HAL
open science

Automatic derivation of I/O complexity bounds for affine programs

Auguste Olivry

To cite this version:

Auguste Olivry. Automatic derivation of I/O complexity bounds for affine programs. Computational Complexity [cs.CC]. Université Grenoble Alpes [2020-..], 2022. English. NNT : 2022GRALM022 . tel-03877029

HAL Id: tel-03877029

https://theses.hal.science/tel-03877029

Submitted on 29 Nov 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

Spécialité : Informatique

Arrêté ministériel : 25 mai 2016

Présentée par

Auguste OLIVRY

Thèse dirigée par **Fabrice RASTELLO**

préparée au sein du **Laboratoire d'Informatique de Grenoble** dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

Calcul automatique de bornes sur la complexité I/O de programmes affines

Automatic derivation of I/O complexity bounds for affine programs

Thèse soutenue publiquement le **8 juin 2022**,
devant le jury composé de :

Monsieur FABRICE RASTELLO
Directeur de recherche, Inria Centre Grenoble Rhône-Alpes,
Directeur de thèse
Monsieur SEBASTIAN HACK
Professeur, Universität des Saarlandes, Rapporteur
Madame LAURA GRIGORI
Directrice de recherche, Inira Centre de Paris, Rapportrice
Monsieur UDAY BONDHUGULA
Professeur associé, Indian Institute of Science, Bangalore, Examinateur
Monsieur SVEN VERDOOLAEGE
Docteur en sciences, Cerebras Systems, Examinateur
Madame LAURE GONNORD
Professeure des Universités, Grenoble INP, Examinatrice,
Présidente du jury

Résumé

L'évaluation de la complexité d'un algorithme est une étape importante lors du développement d'une application, de par son impact sur son temps d'exécution et sa consommation énergétique. La complexité arithmétique, qui est le nombre d'opérations effectuées par un programme, est en général facile à caractériser pour un programme affine, composé de boucles simples avec des bornes statiques. La complexité de mouvement de données (ou complexité I/O) est plus complexe à évaluer. En effet, elle se réfère à la quantité minimale de données à transférer entre une mémoire lente de grande capacité, et une mémoire rapide de capacité limitée, et ce *en considérant tous les ordonnancements valides des opérations*.

Dans cette thèse, nous présentons IOOpt, un outil automatisé capable de calculer des bornes symboliques sur la complexité I/O de programmes affines. Étant donné une description d'un programme affine, il génère : 1. une borne inférieure sur la complexité I/O sous la forme d'une expression symbolique dépendant de la taille de la mémoire rapide et des paramètres du programme ; 2. une borne supérieure qui permet d'évaluer la précision de la borne inférieure 3. une recommandation de tuilage (taille des tuiles et permutation des boucles) qui permet d'atteindre la borne supérieure. Cet outil est disponible en open-source, et peut être testé sur la page `https://iocomplexity.corse.inria.fr/`. Le développement de cet outil a été possible grâce à la combinaison de résultats mathématiques avancés avec des outils puissants d'analyse de programmes, tels la réprésentation polyédrique. Notre modèle de mouvement de données est notamment basé sur celui proposé par Hong&Kung (« Red-blue pebble game »), adapté pour pouvoir décomposer et recomposer des programmes complexes.

L'applicabilité de cette approche est démontrée en combinant des bornes symboliques sur un large panel de programmes affines. Dans de nombreux cas, les bornes trouvées par IOOpt sont optimales, ou proches de la borne optimale par un facteur constant.

Complexité I/O La complexité I/O est définie sur la base d'un modèle de mémoire à deux niveaux : une mémoire d'accès rapide mais de taille limitée, et une mémoire d'accès plus lent mais de capacité illimitée. Lors de l'exécution d'un programme, les données d'entrée se trouvent initialement dans la mémoire lente. Les données peuvent être déplacées entre les deux niveaux de mémoire, et doivent être dans la mémoire rapide pour être utilisées dans un calcul. Le *coût I/O* d'un programme, pour un ordre donné des opérations de calcul et de mémoire, est le nombre de transferts depuis la mémoire lente vers la mémoire rapide. La *complexité I/O* d'un programme est le coût I/O minimum qui peut être obtenu en réordonnant les opérations.

Il est en général impossible de calculer la complexité I/O exacte d'un programme car le nombre d'ordonnancements possible est combinatoire. Ceci motive la recherche de bornes inférieures et supérieures sur cette complexité.

Représentation des programmes Afin de pouvoir représenter les différents ordonnancements possibles, un programme est décrit par ses opérations, et par les dépendances de données entre celles-ci. Cela est naturellement représenté par une structure de graphe orienté, abrégé CDAG en anglais.

Un exemple de tel graphe pour un programme exécutant la multiplication de deux matrices carrées de taille N est présenté sur la figure 2.2b (page 6).

Bornes inférieures Nous utilisons deux approches distinctes pour trouver des bornes inférieures sur la complexité I/O. L'une est basée sur la technique dite du « S-partitionnement », et l'autre sur le front d'onde (*wavefront*) de graphes. La combinaison de ces deux approches permet de traiter des programmes avec différents motifs de dépendances de données, et de les combiner au sein d'un même programme.

Nous détaillons ici l'idée générale de l'approche par partitionnement, qui est de raisonner sur l'*intensité opérationnelle* maximale d'une région du CDAG, c'est-à-dire la quantité maximale d'opérations de calcul qui peuvent être effectuées avec un nombre limité et déterminé d'opérations d'I/O. Plus précisément, étant donné un paramètre T (dont la valeur sera fixé ultérieurement), on cherche à borner la taille du plus grand sous-graphe qui puisse être exécuté avec au plus T *loads* (transferts de la mémoire lente à la mémoire rapide), en supposant que la mémoire rapide contient déjà S données. Par un simple argument de comptage, une telle borne se traduit en une borne inférieure sur le nombre de sous-séquences d'au moins T loads nécessaires pour exécuter le programme complet, ce qui ce traduit enfin en une borne sur le coût I/O du programme.

Pour borner la taille du sous-graphe, il faut remarquer que sa « frontière », composée des prédécesseurs des nœuds du sous-graphe qui n'en font pas partie, est de taille bornée par $S + T$. Il reste à borner la taille d'un sous-graphe en fonction de la taille de sa frontière, et pour cela on utilise le fait que les graphes de programmes affines sont très réguliers et géométriques. Ainsi, les opérations d'un programme constitué de trois boucles parfaitement imbriquées comme la multiplication de matrices peuvent être naturellement représentées par des points au sein d'un cube. La dernière pièce du raisonnement est fournie par une famille d'inégalités mathématiques (l'inégalité de Loomis-Whitney et sa généralisation, l'inégalité de Brascamp-Lieb), qui permet de borner la taille d'un ensemble en fonction de la taille de ses projections en dimension inférieure (voir figure 2.4, page 8).

Pour la multiplication de matrices, la borne obtenue par cette méthode est $\frac{2N^3}{\sqrt{S}}$. Cette borne est optimale : il est possible de trouver un ordonnancement des opérations qui atteint cette borne (en ignorant les termes négligeables).

Bornes supérieures La question qui se pose, une fois une borne inférieure établie, est de savoir si celle-ci est atteignable. Il s'agit pour cela d'exhiber un ordonnancement des opérations du programme le plus efficace possible en terme d'I/O. Autrement dit, une borne supérieure sur la complexité I/O d'un programme est simplement donnée par un ordonnancement valide. Ce qui rend ce problème difficile est qu'on cherche a obtenir des bornes paramétriques, valables pour toutes les valeurs des paramètres (dimensions des tableaux, capacité de la mémoire rapide), et qu'il est nécessaire de trouver un « bon » ordonnancement des opérations pour que la borne soit la plus proche possible de l'optimum.

La principale opération de réordonnancement qui permet d'optimiser les transferts de données est le *tuilage*. Il s'agit de partitionner le domaine d'itération d'un nid de boucles

en sous-domaines appelés tuiles, qui sont ensuite exécutées les unes après les autres. Des exemples de codes tuilés pour la multiplication de matrices se trouvent page 10, figure 2.5.

Notre approche pour trouver des bornes inférieures peut se résumer comme suit. En premier lieu, le programme est analysé pour isoler les parties où la transformation de tuilage peut être appliquée. Une première étape sélectionne ensuite un ensemble réduit de permutations des boucles de tuilage. Ceci procure pour chaque permutation un programme tuilé avec des tailles de tuiles paramétriques. Ensuite, un algorithme calcule une expression symbolique du coût I/O pour chaque permutation, ainsi que des contraintes sur la taille des tuiles. Enfin, ces expressions peuvent être utilisées pour optimiser numériquement la taille des tuiles pour des valeurs données des paramètres, ou pour en déduire une borne symbolique qui ne dépend plus des tailles de tuile mais uniquement de S et des paramètres du programme, et peut donc être directement comparée aux bornes inférieures calculées précédemment.

Sur la multiplication de matrices, notre algorithme génère un tuilage de taille $T_i \times T_j \times 1$, pour un coût I/O de $N^3/T_i + N^3/T_j + N^2$. La minimisation symbolique de cette expression sous la condition que toutes les données d'une tuile tiennent dans la mémoire rapide de taille S donnent une borne supérieure sur la complexité I/O de $\frac{2N^3}{\sqrt{S+1}-1} + N^3$. Cela correspond bien à la borne inférieure, à des termes asymptotiquement négligeables près.

Programmes affines et automatisation L'automatisation des techniques présentées ci-dessus est une contribution clé de cette thèse. Pour que cela soit possible, il a fallu choisir une classe de programme sur laquelle travailler, à savoir la classe des programmes *affines*. Cette classe constitue un bon compromis entre expressivité et applicabilité des techniques de calcul d'I/O, et dispose d'un riche écosystème d'outils d'analyse. Les programmes affines comprennent un large éventail d'algorithmes, notamment l'algèbre linéaire dense, les stencils, les convolutions, et de nombreux programmes utilisant la programmation dynamique.

Un programme affine se compose de boucles for (possiblement imbriquées) et d'expressions, qui opèrent sur des variables et des tableaux multidimensionnels. Les bornes des boucles sont des expressions affines des paramètres du programme et des indices de boucles qui les entourent. De même les accès aux tableaux sont des fonctions affines des indices de boucles et des paramètres. Ainsi, le flot de contrôle d'un programme affine est analysable statiquement.

Les programmes affines peuvent être analysés par les outils de compilation polyédrique, comme ISL et barvinok. Ces outils représentent les programmes par des points dans un espace multi-dimensionel, et peuvent effectuer des opérations sur des ensembles de points, analyser les dépendances de données entre instructions, compter le nombre de points dans un ensemble défini paramétriquement, entre autres. Cela est particulièrement utile pour notre calcul de bornes inférieures, pour mettre en évidence des dépendances régulières qui correspondent à des projections.

Expériences et résultats Ces algorithmes ont été implémentés dans des outils disponibles librement, et évalués sur un certain nombre de programmes, comprenant les 30 programmes de PolyBench, une suite d'évaluation de référence de programmes affines. Notamment, sur les contractions de tenseur et les convolutions, ainsi que sur plusieurs algorithmes d'algèbre linéaire, les bornes inférieures et supérieures sont égales asymptotiquement, ce qui permet de clore la question de l'évaluation de leur complexité I/O.

Abstract

Evaluating the complexity of an algorithm is an important step when developing applications, as it impacts both its time and energy performance. Computational complexity, which is the number of arithmetic operations of a program, is easy to characterize for affine programs, that are composed of simple loops with static bounds. Data movement (or, I/O) complexity is more complex to evaluate as it refers, *when considering all possible valid schedules*, to the minimum required number of I/O between a slow memory with large capacity, and a fast storage location with limited capacity.

This thesis presents IOOpt, a fully automated tool that automatically computes symbolic bounds on the I/O complexity of an affine program. Given a description of an affine program, it automatically computes: 1. a lower bound of the I/O complexity as a symbolic expression of the cache size and program parameters; 2. an upper bound that allows one to assess the tightness of the lower bound; 3. a tiling recommendation (loop permutation and tile sizes) that matches the upper bound. This tool is implemented as an open-source program, and a demonstration is available at `https://iocomplexity.corse.inria.fr/`. This was made possible by combining advanced mathematical results with powerful program analysis tools such as polyhedral representation. Our data movement model is notably based on the one designed by Hong&Kung (red-blue pebble game), adapted to be able de decompose and recompose complex programs.

We demonstrate the effectiveness of our approach by deriving symbolic bounds for a wide range of affine programs. In many cases, the bounds found by IOOpt can be proven optimal, or close to optimal by a constant factor.

Remerciements

Je tiens en premier lieu à remercier mon directeur de thèse, Fabrice Rastello, qui m'a suivi et soutenu depuis mon stage de L3 en 2015, jusqu'à l'aboutissement de ce doctorat, et m'a donné l'opportunité de faire mon stage de M2 aux États-Unis, stage où ont été posées les bases de ce qui est devenu ma thèse. Au delà du cadre scientifique, c'est également une relation humaine de grande qualité qui s'est tissée au cours de ces années, devant le tableau comme en montagne.

Je remercie les rapporteurs, Laura Grigori et Sebastian Hack, d'avoir pris le temps de se plonger dans mon manuscrit de thèse, et pour leurs retours très positifs. Un grand merci également aux autres membres du jury, en particulier Sven Verdoolaege pour ses retours sur le manuscrit, ainsi que Laure Gonnord et Uday Bondhugula.

Merci aux collègues de l'equipe CORSE, passés et présents : Fabian, les deux Nicolas, Théo, Chukri, Manu, Florent, Hugo et tous les autres. Mention spéciale à Guillaume pour avoir pris le temps de se plonger dans mon travail, sa collaboration a été très précieuse et ses relectures toujours pertinentes.

Merci à Imma Presseguer pour son aide administrative d'une efficacité inégalable.

Je remercie également les chercheurs avec qui j'ai eu l'occasion de collaborer, notamment P. Sadayappan avec qui les échanges ont été particulièrement fructueux et qui nous a accueilli à Columbus, Louis-Noël Pouchet qui m'a permis de venir en stage à Fort Collins, ainsi que Julien Langou et Atanas Rountev.

Merci à toutes les personnes, famille et ami·e·s, qui ont été présents toutes ces années : camarades de l'ENS, partenaires de grimpe du club de l'ENS et de Pic & Col, choristes du CUG, colocataires de confinements... Je ne les énumérerai pas de peur d'en oublier mais ils se reconnaîtront.

Un grand merci à mes parents, qui m'ont toujours soutenu avec bienveillance et ont su me transmettre un certain goût pour la science, ainsi qu'à toute ma famille.

Enfin, un merci tout spécial à Solène, qui a été à mes côtés pendant toute cette dernière année de thèse pas toujours facile, et sans qui la vie aurait bien moins de saveur.

Contents

1 Introduction **3**

2 Overview and Background **5**
2.1 I/O Complexity . . . 5
2.2 Program representation . . . 6
2.3 Lower bound derivation . . . 7
2.4 Upper bound derivation . . . 9
2.5 Affine programs and automation . . . 11

3 Lower Bounds **13**
3.1 Foundations . . . 13
3.1.1 CDAG . . . 13
3.1.2 A compact representation of the CDAG: the data-flow graph . . . 15
3.1.3 Partitioning . . . 16
3.1.4 Using projection to bound the cardinality of K-bounded sets . . . 18
3.1.5 DFG Paths . . . 19
3.2 CDAG decomposition . . . 20
3.2.1 Non-disjoint Decomposition Lemma . . . 20
3.2.2 Bounded combination . . . 22
3.2.3 Loop parametrization . . . 23
3.3 K-partition bound derivation . . . 24
3.3.1 Geometric embedding, DFG-paths and projections . . . 25
3.3.2 Finding paths . . . 27
3.3.3 Computing the lower bound . . . 28
3.3.4 Extensions: Small Dimensions and Reduction Detection . . . 30
3.4 Wavefront bound derivation . . . 34
3.4.1 Theoretical results . . . 34
3.4.2 Implementation . . . 35
3.5 Complete framework . . . 36
3.5.1 DFG construction . . . 36
3.5.2 Instances of parameter values . . . 37
3.5.3 Main algorithm . . . 37
3.6 Experimental evaluation . . . 41
3.6.1 Implementation . . . 41
3.6.2 Evaluation on PolyBench . . . 41
3.6.3 Parametric bounds for OI . . . 43

3.6.4 Complete lower bound formulae . . . 43
3.7 Related work . . . 45

4 Upper Bounds **51**
4.1 Background . . . 52
4.1.1 Class of programs . . . 52
4.1.2 Program representation . . . 53
4.2 Loop permutation and tiling . . . 53
4.2.1 Tiling transformation . . . 53
4.2.2 Sub-domains and reuse . . . 54
4.3 Cost model . . . 56
4.3.1 Single array . . . 56
4.3.2 Multiple arrays . . . 57
4.3.3 Extension to multiple memory levels . . . 58
4.3.4 Optimization problem . . . 58
4.4 Loop permutation selection . . . 58
4.4.1 Reuse for an array along a dimension . . . 59
4.4.2 Algorithm for permutation selection . . . 59
4.5 Putting it all together . . . 59
4.6 Experiments . . . 61
4.6.1 Benchmarks . . . 62
4.6.2 Symbolic upper bound expressions . . . 62
4.6.3 Comparison of upper and lower bounds for different cache sizes . . . 65
4.6.4 Implementation . . . 67
4.7 Related work . . . 67

5 Conclusion **70**
5.1 Summary of Results . . . 70
5.2 Limitations . . . 70
5.3 Future work . . . 71

Chapter 1

Introduction

Several factors can impact the execution time of an application on a computer: the most obvious one is the number of arithmetic operations, but the cost of moving data between CPU registers and memory can also be critical. For several decades now, CPUs have seen their computational power (FLOPS) improve at a higher rate than memory latencies and bandwidth. This means that, in order to use a processor to its full capacity, the minimization of data transfers between registers, CPU caches and RAM is getting increasingly important.

While latency costs can usually be hidden by hardware mechanisms like prefetching, the mismatch between computational power and memory bandwidth poses a fundamental limit. The balance between computation and data transfers can be expressed through a program's *operational intensity*, which is the ratio between the number of arithmetic operations and the volume of data transfers. This quantity can be compared with the *machine balance* of a CPU, which is the ratio between peak arithmetic performance and memory bandwidth. Only if a given program has an operational intensity that exceeds the machine balance can it exploit the full potential of the CPU; otherwise its execution speed will be limited by data transfers.

This motivates the need to study the *data movement complexity* of programs, also called *I/O complexity*. Similarly to computational complexity, which characterizes the number of operations made by a program, I/O complexity quantifies the volume of data movement operations to and from memory that are needed to perform a computation, for any valid schedule of operations. This quantity can be very challenging to compute precisely, and has been the subject of a long line of work, starting with the seminal paper of Hong & Kung [29] that first presented a method for deriving lower bounds on the I/O of programs. Concurrently, advanced methods in program analysis such as polyhedral compilation have been developed, enabling the analysis and representation of data dependencies of complex programs (precisely *affine* programs, that are composed of a combination of loop nests accessing multidimensional arrays, where loop bounds and array accesses are affine function of loop counters and program parameters). Elango et al. [21] first demonstrated how these two lines of work could be combined to automate the data movement analysis of affine programs, including stencils and linear algebra kernels.

In this thesis, we present a fully automated method for computing both lower and upper bounds on the I/O complexity of affine programs. It has been implemented as two open-source tools called IOLB and IOUB.

IOLB (for *I/O Lower Bounds*) takes as input an affine program and outputs a *parametric* lower bound *with scaling constant* on its I/O. This means that the bound is a symbolic expres-

sion, depending on program parameters (matrix dimensions for example) and on the size of the fast memory S, and that it does not rely on asymptotic notation. For instance, for the basic multiplication of square matrices of dimensions $N \times N$, the bound output by IOLB is $\frac{2N^3}{\sqrt{S}} - 3S$. As such, this bound can be evaluated for any value of N and S, and compared for example to a cache miss count for a specific execution. Our algorithm for lower bounds is able to decompose a program into sub-regions and to combine different methods on each sub-program to derive a bound as tight as possible. IOLB was extensively evaluated on 30 algorithms specified in PolyBench [39], with several first-time I/O lower bounds demonstrations on these programs.

Similarly, IOUB (for *I/O Upper Bounds*) automatically computes a symbolic expression of an upper bound on the data movement cost of a program, from an abstract representation of its iteration space and array accesses. To do so, it generates a set of candidate tiling transformations, using some pruning techniques to avoid combinatorial explosion, and associates to each of them a symbolic expression of the I/O cost (which is an upper bound in the general case but is actually exact in many cases). It then compares these expressions using numerical evaluation to single out the best ones. For concrete values of fast memory size and program parameters, the tool can also suggest a loop permutation and numerical tile sizes that minimize I/O. Finally, it is able to use computer algebra to determine tile sizes as a function of the fast memory size S, and to get an upper bound that is directly comparable to the lower bound expression from IOLB. For instance, on square matrix-matrix multiplication, the upper bound from IOUB is $\frac{2N^3}{\sqrt{S+1}-1} + 1$, which matches the lower bound up to lower order terms, and effectively proves that the I/O complexity for this problem is $2\frac{N^3}{\sqrt{S}}(1 + o(1))$. IOUB was evaluated on several tensor contraction and convolution programs, giving optimal bounds in many cases.

The rest of this manuscript is organized as follows. Chapter 2 gives a high-level view of the concepts and methods used to derive lower and upper bounds on I/O complexity, using examples to provide intuition to the reader. Chapter 3 delves into the details of automated I/O lower bound derivation, starting with the formal framework needed to reason about them, then presenting the two main methods (K-partitioning and wavefront), as well as how to combine them. We provide detailed algorithms for the automation of I/O lower bound analysis, and an evaluation of our implementation on a large selection of affine programs. Chapter 4 similarly presents the I/O upper bound analysis method, first describing the model and theoretical framework, then explaining how to build an automated program analysis, and showing its application on tensor contraction and convolution kernels. Finally, conclusions and suggestions for future work are provided in Chapter 5.

Chapter 2

Overview and Background

In this chapter, we give a high-level view of the concept of I/O complexity, and illustrate the techniques for deriving lower and upper bounds on an example. The goal is to provide intuitions on the representations and reasoning methods that will be more thoroughly detailed in Chapters 3 and 4.

2.1 I/O Complexity

Our framework is a simple memory model with two levels: a small and fast memory (of fixed size S) and a slow but infinite memory. Initially, all program data resides in the slow memory. Data can be transferred back and forth between the two memory levels, and a piece of data has to be in the fast memory in order to be used in a computation. The *I/O cost* of a program, for a given schedule of arithmetic and I/O operations, is the number of data transfers from slow to fast memory (or *loads*) [1]. Now, for a given program, we are interested in the lowest I/O cost that can be achieved by reordering operations: this quantity is called the *I/O complexity* of a program.

Computing the I/O complexity exactly is usually intractable, as one would have to simulate every possible valid schedule of operations. Hence the need to find lower and upper bounds on this complexity. To get an precise estimation of the I/O complexity of a program,

1. We choose in this work to count only loads (transfers from slow to fast memory) and not stores (transfers from fast to slow memory). This simplifies the reasoning while still providing meaningful values.

Figure 2.1 – Memory model

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

(a) C code.

(b) Graph representation (CDAG) for $N = 2$.

Figure 2.2 – Matrix-matrix multiplication.

the lower and upper bounds should be as close as possible. In some cases (such as matrix multiplication that will be used in this section), it is possible to find matching lower and upper bounds (up to lower order additive terms), but even for simple programs this requires complex reasoning. It is important to note that the bounds that we derive are valid for a particular implementation of an algorithm, and not for any program computing a given problem.

2.2 Program representation

Since we need to represent whether a given schedule of operations is valid or not, a program is represented by its operations, and data dependencies between them. This is naturally abstracted as a directed graph, called a CDAG (computational directed acyclic graph, see Def. 1). For programs that are composed of nested loops, vertices of this graph can be represented as points in a multi-dimensional space.

Example: Matrix Multiplication Let us take the example of matrix-matrix multiplication. The corresponding C code and the graph representation are shown on Fig. 2.2. On Fig. 2.2b, vertices in green, red and light gray respectively represent input data from arrays A, B and C. Vertices in dark gray are instances of statement S (i.e., each vertex corresponds to a multiplication and an addition). Edges represent data dependencies: all direct predecessors of a vertex must be in fast memory to perform the computation.

The arithmetic complexity of this program (counting the multiplication and accumulation as a single operation) is $\mathcal{C} = N^3$.

Considering the original schedule given by Fig. 2.2a, and an optimal memory management (evicting the piece of data that will be needed latest from the small memory when it is full), and supposing the fast memory size S to be much smaller than N^2, the I/O cost for this schedule is on the order of N^3 — intuitively, the condition $S << N^2$ means that all data (matrices A, B and C) does not fit in the fast memory. Indeed, to compute a single row of C, the totality of matrix B needs to be loaded in memory, so at most $S - 1$ words of matrix B can stay in fast memory between two consecutive iterations of the outer loop on i, and each iteration demands at least $N^2 - S + 1$ load operations. The outer loop iterates N times, so the total

number of operations when $S << N^2$ is indeed of the order of N^3. The question is to know whether this can be improved, and up to which point.

2.3 Lower bound derivation

We employ two distinct approaches to find lower bounds on data movement: one based on the S-Partitioning approach [29], and one based on graph wavefronts [22]. Combining these two approaches is essential for handling of a large class of programs, as they are complementary and work on different data dependence patterns. A lower bound for a program exhibiting a combination of both kinds of patterns can combine results from both approaches. In this introductory section, we only present a high-level overview of the S-Partitioning approach, and apply it to matrix multiplication, to familiarize the reader with the reasoning and terminology used.

The general idea of the S-partitioning approach for proving lower bounds is to reason on the maximal *operational intensity* of a sub-CDAG, i.e. the maximum number of computations that can be done with a fixed number of memory operations (loads). More precisely, for a given parameter T (whose value will be chosen later), we are looking for an upper bound on the size of the largest subgraph that can be executed doing only T loads, and supposing the fast memory is full, i.e. S pieces of data are already loaded. Supposing we know such a bound $U(T)$, this implies that any valid schedule must contain at least $\lfloor |V|/U(T) \rfloor$ sub-sequences with T loads, where V is the set of computational vertices in the CDAG. This leads to a lower bound on the number of loads of

$$Q_{\text{low}} = T \cdot \left\lfloor \frac{|V|}{U(T)} \right\rfloor. \tag{2.1}$$

Let us now explain how the upper bound $U(T)$ can be computed. The idea is to use information on the "boundary" of a set of vertices P that can be executed with T load instructions, to derive a bound on the cardinality of P. Intuitively, the "boundary" corresponds to values that need to be in fast memory to execute vertices in P, so its size is bounded. When the dependence pattern in the CDAG is regular, geometric inequalities that relate the cardinality of a set of points in a multi-dimensional space to cardinalities of lower-dimensional projections of those points can be used to derive a bound on $|P|$.

More formally, the In-set $\text{In}(P)$ of P is the set of all predecessors of the vertices in P that do not belong to P (see Fig. 2.3a). $\text{In}(P)$ represents values that are dependencies of operations in P, but are not computed within P. Since all values in $\text{In}(P)$ must be in fast memory in order to execute the operations in P, they must either already be in fast memory at the beginning of the execution, or be explicitly loaded. At most S values from $\text{In}(P)$ can be initially present in the fast memory, and by definition T values can be loaded to execute P. Thus the size of $\text{In}(P)$ must be less than $(S + T)$.

In many programs, CDAGs have a natural embedding in a multi-dimensional space, where each dimension corresponds to a loop counter, and dependences between statements are regular and orthogonal. When this is the case, the size of the In-set $\text{In}(P)$ can be bounded by the size of its projections on hyperplanes. Fig. 2.3b illustrates this relation in a 2-dimensional setting. This relation is one of the key insights to the automation of lower bound computation.

Let us go back to matrix multiplication. In this example, vertices corresponding to statement S are naturally represented as points in a three-dimensional lattice, each dimension

(a) (b)

Figure 2.3 – In-set and projections. When data dependencies are regular and orthogonal, $\mathrm{In}(P)$ is bounded by the size of projections of P.

Figure 2.4 – Loomis-Whitney inequality in three dimensions

representing a loop index (see Fig. 2.2b). With that representation, we observe that the size of the In-set of a vertex set of this particular graph must be greater than or equal to the cardinality of the orthogonal projections of P onto the three elementary planes:

$$|\mathrm{In}(P)| \geq \left|\mathrm{proj}_x(P)\right| + \left|\mathrm{proj}_y(P)\right| + \left|\mathrm{proj}_z(P)\right|. \tag{2.2}$$

Conveniently, a mathematical result known as the *Loomis-Whitney inequality* [34] relates the size of a vertex set in a three-dimensional space to the sizes of its three 2D projections (see Fig. 2.4):

$$|P| \leq \left(\left|\mathrm{proj}_x(P)\right| \cdot \left|\mathrm{proj}_y(P)\right| \cdot \left|\mathrm{proj}_z(P)\right|\right)^{\frac{3}{2}}. \tag{2.3}$$

This result can be generalized to arbitrary dimensions and any set of (not necessarily orthogonal) projections, and is called the Brascamp-Lieb inequality.

Optimizing the conjunction of inequalities (2.2) and (2.3) leads to the following bound on $|P|$: (mathematical details can be found in Sec. 3.3.1)

$$U(T) = \left(\frac{(S+T)}{3}\right)^{\frac{3}{2}}. \tag{2.4}$$

Finally, setting $T = 2S$ and plugging the expression into equation (2.1) gives the optimal bound

$$IO_{\mathrm{mm}} \geq Q_{\mathrm{low}} = 2S \cdot \left\lfloor \frac{N^3}{S^{\frac{3}{2}}} \right\rfloor \approx \frac{2N^3}{\sqrt{S}}. \tag{2.5}$$

A key contribution of this work is the automation of this process.

2.4 Upper bound derivation

Once we found a lower bound, the question is whether it can be attained. An upper bound on I/O complexity is simply the cost of a valid program schedule.

Thus, for a program with fixed parameter values, it is possible to generate a full execution trace, and count the number of data transfers, using an optimal cache policy (always evicting the piece of data that will be used the latest). However, this is costly and only valid for fixed parameter values, while we would like to derive parametric expressions. Moreover, the goal is to find an upper bound as low as possible, so we need to use a "good" schedule.

The main rescheduling transformation to optimize data transfers is *tiling*. It consists in partitioning the complete iteration domain of a loop nest into smaller parts, usually (hyper-)rectangles, called *tiles*, and executing each tile atomically. Fig. 2.5 shows examples of tiled codes for matrix multiplication.

Our approach to finding upper bounds on I/O complexity can be summarized as follows. Beforehand, the program is analyzed to find the part on which the tiling transformation can be applied. A first step selects a subset of tiling loop permutations, using analytical reasoning to reduce the search space. For each permutation, this provides a parametrically tiled program. Then, an algorithm based on polyhedral calculus computes a symbolic expression of the I/O cost for each tiling permutation, as well as constraints on tile sizes. Finally, these expressions can be used to provide actual tile sizes for numerical parameter values, or to provide a symbolic bound depending only on S but not on tile sizes, that can be directly compared with the lower bound.

```
for(i1 = 0; i1 < N; i1 += Ti)
  for(j1 = 0; j1 < N; j1 += Tj)
    for(k1 = 0; k1 < N; k += Tk)
      for(i = i1; i < i1 + Ti; i++)
        for(j = 0; j < j1 + Tj; j++)
          for(k = 0; k < k1 + Tk; k++)
            C[i][j] += A[i][k] * B[k][j];
```

(a)

```
for(j1 = 0; j1 < N; j1 += Tj)
  for(i1 = 0; i1 < N; i1 += Ti)
    for(k1 = 0; k1 < N; k += Tk)
      for(k = 0; k < k1 + Ti; k++)
        for(i = i1; i < i1 + Tj; i++)
          for(j = 0; j < j1 + Tk; j++)
            C[i][j] += A[i][k] * B[k][j];
```

(b)

Figure 2.5 – Two valid tiled schedules for Matrix-matrix multiplication.

```
for(i1 = 0; i1 < N; i1 += Ti)
  for(j1 = 0; j1 < N; j1 += Tj)
    for(k1 = 0; k1 < N; k += Tk)
      for(i = i1; i < i1 + Ti; i++)
        for(j = 0; j < j1 + Tj; j++)
          for(k = 0; k < k1 + Tk; k++)
            C[i][j] += A[i][k] * B[k][j];
```

A	B	C
N^2	N^2	N^2
$Ti \cdot N$	N^2	$Ti \cdot N$
$Ti \cdot N$	$Tj \cdot N$	$Ti \cdot Tj$
$Ti \cdot Tk$	$Tj \cdot Tk$	$Ti \cdot Tj$
Tk	$Tk \cdot Tj$	Tj
Tk	Tk	1
1	1	1

Figure 2.6 – Array footprints

Let us illustrate it on the matrix multiplication example. A tiled schedule for matrix multiplication consists of six loops, and the permutation of the three outer loops and the three inner loops is free. Figure 2.5 shows two valid tiled schedules (supposing for clarity that `Ti`, `Tj` and `Tk` are divisors of `N`). We use a heuristic to select a subset of relevant permutations, which is detailed in Sec. 4.3. For now, let us focus on code (a) in Fig. 2.5.

Figure 2.6 shows the symbolic expression of the memory footprint of each array for each subloopnest. For each array, the last level for which the footprint is supposed to fit in the fast memory is highlighted. This means that, for some fixed values of `i1` and `j1`, at the beginning of the third loop iterating of `k1`, all $Ti \cdot Tj$ values `C[i][j]` for `i = i1...i1 + Ti`, `j = j1...j1 + Tj` are loaded and kept in fast memory for the whole iteration. The same goes at the beginning of the fourth loop (iterating on `i`), for values `A[i][k]` and `B[k][j]`, `i = i1...i1 + Ti`, `j = j1...j1 + Tj`, `k = k1...k1 + Tk`.

This implies that all these pieces of data fit in the fast memory, i.e.

$$Ti \cdot Tk + Tj \cdot Tk + Ti \cdot Tj \leq S \tag{2.6}$$

The outer loops iterating on `i1`, `j1` and `k1` iterate respectively `N/Ti`, `N/Tj` and `N/Tk` times. In

```
for(i1 = 0; i1 < N; i1 += T)
  for(j1 = 0; j1 < N; j1 += T)
    for(k = 0; k < N; k++)
      for(i = i1; i < i1 + T; i++)
        for(j = 0; j < j1 + T; j++)
          C[i][j] += A[i][k] * B[k][j];
```

Figure 2.7 – Best tiled schedule for matrix multiplication.

total, $(\mathtt{N}/\mathtt{Ti})\cdot(\mathtt{N}/\mathtt{Tj})\cdot(\mathtt{N}/\mathtt{Tk})\cdot(\mathtt{Ti}\cdot\mathtt{Tk}) = \mathtt{N}^3/\mathtt{Tj}$ values of A, $(\mathtt{N}/\mathtt{Ti})\cdot(\mathtt{N}/\mathtt{Tj})\cdot(\mathtt{N}/\mathtt{Tk})\cdot(\mathtt{Tj}\cdot\mathtt{Tk}) = \mathtt{N}^3/\mathtt{Ti}$ values of B, $(\mathtt{N}/\mathtt{Ti})\cdot(\mathtt{N}/\mathtt{Tj})\cdot(\mathtt{Ti}\cdot\mathtt{Tj}) = \mathtt{N}^2$ values of C are loaded. The total I/O of this program under this schedule is thus

$$IO = \frac{\mathtt{N}^3}{\mathtt{Ti}} + \frac{\mathtt{N}^3}{\mathtt{Tj}} + \mathtt{N}^2 \tag{2.7}$$

Finally, to be able to compare this expression with the lower bound, Ti and Tj need to be expressed as functions of S and N. As the goal is to minimize I/O, this amounts to finding values of Ti, Tj and Tk that minimize expression (2.7) under condition (2.6). This can be solved manually or using computer algebra, and the optimal value is $\mathtt{Ti} = \mathtt{Tj} = \sqrt{S+1}-1$, and $\mathtt{Tk} = 1$, leading to

$$IO = \frac{2N^3}{\sqrt{S+1}-1} + N^2. \tag{2.8}$$

This expression matches the lower bound up to lower order terms.

Since the optimal value of T_k is 1, one loop of the tiled code can be removed, and the final code attaining this bound is shown on Fig. 2.7.

Our tool, IOOpt, is also able to find a numerical solution for fixed values of N and S. For instance, for $N = 1500$ and $S = 1024$, IO is minimized for $\mathtt{Ti} = \mathtt{Tj} = 31$, and gives a total number of loads $IO = 65516129$.

On an actual machine, several levels of memory are present, so multiple levels of tiling are needed to produce efficient code. IOOpt is able to model such multi-level memory hierarchies and optimize the I/O for all levels at once. The result can then be used in conjunction with additional optimization techniques (vectorization, efficient register use...) to produce highly efficient code for modern CPUs.

2.5 Affine programs and automation

This chapter gave a high-level view of the models and methods to produce I/O bounds, but did not detail so far the automation process, which is a key part of this thesis.

To do so, we have to choose a class of programs to work with: the class of *affine* programs stands in a nice spot combining expressiveness and applicability of our techniques, and comes with a convenient analysis tool ecosystem. Affine programs cover a wide set of key algorithms, as exemplified with the 30 algorithms in PolyBench/C [39] that span popular dense linear algebra, stencils/convolutions, and dynamic programming techniques.

An affine program is composed of (potentially nested) for loops and statements, operating on variables and multi-dimensional arrays. Loop bounds are affine expressions of surrounding loop indices and program parameters (a program parameter is a symbolic constant during

the compilation, like array dimensions). Similarly, access functions made by statements to arrays are affine expressions of surrounding loop indices and program parameters. As such, the control flow of such programs is statically analyzable.

This class corresponds to the class of programs that can be analyzed by polyhedral compilation techniques [25]. Powerful analysis tools such as ISL [53] and barvinok [57] can represent these programs abstractly as points in a multidimensional space, and do some operations on these points such as analyzing data flow dependences, counting the number of points satisfying some property as a symbolic expression... This is particularly interesting for our I/O analysis, as it allows us for instance to uncover regular data dependencies that correspond to projections in our lower bound derivation.

For example, in the matrix multiplication code of figure 2.2a, all iterations of statement S are represented as follows in ISL notation:

$$[N] \rightarrow \{S[i,j,k] : \ 0 \leq i < N \ \wedge \ 0 \leq j < N \ \wedge \ 0 \leq k < N\} \tag{2.9}$$

Here the matrix dimension N is a program parameter (a symbolic constant), and every instance of statement S is represented as an integer point in a 3-dimensional space.

In the same way, data dependencies can be represented parametrically. The following ISL formula represents all dependencies from data in array A:

$$[N] \rightarrow \{A[i,k] \rightarrow S[i,j,k] : \ 0 \leq i < N \ \wedge \ 0 \leq j < N \ \wedge \ 0 \leq k < N\} \tag{2.10}$$

The corresponding vertices and edges are highlighted on the CDAG from Fig. 2.2b on Fig. 2.8.

Figure 2.8 – Ilustration of equations (2.9) and (2.10) on the CDAG for $N = 2$.

Chapter 3

Lower Bounds

In this chapter, we present our results and tools for the derivation of I/O lower bounds. Formal definitions and theoretical background needed for the main analysis are introduced in Section 3.1. Then, Section 3.2 provides insights on how complex programs can be decomposed to derive tighter bounds. Two proof techniques, namely the K-partition and the wavefront based proofs are respectively described in Section 3.3 and Section 3.4. An overview of the complete framework is provided in Sec. 3.5. We demonstrate the power of our approach by running it on a full benchmark suite of affine programs: Sec. 3.6 reports the data movement complexities for the 30 algorithms benchmarked in PolyBench, as well as tensor contraction and convolution programs. Finally, related work is discussed in Sec. 3.7.

3.1 Foundations

Before delving into the main matter, we present some background and discuss prior results needed for the developments in this chapter.

3.1.1 CDAG

The formalism and methodology we use is strongly inspired by the foundational work of Hong & Kung [29]. It was introduced in the context of data movement lower bounds and models the execution of an algorithm on a processor with a two-level memory hierarchy.

In this formalism, an algorithm is abstracted by a graph — called a CDAG —, where vertices model execution instances of arithmetic operations and edges model data dependencies among the operations. The data movement (or *I/O*) complexity of a CDAG is formalized via the red-white pebble game (a variation of Hong & Kung's red-blue pebble game). In this game, a vertex of a CDAG can hold red and white pebbles. Red pebbles represent values in the fast memory (typically a cache or scratchpad), and their total number is limited. White pebbles represent computed values, that can be loaded into the fast memory. A value can be computed only when all its operands reside in the fast memory: a red pebble can be placed on a vertex in the CDAG if all its predecessors hold a red pebble, a white pebble is placed alongside the red one. Values that have been computed can be loaded in and discarded from the fast memory at any time: a red pebble can be placed or removed from a vertex holding a white pebble. However a value can only be computed once: once a vertex holds a white

```
Parameters: N, M;
Input: A[N], C[M]; Output: A[N];
for(t=0;t<M;t++)
  for (i=0; i<N; i++)
    A[i] = A[i] * C[t];
```

(a)

```
Parameters: N, M;
Input: A[N], C[M]; Output: S_{M-1}[N];
for (0 ≤ t < M and 0 ≤ i < N)
  if (t==0): S_{0,i} = A[i] * C[0];
  else: S_{t,i} = S_{t-1,i} * C[t];
```

(b)

(c)

Figure 3.1 – Example 1. **(a)** C-like code. **(b)** Corresponding single assignment form. **(c)** Corresponding CDAG. Input nodes `A[N]` (resp. `C[N]`) are in grey (resp. white), compute nodes are in black.

pebble, it cannot be removed. The *I/O* cost of an execution of the game is the number of loads into the fast memory: the number of times a red pebble is placed alongside a white one.

Contrary to Hong & Kung's original model, this formalism *does not allow recomputation* of the value at a vertex. This follows many previous efforts [4, 3, 10, 11, 19, 22, 21, 30, 46]. This assumption is necessary to be able to derive lower bounds for complex CDAGs by decomposing them into subregions. Another slight difference with prior work is that it only models *loads and not stores* — this means the generated bounds are clearly also valid lower bounds for a model that counts both loads and stores. Since the number of loads dominates stores for most computations, the tightness of the lower bounds is not significantly affected.

Let us define formally the notion of CDAG, and our pebbling game.

Definition 1 (Computational Directed Acyclic Graph (CDAG))**.** *A* Computational Directed Acyclic Graph (CDAG) *is a tuple $G = (V, E, I)$ of finite sets such that (V, E) is a directed acyclic graph, $I \subseteq V$ is called the* input set *and every $v \in I$ has no incoming edges.*

Definition 2 (Red-White Pebble Game)**.** *Given a CDAG $G = (V, E, I)$, we define a complete S-red-white pebble game (S-RW game for short) as follows: In the initial state, there is a white pebble on every input vertex $v \in I$, S red pebbles and an unlimited number of white pebbles. Starting from this state, a complete game is a sequence of steps using the following rules, resulting in a final state with white pebbles on every vertex.*

- **(R1)** *A red pebble may be placed on any vertex that has a white pebble.*
- **(R2)** *If a vertex v does not have a white pebble and all its immediate predecessors have red pebbles on them, a red pebble may be placed on v. A white pebble is placed alongside the red pebble.*
- **(R3)** *A red pebble may be removed from any vertex.*

The cost *of a S-RW game is the number of applications of rule (R1), corresponding to the number of transfers from slow to fast memory.*

Definition 3 (*I/O* complexity)**.** *The I/O (or* data movement*)* complexity *of a CDAG G for a fast memory capacity S, denoted $Q(G)$, is the minimum cost of a complete S-RW game on G.*

3.1.2 A compact representation of the CDAG: the data-flow graph

```
for (0 ≤ t < M and 0 ≤ i < N)
  if (t==0): S[0,i]=A[i]*C[0];
  else: S[t,i]=S[t-1,i]*C[t];
```

(a) Single assignment form

(b) DFG

$$D_A = [N] \to \{A[i] : \ 0 \le i < N\}$$
$$D_C = [N] \to \{C[t] : \ 0 \le t < M\}$$
$$D_S = [M,N] \to \{S[t,i] : \ 0 \le t < M \ \wedge \ 0 \le i < N\}$$
$$|D_S| = MN$$

(c) Node domains

$$R_{e_1} = [N] \to \{A[i] \to S[0,i] : \ 1 \le i < N\}$$
$$R_{e_2} = [M,N] \to \{C[t] \to S[t,i] : \ 0 \le t < M \ \wedge \ 0 \le i < N\}$$
$$R_{e_3} = [M,N] \to \{S[t,i] \to S[t+1,i] : \ 0 \le t < M-1 \ \wedge \ 0 \le i < N\}$$

(d) Edge relations

Figure 3.2 – DFG for Example 1

A CDAG (see Fig. 3.1c) represents a single dynamic execution of a program, and can be very large. To be able to analyze programs of realistic size with reasonable resources, we use a compressed representation called Data-flow graph (DFG). Another advantage of such a representation is that it is *parametric*, i.e. a single DFG can represent CDAGs of different sizes, depending on program parameters. A DFG represents an *affine* computation, which is the class of programs that can be handled by the *polyhedral model* [25]. We use terminology and syntax inspired on the ISL library [53], and illustrate them with the example of Fig. 3.1. Formal definitions can be found in the tutorial [54].

Vertex domains As one can see on Fig. 3.1c, to each loop is associated a "geometric" space dimension (t and i here) so that each *vertex of the CDAG* lives in a multidimensional iteration space, its *domain*, that can be algebraically represented as a union of parametric $\mathbb{Z}$-polyhedra bounded by affine inequalities.

Definition 4. *A* domain *is a $\mathbb{Z}$-polyhedron, i.e. a set of points in the lattice $\mathbb{Z}^d$ for some integer d, bounded by affine inequalities. In ISL notation, it is called a* set *and is written:*

$$S = [p_1, \ldots, p_m] \to \{X[i_1, \ldots, i_d] : \mathcal{P}\}$$

where:

- *$p_1, \ldots, p_m$ are (integer) parameter variables,*
- *X is an identifier for the* space *to which points in S belong (two points with the same coordinates but different identifiers are treated as distinct points),*
- *$i_1, \ldots, i_d$ are coordinate variables,*
- *$\mathcal{P}$ is a Presburger formula on parameter and coordinate variables (in most cases a conjunction of affine inequalities $\mathcal{I}_1 \wedge \ldots \wedge \mathcal{I}_t$).*

It is also often useful to manipulate unions of such domains, and ISL provides appropriate objects for this. Standard operations (union, intersection, difference,...) are available, as well

as a *cardinality* operation (denoted $|D|$), which computes the cardinality of a given domain as a (quasi-)polynomial function of the parameter variables (this is provided by the barvinok library[7, 57]). As an example (see Fig. 3.2c), the *domain* D_S of statement S is a $\mathbb{Z}$-polyhedron with parameters M and N made up of all integer points (t,i) such that $0 \leq t < M$ and $0 \leq i < N$. The number of points in this set (cardinality) is $|D_S| = MN$. Note that the space within which all the points of a statement (S here) live is identified with the name of the statement, using the notation $S[t,i]$.

Edge relations A set of *edges* of the CDAG is represented using a *relation* (ISL map), which is a set of pairs between two spaces, from the *domain* space to the *image* space.

Definition 5. *A* relation *is a binary relation between points in two domains, subject to affine constraints. It is called a map in ISL, and written as follows:*

$$R = [p_1,\ldots,p_m] \rightarrow \{X[i_1,\ldots,i_d] \rightarrow Y[j_1,\ldots,j'_d] : \mathcal{I}_1 \ \wedge \cdots \wedge \mathcal{I}_t\}$$

The domain $\mathrm{Dom}(R)$ *of R is the set of points $X[i_1,\ldots,i_d]$ having an image through R. The* image $\mathrm{Im}(R)$ *of R is the set of points $Y[j_1,\ldots,j'_d]$ that have a predecessor through R.*

As an example (see Fig. 3.2d), the data flow from statement $S[t,i]$ (definition of $A[i]$ in S) to statement $S[t+1,i]$ (use of $A[i]$ in S) is represented using the relation R_{e_3}. In addition to standard set operations, ISL can compute (an over-approximation of) the transitive closure of a relation, denoted R^*. The following operations are also supported: image of a domain D through a relation R (denoted $R(D)$), and composition of two relations R_1 and R_2, denoted $R_1 \circ R_2$ (this is left composition, going the opposite way from usual functional notation). Composition restricts the image space of the resulting relation to points where the composition relation makes sense: $\mathrm{Dom}(R_1 \circ R_2) = R_1^{-1}(\mathrm{Im}(R_1) \cap \mathrm{Dom}(R_2))$, $\mathrm{Im}(R_1 \circ R_2) = R_2(\mathrm{Im}(R_1) \cap \mathrm{Dom}(R_2))$. As with domains, we will sometimes manipulate unions of such relations.

A Data-flow graph (DFG) A DFG is a graph $G = (\mathcal{S}, \mathcal{D})$. Each vertex $S \in \mathcal{S}$ of the graph represents a (static) statement or an input array of the program. Each vertex S is associated with a parametric iteration domain D_S and a list of enclosing loops (empty for input arrays). Each edge $d = (S_a, S_b) \in \mathcal{D}$ represents a flow dependency between statements or input arrays. Each edge is associated with an affine relation R_d between the coordinates of the source and sink vertices. The DFG is a compact (exact) representation of the dynamic CDAG where a single vertex/edge of the DFG represents several vertices/edges of the dynamic CDAG. While all the reasoning and proofs can be done by visualizing a CDAG, the actual heuristic described in this chapter manipulates its compact representation, allowing us to translate graph methods [21] into geometric reasoning. Fig. 3.2b shows the DFG for our simple stencil code.

3.1.3 Partitioning

One key idea from Hong & Kung was the design of a mapping between any valid sequence of moves in the red-blue pebble game and a convex partitioning of the vertices of a CDAG and thereby the assertion of an *I/O* lower bound for any valid schedule in terms of the minimum possible count of the disjoint vertex-sets in any valid $2S$-partition (see below) of the CDAG.

The argument is the following: any execution can be decomposed into consecutive segments doing exactly (but for the last one) S loads. There are at most S vertices in fast memory

before the start of each segment. Considering the set of computed vertices in one of these segments, we can bound the size of its "frontier" by $2S$: there can be at most S vertices in fast memory before the execution of the segment, and by construction there are exactly S loads.

Smith et al. [48] introduced a generalization of this argument, leading to tighter bounds in many cases. The idea is to decompose the execution into segments with T loads. This leads to a $(S+T)$-partitioning lemma instead of the original $2S$. We provide formal definitions below.

Definition 6 (In-set)**.** *Let $G = (V, E)$ be a DAG, $P \subseteq V$ be a vertex set in G. The In-set of P is the set of vertices outside P with a successor inside P. Formally,*

$$In(P) = \{v \in V \setminus P,\ \exists (v, w) \in E \wedge w \in P\}$$

Definition 7 (K-bounded set)**.** *Let $G = (V, E)$ be a DAG. A vertex set $P \subseteq V$ is called* K-bounded *if $In(P) \leq K$.*

Definition 8 (K-partition of a CDAG)**.** *Let $G = (V, E, I)$ be a CDAG. A K-partition of G is a collection of subsets $V_1, V_2, \ldots, V_m$ of $V \setminus I$ such that:*

(1) *$\{V_1, V_2, \ldots, V_m\}$ is a partition of $V \setminus I$, i.e. $\forall i \neq j\ V_i \cap V_j = \emptyset$ and $\bigcup_{i=1}^{m} V_i = V \setminus I$.*

(2) *There is no cyclic dependence between V_i's.*

(3) *Every V_i is K-bounded, i.e. $\forall i\ In(V_i) \leq K$.*

Lemma 1 ($(S+T)$-Partitioning [29])**.** *Let $T > 0$. Any complete calculation $\mathcal{R}$ of the red-white pebble game on a CDAG G using at most S red pebbles is associated with a $(S+T)$-partition of the CDAG such that*

$$Q^{\mathcal{R}} \geq T \cdot (h - 1),$$

where $Q^{\mathcal{R}}$ is the number of applications of rule (R1) in the game and h is the number of subsets in the partition.

In particular, an upper bound U on the size of a $(S+T)$-bounded set directly translates into a data movement lower bound: $Q(G) \geq T \cdot \left(\left\lceil \frac{|V \setminus I|}{U} \right\rceil - 1\right)$. This bound can actually be improved to:

$$Q(G) \geq T \cdot \left\lfloor \frac{|V \setminus I|}{U} \right\rfloor. \tag{3.1}$$

Indeed, the proof of Lemma 1 establishes a correspondence between the number of sets in a $(S+T)$-partition of G and a partition of an execution of the game in segments containing exactly S loads, except maybe the last one. The subtraction by one is due to this last segment. When $|V \setminus I|/U$ is an integer, the segment contains exactly S loads and thus the subtraction is not necessary.

We actually want to be able to compute lower bounds for CDAGs in which no vertices are tagged as input (this is particularly useful when doing decomposition, see Sec. 3.2) The following lemma (Lemma 2) establishes such a result. The main idea is as follows: we tag some vertices as input, getting a new CDAG on which Lemma 1 applies and gives some bound. The additional *I/O* cost is at most the number of input vertices that were added, so we get a lower bound for the original CDAG by subtracting this number from the bound. See [22] for a complete proof.

Definition 9 (Sources)**.** *Let $G = (V, E)$ be a DAG, $P \subseteq V$ be a vertex set in G. The sources of P are the vertices of P with no predecessors in P. Formally,*

$$\text{Sources}(P) = \{v \in P,\ \nexists u \in P, (u, v) \in E\}$$

Lemma 2 ($(S+T)$-Partitioning *I/O* lower bound, no input case [22])**.** *Let S be the capacity of the fast memory, let $G = (V, E, \emptyset)$ be a CDAG, and let h be the minimum number of subsets in a $(S+T)$-partition of $G_I = (V, E, I = \text{Sources}(V))$ for some $T > 0$. Then, the minimum I/O for G satisfies:*

$$Q(G) \geq T \cdot (h-1) - |\text{Sources}(V)|.$$

3.1.4 Using projection to bound the cardinality of K-bounded sets

The key idea behind the automation of data movement lower bound computation is the use of geometric inequalities through an appropriate program representation. Vertices of a CDAG are mapped to points in a multidimensional geometric space $\mathcal{E} \simeq \mathbb{Z}^d$ through some mapping ρ (where dimensions are typically loop indices), and regular data dependencies in the CDAG are represented as projections on a lower-dimensional space.

The condition "set of vertices $P \subset V$ is K-bounded" in the CDAG corresponds to a condition of the form "the size of the projections of $\rho(P)$ in $\mathcal{E}$ is bounded by K". Finding a bound on the size of a K-bounded set in a CDAG can thus be reduced to: finding a bound on the size of a set E in a geometric space, given cardinality bounds on some of its projections. This correspondence is developed in Sec. 3.3. In this section, we take it for granted and only introduce the mathematical notations and results.

There exist inequalities for doing exactly what we need, namely the discrete Brascamp-Lieb inequality, introduced by Christ et al. [16] as a discrete analogue to the one established by Brascamp and Lieb for metric spaces [13].

Theorem 1 (Brascamp-Lieb inequality, discrete case [16])**.** *Let d and d_j be nonnegative integers and $\phi_j : \mathbb{Z}^d \mapsto \mathbb{Z}^{d_j}$ be group homomorphisms for $1 \leq j \leq m$. Let $0 \leq s_1, s_2, \ldots, s_m \leq 1$. Suppose that:*

$$\text{rank}(H) \leq \sum_{j=1}^{m} s_j \cdot \text{rank}\big(\phi_j(H)\big) \text{ for all subgroups } H \text{ of } \mathbb{Z}^d \tag{3.2}$$

Then:

$$|E| \leq \prod_{j=1}^{m} \big|\phi_j(E)\big|^{s_j} \text{ for all nonempty finite sets } E \subseteq \mathbb{Z}^d. \tag{3.3}$$

A special case is when the ϕ_j are the canonical projections on $\mathbb{Z}^{d-1}$ along basis vectors, and $s_1 = \cdots = s_m = \frac{1}{d-1}$, giving a bound of the form $|E| \leq \prod_{i=1}^{d} \big|\phi_j(E)\big|^{1/(d-1)}$. Fig. 2.4 illustrates this special case in three dimensions.

The issue, when trying to apply Theorem 1, is that (3.2) has to be true for *all* subgroups, which can obviously be quite difficult to establish. However, as all the coefficients are integers and bounded by d, the number of distinct inequalities in (3.2) is bounded. The set of admissible s_j is thus a (convex) polyhedron and it has been shown [17] that the polyhedron defined by these inequalities is computable. The algorithm is actually combinatorial and quite complex, so we do not use it in our present work. Instead, we use the following result, which restricts the set of subgroup for which (3.2) has to be verified.

Definition 10 (Lattice of subgroups)**.** *The* lattice of subgroups *generated by subgroups $H_1, H_2, \ldots, H_m$ of a group G is the closure of $\{H_1, H_2, \ldots, H_m\}$ under group sum and intersection.*

Lemma 3 (Lattice of subgroups in Brascamp-Lieb[52])**.** *Theorem 1 holds with the weaker condition:*

$$\operatorname{rank}(H) \leq \sum_{j=1}^{m} s_j \cdot \operatorname{rank}\big(\phi_j(H)\big) \text{ for all subgroups } H \in \mathcal{L}_{\phi_1,\phi_2,\ldots,\phi_m} \tag{3.2b}$$

where $\mathcal{L}_{\phi_1,\phi_2,\ldots,\phi_m}$ is the lattice of subgroups generated by $Ker(\phi_1), Ker(\phi_2), \ldots, Ker(\phi_m)$.

The subgroup lattice is not necessarily finite, so this does not give a tractable algorithm, but this will be sufficient in most cases, as loop nests are usually of quite small dimensions, and data dependencies are not too complex. In our practical implementation, we use a time-out. We add projections: the more projections the tighter the bound. Each time we add a projection we update the lattice of subgroups and check for Conditions 3.2b to be satisfied. The process (adding projections) stops if we reach the time-out. This does not mean that the algorithm fails, rather that the bound will potentially be less tight (cf. Sec. 3.5).

A situation that often occurs is when the subgroups $\operatorname{Ker}(\phi_j)$ are linearly independent: in this case there is no need to compute the generated lattice of subgroups, simply testing on each kernel individually for Conditions 3.2 is sufficient (see proof in [16], Sec. 6.3).

Choosing s_j's The goal is to solve the following problem: *"Given a set of projections (group homomorphisms in $\mathbb{Z}^d$) $\phi_1, \ldots, \phi_m$ and a constant K, find an upper bound (as tight as possible) on the cardinality of a set $E \subset \mathbb{Z}^d$ satisfying $\left|\phi_j(E)\right| \leq K$."*

For any coefficients $s_1, \ldots, s_j$ satisfying (3.2), Theorem 1 gives the following bound on $|E|$:

$$|E| \leq \prod_{j=1}^{m} \left|\phi_j(E)\right|^{s_j} \leq \prod_{j=1}^{m} K^{s_j} = K^{\sum_j s_j}.$$

To get a bound as tight as possible on $|E|$, we want to minimize the right-hand side of this inequality. This amounts to minimizing $\sum_j s_j$ while satisfying the constraints in (3.2). Since these constraints are linear inequalities, the optimal choice for s_j's can be obtained by a linear solver.

In the special case where the ϕ_j's are orthogonal projections along basis vectors (and $m = d$), kernels are linearly independent and the linear program is:

$$\text{Minimize} \sum_j s_j \qquad \text{s.t. } \forall 1 \leq i \leq d,\ 1 \leq \sum_{j \neq i} s_j$$

and its solution is, as expected, $s_1 = \cdots = s_d = \frac{1}{d-1}$.

3.1.5 DFG Paths

A fundamental object in our lower bound analysis is a DFG-path, which is simply a directed path in a DFG. The relation R_p of a DFG-path $p = (e_1, \ldots, e_k)$ is the composition of the relations of its edges: $R_p = R_{e_1} \circ \cdots \circ R_{e_k}$. We are only interested in two specific types of DFG-paths, depending on their relation:

— *chain circuits*, which are cycles from one DFG-vertex S to itself, such that the path relation R_p is a translation $S[\vec{x}] \to S[\vec{x}+\vec{b}]$.
— *broadcast S_a, S_b-paths*, which are elementary paths (from a S_a to S_b – S_b possibly equal to S_a) in which all DFG-edges but the first one are injective edges, such that the inverse of the corresponding relation R_p is an affine function $S_b[\vec{x}] \to S_a[A \cdot \vec{x}+\vec{b}]$, where A is not full-rank.

Intuitively, a chain circuit corresponds in the CDAG to "iterative" dependencies, for instance every statement $S_{i,j}$ in a 2-dimensional loop depending on the result of statement $S_{i-1,j}$. Broadcast paths correspond to some data being reused multiple times, for instance a variable x being used by every statement S_i in a one-dimensional loop. The dimension of the kernel of A in the definition above corresponds to the dimension of the set of statements that use a single piece of data: it is of dimension d if it is used in every iteration of a d-dimensional loop. In both cases, these are regular data reuse patterns that can be exploited by our geometric approach.

In Fig. 3.2, path $p = (e_3)$ is a chain circuit, going from S to itself with translation vector $\vec{b} = (1,0)$. Path $p' = (e_2)$ is a broadcast path, with relation $R_{p'} = R_{e_2} = \{C[t] \to S[t,i] : \ 0 \le t < M \ \wedge \ 0 \le i < N\}$. The inverse relation is the linear function $\vec{I} \mapsto A \cdot \vec{I} + \vec{b}$, with $A = (1 \ \ 0), \vec{b} = (0)$. The kernel of A is $\{(0,i), i \in \mathbb{R}\}$.

3.2 CDAG decomposition

To derive data movement lower bounds for a complex program, it is essential to be able to decompose it into subregions for which we can compute lower bounds, and then sum the complexity for each subregion. The *no recomputation* condition is necessary for such a decomposition. Under this hypothesis, it is quite straightforward to see that a decomposition into disjoint subregions is sufficient. In this section, we provide a more general decomposition lemma, using the fact that vertices of a subregion that will not be counted as loads can also be part of another subregion. We then explain how it is applied on the DFG representation, distinguishing two cases: combining a fixed number of program regions (see example in Fig. 3.4); and summing over all iterations of a loop (see example in Fig. 3.3), which amounts to combining an unbounded (parametric) number of program regions. We stress that the CDAG partitioning method (Sections 3.1.3 and 3.1.4) and the CDAG decomposition method (this section) are two distinct things, used at different stages in the global algorithm.

3.2.1 Non-disjoint Decomposition Lemma

Definition 11 (sub-CDAG, no-spill set)**.** *Let $G = (V,E,I)$ be a CDAG, and $V_i \subset V$. The sub-CDAG $G_{|V_i}$ of G is the CDAG with vertices V_i, edges $E_i = E \cap (V_i \times V_i)$ and input vertices $I_i = I \cap V_i$.*

The no-spill set *of $G_{|V_i}$ is the subset of vertices of $V_i \setminus I_i$ with either:*

1. *no outgoing edges in E_i, or*
2. *no incoming edges in E_i and at most one outgoing edge in E_i*

The may-spill set *of $G_{|V_i}$ is the complement of its no-spill set in V_i.*

Lemma 4 (CDAG decomposition)**.** *Let $G = (V,E,I)$ be a CDAG. Let $V_1, V_2, \ldots, V_k$ be subsets of V such that for any $i \neq j$, the may-spill sets of $G_{|V_i}$ and $G_{|V_j}$ are disjoint. Then, the I/O complexity of*

G is bounded by the I/O complexities of the sub-CDAGs $G_{|V_i}$:

$$Q(G) \geq \sum_{i=1}^{k} Q(G_{|V_i}).$$

Proof. Let $\mathcal{R}$ be an optimal S-RW-game on G, with cost $Q = Q(G)$. For all i, we denote Q_i the cost of $\mathcal{R}$ restricted to V_i, that is the number of applications of rule (R1) on vertices in *the may-spill set of* V_i. Since the may-spill sets are pairwise disjoint, clearly $Q \geq \sum_{i=1}^{k} Q_i$. For all i, we will build from $\mathcal{R}$ a valid game $\mathcal{R}_i$ for $G_{|V_i}$ with cost Q_i. This will show that $Q(G_{|V_i}) \leq Q_i$, from which follows $\sum_{i=1}^{k} Q(G_{|V_i}) \leq \sum_{i=1}^{k} Q_i \leq Q = Q(G)$, establishing the result.

To build the game $\mathcal{R}_i$ from $\mathcal{R}$, we proceed as follows:

1. Remove every move involving vertices outside V_i.
2. For no-spill vertices in V_i without successors, remove every application of rule (R1) and (R3).
3. For no-spill vertices v in V_i with no predecessors and one successor w, move the single application of rule (R2) on v just before the application of (R2) on w, add a (R3) move for v just after this point and remove all subsequent (R1) and (R3) moves on v.

It is clear that after step 1, we have a valid game for $G_{|V_i}$. Indeed conditions for (R1) and (R3) are trivially preserved, and since we kept all (R2) moves on vertices in V_i, there will always be the necessary red pebbles to apply (R2). Step 2 also gives a valid game, since no (R2) moves can depend on such a vertex having a red pebble. Step 3 is also a valid transformation because since v does not have any predecessor in V_i (and is not an input vertex), it can be activated via (R2) and any given point. Therefore activating it just when it is needed, and applying (R3) just after is valid.

This preserves all (R1) moves on the may-spill set of $G_{|V_i}$ and removes all (R1) moves on its no-spill set, thus the cost of $\mathcal{R}_i$ is indeed Q_i. □

IOLB implements two different mechanisms that make use of the non-disjoint decomposition lemma. The basic one (bounded combination – Sec. 3.2.2) simply decomposes the CDAG into a bounded number of sub-CDAGs (e.g.,corresponding to different sub-regions of the code), computes the corresponding *I/O* complexities, and combines them. The more complex one (loop parametrization – Sec. 3.2.3), decomposes the CDAG into an unbounded number of sub-CDAGs by "slicing" the iteration space of a loop nest. IOLB combines the two mechanisms. The following example illustrates the decomposition lemma for loop parametrization.

Illustrating example Consider Example 2 on Fig. 3.3. The CDAG can be decomposed into $M-1$ identical subgraphs, as shown on Fig. 3.3c (each subgraph $G_{|V_t}, t = 1,\ldots,M-1$ corresponds to iteration t of the loop enclosing S_1, and iterations $t-1$ and t of the loop enclosing S_2). On each of these subgraphs, the may-spill set contains the two "bottom" rows (because vertices in the "top" row have no successor in the sub-CDAG). Thus the may-spill sets of these subgraphs are pairwise disjoint and the *I/O* for the whole CDAG is greater than the sum of the individual *I/O* for each subgraph by Lemma 4.

On each subgraph $G_{|V_t}$, the wavefront method (Sec. 3.4) can be applied, giving a lower bound on *I/O* of $Q(G_{|V_t}) \geq N - S$. As the may-spill set of the different subgraphs do not

```
  for(t=0; t<M; t++) {
      s = 0;
      for(i=0; i<N; i++)
S1:       s += A[i];
      for(i=0; i<N; i++)
S2:       A[i] += s;
  }
```

(a) Code

(b) CDAG for M=4, N=4. White vertices correspond to S1, gray vertices to S2.

(c) Decomposition of the CDAG

Figure 3.3 – Example 2

intersect, the individual complexities can be summed over $t = 1, \ldots, M-1$, providing a lower bound for the whole CDAG:

$$Q(G) \geq (M-1)(N-S).$$

3.2.2 Bounded combination

The main procedure of IOLB (Sec. 3.5.3) selects a bounded set of (possibly overlapping) sub-CDAGs and computes their individual complexities. The objective of Alg. 1 is to combine (sum) as many non-interfering (disjoint may-spill sets) complexities as possible. It does so using a greedy approach: Assume there are two sub-CDAGs both with a "high" complexity but with non-disjoint may-spill sets. Alg. 1 will select the one with the highest complexity, recompute the complexity of the second after removing the intersecting part, and then sum them up. The overall set of sub-CDAGs is iteratively processed this way (and the complexities summed-up) until empty or negligible complexities remain. The comparison (what is "higher") is done using *instances of parameter values*, simply evaluating the corresponding symbolic expressions. It should be emphasized that the final bound is a valid lower bound for *any* parameter values, the instances of parameter values are only used for heuristics.

Let us have a look at the example on Fig. 3.4. In the original code (3.4a), notice that `k` is the outer loop index, meaning that `A[k]` will have been modified either in the current loop iteration or the previous one depending on the order between `i` and `k` (Floyd-Warshall exhibits the same pattern, with three loops instead of two). This is made clear in the single-assignment form (3.4b), and can be visualized in the CDAG representation (3.4c). The dependences on input values are grayed out in (3.4b) and omitted in (3.4c), and we will ignore them in the discussion to keep the explanations simple.

```
function combine_subQ
    input  : A DFG G, an instance I, a set of complexities 𝒬
    output: A combined complexity Q^I
    Q^I = 0;
    Let G' be a copy of G;
    while 𝒬 ≠ ∅ do
        let Q such that Q(I) = max_𝒬 Q(I);
        𝒬 = 𝒬 − {Q};
        if Q(I) = 0 then return Q^I;
        if G' ∩ Q.may-spill ≠ ∅ then
            Recompute Q assuming CDAG G';
            𝒬 = 𝒬 ∪ {Q};
        else
            Q^I := Q^I + Q;
            G' = G' − Q.may-spill
    return Q^I
```

Algorithm 1: Summing lower bound expressions by removing interferences

Considering only the statement vertex S in the DFG, the dependency analysis gives the following relations:

$$
\begin{aligned}
R_1 &= \{S[k-1,i] \rightarrow S[k,i] : \ 1 \le k < N \ \wedge \ 0 \le i < N\} \\
R_2 &= \{S[k-1,k] \rightarrow S[k,i] : \ 1 \le k < N \ \wedge \ 0 \le i < k\} \\
R_3 &= \{S[k,k] \ \rightarrow \ S[k,i] : \ 0 \le k < N \ \wedge \ k < i < N\}
\end{aligned}
$$

The image spaces of R_2 and R_3 provide a natural decomposition of the CDAG into two non-interfering sub-CDAGs, as shown in (3.4d). On each part, the pattern is similar to that of Example 1 on page 14, and the geometric approach gives a lower bound (omitting lower order terms) $Q(G_i) \ge \frac{N^2}{2S}$. Since they do not interfere, Alg. 1 will return their sum $Q(G) \ge \frac{N^2}{S}$, independently of the parameter instance.

3.2.3 Loop parametrization

As done on the example above, IOLB can compute the *I/O* complexity of some inner loop nests of a bigger enclosing loop nest and sum them. To this end, our scheme performs what we call *loop parameterization*. Loop parameterization considers each individual sub-CDAG where the outermost indices are fixed (our algebraic formulation allows us to consider such indices as parameters without the need to explicitly enumerate them) enriched by their input vertices. Taking the notations

$$\overline{V_i} = V_i \cup \mathrm{In}(V_i)$$

parametrizing the outer "t" loop with $t = \Omega$ (with Ω a parameter) allows us to compute Q^{Ω}, a (parametric) lower bound for each individual value of $\Omega = 1, \ldots, M-1$ (The sub-CDAG for

```
Parameters: N;
Input: A[N]; Output: A[N];
for(k=0;k<N;k++)
  for (i=0; i<N; i++)
    A[i] = f(A[i],A[k]);
```

(a) C-like code

```
Parameters: N;
Input: A[N]; Output: S_{N-1}[N];
for (0 ≤ k < N and 0 ≤ i < N)
  if (k==i==0): S_{0,i} = f(A[0],A[0]);
  else if (k==0): S_{0,i} = f(A[i],S_{0,0});
  else if (i<=k): S_{k,i} = f(S_{k-1,i}, S_{k-1,k});
  else if (i>k): S_{k,i} = f(S_{k-1,i}, S_{k,k});
```

(b) Corresponding single assignment form

(c) Corresponding CDAG for N=5. Input nodes A[N] are omited.

(d) Decomposition into two non-interfering sub-CDAGs. Sources are in gray. May-spill sets are encircled.

Figure 3.4 – Example 3

$\Omega = 0$ does not have the same pattern so it is ignored), and combine them

$$Q = \sum_{1 \le \Omega < M} Q^{\Omega} = \sum_{1 \le \Omega < M} Q_{|\overline{\{v \in V,\ t = \Omega\}}} = \sum_{1 \le \Omega < M} N - S = (M-1) \cdot (N-S).$$

In more complex cases, the parametric bound can depend on the outer loop parameter Ω, and we use formulas for sum of polynomials.

In ISL terms, this is done by making the outer loop index a parameter. Here the original domain

$$D_{S_1} = [M,N] \to \{S_1[t,i] : 0 < t < M \ \wedge \ 0 < i < N\}$$

becomes

$$D^{\Omega}_{S_1} = [M,N,\Omega] \to \{S_1[t,i] : t = \Omega \ \wedge \ 0 < t < M \ \wedge \ 0 < i < N\}.$$

The corresponding parts of the algorithms are highlighted in Algorithm 6 on page 38.

3.3 K-partition bound derivation

In this section, we explain how to apply the geometrical reasoning of Sec. 3.1.4 on a CDAG $G = (V,E)$, using its compact representation as a DFG. We also present, in 3.3.1, a generaliza-

tion of one of the techniques introduced in [20, 35, 48] that these authors used to derive a tighter lower bound for matrix multiplication.

To apply Lemma 2 on G, we need to find a lower bound on the minimum number of subsets h in any K-partition of G. The general reasoning is as follows:

1. Embed V in a geometric space through a map $\rho : P \subseteq V \mapsto E \subseteq \mathbb{Z}^d$, such that two disjoint subsets of V are mapped to disjoint subsets of $\mathbb{Z}^d$. We have
$$\left|\rho(P)\right| \leq |P|.$$
2. Use the DFG representation to find a subset $V' \subseteq V$ and a set of projections (group homomorphisms) $\phi_1, \ldots, \phi_m$ with the property that:
$$\text{Any } K\text{-bounded set } P \subseteq V' \setminus \text{Sources}(V') \text{ satisfies} \left|\phi_j(\rho(P))\right| \leq K. \tag{3.4}$$
3. Using Theorem 1, derive an upper bound U on $\left|\rho(P)\right|$ for any K-bounded P. This provides a lower bound $\left\lceil \frac{|V' \setminus \text{Sources}(V)|}{U} \right\rceil$ on the number h of disjoint K-bounded sets in $V' \setminus \text{Sources}(V')$.

3.3.1 Geometric embedding, DFG-paths and projections

Let S_k be some fixed DFG-vertex (corresponding to one program statement). Let $Q_1, \ldots, Q_m$ be DFG-paths all ending in S_k, with a common image space $D_k = \{S_k[i_1, \ldots, i_d] : \ldots\}$.

The embedding ρ is defined as:
$$\rho(P) = \{(i_1, \ldots, i_d) \mid S_k[i_1, \ldots, i_d] \in P\}.$$
Vertices corresponding to statement S_k are mapped to their corresponding d-dimensional point, and other vertices are ignored.

Definition 12 (embedded projections)**.** *For a given path Q with relation R_Q, the geometric projection ϕ_Q is defined as follows:*

— *If the path is a broadcast path with $R_Q = \{S_j[j_1, \ldots, j_{d'}] \to S_k[i_1, \ldots, i_d] : \ldots\}$ for some statement S_j (not necessarily $\neq S_k$), then the projection is directly given by the path relation $\phi_Q(i_1, \ldots, i_d) = (j_1, \ldots, j_{d'})$.*

— *If the path is a chain circuit with $R_Q = \{S_k[i_1, \ldots, i_d] \to S_k[i_1 + \delta_1, \ldots, i_d + \delta_d] : \ldots\}$, then the projection is the orthogonal projection on the hyperplane in $\mathbb{Z}^d$ defined by orthogonal vector $\delta = (\delta_1, \ldots, \delta_d)$. Its explicit formulation can be computed but is not needed here.*

In the case of a broadcast, it is straightforward that ϕ_Q satisfies (3.4), because $R_Q^{-1}(P)$ is included in $\text{In}(P)$ for any $P \subseteq V$, so $\left|\phi_Q(\rho(P))\right| \leq |\text{In}(P)| \leq K$ for any K-bounded P.

In the case of a chain circuit, let us call $I_Q(P) = R_Q^{-1}(P) \setminus P$. This is basically the In-set of P restricted to edges corresponding to DFG-path Q, so $I_Q(P) \subset \text{In}(P)$. The projection ϕ_Q associates one point to each straight line directed by δ. Since $P \subseteq V \setminus \text{Sources}(V)$, there is at least one point in $\phi_Q(\rho(P))$ for every nonempty chain in P, and $\left|\phi_Q(\rho(P))\right| \leq |\text{In}(P)| \leq K$.

Example Consider paths $p_1 = (e_2)$ and $p_2 = (e_3)$ in Fig. 3.2. p_1 is a broadcast path with relation $\{C[t] \to S[t,i]\}$, so the corresponding projection is $\phi_1(t,i) = (t)$. p_2 is a chain path with relation $\{S[t,i] \to S[t+1,i]\}$, so the corresponding projection is $\phi_2(t,i) = \text{proj}_{(1,0)}(t,i) = (0,i)$ (see Fig. 2.3b).

Summing projections

In some cases, the parts of the In-set of a vertex set associated with two given path relations are actually disjoint. Let Q_1 and Q_2 be two such paths, such that $R^{-1}_{Q_1}(P) \cap R^{-1}_{Q_2}(P) = \emptyset$ for any $P \subseteq V \setminus \text{Sources}(V)$. If these are two broadcast paths, then since $R^{-1}_{Q_i}(P) \subset \text{In}(P)$, any K-bounded set P satisfies the stronger inequality:

$$\left|\phi_{Q_1}(\rho(P))\right| + \left|\phi_{Q_2}(\rho(P))\right| \leq K$$

The same holds if Q_1 is a chain circuit and $R^{-1}_{Q_1}(P) \cap R^{-1}_{Q_2}(P) = \emptyset$, by a similar argument.

We say two paths Q_1 and Q_2 are *independent* for domain D_S if $R^{-1}_{Q_1}(D_S) \cap R^{-1}_{Q_2}(D_S) = \emptyset$. We can build the *DFG-path interference graph*: vertices are paths $Q_1, \ldots, Q_m$ and there is an edge between any independent pair of paths. In this graph, if vertices $Q_{i_1}, \ldots, Q_{i_t}$ form a clique, then

$$\text{Any } K\text{-bounded set } P \subseteq D_k \text{ satisfies } \sum_s \left|\phi_{i_s}(P)\right| \leq K.$$

Example Looking again at Example 1, it is straightforward to check that paths p_1 and p_2 are independent, so a K-bounded set P actually satisfies $\left|\phi_1(P)\right| + \left|\phi_2(P)\right| \leq K$.

Combining several such inequalities, such that every projection occurs at least once, leads to a general constraint of the form:

$$\sum_{j=1}^{m} \beta_j \left|\phi_j(E)\right| \leq K,$$

for some positive coefficients β_j. This is achieved by finding a set of maximal cliques covering all vertices in the interference graph, and summing the corresponding inequalities. Computing these β_j's is the role of function coeffInterf in Algorithm 4.

In this case, a tighter bound can be derived:

$$|E| \leq \prod_{j=1}^{m} \left|\phi_j(E)\right|^{s_j} \leq \left(\frac{K}{\sum_j s_j}\right)^{\sum_j s_j} \prod_{j=1}^{m} \left(\frac{s_j}{\beta_j}\right)^{s_j}.$$

The following lemma establishes this result.

Lemma 5. *Let $0 \leq s_1, s_2, \ldots, s_m \leq 1$ and $C > 0$. Let x_j be nonnegative integers and $\beta_j > 0$ for $1 \leq j \leq m$ such that $\sum_{j=1}^{m} \beta_j x_j \leq C$. Then*

$$\prod_{j=1}^{m} x_j^{s_j} \leq \left(\frac{C}{\sum_j s_j}\right)^{\sum_j s_j} \prod_{j=1}^{m} \left(\frac{s_j}{\beta_j}\right)^{s_j}. \tag{3.5}$$

Proof. We use Lagrange multipliers to find the constrained maximum of the function $\psi : x \in \mathbb{R}^m \mapsto \prod_{j=1}^{m} x_j^{s_j}$.

$$L(x, \lambda) = \prod_{j=1}^{m} x_j^{s_j} - \lambda \left(\sum_{j=1}^{m} \beta_j x_j - C\right)$$

Partial derivatives are:

$$\frac{\partial L}{\partial x_j} = s_j x_j^{s_j - 1} \prod_{k \neq j} x_k^{s_k} - \lambda \beta_j \quad , 1 \leq j \leq m$$

$$\frac{\partial L}{\partial \lambda} = C - \sum_{j=1}^{m} \beta_j x_j$$

Setting them to be 0, we get:

$$\lambda \beta_j x_j = s_j \prod_{k=1}^{m} x_k^{s_k}, 1 \leq j \leq m$$

Summing for $1 \leq j \leq m$ gives:

$$\lambda \sum_{j=1}^{m} \beta_j x_j = \lambda C = \left(\prod_{k=1}^{m} x_k^{s_k} \right) \cdot \sum_{j=1}^{m} s_j$$

From which we derive:

$$x_j = \frac{C s_j}{\beta_j \sum_j s_j}, 1 \leq j \leq m$$

And finally:

$$\psi(x) \leq \prod_{j=1}^{m} x_j^{s_j} = \prod_{j=1}^{m} \left(\frac{C s_j}{\beta_j \sum_i s_i} \right)^{s_j} = \left(\frac{C}{\sum_j s_j} \right)^{\sum_j s_j} \prod_{j=1}^{m} \left(\frac{s_j}{\beta_j} \right)^{s_j}.$$

□

With this more general formulation, the choice of the s_j coefficients is more involved than the linear optimization problem of Sec.3.1.4, and is developed in Sec. 3.3.3.

Kernel subgroup lattice

As already mentioned, the subgroup lattice (Def. 10 used in Lemma 3) is not necessarily finite, so it is better to build it step-by-step, updating it each time we add a new path. We set a time limit for the computation to converge, and do not add the path if this limit is reached. Function subspace_closure in Algorithm 2 tentatively updates the current subgroup lattice with a new one, returning the original lattice in case of a timeout.

3.3.2 Finding paths

The function that generates the set of paths $\mathcal{P} = \{Q_1, \ldots, Q_m\}$ for a DFG-vertex S is named genpaths (Alg. 3). Starting from S, it uses a simple backward traversal (backward DFS) that favors walking through predecessors with largest domain. As the number of paths can be combinatorial, IOLB sets a timeout to avoid a computational blow-up. For a path $P = (S_1, \ldots, S_t = S)$, we store sub-path relations $R_{S_i \to S}$ for every intermediate statement S_i. This is necessary to "remember" exactly which CDAG vertices are included in the computation.

```
function subspace_closure
    input  : Lattice of subgroups 𝓛, subgroup to add K
    output: updated set of subspaces 𝓛'
    𝓛' = 𝓛;
    while not timeout do
        if ∃H ∈ 𝓛', H ∩ K ∉ 𝓛' then 𝓛' = 𝓛' ∪ {H ∩ K};
        else if ∃H ∈ 𝓛', H + K ∉ 𝓛' then 𝓛' = 𝓛' ∪ {H + K};
        else return 𝓛';
    return 𝓛 if timeout
```

Algorithm 2: Update the subgroup lattice with a new kernel

```
function genpaths
    input  : a Data-flow graph G = (𝒮, 𝒟), a statement S ∈ 𝒟
    output: set of paths 𝒫
    start from S and backward traverse to build any possible path that reaches S;
    drop paths for which R_{S'→S}(D_{S'}) has lower dimensionality than D_S;
    drop paths if ¬(isBroadcast(P) ∨ isChain(P));
```

Algorithm 3: Generate paths

3.3.3 Computing the lower bound

Once we have found a path combination, it is quite straightforward to apply the theoretical results introduced above.

This is detailed in function sub_paramQ_bypartition in Alg. 4. Here, the role of the function call to coeffInterf is to compute the coefficients β_j. It does so by finding a clique cover of the DFG-path independence graph and summing the constraints formed by each clique as explained in Sec. 3.3.1. The values for coefficients s_j that satisfy inequalities 3.2b are then determined using convex optimization (IPOPT [58] in our case) so as to minimize as much as possible the quantity

$$U = \left(\frac{(S+T)}{\sum_j s_j}\right)^{\sum_j s_j} \prod_{j=1}^{m} \left(\frac{s_j}{\beta_j}\right)^{s_j}. \tag{3.6}$$

Indeed this expression being an upper bound on $\left|\rho(P)\right|$ (see Lemma 5), minimizing it has the effect of tightening [1] the computed *I/O* complexity.

The constraints in (3.2b) describe a convex polyhedron, but the objective function (3.6) is not convex. In the basic case when all projections are simply bounded by $(S+T)$, the objective is

$$U := (S+T)^{\sum_j s_j},$$

so a natural objective is to minimize $\sum_j s_j$ in this generalized case. It can be easily checked that (3.6) is indeed equal to this when $\beta_j = \frac{1}{m}$ for all j and $s_1 = \cdots = s_j$.

Notice that the first factor in the expression of U depends only on the value of the sum. So once $\sum s_j$ is fixed, it is natural to minimize the second factor $\prod_{j=1}^{m} \left(\frac{s_j}{\beta_j}\right)^{s_j}$, which is convex as

1. Observe that any values of s_js lead to a correct bound

function *sub_paramQ_bypartition*

input : paths $\mathcal{P} = \{P_1, \ldots, P_m\}$ with domain D and lattice $\mathcal{L}$

output: complexity Q

$I := \bigcup_{P_i \in \mathcal{P}} R_{P_i}^{-1}(D)$;

$d := \dim(D)$;

$(\beta_1, \ldots, \beta_m) := \mathsf{coeffInterf}(\mathcal{P}, D)$;

$(s_1, \ldots, s_m) :=$ convex-opt {

variables: $\{s_1, \ldots, s_m \in \mathbb{Q}^+\}$

objective: minimize $\sum_j s_j$ and then $\prod_j \left(\frac{s_j}{\beta_j}\right)^{s_j}$

constraints: $\forall H \in \mathcal{L},\ \sum_j s_j \mathrm{rank}\Big(\phi_j(H)\Big) \geq \mathrm{rank}(H)$ where $\phi_j = \mathrm{proj}^{\perp}_{\mathsf{Ker}(P_j)}$} ;

$T := \frac{1}{\sum_j s_j - 1} S$;

$U := \prod_{j=1}^{m} \left(\frac{(S+T)s_j}{\beta_j \sum_i s_i}\right)^{s_j}$;

$Q := \max\Big(\left\lfloor \frac{|D|}{U} \right\rfloor \times T - |I|, 0\Big)$;

$Q.\mathsf{may\text{-}spill} := \mathsf{may\text{-}spill}(\mathcal{P}, D)$;

function *coeffInterf*

input : paths $\mathcal{P} = \{P_1, \ldots, P_m\}$, domain D

output: coefficients $(\beta_1, \ldots, \beta_m)$ such that $\sum \beta_j \phi_j(E) \leq K$ for any K-bounded set E

$G :=$ graph with $V := P_1, \ldots, P_m$ and $E := (P_i, P_j),\ R_{P_i}^{-1}(D) \cap R_{P_i}^{-1}(D) \neq \emptyset$;

$\mathcal{I} :=$ set of maximal independent sets of G such that every node belongs to at least one set (greedy construction);

$\beta_j := \#\{I \in \mathcal{I}, P_j \in I\}/|\mathcal{I}|$;

function *may-spill*

input : paths $\mathcal{P} = \{P_1, \ldots, P_m\}$, domain D

output: may-spill set D^{ms}

$D^{\mathrm{ms}} := \emptyset$;

foreach broadcast path $P_i = (S_0, S_1, \ldots, S_t = S) \in \mathcal{P}$ **do**

$D^{\mathrm{ms}} := D^{\mathrm{ms}} \cup \left(\bigcup_{j=0}^{t} R_{S_j \to S}^{-1}(D)\right)$;

foreach chain circuit $P_i = (S_0, S_1, \ldots, S_t = S) \in \mathcal{P}$ **do**

$D^{\mathrm{ms}} := D^{\mathrm{ms}} \cup \left(\bigcup_{j=1}^{t} R_{S_j \to S}^{-1}(D)\right) \cup \left(R_{S_0 \to S}^{-1}(D) \cap \left(\bigcup_{k \neq j} R_{P_k}^{-1}(D)\right)\right)$;

function *Ker*

input : path P_j

output: linear space K such that the orthogonal projection $\phi_j = \mathrm{proj}^{\perp}_{\mathsf{Ker}(P)}$

if $\mathsf{isBroadcast}(P_j)$ **then return** $\mathsf{Ker}(j_1 \ldots j_{d'})$ where $R_{P_j} = \{T[j_1, \ldots, j_{d'}] \to [i_1, \ldots, i_d] : \ldots\}$;

if $\mathsf{isChain}(P_j)$ **then return** $(\delta_1, \ldots, \delta_d)$ where $R_{P_j} = \{S[i_1, \ldots, i_d] \to S[i_1 + \delta_1, \ldots, i_d + \delta_d] : \ldots\}$;

Algorithm 4: Derivation of a lower bound from a path combination with the partition method

```
for (b = 0; b < B; b++)
  for (c = 0; c < C; c++)
    for (f = 0; f < F; f++)
      for (x = 0; x < X; x++)
        for (y = 0; y < Y; y++)
          for (h = 0; h < H; h++)
            for (w = 0; w < W; w++) {
              Out[f,x,y,b] += Image[x+h,y+w,c,b] * Filter[f,h,w,c];
}
```

Figure 3.5 – Convolution kernel.

a function of $(s_1, \ldots, s_m)$. This convex optimization problem can be solved with an appropriate tool, such as IPOPT [58]. The last step amounts to setting an appropriate value for T that provides a lower bound of $\left\lfloor \frac{|D_S|}{U} \right\rfloor \times T - |I|$ (see Lemma 2) as big/tight as possible. Here, D_S corresponds to $V \setminus \text{Sources}(V)$ in the CDAG view, and I is the frontier of the domain, corresponding to $\text{Sources}(V)$. T is chosen as $\frac{1}{\sum_j s_j - 1} S$, because it maximizes the first term asymptotically.

Finally, we store the may-spill set corresponding to the CDAG for which this lower bound is valid.

Taking the example given in Figure 3.1, it is sufficient to check condition (3.2b) on $H_1 = \{(0, i)\}, H_2 = \{(t, 0)\}$, and the optimization problem is:

$$\begin{aligned} \text{Minimize} \quad & s_1 + s_2 \text{ and then } s_1^{s_1} s_2^{s_2} \\ \text{s.t.} \quad & s_1 \geq 1, \;\; s_2 \geq 1 \end{aligned}$$

The solution is $s_1 = s_2 = 1$, and $U = ((S+T)/2)^2 = S^2$ (because $T = \frac{1}{s_1 + s_2 - 1} S = 1 \cdot S$). $|D_S| = MN$ and $|I| = N + M$, so

$$Q \geq \left\lfloor \frac{MN}{S^2} \right\rfloor \times S - N - M.$$

3.3.4 Extensions: Small Dimensions and Reduction Detection

We introduced two extensions to the method described above, that allow IOLB to derive much tighter bounds for some programs. For example, on a convolution (Fig. 3.5), the lower bound derived by the initial algorithm is several asymptotic orders below the upper bound. This lower bound can be improved by taking into account (i) the *reductions* across multiple dimensions, which refines the dependence analysis performed in the first steps of the algorithm, and (ii) *small dimensions*. Small dimensions are dimensions with a much smaller number of iterations than the size of the fast memory S. This property can be used to improve the power factor of the asymptotic bound. Figure 3.6 shows the successive improvements of the initial lower bound expression for convolution, eventually leading to matching the upper bound. Both extensions have been implemented in IOLB, and allow us to obtain asymptotically tight bounds for tensor contraction and convolution computations. These bounds are shown in table 3.2 on page 49.

Initial lower bound:

$$BFXY + CBXY$$

With reduction detection:

$$\frac{BCXYWH}{S}$$

With reduction detection and small dimension handling:

$$\frac{2BCXY\sqrt{WH}}{\sqrt{S}}$$

Upper bound when $W, H << S$ [18]:

$$\frac{2BXYCF\sqrt{WH}}{\sqrt{S}}$$

Figure 3.6 – Lower bounds for convolution, without and with IOLB extensions (only dominant terms are shown).

Using Reductions

A *reduction* is the successive application of an associative and commutative binary operator over a set of values. For example, in the convolution kernel described in Figure 3.5, there is a reduction over 3 dimensions (c, h and w) with the addition operation. We only consider reductions that sum all the values along some dimensions, called *reduced dimensions*. This property can be used to sum the values of a set in any order, instead of having a fixed sequence of summation.

Because the program is expressed as a loop nest, its reduction is sequential. Thus, when the DFG is extracted from a code, there is a chain of dependencies linking nodes along reduced dimensions. When there are more than 2 reduced dimensions, this negatively impacts our study of DFG-paths, thus the quality of the extracted projections ϕ_j and the upper bound found on the size of the K-bounded set E.

We detect a simple class of reductions by performing a pattern-matching on affine dependencies in the DFG. In this class, the reduced dimensions are defined over a rectangular domain and that the summation is performed in a lexicographic order over a permutation of these reduced dimensions. Once a reduction is detected, we replace the sequential chain of dependencies (between the nodes of the reduction) by two sets of broadcast dependencies: (i) starting from the computations using the final result of the reduction to all the nodes of the reduction, and (ii) starting from the initialization of the reduction to the nodes of the reduction.

On the convolution example of Figure 3.5, if we do not detect the reduction, the dependence on `Out` is along dimension s, and the corresponding projection found by IOLB is $\phi_1(b,c,f,x,y,h,w) = (b,c,f,x,y,h,0)$. But if we detect it, the projection of the new path becomes $\phi_1(b,c,f,x,y,h,w) = (b,0,f,x,y,0,0)$, which leads to a bigger kernel and better constraints on s_j.

Adapting the Method to Small Dimensions

The geometrical argument of the K-partition method consists, at a high level, in bounding the size of the largest sub-CDAG P with a given operational intensity, by bounding its extent along each dimension. The bound obtained in the end is some power of the cache size S, which makes sense when loop limits along all dimensions are large (i.e., of the same magnitude and with their product $>> S$). However, in some programs, a subset of dimensions has small sizes in most real life implementations: in convolutions, H and W are almost always less than 10. When this is the case, the extent of P along *small* dimensions is more constrained by the total extent of the CDAG along this dimensions, than by the geometrically derived bound depending on S.

Formally, we assume that some of the dimensions of the iteration domain are *small*, i.e., their size is orders of magnitude smaller than S. For notation simplicity, let us assume that they correspond to the first q dimensions in statement domain $D_k = \{S_k[i_1,\ldots,i_d] : \ldots\}$. We adapt the partitioning method by considering an additional group homomorphism ϕ_{sd} to the application of Theorem 1. This group homomorphism is a projection of the space on the small dimensions:

$$\phi_{sd} = (i_1,\ldots,i_q).$$

Let $N_{sd} = \left|\phi_{sd}(\rho(D_k))\right|$ be the size of the image of D_k through ϕ_{sd} (in most cases, this will be equal to the product of small dimension sizes, e.g., $N_{sd} = H \cdot W$ for convolution).

Introducing a coefficient s_{sd} associated with ϕ_{sd}, we can rewrite Theorem 1 as follows.

Assuming:

$$\operatorname{rank}(H) \leq \sum_{j=1}^{m} s_j \cdot \operatorname{rank}\left(\phi_j(H)\right) + s_{sd} \cdot \operatorname{rank}(\phi_{sd}(H)) \quad \text{for all subgroups } H \in \mathcal{L}_{\phi_1,\phi_2,\ldots,\phi_m} \tag{3.2s}$$

Then:

$$|E| \leq \prod_{j=1}^{m} \left|\phi_j(E)\right|^{s_j} \cdot N_{sd}^{s_{sd}} \quad \text{for all nonempty finite sets } E \subseteq \mathbb{Z}^d. \tag{3.3s}$$

The upper bound on $\left|\rho(P)\right|$ becomes:

$$U = \left(\frac{(S+T)}{\sum_j s_j}\right)^{\sum_j s_j} \prod_{j=1}^{m} \left(\frac{s_j}{\beta_j}\right)^{s_j} N_{sd}^{s_{sd}}. \tag{3.6s}$$

Note that the constraints on the s_j are relaxed due to the contribution of s_{sd}, giving us an opportunity to find smaller values of s_j, compared to the usual method. Moreover, in order to get a bound as tight as possible, we should minimize $\sum s_j$ first, s_{sd} next, and finally $\prod\left(\frac{s_j}{\beta_j}\right)^{s_j}$.

Let us illustrate this method on the convolution example described in Figure 3.5.

Let us start with the "standard" derivation, i.e. without taking small dimensions into account. By examining the affine dependencies of the program, we find three projections ϕ_1, ϕ_2 and ϕ_3, described in Figure 3.7a.

As stated by Theorem 1, we consider the subgroups $\mathcal{H}$ generated from the combination of kernel vectors of the ϕ_j. This yields constraints shown in Figure 3.7b, ignoring the rightmost column. Their corresponding subgroups are shown in the leftmost column.

$$\begin{aligned}
\phi_1(b,c,f,x,y,h,w) &= (b,0,f,x,y,0,0) && \text{(from reduction)}\\
\phi_2(b,c,f,x,y,h,w) &= (x+h,y+w,c,b) && \text{(from Image)}\\
\phi_3(b,c,f,x,y,h,w) &= (f,h,w,c) && \text{(from Filter)}\\
\phi_{sd}(b,c,f,x,y,h,w) &= (h,w) && \text{(small dimensions)}
\end{aligned}$$

(a) Projections (group homomorphisms) used in Brascamp-Lieb inequality.

$(\vec{f})$	$1 \leq s_1 + s_3$	
$(\vec{b})$	$1 \leq s_1 + s_2$	
$(\vec{c})$	$1 \leq s_2 + s_3$	
$(\vec{x})$	$1 \leq s_1 + s_2$	
$(\vec{w})$	$1 \leq s_2 + s_3$	$+s_{sd}$
$(\vec{w}-\vec{x})$	$1 \leq s_1 + s_3$	$+s_{sd}$
$(\vec{w},\vec{x})$	$2 \leq s_1 + s_2 + s_3$	$+s_{sd}$
$(\vec{y})$	$1 \leq s_1 + s_2$	
$(\vec{h})$	$1 \leq s_2 + s_3$	$+s_{sd}$
$(\vec{h}-\vec{y})$	$1 \leq s_1 + s_3$	$+s_{sd}$
$(\vec{h},\vec{y})$	$2 \leq s_1 + s_2 + s_3$	$+s_{sd}$

(b) Constraints on s_j from Brascamp-Lieb inequality.

Figure 3.7 – Partitioning method - Brascamp-Lieb inequality applied to a convolution, without and with small dimensions (H and W).

The three projections are independent so $\beta_1 = \beta_2 = \beta_3$. The optimization problem can be written as:

$$\text{Minimize } s_1 + s_2 + s_3 \text{ under constraints:} \quad \begin{aligned} s_1 + s_2 &\geq 1\\ s_1 + s_3 &\geq 1\\ s_2 + s_3 &\geq 1\\ s_1 + s_2 + s_3 &\geq 2 \end{aligned}$$

The solution is $s_1 = s_2 = s_3 = \frac{2}{3}$ and gives the following upper bound on $\left|\rho(P)\right|$:

$$U = \left(\frac{S+T}{3}\right)^2.$$

Finally we set $T = \frac{1}{s_1+s_2+s_3-1} S = S$, and get the final bound (ignoring lower-order terms corresponding to the input):

$$Q = \frac{BCXYWH}{U} \cdot T = \frac{9}{4}\frac{BCXYWH}{S}.$$

Now, we will see that taking small dimensions into account leads to a tighter bound. Here small dimensions are h and w, and $N_{sd} = W \cdot H$. Compared to the first derivation, we add a projection ϕ_{sd}, shown in Figure 3.7a. The presence of this new projection adds a new coefficient s_{sd} to the constraints on s_js, as shown in Figure 3.7b.

The optimization problem becomes:

$$\text{Minimize } s_1 + s_2 + s_3 \text{, then } s_{sd} \text{ under constraints:} \quad \begin{aligned} s_1 + s_2 &\geq 1\\ s_1 + s_3 &\geq 1\\ s_2 + s_3 &\geq 1\\ s_1 + s_2 + s_3 + s_{sd} &\geq 2 \end{aligned}$$

The solution is now $s_1 = s_2 = s_3 = s_{sd} = \frac{1}{2}$, and we get

$$U = \left(\frac{S+T}{3}\right)^{\frac{3}{2}} \cdot (WH)^{\frac{3}{2}}.$$

Finally we set $T = \frac{1}{s_1+s_2+s_3-1} S = 2S$, and get the final lower bound on I/O (ignoring again lower-order terms):

$$Q = \frac{BCXYWH}{U} \cdot T = \frac{2BCXY\sqrt{WH}}{\sqrt{S}}.$$

This is a tighter lower bound for most real-life configurations.

3.4 Wavefront bound derivation

3.4.1 Theoretical results

An alternative way to derive data movement lower bounds in the no-recomputation model is the *wavefront* abstraction. At any point in an execution of a RW-game, the wavefront is the set of vertices that have been computed but whose result is still needed by some successor (sometimes called the set of *live* vertices). If the size of the wavefront at some point in the execution is greater than the size of the fast memory, then necessarily some vertices have to be spilled to the slow memory and thus loaded using rule ($R1$). This is formalized in the definition and lemma below:

Definition 13 (Wavefront)**.** *Let $\mathcal{R}$ be an execution of the RW-game on a CDAG G, and v a vertex of G. Consider the time t in the execution just before v has been computed (i.e., just after a white pebble has been placed on v using rule ($R2$)). The* wavefront *$W_{\mathcal{R}}(v)$ in execution $\mathcal{R}$ is the set of vertices that, a time t, have been computed (i.e., have a white pebble) but have some successor that does not.*

Lemma 6 (Min-wavefront [22])**.** *Let S be the capacity of the fast memory, and $G = (V, E)$ be a CDAG. Let $w_G^{min} = \min_{\mathcal{R}}(\max_{v \in V} |W_{\mathcal{R}}(v)|)$, so that any valid RW-game on G has a wavefront of size at least w_G^{min}. Then,*

$$Q \geq w_G^{min} - S.$$

This lemma is quite general and cannot be applied directly, as $w_G^{\min}$ is not usually computable. We thus provide the following result, which uses a condition on the structure of the CDAG to get a lower bound on the size of a wavefront in any execution.

Corollary 1. *Let $G = (V, E)$ be a CDAG, and V_1, V_2 be disjoint subsets of V such that every vertex in V_2 is reachable from every vertex in V_1 through some path in G. Let $L_1, \ldots, L_m$ be disjoint paths in G, starting in V_1 and ending in V_2. More formally:*

$$\begin{cases} \forall L_j = (v_1^j, \ldots, v_{l_j}^j), \quad v_1^j \in V_1 \text{ and } v_{l_j}^j \in V_2 \\ V_1 \times V_2 \subset E^* \end{cases}$$

Then,

$$w_G^{min} \geq m.$$

Figure 3.8 – Application of Cor. 1 on a sub-CDAG. Paths L_j (single edge) are shown in bold.

By Lemma 6, this implies:

$$Q \geq m - S.$$

Proof. Let v be the first vertex to be computed among vertices of V_2 in some fixed RW-game on G. By the second condition, every vertex in V_1 must have been computed since v depends on all of them. Just before a red pebble is placed on v, no vertex in V_2 has a red pebble. Therefore there is a live vertex (that has a red pebble and a successor without one) in every path L_j. Thus $w_G^{\min} \geq m$. □

The common case to use this technique to get a strong data movement lower bound is to combine it with the parametric CDAG decomposition (Sec. 3.2.3).

Example In the example of Fig. 3.3, Corollary 1 can be applied on each subgraph, with V_1 and V_2 chosen as shown on Fig. 3.8. It can be easily checked that every vertex in V_1 can reach every vertex in V_2, and there are N disjoint paths L_j, so this gives a lower bound

$$Q(G_{|V_t}) \geq N - S.$$

3.4.2 Implementation

To apply this technique, our algorithm tries to uncover a set of disjoint paths satisfying the hypotheses of Corollary 1. To reduce the search space, Algorithm 5 looks for a much more constrained pattern, in which all disjoint paths L_j begin and end in different instances of some statement S, with an increment in the innermost parametrized loop index (see Sec. 3.2.3). This amounts to finding an injective circuit in the DFG with a relation of the form

$$\{S[I_1 \dots I_d, i_{d+1} \dots i_D] \rightarrow S[I_1 \dots I_d + 1, i_{d+1} \dots i_D]\}.$$

Intuitively, we look at two "slices" of the CDAG, each of them of dimension $D-d$, representing two successive iterations of the body of the loop iterating over dimension d (with index I_d and $I_d + 1$, respectively). We try to find subsets of these two slices (corresponding to V_1 and V_2) such that every vertex in the first one can reach every vertex in the second one.

In Algorithm 5, the first loop computes the following relations:

— $R_{S \rightarrow S}$ is the union of all path relation of elementary circuits from S to itself,

— $RL_{S \rightarrow S}$ is the union of all *affine* path relations (where every subpath relation is affine) of elementary circuits $S \rightarrow S$,

— $R_{S \rightarrow *}$ is the union of all path relations of elementary paths from S to any other DFG-vertex.

This is then used to compute the relation R_{I_d}. Here I_d is the index of the innermost parametrized loop, and R_{I_d} is the restriction of $RL_{S\to S}$ to paths that do exactly one step along dimension d, not changing any other index. This represents a set of disjoint paths going from instances of statement S in "slice" I_d to instances of S in "slice" I_d+1.

To apply Corollary 1, we need to restrict R_{I_d} to a domain W where every starting vertex reaches every ending vertex. To do so, we first compute $X = \left(R_{\text{complete}} - (R_{S\to S})^*\right)\left(\text{Dom}\left(R_{I_d}\right)\right)$, that is, all vertices in "slice" I_d+1 that are *not reachable* from "slice" I_d. We then take $W = \text{Dom}\left(R_{I_d}\right) - R_{I_d}^{-1}(X)$, that is, we only keep vertices in "slice" I_d that can reach every vertex in slice I_d+1. Application of Corollary 1 with $V_1 = W$, $V_2 = R_{I_d}(W)$ and $\{L_j\}_j = R_{I_d}$ gives $Q \geq |W| - S$.

1 **function** *sub_paramQ_bywavefront*
  input : DFG $G = (\mathcal{S}, \mathcal{D})$, vertex $S \in \mathcal{S}$, parametrized dimensions Ω_d
  output: lower bound Q
2 $D := \dim(S)$;
3 $R_{S\to S} := \emptyset$; $RL_{S\to S} := \emptyset$; $R_{S\to *} := \emptyset$;
4 **foreach** $S_j \in \mathcal{S}$ in topological order, from S (excluded) to S (included) **do**
5 $A := \{(S_i, S_j) \in E,\ R_{S_i\to S_j}$ is affine and $R^{-1}_{S_i\to S_j}$ injective$\}$;
6 $RL_{S\to S_j} := \bigcup_{(S_i,S_j)\in A} RL_{S\to S_i} \circ R_{S_i\to S_j}$;
7 $R_{S\to S_j} := \bigcup_{(S_i,S_j)\in E} R_{S\to S_i} \circ R_{S_i\to S_j}$;
8 $R_{S\to *} := R_{S\to *} \cup R_{S\to S_j}$;
9 $R_{I_d} := RL_{S\to S} \cap \{S[I_1 \dots I_d, i_{d+1} \dots i_D] \to S[I_1 \dots I_d + 1, i_{d+1} \dots i_D]\}$;
10 $R_{\text{complete}} := \{S[I_1 \dots I_d, i_{d+1} \dots i_D] \to S[I_1 \dots I_d + 1, i'_{d+1} \dots i'_D]\}$;
11 $W := \text{Dom}\left(R_{I_d}\right) - R_{I_d}^{-1}\left(\left(R_{\text{complete}} - (R_{S\to S})^*\right)(\text{Dom}\left(R_{I_d}\right))\right)$;
12 $Q := \max(|W| - S, 0)$;
13 Q.may-spill $:= W \cup \left(R_{S\to *}(W) \cap R^{-1}_{S\to *}\left(R_{I_d}(W)\right)\right)$

Algorithm 5: Derivation of a lower bound with the wavefront method

3.5 Complete framework

3.5.1 DFG construction

Our front end (PET [56]) takes as input a program in C where the to-be analyzed regions (SCoPs – Static Control Parts) are delimited by `#pragma scop` and `#pragma endscop` annotations. For PET, all array accesses are supposed not to alias with one another. Any scalar data is assumed to be atomic and all of the same size: our CDAG is not weighted (which is a limitation of our implementation and not a conceptual limitation of the approach). As illustrated by the example of Fig. 3.1 and 3.2 (multidimensional-)array accesses are affine expressions of static parameters and loop indices. A static parameter can be the result of any complex calculation but has to be a fixed value for the entire execution of the region. Loop bounds and more generally control tests follow the same rules (affine expressions). As a consequence, the iteration space is a union of (parametric) polyhedra, and memory accesses (read and writes) are

piecewise affine functions. This representation of the region execution that fits into the polyhedral framework [25] allows IOLB to compute data dependencies using standard data-flow analyses.

PET outputs a polyhedral representation of the input C program, from which we extract a *Data-flow graph (DFG)* $G = (\mathcal{S}, \mathcal{D})$ (see Sec. 3.1.2).

3.5.2 Instances of parameter values

As briefly explained in Sec. 3.2, to generate bounds that are as tight as possible, our heuristic needs to take decisions. Such decisions are based on our ability to compare the size of two different domains sizes or even the complexity of two different sub-CDAGs. The overall framework being parametric (it provides complexities that are functions of parameter values and cache size), a total order is obtained by considering a specific instance of parameter values, taken as an additional input alongside the C program. One needs to outline that a specific instance of parameter values is *not* considered by the algorithm as a precondition: For a given instance, the computed lower bound expression is universal, i.e., is correct for *any* parameter values. For completeness, several instances are considered, and to each instance I is associated a complexity Q^I. As we have $Q \geq Q^I$ for any instance, denoting $\mathcal{I}$ the set of all considered instances, they are simply combined as:

$$Q = \max_{I \in \mathcal{I}} \left(Q^I \right).$$

3.5.3 Main algorithm

Alg. 6 contains the skeleton of the main part of IOLB, with links to corresponding subsections.

To make it concrete, we show a step-by-step execution of the algorithm on the `cholesky` kernel. The pseudo-code and associated DFG for `cholesky` are reported in Fig. 3.9.

In this example, the K-partition method is the method that yields the strongest bound. To keep things tractable, we will detail only the parts of the algorithm that contribute to this bound: the iteration of the outer loops (lines 4 and 5) for which $d = 0$ and $S = S_3$, and only the K-partition part (lines 8 to 18, corresponding to Sec. 3.3).

The DFG contains three statement vertices $\{S_1, S_2, S_3\}$ (the vertex corresponding to input array A and the corresponding dependences are omitted as they do not play a role in the lower bound derivation). The main loop of Alg. 6 iterates on those statements and computes some lower bound complexities for each of them.

Procedure genpaths (called at line 11 in Alg. 6) traverses the DFG, searching for chain or broadcast paths ending in S (cf. Sec. 3.1.2). Here, it finds three "interesting paths" for statement S_3. These are the three paths pointing to S_3, namely: $P_1 = (e_1)$ (chain path), $P_2 = (e_2)$ and $P_3 = (e_3)$ (broadcast paths). Their relations are: $R_{P_1} = R_{e_1}$, $R_{P_2} = R_{e_2}$, $R_{P_3} = R_{e_3}$ (cf. Fig. 3.9c).

The corresponding projections and kernels are:

$$\begin{aligned}
\phi_1(k,i,j) &= \mathrm{proj}_{(1,0,0)}(k,i,j) = (0,i,j) & K_1 &= \mathrm{Ker}(\phi_1) = \langle (1,0,0) \rangle \\
\phi_2(k,i,j) &= (k,j) & K_2 &= \mathrm{Ker}(\phi_2) = \langle (0,1,0) \rangle \\
\phi_3(k,i,j) &= (k,i) & K_3 &= \mathrm{Ker}(\phi_3) = \langle (0,0,1) \rangle
\end{aligned}$$

```
function program_Q
    input  : Data-flow graph G = (𝒮, 𝒟), an instance I
    output: lower bound Q_low
    𝒬 = ∅;
    Let D be the max loop depth;
    foreach loop level 0 ≤ d < D do                                  → Sec. 3.2.3
        foreach S ∈ 𝒮 surrounded by at least d + 1 loops do
            Ω_d := [I_1, ..., I_d] → {S[i_1, ..., i_D] : i_1 = I_1 ∧ ... i_d = I_d};
            Let G' be a copy of G;
            while elapsedTime < timeout do
                Let D_S be the parametrized domain of S in G';
                𝒫 := ∅, ℒ := ∅;
                foreach P_i ∈ genpaths(G', S, Ω_d) do
                    if |D_S ∩ Im(P_i)| ≥ γ · |D_S| then
                        K_i := Ker(P_i);
                        if ℒ := subspace_closure(ℬ, K_i) changed then
                            D_S := D_S ∩ Im(P_i);
                            𝒫 := 𝒫 ∪ P_i;
                                                                     Sec. 3.3
                if 𝒫 = ∅ then exit while loop;
                (𝒬, G') = combine_paramQ(𝒬, G',
                    sub_paramQ_bypartition(𝒫, D_S, ℒ, Ω_d));
            (𝒬, G') = combine_paramQ(𝒬, G',        sub_paramQ_bywavefront(S, Ω_d));   Sec. 3.4
    Q_low = input_size(G) + max(0, combine_subQ(𝒬));   Sec. 3.2.2
function combine_paramQ
    input  : set of global bounds 𝒬, DFG G', parametrized bound Q(Ω)
    output: updated 𝒬, G'
    if [Ω ≠ Ω' ⇒ Q.interf(Ω) ∩ Q.interf(Ω') = ∅] then
        Q := ∑_Ω Q(Ω);
        Q.may-spill := ⋃_Ω Q.may-spill(Ω);
        𝒬 = 𝒬 ∪ {Q};
        G' := G' \ Q.may-spill;
```

Algorithm 6: Main procedure that computes Q_{low} for the program by combining lower bound of sub-CDAGs obtained through K-partition or wavefront reasoning

```
for(k = 0; k < n; k++)
   A[k][k] = sqrt(A[k][k]);                    //S1
   for(i = k+1; i < n; i++)
      A[i][k] /= A[k][k];                      //S2
   for(i = k+1; i < n; i++)
      for(j = k+1; j <= i; j++)
         A[i][j] -= A[i][k] * A[j][k];         //S3
```

(a) Source code

(b) DFG (input nodes are omitted)

$$
\begin{aligned}
R_{e_1} &= \{S_3[k-1,i,j] \rightarrow S_3[k,i,j] : \ 1 \le k < N \ \wedge \ k+1 \le i < N \ \wedge \ k+1 \le j \le i\} \\
R_{e_2} &= \{S_2[k,j] \rightarrow S_3[k,i,j] : \ 0 \le k < N \ \wedge \ k+1 \le i < N \ \wedge \ k+1 \le j \le i\} \\
R_{e_3} &= \{S_2[k,i] \rightarrow S_3[k,i,j] : \ 0 \le k < N \ \wedge \ k+1 \le i < N \ \wedge \ k+1 \le j \le i\} \\
R_{e_4} &= \{S_3[k-1,i,k] \rightarrow S_2[k,i] : \ 1 \le k < N \ \wedge \ k+1 \le i < N\} \\
R_{e_5} &= \{S_1[k] \rightarrow S_2[k,i] : \ 0 \le k < N \ \wedge \ k+1 \le i < N\} \\
R_{e_6} &= \{S_1[k-1,k,k] \rightarrow S_1[k] : \ 1 \le k < N \ \wedge \ k+1 \le i < N\}
\end{aligned}
$$

(c) Edge relations

Figure 3.9 – Cholesky decomposition

The domain D_S is initialized at line 9 to

$$D_S := D_{S_3} = \{S_3[k,i,j] : \ 0 \le k < N \ \wedge \ k+1 \le i < N \ \wedge \ k+1 \le j \le i\}.$$

The **foreach** loop then iterates over $\{P_1, P_2, P_3\}$. Here the three paths have domains that almost span the entire domain of S_3, so the condition at line 12 is always true (γ is a constant between 0 and 1). They also have pairwise orthogonal kernels, so the condition at line 14 is also true at each iteration, and at the end of the **foreach** loop:

$$
\begin{aligned}
D_S &= ((D_{S_3} \cap \mathrm{Im}(P_1)) \cap \mathrm{Im}(P_2)) \cap \mathrm{Im}(P_3) \\
&= \{S_3[k,i,j] : \ 1 \le k < N \ \wedge \ k+1 \le i < N \ \wedge \ k+1 \le j \le i\} \\
\mathcal{P} &= \{P_1, P_2, P_3\}
\end{aligned}
$$

At line 18, function sub_paramQ_bypartition derives a lower bound from the set of paths $\mathcal{P}$, which is then added to the set of lower bounds $\mathcal{Q}$ by function combine_paramQ.

We can check that P_1 is independent from P_2 and P_3, but P_2 and P_3 are not (that is $R_{P_1}^{-1}(D) \cap R_{P_2}^{-1}(D) = \emptyset$, $R_{P_1}^{-1}(D) \cap R_{P_3}^{-1}(D) = \emptyset$ and $R_{P_2}^{-1}(D) \cap R_{P_3}^{-1}(D) \neq \emptyset$).

Thus the following inequality holds for any K-bounded set E in the CDAG:

$$\left|\phi_1(E)\right| + \frac{1}{2}\left|\phi_2(E)\right| + \frac{1}{2}\left|\phi_3(E)\right| \leq K. \tag{3.7}$$

Let s_1, s_2, s_3 be the solutions to the following optimization problem:

$$\text{Minimize } \sigma := \sum_j s_j \qquad \text{s.t.} \qquad \begin{array}{l} 0 \leq s_1, s_2, s_3 \leq 1 \\ 1 \leq s_2 + s_3 \\ 1 \leq s_1 + s_3 \\ 1 \leq s_1 + s_2 \end{array} \tag{3.8}$$

The discrete Brascamp-Lieb theorem [16], combined with Lemma 5 applied on projections ϕ_i, guarantee that, for any K-bounded set E:

$$|E| \leq K^{\sigma} \left(\frac{2s_1}{\sigma}\right)^{s_1} \left(\frac{2s_2}{\sigma}\right)^{s_2} \left(\frac{s_3}{\sigma}\right)^{s_3}. \tag{3.9}$$

The solution to (3.8) is $s_1 = s_2 = s_3 = \frac{1}{2}$, so

$$|E| \leq 2 \cdot (K/3)^{3/2}.$$

Lemma 2 tells us that, if U is an upper bound on the size of a $(S+T)$-bounded-set in G, then:

$$Q(G) \geq T \cdot \left\lfloor \frac{|V \setminus \text{Sources}(V)|}{U} \right\rfloor - |\text{Sources}(V)|.$$

Here $V = D \cup R_{P_1}^{-1}(D) \cup R_{P_2}^{-1}(D) \cup R_{P_3}^{-1}(D)$, giving:

$$\begin{aligned} V \setminus \text{Sources}(V) = \{S_3[k,i,j] :\ & 1 \leq k < N \\ & \wedge\ k+1 \leq i < N \ \wedge\ k+1 \leq j \leq i\} \\ \text{Sources}(V) = \{S_3[0,i,j] : 1 \leq i < N\ & \wedge\ 1 \leq j \leq i; \\ S_2[k,i] :\ & 1 \leq k < N \ \wedge\ k+1 \leq i < N\} \end{aligned}$$

So $|V \setminus \text{Sources}(V)| = \frac{N^3}{6}$ and $|\text{Sources}(V)| = N^2$.[2] Taking for U our upper bound on $|E|$ provides the following inequality for which the objective is to set a value for T that maximizes its right hand side:

$$Q \geq T \times \left\lfloor \frac{N^3/6}{2 \cdot (K/3)^{3/2}} \right\rfloor - N^2 \approx \frac{T}{(S+T)^{3/2}} \times \frac{N^3/6}{2 \cdot (1/3)^{3/2}}.$$

Setting $T = 2S$ (so $K = S + T = 3S$) leads to the following lower bound on Q:

$$Q_{\text{low}}^{\infty} = (2S) \times \frac{N^3/6}{2S^{3/2}} = \frac{N^3}{6\sqrt{S}}.$$

2. From here on, we omit lower-order additive terms to keep things readable. The full formula output by IOLB can be found in Sec 3.6.4.

Concerning the parts of the algorithm that were not detailed here: the outermost loop (Line 4) corresponds to the loop parametrization detailed in Sec. 3.2.3: for each loop depth d, outermost indices are fixed (as parameter Ω_d – Line 6), and parametrically computed lower bounds are summed (when not interfering – Line 22) over all iterations (Line 23 in combine_paramQ). The loop on statements S (Line 5) decomposes the full CDAG into as many "S-centric" sub-CDAGs. The so-obtained bounded set of lower bounds $\mathcal{Q}$ are combined using procedure combine_subQ (Line 20) as described in Sec. 3.2.2. To take compulsory misses into account, the size of the input data of the program is added to the expression.

For each statement S, both techniques (K-partition and wavefront, resp. Line 18 and Line 19) generate lower bounds. As opposed to the implicitly considered "S-centric" sub-CDAGs for the wavefront reasoning, an "S-centric" sub-CDAG for the K-partition reasoning (which is built by finding a set $\mathcal{P}$ of DFG-paths that terminate at S – Lines 10-17 through function genpaths) does not necessarily span all the S-vertices (D_S) of the CDAG. So several (non-intersecting) sub-CDAGs can be built until no more interesting lower bound can be derived (Line 17).

3.6 Experimental evaluation

3.6.1 Implementation

IOLB was implemented in C, using ISL-0.13 [53], barvinok-0.37 [57] and PET-0.05 [56]. We also used GiNaC-1.7.4 [9] for the manipulation of symbolic expressions, and PIP-1.4.0 [24] for linear programs. IOLB takes as input an affine C program and outputs a symbolic expression for a lower bound on *I/O* complexity as a function of the problem size parameters of the program and capacity of fast memory. Figure 3.10 shows an example of annotated C code (jacobi-1d from PolyBench), with the output from IOLB.

3.6.2 Evaluation on PolyBench

IOLB was applied to all programs in the PolyBench/C-4.2.1 benchmark suite [39]. For each kernel, our tool outputs an *I/O* lower bound expression Q_{low}, from which we derive an upper bound on operational intensity OI_{up} by forming the ratio of the number of operations and Q_{low}. To evaluate the quality of the results produced by IOLB, we manually generate tiled versions of each kernel, then manually compute parametric data-movement costs as a function of tile sizes and cache size, then manually find the optimal tile sizes and thereby, finally, derive a manually optimized data-movement cost for this kernel. By forming the ratio of the total number of operations and the data-movement cost, we then generate OI_{manual}. In this derivation, we assume that we have explicit control of the cache. Then OI_{manual} is compared with an operational intensity upper-bound obtained by forming the ratio of the number of operations and the data movement lower bound generated by IOLB: OI_{up}.

IOLB runs in less than one second on each of these kernels on a standard computer.

Let us use `jacobi-1d` as an example to illustrate this. IOLB computes a lower bound expression Q_{low} on the number of loads needed for any schedule of the `jacobi-1d` kernel:

$$Q_{\text{low}} = 2 + N + \max\left(0, \frac{TN}{4S} - N - T - \frac{1}{4}\frac{N}{S} - \frac{3}{4}\frac{T}{S} - S + 5\right).$$

```c
void kernel_jacobi_1d(int tsteps, int n, double A[n], double B[n]) {
   int t, i;
#pragma scop
   for (t = 0; t < tsteps; t++) {
      for (i = 1; i < n - 1; i++)
         B[i] = 0.33333 * (A[i-1] + A[i] + A[i+1]);
      for (i = 1; i < n - 1; i++)
         A[i] = 0.33333 * (B[i-1] + B[i] + B[i+1]);
        }
#pragma endscop
}
```

Inputs:

n

Full expression:

$$n + \max\left(0, -S - n - tsteps + 5 + \frac{n \cdot tsteps}{4 \cdot S} - \frac{n}{4 \cdot S} - \frac{3 \cdot tsteps}{4 \cdot S} + \frac{3}{4 \cdot S}\right) + 2$$

Asymptotic expression:

$$\frac{n \cdot tsteps}{4 \cdot S}$$

Simplified expression:

$$\frac{n^2}{4 \cdot S}$$

Figure 3.10 – Input file and output from IOLB

The first term is the input data size, and the second term is obtained by the partitioning technique. Since the expression of Q_{low} can be quite large, we automatically simplify to Q_{low}^{∞} by only retaining the asymptotically dominant terms, assuming all parameters $N, M \ldots$ and cache size S tend to infinity, and $S = o(N, M, \ldots)$. Finally, from Q_{low}^{∞} and the fact that the `jacobi-1d` kernel performs $6TN$ operations, we compute an upper bound for the OI of any schedule of the `jacobi-1d` kernel:

$$Q_{\text{low}}^{\infty} = \frac{TN}{4S} \qquad OI_{\text{up}} = \frac{6TN}{Q_{\text{low}}^{\infty}} = 24S$$

3.6.3 Parametric bounds for *OI*

Table 3.1 reports, for each kernel in PolyBench:

— the size of the input data and the number of operations;

— the simplified *I/O* lower bound Q_{low}^{∞} from IOLB;

— the parametric lower and upper bounds on operational intensity OI_{manual} and $OI_{\text{up}} = \frac{\#\text{ ops}}{Q_{\text{low}}^{\infty}}$, and their ratio;

— the best known published OI_{up}, when it exists.

The 30 reported benchmarks can be divided into four categories, corresponding to table divisions:

1. (19 kernels) The ratio $\frac{\#\text{ ops}}{\#\text{ input data}}$ is high, and we manually find that high OI is achievable through tiling. IOLB gives a non-trivial OI upper bound that is within a constant of the manually obtained OI lower bound OI_{manual}. The bound is asymptotically tight for 9 of them, and within a factor of 2 for an additional 5. Except for matrix multiplication (`gemm`), where it matches the best published bound, these are all improvements over previously published results.
2. (7 kernels) The ratio $\frac{\#\text{ ops}}{\#\text{ input data}}$ is a constant: clearly, these cases do not provide enough operations to enable data reuse. The reported lower bound by IOLB is # input data, which is asymptotically tight for 5 of them, and within a factor of 2 for 1 more.
3. (2 kernels) The ratio $\frac{\#\text{ ops}}{\#\text{ input data}}$ is high which could indicate potential for tiling and high OI. However our best manual schedule leads to a constant OI which is arbitrarily far from this optimistic ratio. IOLB proves that the code is not tileable, so the best achievable OI is a constant. IOLB finds this upper bound on OI thanks to the wavefront technique. This is better by at least a factor of $\sqrt{S}$ than any bound that could be obtained by geometric reasoning.
4. (2 kernels) There is an arbitrarily large discrepancy between OI_{up} and OI_{manual}. Visual examination shows that, for these cases, IOLB is too optimistic. These codes are actually not tileable in all dimensions, and we believe that it is possible, using more advanced techniques that are currently out of the scope of IOLB, to prove a matching OI upper bound.

3.6.4 Complete lower bound formulae

Table 3.2 shows the complete formulae produced by IOLB. Next to the complete formulae are asymptotic formulae as presented in Table 3.1.

kernel	# input data	# ops	Q^{∞}_{low}	$\boldsymbol{OI_{\text{manual}} \leq OI \leq OI_{\text{up}}}$	ratio	Published OI_{up}
2mm	$N_iN_k + N_kN_j$ $+N_jN_l + N_iN_l$	$2(NiN_jN_k$ $+N_iN_jN_l)$	$2(N_iN_jN_k$ $+N_iN_jN_l)/\sqrt{S}$	$\sqrt{S} \leq OI \leq \sqrt{S}$	1	–
3mm	$N_iN_k + N_kN_j$ $+N_jN_m + N_mN_l$	$2(NiN_jN_k + N_jN_lN_m$ $+N_iN_jN_l)$	$2(N_iN_jN_k + N_iN_jN_l$ $+N_jN_lN_m)/\sqrt{S}$	$\sqrt{S} \leq OI \leq \sqrt{S}$	1	–
cholesky	$\frac{1}{2}N^2$	$\frac{1}{3}N^3$	$\frac{1}{6}N^3/\sqrt{S}$	$\sqrt{S} \leq OI \leq 2\sqrt{S}$	2	$8\sqrt{S}$ [2]
correlation	MN	M^2N	$\frac{1}{2}M^2N/\sqrt{S}$	$\sqrt{S} \leq OI \leq 2\sqrt{S}$	2	–
covariance	MN	M^2N	$\frac{1}{2}M^2N/\sqrt{S}$	$\sqrt{S} \leq OI \leq 2\sqrt{S}$	2	–
doitgen	$N_pN_qN_r + N_p^2$	$2N_p^2N_qN_r$	$2N_p^2N_qN_r/\sqrt{S}$	$\sqrt{S} \leq OI \leq \sqrt{S}$	1	–
fdtd-2d	$3N_xN_y + T$	$11N_xN_yT$	$\frac{2}{3\sqrt{3}}N_xN_yT/\sqrt{S}$	$\frac{11}{24}\sqrt{3}\sqrt{S} \leq OI \leq \frac{33}{2}\sqrt{3}\sqrt{S}$	36	–
floyd-warshall	N^2	$2N^3$	$2N^3/\sqrt{S}$	$\sqrt{S} \leq OI \leq \sqrt{S}$	1	$8\sqrt{S}$ [2]
gemm	$N_iN_j + N_jN_k + N_iN_k$	$2N_iN_jN_k$	$2N_iN_jN_k/\sqrt{S}$	$\sqrt{S} \leq OI \leq \sqrt{S}$	1	$\sqrt{S}$ [48]
heat-3d	N^3	$30N^3T$	$\frac{3}{8}\sqrt[3]{2}N^3T/\sqrt[3]{S}$	$\frac{5}{2}\sqrt[3]{S} \leq OI \leq 40 \cdot 2^{2/3}\sqrt[3]{S}$	$16 \cdot 2^{2/3}$	–
jacobi-1d	N	$6NT$	$\frac{1}{4}NT/S$	$\frac{3}{2}S \leq OI \leq 24S$	16	$48S$ [22]
jacobi-2d	N^2	$10N^2T$	$\frac{2}{3\sqrt{3}}N^2T/\sqrt{S}$	$\frac{5}{4}\sqrt{S} \leq OI \leq 15\sqrt{3}\sqrt{S}$	$12\sqrt{3}$	$40\sqrt{2}\sqrt{S}$ [22]
lu	N^2	$\frac{2}{3}N^3$	$\frac{2}{3}N^3/\sqrt{S}$	$\sqrt{S} \leq OI \leq \sqrt{S}$	1	$8\sqrt{S}$ [2]
ludcmp	N^2	$\frac{2}{3}N^3$	$\frac{2}{3}N^3/\sqrt{S}$	$\sqrt{S} \leq OI \leq \sqrt{S}$	1	$8\sqrt{S}$ [2]
seidel-2d	N^2	$9N^2T$	$\frac{2}{3\sqrt{3}}N^2T/\sqrt{S}$	$\frac{9}{4}\sqrt{S} \leq OI \leq \frac{27\sqrt{3}}{2}\sqrt{S}$	$6\sqrt{3}$	–
symm	$\frac{1}{2}M^2 + 2MN$	$2M^2N$	$2M^2N/\sqrt{S}$	$\sqrt{S} \leq OI \leq \sqrt{S}$	1	$8\sqrt{S}$ [2]
syr2k	$\frac{1}{2}N^2 + 2MN$	$2MN^2$	$MN^2/\sqrt{S}$	$\sqrt{S} \leq OI \leq 2\sqrt{S}$	2	$8\sqrt{S}$ [2]
syrk	$\frac{1}{2}N^2 + MN$	MN^2	$\frac{1}{2}MN^2/\sqrt{S}$	$\sqrt{S} \leq OI \leq 2\sqrt{S}$	2	$8\sqrt{S}$ [2]
trmm	$\frac{1}{2}M^2 + MN$	M^2N	$M^2N/\sqrt{S}$	$\sqrt{S} \leq OI \leq \sqrt{S}$	1	$8\sqrt{S}$ [2]
atax	MN	$4MN$	MN	$4 \leq OI \leq 4$	1	–
bicg	MN	$4MN$	MN	$4 \leq OI \leq 4$	1	–
deriche	HW	$32HW$	HW	$\frac{16}{3} \leq OI \leq 32$	6	–
gemver	N^2	$10N^2$	N^2	$5 \leq OI \leq 10$	2	–
gesummv	$2N^2$	$4N^2$	$2N^2$	$2 \leq OI \leq 2$	1	–
mvt	N^2	$4N^2$	N^2	$4 \leq OI \leq 4$	1	–
trisolv	$\frac{1}{2}N^2$	N^2	$\frac{1}{2}N^2$	$2 \leq OI \leq 2$	1	–
adi	N^2	$30N^2T$	N^2T	$5 \leq OI \leq 30$	6	–
durbin	N	$2N^2$	$\frac{1}{2}N^2$	$\frac{2}{3} \leq OI \leq 4$	6	–
gramschmidt	MN	$2MN^2$	$MN^2/\sqrt{S}$	$1 \leq OI \leq 2\sqrt{S}$	$2\sqrt{S}$	–
nussinov	$\frac{1}{2}N^2$	$\frac{1}{3}N^3$	$\frac{1}{6}N^3/\sqrt{S}$	$1 \leq OI \leq 2\sqrt{S}$	$2\sqrt{S}$	–

Table 3.1 – Results on PolyBench benchmarks

We present the complete formulae produced by IOLB for a few reasons. While the lower bounds on *I/O* obtained by IOLB are lower bounds for any values of the parameters (M, N, S, etc), the asymptotic formulae have to be used with care. Indeed, the asymptotic reasoning, while providing simpler and easier to understand formulae, unfortunately removes the lower bound property. (Negligible negative terms are removed during the asymptotic reasoning.) Furthermore, if the asymptotic assumptions are violated, then the asymptotic formula becomes really off. For example if S is not negligible with respect to M and N, or if one dimension in `gemm` is small. Also, the asymptotic reasoning entails some assumptions. We chose to assume all parameters $N, M \ldots$ and cache size S tend to infinity, and $S = o(N, M, \ldots)$). We can imagine other reasonable asymptotic reasoning. With the complete formula, it is possible to derive them at will. Finally, it can be instructive to understand the form of the complete formulae returned by IOLB, and showing the complete next to the asymptotic expansion explains our asymptotic assumption.

Tensor contractions and convolutions This table also shows parametric lower bounds for tensor contraction and 2D convolution kernels, as an application of the extensions presented in 3.3.4.

Because the lower bound derived is still sound, even if the small dimension hypothesis is not satisfied, we have to combine various lower bounds for different small dimension scenarios together.

For convolution kernels, we considered 5 small dimension scenarios, based on array sizes: (i) no small parameters, (ii) H and W small parameters, (iii) H, W and B small parameters, (iv) H, W, X, Y and B small parameters, and (v) C, H, W and B small parameters. Among those bounds, only the first three scenarios lead to interesting bounds. They correspond to the last three rows in the expression. The previous version of the algorithm (without reduction management or small dimensions) fails to find an interesting bound, and returns the sum of array sizes (first row in the expression).

3.7 Related work

The seminal work of Hong & Kung [29] was the first to present an approach to derive lower bounds on data movement for any valid execution schedule of operations in a computational DAG. Their work modeled data movement in a two-level memory hierarchy and presented manually derived decomposability factors (asymptotic order complexity, without scaling constants) for a few algorithms like matrix multiplication and FFT. Several efforts have sought to build on the fundamental lower bounding approach devised by Hong & Kung, usually targeting one of two objectives: i) generalizing the cost model to more realistic architecture hierarchies [46, 10, 11], or ii) providing an *I/O* complexity with (tight) constant for some specific class of algorithms (sorting/FFT [1, 42], relaxation [43], or linear algebra [30, 4, 3, 19]).

In the context of linear algebra, Irony et al. [30] were the first to use the Loomis-Whitney inequality [34] to find a lower bound on data movement. This was in the context of `gemm` (one of the kernels of PolyBench). The asymptotic upper bound on *OI* from this paper is $4\sqrt{2}\sqrt{S}$. IOLB returns $\sqrt{S}$. This result was then extended in [2] to 7 more kernels of PolyBench: `cholesky`, `floyd-warshall`, `lu`, `symm`, `syr2k`, `syrk`, and `trmm`, where their upper bound on *OI* is $8\sqrt{S}$ for all of them. IOLB returns $\sqrt{S}$ for 4 of these kernels, and $2\sqrt{S}$ for the other 3. The

kernel	Complete lower bound	Asymptotic simplified formula
2mm	$\max\Big(N_iN_j + N_jN_k + N_iN_k + N_jN_l + 2,$ $\Big(\frac{2}{\sqrt{S}}N_iN_j(N_k-1) + 2N_i + 2N_j + N_k - 4\sqrt{2}S\Big)$ $+\Big(\frac{2}{\sqrt{S}}N_iN_l(N_j-1) + 2N_i + 2N_l + N_j - 4\sqrt{2}S\Big)$ $-2N_iN_j - 2\Big)$	$\frac{2}{\sqrt{S}}N_iN_jN_k$ $+\frac{2}{\sqrt{S}}N_iN_lN_j$
3mm	$\max\Big(N_iN_k + N_jN_k + N_jN_m + N_lN_m,$ $\Big(\frac{2}{\sqrt{S}}N_iN_j(N_k-1) + 2N_i + 2N_j + N_k - 4\sqrt{2}S\Big)$ $+\Big(\frac{2}{\sqrt{S}}N_iN_l(N_j-1) + 2N_i + 2N_l + N_j - 4\sqrt{2}S\Big)$ $+\Big(\frac{2}{\sqrt{S}}N_jN_l(N_m-1) + 2N_j + 2N_l + N_m - 4\sqrt{2}S\Big)$ $-2N_jN_i - 2N_jN_l - N_iN_l - 6\Big)$	$\frac{2}{\sqrt{S}}N_iN_jN_k$ $+\frac{2}{\sqrt{S}}N_iN_lN_j$ $+\frac{2}{\sqrt{S}}N_jN_lN_m$
adi	$4N^2 + \max\Big(0, (N^2 - 4N - S + 5)(T-2)\Big)$	N^2T
atax	$MN + N + \max\Big(0, \frac{1}{8}\frac{1}{S}((2M-1-8S)(2N-1-8S)-1) - 10S + 2\Big)$	MN
bicg	$MN + M + N + \max\Big(0, \frac{1}{8}\frac{1}{S}((2M-1-8S)(2N-1-8S)-1) - 10S + 2\Big)$	MN
cholesky	$\max\Big(\frac{1}{2}N(N+1), \frac{1}{6}\frac{1}{\sqrt{S}}(N-1)(N-2)(N-3) + \frac{1}{2\sqrt{2}}\frac{1}{S}(N-1)(N-2)$ $-(N-2)(N-7) - 4\sqrt{2}S\Big)$	$\frac{1}{6}\frac{1}{\sqrt{S}}N^3$
correlation	$\max\Big(MN + 2, \frac{1}{2}\frac{1}{\sqrt{S}}M(M-1)(N-1+\frac{\sqrt{2}}{2}\frac{1}{\sqrt{S}}) - \frac{1}{2}(M-3)(M+2N-2) + 2 - 4S\sqrt{2}\Big)$	$\frac{1}{2}\frac{1}{\sqrt{S}}M^2N$

covariance	$\max\left(MN+2, \frac{1}{2}\frac{1}{\sqrt{S}}M(M-1)(N-1+\frac{\sqrt{2}}{2}\frac{1}{\sqrt{S}})-\frac{1}{2}(M-3)(M+2N-2)+1-4S\sqrt{2}\right)$	$\frac{1}{2}\frac{1}{\sqrt{S}}M^2N$
deriche	$HW+1$	HW
doitgen	$\max\left(N_p^2+N_pN_qN_r, \frac{2}{\sqrt{S}}N_qN_rN_p(N_p-1+\frac{1}{\sqrt{2}}\frac{1}{\sqrt{S}})-N_qN_r(N_p-1)+2N_p-8\sqrt{2}S-1\right)$	$2\frac{1}{\sqrt{S}}N_qN_rN_p^2$
durbin	$2N+\max\left(0, \frac{1}{2}(N-3)(N-2-2S)\right)$	$\frac{1}{2}N^2$
fdtd-2d	$\max\left(3N_xN_y-N_y+T-1, \frac{1}{2\sqrt{2}}\frac{1}{\sqrt{S}}(N_x-2)(N_y-2)(T-1)+2(N_x+2)(N_y+2)\right.$ $\left.-T(N_x+N_y-6)-N_y-S-23\right)$	$\frac{1}{2\sqrt{2}}\frac{1}{\sqrt{S}}N_xN_yT$
floyd-warshall	$\max\left(N^2, \frac{1}{\sqrt{S}}(N-1)^3-(6N-19)(N-2)-8\sqrt{2}S\right)$	$\frac{1}{\sqrt{S}}N^3$
gemm	$\max\left(N_iN_j+N_jN_k+N_iN_k+2, \frac{2}{\sqrt{S}}N_iN_j(N_k-1)+2N_i+2N_j+N_k-4\sqrt{2}S\right)$	$2\frac{1}{\sqrt{S}}N_iN_jN_k$
gemver	$N^2+8N+2+\max\left(0, \frac{1}{4}\frac{1}{S}(3N-2)(N-8S)-3S+1\right)$	N^2
gesummv	$2N^2+N+2+\max\left(0, \frac{1}{2}\frac{1}{S}(N-1)(N-8S)-2S\right)$	$2N^2$
gramschmidt	$\max\left(MN, \frac{1}{\sqrt{S}}MN(N-3)-M(N-5-\frac{2}{\sqrt{S}})-\frac{1}{2}(N-1)(N-6)-4\sqrt{2}S-3\right)$	$\frac{1}{\sqrt{S}}MN^2$
heat-3d	$\max\left((N-10)(N+2)^2, \frac{9\sqrt[3]{3}}{16}\frac{1}{\sqrt[3]{S}}(T-1)(N-3)^3-3(T-7)(N-3)(N-4)\right.$ $\left.+42N-T-\frac{9\sqrt[3]{3}}{4\sqrt[3]{4}}S-111\right)$	$\frac{9\sqrt[3]{3}}{16}\frac{1}{\sqrt[3]{S}}N^3T$
jacobi-1d	$\max\left(2+N, \frac{1}{4}\frac{1}{S}(T-1)(N-3)-T-S+7\right)$	$\frac{1}{4}\frac{1}{S}NT$
jacobi-2d	$\max\left((N-2)(N+6), \frac{2}{3\sqrt{3}}\frac{1}{\sqrt{S}}(N-3)^2(T-1)-\frac{4\sqrt{2}}{3\sqrt{3}}S-(T-7)(2N-7)+14\right)$	$\frac{2}{3\sqrt{3}}\frac{1}{\sqrt{S}}N^2T$
lu	$\max\left(N^2, \frac{2}{3}\frac{1}{\sqrt{S}}(N-2)(N^2-4N+6)-2(N^2-10N+18)-8\sqrt{2}S\right)$	$\frac{2}{3}\frac{1}{\sqrt{S}}N^3$
ludcmp	$\max\left(N^2+N, \frac{1}{3}\frac{1}{\sqrt{S}}(2N-3)(N-1)(N-2)\sqrt{2}\frac{1}{S}(N-1)(N-2)-(2N^2-15N+19)-16\sqrt{2}S\right)$	$\frac{2}{3}\frac{1}{\sqrt{S}}N^3$

mvt	$N^2 + 4N + \max\left(0, \frac{1}{6}\frac{1}{S}N(N-1) - 2S - 4N + 4\right)$	N^2
nussinov	$\frac{1}{2}N^2 + \frac{5}{2}N - 1 + \max\left(0, \frac{1}{6}\frac{1}{\sqrt{S}}(N-3)(N-4)(N-5) + \frac{1}{4}\frac{1}{S}\sqrt{2}(3N^2 - 19N + 6)\right.$ $\left.-(N^2 - 13N + 22) - 8\sqrt{2}S\right)$	$\frac{1}{6}\frac{1}{\sqrt{S}}N^3$
seidel-2d	$\max\left(N^2, \frac{2}{3\sqrt{3}}\frac{1}{\sqrt{S}}(N-3)^2(T-1) - (2N-7)(T-5) - \frac{4\sqrt{2}}{3\sqrt{3}}S + 12\right)$	$\frac{2}{3\sqrt{3}}\frac{1}{\sqrt{S}}N^2T$
symm	$\max\left(\frac{1}{2}M(M+1) + 2MN + 2, 2\frac{1}{\sqrt{S}}(M-1)(M-2)N - \frac{1}{2}((4N+M)(M-5))\right.$ $\left.+5(M-2) - 8\sqrt{2}S\right)$	$2\frac{1}{\sqrt{S}}M^2N$
syr2k	$\max\left(2 + 2MN + \frac{1}{2}N(N+1), \frac{1}{\sqrt{S}}(M-1)(N+1)N + M + 4N - 4\sqrt{2}S\right)$	$\frac{1}{\sqrt{S}}MN^2$
syrk	$\max\left(MN + \frac{1}{2}(N+1)N + 2, \frac{1}{2}\frac{1}{\sqrt{S}}(M-1)(N+1)N - (M-4)(N-1) - 2\sqrt{2}S + 4\right)$	$\frac{1}{2}\frac{1}{\sqrt{S}}MN^2$
trisolv	$\frac{1}{2}N(N+1) + N + \max\left(0, \frac{1}{8}\frac{1}{S}(N-1)(N-2) - 2N - S + 5\right)$	$\frac{1}{2}N^2$
trmm	$\max\left(\frac{1}{2}M(M-1) + MN + 1, \frac{1}{\sqrt{S}}(M - 2 + \frac{\sqrt{2}}{\sqrt{S}})(M-1)N - (M-4)(N-2) - 8\sqrt{2}S + 5\right)$	$\frac{1}{\sqrt{S}}M^2N$
TC abcde-efbad-cf	$\max\left(ABCDE + EFBAD + CF, -2 + F - S + 2C + 2ABDE + \frac{2ABCDE(F-1)}{\sqrt{S}}\right)$	$\frac{2}{\sqrt{S}}ABCDEF$
TC abcd-dbea-ec	$\max\left(ABCD + DBEA + EC, -2 + E - S + 2C + 2ABD + \frac{2ABCD(E-1)}{\sqrt{S}}\right)$	$\frac{2}{\sqrt{S}}ABCDE$
TC abc-bda-dc	$\max\left(ABC + BDA + DC, -2 + D - S + 2C + 2AB + \frac{2ABC(D-1)}{\sqrt{S}}\right)$	$\frac{2}{\sqrt{S}}ABCD$
TC abcdef-dega-gfbc	$\max\left(ABCDEF + DEGA + GFBC, -2 + G - S + 2ADE + 2BCF + \frac{2ABCDEF(G-1)}{\sqrt{S}}\right)$	$\frac{2}{\sqrt{S}}ABCDEFG$
TC abc-adec-ebd	$\max\left(ABC + ADEC + EBD, -3 + DE - S + 3AC + 3B - ABC + \frac{2ABC(DE-1)}{\sqrt{S}}\right)$	$\frac{2}{\sqrt{S}}ABCDE$
TC ab-cad-dcb	$\max\left(AB + CAD + DCB, -3 + CD - S + 3A + 3B - AB + \frac{2AB(CD-1)}{\sqrt{S}}\right)$	$\frac{2}{\sqrt{S}}ABCD$
TC ab-ac-cb	$\max\left(AB + AC + CB, -2 + C - S + 2A + 2B + \frac{2AB(C-1)}{\sqrt{S}}\right)$	$\frac{2}{\sqrt{S}}ABC$

TC abcd-aebf-fdec	$\max\left(ABCD + AEBF + FDEC, -3 + EF - S + 3AB + 3CD - ABCD + \frac{2ABCD(EF-1)}{\sqrt{S}}\right)$	$\frac{2}{\sqrt{S}} ABCDEF$
conv-2d-s1	$\max\Big(BC(Y+H-1)(X+W-1) + BFXY + FCHW,$ $-2 - S + C + 4F + BY + BX + 2BXY - 2BXYF + \frac{BFXY(WHC-1)}{S},$ $-2 - S + C + 4F + BY + BX + 2BXY - 2BXFY + \frac{2BXYCF\sqrt{HW}}{\sqrt{S}} - \frac{2BXYF}{\sqrt{HWS}},$ $-2 - S + C + 4F + BY + BX + 2BXY - 2BXFY + \frac{2XYCF\sqrt{BHW}}{\sqrt{S}} - \frac{2XYF\sqrt{B}}{\sqrt{HWS}}\Big)$	$\frac{2}{\sqrt{S}} BXYCF\sqrt{HW}$

Table 3.2 – Complete Lower Bound Formulae for PolyBench, TCCG and convolutions obtained with IOLB

method presented in [2] is limited to a few algorithms. Kwasniewski et al. [31] implemented an algorithm for parallel matrix-matrix multiplication that matches the communication lower bound for any combinations of matrix dimensions, processors counts and memory sizes. See discussion on [16] for more details on these limitations.

The studies that are the most related to this chapter are those from Christ et al. [16], and Elango et al. [22, 21].

The idea of using a variant of the Brascamp-Lieb inequality to derive bounds for arbitrary affine programs comes from Christ et al. [16]. However, the approach they propose suffers from several limitations: 1. The model is based on association of operations with data elements and does not capture data dependencies in a computational DAG. Consequently, it can lead to very weak lower bounds on data movement for computations such as Jacobi stencils. 2. There is no way to (de-)compose the CDAG, and they view all the statements of the loop body (that has to be perfectly nested) as an atomic statement. As a consequence, it is incorrect to use this approach for loop computations where loop fission is possible. 3. The lower bounds modeling is restricted to S-partitioning, leading to very weak lower bounds for algorithms such as `adi` or `durbin`. 4. Obtaining scaling constants, in particular with non-orthogonal reuse directions, is difficult, and only asymptotic order complexity bounds can be derived. 5. No automation of the lower bounding process was proposed, but manually worked out examples of asymptotic complexity as a function of fast memory capacity (without scaling constants) were presented.

Elango et al. [22] used a variant of the red-blue pebble game without recomputation, enabling the composition of several sub-CDAGs, and the use of a lower-bounding approach based on wavefronts in the DAG. Manual application of the approach for parallel execution was done on specific examples, but no approach to automation was proposed.

The later work of Elango et al. [21] was the first to make the connection between paths in the data-flow graph and regular data reuse patterns and to propose an automated compiler algorithm for affine programs. However, their proposed approach suffers from several limitations: 1. Only asymptotic $\Omega(\ldots)$ data movement bounds were obtainable, without any scaling constants. In contrast, IOLB generates meaningful non-asymptotic parametric *I/O* lower bound expressions. From these expressions, we can derive asymptotic lower bounds with scaling constants, critical for use in deducing upper limits on *OI* for a roofline model. 2. Since they were only trying to provide asymptotic bounds without constants, they did not address (de-)composition (asymptotic bounds can be safely summed up even if they interfere). Also, they only considered enumerative decomposition, and not dimension decomposition through loop parameterization that is necessary to obtain a tight bound for their Matmult-Seidel illustrative example. They also only considered the simple non-overlapping notion of interference, and did not allow decomposition of the same statement, required in order to obtain a tight bound for computations like `floyd-warshall`. 3. Finally, their approach only used the S-partitioning paradigm for lower bounds but not the wavefront-based paradigm, thus leading to very weak bounds for benchmarks such as `adi` or `durbin`.

Chapter 4

Upper Bounds

We have presented a method for automatically computing data movement lower bounds for affine programs. This can be very useful to assess the optimization potential of a program, but a first step is to evaluate its actual data movement cost. This raises two questions: 1. how can we estimate the I/O cost of a program under a given schedule, as a parametric expression of parameters and cache size? 2. how do we find an I/O-efficient schedule for a program? Answering these questions can help us assess the tightness of a data movement lower bound, with the best scenario being matching lower and upper bounds: when this is the case, we have found an I/O-optimal schedule. This chapter tries to tackle this challenge for a large class of programs, and presents a method for automating the process, that was implemented in a tool called IOUB.

It includes the first algorithm for computing a symbolic over-approximation of the data movement for a parametric (multi-dimensional) tiled version of an affine code, as well as the first fully automated scheme for expressing as an operations research problem the minimization of this data movement expression.

The chapter is organized as follows. Sec. 4.1 provides some useful formalism along with the required background. Sec. 4.2 defines the tiling transformation, and the associated domains and quantities that are used in the analysis. Sec. 4.3 describes the I/O cost model, and the algorithm that we designed to compute a symbolic expression of the I/O cost of a tiled program. Sec. 4.4 explains how we reduce the search space for tiling loop permutations. Sec. 4.5 connects the previous sections into a complete framework. Sec. 4.6 reports how it was implemented into a tool called IOUB, and reports the results of our experiments on tensor contractions and convolutions. Finally, Sec. 4.7 discusses related work.

```
for(i = 0; i < Ni; i++)
  for(j = 0; j < Nj; j++)
    for(k = 0; k < Nk; k++)
      C[i][j] += A[i][k] * B[k][j];
```

(a)

```
for(i1 = 0; i1 < Ni; i1+=Ti)
  for(j1 = 0; j1 < Nj; j1+=Tj)
    for(k = 0; k < Nk; k++)
      for(i = i1; i < i1+Ti; i++)
        for(j = j1; j < j1+Tj; j++)
          C[i][j] += A[i][k] * B[k][j];
```

(b)

Figure 4.1 – Matrix-matrix multiplication: (a) untiled version (b) tiled version.

```
for(c = 0; c < Nc; c++)
  for(f = 0; f < Nf; f++)
    for(x = 0; x < Nx; x++)
      for(w = 0; w < Nw; w++)
        Out[f][x] += Image[x+w][c] * Filter[f][w][c];
```

Figure 4.2 – Running example - 1D Convolution

4.1 Background

In this section, we present the class of programs, its representations, and the memory model we consider in our analysis.

4.1.1 Class of programs

Imperfectly nested affine loop programs We consider *imperfectly nested* loop programs, which can be recursively defined as a sequence of statements and `for` loops, each being itself composed of imperfectly nested loops. A *program parameter* is a symbolic constant during the compilation, whose value will be known during the execution of the program (for example, array sizes). The access functions made by statements to arrays are *affine expressions* of surrounding loop indices and program parameters. We also assume that conditions on loop indices are affine expressions of surrounding loop indices and program parameters [8].

For example, the 1D-convolution described in Figure 4.2 matches these criteria. This program is even *perfectly nested*: all loops are nested with a single statement inside. Constants `Nc`, `Nf`, `Nx` and `Nw` are the *program parameters*. All loop indices are bounded by affine constraints (e.g. $0 \leq c < \texttt{Nc}$), and all array subscripts are affine expressions (e.g. $(x+w, c)$ for array `Image`).

Fully-tilable program The *tiling transformation* [59] is a key loop transformation to improve the data locality of a program. Given a list of consecutive dimensions (called *loop band*), this transformation groups their iterations into tiles, which are executed atomically. Figure 4.1 shows an example of tiling transformation for the matrix multiplication example: the tiled dimensions are `i` and `j` and the iterations are grouped into rectangles of size $\texttt{Ti} \times \texttt{Tj}$ (*tile shape*). Note that when the tiling shape is rectangular, each tiled dimension (e.g. `i`) is strip-mined, giving rise to the *tile dimension* that will iterate over the tiles (e.g. `i1`), and the *local dimension* that iterates inside a tile (e.g. `i` in the transformed program).

A tiling is *legal* when there is no cycle of dependencies between the computation of different tiles, that is, when the atomicity condition between tiles can be respected by the schedule of the program. The algorithms presented in this chapter are described on the assumption that the input program is *fully tilable using rectangular tiles*, which means that all its dimensions can be legally tiled by using rectangular tiles.

In theory, any affine program could be pre-processed using a polyhedral compiler to provide a fully permutable (that is, tilable using rectangular tiles) loop band, but the reality is more complex, as analyzing a non-regular iteration domain involves being able to cope with the simplification and solving of a complex system of symbolic expressions. Nevertheless, several important classes of computation fit our simplified hypothesis, such as a convolution, all kinds of tensor contractions and many linear algebraic kernels, including matrix mul-

tiplication. Figure 4.2 shows another example of such computation, which is a simplified convolution. It will be used as a running example for this chapter.

4.1.2 Program representation

For theoretical reasoning, we still use the CDAG abstraction for programs. However representing a full graph is not a practical way of handling programs, so our algorithms manipulate a more convenient intermediate representation. The formalism we use in this chapter is slightly different from the DFG we use for lower bounds, as the focus is more on arrays than on statements, but the underlying framework is the same. Let us formally define this representation.

The *iteration domain* $\mathcal{I}_S$ of a statement S is the set of integral values that the surrounding loop indices take during execution. Each point of this space is associated with an execution instance of the statement. Due to the hypotheses made on the loop bounds, the iteration domain of a statement is a *$\mathbb{Z}$-polyhedron*, that is, a set of integral points whose coordinates satisfy a set of affine constraints. The computation of the cardinality of a set E (denoted $|E|$ – symbolic expression as a function of the program parameters), can be efficiently done using the Barvinok algorithm [1] [7].

Each array A is associated with a *memory domain* $\mathcal{M}_A$, which is the multidimensional set of valid array indices for the array A. Given a statement S containing a read or a write to an array A, this occurrence is associated with a *memory access function* $f_A : \mathcal{D}_S \mapsto \mathcal{M}_A$, which is an affine function. For any given sub-domain $D_S \subset \mathcal{I}_S$ defined as a *$\mathbb{Z}$-polyhedron*, the associated data footprint $f_A(D_S)$ is also a *$\mathbb{Z}$-polyhedron* that can be easily computed by polyhedral compilation tools.

For the matrix-matrix multiplication example (Figure 4.1, untiled version), the dimensions are $\mathcal{D} = \{i, j, k\}$ and the iteration domain is $\mathcal{I} = \{(i, j, k) \mid 0 \leq i < N_i \ \wedge \ 0 \leq j < N_j \ \wedge \ 0 \leq k < N_k\}$. A 3-dimensional rectangular subset of this domain will be defined as $[l_i, u_i] \times [l_j, u_j] \times [l_k, u_k]$, where, for each dimension $d \in \mathcal{D}$, $0 \leq l_d \leq u_d < N_d$. The example code also contains three array accesses:

— for array A, with domain $\mathcal{M}_A = \{x, y \mid 0 \leq x < N_i \ \wedge \ 0 \leq y < N_k\}$ and access function $f_A(i, j, k) = (i, k)$;

— for array B, with domain $\mathcal{M}_B = \{x, y \mid 0 \leq x < N_k \ \wedge \ 0 \leq y < N_j\}$ and access function $f_B(i, j, k) = (k, j)$;

— and for array C, with domain $\mathcal{M}_C = \{x, y \mid 0 \leq x < N_i \ \wedge \ 0 \leq y < N_j\}$ and access function $f_C(i, j, k) = (i, j)$.

4.2 Loop permutation and tiling

4.2.1 Tiling transformation

Let us consider the example of Figure 4.1 that reports a tiled and non-tiled code for matrix-matrix multiplication. The original code contains three dimensions i, j, and k. The tiled code

1. In theory, Barvinok's algorithm has an exponential complexity in the number of dimensions of the set. In practice, its execution time is usually less than a few seconds, when applied to the polyhedral sets commonly encountered during program analysis.

```
for(c1 = 0; c1 < Nc; c+=Tc)
  for(f1 = 0; f1 < Nf; f+=Tf)
    for(x = 0; x < Nx; x++)
      for(c = c1; c < c1+Tc; c++)
        for(w = 0; w < Nw; w++)
          for(f = f1; f < f1+Tf; f++)
            Out[f][x] += Image[x+w][c] * Filter[f][w][c];
```

Tile limit

Figure 4.3 – Tiled code for 1D-convolution for tiling schedule $\left((w,c,f,x),\left\{T_c = \mathtt{Tc}, T_f = \mathtt{Tf}, T_x = 1, T_w = \mathtt{Nw}\right\}\right)$

contains five loops on dimensions i, j, k, i, j, from outer to inner. The three outermost loops span the entire iteration domain of size $\mathtt{Ni} \times \mathtt{Nj} \times \mathtt{Nk}$, while the two innermost ones span a polyhedron of size $\mathtt{Ti} \times \mathtt{Tj}$. We refer to the innermost (resp. outermost) part as the intra-tile (resp. inter-tile) dimensions. As we will see later, unlike the permutation of the inter-tile dimensions, that of the intra-tile dimensions is not relevant in our model. In other words, while a different schedule with loops ordered as (from outer to inner) (i1,k,j1,i,j) could lead to a different I/O estimation than for the schedule with loops ordered as (i1,j1,k,i,j) (as in Figure 4.1), our cost model will not be affected if we permute i and j as in loop order (i1,j1,k,j,i). This motivates the following notation to represent the tiling schedule of Figure 4.1: $((i,j,k),\ \left\{T_i = \mathtt{Ti}, T_j = \mathtt{Tj}, T_k = 1\right\})$ where the ordered list $\mathcal{P} = (i,j,k)$ describes the permutation of the inter-tile loop dimensions, and $T = \{T_i = \mathtt{Ti}, T_j = \mathtt{Tj}, T_k = 1\}$ describes the tile sizes. Similarly, for the example in Figure 4.2, the tiling shown in Figure 4.3 is represented as $((w,c,f,x),\ \left\{T_c = \mathtt{Tc}, T_f = \mathtt{Tf},\ T_x = 1, T_w = \mathtt{Nw}\right\})$. Note that the loop on w in the inter-tile loops (outermost loops), and the loop on x in the intra-tile loops are both omitted in the code as they would have only one iteration. The permutation of intra-tile loops in the code is arbitrary.

Let us provide a more formal definition.

Definition 14. *A* tiling schedule *is defined as a tuple* $(\mathcal{P}, T)$ *where* $\mathcal{P} = \left(d_j\right)_{|\mathcal{D}| \geq j \geq 1}$ *is a permutation of dimensions* $\mathcal{D}$ *representing the inter-tile loop order, and* $T = \{T_d\}_{d \in \mathcal{D}}$ *represent the tile dimensions. In* $\mathcal{P}$, $d_{|\mathcal{D}|}$ *represents the outermost loop dimension, while* d_1 *represents the innermost loop dimension that encloses the tile: the permutation order is outer (leftmost) to inner (rightmost). The tile size for dimension* d *is* T_d.

4.2.2 Sub-domains and reuse

To model the I/O cost of a given tiling schedule, we first need some formal definitions.

For an inter-tile loop dimension d_j, we define its *sub-domain* as the set of points in the iteration domain such that the indices of the enclosing loops d_k ($|\mathcal{D}| \geq k \geq j$) have fixed values.

Definition 15. *Let* d_j *be some inter-tile loop dimension. We denote by* $\mathtt{i}k$ *the fixed index value at level* k. *The* sub-domain SD_{d_j} *at level* j *is defined as:*

$$SD_{d_j}(\mathtt{i}D, \ldots, \mathtt{i}j) = \{i_{|\mathcal{D}|}, \ldots, i_1 \in \mathcal{I} \mid \forall k \geq j,\ \mathtt{i}k \leq i_k < \mathtt{i}k + T_{d_k}\}$$

Definition 16. *The* sub-domain data footprint *for an array A at level j is defined as:*

$$SDF_{A,j}(iD,\ldots,i\,j) = \left|f_A\left(SD_{d_j}(iD,\ldots,i\,j)\right)\right|$$

Taking the example of the 1D-convolution of Figure 4.3, the sub-domain $SD_{d_2}(\mathtt{c1},\mathtt{f1})$ for loop dimension $d_j = d_2 = f$ (recall that $d_4 = w$, $d_3 = c$, $d_2 = f$, $d_1 = x$) is $\{(w,c,f,x) \mid 0 \le x < \mathtt{Nx} \wedge \mathtt{c1} \le c < \mathtt{c1{+}Tc} \wedge 0 \le w < \mathtt{Nw} \wedge \mathtt{f1} \le f < \mathtt{f1{+}Tf}\}$.

The sub-domain data footprint at level 2 for array Image for this program is:

$$SDF_{\mathtt{Image},2}(\mathtt{c1},\mathtt{f1}) = (\mathtt{Nx{+}Nw{-}1}) \times \mathtt{Tc}$$

Whenever there is a non-empty overlap between the data used by two consecutive sub-domains, estimating it allows us to refine our cost model. This leads us to define the inter-sub-domain reuse as follows:

Definition 17. *The* inter-sub-domain reuse *for an array A at level j is:*

$$SDR_{A,j}(iD,\ldots,i\,j) = \left|f_A\left(SD_{d_j}(iD,\ldots,i\,j)\right) \cap f_A\left(SD_{d_j}(iD,\ldots,i\,j - T_{d_j})\right)\right|$$

A sharp eye would have observed that the notion of reuse from a "previous" sub-domain is meaningful for all sub-domains but the first one in the loop. To handle this subtlety, for a given loop dimension d_j (assuming its loop index starts at 0), we split the iteration space into two sub-domains: the *front domain*

$$I_{\text{front}} = \{(i_{|\mathcal{D}|},\ldots,i_1) \in \mathcal{I} \mid i_j = 0\}$$

and the *back domain*

$$I_{\text{back}} = \{(i_{|\mathcal{D}|},\ldots,i_1) \in \mathcal{I} \mid i_j \neq 0\}$$

This allows us to define, for a given sub-domain and an array A, its *inverse density* as the ratio:

$$ID_{A,j}(\mathtt{iD},\ldots,\mathtt{i\,j} \neq 0) = \frac{SDF_{A,j}(\mathtt{iD},\ldots,\mathtt{i\,j}) - SDR_{A,j}(\mathtt{iD},\ldots,\mathtt{i\,j})}{\left|SD_{d_j}(\mathtt{iD},\ldots,\mathtt{i\,j})\right|}$$

$$ID_{A,j}(\mathtt{iD},\ldots,0) = \frac{SDF_{A,j}(\mathtt{iD},\ldots,\mathtt{i\,j})}{\left|SD_{d_j}(\mathtt{iD},\ldots,\mathtt{i\,j})\right|}$$

As we will see later, the inverse density is an over-approximation of the best attainable inverse operational intensity (data movement per computation unit). As an example, for 1D-convolution:

$$\begin{aligned}
SD_x(\mathtt{c1},\mathtt{f1},\mathtt{x}) &= \{c,f,w,x \mid 0 \le w < \mathtt{Nw} \wedge x = \mathtt{x} \wedge \\
&\quad\ \mathtt{c1} \le c < \mathtt{c1{+}Tc} \wedge \mathtt{f1} \le f < \mathtt{f1{+}Tf}\} \\
|SD_x(\mathtt{c1},\mathtt{f1},\mathtt{x})| &= \mathtt{Nw} \times \mathtt{Tc} \times \mathtt{Tf} \\
SDF_{\mathtt{Image},1}(\mathtt{c1},\mathtt{f1},\mathtt{x}) &= \mathtt{Nw} \times \mathtt{Tc} \\
SDR_{\mathtt{Image},1}(\mathtt{c1},\mathtt{f1},\mathtt{x}) &= \mathtt{Tc} \times (\mathtt{Nw} - 1) \\
ID_{\mathtt{Image},1}(\mathtt{c1},\mathtt{f1},\mathtt{x} \neq 0) &= 1/(\mathtt{Nw} \times \mathtt{Tf}) \\
ID_{\mathtt{Image},1}(\mathtt{c1},\mathtt{f1},0) &= 1/\mathtt{Tf}
\end{aligned}$$

To avoid complicated expressions for non-rectangular domains, we consider the maximum value for the inverse density that we specialize for the front and the back:

$$ID^{\text{front}}_{A,j} = \max_{\mathtt{iD},\ldots,\mathtt{i\,j}/\mathtt{i\,j}=0} ID_{A,j}(\mathtt{iD},\ldots,\mathtt{i\,j})$$

$$ID^{\text{back}}_{A,j} = \max_{\mathtt{iD},\ldots,\mathtt{i\,j}/\mathtt{i\,j}\neq 0} ID_{A,j}(\mathtt{iD},\ldots,\mathtt{i\,j})$$

In many cases, the sub-domain sizes and intersections do not depend on loop indices, and there is no need to take the maximum.

4.3 Cost model

Now, we want to evaluate the I/O cost of a given tiling schedule. A tiling schedule defines the order of "computation operations" (rule ($R2$) in red-white pebble game terminology), but the order of memory operations still needs to be defined.

4.3.1 Single array

Let us start with the case when there is a single array A. We consider a tiling schedule $(\mathcal{P} = (d_{|\mathcal{D}|},\ldots,d_1), T = \{T_d\}_{d\in\mathcal{D}})$.

The basic idea for scheduling I/O operations is simple: for each tile, first bring all input data in fast memory, then execute all computations inside the tile.

For this to be possible, the data footprint of a single tile has to be smaller than the cache capacity S:

$$SDF_{A,1} \leq S.$$

For simplicity, we restrict our I/O cost model to situations where this condition holds. It would be possible to compute it when is it not the case, but the I/O cost would be high. Since our goal is to derive an upper bound as close as possible to the optimal, this is not necessary.

The basic idea described above can be improved: sometimes, consecutive tiles actually use the same data. To be more accurate, we want to find the largest loop nest such that the input data for all tiles in the loop nest still fits in fast memory. The data then only needs to be brought in fast memory once for each such "macro-tile". Finally, if there is some overlap in the input data for two consecutive tiles, only the additional data needs to be fetched.

Formally, we define the *outermost reuse dimension* for array A, as the "first" (smallest possible l) dimension d_l in $\mathcal{P}$ such that the sub-domain data footprint of A is less than the cache capacity S.

Definition 18. *The* outermost reuse dimension *l for array A is the leftmost index in $\mathcal{P}$ such that*

$$SDF_{A,l} \leq S.$$

The total I/O cost for array A derived from the inverse density as follows constitutes an upper bound on the cost of an optimal red-white pebble game:

$$IO_A = ID^{\text{front}}_{A,l} \times |I_{\text{front}}| + ID^{\text{back}}_{A,l} \times |I_{\text{back}}|$$

Indeed, by construction, each sub-domain can be executed by bringing only once (before starting execution) all the required data. Every sub-domain in the back can reuse the data

used in the previous sub-domain and the corresponding I/O can thus be saved. This means that the optimal inverse operational intensity for any sub-domain in the front (resp. in the back) is no more than $ID^{\text{front}}_{A,l}$ (resp. $ID^{\text{back}}_{A,l}$).

4.3.2 Multiple arrays

Now when there are multiple arrays $A_1,\dots,A_s$, we "cut" the cache into array-specific regions of sizes $S_1,\dots,S_s$, such that $S_1+\cdots+S_s=S$. The idea is then simply to do the same as before, with each region holding only data from its array. This is slightly restrictive, as one could imagine the optimal I/O schedule using some memory space alternatively for different arrays. But it is necessary to make the problem tractable, and does not prevent IOUB from giving optimal or close to optimal bounds in many cases.

The outermost reuse dimension for array A_i is defined is the same way as in the one array case:

Definition 19. *The* outermost reuse dimension l_i *for array* A_i *is the leftmost index in* $\mathcal{P}$ *such that*

$$SDF_{A,l_i} \leq S_i.$$

The result is straightforwardly derived from the one array case, and is summed up in the following lemma:

Lemma 7. *Let us consider a program with s arrays* $A_1,\dots,A_s$*, and a tiling* $(\mathcal{P}=(d_{|\mathcal{D}|},\dots,d_1),T=\{T_d\}_{d\in\mathcal{D}})$ *for this program. Let* $S_1,\dots,S_s$ *be integers such that* $S_1+\cdots+S_s=S$*, and for all* $i=1,\dots,s$*:*

$$SDF_{A_i,1} \leq S_i. \tag{4.1}$$

Then the following expression is an upper bound on the cost of a red-white pebble game for this program:

$$IO = IO_{A_1}+\cdots+IO_{A_s}$$

where

$$IO_{A_i} = ID^{front}_{A_i,l_i}\times\left|I_{front}\right| + ID^{back}_{A_i,l_i}\times\left|I_{back}\right| \tag{4.2}$$

Conditions (4.1) can be replaced by their sum, avoiding the need to explicitly define S_is:

$$\sum_i SDF_{A,l_i} \leq S.$$

However this notation makes the assumptions more explicit, and makes it possible the cache partition when solving the equations numerically.

Example On our running example, for array Image, supposing the outermost reuse dimension is x:

$$\begin{aligned}
IO_{\texttt{Image}} &= ID^{\text{front}}_{\texttt{Image},1}\times|I_{\text{front}}| + ID^{\text{back}}_{\texttt{Image},1}\times|I_{\text{back}}| \\
&= \frac{1}{\texttt{Tf}}\times\texttt{Nw.Nc.Nf} + \frac{1}{\texttt{Nw.Tf}}\times\texttt{Nw.Nc.Nf.(Nx}-1) \\
&= \frac{\texttt{Nc.Nf.(Nx}+\texttt{Nw}-1)}{\texttt{Tf}}
\end{aligned}$$

The same computation for arrays `Out` and `Filter` yields (assuming the outermost reuse dimensions are respectively x and f):

$$IO_{\texttt{Out}} = \texttt{Nc.Nf.Nx/Tc}$$
$$IO_{\texttt{Filter}} = \texttt{Nc.Nf.Nw}$$

Conditions on outermost reuse dimensions are:

$$\begin{aligned} SDF_{\texttt{Image},1}(\texttt{c1},\texttt{f1},\texttt{x}) = &\quad \texttt{Nw} \times \texttt{Tc} &\leq S_{\texttt{Image}} \\ SDF_{\texttt{Out},1}(\texttt{c1},\texttt{f1},\texttt{x}) = &\quad \texttt{Tf} &\leq S_{\texttt{Out}} \\ SDF_{\texttt{Filter},2}(\texttt{c1},\texttt{f1}) = &\quad \texttt{Tf} \times \texttt{Nw} \times \texttt{Tc} &\leq S_{\texttt{Filter}} \end{aligned}$$

This leads to the following bound:

$$IO = \frac{\texttt{Nc.Nf.(Nx+Nw-1)}}{\texttt{Tf}} + \frac{\texttt{Nc.Nf.Nx}}{\texttt{Tc}} + \texttt{Nc.Nf.Nw}$$
$$\text{Assuming} \quad \texttt{Nw} \times \texttt{Tc} + \texttt{Tf} + \texttt{Tf} \times \texttt{Nw} \times \texttt{Tc} \leq S.$$

4.3.3 Extension to multiple memory levels

The red-white pebble game and the corresponding lower bound reasoning frameworks can be extended to multi-level memory hierarchies [46]. The above computation of I/O can also be extended by simply considering one tiling band per cache level and independently applying the previous reasoning to each level. One only needs to add constraints on tile sizes so that tiles get bigger at each level.

The question that arises is: what is the I/O cost of a multi-level tiling? Our model can provide a cost for each memory level, but how to aggregate these costs is not straightforward. In a real-life CPU with multiple levels of cache, it is likely that the I/O cost is dominated by the level where the cost is the closest to the bandwidth. In this work, we chose to use a weighted sum, where the cost at each level is multiplied by the inverse bandwidth.

4.3.4 Optimization problem

Having a symbolic expression as a function of the tile sizes makes it possible to express the problem of choosing tile sizes as a non-linear optimization problem (NLP). Thus, for fixed values of S and program parameters, an NLP solver such as IPOPT [58] can be used to find optimizing tile sizes for this particular permutation. One still needs to be cautious, as the tile sizes found by such tools are not guaranteed to be integers, and rounding can have an impact on the I/O cost.

This is only valid for a fixed permutation of tiling loops, as well as fixed outermost reuse dimensions for each array. Indeed, since tile sizes are variables, conditions (4.1) come as constraints that are verified afterwards. The next section explains how to choose these parameters.

4.4 Loop permutation selection

We have so far only focused on computing the required I/O cost for a given permutation of the inter-tile loops. Unfortunately, the search space of all possible permutations grows exponentially with the number of dimensions and with the memory depth.

However, it is straightforward to see that many permutations are equivalent in terms of I/O cost: for example, switching the two outer loops `c1` and `f1` in the conv-1D example in Figure 4.3 would not change the I/O costs. Moreover, some permutations are better than others: for example, it is not worth exploring the permutation whose innermost dimensions do not allow any data reuse between two successive tiles. In this section, we provide some insights into how to select a subset of relevant permutations for a given code.

4.4.1 Reuse for an array along a dimension

First, we define a notion of *reuse*: for a given dimension d and an array A, there is *reuse* for array A on dimension d if, when putting dimension d innermost (that is, setting $d_1 = d$), the sub-domain data footprint at level 2 for A *does not increase much* compared to level 1:

$$SDF_{A,2} - SDF_{A,1} \ll SDF_{A,1}$$

This allows us to decide which dimensions are worth putting at the innermost levels.

In simple cases, this notion is not ambiguous: for array `Out` in the conv-1D example, c is a reuse dimension as setting it innermost would lead to $SDF_{\texttt{Out}_2} = SDF_{\texttt{Out}_1} = \texttt{Tf} \times \texttt{Tx}$, while f is not a reuse dimension as setting it innermost would lead to $SDF_{\texttt{Out}_2} - SDF_{\texttt{Out}_1} = (\texttt{Nf} - \texttt{Tf}) \times \texttt{Tx}$ (unless $\texttt{Nf} = \texttt{Tf}$ but in that case it means that f iterates only once). For more complex subscript expressions, such as for array `Image`, the criterion is less clear: setting w as innermost and assuming it is not part of the tile would lead to comparing $\texttt{Tw} - 1$ with $\texttt{Tx}$ (as $SDF_{\texttt{Image}_2} = (\texttt{Tx} + \texttt{Tw} - 1) \times \texttt{Tc}$, $SDF_{\texttt{Image}_1} = \texttt{Tx} \times \texttt{Tc}$). In other words, a reuse criterion requires an "oracle" (such as the user) to provide information about small and large dimensions.

This also provides information on how to choose outermost reuse dimensions, defined in the previous section. For an array A, this dimension will be the highest one such that all lower dimensions are reuse dimensions for array A.

4.4.2 Algorithm for permutation selection

Assuming such an oracle and a reuse criterion, Algorithm 7 selects a set of representative permutations. This algorithm builds a permutation from the innermost to the outermost dimensions while keeping track of the set of remaining dimensions (variable $\mathcal{D}'$), and of the set S of arrays having a potential reuse along each dimension d (variable R). When a dimension is chosen, the set of potential reuse is updated. When there is no more potential reuse, an arbitrary permutation of the remaining dimensions is chosen.

Figure 4.4 illustrates the steps of the algorithm on the 1D convolution (Figure 4.2), and how it narrows the loop permutation space to three permutations (one of them is the one used for the running example in Figure 4.3). Note that some permutations are pruned during the selection process (red cross in Figure 4.4), because they have strictly less reuse potential than others. For example, selecting dimension c as the innermost dimension allows us to have reuse on array `Out`, but dimension w allows us to have reuse on one more array (`Out` and `Image`).

4.5 Putting it all together

The IOUB tool combines the permutation selection and I/O cost computation as follows: it first selects a list of permutations and generates the corresponding set of tiled versions (with

input : dimensions $\mathcal{D}$, arrays $\mathcal{A}$, reuse oracle
output: list of permutations $\mathcal{P}$

```
1  function genPerm(D', R)
2      if D' = ∅ then
3          return {()};
4      if S = ∅ for all (d, S) ∈ R then
5          return {P} where P is an arbitrary permutation of D';
6      P := ∅;
7      forall (d, S) ∈ R such that ∄(d', S') ∈ R, S ⊊ S' do
8          D'_d := D' \ {d};
9          R_d := {(d', S' ∩ S), (d', S') ∈ R \ {(d, S)}};
10         P_d := genPerm(D'_d, R_d);
11         P := P ∪ {(P, d), P ∈ P_d};
12     return P;
13 R := {(d, {A ∈ A, reuse(A, d)}), d ∈ D};
14 return genPerm (D, R);
```

Algorithm 7: Generation of the list of loop permutations that maximize reuse

Figure 4.4 – Application of Algorithm 7 to the 1D convolution kernel

Figure 4.5 – Summary of the full framework of IOUB, illustrated on matrix-matrix multiplication

tile sizes as parameters) using Algorithm 7. For each permutation, it computes a symbolic expression (as a function of program parameters, tile sizes, and cache size) that represents the required I/O, using the method described in Sec. 4.2 and 4.3, as well as constraints on tile sizes. Then, this can be fed to an NLP solver to find the best tile sizes for each permutation. The final I/O cost is the minimum over all versions, and IOUB also provides a basic tiled code that implements the corresponding tiling scheme. It can also be useful to have a symbolic bound that does not depend on tile sizes, but only on program parameters and cache size. A discussion on how to derive such an expression for the examples we used in experiments can be found in Sec. 4.6.2. This is summarized in Fig. 4.5.

4.6 Experiments

Using our new I/O cost computation method, and our improvements to the lower bound algorithm for computing I/O complexity, IOUB is able to provide tight bounds for both tensor contractions (even with small dimensions) and convolutions. The general code for convolution is shown on Fig. 4.7. An example of tensor contraction is given on Fig. 4.6.

```
for (a = 0; a < A; a++)
  for (b = 0; b < B; b++)
    for (c = 0; c < C; c++)
      for (d = 0; d < D; d++)
                for (e = 0; e < E; e++)
          Out[a,b,c] += In1[a,d,e,c] * In2[e,b,d];
```

Figure 4.6 – Tensor contraction kernel (abc-adec-ebd)

```
for (b = 0; b < B; b++)
  for (c = 0; c < C; c++)
    for (f = 0; f < F; f++)
      for (x = 0; x < X; x++)
        for (y = 0; y < Y; y++)
          for (h = 0; h < H; h++)
            for (w = 0; w < W; w++)
               Out[f,x,y,b] += Image[x+h,y+w,c,b] * Filter[f,h,w,c];
```

Figure 4.7 – Convolution kernel

4.6.1 Benchmarks

We illustrate this by running it on representative benchmarks: For convolutions, we considered 11 different layers of Yolo9000 [44]. The parameter values for each layer are shown in Fig. 4.8. For tensor contractions, we considered the ones from TCCG [50].

The benchmark Python script from TCCG source code gathers 73 tensor contraction kernels (originating from various sources), that can be reduced to 49 different kernels, once duplicates are identified. For each kernel, TCCG selects the sizes of every dimension so that they are multiple of 8, roughly equal, and so that the product of all sizes is around 200×2^{20}.

We can further reduce the number of relevant kernels that need to be considered in our analysis, by grouping them according to the number of dimensions of each array, and the number of dimensions shared between them. Indeed, the array layouts (dimension order) do not impact our analysis. This yields eight classes of tensor contraction kernels, described in Figure 4.9.

4.6.2 Symbolic upper bound expressions

The method presented in this chapter provides a symbolic expression for the I/O of a program, as a function of program parameters and tile sizes. To compare them with the parametric lower bound expressions, we would like to remove tile sizes from the expression, and express them as functions of the cache size instead. In general, this is too complicated to solve. However, for the benchmarks we consider, this is possible by using only a few assumptions.

Let us start with the matrix multiplication example from Figure 4.1, with tiling

$$\left((i,j,k),\left\{T_i = \mathsf{Ti}, T_j = \mathsf{Tj}, T_k = 1\right\}\right).$$

Layer	F	C	X	Y	W	H
Yolo9000-0	32	3	544	544	3	3
Yolo9000-2	64	32	272	272	3	3
Yolo9000-4	128	64	136	136	3	3
Yolo9000-5	64	128	136	136	1	1
Yolo9000-8	256	128	68	68	3	3
Yolo9000-9	128	256	68	68	1	1
Yolo9000-12	512	256	34	34	3	3
Yolo9000-13	256	512	34	34	1	1
Yolo9000-18	1024	512	17	17	3	3
Yolo9000-19	512	1024	17	17	1	1
Yolo9000-23	28272	1024	17	17	1	1

Figure 4.8 – Parameter values for convolutional layers of Yolo9000

Kernel	dim.	s. d.	Problem sizes
abcde-efbad-cf	5 5 2	4 1 1	48/32/24/32/48/32
abcd-dbea-ec	4 4 2	3 1 1	72/72/24/72/72
abc-bda-dc	3 3 2	2 1 1	312/312/296/312
abcdef-dega-gfbc	6 4 4	3 3 1	24/16/16/24/16/16/24
abc-adec-ebd	3 4 3	2 1 2	72/72/72/72/72
ab-cad-dcb	2 3 3	1 1 2	312/296/312/312
ab-ac-cb	2 2 2	1 1 1	5136/5136/5120
abcd-aebf-fdec	4 4 4	2 2 2	72/72/72/72/72/72

Figure 4.9 – Classes of Tensor Contraction kernels from the TCCG benchmarks. The classes are determined from the number of dimensions of the arrays (Out / In1 / In2), and the number of shared dimensions (s. d.) between arrays (Out+In1 / Out+In2 / In1+In2)

Kernel	I/O upper bound
TC abcde-efbad-cf	UB = $\frac{2ABCDEF}{\sqrt{S+1}-1} + CF$
TC abcd-dbea-ec	UB = $\frac{2ABCDE}{\sqrt{S+1}-1} + CE$
TC abc-bda-dc	UB = $\frac{2ABCD}{\sqrt{S+1}-1} + CD$
TC abcdef-dega-gfbc	UB = $\frac{2ABCDEFG}{\sqrt{S+1}-1} + BCFG$
TC abc-adec-ebd	UB = $\frac{2ABCDE}{\sqrt{S+1}-1} + BDE$
TC ab-cad-dcb	UB = $\frac{2ABCD}{\sqrt{S+1}-1} + AB$
TC ab-ac-cb	UB = $\frac{2ABC}{\sqrt{S+1}-1} + BC$
TC abcd-aebf-fdec	UB = $\frac{2ABCDEF}{\sqrt{S+1}-1} + CDEF$
2D Convolution	UB = $CFHWXY\left(\frac{1}{XY} + \frac{1}{H\Delta W} + \frac{(H+\Delta-1)(W+X-1)}{H\Delta^2 WX}\right)$ where $\Delta = \frac{-HW+W+\sqrt{H^2W^2+4HSW-2HW^2+4SW+4S+W^2}}{2(HW+W+1)}$

Figure 4.10 – Combined parametric I/O bounds of tensor contraction (TC) and 2D convolution kernels. S is the small memory size and other uppercase letters are problem sizes (for TC kernels A…F are tensor dimensions, for convolution see Fig. 4.7).

The IO cost expression and the constraints on tile sizes are:

$$IO = IO_A + IO_B + IO_C = \mathtt{Ni} \cdot \mathtt{Nj} \cdot \mathtt{Nk} \cdot \left(\frac{1}{\mathtt{Ti}} + \frac{1}{\mathtt{Tj}} + \frac{1}{\mathtt{Nk}} \right) \tag{4.3}$$

$$SDF_{A,1} + SDF_{B,1} + SDF_{C,1} = \mathtt{Ti} + \mathtt{Tj} + \mathtt{Ti} \cdot \mathtt{Tj} \leq S \tag{4.4}$$

Then, we consider square tiles, by assuming that Ti and Tj are equal to the same value T. We also assume that the tile completely fills the cache, which means that we consider that inequality (4.4) is actually an equality. While this may not be the best solution, these hypotheses will still provide a valid bound. We obtain the following equality:

$$2T + T^2 = S.$$

It has a unique positive solution:

$$T = \sqrt{S+1} - 1.$$

We can then plug this value back into expression (4.3) to get our symbolic bound:

$$IO = \mathtt{Ni} \cdot \mathtt{Nj} \cdot \left(\frac{2\mathtt{Nk}}{\sqrt{S+1} - 1} + 1 \right)$$

Note that the dominant term matches the dominant term of the I/O lower bound for matrix multiplication.

Here, we chose the tiling scheme manually, but this can be automated using the NLP solver with some relevant numerical values of parameters. It will return a permutation and numerical tile sizes, and we can use the information on which dimensions have $T_d = 1$ or $T_d = N_d$ to guide our symbolic solving. If the problem sizes are significantly bigger than $\sqrt{S}$, the best tiling found by the solver will precisely be this one.

To generalize this reasoning to tensor contraction kernels, we only need to add some extra information on the expected order of magnitude of tile sizes.

Dimensions in a tensor contraction computation can be divided into three groups such that after "merging" the dimensions in each group, the computation is equivalent to a matrix multiplication. This corresponds to the "shared dimensions" in Fig. 4.9. The condition we impose is that the products of tile sizes inside each group are equal. For example abc-adec-ebd, the three groups are $\{a,c\}, \{b\}$ and $\{d,e\}$, and the condition is $T_a T_c = T_b = T_d T_e$.

The expressions are shown in Figure 4.10. They correspond to the "general case" where the input size is not the bottleneck, i.e., when parameters are sufficiently large compared to S.

The expression for convolution is quite complex, but an asymptotic analysis shows that the highest order term is $\frac{2CFXY\sqrt{HW}}{\sqrt{S}}$ when C, F, X and Y are sufficiently large, which matches the third term in the lower bound (the B factor does not appear since it is equal to 1 in all our benchmarks).

4.6.3 Comparison of upper and lower bounds for different cache sizes

Figure 4.11 shows a comparison between lower and upper bounds for the considered benchmarks. For several cache sizes (from 16 kB to 4MB), the lower bound is computed by plugging the actual values in the symbolic expressions shown above, and the upper bound is obtained by running the NLP solver on the optimization problems generated by IOUB.

Figure 4.11 – I/O bounds (lower is blue and upper is orange) of tensor contractions (TCCG) and convolutions (Yolo) for different cache sizes

As a sanity check, we confirm that the upper bound found is always above the lower bound, which was not trivial for the convolution bounds, and that both curves are decreasing or are constant along S. Both bounds are close to each other, ranging from at most a factor of 3 between them for Yolo9000-2 on the smallest value of S, to almost the same values for the *ab-ac-cb* TC kernel, which is also known as matrix multiplication. Moreover, the upper bound and the lower bound come closer for the largest values of S, which shows a match between the asymptotic dominant term of both bounds. When the cache size becomes very large, the dominant cost becomes the initial loading of the input data, hence the matching bounds and the horizontal line for some benchmarks.

4.6.4 Implementation

IOUB is implemented as a command-line tool, written mostly in Python and using the ISL [53] and Sympy [37] libraries, as well as the IPOPT [58] NLP solver. Figure 4.12 show an example of input file and output from IOUB for a convolution kernel and a 3-level memory hierarchy. In its current version, the input format is a yaml file describing loops and arrays manually, but IOUB could be combined with polyhedral tools like PET [56] and PluTo [12] to generate this intermediate representation directly from C code.

4.7 Related work

Computing an I/O complexity upper bound for an algorithm is the most reasonable way to assess the tightness of a lower bound. While this computation is usually done by hand using ad hoc techniques specific to each studied algorithm [38, 18, 1, 48, 31, 43], Fauzia et al. [23] proposed a heuristic that directly reasons on the CDAG, which unfortunately does not scale to real programs. Finding an upper bound for a fixed architecture can also be viewed as finding an optimized program transformation that minimizes data movement costs, which also implies being able to evaluate this cost. Thus, restricting the analysis to affine programs and using the polyhedral framework appears to be appropriate for this problem. However, while the seminal scheduling algorithm from PluTo [12] is able to expose tilable loops and generate tiled code, it can only handle fixed tile sizes. This is because parametric tiling is not an affine transformation: a tiled affine code with parametric tile sizes is no longer affine. A consequence is that existing polyhedral tools cannot be used to evaluate the I/O cost of such a code. These tools include PolyFeat [5], which computes an approximation of the number of capacity misses for arbitrary affine programs, as well as other algorithms [6, 14, 27] which focus on precisely modeling conflict misses. Some of these algorithms are restricted to a small class of programs, and most of them can only model a one-level cache. They all need to consider fixed parameters and fixed tile sizes. Other works [60, 47, 32] implemented ad hoc computation of this cost function for very restricted sub-classes of programs, but our algorithm is the first to be able to automatically generate a symbolic expression of it for arbitrary parametric tiled affine programs, in a multi-level cache setting.

We showed how we use this cost function to find the best loop permutations and tile sizes for a given architecture, using operations research. Once again, this is out of the scope of polyhedral analysis, as cost functions are not affine. So the usual optimization strategies used by polyhedral compilers [28, 12, 55], mostly based on parametric integer programming [24], cannot be used. Some literature exists on tile size selection, but it mostly deals with performance

```
dims : [b, f, c, x, y, w, h]

arrays:
   O : [ [b, f, x, y], 2]
   I : [ [b, c, x+w, y+h], 1]
   K : [ [f, c, w, h], 1]
reuse :
    O: [c, w, h]
    I: [f, w, h]
    K: [b, x, y]

values:
  n_levels: 2
  cache: [8192, 262144]
  dims: [1, 32, 128, 96, 96, 3, 3]
```

Cache sizes (in words): [8192, 262144]
Best solution:

Permutation: L1:[y,f,c,w,h] L2:[x,y] MEM:[w,h,c,b,f,x,y]

Footprint: $\{L1: Tb_0 \cdot Tc_0 \cdot (Th_0 + Ty_0 - 1) \cdot (Tw_0 + Tx_0 - 1) + Tb_0 \cdot Tf_0 \cdot Tx_0 \cdot Ty_0 + Tc_0 \cdot Tf_0 \cdot Th_0 \cdot Tw_0,\ L2: Tb_1 \cdot Tc_1 \cdot (Th_1 + Ty_1 - 1) \cdot (Tw_1 + Tx_1 - 1) + Tb_1 \cdot Tf_1 \cdot Tx_1 \cdot Ty_1 + Tc_1 \cdot Tf_1 \cdot Th_1 \cdot Tw_1\}$

Analytical cost expression:

$$\left\{L1: Nb \cdot Nc \cdot Nf \cdot Nh \cdot Nw \cdot Nx \cdot Ny \cdot \left(\frac{(Th_0 + Ty_0 - 1) \cdot (Tw_0 + Tx_1 - 1)}{Tf_0 \cdot Th_0 \cdot Tw_0 \cdot Tx_1 \cdot Ty_0} + \frac{1}{Tc_0 \cdot Th_0 \cdot Tw_0} + \frac{1}{Tb_1 \cdot Tx_1 \cdot Ty_1}\right),\ L2: Nb \cdot Nc \cdot Nf \cdot Nh \cdot Nw \cdot Nx \cdot Ny \cdot \left(\frac{1}{Tb_1 \cdot Tx_1 \cdot Ty_1} + \frac{(Nh + Ty_1 - 1) \cdot (Nw + Tx_1 - 1)}{Nh \cdot Nw \cdot Tf_1 \cdot Tx_1 \cdot Ty_1} + \frac{1}{Nc \cdot Nh \cdot Nw}\right)\right\}$$

$$Nb \cdot Nc \cdot Nf \cdot Nh \cdot Nw \cdot Nx \cdot Ny \cdot \left(\frac{1}{Tb_1 \cdot Tx_1 \cdot Ty_1} + \frac{(Nh + Ty_1 - 1) \cdot (Nw + Tx_1 - 1)}{Nh \cdot Nw \cdot Tf_1 \cdot Tx_1 \cdot Ty_1} + \frac{1}{Nc \cdot Nh \cdot Nw}\right)$$

$$+ Nb \cdot Nc \cdot Nf \cdot Nh \cdot Nw \cdot Nx \cdot Ny \cdot \left(\frac{(Th_0 + Ty_0 - 1) \cdot (Tw_0 + Tx_1 - 1)}{Tf_0 \cdot Th_0 \cdot Tw_0 \cdot Tx_1 \cdot Ty_0} + \frac{1}{Tc_0 \cdot Th_0 \cdot Tw_0} + \frac{1}{Tb_1 \cdot Tx_1 \cdot Ty_1}\right)$$

Optimal sizes: $Tx_0=1$ $Tb_0=1$ $Ty_0=11$ $Tf_0=32$ $Tc_0=23$ $Tw_0=3$ $Th_0=3$ $Tx_1=96$ $Tb_1=1$ $Ty_1=46$ $Tf_1=32$ $Tc_1=23$ $Tw_1=3$ $Th_1=3$

Parameters: $C_1=8192$ $C_2=262144$ $Nb=1$ $Nf=32$ $Nc=128$ $Nx=96$ $Ny=96$ $Nw=3$ $Nh=3$

Total Cost: 4769783

Data volume: L1: 3141356 L2: 1628427

Data footprint: L1: 7873 L2: 256128

Schedule:

```
for (y2=0; y2<96; y2+=46)
  for (c2=0; c2<128; c2+=23) {
    Sy2=min(46,96-y2) // in {4, 46}
    for (y1=y2; y1<y2+Sy2; y1+=11)
      for (x1=0; x1<96; x1+=1)
        for (h=0; h<3; h+=1)
          for (w=0; w<3; w+=1) {
            Sc2=min(23,128-c2) // in {13, 23}
            for (c=c2; c<c2+Sc2; c+=1)
              for (f=0; f<32; f+=1) {
                Sy1=min(11,y2+Sy2-y1) // in {2, 11, 4}
                for (y=y1; y<y1+Sy1; y+=1)
                  <bb>;
}}}
```

Figure 4.12 – Input file and output from IOUB

model design and is either handcrafted for specific kernels and access function patterns [33, 40] or uses machine learning [60] and even cache simulators [36]. Polyhedral analysis is only used for the tiling transformation. Renganarayana and Rajopadhye [45] showed that most performance models for tile size selection used in the literature are polynomials that are operations research-friendly. Our automatically generated cost function, which matches the distinct-access model promoted by Ferrante et al. [26], fits into this category.

Chapter 5

Conclusion

5.1 Summary of Results

This thesis presents a set of methods and a theoretical framework to automatically analyze the data movement complexity of affine programs. More precisely, for a two-level memory hierarchy with a fast memory of limited size S, our algorithms are able to find:

— a symbolic, non-asymptotic lower bound on the number of loads (transfers from slow to fast memory) that are necessary to execute this program under any valid schedule, as a function of S and program parameters;

— a symbolic upper bound on the number of loads, obtained by comparing several possible tiling transformations;

— a concrete tiled code with numerical tile sizes for fixed values of program parameters and fast memory size.

These two algorithms have been implemented as open-source tools, and evaluated on multiple examples. On tensor contractions and convolution kernels, the expressions output by both tools match up to lower-order terms, effectively closing the theoretical problem of their I/O complexity. The lower bound algorithm was also applied on all programs in PolyBench, and is able to match or even improve over the state-of-the-art in many cases.

5.2 Limitations

Despite these promising results, IOLB and IOUB still suffer from limitations, due to the inherent complexity of these problems.

Lower bounds One category of programs for which the bounds we get are not tight are stencils (e.g. jacobi and seidel in PolyBench). This type of computation exhibits complex data dependencies (array subscripts like A[i+1][j-1]) that are not found in linear algebra kernels for instance. While some careful manual analysis can be used to slightly improve the constant factor of the expression compared to the automatic analysis, bridging the gap between lower and upper bounds is still an open problem. Moreover, the techniques presented here (projection and wavefront-based) do not cover all possible data dependency patterns, even inside the class of affine programs, and some other methods would have to be developed in order to prove tight lower bounds on gramschmidt for instance.

Upper bounds Here the limitations come more from the rapidly increasing complexity of expressions when trying to tackle more involved programs. When iteration domains get complex, for instance when loop counters depend on one another, or loop bounds are more involved functions of parameters, cardinalities computed by Barvinok's algorithm also get more complex, with several possible expressions depending on relations between parameters. This can greatly increase the size of the optimization problem, making it currently intractable. Another limit is the method to generate symbolic upper bounds: currently we rely on solving a polynomial equation, which is not generally doable when the degree exceeds four.

Finally, such automatic analyses can only be applied on a specific class of programs. It seems realistic to extend this work to get tight bounds on a large part of practical polyhedral programs, and even on some non-polyhedral ones, but many programs will certainly stay out of the scope of this type of automated analysis anyway.

5.3 Future work

The work presented here could be continued in many directions. Some of the limitations we pointed out could be overcome: for instance the affine requirement could be relaxed by using under- or over-approximations, and more complex wavefront patterns could be included in IOLB. The technique for generating upper bounds that do not depend on tile sizes in IOUB could also be generalized to cases when the polynomial equation is not solvable exactly, by relaxing the problem of finding the precise expression of a tile size that completely fills the cache to only finding a size that does not exceed the cache capacity.

Applications As we explained earlier IOUB is able to provide numerical tile sizes suggestions that minimize I/O in our model. The next step is to integrate these suggestions into a code generation framework, and get running programs that effectively use a CPU as close to its peak performance as possible. As of today, producing high-performance code is a difficult problem, for which a lot of manual tuning and machine-specific expertise is often needed. More and more solutions for automatic code generation are developed (TVM [15], Halide [41]), with various compromises between expressiveness and code performance. Multi-level tiling is a key transformation to achieve this goal, but it must be combined with fine-tuned basic blocks that are able to exploit modern CPU mechanisms such as vectorization, memory prefetching, instruction-level parallelism, and adapt to machine-specific details. Choosing tiling loop sizes and permutations is a crucial step, and an analytical model such as the one we developed can be very useful to guide the search in a highly combinatorial space. It should be coupled with some autotuning, in order to take into account various aspects that we do not model, such as data layout, cache line sizes, partial tiles handling, control-flow overhead, etc.

One notable project is BLIS [61], which aims at proving a portable high-performance implementation of BLAS-like linear algebra. They use a static tiling scheme, manually derived from an analytical model, for all configurations. This scheme is the one automatically derived for matrix-matrix multiplication by IOUB in a 4-level memory setting with ordinary matrix dimensions. In the context of tensor computations and convolutions, great efforts have been done by Nicolas Tollenaere et al. [51] They achieve performance very close to native libraries like oneDNN, with higher flexibility and automation.

More general extensions Let us suggest two interesting directions. The first one concerns parallelism: the work presented here mostly deals with sequential programs, however most applications in today's computing exhibit some level of parallelism. While some data-movement lower bounds techniques can be transposed in a parallel setting to prove bounds on the volume of communication, this is not always generalizable, and the challenge of generating and analyzing highly communication-efficient parallel programs remains vastly unsolved. A first major step would be to extend the analytical model from IOUB to distributed memory communication volume, and be able to provide data distribution and communication scheme suggestions. Manual exploration shows that it should be possible to derive communication-efficient algorithms like 2.5D matrix multiplication [49] with a similar approach.

Another direction of research would be to extend our analytical I/O cost model to include more complex aspects, such as data layout: on an actual machine data is grouped in cache lines, and conflicts can appear, so evaluating the costs and benefits of reorganizing the way data is stored (packing) could be valuable.

Bibliography

[1] Alok Aggarwal and Jeffrey S. Vitter. « The Input/Output Complexity of Sorting and Related Problems ». In: *Communications of the ACM* 31 (9 1988), pp. 1116–1127. URL: https://doi.org/10.1145/48529.48535.

[2] Grey Ballard, Erin Carson, James Demmel, Mark Hoemmen, Nick Knight, and Oded Schwartz. « Communication lower bounds and optimal algorithms for numerical linear algebra ». In: *Acta Numerica* 23 (2014), pp. 1–155.

[3] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. « Graph Expansion and Communication Costs of Fast Matrix Multiplication ». In: *Journal of the ACM* 59.6 (Jan. 2013). DOI: 10.1145/2395116.2395121.

[4] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. « Minimizing Communication in Numerical Linear Algebra ». In: *SIAM J. Matrix Analysis Applications* 32.3 (2011), pp. 866–901. DOI: 10.1137/090769156.

[5] Wenlei Bao, Changwan Hong, Sudheer Chunduri, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, and P. Sadayappan. « Static and Dynamic Frequency Scaling on Multicore CPUs ». In: *ACM Transactions on Architecture and Code Optimization* 13.4 (2016). DOI: 10.1145/3011017.

[6] Wenlei Bao, Sriram Krishnamoorthy, Louis-Noël Pouchet, and P. Sadayappan. « Analytical modeling of cache behavior for affine programs ». In: *Proceeding of the ACM on Programming Languages* 2.POPL (2018). URL: https://doi.org/10.1145/3158120.

[7] Alexander I. Barvinok. « A Polynomial Time Algorithm for Counting Integral Points in Polyhedra when the Dimension is Fixed ». In: *Mathematics of Operations Research* 19.4 (1994), pp. 769–779. DOI: 10.1287/moor.19.4.769.

[8] Cédric Bastoul, Albert Cohen, Sylvain Girbal, Saurabh Sharma, and Olivier Temam. « Putting Polyhedral Loop Transformations to Work ». In: *Languages and Compilers for Parallel Computing, 16th International Workshop (LCPC)*. Vol. 2958. Lecture Notes in Computer Science. Springer, 2003, pp. 209–225. DOI: 10.1007/978-3-540-24644-2_14.

[9] Christian Bauer, Alexander Frink, and Richard Kreckel. « Introduction to the GiNaC Framework for Symbolic Computation within the C++ Programming Language ». In: *J. Symbolic Computation* 33 (2002), pp. 1–12.

[10] Gianfranco Bilardi and Enoch Peserico. « A characterization of temporal locality and its portability across memory hierarchies ». In: *Automata, Languages and Programming* (2001), pp. 128–139.

[11] Gianfranco Bilardi, Michele Scquizzato, and Francesco Silvestri. « A Lower Bound Technique for Communication on BSP with Application to the FFT ». In: *Euro-Par 2012 Parallel Processing - Proceedings*. Ed. by Christos Kaklamanis, Theodore S. Papatheodorou, and Paul G. Spirakis. Vol. 7484. Lecture Notes in Computer Science. Springer, 2012, pp. 676–687. DOI: 10.1007/978-3-642-32820-6_67.

[12] 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. DOI: 10.1145/1375581.1375595.

[13] Herm Jan Brascamp and Elliott H Lieb. « Best constants in Young's inequality, its converse, and its generalization to more than three functions ». In: *Advances in Mathematics* 20.2 (1976), pp. 151–173.

[14] Siddhartha Chatterjee, Erin Parker, Philip J. Hanlon, and Alvin R. Lebeck. « Exact Analysis of the Cache Behavior of Nested Loops ». In: *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2001, pp. 286–297. DOI: 10.1145/378795.378859.

[15] Tianqi Chen et al. « TVM: An Automated End-to-End Optimizing Compiler for Deep Learning ». In: *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*. Ed. by Andrea C. Arpaci-Dusseau and Geoff Voelker. USENIX Association, 2018, pp. 578–594.

[16] Michael Christ, James Demmel, Nicholas Knight, Thomas Scanlon, and Katherine Yelick. *Communication Lower Bounds and Optimal Algorithms for Programs that Reference Arrays – Part 1*. 2013. arXiv: 1308.0068v1 [math.CA].

[17] Michael Christ, James Demmel, Nicholas Knight, Thomas Scanlon, and Katherine Yelick. « On Holder-Brascamp-Lieb inequalities for torsion-free discrete Abelian groups ». In: *arXiv preprint arXiv:1510.04190* (2015).

[18] James Demmel and Grace Dinh. *Communication-Optimal Convolutional Neural Nets*. 2018. arXiv: 1802.06905v2 [cs.DS].

[19] James Demmel, Laura Grigori, Mark Hoemmen, and Julien Langou. « Communication-optimal Parallel and Sequential QR and LU Factorizations ». In: *SIAM Journal on Scientific Computing* 34.1 (2012), A206–A239. DOI: 10.1137/080731992.

[20] Jack Dongarra, Jean-François Pineau, Yves Robert, Zhiao Shi, and Frédéric Vivien. « Revisiting matrix product on master-worker platforms ». In: *International Journal of Foundations of Computer Science* 19.6 (2008), pp. 1317–1336.

[21] Venmugil Elango, Fabrice Rastello, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan. « On Characterizing the Data Access Complexity of Programs ». In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 2015, pp. 567–580. DOI: 10.1145/2676726.2677010.

[22] Venmugil Elango, Fabrice Rastello, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan. « On characterizing the data movement complexity of computational DAGs for parallel execution ». In: *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures, (SPAA)*. 2014, pp. 296–306.

[23] Naznin Fauzia, Venmugil Elango, Mahesh Ravishankar, J. Ramanujam, Fabrice Rastello, Atanas Rountev, Louis-Noël Pouchet, and P. Sadayappan. « Beyond Reuse Distance Analysis: Dynamic Analysis for Characterization of Data Locality Potential ». In: *ACM Transaction on Architecture and Code Optimization* 10.4 (Dec. 2013). DOI: 10.1145/2541228.2555309.

[24] Paul Feautrier. « Parametric Integer Programming ». In: *RAIRO Recherche Opérationnelle* 22.3 (1988), pp. 243–268.

[25] Paul Feautrier and Christian Lengauer. « Polyhedron model ». In: *Encyclopedia of Parallel Computing*. 2011, pp. 1581–1592.

[26] Jeanne Ferrante, Vivek Sarkar, and W. Thrash. « On Estimating and Enhancing Cache Effectiveness ». In: *Proceedings of the Languages and Compilers for Parallel Computing (LCPC), Fourth International Workshop*. Vol. 589. Lecture Notes in Computer Science. Springer, 1991, pp. 328–343. DOI: 10.1007/BFb0038674.

[27] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. « Cache Miss Equations: a Compiler Framework for Analyzing and Tuning Memory Behavior ». In: *ACM Transaction on Programming Languages and Systems* 21.4 (1999), pp. 703–746. DOI: 10.1145/325478.325479.

[28] Tobias Grosser, Armin Größlinger, and Christian Lengauer. « Polly - Performing Polyhedral Optimizations on a Low-Level Intermediate Representation ». In: *Parallel Processing Letter* 22.4 (2012). DOI: 10.1142/S0129626412500107.

[29] Jia-Wei Hong and H. T. Kung. « I/O complexity: The Red-Blue Pebble Game ». In: *Proceedings of the 13th Annual ACM Symposium on Theory of Computing (STOC)*. Milwaukee, Wisconsin, United States: ACM, 1981, pp. 326–333. DOI: 10.1145/800076.802486.

[30] Dror Irony, Sivan Toledo, and Alexandre Tiskin. « Communication Lower Bounds for Distributed-Memory Matrix Multiplication ». In: *Journal of Parallel and Distributed Computing* 64.9 (2004), pp. 1017–1026. DOI: 10.1016/j.jpdc.2004.03.021.

[31] Grzegorz Kwasniewski, Marko Kabic, Maciej Besta, Joost VandeVondele, Raffaele Solcà, and Torsten Hoefler. « Red-blue Pebbling Revisited: Near Optimal Parallel Matrix-Matrix Multiplication ». In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 2019. DOI: 10.1145/3295500.3356181.

[32] Rui Li, Aravind Sukumaran-Rajam, Richard Veras, Tze Meng Low, Fabrice Rastello, Atanas Rountev, and P. Sadayappan. « Analytical Cache Modeling and Tilesize Optimization for Tensor Contractions ». In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. Ed. by Michela Taufer, Pavan Balaji, and Antonio J. Peña. ACM, 2019. DOI: 10.1145/3295500.3356218.

[33] Junyi Liu, John Wickerson, and George A. Constantinides. « Tile Size Selection for Optimized Memory Reuse in High-level Synthesis ». In: *27th International Conference on Field Programmable Logic and Applications, FPL 2017, Ghent, Belgium, September 4-8, 2017*. Ed. by Marco D. Santambrogio, Diana Göhringer, Dirk Stroobandt, Nele Mentens, and Jari Nurmi. IEEE, 2017, pp. 1–8. DOI: 10.23919/FPL.2017.8056810.

[34] Lynn H. Loomis and Hassler Whitney. « An inequality related to the isoperimetric inequality ». In: *Bulletin of the American Mathematical Society* 55 (1949), pp. 961–962.

[35] Bradley Lowery and Julien Langou. *Improving the Communication Lower Bounds for Matrix-Matrix Multiplication*. 9th Scheduling for Large Scale Systems Workshop, July 1-4, Lyon, France. 2014.

[36] Sanyam Mehta, Gautham Beeraka, and Pen-Chung Yew. « Tile Size Selection Revisited ». In: *ACM Trans. Archit. Code Optim.* 10.4 (2013), 35:1–35:27. DOI: 10.1145/2541228.2555292.

[37] Aaron Meurer et al. « SymPy: Symbolic Computing in Python ». In: *PeerJ Computer Science* 3 (Jan. 2017), e103. ISSN: 2376-5992. DOI: 10.7717/peerj-cs.103.

[38] Auguste Olivry, Julien Langou, Louis-Noël Pouchet, P. Sadayappan, and Fabrice Rastello. « Automated Derivation of Parametric Data Movement Lower Bounds for Affine Programs ». In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2020, pp. 808–822. DOI: 10.1145/3385412.3385989.

[39] Louis-Noël Pouchet and Tomofumi Yuki. *PolyBench/C 4.2*. https://sourceforge.net/projects/polybench/. 2015.

[40] Nirmal Prajapati, Waruna Ranasinghe, Sanjay V. Rajopadhye, Rumen Andonov, Hristo Djidjev, and Tobias Grosser. « Simple, Accurate, Analytical Time Modeling and Optimal Tile Size Selection for GPGPU Stencils ». In: *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Austin, TX, USA, February 4-8, 2017*. Ed. by Vivek Sarkar and Lawrence Rauchwerger. ACM, 2017, pp. 163–177. DOI: 10.1145/3018743.3018744.

[41] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. « Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines ». In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. Ed. by Hans-Juergen Boehm and Cormac Flanagan. ACM, 2013, pp. 519–530. DOI: 10.1145/2491956.2462176.

[42] Desh Ranjan, John E. Savage, and Mohammad Zubair. « Strong I/O Lower Bounds for Binomial and FFT Computation Graphs ». In: *Computing and Combinatorics - 17th Annual International Conference, COCOON 2011, Dallas, TX, USA, August 14-16, 2011. Proceedings*. Ed. by Bin Fu and Ding-Zhu Du. Vol. 6842. Lecture Notes in Computer Science. Springer, 2011, pp. 134–145. DOI: 10.1007/978-3-642-22685-4_12.

[43] Desh Ranjan, John E. Savage, and Mohammad Zubair. « Upper and Lower I/O Bounds for Pebbling r-Pyramids ». In: *J. Discrete Algorithms* 14 (2012), pp. 2–12. DOI: 10.1016/j.jda.2011.12.005.

[44] Joseph Redmon and Ali Farhadi. « YOLO9000: Better, Faster, Stronger ». In: *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*. IEEE Computer Society, 2017, pp. 6517–6525. DOI: 10.1109/CVPR.2017.690.

[45] Lakshminarayanan Renganarayanan and Sanjay V. Rajopadhye. « Positivity, Posynomials and Tile Size Selection ». In: *Proceedings of the ACM/IEEE Conference on High Performance Computing (SC)*. IEEE/ACM, 2008, p. 55. DOI: 10.1109/SC.2008.5213293.

[46] John E. Savage. « Extending the Hong-Kung Model to Memory Hierarchies ». In: *Proceedings of the First Annual International Conference on Computing and Combinatorics*. COCOON '95. Berlin, Heidelberg: Springer-Verlag, 1995, pp. 270–281. DOI: 10.1007/BFb0030842.

[47] Jun Shirako, Kamal Sharma, Naznin Fauzia, Louis-Noël Pouchet, J. Ramanujam, P. Sadayappan, and Vivek Sarkar. « Analytical Bounds for Optimal Tile Size Selection ». In: *Proceedings of Compiler Construction - 21st International Conference (CC)*. Vol. 7210. Lecture Notes in Computer Science. Springer, 2012, pp. 101–121. DOI: 10.1007/978-3-642-28652-0_6.

[48] Tyler Michael Smith, Bradley Lowery, Julien Langou, and Robert A. van de Geijn. « A Tight I/O Lower Bound for Matrix Multiplication ». In: (2019). arXiv: 1702.02017v2.

[49] Edgar Solomonik and James Demmel. « Communication-Optimal Parallel 2.5D Matrix Multiplication and LU Factorization Algorithms ». In: *Euro-Par 2011 Parallel Processing - 17th International Conference, Euro-Par 2011, Bordeaux, France, August 29 - September 2, 2011, Proceedings, Part II*. Vol. 6853. Lecture Notes in Computer Science. Springer, 2011, pp. 90–109. DOI: 10.1007/978-3-642-23397-5_10.

[50] Paul Springer and Paolo Bientinesi. *Design of a High-Performance GEMM-like Tensor-Tensor Multiplication*. 2016. arXiv: 1607.00145.

[51] Nicolas Tollenaere, Auguste Olivry, Guillaume Iooss, Hugo Brunie, Albert Cohen, P Sadayappan, and Fabrice Rastello. « Efficient convolution optimisation by composing micro-kernels ». working paper or preprint. Oct. 2021. URL: https://hal.archives-ouvertes.fr/hal-03149553.

[52] S. I. Valdimarsson. « The Brascamp-Lieb polyhedron. » In: *Canadian Journal of Mathematics* 62.4 (2010), pp. 870–888.

[53] Sven Verdoolaege. « ISL: An Integer Set Library for the Polyhedral Model ». In: *Mathematical Software–ICMS 2010*. 2010, pp. 299–302.

[54] Sven Verdoolaege. *Presburger Formulas and Polyhedral Compilation*. https://libisl.sourceforge.io/tutorial.pdf. 2021.

[55] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. « Polyhedral Parallel Code Generation for CUDA ». In: *ACM Transactions on Architecture and Code Optimization* 9.4 (Jan. 2013). DOI: 10.1145/2400682.2400713.

[56] Sven Verdoolaege and Tobias Grosser. « Polyhedral Extraction Tool ». In: *Second International Workshop on Polyhedral Compilation Techniques (IMPACT'12)*. 2012.

[57] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. « Counting Integer Points in Parametric Polytopes Using Barvinok's Rational Functions ». In: *Algorithmica* 48.1 (2007), pp. 37–66. DOI: 10.1007/s00453-006-1231-0.

[58] Andreas Wächter and Lorenz T. Biegler. « On the Implementation of an Interior-Point Filter Line-Search Algorithm for large-scale nonlinear programming ». In: *Math. Program.* 106.1 (2006), pp. 25–57. DOI: 10.1007/s10107-004-0559-y.

[59] Jingling Xue. *Loop Tiling for Parallelism*. Vol. 575. Kluwer International Series in Engineering and Computer Science. Kluwer, 2000. DOI: 10.1007/978-1-4615-4337-4.

[60] Tomofumi Yuki, Lakshminarayanan Renganarayanan, Sanjay Rajopadhye, Charles Anderson, Alexandre E. Eichenberger, and Kevin O'Brien. « Automatic Creation of Tile Size Selection Models ». In: *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. ACM, 2010, pp. 190–199. DOI: 10.1145/1772954.1772982.

[61] Field G. Van Zee and Robert A. van de Geijn. « BLIS: A Framework for Rapidly Instantiating BLAS Functionality ». In: *ACM Trans. Math. Softw.* 41.3 (2015), 14:1–14:33. DOI: 10.1145/2764454.