general way is to force the shape of the

subexpression to be an orthogonal prism, whose base is the tile shape of S and which spans across the projected dimensions.

3.4 Discussion

Adaptation to fixed-size partitioning It is possible to obtain a fixed-size partitioned code from a monoparametric partitioned code. For example, if we want to apply a rectangular partitioning with constant tile sizes $t_1 \times t_2 \times \cdots \times t_n$, we take as a block size parameter $b = gcd(t_1, t_2, \ldots, t_n)$ and we use the ratio $D = Diag(\frac{t_1}{b}, \frac{t_2}{b}, \ldots, \frac{t_n}{b})$.

Adaptation between different partitioning Let us consider an identity function $(\vec{i} \mapsto \vec{i})$, with different tilings on both sides of the function. By computing the monoparametric partioned version of this function, we obtain a piecewise quasi-affine function which can be used to adapt the indices of two statements with two different partitionings. For example, we can theoretically mix an hexagonal and a rectangular partitioning in a program.

Partitioning a subset of the indices When we apply the monoparametric partitioning transformation to a polyhedron or an affine function, we are forced to decompose **all** the original indices $\vec{i}$ into their block and local counterpart ($\vec{i_b}$ and $\vec{i_l}$). We can relax slightly this condition, by asking that an index which is inside the same constraint or affine expression than another partitioned index must be also partitioned. For example, we have a constraint $i \leq j$ and if we try to partition only i, we obtain a constraint of the form $i_b.d.b + i_l \leq j$ (where d is the ratio for the ith dimension) which cannot be transformed into an affine constraint. Hence, we also have to partition j.

Therefore, if a set of indices does not interact with the partitioned indices, we can avoid decomposing them. For example, if we consider a matrix multiplication computation $((\forall 0 \leq i < N, \forall 0 \leq j < M)\ C[i,j] = \sum_{0 \leq k < K} A[i,k] \times B[k,j])$, if we partition the indices i and j, we are not forced to partition the index k, because it does not interact with i and j.

Partitioning all the indices does not mean we need to tile all the dimensions We emphasize the fact that partitioning all the dimensions does not necessarily implies that we must eventually tile all the dimensions. Indeed, the monoparametric partitioning transformation is just a reindexing transformation which replace the original indices into new block and local indices. This transformation does not change the schedule of the program. The newly introduced indices are needed to be able to express the new tiling schedule, but we are not forced to use all of them.

For example, if we consider a variable with a 3-dimensional domain (over i, j, k) and assuming that we already have a schedule, which iterates over this domain through a lexicographic order. After partitioning, the schedule becomes $(\underbrace{i_b, i_l}_{i}, \underbrace{j_b, j_l}_{j}, \underbrace{k_b, k_l}_{k})$, which iterates over all the points exactly at the same order as the original schedule.

If we want to tile only the dimensions j and k, we can use the schedule $(j_b, k_b, \underbrace{i_b, i_l}_{i}, j_l, k_l)$, in which each tile (j_b, k_b) is a strip of computation along the i dimension. If we want to use the original index i later, inside the generated code, we can recover it though the non-linear equality $i = i_b.d.b + i_l$.

Flexibility of polyhedral code generator and monoparametric tiling Because the general parametric tiling transformation is not a polyhedral transformation, the current polyhedral compilers hard-code this transformation in their code generator. This means that if we want to change the analysis or transformations performed after the parametric tiling transformation, we have to modify the code generator. A typical example can be found in [44] where two code generators where implemented in order to exploit wavefront parallelism or canonic parallelism. Thus, we lose in flexibility in our compiler framework.

The monoparametric partitioning transformation is a polyhedral transformation, which means that the transformed program is still polyhedral. Thus, we are still able to apply any polyhedral analysis or transformation after partitioning. For example, we can introduce a new level of

partitioning almost for free, just by applying the partitioning transformation on the newly introduced local indices (which do not interact with the block indices, thus which can be partitioned independently), and without having to implement a new code generator for this strategy.

Intra-tile dependence analysis and legality condition of tiling It is possible to recover the informations about dependence between tiles from a partitioned program. Indeed, the information about which tiles depend on which tiles is explicitly given by the blocked dimensions of the partitioned dependence functions. Notice that we do not need to apply the monoparametric partitioning transformation to the whole program to recover these informations.

The intra-tile dependences are useful to determine if a tiling is legal, i.e. if there is no cyclic dependences between tiles. The method used by current polyhedral compilers is to check that all the dependences cross the tiling hyperplanes in the same direction. Instead of using this sufficient condition, we can project the domains and dependences of the program on their blocked dimensions, to build a graph whose nodes are the tiles and whose edges are the dependences between two different tiles. Checking the legality condition of tiling is equivalent to checking that there is no cyclic dependences in this graph. We can consider this graph as a reduced dependence graph and check if the corresponding program admits a schedule (using a scheduling algorithm such as in [15, 25]). This way of checking the legality of a tiling is more expensive than the commonly-used sufficient condition, but is necessary and sufficient.

An example of program on which the tiling is legal, but the hyperplane condition fails is the following:

$$(\forall -N \leq i \leq N, 0 \leq j < N) A[i,j] = A[-b-1-2i, j];$$

where b is the size of a tile. Note that this example also works for a constant tile size. Because of the dependence $(i, j \mapsto -b-1-2i, j)$, the dependences cross any tiling hyperplane in both directions. Let us show that the tiling is still legal, and that we can conclude so through our method.

When we apply the monoparametric partitioning transformation to the dependence function, we obtain the following piecewise function:

$$(i_b, j_b,\ i_l, j_l) \mapsto \begin{cases} (-2i_b - 2, j_b,\ b - 1 - 2i_l, j_l) & \text{if } -b \leq -1 - 2i_l < 0 \\ (-2i_b - 3, j_b,\ 2b - 1 - 2i_l, j_l) & \text{if } -2b \leq -1 - 2i_l < -b \end{cases}$$

When extracting the blocked part of this piecewise dependence, we build the graph of dependences between tiles, where there is only two edges: $(i_b, j_b \mapsto -2i_b - 2, j_b)$ and $(i_b, j_b \mapsto -2i_b - 3, j_b)$. By enumerating all possibilities, we can prove that there is no cycle in this graph. Thus, there is no cycle between tiles and the rectangular tiling is legal.

Extension to (fully) parametric partitioning In the next few paragraph, we will study if is it possible to adapt the monoparametric partitioning to have a fully parametric partitioning, and what makes such adaptation not possible in some situations.

In certain situations, we can use the monoparametric partitioning transformation to obtain a full-parametric partitioning. Indeed, we can partition groups of indices of a program using different tile size parameters $b_1, b_2, \ldots$ if these groups of indices do not interact with each other. For example, if we consider a matrix multiplication computation between rectangular matrices, each index does not interact with the others, thus we can obtain a fully parametrized tiled code through the monoparametric partitioning transformation while obtaining an affine program.

However, as soon as two indices with different tile size parameter interact with each other, when we try to follow the same derivation than the monoparametric partitioning transformation, we obtain a term of the form $\left\lfloor \frac{b_1}{b_2} \right\rfloor$, which is not affine.

To get an intuition of why it cannot work, let us consider a polyhedron containing only the constraint $i + j \leq 0$, and let us see what happens when we try to tile it with a parametric tile size $b_1 \times b_2$. As shown by Figure 3.14, there are two reasons why the result cannot be expressed in the polyhedral model:

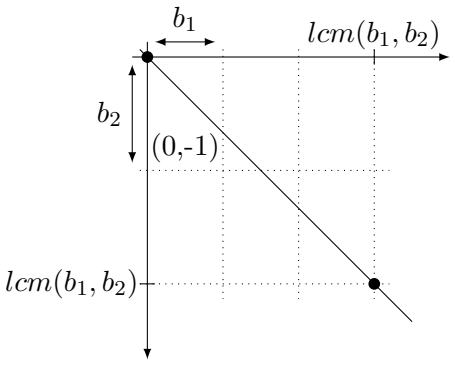


FIGURE 3.14: Parametric tiling with $b_1 \times b_2$ rectangular tile sizes on the polyhedron $i + j \leq 0$. If we study the shape of the tiles on the diagonal between $(0, 0)$ and $(lcm(b_1, b_2), lcm(b_1, b_2))$, we already have a parametric number of different tile shapes

- Let us estimate the number of different tile shapes on the diagonal. The constraints $i \leq j$ goes through the integer point $(0, 0)$, and we can show that the next integer point it is going through is $(lcm(b_1, b_2), lcm(b_1, b_2))$ where $lcm(x, y)$ is the least common multiple of x and y. Thus, we have $O(lcm(b_1, b_2)/b_1) \approx O(b_1 + b_2)$ different type of tiles.

- If we consider the shape of the diagonal tile $(i_b, j_b) = (0, -1)$, this shape happens for every tile (i_b, j_b) such that $b_1.i_b + b_2(j_b + 1) = 0$, which is not an affine constraint.

Also note that the constraints of the shapes themselves are polyhedral (ex: $i_l + j_l \leq b_2$ for the diagonal tile $(i_b, j_b) = (0, -1)$). Therefore, it is not possible to express this union as a polyhedral union, even if it might be possible to exploit the fact that each shape is polyhedral.

Chapter 4

From Partitioning to Tiling

In the previous chapter, we have presented the monoparametric partitioning transformation, which is a reindexing transformation. It introduces a new set of indices, identifying which block contains a given point, and what are its local coordinate inside such block. In this chapter, we use this transformation to express a tiling. The main addition of the monoparametric tiling transformation compared to the partitioning equivalent is that the tiles are atomic. We have seen in Section 3.4 that we can force this atomicity by changing the schedule of the program, if it admits one.

We start this chapter by presenting an extension of our program representation in Section 4.1. We allow hierarchical programs where it is possible to "call" other subprograms (called *subsystems* [19]). In addition, we impose that the subsystems are atomic. This allows us to express a tiling without having to consider the schedule of a program. It also allows us to isolate explicitly the computation of each tile, so that we can consider them independently later in Chapter 6.

In Section 4.2, we will show how to apply the monoparametric tiling transformation on a program which does not contain reductions. We first introduce the notion of *tile group* which identify set of variables which will share the same tile space. Then, we describe how we build the different subsystems corresponding to the tiles. The key part of this transformation is the classification of the tiles according to their computation, i.e., into *kind of tiles*. We show that there is a finite

non-parametric number of them, thus we can generate one subsystem per kind of tile. Then, we have to identify the inputs and outputs of each subsystem. Finally, we need to create a main system which will call the other subsystems and communicate the correct values between each of them.

In Section 4.3, we will consider program which contains reductions. In particular, because the projected dimensions of the reductions are also tiled, we have to create a new variable for each reduction. Such a variable requires a special management in order to keep the legality of the tiling. Finally, we present the extension of the monoparametric tiling transformation to program with reductions. We evaluate the scalability of our transformation in Section 4.4.

4.1 Hierarchical programs

This section presents an extension of our program representation which allows us to call other programs, i.e., structuration. We introduce a new type of equations, called *use equations*. A use equation corresponds to a call another program (called a *subsystem* [19]), provides the inputs to this program, and retrieves its outputs. Contrary to the formalism introduced in [19], we assume in this document that the subsystems are atomic. This means that all of their inputs must be ready before calling a subsystem, and all of their outputs can be retrived at once.

The syntax of a use equation is the following:

`use` $\mathcal{D}_{ext}$ name[parameters] (list of input expressions) `returns` (list of output variables);

where the extension domain $\mathcal{D}_{ext}$ is optional. The role of each object and the semantic of this use equation will be introduced incrementaly in the rest of this section. We first consider the case where a use equation does not have an extension domain, then the case where it does.

Use equation without extension domain Let us first consider the case where there is no $\mathcal{D}_{ext}$ (which is called the *extension domain*). The meaning of this equation is the following. First, the main system computes the input expressions, before calling the subsystem (called

"name") on these inputs, with a list of affine expressions of the parameters (corresponding to the parameters of the new system). The subsystem performs its computation atomically, i.e., independently for the rest of the computation. Finally, the outputs of the subsystem are retrieved and stored inside some variables of the main system.

Example 4.1. *Let us consider the following example, in which the main program computes the mean value of a vector of size N, and calls a subsystem which computes the sum of the values of a vector:*

Program "sum" :
input: $Vect$ (defined on $\{i|0 \leq i < N\}$)
output: Res (scalar)
parameter: N

$$Res = \sum_{0 \leq k < N} Vect[k];$$

Program "mean" :
input: A (defined on $\{i|0 \leq i < N\}$)
output: $Mean$ (scalar)
local: temp (scalar)
parameter: N

use sum [N] (A) returns (temp);
$Mean = temp/N;$

If we consider the PRDG view of our program representation, a use equation corresponds to a special kind of hyperedge, labelled by a program name, connected to multiple inputs and outputs nodes. For example, the PRDG of the program described in Example 4.1 is shown in Figure 4.1

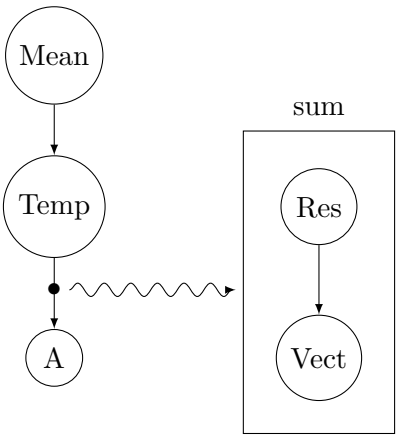


FIGURE 4.1: PRDG of the program described in Example 4.1. The use equation is represented by an hyperedge, with one source per output and one destination per input.

Use equation with extension domain Let us consider a program which performs a scalar product and let us assume that we want to use it as a subsystem to perform a matrix multiplication. In the matrix multiplication program, we need to instantiate the scalar product subsystem a parametric number of times. It can be done by using the extension domain $\mathcal{D}_{ext}$. Each integer point $i_{ext} \in \mathcal{D}_{ext}$ corresponds to one subsystem instance. This i_{ext} can be used as parameters in the rest of the use equation. More precisely:

- The indices can be used to specify the parameters of the subsystem.
- The first dimensions of the input expressions correspond to the dimensions of the extension domain (like in a functional "map")
- The first dimensions of the output variables correspond to the dimensions of the extension domain. All the results from every subsystem call are gathered in the same common variables (like in a "map")

Example 4.2. *Let us assume that we want to implement a matrix-vector product, where the matrix is lower-triangular, by using a subsystem which implements a scalar product:*

Program "scalProd" : *inputs:* $Vect1, Vect2$ *(both defined on* $\{i|0 \leq i < M\}$ *)*
output: Res (scalar)
parameter: M

$$Res = \sum_{0 \leq k < M} Vect1[k] * Vect2[k];$$

Program "triMatVectProd" : *inputs:* $Vect$ *(defined on* $\{i|0 \leq i < N\}$ *)*
L *(defined on* $\{i, j|0 \leq i \leq j < N\}$ *)*
output: vectRes (defined on $\{i|0 \leq i < N\}$ *)*
parameter: N

use$\{k|0 \leq k < N\}$ *scalProd* $[k]$
$((k, i \rightarrow i)@Vect, \quad L)$
returns $(vectRes)$;

where $(k, i \rightarrow i)@Vect$ *is a 2-dimensional expression whose value at* (k, i) *is* $Vect[i]$.

In this example, we have N different subsystems call. The k-th call computes the product of two vectors of size k. The first one is the first k elements of $Vect$*, the second one is the kth row of L. The value produced by the k-th instance of the subsystem is the k-th element of vectRes.*

4.2 Monoparametric tiling without reduction

We present how to apply the monoparametric tiling transformation such that the computation of each tile is separated into a different subsystem. In this section, we will consider programs

without reductions, before removing this restriction in the following section. The monoparametric tiling transformation is a combination of the monoparametric partitioning transformation with an *outlining* transformation. An outlining transformation is a transformation which encapsulates a portion of the computation of a system inside a new subsystem. Its reverse is called the inlining transformation.

In Subsection 4.2.1, we present the kind of code we want to obtain after the monoparametric tiling transformation through an example. Then, in Subsection 4.2.2, we talk about an adaptation to the monoparametric partitioning transformation which exposes the blocked and local indices. In order to allow tile spaces shared by several variables (which might be required by the legality conditions), we introduce the notion of *tile group* in Subsection 4.2.3. Finally, we present our transformation in Subsection 4.2.4.

4.2.1 Example - Smith Waterman

In this subsection, we present an example of application of the monoparametric tiling transformation. We consider the following program which corresponds to a Smith-Waterman computation, with no diagonal dependence:

$$\begin{aligned}
Out = \;& A[N-1, N-1] \\
(\forall i = j = 0)\; A[i,j] = \;& w[0,0]; \\
(\forall i = 0 < j < N)\; A[i,j] = \;& A[i,j-1] + w[i,j-1]; \\
(\forall j = 0 < i < N)\; A[i,j] = \;& A[i-1,j] + w[i-1,j]; \\
(\forall 0 < (i,j) < N)\; A[i,j] = \;& min(A[i-1,j] + w[i-1,j], A[i,j-1] + w[i,j-1]);
\end{aligned}$$

where w is an input of the program, and N a program parameter. A graphical representation of this program is shown in Figure 4.2.

First of all, let us remark that the rectangular tiling is legal, and let us consider a monoparametric tiling transformation with square tiles (1×1 ratio). We also assume for simplicity that the tile size parameter b divides N. We also tile the variables A and Out separately. This tiling is also shown in Figure 4.2.

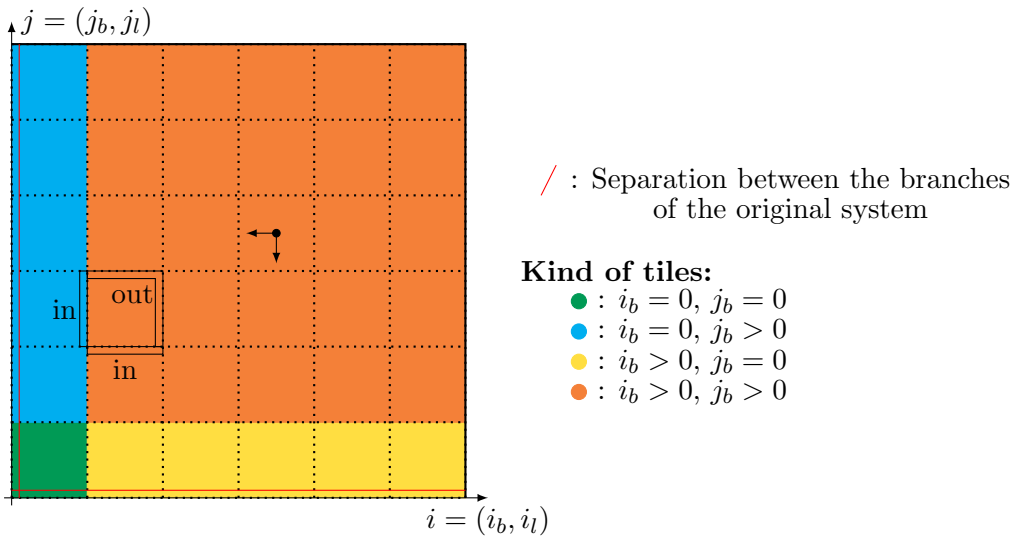


FIGURE 4.2: Example of Subsection 4.2.1: Monoparametric tiling of a Smith-Waterman computation

Kind of tiles In the tiled code we want to generate, the computation of each tile is enclosed inside a subsystem. If two tiles have a different computation, we have to use two different subsystems. If two tiles have the same computation (but are called using different input values), then we can reuse the same subsystem for both tiles. This means that we have to classify the tiles according to their computation, in order to generate one subsystem for each different computation. This classification is called *kind of tiles*.

In this example, we have 4 different kinds of tiles, as shown in Figure 4.2, because of the special computation performed in the first row and column ($i = 0$ and $j = 0$). The first kind of tile (occurs at $i_b = j_b = 0$, in green in the figure) has a special computation on its first row and column ($i_l = 0$ and $j_l = 0$). The second kind of tile (occurs for all the $i_b = 0$ and $j_b > 0$, in yellow in the figure) has a special computation only on its first row ($i_l = 0$). The third kind of tile (occurs for all the $i_b > 0$ and $j_b = 0$, in blue in the figure) has a special computation only on its first column ($j_l = 0$). The fourth kind of tile (occurs for all the $i_b > 0$ and $j_b > 0$, in orange in the figure) has no special computation. Thus, we will have 4 different subsystems generated.

Building the subsystems Let us consider a kind of tile. In order to build the corresponding subsystem, we need to know its computation, and its inputs/outputs. Its computation can be

determined by applying the monoparametric partitioning transformation to the program, then classify the equations according to the constraints on the block indices. For example, for the kind of tile • $(i_b > 0, j_b > 0)$, the corresponding equations in the partitioned system are the following:

$$
\begin{aligned}
&(\forall 0 < i_b, j_b)\ (\forall 0 = i_l = j_l)\ \hat{A}[i_b, j_b, i_l, j_l] = min(\hat{A}[i_b - 1, j_b, b - 1, j_l] \\
&\qquad +\hat{w}[i_b - 1, j_b, b - 1, j_l], \hat{A}[i_b, j_b - 1, i_l, b - 1] + \hat{w}[i_b, j_b - 1, i_l, b - 1]); \\
&(\forall 0 < i_b, j_b)\ (\forall 0 = i_l < j_l)\ \hat{A}[i_b, j_b, i_l, j_l] = min(\hat{A}[i_b - 1, j_b, b - 1, j_l] \\
&\qquad +\hat{w}[i_b - 1, j_b, b - 1, j_l], \hat{A}[i_b, j_b, i_l, j_l - 1] + \hat{w}[i_b, j_b, i_l, j_l - 1]); \\
&(\forall 0 < i_b, j_b)\ (\forall 0 = j_l < i_l)\ \hat{A}[i_b, j_b, i_l, j_l] = min(\hat{A}[i_b, j_b, i_l - 1, j_l] \\
&\qquad +\hat{w}[i_b, j_b, i_l - 1, j_l], \hat{A}[i_b, j_b - 1, i_l, b - 1] + \hat{w}[i_b, j_b - 1, i_l, b - 1]); \\
&(\forall 0 < i_b, j_b)\ (\forall 0 < i_l, j_l)\ \hat{A}[i_b, j_b, i_l, j_l] = min(\hat{A}[i_b, j_b, i_l - 1, j_l] \\
&\qquad +\hat{w}[i_b, j_b, i_l - 1, j_l], \hat{A}[i_b, j_b, i_l, j_l - 1] + \hat{w}[i_b, j_b, i_l, j_l - 1]);
\end{aligned}
$$

Once we have identify the computation of each kind of tile, we need to find what are the inputs and outputs of each kind of tile. The inputs can be determined by examining the dependences of the computation of the subsystem. Because the block indices are explicit, we can immediately identify when a value is produced outside of the current tile. In our case, we obtain the following

subsystem:

Parameters: $N_l(=0), b$

Input variables:

$Ain1$, defined over $\{i_l, j_l | i_l = b-1\}$ and corresponding to the block $A[i_b - 1, j_b]$

$Ain2$, defined over $\{i_l, j_l | j_l = b-1\}$ and corresponding to the block $A[i_b, j_b - 1]$

$win0$, defined over $\{i_l, j_l | 0 \leq i_l, j_l < b\}$ and corresponding to the block $w[i_b, j_b]$

$win1$, defined over $\{i_l, j_l | i_l = b-1\}$ and corresponding to the block $w[i_b - 1, j_b]$

$win2$, defined over $\{i_l, j_l | j_l = b-1\}$ and corresponding to the block $w[i_b, j_b - 1]$

Local variable:

$Aloc$, defined over $\{i_l, j_l | 0 \leq i_l, j_l < b\}$ and corresponding to the block $A[i_b, j_b]$

Output variable: Not built yet...

Equations:

$$(\forall i_l = j_l = 0)\ Aloc[i_l, j_l] = min(Ain1[b-1, j_l] + win1[b-1, j_l], \\ Ain2[i_l, b-1] + win2[i_l, b-1]);$$

$$(\forall i_l = 0, j_l > 0)\ Aloc[i_l, j_l] = min(Ain1[b-1, j_l] + win1[b-1, j_l], \\ Aloc[i_l, j_l - 1] + win0[i_l, j_l - 1]);$$

$$(\forall i_l > 0, j_l = 0)\ Aloc[i_l, j_l] = min(Aloc[i_l - 1, j_l] + win0[i_l - 1, j_l], \\ Ain2[i_l, b-1] + win2[i_l, b-1]);$$

$$(\forall i_l, j_l > 0)\ Aloc[i_l, j_l] = min(Aloc[i_l - 1, j_l] + win0[i_l - 1, j_l], \\ Aloc[i_l, j_l - 1] + win0[i_l, j_l - 1]);$$

Finally, we just need to determine the outputs of this subsystem. After building all the subsystems without their outputs, we know which values of a variable are needed outside of its tile. In our case, we know that the last row and last column of a block of A can be asked by its neighbor tiles ($(i_b, j_b + 1)$ and $(i_b + 1, j_b)$, if these tiles exist). We create two outputs variables corresponding to the data which might be asked by these tiles, and the corresponding copy equations from $Aloc$.

Thus, we just need to add the following output variables and equations to the subsystem we have previously presented:

Output variables:

$Aout1$, defined over $\{i_l, j_l | i_l = b - 1\}$ which might be asked by the block $A[i_b + 1, j_b]$

$Aout2$, defined over $\{i_l, j_l | j_l = b - 1\}$ which might be asked by the block $A[i_b, j_b + 1]$

Equations:

$Aout1[i_l, j_l] = Aloc[i_l, j_l];$

$Aout2[i_l, j_l] = Aloc[i_l, j_l];$

Building the main system The main system of the tiled program contains the use equations, and do not contains any actual information. For our example, the main system is described in Figure 4.3. Because we have 4 kinds of tile, we have 4 use equations, calling the 4 different subsystems. We have one local variable per output of the use equations (the $AoutXY$, where $X = 1 \dots 4$ and $Y = 1, 2$)

About the inputs of the use equations, because the values passed might come from a tile which belongs to a different kind of tile, we need to have a local variable to gather the values of all the outputs of the same type ($Aout1$ and $Aout2$ for the last row and last column respectively). These variables are used inside the input expressions of the use equations.

In the rest of this section, we will describe formally the concepts and algorithms we used on this example to obtain the tiled program.

4.2.2 Preprocessing - Preparing for the outlining

First of all, we need to separate physically each computation, according to the block to which they belong. In the previous chapter, we applied the monoparametric partitioning transformation syntactically, before normalizing the result. After the normalization step, each variable

Parameters: $N_b, N_l(=0), b$
Input variable:
w, defined over $\{i_b, j_b, i_l, j_l | 0 \leq i_b, j_b < N_b, 0 \leq i_l, j_l < b\}$
Output variable:Out, scalar
Local variables:
$Aout1$, defined over $\{i_b, j_b, i_l, j_l | 0 \leq i_b, j_b < N_b, i_l = b-1, 0 \leq j_l < b\}$
$Aout2$, defined over $\{i_b, j_b, i_l, j_l | 0 \leq i_b, j_b < N_b, 0 \leq i_l < b, j_l = b-1\}$

$Aout11$, defined over $\{i_b, j_b, i_l, j_l | i_b = j_b = 0, i_l = b-1, 0 \leq j_l < b\}$
$Aout21$, defined over $\{i_b, j_b, i_l, j_l | 0 = i_b < j_b, i_l = b-1, 0 \leq j_l < b\}$
$Aout31$, defined over $\{i_b, j_b, i_l, j_l | 0 = j_b < i_b, i_l = b-1, 0 \leq j_l < b\}$
$Aout41$, defined over $\{i_b, j_b, i_l, j_l | 0 < i_b, j_b, i_l = b-1, 0 \leq j_l < b\}$
$Aout12$, defined over $\{i_b, j_b, i_l, j_l | i_b = j_b = 0, 0 \leq i_l < b, j_l = b-1\}$
$Aout22$, defined over $\{i_b, j_b, i_l, j_l | 0 = i_b < j_b, 0 \leq i_l < b, j_l = b-1\}$
$Aout32$, defined over $\{i_b, j_b, i_l, j_l | 0 = j_b < i_b, 0 \leq i_l < b, j_l = b-1\}$
$Aout42$, defined over $\{i_b, j_b, i_l, j_l | 0 < i_b, j_b, 0 \leq i_l < b, j_l = b-1\}$
Equations:
$(\forall i_b = j_b = 0)\ (\forall 0 \leq j_l \leq b-1 = i_l)\ Aout1 = Aout11[i_b, j_b, i_l, j_l];$
$(\forall 0 = i_b < j_b)\ (\forall 0 \leq j_l \leq b-1 = i_l)\ Aout1 = Aout21[i_b, j_b, i_l, j_l];$
$(\forall 0 = j_b < i_b)\ (\forall 0 \leq j_l \leq b-1 = i_l)\ Aout1 = Aout31[i_b, j_b, i_l, j_l];$
$(\forall 0 < i_b, j_b)\ (\forall 0 \leq j_l \leq b-1 = i_l)\ Aout1 = Aout41[i_b, j_b, i_l, j_l];$
$(\forall i_b = j_b = 0)\ (\forall 0 \leq i_l \leq b-1 = j_l)\ Aout2 = Aout12[i_b, j_b, i_l, j_l];$
$(\forall 0 = i_b < j_b)\ (\forall 0 \leq i_l \leq b-1 = j_l)\ Aout2 = Aout22[i_b, j_b, i_l, j_l];$
$(\forall 0 = j_b < i_b)\ (\forall 0 \leq i_l \leq b-1 = j_l)\ Aout2 = Aout32[i_b, j_b, i_l, j_l];$
$(\forall 0 < i_b, j_b)\ (\forall 0 \leq i_l \leq b-1 = j_l)\ Aout2 = Aout42[i_b, j_b, i_l, j_l];$

$use\ \{i_b, j_b | i_b = j_b = 0\}\ subsyst1[Nl, b]\ (w)\ return(Aout11, Aout12);$
$use\ \{i_b, j_b | 0 = i_b < j_b\}\ subsyst2[Nl, b]\ (Aout2[i_b, j_b - 1, \bullet, \bullet], w, w[i_b, j_b - 1, \bullet, \bullet])$
$return(Aout21, Aout22);$
$use\ \{i_b, j_b | 0 = j_b < i_b\}\ subsyst3[Nl, b]\ (Aout1[i_b - 1, j_b, \bullet, \bullet], w, w[i_b - 1, j_b, \bullet, \bullet]$
$return(Aout31, Aout32);$
$use\ \{i_b, j_b | 0 < i_b, j_b\}\ subsyst4[Nl, b]\ (Aout1[i_b - 1, j_b, \bullet, \bullet], Aout2[i_b, j_b - 1, \bullet, \bullet], w$
$w[i_b - 1, j_b, \bullet, \bullet], w[i_b, j_b - 1, \bullet, \bullet])\ return(Aout41, Aout42);$
$Out = A[N_b - 1, N_b - 1, b - 1, b - 1];$

FIGURE 4.3: Main system after applying the monoparametric tiling transformation to the example of Subsection 4.2.1. $A[f(i_b), g(j_b), \bullet, \bullet]$ is a variable whose value at (i_l, j_l) is $A[f(i_b), g(j_b), i_l, j_l]$.

Var of a system has a list of equations of the following form:

$$\begin{aligned}(\forall \vec{i} \in \mathcal{D}_1) \quad & Var[\vec{i}] = \mathrm{SExpr}_1[\vec{i}];\\ (\forall \vec{i} \in \mathcal{D}_2) \quad & Var[\vec{i}] = \mathrm{SExpr}_2[\vec{i}];\\ & \dots\end{aligned}$$

where the SExpr_k are expressions and the $\mathcal{D}_k$ are disjoint.

As shown in Theorems 3.2 and 3.3, there is a clear separation between the constraints on the block indices and the local indices. Thus, it is possible to keep this separation inside a program, to obtain the following form of (slightly modified) equation:

$$\begin{aligned}(\forall \vec{i_b} \in \mathcal{D}_{bl,1})(\forall \vec{i_l} \in \mathcal{D}_{loc,1,1}) \quad & Var[\vec{i_b}, \vec{i_l}] = \mathrm{SExpr}_{1,1}[\vec{i_b}, \vec{i_l}];\\ (\forall \vec{i_b} \in \mathcal{D}_{bl,1})(\forall \vec{i_l} \in \mathcal{D}_{loc,1,2}) \quad & Var[\vec{i_b}, \vec{i_l}] = \mathrm{SExpr}_{1,2}[\vec{i_b}, \vec{i_l}];\\ \dots \qquad & \dots\\ (\forall \vec{i_b} \in \mathcal{D}_{bl,2})(\forall \vec{i_l} \in \mathcal{D}_{loc,2,1}) \quad & Var[\vec{i_b}, \vec{i_l}] = \mathrm{SExpr}_{2,1}[\vec{i_b}, \vec{i_l}];\\ (\forall \vec{i_b} \in \mathcal{D}_{bl,2})(\forall \vec{i_l} \in \mathcal{D}_{loc,2,2}) \quad & Var[\vec{i_b}, \vec{i_l}] = \mathrm{SExpr}_{2,2}[\vec{i_b}, \vec{i_l}];\\ \dots \qquad & \dots\end{aligned}$$

where the polyhedra $\mathcal{D}_{bl,k}$ only contain constraints on the block indices, and the polyhedra $\mathcal{D}_{loc,k,l}$ only contain constraints on the local indices.

For a given $\mathcal{D}_{bl,k}$, we can see the list of equations whose block indices belong to $\mathcal{D}_{bl,k}$ as the computation performed in a tile. This computation is the same for all the tiles whose block indices satisfy the constraints of $\mathcal{D}_{bl,k}$. In order to classify the tiles according to their computation, we introduce the notion of *kind of tile*.

Definition 4.1. A *kind of tile* is a collection of tiles which share the same computation, i.e., whose computations are Herbrand-equivalent

Theorem 4.2. *For a given partitioned program, there is a finite non-parametric number of kind of tile.*

Proof. Note that each polyhedron and affine function we partition leads to either a union of a finite non-parametric polyhedron, or a piecewise quasi-affine function with a finite non-parametric number of branches. Therefore, we will also have a finite non-parametric number of $\mathcal{D}_{bl,k}$ after normalization. Thus, we will have a finite non-parametric number of kind of tile. □

This property is crucial for the construction of the monoparametric tiled code: indeed, we cannot have a parametric number of subsystems in our tiled code. It is also especially useful for our template recognition framework. Indeed, instead of having to consider the computation of a parametric number of tiles, we will be able to just consider the computation of a finite non-parametric number of kinds of tile.

For each kind of tile, the local computation of this tile is described by the corresponding equations. Intuitively, the outlining transformation consists of putting this computation inside a separated subsystem (with one subsystem per kind of tile) and managing the input/outputs of this tile.

Example 4.3. *Let us consider a computation with a Smith-Waterman pattern of dependences:*

$$\begin{aligned}
Out = &\ A[N,N] \\
(\forall i = j = 0)\ A[i,j] = &\ w[0,0]; \\
(\forall i = 0 < j)\ A[i,j] = &\ A[i,j-1] + w[i,j-1]; \\
(\forall j = 0 < i)\ A[i,j] = &\ A[i-1,j] + w[i-1,j]; \\
(\forall 0 < i,j < N)\ A[i,j] = &\ min(A[i-1,j] + w[i-1,j], A[i,j-1] + w[i,j-1]);
\end{aligned}$$

where N is a parameter of the program. We consider square tiles $b \times b$.

If we assume that the program parameter N is divisible by the tile size b, the first row and first column of A have a different computation than the rest of the domain of A. Therefore, the first row and column of tile will have a different computation (respectively on their first row and on their first column) compared to the rest of the tiles. The tile $i_b = j_b = 0$ is even more special and has a different computation for both its first row and column. Therefore, we have 4 kinds of tiles: $(i_b = j_b = 0)$, $(0 = i_b < j_b)$, $(0 = j_b < i_b)$ and $(0 < i_b, j_b)$.

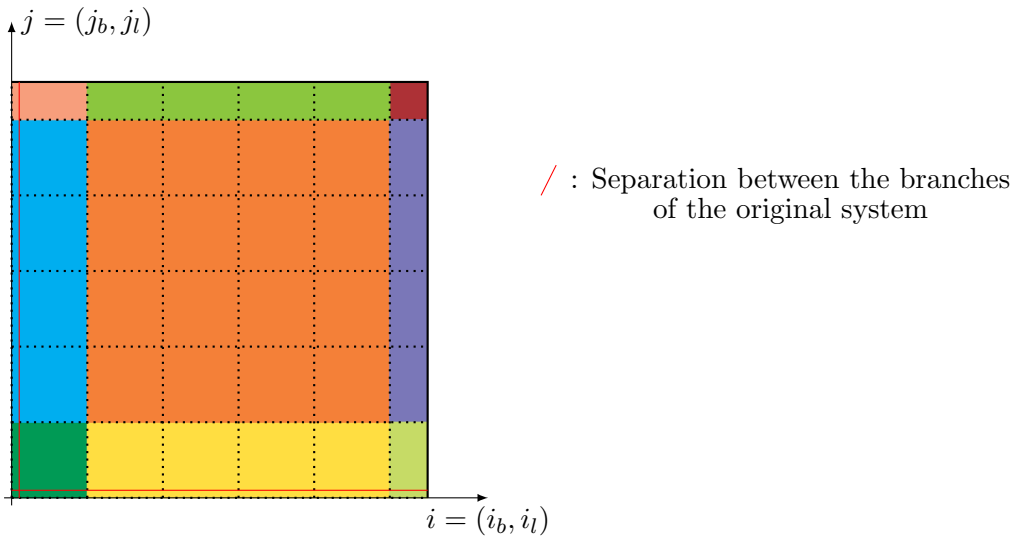


FIGURE 4.4: Example 4.3: different kind of tiles when the tile size parameter b does not divide the program parameter N.

If we assume that N is not divisible by the block size, we have boundary tiles which are not full tiles. Thus, in addition to the 4 kind of tiles discovered previously for the full tiles, we have 5 additional kind of tiles (for a total of 9 kinds of tiles), as shown in Figure 4.4:

- *A small square tile ●, corresponding to $i_b = j_b = N_b$.*
- *The left rectangle ●, corresponding to $i_b = N_b$ and $0 < j_b$.*
- *The left rectangle which is the first of its column ●, corresponding to $i_b = N_b$ and $j_b = 0$.*
- *The bottom rectangle ●, corresponding to $j_b = N_b$ and $0 < i_b$.*
- *The bottom rectangle which is the first of its row ●, corresponding to $j_b = N_b$ and $i_b = 0$.*

4.2.3 Tile group

We have seen in Chapter 3 that the monoparametric partitioning transformation is just a reindexing transformation, which replaces all the indices of the original program into their corresponding block and local indices. This transformation does not ask for the atomicity of its

tiles. Now that we consider the monoparametric tiling transformation, the tiles are atomic. In particular, we have to care about the legality condition of tiling, i.e., we need to ensure that there is no cyclic dependences between tiles.

In order to make a tiling legal, a possibility is to adjust the domain and dependences of the variables by using a Change of Basis transformation beforehand, as explained in Section 2.3. Another possibility is to tile several variables together, such that the same set of tiles compute all of them, instead of having a set of tile per variable. This information allows us to manage cyclic dependences between variables, and avoid that such cyclic dependences occurs between tiles.

In order to specify which variables share their tiles, we introduce the concept of *tile group*:

Definition 4.3. A *tile group* is a set of variables which will be tiled together, and will share the same tiling spaces.

A variable can belong to at most one tile group. All the variables of the same tile group share the same kinds of tile, thus will share the same subsystems.

In some situation, we might want to have a tile group which contains variables whose domains do not have the same number of dimensions. In order to be able to share tiles, we need to arrange the domain of these variables so that their domains have the same number of dimensions. This process is called *alignment*, and can be performed through some Change of Basis transformations. After these transformations, all the variables of the same tile group must have the same number of dimensions.

Example 4.4. *Let us consider the following modified version of Jacobi1D. In this version, we use an additional local variable temp*2 *and perform some extra copy between the two local*

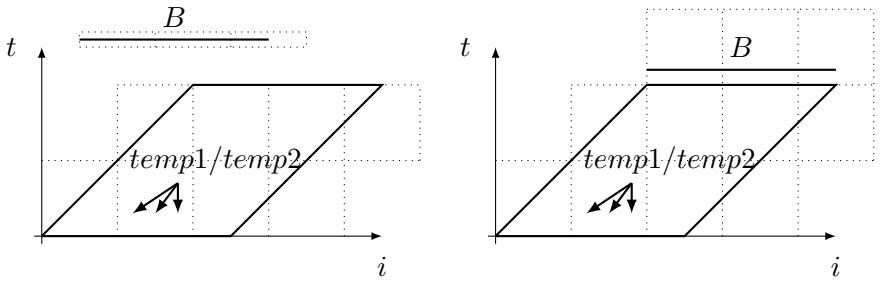
variables at every time step t:

$$
\begin{array}{rl}
\textit{Program ``Jacobi1Dcopy'':} & \textit{input: } A \textit{ (defined on } \{i|0 \leq i < N\} \textit{)} \\
& \textit{output: } B \textit{ (defined on } \{i|0 \leq i < N\} \textit{)} \\
& \textit{local: } temp1 \textit{ (defined on } \{i,t|0 \leq i < N \wedge 0 < t < T\} \textit{)} \\
& \quad temp2 \textit{ (defined on } \{i,t|0 \leq i < N \wedge 0 \leq t < T\} \textit{)} \\
& \textit{parameters: } T, N
\end{array}
$$

$$
\begin{array}{rl}
(\forall 0 \leq i < N) & B[i] = temp1[i, T-1] \\
(\forall t = 0 \wedge 0 \leq i < N) & temp2[i,t] = A[i] \\
(\forall 0 < t < T \wedge i = 0) & temp2[i,t] = temp1[i, t-1] \\
(\forall 0 < t < T \wedge i = N-1) & temp2[i,t] = temp1[i, t-1] \\
(\forall 0 < t < T \wedge 0 < i < N-1) & temp2[i,t] = (temp1[i-1, t-1] + \\
& \qquad temp1[i, t-1] + temp1[i+1, t-1])/3; \\
(\forall 0 \leq t < T \wedge 0 \leq i < N) & temp1[i,t] = temp2[i,t];
\end{array}
$$

If we try to tile $temp1$ and $temp2$ separately, the tiling cannot be legal. Indeed, we will obtain a cyclic dependence between the tiles of $temp1$ and $temp2$, as soon as we try to tile across the time dimension t. Thus, we need to have a single tile group for both variables and have a single set of tile which computes the values of both variables. Moreover, in order to make the rectangular tiling legal, we need to apply the loop skewing transformation [85] beforehand. Therefore, a possible way to preprocess the program in order to make the rectangular tiling legal is the following:

- *First group of tiles:*
  - *$temp1$, preprocess with a Cob using the affine function: $(i, t \mapsto i + t, t)$*
  - *$temp2$, preprocess with a Cob using the affine function: $(i, t \mapsto i + t, t)$*
- *Second group of tiles: B with no preprocessing*



First alignment info
B and $temp1/2$ live in **different** tiles

Second alignment info
B and $temp1/2$ live in the **same** tiles

FIGURE 4.5: Two valid preprocessings and tile groups for the modified Jacobi1D computation introduced in Example 4.4.

Another possibility is to put all three variables into the same group of tiles. In that case, we need to adapt the domain of B (which is 1-dimensional) to make it compatible with the other 2-dimensional domains. It is possible by applying a Cob using, for example, the following affine function: $(i \mapsto i + T - 1, T - 1)$. *Figure 4.5 shows a graphical representation of both examples.*

In the rest of this document, we will assume that the legality issues were already taken care of. This means that we assume that the preprocessing has already been performed, and that the tile groups are specified, such taht the rectangular tiling is legal. We can double-check for the legality of the tiling specified by these information through the method explained in Section 3.4. In the rest of this section, we will focus on how to perform the monoparametric tiling transformation, using these informations.

4.2.4 Monoparametric Tiling with outlining without reduction

In the context of this document, we assume that we tile all variables across all dimensions. The transformation can be potentially extended to tile only a subset of the dimensions, but we still need to partition all dimensions, thus we still need to introduce the block and local indices for all dimensions.

The main intuition of this transformation is to create one subsystem per kind of tile. Then, the main program will call the corresponding subsystems using use equations and manage their

Main system:

- Variables: monoparametric tiled version of the original system variable
- Local variables for the output of use equations: $VarOut\_k$
- Copy equations: $(\forall \vec{i} \in \mathcal{D}_k^{Var})\ Var = VarOut\_k$
- UseEquations: use $\mathcal{D}_k^{Var}$ subsyst_k $(\dots)$ returns $(VarOut\_k)$

Subsystem for the kth kind of tile:

- *Inputs:* data computed by other tiles, needed by this tile
- *Locals:* data computed by this tile
- *Outputs:* copy of local variable, needed by other tiles
- Equations corresponding to the computation of this kind of tile

FIGURE 4.6: Form of the main system and the subsystem after applying the CART with outlining transformation. In the main system $\mathcal{D}_k^{Var}$ is the domain of the kth kind of tiles.

inputs and outputs. The structures of the subsystems and the main program are summarized in Figure 4.6.

Assuming that the preprocessing described in Subsection 4.2.2 was already applied, the algorithm builds the main system and the subsystems in the following order:

1. Computing the kind of tiles of the program
2. Building the subsystems
   (a) Computing the domains of the local variables of the subsystems
   (b) Obtaining the equations of the subsystems and tracking down their inputs
   (c) Adding the outputs of the subsystems
3. Building the main system

Step 1 - Computing the kind of tiles After applying the monoparametric partitioning transformation while preparing for outlining, the obtained program has a specific form, in which

the constraints on the blocked and local indices are separated:

$$\begin{array}{ll}
(\forall \vec{i_b} \in \mathcal{D}_{bl,1})(\forall \vec{i_l} \in \mathcal{D}_{loc,1,1}) & Var[\vec{i_b}, \vec{i_l}] = \mathrm{SExpr}_{1,1}[\vec{i_b}, \vec{i_l}]; \\
(\forall \vec{i_b} \in \mathcal{D}_{bl,1})(\forall \vec{i_l} \in \mathcal{D}_{loc,1,2}) & Var[\vec{i_b}, \vec{i_l}] = \mathrm{SExpr}_{1,2}[\vec{i_b}, \vec{i_l}]; \\
\dots & \dots \\
(\forall \vec{i_b} \in \mathcal{D}_{bl,2})(\forall \vec{i_l} \in \mathcal{D}_{loc,2,1}) & Var[\vec{i_b}, \vec{i_l}] = \mathrm{SExpr}_{2,1}[\vec{i_b}, \vec{i_l}]; \\
(\forall \vec{i_b} \in \mathcal{D}_{bl,2})(\forall \vec{i_l} \in \mathcal{D}_{loc,2,2}) & Var[\vec{i_b}, \vec{i_l}] = \mathrm{SExpr}_{2,2}[\vec{i_b}, \vec{i_l}]; \\
\dots & \dots
\end{array}$$

In this step, we want to distinguish the different tiles of a tile group according to their computation, i.e., according to which equations contributes to this tile. In order to do this, for each variable, we retrieve the constraints $\mathcal{D}_{bl,k}$ on the blocked indices of the domain of their equations. These domains form a partition of the tiles in which the variable Var contributes, and there is, by construction, only a finite non-parametric number of them.

Then, we consider each tile group separately. If we have a single variable inside the considered tile group, we have as many kinds of tile than domains on the block indices $\mathcal{D}_{bl,k}$. Moreover, the corresponding equations of the k-th kind of tile are the set of equations whose constraints on the blocked indices are $\mathcal{D}_{bl,k}$.

If we have multiple variables in the considered tile group, we consider each family of block constraints $(\mathcal{D}_{bl,k}^{Var})_k$ coming from each variable Var of the tile group. The list of non-empty intersections of these families corresponds to the different kind of tiles. The corresponding equation of each one of these kind of tiles are the ones which contributes to the intersection.

Step 2 - Building the subsystems For each kind of tile, we have to build the corresponding subsystem which perform its computation. The equations of such subsystem can be obtained by removing the blocked dimensions of every variable and dependence functions. This means that if we have the following equation:

$$(\forall \vec{i_b} \in \mathcal{D}_{bl})(\forall \vec{i_l} \in \mathcal{D}_{loc}) \quad Var[\vec{i_b}, \vec{i_l}] = f(\mathrm{Var}_1[u_{b,1}(\vec{i_b}), u_{l,1}(\vec{i_l})], \dots, \mathrm{Var}_k[u_{b,k}(\vec{i_b}), u_{l,k}(\vec{i_l})]);$$

we remove the blocked dimensions $\vec{i_b}$ to obtain:

$$(\forall \vec{i_l} \in \mathcal{D}_{loc}) \quad Var'[\vec{i_b}, \vec{i_l}] = f(\text{Var'}_1[\vec{i_l}], \ldots, \text{Var'}_k[\vec{i_l}]);$$

Note that this is possible only because the block and local indices are cleanly separated in a partitioned affine function, as shown by Theorem 3.3.

In the previous equation, Var' is a local variable of the subsystem, corresponding to the block of Var computed by the current tile. $\text{Var'}_1, \ldots, \text{Var'}_k$ can be either local variable (if the data accessed is computed in the same subsystem) or an input of the subsystem (if the data accessed is computed outside of the subsystem). Thus, while obtaining the equations of the subsystem, we examine these variables to determine the inputs of the subsystem. We create exactly one input variable of the subsystem per block accessed, whose domain corresponds to the data accessed from this block.

About the parameters of a subsystem, we need at least the local parameters $\vec{p_l}$ and the block size parameter b, which are still present in the equations of the subsystem. About the block parameters $\vec{p_b}$, because we have removed all the block indices of the equations, there is no longer any constraints involving the block indices or the block parameters in the subsystem. Thus, we can omit them in the parameters of the subsystem.

The inputs can be determined by examining the dependences of the computation of the subsystem. Because the block indices are explicit, we can immediately identify when a value is produced outside of the current tile. For example, if we have originally a dependence $Var[i_b - 1, j_b - 1, b - 1, b - 1]$, we can immediately deduce that we need a data from the block $(i_b - 1, j_b - 1)$ of the tile group of the variable Var. We create one input variable in the subsystem, per external block accessed in the computation of the subsystem.

About the outputs of a subsystem, a simple solution would be to transfer back all the data computed in a tile to the main system. However, this causes a lot of unnecessary communications between the subsystem and the main system, because most of these values will never be used. A better solution consists on determining which data from a tile is needed by other tiles. We

classify this data according to the tile accessing it and create one output variable for each external tile. For example, if the data of a tile is used by the tile $(i_b + 1, j_b)$ and (i_b, i_b), we create two outputs, the first one corresponding to the data of the tile accessed by the tile $(i_b + 1, j_b)$, the second one corresponding to the data of the tile accessed by the tile (i_b, i_b).

Given a tile, depending on the kind of the neighboring tiles, this set of data accessed might change. In the example of Subsection 4.2.1, each tiles admits 2 outputs (corresponding to the last row and the last column of the tile). However, at least one of the outputs of the tiles of the last column or the last row are not used. To simplify the problem, we do not consider the nature of the neighboring tiles and take the union of all the set of data which might be asked by other tiles. This is an overapproximation compared to the exact set of output needed.

Step 3 - Building the main system Finally, we need to form the main system. In particular, we need to gather the outputs of the subsystems to send them as input of others. The form of the main system is given in Figure 4.6.

We first create one use equation per subsystem generated, whose extension domain correspond to the kind of tile. We also create one new local variable per outputs of the use equation, in order to retrieve the results of the subsystem. We also create local variables to gather the values of all the outputs of the same type and the same variable. These variables are used inside the input expressions of the use equations.

Example 4.5. *Let us consider a Skewed Jacobi1D computation:*

$$
\begin{aligned}
&(\forall 0 < i < N)\ Out[i] = temp[T-1, i+T-1];\\
&(\forall t = 0, 0 < i < N)\ temp[t,i] = A[i];\\
&(\forall t = i > 0)\ temp[t,i] = temp[t-1, i-1];\\
&(\forall t > 0, i = N-1+t)\ temp[t,i] = temp[t-1, i-1];\\
&(\forall t > 0, t < i < N-1+t)\ temp[t,i] = (temp[t-1, i-2] +\\
&\qquad\qquad temp[t-1, i-1] + temp[t-1, i])/3;
\end{aligned}
$$

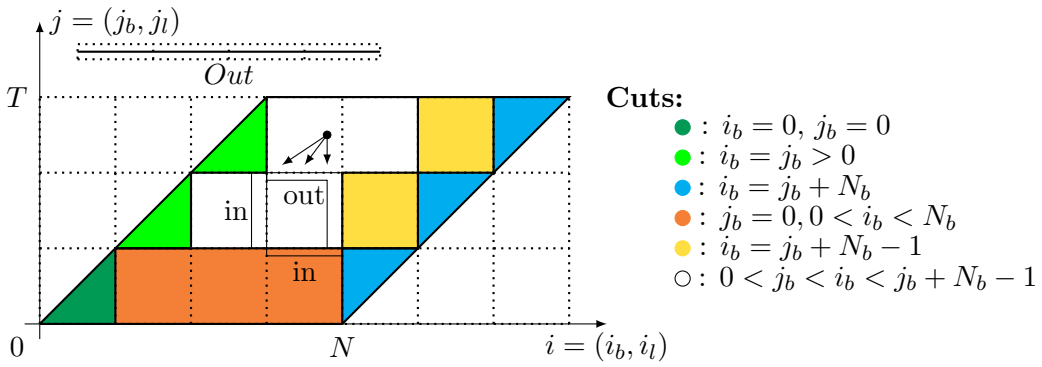


FIGURE 4.7: Example 4.5: Kinds of tile for a Jacobi1D skewed program

We assume that we want to apply the monoparametric partitioning transformation with an aspect ratio of 1×1*, and that the parameters* N *and* T *are divisible by the size parameter* b*. The resulting system contains about 20 different equations. We choose to put the variables temp and Out into two separate tile groups, and no preprocessing is needed to make the tiling legal.*

First of all, we compute the kinds of tile of the program. Because of the boundary conditions, we have 7 kinds of tiles: 6 for the temp variable (listed in the figure 4.7), and one for the Out variable. Once we have determined the equations and the inputs of each subsystem, we determine that the output of a tile are the 2 last columns on the right and the last row (needed for the right, above and the diagonal above-right tiles).

For example, the subsystem corresponding to the kind of tile ● $(i_b = j_b + N_b - 1)$ *is shown in Figure 4.8.*

4.3 Monoparametric tiling with reduction

In this section, we show how to adapt the transformation described in the previous section to manage reductions.

Parameters: $Nl(=0), b$
Input variables:
$tempin1$, defined over $\{i_l, j_l | b-2 \leq i_l < b\}$ $(\leftrightarrow temp[i_b - 1, j_b])$
$tempin2$, defined over $\{i_l, j_l | j_l = b-1\}$ $(\leftrightarrow A[i_b, j_b - 1])$
$tempin3$, defined over $\{i_l, j_l | b-2 \leq i_l < b, j_l = b-1\}$ $(\leftrightarrow A[i_b - 1, j_b - 1])$
Local variable:
$temploc$, defined over $\{i_l, j_l | 0 \leq i_l, j_l < b\}$ $(\leftrightarrow temp[i_b, j_b])$
Output variables:
$tempOut1$, defined over $\{i_l, j_l | j_l = b-1\}$ $(\leftrightarrow temp[i_b, j_b + 1])$
$tempOut2$, defined over $\{i_l, j_l | b-2 \leq i_l < b, j_l < b-1\}$ $(\leftrightarrow temp[i_b + 1, j_b])$
$tempOut3$, defined over $\{i_l, j_l | b-2 \leq i_l < b, j_l = b-1\}$ $(\leftrightarrow temp[i_b + 1, j_b + 1])$
Equations:

$$(\forall i_l = j_l = 0)\ temploc[i_l, j_l] = (tempin3[b-1, b-2] + tempin3[b-1, b-1] + tempin2[b-1, 0])/3;$$

$$(\forall i_l = 1, j_l = 0)\ temploc[i_l, j_l] = (tempin3[b-1, b-1] + tempin2[b-1, 0] + tempin2[b-1, 1])/3;$$

$$(\forall 1 < i_l < b-1, j_l = 0)\ temploc[i_l, j_l] = (tempin2[b-1, i_l - 2] + tempin2[b-1, i_l - 1] + tempin2[b-1, i_l])/3;$$

$$(\forall i_l = b-1, j_l = 0)\ temploc[i_l, j_l] = tempin2[b-1, b-2];$$

$$(\forall i_l = 0, j_l > 0)\ temploc[i_l, j_l] = (tempin1[j_l - 1, b-2] + tempin1[j_l - 1, b-1] + temploc[j_l - 1, i_l])/3;$$

$$(\forall i_l = 1, j_l > 0)\ temploc[i_l, j_l] = (tempin1[j_l - 1, b-1] + temploc[j_l - 1, i_l - 1] + temploc[j_l - 1, i_l])/3;$$

$$(\forall i_l > 1, j_l > 0)\ temploc[i_l, j_l] = (temploc[j_l - 1, i_l - 2] + temploc[j_l - 1, i_l - 1] + temploc[j_l - 1, i_l])/3;$$

$$(\forall 0 \leq i_l < b, j_l = b-1)\ tempOut1[i_l, j_l] = temploc[i_l, j_l];$$

$$(\forall b-2 \leq i_l < b, j_l < b-1)\ tempOut2[i_l, j_l] = temploc[i_l, j_l];$$

$$(\forall b-2 \leq i_l < b, j_l = b-1)\ tempOut3[i_l, j_l] = temploc[i_l, j_l];$$

FIGURE 4.8: Subsystem of the kind of tile ($i_b = j_b + N_b - 1$) in Example 4.5.

4.3.1 Monoparametric partitioning with reductions

A reduction introduces extra dimensions which are projected by the projection function. These dimensions are partitioned by the monoparametric partitioning transformation and also need to be considered in the tiling. We recall that all reductions of a program are preprocessed to make their projection function canonic (i.e., of the form $(\vec{i_1}, \vec{i_2} \mapsto \vec{i_1})$, see Subsection 3.2.2).

Motivating Example We first consider an example to provide an intuition of how reductions can be managed during the monoparametric tiling transformation. Let us consider a matrix

multiplication program with reduction:

$$(\forall 0 \leq i, j < N)\ C[i, j] = \sum_{k=0}^{N-1} A[i, k] * B[k, j]$$

If we simply apply the partitioning transformation and assuming that N is divisible by the block size b, we obtain the following program:

$$(\forall 0 \leq i_b, j_b < N_b)(\forall 0 \leq i_l, j_l < b)\ C[i_b, j_b, i_l, j_l] = \sum_{k_b, k_l} A[i_b, k_b, i_l, k_l] * B[k_b, j_b, k_l, j_l];$$

Note that the reduction sums over several tiles (the $A[i_b, \bullet]$ and $B[\bullet, j_b]$). In order to differentiate the computation according to the tiles accessed, we can split the reduction into the following two reductions:

$$C[i_b, j_b, i_l, j_l] = \sum_{k_b} TempRed[i_b, j_b, k_b, i_l, j_l];$$
$$TempRed[i_b, j_b, k_b, i_l, j_l] = \sum_{k_l} A[i_b, k_b, i_l, k_l] * B[k_b, j_b, k_l, j_l];$$

in which $TempRed[i_b, j_b, k_b, i_l, j_l]$ corresponds to the intermediate result of the accumulation over the k_bth tile.

As shown in Figure 4.9, all values of $TempRed$ are summed together (in the equation defining C) in order to obtain the value of the full reduction. Note that we are using the associativity property of the reduction operator to group the summation of the terms inside a tile, thus this transformation uses the semantic properties of a reduction. The equation of $TempRed$ only uses one block of A and one block of B instead of the whole row/column.

General Case In general, let us consider a reduction of the form:

$$Var[\vec{i}_1] = \sum_{\pi(\vec{i}_1, \vec{i}_2) = \vec{i}_1} Expr[\vec{i}_1, \vec{i}_2]$$

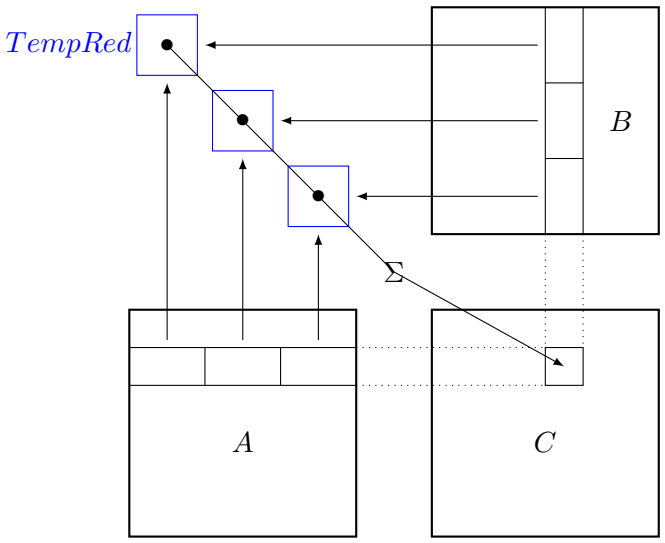


FIGURE 4.9: Representation of the partitioned matrix multiplication program. In order to compute a tile of C, we have a summation over the tiles of A from the same row, and the tiles of B from the same column. We introduce a new temporary variable called $TempRed$ which corresponds to the partial results of this summation. Then we sums all the values of $TempRed$ to obtain the value of the tile of C.

, where $\pi : (\vec{i_1}, \vec{i_2} \mapsto \vec{i_1})$. After partitioning, this reduction becomes:

$$Var[\vec{i}_{b,1}, \vec{i}_{l,1}] = \sum_{\vec{i}_{b,2}} \sum_{\vec{i}_{l,2}} Expr[\vec{i}_{b,1}, \vec{i}_{b,2}, \vec{i}_{l,1}, \vec{i}_{l,2}]$$

We are able to separate the two summations because of the associativity and commutativity property of the reduction operator.

We introduce a new variable $TempRed[\vec{i_{b,1}}, \vec{i_{b,2}}, \vec{i_{l,1}}]$ to represent the intermediate result of the summation on one block of the reduction (corresponding to the result of the second summation in the previous equation). The original reduction equation becomes:

$$Var[\vec{i}] = \sum_{\vec{i_{b,2}}} TempRed[\vec{i}_{b,1}, \vec{i}_{b,2}, \vec{i}_{l,1}]$$

The equation of $TempRed$ is:

$$TempRed[\vec{i}_{b,1}, \vec{i}_{b,2}, \vec{i}_{l,1}] = \sum_{\vec{i}_{l,2}} \hat{Expr}[\vec{i}_{b,1}, \vec{i}_{b,2}, \vec{i}_{l,1}, \vec{i}_{l,2}];$$

where $\hat{Expr}$ is the partitioned version of $Expr$. Both equations can be put under the form introduced in Subsection 4.2.2, in which the blocked and local constraints and dependences are separated.

Note that we use the associativity property of the reduction operator, when we separate the reduction over $(\vec{i}_{b,2}, \vec{i}_{l,2})$ into two reductions (one over $\vec{i}_{b,2}$, and one over $\vec{i}_{l,2}$)).

4.3.2 Tile groups and reduction

In the previous subsection, we showed that reductions can be supported by introducing a new variable $TempRed$ for each reduction. Because the tile groups were specified before the monoparametric partitioning transformation, it does not contain any informations about how to tile $TempRed$. In the rest of this subsection, we show how to infer automatically in which tile group we should include $TempRed$.

The main intuition is the following: because the tile space of $TempRed$ has more dimensions of the tile group it originates, we choose to create a new tile group for them. However, we might have some cyclic dependences between the tiles of $TempRed$ and a tile from the original tile group. We show how to identify these tiles and split them from the rest of the tiles of $TempRed$.

Let us consider an equation containing a reduction: $Var[\vec{i_b}, \vec{i_l}] = \sum_{k_b} TempRed[\vec{i_b}, \vec{k_b}, \vec{i_l}]$ where $TempRed$ is the variable introduced by the partitioning of the reduction. In which tile group should we add $TempRed$, such that the tiling is still legal (i.e., no cycle between tiles is introduced)?

Let us consider a tile $(\vec{i_b}, \vec{k_b})$ of $TempRed$, and let us study the dependences involving this tile. Figure 4.10 presents the possible dependences involving a tile $TempRed[\vec{i_b}, \vec{k_b}, \vec{i_l}]$.

By construction, a variable $TempRed$ is introduced every times we have a reduction, and occurs only on the right-hand side of the equation of Var. Thus, the only dependence whose destination is a tile $TempRed[\vec{i_b}, \vec{k_b}]$ comes from $Var[\vec{i_b}]$. The dependences coming from $TempRed$ are the ones from the reduction body. They can either go to another variable not in the same tile group of Var (called $VarExti$ in Figure 4.10). Because the tiling was already valid before introducing

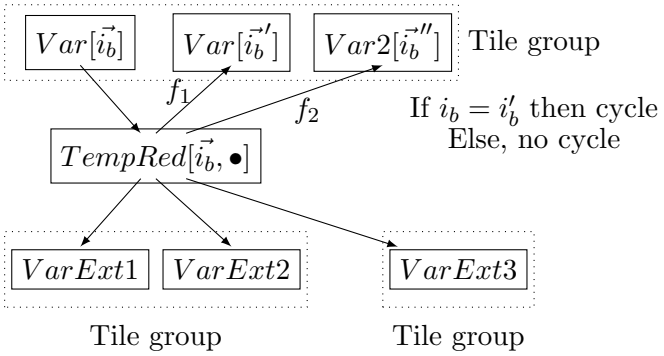


FIGURE 4.10: Dependences across tiles involving the tile $TempRed[\vec{i_b}, \vec{k_b}]$. A rectangle represents a tile and an arrow from a tile X to a tile Y means that the tile X *depends on* the tile Y. $\vec{i_b}$, $\vec{i_b}'$ and $\vec{i_b}''$ are instances of tiles for the tile group of Var. $VarExt1$, $VarExt2$ and $VarExt3$ are variables from other tile groups. $f1$ and $f2$ are block components of the dependence functions.

$TempRed$, there is no cycle possible involving the tile $Var[\vec{i_b}]$ and a tile from another tile group. Thus, the dependences leaving $TempRed$ to $VarExti$ cannot be part of a cycle between tiles.

Now, let us consider the dependences from the tiles $TempRed[\vec{i_b}, \vec{k_b}]$ to some tiles of variables of the same tile group that Var. Because of the legality of the tiling before introducing $TempRed$, there is no cycle between the tile computing $Var[\vec{i_b}]$ and any other tile (computing $Var[\vec{i_b}']$, where $\vec{i_b}' \neq \vec{i_b}$). Thus, if a dependence is going to $Var1[\vec{i_b}']$ where $\vec{i_b}' \neq \vec{i_b}$, then this dependence cannot be part of a cycle between tiles.

Last case: if we have a dependence from $TempRed[\vec{i_b}, \vec{k_b}]$ to a tile $\vec{i_b}$ of a variable of the same tile group of Var, then we have to compute $TempRed[\vec{i_b}, \vec{k_b}]$ in the same tile as $Var[\vec{i_b}]$ to avoid cycles across tiles. A naive solution is to compute all the $TempRed[\vec{i_b}, \bullet]$ in the same tile which computes $Var[\vec{i_b}]$. This will always give us a legal tiling. However, this implies that we do not tile the dimensions $\vec{k_b}$ (even if they are blocked). In many cases, this might be overkill since it would preclude a potential legal tiling. Thus, we must do this analysis in an *instance-wise* manner.

In the general case, we have to include at least the blocks $TempRed[\vec{i_b}, \vec{k_b}]$, which loop back to $Var[\vec{i_b}]$, in the same tile as $Var[\vec{i_b}]$. This means that the set of blocks to be included in the

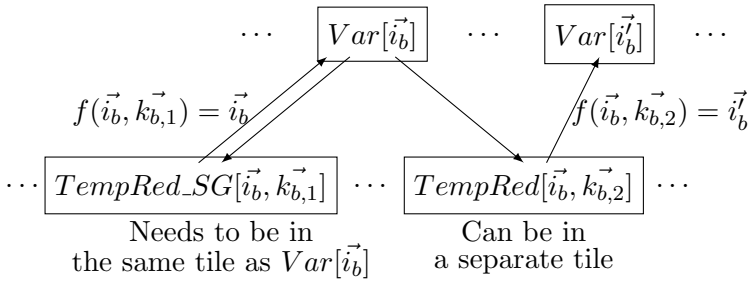


FIGURE 4.11: Split of *TempRed* according to the tiles which can be put in a separate tile group and those which must stay in the same tile group

same tile as $Var[\vec{i_b}]$ must contain at least the following set of tiles of $TempRed$:

$$\{\vec{i_b}, \vec{k_b} \mid f_1(\vec{i_b}, \vec{k_b}) = \vec{i_b} \ \vee \ f_2(\vec{i_b}, \vec{k_b}) = \vec{i_b} \ \vee \ \dots\}$$

where f_1, f_2, ... are the blocked components of the dependence functions from $TempRed[\vec{i_b}, \vec{k_b}]$ to a variable of the same tile group of Var.

We use this set to split the variable $TempRed$ into two variables, as shown in Figure 4.11: $TempRed\_SG$ (Same Group), corresponding to the tiles which must be put into the same tile group as Var because of the legality condition, and $TempRed$, corresponding to the tiles which can be tiled separately. A similar analysis was proposed by Wonnacott [18] in "almost-tilable" loops, but is limited for fixed-size tiling.

Example 4.6 (Forward substitution)**.** *Let us consider a program which solves the linear system* $L.\vec{x} = \vec{b}$ *where* L *is a lower-triangular matrix:*

$$(\forall 0 \leq i < N) \ x[i] = (b[i] - \sum_{k<i} L[i,k] \times x[k])/L[i,i];$$

We assume that x *and temp belong to the same tile group. The partitioning step introduces a new variable* $TempRed$ *and transform the program into the following equations, assuming that*

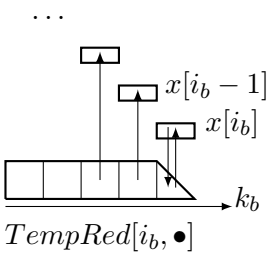


FIGURE 4.12: Dependences between the tiles of $TempRed$ and the tiles of $x/temp$

the parameters are divisible:

$$(\forall 0 \leq i_b < N_b)\ (\forall 0 \leq i_l < b)\ x[i_b, i_l] = (b[i_b, i_l] - \sum_{k_b \leq i_b} TempRed[i_b, k_b, i_l])/L[i_b, i_b, i_l, i_l];$$
$$(\forall 0 \leq i_b = k_b < N_b)\ (\forall 0 \leq i_l < b)\ TempRed[i_b, k_b, i_l] = \sum_{0 \leq k_l < i_l} L[i_b, k_b, i_l, k_l] \times x[k_b, k_l];$$
$$(\forall 0 \leq k_b < i_b < N_b)\ (\forall 0 \leq i_l < b)\ TempRed[i_b, k_b, i_l] = \sum_{0 \leq k_l < b} L[i_b, k_b, i_l, k_l] \times x[k_b, k_l];$$

Let us analyze the dependences involving $TempRed$ to decide in which tile group we should insert it. The only dependence which might introduce a cycle is the one corresponding to $x[k_b, k_l]$ in the equations of $TempRed$ (as shown in Figure 4.12). A cycle is introduced when $k_b = i_b$, thus we need to split this tile of TempRed from the other tiles. Therefore, we obtain the following program after normalization:

$$(\forall 0 \leq i_b < N_b)\ (\forall 0 \leq i_l < b)\ x[i_b, i_l] = (b[i_b, i_l] - \sum_{k_b < i_b} TempRed[i_b, k_b, i_l] - \sum_{k_b = i_b} TempRed\_SG[i_b, k_b, i_l])/L[i_b, i_b, i_l, i_l];$$
$$(\forall 0 \leq i_b = k_b < N_b)\ (\forall 0 \leq i_l < b)\ TempRed\_SG[i_b, k_b, i_l] = \sum_{0 \leq k_l < i_l} L[i_b, k_b, i_l, k_l] \times x[k_b, k_l];$$
$$(\forall 0 \leq k_b < i_b < N_b)\ (\forall 0 \leq i_l < b)\ TempRed[i_b, k_b, i_l] = \sum_{0 \leq k_l < b} L[i_b, k_b, i_l, k_l] \times x[k_b, k_l];$$

We have two tile groups: one containing the variables (x and $TempRed\_SG$), and another containing the variable $TempRed$. As a side note, we can notice that each tile of the first tile group correspond to a small forward substitution computation, and that each tile of the second tile group correspond to a small matrix multiplication.

Example 4.7 (Nussinov/Optimal String Parenthization [18]). *Let us consider the following program:*

$$(\forall 0 \leq i < j < N)\ N[i,j] = max_{i \leq k < j}(N[i,k] + N[k+1,j]);$$

After the partitioning and normalization step, we obtain the following program, a graphical representation being shown in Figure 4.13:

$$(\forall 0 \leq i_b \leq j_b < N_b)\ (\forall 0 \leq i_l < j_l < b)\ N[i_b, j_b, i_l, j_l] = max_{i_b \leq k_b \leq j_b} TempRed[i_b, j_b, k_b, i_l, j_l];$$

$$(\forall 0 \leq i_b = k_b = j_b)\ (\forall 0 \leq i_l < j_l < b)\ TempRed[i_b, j_b, k_b, i_l, j_l] = max_{i_l \leq k_l < j_l} N[i_b, k_b, i_l, k_l] + temp[k_b, j_b, k_l, j_l];$$

$$(\forall 0 \leq i_b = k_b < j_b)\ (\forall 0 \leq i_l < j_l < b)\ TempRed[i_b, j_b, k_b, i_l, j_l] = max_{i_l \leq k_l < b} N[i_b, k_b, i_l, k_l] + temp[k_b, j_b, k_l, j_l];$$

$$(\forall 0 \leq i_b < k_b = j_b)\ (\forall 0 \leq i_l < j_l < b)\ TempRed[i_b, j_b, k_b, i_l, j_l] = max_{0 \leq k_l < j_l} N[i_b, k_b, i_l, k_l] + temp[k_b, j_b, k_l, j_l];$$

$$(\forall 0 \leq i_b < k_b < j_b)\ (\forall 0 \leq i_l < j_l < b)\ TempRed[i_b, j_b, k_b, i_l, j_l] = max_{0 \leq k_l < b} N[i_b, k_b, i_l, k_l] + temp[k_b, j_b, k_l, j_l];$$

$$(\forall 0 \leq i_b \leq k_b \leq j_b)\ (\forall 0 \leq i_l < b, 0 \leq j_l < b-1)\ temp[k_b, j_b, k_l, j_l] = N[k_b, j_b, k_l + 1, j_l];$$

$$(\forall 0 \leq i_b \leq k_b < j_b)\ (\forall 0 \leq i_l < j_l = b-1)\ temp[k_b, j_b, k_l, j_l] = N[k_b + 1, j_b, 0, j_l];$$

Let us analyze the dependences involving $TempRed$/$temp$ to decide in which tile group we should insert it. By examining the equations of $TempRed$ and $temp$, we identify in total 3 dependences which might introduce a loop involving $N[i_b, j_b]$:

- $N[i_b, k_b]$ *in the equation of* $TempRed[i_b, j_b, k_b]$
- $N[k_b, j_b]$ *in the equation of* $temp[i_b, j_b, k_b]$
- $N[k_b + 1, j_b]$ *in the equation of* $temp[i_b, j_b, k_b]$

Let us examine each of these dependences separately:

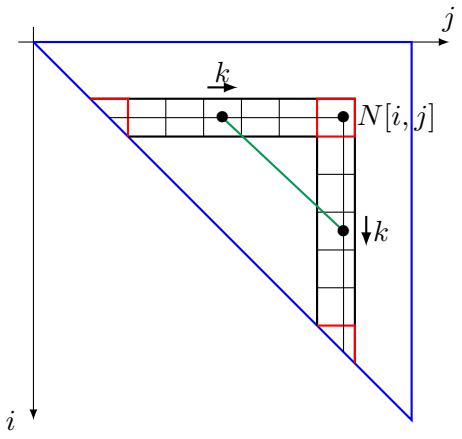


FIGURE 4.13: Partitioned Optimal String Parenthization. The two black lines correspond to the data needed to compute a single point $N[i, j]$. The green line links the two data needed to compute one instance of $TempRed[i, j, k]$ and the black arrows shows how these data accessed move according to k.
The two stripes of tiles correspond to the data needed to compute a single tile $N[i_b, j_b]$. The tiles in red (two diagonal and middle one) corresponds to the tiles of $TempRed[i_b, j_b, k_b]$ which have a cycle with the tile $N[i_b, j_b]$, thus which must be separated from the rest of the computation of $TempRed[i_b, j_b, k_b]$.

- *The first dependence introduces a loop between tiles iff* $(i_b, k_b) = (i_b, j_b)$*, i.e., when* $k_b = j_b$*. Physically, this situation corresponds to the case where the data needed by* $N[i, k]$ *belongs to the tile* $N[i_b, j_b]$ *which is currently computed.*

- *The second dependence introduces a loop between tiles iff* $(k_b, j_b) = (i_b, j_b)$*, i.e., when* $k_b = i_b$*. Physically, this situation corresponds to the case where the data needed by* $N[k + 1, j]$ *belongs to the tile* $N[i_b, j_b]$ *which is currently computed.*

- *The third dependence introduces a loop between tiles iff* $(k_b + 1, j_b) = (i_b, j_b)$*, i.e., when* $k_b + 1 = i_b$*. Because of the constraints of the equations in which this dependence happens* $(i_b \leq k_b)$*, this situation never occurs.*

Therefore, the only tiles of $TempRed/temp$ which we have to include into the same tile group as N are $k_b = i_b$ and $k_b = j_b$. After splitting, we obtain the following program:

$$(\forall 0 \leq i_b \leq j_b < N_b)\ (\forall 0 \leq i_l < j_l < b)\ N[i_b, j_b, i_l, j_l] = max(TempRed\_SG[i_b, j_b, i_b, i_l, j_l],$$
$$TempRed\_SG[i_b, j_b, j_b, i_l, j_l], max_{i_b < k_b < j_b} TempRed[i_b, j_b, k_b, i_l, j_l];$$
$$(\forall 0 \leq i_b < k_b < j_b)\ (\forall 0 \leq i_l < j_l < b)\ TempRed[i_b, j_b, k_b, i_l, j_l] = max_{0 \leq k_l < b}$$
$$N[i_b, k_b, i_l, k_l] + temp[k_b, j_b, k_l, j_l];$$
$$(\forall 0 \leq i_b \leq k_b \leq j_b)\ (\forall 0 \leq i_l < b, 0 \leq j_l < b-1)\ temp[k_b, j_b, k_l, j_l] = N[k_b, j_b, k_l + 1, j_l];$$
$$(\forall 0 \leq i_b \leq k_b < j_b)\ (\forall 0 \leq i_l < j_l = b-1)\ temp[k_b, j_b, k_l, j_l] = N[k_b + 1, j_b, 0, j_l];$$

$$(\forall 0 \leq i_b = k_b \leq j_b)\ (\forall 0 \leq i_l < j_l < b)\ TempRed\_SG[i_b, j_b, k_b, i_l, j_l] = max_{0 \leq k_l < b}$$
$$N[i_b, k_b, i_l, k_l] + temp\_SG[k_b, j_b, k_l, j_l];$$
$$(\forall 0 \leq i_b \leq k_b = j_b)\ (\forall 0 \leq i_l < j_l < b)\ TempRed\_SG[i_b, j_b, k_b, i_l, j_l] = max_{0 \leq k_l < b}$$
$$N[i_b, k_b, i_l, k_l] + temp\_SG[k_b, j_b, k_l, j_l];$$
$$(\forall 0 \leq i_b \leq k_b = j_b)\ (\forall 0 \leq i_l < b, 0 \leq j_l < b-1)\ temp\_SG[k_b, j_b, k_l, j_l] = N[k_b, j_b, k_l + 1, j_l];$$
$$(\forall 0 \leq i_b = k_b \leq j_b)\ (\forall 0 \leq i_l < b, 0 \leq j_l < b-1)\ temp\_SG[k_b, j_b, k_l, j_l] = N[k_b, j_b, k_l + 1, j_l];$$
$$(\forall 0 \leq i_b = k_b < j_b)\ (\forall 0 \leq i_l < j_l = b-1)\ temp\_SG[k_b, j_b, k_l, j_l] = N[k_b + 1, j_b, 0, j_l];$$

We have two tile groups: one which contains the variables $(N, TempRed\_SG, temp\_SG)$ and another which contains the variables $(TempRed, temp)$.

Example 4.8 (Recursive reduction)**.** *Let us consider the following program:*

$$(\forall 0 < i < N)\ A[i] = \sum_{0 \leq k < i} A[i-1] * A[k];$$
$$(\forall i = 0)\ A[i] = 1;$$

After partitioning, we obtain the following program:

$$(\forall 0 < i_b < N_b)\ (\forall 0 \leq i_l < b)\ A[i_b, i_l] = \sum_{0 \leq k_b \leq i_b} TempRed[i_b, k_b, i_l];$$
$$(\forall i_b = 0)\ (\forall 0 < i_l < b)\ A[i_b, i_l] = \sum_{0 \leq k_b \leq i_b} TempRed[i_b, k_b, i_l];$$
$$(\forall i_b = 0)\ (\forall i_l = 0)\ A[i_b, i_l] = 1;$$

$$(\forall 0 \leq k_b = i_b < N_b)\ (\forall 0 \leq i_l < b)\ TempRed[i_b, k_b, i_l] = \sum_{0 \leq k_l < i_l} temp[i_b, k_b, i_l, k_l] * A[k_b, k_l];$$

$$(\forall 0 \leq k_b < i_b < N_b)\ (\forall 0 \leq i_l < b)\ TempRed[i_b, k_b, i_l] = \sum_{0 \leq k_l < b} temp[i_b, k_b, i_l, k_l] * A[k_b, k_l];$$

$$(\forall 0 \leq k_b \leq i_b < N_b)\ (\forall 0 = i_l \leq k_l < b)\ temp[i_b, k_b, i_l, k_l] = A[i_b - 1, b - 1];$$

$$(\forall 0 \leq k_b \leq i_b < N_b)\ (\forall 0 < i_l \leq k_l < b)\ temp[i_b, k_b, i_l, k_l] = A[i_b, i_l - 1];$$

After analyzing the equations of $TempRed/temp$, we identify 3 dependences which might introduce a loop:

- $A[k_b]$ *in the equation of* $TempRed[i_b, k_b]$
- $A[i_b - 1]$ *in the equation of* $temp[i_b, k_b]$
- $A[i_b]$ *in the equation of* $temp[i_b, k_b]$

Let us examine each dependences separately:

- *The first dependence introduces a cycle between tiles iff* $k_b = i_b$.
- *The second dependence introduces a cycle between tiles iff* $i_b - 1 = i_b$, *which is not feasible*
- *The third dependence introduces a cycle between tiles iff* $i_b = i_b$, *i.e., always. This means that every block of* $TempRed[i_b, k_b]$ *requires some data coming from the block* $A[i_b]$, *thus that we have a cycle between the tile* $A[i_b]$ *and every tiles* $TempRed[i_b, \bullet]$.

Therefore, we have to keep a single tile group. Each tile of this tile group will compute the blocks $A[i_b]$ *and the whole stripes* $TempRed[i_b, \bullet]$ *and* $temp[i_b, \bullet]$. *Physically, this means that it is not possible to tile the dimension of the reduction.*

4.3.3 Monoparametric Tiling with reductions

Now, let us show how to adapt the algorithm presented in Subsection 4.2.4 to manage reductions. We recall that the three steps of this algorithm were the following:

1. Computing the kind of tiles of the program
2. Building the subsystems
   (a) Computing the domain of the local variables of the subsystems
   (b) Obtaining the equations of the subsystems and tracking down their inputs
   (c) Adding the outputs of the subsystems
3. Building the main system

The main difference with the previous algorithm is that we might not remove all the block indices of a variable inside a tile group, which happens only for the variables $TempRed\_SG$. For example, if we consider the variables inside Example 4.6, the variable $TempRed\_SG[i_b, k_b, i_l]$ is inside the same tile group as the variable $x[i_b, i_l]$. Thus, the subsystem computing the tile $x[i_b, \bullet]$ will also compute $TempRed\_SG[k_b, \bullet]$ for $k_b = i_b$ and the dimension corresponding to k_b will stay in the equations of the subsystem. The fact that block indices are not fully removed can means that an entire slice of $TempRed\_SG$ is computed inside a tile.

Because some blocked indices remain in the subsystem and because these indices might interact with the other blocked indices (through constraints, like "$k_b \leq i_b$" in Example 4.8), thus we need to keep all the previously removed blocked indices as parameters.

The main modification of the algorithm comes in step 2, when we form the equations of the subsystem while tracking down the inputs and outputs. Indeed, variables whose block indices are not fully removed must be handled slightly differently. We have 4 kinds of dependences:

- Dependences going from a normal variable to a normal variable (this was always the case in the previous algorithm)
- Dependences going from a normal variable to a $TempRed\_SG$ variable
- Dependences going from a $TempRed\_SG$ variable to a normal variable
- Dependences going from a $TempRed\_SG$ variable to a $TempRed\_SG$ variable

By construction of the $TempRed$ variable, all dependences toward a $TempRed\_SG$ variable are identity dependences and remain inside the same tile group (thus do not create inputs or outputs), thus do not cause any issue. Let us consider the dependences from a $TempRed\_SG$ variable to a normal variable: because not all block indices are removed, such block indices might impact the tiles accessed, meaning that we might require a collection of block as an input, instead of a single one. For example, if we consider the dependence from $TempRed$ to A in Example 4.8, we need all the $A[k_b]$ where $k_b \leq i_b$ as an input of a tile to be able to compute a tile of $TempRed$. We need to differentiate the tiles accessed by such dependences when the data required is coming from the tile itself. For example, if $k_b = i_b$, the data asked by the dependence $A[k_b, k_l]$ is computed internally, thus has to be separated from the data coming from other tiles ($k_b < i_b$, constituting the inputs).

The rest of the algorithm is similar to the outlining algorithm without reduction.

Example 4.9 (Cholesky)**.** *Let us consider the Cholesky computation, in which A is the input ($N \times N$ matrix) and L is the output (lower triangular matrix), where $A = L.L^T$ is a symmetric semi-definite positive matrix:*

$$
\begin{aligned}
&(\forall i = j = 0)\ L[i,j] = \sqrt{A[i,i]};\\
&(\forall i = j > 0)\ L[i,j] = \sqrt{A[i,i] - \sum_{k<j} L[i,k] * L[i,k]};\\
&(\forall i > j = 0)\ L[i,j] = A[i,j]/L[j,j];\\
&(\forall i > j > 0)\ L'[i',j'] = \left(A[i,j] - \sum_{k<j} L[i,k] * L[j,k]\right)/L[j,j];
\end{aligned}
$$

Let us assume that the aspect ratio of L and A are both 1×1. After partitioning, we obtain the system described in Figure 4.14. We have only one variable originally, thus a single tile group at the start.

After analyzing all the dependences involving $TempRed1$ and $TempRed2$, we find that the tiles $TempRed1[i_b, j_b, j_b]$ and $TempRed2[i_b, j_b, j_b]$ are the only tiles which admit a cyclic dependence with the tile $L[i_b, j_b]$ (physically, they correspond to the portions of $TempRed1/2$ which needs values from the tile $L[i_b, j_b]$ to be computed). Therefore, we split these tiles of

$$(\forall i_b = j_b = 0)\ (\forall i_l = j_l = 0)\ L[i_b, j_b, i_l, j_l] = \sqrt{A[i_b, i_b, i_l, i_l]};$$
$$(\forall i_b = j_b = 0)\ (\forall i_l = j_l > 0)\ L[i_b, j_b, i_l, j_l] = \sqrt{A[i_b, i_b, i_l, i_l] - \sum_{k_b \leq i_b} TempRed1[i_b, j_b, k_b, i_l, j_l]};$$
$$(\forall i_b = j_b = 0)\ (\forall i_l > j_l = 0)\ L[i_b, j_b, i_l, j_l] = A[i_b, j_b, i_l, j_l]/L[j_b, j_b, j_l, j_l];$$
$$(\forall i_b = j_b = 0)\ (\forall i_l > j_l > 0)\ L[i_b, j_b, i_l, j_l] = \left(A[i_b, j_b, i_l, j_l] - \sum_{k_b \leq i_b} TempRed2[i_b, j_b, k_b, i_l, j_l]\right) / L[j_b, j_b, j_l, j_l];$$

$$(\forall N_b > i_b > j_b = 0)\ (\forall 0 = j_l \leq i_l < b)\ L[i_b, j_b, i_l, j_l] = A[i_b, j_b, i_l, j_l]/L[i_b, i_b, i_l, i_l];$$
$$(\forall N_b > i_b > j_b = 0)\ (\forall 0 \leq i_l < b, j_l > 0)\ L[i_b, j_b, i_l, j_l] = \left(A[i_b, j_b, i_l, j_l] - \sum_{k_b \leq i_b} TempRed2[i_b, j_b, k_b, i_l, j_l]\right) / L[j_b, j_b, j_l, j_l];$$

$$(\forall N_b > i_b = j_b > 0)\ (\forall 0 \leq i_l = j_l < b)\ L[i_b, j_b, i_l, j_l] = \sqrt{A[i_b, i_b, i_l, i_l] - \sum_{k_b \leq i_b} TempRed1[i_b, j_b, k_b, i_l, j_l]};$$
$$(\forall N_b > i_b = j_b > 0)\ (\forall 0 \leq j_l < i_l < b)\ L[i_b, j_b, i_l, j_l] = \left(A[i_b, j_b, i_l, j_l] - \sum_{k_b \leq i_b} TempRed2[i_b, j_b, k_b, i_l, j_l]\right) / L[j_b, j_b, j_l, j_l];$$

$$(\forall N_b > i_b > j_b > 0)\ (\forall 0 \leq i_l, j_l < b)\ L[i_b, j_b, i_l, j_l] = \left(A[i_b, j_b, i_l, j_l] - \sum_{k_b \leq i_b} TempRed2[i_b, j_b, k_b, i_l, j_l]\right) / L[j_b, j_b, j_l, j_l];$$

$$(\forall 0 \leq k_b = j_b = i_b < N_b)\ (\forall 0 \leq j_l \leq i_l < b)\ TempRed1[i_b, j_b, k_b, i_l, j_l] = \sum_{0 \leq k_l < j_l} L[i_b, k_b, i_l, k_l] * L[i_b, k_b, i_l, k_l]$$
$$(\forall 0 \leq k_b < j_b = i_b < N_b)\ (\forall 0 \leq j_l \leq i_l < b)\ TempRed1[i_b, j_b, k_b, i_l, j_l] = \sum_{0 \leq k_l < b} L[i_b, k_b, i_l, k_l] * L[i_b, k_b, i_l, k_l]$$

$$(\forall 0 \leq k_b = j_b \leq i_b < N_b)\ (\forall 0 \leq j_l \leq i_l < b)\ TempRed2[i_b, j_b, k_b, i_l, j_l] = \sum_{0 \leq k_l < j_l} L[i_b, k_b, i_l, k_l] * L[j_b, k_b, j_l, k_l]$$
$$(\forall 0 \leq k_b < j_b \leq i_b < N_b)\ (\forall 0 \leq j_l \leq i_l < b)\ TempRed2[i_b, j_b, k_b, i_l, j_l] = \sum_{0 \leq k_l < b} L[i_b, k_b, i_l, k_l] * L[j_b, k_b, j_l, k_l]$$

FIGURE 4.14: Cholesky computation after the partitioning transformation and the introduction of the temporary variables $TempRed$

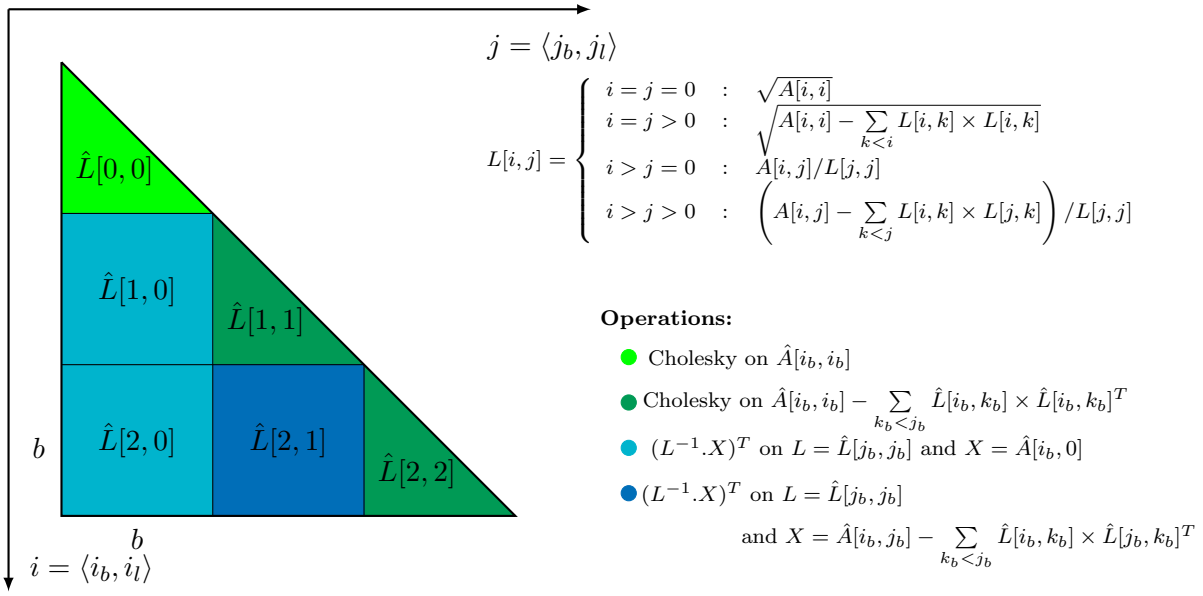


FIGURE 4.15: Cholesky - blocked computation with a tile size $b \times b$. We start from the system of equations in the top-right part of the figure. The left diagram represent the domain of L. After tiling the computation, we can regroup the tiles according to their computation (as shown by the color coding). Finally, we can recognize each kind of tiles as a combination of matrix operations.

$TempRed1$ and $TempRed2$ from the rest of the computation, forming in total 3 tiles groups $(L, TempRed1\_SG, TempRed2\_SG)$, $(TempRed1)$ and $(TempRed2)$.

Because the first tile group admit 4 kinds of tiles, we will obtain 6 subsystems in total:

1. *One computing $L[0,0]$, $TempRed1\_SG[0,0,0]$ and $TempRed2\_SG[0,0,0]$.*
2. *One computing $L[i_b,0]$ and $TempRed2\_SG[i_b,0,0]$ for $i_b > 0$.*
3. *One computing $L[i_b,i_b]$, $TempRed1\_SG[i_b,i_b,i_b]$ and $TempRed2\_SG[i_b,i_b,i_b]$ for $i_b > 0$.*
4. *One computing $L[i_b,j_b]$ and $TempRed2\_SG[i_b,j_b,j_b]$ for $i_b > j_b > 0$*
5. *One computing $TempRed1[i_b,j_b,k_b]$ for $k_b < j_b$ (corresponding to the accumulation needed to compute $L[i_b,j_b]$)*
6. *One computing $TempRed2[i_b,j_b,k_b]$ for $k_b < j_b$ (corresponding to the accumulation needed to compute $L[i_b,j_b]$)*

As showed by Figure 4.15, we can recognize the computation performed inside these subsystem as matrix operations: subsystems 1 and 3 corresponding to a mini-Cholesky computation, 2 and 4 corresponding to the operation "$(L^{-1}.X)^T$" (which is an instance of the $xTRSM$ operation in BLAS), 5 and 6 corresponding to a transposed matrix product. In Chapter 5, we will present a method to recognize these operations from the subsystem we obtained in this section.

4.4 Experimental Validation

In this section, we present our implementation of the monoparametric tiling transformation and evaluate its scalability.

Implementation The rectangular monoparametric tiling transformation has been implemented in Java, using the *AlphaZ* compiler framework [89], on top of our monoparametric partitioning transformation presented in Subsection 3.2.3.

We have implemented several options to the monoparametric tiling transformation, in addition to the options to the monoparametric partitioning transformation:

- We can *remove the classification per tile of the outputs of the subsystem.* For the Smith-Watterman example of SubSection 4.2.1, this means that instead of having 2 outputs (corresponding to the values sent to the tile on the right and on the top), we will have a single output, which domain is the set of data needed outside of the tile (i.e., the corner formed by the last column and the last row).
- We can *homogenize the domains of the outputs of the subsystems* across the different kind of tiles. For the Jacobi1D example (Example 4.5 Page 96), if we consider the set of values sent to the tile diagonally above-right in Figure 4.7, we either send 0, 1 or 2 values, depending on the kind of tile of the current tile. By default, when we regroup the values of all these outputs in the main system in a single variable, the domain of this variable will be a union of at least 3 polyhedra. The union of this domain can be much larger for other programs and slow down the following analyses.

We solve this issue by padding the smallest domains. For the Jacobi1D example, this means that we will systematically send 2 values, by adding 0 values for the missing parts. Thus, the domain of the local variable of the main system which regroups all of the corresponding outputs will be a single polyhedron, at the price of a slight increase in the communication.

By default, each variable is placed in a different tile group, with no change of basis preprocessing step.

Experiment on the scalability of the monoparametric tiling transformation We want to study the scalability of our implementation of the monoparametric tiling transformation. This means that we want to check that the time performed by our transformation in a compiler is reasonable.

As our set of benchmark, we use Polybench/Alpha[1] benchmarks, an hand-written *Alpha* implementation of the *Polybench 4.0* benchmark suite. We run our experiment on a machine with an Intel Xeon E5-1650 CPU with 12 cores running at 1.6 GHz (max speed at 3.8GHz), and 31GB of memory.

Because we are considering a tiling, we have to consider the legality condition. We found that the default rectangular tiling (all variables are tiled separately) is legal for all benchmarks, except:

- Some of the linear algebra solvers (`durbin`, `gramschmidt`, `lu`, `ludcmp`)
- All of the stencils (`adi`, `fdtd-2d`, `jacobi-1d`, `jacobi-2d`, `seidel-2d`, `heat-3d`).

For `durbin` and `lu`, because of mutual dependences, we need to have a single tile group for all the variables of these programs. `ludcmp` is the same than a `lu`, plus two forward substitution computations ($\vec{x} = L^{-1}.\vec{b}$) which can be tiled in separate tile groups.

[1] `http://www.cs.colostate.edu/AlphaZsvn/Development/trunk/mde/edu.csu.melange.alphaz.polybench/polybench-alpha-4.0/`

For `gramschmidt`, this program does not admit a legal two-dimensional tiling. Because our current implementation of the monoparametric tiling transformation forces us to tile all dimensions, we cannot apply it legally.

For the stencils kernels, it is possible to obtain a legal tiling by skewing the iteration space beforehand, and, in the case of `adi` and `fdtd-2d`, tiling the mutual dependent local variables together.

For each kernel, we apply the monoparametric tiling transformation and report the following informations:

- The time taken by the monoparametric partitioning transformation.
- The time taken by the preprocessing step, after the monoparametric partitioning transformation and before the monoparametric tiling part. In particular, this preprocessing step includes the management of reductions, the normalization of the program, if the newly introduced variables $TempRed$ are split. We also compute the context domain of the subexpressions of the form $Var[u_b(\vec{i_b}), u_l(\vec{i_l})]$, because this information is needed in order to determine the inputs and outputs of a subsystem.
- The time taken by the three steps of the monoparametric tiling transformation (computing the kinds of tile, building the subsystems and building the main system).
- The total time spent in the transformation.
- The time taken by the computation of the context domains of all the subexpressions of the program (in order to compare this time with the ones from the monoparametric partitioning)
- The number of subsystems generated.
- The average number of nodes in the AST of a subsystem, in order to give an idea of the size of a subsystem.
- The number of equations inside the main program, in order to give an idea of the size of the main system.

Time taken (ms)	correlation	covariance	gemm	gemver	gesummv	symm	syr2k	syrk	trmm	2mm	3mm
Partitioning	166	102	109	91	53	190	110	69	66	233	433
Preprocessing	432	322	276	160	137	905	244	126	296	259	382
Step 1 Kind of tiles	4	3	2	2	1	6	2	1	2	4	7
Step 2 Subsystems	382	277	282	75	55	660	176	112	87	646	1479
Step 3 Main System	34	18	12	18	7	19	13	9	8	25	44
Total Time	1206	817	754	457	309	1928	658	383	516	1297	2471
Context Domain	1030	651	390	363	237	2617	456	274	277	632	837
Num SubSystem	10	6	2	5	3	12	3	2	3	4	6
Average num of nodes in subsystem	14	14	12	11	11	15	13	12	9	10	8
Num Equations Main	24	14	5	15	7	25	7	5	7	9	13

Time taken (ms)	atax	bicg	doitgen	mvt	cholesky	durbin	gramschmidt	lu	ludcmp	trisolv	deriche
Partitioning	71	217	152	62	165	177	–	313	450	52	329
Preprocessing	144	175	178	143	1139	1130	–	1522	1946	193	546
Step 1 Kind of tiles	1	7	3	1	4	4	–	6	7	1	4
Step 2 Subsystems	148	236	406	68	224	230	–	304	426	52	227
Step 3 Main System	12	13	18	9	25	49	–	33	53	7	69
Total Time	436	706	828	346	1806	1772	–	2350	3188	363	1928
Context Domain	871	273	300	265	1057	2143	–	1555	2444	358	1719
Num SubSystem	4	4	2	4	7	8	–	10	16	3	24
Average num of nodes in subsystem	7	7	8	8	36	70	–	33	30	23	30
Num Equations Main	9	10	5	10	31	63	–	44	67	9	59

FIGURE 4.16: Time taken by the hyperrectangular monoparametric tiling transformation inside the compiler - Part 1. We also report the number of subsystems produced, the average number of nodes of a AST of a subsystem, and the number of equations (use and normal) in the main system.

Time taken (ms)	floyd-warshall	nussinov	adi	fdtd-2d	jacobi-1d	jacobi-2d	seidel-2d	heat-3d
Partitioning	72	248	1907	1301	119	529	538	6088
Preprocessing	78	3605	18535	19138	867	19037	28497	$8m2s$
Step 1 Kind of tiles	2	8	72	122	4	28	28	297
Step 2 Subsystems	19	592	1874	2740	107	1764	2823	88267
Step 3 Main System	28	51	427	940	28	242	215	2905
Total Time	302	4689	23719	24871	1251	21859	32474	$9m40s$
Context Domain	629	4968	53348	98721	1252	28482	34515	$21m44s$
Num SubSystem	3	25	48	53	9	33	33	129
Average num of nodes in subsystem	16	45	319	234	54	184	259	1029
Num Equations Main	15	118	676	1067	45	293	333	1985

FIGURE 4.17: Time taken by the hyperrectangular monoparametric tiling transformation inside the compiler - Part 2. We also report the number of subsystems produced, the average number of nodes of a AST of a subsystem, and the number of equations (use and normal) in the main system. All the considered stencil have an order of 1.

The result of our experiments are presented in Figure 4.16 and 4.17.

Most of the time is spent during the preprocessing step and the construction of the subsystems. The preprocessing step contains a traversal of the monoparametric partitioned program, in order to compute the context domain of the subexpressions of the form $Var[u_b(\vec{i_b}), u_l(\vec{i_l})]$. Notice that this step is faster compared to the full context domain computation we considered in Subsection 3.2.3, because we do not need to compute the context domain for all the subexpressions of the program. The construction of the subsystems also contains a traversal of the monoparametric partitioned program, in order to build the equations of the subsystems. Thus, the time taken by this transformation is mostly caused by the size of the program after applying the monoparametric partitioning transformation.

We also notice that the time taken by a context domain computation following the monoparametric tiling transformation is reduced compared to the time taken by the same polyhedral

analysis after the monoparametric partitioning transformation (cf Figure 3.9 Page 59). Thus, distributing the computation across subsystem helps reducing the time taken by the polyhedral analysis on the transformed program.

We cannot reduce the size of the tiled program while keeping the full representation of the tiled program. Indeed, each subsystem contains a different computation (by definition of the kind of tiles), thus need to be generated. We also remark that the size of the program after tiling is independent from the monoparametric nature of the transformation, and is caused by our choice of keeping the full representation of the tiled program.

Chapter 5

Template Recognition

In Chapters 3 and 4, we introduced monoparametric tiling. This transformation allows us to partition the computation of a program into tiles, and isolate the computation of each tile into a separate subsystem. These subsystems are studied separately by our template recognition framework described in Chapter 6.

In this chapter, we present our template recognition algorithm, which is used by our framework to detect linear algebra operations. This template recognition algorithm is an adaptation of the equivalence algorithm from Barthou et al. [8], whose main concepts are briefly reviewed in Section 5.1. We present the algorithm itself in Section 5.2 and present some examples of its application in Section 5.3.

In Section 5.4, we present several adaptations of our algorithm in order to manage the semantic properties commonly encountered in linear algebra computations, such as the associativity and commutativity of binary operators. Finally, we evaluate our algorithm in Section 5.5 before concluding this chapter with some additional remarks in Section 5.6.

5.1 Barthou's equivalence algorithm

Barthou's equivalence algorithm [8] (see Section 2.4) consists of two steps: the first step builds an equivalence automaton and the second step checks some reachability properties in this automaton. The equivalence automaton captures the equivalence problem between two programs. Each state of the equivalence automaton corresponds to a comparison between two computations. Progressing in this automaton corresponds to unrolling both programs, and progressively eliminating the matching computation encountered. The final states of the automaton corresponds to comparison where nothing can be eliminate or further unrolled. There are two kinds of final states: failure states (which denotes comparisons between two expressions which are obviously not equivalent) and accept states (which denotes comparisons between two expressions which might be equivalent, depending on the indices of the expressions).

After building the equivalence automaton, we examine the reachability set of the success and failure states. The two compared programs are equivalent iff any path in the automaton which starts from the initial states with equal indices for the output of both programs (i) does not reach any failure state (ii) reaches an accept state only when the indices of both inputs of the accept state are equal. If these properties are satisfied, the two programs are performing exactly the same sequence of operations, i.e. they are Herbrand-equivalent.

5.2 Adapting the equivalence algorithm into a template algorithm

The main difference between an equivalence algorithm and a template recognition algorithm is that the inputs of a template are unknown and might correspond to an arbitrarily elaborate computation. Thus, one of the main challenges of template recognition is to deduce these inputs. In particular, if an input appears in several places in a template, we should check that the corresponding computation is coherent across all of these places.

Step 1 - Construction of the equivalence automaton In this step, we reuse the equivalence automaton construction process of Barthou [8], while modifying the notion of success and failure state of a equivalence automaton to account for the inputs of a template.

Definition 5.1 (Template final state)**.** Considering an equivalence automaton between a program P and a template P':

- A *template-accept state* is a state which is labeled by an equation of the form $Expr = I'$, where I' is an input of the template. This is more relaxed compared to the notion of accept state, which imposes that exactly the same computation occurs in both side of the equation.
- A *template-failure state* is a state which is labeled by an equation of either:
  - $f(\dots) = f'(\dots)$ where f and f' are different operators
  - $I = f'(\dots)$ where f' is an operator and I is an input of the program

Intuitively, a template-failure state corresponds to a comparison between a sub-expression of a program and of a template, which trivially cannot match, whatever values of the input of the template. The notion of template-accept state is more relaxed than the notion of accept state and the notion of template-failure state is more restricted than the notion of failure state. The rest of the definitions of the equivalence automaton and its constructions rules stay unchanged.

Because we assume that the output of the template matches the output of the program, it might impose some constraints on the parameters of the template (typically, both output arrays must be of the same size). We extract these constraints and keep them.

Step 2 - Extracting the constraints on the inputs of the template Now that the automaton is built, we need to check that the template-failure states are not accessible, and we need to check that there exist some valid input of the template which simultaneously satisfies all the accessible template-accept states.

As for the template-failure states, we compute their accessibility set and, because they are not supposed to be reachable, we check that these sets are empty. If a template-failure state is accessible for any values of the template parameters, then we can conclude that the template does not match. If a template-failure state is never accessible, for any values of the template parameters, we can safely ignore it for the rest of the algorithm. If a template-failure state is accessible only for certain values of the template parameters, we can extract the constraints on the template parameters which makes the corresponding accessibility set empty and consider them as constraints on the parameters of the template.

For example, if we compare a program $O = I_1 + I_2$ and we try to match it to a template $O' = I' + I'$, we obtain two template-accept states: $I_1 = I'$ and $I_2 = I'$. The first template-accept state can be satisfied by taking $I' = I_1$ as the input of the template. The second template-accept state can be satisfied by taking $I' = I_2$ as the input of the template. However, it is not possible to satised both template-accept state at the same time, when both of them are accessible. Therefore, the template does not match with the program.

We examine the automaton and extract the constraints on the inputs of the template by examining the template-accept state. Because a template-accept state is always of the form $Expr = I'$, for each input of the template I', we can list the $Expr$ that are matched to this input, and compute the corresponding accessibility set. Formally, we obtain the following list, for every template input I':

$$\begin{cases} \dots \\ (\forall(\vec{i}, \vec{i'}) \in S_{I',k}) \quad I'[\vec{i'}] = Expr_k[\vec{i}] \\ \dots \end{cases}$$

where $S_{I',k}$ is the accessibility set of the template-accept state $Expr_k[\vec{i}] = I'[\vec{i'}]$.

Step 3 - Determining the inputs of the template We independently consider each input I' of the template and its associated constraints, and try to determine a valid value of such input. For each $\vec{i'}$, we examine how many pairs $(k, \vec{i})$ there exist such that $I'[\vec{i'}] = Expr_k[\vec{i}], (\vec{i}, \vec{i'} \in S_{I',k})$, i.e., how many expressions $Expr_k[\vec{i}]$ are matched to the same $\vec{i'}$.

In practice, it is not possible to iterate over all $\vec{i'}$, because there is a parametric number of them. Instead, we can compute separately the projections of the $S_{I',k}$ on $\vec{i'}$, then consider the non-empty intersections pieces between a subset of these projected sets. There is only a finite non parametric number of these intersections, and, in any of these intersections, all the $\vec{i'}$ will have the same expressions $Expr_k[\vec{i}]$ mapped to them. Thus, by iterating over these intersections, we can cover all the cases encountered by the $\vec{i'}$.

If there is only one expression $Expr_k[\vec{i}]$ for a given template input ($(\forall(\vec{i},\vec{i'}) \in S_{I'})\ I'[\vec{i'}] = Expr[\vec{i}]$) and if, for every $\vec{i'}$, there is only one single expression $Expr[\vec{i}]$, then we can trivially set as the value of our template:

$$I[\vec{i'}] = Expr[u(\vec{i'})] \text{ where } \vec{i} = u(\vec{i'})$$

If there are several expressions $Expr_k[\vec{i}]$ associated to a given template input, but, for each $\vec{i'}$, there exist only one pair $(k,\vec{i})$, then we can set the value of our template input as a disjunction of values, defined over disjoint domains:

$$(\forall \vec{i'} \in \pi(S_{I',k}))\ I[\vec{i'}] = Expr_k[u_k(\vec{i'})] \text{ where } \vec{i} = u_k(\vec{i'})$$

where $\pi(\vec{i},\vec{i'}) = \vec{i'}$ is a projection function.

In general, we might have several expressions $Expr_k[\vec{i}]$ which are mapped to the same $I'[\vec{i'}]$. In that situation, we have to ensure that the pairs are equivalent before selecting one of them as the value of our template input. If this is not the case, this means that two non-equivalent expressions are mapped to the same portion of the same input of the template, thus that the program does not match the template. If all the pairs mapped to the same $I'[\vec{i'}]$ are equivalent, we can select any of them. Another possibility is that the pairs are equivalent only for some values of the parameters: in that case, we extract the constraints on the parameters.

Final step If a value is found for every input of the template, and if the constraints on the parameters are satisfiable, then the template matches the program. In some situations, several values of the template parameters are valid: in that case, we choose to select the biggest values

of the parameters, such that we match as much operations as possible from the program with the template.

The whole algorithm is summarized in Algorithm 1.

Algorithm 1 Template Recognition Algorithm adapted from Barthou's equivalence algorithm

Require: Program P, Template T
Ensure: Does the template match the program? If yes, valid inputs of the template

Build the template-equivalence automaton ▷ **Building the automaton**
Extract the constraints on the template parameters from the outputs

for each template-failure state **do** ▷ **Template-failure states**
    Compute the accessibility set of this state
    Compute the set of template parameters for which this set is accessible
    Add their negation to the constraints on the template parameters
    If the constraints on the template parameters are not satisfiable, return "DO NOT MATCH"
end for

for each template-access state "$Expr[\vec{i}] = I'[\vec{i'}]$" **do** ▷ **Template-accept states**
    Compute the accessibility set
    Add it to the list of constraint on the template input I'
end for

for each template input I' **do** ▷ **Solving the constraints**
    for all $\vec{i'}$ such that $I'[\vec{i'}]$ is matched to several expressions **do**
        Use an equivalence algorithm to check if these expressions are equivalent on the domain they intersect.
        If they are equivalent only for some conditions on the template parameters, add them to the constraints on the template parameters
        If they are not equivalent or if the constraints on the template parameters are not satisfiable, return "DO NOT MATCH"
    end for
    Select one expression which is matched to $I'[\vec{i'}]$ as the value of the input of the template on this domain.
end for

Return "MATCH", and the list of inputs found for the template.

It is possible to speed up the recognition algorithm (resp. the equivalence algorithm) by detecting when a (template) failure state is trivially accessible. While building the automaton, we can compute a subset of the accessibility set on-the-fly, corresponding to the set $(\vec{i}, \vec{i'})$ on which we might end up on a given state, without taking any loops. If this subset is not empty for

a failure state, we can immediately interrupt the construction of the automaton, and conclude that the template does not match the program (resp. both programs are not equivalent).

The effectiveness of this optimization overlaps with the scalar operation classification of our template library: if the first operator encountered by our template and the library are different, then we can trivially conclude that the template does not match.

As with Barthou's equivalence algorithm, this template recognition algorithm relies on a transitive closure, which might not be exact. If we have an overapproximation of the transitive closure instead, then the template recognition algorithm is still sound:

- If the reachability set of a template failure-state is overapproximated, because we consider its negation to extract constraints on the parameters, these constraints might be more restrictive than needed, but are sound.
- If the reachability set of a template accept-state is overapproximated, then we have a constraint on an input of a template which spans over a larger domain than needed. It might create an intersection with another constraint (and trigger a check of equivalence between the two conflicting constraints) and might fail the algorithm. Nevertheless, the algorithm stays also sound on that part.

5.3 Examples

Example 5.1. *Let us consider the following (simple) program*

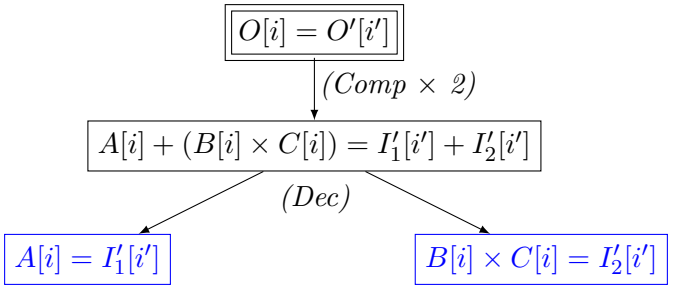
$$(\forall 0 \leq i < N)\ O[i] = A[i] + (B[i] \times C[i]);$$

where A, B and C are inputs of the program. Let us try to match this program with the following template (corresponding to the addition of two vectors of size N'):

$$(\forall 0 \leq i' < N')\ O'[i'] = I'_1[i'] + I'_2[i'];$$

where I'_1 and I'_2 are inputs of the template.

First of all, we build the equivalence automaton:



We have one constraints on the parameters coming from the outputs: the size of O' must be the same than the size of O. Therefore, $N = N'$. We also have two template-accept state: $A[i] = I'_1[i']$ and $B[i] \times C[i] = I'_2[i']$. The accessibility set of both template-accept state are both $\{i, i' \mid 0 \leq i = i' < N = N'\}$.

Let us consider the first input of the template I'_1: for every $0 \leq i' < N'$, there is only one expression which is mapped to $I'_1[i']$ in the automaton, which is $A[i]$, where $i = i'$. Therefore, $I'_1[i'] = A[i']$ is a valid input of the template.

Let us consider the second input of the template I'_2: for every $0 \leq i' < N'$, there is only one expression which is mapped to $I'_2[i']$ in the automaton, which is $B[i] \times C[i]$, where $i = i'$. Therefore, $I'_2[i'] = B[i'] \times C[i']$ is a valid input of the template.

The constraints on the parameters of the template are satisfiable ($N' = N$) and we found valid inputs of the template, thus we conclude that the template matches.

Example 5.2. *Let us consider the following program:*

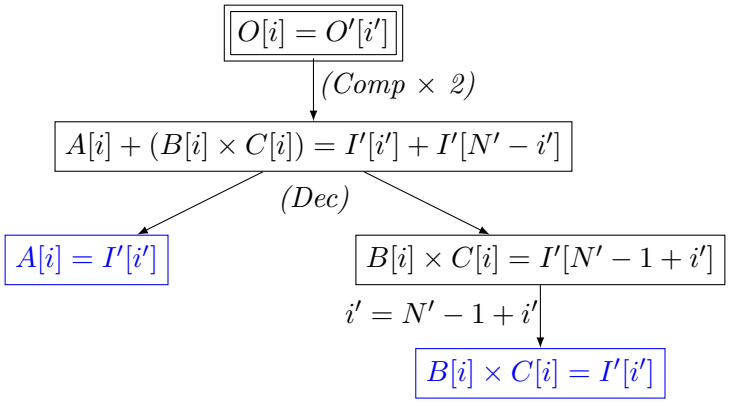
$$(\forall 0 \leq i < N)\ O[i] = A[i] + (B[i] \times C[i]);$$

where A, B and C are inputs of the program. Let us try to match this program with the following template (corresponding to the addition between a vector of size N' and its reverse):

$$(\forall 0 \leq i' < N')\ O'[i'] = I'[i'] + I'[N' - i'];$$

where I' is the input of the template.

First of all, we build the equivalence automaton:



We have one constraints on the parameters coming from the outputs, which imposes $N = N'$. We have two template-accept state: $A[i] = I'[i']$ and $B[i] \times C[i] = I'[i']$. The accessibility set are both $\{i, i' \mid 0 \leq i = i' < N = N'\}$.

Let us consider the unique input of the template I'. We have two expressions mapped to $I'[i']$ for every $0 \leq i' < N'$, which are $A[i]$ (where $i = i'$) and $B[i] \times C[i]$ (where $i = N' - i'$). However, these expressions are not equivalent. Therefore, we conclude that the template does not match (because there is no value for the input I' which satisfies both template-accept states at the same time).

Example 5.3. *Let us consider the following program:*

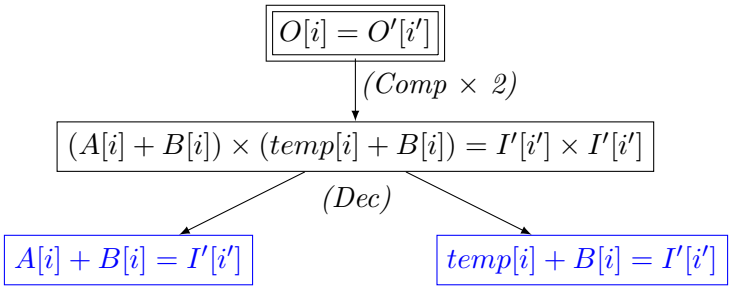
$$(\forall 0 \leq i < N)\ O[i] = (A[i] + B[i]) \times (temp[i] + B[i]);$$
$$(\forall 0 \leq i < N)\ temp[i] = A[i];$$

where A and B are inputs of the program. Let us try to match this program with the following template (corresponding to the multiplication between a vector of size N' with itself):

$$(\forall 0 \leq i' < N')\ O'[i'] = I'[i'] \times I'[i'];$$

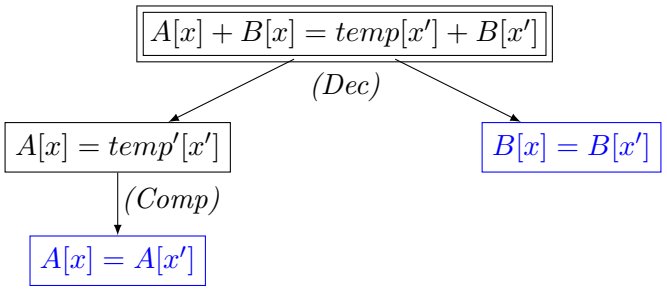
where I' is the input of the template.

First of all, we build the equivalence automaton:



We have one constraints on the parameters coming from the outputs, which imposes $N = N'$*. We have two template-accept state:* $A[i] + B[i] = I'[i']$ *and* $temp[i] + B[i] = I'[i']$*. The accessibility set are both* $\{i, i' \mid 0 \leq i = i' < N = N'\}$*.*

Let us consider the unique input of the template I'*. We have two expressions mapped to* $I'[i']$ *for every* $0 \leq i' < N'$*, which are* $A[i] + B[i]$ *(where* $i = i'$*) and* $temp[i] + B[i]$ *(where* $i = i'$*). We need to check if both expressions are equivalent on the domain* $0 \leq i < N$*. The corresponding equivalence automaton is:*



Both accept states are accessible, and compare the same array cells. Thus, both expressions are equivalent. Stepping back, this means that both $A[i'] + B[i']$ *and* $temp[i'] + B[i']$ *are valid values for the input of the template* $I'[i']$*, for* $0 \leq i' < N'$*. Thus, we conclude that the template matches, and the input of the template will be* $I[i'] = A[i'] + B[i']$ *(or* $I[i'] = temp[i'] + B[i']$*, if we pick the other expression).*

Example 5.4. *Let us consider the following program, corresponding to a serialized reduction over two arrays of size* N *(*I_2 *and* I_1*,* I_2 *being summed in the reverse order), and an element*

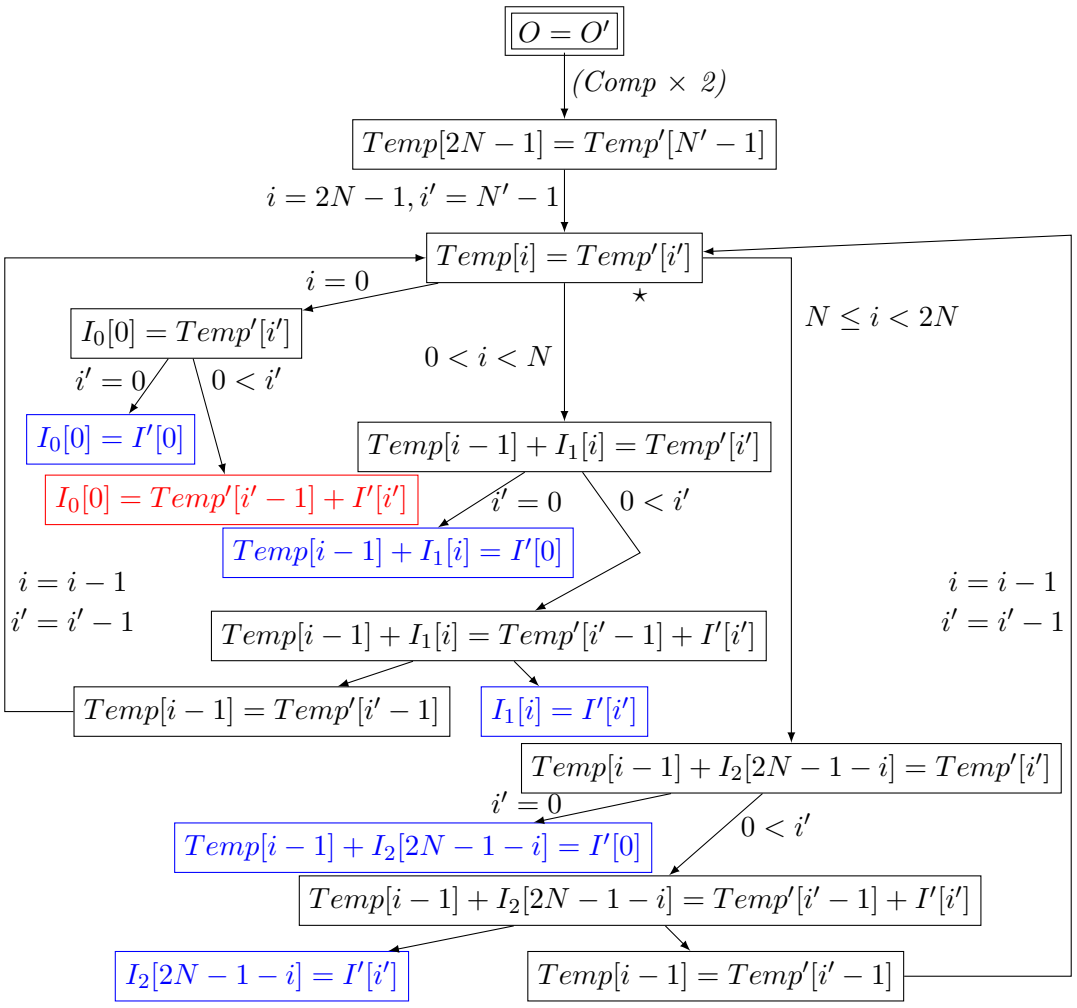
$I_0[0]$*:*

$$\begin{aligned} O = &\quad Temp[2N-1] \\ (\forall N \leq i < 2N)\ Temp[i] = &\quad Temp[i-1] + I_2[2N-1-i] \\ (\forall 0 < i < N)\ Temp[i] = &\quad Temp[i-1] + I_1[i] \\ (\forall i = 0)\ Temp[i] = &\quad I_0[0] \end{aligned}$$

where I_0*,* I_1 *and* I_2 *are inputs of the program. Let us try to match this program with the following template (corresponding to a serialized reduction along an array of size* N'*):*

$$\begin{aligned} O' = &\quad Temp'[N'-1] \\ (\forall 0 < i' < N')\ Temp'[i'] = &\quad Temp'[i'-1] + I'[i'] \\ (\forall i' = 0)\ Temp'[i'] = &\quad I'[0] \end{aligned}$$

First of all, we build the equivalence automaton:



The outputs of the template and the program are both scalar, thus we do not have any constraint on the parameters of the template coming from them. While computing the accessibility set and applying a transitive closure, we find that the accessibility set of the state " $Temp[i] = Temp[i']$ *" (the state besides the star* $\star$*) is* $\{i, i' | (\exists k) i = 2N-1-k \wedge i' = N'-1-k\} = \{i, i' | i = 2N-N'+i'\}$.

We have one template-failure state $I_0[0] = Temp'[i'-1]+I'[i']$*, whose accessibility set is* $\{i, i' | i' = N' - 2N + i \ \wedge \ i = 0 \ \wedge \ 0 < i'\} = \{i, i' | 2N < N' \ \wedge \ i = 0 \ \wedge \ 0 < i'\}$*. Therefore, so that this set is no longer accessible, we have the constraint* $N' \leq 2N$*. Physically, this means that the reduction we try to detect with our template must not be too long:* $N' = 2N$ *corresponds to detecting the whole program as a reduction (with a piece-wise input) and* $N' < 2N$ *corresponds to detecting only part of the program as a reduction.*

Let us examine the template-success state. We have 5 of them, all of them on the template input I', the corresponding constraints being:

- $(i,i') \in \{i,i' | i = i' = 0 \ \wedge \ i = 2N - N' + i'\}\ I_0[0] = I'[0]$
- $(i,i') \in \{i,i' | 0 < i < N \ \wedge \ i' = 0 \ \wedge \ i = 2N - N' + i'\}\ Temp[i-1] + I_1[i] = I'[0]$
- $(i,i') \in \{i,i' | 0 < i < N \ \wedge \ 0 < i' \ \wedge \ i = 2N - N' + i'\}\ I_1[i] = I'[i']$
- $(i,i') \in \{i,i' | N \leq i < 2N \ \wedge \ i' = 0 \ \wedge \ i = 2N - N' + i'\}\ Temp[i-1] + I_2[2N-i-1] = I'[0]$
- $(i,i') \in \{i,i' | N \leq i < 2N \ \wedge \ 0 < i' \ \wedge \ i = 2N - N' + i'\}\ I_2[2N-i-1] = I'[i']$

Let us determine the value of the template input I'. For $i' = 0$, we have 3 constraints which maps 3 different expressions to $I'[0]$ ($I_0[0]$, $Temp[i-1] + I_1[i]$ and $Temp[i-1] + I_2[2N-i-1]$). The first constraint imposes that the template parameter N' is equal to $2N$. The second constraint imposes that $0 < 2N - N' < N$, i.e., $N < N' < 2N$ and the third one that $N \leq 2N - N' < 2N$, i.e. $0 < N' \leq N$. Therefore, these 3 constraints are disjoints, and we have:

$$I'[0] = \begin{cases} N' = 2N & : \ I_0[0] \\ N < N' < 2N & : \ Temp[2N - N' - 1] + I_1[2N - N'] \\ 0 < N' \leq N & : \ Temp[2N - N' - 1] + I_2[N' - 1] \end{cases}$$

For $0 < i'$, we have 2 constraints which maps 2 different expressions to $I'[i']$ ($I_1[i]$ and $I_2[2N - i - 1]$). The first constraint imposes $0 < 2N - N' + i' < N$, i.e., $N' - 2N < i' < N' - N$. Because we have already determined that $N' \leq 2N$, $0 \leq i' < N' - N$. The second constraint imposes $N \leq 2N - N' + i' < 2N$, i.e., $N' - N \leq i' < N'$. Thus, both of them are disjoints and we have:

$$I'[i'] = \begin{cases} 0 < i' < N' - N & : \ I_1[2N - N' + i'] \\ N' - N \leq i' < N' & : \ I_2[N' - 1 - i'] \end{cases}$$

Therefore, the template matches for any $N' \leq 2N$. To maximize the part of the program covered by the template, we pick $N' = 2N$, which gives us, as the input of the template:

$$
\begin{aligned}
&(\forall i' = 0)\ I'[i'] = I_0[0] \\
&(\forall 0 < i' < N)\ I'[i'] = I_1[i'] \\
&(\forall N \leq i' < 2N)\ I'[i'] = I_2[2N - 1 - i']
\end{aligned}
$$

Therefore, we conclude that the template matches.

Example 5.5. *Let us consider a Cholesky computation and let us apply the transformation we have presented in the previous chapter, for square tile sizes ($b \times b$). This example was already discussed in Example 4.9 and Figure 4.15 subsumes the different blocks we obtain. Let us consider the equations obtained for the dark green tiles (tiles whose blocked indices satisfy $0 < j_b$ and $i_b = j_b$):*

$$
\begin{aligned}
&(\forall 0 = j_l = i_l < b)\ Lloc[i_l, j_l] = \sqrt{Ain[j_l, j_l] - \sum_{k_b < i_b} TR0in[k_b, i_l, j_l]}; \\
&(\forall 0 < j_l = i_l < b)\ Lloc[i_l, j_l] = \sqrt{Ain[j_l, j_l] - \sum_{k_b < i_b} TR0in[k_b, i_l, j_l] - TR0\_SG[i_b, i_l, j_l]}; \\
&(\forall 0 = j_l < i_l < b)\ Lloc[i_l, j_l] = \left(Ain[i_l, j_l] - \sum_{k_b < i_b} TR1in[k_b, i_l, j_l] \right) / Lloc[j_l, j_l]; \\
&(\forall 0 < j_l < i_l < b)\ Lloc[i_l, j_l] = \left(Ain[i_l, j_l] - \sum_{k_b < i_b} TR1in[k_b, i_l, j_l] - TR1\_SG[i_b, i_l, j_l] \right); \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad / Lloc[j_l, j_l];
\end{aligned}
$$

$$
\begin{aligned}
&(\forall k_b = i_b, 0 < j_l = i_l < b)\ TR0\_SG[k_b, i_l, j_l] = \sum_{k_l < j_l} Lloc[j_l, k_l] \times Lloc[j_l, k_l]; \\
&(\forall k_b = i_b, 0 < j_l < i_l < b)\ TR1\_SG[k_b, i_l, j_l] = \sum_{k_l < j_l} Lloc[i_l, k_l] \times Lloc[j_l, k_l];
\end{aligned}
$$

where b, i_b, j_b are parameters of the program. $Ain[i_l, j_l]$, $TR0in[k_b, i_l, j_l]$ and $TR1in[k_b, i_l, j_l]$ are inputs of the program. Ain corresponds to the block $A[i_b, j_b]$ of the program, $TR0in[k_b, i_l, j_l]$ corresponds to the partial accumulation of the $\sum_k L[j,k] \times L[j,k]$ over the block (i_b, j_b, k_b) and $TR1in[k_b, i_l, j_l]$ corresponds to the partial accumulation of the $\sum_k L[i,k] \times L[j,k]$ over the block (i_b, j_b, k_b).

Let us compare this program with the following template, corresponding to a scalar Cholesky computation:

$$
\begin{aligned}
&(\forall i' = j' = 0)\ L'[i', j'] = \sqrt{A'[i', i']};\\
&(\forall 0 < j' = i' < N')\ L'[i', j'] = \sqrt{A'[i', i'] - \sum_{k' < j'} L'[i', k'] \times L'[i', k']};\\
&(\forall 0 = j' < i' < N')\ L'[i', j'] = A[i', j']/L'[j', j'];\\
&(\forall 0 < j' < i' < N')\ L'[i', j'] = \left(A'[i', j'] - \sum_{k' < j'} L'[i', k'] \times L'[j', k']\right)/L'[j', j'];
\end{aligned}
$$

where A' is the input of the template and N' a parameter of the template.

Some of the operators considered are associative and commutative. The template recognition algorithm can manage these semantic properties, as it will be shown in Section 5.4, by considering the possible permutations of their elements. However, in the context of this example, we will not consider these semantic properties.

In particular, this means that we consider a reduction as an operator, admitting a parametric number of elements. Thus, two reductions are considered equivalent iff every subexpression at the same position in both sides are equivalent (i.e., the third subexpression of the left reduction must be equivalent to the third subexpression of the right reduction, and no reordering of the subexpressions under both reductions is allowed). Also, because the considered reduction operator admit a parametric number of elements, this number must be the same, giving us an additional constraint on the template parameters.

The equivalence automaton is shown in two parts, in Figure 5.1 and Figure 5.2.

We have one constraints on the parameters coming from the outputs: $b = N'$. While computing the accessibility sets and applying a transitive closure, we find that the accessibility set of the state $Lloc[i_l, j_l] = L'[i', j']$ is $\{i_l, j_l, i', j' \mid j_l = j' \leq i_l = i'\}$. Because of this, none of the template-failure states are accessible in the automaton, thus we do not have any additional constraints on the parameters of the template.

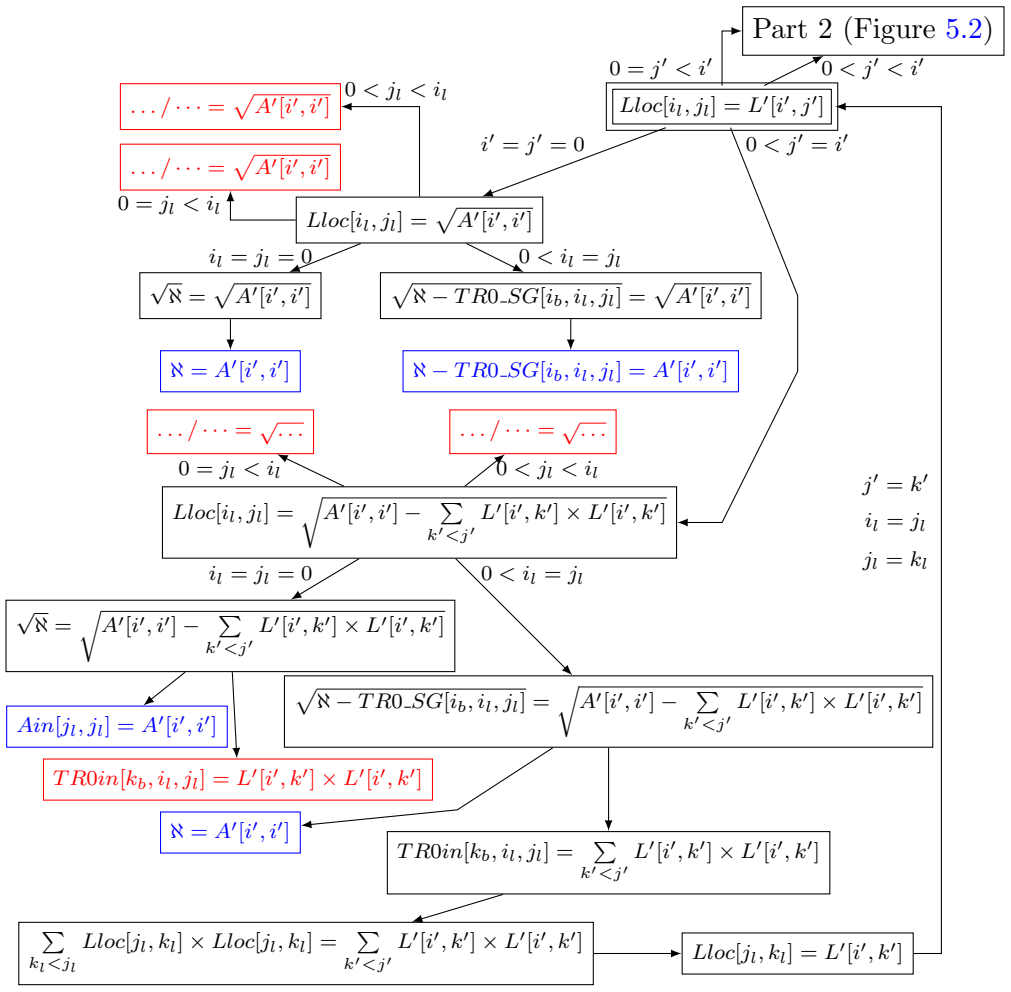


FIGURE 5.1: Equivalence automaton of Example 5.5 (Part 1), where a diagonal parametrized tile of a Cholesky computation is checked for a recursive Cholesky call. To reduce the space taken by the drawing of this automaton, we use the following shortcuts: $\aleph = Ain[j_l, j_l] - \sum_{k_b < i_b} TR0in[k_b, i_l, j_l]$

Let us examine the template-accept states. Only 4 of them are accessible, and their constraints are the following:

- $(\forall i_l = j_l = 0 = i' = j' = 0)$ $\aleph = A'[i', i']$ *(top-left element of A')*
- $(\forall 0 < i_l = j_l = i' = j')$ $\aleph = A'[i', i']$ *(other diagonal elements of A')*
- $(\forall 0 = j_l = j' < i_l = i')$ $\beth = A'[i', j']$ *(first column of non-diagonal elements of A')*
- $(\forall 0 < j_l = j' < i_l = i')$ $\beth = A'[i', j']$ *(other non-diagonal elements of A')*

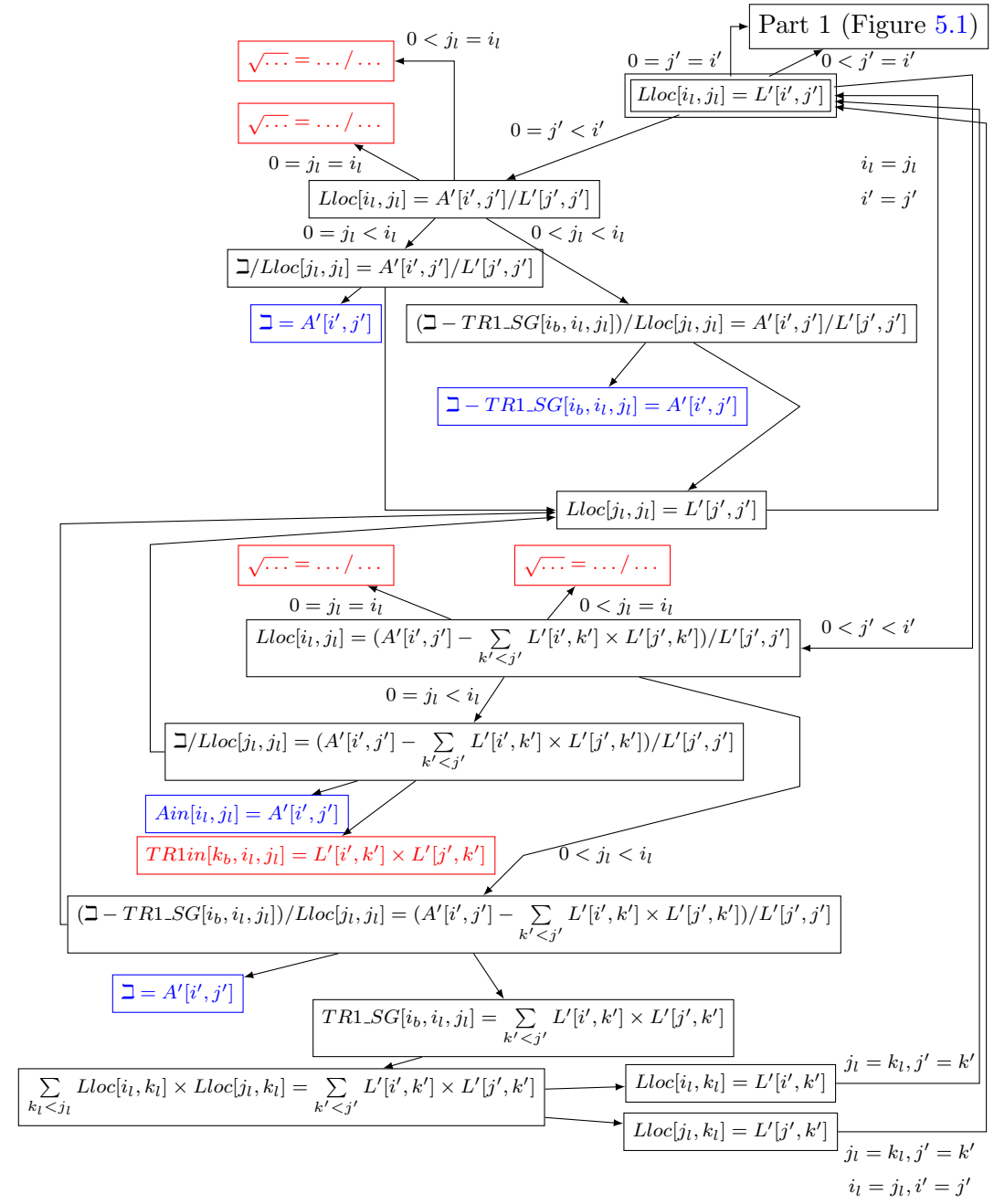


FIGURE 5.2: Equivalence automaton of Example 5.5 (Part 2), where a diagonal parametrized tile of a Cholesky computation is checked for a recursive Cholesky call. To reduce the space taken by the drawing of this automaton, we use the following shortcuts: $\beth = Ain[i_l, j_l] - \sum_{k_b < i_b} TR1in[k_b, i_l, j_l]$

where $\aleph = Ain[j_l, j_l] - \sum_{k_b<i_b} TR0in[k_b, i_l, j_l]$ *and* $\beth = Ain[i_l, j_l] - \sum_{k_b<i_b} TR1in[k_b, i_l, j_l]$.

Let us determine the value of the template input A'*: there is 4 associated constraints, but each one of them are concerning disjoints portions of* A'*. Therefore, we can simply take as the template input:*

$$(\forall j' = i' = 0)\ A'[i', j'] = Ain[0, 0] - \sum_{k_b<i_b} TR0in[k_b, i_l, j_l]$$
$$(\forall 0 < j' = i')\ A'[i', j'] = Ain[j', j'] - \sum_{k_b<i_b} TR0in[k_b, i_l, j_l]$$
$$(\forall 0 = j' < i)\ A'[i', j'] = Ain[i', 0] - \sum_{k_b<i_b} TR1in[k_b, i_l, j_l]$$
$$(\forall 0 < j' < i)\ A'[i', j'] = Ain[i', j'] - \sum_{k_b<i_b} TR1in[k_b, i_l, j_l]$$

Therefore, we conclude that the template matches the program, and we just have recognize a recursive call to a smaller Cholesky at the level of the diagonal blocks of a Cholesky computation.

5.4 Managing semantic properties

In Section 5.2, we described a template matching algorithm, based on Barthou's equivalence algorithm (cf Sections 2.4 and 5.1). Both algorithms do not consider any semantic properties. In this section, we show how to extend the template matching algorithm to deal with the common semantic properties usually encountered in a linear algebra computation.

The computation considered in our template library are linear algebraic operations, whose data belongs to a ring $(\mathbb{R}, +, \times)$. Because of the algebraic properties of this ring, given a program, there exist several variations which computes the same result. For example, in Example 5.5, we compare $\left(Ain[j_l, j_l] - \sum_{k_b<i_b} TR0in[k_b, i_l, j_l]\right) - TR0\_SG[i_b, i_l, j_l]$ with $\left(Ain[j_l, j_l] - \sum_{k_b<i_b} TR1in[k_b, i_l, j_l]\right) - TR1\_SG[i_b, i_l, j_l]$, but these three terms might be reordered differently (using the associativity and commutativity properties of the addition). If we do not take account of these algebraic property, the slightest variation of the computation will make the template recognition algorithm fails.

In this section, we show how to deal with some of the semantic properties encountered for the linear algebra operations, in order to improve the capability of our template recognition algorithm. All of these properties, other than associativity, commutativity and distributivity, are managed through a set of rewriting rules. The implementation of our templates are already normalized according to these rules, and the compared program are normalized through these rules. The associativity and commutativity properties are managed within the template recognition algorithm, instead of being a preprocessing step.

Managing semantic properties through rewriting rules We deal with most of the semantic properties by using rewriting rules, to be applied to both the program and the template before performing the template recognition algorithm. Because our templates are stored in a library, we can apply this preprocessing step once and for all. The rewriting rules are the following:

- Neutral element: we remove the useless contribution
  - $0 + A \to A, \quad A + 0 \to A$
  - $1.A \to A, \quad A.1 \to A$
- Annihilator element: we propagate it
  - $0.A \to 0$
- Inverse: we explicit the addition/multiplication
  - $A - B \to A + (-B)$
  - $A/B \to A.(1/B)$
- Reduction over a single element: we remove the reduction
  - $\sum_{k=f(\vec{i},\vec{p})} Expr[k] \to Expr[f(\vec{i},\vec{p})]$, where $f(\vec{i},\vec{p})$ is an affine function of the surrounding indices $\vec{i}$ and parameters $\vec{p}$.

These rules allows us to partially normalize the expression of a template or program, to allow our template recognition algorithm to recognize equivalent pattern, while taking these algebraic properties into account. Note that all of these rules are local modifications of the expression syntactic tree, thus can be applied easily.

Distributivity management We cannot deal with the distributivity semantic rule by using a rewriting rule. Indeed, we have the choice of either distributing ($A \times (B+C) \rightarrow A \times B + A \times C$) or factorizing ($A \times B + A \times C \rightarrow A \times (B+C)$) the terms. However, each case, we might prevent the other template from being recognized.

Indeed, if our program is $Out = (A+B) \times C$ and our template $Out' = I'_1 \times I'_2$, distributing C over $(A+B)$ prevents this template to be matched. Likewise, if our program is $Out = A \times C + B \times C$ and our template $Out' = A' + B'$, factorizing C prevents this template to be matched. Thus, forcing either way through a rewriting rule might hurt the recognition process.

In our context, we partially solve this problem by creating multiple versions of the template in which the distributivity property might apply: one in which the terms are factorized, one in which the terms are distributed.

Associativity and commutativity management We deal with associativity and commutativity rules by generating several variants of the template equivalence automaton, during the template recognition algorithm. For example, given a state of a template equivalence automaton $A + B = A' + B'$, we can match A with either A' or B' (resp. B with either B' or A'). Therefore, we generate two versions of the automaton: one in which the state $A + B = A' + B'$ leads to the states $A = A'$ and $B = B'$ through a computation rule, and another one in which this state leads to the states $A = B'$ and $B = A'$ through a computation rule (corresponding to the choice we are taking).

Let us first consider a state, during the construction of the template equivalence automaton, comparing two summations in which no term is a reduction:

$$\text{SExpr}_1 \oplus \cdots \oplus \text{SExpr}_k = \text{SExpr'}_1 \oplus \cdots \oplus \text{SExpr'}_{k'}$$

The main idea is that a term $\text{SExpr'}_{i'}$ on the template (right) side is mapped to one or many term(s) SExpr_i on the program (left) side. Therefore, if we have fewer terms on the program side than on the template side (i.e., if $k < k'$), we cannot match a term on the template side with a term on the program side, thus fall back into the default strategy of considering the operator as non-commutative and non-associative.

If we have at least as many terms on the program side as in the template side, then we can associate at least one term of the program side to the template side. Therefore, we generate all possible combinations and create the corresponding automaton for each combination. If the number of terms in each side is equal, it amounts to considering all the permutations [80].

The maximum number of terms in a summation does not exceed the maximum number of terms in a summation inside the input program or template and is, in practice, reasonably small. Thus, the number of automaton generated stays reasonable in practice.

This method of managing associativity and commutativity is not perfect. For example, one limitation is that, once we pick a combination, the choice is fixed once for all for this automaton, even if we encounter exactly the same state later. Indeed, when we try to add a new state to the template equivalence automaton, we check first if the state already exists, and, if it does, we reuse this existing state (this is the mechanism which allows us to have loops inside the template equivalence automaton). Therefore, if we have a comparison between two summations inside a loop of the template equivalence automaton, because the states were created the first time they were encountered, the choice made the first time cannot be changed. However, in the context of recognizing linear algebra operations, our method is enough.

In the general case, let us consider a state during the construction of the template equivalence automaton, comparing two summations (for any associative and commutative binary operator),

some terms being potentially reductions:

$$\begin{aligned}\mathrm{SExpr}_1 \oplus \cdots \oplus \mathrm{SExpr}_k \oplus \bigoplus \mathrm{SExprRed}_1 \oplus \cdots \oplus \bigoplus \mathrm{SExprRed}_l = \\ \mathrm{SExpr'}_1 \oplus \cdots \oplus \mathrm{SExpr'}_k \oplus \bigoplus \mathrm{SExprRed'}_1 \oplus \cdots \oplus \bigoplus \mathrm{SExprRed'}_l\end{aligned}$$

A reduction can be viewed as the summation over a parametric number of terms, therefore, the natural extension of the previous strategy consist on mapping any term of the template side (including a specific term inside a reduction) to one or many term(s) of the program side. In particular, this allows potential permutations of the summation order in the reductions. This idea was applied for an equivalence checking algorithm in order to manage reduction by Iooss and al. [34].

In our case, in order to simplify the template equivalence algorithm (and to avoid inferring a suitable permutation), we choose not to exploit potential permutations of the order of summation inside a reduction. It means that a reduction $\bigoplus \mathrm{SExprRed}_k$ is considered as some kind of unary operator. Also, when comparing two reductions, we need to check that the number of terms summed is the same. This can introduce some constraints on the parameters of the template (for example, if we compare a reduction over N terms in the program side with a reduction over N' terms in the template side, we must have $N = N'$).

A reduction term of the template side ($\mathrm{SExprRed'}_{i'}$) must be mapped to a single reduction term on the program side ($\mathrm{SExprRed}_i$). A non-reduction term of the template side can be mapped to any combination of terms on the program side. Therefore, to be able to apply this strategy, we need at least as many reductions on the program side as in the template side, and the total number of terms on the program side must be greater than or equal to the number of terms on the template side. We generate all the possible combinations, then generate one version of the template equivalence automaton per combination.

5.5 Experimental validation

In this section, we evaluate the scalability of our implementation of the template recognition algorithm described previously in this section. The implementation was done in Java, using the *AlphaZ* compiler framework [89]. The Integer Set Library (isl [79]) was used in order to perform the transitive closure.

Our set of test cases consist in the examples we have developed in the previous sections, plus the following additional template recognition problems:

- *Matmult*: compares a matrix multiplication computation (with reduction, i.e., $(\forall i, j)$ $C[i, j] = \sum_k A[i, k] * B[k, j]$) with a matrix multiplication template containing the same equation. This is a simple test case with a reduction to be managed.
- *Cholesky_Lbl_Tile1*: the program corresponds to one of the subsystems we obtain after applying the monoparametric tiling transformation with outlining on a Cholesky computation (see the light blue tiles of the left column in Figure 4.15, Page 112). The compared template corresponds to the linear algebra computation $C \leftarrow B.U^{-1}$ (`xTRSM` in BLAS).
- *Cholesky Commutation*: the computation is the same as the one of Example 5.5. However, the order of the summations was changed.

We run our experiments on a machine with an Intel Xeon E5-1650 CPU with 12 cores running at 1.6 GHz (max speed at 3.8GHz), and 31GB of memory. Figure 5.3 reports the time taken by each step of the algorithm. We also report the number of equivalence automata built during step 1 (the multiple versions being caused by the associativity and commutativity properties). We report the number of template equivalence automata we have considered in Step 2 and 3, until a matching was found, or all the automata were checked.

The steps 1 (equivalence automata construction) and 2 (constraint extraction) are the main contributors toward the total time. About the equivalence automata construction step of the algorithm, this step takes a lot of time when the number of variants is large. In Chapter 6, we will use this algorithm intensively in order to identify linear algebra operations inside a computation,

Template Matching Problem	Step 1 - Time Automaton Construction	Automata Considered /Automaton Built	Step 2 Extracting Constraints	Step 3 Determining Inputs	Number of States of the Automata Considered	Total Time
Example 5.1 Page 125 (Simple)	49	1/2	20	9	5	82
Example 5.2 Page 126 (NMatch)	41	2/2	23	89	12	156
Example 5.3 Page 127 (Unroll)	45	1/2	20	78	5	146
Matmult	58	1/2	35	8	10	104
Example 5.4 Page 128 (Reduction)	317	1/2	260	6	35	585
Example 5.5 Page 132 (Cholesky Tile2)	617	1/4	371	15	80	1008
Cholesky Lbl_Tile1	1806	1/28	628	24	41	2461
Example 5.5 Modified (Cholesky Comm)	7406	4/9	2460	18	396	9884

FIGURE 5.3: Experimental validation of our template recognition algorithm. The times are in milliseconds (ms). Because Step 1 is generating a list of automata (corresponding to several version of matching, due to the associativity and commutativity properties), we consider them one by one during Step 2 and 3, until a matching is found (or all of them are considered). The number of state of the automata considered are the sum of the number of states of the automata on which we went through step 2 and 3. The times reported for Step 2 (resp. 3) are the sums of the time spent in each Step 2 (resp. 3) phase, for each automata considered

and some of the instances of the template equivalence algorithm reach the hundreds of automata built. In the constraint extraction step, the most expensive operation is the transitive closure.

5.6 Discussion

Equivalence of reduction We have proposed [34] an extension to Barthou's equivalence algorithm to manage the associativity and commutativity properties of reductions. Because of

the properties of the reduction operator, the terms accumulated might be in a different order. Hence, the main challenge in this extension is to find a mapping between the terms of two compared reductions, so that we can conclude for equivalence or not.

The extension is performed in the following manner:

- *Equivalence automaton construction:* we add a new rule to manage reductions, called *Decompose Reduce*.

$$\boxed{\bigoplus_{\pi(\vec{k})=\vec{i}} E[\vec{k}] = \bigoplus_{\pi'(\vec{k'})=\vec{i'}} E'[\vec{k'}]}$$
$$\downarrow \sigma(\vec{k}) = \vec{k'}$$
$$\boxed{E[\vec{k}] = E'[\vec{k'}]}$$

  The idea of this rule is to map every instance of the left reduction $E[\vec{k}]$ to an equivalent instance $E'[\vec{k'}]$ on the right reduction, such that these two instances are equivalent. In other words, if we manage to find a bijection σ between the instances $\vec{k}$ of the left reduction and the occurrences $\vec{k'}$ of the right reduction such that $E[\vec{k}]$ is equivalent to $E'[\vec{k'}]$, then both reductions are equivalent. During the equivalence automaton construction step, we leave σ as a symbolic function (it does not impact the construction of the rest of the automaton). However, we still need to prove the existence of such σ, and the rest of the algorithm will focus on inferring it.

  Because this rule is based on a bijection which associates exactly one instance from the left reduction to another from the right reduction, we cannot manage situations where a left-instance must be mapped to the sum of several right-instances (or vice versa). In such situations, we will not be able to find a correct σ and we will be unable to conclude if both reductions are equivalent or not. However, in the situation of our transformation, this case should not happen.

- *Derivation of the mapping σ:* Once the equivalence automaton is built, if we did encounter a reduction, we need to prove the existence of σ, the bijection which associates equivalent terms from both reductions. We do this constructively, by inferring it, from the equivalence

automaton (which contains all the information needed). The inference algorithm consists of 3 steps:

- Extracting the constraints on σ: this step is just the computation of accessibility relations.
- Rearranging the constraints in order to obtain partial bijection $\tilde{\sigma}_i$. A partial bijection is a bijection which is defined only on a subset of a domain. The constraints extracted on σ have a special form: the only constraints in which we have indices from both programs are equalities (i.e., the indices of both programs do not mix in our constraints, except for some equality constraints). Using this property, we can transform our constraints into partial bijections $\tilde{\sigma}$.
- Combining the partial bijections $\tilde{\sigma}$ into a full bijection σ, which is our mapping. This problem is an instance of the *bipartite graph perfect matching* problem, over a particular kind of graph: the set of nodes of this graph corresponds to all the points of the antecedent and the image domain, and the edges corresponds to the partial bijections. Thus, this graph has a parametrized number of nodes, but has only a finite number of "type" of edges (one type for each partial bijections). We have proposed [34] several heuristics to find a perfect matching on such a graph: a greedy algorithm and one inspired of the augmenting path algorithm, which solves the perfect matching problem for finite graphs.

More details about this adaptation of the equivalence algorithm can be found in our paper [34]. This work can probably be extended to a template equivalence algorithm which manages the associativity and commutativity properties of reductions (like what we did in Section 5.2 with the original Bathou's equivalence algorithm). However, we will have to infer both the expression corresponding to the inputs of the template, and the bijection σ at the same time.

Moreover, in the context of our work, this level of flexibility for the associativity and commutativity properties of reduction operators is not needed. Indeed, in practice, we only need these properties to cut a reduction according to the tiling considered. This is already done automatically during the monoparametric tiling transformation (see Section 4.3).

Adjusting the domain of output variables As shown in Algorithm 1 Page 124, we extract some constraints between the parameters of the program and the template through the domain of output variables. This is done by comparing the domain of the output of the program with the domain of the output of the template, and by deducing the constraints on both sets of parameters to make them match.

During the application of our template detection framework, we might encounter subsystems which are parts of a larger linear algebra operation. For example, we can have a subsystem whose output is the strict lower triangular part of the result of a matrix multiplication between two square $N \times N$ matrices A and B:

$$(\forall 0 \leq j < i < N)\ C[i,j] = \sum_{0 \leq k < N} A[i,k] \times B[k,j]$$

If we try to compare this subsystem with the matrix multiplication template, because the domain of the output of the subsystem has a triangular shape and the domain of the template a rectangular shape, we will conclude that the subsystem does not match the template.

A first possible option to fix that issue is to create a new template per output shape (for example, a matrix multiplication template with a lower triangular output, then another one with an upper triangular output). However, this option forces us to duplicate many templates, which will slow down the recognition process.

Another option is to adapt the template recognition algorithm to allow the inclusion of the domain of the output variable of the subsystem in the domain of the output variable of the template, instead of an equality. However, such an extension causes the algorithm to fail when we try to fix the actual value of the template. For example, we would be able to match a matrix multiplication between two $N \times N$ matrices with a matrix multiplication template between a $N' \times N$ matrix and a $N \times N'$ matrix, for $N' > N$. Thus, taking the maximal value of N' does not maximize the amount of computation matched anymore, but the number of useless computation on such matching.

Finally, the option we chose is to extend the output domain of the subsystem in order to have a rectangular shape. If the equations of the output variable are valid for the new part of the domain of the output variable, we reuse them. Else, we add a new equation which sets the domain of the output variable to 0 in this new part. For example, if we consider the triangular subset of a matrix multiplication equation we considered previously, because the expression $\sum_{0 \leq k < N} A[i,k] \times B[k,j]$ can be defined other the whole square domain $\{i,j | 0 \leq i < N, 0 \leq j < N\}$, we can extend this equation before comparing the subsystem with the matrix multiplication template.

Using a similar reasoning, if the output domain of a subsystem contains equalities which reduce its dimensionality (for example, $\{i,j \mid i = j \dots\}$), we transform it to make it full dimensional (for the last example, we transform the 2D domain into a 1D domain) and adapt the corresponding equations.

Chapter 6

Recognizing subcomputations

In this chapter, we present our linear algebra subcomputation recognition framework. This framework is based on the contributions presented in the previous chapters, i.e., the monoparametric tiling transformation from Chapter 3 and 4, and the template recognition algorithm from Chapter 5. We present the remaining pieces in this chapter.

The main idea is to first partition the computation into tiles, using the monoparametric tiling transformation, then try to recognize the computation of each tile as a combination of linear algebra operators. These operators are listed in a library of template, which is inspired by the BLAS specification [46]. We present this library in Section 6.1.

Then, we present the structure of our framework in Section 6.2, and apply it to various linear algebra and non-linear algebra applications in Section 6.3. We conclude this chapter with several additional remarks in Section 6.4.

6.1 Template library

In this section, we present our library of linear algebra templates. Starting from the BLAS specification, we justify our design choices, which aim at minimizing the time spent to search for a matching template.

The BLAS specification The operations of the BLAS specification are classified into 3 levels, depending on the data structure returned. The output of a level 1 operator is a scalar, the output of a level 2 operator is a vector and that of a level 3 a matrix. Each operations have up to 4 variants, depending on the data type of the structure returned (single precision, double precision, complex and double precision complex). We will focus only on the double precision variant, but our approach can easily be extended to any other data types.

The list of operations and their names are described in Figure 6.1. Notice that some of these operations (such as DGEMM) overwrite their inputs, i.e. are inplace, which is not allowed in our program representation. Thus, we adapt these operations to add an additional copy and have "single-assignment" templates. A simple post-processing can be applied to check if this copy is necessary. Some templates (such as DSWAP or DCOPY) do not make sense in a such single-assignment context, and are removed.

For example, if we consider DTRMM, the operation we will consider instead is $C \leftarrow \alpha.L^X.B$, which corresponds to first copying the matrix B into the matrix C, then the in-place operation DTRMM of BLAS. If the matrix B is not used afterward, the copy can be skipped.

Reducing the number of templates In our template recognition algorithm, we will deal with the associativity and commutativity properties of binary operators. Other algebraic properties (such that distributivity of an operator over another, absorptive and neutral elements) are not managed by the template recognition algorithm. Therefore, in some BLAS operations, we have a special case when $\alpha, \beta = 1, 0$ or -1.

For example, for $\alpha = 1$, DGEMM becomes $C \leftarrow A^X.B^X$ and its computation has one multiplication less than the same operation when $\alpha = 2$, for example. Thus, to deal with the fact that 1 is neutral for the multiplication, we need to separate (at least) the case where $\alpha = 1$ and $\alpha \neq 1$ into two different templates.

To reduce the number of templates, we assume that $\alpha = 1$ everywhere, and add the operations $C \leftarrow \alpha.A$ (where $\alpha \neq 1$). This allows us to split the operation into 2 operations (one which contains the main matrix multiplication operation, and the other which contains the scalar

Level 1 BLAS:

- DSWAP : $x \leftrightarrow y$
- DSCAL : $x \leftarrow \alpha.x$
- DCOPY : $y \leftarrow x$
- DAXPY : $y \leftarrow \alpha.x + y$
- DDOT : $\alpha \leftarrow \vec{x}^T.\vec{y}$

Level 2 BLAS:

- DGEMV : $\vec{y} \leftarrow \alpha.A.\vec{x} + \beta.\vec{y}$
- DSYMV : $\vec{y} \leftarrow \alpha.S.\vec{x} + \beta.\vec{y}$ where S is symmetric
- DTRMV : $\vec{y} \leftarrow L^X.\vec{x}$ where L is lower-triangular
  $\vec{y} \leftarrow U^X.\vec{x}$ where U is upper-triangular
- DTRSV : $\vec{y} \leftarrow L^{-X}.\vec{x}$ where L is lower-triangular
  $\vec{y} \leftarrow U^{-X}.\vec{x}$ where U is upper-triangular
- DGER : $A \leftarrow \alpha.\vec{x}.\vec{y}^T + A$
- DSYR : $A \leftarrow \alpha.\vec{x}.\vec{x}^T + A$
- DSYR2 : $A \leftarrow \alpha.(\vec{x}.\vec{y}^T + \vec{y}.\vec{x}^T) + A$

Level 3 BLAS:

- DGEMM : $C \leftarrow \alpha.A^X.B^X + \beta.C$
- DSYMM : $C \leftarrow \alpha.S.B + \beta.C$ or $C \leftarrow \alpha.B.S + \beta.C$ where S is symmetric
- DSYRK : $C \leftarrow \alpha.A.A^T + \beta.C$ or $C \leftarrow \alpha.A^T.A + \beta.C$
- DSYR2K : $C \leftarrow \alpha.(A.B^T + B.A^T) + \beta.C$
- DTRMM : $B \leftarrow \alpha.L^X.B$ or $B \leftarrow \alpha.B.L^X$ where L is lower-triangular
  $B \leftarrow \alpha.U^X.B$ or $B \leftarrow \alpha.B.U^X$ where U is upper-triangular
- DTRSM : $B \leftarrow \alpha.L^{-X}.B$ or $B \leftarrow \alpha.B.L^{-X}$ where L is lower-triangular
  $B \leftarrow \alpha.U^{-X}.B$ or $B \leftarrow \alpha.B.U^{-X}$ where U is upper-triangular

FIGURE 6.1: List of BLAS operations corresponding to linear algebra operations, for double-precision floating point. A lower case letter ($x, y, \alpha, \beta \dots$) denotes a *scalar*, a lower case letter with an arrow ($\vec{x}$, $\vec{y}$) denotes a vector, and an upper case letter ($A, B, C, \dots$) denotes a matrix. $A^X = A$ or A^T, and $A^{-X} = A^{-1}$ or A^{-T}. We ignore the different versions caused by the different memory storage

multiplication). A post-processing can be used to merge these two operations, if they are detected in succession, so that a single BLAS kernel can be used instead of two.

We also notice that BLAS has many variants of the same operation, depending on whether or not one of its argument is transposed. For example, for DGEMM, we have in total 4 variants ($C \leftarrow A.B$, $C \leftarrow A.B^T$, $C \leftarrow A^T.B$ and $C \leftarrow A^T.B^T$). To reduce the number of variants, we separate the transpose operation ($C \leftarrow A^T$) from the matrix multiplication ($C \leftarrow A.B$), and we will only have to consider a single variant of the template. Once again, a post-processing can merge the transpose operation with the matrix multiplication operation, if these operations are detected in succession.

The list of template operations we obtain after these simplification is described in Figure 6.2. In addition to these template, we consider the whole program (before tiling) as a potential template, in order to recognize some tiles as a recursive call on smaller instances.

Classification per scalar operations In order to recognize a system as a linear algebra operation, we consider each operation of the library independently and try to match it with the system. If none of the templates in the library match, then we conclude that the current system cannot benefit from any operation in our library. However, if the template library is big, going over it will take a lot of time.

In order to accelerate this process, we need to reduce the number of templates considered. One option is to classify the template of the library according to their corresponding scalar operation, i.e., the operation obtained when we assume that the size of the matrix and vector is 1. For example, if we consider DGEMM ($C \leftarrow A.B$), for matrix sizes of 1, we obtain a multiplication between 2 scalars a and b.

In our context, we compare the template to the computation of a tile of parametric size. When the size of this tile is 1^k, the computation performed is a scalar operation. If this operation is different from the corresponding scalar operation of a template, then there is no hope that the template matches. Therefore, by using this classification, we can immediately restrict the set of template which might match with a given tile.

Extra:

- Transpose: $C \leftarrow A^T$
- Scalar multiplication - vector : $\vec{y} \leftarrow \alpha.\vec{x}$ where $\alpha \notin \{0, 1\}$
- Scalar multiplication - matrix : $C \leftarrow \alpha.A$ where $\alpha \notin \{0, 1\}$
- Addition - vector : $\vec{y} \leftarrow \vec{x_1} + \vec{x_2}$
- Addition - matrix : $C \leftarrow A + B$
- Reduction - vector : $\vec{y} \leftarrow \sum_k \vec{x_k}$
- Reduction - matrix : $C \leftarrow \sum_k A_k$

Level 1:

- DSCAL : $y \leftarrow \alpha.x$
- DDOT : $\alpha \leftarrow \vec{x}^T.\vec{y}$

Level 2:

- DGEMV : $\vec{y} \leftarrow A.\vec{x}$
- DSYMV : $\vec{y} \leftarrow S.\vec{x}$ where S is symmetric
- DTRMV : $\vec{y} \leftarrow L.\vec{x}$ where L is lower-triangular
  $\vec{y} \leftarrow U.\vec{x}$ where U is upper-triangular
- DTRSV : $\vec{y} \leftarrow L^{-1}.\vec{x}$ where L is lower-triangular
  $\vec{y} \leftarrow U^{-1}.\vec{x}$ where U is upper-triangular
- DGER : $A \leftarrow \vec{x}.\vec{y}^T$
- DSYR : $A \leftarrow \vec{x}.\vec{x}^T$
- DSYR2 : $A \leftarrow \vec{x}.\vec{y}^T + \vec{y}.\vec{x}^T$

Level 3:

- DGEMM : $C \leftarrow A.B$
- DSYMM : $C \leftarrow S.B$ or $C \leftarrow B.S$ where S is symmetric
- DSYRK : $C \leftarrow A.A^T$
- DSYR2K : $C \leftarrow A.B^T + B.A^T$
- DTRMM : $C \leftarrow L.B$ or $C \leftarrow B.L$ where L is lower-triangular
  $C \leftarrow U.B$ or $C \leftarrow B.U$ where U is upper-triangular
- DTRSM : $C \leftarrow L^{-1}.B$ or $C \leftarrow B.L^{-1}$ where L is lower-triangular
  $C \leftarrow U^{-1}.B$ or $C \leftarrow B.U^{-1}$ where U is upper-triangular

FIGURE 6.2: List of templates in our library, after simplification

Order of template comparison We notice that some templates are actually generalization of others templates. For example, DSYMM is a special case of DGEMM (which means that the template DSYMM can be considered as an instance of the template DGEMM for some specific inputs). In order to find the most specialized operation, the recognition framework considers the most specialized one first, i.e., we try to match DSYMM before DGEMM.

This leads us to the list of templates described in Figure 6.3, classified by scalar operations and number of dimensions of the output and ordered from the most specialized one to the most general.

Note that the transpose operation corresponds to a scalar "no operation". Therefore, it might happen anytime we have a matrix. Thus, if no operation is recognized after a first pass, for any scalar operation, we apply a "transpose" and try to recognize a new operation following. If no operation is recognized after that, we conclude that the considered system does not correspond to any linear algebra operation we have in our library.

In addition, we add the following templates to our library. They do not appear in BLAS, but occur in some applications:

- Point-to-point multiplication (resp. division): the equation of the template is $C[i,j] = A[i,j] \times B[i,j]$ (resp. $C[i,j] = A[i,j]/B[i,j]$).
- Diagonal matrix multiplication: this template is a specialization of a matrix multiplication: its output is a vector corresponding to the diagonal of the output matrix. Its equation is $y[i] = \sum_k A[i,k] \times B[k,i]$.
- Sum of triangular reduction: the computation of this template is $C[i,j] = A[i,j] + \sum_{0 \leq k < j} L[i,j,k]$.

- Scalar output:
  - (×) DSCAL : $z \leftarrow \alpha.x$
  - (×) DDOT : $z \leftarrow \vec{x}^T.\vec{y}$
- Vector output:
  - (×) DSYMV : $\vec{y} \leftarrow S.\vec{x}$ where S is symmetric
  - (×) DTRMV : $\vec{y} \leftarrow L.\vec{x}$ where L is lower-triangular
    $\vec{y} \leftarrow U.\vec{x}$ where U is upper-triangular
  - (×) Scalar multiplication - vector : $\vec{y} \leftarrow \alpha.\vec{x}$ where $\alpha \notin \{0, 1\}$
  - (×) DGEMV : $\vec{y} \leftarrow A.\vec{x}$
  - (div) DTRSV : $\vec{y} \leftarrow L^{-1}.\vec{x}$ where L is lower-triangular
    $\vec{y} \leftarrow U^{-1}.\vec{x}$ where U is upper-triangular
  - (+) Addition - vector : $\vec{y} \leftarrow \vec{x_1} + \vec{x_2}$
  - (+) Reduction - vector : $\vec{y} \leftarrow \sum_k \vec{x_k}$
- Matrix output:
  - (×) DSYRK : $C \leftarrow A.A^T$
  - (×) DSYMM : $C \leftarrow S.B$ or $C \leftarrow B.S$ where S is symmetric
  - (×) DTRMM : $C \leftarrow L.B$ or $C \leftarrow B.L$ where L is lower-triangular
    $C \leftarrow U.B$ or $C \leftarrow B.U$ where U is upper-triangular
  - (×) DSYR : $A \leftarrow \vec{x}.\vec{x}^T$
  - (×) DGER : $A \leftarrow \vec{x}.\vec{y}^T$
  - (×) Scalar multiplication - matrix : $C \leftarrow \alpha.A$ where $\alpha \notin \{0, 1\}$
  - (×) DGEMM : $C \leftarrow A.B$
  - (div) Inverse of a triangular matrix: L^{-1}
  - (div) DTRSM : $C \leftarrow L^{-1}.B$ or $C \leftarrow B.L^{-1}$ where L is lower-triangular
    $C \leftarrow U^{-1}.B$ or $C \leftarrow B.U^{-1}$ where U is upper-triangular
  - (+) DSYR2K : $C \leftarrow A.B^T + B.A^T$
  - (+) DSYR2 : $A \leftarrow \vec{x}.\vec{y}^T + \vec{y}.\vec{x}^T$
  - (+) Addition - matrix : $C \leftarrow A + B$
  - (+) Reduction - matrix : $C \leftarrow \sum_k A_k$
  - (nothing) Transpose: $C \leftarrow A^T$

FIGURE 6.3: Final list of template, classified by scalar operations and number of dimensions of the output, and ordered

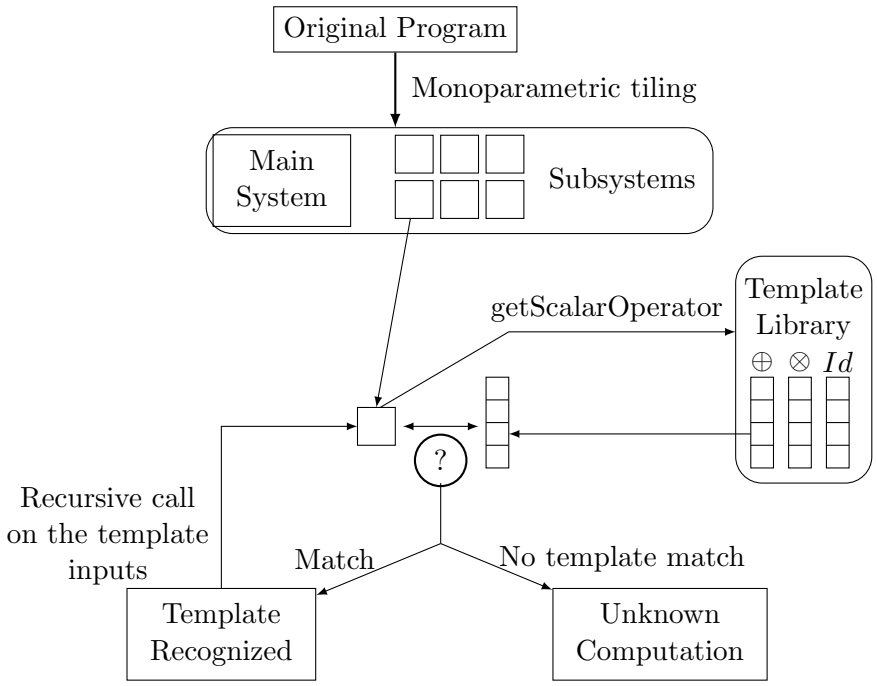


FIGURE 6.4: Template recognition procedure: we first apply the monoparametric tiling transformation, then we consider each produced subsystem independently. The template library is classified according to the corresponding scalar operation of each template. Each subsystem is analyzed in order to detect its scalar operation. We retrieve the list of template corresponding to this scalar operation, from the template library. Then, we compare the subsystem with each template of this list, one by one. Two situations might occur: either none of the templates match, and the computation is considered as unknown, or a template matches. Then, we check the inputs of this template, and recursively call the template matching algorithm on each input of the template that is not an input of the subsystem.

6.2 Linear algebra operation recognition framework

In this section, we describe how we combine the monoparametric tiling transformation (cf Chapter 4) with our template matching algorithm (cf Section 5.2) in order to recognize instances of templates from our template library (cf Section 6.1). The whole process is shown in Figure 6.4.

As a preprocessing step, we apply to the original program the rewriting rules we have presented in Section 5.4, which allows us to manage most of the algebraic properties of a ring. This is more efficient to do it before the monoparametric tiling transformation, such that the changes propagate over all the subsystems.

Monoparametric tiling The first step of our algorithm is to apply the monoparametric tiling transformation. We use square tiles for every variable and place each variable of the program (i.e. identity ratios), and assign a single variable per tile group whenever possible. This transformation produces a main program and a list of subsystems. The main program does not contain any computation, but each subsystem contain the computation of a tile. Thus, we consider each subsystem independently in the rest of the procedure.

In the original monoparametric tiling transformation, we compute the set of values needed by other tiles, in order to form the outputs of the subsystems. In particular, we classify the output data depending on which tile requires the information. However, in the context of template recognition, we want a single output per tiled variable for each subsystem, instead of spliting it into several output variables. Thus, as discussed in Section 4.4, we disable this feature in the context of template recognition.

Retrieving the list of templates Given a specific subsystem, we want first to identify its corresponding scalar operator (i.e., which operator the subsystem corresponds to, when the tile size parameter is equal to 1), so that we can select the corresponding category in our template library (cf *getScalarOperator* in Figure 6.4).

In order to determine the scalar operator, we extract the top-most operator of the operations leading to the output of the subsystem. Several situations might occurs:

- If a scalar operator is found and is managed by our template library, we return it.
- If no operator is found (for example, the output variable of the subsystem is a constant, or the copy of an input variable), or an operator which is not considered by the template library is found (e.g., a square root for a Cholesky computation), then we do not retrieve any template from our library.
- If we encounter a reduction, because the reductions inside a subsystem was created from a larger reduction of the original program, these reductions are accumulating over a parametric number of elements. Thus, when the tile size of a subsystem is set to 1, the

reduction accumulates over a single element for a tile size of 1, thus disappears. Therefore, we just ignore the corresponding reduction operator and continue the search inside the reduction. If no operator occurs inside the reduction, we take the operator of the reduction by default. Note that this strategy is not valid in general (for example, if we have a reduction over 3 elements). However, in our context, such a situation should not occur.

Moreover, we have to be careful about detected multiplication operators that are actually divisions (because of the rewriting rule $A/B \rightarrow A.(1/B)$ we applied to our original program).

Once we obtain the scalar operator of a subsystem, we combine it with the number of dimensions of the output to determine the corresponding *template category*, and to retrieve the corresponding list of templates from our library (cf Figure 6.3). If no operation was found, then this list of templates is empty.

We add at the beginning of our list of templates a special template called *recursive call*. The equations of this template are exactly the ones from the original program. This template allows us to identify the recursive call to our original program, on smaller instances. A typical example was shown for a Cholesky computation (in Figure 4.15 Page 112) where the top most operation in each diagonal block are smaller instances of a Cholesky.

If the domain of the output variable is two dimensional, then we have to deal with the *transpose* template. Because the transpose operation is idempotent (i.e. $(A^T)^T = A$), we prevent its template to be applied twice consecutively. Moreover, because this template does not have an associated scalar operators, we add it at the end of our template list.

Output of the procedure and recursion The output of our procedure is a *tree of templates*. Each node of this tree corresponds to a template, whose inputs are the children of this node. The leaves of the template tree are either an input of the program, a constant, or a non recognized computation.

Given a subsystem and a freshly retrieved list of templates, we start trying to match the subsystem with each template of the list, using our template recognition algorithm from Section 5.2. If the template does not match, we continue with the next template in the list. If the end of the list of templates is reached, then we return a tree with a single node corresponding to a non recognized computation.

If a template is matched to a subsystem, we build a node corresponding to this template. Then, we examine the expressions of the subsystem that correspond to the inputs of the template. For each of these expressions, if it is an input variable of the subsystem, or a constant, or a switch between input variable and contants, then we build the corresponding leaf and link it to the node of the recognized template. If the expression is more complicated, we build a new subsystem which corresponds to the reminder of the computation, and apply our procedure recursively on this new system. Then, we retrieve the produced tree of template and link it to the node of the recognized template.

Example 6.1. *To illustrate our procedure, let us apply it to a matrix multiplication computation. The original program is the following:*

$$(\forall 0 \leq i, j < N)\ C[i,j] = \sum_{0 \leq k < N} A[i,k] \times B[k,j];$$

where A and B are input variables, both defined over the domain $\{i, j | 0 \leq i, j < N\}$, and C is the output variable.

The preprocessing step to manage algebraic properties does not do anything. Then, we apply a monoparametric tiling transformation, only using the identity ratio. Because of the reduction, we obtain two subsystems: one corresponding to a small matrix multiplication, an another summing all the outputs of the small matrix multiplication to form the final result.

The equations of the first subsystem are:

$$(\forall 0 \leq i_l, j_l < b)\ TempRed[i_l, j_l] = \sum_{0 \leq k_l < b} Ain[i_l, k_l] \times Bin[k_l, j_l];$$

where TempRed is the output and Ain and Bin are the inputs of the subsystem (corresponding to the tiles $A[i_b, k_b]$ and $B[k_b, j_b]$ in the original program).

The equations of the second subsystem are:

$$(\forall 0 \leq i_l, j_l < b)\ C[i_l, j_l] = \sum_{0 \leq k_b < N_b} TempRedin[k_b, i_l, j_l];$$

where C is the output (corresponding to the tile $C[i_b, j_b]$ in the original program) and TempRed is the input of the subsystem (corresponding to the collection of results of the partial summation).

We start our procedure by examining the first subsystem, and try to determine its associated scalar operator. The first operator encountered is the one from the summation $\sum$, but, because it comes from a reduction, we ignore it. The next one is a multiplication. Therefore, we retrieve the list of templates corresponding to a multiplication for matrix in our template library. We append at the start of this list of templates the recursive call *template (thus, which is a matrix multiplication), and, because the output is two dimensional, the* transpose *template at the end.*

Then, we try to match the first subsystem with the selected templates. The first one (recursive call) matches, and the expressions corresponding to the inputs of the template are:

$$\begin{cases} A' \leftrightarrow Ain \\ B' \leftrightarrow Bin \end{cases}$$

Both of them are inputs of the subsystem, thus we do not have a recursive call. Thus, the procedure is done for the first subsystem.

We now consider the second subsystem. When considering the associated operation, we encounter a reduction, but no operation afterward. Thus, we retrieve the list of template corresponding to an addition for matrix, append at the start the recursive call *template, and at the end the* transpose *template.*

We try to match the first subsystem with the selected templates, and no template match until we end up on "Reduction - matrix" (summarized as $C \leftarrow \sum_k Ak$ in Figure 6.3). The expressions

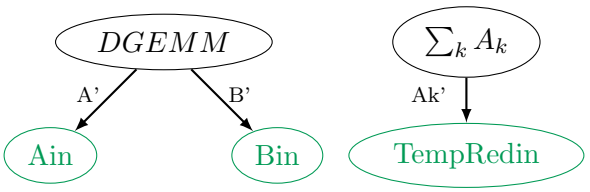


FIGURE 6.5: Returned template tree for the matrix multiplication computation. The green nodes correspond to the input of the template

corresponding to the inputs of the template are:

$$Ak' \leftrightarrow TempRedin$$

This is an inputs of the subsystem, thus we do not have a recursive call. No subsystem remains, thus the procedure ends. The returned template tree are shown in Figure 6.5

In the next section, we will present several examples of application of this procedure.

6.3 Applications

In the previous section, we described a framework that applies the monoparametric tiling transformation to a program, then applies recursively a template recognition algorithm to each generated subsystems independently. In this section, we present several experiments in order to validate this framework. In particular, we apply this framework to several programs, in order to check the scalability of our approach, then we study the amount of computation our framework is able to recognize as a template.

We will apply our framework to two kinds of applications in this chapter: linear algebra applications (which should be almost completely covered by the templates we recognize), and non-linear algebra applications (in which only some specific parts of the computation should be recognized as a template). We will first study the linear algebra applications (Symmetric Positive semi-Definite Matrix Inversion and Silvester Equations), then the non-linear algebra applications (Algebraic Path Problem and Mc Caskill).

Parameters: N
Inputs:
A, defined over $\{i,j|0 \leq i,j < N\}$
Local:
L, defined over $\{i,j|0 \leq j \leq i < N\}$
$InvL$, defined over $\{i,j|0 \leq j \leq i < N\}$
Output:
$InvA$, defined over $\{i,j|0 \leq i,j < N\}$

$(\forall i = j = 0)\ L[i,j] = \sqrt{A[i,i]};$ // Cholesky:$A = L.L^T$

$(\forall i = j > 0)\ L[i,j] = \sqrt{A[i,i] - \sum_{k<i} L[i,k] * L[i,k]};$

$(\forall i > j = 0)\ L[i,j] = A[i,j]/L[i,i];$

$(\forall i > j > 0)\ L[i,j] = \left(A[i,j] - \sum_{k<j} L[i,k] * L[j,k]\right)/L[i,i];$

$(\forall i = j \geq 0)\ InvL[i,j] = 1/L[i,i];$ // $InvL = L^{-1}$

$(\forall i > j \geq 0)\ \left(-\sum_{j \leq k < i} L[i,k] * InvL[k,j]\right)/L[i,i];$

$(\forall 0 \leq i,j < N)\ InvA[i,j] = \sum_k InvL[k,i] * InvL[k,j];$ // $InvA = InvL^T.InvL$

FIGURE 6.6: Original program for the Symmetric Positive semi-Definite Matrix Inversion. The input is a semi-definite positive square matrix A of size $N \times N$. This program is the composition of a Cholesky computation (whose result is L), followed by a triangular matrix inversion (whose result is $InvL$), and a transpose matrix multiplication (whose result is $InvA$, which is also the output of the program)

The experiments presented in this section were run on a machine with an Intel Xeon E5-1650 CPU with 12 cores running at 1.6 GHz (max speed at 3.8GHz), and 31GB of memory.

6.3.1 Dense Linear algebra applications

Symmetric Positive semi-Definite Matrix Inversion The first application we consider is called *Symmetric Positive semi-Definite Matrix Inversion* (SPDMI). The input is a symmetric semi-definite matrix A, i.e., a square matrix which can be decomposed as $A = L.L^T$, where L is a lower-triangular matrix. The output is the inverse of this matrix. This output is computed by using the Cholesky factorization algorithm on A to retrieve the lower triangular matrix L, then a triangular matrix inversion to compute L^{-1}, then a transpose matrix product to compute $A^{-1} = L^{-T}.L^{-1}$. The equations of such a program are described in Figure 6.6.

After analyzing the dependences, it is legal to tile separately the three variables L, $InvL$ and $InvA$, using a square monoparametric tiling.

After applying the monoparametric tiling transformation, we obtain in total 16 subsystems: 7 coming from the equations of L (including 3 from the two reductions), 4 coming from the equations of $InvL$ (including 2 from the reduction), and 5 coming from the equations of $InvA$ (including 4 from the reduction). The time taken by the monoparametric tiling transformation, plus some post-processing normalization steps (such as reducing the number of dimensions of some inputs and outputs) is about 5.7 seconds.

Across all subsystems, we perform 200 comparisons between a program and a template. We consider in total 429 equivalence automata (we count only the automata for which we extract some constraints, and not the total number of automata built), containing in total 9815 states. Also, 52 equivalence subproblems are considered. The total time taken by the whole process (including the monoparametric tiling transformation) is 439.3 seconds (about 7 minutes 19 seconds). This means that we spend in average about 2 seconds for each instance of template-match comparison.

In total, we have detected 27 templates in this computation (if we ignore the 3 "transpose" node that precedes a "non-recognized" node). The corresponding template trees are presented in Figure 6.7.

We managed to recognize completely the computation, except in 5 places:

- For the subsystems L_bl_Tile0 and L_bl_Tile2, these tiles correspond to the diagonal blocks of a Cholesky computation. Because we do not have a Cholesky template in our library, no operation is recognized.
- For the subsystem L_bl_Tile3 (corresponding to the dark blue part in Figure 4.15, Page 112), the two computations which are not recognized are both a switch between an input variable (for $i = 0$), and a sum of matrix ($C \leftarrow \sum A_k$, for $i > 0$). We do not have a corresponding template to recognize this kind of pattern.

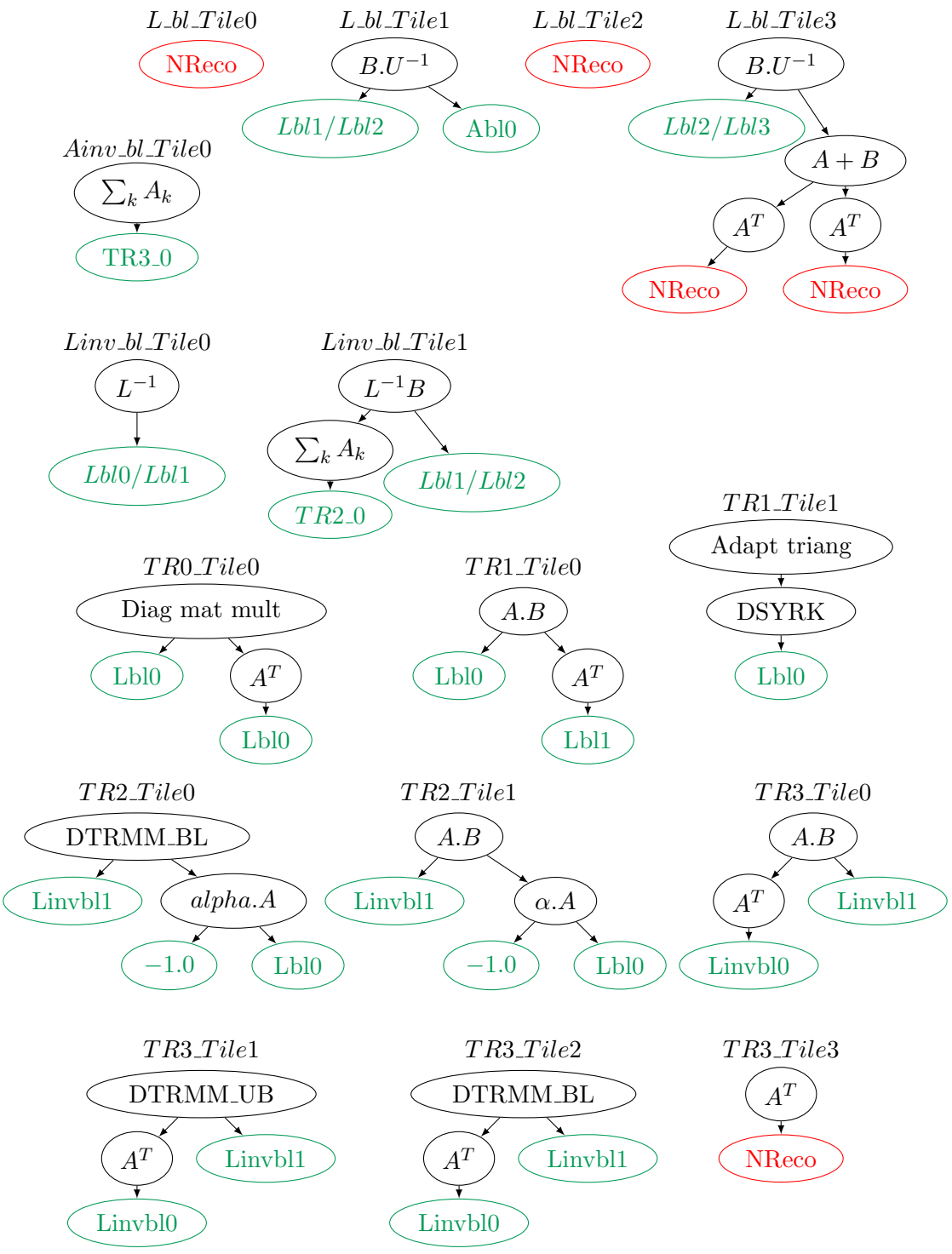


FIGURE 6.7: Output of our template recognition framework: trees of recognized templates for the SPDMI example. The nodes in green correspond to the input of the template, a constant, or a switch between inputs and constants. The nodes in red correspond to the non-recognized computation.

- For the subsystem $TR3\_Tile3$ (coming from the reduction inside the equation of $Linv$, and occurring only for the tile at $i_b = j_b = k_b$), its computation is:

$$(\forall 0 \leq (i,j) < N)\ Out[i,j] = \sum_{\substack{i \leq k < b \\ j \leq k}} Linvbl0[k,i] \times Linvbl0[k,j]$$

  Because of the bounds of the domain of the summation, none of our template matches.

Thus, we managed to match almost all the computation with our template library. Also, because there is only a quadratic number of tiles whose computation is not fully covered by templates, among a cubic number of tiles, the most frequently used parts of the computation were recognized.

Sylvester Equation Solver A *Sylvester equation* is an equation of the form $A.X + X.B = C$, where A, B and C are given square matrices, and X is an unknown square matrix.

We will explain the well-known algorithm to solve this equation, and then apply our template recognition framework to this. We will first show why there is no loss of generality if we assume that A and B are upper-triangular. We can simplify this equation by considering the Schur decomposition of the matrix A, i.e., we have $A = Q_A.U_A.Q_A^{-1}$, where U_A is upper-triangular and Q_A is a unitary matrix (i.e., $Q_A^{-1} = Q_A^H$, the conjugate of the transpose of A). Likewise, we consider the Schur decomposition of the matrix B: $B = Q_B.U_B.Q_B^{-1}$ where Q_B is a unitary matrix and U_B an upper-triangular matrix. By replacing A and B by their decomposition in the main equation, we obtain $U_A.(Q_A^{-1}.X.Q_B) + (Q_A^{-1}.X.Q_B).U_B = Q_A^{-1}.C.Q_B$. Thus, by setting $X' = Q_A^{-1}.X.Q_B$ and $C' = Q_A^{-1}.C.Q_B$, we obtain the following equation: $U_A.X' + X'.U_B = C'$.

The program which solves a Sylvester equation, when A and B are upper-triangular is the following:

$$
\begin{aligned}
&(\forall i = N-1, j = 0)\ X[i,j] = C[i,j]/(A[i,i] + B[j,j]);\\
&(\forall i = N-1, 0 < j < N)\ X[i,j] = \left(C[i,j] - \sum_{0 \le k < j} X[i,k] \times B[k,j]\right)/(A[i,i] + B[j,j]);\\
&(\forall 0 \le i < N-1, j = 0)\ X[i,j] = \left(C[i,j] - \sum_{i < k < N} A[i,k] \times X[k,j]\right)/(A[i,i] + B[j,j]);\\
&(\forall 0 \le i < N-1, 0 < j < N)\ X[i,j] = \Big(C[i,j] - \sum_{0 \le k < j} X[i,k] \times B[k,j]\\
&\qquad\qquad - \sum_{i < k < N} A[i,k] \times X[k,j]\Big)/(A[i,i] + B[j,j]);
\end{aligned}
$$

A square monoparametric tiling is legal: because all the dependences on X are increasing along the i dimension and decreasing along the j dimension, they satisfy the hyperplane condition for the legality of tiling.

After applying the monoparametric tiling transformation, we obtain in total 8 subsystems: 4 which compute the value of X, and one for each reductions of the program. The time taken by the monoparametric tiling transformation is about 6.6 seconds.

During the recognition process, we have an issue with 4 of the 8 subsystems (corresponding to X), for which the computation of a transitive closure takes a significant amount of time. Thus, we are forced to skip the recognition process for these subsystems and consider them as "not recognized".

Across all remaining subsystems, we perform 28 comparisons between a program and a template. We consider in total 148 equivalence automata (we count only the automata for which we extract some constraints, and not the total number of automata built), containing in total 1602 states. Also, 15 equivalence subproblems are considered. The total time taken by the whole process (including the monoparametric tiling transformation) is 104.7 seconds (about 1 minute 45 seconds). This means that we spend in average about 1.5 seconds for each instance of template-match comparison.

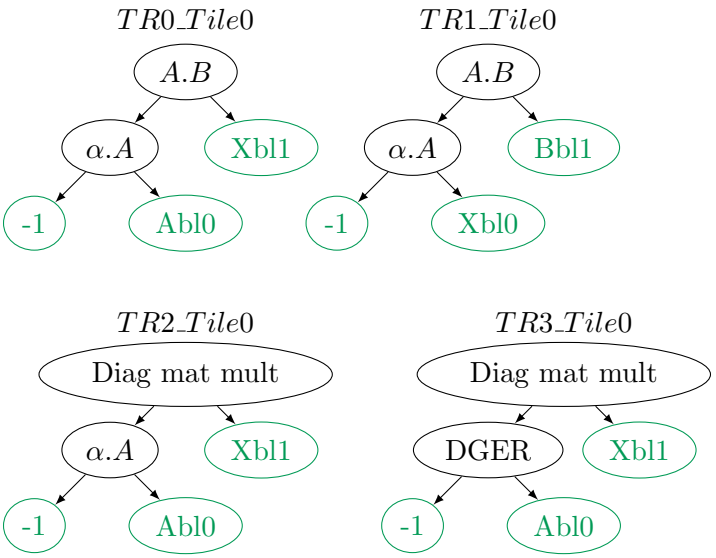


FIGURE 6.8: Output of our template recognition framework: trees of recognized templates for the Sylvester equation solver example. The nodes in green correspond to the input of the template, a constant, or a switch between inputs and constants. The nodes in red correspond to the non-recognized computation.

In total, we have detected 8 templates in this computation. The corresponding template trees are presented in Figure 6.8. These templates cover completely the subsystems created from reductions, which contains the majority of the computation of the program.

6.3.2 Applications outside of dense linear algebra

Algebraic Path Problem The Algebraic Path Problem (APP) is a graph algorithm which can be viewed as a generalization of the Floyd-Warshall algorithm. Its equations are the following:

$$
\begin{aligned}
&(\forall 0 \leq (i,j) < N) && Out[i,j] = F[i,j,N-1] \\
&(\forall 0 \leq (i,j) < N, k = -1) && F[i,j,k] = A[i,j] \\
&(\forall 0 \leq i = j = k < N) && F[i,j,k] = clos(F[k,k,k-1]) \\
&(\forall 0 \leq i = k < N, j \neq k) && F[i,j,k] = F[k,k,k] \times F[k,j,k-1] \\
&(\forall 0 \leq j = k < N, i \neq k) && F[i,j,k] = F[i,k,k-1] \times F[k,k,k] \\
&(\forall 0 \leq (i,j,k) < N, i \neq k, j \neq k) && F[i,j,k] = F[i,j,k-1] + (F[i,k,k] \times F[k,j,k-1])
\end{aligned}
$$

where A is an input variable, Out the output variable and $clos$ is a closure operator.

Let us explain the equations of this program. We can consider A as the adjacency matrix of a weighted directed graph (which has N nodes). The weight of a path is the product of the weight of the edges of this paths. Then, $Out[i,j]$ corresponds to the summation of the weight of all the paths starting from the node i and finishing on the node j. $F[i,j,k]$ represents the summation of the weights of all the paths from node i to node j, such that all the intermediate nodes of this path are the nodes 0 to k.

The closure operator manages the loops on the graph: indeed, the set of paths from k to k using all the nodes whose labels are bellow k ($F[k,k,k]$) can be decomposed as a succession of loops from k to k, using the nodes whose labels are below $k-1$. The multiplication operator can be viewed as a composition of paths. For example, if $i = k$, all the paths from i to j ($\neq i$) using the nodes whose labels are below k ($F[k,j,k]$) can be viewed as the combination of self-loops from i to i ($F[k,k,k]$), then a path from k to j, which is not using the node k again ($F[k,j,k-1]$). Likewise, the addition operator corresponds to a disjoint union of paths. We notice that if we take as a closure operator $min(x)$, and as a semi-ring $(min,+)$, we obtain exactly Floyd's algorithm, which computes the shortest path between all pairs of nodes. Likewise, this program can be used to compute accessibility relation inside a graph.

The equations of the APP do not contain any reductions. However, if we study the computation, we can recognize a reduction along the k axis. More precisely, if we analyze the computation needed to compute a given $Out[i,j]$, we first have a decreasing accumulation from $F[i,j,N-1]$ to $F[i,j,max(i,j)]$, then from $F[i,j,max(i,j)-1]$ to $F[i,j,min(i,j)]$, then from $F[i,j,min(i,j)-1]$ to $F[i,j,-1]$. We can arrange automatically the program to explicit this reduction.

Also, if we analyze the dependences of the program, each $F[i,j,k]$ are used exactly once, except the ones on the planes $i = k$ and $j = k$. Thus, we can replace the local variable F by the following local variables:

- $cross[i,j,k]$ defined for $i = k$ or $j = k$, and which corresponds to the special computations.

- $temp1[i,j]$ corresponding to $F[i,j,max(i,j)-1]$, and which is the top-most element of the middle reduction ($k = max(i,j)$ to $min(i,j)$) and defined for $i \neq j$ and $0 \leq k < N$.
- $temp2[i,j]$ corresponding to $F[i,j,min(i,j)-1]$, and which is the top-most element of the bottom reduction ($k = min(i,j) - 1$ to 0) and defined for $0 \leq (i,j,k) < N$.

In addition, in order to avoid unions of polyhedra in the domains of these variables, we split the variable $temp1$ into $temp1Maxi$ (for $i > j$) and $temp1Maxj$ (for $j > i$). Likewise, we split the variable $temp2$ into $temp2Mini$ (for $i < j$) and $temp2Minj$ (for $j \leq i$). The variable $cross$ is split in 5 fragments: $crossMiddle$ (for $i = j = k$), $crossUp$ (for $i = k < j$), $crossBottom$ (for $j < i = k$), $crossLeft$ (for $i < j = k$) and $crossRight$ (for $j = k < i$). The resulting program is shown in Figure 6.9.

Now, let us find a legal tiling for this program. By studying the self-dependences, we identify in total 6 tile groups:

1. Out,
2. $crossBottom$,
3. $crossUp$,
4. $temp1Maxi$,
5. $crossLeft$ and $temp1Maxj$,
6. $crossMiddle$, $crossRight$, $temp2Mini$ and $temp2Minj$.

Also, rectangular tiling is legal, thus there is no need to apply a change of basis on any of these variables beforehand.

After applying the monoparametric tiling transformation, we obtain in total 60 subsystems, 34 of them coming from reductions. The time taken by the monoparametric tiling transformation is about 20.5 seconds.

$$(\forall 0 \leq i = j < N)\ Out[i,j] = crossMiddle[i,i,i] + \sum_{i<l<N} crossLeft[i,l,l] \times temp1Maxi[l,j,l-1];$$
$$(\forall 0 \leq i < j < N)\ Out[i,j] = crossLeft[i,j,j] + \sum_{j<l<N} crossLeft[i,l,l] \times temp1Maxi[l,j,l-1];$$
$$(\forall 0 \leq j < i < N)\ Out[i,j] = crossBottom[i,j,j] + \sum_{i<l<N} crossLeft[i,l,l] \times temp1Maxi[l,j,l-1];$$

$$(\forall 0 \leq j < i < N, k = i-1)\ temp1Maxi[i,j,k] = crossRight[i,j,j] + \sum_{j<l<i} crossRight[i,l,l] \times temp1Maxi[l,j,l-1];$$
$$(\forall 0 \leq i < j < N, k = j-1)\ temp1Maxj[i,j,k] = crossUp[i,j,i] + \sum_{i<l<j} crossLeft[i,l,l] \times temp2Mini[l,j,l-1];$$

$$(\forall 0 \leq i < j < N, k = i-1)\ temp2Mini[i,j,k] = A[i,j] + \sum_{0\leq l<i} crossRight[i,l,l] \times temp2Mini[l,j,l-1];$$
$$(\forall 0 \leq j \leq i < N, k = i-1)\ temp2Minj[i,j,k] = A[i,j] + \sum_{0\leq l<j} crossRight[i,l,l] \times temp2Mini[l,j,l-1];$$

$$(\forall 0 \leq i = j = k < N)\ crossMiddle[i,j,k] = clos(temp2Minj[k,k,k-1]);$$
$$(\forall 0 \leq j = k < i < N)\ crossRight[i,j,k] = temp2Minj[i,k,k-1] \times crossMiddle[k,k,k];$$
$$(\forall 0 \leq i < j = k < N)\ crossLeft[i,j,k] = temp1Maxj[i,k,k-1] \times crossMiddle[k,k,k];$$
$$(\forall 0 \leq i = k < j < N)\ crossUp[i,j,k] = crossMiddle[k,k,k] \times temp2Mini[k,j,k-1];$$
$$(\forall 0 \leq j < i = k < N)\ crossBottom[i,j,k] = crossMiddle[k,k,k] \times temp1Maxi[k,j,k-1];$$

FIGURE 6.9: Equations of the APP program, after detecting the reductions and reorganizing the local variables. For concision, we do not consider the special equations which manages the case when a reduction sums over no element (for example, when $i = N-1$ in the first equation), and will just consider that the value of the reduction is 0.

Across all remaining subsystems, we perform 660 comparisons between a program and a template. We consider in total 698 equivalence automata, containing in total 13054 states. Also, 93 equivalence subproblems are considered. The total time taken by the whole process (including the monoparametric tiling transformation) is 1578.4 seconds (about 26 minutes 18 seconds). This means that we spend in average about 2.26 seconds for each instance of template-match comparison.

We detect in total 44 templates (without counting the 16 of them which are a "transpose" detected right before a non-recognized computation). The operations detected are mostly matrix

$$(\forall 0 \leq i < j-4 < N-4)\ Q[i,j] = 1 + \sum_{\substack{i < d \leq j-4 \\ d+4 \leq e \leq j}} Q[i,d-1] \times Qb[d,e];$$
$$(\forall 0 \leq i = j-4 < N-4)\ Q[i,j] = 1 + \sum_{i+4 \leq e \leq j} Qb[i,e];$$
$$(\forall 0 \leq j-4 < i \leq j < N-4)\ Q[i,j] = 1;$$

$$\begin{aligned}(\forall 0 \leq i < j-4 < N-4)\ Qm2[i,j] = & \sum_{i+4 \leq e \leq j} Qb[i,e] \times emulti01[j-e] \\ & + \sum_{\substack{i < d \leq j-4 \\ d+4 \leq e \leq j}} Qb[d,e] \times emulti01[j+d-i-e] \\ & + \sum_{\substack{i < d \leq j-4 \\ d+4 \leq e \leq j}} Qm2[i,d-1] \times Qb[d,e] \times emulti01[j-e];\end{aligned}$$
$$(\forall 0 \leq i = j-4 < N-4)\ Qm2[i,j] = \sum_{i+4 \leq e \leq j} Qb[i,e] \times emulti01[j-e];$$
$$(\forall 0 \leq i \leq j < i+4 \leq N)\ Qm2[i,j] = 0;$$

$$(\forall 0 \leq i \leq j < N)\ Qb[i,j] = QbTemp[i,j] \times base\_pair(seq[i], seq[j]);$$
$$\begin{aligned}(\forall 0 \leq i < j-6 < N-6)\ QbTemp[i,j] = & eh[i,j] + \sum_{i+5 \leq e < j} esbi[i,j,i+1,e] \times Qb[i+1,e] \\ & + \sum_{\substack{i+1 < d \leq j-5 \\ d+4 \leq e < j}} Qm2[i+1,d-1] \times Qb[d,e] \times emulti11[j-e-1] \\ & + \sum_{\substack{i+1 < d \leq j-5 \\ d+4 \leq e < j}} esbi[i,j,d,e] \times Qb[d,e];\end{aligned}$$
$$(\forall 0 \leq i = j-6 < N-6)\ QbTemp[i,j] = eh[i,j] + \sum_{i+5 \leq e < j} esbi[i,j,i+1,e] \times Qb[i+1,e];$$
$$(\forall 0 \leq i < j-3 < i+3 < N)\ QbTemp[i,j] = eh[i,j];$$
$$(\forall 0 \leq i \leq j \leq i+3 < N)\ QbTemp[i,j] = 0;$$

FIGURE 6.10: Equations of the McCaskill program. The output of the program is Q.

multiplications ($A.B$, $B.U$ where U is upper-triangular, diagonal matrix multiplication), but also some matrix and vector additions, point to point multiplications, reduction on a vector ($\vec{y} = \sum_k \vec{x}_k$).

The subsystems which are the most frequently used are the ones coming from reductions and which does not correspond to border cases. We managed to recognize the totality of the computation of 5 of these subsystems, over 6.

McCaskill This application is a subset of the computation of a bioinformatics application called *piRNA* (Partition function of Interacting RNAs [16]). Its equations are shown in Figure 6.10.

About the legality of tiling, we have two tile groups: Q (which is the output of the program, but never used in the equations), and $(Qb, QbTemp, Qm2)$. We also notice that all the dependences are always positive along the first dimension, and negative along the second dimension. Thus, rectangular tiling is legal.

After applying the monoparametric tiling transformation, we obtain in total 113 subsystems, 99 of them coming from the reductions of the program. The tile taken by the monoparametric tiling transformation is about 57 seconds.

During the recognition process, we have an issue with 2 subsystems, for which the computation of a transitive closure takes a significant amount of time. Across all the remaining subsystems, we perform 2245 comparisons between a program and a template. We consider in total 4566 equivalence automata, containing in total 90812 states. In addition, 26 equivalence subproblems were considered. The total time taken by our framework is 4196.5 seconds, which is about 1 hour and 10 minutes. In average, we spend 1.87 seconds per instance of template-match comparison.

We managed to detect 80 templates in total, however, only 8 of them are not a "transpose" preceding a non-recognized computation. This poor result can be explained by the fact that using a linear algebra library for this computation is not a good fit.

Indeed, the subsystems could not match our linear algebra templates for several reasons. For example, we have several subsystems whose top computation is a reduction, summing over the dimension k, but such that the boundary conditions on k are strange (such as $j \leq k \leq i + 4$). None of our template manages to match a suitable reduction with the same number of terms summed for every values of (i, j). Also, several subsystems contain reductions which project 2 dimensions at once, whose result is two dimensional (i.e., $Out[i, j] = \sum_{k,l} temp[i, j, k, l]$). Even if we ignore the issue on the bounds on k and l, we do not have any template which accumulates over 2 dimensions at once.

Therefore, a linear algebra library of template is not suitable for this computation. However, the computation of many subsystems have the same kind of structure. Thus, we might be able to identify a common operator which can be recognized over many subsystems. Then, we can create an highly-efficient implementation of this operator, and add it to our template library.

6.4 Discussion

Post-processing: merging the templates After obtaining a tree of template which corresponds to our program, we can merge some nodes of this tree. This process is particularly important to manage transposition and scalar multiplication in an efficient way. Indeed, as presented in Section 6.1, the operations of the BLAS library have several options. For example, DGEMM has a option to transpose both of its input matrices, and can multiply the result with a scalar. Thus, if we detect a matrix multiplication template, followed by a transposition, for example, we can use a single call to BLAS instead of two function calls.

Another situation where merging templates is advantageous is when we have adaptation of output domains. Indeed, if we detect an adaptation for a triangular output domain, followed by a matrix multiplication, having an implementation of a matrix multiplication which only computes the triangular part of the domain instead of the full domain will avoid useless operations.

Post-processing: optimizing the algorithm itself We can also use the information summarized in the template tree to optimize the algorithm. For example, if we detect some redundant operations among the templates detected, we can reorganize the templates to reuse the result of such operations. If we combine such mechanism with a cost function which estimates the operational complexity, we can explore different versions of an algorithm and select the best version, before generating the BLAS calls.

For example, if we consider the Cholesky computation (Figure 4.15, Page 112), we notice that every tile of a column computes the inverse of the same lower triangular matrix while multiplying it with a square matrix. Thus, we can examine another version of this algorithm where the inverse of this lower triangular matrix is computed once separately, and each tile is performing a triangular matrix multiplication. However, after examining the complexity of each versions, the latter one turns out to be more costly (and also requires more space).

Towards code generation After obtaining the tree of templates, we still have several issues to solve before being able to generate some code. In particular, we have to be careful about the

memory management. Indeed, in BLAS, most of the operations are in-place, i.e., they reuse one of the input matrix as an output (for example, $C \leftarrow \alpha.A.B + \beta.C$ for DGEMM). Our templates are purely functional, i.e., they assume that the output and the input matrices are allocated in different places. Thus, we need to determine if and when we need to copy a matrix in order to use the in-place operations from BLAS.

There are also several options for the storage mapping of the matrices manipulated in BLAS (row major and column major for square matrices, different storage methods for triangular matrices). Evaluating and selecting the best option is another piece which is required before generating some code.

Another feature would be to switch between implementations of a template, depending on some properties, such as the size of the template (for example, we can imagine a switch between a BLAS implementation, and a code generated through LGEN [77] which outperforms BLAS for small problem sizes).

Extending the template library The template library we have presented in Section 6.1 corresponds to the operations which can be found in BLAS. It is possible to extend this library to include more operations, such that the ones from LAPACK [6]. However, we have to be careful about the size of the template library, which impacts directly the time taken by our framework. Hence, we might need to refine the classification of our library to reduce the number of template to be considered at each steps.

Another extension is to change the vector space our linear-algebra operations operates. For example, instead of considering the vector space $(\mathbb{R}, +, \times)$, we can consider the semi-ring $(\mathbb{R} \cup \{-\infty\}, max, +)$ which is useful for some dynamic programming applications.

Chapter 7

Related Work

In this chapter, we present the links between our contributions and others. We will first present in Section 7.1 the work about the tiling transformation, and how it relates to our monoparametric tiling transformation. Then, in Section 7.2 we present the work related to program equivalence and template recognition, and how our template recognition algorithm contributes. Finally, we list in Section 7.3 the body of work on dense linear algebra algorithm derivation, and show their relations with our template detection framework.

7.1 Tiling transformation and code generation

We have presented the tiling transformation [35, 87] in Section 2.3, and its characteristics (such as tile shape, fixed-size vs parametric, legality condition) were already discussed there. In this section, we focus on how tiling is managed in the current polyhedral compilers. We will first consider the case of fixed-size tiling, before considering parametric tiling.

Code generation for fixed-size tiling Fixed-size tiling is a polyhedral transformation, i.e., the transformed program is still polyhedral. This means that we have two options when applying the fixed-size tiling transformation: either we compute the intermediate representation of the

program after transformation, or we generate directly the code using a polyhedral code generator (such as Cloog [10]).

The Pluto [15] polyhedral compiler is a fully automatic source-to-source compiler that generates fixed-size tiled and parallel code. It finds automatically a set of valid tiling hyperplanes by formulating and solving an integer linear programming problem. Because of the problem formulation, the normal vector of hyperplanes are forced to be positive in the original paper, however this limitation was removed in a recent work [1]. After deciding on a set of hyperplanes, Pluto tiles specifically identified bands of the *scattering functions* (i.e., the scheduling functions) and generates immediately the syntax tree of the tiled code using Cloog.

In comparison, our monoparametric tiling transformation computes explicitly the intermediate representation of the tiled program. Because of the size of the resulting program, it might cause some scalability issues for the later polyhedral analysis. However, in our context, we need to keep all the information about the computation of each tile, thus we do not have a choice. For other purposes (such as code generation), it might be enough to retain only part of the information about the tiled program. For example, Kong et al [43] use a similar classification (called *signature* in their paper) to our notion of *kind of tile* for their dynamic dataflow compiler framework. However, instead of differentiating each tile according to its computation, they differentiate tiles according to their incoming and outgoing intra-tile dependences.

Code generation for parametric tiling Because parametric tiling is a non-polyhedral transformation and prevents any polyhedral analysis afterward, current compilers integrate this transformation in the code generation phase. It also prevents any further polyhedral transformation or analysis, which was not hard-coded in the code generator.

Parametric tiling is trivial when the iteration domain is rectangular, the easiest solution is to use a rectangular bounding box of the iteration space and tile it. However, if the iteration domain is, for example, triangular, many of the executed tiles are empty and such a method becomes inefficient.

Renganarayanan et al [68, 69] presented a parametric tiled code generator for perfectly nested loops and rectangular tiling, which only iterates over the non-empty tiles. The main idea of this approach is to compute the set of non-empty tiles (called *outset*) and the set of full tiles (called *inset*) in a simple way, then use these information to enable efficient code generation. This work was later extended to manage multi-level tiling [41, 69]. We notice that the outset and inset appears in our monoparametric tiling transformation: the outset is the union of the domains of all our kind of tiles, and the inset is the union of all the domains of our kind of tiles which are full-tiles.

Kim [39] proposed another parametric code generator called *D-tiling* for perfectly nested loop, following the work from Renganarayann. Its main insight is the idea that code generation can be done syntactically on each tiled loop incrementally, instead of all at once. It has been extended in order to manage imperfectly nested [40].

Independently, Hartono et al [33] have presented a code generation scheme called *PrimeTile* which also manages imperfectly nested loop. The main idea is to cut the computation into stripes, and to place the first tile origin on this stripe at the position where we are starting to have full tiles in this stripe. The generated code is sequential and efficient [78]. Because the tile origins of different stripes are not aligned, we cannot find a wavefront parallelism and this scheme cannot be adapted to generate parallel tiled code.

Later, Hartono et al [32] have presented a code generation scheme called *DynTile* which manages to generate parallel tiled code for imperfect nested loop. The idea is to consider the convex hull of all statements, then to rely on a dynamic inspector to determine the wavefronts of tiles, which are scheduled in parallel. Finally, Baskaran et al [9] have presented *PTile* which allows parametrized parallel tiled code for imperfectly nested affine loops. This algorithm is identical to the one used in D-tiler, and was independently developed. A survey [78] compares the effectiveness of the sequential, and the parallel code generated by Primetile, Dyntile and PTile.

Another approach is to adapt the Fourier-Motzkin elimination procedure to manage parametric coefficient. This has been done by Amarasinghe [4] who integrated the possibility of managing linear combination of parametric coefficient in the SUIF tool set (such as $(N+2M).i$, where N

and M are parameters, and i is a variable), but no details have been provided and only perfectly nested loops were managed. Lakshminarayanan et al [69] (Appendix B) extended this to the case where the coefficients of a linear inequality can be parameters.

More generally, several people have been looking at extending the polyhedral model to be able to manage parametric tiling naturally. Größlinger et al [29] extended the polyhedral model to deal with parametrized coefficients, and have showed how to adapt Fourier-Motzkin and the simplex algorithm. In particular, these coefficients can be rational fractions of polynomials of parameters (such as $\frac{0.3 * N^2}{0.7 * N * M + 3}$). However, they have to rely on quantifier elimination, thus their method has scaling issues. Achtziger et al [2] studied how to find a valid quadratic schedules for an affine recurrence equation. Recently, Feautrier [26] considered polynomial constraints and has presented an extension of Farkas lemma. This class encompasses the parametric tiling transformation, at the cost of the complexity of the analysis.

7.2 Program equivalence and template recognition

In this section, we present the state-of-the-art on the program equivalence algorithm, then on the template recognition algorithm, and how it relates to our template recognition algorithm.

7.2.1 Program equivalence

The equivalence problem between two programs is known to be undecidable [8]. However, many approaches and semi-algorithms were proposed in the last few years to tackle partially this problem.

A first approach to the equivalence problem consists on comparing directly the computations of both programs, by "unrolling" them simultaneously and step by step, while managing their recursions.

Barthou et al [8] proposed a semi-algorithm for System of Affine Recurrence Equations, which encodes the equivalence problem into a reachability problem of a Presburger automaton (i.e., a

finite automaton whose states are associated with an integer vector, and whose transitions can test and modify these values). This reachability problem is also undecidable, but some efficient heuristics exist. This algorithm only considers Herbrand equivalence and no semantic properties are considered.

Shashidhar et al [75] proposed another equivalence algorithm based on *Array Data Dependence Graph* (ADDG). This graph is a representation of the operations done by a program, and the data dependences between them. Their algorithm manages associativity and commutativity (by transforming locally the ADDG), but only over a finite number of elements. They manage recurrences by unfolding the loops from both programs as many times as needed until obtaining a comparison between the same states again.

Verdoolaege et al [81] proposed an improved formalism based on a *dependence graph*, that allows them to manage parametrized programs. They also present an alternative way to deal with recurrences, based on the widening operation. Commutativity is managed by testing every permutation of the arguments of operators until we find a good one. This approach is no longer possible if the number of arguments is parametrized (as it is in the case of reduction).

If we assume that the size of the programs we compare are fixed at compile time and small, a pragmatic approach to prove equivalence is to unroll the computation, to normalize it and to check that the same operations are performed in the same data. This approach has been explored by Schordan et al. [73], but, for obvious reasons, does not scale well, is not adaptable to parametric loops and does not manage semantic properties.

Pnueli et al. [58] introduced a method called *translation validation*. The idea is to create an automaton representing the possible states of a program (called a Synchronous Transition System), then to prove that there exist a bisimulation between the two automata.

Symbolic analysis [31] is another way of proving the equivalence of two programs, by deriving a symbolic expression for the outputs, as functions of the program inputs. Then, we just have to prove that both expressions are equivalent, potentially modulo some semantic properties.

Menon et al. [53] introduced fractal symbolic analysis. It consists of producing a new equivalence problem with simpler programs, such that if the new programs are equivalent, then the original programs were equivalent. This new problem is an approximation of the original equivalence problem. By applying the same technique recursively, they manage to obtain programs which are simple enough to be managed by a classical symbolic analysis.

Karfa et al. [36] proposed an algorithm to decide equivalence based on ADDG, inspired by symbolic analysis. The idea behind their equivalence checking is to build an arithmetic expression corresponding to the computation done by the considered program. By normalizing this expression, they are able to manage the semantic properties of binary operators. However, because they need to have a finite arithmetic expression, they are not able to manage recursion and reductions.

Lopes et al. [49] used a similar approach and manages uninterpreted function symbols. The idea is to replace these uninterpreted function symbols by an affine expression with parametric coefficients, then to find an arithmetic expression of the outputs as a function of the inputs. They manage loops by considering it as a recurrence, and by solving it (i.e., by finding a closed form of the state of the loop after a given number of iterations), which is not always feasible.

7.2.2 Template recognition

Template recognition algorithm We can classify the current state-of-the-art template recognition algorithm into two categories: those based on dependence graphs [57] and those based on Abstract Syntax Tree [12, 38, 54].

Pinter and Pinter's recognition algorithm [57] is based on the Program Dependence Graph. After building and normalize it, they try to recognize patterns within it, using a graph grammar. If a portion of the graph matches, then a computation is detected.

Both Kessler's PARAMAT [38] and Bhansali's system [12] are based on the AST of the program. In the case of PARAMAT, the program is first normalized (by doing various transformation such as constant propagation, or dead-code elimination) before matching exactly the AST with the

template. In the case of Bhansali's system, there is no normalization step before this matching. This last work contains a library of templates which is similar to our framework: their templates (which are called *patterns* and described using a DSL) are organized into categories (which are the application domain of the templates, for example "linear algebra solver"), in order to prune the space of template to be matched.

Alias's template recognition algorithm [3] is the closest to our contribution. The algorithm is composed of two steps. The first step (called *slicing*) gathers candidate portions of the code which can potentially match with the template. The second step (*instanciation test*) considers the previously extracted slices, and determines which ones correspond to the template we aim to recognize. This method is based on a unification tree-automaton, which unrolls the computation of both the template and the slice and unifies the template with the program.

Compared to our contribution, the template considered can be function of the first order, which means that an operator in a template might be an unknown part to be matched. However, they assume that the templates are linear, which means that the inputs of a template can only occur once. About the recognition algorithm itself, Alias's algorithm can recognize a template anywhere in the program whereas, in our case, the output of the template and the program must match. However, because of this, it is possible for them to detect several overlapping templates, which forces them to select which template to keep.

We also notice that none of the recognition algorithm described above consider semantic properties.

Reduction and scan detection Many work focus on detecting reductions and scans inside a polyhedral program, which can be viewed as a special case of template recognition. The earliest work was by Redon and Feautrier [65]. This paper focuses on detecting recurrences inside a system of recurrence equations, thus can be used to detect reductions and scans (because they are special cases of recurrences). Their approach is based on a pattern-matching mechanism which is able to detect multidimensional recurrences, but fails if a reduction or scan spans

other multiple equations (mutual dependent variables) or is higher-order (i.e., the recursion uses multiple elements from the previous iterations).

Sato and al. [71] detects loops as instance of matrix vector multiplication, which can be implemented by a reduction operator. Because of this formalism, they are able to manage high-order recursions: for example, if we consider a Fibonnaci computation $F_i = F_{i-1} + F_{i-2}$, the recognized matrix vector multiplication will be:

$$\begin{bmatrix} F_i \\ F_{i-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} . \begin{bmatrix} F_{i-1} \\ F_{i-2} \end{bmatrix}$$

However, this methods does not manage to recognize multi-dimensional reductions or scans, or when a scan or a reduction is inside a multi-dimensional loop.

Zou and Rajopadhye [90] have managed to combine the two previous contributions and overcome their respective limitations.

Menon et al. [52] have presented a system which detect matrix multiplication operations inside a Matlab program, in order to replace it by a BLAS library call. The reduction detected are straight-forward, and a set of rewriting rules (called *axioms*) are used to normalize the program, in order to identify matrix multiplications.

7.3 Dense linear algebra algorithm derivation

In this section, we present the state-of-the-art on dense linear algebra algorithm derivation and show how it relates to our template detection framework.

FLAME Van de Geijn's group have developed FLAME [13, 30], a Formal Linear Algebra Methodology Environment. The input of this environment is a precondition and a post-condition of a linear algebra computation, expressed as a high-level equation on the input and output matrices. For example, in order to derive a Cholesky algorithm, the input to FLAME would have been a matrix A, the output a matrix L, the property that L is a lower-triangular matrix,

and the equation $A = L.L^T$. Given this information, they are able to derive a list of in-place algorithms which satisfy this specification.

The derivation of the algorithm is based on an *algorithmic skeleton*, which consists mainly in a while loop, in which each iteration builds a larger portion of the output matrix. Each step of the derivation aims at completing this skeleton to obtain the full algorithm, starting by deriving the invariant of the while loop, and finishing by the computation performed inside. For every option encountered during these steps, a different version of the algorithm is generated. The result of this derivation is a pseudo-code algorithm which manipulates rows and columns of block of matrices. This pseudo-code algorithm is then used to generate an efficient code.

An iteration of the while loop corresponds to a progression of one row (or one column) in the in-place computation of the output. As an option to their derivation, they can make one iteration of the while loop correspond to a progression of b rows (or b columns), where b is a parameter: at each new iteration, instead of considering only one extra row/column, they can consider b extra rows/columns at once. In that case, the derived computations inside the while loop are dealing with sub-matrices and vectors of size b, which are similar to the template we detect with our framework.

Hydra Duchâteau et al have developed Hydra [56], which is also a system to derive linear algebra algorithms. They start from an equation (called *signature*) specifying the algorithm they aim to derive (such as $L.X = B$ where L is lower-triangular and X is marked as the *unknown matrix* and is the output of the algorithm). The main idea of their derivation consists on using a divide and conquer strategy to recursively cut their matrices into smaller blocks, and propagate this division inside the matrix equation. For example, if we consider $L.X = B$ and cut all these matrices into 4 submatrices, we obtain the following equations:

$$\begin{cases} L_{0,0}.X_{0,0} & = & B_{0,0} \\ L_{1,0}.X_{0,0} + L_{1,1}.X_{1,0} & = & B_{1,0} \\ L_{0,0}.X_{0,1} & = & B_{0,1} \\ L_{1,0}.X_{0,1} + L_{1,1}.X_{1,1} & = & B_{1,1} \end{cases}$$

When the matrices are small enough, they stop the recursive divide and conquer strategy and rely on a library call (instead of stopping at the scalar level). Then, the next step is to produce a task graph, so that they can figure out in which order they should compute the sub-blocks of the unknown matrix. Then, using dynamic scheduling algorithms to avoid load-balancing issues, they generates a parallel code corresponding to their specification.

LGen Spampinato et al have developed LGen [77], which focuses on deriving linear algebra implementations for very small and fixed problem sizes (e.g., 5×9 matrices), called *BLAC* (Basic Linear Algebra Computations). The computation is specified through a linear algebra equation, in which the left-hand side is the output of the computation and the right-hand side is an expression of the inputs of the computation. The first step of their derivation is to use a tiling, decide for its tile sizes and propagate it to the rest of the equations. Then, they makes the access pattern and loop explicits, before performing various optimizations (such as loop unrolling, scalar replacement) and obtaining an efficient vectorized C code. The best version is picked by using auto-tuning. Their methodology is inspired by SPIRAL [60], which targets Digital Signal Processing computations.

Autotuning framework and specialized compiler for linear algebra Many other works [11, 14, 76, 82, 83] aim to find the best implementation possible for linear algebra computation, through autotuning. Compared to the frameworks described previously in this section, their starting specifications already describe the computation, instead of specifying it then deriving it. Some details (such that the value of the tile size parameters) are determined through exploration, but no new piece of computation is generated.

Comparison with our framework All these previous works are deriving a dense linear algebra algorithm from a high-level specification, which consists on an equation between matrices and vectors. Our framework aims to do the reverse: given a computation, we want to retrieve the high-level properties of the program through template recognition, in order to place a library call whenever possible.

Chapter 8

Conclusion

We conclude this document in Section 8.1. Then, we present some interesting unexplored research directions which are directly in the continuation of our work in Section 8.2.

8.1 Conclusion

Nowadays, architectures are becoming more and more complex, and it is increasingly difficult to use them at their full capabilities. This has caused a gap in performance between a code which are automatically generated through a compiler, and a code from a high-performance library, which was finely tuned. Thus, in order to improve the performance of a compiler-generated code, we want to be able to place calls to operations from a high-performance library. In this dissertation, we consider dense linear algebra operations and focus on the following problem: given a polyhedral computation, how can we detect subcomputations that corresponds to dense linear algebra operations?

This dissertation makes three contributions: a program transformation called monoparametric tiling, a template recognition algorithm and a framework which combines these two previous contributions to address our problem.

The monoparametric tiling is a tiling transformation in which the tile sizes are multiples of a common tile size parameter. This transformation is in-between fixed-size tiling and parametric tiling: indeed, this transformation is still polyhedral (like the fixed-size tiling transformation), while having parametric tile sizes of fixed shapes. We first consider the first half of this transformation, called monoparametric partitioning, which is just a reindexing of the spaces of a program in order to introduce the dimensions used to express a tiling. We show how to apply this partitioning transformation on polyhedra, affine functions then program, both for hyper-rectangular and general tile shapes. Then, we present the second half of the transformation, which isolates the computation of each tiles inside an atomic subprogram.

We introduce a template recognition algorithm, an extension of Barthou's program equivalence algorithm [8]. This algorithm is able to deal with semantic properties commonly found in dense linear algebra applications, such as associativity and commutativity of binary operators. To the best of our knowledge, our template recognition algorithm is the first template recognition algorithm powerful enough to be able to recognize any operation from BLAS.

Finally, we introduce our template detection framework. This framework first applies the monoparametric tiling transformation, then considers each subprogram independently, trying to recognize it as a finite combination of templates. Our templates are coming from a template library inspired by BLAS [46]. Our framework successfully recognizes most of the computation of dense linear algebra applications, and recognizes some portions of applications outside of the dense linear algebra domain. Then, the piece of code recognized as a linear algebra operation can be substituted by a library call, which will improve the performance of the code, or we can use this newly acquired high-level information to perform some optimization of the algorithm itself.

8.2 Future directions

In this section, we discuss the future research directions which span from our contributions and were not addressed yet. We consider each one of our contributions in the order of this

dissertation.

8.2.1 Monoparametric tiling transformation

Necessary and sufficient condition for the legality of tiling Currently, we do not perform any legality check of the provided tiling informations. As mentioned in Section 3.4, we can check for the legality condition after applying the partitioning transformation to the whole program, by collecting the block contributions of the partitioned dependence functions. We also showed that this criterion is more precise than the legality condition based on the tiling hyperplanes. However, partitioning the whole program is costly if we just want to check the legality of a tiling. It might be possible to avoid this cost by focusing on the dependences functions, paired with their context domain (i.e., for which indices a dependence function is used).

Monoparametric tiled code generation for any tile shape If we combine our monoparametric tiling transformation with a polyhedral code generator, we obtain a monoparametric tiled code generator. As shown in Section 4.4, the main issue with our transformation is the size of the generated tiled code. This is caused by the fact we build a full program representation of the tiled code. We might be able to avoid building this program representation by generating immediately the tiled code, like Pluto does. However, this will also prevent any polyhedral analysis to be applied after the tiling.

Assuming that we have built such tiled code generator, the tile shapes supported are hyperrectangular tile shapes, or parallelogram tile shapes with some preprocessing. It is possible to extend such code generator to support any tile shape. In addition, because of the monoparametric nature of our tiling transformation, the tiles of the generated code will be monoparametrized.

We claim that, except for the partitioning part of the tiling transformation (whose generalization was presented in Section 3.3), the rest of the machinery can be completely abstracted from the tile shape. Indeed, once the partitioning has been applied, at no point the tile shape plays a role in the construction of the subsystems and main system of the tiled code, as shown in

Chapter 4. Because our legality condition is built on top of the partitioning and analyze the block contribution, it is also independent of the tile shape.

Monoparametric tiling and fixed-size tiling We hypothesize that monoparametric tiling is strictly better than fixed-size tiling. In other words, anything which can be done with fixed-size tiling can also be done with monoparametric tiling, only better (because of the limited amount of parametrization of a monoparametric tiled code).

In order to verify this claim, we can consider a fixed-size tiling code generator (e.g., Pluto [15]) and create an associated monoparametric tiling code generator, such that if we substitute the block size parameter by a constant value in the monoparametric tiled code, we obtain exactly the fixed-size tiled code. Intuitively, because both transformations are polyhedral, all the information collected in order to generate a fixed-size tiled code (such as the tiled version of the iteration space, ...) can also be collected for the monoparametric case. Thus, in order to obtain such monoparametric tiling code generator, we can combine these informations exactly in the same way than the fixed-size code generator.

If this claim is verified, then there is no benefit to use fixed-size tiling over monoparametric tiling.

8.2.2 Template recognition algorithm

We can enhance further the recognition power of our algorithm, by improving the management of some semantic properties.

Managing the semantic properties of reduction operators Our current template recognition algorithm does not consider the associativity and commutativity of the reduction operators. This means that if we compare two reductions, we have to compare their subexpressions exactly in the same order of accumulation. In particular, we cannot consider permutations in the order of accumulation. For example, this prevents us to recognize a match between $\sum_{k=0}^{N} I[k]$ and $\sum_{k'=0}^{N'} I'[N' - k']$, because the order of summation is reversed.

In Section 5.6, we discussed an extension of Barthou's equivalence algorithm we have proposed in [34], in order to manage the associativity and commutativity of the reduction operators. The next step would be to adapt this equivalence algorithm to a template recognition algorithm, like we did in Section 5.2 for the original equivalence algorithm. The main difficulty of this extension would come from the fact that we have to infer the mappings between two compared reductions, and we have to find simultaneously the inputs of a template.

Note that, in the context of our framework, we manage partially the associativity and commutativity properties of reductions operators. Indeed, as mentioned in Section 4.3, when the monoparametric tiling transformation tiles a reduction, this transformation decomposes it into smaller reductions of the size of a tile and introduces the partial result of a tile as a new variable (called $TempRed$). The original reduction becomes an accumulation over $TempRed$, each element of $TempRed$ being the accumulation over a tile. Thus, the associativity and commutativity properties were used in order to cut the reduction along the tiles boundaries. In practice, we showed that this reordering is enough in order to recognize linear algebraic templates with our framework.

Managing the distributivity property As discussed in Section 5.4, we manage most of the semantic properties commonly found in a dense linear algebra computation through a set of rewriting rules. However, this approach is not satisfactory for the distributivity property. Indeed, we have shown that distributing or factorizing any term encounter indiscriminately might prevent the recognition of some template. We proposed a solution based on multiple version of a template: one in which the terms are distributed and one in which the terms are factorized. This fix is good enough in our context, but we might want a cleaner solution.

It might be possible to manage the distributivity property by adapting the template recognition algorithm. For example, we can adopt a similar approach than the management of the associativity and commutativity properties, and generate different versions of the equivalence automaton, depending on whether we choose to distribute/factorize a term or not. This requires us to keep track of the surrounding factorized terms when analyzing a state of the automata.

8.2.3 Template recognition framework

Enriching the template library Currently, our template library is mostly composed of operations which can be found in the BLAS specification. We could extend this library to include operations which can be found in LAPACK. For example, we could include a Cholesky computation, whose corresponding scalar operation is a square root. Another possible extension is to consider operations from alternate semi-ring, such as $(max, +)$, which is useful for dynamic programming computations.

Another idea is to build automatically the template library, based on the unrecognized computations encountered. More precisely, every times a computation is not recognized by our template library, we can register this computation (or detect a portion of the computation which might correspond to the top-most operation, then register it). If a computation is encountered multiple times, we can decide to add it to the template library and notify the user that it might be interesting to have a corresponding efficient implementation. We already have a limited form of this idea, through our "recursion" template (which tries to find smaller instances of our original computation).

Using template recognition to improve performance Our framework detects subcomputations of a program as matricial operations. These high-level information can be exploited by using semantic properties of linear algebra operations to change the computation itself. For example, if we recognize $L.L^{-1}$, we can replace the corresponding computation by an identity matrix. If we detect a succession of matrix multiplications $A \otimes (B \otimes (C \otimes D))$, we can rearrange them into $(A \otimes B) \otimes (C \otimes D)$, in order to enable parallelism. These optimizations are much more powerful compared to what could have been done without the recognition process.

Then, when we decide to generate code, we can place library calls corresponding to these matricial operations. As discussed in Section 6.4, we need to preprocess the template tree we obtain, so that we merge them and minimize the number of library calls issued. We also need to decide for a memory allocation (most of the BLAS computation being in-place) and the memory storage for each matrix.

Appendix A

Résumé du travail de thèse

Ce chapitre consiste en un résumé étendu du travail de thèse écrit en Français. Son organisation suit la structure du document, c'est à dire que les sections correspondent aux chapitres du document. Cependant, même si le discours principal est identique, ce résumé présente moins de détails (preuves, exemples, commentaires secondaires). Ainsi, le lecteur est invité à se référer au document complet en Anglais pour des explications complètes.

A.1 Introduction

De nos jours, du à la complexité croissante des architectures, il est de plus en plus difficile de les exploiter pleinement afin d'exécuter une application le plus rapidement possible. En réponse à ce problème, des librairies qui proposent des implémentations à haute performance pour certaines opérations ont été créées. Ces implémentations ont été finement calibrées manuellement et leur performance ne sont généralement pas atteignable par un code généré par un compilateur.

Cependant, les appels à ces implémentations à haute-performance doivent être faites à la main, ce qui pose plusieurs problèmes. Tout d'abord, cela demande une compréhension profonde de l'algorithme de la part de l'auteur du programme, afin de pouvoir reconnaître, délimiter et remplacer les bouts correspondant par un appel de fonction. Ensuite, cette compréhension peut

être imparfaite, au sens où certains appels de fonction intéressants peuvent avoir été manqués. Ainsi, l'idéal serait de permettre aux compilateurs de placer automatiquement ces appels à des librairies, ce qui n'est, pour le moment, pas fait.

Le problème clef est de reconnaître un calcul qui correspond à une opération ayant une implémentation optimisée. Plus précisément, nous cherchons à reconnaître des sous-calculs (par opposition au programme entier) afin de pouvoir les remplacer par des appels de fonction correspondants. Dans le contexte de notre travail de thèse, nous nous intéressons à des opérations d'algèbre linéaire, pour lesquels plusieurs librairies existent (telles que BLAS [46], LAPACK [6]), et qui contiennent des opérations communément présents dans de nombreux domaines d'application.

Nous considérons donc le problème suivant: comment reconnaître des sous-calculs correspondant à des opérations d'algèbre linéaire dans un programme polyédrique? Ce problème soulève plusieurs défis. Comme on s'intéresse à des sous-calculs, on doit faire attention aux recouvrements entre opérations détectées. Le fait que l'on cible des opérations d'algèbre linéaire veut dire que l'on doit gérer les propriétés sémantiques associées à ce domaine. Enfin, nous devons faire attention à la scalabilité du procédé de reconnaissance.

Contributions L'idée principale de notre solution est de découper préemptivement le calcul en blocs avant d'effectuer la reconnaissance d'opérations. Vu que l'on considère des opérations d'algèbre linéaire qui raisonnent sur des matrices qui sont rectangulaires, on partitionne l'espace des données en tuiles et utilise ce tuilage pour différencier les calculs en fonction de la tuile utilisée. Ainsi, le tuilage sur l'espace des données est propagé à l'espace des calculs. L'hypothèse faite est que ces sous-calculs correspondent à des combinaisons d'opérations d'algèbre linéaire. L'avantage de cette approche est, d'une part, d'éviter d'avoir des recouvrements entre opérations reconnues et, d'autre part, de fournir une liste d'endroits dans le flot de calcul où commencer à chercher à reconnaître un début de sous-calcul.

Ainsi, nous proposons les contributions suivantes:

- **Tuilage monoparamétrique:** Nous introduisons une nouvelle transformation de programme appellée le *tuilage monoparamétrique*. Un tuilage peut utiliser des tuiles de taille

fixe (les tailles de tuiles sont constantes et ne peuvent pas être changées après compilation) ou de taille paramétré (les tailles de tuiles sont des paramètres du programme et donc peuvent être changées juste avant exécution, mais le programme après transformation n'est plus polyédrique). Nous montrons que si nous considérons des tuiles dont les tailles sont des multiples d'un *unique paramètre*, le programme après transformation reste polyédrique, tout en permettant une paramétrisation limitée après compilation. Nous proposons ensuite une variante de cette transformation qui isole le calcul effectué par une tuile dans un sous-programme. Cela est possible du fait qu'il y a un nombre fini non-paramétrique de calculs différents effectués par les tuiles du programme, et donc on a besoin seulement d'un nombre fini non-paramétrique de sous-programmes.

- **Algorithme de reconnaissance de template:** Nous proposons une extension d'un algorithme d'équivalence de programme [8] en un algorithme de reconnaissance de template. Les templates que nous considérons dans ce document sont des programmes dont les entrées peuvent correspondre à des expressions inconnues. Ainsi, par rapport à un algorithme d'équivalence de programme, l'algorithme de reconnaissance de template doit également tenir compte de ces inconnues et déterminer leurs valeurs. Nous étendons par la suite cet algorithme de reconnaissance de template, de manière à gérer les propriétés sémantiques communément rencontrées en algèbre linéaire (associativité, commutativité, distributivité, ...).

- **Procédé de reconnaissance de sous-calculs d'algèbre linéaire:** Nous combinons les deux contributions précédentes de la manière suivante: nous appliquons d'abord le tuilage monoparamétrique pour séparer les calculs des tuiles en sous-programmes isolés. Puis, nous considérons chaque sous-programme indépendamment et nous essayons de les reconnaître comme une combinaison d'opérateurs d'algèbre linéaire. Ces opérateurs sont définis à travers une librairie de templates, inspirée de BLAS [46]. A chaque fois qu'un opérateur est reconnu, l'algorithme est appliqué récursivement sur les expressions correspondants aux entrées du template. Le résultat du procédé est donc un arbre de templates par sous-programmes, chaque noeud correspondant à un template reconnu.

Plan Le reste du résumé est structuré de la manière suivante: dans la Section A.2, nous introduisons les définitions et notations nécessaires pour comprendre le reste du travail. Nous introduisons en deux parties la transformation de tuilage monoparamétrique dans les Sections A.3 et A.4. La Section A.3 présente la transformation de partitionnement monoparamétrique, qui effectue une réindexation de tous les indices du programmes. Nous étudions tout d'abord le cas d'un tuilage rectangulaire, avant d'étendre la transformation à n'importe quelle forme de tuile. Les indices introduits par le partitionnement monoparamétrique sont ensuite utilisés dans la Section A.4 pour exprimer le tuilage, tout en isolant les calculs effectués par les tuiles dans des sous-programmes séparés.

La Section A.5 présente un algorithme de reconnaissance de template, basé sur un algorithme d'équivalence de programme proposé par Barthou et al [8]. Plusieurs extensions sont proposés pour permettre la gestion de propriétés sémantiques usuellement rencontrées en algèbre linéaire. La Section A.6 combine les contributions précédentes en un seul procédé, qui décompose le calcul d'un programme en tuiles, avant d'essayer de reconnaître chaque tuile en tant que combinaison d'opérateurs classique d'algèbre linéaire. Notamment, ce procédé utilise une bibliothèque de templates, inspirés de BLAS [46]. Nous concluons finalement ce travail dans la Section A.7, et proposons quelques pistes de recherche.

A.2 Définitions et notations

Cette section présente les définitions et notations qui serons utilisées dans la suite de cette thèse. Nous définissons tout d'abord la représentation de programme choisie, puis montrons un bref aperçu des transformations de programme considérées par la suite (c'est à dire, changement de base, et tuilage), puis finissons par présenter les intuitions principales derrière l'algorithme d'équivalence de programme.

Représentation de programme La représentation de programme que nous choisissons est la suivante:

Definition A.1. Un programme polyédrique est un programme dont le calcul peut être représenté par une liste d'équations, de la forme suivante:

$$\vec{i} \in \mathcal{D} : Var[\vec{i}] = Expr(Var_1[u_1(\vec{i})], \ldots, Var_d[u_d(\vec{i})])$$

où $\mathcal{D}$ est un polyèdre, c'est à dire un ensemble d'entier satisfaisant des contraintes affines et les u_k sont des fonctions affines, appelées *fonctions de dépendances* qui lie chaque lecture à son site de définition $Var_k[u_k(\vec{i})]$. Var est une variable du programme, qui peut être soit une variable d'entrée, soit une variable de sortie, soit une variable locale. $\vec{i}$ est appelé *vecteur d'itération.* $Expr$ est une expression est peut-être des formes suivantes:

- Une variable $S[u[\vec{i}]$
- Une opération $op(Expr_1, \ldots, Expr_k)$ où l'arité de l'opération est k. Une constante est un opérateur d'arité 0.
- Un fonction des indices $f(\vec{i})$

Une variable peut avoir plusieurs équations définissant ses valeurs, sous réserve que ces définitions concernent des ensembles de vecteur d'itération disjoints. Le domaine d'une variable est l'union de tous ces ensembles, et correspond à l'ensemble des points pour lesquels cette variable est définie à travers une des équations du programme.

Nous rajoutons à cette définition la notion de réduction. Une réduction est une application successive d'un opérateur binaire associatif et commutatif sur un ensemble de valeur. Un exemple typique de réduction rencontré en algèbre linéaire est une sommation sur un nombre paramétrique de valeurs:

$$C[i,j] \quad = \quad \sum_{k=0}^{k<N} A[i,k] * B[k,j];$$

Nous intégrons les réductions à notre représentation de programme en tant que nouveau type d'équation, de la forme suivante:

$$\vec{i} \in \mathcal{D}_r : Var[\vec{i}] = \bigoplus_{\substack{\vec{j} \in \mathcal{D} \\ \vec{i} = \pi(\vec{j})}} Expr(Var_1[f_1(\vec{j})], \ldots, Var_d[f_d(\vec{j})])$$

où π est une fonction affine appelée *fonction de projection*, qui détermine les directions selon lesquelles sommer les valeurs de la sous-expression. Afin de faciliter l'écriture de programmes, nous autorisons l'utilisation de réductions comme arguments d'une expression.

Transformation de programme Dans le reste du document, nous nous intéresserons principalement à deux transformations de programme: la transformation de *changement de base* et la transformation de *tuilage*.

Un changement de base est une transformation qui modifie le domaine d'une variable en utilisant une fonction unimodulaire (c'est à dire, une bijection dont le déterminant vaut 1 ou -1). Le nouveau domaine de la variable est l'image de l'ancien domaine par cette fonction unimodulaire, et les équations du programme sont adaptées pour tenir compte de ce changement. Ainsi, cette transformation est juste une fonction de réindexage du domaine d'une variable et ne modifie en aucun cas le calcul effectué par un programme.

Un tuilage est une transformation qui regroupe les calculs en groupes (appelées *tuiles*) qui sont exécutés de manière atomiques. La Figure A.1 montre un exemple de tuilage pour des tuiles carrées de taille 3 par 3.

Parce que les tuiles sont exécutées de manière atomique, on ne peut pas avoir de cycle de dépendances entre elles. Par exemple, dans la Figure A.1, chaque tuiles dépendent de la tuile à leur gauche et en dessous, et il n'y a pas de cycle de dépendence entre différentes tuiles. Par conséquent, le tuilage est légal. Des changements de base sont fréquemment utilisés pour arranger les dépendances d'un programme et rendre un tuilage légal.

$$
\begin{aligned}
(\forall i = j = 0)\ & A[i,j] = 1 \\
(\forall j = 0 < i)\ & A[i,j] = A[i-1,j] \\
(\forall i = 0 < j)\ & A[i,j] = A[i,j-1] \\
(\forall 0 < i,j)\ & A[i,j] = A[i-1,j] + A[i,j-1]
\end{aligned}
$$

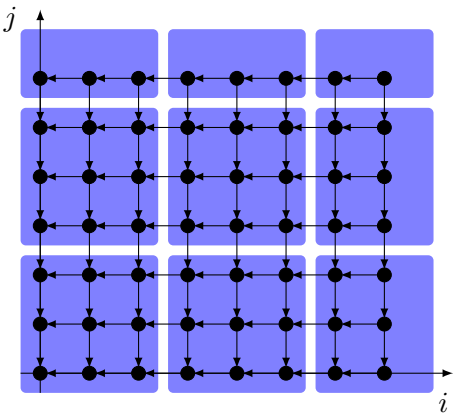


FIGURE A.1: Exemple de tuilage, avec pour tuiles des carrés de taille 3×3

Différentes variations du tuilage existent, par exemple en jouant sur la forme de la tuile considérée (parallélépipède, trapézoïde, hexagone, ...). Indépendamment, la nature des tailles de tuiles est un autre critère de variation du tuilage considéré. Si les tailles d'une tuile sont des constantes (par exemple 16×32), alors le tuilage est de taille fixe et cette transformation est polyédrique (c'est à dire, le programme transformé reste polyédrique). Le désavantage de cette transformation est que les tailles de tuile est fixée pendant la compilation, et donc on est obligé de recompiler le programme à chaque fois que l'on veut changer ces tailles, ce qui est gênant si on veut découvrir la taille de tuile qui donne les meilleures performances.

Si les tailles d'une tuile sont des paramètres (par exemple $b_1 \times b_2$), cette transformation n'est plus polyédrique (c'est à dire, le programme transformé n'est plus polyédrique, à cause de contraintes quadratiques introduites). Dans ce dernier cas, parce qu'on sort du modèle polyédrique, il n'est plus possible de composer des transformations ou analyses polyédriques à la suite d'un tuilage paramétrique.

Algorithme d'équivalence de programme Dans la Section A.5, nous allons étendre un algorithme d'équivalence de programme en un algorithme de reconnaissance de template. La notion d'équivalence utilisée par cet algorithme s'appelle l'*équivalence d'Herbrand*: deux programmes sont équivalents s'ils font exactement les mêmes opérations sur les mêmes données afin d'obtenir leur sorties. Notez que cette notion d'équivalence ne tient compte d'aucune propriété

sémantique. De plus, toute transformation de programme qui respecte les dépendences préserve cette équivalence.

L'algorithme d'équivalence de programme qui constitue notre point de départ est celui proposé par Barthou et al [8]. La description et la formalisation complète de l'algorithme peuvent se trouver dans la Section 2.4. Nous nous contentons ici de donner les intuitions principales de l'algorithme. La représentation de programme considérée est celle des Systèmes d'Équations Récurrentes Affines (SERA), qui est similaire à la représentation de programme que nous avions introduit précédemment. Aussi, le problème de décider l'équivalence de deux programmes est indécidable, et donc l'algorithme d'équivalence de programme est en fait un *semi-algorithme* (il est possible que l'algorithme échoue à conclure une équivalence ou une non-équivalence)

L'algorithme d'équivalence de programme de Barthou repose sur la notion d'*automate d'équivalence*. Cet automate est un automate de Presburger, ce qui veut dire que chaque état est associé avec un vecteur de valeurs entières et que les transitions peuvent inspecter et modifier ces valeurs. Dans le cas d'un automate d'équivalence, les états de cet automates correspondent à une comparaison entre deux sous-calculs de chaque programme "$Expr_1 = Expr_2$" et les vecteurs correspondent aux indices de ces sous-calculs $(\vec{i_1}, \vec{i_2})$. L'intuition principale d'un automate d'équivalence est que progresser dans l'automate revient à dérouler symboliquement les calculs effectuées par chaque programme, en partant des sorties, tout en éliminant les opérateurs identiques qui occurrent de chaque côté.

Ainsi, conformément à cette intuition, l'état initial d'un automate d'équivalence compare les sorties des deux programmes. L'automate admet deux sortes d'état final: les *états d'échec* qui correspondent à une comparaison trivialement fausse (par exemple, comparer une entrée d'un côté avec un opérateur de l'autre) et les *états de réussite* qui correspondent à des comparaisons entre entrées correspondantes. Les transitions de l'automate sont construites en suivant 3 règles de constructions, qui, intuitivement, déroulent les calculs et éliminent les opérateurs présents de chaque côté.

Barthou et al ont montré que deux programmes sont équivalents si et seulement si tout chemin qui part de l'état initial en prenant des indices égaux $(\vec{i}, \vec{i})$ (ce qui correspond à comparer la

même sortie):

- N'arrive jamais à accéder un état final d'échec
- Accède un état final de réussite uniquement quand les indices des deux entrées comparées sont égaux (ce qui veut correspond à comparer la même entrée)

Ainsi, le problème de décider l'équivalence de deux programmes peut se réduire au problème de calculer l'ensemble d'accessibilité de certains états dans un automate de Presburger. Ce dernier problème est lui-même indécidable, mais plusieurs heuristiques existent pour le résoudre.

Cet algorithme d'équivalence sera étendu en un algorithme de reconnaissance de *templates* dans la Section A.5. Les templates que nous considérons dans ce document sont des programmes dont les entrées peuvent correspondre à des expressions inconnues. Le problème de reconnaissance de template prend en argument un programme et un template et essaye de trouver des valeurs aux entrées du template qui le rend équivalent au programme. Il s'agit d'une définition plus faible que celle retenue par Alias [3], qui considère des templates comportant des fonctions inconnues.

A.3 Partitionnement monoparamétrique

Dans cette section, nous nous intéressons à la première partie de la transformation de tuilage monoparamétrique, appelée *partitionnement monoparamétrique*. La seconde partie de cette transformation est décrite dans la Section A.4. Afin de simplifier le formalisme, nous nous concentrons, au début de cette section, sur le cas des tuiles rectangulaires, avant de généraliser nos résultats au cas général.

Commençons par définir la transformation de partitionnement monoparamétrique. On considère un pavage de l'espace de chaque variable par des tuiles rectangulaires de taille $(d_1.b)\times\cdots\times(d_k.b)$, où les d_i sont des constantes et b est un paramètre du programme. Ainsi, chaque point de l'espace original $\vec{i}$ se retrouve dans une unique tuile rectangulaire de ce pavage. Il est possible d'introduire de nouveaux indices qui identifient la position de ce point dans le nouveau pavage.

Afin d'identifier une tuile (respectivement la position d'un point dans une tuile), de nouveaux indices appelés *indices tuilés* $\vec{i_b}$ (respectivement *indices locaux* $\vec{i_l}$) sont introduits, tels que $\vec{i} = D.b.\vec{i_b} + \vec{i_l}$, $\vec{0} \leq \vec{i_l} < D.\vec{1}$ et D est une matrice (appelé *ratio*) dont les coefficients diagonaux sont les d_i.

La transformation de partitionnement monoparamétrique est simplement une réindexation de tous les indices du programme, qui remplace les indices originaux $\vec{i}$ par les indices tuilés et locaux $(\vec{i_b}, \vec{i_l})$. Ainsi, le nombre de dimensions de tous les espaces du programme transformé sont doublés par rapport au programme original. Dans le reste de cette section, nous allons tout d'abord montrer que, bien que ce changement d'indice n'est pas affine, nous avons tout de même des propriétés de stabilité qui permettent d'obtenir un programme transformé polyédrique.

Propriété de stabilité dans le cas des tuiles rectangulaires Tout d'abord, étudions l'application de la transformation de partitionnement monoparamétrique sur un polyèdre, puis sur une fonction affine. En effet, ces deux objets mathématiques sont les seuls qui interagissent avec les indices d'un programme. Ainsi, substituer ces objets par leur version partitionnée est le cœur de la transformation de partitionnement monoparamétrique.

Considérons un polyèdre $\mathcal{D}$. L'ensemble Δ obtenu en appliquant la transformation de partitionnement monoparamétrique sur ce polyèdre est une union finie non paramétrique de polyèdres admettant les propriétés suivantes:

- Chaque polyèdre de Δ correspond à une forme de tuile
- Les contraintes de chaque polyèdre peuvent être séparées en deux ensembles: les contraintes qui concernent les indices tuilés et les contraintes qui concernent les indices locaux. Il n'y a aucune contrainte qui font intervenir les deux types d'indices.

Ainsi, Δ décrit les différentes formes de tuiles qui arrivent après tuilage et les contraintes sur les indices tuilés qui spécifient où chaque forme de tuiles se trouvent. Par exemple, la Figure A.2 montre un exemple de partitionnement d'un triangle bi-dimensionnel, en utilisant un tuilage de taille $b \times b$

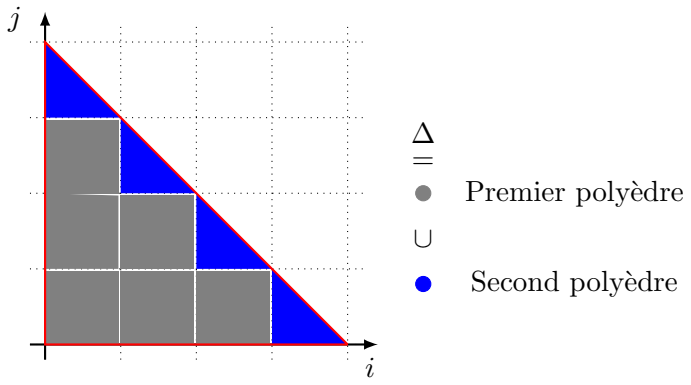


FIGURE A.2: Union de polyèdres Δ obtenus après partitionnement. Le polyèdre original est un triangle, et nous avons supposé, pour simplifier la présentation, que les tailles de tuile divisent la taille de ce triangle. Après partitionnement, nous obtenons une union de deux polyèdres dans Δ: un polyèdre qui correspond aux tuiles pleines, et un autre polyèdre qui correspond aux triangles inférieurs (sur la diagonale)

Considérons une fonction affine f. Tout d'abord, notons que cette fonction affine interagit avec deux espaces (correspondant à ses entrées et sorties): on doit donc considérer deux partitionnements sur ces deux espaces. La fonction ϕ obtenue en appliquant la transformation de partitionnement monoparamétrique sur cette fonction affine est une fonction affine par morceaux. Les branches de cette fonction affine par morceaux ont les propriétés suivantes:

- La valeur de chaque branche est une fonction affine et est différente des autres
- Les conditions de chaque branche est une conjonction de contraintes affines (càd, de la forme $\vec{a}.\vec{i} + b \geq 0$ avec $\vec{a}$ et b des constantes) et de contraintes modulo (càd de la forme $g(\vec{i_b})\%M = C$, où $0 \leq C < M$ sont des constantes et g est une fonction affine sur les indices tuilés i_b).
- Les branches ne contiennent aucune contrainte modulo si et seulement si une contrainte de divisibilité faisant intervenir les ratios des partitions est satisfaite. Plus précisément, si D est une matrice diagonale dont les coefficients sont les ratios du partitionnement sur l'espace d'entrée, si D' est cette même matrice pour le partitionnement de l'espace de sortie et si Q est la matrice des coefficients de f, la condition est que "$D'^{-1}.Q.D$ est une matrice entière".

Partitionnement dans le cas des tuiles rectangulaires En utilisant ces propriétés de stabilité, la transformation de partitionnement consiste simplement à substituer tous les polyèdres et fonctions affines d'un programme par leur versions partitionnées. Les fonctions de dépendances pouvant devenir des fonctions affines par morceaux, il est nécessaire de les aplatir, afin de créer une équation par branches de cette fonction. Il est important d'éliminer progressivement les branches non-satisfiables pendant cet aplatissement afin d'éviter toute explosion combinatoire.

Afin d'appliquer cette transformation, il est nécessaire d'assigner un partitionnement à tous les espaces intervenant dans un programme, c'est à dire, à tous les domaines des variables d'un programme. Cependant, il faut faire attention à ce que ces ratios n'introduisent pas de conditions de modulo lors du partitionnement des fonctions de dépendances (au risque de rendre le programme transformé non polyédrique). Par défaut, prendre des ratios carrés $(1 \times 1 \times \cdots \times 1)$ pour toutes les partitionnements des variables est suffisant pour éviter toute condition modulo.

Parce que cette spécification peut être lourde du point de vue de l'utilisateur, nous proposons que l'utilisateur ne définisse qu'une partie des ratios, et qu'un algorithme dérive les ratios manquants qui n'introduisent aucune condition modulo. Cet algorithme parcourt des équations du programme de bas en haut et trouve le ratio minimum qui n'introduit pas de modulo pour chaque variables. De plus, si cet algorithme échoue, c'est qu'il n'existe aucun ratio qui n'introduit aucun modulo, étant donné les spécifications fournies par l'utilisateur.

Partitionnement pour des formes de tuile quelconque Il est possible d'étendre les résultats précédents à des partitionnements avec n'importe quelle forme de tuiles. Tout d'abord, un partitionnement monoparamétrique pour une forme de tuile quelconques est définit à travers 3 objets:

- La forme de la tuile, qui est un agrandissement d'un polyèdre non paramétré par un facteur b, b étant le paramètre de taille de tuile
- Un treillis des origines de tuiles,

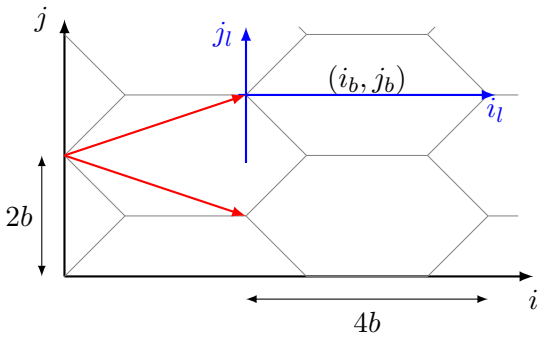


FIGURE A.3: Exemple d'un partitionnement monoparamétrique hexagonal pour un espace 2D. (i_b, j_b) sont les indices tuilés, qui identifient une tuile, (i_l, j_l) sont les indices locaux, qui identifient la position d'un point dans une tuile. La forme de la tuile est un hexagone dont les pentes sont à 45° et qui est de taille $4b \times 2b$. Cette tuile peut être vu comme l'agrandissement d'un hexagone de taille 4×2. Les flèches rouges correspondent à la base du treillis des origines des tuiles.

- Une fonction de décomposition qui, étant donné un point $\vec{i}$, retourne son indice tuilé $\vec{i_b}$ et local $\vec{i_l}$, qui identifie la tuile et les coordonnées locales de où se trouve ce point.

Cette définition est une généralisation du cas rectangulaire. Dans le cas rectangulaire, la forme de la tuile est un agrandissement d'un rectangle de taille constante $d_1 \times \cdots \times d_k$ par un facteur b, le treillis des origines de tuiles admet pour base les vecteurs $(d_i.\vec{e_i})_i$ où $\vec{e_i}$ est le ième vecteur canonique, et la fonction de décomposition consiste en une division entière. La Figure A.3 montre un autre exemple de partitionnement dans le cas des tuiles hexagonales.

Les propriétés de stabilité sur les polyèdres et fonctions affines sont toujours valables dans le cas des formes de tuile quelconques. À propos des fonctions affines, le critère sur les ratio pour éviter les modulos devient un critère sur la base du treillis des origines de tuiles. Cela veut notamment dire que la forme d'une tuile n'a aucun impact sur la présence de modulo dans une fonction partitionnée. L'application de ces propriétés de stabilité à un programme et l'algorithme de dérivation associé reste identique au cas rectangulaire.

A.4 Du partitionnement au tuilage

Dans cette section, nous présentons la seconde partie de la transformation de tuilage monoparamétrique. On suppose que la première partie de la transformation (partitionnement monoparamétrique, décrite dans la section précédente) a été effectuée, et nous nous servons des nouveaux indices introduits pour exprimer le tuilage.

Nous commençons par décrire une extension de notre représentation de programme. Cette extension autorise un programme à appeler des sous-programmes, appelés *sous-systèmes*, qui sont exécutés de manière atomique. Ensuite, nous décrivons la transformation de tuilage d'abord dans le cas de programme sans réductions, puis dans le cas de programme contenant des réductions. Le calcul de chaque tuiles est encapsulé dans un sous-système, ce qui nous permet d'imposer la propriété d'atomicité des tuiles, et d'isoler leurs calculs. Ces sous-systèmes seront considérés séparément dans la Section A.6 afin de tenter de reconnaître des combinaisons d'opérateur d'algèbre linéaire.

Sous-systèmes Nous introduisons une extension à notre représentation de programme qui autorise un programme à utiliser un autre programme (appelé *sous-système*) durant son exécution. Cet appel s'effectue via un type d'équations spécial appelé *équation d'utilisation*, de la forme suivante:

`use` $\mathcal{D}_{ext}$ nomSousSysteme[paramètres] (liste des entrées)
`returns` (liste des variables de sortie);

Cette équation d'utilisation appelle le programme "nomSousSysteme" avec les paramètres et valeur en entrées spécifiées, et récupère ses résultats dans les variables de sortie. Le polyèdre $\mathcal{D}_{ext}$ s'appelle le *domaine d'extension* et permet de paramétrer les appels de la manière suivante: chaque point $\vec{i}_{ext}$ de ce polyèdre correspond à un appel au programme "nomSousSysteme", et les indices $\vec{i}_{ext}$ peuvent être utilisés dans la spécification des paramètres et des entrées de ces appels. Ainsi, il est possible de spécifier un nombre paramétrique d'appels à travers une seule équation d'utilisation.

Tuilage pour des programmes sans réductions Nous supposons qu'un tuilage légal est spécifié par l'utilisateur en entrée (c'est à dire, quel changement de base et quelles variables on doit tuiler ensemble pour éviter toute dépendance cyclique entre tuiles). L'idée principale de la transformation de tuilage est de distribuer le calcul des tuiles dans des sous-systèmes, de telle sorte que le programme principal gère les communications entre tuiles et les appels aux sous-systèmes correspondants, tandis que les sous-systèmes contiennent le calcul effectué par le programme. Cependant, un programme tuilé possède habituellement un nombre paramétrique de tuiles, tandis qu'il n'est pas possible d'avoir un nombre paramétrique de sous-systèmes dans un programme.

Ce problème est résolu avec l'introduction de la notion de *type de tuile*. Il est possible de classifier les tuiles d'un programme selon le calcul qu'elles effectuent. Un type de tuile est une de ces classes, et on peut montrer qu'il n'y en a qu'un nombre fini non paramétrique. Ainsi, il est possible de créer un sous-système par type de tuile et de faire appel à ce sous-système à chaque fois qu'on veut exécuter une tuile de type associé. On a donc besoin de créer qu'un nombre fini non paramétrique de sous-système, ce qui rend la transformation de tuilage possible.

Le programme tuilé possède un système principal et une collection de sous-systèmes. Les équations d'un sous-système correspondent au calcul associé au type de tuile correspondant. Les entrées d'un sous-système sont les données minimales dont les équations d'un sous-système ont besoin, qui ne sont pas calculées à l'intérieur du sous-système en question. Les sorties d'un sous-système sont les données calculées par les équations du sous-système dont d'autres sous-systèmes ont besoin. De multiples entrées et sorties sont crées en fonction de la tuile qui produit la donnée fournie au sous-système (pour les entrées) ou qui nécessite la donnée fournée par le sous-système (pour les sorties).

Le système principal contient une équation d'utilisation par type de tuile, leur domaine d'extension correspondant au domaine où ce type de tuile est présent. Les sorties des équations d'utilisations sont stockées dans des variables locales, qui sont ensuite regroupées avant d'être réutilisées dans les entrées des équations d'utilisation. Cela nous permet d'éviter de dissocier selon si une entrée vient d'un type de tuile ou d'un autre.

Tuilage pour des programmes avec réductions Les réductions d'un programme introduisent des indices supplémentaires qui sont partitionnées et qui introduisent des indices tuilés supplémentaires. Par exemple, considérons un programme qui effectue une multiplication de matrices entre deux matrices carrées de taille N:

$$(\forall 0 \leq i, j < N)\ C[i, j] = \sum_{k=0}^{N-1} A[i, k] * B[k, j]$$

Après partitionnement, si on suppose que le paramètre N est divisible par la taille de tuile b, on obtient le programme suivant:

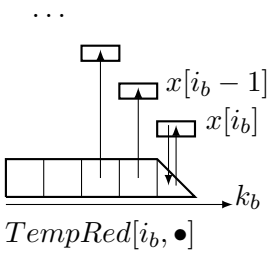
$$(\forall 0 \leq i_b, j_b < N_b)(\forall 0 \leq i_l, j_l < b)\ C[i_b, j_b, i_l, j_l] = \sum_{k_b, k_l} A[i_b, k_b, i_l, k_l] * B[k_b, j_b, k_l, j_l];$$

Notez à ce point que la réduction somme sur un ensemble de tuile indexées par k_b. Ainsi, afin de séparer les calculs de chacune de ces tuiles, nous décomposons la réduction, en introduisant une variable temporaire d'accumulation, nommée $TempRed$:

$$C[i_b, j_b, i_l, j_l] = \sum_{k_b} TempRed[i_b, j_b, k_b, i_l, j_l];$$
$$TempRed[i_b, j_b, k_b, i_l, j_l] = \sum_{k_l} A[i_b, k_b, i_l, k_l] * B[k_b, j_b, k_l, j_l];$$

Ainsi, chaque réduction du programme introduit une nouvelle variable temporaire d'accumulation. On remarque que, par le simple fait d'introduire cette variable temporaire d'accumulation, les propriétés d'associativité et de commutativité de l'opérateur de réduction ont été utilisées. Ainsi, cette transformation ne préserve pas l'équivalence d'Herbrand, et est donc une *transformation sémantique*.

Ces nouvelles variables n'apparaissant pas dans le tuilage spécifié en entrée par l'utilisateur, on doit adapter cette spécification afin d'en tenir compte, tout en prenant garde à respecter la légalité du tuilage.



Original program:
$(\forall 0 \leq i < N)\ x[i] = (b[i] - \sum_{k<i} L[i,k] \times x[k])/L[i,i]$

FIGURE A.4: Dependances entre les tuiles de $TempRed$ et les tuiles de $x/temp$.

Par exemple, la Figure A.4 montre les tuiles d'un programme qui résout une équation de la forme $L.\vec{x} = \vec{b}$ où $\vec{x}$ est l'inconnue et L une matrice triangulaire inférieure. Notez que la variable $TempRed$ dépend des valeurs précédentes de x, et donc que la dernière tuile de $TempRed$ admet une dépendance cyclique avec la tuile calculant les $x[i_b, \bullet]$. Ainsi, le sous-système qui calcule les valeurs $x[i_b, \bullet]$ doit aussi calculer les valeurs de $TempRed[i_b, i_b, \bullet]$ $(k_b = i_b)$, et on peut avoir un autre sous-système qui calcule les autre tuiles de $TempRed$.

En analysant les dépendances entre tuiles, nous détectons quelles tuiles de $TempRed$ peuvent être tuilée séparément sans introduire de dépendances cycliques entre tuiles. Ces tuiles peuvent former leur propres sous-systèmes, tandis que le calcul des autres tuiles doivent être inclus dans des sous-systèmes existants.

A.5 Reconnaissance de templates

Dans cette section, nous introduisons un algorithme de reconnaissance de template, qui est une adaptation de l'algorithme d'équivalence de programme dont les concepts principaux ont été rapidement décrit dans la Section A.2.

Algorithme de reconnaissance de template Commençons par décrire l'algorithme de reconnaissance de template. Cet algorithme prend en entrée un programme et un template, et détermine si le template matche le programme (c'est à dire, s'il existe des valeurs des paramètres et des entrées du template qui rend le template équivalent au programme). De plus, si le template matche, des valeurs du paramètres et des entrées du template sont inférés automatiquement.

La notion d'équivalence utilisée est l'équivalence de Herbrand, mais va être enrichie plus tard afin de gérer les propriétés sémantiques présentent en algèbre linéaire.

La première étape de l'algorithme consiste à construire l'automate d'équivalence du problème de reconnaissance de template. L'automate d'équivalence que l'on considère dans l'algorithme de reconnaissance de template est légèrement modifié par rapport à l'algorithme d'équivalence de programme. En l'occurence, on modifie la notion d'état final de réussite: dans le cas de la reconnaissance de template, un état final de réussite est tout état final de la forme " $\cdots = I'$ ", avec I' une entrée du template. En effet, intuitivement, une entrée de template peut potentiellement correspondre à n'importe quelle expression du programme.

Une fois l'automate d'équivalence construit, la seconde étape de l'algorithme de reconnaissance de template consiste à extraire les contraintes sur les entrées du template. Cela est fait en calculant les ensembles d'accessibilité de chaque état final de l'automate de template. L'ensemble d'accessibilité d'un état est l'ensemble des indices $(\vec{i}, \vec{i'})$ tels que il existe un chemin dans l'automate partant de l'état initial et arrivant sur l'état considéré avec ces valeurs d'indices.

Pour les états finaux d'échec, les ensemble d'accessibilité correspondants doivent être vides (c'est à dire, ces états ne doivent pas être accessibles). Pour les états finaux de réussite, les contraintes sont de la forme:

$$(\forall(\vec{i}, \vec{i'}) \in \mathcal{S}) \quad I'[\vec{i'}] = Expr_k[\vec{i}]$$

où I' est une entrée de template et $\mathcal{S}$ est l'ensemble d'accessibilité de l'état final de réussite. Notez qu'une clôture transitive peut être nécessaire pour calculer cet ensemble d'accessibilité et donc que des sur-approximations peuvent intervenir lors de ce calcul.

La troisième et dernière étape de l'algorithme de reconnaissance de template consiste à résoudre les contraintes que l'on vient d'extraire, afin d'en déduire les valeurs des entrées du template. Pour cela, on classifie les contraintes suivant l'entrée de template I' qu'il fait intervenir et on examine les ensembles d'accessibilité. Pour chaque entrée de template, deux situations peuvent arriver:

- Pour chaque valeur de la variable d'entrée du template $I'[\vec{i'}]$, il n'y a qu'une seule expression $Expr_k[\vec{i}]$ du programme qui lui est associée *via* une des contraintes. Dans ce cas, on peut simplement construire une disjonction de cas entre les différentes valeurs associées à l'entrée de template I'.

- De multiples valeurs $Expr_k[\vec{i}]$ sont associés à la même valeur de la variable d'entrée du template $I'[\vec{i'}]$. Dans ce cas, on doit d'abord vérifier que ces valeurs sont équivalentes, *via* un appel à un algorithme d'équivalence de programme. En pratique, le coût de cet appel est raisonnable. Si ce n'est pas le cas, cela veut dire que la variable d'entrée du template doit prendre deux valeurs différentes en même temps, ce qui est impossible. D'où on conclue que le template ne matche pas. Si c'est le cas, on choisit une des deux valeurs (le choix n'étant pas important du fait de l'équivalence) et construit la disjonction comme vu dans le cas précédent.

A propos de l'inférence des paramètres du template, des contraintes sur les paramètres sont obtenues depuis plusieurs endroits dans l'algorithme: les domaines des variables de sortie doivent correspondre, ce qui introduit des égalités entre paramètres du template et du programme. Certains états finaux d'échec peuvent n'être accessibles que pour certaines valeurs de paramètre de template, donc la négation de ces contraintes doit être prise. De même, lors des appels à un algorithme d'équivalence, certaines expressions ne sont équivalentes que pour certaines valeurs de paramètres. Enfin, lorsque l'on compare deux réductions dans l'automate d'équivalence, on demande que les nombres d'éléments sommés soit égaux (ce qui peut introduire des contraintes d'égalité entre paramètres).

Si, après avoir regroupé toutes ces contraintes sur les paramètres, elles ne sont pas satisfiables, on conclut que le template ne matche pas le programme. Il se peut aussi que la valeur des paramètres du template ne soit pas fixée: dans ce cas, on fait la supposition que plus les valeurs des paramètres du template sont grandes, plus le template fait de calculs, et nous sélectionnons la valeur maximale des paramètres du template.

L'Exemple 5.4 Page 128 illustre un grand nombre de mécanismes de cet algorithme.

Gestion des propriétés sémantiques Nous proposons plusieurs extensions à notre algorithme de reconnaissance de template, afin de gérer des propriétés sémantiques usuellement rencontrées en algèbre linéaire.

Les propriétés d'associativité et de commutativité des opérateurs binaires sont gérées pendant la construction de l'automate de la manière suivante. Si un état compare deux expressions "$A_1 + \cdots + A_k = B_1 + \cdots + B_k$", l'algorithme matche A_i avec B_i par défaut. Cependant, du fait des propriétés d'associativité et de commutativité, n'importe quel A_i peut être matché à n'importe quel B_j, et on a autant de possibilité de matchage que de permutations. Ainsi, on génère toutes les versions de l'automate et applique le reste de l'algorithme à ces versions. Si une version de l'automate d'équivalence arrive à matcher le template au programme, on arrête le parcours des versions et retourne le résultat que l'on vient d'obtenir. Si aucune version de l'automate d'équivalence arrive à matcher le template, l'algorithme conclut que le template ne matche pas.

La propriété de distributivité est gérée en ayant différentes versions du template: une où les expressions sont factorisées complètement, et une où les expressions sont distribuées complètement. Le reste des propriétés (élément neutre, absorbant, gestion des soustractions et divisions, ...) sont des modifications locales, et sont gérées via des règles de réécritures appliquées avant d'exécuter le reste de l'algorithme.

A.6 Reconnaissance de sous-calculs

Cette section combine les contributions précédentes en un procédé de reconnaissance d'opérations d'algèbre linéaire en tant que sous-calcul d'un programme polyédrique. Nous introduisons d'abord la librairie de templates qui correspond aux opérations que l'on essaye de reconnaître. Cette librairie est une des composantes du procédé, que l'on introduit par la suite. Enfin, nous présentons quelques résultats expérimentaux et discutons de ses performances.

Librairie de templates Nous construisons une librairie de template qui correspond aux opérations décrites dans la spécification BLAS. Lors de la construction de cette librairie, l'objectif principal est de réduire le plus possible le nombre de templates.

Par exemple, considérons l'opération DGEMM: $C \leftarrow \alpha.A^X.B^X + \beta.C$, où A et B sont des matrices, $A^X = A$ ou A^T et α, β sont des scalaires. Si on n'effectue aucun traitement préliminaire sur cet opération, nous devons implémenter de multiples versions de ce template, pour prendre en compte les valeurs spéciales de α et β, ou de la présence d'une transposée. À la place, nous décomposons DGEMM comme une combinaison des opérations fondamentales suivantes: $C \leftarrow A.B$ (multiplication de matrices), $C \leftarrow A^T$ (transposition de matrice), $C \leftarrow A + B$ (addition de matrices) et $C \leftarrow \alpha.A$ avec $\alpha \neq 0, 1$ (multiplication d'une matrice par un scalaire). Ainsi, 4 templates suffisent pour couvrir toutes les variantes de DGEMM.

Afin de limiter le nombre de template à comparer avec un sous-système donné, on classifie chaque template selon leur opération scalaire correspondante. L'opération scalaire d'un template est le calcul obtenu quand l'on impose que les tailles des matrices et vecteurs considérés par le template sont égales à 1. L'opération scalaire d'un sous-système doit être identique à l'opération scalaire d'un template (vu que la comparaison se fait directement entre ces opérations, dans le cas particulier où les tailles de leurs matrices et vecteurs sont 1).

Un cas particulier est l'opération de transposition, qui n'a pas d'opération scalaire associée et peut être potentiellement appliquée à n'importe quel endroit. Aussi, ce template présente le risque d'être reconnu indéfiniment (du fait de sa propriété d'idempotence). Ainsi, le template correspondant à l'opération de transpose est testé uniquement après tous les autres templates pouvant correspondre au sous-système considéré. De plus, si le dernier template reconnu est un transposition, on ne cherchera pas à re-reconnaître une transposition immédiatement après.

Un dernier aspect à considérer est l'ordre de comparaison des templates. Ce dernier est effectué du template le moins général au plus général. Cela permet l'opportunité de reconnaître, par exemple, une multiplication de matrice symétrique $C \leftarrow S.B$ avant de tester une multiplication de matrices générale $C \leftarrow A.B$, et donc avoir des informations plus riches sur les opérations reconnues.

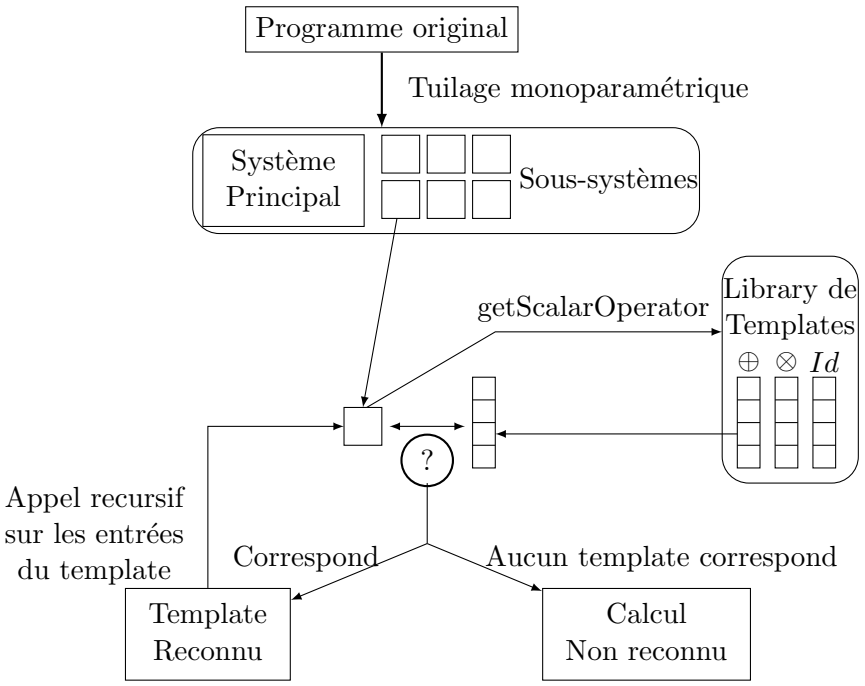


FIGURE A.5: Procédé de reconnaissance de template.

Procédé de reconnaissance d'opérations d'algèbre linéaire Le procédé de reconnaissance d'opérations d'algèbre linéaire est décrit dans la Figure A.5. Nous appliquons d'abord la transformation de tuilage monoparamétrique, puis nous considérons chaque sous-systèmes produit indépendamment. La librairie de template est triée en fonction de l'opération scalaire correspondant à chaque template. Chaque sous-système est analysé afin de détecter son opération scalaire, qui est utilisée pour récupérer la liste de templates correspondant à cette opération. Ensuite, on compare le sous-système avec chaque template successivement de cette liste. Deux situations peuvent se produire: soit aucun template ne correspond, et le calcul n'est pas reconnu, soit un template correspond. Dans le dernier cas, on récupère l'expression correspondant à chaque entrée du template et on appelle récursivement l'algorithme de reconnaissance de template sur chacune d'entre elles.

Résultats expérimentaux Nous avons évalué notre procédé de reconnaissance d'opérations d'algèbre linéaire sur des applications d'algèbre linéaire et hors du domaine de l'algèbre linéaire.

Dans le cas des applications d'algèbre linéaire (inversion de matrices symétriques définies positives, et résolution d'équation de Silvester), nous sommes parvenu à reconnaître la quasi-totalité des calculs comme une combinaison de templates de notre librairie. Les calculs non reconnus correspondent soit à un calcul de clôture transitive qui prend trop de temps, ou à une opération qui n'est pas présente dans la librairie (parce qu'elle est trop spécifique). Dans les deux cas, les sous-systèmes les plus fréquemment appelés sont complètement reconnus.

Dans le cas des applications hors du domaine de l'algèbre linéaire (Algebraic Path Problem (APP), McCaskill qui est une application de bio-informatique), une bonne partie du programme ne correspond pas à des opérations d'algèbre linéaire. Dans le cas de l'APP, 5 des 6 sous-systèmes les plus fréquemment appelés ont été complètement reconnus. Dans le cas de McCaskill, notre procédé reconnaît presque aucun sous-système comme opération d'algèbre linéaire. Cela est du au fait que la majorité des calculs des sous-système sont des opérations sur des tenseurs: la librairie de template que l'on a choisit est donc inadaptée à cette application, mais la décomposition arrive tout de même à isoler les calculs de l'application de manière pertinente.

A.7 Conclusion

Contributions Dans ce travail de thèse, nous avons présenté un mécanisme de reconnaissance d'opérations d'algèbre linéaire présents dans un programme polyédrique. Afin de construire ce mécanisme, trois contributions sont faites dans ce travail de thèse: une transformation de programme appelée tuilage monoparamétrique, un algorithme de reconnaissance de template et le mécanisme en lui-même.

A propos du tuilage monoparamétrique, cette transformation de programme est un tuilage dont les tailles de tuile sont des multiples d'un paramètre commun de taille de tuile. Cette transformation est à mi-chemin entre le tuilage à taille fixe et le tuilage paramétrique: en effet, le tuilage monoparamétrique est une transformation polyédrique, tout en permettant une forme limitée de paramétrisation des tailles de tuile. Nous avons étudié cette transformation en deux

parties. La première partie de la transformation, nommée partitionnement monoparamétrique, est juste une réindexation de tous les espaces d'un programme, de manière à introduire les indices tuilés et locaux. La seconde partie de la transformation distribue et isole les calculs de chaque tuile dans des sous-programmes séparés.

L'algorithme de reconnaissance de template est une autre des composantes principales du mécanisme. Cet algorithme est une extension d'un algorithme d'équivalence de programme proposé précédemment par Barthou et al [8]. Cet algorithme a été étendu afin de gérer les propriétés sémantiques communément rencontrées dans le domaine de l'algèbre linéaire. Cet algorithme de reconnaissance de template est le premier algorithme qui est suffisament puissant pour reconnaître n'importe quel opérations de BLAS.

Finallement, nous utilisons ces deux contributions pour construire un mécanisme de reconnaissance d'opérations d'algèbre linéaire. Un tuilage monoparamétrique est d'abord utilisé pour séparer le calcul selon leur tuiles, puis le calcul de chaque tuile est considéré séparément, afin de les reconnaître comme une combinaison de template. Les templates proviennent d'une librairie inspirée par BLAS. Lorsque l'on utilise notre mécanisme sur des applications d'algèbre linéaire, la majorité des calculs sont reconnus. L'application de ce mécanisme sur des applications qui ne sont pas du domaine d'algèbre linéaire est moins efficace, mais arrive tout de même à reconnaître des portions de calcul fréquemment utilisées.

Bibliography

[1] Aravind Acharya and Uday Bondhugula. Pluto+: Near-complete modeling of affine transformations for parallelism and locality. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2015.

[2] Wolfgang Achtziger and Karl-Heinz Zimmermann. Finding quadratic schedules for affine recurrence equations via nonsmooth optimization. *Journal of VLSI Signal Processing Systems*, 25(3):235–260, July 2000.

[3] Christophe Alias. *Program Optimization by Template Recognition and Replacement*. PhD thesis, Université de Versailles, December 2005.

[4] Saman Prabhath Amarasinghe. *Parallelizing Compiler Techniques Based on Linear Inequalities*. PhD thesis, Stanford University, 1997.

[5] Corinne Ancourt and François Irigoin. Scanning polyhedra with DO loops. *SIGPLAN Notices*, 26(7):39–50, April 1991.

[6] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[7] Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. Tiling stencil computations to maximize parallelism. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 1–11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[8] Denis Barthou, Paul Feautrier, and Xavier Redon. On the equivalence of two systems of affine recurrence equations. Technical Report RR-4285, INRIA, 2001.

[9] Muthu Manikandan Baskaran, Albert Hartono, Sanket Tavarageri, Thomas Henretty, J. Ramanujam, and P. Sadayappan. Parameterized tiling revisited. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 200–209, New York, NY, USA, 2010. ACM.

[10] Cédric Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, pages 7–16, Juan-les-Pins, France, September 2004.

[11] Geoffrey Belter, E. R. Jessup, Ian Karlin, and Jeremy G. Siek. Automating the generation of composed linear algebra kernels. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 59:1–59:12, New York, NY, USA, 2009. ACM.

[12] S. Bhansali and J. R. Hagemeister. A pattern-matching approach for reusing software libraries in parallel systems. In *First International Workshop on Knowledgebased Systems for the ReUse of Program Libraries*, 1995.

[13] Paolo Bientinesi, John A. Gunnels, Margaret E. Myers, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. The science of deriving dense linear algebra algorithms. *ACM Transactions on Mathematical Software*, 31(1):1–26, March 2005.

[14] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology. In *Proceedings of the 11th International Conference on Supercomputing*, ICS '97, pages 340–347, New York, NY, USA, 1997. ACM.

[15] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral program optimization system. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2008.

[16] Hamidreza Chitsaz, Raheleh Salari, S. Cenk Sahinalp, and Rolf Backofen. A partition function algorithm for interacting nucleic acid strands. *Bioinformatics*, 25(12):i365–i373, 2009.

[17] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, and Glenn Reinman. Architecture support for accelerator-rich CMPs. In *49th Design Automation Conference (DAC)*, pages 843–849, June 2012.

[18] Wonnacott David, Tian Jin, and Allison Lake. Automatic tiling of "mostly-tileable" loop nests. In *Proceedings of the 5th International Workshop on Polyhedral Compilation Techniques*, Amsterdam, The Netherlands, January 2015.

[19] Florent de Dinechin, Patrice Quinton, and Tanguy Risset. Structuration of the Alpha language. In *Massively Parallel Programming Models*, pages 18–24. IEEE, 1995.

[20] R.H. Dennard, V.L. Rideout, E. Bassous, and A.R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, Oct 1974.

[21] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. *SIGARCH Computer Architecture News*, 39(3):365–376, June 2011.

[22] Paul Feautrier. Array expansion. In *Proceedings of the 2nd International Conference on Supercomputing*, ICS'88, pages 429–441, New York, NY, USA, 1988. ACM.

[23] Paul Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988.

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

[25] Paul Feautrier. Some efficient solutions to the affine scheduling problem: I. one-dimensional time. *International Journal of Parallel Programming*, 21(5):313–348, October 1992.

[26] Paul Feautrier. The power of polynomials. In Alexandra Jimborean and Alain Darte, editors, *5th International Workshop on Polyhedral Compilation Techniques (IMPACT'15)*, Amsterdam, Netherlands, January 2015.

[27] Pierrick Gachet, Christophe Mauras, Patrice Quinton, and Yannick Saouter. Alpha du centaur: A prototype environment for the design of parallel regular alorithms. In *Proceedings of the 3rd International Conference on Supercomputing*, ICS'89, pages 235–243, New York, NY, USA, 1989. ACM.

[28] Tobias Grosser, Albert Cohen, Justin Holewinski, P. Sadayappan, and Sven Verdoolaege. Hybrid hexagonal/classical tiling for GPUs. In *12th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '14*, page 66, February 2014.

[29] Armin Grosslinger, Martin Griebl, and Christian Lengauer. Introducing non-linear parameters to the polyhedron model. In *Proceedings of the 11th Workshop on Compilers for Parallel Computers (CPC 2004)*, pages 1–12, 2004.

[30] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. FLAME: Formal linear algebra methods environment. *ACM Transactions on Mathematical Software*, 27(4):422–455, December 2001.

[31] C.A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of computing. MIT Press, 1992.

[32] A. Hartono, M.M. Baskaran, J. Ramanujam, and P. Sadayappan. Dyntile: Parametric tiled loop generation for parallel execution on multicore processors. In *International Symposium on Parallel Distributed Processing (IPDPS),*, pages 1–12, April 2010.

[33] Albert Hartono, Muthu Manikandan Baskaran, Cédric Bastoul, Albert Cohen, Sriram Krishnamoorthy, Boyana Norris, J. Ramanujam, and P. Sadayappan. Parametric multi-level tiling of imperfectly nested loops. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 147–157, New York, NY, USA, 2009. ACM.

[34] Guillaume Iooss, Christophe Alias, and Sanjay Rajopadhye. On program equivalence with reductions. In Markus Muller-Olm and Helmut Seidl, editors, *Static Analysis Symposium*,

volume 8723 of *Lecture Notes in Computer Science*, pages 168–183. Springer International Publishing, 2014.

[35] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'88, pages 319–329, January 1988.

[36] C. Karfa, K. Banerjee, D. Sarkar, and C. Mandal. Verification of loop and arithmetic transformations of array-intensive behaviors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(11):1787–1800, Nov 2013.

[37] Richard M Karp, Raymond E Miller, and Shmuel Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3):563–590, 1967.

[38] Christoph W. Kessler. Pattern-driven automatic parallelization. *Scientific Programming*, 5(3):251–274, August 1996.

[39] DaeGon Kim and Sanjay Rajopadhye. Efficient tiled loop generation: D-tiling. In *Proceedings of the 22Nd International Conference on Languages and Compilers for Parallel Computing*, LCPC'09, pages 293–307, Berlin, Heidelberg, 2010. Springer-Verlag.

[40] DaeGon Kim and Sanjay V. Rajopadhye. Parameterized tiling for imperfectly nested loops. Technical Report CS-09-101, Colorado State University, February 2009.

[41] DaeGon Kim, Lakshminarayanan Renganarayanan, Dave Rostron, Sanjay V. Rajopadhye, and Michelle Mills Strout. Multi-level tiling: M for the price of one. In *Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing, SC 2007, November 10-16, 2007, Reno, Nevada, USA*, page 51, 2007.

[42] Induprakas Kodukula, Nawaaz Ahmed, and Keshav Pingali. Data-centric multi-level blocking. *SIGPLAN Notices*, 32(5):346–357, May 1997.

[43] Martin Kong, Antoniu Pop, Louis-Noël Pouchet, R. Govindarajan, Albert Cohen, and P. Sadayappan. Compiler/runtime framework for dynamic dataflow parallelization of tiled programs. *ACM Transactions on Architecture and Code Optimization*, 11(4):61:1–61:30, January 2015.

[44] Athanasios Konstantinidis, Paul H.J. Kelly, J. Ramanujam, and P. Sadayappan. Parametric gpu code generation for affine loop programs. In Calin Cascaval and Pablo Montesinos, editors, *Languages and Compilers for Parallel Computing*, volume 8664 of *Lecture Notes in Computer Science*, pages 136–151. Springer International Publishing, 2014.

[45] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. *SIGPLAN conference of Programing Language Design and Implementation*, 42(6):235–244, June 2007.

[46] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, September 1979.

[47] H. Le Verge. Reduction operators in alpha. In Daniel Etiemble and Jean-Claude Syre, editors, *PARLE'92 Parallel Architectures and Languages Europe*, volume 605 of *Lecture Notes in Computer Science*, pages 397–411. Springer Berlin Heidelberg, 1992.

[48] Hervé Le Verge. *Un environnement de transformations de programmes pour la synthèse d'architectures régulières*. PhD thesis, Université de Rennes 1, October 1992.

[49] Nuno P. Lopes and José Monteiro. Automatic equivalence checking of programs with uninterpreted functions and integer arithmetic. *International Journal on Software Tools for Technology Transfer*, pages 1–16, 2015.

[50] David B. Loveman. Program improvement by source-to-source transformation. *Journal of the ACM*, 24(1):121–145, January 1977.

[51] Michael J. Lyons, Mark Hempstead, Gu-Yeon Wei, and David Brooks. The accelerator store: A shared memory framework for accelerator-based systems. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):1–22, January 2012.

[52] Vijay Menon and Keshav Pingali. High-level semantic optimization of numerical codes. In *Proceedings of the 13th International Conference on Supercomputing*, ICS'99, pages 434–443, New York, NY, USA, 1999. ACM.

[53] Vijay Menon, Keshav Pingali, and Nikolay Mateev. Fractal symbolic analysis. *ACM Transaction on Programming Languages and Systems*, 25(6):776–813, November 2003.

[54] R. Metzger and Z. Wen. *Automatic Algorithm Recognition: A New Approach to Program Optimization*. MIT Press, 2000.

[55] Gordon E. Moore. Progress in digital integrated electronics. In *International Electron Devices Meeting 1975*, volume 21, pages 11–13, 1975.

[56] David Padua, Denis Barthou, and Alexandre X. Duchateau. Hydra: Automatic algorithm exploration from linear algebra equations. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO'13, pages 1–10. IEEE Computer Society, 2013.

[57] Shlomit S. Pinter and Ron Y. Pinter. Program optimization and parallelization using idioms. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'91, pages 79–92, New York, NY, USA, 1991. ACM.

[58] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In Bernhard Steffen, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *Lecture Notes in Computer Science*, pages 151–166. Springer Berlin Heidelberg, 1998.

[59] Michael Powell, Se-Hyun Yang, Babak Falsafi, Kaushik Roy, and T. N. Vijaykumar. Gated-vdd : A circuit technique to reduce leakage in deep-submicron cache memories. In *Proceedings of the 2000 International Symposium on Low Power Electronics and Design (ISLPED '00)*, pages 90–95, 2000.

[60] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[61] Fabien Quilleré, Sanjay Rajopadhye, and Doran Wilde. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming*, 28(5):469–498, October 2000.

[62] Patrice Quinton, Sanjay Rajopadhye, and Doran Wilde. Deriving imperative code from functional programs. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, FPCA'95, pages 36–44, New York, NY, USA, 1995. ACM.

[63] Patrice Quinton and Vincent van Dongen. The mapping of linear recurrence equations on regular arrays. *Journal of VLSI signal processing systems for signal, image and video technology*, 1(2):95–113, 1989.

[64] Sanjay V. Rajopadhye, S. Purushothaman, and Richard M. Fujimoto. On synthesizing systolic arrays from recurrence equations with linear dependencies. In Kesav V. Nori, editor, *Foundations of Software Technology and Theoretical Computer Science*, volume 241 of *Lecture Notes in Computer Science*, pages 488–503. Springer Berlin Heidelberg, 1986.

[65] Xavier Redon and Paul Feautrier. Detection of recurrences in sequential programs with loops. In Arndt Bode, Mike Reeve, and Gottfried Wolf, editors, *PARLE'93 Parallel Architectures and Languages Europe*, volume 694 of *Lecture Notes in Computer Science*, pages 132–145. Springer Berlin Heidelberg, 1993.

[66] Xavier Redon and Paul Feautrier. Detection of scans in the polytope model. *Parallel Algorithms and Applications*, 15(3-4):229–263, 2000.

[67] D. A. Reed, L. M. Adams, and M. L. Partick. Stencils and problem partitionings: Their influence on the performance of multiple processor systems. *IEEE Transactions on Computers*, 36(7):845–858, July 1987.

[68] Lakshminarayanan Renganarayanan, DaeGon Kim, Sanjay V. Rajopadhye, and Michelle Mills Strout. Parameterized tiled loops for free. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 405–414, June 2007.

[69] Lakshminarayanan Renganarayanan, DaeGon Kim, Sanjay V. Rajopadhye, and Michelle Mills Strout. Parameterized loop tiling. *ACM Trans. Program. Lang. Syst.*, 34(1):3, 2012.

[70] Yannick Saouter and Patrice Quinton. Computability of recurrence equations. *Theoretical Computer Science*, 116(2):317–337, August 1993.

[71] Shigeyuki Sato and Hideya Iwasaki. Automatic parallelization via matrix multiplication. *SIGPLAN Notice*, 46(6):470–479, June 2011.

[72] Robert R. Schaller. Moore's law: past, present and future. *IEEE Spectrum*, 34(6):52–59, Jun 1997.

[73] Markus Schordan, Pei-Hung Lin, Dan Quinlan, and Louis-Noël Pouchet. Verification of polyhedral optimizations with constant loop bounds in finite state space computations. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications*, volume 8803 of *Lecture Notes in Computer Science*, pages 493–508. Springer Berlin Heidelberg, 2014.

[74] Muhammad Shafique, Siddharth Garg, Tulika Mitra, Sri Parameswaran, and Jörg Henkel. Dark silicon as a challenge for hardware/software co-design: Invited special session paper. In *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis*, CODES'14, pages 1–10. ACM, 2014.

[75] K.C. Shashidhar, Maurice Bruynooghe, Francky Catthoor, and Gerda Janssens. Verification of source code transformations by program equivalence checking. In Rastislav Bodik, editor, *Compiler Construction*, volume 3443 of *Lecture Notes in Computer Science*, pages 221–236. Springer Berlin Heidelberg, 2005.

[76] Jeremy G. Siek, Ian Karlin, and E. R. Jessup. Build to order linear algebra kernels. In *IEEE International Symposium on Parallel and Distributed Processing, 2008. IPDPS 2008*, pages 1–8, April 2008.

[77] Daniele G. Spampinato and Markus Püschel. A basic linear algebra compiler. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 23–32. ACM, 2014.

[78] Sanket Tavarageri, Albert Hartono, Muthu Baskaran, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan. Parametric tiling of affine loop nests. In *Proceedings of the 15th Workshop on Compilers for Parallel Computers*, 2010.

[79] Sven Verdoolaege. isl: An integer set library for the polyhedral model. In Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama, editors, *Lecture Notes in Computer Science, International Congress on Mathematical Software (ICMS 2010)*, pages 299–302. Springer, September 2010.

[80] Sven Verdoolaege, Gerda Janssens, and Maurice Bruynooghe. Equivalence checking of static affine programs using widening to handle recurrences. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 34(3):1–35, November 2012.

[81] Sven Verdoolaege, Gerda Janssens, and Maurice Bruynooghe. Equivalence checking of static affine programs using widening to handle recurrences. *ACM Transactions on Programming Languages and Systems*, 34(3):1–35, November 2012.

[82] Richard Vuduc, James W Demmel, and Katherine A Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Institute of Physics Publishing*, 2005.

[83] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, SC '98, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society.

[84] Doran Wilde and Sanjay Rajopadhye. The naive execution of affine recurrence equations. In *Proceedings of the IEEE International Conference on Application Specific Array Processors*, ASAP '95, pages 1–, Washington, DC, USA, 1995. IEEE Computer Society.

[85] David Wonnacott. Time skewing for parallel computers. In *In Proceedings of the Twelfth Workshop on Languages and Compilers for Parallel Computing*, pages 477–480. Springer-Verlag, 1999.

[86] David Wonnacott. Achieving scalable locality with time skewing. *International Journal of Parallel Programming*, 30(3):181–221, 2002.

[87] Jingling Xue. *Loop Tiling for Parallelism.* Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[88] Kamen Yotov, Xiaoming Li, Gang Ren, Michael Cibulskis, Gerald DeJong, Maria Garzaran, David Padua, Keshav Pingali, Paul Stodghill, and Peng Wu. A comparison of empirical and model-driven optimization. *SIGPLAN Notices*, 38(5):63–76, May 2003.

[89] Tomofumi Yuki, Gautam Gupta, DaeGon Kim, Tanveer Pathan, and Sanjay V. Rajopadhye. Alphaz: A system for design space exploration in the polyhedral model. In *Languages and Compilers for Parallel Computing, 25th International Workshop, LCPC 2012*, pages 17–31, September 2012.

[90] Yun Zou and Sanjay Rajopadhye. Scan detection and parallelization in "inherently sequential" nested loop programs. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO'12, pages 74–83. ACM, 2012.