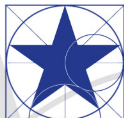


PSL UNIVERSITÉ PARIS

THÈSE DE DOCTORAT DE L'UNIVERSITÉ PSL

Préparée à MINES ParisTech

Analyse et compilation de langages de programmation parallèle

Analysis and Compilation of Parallel Programming Languages

Soutenue par

Adilla SUSUNGI

Le 26 novembre 2018

École doctorale n°432

Sciences des métiers de l'ingénieur

Spécialité

Informatique temps-réel, robotique et automatique

Composition du jury :

Gaétan HAINS Professeur, Université Paris-Est Créteil	*Président*
Philippe CLAUSS Professeur, Université de Strasbourg	*Rapporteur*
P. SADAYAPPAN Professeur, Ohio State University	*Rapporteur*
Christine EISENBEIS Directrice de recherche, Inria	*Examinateur*
Claude TADONKI Directeur de recherche, MINES ParisTech	*Directeur de thèse*
Albert COHEN Directeur de recherche, Inria	*Co-directeur de thèse*

MINES ParisTech | PSL

Remerciements
Acknowledgements

Je souhaiterais exprimer ma reconnaissance envers toute personne ayant contribué, de près ou de loin, à la complétion de cette thèse.

Je remercie tout d'abord François Irigoin de m'avoir accueillie au sein du centre de recherche en informatique (CRI) à Fontainebleau et de m'avoir donné l'opportunité de travailler sur le projet ACOPAL, financé par l'université Paris Sciences et Lettres.

Je ne prendrai jamais pour acquis le fait d'avoir été dirigée par Claude Tadonki et Albert Cohen. Tout le long de cette thèse vous m'avez impressionnée tant par votre expertise scientifique que par votre amabilité, votre sens de l'encouragement et votre intégrité.

Je remercie chaleureusement les membres de mon jury: Gaétan Hains ayant accepté de présider le jury, Philippe Clauss et P. Saday ayant rapporté mon manuscrit de thèse, Christine Eisenbeis ayant participé en tant qu'examinatrice, ainsi que Jeronimo Castrillon et Norman Rink, nos invités. *I would like to particularly thank Jeronimo who gave me the opportunity to visit his lab, the Chair for Compiler Construction at TU Dresden, and collaborate with him and Norman. I owe half of this thesis work to our collaboration and, especially, Norman's investment and precious help. Also, thanks to Immo Huismann, Jörg Stiller and Jochen Fröhlich for providing access to their applications.*

J'ai énormément de chance d'avoir été entourée par les membres du CRI (anciens ou actuels), c'est-à-dire, Corinne Ancourt, Arthur Aubertin, Maksim Berezov, Laila Bouhouch, Catherine Le Caër, Fabien Coelho, Laurent Daverio, Emilio Gallego, Guillaume Genestier, Florian Gouin, Pierre Guillou, Olfa Haggui, Olivier Hermant, Pierre Jouvelot, Patryk Kiepas, Nelson Lossing, Claire Medrala, Luc Perera, Benoît Pin, Bruno Sguerra, Lucas Sguerra, Pierre Wargnier et, bien que n'étant pas membre du CRI, je n'oublie certainement pas Monika Rakoczy; ils sont toujours prêts à aider si besoin (scientifiquement parlant ou de façon purement pratique) et sont tout simplement sympathiques, drôles, voire fous à lier ;-)

De par mes nombreuses visites à l'ENS, j'ai eu le plaisir de côtoyer Ulysse Beaugnon, Andi Drebes, Guillaume Iooss, Chandan Reddy, Oleksandr Zinenko et Jie Zhao. *My visit at the Chair for Compiler Construction also allowed me to meet great peers including Hasna Bouraoui, Sebastian Ertel, Andrés Goens, Fazal Hameed, Asif Ali Khan, Robert Khazanov, Nesrine Khouzami, Christian Menard and Lars Schütze.*

Enfin, merci à ma famille et mes amis d'avoir plus cru en moi que moi même.

Table of Contents

Table of Contents iii

1 Introduction **1**
1.1 Parallel Architectures, Programming Languages and Challenges 6
1.2 Research Context 9
1.3 Thesis Contributions and Outline 12

2 Intermediate Representation for Explicitly Parallel Programs: State-of-the-art **13**
2.1 Intermediate languages 14
2.2 Program Representations Using Graphs 17
2.3 The Static Single Assignment Form 21
2.4 The Polyhedral Model 22
2.5 Discussion 25
2.5.1 Observations 25
2.5.2 Perspectives 29

3 The Tensor Challenge **31**
3.1 Numerical Applications 32
3.2 Computational Fluid Dynamics Applications: Overview 33
3.3 CFD-related Optimization Techniques 34
3.3.1 Algebraic Optimizations 35
3.3.2 Loop Transformations 37
3.4 Envisioned Tool Flow 39
3.5 Existing Optimization Frameworks 40
3.5.1 Linear and Tensor Algebra Frameworks 40
3.5.2 Levels of Expressiveness and Optimization Control 43
3.6 Outcomes 43

4 The NUMA Challenge **45**
4.1 NUMA Architectures: Topologies and Management Solutions 46
4.1.1 Operating Systems 46
4.1.2 NUMA APIs 48

4.1.3 Languages extensions . . . 50
4.2 Case Study: Beyond Loop Optimizations for Data Locality . . . 51
4.2.1 Experimental Setup . . . 51
4.2.2 Observations . . . 52
4.3 Refining NUMA Memory Allocation Policies . . . 57
4.3.1 Thread array regions and page granularity . . . 58
4.3.2 Implementation . . . 59
4.3.3 Limitations . . . 60
4.4 Data Replications Implementation . . . 61
4.4.1 Conditional Branching . . . 61
4.4.2 Replication Storage . . . 63
4.5 On Run-time Decisions . . . 64
4.6 Outcomes . . . 65

5 TeML: the Tensor Optimizations Meta-language 67
5.1 Overview . . . 68
5.2 Tensors . . . 69
5.2.1 $N-$dimensional Arrays . . . 70
5.2.2 Compute Expressions . . . 73
5.2.3 Tensor Operations . . . 74
5.2.4 Initializations . . . 77
5.3 Loop Generation and Transformations . . . 77
5.4 Memory Allocations . . . 81
5.5 Implementation and Code Generation . . . 81
5.6 On Data Dependencies . . . 83
5.6.1 Dependency Checks using Explicit Construct . . . 84
5.6.2 Decoupled Management . . . 85
5.7 Evaluation . . . 87
5.7.1 Expressing Tensor Computations . . . 87
5.7.2 Reproducing Loop Optimization Paths . . . 89
5.7.3 Performance . . . 89
5.8 Conclusion . . . 94

6 Formal Specification of TeML 95
6.1 Formal specification . . . 96
6.1.1 Domains of trees . . . 96
6.1.2 State . . . 97
6.1.3 Valuation functions . . . 97
6.2 Compositional definitions . . . 104
6.2.1 Loop transformations . . . 104
6.2.2 Tensor operations . . . 108
6.3 Range Inference . . . 110

6.4 Towards Type Safety 111
6.5 Conclusion 112

7 Conclusion and Perspectives **115**

Bibliography **119**

A ICC Optimizations Reports **i**
A.1 Naive Interpolation i
A.2 Naive Interpolation with Loop-invariant Code Motion v

Chapter 1

Introduction

Traduction en Français

La puissance des architectures parallèles ne cesse de croître depuis leur apparition ; les supercalculateurs les mieux classés ces dernières années illustrent bien cette tendance. En effet, les performances maximales pour *ASCI Red* (1998), *Roadrunner* (2008) et *Summit* (2018) sont respectivement de l'ordre des teraFLOPS, petaFLOPS et exaFLOPS. Cette différence notable peut s'expliquer par deux principaux facteurs. Premièrement, de plus en plus de cœurs sont intégrés afin d'offrir plus de parallélisme : 9 152 cœurs pour *ASCI Red*, 129 600 cœurs pour *Roadrunner* et 2 282 544 cœurs pour *Summit*. Deuxièmement, divers aspects liés au matériel (incluant la hiérarchie de la mémoire, la capacité de stockage et les interconnexions) sont améliorés ; par exemple l'ordonnancement des threads, les instructions vectorielles ou la gestion de la mémoire.

Un tel potentiel de puissance de calcul vient au prix d'une conception de plus en plus complexe et parfois difficile à appréhender. Il est pourtant nécessaire de comprendre ces architectures afin de les exploiter. Un premier pas vers cette compréhension est la reconnaissance des différents types modèles mémoires pouvant composer une architecture parallèle. Dans les *systèmes à mémoire partagée*, les cœurs partagent généralement le même espace mémoire. En revanche, les *systèmes à mémoire distribuée* sont des grappes de machines reliées par le réseau, où chaque machine possède sa propre mémoire. Ainsi, l'accès à des données situées sur des machines distantes n'est possible que si des communications explicites sont instanciées. Dans le contexte du calcul accélérée, un système *hétérogène* comprend des processeurs manycore tels que les GPUs et les FPGAs contrôlés à partir d'une machine standard (l'hôte) via un bus PCI.

De plus, certaines architectures peuvent être intrinsèquement hétérogènes. Par exemple, ARM `big.LITTLE` associe des cœurs de processeurs économes et lents avec des cœurs rapides à forte consommation. D'autres exemples sont les APU AMD qui combinent des processeurs multicœurs et un processeur graphique sur une seule puce.

En fonction du type de système ciblé, différents langages de programmation parallèle et APIs doivent être utilisés. OpenMP [9], Pthreads [12] ou Cilk [6] ont d'abord été conçus pour

les systèmes à mémoire partagée, MPI [7] pour les systèmes à mémoire distribuée et CUDA [2] ou OpenCL [8] pour les accélérateurs. En outre, les superordinateurs étant généralement des grappes de processeurs combinées à des accélérateurs, il est courant de combiner des langages en fonction des types de modèles mémoires présents (par exemple, OpenMP + CUDA, MPI + Pthreads, etc). Cependant, une autre tendance est également d'apporter plus de transparence dans la programmation hétérogène avec des extensions de langage (par exemple, le support du calcul accélérée dans OpenMP 4.0 ou encore l'introduction de communications unilatérales dans MPI 3.0). Une autre alternative est l'introduction du modèle PGAS (*Partitioned Global Access Space* ou encore “modèle de mémoire partagée distribuée”). Dans ce type de modèle de mémoire, la mémoire partagée est partitionnée de manière logique afin que différents threads aient leur propre espace d'adressage. Les nouveaux langages tels que X10 [15], Chapel [1] ou UPC [14] suivent le modèle PGAS.

La programmation parallèle peut être néanmoins difficile. En effet, plusieurs problèmes inhérents au parallélisme existent.

Les *data races*

Survenant lorsque, sans synchronisation appropriée, différents threads accèdent au même emplacement de mémoire et, sur au moins un de ces accès, une écriture est effectuée. Ceci entraîne des résultats inexacts. Cependant, certaines *data races* peuvent être délibérées : celles-ci sont dites bénignes.

Les *deadlocks*

On parle de deadlock lorsqu'un thread attend un événement qui ne se produira jamais ou lorsque deux threads s'attendent simultanément. Par conséquent, le programme ne se termine jamais. Cela peut se produire tant bien dans le modèle de mémoire partagée que dans le modèle de mémoire distribuée.

Les synchronisations

Une mauvaise gestion des synchronisations de threads nuit aux performances. Par exemple, des sections critiques avec trop de charge de travail impliquent l'attente de threads. Le temps d'inactivité de threads qui attendent trop longtemps doit donc être réduit.

Les faux partages

Les faux partages (*false sharing*) sont inhérents aux systèmes à mémoire partagée avec une cohérence de cache trop conservatrice (notamment celle basées sur les protocoles MESI ou MOESI). Ils se produisent lorsque plusieurs cœurs partagent la même ligne de cache. En effet, une ligne de cache partagée est invalidée chaque fois qu'il y a une écriture. Par conséquent,

les autres threads lisant leurs données doivent récupérer fréquemment la ligne de cache, ce qui dégrade les performances.

Localité de données et latences de transferts

Les latences de transferts se produisent à différentes échelles. Nous pourrions par exemple distinguer les communications entre processus dans un modèle de mémoire distribuée (ou hétérogène) des transferts de données standard au sein d'un système à mémoire partagée. Nous pourrions encore distinguer les transferts à l'échelle du registre vectoriel dans les architectures SIMD. Cependant le principe reste le même ; une mauvaise localité des données implique des latences de transfert plus élevées.

Les goulots d'étranglement

Les goulots d'étranglement surviennent lorsque trop de threads accèdent à la même mémoire, saturant ainsi le bus de communication. Il s'agit d'un facteur spécifique de latences de transferts élevées, typiques des applications ne tenant pas compte des architectures NUMA (*Non-Uniform Memory Access*).

Contexte de recherche

Relever les défis du parallélisme nécessite une connaissance approfondie de divers aspects liés au programme à optimiser ainsi que l'architecture sous-jacente.

Premièrement, l'algorithme en question pourrait être modifié afin d'exposer le parallélisme et améliorer l'efficacité des accès mémoires. Dans certains cas, une programmation parallèle inefficace provient d'un manque de connaissance du langage de programmation utilisé. Par exemple, une utilisation incorrecte des constructeurs de synchronisation entraîne une exécution incorrecte du programme (éventuellement provoquée par des data races).

Deuxièmement, les différents types de systèmes décrits au préalable peuvent impliquer différents types de stratégies pour exploiter pleinement leur potentiel. Par conséquent, il peut être difficile d'optimiser une application s'exécutant sur un cluster hétérogène. En outre, nous devons non seulement étudier quelle transformation de programme est adéquate sur une architecture donnée, mais nous avons également parfois besoin d'une connaissance approfondie de certains langages de programmation pour tirer le meilleur parti du matériel. Un exemple typique est CUDA, spécialement conçu pour les GPU NVIDIA : son abstraction de bas niveau peut augmenter la courbe d'apprentissage. Par conséquent, une programmation efficace sur des architectures hétérogènes implique un large éventail de connaissances qu'un programmeur doit posséder.

Les compilateurs sont des outils dont le flux est souvent caractérisé par (i) l'analyse d'un code source et sa traduction en une (ou plusieurs) représentation(s) intermédiaire(s), (ii) l'application de différentes techniques de transformation pour optimiser le programme et (iii) la traduction de la version optimisée du code en code assembleur (compilation traditionnelle) ou en un code source de haut niveau (compilation source-à-source). L'application

de transformations sur les représentations intermédiaires afin d'optimiser le programme implique que l'analyse et la vérification du programme font également partie du processus. Par conséquent, les compilateurs peuvent aider à alléger la tâche d'un programmeur en prenant en charge une partie de tout le refactoring de code nécessaire pour utiliser efficacement l'architecture cible.

Malheureusement, les technologies de compilation actuelles ne semblent pas répondre aux besoins de la compilation parallèle. En effet, leurs *représentations intermédiaires* (RI), qui sont des clés pour effectuer des optimisations, ont été originellement conçues pour des programmes séquentiels uniquement. Cela ne restreint pas la possibilité de compiler des programmes parallèles car les constructions parallèles sont traduites en appels runtime. Mais certaines limitations persistent:

1. **Les représentations intermédiaires n'intègrent pas la sémantique des constructeurs parallèles**. Sans sémantique parallèle, les RIs ont une expressivité restreinte pour adapter les techniques d'optimisation courantes aux programmes parallèles et permettre des techniques spécifiques au parallélisme. Par exemple, comment un compilateur peut-il analyser les instances de synchronisation pour éviter des data races si sa RI ne contient aucune abstraction des synchronisations ?
2. **Les appels runtime empêchent l'application de techniques d'optimisations classiques sur les parties séquentielles d'un programme parallèle**. Comme le compilateur ne connaît pas les effets secondaires de ces appels d'exécution, il abandonne tout choix d'optimisation. Même les compilateurs pour les programmes parallèles usent d'appels runtime, comme en est le cas du compilateur Chapel qui traduit le code source en C ou C ++ avec des appels runtime.

Par conséquent,

Comment pourrions-nous repenser la conception des représentations intermédiaires pour les architectures parallèles?

Contributions de cette thèse

Cette thèse contribue à divers aspects liés aux représentations intermédiaires pour les architectures parallèles, dans le cadre de la compilation statique. Elle inclue :

1. un état de l'art des représentations intermédiaires parallèles ;
2. la conception, l'implémentation et la formalisation de TeML, un méta-langage pour l'optimisation d'applications tensorielles, à la lumière des perspectives de recherche exposées par l'état de l'art.

Publications

Les travaux de cette thèse ont conduit à plusieurs publications (et une soumission en cours d'évaluation) sur lesquelles certaines parties de ce manuscrit sont basées, voire sont des extraits.

Journaux

A. Susungi, C. Tadonki. Intermediate Representations for Explicitly Parallel Programs. Submitted to *ACM Computing Surveys* in November 2018.

Conférences internationales

A. Susungi, N. A. Rink, A. Cohen, J. Castrillón, C. Tadonki. Meta-programming for cross-domain tensor optimizations. In Proceedings of *Generative Programming: Concepts & Experiences (GPCE)*, Boston, November 2018.

A. Susungi. On the Semantics of Loop Transformation Languages (Extended abstract). In Proceedings of *2nd International Conference on the Art, Science, and Engineering of Programming (<Programming'18> Companion)*, Nice, April 2018.

A. Susungi, N. A. Rink, J. Castrillón, I. Huismann, A. Cohen, C. Tadonki, J. Stiller, J. Fröhlich. Towards Generative and Compositional Tensor Optimizations (Short paper). In Proceedings of *Generative Programming: Concepts & Experiences (GPCE)*, Vancouver, October 2017.

Workshop internationaux

N. A. Rink, I. Huismann, **A. Susungi**, J. Castrillón, J. Stiller, J. Fröhlich, C. Tadonki, A. Cohen. CFDlang: High-level Code Generation for High-Order Methods in Fluid Dynamics. In Proceedings of *International Workshop on Real World Domain-specific Languages (RWDSL)*, Vienna, February 2018.

A. Susungi, A. Cohen, C. Tadonki. More Data Locality for Static Control Programs on NUMA Architectures. In Proceedings of *International Workshop on Polyhedral Compilation Techniques (IMPACT)*, Stockholm, January 2017.

* ♣ * ♣ *

1.1 Parallel Architectures, Programming Languages and Challenges

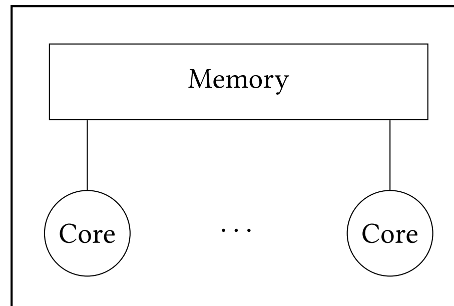
The power of parallel architectures has been ever increasing since their emergence and top-ranked supercomputers in these last years illustrate this trend. The order of magnitude of peak performances for *ASCI Red* (1998), *Roadrunner* (2008) and *Summit* (2018) are respectively teraFLOPS, petaFLOPS and exaFLOPS. Such differences every decade can be explained by two main factors. First, more and more cores are integrated for greater parallelism: 9,152 cores for *ASCI Red*, 129,600 cores for *Roadrunner* and 2,282,544 cores for *Summit*. Second, hardware organization (including memory hierarchy, storage capacity, and interconnections) is revisited to improve various aspects such as thread scheduling, vector instructions, and memory management. The potential of recent hardware comes along with a more complex design that can be difficult to grasp, but necessary to understand to leverage them.

The first step towards understanding is acknowledging the different types of memory systems composing a parallel architecture (cf. Figure 1.1). In *shared memory systems*, cores share the same memory and addressing space. On the other hand, *distributed memory systems* are clusters of machines linked through the network where each machine has its own private memory. Thus, accesses to data located on remote machines require the instantiation of explicit communications. In the context of accelerated computing, *heterogeneous* systems include manycore processors such as GPUs and FPGAs monitored from a standard machine (the host) through a PCI bus.

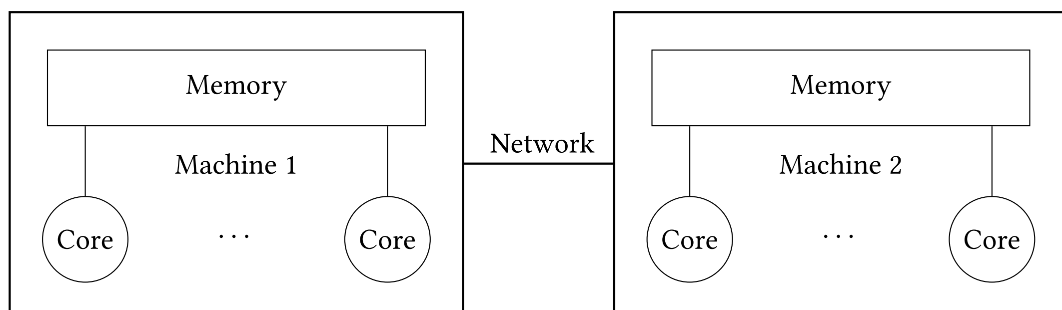
Heterogeneity can be inherent to certain architectures. For instance, the ARM `big.LITTLE` (cf. Figure 1.2) couples slow processor cores with high-speed cores. Other examples are AMD APUs that combine multi-core processors and a GPU on a single chip (cf. Figure 1.3).

Different parallel programming languages and APIs are used, depending on the targeted system. OpenMP [9], Pthreads [12] or Cilk [6] were first designed for shared memory systems, MPI [7] for distributed memory systems and CUDA [2] or OpenCL [8] for accelerators. Furthermore, as supercomputers are clusters of CPUs combined with accelerators, it is common to combine languages targeting different memory systems (i.e. OpenMP + CUDA, MPI + Pthreads, etc). Though, another trend is to offer more transparency in heterogeneous programming with language extensions (i.e. support of accelerated computing in OpenMP 4.0, the introduction of one-sided communications in MPI 3.0). Another alternative is the introduction of the PGAS model (also known as “distributed-shared memory model”). This type of memory model is based on a logical partitioning of the shared memory so that different threads have their own addressing space. New languages such as X10 [15], Chapel [1] or UPC [14] follow the PGAS model.

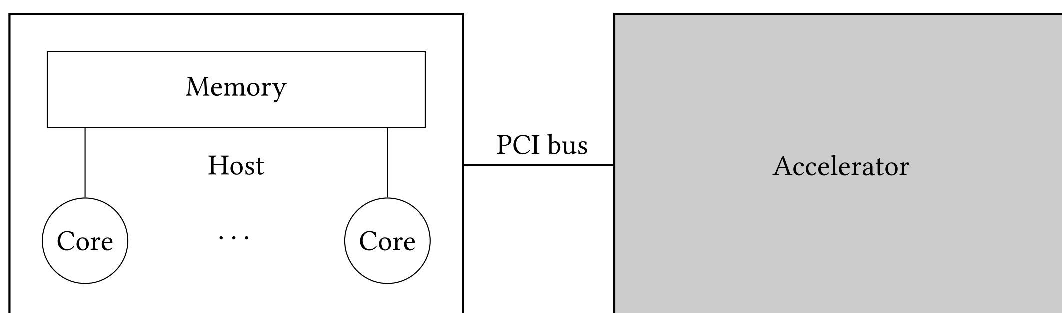
Due to various issues inherent to parallelism, parallel programming can be challenging.



(a) Shared memory system



(b) Distributed memory system



(c) Heterogeneous memory system

Figure 1.1: Different types of memory systems in parallel architectures

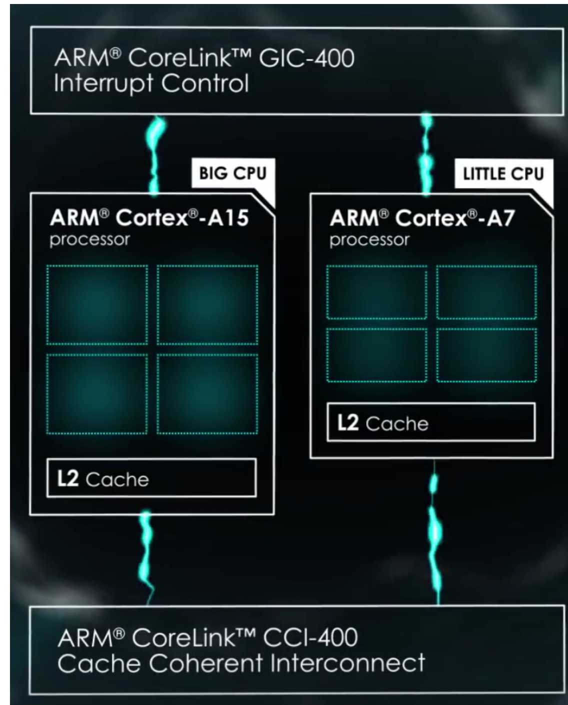


Figure 1.2: ARM `big.LITTLE`

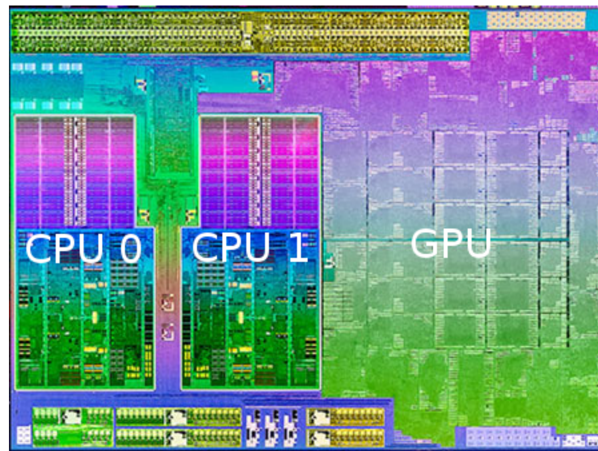


Figure 1.3: AMD Trinity APU die shot

Data races

Data races occur when, without proper synchronizations, different threads access to the same memory location and at least on these accesses is a write. These lead to inaccurate results. Nevertheless, *benign data races* are to be distinguished as they are deliberate.

Deadlocks

Deadlocks, that lead to programs that never terminates, can be created in different circumstances, e.g. a thread waiting for an event that will never occur, or two threads waiting for each other to complete their tasks. Deadlocks can be encountered in the shared memory model as well as in the distributed memory model.

Synchronizations

An improper use of synchronization constructs can lead to poor performances. For example, critical sections containing too much workload can result into thread stalling.

False sharing

Inherent to shared memory systems with too conservative cache coherence (especially when based on MESI or MOESI protocols), it occurs when several cores share the same cache line. The shared cache line is invalidated whenever one of the cores writes to it. Therefore, other threads reading their data need to re-fetch the cache line. Frequent data re-fetching degrades performances.

Data locality, transfer latencies

Poor data locality can increase transfer latencies and the number of memory accesses. These issues can happen at different scales; we can distinguish communications between processes in a distributed memory or a heterogeneous model from standard data transfer within a shared memory system, and even from transfers at the vector register scale in SIMD architectures.

Traffic contention

Traffic contention occurs when too many threads access the same memory, thus creating a bus bottleneck. This is a specific factor of high transfer latencies typical to NUMA-unaware applications.

1.2 Research Context

Tackling parallel-specific challenges requires a deep knowledge of aspects related to the program as well as the underlying architecture. First, the program's algorithm may need to be

rewritten to expose parallelism and enhance efficient memory accesses. In some cases, such rewriting can target the proper use of the parallel programming language. For instance, improper use of synchronization constructs when necessary leads to incorrect program execution (sometimes caused by data races or deadlocks). Second, the different types of systems depicted above may require different types of strategies to leverage their potential. Consequently, it can be challenging to optimize an application running on a heterogeneous cluster. Furthermore, not only should we investigate which program transformations are adequate on a given architecture, but we also sometimes need a good knowledge of certain programming languages to make the most of the hardware. A typical example is CUDA that is designed for NVIDIA GPUs; its low-level abstraction can increase one's learning curve. Therefore, efficient programming on heterogeneous architectures involves a wide spectrum of knowledge that a programmer needs to grasp.

Compilers are tools which flow is often characterized by (i) parsing a source language and translating it into an (or several) intermediate representation(s) (IR), (ii) applying various transformation techniques on the IRs to optimize the program then (iii) translating the optimized version of the code either into assembly code (traditional compilation) or a high-level program (source-to-source compilation). IR transformations to optimize the program implies that program analysis and verification are also part of the compilation process. Therefore, compilers can help alleviate a programmer's task by taking in charge part of all the code refactoring required for an efficient use of the target architecture.

Unfortunately, current compiler technologies seem not to catch up with the needs of parallel compilation. Indeed, their *intermediate representations*, that are keys to perform optimizations, were originally designed for sequential programs only. This does not restrain the possibility of compiling parallel programs since parallel constructs are translated to runtime calls. But such characteristics raise some limitations:

1. **Intermediate representations do not integrate the semantics of parallel constructs**. Therefore, it is difficult to adapt existing optimization techniques to parallel programs. This also limits the ability to apply parallel-specific techniques. For instance, how could a compiler analyze synchronization instances to avoid data races if its IR does not have any representation of synchronizations?

2. **Runtime calls impede the application of common optimizations on sequential parts in a parallel program**; with unknown side-effects of a runtime call, the compiler becomes conservative. For example, in Figure 1.4, constant propagation and dead store elimination are, as expected, applied when no MPI communications are used. However, Figure 1.5 shows that none of these optimizations have been performed due to the presence of the MPI constructs. Even compilers for parallel programs may rely on runtime calls; this is the case for the Chapel compiler that translates the source code into C or C++ with runtime calls.

Therefore,

```
1   if (taskid == 0) {
2    a = 0;
3    b = a;
4   }
5   else if (taskid == 1) {
6    a = 1;
7    b = 2;
8    printf("%d\n", b);
9   }
```

```
1   taskid.0_6 = taskid;
2   if (taskid.0_6 == 1)
3   goto <bb 3>;
4   else
5   goto <bb 4>;
6   <bb 3>:
7   # DEBUG a => 1
8   # DEBUG b => 2
9   # DEBUG __fmt => "%d\n"
10  __printf_chk (1, "%d\n", 2);
```

Figure 1.4: Original code sample without MPI communications (left) and its corresponding GCC IR (right) after applying -O2 optimizations

```
1   if (taskid == 0) {
2    a = 0;
3    b = a;
4    MPI_Recv(&a, 1, MPI_INT, 1, 0,
  ↪ MPI_COMM_WORLD, NULL);
5   }
6   else if (taskid == 1) {
7    a = 1;
8    b = 2;
9    printf("%d\n", b);
10   MPI_Send(&b, 1, MPI_INT, 0, 0,
  ↪ MPI_COMM_WORLD);
11  }
```

```
1   taskid.0_6 = taskid;
2   if (taskid.0_6 == 0)
3   goto <bb 3>;
4   else
5   goto <bb 4>;
6   <bb 3>:
7   a = 0;
8   b = 0;
9   MPI_Recv (&a, 1, 1275069445, 1, 0,
  ↪  1140850688, 0B);
10  goto <bb 6>;
11  <bb 4>:
12  if (taskid.0_6 == 1)
13  goto <bb 5>;
14  else
15  goto <bb 6>;
16  <bb 5>:
17  a = 1;
18  b = 2;
19  # DEBUG __fmt => "%d\n"
20  __printf_chk (1, "%d\n", 2);
21  MPI_Send (&b, 1, 1275069445, 0, 0,
  ↪  1140850688);
```

Figure 1.5: Original code sample with MPI communications (left) and its corresponding GCC IR (right) after applying -O2 optimizations

How do rethink intermediate representation design for parallel architectures?

1.3 Thesis Contributions and Outline

This thesis contributes to various aspects related to the intermediate representation for static parallel source-to-source compilation:

1. A survey of parallel intermediate representations;
2. The design, implementation, and formalization of TeML, a tensor optimization meta-language, in the light of research perspectives exposed by the survey.

We focus on expressiveness and functionality, thereby providing a flexible framework for suitable static optimizations, as well as prospective dynamic ones. This document is outlined as follows. Chapter 2 is a survey of contributions to parallel intermediate representations. Chapter 3 recalls characteristics of numerical applications and exposes challenges in tensor optimizations with, as an example, issues encountered in computational fluid dynamics. We then study the challenges of code transformations in the context of NUMA architectures in Chapter 4. Chapter 3 and 4 are preliminary to understand choices in the design of TeML, presented in Chapter 5. Chapter 6 presents the formal specification TeML. We then conclude this dissertation with Chapter 7, mentioning future work.

Chapter 2

Intermediate Representation for Explicitly Parallel Programs: State-of-the-art

---

Dans ce chapitre, nous étudions les représentations intermédiaires parallèles proposées ces dernières années. Contrairement à l'état-de-l'art proposé par Belwal et Surdashan [31], nous nous concentrons exclusivement sur les représentations de programmes explicitement parallèles. Nous prenons donc en compte un nombre considérable de contributions non incluses dans [31].

Les contributions sont classées en 4 catégories de représentations : les langages intermédiaires, les graphes, la forme d'assignation unique statique (SSA) et la représentation polyédrique. Ces différents types de représentations ont des rôles complémentaires dans un flot de compilation. En effet, les graphes sont utiles pour représenter, par exemples, différents attributs du flot d'un programme (par exemple le flux parallèle ou les dépendances). La forme SSA facilite l'analyse et l'optimisation du flot de données. Le modèle polyédrique quant à lui est puissant pour analyser et optimiser les boucles imbriquées. Et finalement, les langages servent de portes d'entrées pour générer ces différentes représentations.

Les contributions aux RIs parallèles sont principalement axées sur l'inclusion de la sémantique parallèle dans ces différents types de représentations. Mais ce domaine de recherche en est encore à ses débuts, ce qui laisse plusieurs perspectives de recherches encore possibles tels que la prise en compte des hiérarchies mémoires, la possibilité d'analyser les pointeurs, l'expressivité des RIs relativement à différents domaines d'applications, leur intégration concrète dans les outils de compilation, les méthodologies d'optimisations impliquant leur usage, ainsi que le rôle de la compilation dynamique.

* ♣ * ♣ *

In this chapter, we survey parallel intermediate representations (PIR) proposed in recent years in the context of static compilation. Unlike the survey proposed by Belwal and Surdashan [31], we focus exclusively on representations for explicitly parallel programs. We, therefore, take into account a considerable amount of contributions not included in [31].

Contributions are classified in 4 categories with complementary roles in a compilation flow: intermediate languages, graphs, static single assignment form, and polyhedral representation. Indeed, graphs are useful to represent different program behaviors (e.g the parallel flow or dependences), the static single assignment form eases data-flow analyses and optimization, the polyhedral model is powerful to analyze and optimize nested loops and finally, intermediate languages are gateways to generate these different representations.

In the following, we address each category with an introduction and related contributions.

2.1 Intermediate languages

Intermediate languages ease program analysis. The C language, for instance, is difficult to analyze for several reasons including its large set of constructs and the difficulty of defining its formal semantics. An intermediate language such as CIL [97] is more convenient as it is an analyzable subset of C with fewer keywords. Another interest is to serve as a common intermediate language for compilers that support multiple languages. We can mention PIPS IR [10], designed for C and Fortran programs.

In the context of parallel programs, several intermediate languages have been proposed at different levels of abstraction. The main interest in designing such a language is to capture parallel semantics including expressing parallel loops, describing variables properties (e.g., shared or private) or abstract synchronizations.

Selected contributions in the literature

Erbium Miranda et al. [93, 92] proposed Erbium, an intermediate language specifically designed to support classic or platform-specific optimizations for streaming applications. Erbium features data structures and functions for the representation of data streaming and resource management.

Threads creation are abstracted as *processes* communicating and synchronizing through *event records*. Records abstract FIFO channels with read or write *views*, that is, unbounded streams addressable through non-negative indexes. Live elements of read or write views are stored in a *sliding window*. The size of such a window is a view's *horizon*.

Several primitives are useful for the management of resources: *commit*, *update*, *release* and *stall*. The following producer-consumer flow illustrates their usage. On a producer's side, a view is on read and write access until the producer *commits* it; at this point, the view becomes read-only and is made available to the consumer process. However, unless the consumer *updates* its view with the corresponding index, it will not actually consume the view. After being consumed, a view can be *released* and the storage location of the released view can be reused by a producer when using the *stall* primitive.

Erbium has been implemented in an experimental branch of GCC 4.3.

GCC IR In order to adapt GCC optimizations to parallel programs, one could think of re-implementing optimization passes. However, Pop and Cohen [104] attempt to tackle the lack of parallel semantics in GCC by preserving both parallel semantics and optimizations passes implementation. For this purpose, they propose a set of annotations.

Dataflow annotations abstract informations about accessed variables within a block such as read, write or reduced variables. *Control-flow annotations* abstract barriers and synchronizations (e.g, execution by a single thread, barrier, synchronization point, memory flush). Finally, *user hints* store any information provided by the programmer, including the schedule of a loop or the number of threads. These annotations characterize a parallel region through calls to factitious built-in functions in conditional statements.

Pragma-based programming languages such as OpenMP or HMPP could benefit from such abstractions.

PLASMA IR PLASMA [99] is a programming framework for writing portable SIMD programs. Supporting multiple types of high-level programming languages, its main component is the intermediate representation (PLASMA IR). Abstractions are provided for programs to be compiled to different SIMD architectures including accelerators or processors with SIMD extension.

The IR has four main categories of constructs: operators, vectors, distributors and vector compositions. *Operators* act on either primitive types (e.g add, multiply elements) or aggregations of primitive types called blocks (e.g permute blocks, retrieve maximum or minimum). *Vectors* are blocks with a length attribute. *Distributors* are parallel operations performed on vectors (e.g parallel addition of vectors, reduction of elements in a vector). Finally, *vector composition* allow the encoding of operations such as concatenation, slicing or gathering.

The input source file is a PLASMA source, written independently of any SIMD device and the generated code targets CPUs and NVIDIA GPUs. General loop transformations (e.g, loop fusion, tiling [138, 75]), communication optimizations or more specific GPU optimizations (e.g, memory coalescing [59]) can be performed within the IR.

PENCIL Baghdadi et al. designed PENCIL [24, 25], a portable intermediate language to ease highly optimized GPU code generation, lowered from domain-specific languages (DSLs). PENCIL extends a subset of C99 with specific constructs including directives, function attributes, and built-in functions. Pointer manipulation as arguments of functions is not allowed, as well as recursive function calls. Furthermore, a `for` loop must have a single iterator, non-changing start and stop values within the loop and a constant increment.

The main constructs of PENCIL are:

- the *assume predicate* which guarantees that a boolean condition `e` is held whenever the control flow reaches the predicate,
- the *independant directive* which is a loop annotation indicating that a loop has no dependencies,

- *summary functions* that describe memory access patterns of functions in order to analyze memory footprints
- the *kill statement* that signals that a variable or an array element is dead at the point where the statement is called.

PENCIL is used in a polyhedral compilation flow that generates OpenCL code, using PPCG [137], for programs written in VOBLA [30], a DSL for linear algebra.

PIR Zhao and Sarkar [141] designed PIR for Habanero-Java [44]. Constructed from an abstract syntax tree (AST), PIR is a three-level IR. The high-level PIR (HPIR) is a tree, the Region Structure Tree, based on the syntax of Habanero-Java constructs (`finish, async, isolated, foreach, forall`) and more general ones such as `for` and `while`. When reaching the middle-level (MPIR) parallel constructs are lowered to `async, finish` and `isolated`; constructs that cannot be directly expressed become a combination of these, for instance `foreach` is translated to a `for` with an `async` body. Finally the low-level (LPIR) is alike a sequential flat IR where parallel constructs are expressed as runtime APIs.

The May Happen in Parallel (MHP) analysis [21] or parallel loop chunking in the presence of synchronizations [118] can be performed at the HPIR, whereas other analysis such as load elimination or data race detection can rather be performed at the MPIR.

SPIRE Unlike other IRs, Khaldi et al. [77] proposed a methodology to extend sequential IRs with parallel semantics. Named SPIRE (Sequential to Parallel Intermediate Representation Extension), it introduces a few key constructs that abstract execution, synchronization, data distribution, communications, and the memory model.

Data and task parallelism are respectively represented with the `parallel` and `spawn` construct. Reductions can be specified using `reduced`. Collective synchronizations are abstracted using `barrier`, whereas `atomic` is used for mutual exclusions. The `single` construct defines the sequential execution of statements within a spawned section. Furthermore, memory information can be specified for variables using `private`, `shared` or `pgas`. Communications and point-to-point synchronization are handled with intrinsic functions: `send(), recv(), signal()` and `wait()`.

A proof of concept is demonstrated with the generation of OpenMP task parallelism in PIPS [76] and the optimization of OpenSHMEM communications in LLVM [78].

INSPIRE INSPIRE [73] is the parallel IR of the Insieme[1] compiler infrastructure.

The parallel execution of instructions is modeled as a *job* cooperatively executed by a thread group. `pfor` is the main construct for work distribution. Data distribution within a group is performed using `redistribute`. Point-to-point communications can be specified using `channels`. In addition, several built-in functions are used for parallelism management.

[1]Insieme Compiler project from the University of Innsbruck: www.insieme-compiler.org

The functions `getThreadID()` and `getNumThreads()` respectively return the thread identification and the total number of threads. Threads are allowed to create sub-thread groups with `spawn()`, which can be merged using `merge()` or `mergeAll()`.

Several contributions demonstrate the usability of INSPIRE; Insieme's runtime system [131], a framework for the implementation and optimization of MPI programs based on the Insieme compiler and runtime system [100].

2.2 Program Representations Using Graphs

As shown in the survey of Stanier and Watson [126], graphs are popular in modern compilers. One of the most common is the control flow graph (CFG) [23]. The CFG represents all possible paths of a program during its execution. Formally, it is a graph $CFG = (V, E, Entry, Exit)$ where:

- V is a set of nodes representing *basic blocks* in the program;
- E is a set of edges representing sequential control flow in the program;
- and $Entry$ and $Exit$ are respectively nodes representing the entry and exit points of the program.

Basic blocks are sequences of instructions with no jumps, nor conditional branching. Conditional branches mark the end of a basic block with two outgoing edges representing the $true$ of $false$ paths. CFGs are useful for control- and data-flow analyses (e.g reaching definitions, available expression, liveness analysis) and optimizations (e.g, unreachable code elimination, common sub-expression elimination, copy or constant propagation) [22].

However, classic CFG properties are not sufficient for parallel programs. Indeed, the flow of parallel programs involves in addition concurrency, synchronization, mutual exclusion and sometimes, explicit communications. The wide implementation of the CFG in compilers and their unsuitability for parallel programs is the main motivations for contributions to provide adequate CFGs for parallel programs.

Another main representation is the program dependence graph (PDG) [64]. It is defined as a graph $PDG = (V, E)$ where:

- V are nodes representing instructions;
- E either represent control or data dependences within the program.

Useful for dependence analyses, it can be applied to parallel programs if the parallel flow is properly captured. Figure 2.1 is an example of a program and its corresponding CFG and PDG.

Selected contributions in the literature

Extended flow graph Srinivasan and Wolfe [125] proposed the extended flow graph (EFG), a hierarchical representation combining parallel control flow graphs (PCFG) and parallel precedence graphs (PPG).

```
1  y = x + 7;
2  if (y < 5)
3   a = a + 7;
4  else
5   a = a - 7;
6  y += a;
```

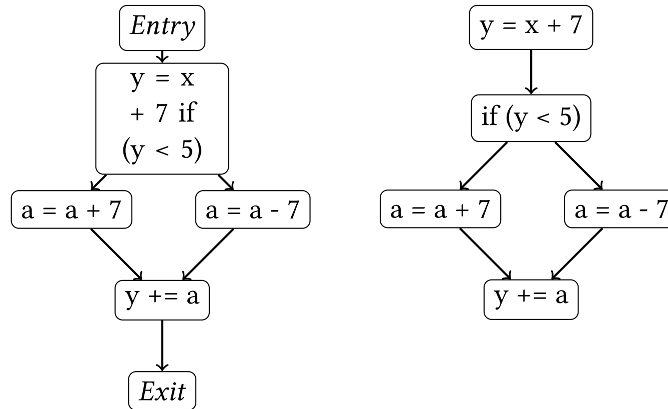


Figure 2.1: A program, its CFG (mid) and its PDG (left)

The PCFG is a graph $G = (V_G, E_G, Entry_G, Exit_G)$ where:

- V_G is a set of basic block nodes or parallel block (called *supernode*);
- E_G is a set of edges representing potential flow of control in the program;
- $Entry_G$ is the unique start node;
- $Exit_G$ is the unique exit node.

Within a supernode, the parallel execution is represented by the PPG. The PPG is a graph $P = (V_P, E_P, Entry_P, Exit_P)$ where:

- V_P is a set of sections in a parallel block;
- E_P is a set of *wait-dependence* arcs;
- $Entry_P$ is the co-begin node, where co-begin denotes the beginning of a parallel section;
- $Exit_P$ the co-end node, denoting the end of the parallel section.

Finally, each section in a parallel block is also represented by a PCFG $S = (V_S, E_S, Entry_S, Exit_S)$ where $Entry_S$ and $Exit_S$ respectively represent the entry into and the exit from the section.

The PCFG comes along with a parallel SSA form presented in the next section.

Parallel program graph Sarkar and Simons [116] proposed the parallel program graph (PPG), a graph $PPG = (N, E_{cont}, E_{sync}, TYPE)$ where:

- N is a set of node representing arbitrary sequential computation;
- E_{cont} are control edges;
- E_{sync} are synchronization edges;
- $TYPE$ is a node type mapping.

Nodes can be identified as: *START* (beginning of the program), *PREDICATE* (conditional statements, with two outgoing edges either *true* or *false*), *COMPUTE* (nodes with only one outgoing edge) and *MGOTO* (creation of a parallel section where each successor of the *MGOTO* node is a thread).

Authors demonstrate how the reaching definition analysis may be applied using the PPG [115]. However, such representation is restricted to deterministic parallel programs.

Concurrent control flow graph Lee et al. [84] proposed the concurrent control flow graph (CCFG). They introduce the notion of concurrent basic blocks (CBB) to incorporate thread interaction. A CBB has the fundamental definition of a basic block in a sequential program, but with three additional properties involving parallel program constructs: (*i*) only the *first* statement can be a **`wait`** or contain a use of a conflicting variable, (*ii*) only the *last* statement can be a **`post`** or contain a definition of a conflicting variable and (*iii*) if a CBB contains a parallel execution, a mutual exclusion construct or **`barrier`** statement, then that statement is the only one in the concurrent basic block except **`end parallel do`**.

A node that contains an **`end parallel do`** contains an assignment to the loop index variable.

Based on this, a CCFG is a directed graph $G = (N, E, Ntype, Etype)$ where:

- N is the set of nodes representing basic blocks;
- E is the set of edges composed of control flow edges, synchronization edges and conflict edges;
- $Ntype$ is a function telling the class of nodes;
- $Etype$, a function telling the type of edges.

The CCFG comes along with the concurrent SSA form presented in the next section.

OpenMP control flow graph Lin [85] introduced the OpenMP CFG (OMPCFG) to model control flow in OpenMP programs. The graphs is composed of: *basic nodes* for basic blocks and *directive nodes* for OpenMP directive blocks. *Entry* and *Exit* nodes are respectively the unique entry and exit points of the graph. The OMPCFG may serve for concurrency analysis and static data race detection.

SPIR Choi et al. [51] proposed the Signal Processing Intermediate Representation (SPIR), a dataflow graph for efficient mapping of streaming applications. It is graph $G = (N, E)$ where:

- N is a set of nodes corresponding to either a task or the condition of an **`if`** statement;
- E are edges representing data transfer between tasks. Edges following **`if`** conditions are annotated with *true* or *false*.

A possible application is task mapping onto processors maximizing throughput under memory constraints.

Kimble IR Benoit and Louise [33] introduced the Kimble IR, meant to support explicit parallelism and perform automatic parallelization. It is a direct acyclic graph $G = (N, D, E)$ where:

- N is a set of nodes corresponding to program constructs: function, loop, region, cluster (a sequence of dependent statements), statement, guard or function call;
- D is a set of ordered pairs of nodes denoting a dependency between two nodes;
- E is a set of ordered pairs of nodes expressing a hierarchical relationship.

Gomet [32], an extension of GCC to support multi-grain parallelism on MPSoCs uses Kimble IR. Program transformations such as reordering, function inlining and outlining can be performed within the IR. In [34], they describe how it is used to determine the execution cost on a given processor.

MPI control flow graphs The MPI-CFG [119] extends the CFD with *communication nodes*, which are separate basic blocks expressing communication statements, and edges denoting data transfer. Edges are also annotated with a value providing pieces of information about process IDs whenever possible. Additional information may be provided if such ID is unknown. An unknown process proven to be the only one executing a communication is annotated *single*. If one or more processes may be involved, *unknown* is the annotation. Finally, *multiple* may be used if it is guaranteed that more than one process executes the communication.

Point-to-point communications are typically abstracted using single communication edges. But specific representations have been provided for each type of collective communications such as broadcasting or scattering in order to reflect their unique semantics.

Strout et al. [129] demonstrate the MPI-CFG could be useful for static analysis. Furthermore, they propose the MPI-ICFG for inter-procedural MPI calls, obtained through an inter-procedural CFG [83] enhanced with communication edges.

Delite IR Delite [42] is a framework for the compilation of domain-specific languages to heterogeneous targets. Its IR is three-level.

The lowest level, the *generic* IR, is a composed of a collection of nodes only linked if their exist read-after-read or control dependencies; the nodes are free otherwise, hence named *sea of nodes*. The mid-level is the *parallel* IR, extending the generic IR to express particular parallelism patterns. For instance, a node with the label *sequential op* indicates that the node is executed sequentially and another one with a *reduce op* label indicates are reductions. Finally, since Delite's input programs can be written in various DSLs, the highest level is the *domain-specific* IR.

Different types of optimizations are performed depending on the level of IR. Domain-specific and parallel-specific optimization are respectively applied in the high-level and mid-level IR. Classic optimizations are mainly performed in the low-level IR.

Tapir Tapir [117] extended LLVM IR with fork-join parallelism. Using three instructions, that is, **`detach`**, **`reattach`** and **`sync`**, it extends the CFG to represent parallel task and synchronizations. Parallel loops are represented as spawned blocks into a number of tasks corresponding to the number of loop iterations.

A certain number of LLVM analyses and optimization passes have been adapted to Tapir including alias, data-flow analyses, and classic optimization passes.

2.3 The Static Single Assignment Form

The static single assignment (SSA) form [56] is a popular compilation technique that consists in assigning variables only once. That is, if a variable v is assigned 3 times, in the SSA form, each assignment will be uniquely identified as v_i with $i = 1..3$. When necessary to deal with control flow merges due to multiple possible paths (e.g after a **`if-then-else`** section), a ϕ-function is used to abstract merging points. For example, the SSA form of the code sample in Figure 2.1 is:

```
1  y1 = x1 + 7;
2  if (y1 < 5)
3   a2 = a1 + 7;
4  else
5   a3 = a1 - 7;
6  y2 = φ(a2, a3);
```

The SSA form is often combined with the CFG as it eases the identification of data dependencies and thus enables more efficient data-flow optimization techniques. The SSA form cannot accurately be applied on array computations within loops. More accurate representations, *array* SSA forms, therefore exist [82, 114].

However, applying the SSA form in a parallel program exhibits issues: How do we handle merging points at the end of a parallel section? How do we handle the concurrent update of a shared variable within a task? Indeed, parallel sections may include writes by multiple threads on the same variable. As the order of thread execution may be non-deterministic, merging points at the end of such sections are necessary.

Selected contributions in the literature

Srinivasan et al. [124] pioneered parallel SSA (PSSA) forms. Their PSSA form features, in addition to the ϕ-function, a ψ-function denoting parallel merge nodes. Nascent [128], a Fortran parallelizing compiler, uses the PSSA form for optimizations forward substitution and constant propagation in parallel programs. However, the PSSA form is limited to the PCF Fortran standard, with a weak memory consistency model.

For better generalization, Lee et al. [84] also proposed the concurrent SSA form (CSSA) similar to PSSA form but also capturing the update of a shared variable within a concurrent task using π-functions. They redefined a larger set of compiler analysis and optimizations

including copy propagation, dead code elimination, common sub-expression elimination, and redundant load/store elimination.

As the CSSA form focuses only on event-based synchronizations, Novillo et al. [98] proposed the CSSAME framework based on an extended CSSA form [84] to deal with mutual exclusion. The CSSAME framework does not introduce any new term; they incorporate the semantics of mutual exclusion synchronizations to be taken into account when building the CSSA form. They show how constant propagation, dead code elimination, and lock independent code motion can be performed in the presence of mutual exclusion.

Collard [55] aimed at representing accurately the array SSA form in the presence of concurrency, event-based synchronization or mutual exclusion. The idea is to clearly define the order of memory updates then applies array SSA form accordingly.

Chakrabarti and Banerjee [45] proposed an array SSA form in the context of the automatic generation of message-passing programs from a program similar to High-Performance Fortran standards. This work aims at maintaining accurate information about distributed data in such a context. Therefore, they not only distinguish merging at dominance frontier (through the Φ_d-term) and after each non-killing write of an array (through the Φ_w-term) but also before any read that occurs on a distributed array variable (through the Φ_r-term). The array SSA form in an automatic parallelizing compiler named PARADIGM in order to perform data placement, generation and optimizations of data transfers for MPI final code generation.

2.4 The Polyhedral Model

The polyhedral model [63] is a mathematical representation of a subset of imperative languages called static control programs (SCoP). A SCoP is a maximal set of consecutive instructions where the loop bounds (`for` loops), conditions (`if` statements) and arrays (excluding pointers) are affine functions depending only on outer loop indexes (e.g, i, j) and constant parameters (e.g, n). Each statement S can be associated to an iteration vector $x \in \mathbb{Z}^p$, where the ith element of the vector is the loop index. The set of possible values for the iteration vector is called the *iteration domain*. The iteration domain can be therefore specified by a set of linear inequalities defining an integer polyhedron:

$$D_S = \{\vec{x} \in \mathbb{Z}^p \mid A\vec{x} + \vec{c} > 0\} \tag{2.1}$$

where $\vec{x}$ is the iteration vector, A is a constant matrix and $\vec{c}$ is a constant vector. Figure 2.2 illustrates the representation of a program in the polyhedral model.

Another characteristic of a SCoP is the *scheduling function* that describes the logical date of execution for each statement instance. For instance, given the following code sample where S_1, S_2 and S_3 are instructions

```
1  do i = 1, n
2   do j = 1, n
3    if (i <= n+2-j)
4     S1;
```

$$\begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ -1 & -1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 1 \end{pmatrix} (n) + \begin{pmatrix} -1 \\ 0 \\ -1 \\ 0 \\ 2 \end{pmatrix} \geq 0$$

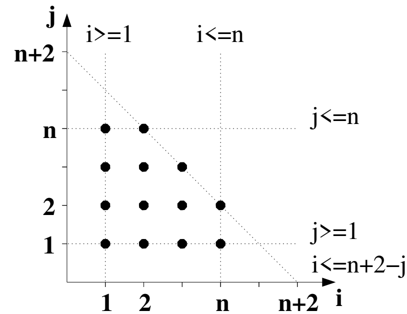


Figure 2.2: A loop, its corresponding iteration domain (bottom left) and polyhedral representation (right).

```
1  for (i = 0; i < N; i++)
2   S1;
3  for (j = 0; j < N ; j++)
4   S2;
5   S3;
```

the corresponding scheduling function is:

$$S_1(i) = (i, 0)$$
$$S_2(i,j) = (i, 1, j, 0)$$
$$S_3(i,j) = (i, 1, j, 1)$$

The polyhedral model enables exhaustive analysis of nested loops. Several state-of-the-art tools are based on polyhedral compilation techniques: Clan [16] or Pet [134] as extractors of the polyhedral representation of a program; isl [135] or PolyLib [87] as libraries for manipulating polyhedra; CLooG [27] as a polyhedral-to-C code generator; PPCG [137] as C-to-CUDA code generator.

Several compilers have integrated polyhedral compilation: PIPS [10], GCC through Graphite [106], LLVM through Polly [66], IBM XL through PluTo [38] and a R-stream [91].

Consequently, advances in parallelism support impact both research and industrial compilers.

Selected contributions in the literature

Contributions in the polyhedral model exhibit two approaches. Most researches attempt to identify at least subsets of parallel programs compatible with the polyhedral representation. However, extending the representation has also been addressed.

Basupalli et al. [28] integrated an OpenMP program verifier, OmpVerify, in the Eclipse IDE for data race detection. To perform such an analysis, they first extract the initial de-

pendences of the program (ignoring the parallel constructs), then they consider the parallel construct `omp for` as a program transformation that assigns new time-stamps to instances in the program. Finally, the task of the verifier is to ensure that this transformation does not introduce any data race. However, some dynamic concurrent accesses may be difficult to detect.

Liveness analysis for register allocation requires the computation of a set of conflicting variables in a live range. Such computation is based on the notion of total order among each iteration, which is not applicable to parallel programs. To put liveness analysis to work with parallel programs, Darte et al. [57] proposed a method where conflicting variables are computed based on the notion of partial order *and* the happens-before relationship that can be computed with Presburger sets and relations [136].

The Insieme compiler also features the translation from INSPIRE [73] to polyhedral representation. Therefore, Pellegrino et al. [101] developed a polyhedral-based approach to exact dependence analysis of MPI communications. They define, for MPI point-to-point and collective communications, semantically equivalent loops that fit in the polyhedral model. Communications must be first rewritten in a *normal form*, in which an MPI program only contains `MPI_Send` and `MPI_Recv`. Then a data dependence graph is generated in order to apply a set a transformation including loop fission or code motion.

In order to perform static data race detection in X10 programs, Yuki et al. [140] adapted the array dataflow analysis to a subset of the language, fitting in the polyhedral model. The subset includes sequences of instructions, sequential `for` loops, the parallel activation construct `async` and the termination construct `finish`. Similarly to [57] for liveness analysis, the array dataflow analysis is based on the happens-before relationship of instances.

Cohen et al. [52] studied polyhedral techniques applied to OpenStream [105], a stream-programming extension of OpenMP, mainly for static analyses such as dependence analysis and deadlock detection. The main idea is to exhibit the properties of OpenStream programs that can be represented in polyhedral analysis.

Unlike previous approaches, Chatarasi et al. [47] proposed extensions to the polyhedral model. They introduce the notion of *space mapping*, abstracting the multi-threaded loops and *phase mapping* to distinguish different parallel phases (e.g. separated by barriers). They demonstrate its usefulness for static data race detection.

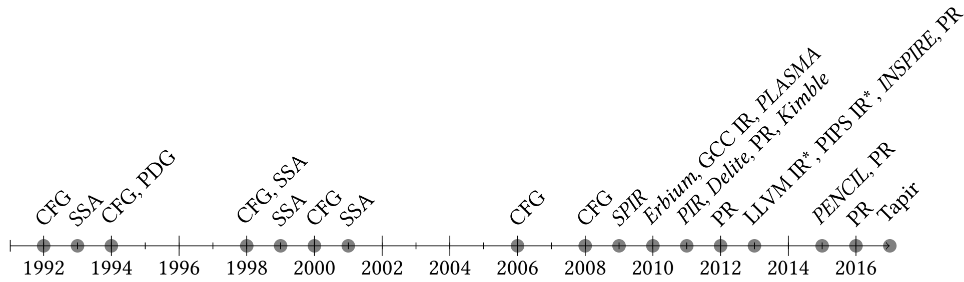


Figure 2.3: Timeline of attempts of revisiting existing IRs or designing new ones. IRs marked with * are those that were extended with SPIRE. New IRs are in italic. PR stands for polyhedral representation

2.5 Discussion

Using further classifications, we may observe different trends in terms of language support, code generation, memory model or concrete application in compilers. Table 2.1 summarizes the different contributions, their year of appearance and the characteristics of the type of parallel programs supported.

2.5.1 Observations

Timeline. Figure 2.3 exhibits two eras of parallel IR proposals; flow graphs and SSA forms prior to 2008, then from 2010 a variety of intermediate languages and extension of research compiler IRs.

Language support Most contributions target only one parallel language or a restricted set. PLASMA, SPIRE and INSPIRE are exception attempting to support a wider range of languages.

Levels of abstractions for intermediate languages We may generally consider that any contribution targeting GCC IR, LLVM IR or common graph representation are general-purpose. However, Erbium, while being implemented in GCC, remains specific to streaming applications. On the other hand, PLASMA IR is architecture-specific.

Erbium and PENCIL, unlike other languages, may be used as user programming languages.

IRs structure PIR and Delite IR distinguished themselves with their multi-level IRs enabling various types of optimizations. With the complementarity of the CFG and SSA form, contributions of Srinivasan et al. [125, 124], and Lee et al. [84] can also be considered as multi-level IRs.

Table 2.1: Year of contributions and characteristics of their supported parallel programs

Contributions	Characteristics of supported parallel programs	Year
	Intermediate languages	
[93] (Erbium)	Streaming applications	2010
[99] (PLASMA IR)	SIMD parallelism	2010
Pop, Cohen [105]	Directive-based	2011
[141] (PIR)	Habanero-Java	2011
[77] (SPIRE)	Multithreaded, distributed, PGAS	2013
[73] (INSPIRE)	Multithreaded, distributed, with accelerators	2013
[24] (PENCIL)	DSLs	2015
[117] (Tapir)	Multithreaded	2017
	Graphs	
[125] (EFG)	PCF Fortran	1992
[116] (PPG)	Deterministic multithreaded	1994
[84] (CCFG)	Multithreaded	1998
[119] (MPI-CFG)	MPI	2000
[129] (MPI-ICFG)	MPI	2006
[85] (OMPCFG)	OpenMP	2008
[51] (SPIR)	Streaming applications	2009
[32] (Kimble IR)	Multithreaded	2010
[42] (Delite IR)	DSLs	2011
	SSA form	
[124] (PSSA)	PCF Fortran	1993
[84] (CSSA)	Multithreaded with event-based synchronizations	1998
[98] (CSSAME)	Multithreaded with synchronizations and mutual exclusion	1998
[55]	Multithreaded with synchronizations and mutual exclusion	1999
[45]	HPF-like	2001
	Polyhedral model	
[28]	OpenMP	2011
[101]	MPI	2012
[140]	X10	2013
[46]	Multithreaded	2015
[47]	Multithreaded	2016
[57]	Multithreaded	2016
[52]	OpenStream [105]	2016

Table 2.2: Intermediate languages and some examples of their applications for language support, code generation and integration in compilers

Intermediate languages	Language support	Code generation	Compilers
[93] (Erbium)	OpenStream [105]	-	GCC [92]
Pop and Cohen [105]	OpenMP, OpenACC	-	GCC
[99] (PLASMA IR)	OpenMP, Matlab, StreamIT	CUDA	PLASMA
[24] (PENCIL)	VOBLA [30]	OpenCL, CUDA	DSL compilers
[77] (SPIRE)	OpenMP, MPI, Chapel, OpenSHMEM	OpenMP	LLVM [78], PIPS [76]
[141] (PIR)	Habanero-Java [44]	-	Habanero-Java compiler
[73] (INSPIRE)	OpenMP, MPI, Cilk, OpenCL	OpenMP, MPI	Insieme [131, 100]

Table 2.3: IRs and their memory models scope

Contributions	Shared	Distributed	Host-accelerator	PGAS
[93] (Erbium)	✓			
Pop and Cohen [105]	✓		✓	
[99] (PLASMA IR)	✓	✓	✓	
[24] (PENCIL)			✓	
[77] (SPIRE)	✓	✓		✓
[73] (INSPIRE)	✓	✓	✓	
[117] (Tapir)	✓			
[125, 124] (EFG + PSSA)	✓			
[84] (CCFG + CSSA)	✓			
[116] (PPG)	✓			
[85] (OMPCFG)	✓			
[51] (SPIR)	✓			
[32] (Kimble IR)	✓			
[119] (MPI-CFG)		✓		
[129] (MPI-ICFG)		✓		
[141] (PIR)				✓
[42] (Delite IR)			✓	
[98] (CSSAME)	✓			
Collard [55]	✓			
Chakrabarti and Banerjee [45]		✓		
Basupalli et al. [28]	✓			
Chatarasi et al. [47]	✓			
Darte et al. [57]	✓			
Pellegrini et al. [101]		✓		
Yuki et al. [140]	✓			
Cohen et al. [52]	✓			

Table 2.4: Examples of applications for analyses/optimizations of graph-based IRs, parallel SSA forms and polyhedral model.

Contributions	Analyses	Optimizations
[125] (EFG + PSSA)	N/A	Constant propagation [128]
[84] (CCFG + CSSA)	Data-flow analyses	Redundant load/store elimination
[116] (PPG)	Reaching definitions analysis [115]	N/A
[85] (OMPCFG)	Data race detection	N/A
[51] (SPIR)	Processor assignment	Task scheduling
[32] (Kimble IR)	Execution cost [34]	Function inlining/outlining
[119] (MPI-CFG)	N/A	N/A
[129] (MPI-ICFG)	Activity analysis	N/A
[141] (PIR)	May-happen-in-parallel analysis	Dead code elimination
[42] (Delite IR)	Data-flow analyses	Pattern matching, domain-specific
[98] (CSSAME)	Detection of mutual exclusion	Code motion
Collard [55]	N/A	N/A
Chakrabarti and Banerjee [45]	Communication placement	Communication optimization
Basupalli et al. [28]	Data race detection	N/A
Chatarasi et al. [46, 47]	Data race detection	N/A
Darte et al. [57]	Liveness analysis	N/A
Pellegrini et al. [101]	Dependence analysis	Communication optimization
Yuki et al. [140]	Array data-flow analysis	N/A
Cohen et al. [52]	Deadlock detection	N/A

Memory models Table 2.3 summarizes memory models scopes. Most IRs apply to the shared memory model. Very few contributions concern distributed and host-accelerator memory models. This could be surprising for the distributed model in particular as it has been exploited for a very long time. However, the fact that distributed memory systems involve a more restricted set of analyses and transformations could explain the limited amount of contributions. Indeed, the main concern in distributed computing is data transfer. The PGAS model is addressed only twice.

Analyses and optimizations Table 2.4 gives examples of analyses and optimizations that were applied for parallel graphs, SSA forms, and the polyhedral model. Classic data-flow analyses and optimizations are often targeted. As for parallel-specific ones, data race detection is recurrent, mostly in the polyhedral model.

Type of compilation flow We may assume that contributions prior to 2008 could be applicable either in a source-to-source or a source-to-binary compilation flow. Polyhedral optimization is rather in the context of source-to-source. Erbium and Pop and Cohen's contribution [105] are more in a source-to-binary context. SPIRE has been demonstrated in both contexts.

Concrete implementation in compilers Very few IRs have been concretely implemented in an available compiler. Graphs and SSA forms applications seem very isolated. After 2008, GCC and LLVM were the most targeted compilers but Tapir seems to be the most

viable contribution. PIR and INSPIRE were respectively designed for the Habanero-Java [44] compiler and the Insieme[2] compiler.

2.5.2 Perspectives

In the context of static compilation, the body of research in parallel IRs mainly focused on including parallel semantics in the different type of representations. This has been useful especially for parallel-specific analyses and transformations. The adaptation of classic optimization passes is also a step forward in the suitability of common compilers for parallel programs. But this research area is still at early stages. Future research directions are therefore possible.

Memory hierarchy It is clear that the efficiency of parallel programs depends on the underlying architecture. More specifically, the memory hierarchy and data placements is a key factor of performance. Yet, very few contributions seem to take into account the target architecture to some extent, especially for SIMD architectures. Furthermore, despite the considerable amount of proposed IRs in the shared memory model, no contribution seem to address challenges related to NUMA architectures. Complementary to memory placement, data layouts also considerably impact performance. However, no contribution takes this aspect into account.

Pointer analysis Most popular parallel programming languages are based on the C language. This makes inevitable the question of pointer analysis which is not addressed by any of the proposed IRs. This is not surprising as, in the sequential context, pointer analysis is already known to be complex due to the difficult understanding of the semantics of the C language. Note however that many industrial applications that need to be parallelized include array traversal based on pointer arithmetics. Therefore, automatic parallelization for such applications would require pointer analysis. Current automatic parallelizer do not feature such analyses and explicitly forbid the use of pointers in input programs. Consequently, before even addressing this aspect in parallel compilation specifically, pointer analysis still needs to mature in current compilers.

Expressiveness Besides performance factors, the level of expressiveness also matters. Most contributions have a general-purpose application scope. However, the variety of applications makes it impossible to rely only on general-purpose solutions. Domain-specificities are therefore mandatory to complement existing approaches. Narrower scopes for IRs started to emerge with Erbium, PENCIL, and Delite. PENCIL (coupled with VOBLA and PPCG) and Delite are particularly interesting as their compilation flow clearly exhibit domain-specificities along with general-purpose optimizations. Further investigations in

[2] Insieme Compiler project from the University of Innsbruck: www.insieme-compiler.org

this direction would be interesting for other types of application such as stencil computations.

Applicability As shown in the survey, very few contributions were actually applied in real compilers. While this may look disappointing, several aspects explain the difficulty of application. Compilers such as GCC and LLVM have been in active development for years; GCC since 1987 and LLVM since 2003. Extending their internal representations would, therefore, require major, and probably difficult, refactoring. While SPIRE and Tapir are proofs that such a task is not impossible, a lot of time is still necessary to (i) understand how classic optimization passes extend with parallel specificities and (ii) implement these extensions. Indeed, more complex memory models will probably be required to deal with the heterogeneity of upcoming architectures; contributions prior to 2008 may need to be revisited in light of modern architectures. Though, the INSIEME and Habanero Java compilers also demonstrate that another way is considering dedicated parallel compilation.

Optimization methodology One last aspect that can be discussed is the general approach to optimizations. All proposed contributions are in the context of a classic compilation approach, i.e., fully automated with predefined heuristics. However, with the increasing complexity of architecture, compilers become more and more limited in finding efficient transformations. Consequently, other compilation approaches become appealing. For instance, there is *autotuning*, that is, the dynamic search for efficient program variants. Note also that hand-optimization by experts is an alternative. IRs designed with respect to such need are therefore required. Among the contributions in this survey, PENCIL and Erbium seem to provide, to some extent, facilities for this purpose.

Dynamic compilers Even though our focus is static compilation (and parallel IRs have been mainly proposed in this context), the inherent non-deterministic nature of parallelism at runtime suggests that dynamic compilation should play an important role in parallel compilation. Automatic parallelization based on dynamic compilation already exists (e.g, thread-level speculation [74, 72, 43, 86]), in which other types of program information, not accessible from static analyses, are manipulated. For instance, pieces of information about memory accesses enable pointer analysis. Further studies on parallel IRs for dynamic compilers, worth investigating, will probably expose differences between approaches.

Chapter 3

The Tensor Challenge

Un large éventail de domaines tels que le machine learning, la vision par ordinateur, le traitement graphique ou du signal utilisent des algorithmes basés sur des principes d'algèbre linéaire et tensoriel. Malgré leurs spécificités respectives, ces principes impliquent des calculs fondamentaux sur des vecteurs, des matrices et des tenseurs. Par conséquent, les applications numériques provenant de différents domaines partagent des noyaux de calculs communs (par exemple, une multiplication matrice-matrice ou matrice-vecteur). En programmation, les noyaux d'algèbre linéaire et tensorielle sont exprimés sous forme de calculs sur des tableaux $N-$dimensions dans des nids de boucles.

Selon le type de données manipulé ou le type d'opération, les nids de boucles peuvent nécessiter beaucoup de calculs. Les optimisations visant à réduire les temps d'exécution de ces types de programmes englobent différents axes de techniques de compilation, y compris les transformations de boucles, de disposition des données ou transformations algébriques.

Nous avons eu l'occasion d'étudier des problèmes spécifiques à des applications en dynamique des fluides pour lesquelles des transformations algébriques et de boucles sont nécessaires. Nous aurions pu compter uniquement sur le compilateur ICC pour compiler efficacement ces programmes. Pourtant, une étude avec ICC montre qu'il peut y avoir des heuristiques intéressantes qui ne sont pas choisies par le compilateur. Cela soulève le besoin d'avoir plus de flexibilité dans les choix d'optimisation ; avoir la possibilité d'appliquer différentes heuristiques de transformation est plus attrayante.

Bien qu'il existe plusieurs frameworks pour l'optimisation d'applications tensorielles, ceux-ci semblent ne pas être aisément utilisable dans une variété de domaines. Ceci rend donc difficile leur usage pour optimiser certaines applications en dynamique des fluides.

∗ ♣ ∗ ♣ ∗

We explore the research perspectives mentioned in Chapter 2 in the context of *numerical applications*. This chapter introduces numerical applications and transformation techniques often used for their optimization. We then focus on challenges encountered in the domain of computational fluid dynamics to expose limitations in current relevant optimizing frameworks.

This chapter is based on:

Adilla Susungi, Norman A. Rink, Jerónimo Castrillón, Immo Huismann, Albert Cohen, Claude Tadonki, Jörg Stiller, and Jochen Fröhlich. ***Towards compositional and generative tensor optimizations***. In Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2017). DOI: https://doi.org/10.1145/3136040.3136050

3.1 Numerical Applications

A wide range of domains such as *machine learning*, *computer vision*, *signal* or *graphics processing* use mathematical algorithms based on linear and tensor algebra principles. Despite their respective domain specificities, such applications involve fundamental computations with vectors, matrices, and tensors, thereby sharing common kernels (e.g., matrix-matrix or matrix-vector multiplication). In programming, linear and tensor algebra kernels are abstracted as computations over $N-$dimensional arrays within loop nests.

Depending on the data manipulated, or the operation involved, loop nests can be compute-intensive. Optimizations to reduce runtime execution for this class of programs encompass different axes of compilation techniques including (but not limited to) *loop*, *data layout* or *algebraic transformations*.

Loop Transformations

A set of loop transformations techniques [75] exist, among which we can cite popular ones and their impact.

Loop interchange swaps dimensions in a loop nest. Given two loop nests, *loop fusion* merges adjacent dimensions into a single dimension. Conversely, *loop distribution* split statements of the same dimension into two adjacent dimensions. *Loop unrolling* unwinds several instructions. *Loop peeling* isolates first or last iterations. *Loop tiling* divides the iteration space into smaller blocks.

Minimizing memory accesses is an important concern in optimizing numerical applications. An example of solution is promoting cache reuse using techniques such as interchange, fusion, distribution or tiling. Loop unrolling eliminates some loop control, leading to more speed-up. Loop peeling can be useful to eliminate data dependencies and expose parallelism. It can also serve as a step to peel off iterations before applying vectorization.

Data layout Transformations

Data layout transformations complement loop transformation for different purposes, e.g. memory footprint reduction on memory-constrained systems, optimal data access per threads in parallel programs or cache reuse improvement.

We can identify specific layout transformations. *Data transposition* consists in permuting axes of an N-dimensional array. A typical use is to align data accesses with the storage policy

of arrays in a programming model (e.g. row-major in C, column-major in Fortran). *Padding* insert additional data to change the dimensions of an array. It can be used to align data in the cache to avoid false sharing. Another example of improving thread access to memory using layout transformations is *memory coalescing* for aligned accesses.

Specific types layout transformations are required for applications such as stencil computations [67] or sparse data structures [96].

Algebraic Transformations

Previous work [29, 142, 88] show the benefits of algebraic transformations thanks to associativity, distributivity and commutativity properties. For example, it is possible to considerably reduce the algorithmic complexity of a program through transformations such as expression splitting or factorization.

This far, we have presented several general transformation strategies. However, an important aspect is finding relevant compositions of transformations, which can be tightly domain-specific. As an example, we show in the next sections how this applies to applications in computational fluid dynamics.

3.2 Computational Fluid Dynamics Applications: Overview

Numerical analysis is the study of problems based on continuous mathematics using numerical approximations. Equation discretization is a fundamental process that can be performed with different *orders of approximation*; for a nth-order approximation, the order of magnitude of error is $O(x^{n+1})$. Therefore, the greater the order of approximation, the more precise the discretization. The evolution of numerical methods has a direct correlation with that of computers; with the help of increasingly powerful machines, the state-of-art of numerical analysis advances with methods producing more and more accurate results.

Computational fluid dynamics (CFD) is the study of fluid flows using numerical methods. Methods used in CFD can be classified between *low order methods* (first-order, second-order) or *high order methods* (greater than second-order). While low order methods are widely used in engineering applications for their robustness and reliability, high order methods instead offer more accuracy — needed for specific types of flows — at the cost of greater computational intensity [60].

Navier-Stokes equations (NSE) govern fluid flows. They are crucial for the understanding of various types of fluid dynamics including the movement of water currents, atmospheric air mass or air motion surrounding vehicles. Several higher order numerical methods exist to solve NSE including the *spectral element*, *finite difference* or *finite element methods*. Our method of interest is the spectral element methods which involve using the *Helmholtz equation* in NSE solvers.

Fast and efficient NSE solvers require to focus on the Helmholtz operator. Huismann et al. [69, 70] demonstrate that using a tensor-based representation of the Helmholtz operator

enables algebraic transformations that lead to drastically reduced execution time when coupled with the static condensation method [109], a well established CFD-related technique.

In a typical CFD application, the volume of interest is divided into thousands of mesh points, where a computation is performed for each element of the mesh. For a given mesh point e, the numerical solution is computed in a 3-dimensional spatial context. Each dimension has a maximum value of p, which is the polynomial order used to approximate the solutions of the computation domain. Two CFD operators often computed in these meshes are the *Interpolation* defined as

$$\mathbf{v}_e = (\mathbf{A} \otimes \mathbf{A} \otimes \mathbf{A})\mathbf{u}_e \ , \tag{3.1}$$

and the *Inverse Helmholtz*:

$$\mathbf{v}_e = (\mathbf{S} \otimes \mathbf{S} \otimes \mathbf{S})\mathbf{D}_e^{-1}(\mathbf{S}^T \otimes \mathbf{S}^T \otimes \mathbf{S}^T)\mathbf{u}_e \ . \tag{3.2}$$

Computations are repeated for hundreds of thousands times, leading to execution times of several weeks, even when parallelized across thousands of cores.

3.3 CFD-related Optimization Techniques

From a programming point of view, *Interpolation* and *Inverse Helmholtz* can be implemented following the formula

$$v_{ijk} = \sum_{l,m,n} A_{kn} \cdot A_{jm} \cdot A_{il} \cdot u_{lmn} \ . \tag{3.3}$$

for *Interpolation* and

$$t_{ijk} = \sum_{l,m,n} A^T_{kn} \cdot A^T_{jm} \cdot A^T_{il} \cdot u_{lmn} \tag{3.4}$$

$$p_{ijk} = D_{ijk} \cdot t_{ijk} \tag{3.5}$$

$$v_{ijk} = \sum_{l,m,n} A_{kn} \cdot A_{jm} \cdot A_{il} \cdot p_{lmn} \tag{3.6}$$

for *Inverse Helmholtz.* A naive implementation corresponds to Listing 3.1a and 3.1b, in which we can recognize elements of the domain informations. (1) The outermost dimension with loop index e corresponds to iteration points of the mesh where M is the maximum number of elements. (2) The shape of tensor v reflects the 3-dimensional spatial context for which a coefficient is computed for a given mesh point. (3) N the size of inner tensor dimensions is equal to $p + 1$, where p is the polynomial order previously mentioned.

In the following, we present two levels of program transformations necessary for improved speed-ups. Note that for conciseness, we no longer explicit dimensions of loops and tensors involved in the mesh.

```
1   for (e = 0; e < M; e++)
2    for (i = 0; i < N; i++)
3     for (j = 0; j < N; j++)
4      for (k = 0; k < N; k++)
5       for (l = 0; l < N; l++)
6        for (m = 0; m < N; m++)
7         for (n = 0; n < N; n++)
8          v[e][i][j][k] = A[k][n] * A[j][m] * A[i][l] * u[e][l][m][n];
```

(a) Interpolation

```
1   for (e = 0; e < M; e++) {
2    for (i = 0; i < N; i++)
3     for (j = 0; j < N; j++)
4      for (k = 0; k < N; k++)
5       for (l = 0; l < N; l++)
6        for (m = 0; m < N; m++)
7         for (n = 0; n < N; n++)
8          t[e][i][j][k] = A[n][k] * A[m][j] * A[l][i] * u[e][l][m][n];
9
10   for (i = 0; i < N; i++)
11    for (j = 0; j < N; j++)
12     for (k = 0; k < N; k++)
13      p[e][i][j][k] = D[e][i][j][k] * t[e][i][j][k];
14
15   for (i = 0; i < N; i++)
16    for (j = 0; j < N; j++)
17     for (k = 0; k < N; k++)
18      for (l = 0; l < N; l++)
19       for (m = 0; m < N; m++)
20        for (n = 0; n < N; n++)
21         v[e][i][j][k] = A[k][n] * A[j][m] * A[i][l] * p[e][l][m][n];
22  }
```

(b) Inverse Helmholtz

Listing 3.1: Naive implementations of the CFD kernels in C

3.3.1 Algebraic Optimizations

Interpolation is a good candidate for algebraic optimizations. Listing 3.1a depicts its naive implementation. However, adding parentheses to Equation 3.3 enforces different legal evaluation orders thanks to associativity (c.f. Listings 3.2a, 3.2b and 3.2c for corresponding implementations):

$$v_{ijk} = \sum_{l,m,n} \left(A_{kn} \cdot \left(A_{jm} \cdot \left(A_{il} \cdot u_{lmn}\right)\right)\right), \tag{3.7}$$

$$v_{ijk} = \sum_{l,m,n} \left(A_{kn} \cdot A_{jm}\right) \cdot \left(A_{il} \cdot u_{lmn}\right), \tag{3.8}$$

$$v_{ijk} = \sum_{l,m,n} \left(A_{kn} \cdot \left(\left(A_{jm} \cdot A_{il}\right) \cdot u_{lmn}\right)\right). \tag{3.9}$$

```
for (i1 = 0; i1 <= 6; i1 += 1)
 for (i2 = 0; i2 <= 6; i2 += 1)
  for (i3 = 0; i3 <= 6; i3 += 1)
   for (i4 = 0; i4 <= 6; i4 += 1)
    tmp1[i1][i2][i3] += A[i1][i4] * u[i4][i2][i3];

for (i1 = 0; i1 <= 6; i1 += 1)
 for (i2 = 0; i2 <= 6; i2 += 1)
  for (i3 = 0; i3 <= 6; i3 += 1)
   for (i4 = 0; i4 <= 6; i4 += 1)
    tmp2[i1][i2][i3] += A[i1][i4] * tmp1[i2][i4][i3];

for (i1 = 0; i1 <= 6; i1 += 1)
 for (i2 = 0; i2 <= 6; i2 += 1)
  for (i3 = 0; i3 <= 6; i3 += 1)
   for (i4 = 0; i4 <= 6; i4 += 1)
    v[i1][i2][i3] += A[i1][i4] * tmp2[i2][i3][i4];
```

(a) Equation 3.7

```
for (i1 = 0; i1 < N; i1 += 1)
 for (i2 = 0; i2 < N; i2 += 1)
  for (i3 = 0; i3 < N; i3 += 1)
   for (i4 = 0; i4 < N; i4 += 1)
    tmp1[i1][i2][i3][i4] += A[i1][i2] * A[i3][i4];

for (j1 = 0; j1 < N; j1 += 1)
 for (j2 = 0; j2 < N; j2 += 1)
  for (j3 = 0; j3 < N; j3 += 1)
   for (j4 = 0; j4 < N; j4 += 1)
    for (j5 = 0; j5 < N; j5 += 1)
     tmp2[j1][j2][j3] += tmp1[j1][j4][j2][j5] * u[j5][j4][j3];

for (k1 = 0; k1 < N; k1 += 1)
 for (k2 = 0; k2 < N; k2 += 1)
  for (k3 = 0; k3 < N; k3 += 1)
   for (k4 = 0; k4 < N; k4 += 1)
    v[k1][k2][k3] += A[k1][k4] * tmp2[k2][k3][k4];
```

(b) Equation 3.8

```
for (i1 = 0; i1 < N; i1 += 1)
 for (i2 = 0; i2 < N; i2 += 1)
  for (i3 = 0; i3 < N; i3 += 1)
   for (i4 = 0; i4 < N; i4 += 1)
    tmp1[i1][i2][i3][i4] += A[i1][i2] * A[i3][i4];

for (j1 = 0; j1 < N; j1 += 1)
 for (j2 = 0; j2 < N; j2 += 1)
  for (j3 = 0; j3 < N; j3 += 1)
   for (j4 = 0; j4 < N; j4 += 1)
    tmp2[j1][j2][j3] += A[j1][j4] * u[j4][j2][j3];

for (k1 = 0; k1 < N; k1 += 1)
 for (k2 = 0; k2 < N; k2 += 1)
  for (k3 = 0; k3 < N; k3 += 1)
   for (k4 = 0; k4 < N; k4 += 1)
    for (k5 = 0; k5 < N; k5 += 1)
     v[k1][k2][k3] += tmp1[k1][k4][k2][k5] * tmp2[k3][k5][k4];
```

(c) Equation 3.9

Listing 3.2: Variants of Interpolation in C

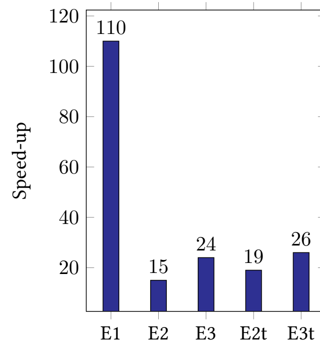


Figure 3.1: Speed-ups for variants of *Interpolation* compared to naive evaluation.

The runtime complexity for Listing 3.1a is $O(p^6)$. Yet finding a good evaluation order can reduce it up to $O(p^4)$. We conducted experiments to evaluate the speed-ups obtained for different variants; the results are shown in Figure 3.1. Starting with a naive evaluation of the tensor contractions, i.e. without any smart reordering of operations to reduce the algorithmic complexity, parallelization of the mesh loop across 24 cores leads to a speed-up of 20×. In the following, this parallelized naive evaluation serves as our baseline. Figure 3.1 shows the speed-ups compared to this baseline for parallelized implementations of the three *Interpolation* variants. E1, E2, E3 respectively correspond to Equations (3.7), (3.8), and (3.9). E1 appears to be the best CPU-based variant as it best minimizes the algorithmic complexity.

3.3.2 Loop Transformations

Additional loop fusions and permutations can be applied to E1, E2, and E3. The results of applying such transformations on E2 and E3 correspond to variant E2t and E3t in Figure 3.1; this shows that fusions and permutations are not sufficient to reach the performance produced by the best algebraic transformation. Note that it is also best not to perform fusions on E1. Indeed, p the polynomial order generally ranges from 1 to 13; hence tensor dimensions are very small. This implies that usual transformations such as tiling or fusion are actually performance degrader. The main transformations that contribute to speeding up the program are loop unrolling, vectorization and permutations. Loop fusions have the particular effect of hindering permutations opportunities, hence limiting optimal performances.

As an alternative to the algebraic transformation resulting into variant E1, one could think of performing some loop invariant code motion. For example, it is possible to hoist out `A[i1][i4]` and `A[i2][i5]`:

```
1 for (i1 = 0; i1 < N; i1 += 1)
2  for (i2 = 0; i2 < N; i2 += 1)
3   for (i3 = 0; i3 < N; i3 += 1)
4    for (i4 = 0; i4 < N; i4 += 1) {
5     a = A[i1][i4];
6     for (i5 = 0; i5 < N; i5 += 1) {
```

```
7         b = A[i2][i5];
8         for (i6 = 0; i6 < N; i6 += 1)
9          v[i1][i2][i3] += a * b * A[i3][i6] * u[i4][i5][i6];
10       }
11     }
```

However, this induces the same issue caused by loop fusions; this transformation is not beneficial as it inhibits good permutation opportunities (cf. Appendix A.2). Consequently, such a variant is even slower than the original naive version.

Despite the small set of transformations applicable, it is not always clear when and how to apply them; with the iterative process, polynomial orders may vary from one execution instance to the other. For a better understanding of transformation heuristics, we conducted a series of experiments to study optimizations heuristics chosen by ICC, the Intel compiler, given a polynomial order.

Since ICC does not handle algebraic optimizations (cf. Appendix A.1), we assume as input program the variant of Equation 3.7 (cf. Listing 3.2a). We name L_1, L_2, L_3 the three successive loop nests.

Experiments and outcomes

Using ICC version 18.0, our compilation instruction is **`icc -O3 -xHost -qopt-report=1|2|5 -qopt-report-phase=vec,loop`** to generate heuristics reports with different verbosities. Observations are exposed in Table 3.1.

<table>
<tr><th>N = p + 1</th><th>ICC heuristics</th></tr>
<tr><td>2</td><td rowspan="6">ICC fully unrolls all loop dimensions in all three loop nests
and performs loop interchanges in L_1 and L_2 as they present such opportunity.
Loop interchange permutes the two innermost dimensions (i.e., (1 2 3 4) –> (1 2 4 3))</td></tr>
<tr><td>3</td></tr>
<tr><td>4</td></tr>
<tr><td>5</td></tr>
<tr><td>6</td></tr>
<tr><td>7</td></tr>
<tr><td>8</td><td rowspan="2">Loop collapsing of <code>i2</code> and <code>i3</code> (c.f. Listing 3.2a) and vectorization (with peeling for N = 9)
of innermost loops in L_1 and L_3 only.</td></tr>
<tr><td>9</td></tr>
<tr><td>10</td><td rowspan="3">Compared to N=8,9, heuristics change for L_2 only: loop interchange is applied,
then vectorization is performed on the 2nd dimension.</td></tr>
<tr><td>11</td></tr>
<tr><td>12</td></tr>
<tr><td>13</td><td rowspan="2">All innermost loops are vectorized (preceded by peeling for N = 13)
in addition to collapsing in L_1 and L_3 and loop interchange in L_2.</td></tr>
<tr><td>14</td></tr>
</table>

Table 3.1: ICC heuristics for different polynomial orders

These different results exhibit several aspects:

- Full unrolling is the de facto transformation for sizes below 8;

- A unique loop interchange opportunity is taken into account in L_1 and L_2 (i.e., (1 2 3 4) –> (1 2 4 3)). Yet, L_1 presents another opportunity, that is (1 2 3 4) –> (1 4 2 3). In addition, it seems that this latter interchange strategy orders memory accesses more efficiently than ICC's choice;
- In L_1, the loop collapsing opportunity from sizes above 8 is prioritized over the loop interchange opportunities. Yet, with further manual experiments, collapsing did not seem to be more efficient than interchanging.
- Vectorization in L_2 only appears from sizes above 10. It is not fully clear why ICC's heuristics consider vectorization this late, in comparison to the other loops. A possible explanation is the coupling with loop collapsing in L_1 and L_3. Perhaps the cost model estimation assumes collapsing enables good vectorization. Furthermore, the vectorized dimension is the 2nd one. We cannot clearly state that the following dimensions were unrolled before; reports did not provide such indications.

The clear tendency of chosen heuristics per polynomial order ranges is informative. Nevertheless, other heuristics choices exist that would now need to be explored by hand.

3.4 Envisioned Tool Flow

Domain practitioners would spend a lot of time hand-writing different variants per polynomial order to find an efficient one (and that is if they have a good knowledge of code optimization techniques). An automated process, from a DSL to optimized C code generation as depicted in Figure 3.2, is more convenient. The iterative nature of the execution of computations makes approaches such as iterative compilation [81] useful to find beneficial transformations.

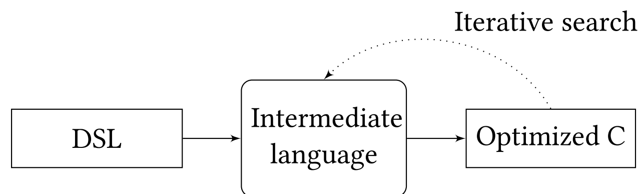


Figure 3.2: Envisioned tool flow

CFDlang

CFDlang [112] is the DSL designed in our context. Using a compact representation of tensors, it proposes a notation for tensor expression close to mathematics. The particularity of CFDlang is its expressiveness scope. Full numerical applications are not meant to be written but rather performance critical tensor expressions, such as *Interpolation* that are part of larger applications.

The following code sample shows the implementation of *Interpolation* with CFDlang.

```
1 var input A : [7 7]
2 var input u : [7 7 7]
3 var output v : [7 7 7]
4
5 elem [u v] 216
6
7 v = A # A # A # u . [[1 6] [3 7] [5 8]]
```

Tensors are declared using the **var** keyword, accompanied by the **input** or **output** qualifiers to explicit the program dataflow and a list of integers specifying the shape. CFDlang decouples mesh dimensions from tensor shapes; instead of specifying `v : [216 7 7 7]`, where 216 is the mesh size, CFDlang provides the **elem** keyword to identify tensors with mesh dimensions. The list of tensors is then followed by the actual size of the mesh. The actual tensor expression is written similarly to the mathematical expression. The $\otimes$ operator which is a tensor product directly corresponds to CFDlang's **#** (hash) operator, which concatenates all tensors. In the presence of tensor contractions, expressions are followed by a list of pairs of axes to be contracted, assuming axes of the concatenated tensor. As the domain practitioners generally write Fortran code, CFDlang is designed to be integrated in a Fortran interface as in Listing 3.3.

CFDlang is meant to serve as a front-end to an intermediate engine generating efficient code. In the next section, we go through several potential frameworks for this purpose.

3.5 Existing Optimization Frameworks

In our CFD context, we need an intermediate engine that has three main features: the ability to (1) perform algebraic transformations, (2) choose specific transformations and (3) search optimizations iteratively. Furthermore, with CFDlang as a frontend, several types of tensor operations need to be expressible (e.g. contractions or entrywise multiplications).

Several potential frameworks exist; we can assess their suitability in light of our requirements.

3.5.1 Linear and Tensor Algebra Frameworks

We focus on most relevant optimizing frameworks, therefore we do not take into account scientific libraries such as Numpy [17], Libtensor [71] or Xtensor [19].

Pluto

Pluto [39] is a polyhedral-based source-to-source automatic parallelization tool. Using C programs as input code, Pluto's main goal is finding affine transformations for efficient tiling of kernels delimited by `#pragma scop/endscop`. Generated code includes thread-level parallelism for multicore CPUs using OpenMP as well as vectorization. While Pluto performs polyhedral optimizations automatically, users are free to enable or disable transformations through appropriate options.

```
subroutine Interpolation__CFDlang_Init(h, u)
type(Tensor_Handle),   intent(inout) :: h
real(RDP), contiguous, intent(in)    :: u(:,:,:,:)

integer              :: n, n_element
character(len=8)     :: n_str, ne_str
character(len=1024) :: source

n         = size(u,1)
n_element = size(u,4)

write (n_str,  "(I8)") n
write (ne_str, "(I8)") n_element

source = "type matrix    : ["//n_str//" "//n_str//"]"//c_new_line//&
"type tensorIN   : ["//n_str//" "//n_str//" "//n_str//" "//ne_str//"]"//c_new_line//&
"type tensorOUT : ["//n_str//" "//n_str//" "//n_str//" "//ne_str//"]"//c_new_line//&
"                                        "//c_new_line//&
"var input A    : matrix                 "//c_new_line//&
"var input u    : tensorIN               "//c_new_line//&
"var input output v   : tensorOUT        "//c_new_line//&
"                                        "//c_new_line//&
"v = A # A # A # u . [[5 8] [3 7] [1 6]]"//c_new_line//&
c_null_char

call Tensor_Init_Context(h%ctx)
call Tensor_Init_Code_Gen(h%ctx, h%cg, source, &
rowMajor = 0, &
fuseElementLoop = 1, &
restrictPointer = 1, &
graphCodeGen = 1  &
)

call Tensor_Generate_C_Code(h%ctx, h%cg)

call Tensor_Init_Kernel(h%ctx, h%cg, h%k, cleanOnDestruction = 0 )
call Tensor_Build_Kernel(h%ctx, h%k)
call Tensor_Final_Code_Gen(h%ctx, h%cg)

call Tensor_Init_Execution(h%ctx, h%k, h%ex)
end subroutine Interpolation__CFDlang_Init
```

Listing 3.3: Fortran interface including a CFDlang code sample

The Tensor Contraction Engine (TCE)

TCE [29] is a domain-specific framework for quantum chemists; it focuses on tensor contractions which represent the main causes of bottlenecks in the scientific computations encountered in this domain. TCE is integrated in NWCHEM [132], an open-source computational chemistry package. TCE translates the high-level specification of the program (in a mathematical form) into C/Fortran code. The output program is optimized using techniques to minimize algorithmic complexity, memory footprint and communication costs.

The Tensor Transposition Compiler (TTC)

TTC [123] is a compiler that generates high-performance C++/CUDA C code for tensor transpositions. TTC explores different search spaces for opportunities of transformations. When finding a good candidate, the key idea is to apply a blocking technique; the transposition is broken down into multiple independent 2D transpositions.

The Tensor Algebra Compiler (TACO)

TACO [79] is a C++ library for the compilation of dense and sparse linear and tensor algebra expressions. TACO requires as input expressions using an index notation and a format descriptor of tensors. Format descriptors include the specification of each tensor dimension as dense or sparse and the order in which they are stored. Different sparse tensor storage format as supported thanks to TACO's representation of tensors based on trees. TACO is able to generate code for tensor expressions involving dense or sparse operands, or both.

TensorFlow

TensorFlow [20] is an API for high-performance numerical computation mainly used for deep neural networks. Computations can be deployed on various platforms, including CPUs, GPUs, and TPUs (Tensor Processing Units). TensorFlow's computation model is based on data flow graphs. Computations are further optimized with XLA [18], a JIT-compiler. TensorFlow shares similarities with Theano [35].

Tensor Comprehensions (TC)

TC [133] is a C++ library coming along with an index notation language for the compilation of machine learning kernels. Its compilation flow includes various optimizations ranging from parallel-specific transformations thanks to Tapir LLVM [117] to polyhedral transformations thanks to ISL [135]. TC mainly generates CUDA code using autotuning techniques but users still have a level of control on what optimizations should be applied through TC's API interface.

LGen and SLinGen

LGen [121, 122] is a compiler for high-performance basic linear algebra computations using autotuning techniques. Input code provided in a mathematical form is successively transformed into lowered form. Each lowering stage incorporates high-level loop optimizations using ISL as well as C-code level transformations. A variant of LGen, SLinGen [120], has been specifically designed for small-scale applications.

3.5.2 Levels of Expressiveness and Optimization Control

None of the above-cited frameworks feature all requirements for the CFD tool flow. However, there are different degrees of unsuitability.

Due to its restriction to tensor contractions, TCE seems not to have suitable expressiveness. This is unfortunate as it is the only framework that considers algebraic transformations problematics. Furthermore, optimizations heuristics proposed by TCE seeks trade-offs between program transformations (including loop fusions) and memory footprint. Our CFD context does not present the same issues: the sizes of tensors makes memory footprint optimizations irrelevant for small sizes.

At this time, TACO is limited as it can handle only one kernel at a time. The scope of TACO also seems to highly focus on sparse tensors problematics, yet the CFD applications involve dense tensors only. Furthermore, TACO does not yet handle classic loop transformations either.

TTC's asset, despite its restricted scope, is the search for good candidates for transpositions. Unfortunately, this is not the primary need for the CFD applications.

TensorFlow and Theano operate differently; tensor computations are first built as computation graphs. Before the actual execution of such graphs, optimizations may be applied but they remain specific to the graphs structures. They also do not incorporate loop-level optimizations.

Generally, frameworks that do not allow control over specific transformations are black-boxes almost impossible to adapt to the CFD needs. Conversely, those that do allow some control generally provide the capability of choosing a set of transformations to apply, which is much more appealing. We consider Pluto and TC in this latter category which makes them the most suitable candidates here. SLinGen is also a potential candidate thanks to its autotuning features. Note that if the only unmatching criterion is the lack of algebraic transformations, this level of transformation could be handled at the CFDlang level.

3.6 Outcomes

We exposed characteristics and optimization needs for certain CFD applications. One could rely on the ICC compiler only to efficiently compile programs. Yet a study with ICC shows that there may be heuristics that are not chosen by the compiler. This raises the need to have more flexibility in optimization choices; having the ability to enforce different transformation orders is more appealing. Several tensor optimization frameworks exist, but they often do not meet the needs of the CFD kernels. Beyond this context, they seem not to generalize well from one domain to another.

Chapter 4

The NUMA Challenge

Contrairement à la plupart des contributions présentées dans l'état-de-l'art, nous abordons plutôt les problèmes liés à l'optimisation de la mémoire. Nous nous intéressons particulièrement aux questions de placements de données sur les architectures NUMA (Non-uniform memory access – accès à la mémoire non uniforme) qui sont souvent négligées. Comme le montre notre étude de cas sur un outil de compilation polyédrique [130], la prise en compte du NUMA peut être cruciale pour complémenter les techniques de transformations usuelles. Généralement, cette prise en compte implique l'usage de différents types de placement de données standard sur le NUMA telles que la réplication des données accédées en lecture ou encore la répartition d'un tableau sur différents nœuds NUMA. Pour aller plus loin, les méthodes de placement pourraient être étendues et affinées pour plus de précision. Cependant, des décisions à l'exécution du programme, telles que l'ordonnancement des threads, doivent complémenter cette démarche.

∗ ♣ ∗ ♣ ∗

As exposed in Chapter 2, intermediate representations do not always feature abstractions for memory optimizations on parallel archtectures. We, therefore, study this aspect with a focus on data placement on NUMA (Non-Uniform Memory Access) architectures. NUMA-awareness is often overlooked, yet, important to complement the transformation techniques mentioned in Chapter 3.

In this chapter, we explain what distinguishes such architectures and summarize various NUMA management solutions. We also demonstrate the complementarity of data placements with other transformation techniques through a case study of data locality enhancement, using Pluto [39] as a proof-of-concept. We show that codes generated by Pluto can benefit from more data locality with additional NUMA placement heuristics. We then focus on two placement techniques, i.e. *interleaved allocation* and *replications*, to study how far we can exploit them in source-to-source compilation.

This chapter is based on:

Adilla Susungi, Albert Cohen, Claude Tadonki. ***More data locality for static control programs in NUMA architectures***. In Proceedings of the 7th International Workshop on Polyhedral Compilation Techniques (IMPACT 2017).

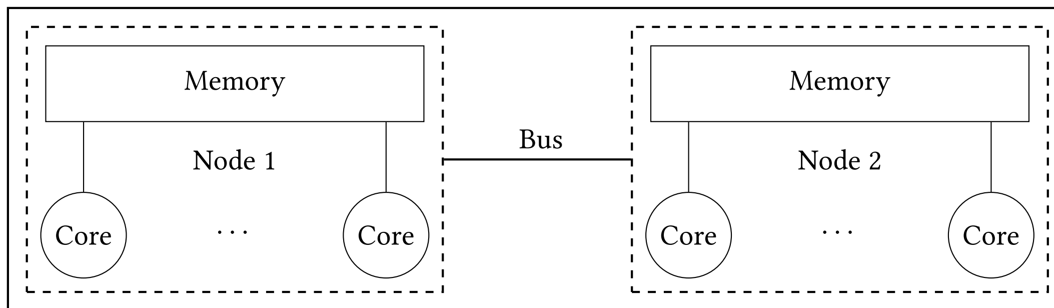


Figure 4.1: 2-nodes NUMA machine

4.1 NUMA Architectures: Topologies and Management Solutions

NUMA architectures constitute an important class of parallel architectures. These large-scale systems organize the physically shared memory across several nodes connected through cache-coherent high-performance links. With such a design, threads either access the memory resources located on the same node as the core executing the thread —the local node— or on a different node —a remote node (cf. Figure 4.1). Consequently, uncontrolled data placement can yield traffic contention when all threads are accessing the same memory bank. Furthermore, remote data access increase memory latencies.

NUMA systems were introduced as a cure to the scalability issues of symmetric multi-processors with Uniform Memory Access (UMA). Unfortunately, NUMA-unaware programs running on NUMA platforms tend not to benefit from the additional computational power and memory bandwidth offered by the multiple nodes of the system.

Figures 4.2 and 4.3 depict two different 2-nodes NUMA core distributions; The *Pau* machine follows a cyclic distribution of 32 cores (including hyperthreading) whereas the *Taurus* machine follows a block distribution of 24 cores (without hyperthreading).

Different types of management solutions exist, ranging from operating systems features to research contributions in the area of parallel programming languages.

4.1.1 Operating Systems

Many operating systems provide NUMA-aware data placements. The *first touch* policy is common and performed by default when using the `malloc` function. It consists in not assigning a memory page at the time a `malloc` is specified, but instead, when the *first* write is performed. A programmer can consider this policy for optimal data accesses. But this approach becomes irrelevant when read accesses following the first write are performed by another thread on a different node.

The *affinity-on-next-touch* policy addresses limitations of the first touch policy as it allows dynamic page migration with respect to read accesses. Goglin and Furmento [65] implemented this policy in the Linux kernel.

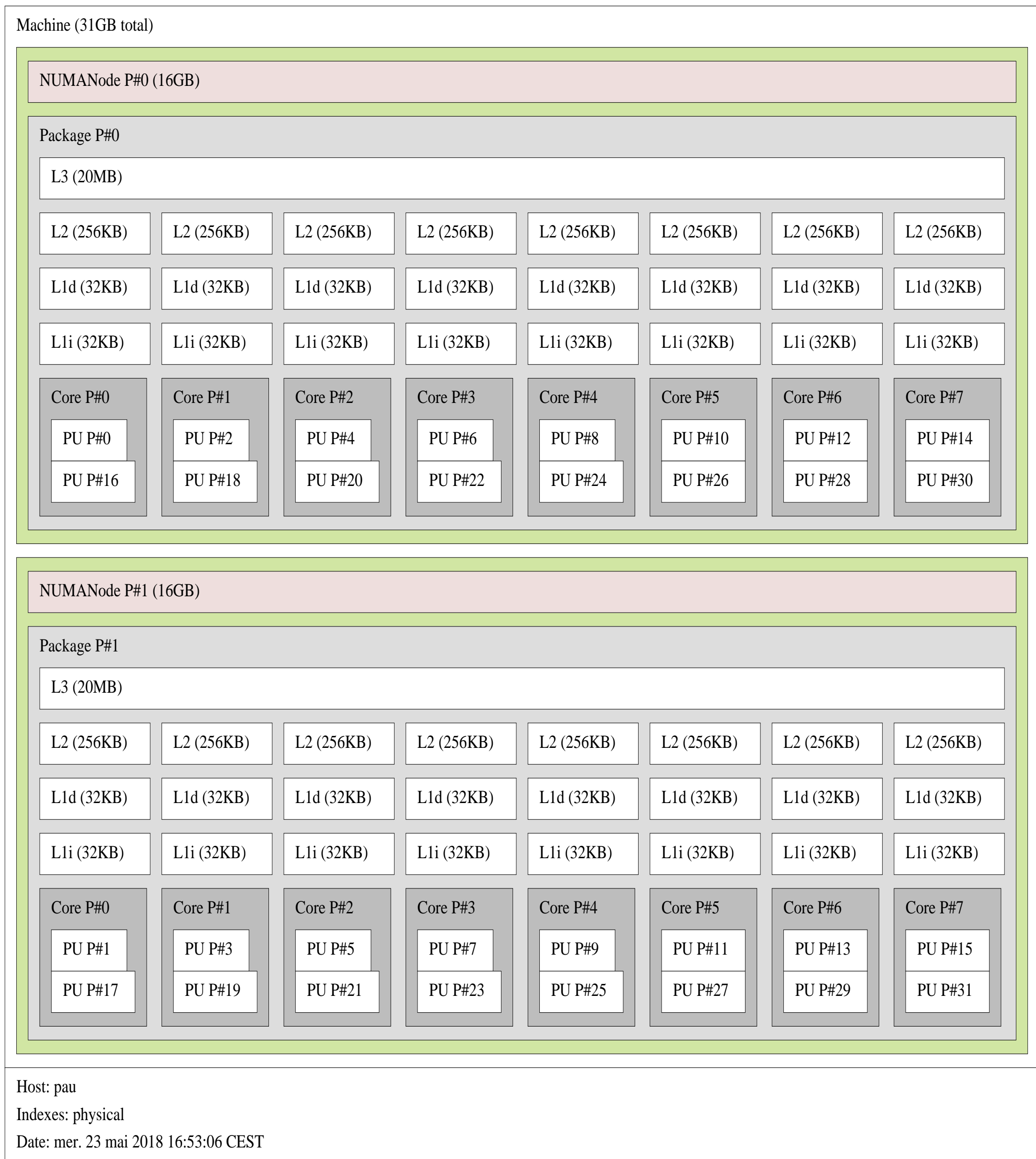


Figure 4.2: The *Pau* machine following a cyclic distribution of cores.

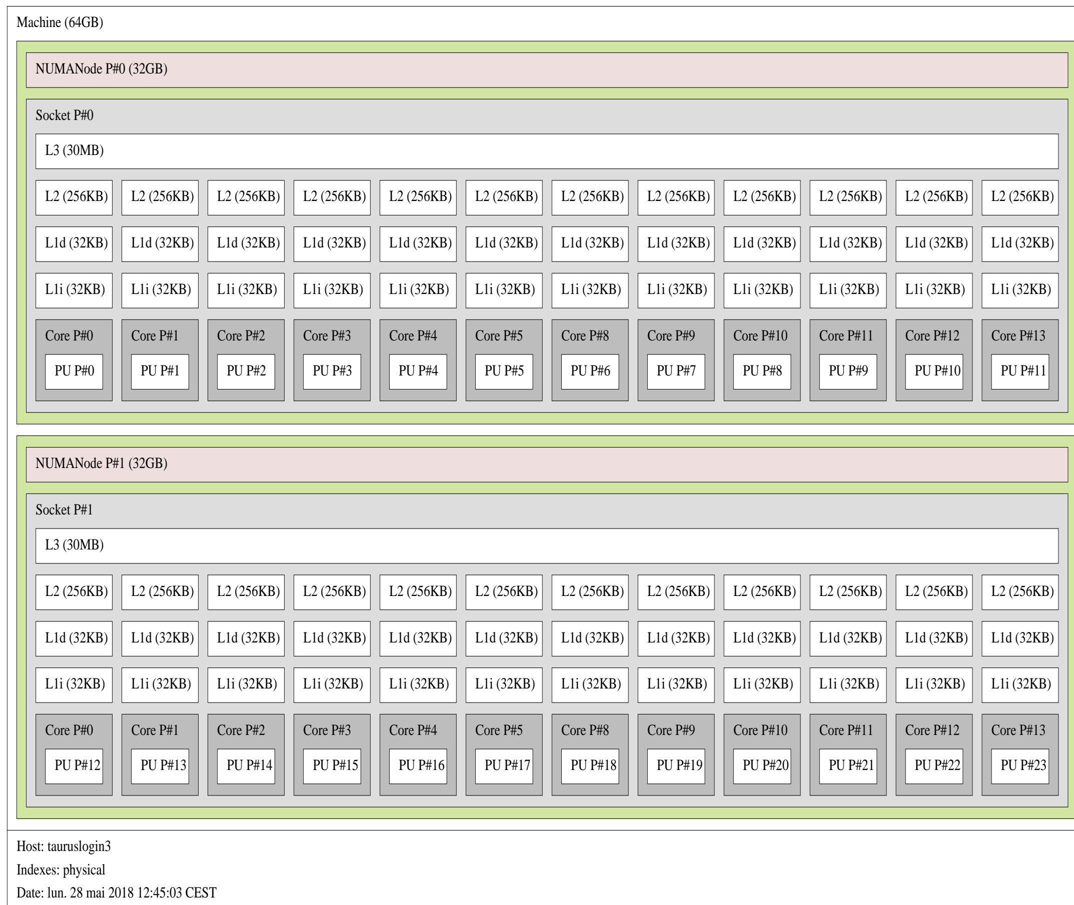


Figure 4.3: The *Taurus* machine following a block distribution cores.

Another approach, designed for the Linux kernel, is Carrefour [58] which focuses on avoiding memory contention through decisions for page co-location (for the placement of a page on the same node as the core accessing it), page interleaving (round-robin placement of pages across nodes) or replication of pages on several nodes and thread clustering to promote thread affinity according to the shared data.

4.1.2 NUMA APIs

While the above-mentioned strategies are implemented at the kernel level, several NUMA APIs are accessible to users.

Operating system related

Linux *Libnuma* is the NUMA API for Linux. It features memory and process management functions related to the following policies:

- *Local* – allocation is performed on the node on which the process is currently running. This is the default policy and is also based on the first touch policy.
- *Preferred* – allocation is first performed on a given set of nodes but if it fails, it is performed on the closest nodes.
- *Bind* – allocation is strictly performed on a given set of nodes. If the allocation fails, no fall-back is possible.
- *Interleaved* – allocation is performed in a round-robin fashion across NUMA nodes, on a set of pages selected. As this operates at a page size granularity, this is ineffective if the size of data allocated is smaller than a page size.

Libnuma comes along with *numactl*, a tool providing the same features as command lines. However, changes through command lines apply only at the level of the entire program.

Solaris The Locality Group Library (*liblgrp*) is included of Solaris. In this library, a *lgroup* is a set containing multiple CPUs and a memory that is accessed with the same latency for all CPUs. In multiple lgroups, any CPU can access any memory, with a variation according to the distance between the CPU and the requested memory. It is then possible to retrieve pieces of information about a given lgroup (number of resources, CPUs, size of free memory, etc). Threads and memory placements can be performed with respect to lgroups.

Windows With Windows's NUMA API, it is also possible to retrieve pieces of information about the hardware and allocate memory on a specific NUMA node. However, unlike libnuma, there is no such thing as memory allocation policies.

Operating system independent

`hwloc` The Portable Hardware Locality (`hwloc`) [40] is a portable software across several operating systems including Linux, Solaris, Windows, BSD or Darwin / OS X. Its purpose is to gather information about hardware through an API interface or command lines. It is also possible to bind threads or processes using the API.

MAi The Minas Framework [107] includes high-level functions for data allocation and placement for arrays. Several memory policy groups exist:

- The *Bind* group either divides data into blocks independently from the number of threads or place data in one or a set of restricted nodes.
- The *cyclic* group distributes data in a round-robin fashion across memory pages.
- In the *random* group, memory pages are placed randomly on CC-NUMA nodes.

Data distribution over nodes can be performed using entire arrays or a block distribution. Sizes for the block distribution can be either chosen by the programmer or automatically by MAi.

Majo and Gross API primitives in [89, 90] are proposed to specify block-cyclic or block-exclusive data distribution on each dimension of an array. It is then possible to use OpenMP as a front-end where the data distribution patterns are encoded as clauses of the `schedule` construct.

4.1.3 Languages extensions

It is interesting to observe the extent of NUMA support in actual implementations of popular languages.

HPF [4], Co-array Fortran or Chapel feature constructs for data distribution across processes. Chapel also includes a NUMA model in its locale models, even though not fully mature. OpenMP only provides the `OMP_PROC_BIND` environment variable to disable thread migration at execution time. GNU UPC [3] is a toolset providing a NUMA-aware execution environment (including thread scheduling and memory allocations based on numactl) for programs written in UPC. Current MPI implementations also feature options at execution time for process binding. Finally, HPX [5] is an effort currently made to provide high-level parallel programming solutions in the C++ language specification, including data locality and thread placement constructs within the language as well as at execution time.

Therefore, NUMA support, when existing, mainly includes process binding at execution time only. Very few languages provide means for high-level data distribution on NUMA nodes. In the following, we present contributions proposing to fill this gap.

Case with OpenMP

Bircsak et al. [36] proposed an HPF-like data distribution: for each dimension, we can either choose a block, cyclic or no distribution. Users do not specify the granularity; it systematically depends on the number of memories available. Directives for the location of computations and thread migration are available.

Huang et al. [68] proposed block distributions with variable dimensions, data replications and thread affinity. They also introduce the concept of location as Chapel's locales or X10's places so that users may specify which set of cores or which NUMA nodes a computation must run on. Note that these are considered as hints only as they do not expect the programmer to have a deep understanding of the underlying hardware.

Muddukrishna, Jonsson and Brorsson [94] extended the OpenMP API with memory allocation functions that enable the programmer to specify simple data distribution patterns such as:

- *standard* – as a standard malloc;
- *fine* – element-wise distribution of arrays on interleaved pages among nodes;
- *coarse* – interleaved distribution of entire arrays among nodes.

With ForestGOMP, Broquedis et al. [41] extended the GNU OpenMP runtime implementation for NUMA support.

In the MPC framework [102] there is no construct extensions; it is a complete NUMA-aware implementation of OpenMP with full respect to the OpenMP 3.1 standard. Executing a program with MPC requires only to specify for instance the number of nodes available.

Case with MPI and Pthreads

Beyond OpenMP, the MPC framework also provides an implementation of Pthreads and the MPI 1.3 standard, with a few features from more recent standards. Generally, the framework is implemented in order to provide efficient hybrid OpenMP/Pthreads/MPI programming, which is known to be difficult to properly exploit on NUMA clusters.

Case with Intel Threading Building Blocks (TBB)

Majo and Gross [90] introduce TBB-NUMA, a NUMA-aware parallel programming library based on Intel TBB in which various data distribution patterns are available. Custom distribution patterns can also be specified.

4.2 Case Study: Beyond Loop Optimizations for Data Locality

The purpose of this section is to highlight the benefits of integrating NUMA-awareness in optimizing compilers for data locality. As a proof-of-concept, Pluto [39] is our demonstration framework. Pluto is based on the polyhedral model, intensively used for the optimization of loop nests that fit the model's constraints (and many numerical applications do). Today, polyhedral tools are capable of producing highly optimized parallel code for multicore CPUs [39], distributed systems [37], GPUs [137] or FPGAs [103]. It is, however, interested to note that NUMA optimizations are not applied.

In the tool flow of our demonstration framework [130], we clearly seperate optimizations handled by Pluto with those we introduce; Pluto is in charge of all control flow optimizations and parallelism extraction, whereas our post-pass implements data placement on NUMA systems. Furthermore, as we did not emphasize on layout transformations in the previous chapter, we take this opportunity to introduce them — more specifically transpositions — along with NUMA optimizations. Indeed, layout transformations are often not first-class citizens in polyhedral tools.

As Pluto outputs may be complex, we handle affordable cases.

4.2.1 Experimental Setup

We present case studies of several PolyBench [11] programs: Gemver, Gesummv, Gemm, and Covariance. The minimal set of Pluto options that we use are tiling for L1 cache, parallelism, and vectorization. All programs are compiled with `gcc -O3 -march=native` which enables vectorization by default. We execute them on a 2 sockets NUMA system with 36 Intel Xeon cores E5-2697 v4 (Broadwell) @2.30 GHz distributed across 4 nodes (9 cores per node, L1 cache: 32K).

```
1  for (i = 0; i < _PB_N; i++)
2   for (j = 0; j < _PB_N; j++)
3    A[i][j] = A[i][j] + u1[i] * v1[j] + u2[i] * v2[j];
4
5  for (i = 0; i < _PB_N; i++)
6   for (j = 0; j < _PB_N; j++)
7    x[i] = x[i] + beta * A[j][i] * y[j];
8
9  for (i = 0; i < _PB_N; i++)
10  x[i] = x[i] + z[i];
11
12 for (i = 0; i < _PB_N; i++)
13  for (j = 0; j < _PB_N; j++)
14   w[i] = w[i] +  alpha * A[i][j] * x[j];
```

Figure 4.4: Original Gemver code

We apply interleaved allocations and data placements as NUMA optimizations. Simple guidelines are used to decide which policy to choose:

- An array is replicated on all nodes if each thread performs *read-only* accesses on the *entire* array. As replicating written arrays would require additional heuristics to ensure data coherence, we do not handle such cases.
- If an array is not replicated, then it is interleaved on all nodes (especially if it is multi-dimensional) to reduce traffic contention.

Following this rule, Gemver appears to be the only program in which we apply additional data replications.

Transpositions are performed at initialization time using indexes permutation only.

4.2.2 Observations

Gemver

We compare different versions of Gemver generated by Pluto:

- The default output of Pluto (named Default);
- The addition of NUMA placement only, considering replication and interleaved allocation (named NUMA);
- The addition of transpositions only (named Layout);
- The addition of combined NUMA placement and transposition (named NUMA-Layout).

Moreover, we consider two different Pluto outputs: the first output is generated using the *no fuse* heuristic and the second using the *smart fuse* heuristic. The *max fuse* option is not suitable for Gemver, therefore, we do not consider it. Figure 4.5 shows, for both outputs, the speed-ups over their respective Default version executed on 1 core.

The *smart fuse* version scales better than the *no fuse* version. This is due to the enhancement of cache reuse thanks to the fusion of the two first loops. However, compared to a naive parallel version of Gemver (i.e. the Default version executed in parallel), the two Pluto outputs fare no better. This means that optimizing the temporal locality only is not sufficient. As 10 cores is the threshold from which NUMA effects appear on the considered machine, we can also see in Figure 4.5 that they poorly scale on 16 and 36 cores.

According to the aforementioned guidelines, we applied the exact same NUMA placement in both outputs. This solution improves both, but *no fuse* provides the best performance on 36 cores. When using interleaved allocation for an array, the different threads accessing it must perform row-major accesses to preserve node locality as much as possible. This is what occurs with *no fuse*. However, with *smart fuse*, the first loop is permuted in order to perform the fusion legally. Despite the benefit of this transformation to enhance cache reuse, some node locality is lost since column-major accesses are still performed.

Thread binding using `OMP_PROC_BIND` seem not to significantly improve the performances of the NUMA and NUMA-Layout versions. It may even lower the speed-up due to inadequate thread binding with respect to the interleaved allocation. In this case, a finer binding heuristic using, for example, **`GNU_CPU_AFFINITY`** can be useful. Another reason for performance decrease can also be load imbalance. This can be adjusted thanks to a runtime system.

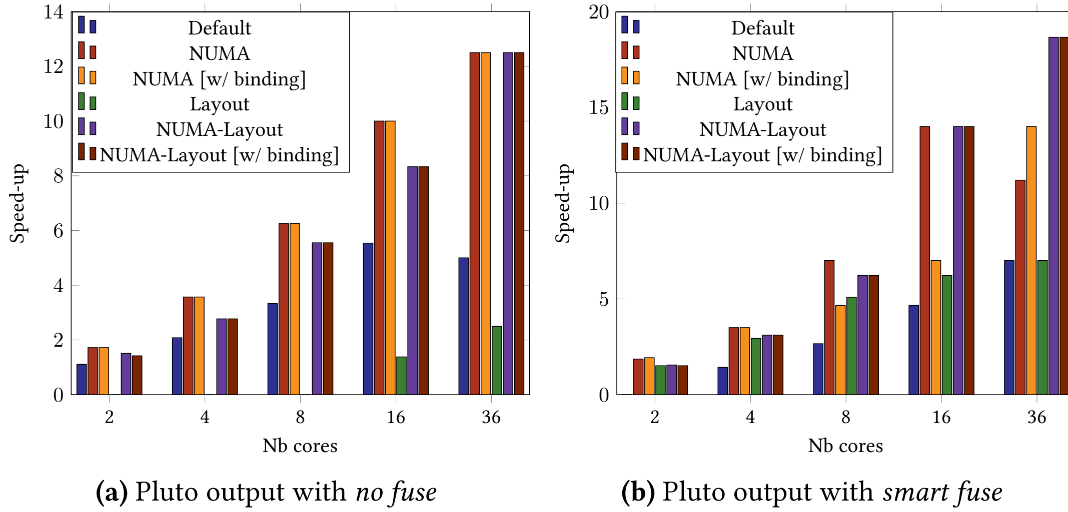
We noticed that, when considering replications alone as NUMA placements, the positive effects of thread binding are much more noticeable despite less speed-up. Interleaved allocations, therefore, seem to inhibit the effects of thread binding since they may reduce node locality per thread. Thread migration can then operate as a counterbalance.

On the other hand, layout transformations are better suited to *smart fuse* because data reuse is much more enhanced. Furthermore, this allows threads to perform row-major accesses with respect to an interleaved mapping. Such a systematic transformation does not align well with the schedule of the *no fuse* version, hence the degraded performance. Therefore, at this level of optimizations, the best version of Gemver appears to involve smart fuse, NUMA placement, and transposition.

Additional data replications using `memcpy` add 0.3 ms to all execution instances of versions with NUMA placement. They, therefore, have a negligible impact on the performance observed.

Gesummv

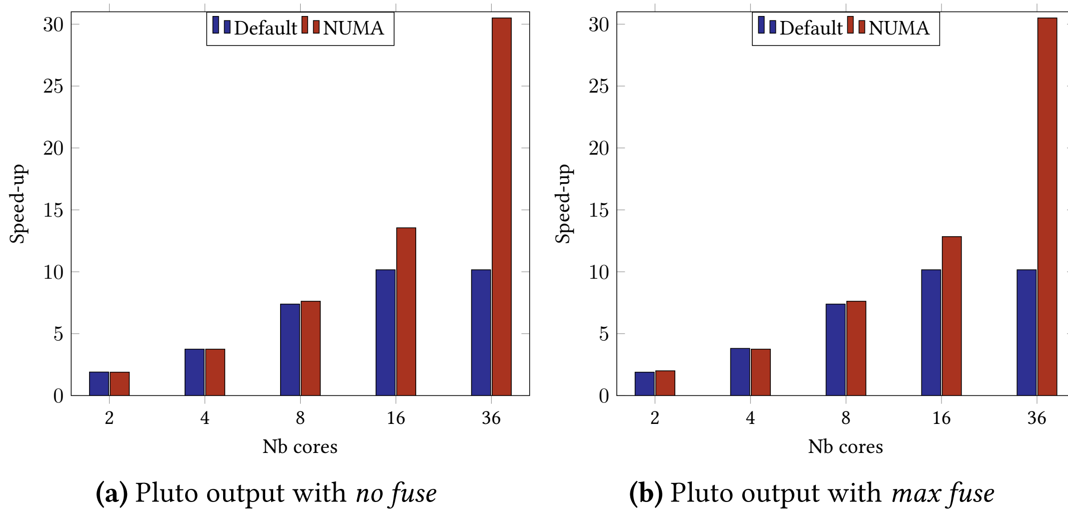
For Gesummv, we consider Pluto outputs with *no fuse* and *max fuse* (*no fuse* and *smart fuse* heuristics result into the same generated code). Results are depicted in Figure 4.6. Similarly to the case of Gemver, optimized temporal locality alone does not provide better performance than the naive parallel version for codes generated by Pluto (i.e. the Default version). Both outputs also scale poorly without NUMA-aware placements. We applied interleaved allocation, which brings a 3× speed-up on 16 cores and 4× speed-up on all 36 cores. As



(a) Pluto output with *no fuse* **(b)** Pluto output with *smart fuse*

Figure 4.5: Speed-ups over Default versions executed on 1 core for Gemver

.



(a) Pluto output with *no fuse* **(b)** Pluto output with *max fuse*

Figure 4.6: Speed-ups over Default versions executed on 1 core for Gesummv

we observed that thread binding may not match with interleaved allocations, we consider exploring the effects of changing the granularity of interleaving for further improvements.

Covariance

In the following, we no longer consider different Pluto outputs. Instead, we consider a naive parallel version of Covariance (i.e. a parallel code without loop transformations that we name Naive) and a Pluto output with the most suitable heuristic. For both programs, we compare again versions with (i) NUMA placements only (interleaved data allocation, named NUMA), (ii) transposition as shown in Listing 4.1 (named Layout), (iii) and both together

(named NUMA-Layout). Their respective Default version correspond, again, to versions without any NUMA or layout transformations.

Figure 4.7 shows that the Default version of Pluto delivers much better performances than that of the Naive program. But Naive can be improved thanks to the data transposition. However, applying the same transposition to the Pluto output considerably reduces the speed-up, similarly to what we observe for Gemver with *no fuse*.

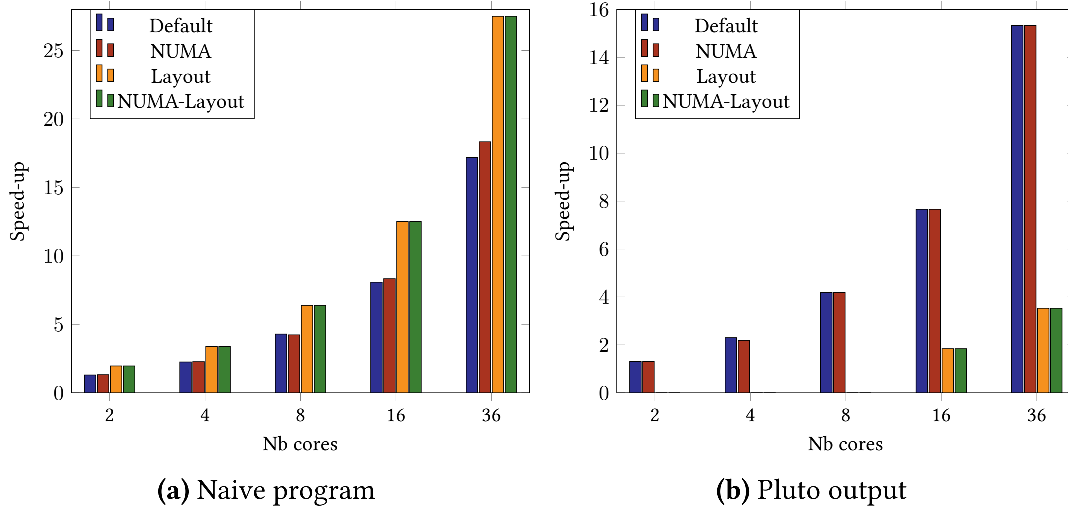
NUMA allocations have a little positive impact on the Naive program and none on the Pluto output. This is due to the fact that temporal locality already optimizes cache usage.

```
1  // Initializing the transposition
2  for (i = 0; i < N; i++)
3   for (j = 0; j < M; j++)
4    data[j][i] = ((DATA_TYPE) i*j) / M;
5
6  // Computation
7  #pragma omp parallel for private(i)
8  for (j = 0; j < _PB_M; j++) {
9   mean[j] = SCALAR_VAL(0.0);
10  for (i = 0; i < _PB_N; i++) {
11   mean[j] += data[j][i];
12   mean[j] /= float_n;
13  }
14 }
15
16 #pragma omp parallel for private(j)
17 for (i = 0; i < _PB_N; i++)
18  for (j = 0; j < _PB_M; j++)
19   data[i][j] -= mean[i];
20
21 #pragma omp parallel for private(j,k)
22 for (i = 0; i < _PB_M; i++)
23  for (j = i; j < _PB_M; j++) {
24   cov[i][j] = SCALAR_VAL(0.0);
25   for (k = 0; k < _PB_N; k++)
26    cov[i][j] += data[i][k] * data[j][k];
27   cov[i][j] /= (float_n - SCALAR_VAL(1.0));
28   cov[j][i] = cov[i][j];
29 }
```

Listing 4.1: Covariance with transposition of array `data`

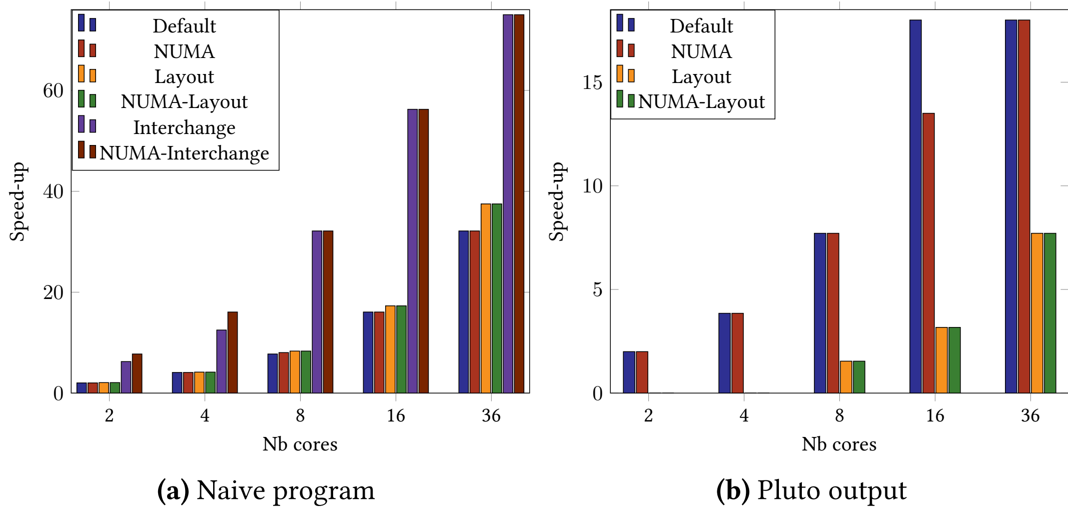
Gemm

For Gemm, we also consider a Naive program and a Pluto output. As shown in Figure 4.8, all versions scale very well, but additional locality optimizations and vectorization in outputs generated by Pluto considerably improve performances. We measure the execution time of multiple parallel versions without loop transformation (i.e. a Naive program without loop tiling in this case) including two methods for eliminating column-major accesses. The first



(a) Naive program **(b)** Pluto output

Figure 4.7: Speed-ups over Default versions executed on 1 core for Covariance



(a) Naive program **(b)** Pluto output

Figure 4.8: Speed-ups over Default versions executed on 1 core for Gemm

version eliminates such accesses through data transposition, and the second version is obtained through loop interchange (ikj). Loop interchange is also the transformation that Pluto performs, in addition to loop tiling. On this machine, loop interchange seems more appropriate than data transposition on the Naive program. We reproduced these experiments on two other machines: a 4 core Intel Core i7-4910MQ CPU (Haswell) at 2.90GHz and a 16 core Intel Xeon CPU E5-2660 (Sandy Bridge) at 2.20GHz. We observed the same tendency on the Haswell with or without the `-O3` option, but we noticed the opposite on the Sandybridge when disabling this option. This is probably due to the lower AVX computation throughput compared to memory bandwidth: the program (with these optimizations) is definitely compute-bound on Sandybridge and NUMA optimizations have little impact.

Outcomes

These results show that, on one hand, bandwidth-bound programs that cannot be improved using Pluto's default heuristics (Gemver and Gesummv) do benefit from NUMA placements and transpositions. On the other hand, programs already benefiting a lot from Pluto's heuristics do not require additional NUMA placement since most data accesses hit the cache. Moreover, in some cases, and depending on the machine, it seems wiser to rely on loop transformations rather than data layout transformations.

These case studies show the advantages of complementing polyhedral tools with data placement on NUMA nodes and transposition. NUMA placements tend to improve a program's scalability, especially from the threshold from which the NUMA effect appears. In addition, transpositions helps to improve the speed-up in general. Combining both is an interesting alternative even though further optimizations are still necessary.

This far, our experiments involved the use of standard API constructs for data placements. In the prospect of an intermediate language featuring NUMA abstractions, expressiveness for both interleaved allocation and replication is required. But the API provides means to implement more precise NUMA operations. These could, therefore, be reflected in the language expressiveness. The following sections discuss possible refinements for data placement policies.

4.3 Refining NUMA Memory Allocation Policies

The implementation of **`numa_alloc_interleaved`** (and **`numa_alloc_onnode`**) is based on the combination of two main functions:

- **`mmap`** that creates a new mapping in the virtual address space of the calling process.
- **`mbind`** that complements **`mmap`** by setting the NUMA memory policy. It requires to choose, for a given memory range, an allocation policy mode for a certain number of nodes.

Listing 4.2 shows the implementation of **`numa_alloc_interleaved`** in Libnuma [1].

Regardless of the input program, interleaved allocation generally reduces traffic contention. However, to promote more data locality, we can explore the possibility of fine-tuning interleaved allocation with respect to a given program, i.e. compute the exact page granularity of interleaving that maximizes data locality. This requires to implement a variation of **`numa_alloc_interleaved`** thanks to **`mmap`** and **`mbind`**.

To compute the exact page granularity, we need to determine *thread array regions* that are obtained using the thread scheduling policy, with respect to the NUMA topology. Hypotheses for such statement implies that thread migration is deactivated and threads are binded to their respective cores.

[1] Source available at: https://github.com/numactl/numactl/blob/master/libnuma.c

```
1 void * numa_alloc_interleaved(size_t size) {
2     char *mem = mmap(0, size, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, 0, 0);
3     if (mem == (char *)-1)
4         return NULL;
5     if (mbind(mem, size, pol, bmp ? bmp->maskp : NULL, bmp ? bmp->size + 1 : 0,
  ↪  mbind_flags) < 0)
6         numa_error("mbind");
7     return mem;
8 }
```

Listing 4.2: Implementation of **`numa_alloc_interleaved`**

4.3.1 Thread array regions and page granularity

In OpenMP, the **`schedule(static, C)`** directive, where C is a chunk size, describes a round-robin distribution of the iteration range per chunks of C iterations among T threads (where T is the maximum number of threads executing the loop). The maximum value of the iteration range in noted N.

Let $M = N \times \frac{C}{T}$. The iteration range I_k of a thread k over a parallel dimension (inner dimensions excluded) is

$$I_k = \bigcup_{p=0}^{M} S_p, \quad S_p = \{i \mid C \cdot (p \cdot T + k) \leq i < C \cdot (p \cdot T + k) + C\}. \tag{4.1}$$

Further simplifications of I are possible depending on the type of distribution. Typically, a block distribution is specified with **`schedule(static, N/T)`**. In this case,

$$I_k = \{i \mid N \cdot \frac{k}{T} \leq i < N \cdot \frac{k}{T} + C\}. \tag{4.2}$$

A cyclic distribution produces a much simpler definition:

$$I_k = \bigcup_{p=0}^{M} S_p, \quad S_p = p \cdot T + k. \tag{4.3}$$

Let A an array and $i_1, \cdots, i_m$ the indexes of the loop iterating over A. We note $i_{//}$ the loop index iterating over a parallel dimension. We can determine the thread array region of a thread k using I:

$$TAR_k = \{A(i_1, \cdots, i_m) \mid \begin{cases} i_q \in I_k, \text{ if } i_q = i_{//} \\ i_q \in [0, max], \text{ otherwise} \end{cases} \}$$

where $[0, max]$ represents the full iteration range of i_q.

Let $ELTSIZE$ the size of elements in an array. We note $|TAR_k|$ the cardinal of TAR_k. If TAR_k is a full set, the page granularity for a given threadG_k is computed as follows:

$$G_k = \frac{|TAR_k| \times ELTSIZE}{PAGESIZE}.$$

If TAR_k is a disjoint set $G_k = g_1 \cup \cdots \cup g_m$ such that $g_1 = \cdots = g_m$, then

$$G_k = \frac{|g| \times ELTSIZE}{PAGESIZE}$$

where $g = g_1 \vee \cdots \vee g_m$.

Defining the final page granularity to be passed onto the allocation function is easy when $G_1 = \cdots = G_T$. However, approximations are necessary when TAR_k is a disjoint set and/or $G_1 \neq \cdots \neq G_T$.

Example: Block distribution

We assume $T = 4$, $N = 4096$, $sizeof = 8$ and $PAGESIZE = 4096$. We illustrate these principle using the first kernel of Gemver.

```
1 #pragma omp parallel for
2 for (i = 0; i < N; i++)
3  for (j = 0; j < N; j++)
4   A[i][j] += u1[i] * v1[j] + u2[i] * v2[j];
```

We focus on arrays A and $u1$ as their iteration ranges involve the parallel dimension. With no thread scheduling specification, OpenMP defaults to `schedule(static, N/T)` which is a block distribution. Computing all I_k following Equation 4.2:

For array A

$$TAR_0 = \{A(i,j) \mid i \in [0,1023] \text{ and } j \in [0,4095]\}$$
$$TAR_1 = \{A(i,j) \mid i \in [1024,2047] \text{ and } j \in [0,4095]\}$$
$$TAR_2 = \{A(i,j) \mid i \in [2048,3071] \text{ and } j \in [0,4095]\}$$
$$TAR_3 = \{A(i,j) \mid i \in [3072,4095] \text{ and } j \in [0,4095]\}$$

Therefore, $|TAR_0| = \cdots = |TAR_3| = 4194304$ and $G_A = 8192$ pages.

For array $u1$

$$TAR_0 = \{u_1(i) \mid i \in [0,1023]\}$$
$$TAR_1 = \{u_1(i) \mid i \in [1024,2047]\}$$
$$TAR_2 = \{u_1(i) \mid i \in [2048,3071]\}$$
$$TAR_3 = \{u_1(i) \mid i \in [3072,4095]\}$$

Therefore, $|TAR_0| = \cdots = |TAR_3| = 1024$ and $G_{u_1} = 2$ pages.

4.3.2 Implementation

A custom interleaving function is dependent on the type of core distribution across NUMA nodes. The example of implementation in Listing 4.3 tightly matches the cyclic distribution illustrated in Figure 4.2; this characteristic is emphasized from lines 21 to 25.

```
// bsize is the page granularity.
void *numa_alloc_block(size_t size, int bsize) {
 void *mem, *offset;
 struct bitmask *bmp;
 int nodeid;
 size_t i;

 bmp = numa_allocate_nodemask();
 mem = mmap(NULL, size, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0);

 nodeid = 0; // start with first NUMA node
 offset = mem;

 int lbsize = 0;
 numa_bitmask_setbit(bmp, nodeid);
 for (i = 0; i < size; i+=PAGESIZE) {
  mbind(offset, PAGESIZE, MPOL_BIND, bmp->maskp, bmp->size, 0);
  *((int *)offset) = 0;

  offset = (void *)((uintptr_t)offset + PAGESIZE);
  lbsize++;
  if (lbsize == bsize) {
   numa_bitmask_clearbit(bmp, nodeid);
   nodeid = (nodeid == 1) ? 0 : nodeid + 1;
   numa_bitmask_setbit(bmp, nodeid);
   lbsize = 0;
  }
 }
 numa_bitmask_clearbit(bmp, nodeid);
 return mem;
}
```

Listing 4.3: An implementation of a custom interleaving function following a cylic distribution of cores over 2 NUMA nodes.

4.3.3 Limitations

There are several limitations to this approach. First, we may not observe significant speed-up depending on the topology. While the goal of this methodology is to enhance data locality, it becomes irrelevant if the main issue of an application is traffic contention and there is no significant distance between nodes. In this case, page granularity does not matter. We performed a few experiments on the *Pau* machine (c.f. Figure 4.2) and observed variable results that did not allow us to conclude that the methodology is efficient.

Furthermore, these analyses need to meet several preconditions, especially concerning the thread binding. OpenMP proposes `OMP_PROC_BIND`, which seems to bind a thread k to a core k. This has been observed during experiments, however, in case of doubt, `GNU_CPU_AFFINITY` allows to precisely bind threads to cores.

Another limitation is that if N or C are not powers of 2, we need to approximate the page granularity. Finally, the application scope is restricted; typical programs on which this

could work are static control programs (i.e. programs that fit in the polyhedral model).

4.4 Data Replications Implementation

In this section, we discuss data replications. Note that we only consider replication on read-only variables; coherency management is required for write accesses, which is not in the scope of this thesis.

In *gemver*, we suggested that A, $u1$ and $u2$ could be interleaved due to the thread access pattern. Similarly, we can also assume that $v1$ and $v2$ could be replicated. Indeed,

$$TAR_k = \{v_1(j) \mid j \in [0, max_j]\}$$

where j is inner to the parallel dimension i.

4.4.1 Conditional Branching

In [130], we relied on a replication scheme dependent on the specification of conditions on parallel loop indexes. In Listing 4.4, we present the generated code following this idea.

```
#pragma omp parallel for
for (i = 0; i < n; i++) {
 if (0 <= i && i <  (n/N_n))
  for (j = 0; j < n; j++)
   A[i][j] += u1[i] * v1_1[j] + u2[i] * v2_1[j];

 if (n/N_n <= i && i < 2 * (n/N_n))
  for (j = 0; j < n; j++)
   A[i][j] += u1[i] * v1_2[j] + u2[i] * v2_2[j];

 ...
}
```

Listing 4.4: Implementation of a replication with conditional branching

The potentially excessive conditional branching is not appealing. With no parallelism, classic loop transformations such as hoisting out the `if` conditions or index splitting could be applied. The context of parallel programming introduces several challenges.

Challenge 1: Optimizations depend on the organization of cores.

The choice of transformations is tightly dependent on the organization of cores. In Listings 4.4 and 4.8, we present the replication on a *block* distribution of cores in the NUMA system. Therefore, an appealing optimization here is index splitting. If we consider a replication on a cyclic distribution of cores as in Figure 4.2, the replication can be rewritten as in Listing 4.5.

```
1  #pragma omp parallel for private(j) schedule(static,1)
2  for (i = 0; i < N; i++) {
3   if ((i % 2) == 0)
4    for (j = 0; j < N; j++)
5     A[i][j] += u1[i] * v1_1[j] + u2[i] * v2_1[j];
6
7   if ((i % 2) != 0)
8    for (j = 0; j < N; j++)
9     A[i][j] += u1[i] * v1_2[j] + u2[i] * v2_2[j];
10 }
```

Listing 4.5: Replication on a 2-nodes NUMA topology with cyclic distribution of cores

An optimized version could be as in Listing 4.6 where we apply a technique similar to loop tiling; the "tile" loop (i.e. the nesting level where t is the loop index) is, in fact, a loop iterating over the range of thread identifiers and the "point" loop is the former parallel dimension.

```
1  #pragma omp parallel for schedule(static, C) private(this_i, i, j)
2  for (t = 0; t < T; t++)
3   for (i = C*t; i < C*(t+1); i++) {
4    this_i = ((t % 2) * N)+i;
5    for (j = 0; j < _PB_N; j++)
6     A[i][j] = A[i][j] + vu1[this_i] * v1[j] + vu2[this_i] * v2[j];
7   }
```

Listing 4.6: Replication without conditional branching on a 2-nodes NUMA topology with cyclic distribution of cores

Some types of distributions will probably be left out. Perhaps could we assume that common patterns are cyclic or block distributions. But topologies remain unpredictable.

Challenge 2: The parallel programming language itself can be the hindering factor.

The semantics of the parallel programming language can be a limiting factor. Until now, we presented examples of OpenMP programs. However, we have encountered issues due to the semantics of the parallel section.

Considering Listing 4.4 where the replication follows the block distribution of cores, we mentioned earlier that index-splitting could be an optimization technique corresponding to Listing 4.7 a priori. The main idea is to create distinct parallel sections. Unfortunately, OpenMP does not provide enough user control over thread pools to ensure that Section 1 and Section 2 have different thread pools. Any `#pragma omp parallel` section starts the thread counter to 0. Even creating a parallel section using `#pragma omp sections` with inner parallel sections does not provide more flexibility. But contrary to OpenMP, Pthreads seems to propose means allowing such implementations.

```
// Section 1
#pragma omp parallel for
for (i = 0; i < n/N_n; i++)
 for (j = 0; j < n; j++)
  A[i][j] += u1[i] * v1_1[j] + u2[i] * v2_1[j];

// Section 2
#pragma omp parallel for
for (i = n/N_n; i < 2* n/N_n; i++)
 for (j = 0; j < n; j++)
  A[i][j] += u1[i] * v1_2[j] + u2[i] * v2_2[j];

...
```

Listing 4.7: Idealistic implementation of data replication with index-splitting

4.4.2 Replication Storage

This far, we considered cases where replications are stored in distinct arrays. But the storage mode impacts the implementation.

Alternatively to Listing 4.4, we could store all replication into the same array with an extra dimension which size corresponds to the number of replications. In this case, the generated code matches the implementation presented in Listing 4.8; this does not require to change access patterns.

```
#pragma omp parallel for
for (i = 0; i < n; i++)
 for (k = 0; k <= N_n; k++) {
  if (k * (n/N_n) <= i && i < (k+1) * (n/N_n))
   for (j = 0; j < n; j++)
    A[i][j] += u1[i] * v1_1[k][j] + u2[i] * v2_1[k][j];
    }
```

Listing 4.8: Implementation of replication with conditional branching and additional looping

Assuming an idealistic implementation of index splitting where it is possible to custom thread pools, a compact replication with a change on the access pattern could be performed as in Listing 4.9. Given an array $A(M)$, let $B = compactreplicate(A)$, resulting into $B(M*N)$ where $B(i) = B(i+k*M), 1 \leq k \leq N, 0 \leq i < M$.

The presented solutions can be, however, difficult to automatize. Various aspects need to be verified such as array bounds, thread identifiers or aligned array allocations. One last alternative, probably the easiest to implement and the most reliable, is as in Listing 4.10; we store replications in a two-dimensional array and dynamically find the proper node indexation. Unlike other variants that need to ensure that the thread binding is accurate,

```
#pragma omp parallel for
for (i = 0; i < n/N_n; i++)
 for (j = 0; j < n; j++)
  A[i][j] += u1[i] * new_v1[j] + u2[i] * new_v2[j];

#pragma omp parallel for
for (i = n/N_n; i < 2*n/N_n; i++)
 for (j = n; j < 2*n; j++)
  A[i][j] += u1[i] * new_v1[j] + u2[i] * new_v2[j];

...
```

Listing 4.9: Compact storage of replications

such a dynamic retrieval of node identification allows a portable implementation of `select`, but at the cost of function calls overheads.

```
#pragma omp parallel for
for (i = 0; i < n; i++) {
 k = get_node_current_thread();
 for (j = 0; j < n; j++)
  A[i][j] += u1[i] * new_v1[k][j] + u2[i] * new_v2[k][j];
}
```

Listing 4.10: Replication relying on dynamic retrieval of node identifiers.

4.5 On Run-time Decisions

Data placements are necessary but they may not be relevant without proper runtime decisions. In this section, we exhibit an example of interaction between a data placement policy and thread binding policies at runtime.

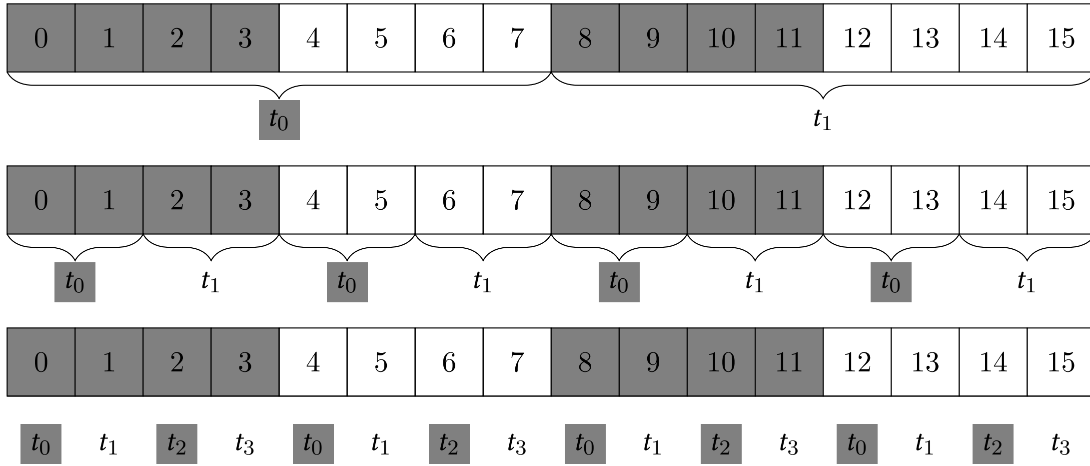
On platforms with a cyclic core distribution, an interleaved allocation policy lacking data locality may not be solvable through adjustable page sizes. Let us consider an array $u[16]$ and its mapping on a cyclic core distribution across 2 nodes. Assuming an interleaved allocation per 1 page size that contains only 4 elements, Figure 4.9a shows an example of a poor combination of work distribution and thread binding. It is appealing to readjust the thread binding with respect to data placement if means are provided for a fine control, and the programmer knows what he is doing. Otherwise, it is best to let the operating system decide on the proper migration of threads and pages.

Further adjustments at the static level may be performed as in Figure 4.9b with programming languages providing constructs for thread scheduling. This also allows the adjustment of thread workloads to some extent. For programming languages that do not provide such constructs, it is necessary to reason with the understanding of how the operating system takes decisions.

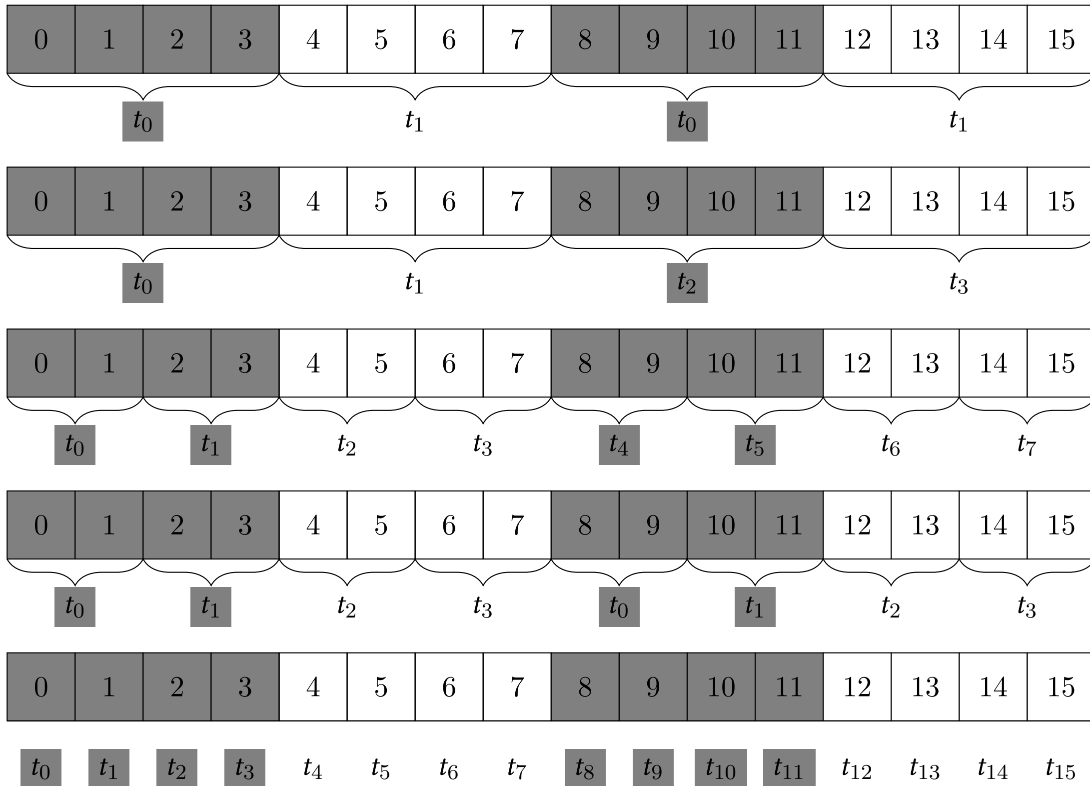
4.6 Outcomes

We presented NUMA architectures and how NUMA-awareness can complement loop and data layout transformations. NUMA-awareness must involve runtime decisions such as thread binding. However, in the context of static data placement, two main types of data allocation can be considered. On one hand, we have interleaved allocations that might be refined for further data locality if traffic contention is not the only issue. On the other hand, we have data replications and their implementation that may involve different storage mode and code transformations methods. It is therefore interesting to take into account, at least, abstractions for standard NUMA data placements. In addition, more abstractions can be proposed for further (and easier) placement exploration.

*** Background fill *gray* represents elements mapped on and threads bind to cores on node 0 (respectively node 1 with *white*). ***



(a) Example of poor combination resulting to unoptimized locality



(b) Combination for optimal data locality according to the number of threads.

Figure 4.9: Different combinations of work distribution and binding policies for a 1-dimensional array of size 16

Chapter 5

TeML: the Tensor Optimizations Meta-language

Nous présentons la conception et l'implémentation de TeML, un langage intermédiaire génératif et extensible doté de capacités de méta-programmation. TeML se distingue des autres méta-langages pour l'optimisation d'application tensorielles de par son expressivité de haut niveau et son degré de compositionnalité.

Le niveau d'abstraction adopté pour exprimer les calculs tensoriels est proche de celui utilisé par des outils tels que TensorFlow ou Theano. En outre, TeML permet de décrire en quelques lignes des schémas d'optimisations (qui peuvent être effectué par des outils tels que Pluto) à appliquer aux calculs tensoriels spécifiés. Nous fournissons également le moyen de caractériser des opérateurs tensoriels de haut niveau ainsi que des concepts liés au placement de données sur les architectures NUMA. De plus, sa conception basées sur une programmation de type fonctionnelle permet de composer et de générer plusieurs variantes de programme à partir du même programme TeML. Cette caractéristique permet à TeML d'être exploité dans la conception d'outils pour différents domaines d'applications tensorielles. Dans l'évalution de TeML, nous parvenons à démontrer que la flexibilité offerte permet d'exprimer des schémas d'optimisations plus efficaces que celles proposées par Pluto.

* ♣ * ♣ *

Our case study in Chapter 3 showed that tensor optimizations tools do not generalize well from one domain to another. An example of hindering factor is using as a front-end a language which expressiveness is irrelevant for a given domain. This is all the more true if the tool's compilation flow is domain-specific or tailored for restricted optimization techniques. In addition, there may be a lack of flexibility to extend such a tool.

Another issue, whether concerning domain-specific or more general solutions, is that there is real difficulty in composing complex transformations; after a few steps, it is hard to keep track of the sequence of transformations due to the core design of the intermediate representation used or the lack of formal semantics.

We, therefore, present the design and implementation of TeML, a generative and extensible intermediate language with meta-programming capabilities to close this gap. Although NUMA optimizations cannot be fully managed statically, as we have seen in Chapter 4 that it is beneficial to couple runtime solutions with static data placements, TeML also incorporate NUMA-awareness to mitigate the severe lack of high-level abstractions for NUMA placements.

This chapter is based on:

> Adilla Susungi, Norman A. Rink, Jerónimo Castrillón, Immo Huismann, Albert Cohen, Claude Tadonki, Jörg Stiller, and Jochen Fröhlich. ***Towards compositional and generative tensor optimizations***. In Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '17). `https://doi.org/10.1145/3136040.3136050`

> Adilla Susungi, Norman A. Rink, Albert Cohen, Jerónimo Castrillón, and Claude Tadonki. ***Meta-programming for Cross-Domain Tensor Optimizations***. In Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '18). `https://doi.org/10.1145/3278122.3278131`

5.1 Overview

TeML is a *transformation meta-language*, i.e. a language that has the ability to express program transformations using high-level constructs. This provides an appealing level of flexibility in composing sequences of transformations. Transformation meta-languages have an advantage over standard intermediate languages that make it difficult to compose and trace transformations paths. Therefore, they have been used in various contexts such as empirical tuning, interactive compilation or optimization meta-programming. Transformation meta-languages come in various forms such as scripting languages, intermediate languages, pragma-based or multi-stage and rely on various levels of abstraction for the input programs.

In our domain of interest, i.e. loop nests optimizations, a limited number of meta-languages exist that we can name: Poet [139], Goofi [95], X [61], URUK [53, 54], CHiLL [48, 113], Loo.py [80], Halide [110], XFOR [62], TVM [50], Lift [127], Clay [26]. Note that we can distinguish those that are based (at least to some extent) on the polyhedral model (e.g. URUK, CHiLL, XFOR, Clay) from those that are not (e.g. Halide, TVM, Lift). Loo.py is partially based on the polyhedral model. Also, despite being included among transformation meta-languages, Lift is in a different league: it relies on a traditional functional programming style using *combinators* and transformations are described through *rewriting rules*. Hence, contributions of this thesis relate to concepts in above-cited meta-languages *except* Lift.

TeML distinguishes itself from its predecessor on three main differences:

Program abstraction. In TeML, we propose high-level abstractions for tensor computations allowing to express and easily compose algebraic transformations with other types of transformations. Existing meta-language do not propose such abstractions. Those tightly based on the polyhedral model do not feature means for computation specification along with the sequence of transformations. Hence, they do not have expressiveness for composing algebraic transformations. On the other hand, meta-languages enabling both specification of computations and transformations propose constructs for simpler optimization problems, sometimes domain-specific.

Degree of compositionality. The second difference is that TeML integrates meta-programming capabilities following a unified functional design for program specification and transformation. This allows TeML to be much more compositional. Indeed, existing meta-language rely on an imperative composition of transformations and, when possible, a functional declaration of computations. Such level of unification and expressiveness is key for TeML's cross-domain applicability.

Architectural awareness. Finally, we incorporate some level of NUMA-awareness but not only through providing high-level constructs mapping to a NUMA API. We also propose abstractions concepts such as the type of program transformations inherent to data replications. This type of expressiveness does not exist in other languages.

TeML is designed with one goal in mind: modular and generative program specifications and transformations with unambiguous semantics. We achieve this by considering a program as a list of declarations using immutable functions. In addition, tensors have decoupled data layouts and NUMA allocation policy control. As shown in Figure 5.1, besides the declarative part of a TeML program, additional constructs for initialization, allocation and code generation specifications are available.

Declarations are performed using functions returning either tensors or loops through $\langle Texpression \rangle$ and $\langle Lexpression \rangle$ respectively. Returned objects may be stored in a *physical* or a *virtual* memory depending on the type of construct used: if a construct starts with `v` or `@`, then the returned object is stored in the virtual memory. The physical memory (the main memory) is a collection of memory blocks distributed over different NUMA nodes whereas the virtual memory is a single block. Most objects are stored in the main memory but the virtual memory specifically serves to store abstract views of objects in the main memory.

In the following section, we present different aspects of TeML's design.

5.2 Tensors

The underlying structure of a tensor is an N-dimensional array. In this section, we introduce different ways of declaring $N-$dimensional arrays.

```
⟨program⟩       ::= ⟨stmt⟩ ⟨program⟩
                  |  ϵ
⟨stmt⟩          ::= ⟨id⟩ = ⟨expression⟩
                  |  ⟨id⟩ = @⟨id⟩ : ⟨expression⟩
                  |  codegen (⟨ids⟩)
                  |  init (...)
⟨expression⟩    ::= ⟨Texpression⟩
                  |  ⟨Lexpression⟩
⟨Texpression⟩   ::= scalar ()
                  |  tensor ([⟨ints⟩])
                  |  eq (⟨id⟩, ⟨iters⟩? → ⟨iters⟩)
                  |  vop (⟨id⟩, ⟨id⟩, [⟨iters⟩?, ⟨iters⟩?])
                  |  op (⟨id⟩, ⟨id⟩, [⟨iters⟩?, ⟨iters⟩?] → ⟨iters⟩)
⟨Lexpression⟩   ::= build (⟨id⟩)
                  |  stripmine (⟨id⟩, ⟨int⟩, ⟨int⟩)
                  |  interchange (⟨id⟩, ⟨int⟩, ⟨int⟩)
                  |  fuse (⟨id⟩, ⟨id⟩, ⟨int⟩)
                  |  unroll (⟨id⟩, ⟨int⟩)
⟨iters⟩         ::= [⟨ids⟩]
⟨ids⟩           ::= ⟨id⟩ (, ⟨id⟩)*
⟨ints⟩          ::= ⟨int⟩ (, ⟨int⟩)*
```

Figure 5.1: TeML core grammar

5.2.1 N−dimensional Arrays

Basic declaration

A simple multi-dimensional array declaration is performed using:

- **`tensor`** $(T, [s_1, ..., s_N])$,

where T is the data type of the array elements and $[s_1, ..., s_N]$ are the sizes of the dimensions, i.e. the shape. We provide **`scalar`** (T) for pure scalar variables. However, scalars can also be viewed as 1-dimensional arrays with 1 element which can be declared using **`tensor`**.

Transpositions

We support two types of transposition for different purposes:

- **`transpose`**$(t, [r_1, r_2])$ for transposing the physical *or* virtual tensor t into a *physical* tensor,

- **vtranspose**(t, $[r_1, r_2]$) for transposing the physical *or* virtual tensor t into a *virtual* tensor.

Both operations produce a new tensor object in the intermediate language by swapping the dimensions r_1, r_2 of t.

Using either transpose or vtranspose have different impact on the generated code. The **transpose** construct generates a loop explicitly performing data transposition. As an example, we consider the following code sample.

```
A   = tensor(int, [N,N])
At = transpose(A, [1, 2])

# Computation using At
# ...
```

The C code generation from such a code includes the declaration of both A and A^T, the initialization of A followed by its transposed copy stored in A^T, then the use of A^T in the computation to eliminate column-major accesses.

```
// Declaration of both A and At
int A[N][N];
int At[N][N];

// Initialization of A
for (i = 0; i < N; i++)
 for (j = 0; j < N; j++)
  A[i][j] = ...

// Transposition into At
for (i = 0; i < N; i++)
 for (j = 0; j < N; j++)
  At[i][j] = A[j][i];

// Computation using At
for (i = 0; i < N; i++)
 for (j = 0; j < N; j++)
  ... = At[i][j];
```

However, vtranspose serves us to abstract and modify the access function of a tensor. Therefore, a tensor created with vtranspose will not appear in the generated code. For example, the following code sample

```
A   = tensor(2, int, [N,N])
At = vtranspose(A, [1, 2])

# Computation using At
```

generates a C code that only includes a declaration of A, then its transposed access in the computation as a result of a **vtranspose** specification in TeML.

$$\begin{pmatrix}0&0&0&0\\1&0&0&0\\0&1&0&0\\0&0&1&0\end{pmatrix}\begin{pmatrix}0&1&0&0\\0&0&1&0\\0&0&0&1\\0&0&0&0\end{pmatrix}\begin{pmatrix}1&\frac{1}{2}&\frac{1}{2}&\frac{1}{2}\\\frac{1}{2}&\frac{1}{3}&\frac{1}{4}&\frac{1}{5}\\\frac{1}{3}&\frac{1}{4}&\frac{1}{5}&\frac{1}{6}\\\frac{1}{4}&\frac{1}{5}&\frac{1}{6}&\frac{1}{7}\end{pmatrix}$$

Figure 5.2: From left to right: shift-lower, shift-upper, Hilbert matrices

```
// Declaration of A only
int A[N][N];

// Initialization of A
for (i = 0; i < N; i++)
 for (j = 0; j < N; j++)
  A[i][j] = ...

// Computation using transposed access to A
for (i = 0; i < N; i++)
 for (j = 0; j < N; j++)
  ... = A[j][i];
```

Special cases

If initialization of special cases is needed, dedicated constructs can be used:

- **zeros**(T, [s_1, ..., s_N])
- **ones**(T, [s_1, ..., s_N])
- **identity**(T, [s_1, s_2])
- **shift_lower**(T, [s_1, s_2])
- **shift_upper**(T, [s_1, s_2])
- **hilbert**(T, [s_1, s_2])

Such constructs use the same parameters as **tensor**. Besides **zeros** and **ones**, other specifications are restricted to matrices. Note that in the case of the lower- and upper- shift matrices, one type may be obtained with the transposition of its counterpart. For instance,

```
B = shift_upper(double, [4, 4])
```

corresponds to

```
A = shift_lower(double, [4, 4])
B = transpose(A, [1, 2])
```

Replications and multiplexing

In Chapter 4, Section 4.4, we exposed potential implementations of data replications. Therefore, NUMA-awareness consists in introducing appropriate program abstractions, in addition to allocation functions. The particularity of replications is the management of multiple copies; we need to choose within a parallel loop different copies of the same data structure, depending on the thread scheduling.

Replications can be performed using a simple assignment such as `t_1 = t`. We then handle the choice of different replications within a loop using virtual tensors carrying multiplexing properties; i.e. they point to concrete tensors that are used depending on conditions. For more clarity, let us recall an implementation of replications using conditional branching where v_{1_2} is a replication of v_{1_1} (resp. v_{2_2} and v_{2_1}).

```
1  #pragma omp parallel for
2  for (i = 0; i < n; i++) {
3   if (0 <= i && i <  (n/N_n))
4    for (j = 0; j < n; j++)
5     A[i][j] += u1[i] * v1_1[j] + u2[i] * v2_1[j];
6
7   if (n/N_n <= i && i < 2 * (n/N_n))
8    for (j = 0; j < n; j++)
9     A[i][j] += u1[i] * v1_2[j] + u2[i] * v2_2[j];
10 }
```

We then assume tensor V_1 to abstract the conditional use of v_{1_1} and v_{1_2} (resp. V_2 for v_{2_2} and v_{2_1}). This is the idea of the TeML construct used for this purpose, that is:

- `select`$(t_1, \cdots, t_n)$,

where $t_1, \cdots, t_n$ are replication to be mapped on different nodes. The number of arguments n is equal to the number of NUMA nodes considered. The TeML equivalent of the C code above is then:

```
1  v1_1 = tensor(double, [N, N])
2  v2_1 = tensor(double, [N, N])
3  v1_2 = v1_1
4  v2_2 = v2_1
5
6  V1 = select(v1_1, v1_2)
7  V2 = select(v2_1, v2_2)
8
9  // Computation using V1 and V2
10 // instead of v1_1, v1_2, v2_1 and v2_2
```

5.2.2 Compute Expressions

New tensors may be declared as the results of computations. Computations are defined as compositions of 3-address binary operations. The corresponding syntax in TeML is:

- `op(`$t_1, t_2, [[s_{1_1}, \cdots, s_{1_n}], [s_{2_1}, \cdots, s_{2_n}]]$ `->` $[s_{3_1}, \cdots, s_{3_n}]$`),`

where $[s_{1_1}, \cdots, s_{1_n}]$ and $[s_{2_1}, \cdots, s_{2_n}]$ are the subscripts of t_1 and t_2 respectively. The connector `->` indicates that the following subscript, here $[s_{3_1}, \cdots, s_{3_n}]$, should be associated to the output tensor. Tensor objects in TeML carry as an attribute the compute expression from which they result.

We use a naming convention to enforce a nesting order. For instance, a loop index i_k iterates over the dimension at depth k.

In order to compose beyond 3-address expressions, an `op` can be combined with their counterpart functions `vop` generating virtual tensors;

- `vop` $(t_1, t_2, [[s_{1_1}, \cdots, s_{1_n}], [s_{2_1}, \cdots, s_{2_n}]])$.

`vop` constructs return tensor images holding sub-expressions eventually expanded recursively at instances of `op`. Therefore, the shape of such virtual tensor is irrelevant. Furthermore, it is not necessary to use the `->` connector in a `vop`.

For both `op` and `vop` constructs, t_1 and t_2 may result from other `op` and `vop`. This means that if an input tensor is virtual, a subscript *should not* be associated with it, as they serve as temporary tensors. Therefore, we could redefine `op` and `vop` as:

- `op(`$t_1, t_2, [([s_{1_1}, \cdots, s_{1_n}]), ([s_{2_1}, \cdots, s_{2_n}])]$ `->` $[s_{3_1}, \cdots, s_{3_n}]$`)`
- `vop` $(t_1, t_2, [([s_{1_1}, \cdots, s_{1_n}]), ([s_{2_1}, \cdots, s_{2_n}])])$,

where the parenthesis around subscripts indicate that their specification is optional, with respect to the nature of their respective tensors.

Data replication through a simple assignment as proposed in the previous section implies the use of the C function `memcpy`. However, one can enforce copies to be performed through loops using `eq`, which enables layout manipulations:

- `eq(`$t, ([s_1, \cdots, s_n])$ `->` $[s_{1_1}, \cdots, s_{1_n}]$`).`

Using immutable functions enforces a single-assignment form for output dependencies. Consequently, we lose track of such re-computations at code generation. To avoid this, the keyword `@T` aliases an output tensor to T whenever preceding an operator. When generating code, any tensor that appears to be an alias is replaced by the original tensor.

5.2.3 Tensor Operations

Low-level arithmetic operators offer a high degree of expressiveness. But it is possible to characterize patterns in tensor computations and thus propose higher-level constructs.

```
1 A = tensor(double, [2, 2, 2])
2 B = tensor(double, [2, 2, 2])
3 C = mul(A, B, [[i1, i2, i3], [i1, i2, i3]] -> [i1, i2, i3])
```

```
1 for i1 from 0 to N1
2  for i2 from 0 to N2
3   for i3 from 0 to N3
4    C[i1][i2][i3] = A[i1][i2][i3] * B[i1][i2][i3]
```

(a) op construct

```
1 A = tensor(double, [2, 2, 2])
2 B = tensor(double, [2, 2, 2])
3 C = vmul(A, B, [[i1, i2, i3], [i1, i2, i3]])
4 D = mul(A, C, [[i1, i2, i3], ] -> [i1, i2, i3])
```

```
1 for i1 from 0 to N1
2  for i2 from 0 to N2
3   for i3 from 0 to N3
4    C[i1][i2][i3] = A[i1][i2][i3] * B[i1][i2][i3]
```

(b) vop and op constructs

```
1 A = tensor(double, [2, 2, 2])
2 B = eq(A, [i1, i2, i3] -> [i1, i2, i3])
```

```
1 for i1 from 0 to N1
2  for i2 from 0 to N2
3   for i3 from 0 to N3
4    B[i1][i2][i3] = A[i1][i2][i3];
```

(c) eq construct

```
1 A = mul(B, C, [[i1, i2, i3], [i1, i2, i3]] -> [i1, i2, i3])
2 A1 = @A:mul(B, D, [[i3, i2, i1], [i1, i2, i3]] -> [i3, i2, i1])
```

```
1 A[i1][i2][i3] = B[i1][i2][i3] * C[i1][i2][i3];
2 A[i3][i2][i1] = B[i3][i2][i1] * D[i1][i2][i3];
```

(d) Aliasing

Listing 5.1: Examples of TeML code samples and their corresponding C code

Tensor contraction. If, say, the 3rd dimension of A and the 1st dimension of B have the same size d, a contraction along these dimensions is

$$C_{ijlm} = \sum_{k=1}^{d} A_{ijk} \cdot B_{klm} \,. \tag{5.1}$$

The resulting tensor C is 4-dimensional. From Equation (5.1) it is clear that the contraction generalizes the matrix-matrix multiplication. Note, however, that one can contract along with any pair of dimensions of A and B, not only the last and first respectively, as long as the contracted dimensions have the same size.

Outer product. The outer product of A and B produces a 6-dimensional tensor C such that

$$C_{ijklmn} = A_{ijk} \cdot B_{lmn} \,. \tag{5.2}$$

Entry-wise operations. The entry-wise tensor multiplication of A and B produces a 3-dimensional tensor C:

$$C_{ijk} = A_{ijk} \, \langle op \rangle \, B_{ijk} \,. \tag{5.3}$$

where $\langle op \rangle$ is an arithmetic operation.

These operations are mapped directly to the following constructs in TeML:

- **`outerproduct`** $(t_1, \cdots, t_n)$,
- **`contract`** $(t_1, t_2, [[r_{1_1}, r_{2_1}], ..., [r_{1_n}, r_{2_n}]])$,
- **`entrywise_add`** $(t_1, \cdots, t_n)$, respectively **`_mul`**, **`_sub`** and **`_div`**.

The semantics of `contract` and its arguments are as follows. The last argument is a list of pairs, in which each pair $[r_{1_i}, \; r_{2_i}]$ specifies that the dimension r_{1_i} of t_1 and r_{2_i} of t_2 are contracted together. If t_1 is N-dimensional, then r_{1_i} can take numerical values ranging from 1 to N, and analogously for t_2. Note in particular that the order of the first two arguments of `contract` matters since the ranks in the last argument are tied to either t_1 or t_2.

The use of such high-level tensor operators can be illustrated through the different evaluations of *Interpolation*. Listings 5.2a–5.2c show the intermediate language code for the different variants of Interpolation introduced in Chapter 3, i.e. Equations 3.7–3.9 that we recall respectively:

$$v_{ijk} = \sum_{l,m,n} (A_{kn} \cdot (A_{jm} \cdot (A_{il} \cdot u_{lmn}))$$
$$v_{ijk} = \sum_{l,m,n} (A_{kn} \cdot A_{jm}) \cdot (A_{il} \cdot u_{lmn})$$
$$v_{ijk} = \sum_{l,m,n} (A_{kn} \cdot ((A_{jm} \cdot A_{il}) \cdot u_{lmn}))$$

Equations 3.8 and 3.9 each contain a term $(A \cdot A)$ that does not involve a contraction and therefore gives rise to the `outerproduct` in Figures 5.2b and 5.2c respectively.

```
1   A = tensor(double, [N, N])
2   u = tensor(double, [N, N, N])
3
4   tmp1 = contract(A, u, [[2, 1]])
5   tmp2 = contract(A, tmp1, [[2, 2]])
6   v = contract(A, tmp2, [[2, 3]])
```

(a) Equation 3.7

```
1   A = tensor(double, [N, N])
2   u = tensor(double, [N, N, N])
3
4   tmp1 = outerproduct(A, A)
5   tmp2 = contract(A, u, [2, 1])
6   v = contract(tmp1, tmp2, [[2, 3], [4, 2]])
```

(b) Equation 3.8

```
1   A = tensor(double, [N, N])
2   u = tensor(double, [N, N, N])
3
4   tmp1 = outerproduct(A, A)
5   tmp2 = contract(tmp1, u, [[2, 2], [4, 1]])
6   v = contract(A, tmp2, [[2, 3]])
```

(c) Equation 3.9

Listing 5.2: Intermediate language codes for equations

5.2.4 Initializations

Generally, tensor declarations must have their corresponding initialization using `init`. However, it is not required in specific cases, especially for tensor types distinguished by their layout or initialization values.

Replicated arrays need not have their corresponding `init` instance. Matrices such as the triangular or skew-symmetric would require explicit `init` instances but distinguishing them enforces an appropriate initialization pattern at code generation. On the other hand, special cases such as in Figure 5.2 have pre-defined initialization values and therefore do not require an explicit `init` instance. But they may be distinguished by their layout or initialization values. Languages such as Matlab or Scipy provide such expressiveness, which can, therefore, be part of TeML's expressiveness too.

5.3 Loop Generation and Transformations

The compute expression of tensors need to be expanded into concrete loop nests. To do so, `build` is the construct that is used:

- `build` (t),

taking as input a tensor t and returning a *loop*. In other words, the **build** construct is the bridge between tensor expression specifications and their corresponding loop objects on which transformations can be applied using dedicated constructs. Note that **build** cannot be used on simple N-dimensional arrays, i.e. generated with primitive or special case initialization constructs.

Transformation constructs generally take as parameters loops and integers denoting the dimension depth at which the transformation should occur. Given loops l, l_1 and l_2, and dimension depths d, d_1, d_2, we implemented primitives such as:

- **stripmine** (l, d, v), where v is the stripmining factor;
- **interchange** $(l, [d_1, d_2])$;
- **tile(l, v)**, where v is the tiling factor;
- **unroll** (l, d);
- **part_unroll** (l, d, v);
- **fuse** (l_1, l_2, d);
- **unroll_and_fuse** (l_1, l_2, d);
- **parallelize** (l, d, s), where s is a thread schedule (corresponds to OpenMP **schedule**);
- **vectorize** (l, d).

For illustration purposes, we provide a full picture of writing optimizations for *Inverse Helmholtz* (c.f. Chapter 3):

$$t_{ijk} = \sum_{l,m,n} A^T_{kn} \cdot A^T_{jm} \cdot A^T_{il} \cdot u_{lmn}$$

$$p_{ijk} = D_{ijk} \cdot t_{ijk}$$

$$v_{ijk} = \sum_{l,m,n} A_{kn} \cdot A_{jm} \cdot A_{il} \cdot p_{lmn}$$

Step 1: Declaration of tensors.

We declare A, u, and D.

```
1  A = tensor(double, [N, N])
2  u = tensor(double, [N, N, N])
3  D = tensor(double, [N, N, N])
```

We prepare the transposed view of A using a virtual transposition to express A^T.

```
1  At = vtranspose(A, 1, 2)
```

Having declared the arrays, we proceed with the tensor computations specifications. The contractions are decomposed as in Equation (3.7). We also express the entry-wise multiplication between D and $tmp3$ (see Equation (3.5)), where $tmp3$ is the last intermediate value of the first set of contractions. We use the **vtranspose** construct over a few tensors to match future loop interchanges.

```
tmp2t = vtranspose(tmp2, 1, 2)
tmp3t = vtranspose(tmp3, 1, 3)
tmp4t = vtranspose(tmp4, 1, 3)
tmp5t = vtranspose(tmp5, 2, 3)
tmp1 = contract(At, u, [2, 1])
tmp2 = contract(At, tmp1, [2, 2])
tmp3 = contract(At, tmp2t, [2, 3])
tmp4 = entrywise(D, tmp3t)
tmp5 = contract(A, tmp4t, [2, 1])
tmp6 = contract(A, tmp5t, [2, 2])
v = contract(A, tmp6, [2, 3])
```

Later we might want to apply transpositions to the first set of contractions, which would lead to transposed accesses of D. Therefore, we already declare a tensor Dt that is a transposed version of D.

```
Dt = transpose(D, 1, 3)
```

Step 2: Building the loops.

We then build the loops corresponding to the computations.

```
ld = build(Dt)
l1 = build(tmp1)
l2 = build(tmp2)
l3 = build(tmp3)
l4 = build(tmp4)
l5 = build(tmp5)
l6 = build(tmp6)
lv = build(v)
```

Step 4: Applying transformations.

We now apply transformations. Several dimensions can be interchanged in the loops computing $tmp1$, $tmp2$, $tmp3$, $tmp4$, and $tmp5$.

```
l1a = interchange(l1, [4, 3])
l1b = interchange(l1a, [4, 2])
l2a = interchange(l2, [2, 1])
l2b = interchange(l2a, [1, 4])
l3a = interchange(l3, [3, 2])
l3b = interchange(l3a, [1, 3])
l3c = interchange(l3b, [1, 2])
l3d = interchange(l3c, [1, 4])
l4a = interchange(l4, [3, 1])
l5a = interchange(l5, [3, 2])
```

```
1    for (i1 = 0; i1 <= 12; i1 += 1)
2    for (i2 = 0; i2 <= 12; i2 += 1)
3    for (i3 = 0; i3 <= 12; i3 += 1)
4    for (i4 = 0; i4 <= 12; i4 += 1)
5    tmp1[i1][i2][i3] += A[i4][i1] *
  ↪   u[i4][i2][i3];
6
7    for (i1 = 0; i1 <= 12; i1 += 1)
8    for (i2 = 0; i2 <= 12; i2 += 1)
9    for (i3 = 0; i3 <= 12; i3 += 1)
10   for (i4 = 0; i4 <= 12; i4 += 1)
11   tmp2[i1][i2][i3] += A[i4][i1] *
  ↪   tmp1[i2][i4][i3];
12
13   for (i1 = 0; i1 <= 12; i1 += 1)
14   for (i2 = 0; i2 <= 12; i2 += 1)
15   for (i3 = 0; i3 <= 12; i3 += 1)
16   for (i4 = 0; i4 <= 12; i4 += 1)
17   tmp3[i1][i2][i3] += A[i4][i1] *
  ↪   tmp2[i2][i3][i4];
18
19   for (i1 = 0; i1 <= 12; i1 += 1)
20   for (i2 = 0; i2 <= 12; i2 += 1)
21   for (i3 = 0; i3 <= 12; i3 += 1)
22   tmp4[i1][i2][i3] = D[i1][i2][i3] *
  ↪   tmp3[i1][i2][i3];
23
24   for (i1 = 0; i1 <= 12; i1 += 1)
25   for (i2 = 0; i2 <= 12; i2 += 1)
26   for (i3 = 0; i3 <= 12; i3 += 1)
27   for (i4 = 0; i4 <= 12; i4 += 1)
28   tmp5[i1][i2][i3] += A[i1][i4] *
  ↪   tmp4[i4][i2][i3];
29
30   for (i1 = 0; i1 <= 12; i1 += 1)
31   for (i2 = 0; i2 <= 12; i2 += 1)
32   for (i3 = 0; i3 <= 12; i3 += 1)
33   for (i4 = 0; i4 <= 12; i4 += 1)
34   tmp6[i1][i2][i3] += A[i1][i4] *
  ↪   tmp5[i2][i4][i3];
35
36   for (i1 = 0; i1 <= 12; i1 += 1)
37   for (i2 = 0; i2 <= 12; i2 += 1)
38   for (i3 = 0; i3 <= 12; i3 += 1)
39   for (i4 = 0; i4 <= 12; i4 += 1)
40   v[i1][i2][i3] += A[i1][i4] *
  ↪   tmp6[i2][i3][i4];
```

(a) Before transformations

```
1    for (i1 = 0; i1 <= 12; i1 += 1)
2    for (i2 = 0; i2 <= 12; i2 += 1)
3    for (i3 = 0; i3 <= 12; i3 += 1)
4    Dt[i1][i2][i3] = D[i3][i2][i1];
5
6    for (i1 = 0; i1 <= 12; i1 += 1)
7    for (i2 = 0; i2 <= 12; i2 += 1)
8    for (i3 = 0; i3 <= 12; i3 += 1)
9    for (i4 = 0; i4 <= 12; i4 += 1)
10   tmp1[i1][i4][i2] += A[i3][i1] *
  ↪   u[i3][i4][i2];
11
12   for (i1 = 0; i1 <= 12; i1 += 1)
13   for (i2 = 0; i2 <= 12; i2 += 1)
14   for (i3 = 0; i3 <= 12; i3 += 1)
15   for (i4 = 0; i4 <= 12; i4 += 1)
16   tmp2[i2][i4][i3] += A[i1][i2] *
  ↪   tmp1[i4][i1][i3];
17
18   for (i1 = 0; i1 <= 12; i1 += 1)
19   for (i2 = 0; i2 <= 12; i2 += 1)
20   for (i3 = 0; i3 <= 12; i3 += 1)
21   for (i4 = 0; i4 <= 12; i4 += 1)
22   tmp3[i3][i2][i4] += A[i1][i3] *
  ↪   tmp2[i4][i2][i1];
23
24   for (i1 = 0; i1 <= 12; i1 += 1)
25   for (i2 = 0; i2 <= 12; i2 += 1)
26   for (i3 = 0; i3 <= 12; i3 += 1)
27   tmp4[i3][i2][i1] = Dt[i3][i2][i1] *
  ↪   tmp3[i1][i2][i3];
28
29   for (i1 = 0; i1 <= 12; i1 += 1)
30   for (i2 = 0; i2 <= 12; i2 += 1)
31   for (i3 = 0; i3 <= 12; i3 += 1)
32   for (i4 = 0; i4 <= 12; i4 += 1)
33   tmp5[i1][i3][i2] += A[i1][i4] *
  ↪   tmp4[i2][i3][i4];
34
35   for (i1 = 0; i1 <= 12; i1 += 1)
36   for (i2 = 0; i2 <= 12; i2 += 1)
37   for (i3 = 0; i3 <= 12; i3 += 1)
38   for (i4 = 0; i4 <= 12; i4 += 1)
39   tmp6[i1][i2][i3] += A[i1][i4] *
  ↪   tmp5[i2][i3][i4];
40
41   for (i1 = 0; i1 <= 12; i1 += 1)
42   for (i2 = 0; i2 <= 12; i2 += 1)
43   for (i3 = 0; i3 <= 12; i3 += 1)
44   for (i4 = 0; i4 <= 12; i4 += 1)
45   v[i1][i2][i3] += A[i1][i4] *
  ↪   tmp6[i2][i3][i4];
```

(b) After transformations

Listing 5.3: Generated codes for *Inverse Helmholtz*

5.4 Memory Allocations

TeML features several allocation policy mapping to the C language and corresponding API calls:

- `alloc` (t) for standard allocation corresponding to `malloc`;
- `alloc_align` (t, n) for data alignment with a factor of n, corresponding to `_mm_malloc`;
- `alloc_interleaved` (t) for interleaved allocation on NUMA nodes corresponding to `numa_alloc_interleaved`. Refined allocations can be performed with `alloc_interleaved` (t, v) where v is a value denoting the number of pages to be mapped per round robin tour. This corresponding to our custom `numa_alloc_block` function;
- `alloc_onnode` (t, n) for allocation on a specific node n corresponding to `numa_alloc_onnode`.

Listing 5.4 depicts a TeML implementation of NUMA placements for the first kernel of Gemver.

5.5 Implementation and Code Generation

In this section, we provide more insight about the TeML-to-C process. We generate C code using the command `python tmc.py input_file.tml`. The `tmc.py` file implements 3 stages of processing: parsing, full syntax tree processing, and code translation.

Parsing

Most of TeML's syntax matches that of Python. Therefore we use RedBaron [13], a parser for Python programs, to process TeML programs. As some TeML elements are not part of Python's syntax, we first pre-process our inputs to transform them into fully Python-compliant programs. A typical example is the connector `->` which is transformed into a comma.

Redbaron generates the full syntax tree (FST) of the program in which different types of nodes can be distinguished (cf. Figure 5.3). Our two main nodes of interests are *assignment* and *atomtrailers* nodes; all declarations are captured as assignment nodes whereas memory allocations, initializations, and code generation instructions are captured as atomtrailers nodes.

Full syntax tree processing

The FST processing consists into creating tensor and loop data structures out of assignment nodes.

```
A = tensor(double, [n, n])
u1 = tensor(double, [n])
v1 = tensor(double, [n])
u2 = tensor(double, [n])
v2 = tensor(double, [n])

u1_1 = u1
u1_2 = u1
u1_3 = u1
u2_1 = u2
u2_2 = u2
u2_3 = u2

U1 = select(u1, u1_2, u1_3, u1_4)
U2 = select(u2, u2_2, u2_3, u1_4)

t1 = vmul(U2, v2, [[i1], [i2]])
t2 = vmul(U1, v2, [[i1], [i2]])
t3 = vadd(t1, t2)
t4 = @A:add(A, t3, [[i1, i2], ] -> [i1, i2])

alloc_interleaved(A)
alloc_onnode(u1, 0)
alloc_onnode(u1_1, 1)
alloc_onnode(u1_2, 2)
alloc_onnode(u1_3, 3)
alloc_onnode(u2, 0)
alloc_onnode(u2_1, 1)
alloc_onnode(u2_2, 2)
alloc_onnode(u2_3, 3)
```

Listing 5.4: An example of NUMA placement in Gemver using TeML

The main attributes of a tensor data structure are the name, the data type, the shape, the compute expression returning the tensor and its ancestor (which is another tensor from which the current is derived).

Each tensor instruction returns a new data structure with attributes filled with respect to the operation used. This tensor object is then added to either a list of concrete tensors and a list of virtual tensors. If an attribute does not apply, it is left to the nil value. Tensor initialization characteristics and allocation policies are updated after the creation of data structures.

Loop structures are linked list of loop dimensions. We also maintain a list of all loops, whether created with **`build`** or transformations. However, while **`build`** and **`fuse`** instantiate new loop structures, other transformation constructs return altered copies of their input loops.

```
1  8 ----------------------------------------------------
2  AssignmentNode()
3  # identifiers: assign, assignment, assignment_, assignmentnode
4  operator=''
5  target ->
6  NameNode()
7  # identifiers: name, name_, namenode
8  value='l'
9  value ->
10 AtomtrailersNode()
11 # identifiers: atomtrailers, atomtrailers_, atomtrailersnode
12 value ->
13 * NameNode() ...
14 * CallNode() ...
15 9 ----------------------------------------------------
16 EndlNode()
17 # identifiers: endl, endl_, endlnode
18 value='\n'
19 indent=''
20 10 ----------------------------------------------------
21 EndlNode()
22 # identifiers: endl, endl_, endlnode
23 value='\n'
24 indent=''
25 11 ----------------------------------------------------
26 AtomtrailersNode()
27 # identifiers: atomtrailers, atomtrailers_, atomtrailersnode
28 value ->
29 * NameNode()
30 # identifiers: name, name_, namenode
31 value='init'
32 * CallNode()
33 # identifiers: call, call_, callnode
34 value ->
35 * CallArgumentNode() ...
36 * CallArgumentNode() ...
```

Figure 5.3: A RedBaron FST sample

Code translation

The **codegen** construct takes as input a list of loops. Such loops may be results of simple builds or sequences of transformations. We, therefore, assume loops to be passed onto **codegen** to be ready for direct translation into C code. Of course, prior to the loops, we also print tensor declarations, allocations and initializations.

5.6 On Data Dependencies

The question of data dependencies is inherent to the principles of loop transformations. Throughout the thesis work, dependency management has been thought of in various ways

that we discuss in this section. We first recall dependency definitions.

Definition 5.6.1 (Dependency)**.** *Let O_1 and O_2 be two operations. There is a dependency from O_1 to O_2, noted $O_1 \rightarrow O_2$, if O_1 is executed before O_2 and they both have access to the same memory location.*

Different types of dependencies exist:

- Read-after-write (flow dependency), when O_1 writes a datum read by O_2;
- Write-after-read (anti dependency), when O_1 read a datum later written by O_2;
- Write-after-write (output dependency), when O_1 and O_2 write the same datum.

In the context of loop executions, the Definition 5.6.1 may be considered at different granularity: inter- and intra-loop dependencies. Different stages of TeML's design greatly influenced our vision of dependency management.

5.6.1 Dependency Checks using Explicit Construct

The early design of TeML[1] was much less functional. We represented the input program using an abstraction closer to imperative nested loops, that is,

```
1 with i as piter:
2  with j as siter:
3   A[i][j] = k1(A[i][j], u1[i], v1[j], u2[i], v2[j])
```

were we:

- used a **with** construct accompanied with **as piter** or **as siter** to qualify declared loop indexes as used in a parallel or a sequential loop, respectively;
- abstracted statements as functions performing element-wise operations and kept implicit the actual arithmetic operators[2].

With such a design, the order in which loops are specified directly determines (at least part of) inter-loop dependencies.

As for intra-loop dependencies, we considered explicit constructs carrying meta-informations. For instance, the following code sample exposes different types of dependencies.

```
1 with i1 as siter:
2  A[i1] = f(B[i1])
3
4 with i2 as siter:
5  with i3 as siter:
6   C[i2] = f(C[i2], A[i3])
```

[1]That is, Ivie in [130]
[2]Still preserved in the compilation flow thanks to pieces of information from the input code

We can identify:

- RAW dependencies from the initialization of B to B[i1], from A[i1] to A[i3], from the initialization of C to C[i2], then from C[i2] to C[i2];
- A WAW dependency from C[i2] to C[i2];
- A WAR dependency from C[i2] to C[i2].

Consequently, we considered representing dependencies as follows:

```
with i1 as siter:
 A[i1] = f(B[i1])

B[i1].__raw(_INIT_)

with i2 as siter:
 with i3 as siter:
  C[i2] = f(C[i2], A[i3])

A[i3].__raw(A[i1])
C[i2].__raw(_INIT_, C[i2])
C[i2].__waw(C[i2])
C[i2].__war(C[i2])
```

Such an idea could be used in the current state of TeML since subscripts may be exposed when using an `op` construct. However, additional features would be required to capture dependencies across high-level tensor operators in which subscripts are implicit.

5.6.2 Decoupled Management

With the current design of TeML, we find the following approach to dependencies more convenient.

Inter-loop dependencies We are interested in inter-loop dependencies, especially for the general data-flow. In this case, it is most importantly read-after-write dependencies that we need. Dependencies at this level can be represented as a data-flow graph $G = (N, E)$ where:

- N is the set of nodes representing tensor computations;
- E is the set of edges denoting the data-flow.

Intuitively, such a graph is built using the following rule. Each instance of `build` creates a new node in G. If $G \neq \emptyset$, an edge from an existing node n to the new node n' is created if the output of the computation represented in n is an input to that of n'.

A finer granularity for dependencies may then be attached to each node.

```
for i in range(0, N):
 for j in range(0, N):
  A(i) += A(i+1) * A(i) * A(i-1)
```

	Dependency	Direction vector	Combinator vectors
D_1	A(i) - A(i+1)	(>, =)	(seqpos, reduce)
D_2	A(i) - A(i-1)	(<, =)	(seqneg, reduce)
D_3	A(i) - A(i)	(=, =)	(map, reduce)

Figure 5.4: A loop and its dependency informations

Intra-loop dependencies Dependencies within a loop concern operations between instances of loop execution, i.e. precise computation of tensor elements. At this level, in addition to what type of dependency occurs, we are interested in the direction of the dependency.

Definition 5.6.2 (Direction vector)**.** *Let a loop l of depth n. Let O_1 and O_2, two dependent operations in l occurring respectively at instance $I = (i_1, \cdots, i_n)$ and $J = (j_1, \cdots, j_2)$. The direction vector $D(I, J) = (d(i_1, j_1), \cdots, d(i_n, j_n))$ such that*

$$d(i_k, j_k) = \begin{cases} >, \text{if } j_k - i_k > 0 \\ <, \text{if } j_k - i_k < 0 \\ =, \text{if } j_k - i_k = 0 \end{cases} \tag{5.4}$$

We may collect all dependencies in a loop to form the *direction matrix.*

Definition 5.6.3 (Direction matrix)**.** *Let n be the depth of a loop and m the number of dependencies. The direction matrix is a matrix M of size $n \times m$ where $M_i = D_i(I, J), i < m$.*

We adapt the notion of direction vector to our needs. The notations ">", "<", and "=" can be substituted with combinators which semantics respect the directions. Consequently, "=" is substituted with two type of combinators: *map* and *reduce.* We can preserve ">" and "<" with their respective substitute *seqpos* and *seqneg.* However, if there is no need to distinguish both directions, we use *seq.* Using combinators is especially interesting to formalize effects on different types of parallel loops.

Transformation effects on direction matrices

We assume that no transformation exposing parallelism can be performed. (In any case, we do not support skewing.)

Strip-mining, unrolling, peeling, parallelization and vectorization do not modify the direction matrix. Loop interchange, however, permutes it. We consider loop fusion to recreate a direction matrix for the newly form loop.

Determining preservation of dependencies

As transformations can be freely applied regardless of legality, semantic equivalence of code is to be ensured at code generation. We do not support operations exposing task parallelism explicitly. Therefore, we assume the task parallelism to be already exposed. The list of loop provided at `codegen` are generated in the order dictated by the initial builds steps. Indeed, when deriving new loops from transformations over old loops, pieces of information on the flow are preserved. However, within a loop, we still need to establish that (i) the list of dependencies is preserved and (ii) the direction matrices respect the original ones.

5.7 Evaluation

Table 5.1 describes a range of common tensor kernels from different application domains. These kernels serve as benchmarks for our evaluation of TeML that assess three different aspects of the language:

1. The capability to express tensor computations efficiently. Here we compare with TensorFlow [20], whose abstractions for tensor expressions are similar to TeML's tensor operations from Section 6.2.2.
2. The ability to reproduce loop optimization paths of existing tools. It is natural to compare with Pluto [39], which gives access to powerful polyhedral-based transformations through an interface allowing to enable or disable transformation heuristics.
3. The ability to easily extend optimization paths by composing with additional transformations, leading to the generation of C programs that outperform the ones generated with Pluto.

As this section unfolds, it will become clear that not all kernels from Table 5.1 can be meaningfully used in the evaluations of all of the three aspects.

All experiments reported in this section were performed on an Intel(R) Core(TM) i7-4910MQ CPU (2.90GHz, 8 cores when hyperthreading is enabled, 8192 KB of shared L3 cache) running the Ubuntu 16.04 operating system. The generated C programs (either from TeML or Pluto) were compiled with the Intel C compiler ICC 18.02 using the flags `-O3 -xHost -qopenmp`. We used Pluto version 0.11.4, and TensorFlow version 1.6 with support for AVX, FMA, SSE, and multi-threading.

5.7.1 Expressing Tensor Computations

Most benchmark kernels can be implemented using TeML's more abstract tensor operations, which keeps the kernel code short. As shown in Table 5.2, TensorFlow, and TeML programs are of comparable sizes in terms of lines of code (LOC). We provide a full implementations of *sddmm* in Figure 5.5.

Most TensorFlow kernels use either `einsum` or `tensordot`. The latter is the equivalent of TeML's `contract`. TensorFlow's `einsum` operation is more low-level than `tensordot`

Table 5.1: Kernels used in experiments and their respective application domains

	Name	Domain
Matrix Multiplication	*mm*	Linear algebra
transposed	*tmm*	Deep learning
batch	*bmm*	
Grouped Convolutions	*gconv*	
Matricized Tensor Times Khatri-Rao product	*mttkrp*	Machine learning
Sampled Dense-Dense Matrix Product	*sddmm*	Data analytics
Interpolation	*interp*	Fluid dynamics
factorized	*finterp*	
Helmholtz	*helm*	
Blur	*blur*	Image processing
Coarsity	*coars*	

Table 5.2: Comparison of benchmarks implementations in Tensorflow and TeML

	Tensorflow		**TeML**	
	N° lines	Constructs used	N° lines	Constructs used
Matrix Multiplication	3	matmul	3	contract
transposed	3	matmul:transpose=True	4	transpose, contract
batch	3	einsum or tensordot	3	contract
Grouped Convolutions	N/A	Not implemented. Incompatible with einsum.	5	vmul, add
Matricized Tensor Times Khatri-Rao product	4	einsum or tensordot, multiply	5	vcontract, contract
Sampled Dense-Dense Matrix Product	4	einsum or tensordot, multiply	6	vcontract, entrywise_mul
Interpolation	3	einsum or tensordot	5	vcontract, contract
factorized	6	einsum or tensordot	6	contract
Helmholtz	N/A	Required division not well supported.	9	contract, entrywise_mul, div, outerproduct
Blur	N/A	No stencil support.	64	op, vop
Coarsity	6	einsum or multiply, subtract	6	ventrywise_mul, entrywise_sub

and can thus be used when the semantics of **tensordot** are too restrictive, e.g. in batched matrix multiplication. In TeML, the same functionality can be implemented using also low-level operations, i.e. **add** and **mul**. Batched matrix multiplication ($C = AB$) must be built with more low-level operations in both frameworks due to the batch index b:

```
1 C[b][i][j] += A[b][i][k] * B[b][k][j]
```

Note that **einsum** is not flexible enough to implement all kernels that can be expressed in TeML, as evidenced by the convolution kernel *gconv* and the stencil kernel *blur* in Table 5.1. Tensorflow does also not have a dedicated construct for outer products, but its documentation explicitly recommends using **einsum** for this purpose.[3]

In summary, practically all of TensorFlow's constructs for building tensor expressions have equivalents in TeML that can be used as effectively. Moreover, kernels that are not well-supported by TensorFlow, e.g. the stencil kernel *blur*, can be expressed in TeML using its fundamental arithmetic operations. Extending TeML with a more abstract operation for stencils is left for future work.

5.7.2 Reproducing Loop Optimization Paths

For the tensor kernels from Table 5.1 we have identified the fastest program variants that can be generated with Pluto by manipulating its heuristics for loop fusion, tiling, interchange, vectorization, and thread parallelism. Table 5.3 lists the TeML equivalents of the loop optimization paths that caused Pluto to generate the fastest programs. Note that the TeML transformations **vectorize** and **parallelize** have been implemented with compiler-specific pragmas for vectorization and OpenMP pragmas for thread parallelization.

The stencil kernel *blur* is not included in Table 5.3 since Pluto's best optimization path for this kernel performs loop skewing, which cannot yet be expressed in TeML. Also, as standard matrix multiplication has been thoroughly studied, our analysis focuses on *bmm*, which presents a less conventional computation pattern.

Since the sequences of TeML loop transformations in Table 5.3 reproduce the effects of Pluto's optimizations, C programs generated either from TeML or Pluto for the kernels listed in Table 5.3 have equal execution times. For space reasons we have omitted plots of these execution times since they would only show relative speed-ups of 1.0× (within measurement accuracy) between Pluto and TeML.

5.7.3 Performance

We now study opportunities for composing optimization paths with TeML that lead to generated C programs that outperform programs generated with Pluto. For completeness, we also indicate the performance of the corresponding TensorFlow kernels. Figure 5.6 shows

[3] https://www.tensorflow.org/api_docs/python/tf/einsum

```python
import tensorflow as tf
from tensorflow.python.client import timeline
import sys

B = tf.random_normal([4096, 4096])
C = tf.random_normal([4096, 4096])
D = tf.random_normal([4096, 4096])

tmp = tf.tensordot(C, D, [[1], [0]])
A = tf.multiply(B, tmp)


T = int(sys.argv[1])

sess = tf.Session(config=tf.ConfigProto(intra_op_parallelism_threads=T))

options = tf.RunOptions(trace_level=tf.RunOptions.FULL_TRACE)
run_metadata = tf.RunMetadata()
sess.run(A, options=options, run_metadata=run_metadata)
```

(a) TensorFlow

```
B = tensor(double, [4096, 4096])
C = tensor(double, [4096, 4096])
D = tensor(double, [4096, 4096])

tmp = vcontract(C, D, [[2, 1]])
A = entrywise_mul(B, tmp)

l = build(A)

init(B, 1)
init(C, 1)
init(D, 1)
init(A, 0)

codegen([l])
```

(b) TeML

Figure 5.5: Implementations of *sddmm* which computation corresponds to `A(i,j) = B(i,j) * C(i,k) * D(k,j)` in index notation.

Table 5.3: Equivalents of Pluto optimization paths in TeML. Kernel data sizes in parentheses

mttkrp (250*250*250)	*sddmm* (4096*4096)	*bmm* (8192*72*26)	*gconv* (32*32*32*32*7*7)	*interp* (50000*7*7*7)	*helm* (5000*13*13*13)	*coars* (4096*4096)
parallelize(l, 1) interchange(l, 2, 3)	interchange(l, 2, 3), parallelize(l, 1), vectorize(l, 3)	tile(l, 32) interchange(l, 7,8) paralellize(l, 1) vectorize(l, 8)	interchange(l1, 4, 5) interchange(l1, 5, 6) parallelize(l1, 1) vectorize(l1, 9) paralellize(l2, 1) vectorize(l2, 9)	interchange(l1, 4, 5), vectorize(l1, 5), interchange(l2, 4, 5), vectorize(l2, 5), parallelize(l1, 1), parallelize(l2,1), parallelize(l3, 1)	fuse_outer(l4, l5, 5), fuse_outer(l4, l6, 5), parallelize(l1, 1), parallelize(l2, 1), parallelize(l3, 1), parallelize(l4, 1), vectorize(l1, 2), vectorize(l2, 3), vectorize(l3, 4)	tile(l, 32) parallelize(l, 1) vectorize(l, 4)

speed-ups relative to the sequential C implementation that is the starting point for applying optimizations with Pluto. Speed-ups are shown for the best program variants generated with Pluto and TeML, and for TensorFlow kernels whenever they exist (cf. Table 5.1).

Pluto's best variants for *mttkrp*, *bmm*, and *sddmm* still offer opportunities for data transposition, which can significantly improve performance if copy overheads are negligible. The relevant loop nest in the *mttkrp* kernel is this:

```
1  for (int i = 0; i <= (I-1); i++)
2   for (int j = 0; j <= (J-1); j++)
3    for (int k = 0; k <= (K-1); k++)
4     for (int l = 0; l <= (L-1); l++)
5      A[i][j] = B[i][k][l] * D[l][j] * C[k][j];
```

The loop interchange $j \leftrightarrow l$ would eliminate the column-major access of D, and $j \leftrightarrow k$ would eliminate that of C. Both column-major accesses cannot be eliminated at the same time (without negatively affecting the access patterns of A and B). Pluto chooses to interchange $j \leftrightarrow k$ (cf. Table 5.3), which only eliminates the column-major access of C. The following TeML meta-program implements *mttkrp* and resolves the column-major access of D by means of data transposition, yielding a speed-up of $1.74\times$ compared to Pluto, with the cost of transposition included (Figure 5.6a).

```
1  B = tensor(double, [250, 250, 250])
2  C = tensor(double, [250, 250])
3  D = tensor(double, [250, 250])
4  Dt = transpose(D, [[1, 2]])
5  tmp = vcontract(B, Dt, [3, 1])
6  A = contract(tmp, C, [2, 1])
7  ld = build(Dt)
8  l = build(A)
9  l1 = parallelize(l, 1)
10 l2 = interchange(l1, [2, 3])
```

For *bmm*, however, when executing on more than two cores, the copy overhead that results from the transposition loop is greater than the cost of transposed data accesses, making it

difficult to outperform Pluto (Figure 5.6b). Similarly, an additional transposition makes *sddmm*'s execution about 2× slower. When TeML does not apply the additional transposition to the *bmm* and *sddmm* kernels, performance is on par with the generated Pluto program as explained in Section 5.7.2. This shows that initially appealing transformations do not always guarantee a gain in performance.

Sequences of contractions are central to both the *interp* and the *helm* kernel. In *interp*, the C implementation of the first contraction is as follows.

```
1 for (int i1 = 0; i1 <= (N-1); i1++)
2  for (int i2 = 0; i2 <= (N-1); i2++)
3   for (int i3 = 0; i3 <= (N-1); i3++)
4    for (int i4 = 0; i4 <= (N-1); i4++)
5     t[i1][i2][i3] += A[i1][i4] * u[i4][i2][i3];
```

Pluto chooses to permute the loops into `i1`, `i2`, `i4`, `i3` . However, all transposed accesses are eliminated if the loops are ordered into `i1`, `i4`, `i2`, `i3` with TeML. Figure 5.6e shows that this potentially yields slightly better performance. Applying the analogous permutation to *helm* inhibits the fusions that Pluto chooses to apply (cf. Table 5.3) but leads to noticeably better performance than Pluto's fusions (cf. Figure 5.6f).

Loop unrolling considerably speeds up *gconv* (Figure 5.6d) as small inner dimensions lend themselves well to full unrolling. While Figure 5.6d looks promising, one has to be cautious when comparing with Pluto since it cannot apply an unrolling heuristic analogous to TeML.

For the *coarsity* kernel we were unable to identify transformations that outperform Pluto's heuristics.

Concerning TensorFlow

Performance comparison against TensorFlow, in Figure 5.6, is not fair. Unlike Pluto and TeML, TensorFlow does not let the programmer configure loop optimizations to be applied to a kernel flexibly. Instead, TensorFlow operators are optimized atomically thus missing possible cross-operator optimizations opportunities [108]. Another point is, since TensorFlow is targeted at the machine learning domain, it is not reasonable to expect it to perform well at optimizing kernels from other application domains such as *interp*. Finally, the version used may influence the results. As explained in the experimental setup, the TensorFlow version used is a version built with AVX and multi-threading support. Had we built a version with XLA, different results could have been obtained. Unfortunately, we were unable to properly built a version with both AVX and XLA support.

Comparing against other APIs

Performance comparisons with other APIs such as Halide or TVM would require this work to be more focused on compilation techniques. In this case, performance evaluation on GPUs is mandatory since most of the presented benchmarks would typically run on GPUs. At this

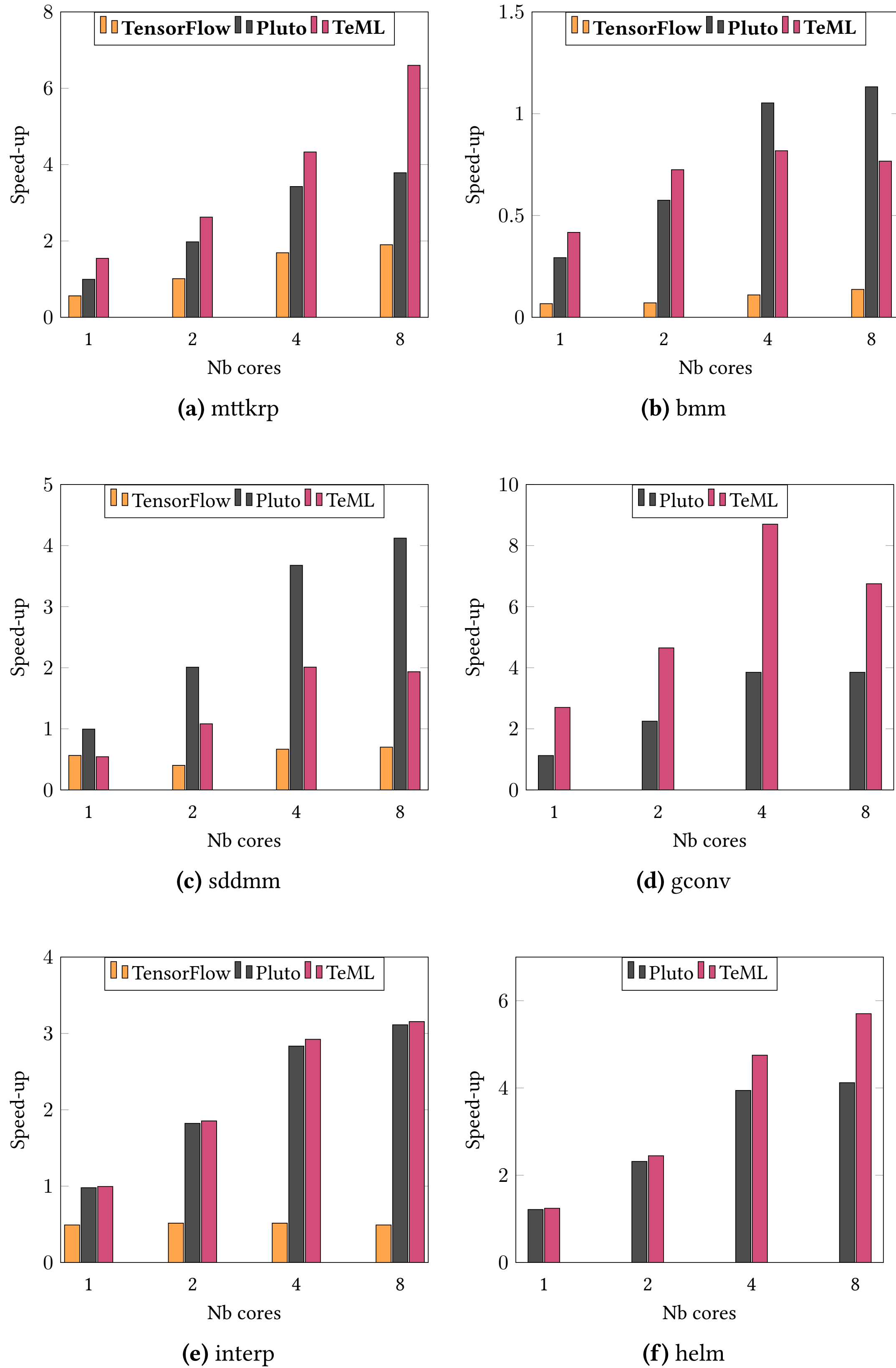


Figure 5.6: Speed-ups relative to sequential (i.e. single-core) C implementations

point in our work we are more focused on language design and expressiveness. Hence the focus of our experiments on expressiveness comparisons with Tensorflow and the capability at least reproduce performances of programs generated by Pluto to ensure that TeML meets this essential requirement.

5.8 Conclusion

This chapter provides an informal description of TeML. TeML distinguishes itself from other transformation meta-languages for loop nest optimizations by its level of abstraction and degree of compositionality. Its level of abstraction is close in spirit to languages such as TensorFlow or Theano and allows describing, in a few lines, optimization schemes of Pluto, a state-of-art polyhedral compiler, as well as more specific, platform- or input-dependent ones. We provide means to abstract high-level tensor operators as well as NUMA-related concepts. Furthermore, its highly compositional design allows to compose and generate multiple program variants from the same TeML (meta-)program.

Chapter 6

Formal Specification of TeML

La sémantique de TeML est expliquée au moyen de représentations arborescentes permettant de raisonner sur les interactions entre les expressions tensorielles et les transformations de boucles. La représentation arborescente d'une expression tensorielle permet de capturer des informations telles que les dimensions du tenseur ou encore le pattern de calcul auquel le tenseur est associé. Celle des nids de boucles, quant à elle, permet de capturer les informations sur les itérateurs de boucles ainsi que les expressions tensorielles desquelles les nids de boucles sont inférées.

Nous spécifions d'abord la sémantique d'opérateurs de bas niveau. Ceci permet de définir celles d'opérateurs de haut niveau au travers de compositions. D'autre part, la sémantique de certaines transformations de boucles peut s'exprimer au travers de la composition d'autres types de transformations de boucles.

En raison de l'utilisation de la notation d'index, une inférence de domaine est requise. Nous la calculons en utilisant des opérations sur des ensembles et des relations de Presburger. Nous démarrons également une discussion concernant la spécification des règles de typage ainsi que la possibilité de les combiner avec la formalisation de l'inférence de domaine. Une telle combinaison pourrait permettre de filtrer les programmes incohérents, i.e. ne respectant pas les règles de définition des différents opérateurs tensoriels et transformations de boucles, ainsi que ceux incompatible avec une inférence de domaine.

* ♣ * ♣ *

In this chapter, we formally specify TeML. We first provide its denotational semantics, then we detail its range inference algorithm.

The denotational semantics is an extract of:

> Adilla Susungi, Norman A. Rink, Albert Cohen, Jeronimo Castrillon, and Claude Tadonki. ***Meta-programming for Cross-Domain Tensor Optimizations***. In Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '18). https://doi.org/10.1145/3278122.3278131

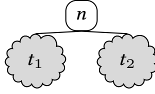


Figure 6.1: $\langle n, [t_1, t_2]\rangle$

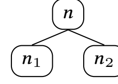


Figure 6.2: $\langle n, [\langle n_1, []\rangle, \langle n_2, []\rangle]\rangle$

6.1 Formal specification

This section gives the denotational semantics of TeML. As explained in the previous chapter, TeML manipulates tensor expressions and loop nests. As both can be represented as trees, domain of trees feature prominently in TeML's denotational semantics.

6.1.1 Domains of trees

Trees are pairs $\langle n, ts\rangle$, where n is the root node and ts is a list of subtrees whose roots are the children of n. Thus, the pair $\langle n, []\rangle$, where $[]$ is the empty list, is a leaf node. Figures 6.1 and 6.2 illustrate this notation of trees.

Tensor expressions are represented as trees whose nodes have a certain structure: the nodes are triples (op, S, I), where op is an operation or identifier, S is the shape of the tensor expression (i.e. a list of its dimensions), and I is a list of iterators. The trees in Figures 6.3–6.5 provide examples of this representation. Note that only identifiers, i.e. nodes (op, S, I) with $op = id$, may appear as leaf nodes. Internal nodes represent operations. It is not meaningful to give an iterator list for an operation, which is indicated by the special value $\bullet$ in Figures 6.3–6.5. Thus, the domain $\mathbf{T}$ of tensor expressions can be given the following recursive definition:

$$\begin{aligned}\mathbf{T} = \{\, \langle (op, S, I), ts\rangle \mid (ts = []) \vee (ts = [t_1, \ldots, t_k] \wedge t_i \in \mathbf{T})\,, \\ I \neq \bullet \Rightarrow ts = [] \wedge op = id \,\}\,.\end{aligned} \tag{6.1}$$

For identifiers we also allow $I = \epsilon$, indicating that the iterator list has not been set yet. Note that $\epsilon \neq []$, and also $\epsilon \neq \bullet$.

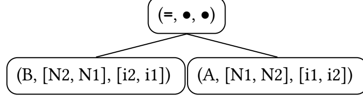
Shapes can also take the special value $\bullet$. This happens only for intermediate tree nodes $(=, \bullet, \bullet)$ that represent assignment operations. TeML assignments do not produce tensor-valued expressions in the target language and thus have no meaningful shape (viz. dimensions) at the meta-level.

Loop nests are also represented as trees of a certain structure: the nodes are iterators and the children of a node are either loop nests or tensor expressions. Thus, the domain $\mathbf{L}$ of loop nests can be recursively defined as

$$\mathbf{L} = \{\, \langle id, [x_1, \ldots, x_k]\rangle \mid x_i \in \mathbf{L} \cup \mathbf{T} \,\}\,. \tag{6.2}$$

This is sufficient to represent arbitrary loop nests, as exemplified by Figure 6.6. Figures 6.3–6.5 also give the tree representations (underlined, in the $\langle\, , [\,]\rangle$ notation) of the perfect loop nests returned by `build`.

```
1  A = tensor([N1, N2])
2  B = eq(A, [i1, i2] -> [i2, i1])
```



$\mathcal{E}_l[\![\texttt{build}(B)]\!]\sigma_2 = \langle \texttt{i1}, [\langle \texttt{i2}, [\sigma_2(B)]\rangle]\rangle$:

```
1  for (int i1 = 0; i1 <= (N1-1); i1++)
2   for (int i2 = 0; i2 <= (N2-1); i2++)
3    B[i2][i1] = A[i1][i2];
```

$$\sigma_1 = \mathcal{P}_{stmt}[\![A = \texttt{tensor}([N1, N2])]\!]\emptyset = \{\, A \mapsto \langle (A, [N1, N2], \epsilon), []\rangle \,\}$$

$$\begin{aligned}\sigma_2 &= \mathcal{P}_{stmt}[\![B = \texttt{eq}(A, [i1, i2] \to [i2, i1])]\!]\sigma_1 \\ &= \{\, A \mapsto \langle (A, [N1, N2], \epsilon), []\rangle\,, \\ &\quad\; B \mapsto \langle (=, \bullet, \bullet), [\langle (B, [N2, N1], [i2, i1]), []\rangle, \\ &\qquad\qquad\qquad\quad \langle (A, [N1, N2], [i1, i2]), []\rangle]\rangle \,\}\end{aligned}$$

Figure 6.3: Matrix transposition implemented with **eq**. The tree on the left depicts $\sigma_2(B)$. The target language (C) code on the right results from $\texttt{build}(B)$.

So far we have always referred to iterators by their names.However, the iterators that appear as nodes in the tree representation of loop nests carry additional information, namely their inferred ranges. The range of an iterator is a triple (lb, ub, st) consisting of integer values lb (lower bound), ub (upper bound), and st (step).

6.1.2 State

The state of a TeML meta-program maps identifiers to trees representing either tensor expressions or loop nests. Thus, the domain **S** of states is defined as

$$\mathbf{S} \;=\; \mathit{identifier} \to (\mathbf{T} + \mathbf{L})\,. \tag{6.3}$$

Hence, a specific state $\sigma \in \mathbf{S}$ is formally a function

$$\sigma : \mathit{identifier} \to (\mathbf{T} + \mathbf{L})\,. \tag{6.4}$$

6.1.3 Valuation functions

We now specify the behavior of TeML programs in terms of valuation functions. For each syntactic category (statement, expression etc.) there is a valuation function that defines how a syntactic entity manipulates a specific state $\sigma \in \mathbf{S}$:

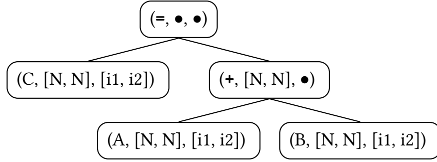
$$\mathcal{P}_{prog} : \mathit{program} \to (\mathbf{S} \to \mathbf{S})\,, \tag{6.5}$$

$$\mathcal{P}_{stmt} : \mathit{stmt} \to (\mathbf{S} \to \mathbf{S})\,, \tag{6.6}$$

$$\mathcal{E}_t : \mathit{Texpression} \to (\mathbf{S} \to \mathbf{T})\,, \tag{6.7}$$

$$\mathcal{E}_l : \mathit{Lexpression} \to (\mathbf{S} \to \mathbf{L})\,. \tag{6.8}$$

```
1    A = tensor([N, N])
2    B = tensor([N, N])
3    C = tensor([N, N])
4    D = @C:add(A, B, [[i1, i2], [i1, i2]]
 ↪   -> [i1, i2])
```



$\mathcal{E}_l[\![\texttt{build}(D)]\!]\sigma_4 = \langle \texttt{i1}, [\langle \texttt{i2}, [\sigma_4(D)]\rangle]\rangle :$

```
1    for (int i1 = 0; i1 <= (N-1); i1++)
2     for (int i2 = 0; i2 <= (N-1); i2++)
3      C[i2][i1] = A[i1][i2] + B[i1][i2];
```

$$\sigma_1 = \mathcal{P}_{stmt}[\![A = \texttt{tensor}([N,N])]\!]\emptyset = \{\, A \mapsto \langle (A,[N,N],\epsilon),[]\rangle \,\}$$

$$\begin{aligned}\sigma_2 &= \mathcal{P}_{stmt}[\![B = \texttt{tensor}([N,N])]\!]\sigma_1 \\ &= \{\, A \mapsto \langle (A,[N,N],\epsilon),[]\rangle,\ B \mapsto \langle (B,[N,N],\epsilon),[]\rangle \,\}\end{aligned}$$

$$\begin{aligned}\sigma_3 &= \mathcal{P}_{stmt}[\![C = \texttt{tensor}([N,N])]\!]\sigma_2 \\ &= \{\, A \mapsto \langle (A,[N,N],\epsilon),[]\rangle,\ B \mapsto \langle (B,[N,N],\epsilon),[]\rangle, \\ &\quad\ C \mapsto \langle (C,[N,N],\epsilon),[]\rangle \,\}\end{aligned}$$

$$\begin{aligned}\sigma_4 &= \mathcal{P}_{stmt}[\![D = \texttt{@}C\!: \texttt{add}(A,B,[[i1,i2],[i1,i2]] \to [i1,i2])]\!]\sigma_3 \\ &= \{\, A \mapsto \langle (A,[N,N],\epsilon),[]\rangle,\ B \mapsto \langle (B,[N,N],\epsilon),[]\rangle, \\ &\quad\ C \mapsto \langle (C,[N,N],\epsilon),[]\rangle, \\ &\quad\ D \mapsto \langle (=),\bullet,\bullet),[\langle (C,[N,N],[i1,i2]),[]\rangle, y]\rangle \,\}, \\ &\text{where } y = \langle (+,[N,N],\bullet),[\langle (A,[N,N],[i1,i2]),[]\rangle, \\ &\qquad\qquad \langle (B,[N,N],[i1,i2]),[]\rangle]\rangle\end{aligned}$$

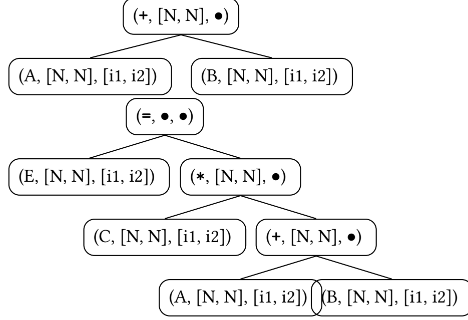
Figure 6.4: The @*id* construct. The tree on the left depicts $\sigma_4(D)$. The C code on the right results from $\texttt{build}(D)$.

$\mathcal{P}_{prog}$ is given in Figure 6.7 and is straightforward: for the empty program ϵ, $\mathcal{P}_{prog}$ yields the identity on **S**; for programs with at least one statement, $\mathcal{P}_{prog}$ relies on function composition ($\circ$) and $\mathcal{P}_{stmt}$. The definition of $\mathcal{P}_{stmt}$ is split across Figures 6.8 and 6.9 since it depends on whether the right-hand side of an assignment is a tensor or a loop expression.

6.1.3.1 Tensor expressions

Figure 6.8 defines the valuation function $\mathcal{E}_t$. Given a tensor expression as its argument, $\mathcal{E}_t$ constructs the tree that represents this tensor expression. The tree that is constructed for **`scalar`** consists of the node $(\square, [], \bullet)$, which sets the shape of a scalar to the empty list $[]$ and forbids iterators to index scalars since the last component of the triple is $\bullet$. When a tree is constructed for **`tensor`**, the last component of the triple is left empty, indicated by ϵ in

```
1   A = tensor([N, N])
2   B = tensor([N, N])
3   C = tensor([N, N])
4   D = vadd(A, B, [[i1, i2], [i1, i2]])
5   E = mul(C, D, [[i1, i2], ] -> [i1, i2])
```



$\mathcal{E}_l[\![\texttt{build}(E)]\!]\sigma_5 = \langle \texttt{i1}, [\langle \texttt{i2}, [\sigma_5(E)]\rangle]\rangle :$

```
1  for (int i1 = 0; i1 < (N-1); i1++)
2   for (int i2 = 0; i2 < (N-1);
↪    i2++)
3    E[i1][i2] = C[i1][i2] *
↪    (A[i1][i2] + B[i1][i2]);
```

$$\sigma_1 = \mathcal{P}_{stmt}[\![A = \texttt{tensor}([N, N])]\!]\emptyset = \{\, A \mapsto \langle (A, [N, N], \epsilon), [] \rangle \,\}$$

$$\begin{aligned}\sigma_2 &= \mathcal{P}_{stmt}[\![B = \texttt{tensor}([N, N])]\!]\sigma_1 \\ &= \{\, A \mapsto \langle (A, [N, N], \epsilon), [] \rangle,\ B \mapsto \langle (B, [N, N], \epsilon), [] \rangle \,\}\end{aligned}$$

$$\begin{aligned}\sigma_3 &= \mathcal{P}_{stmt}[\![C = \texttt{tensor}([N, N])]\!]\sigma_2 \\ &= \{\, A \mapsto \langle (A, [N, N], \epsilon), [] \rangle,\ B \mapsto \langle (B, [N, N], \epsilon), [] \rangle, \\ &\quad\ C \mapsto \langle (C, [N, N], \epsilon), [] \rangle \,\}\end{aligned}$$

$$\begin{aligned}\sigma_4 &= \mathcal{P}_{stmt}[\![D = \texttt{vadd}(A, B, [[i1, i2], [i1, i2]])]\!]\sigma_3 \\ &= \{\, A \mapsto \langle (A, [N, N], \epsilon), [] \rangle,\ B \mapsto \langle (B, [N, N], \epsilon), [] \rangle, \\ &\quad\ C \mapsto \langle (C, [N, N], \epsilon), [] \rangle, \\ &\quad\ D \mapsto \langle (+, [N, N], \bullet), [\langle (A, [N, N], [i1, i2]), [] \rangle, \\ &\qquad\qquad\qquad\qquad \langle (B, [N, N], [i1, i2]), [] \rangle] \rangle \,\}\end{aligned}$$

$$\begin{aligned}\sigma_5 &= \mathcal{P}_{stmt}[\![E = \texttt{mul}(C, D, [[i1, i2],] \rightarrow [i1, i2])]\!]\sigma_4 \\ &= \{\, A \mapsto \langle (A, [N, N], \epsilon), [] \rangle,\ B \mapsto \langle (B, [N, N], \epsilon), [] \rangle, \\ &\quad\ C \mapsto \langle (C, [N, N], \epsilon), [] \rangle, \\ &\quad\ D \mapsto \langle (+, [N, N], \bullet), [\langle (A, [N, N], [i1, i2]), [] \rangle, \\ &\qquad\qquad\qquad\qquad \langle (B, [N, N], [i1, i2]), [] \rangle] \rangle, \\ &\quad\ E \mapsto \langle (=, \bullet, \bullet), [\langle (E, [N, N], [i1, i2]), [] \rangle, \\ &\qquad\qquad \langle (*, [N, N], \bullet), [\langle (C, [N, N], [i1, i2]), [] \rangle, \sigma_4(D)] \rangle] \rangle \,\}\end{aligned}$$

Figure 6.5: More complex tensor expression, including assignment to tensor E. The trees on the left depict $\sigma_4(D)$ (top tree) and $\sigma_5(E)$ (bottom tree) respectively. The target language (C) code on the right results from $\texttt{build}(E)$.

```
for (int i1 = 0; i1 <= (N-1); i1++) {
 C[i1] = A[i1] - B[i1]; // tC
 for (int i2 = 0; i2 <= (N-1); i2++) {
  E[i1][i2] = D[i2] * C[i1]; // tE
  F[i1][i2] = E[i2][i1]; // tF
 }
 for (int i3 = 0; i3 <= (N-1); i3++) {
  G[i1] = G[i1] + F[i1][i3] // tG
 }
}
```

$\langle \texttt{i1}, [\texttt{tC},$
$\quad \langle i2, [\texttt{tE}, \texttt{tF}]\rangle,$
$\quad \langle i3, [\texttt{tG}]\rangle$
$\quad]\rangle$

Figure 6.6: Loop nest (in C) and its tree representation. tC, tE, tF, tG refer to the trees for appropriate tensor expressions.

$$\mathcal{P}_{prog}[\![\epsilon]\!] = \mathrm{id}_{\mathbf{S}} \tag{6.9}$$
$$\mathcal{P}_{prog}[\![s\ p]\!] = \mathcal{P}_{prog}[\![p]\!] \circ \mathcal{P}_{stmt}[\![s]\!] \tag{6.10}$$

Figure 6.7: Definition of $\mathcal{P}_{prog}$. ϵ denotes an empty program, s a statement, and p a program.

.

Equation (6.12). A list of iterators is filled in for ϵ when the tensor is used in an expression. The example in Figure 6.3 demonstrates this: the tensor referred to by A has no iterator list (in either state σ_1 or σ_2); but when A is used in the expression **eq**, an iterator list is filled in based on the arguments of **eq** (cf. state σ_2).

The symbol $\square$ denotes a placeholder for an identifier, which can only be filled in once the identifier on the left-hand side of an assignment has been seen . Therefore, filling in an identifier for $\square$ is deferred to the valuation function $\mathcal{P}_{stmt}$, cf. Equations (6.16), (6.17) and their discussion below.

For **eq**, the function $\mathcal{E}_t$ constructs a tree representing an assignment operation. This also uses a placeholder to defer filling in the identifier for the tensor on the left-hand side of the constructed assignment. The grammar in Figure 5.1 allows the iterator list I_0 to be absent, i.e. $I_0 = \epsilon$. The where-clause after Equation (6.13) specifies when this is possible and, indeed, also required. When $I_0 = \epsilon$, the identifier t must refer to a tensor expression that already has a valid iterator list ($I' \neq \epsilon$), and this then becomes the iterator list of y ($I'' = I'$). However, if $I_0 \neq \epsilon$, then t must refer to a tensor expression with no iterator list ($I' = \epsilon$), and then I_0 is filled in as the iterator list of y ($I'' = I_0$). The handling of the optional arguments I_0 and I_1 of **vop** is completely analogous.

op is syntactic sugar for **vop** followed by **eq**. Equation (6.15) makes this precise. The notation $\sigma\{t \mapsto x\}$ means that the map σ is augmented with a mapping of the identifier t to x. Generally, an existing mapping for t in σ ($t \in dom(\sigma)$) is overwritten. In Equation (6.15), however, t is chosen to be a fresh identifier not already mapped by σ.

Note that the arguments I_0, I_1 of **op** are simply passed on to **vop** in Equation (6.15). The

$$\mathcal{E}_t[\![\texttt{scalar}()]\!] = \lambda\sigma.\langle(\square, [], \bullet), []\rangle \tag{6.11}$$

$$\mathcal{E}_t[\![\texttt{tensor}(S)]\!] = \lambda\sigma.\langle(\square, S, \epsilon), []\rangle \tag{6.12}$$

$$\begin{aligned}
&\mathcal{E}_t[\![\texttt{eq}(t, I_0 \to I_1)]\!] = \\
&\quad \lambda\sigma.\text{let } \langle(op, S, I'), ys\rangle = \sigma(t) \\
&\qquad\qquad y = \langle(op, S, I''), ys\rangle \\
&\qquad\qquad x = \langle(\square, S', I_1), []\rangle \\
&\qquad \text{in } \langle(\texttt{=}, \bullet, \bullet), [x, y]\rangle\,, \\
&\qquad \text{where } \begin{cases} I' \neq \epsilon \wedge I'' = I'\,, & \text{if } I_0 = \epsilon \\ I' = \epsilon \wedge I'' = I_0\,, & \text{if } I_0 \neq \epsilon \end{cases}
\end{aligned} \tag{6.13}$$

$$\begin{aligned}
&\mathcal{E}_t[\![\texttt{vop}(t_0, t_1, [I_0, I_1])]\!] = \\
&\quad \lambda\sigma.\text{let } \langle(op_0, S_0, I'_0), ys_0\rangle = \sigma(t_0) \\
&\qquad\qquad \langle(op_1, S_1, I'_1), ys_1\rangle = \sigma(t_1) \\
&\qquad\qquad y_0 = \langle(op_0, S_0, I''_0), ys_0\rangle \\
&\qquad\qquad y_1 = \langle(op_1, S_1, I''_1), ys_1\rangle \\
&\qquad \text{in } \langle(\texttt{op}, S', \bullet), [y_0, y_1]\rangle\,, \\
&\qquad \text{where } \begin{cases} I'_i \neq \epsilon \wedge I''_i = I'_i\,, & \text{if } I_i = \epsilon \\ I'_i = \epsilon \wedge I''_i = I_i\,, & \text{if } I_i \neq \epsilon \end{cases}
\end{aligned} \tag{6.14}$$

$$\begin{aligned}
&\mathcal{E}_t[\![\texttt{op}(t_0, t_1, [I_0, I_1] \to I_2)]\!] = \\
&\quad \lambda\sigma.\text{let } x = \mathcal{E}_t[\![\texttt{vop}(t_0, t_1, [I_0, I_1])]\!]\sigma \\
&\qquad \text{in } \mathcal{E}_t[\![\texttt{eq}(t, \epsilon \to I_2)]\!]\,(\sigma\{t \mapsto x\})\,, \\
&\qquad \text{where } t \text{ is an identifier not in } dom(\sigma)
\end{aligned} \tag{6.15}$$

$$\begin{aligned}
\mathcal{P}_{stmt}[\![id \texttt{=} e_t]\!] = \lambda\sigma.\text{let } & x = \mathcal{E}_t[\![e_t]\!]\sigma \\
& x' = x[id/\square] \\
\text{in } & \sigma\{id \mapsto x'\}
\end{aligned} \tag{6.16}$$

$$\begin{aligned}
\mathcal{P}_{stmt}[\![id_1 \texttt{ = @}id_2 : e_t]\!] = \lambda\sigma.\text{let } & x = \mathcal{E}_t[\![e_t]\!]\sigma \\
& x' = x[id_2/\square] \\
\text{in } & \sigma\{id_1 \mapsto x'\}
\end{aligned} \tag{6.17}$$

Figure 6.8: Valuation functions $\mathcal{E}_t$ and $\mathcal{P}_{stmt}$ for tensor expressions. The placeholder $\square$ is filled with an identifier by $\mathcal{P}_{stmt}$. The shape S' in `eq` and `vop` must be inferred.

mul operation in Figure 6.5 is an example of when one of the iterator list arguments must be omitted, specifically $I_1 = \epsilon$.

The purpose of $\mathcal{P}_{stmt}$ is to produce a new state from its argument σ by adding a new mapping for an identifier (*id* in Equation (6.16) or id_2 in Equation (6.17)). The target of this new mapping is an expression tree x' that is produced by first evaluating $\mathcal{E}_t[\![e_t]\!]$ in the state σ and then, in Equation (6.16), replacing potential occurrences of $\square$ with the identifier *id* (in symbols: $x[id/\square]$). Now *id* occurs in the expression tree x' and will thus make it into the target language program generated by TeML; but *id* is also mapped in the new state $\sigma\{id \mapsto x'\}$. This is the double meaning of *id* made precise. Equation (6.17) does not lead to this double meaning since $\square$ is replaced with id_2 from the **@**id_2 construct, of which Figure 6.4 gives a worked example. Generally, the right panes of Figures 6.3–6.5 give step-by-step evaluations of $\mathcal{P}_{prog}$, broken down into evaluations of $\mathcal{P}_{stmt}$.

Since TeML assignments have a 3-address format, nested evaluations of $\mathcal{E}_t$ do not occur. This is also clear from Figure 6.8. Therefore, there can never be more than once occurrence of $\square$ in the tree x in Equations (6.16) and (6.17).

In Equation (6.13) the shape S' must be inferred from the properties of y; and S' in Equation (6.14) must be inferred from y_0, y_1. Inferring the shape of a tensor expression from the shapes and iterator lists of its component expressions is straightforward and follows a strategy analogous to the typing of tensor expressions in [111]. Inference fails for malformed tensor expressions, in which case TeML cannot generate a valid target language program. Also analogous to [111], the assignments in Equations (6.16), (6.17) are malformed if *id* or id_2, respectively, has not been introduced with **tensor**.

6.1.3.2 Loop expressions

The definitions of $\mathcal{E}_l$ in Figure 6.9 freely identify iterators with triples (lb, ub, st) that specify iterator ranges (cf. the last paragraph of Section 6.1.1). When a loop nest is built around a tensor expression in Equation (6.18), the upper bounds ub_k must be inferred based on the positions in which iterators are used to index tensors in the expression $\sigma(t)$. This is done straightforwardly during shape inference. A special case occurs if $x = \langle(id, S, [\texttt{i1}, \ldots, \texttt{ir}]), []\rangle$, i.e. all iterators in $\sigma(t)$ appear on the left-hand side of the assignment. Then $ub_k = S_k - 1$, where S_k is the k-th dimension in S. Note that the where-clause after Equation (6.18) enforces that loop nests are only built around assignments of tensors. Example evaluations of $\mathcal{E}_l[\![\texttt{build}(id)]\!]$ appear (underlined) in the bottom left corners of Figures 6.3–6.5, together with the corresponding loop nests in the target language, i.e. C.

Equation (6.19) implements a version of stripmining that does not transform tensor indices inside tensor expressions. The definition for **interchange** in Equation (6.20) is straightforward: it swaps the position of iterators. Note that **fuse** in Equation (6.21) fuses identical outer iterators of two loop nests, and **fuse_inner** in Equation (6.22) fully fuses loops at nesting level r. Observe that the definitions of both fusions require that the forms of the loop arguments satisfy certain matching conditions.

$$\begin{aligned}\mathcal{E}_l[\![\texttt{build}(t)]\!] = \lambda\sigma.\text{let } & r = \text{“number of iterators in } \sigma(t)\text{”}\\ & i_k = (0, ub_k, 1) \text{ for } k = 1, \ldots, r\\ \text{in } & \langle i_1, \ldots, \langle i_r, [\sigma(t)]\rangle \ldots\rangle,\\ \text{where } & \sigma(t) = \langle (=, \bullet, \bullet), [x, y]\rangle\end{aligned} \tag{6.18}$$

$$\begin{aligned}&\mathcal{E}_l[\![\texttt{stripmine}(l, r, v)]\!] =\\ &\quad \lambda\sigma.\text{let } \langle i_1, \ldots \langle i_r, xs\rangle \ldots\rangle = \sigma(l)\\ &\qquad (b, e, 1) = i_r\\ &\qquad i'_r = (0, (e-b)/v - 1, 1)\\ &\qquad i'_{r+1} = (b + v \cdot i'_r, b + v \cdot i'_r + (v-1), 1)\\ &\quad \text{in } \langle i_1, \ldots \langle i'_r, [\langle i'_{r+1}, xs\rangle]\rangle \ldots\rangle\end{aligned} \tag{6.19}$$

$$\begin{aligned}&\mathcal{E}_l[\![\texttt{interchange}(l, r_1, r_2)]\!] =\\ &\quad \lambda\sigma.\text{let } \langle i_1, \ldots \langle i_{r_1}, \ldots \langle i_{r_2}, xs\rangle \ldots\rangle \ldots\rangle = \sigma(l)\\ &\quad \text{in } \langle i_1, \ldots \langle i_{r_2}, \ldots \langle i_{r_1}, xs\rangle \ldots\rangle \ldots\rangle\end{aligned} \tag{6.20}$$

$$\begin{aligned}&\mathcal{E}_l[\![\texttt{fuse}(l_1, l_2, r)]\!] =\\ &\quad \lambda\sigma.\text{let } \langle i_1, \ldots \langle i_r, xs\rangle \ldots\rangle = \sigma(l_1)\\ &\qquad \langle i_1, \ldots \langle i_r, ys\rangle \ldots\rangle = \sigma(l_2)\\ &\quad \text{in } \langle i_1, \ldots \langle i_r, xs \mathbin{||} ys\rangle \ldots\rangle\end{aligned} \tag{6.21}$$

$$\begin{aligned}&\mathcal{E}_l[\![\texttt{fuse_inner}(l, r)]\!] =\\ &\quad \lambda\sigma.\text{let } \langle i_1, \ldots \langle i_{r-1}, xs\rangle \ldots\rangle = \sigma(l)\\ &\qquad [\langle i_r, xs_1\rangle, \ldots, \langle i_r, xs_n\rangle] = xs\\ &\quad \text{in } \langle i_1, \ldots \langle i_{r-1}, [\langle i_r, xs_1 \mathbin{||} \ldots \mathbin{||} xs_n\rangle]\rangle \ldots\rangle\end{aligned} \tag{6.22}$$

$$\begin{aligned}&\mathcal{E}_l[\![\texttt{unroll}(l, r)]\!] =\\ &\quad \lambda\sigma.\text{let } \langle i_1, \ldots \langle i_{r-1}, [\langle i_r, xs\rangle]\rangle \ldots\rangle = \sigma(l)\\ &\qquad (b, e, s) = i_r\\ &\qquad k = (e-b)/s + 1\\ &\quad \text{in } \langle i_1, \ldots \langle i_{r-1}, xs'_0 \mathbin{||} \ldots \mathbin{||} xs'_{k-1}\rangle \ldots\rangle,\end{aligned} \tag{6.23}$$

$$\begin{aligned}&\quad \text{where } xs'_j \text{ is obtained from } xs \text{ by replacing}\\ &\quad \text{the iterator } i_r \text{ with the constant } s \cdot j + b\end{aligned} \tag{6.24}$$

$$\mathcal{P}_{stmt}[\![id = e_l]\!] = \lambda\sigma.\text{let } x = \mathcal{E}_l[\![e_l]\!]\sigma \text{ in } \sigma\{id \mapsto x\} \tag{6.25}$$

Figure 6.9: Valuation functions $\mathcal{E}_l$ and $\mathcal{P}_{stmt}$ for loop expressions. The symbol || denotes concatenation of lists.

The definition for `unroll` in Equation (6.24) is standard, but note that unrolling the outermost loop, i.e. $r = 1$ in Equation (6.24), generally results in a sequence of loops and tensor expressions. Technically, such a sequence is not an element of the domain $\mathbf{L}$. Therefore, we should instead work with the domain $\mathbf{L}'$ defined as

$$\mathbf{L}' = \mathbf{L} \cup \{ \langle \bullet, [x_1, \ldots, x_k] \rangle \mid x_i \in \mathbf{L} \cup \mathbf{T} \} . \tag{6.26}$$

The trees $\langle \bullet, [x_1, \ldots, x_k] \rangle$ represent sequences of top-level loops and tensor expressions $x_1, \ldots, x_k$, i.e. loops and tensor expressions not nested inside a loop. A tree of the form $\langle \bullet, [x] \rangle$ also result when `build` is applied to a tensor expression that has no iterators, i.e. an expression formed of scalars only. We have omitted such corner cases of degenerate loops from the general exposition since TeML's focus is on meaningful transformations of real, i.e. non-degenerate loops.

The definition of $\mathcal{P}_{stmt}$ in Equation (6.25) is completely standard. No placeholders for identifiers are required for loop expressions. This is ultimately because loops have no meaningful names at the level of target language programs.

6.1.3.3 Parallelization

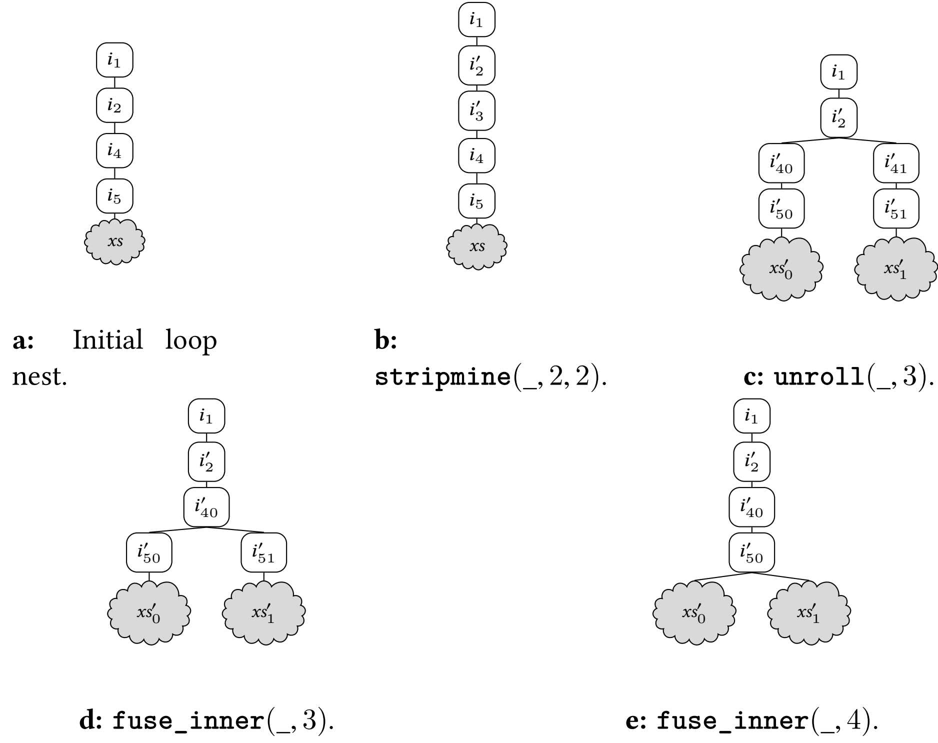
The TeML core language (cf. Figure 5.1) can be extended with the loop expressions `parallelize` and `vectorize` that specify that a loop be parallelized or vectorized, respectively, using pragmas or intrinsic functions. Our representation of loops as trees in $\mathbf{L}$ does not capture this, which is not a problem since neither parallelization nor vectorization changes the structure of a loop. Further work is required to enable more fine-grained control over parallelization, including for instance the generation of atomic sections or optimized parallel reductions. This would necessitate extending both TeML's expressiveness and its code generation process.

6.2 Compositional definitions

In this section, we extend TeML with additional loop and tensor expressions that implement more abstract operations on loops and tensors. These abstract operations enhance TeML's expressiveness and, from the perspective of a TeML implementation, are considered built into the language. Nonetheless, the key observation in this section is that more abstract operations can be defined in terms of the TeML core language by means of the composition in Equation (6.10). This facilitates a modular implementation of TeML and also demonstrates the flexibility of the language.

6.2.1 Loop transformations

The loop transformations in the TeML core language, as defined in Figure 6.9, appear to be rather restrictive. For example, `interchange` only operates on immediately adjacent loops,



a: Initial loop nest. **b:** $\texttt{stripmine}(_, 2, 2)$. **c:** $\texttt{unroll}(_, 3)$.

d: $\texttt{fuse_inner}(_, 3)$. **e:** $\texttt{fuse_inner}(_, 4)$.

Figure 6.10: The standard loop transformation *unroll-and-jam* as a sequence of TEML core transformations.

and **unroll** performs complete unrolling of a loop. We now demonstrate how common, more flexible loop transformations can be composed from the ones in the core language.

6.2.1.1 Partial unrolling

Partial unrolling of a loop nest l by a factor v can be achieved by first stripmining l, also with the factor v. According to Equation (6.19), this introduces a new loop rooted at the iterator i'_{r+1}. Fully unrolling the new loop will result in the partial unrolling of the original loop l. Hence,

$$\mathcal{P}_{stmt}[\![l' = \texttt{part_unroll}(l, r, v)]\!] = \mathcal{P}_{prog}\left[\!\!\left[\begin{array}{l} l_1 = \texttt{stripmine}(l, r, v) \\ l' = \texttt{unroll}(l_1, r + 1) \end{array}\right]\!\!\right] \tag{6.27}$$

Note that in the present section all identifiers that appear only in the argument of the valuation function $\mathcal{P}_{prog}$ are considered *fresh*, i.e. they are not in $dom(\sigma)$ for any state σ that $\mathcal{P}_{stmt}$ is applied to. An implementation of TEML can always introduce as many fresh identifiers as needed.

6.2.1.2 Unroll-and-jam

Unroll-and-jam is a variant of loop unrolling typically applied when considering non-innermost loop nests. When applied to a loop nest l, unroll-and-jam first unrolls a loop that is not the most deeply nested one in l. (This is the *unroll* part.) This typically results in several copies of identical nested loops, which are subsequently fused. (This is the *jam* part.) Figure 6.10 illustrates the steps taken when unroll-and-jam is applied at nesting level 2 to a loop nest that initially has depth 4. Figure 6.10 immediately generalizes to a definition of `unroll_jam` in TeML,

$$\mathcal{P}_{stmt}[\![l' = \texttt{unroll_jam}(l, r, v)]\!] = \mathcal{P}_{prog}\left[\!\!\left[\begin{array}{l} l_0 = \texttt{part_unroll}(l, r, v) \\ l_1 = \texttt{fuse_inner}(l_0, r+1) \\ l_2 = \texttt{fuse_inner}(l_1, r+2) \\ \dots \\ l_{d-1-r} = \texttt{fuse_inner}(l_{d-2-r}, d-1) \\ l' = \texttt{fuse_inner}(l_{d-1-r}, d) \end{array}\right]\!\!\right], \tag{6.28}$$

where d is the depth of the original loop nest l. Note that the steps `stripmine` and `unroll` from Figure 6.10 have been combined into the previously defined `part_unroll`.

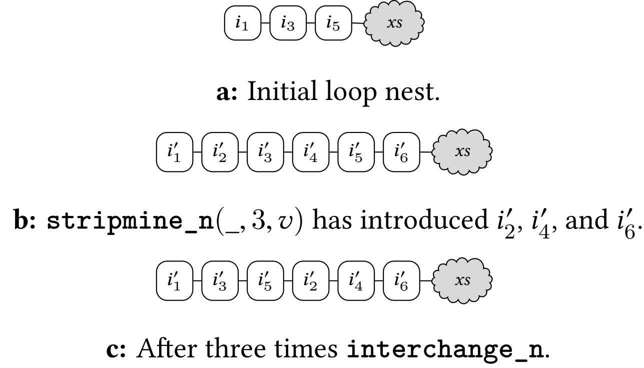
6.2.1.3 Tiling

In order to increase data locality, tiling organizes the iteration space of a loop nest into blocks. This can be achieved by multiple applications of stripmining, each of which introduces blocks into a single loop, cf. Equation (6.19). The auxiliary transformation `stripmine_n` expands into the multiple applications of stripmining required for tiling,

$$\mathcal{P}_{stmt}[\![l' = \texttt{stripmine_n}(l, n, v)]\!] = \mathcal{P}_{prog}\left[\!\!\left[\begin{array}{l} l_1 = \texttt{stripmine}(l, 1, v) \\ l_2 = \texttt{stripmine}(l_1, 3, v) \\ \dots \\ l_{n-1} = \texttt{stripmine}(l_{n-2}, 2(n-1)-1, v) \\ l' = \texttt{stripmine}(l_{n-1}, 2n-1, v) \end{array}\right]\!\!\right]. \tag{6.29}$$

Note that if d is the depth of the original (perfect) loop nest l, then the depth of the resulting loop nest l' is $d + n$.

To arrange the new loops that have been introduced by stripmining into the right order for tiling, several interchanges are needed. To this end, we introduce `interchange_n` that permutes an iterator i in a loop nest through the next n iterators (towards the innermost



a: Initial loop nest.

b: `stripmine_n`$(_, 3, v)$ has introduced i'_2, i'_4, and i'_6.

c: After three times `interchange_n`.

Figure 6.11: Tiling.

loop),

$$\mathcal{P}_{stmt}[\![l' = \texttt{interchange_n}(l, r, n)]\!] = \mathcal{P}_{prog}\left[\!\!\left[\begin{array}{l} l_1 = \texttt{interchange}(l, r, r+1) \\ l_2 = \texttt{interchange}(l_1, r+1, r+2) \\ \ldots \\ l_{n-1} = \texttt{interchange}(l_{n-2}, r+n-2, \\ \qquad\qquad r+n-1) \\ l' = \texttt{interchange}(l_{n-1}, r+n-1, \\ \qquad\qquad r+n) \end{array}\right]\!\!\right]. \quad (6.30)$$

Finally, TEML's loop transformation `tile` can be defined,

$$\mathcal{P}_{stmt}[\![l' = \texttt{tile}(l, v)]\!] = \mathcal{P}_{prog}\left[\!\!\left[\begin{array}{l} l_0 = \texttt{stripmine_n}(l, d, v) \\ l_1 = \texttt{interchange_n}(l_0, 2, 2d-2) \\ l_2 = \texttt{interchange_n}(l_1, 3, 2d-3) \\ \ldots \\ l_{d-1} = \texttt{interchange_n}(l_{d-2}, d, d)) \\ l' = \texttt{interchange_n}(l_{d-1}, d+1, d-1) \end{array}\right]\!\!\right], \quad (6.31)$$

where d is the depth of the original loop nest l, and the resulting loop nest l' has depth $2d$. Figure 6.11 demonstrates how `tile` handles a loop nest of initial depth 3. First, stripmining introduces the additional iterators i'_2, i'_4, and i'_6 in Figure 6.11b. Then, three applications of `interchange_n` move these iterators towards the deep positions in the loop nest, cf. Figure 6.11c.

6.2.2 Tensor operations

Tensor operations that are more abstract than the fundamental arithmetic operations, such as `add` and `mul`, allow algorithms to be expressed more concisely in TeML. In particular, more abstract operations largely hide the explicit, and therefore error-prone, manipulation of iterator lists. As was the case for loop transformations, abstract tensor operations can be defined in terms of TeML's core operations, requiring only a few simple side rules for iterator lists and shapes that a TeML implementation can easily check.

6.2.2.1 Entrywise operations

If the tensors t_0 and t_1 have the same shapes, the core arithmetic operations `add`, `sub` and `mul` can be applied to all entries (viz. components) of $t0$ and $t1$ simultaneously. Let r denote the length of the shapes of t_0 and t_1. The entrywise application of arithmetic operations is enforced in the following definitions by the fact that the same iterator list $I = [\texttt{i1}, \ldots \texttt{ir}]$ is used for both $t0$ and $t1$,

$$\mathcal{E}_t[\![\texttt{ventrywise_op}(t_0, t_1)]\!] = \mathcal{E}_t[\![\texttt{vop}(t_0, t_1, [I, I])]\!]\,, \tag{6.32}$$

$$\mathcal{E}_t[\![\texttt{entrywise_op}(t_0, t_1)]\!] = \mathcal{E}_t[\![\texttt{op}(t_0, t_1, [I, I] \to I)]\!]\,. \tag{6.33}$$

Theses formulae generalize to arbitrary numbers of arguments. In fact, TeML's entrywise operations accept an arbitrary number of tensors are arguments.

6.2.2.2 Transposition

Transposition amounts to reordering an iterator list:

$$\mathcal{E}_t[\![\texttt{transpose}(t, [[r_1, r_2], \ldots, [r_{2k-1}, r_{2k}]])]\!] = \mathcal{E}_t[\![\texttt{eq}(t, I \to I')]\!]\,, \tag{6.34}$$

where I is a list of fresh iterators for t, and I' is obtained from I by swapping pairs of iterators at positions r_{2i-1} and r_{2i}, for $i = 1, \ldots, k$. Since `transpose` is defined in terms of `eq`, it results in an assignment to a real tensor in the target language program, with different iterator lists for the assignee and the tensor t. The corresponding virtual operation `vtranspose` does not introduce any operations into the target language program, it simply causes TeML to reorganize iterator lists.

6.2.2.3 Contraction

Contraction generalizes matrix multiplication to tensors of higher dimensions. Thus, contraction is fundamental to many complex algorithms that operate on tensors. In particular, the vector dot product and matrix-vector multiplication are low-dimensional instances of contraction.

Let S_0 and S_1 be the shapes of tensors t_0 and t_1, and let these shapes have lengths s_0 and s_1 respectively. The tensors t_0 and t_1 can be contracted along dimensions r_0 and r_1,

respectively, if the r_0-th dimension of t_0 has the same extent as the r_1-th dimension of t_1; in other words, if the r_0-th entry of S_0 equals the r_1-th entry of S_1. Then,

$$\mathcal{P}_{stmt}[\![t' = \texttt{contract}(t_0, t_1, [r_0, r_1])]\!] = \mathcal{P}_{prog}\left[\!\!\left[\begin{array}{l} t_2 = \texttt{vmul}(t_0, t_1, [I, J]) \\ t' = \texttt{add}(t', t_2, [I', \epsilon] \to I') \end{array}\right]\!\!\right], \quad (6.35)$$

where

$$\begin{aligned} I &= [\texttt{i0}, \ldots, \texttt{i}(r_0 - 1), \texttt{k}, \texttt{i}(r_0 + 1), \ldots, \texttt{i}s_0], \\ J &= [\texttt{j0}, \ldots, \texttt{j}(r_1 - 1), \texttt{k}, \texttt{j}(r_1 + 1), \ldots, \texttt{j}s_1], \\ I' &= (I \backslash \{\texttt{k}\}) \mid\mid (J \backslash \{\texttt{k}\}). \end{aligned}$$

The iterator k appears at position r_0 in I, and at position r_1 in J; it no longer appears in the iterator list I' for the resulting tensor t'. Contraction of more than one pair of dimensions is defined analogously, but the formulae become unwieldy.

Contraction is a reduction operation and therefore fundamentally requires an accumulator. In Equation (6.35), the resulting tensor t' also acts as the accumulator, which is why t' appears both as the result and the argument of the **add** operation. Because of the **add** in Equation (6.35), t' is a real tensor, which in the target language program is backed by memory. This is why the virtual counterpart **vcontract** cannot be given a definition in terms of more fundamental TeML operations: in a virtual operation, TeML does not have at its disposal a real tensor that can play the role of the accumulator. Therefore, **vcontract** must be considered fundamental. Of course, when generating target language code, TeML can introduce a temporary variable in the target language that acts as the accumulator.[1]

6.2.2.4 Outer product

The outer product, often simply referred to as the *tensor product*, combines the components of tensors $t_1, \ldots, t_n$ into a single tensor whose shape then is the concatenation of the shapes of $t_1, \ldots, t_n$. In formulae,

$$\mathcal{P}_{stmt}[\![t' = \texttt{vouterproduct}(t_1, \ldots, t_n)]\!] = \mathcal{P}_{prog}\left[\!\!\left[\begin{array}{l} s_{n-1} = \texttt{vmul}(t_{n-1}, t_n, [I_{n-1}, I_n]) \\ s_{n-2} = \texttt{vmul}(t_{n-2}, s_{n-1}, [I_{n-2}, \epsilon]) \\ \ldots \\ s_2 = \texttt{vmul}(t_2, s_3, [I_2, \epsilon]) \\ t' = \texttt{vmul}(t_1, s_2, [I_1, \epsilon] \to I') \end{array}\right]\!\!\right], \quad (6.36)$$

[1] It is possible to define `vcontract` by fully unrolling the iterator k in Equation (6.35), which would effectively amount to removing t' as an argument to **add**. However, this would eliminate the loop rooted at iterator k and thus deprive TeML of additional opportunities for loop transformations.

$$\mathcal{P}_{stmt}[\![t' = \texttt{outerproduct}(t_1, \ldots, t_n)]\!] = \mathcal{P}_{prog}\left[\!\!\left[\begin{array}{l} s = \texttt{vouterproduct}(t_1, \ldots, t_n) \\ t' = \texttt{eq}(s, \epsilon \to I') \end{array}\right]\!\!\right], \quad (6.37)$$

where $I' = I_1 \mid\mid \ldots \mid\mid I_n$, and the iterator lists $I_1, \ldots, I_n$ are pairwise disjoint, so that no iterators appear more than once in the concatenated list I'. Note that the implementation of `outerproduct` is not quite as modular as Equation (6.37) suggests: to construct the final iterator list I', the iterator lists $I_1, \ldots, I_n$ from the definition of `vouterproduct` are needed.

6.3 Range Inference

We mentioned throughout this chapter the notion of range inference, that is, understanding the use of loop indices in tensor subscripts in order to deduce tensor shapes, as well as iteration ranges. Such a concept is necessary for languages using index notation. Halide implements range inference heuristics based on an iterative approach. Other languages and APIs using index notation (and exploiting Halide in their tool flow) benefit from it, i.e, TVM and Tensor Comprehensions.

Our range inference algorithm is not iterative and is based on Presburger *sets* and *relations.*

Background

Definition 6.3.1. *Let p, q and r be positive natural integers.* $\longrightarrow \subseteq \mathbb{Z}^p \times \mathbb{Z}^q$ *is a* ***(p,q)-atomic affine relation*** *parametrized by* $\vec{n} \in \mathbb{Z}^r$ *if there exists* $A \in \ell, p+q(\mathbb{Z})$, $A_\exists \in \ell, s(\mathbb{Z})$, $B \in \ell, r(\mathbb{Z})$ *and* $\vec{c} \in \mathbb{Z}^\ell$ *such that:*

$$\vec{x} \longrightarrow \vec{y} \Leftrightarrow \exists \vec{z} \in \mathbb{Z}^s : A\begin{pmatrix}\vec{x}\\ \vec{y}\end{pmatrix} + A_\exists \vec{z} + B\vec{n} + \vec{c} \geq \vec{0}$$

Definition 6.3.2. *A* ***Presburger relation*** $\longrightarrow$ *is a finite union of* (p,q)*-atomic affine relations.*

Also, we define the ***domain****:* $dom \longrightarrow = \{\vec{x} \in \mathbb{Z}^p \mid \vec{x} \longrightarrow \vec{y}\}$ *and the* ***range****:* $ran \longrightarrow = \{\vec{y} \in \mathbb{Z}^q \mid \vec{x} \longrightarrow \vec{y}\}$.

Affine relations come along with useful operations to compute range inference. We are particularly interested in a range's *lexicographic maximum.*

Definition 6.3.3. *Let be a relation* $\longrightarrow \subseteq E \times F$ *and a total order* $\prec$ *over F. The* ***maximum of*** $\longrightarrow$ *w.r.t.* $\prec$ *is the relation defined as follows:*

$$\min_{\prec} \longrightarrow = \{(x, y_{\max}) \mid y_{\max} = \max_{\prec}\{y \mid x \longrightarrow y\}\} \quad (6.38)$$

The algorithm

Range inference using sets and relations is performed as in Algorithm 1. We consider the set of constraints as a Presburger set in which (a) no qualifier is used and (b) $\wedge$ is the only possible logical operator. We define a relation in $\mathbb{Z}^p \times \mathbb{Z}^q$ where $\mathbb{Z}^p$ is the set of constraints and $\mathbb{Z}^q$ is the set of writes (in other words, the subscript of the output tensor). Using operations over relations, we can directly compute the minimum valid shape of the output tensor (and the iteration domain of the loop). Figure 6.12 is a concrete example demonstrating how this can be computed using ISCC an interface to isl. TeML's implementation includes building such ISCC scripts and extracting its results. While there exists Python wrapper for isl, islpy[2], we find more convenient to make use of ISCC scripts due to ISCC features that are not in islpy.

This approach to range inference may be complemented with typing rules to discard programs on which such an algorithm cannot be performed.

Algorithm 1 Non-iterative range inference using Presburger sets and relations

Input: A the set of all subscripts, T the set of all input tensors.
Output: *Dom*, the iteration domain and *LexRan*, the shape of the output tensor

1. Create a set D bounding all subscript terms with the shape of their corresponding tensor:
   for a in A **do**
       Let $S = shape(tensor(a))$
       for t in $terms(a)$ **do**
           Let k be the position of t in a
           Add $0 \leq t < S[k]$ to D
       end for
   end for
2. Build $Write$, the set of writes intersected with D;
3. Compute *Dom* and *Ran*, the domain and range of $Write$;
4. Compute *LexRan*, the lexicographic maximum of *Ran*
5. Return *Dom* and *LexRan*.

6.4 Towards Type Safety

Though deeper investigation is need to define TeML's type system, we discuss the definition of typing rules.

We can use a polymorphic type $Tens(N_1, \cdots, N_{\bar{r}})$ for tensors; the polymorphism allows to encode the shape of a tensor. As for loops, we consider a static type *Loop* since the denotational semantics of constructs returning loops already carry all necessary information for loop construction.

[2]islpy: https://github.com/inducer/islpy

```
# TeML script
A = tensor(double, [500, 26, 72])
B = tensor(double, [500, 72, 26])

C = mul(A, B, [[i1, i2, i4], [i1, i4, i3]] -> [i1, i2, i3])
l = build(C)
```

```
# ISCC script
C := { S[i1, i2, i4, i3]: 0 <= i1 < 500 and 0 <= i2 < 26 and 0 <= i4 < 72 and 0 <=
↪ i3 < 26 };
W := { S[i1, i2, i4, i3] -> C[i1, i2, i3]} * C;
R := ran W;
D := dom W;
L := lexmax R;
print D;
print L;
```

which returns

```
# Iteration domain
{ S[i1, i2, i4, i3] : 0 <= i1 <= 499 and 0 <= i2 <= 25 and 0 <= i4 <= 71 and 0 <=
↪ i3 <= 25 }

# Range inference of C (incrementing with 1 gives the actual shape)
{ C[499, 25, 25] }
```

Figure 6.12: Range inference for batch matrix multiplication

A set of typing rules is exposed in Figures 6.13 and 6.14. `scalar` and `tensor` require strings but only those denoting existing data types: *string* $\in$ [“int”, “float”, “int64”, $\cdots$] (for C code generation). In addition, the shape required for `tensor` must contain a list of positive integers. A well-formed arithmetic operation means that (i) the size of subscript lists match the size of the input tensor shapes, (ii) concrete tensors must be associated to a subscript list, whereas virtual ones should not. Therefore, if no subscript list is associated with both input tensors, then they must all be virtual and have an irrelevant shape. Most high-level tensor operators are constrained by parameters such as ranks and shapes with respect to their semantics. This level ensures that tensors are well formed at least regardless of the application domain. Though, one could relax or enforce such constraints with respect to a domain [49]. Our typing rules for loops only discard constructs with illegal parameters (.e.g., interchanging a loop with out of legal range ranks).

6.5 Conclusion

In this chapter, we formally specify TeML. The semantics of TeML is explained by means of tree representations allowing to reason about interactions between tensor expressions and loop transformations. Due to the use of index notation, range inference is required, which is directly computed using operations over Presburger sets and maps. This enables possible

$$\frac{T : string}{\texttt{scalar}(T) : Scal}\ \text{T-Scalar}$$

$$\frac{T : string \quad \forall n \in S, n \in \mathbb{N}^{+}}{\texttt{tensor}(T, S) : Tens(S)}\ \text{T-Array}$$

$$\frac{t : Tens(S) \quad 1 \leq r_1, r_2 < \overline{r} \quad r_1 \neq r_2}{\texttt{transpose}(t, r_1, r_2) : Tens(swap(S, r_1, r_2))}\ \text{T-Transpose}$$

$$\frac{t_1 : Tens(S_1) \quad \cdots \quad t_n : Tens(S_n)}{\texttt{outerproduct}(t_1, \cdots, t_n) : Tens(S_1 \mid\mid \cdots \mid\mid S_n)}\ \text{T-Outerproduct}$$

$$\frac{\begin{array}{c} t_1 : Tens(S_1) \quad t_2 : Tens(S_2) \quad S_1[r_{1_k}] = S_2[r_{2_k}] \\ 1 \leq r_{1_k} \leq \overline{r_1} \quad 1 \leq r_{2_k} \leq \overline{r_2} (\forall k \in [1, p]) \end{array}}{\texttt{contract}(t_1, t_2, [[r_{1_1}, r_{2_1}] \cdots [r_{1_p}, r_{2_p}]]) : Tens(S')}\ \text{T-Contract}$$

where

$S' = S_1 \setminus \{r_{1_1}, \cdots, r_{k_1}\} \mid\mid S_2 \setminus \{r_{1_2}, \cdots, r_{k_2}\}$

$$\frac{t : Tens}{\texttt{build}(t) : Loop}\ \text{T-build}$$

$$\frac{l : Loop \quad 1 \leq r \leq \overline{r} \quad 1 < v \leq \overline{m}}{\texttt{stripmine}(l, r, v) : Loop}\ \text{T-Stripmine}$$

$$\frac{l : Loop \quad 1 \leq r_1, r_2 < \overline{r} \quad r_1 \neq r_2}{\texttt{interchange}(t, r_1, r_2) : Loop}\ \text{T-Interchange}$$

$$\frac{l_1 : Loop \quad l_2 : Loop \quad 1 < r < \overline{r_1} \quad 1 < r < \overline{r_2}}{\texttt{fuse}(l_1, l_2, r) : Loop}\ \text{T-Fuse}$$

$$\frac{l : Loop}{\texttt{unroll}(l) : Loop}\ \text{T-Unroll}$$

$$\frac{l : Loop \quad 1 \leq r \leq \overline{r}}{\texttt{unroll_jam}(l, r) : Loop}\ \text{T-Unrollljam}$$

$$\frac{l : Loop \quad 1 \leq r \leq \overline{r} \quad 1 < v \leq \overline{m}}{\texttt{part_unroll}(l, r, v) : Loop}\ \text{T-Partunroll}$$

$$\frac{l : Loop \quad 1 < v \leq \overline{m}}{\texttt{tile}(l, v) : Loop}$$

Figure 6.13: General typing rules. $\overline{m}$ represents the maximum bound of the dimension denoted by r. $S = (N_1, \cdots, N_{\overline{r}})$

$$\frac{\begin{array}{c} t_1 : Tens(S_1) \quad t_2 : Tens(S_2) \\ F_1 = [r_{1_1}, \cdots, r_{n_1}] \quad F_2 = [r_{1_2}, \cdots, r_{n_2}] \quad F' = [r'_1, \cdots, r'_n] \\ \text{OR} \\ t_1 : Tens(S_1) \quad t_2 : Tens(\epsilon) \\ F_1 = [r_{1_1}, \cdots, r_{n_1}] \quad F' = [r'_1, \cdots, r'_n] \\ \text{OR} \\ t_1 : Tens(\epsilon) \quad t_2 : Tens(S_2) \\ F_2 = [r_{1_2}, \cdots, r_{n_2}] \quad F' = [r'_1, \cdots, r'_n] \\ \text{OR} \\ t_1 : Tens(\epsilon) \quad t_2 : Tens(\epsilon) \quad F' = [r'_1, \cdots, r'_n] \\ \text{AND} \\ r_{k_1}, r_{k_2} \in \mathbb{N}^+ \\ length(F_k) = length(S_k) \ (1 < k \leq 2) \end{array}}{\texttt{op}(t_1, t_2, [F_1?, F_2?] \rightarrow F') : Tens(S')}$$

where (S') must be inferred.

Figure 6.14: Typing rule of op. The same applies to eq and vop. Note that $\texttt{vop}(t_1, t_2, [F_1?, F_2?]) : Tens(\epsilon)$. $S = (N_1, \cdots, N_{\overline{r}})$

coupling with a type system to ensure that programs that are not compliant to proper range inference are filtered out.

Chapter 7

Conclusion and Perspectives

Traduction en Français

Diverses représentations intermédiaires pour les programmes explicitement parallèles ont été proposées ces 25 dernières années. Ces contributions portent essentiellement sur la création de nouveaux langages ou sur l'extension de RIs populaires telles que le graphe de flot de contrôle, la forme d'assignation unique ou le modèle polyédrique. De nombreuses perspectives de recherches demeurent donc ouvertes. Premièrement, la conception de base des RIs est discutable. En raison de problèmes tels que la portabilité ou la recherche de transformations pertinentes et puissantes pour un programme donné, les RIs actuels de compilateurs rendent difficile la résolution de ces problèmes. Par conséquent, différentes alternatives aux chaînes de compilation classiques ont été explorées, incluant par exemple des chaînes de compilation multi-couches où différents niveaux d'expertise sont assemblés pour composer une chaîne puissante, ou encore des outils de recherche empiriques d'optimisations. Dans de tels contextes, différents types de représentations intermédiaires peuvent être nécessaires pour (i) composer efficacement des séquences de transformations et (ii) traiter la génération de plusieurs versions d'un programme.

Deuxièmement, l'expressivité est discutable. Très peu de contributions abordent la question des spécificités du domaine d'application et de l'architecture. La plupart des RIs de l'état-de-l'art sont à portée générale et ignorent les architectures cibles. Des RIs générales sont bien entendu nécessaires mais une complémentarité apportée par des RIs spécifiques à un domaine est importante. De plus, la compilation parallèle sans tenir compte de l'architecture est forcément incomplète.

Nous avons donc conçu TeML avec plusieurs spécificités ; TeML est:

- Un méta-langage fonctionnel avec une grande capacité de composition pour composer et générer facilement différentes séquences de transformations (notamment les transformations de boucle et de layout) ;
- Spécifiquement conçu pour l'optimisation d'applications tensorielles ;

- *NUMA-aware*, c'est-à-dire, qu'il intègre un support de haut niveau pour les emplacements de données sur NUMA.

Nous avons également spécifié formellement la sémantique de TeML, y compris l'algorithme d'inférence de domaines sur lequel elle s'appuie.

Le travail présenté dans cette thèse peut être naturellement suivi de deux perspectives principales.

Une conception et une implémentation robuste

Divers aspects de TeML restent à être améliorés.

- L'état actuel de TeML fournit principalement une abstraction pour des boucles parfaitement imbriquées, de la création de boucles à leurs transformations. Des abstractions adéquates pour les boucles imparfaitement imbriquées sont donc nécessaires.
- Nous utilisons une approche conservatrice de la notation d'index afin de faciliter l'identification des dimensions de boucles. Si TeML est utilisé uniquement comme langage intermédiaire, cela ne pose aucun problème. Cependant, pour une utilisation en méta-programmation, il semble préférable d'assouplir une telle contrainte et d'améliorer l'identification de la boucle.
- Nos travaux et expériences se concentrent sur les tenseurs de type dense. Une application plus large nécessite la prise en charge de tenseurs *sparse*, y compris une représentation correcte (différentes méthodes de stockage telles que *Coordinate* (COO), *Compression Sparse Row* (CSR), *Block CSR*), les formats *DIA* ou *ELL*) ou la prise en charge de transformations de disposition et de code spécifiques.
- Une étude des problématiques liées à la disposition des données en vue d'une vectorisation efficace, ainsi que la manière dont cela pourrait être représenté dans TeML, serait intéressante et pourrait être étendue au support pour les GPUs.

Une spécification formelle complète

Nous avons mis l'accent sur la sémantique formelle décrivant la construction et la transformation du programme. Cependant, la sémantique du placement des données dans les hiérarchies de mémoire reste à être spécifiée. De plus, la gestion de la dépendance des données dans la sémantique pourrait être utile. Il serait également intéressant d'étudier en profondeur la spécification du système de type et de renforcer sa connexion avec la sémantique dénotationnelle et l'inférence de domaines non itérative. Il s'agit d'un sujet intéressant car des questions concernant les propriétés des transformations méta-programmées seraient mises en avant. Etant donné que TeML partage des fonctionnalités avec d'autres méta-langages d'optimisation, nous pensons qu'un tel travail est une étape vers la formalisation de ce type de méta-langage.

* ♣ * ♣ *

Various parallel intermediate representations have been proposed for the last 25 years, for which our survey provides an overview. Belwal and Surdachan [31] published a survey on IRs for heterogeneous multi-core; they include IRs for automatic parallelization as well as the compilation of explicitly parallel programs. Our survey takes a different point of view, much more focused on explicitly parallel programs. Consequently, we take into account a considerable amount of contributions not included in [31]: extensions of commonly used IRs such as the control flow graph and the static single assignment form, different approaches of polyhedral compilation and parallel intermediate languages.

Contributions of the state-of-art generally focus on extending existing IRs for the support of parallel execution. This leaves rooms for several research perspectives. First, the core design of IR is questionable. Due to issues such as portability or finding relevant and powerful transformations for a given program, current compiler IRs make it difficult to tackle such challenges. Therefore, different alternative to classic compilation chains have been explored; this includes, for example, multi-layer compilation chains where different levels of expertise are put together to compose a powerful chain or empirical autotuning tools in which optimizations are iteratively performed using performance feedback until a program variant, suitable for the target architecture, is found. In such contexts, different types of intermediate representations may be required to (i) efficiently compose sequences of transformations and (ii) address the generation of multiple versions of a program.

Second, the expressiveness is questionable. Very few contributions address the question of domain- and architecture-specificities. Most of IRs in the state-of-art are general purpose and unaware of target architectures. General purpose IRs are necessary, as well as domain-specific ones. Moreover, parallel compilation without architecture awareness is *incomplete*.

Therefore we have designed TeML with several specificities; TeML is:

- A functional meta-language with a high-degree of composition capability to compose and generate easily different sequences of transformations (include loop-level and layout transformations);
- Domain-specific to tensor optimizations in numerical applications;
- NUMA-aware with high-level support for data placements.

We also formally specified the semantics of TeML, including the range inference algorithm on which it relies.

The work presented in this thesis can be naturally followed up with two main perspectives.

A robust design and implementation

There is room for further improvements in various aspects of TeML.

- The current state of TeML mainly provides abstractions for perfectly nested loops, from the creation of loops to their transformations. Adequate abstractions for imperfectly nested loops are therefore required.

- We make use of a conservative approach to the index notation in order to ease the identification of loop dimensions. If TeML is purely used as an intermediate language, this is not problematic. However, for a use in meta-programming, it seems preferable to relax such a constraint and improve loop identification.
- Our work and experiments are focused on dense tensors. A wider application scope requires the support for sparse tensor algebra including proper representation (i.e. different storing methods such as the *Coordinate* (COO), *Compressed Sparse Row* (CSR), *Block CSR* (BCSR), *DIA* or *ELL* formats) or support for specific layout and code transformations.
- It is also crucial to extend TeML for GPU support. Indeed, many numerical applications rather run on GPUs. Data placement problematics related to GPUs could be therefore investigated. Generally, a good support for heterogeneous computing is required. Note that the CFD applications on their larger scope do leverage heterogeneous execution involving GPUs and MPI processes.

A full formal specification

We have emphasized on formal semantics describing program construction and transformation. However, semantics for data placement on memory hierarchies are still left to do. In addition, encoding data dependency management in the semantics may be useful. It would also be interesting to deeply investigate TeML's type system specification and strengthening its connection with the denotational semantics and non-iterative range inference. This is an interesting topic as questions regarding properties of meta-programmed transformations would be raised. As TeML shares features with other optimization meta-languages, we believe that such a work is a step towards the formalization of such type of languages.

Bibliography

[1] Chapel. https://www.chapel.cray.com.

[2] CUDA C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[3] GNU UPC. http://www.gccupc.org.

[4] HPF: High Performance Fortran. http://hpff.rice.edu.

[5] HPX: High Performance ParalleX. http://www.stellar.cct.lsu.edu/projets/hpx.

[6] Intel® Cilk™ Plus. https://www.cilkplus.org.

[7] MPI: Message Passing Interface. http://www.mpi-forum.org.

[8] OpenCL. https://www.khronos.org/opencl/.

[9] OpenMP. https://www.openmp.org.

[10] PIPS. https://www.pips4u.org.

[11] PolyBench: Polyhedral Benchmark Suite. https://sourceforge.net/projects/polybench/.

[12] POSIX Threads. urlhttps://computing.llnl.gov/tutorials/pthreads/.

[13] RedBaron. http://redbaron.readthedocs.io/en/latest/.

[14] UPC: Unified Parallel C. http://www.upc.lbl.gov.

[15] X10. http://www.x10-lang.org.

[16] Clan, A Polyhedral Representation Extraction Tool for C-Based High Level Languages. http://icps.u-strasbg.fr/~bastoul/development/clan/, 2014.

[17] NumPy, package for scientific computing with Python. http://www.numpy.org/, 2017.

[18] XLA: Accelerated Linear Algebra. https://www.tensorflow.org/performance/xla/, 2017.

[19] Xtensor, Multi-dimensional arrays with broadcasting and lazy computing. https://github.com/QuantStack/xtensor, 2017.

[20] Abadi, M., and et al., A. A. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. http://download.tensorflow.org/paper/whitepaper2015.pdf, 2015.

[21] Agarwal, S., Barik, R., Sarkar, V., and Shyamasundar, R. K. May-happen-in-parallel Analysis of X10 Programs. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2007), PPoPP '07, ACM, pp. 183–193.

[22] Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[23] Allen, F. E. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization* (New York, NY, USA, 1970), ACM, pp. 1–19.

[24] Baghdadi, R., Beaugnon, U., Cohen, A., Grosser, T., Kruse, M., Reddy, C., Verdoolaege, S., Betts, A., Donaldson, A. F., Ketema, J., Absar, J., Van Haastregt, S., Kravets, A., Lokhmotov, A., David, R., and Hajiyev, E. PENCIL: A Platform-Neutral Compute Intermediate Language for Accelerator Programming. In *Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques* (2015), PACT '15.

[25] Baghdadi, R., Cohen, A., Grosser, T., Verdoolaege, S., Lokhmotov, A., Absar, J., Van Haastregt, S., Kravets, A., and Donaldson, A. PENCIL Language Specification. Research Report RR-8706, INRIA, May 2015.

[26] Bagnères, L., Zinenko, O., Huot, S., and Bastoul, C. Opening polyhedral compiler's black box. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization* (New York, NY, USA, 2016), CGO '16, ACM, pp. 128–138.

[27] Bastoul, C. Code Generation in the Polyhedral Model is Easier than You Think. In *Proceedings of the Inernation Conference on Parallel Architectures and Compilation Techniques (PACT'04)* (2004).

[28] Basupalli, V., Yuki, T., Rajopadhye, S., Morvan, A., Derrien, S., Quinton, P., and Wonnacott, D. ompVerify: Polyhedral Analysis for the OpenMP Programmer. In *Proceedings of the 7th International Conference on OpenMP in the Petascale Era* (Berlin, Heidelberg, 2011), IWOMP'11, Springer-Verlag, pp. 37–53.

[29] Baumgartner, G., Auer, A., Bernholdt, D. E., Bibireata, A., Choppella, V., Cociorva, D., Gao, X., Harrison, R. J., Hirata, S., Krishnamoorthy, S., Krishnan, S., chung Lam, C., Lu, Q., Nooijen, M., Pitzer, R. M., Ramanujam, J., Sadayappan, P., and Sibiryakov, A. Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. *Proceedings of the IEEE 93*, 2 (Feb 2005), 276–292.

[30] Beaugnon, U., Kravets, A., van Haastregt, S., Baghdadi, R., Tweed, D., Absar, J., and Lokhmotov, A. VOBLA: A Vehicle for Optimized Basic Linear Algebra. *SIGPLAN Notices 49*, 5 (2014), 115–124.

[31] Belwal, M., and Sudarshan, T. Intermediate Representation for Heterogeneous Multi-Core: A Survey. In *VLSI Systems, Architecture, Technology and Applications (VLSI-SATA), 2015 International Conference on* (Jan 2015), pp. 1–6.

[32] Benoit, N., and Louise, S. Extending GCC with a Multi-Grain Parallelism Adaptation Framework for MPSoCs. In *GCC for Research Opportunities Workshop* (2010).

[33] Benoit, N., and Louise, S. Kimble: a Hierarchical Intermediate Representation for Multi-Grain Parallelism. In *Proceedings of the Workshop on Intermediate Representations* (Chamonix, France, 2011), WIR-1, pp. 21–28.

[34] Benoit, N., and Louise, S. Using an Intermediate Representation to Map Workloads on Heterogeneous Parallel Systems. In *PDP'16* (2016), IEEE Computer Society, pp. 811–819.

[35] Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D., and Bengio, Y. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)* (June 2010).

[36] Bircsak, J., Craig, P., Crowell, R., Cvetanovic, Z., Harris, J., Nelson, C. A., and Offner, C. D. Extending OpenMP for NUMA Machines. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing* (Washington, DC, USA, 2000), SC '00, IEEE Computer Society.

[37] Bondhugula, U. Compiling Affine Loop Nests for Distributed-memory Parallel Architectures. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2013), SC '13, ACM, pp. 33:1—-33:12.

[38] Bondhugula, U., Hartono, A., Ramanujam, J., and Sadayappan, P. A practical automatic polyhedral program optimization system. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2008).

[39] Bondhugula, U., Hartono, A., Ramanujam, J., and Sadayappan, P. A practical automatic polyhedral program optimization system. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (June 2008).

[40] Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., and Namyst, R. hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing* (Pisa, Italy, Feb. 2010), IEEE, Ed.

[41] Broquedis, F., Furmento, N., Goglin, B., Wacrenier, P.-A., and Namyst, R. Forestgomp: An efficient openmp environment for numa architectures. *International Journal of Parallel Programming 38*, 5 (2010), 418–439.

[42] Brown, K. J., Sujeeth, A. K., Lee, H. J., Rompf, T., Chafi, H., Odersky, M., and Olukotun, K. A Heterogeneous Parallel Framework for Domain-Specific Languages. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques* (Washington, DC, USA, 2011), PACT '11, IEEE Computer Society, pp. 89–100.

[43] Bruening, D., Devabhaktuni, S., and Amarasinghe, S. Softspec: Software-based speculative parallelism. In *In 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3* (1998).

[44] Cavé, V., Zhao, J., Shirako, J., and Sarkar, V. Habanero-Java: The New Adventures of Old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java* (New York, NY, USA, 2011), PPPJ '11, ACM, pp. 51–61.

[45] Chakrabarti, D. R., and Banerjee, P. Static Single Assignment Form for Message-Passing Programs. *Int. J. Parallel Program. 29*, 2 (Apr. 2001), 139–184.

[46] Chatarasi, P., Shirako, J., and Sarkar, V. Polyhedral Transformations of Explicitly Parallel Programs. In *Proceedings of the Fifth International Workshop on Polyhedral Compilation Techniques* (2015), IMPACT '15.

[47] Chatarasi, P., Shirako, J., and Sarkar, V. Static Data Race Detection for SPMD Programs via an Extended Polyhedral Representation. In *Proceedings of the Sixth International Workshop on Polyhedral Compilation Techniques* (2016), IMPACT '16.

[48] Chen, C., Chame, J., and Hall, M. Chill: A framework for composing high-level loop transformations. Tech. rep., Technical Report 08-897, University of Southern California, 2008.

[49] Chen, T. Typesafe abstractions for tensor operations (short paper). In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala* (New York, NY, USA, 2017), SCALA 2017, ACM, pp. 45–50.

[50] Chen, T., Moreau, T., Jiang, Z., Shen, H., Yan, E. Q., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A. TVM: end-to-end optimization stack for deep learning. *CoRR abs/1802.04799* (2018).

[51] Choi, Y., Lin, Y., Chong, N., Mahlke, S., and Mudge, T. Stream Compilation for Real-Time Embedded Multicore Systems. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Washington, DC, USA, 2009), CGO '09, IEEE Computer Society, pp. 210–220.

[52] Cohen, A., Darte, A., and Feautrier, P. Static Analysis of OpenStream Programs. In *Proceedings of the Sixth International Workshop on Polyhedral Compilation Techniques* (2016), IMPACT '16.

[53] Cohen, A., Girbal, S., and Temam, O. *A Polyhedral Approach to Ease the Composition of Program Transformations.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 292–303.

[54] Cohen, A., Sigler, M., Girbal, S., Temam, O., Parello, D., and Vasilache, N. Facilitating the search for compositions of program transformations. In *Proceedings of the 19th Annual International Conference on Supercomputing* (New York, NY, USA, 2005), ICS '05, ACM, pp. 151–160.

[55] Collard, J.-F. Array SSA for Explicitly Parallel Programs. In *Proceedings of the 5th International Euro-Par Conference on Parallel Processing* (London, UK, UK, 1999), Euro-Par '99, Springer-Verlag, pp. 383–390.

[56] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst. 13*, 4 (Oct. 1991), 451–490.

[57] Darte, A., Isoard, A., and Yuki, T. Liveness Analysis in Explicitly Parallel Programs. In *Proceedings of the Sixth International Workshop on Polyhedral Compilation Techniques* (2016), IMPACT '16.

[58] Dashti, M., Fedorova, A., Funston, J., Gaud, F., Lachaize, R., Lepers, B., Quema, V., and Roth, M. Traffic management: A holistic approach to memory placement on numa systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2013), ASPLOS '13, ACM, pp. 381–394.

[59] Davidson, J. W., and Jinturkar, S. Memory access coalescing: A technique for eliminating redundant memory accesses. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation* (New York, NY, USA, 1994), PLDI '94, ACM, pp. 186–195.

[60] Deng, X., Mao, M., Tu, G., Zhang, H., and Zhang, Y. High-order and high accurate cfd methods and their applications for complex grid problems. *Communications in Computational Physics 11*, 4 (2012), 1081–1102.

[61] Donadio, S., Brodman, J., Roeder, T., Yotov, K., Barthou, D., Cohen, A., Garzarán, M. J., Padua, D., and Pingali, K. *A Language for the Compact Representation of Multiple Program Versions.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 136–151.

[62] Fassi, I., and Clauss, P. Xfor: Filling the gap between automatic loop optimization and peak performance. In *2015 14th International Symposium on Parallel and Distributed Computing* (June 2015), pp. 100–109.

[63] Feautrier, P., and Lengauer, C. *Polyhedron Model.* Springer US, Boston, MA, 2011, pp. 1581–1592.

[64] Ferrante, J., Ottenstein, K. J., and Warren, J. D. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst. 9*, 3 (July 1987), 319–349.

[65] Goglin, B., and Furmento, N. Enabling High-performance Memory Migration for Multithreaded Applications on LINUX. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing* (Washington, DC, USA, 2009), IPDPS '09, IEEE Computer Society, pp. 1–9.

[66] Grosser, T., Zheng, H., Aloor, R., Simbürger, A., Grösslinger, A., and Pouchet, L.-N. Polly - Polyhedral Optimization in LLVM. In *Proceedings of the Sixth International Workshop on Polyhedral Compilation Techniques* (2011), IMPACT '11.

[67] Henretty, T., Stock, K., Pouchet, L.-N., Franchetti, F., Ramanujam, J., and Sadayappan, P. Data layout transformation for stencil computations on short-vector simd architectures. In *Proceedings of the 20th International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software* (Berlin, Heidelberg, 2011), CC'11/ETAPS'11, Springer-Verlag, pp. 225–245.

[68] Huang, L., Jin, H., Yi, L., and Chapman, B. Enabling Locality-aware Computations in OpenMP. *Sci. Program. 18*, 3-4 (Aug. 2010), 169–181.

[69] Huismann, I., Stiller, J., and Fröhlich, J. *Fast Static Condensation for the Helmholtz Equation in a Spectral-Element Discretization.* Springer International Publishing, Cham, 2016, pp. 371–380.

[70] Huismann, I., Stiller, J., and Fröhlich, J. Factorizing the factorization — a spectral-element solver for elliptic equations with linear operation count. *Journal of Computational Physics 346* (2017), 437–448.

[71] Ibrahim, K. Z., Williams, S. W., Epifanovsky, E., and Krylov, A. I. Analysis and tuning of libtensor framework on multicore architectures. In *21st International Conference on High Performance Computing, HiPC 2014, Goa, India, December 17-20, 2014* (2014), pp. 1–10.

[72] Jimborean, A., Clauss, P., Dollinger, J.-F., Loechner, V., and Martinez Caamaño, J. M. Dynamic and speculative polyhedral parallelization using compiler-generated skeletons. *Int. J. Parallel Program. 42*, 4 (Aug. 2014), 529–545.

[73] Jordan, H., Pellegrini, S., Thoman, P., Kofler, K., and Fahringer, T. INSPIRE: The Insieme Parallel Intermediate Representation. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques* (Piscataway, NJ, USA, 2013), PACT '13, IEEE Press, pp. 7–18.

[74] Juan Manuel Martinez Caamano, Aravind Sukumaran-Rajam, A. B. M. S., and Clauss, P. APOLLO: Automatic speculative POLyhedral Loop Optimizer. In *Proceedings of the Sixth International Workshop on Polyhedral Compilation Techniques*, IMPACT '17.

[75] Kennedy, K., and Allen, J. R. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[76] Khaldi, D. *Automatic Resource-Constrained Static Task Parallelization.* PhD thesis, MINES ParisTech, PSL Research University, 2013.

[77] Khaldi, D., Jouvelot, P., Irigoin, F., and Ancourt, C. SPIRE : A Methodology for Sequential to Parallel Intermediate Representation Extension. In *HiPEAC Computing Systems Week* (Paris, France, 2013).

[78] Khaldi, D., Jouvelot, P., Irigoin, F., Ancourt, C., and Chapman, B. LLVM Parallel Intermediate Representation: Design and Evaluation Using OpenSHMEM Communications. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC* (New York, NY, USA, 2015), LLVM '15, ACM, pp. 2:1—-2:8.

[79] Kjolstad, F., Kamil, S., Chou, S., Lugato, D., and Amarasinghe, S. The tensor algebra compiler. *Proc. ACM Program. Lang. 1*, OOPSLA (Oct. 2017), 77:1–77:29.

[80] Klöckner, A. Loo.py: Transformation-based code generation for gpus and cpus. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming* (New York, NY, USA, 2014), ARRAY'14, ACM, pp. 82:82–82:87.

[81] Knijnenburg, P. M. W., Kisuki, T., and O'Boyle, M. F. P. Embedded processor design challenges. Springer-Verlag New York, Inc., New York, NY, USA, 2002, ch. Iterative Compilation, pp. 171–187.

[82] Knobe, K., and Sarkar, V. Array SSA Form and Its Use in Parallelization. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1998), POPL '98, ACM, pp. 107–120.

[83] Landi, W., and Ryder, B. G. A safe approximate algorithm for interprocedural aliasing. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation* (New York, NY, USA, 1992), PLDI '92, ACM, pp. 235–248.

[84] Lee, J., Padua, D. A., and Midkiff, S. P. Basic Compiler Algorithms for Parallel Programs. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 1999), PPoPP '99, ACM, pp. 1–12.

[85] Lin, Y. Static Nonconcurrency Analysis of OpenMP Programs. In *OpenMP Shared Memory Parallel Programming*. Springer, 2008, pp. 36–50.

[86] Liu, W., Tuck, J., Ceze, L., Ahn, W., Strauss, K., Renau, J., and Torrellas, J. Posh: A tls compiler that exploits program structure. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2006), PPoPP '06, ACM, pp. 158–167.

[87] Loechner, V. PolyLib: A library for manipulating parameterized polyhedra, 1999.

[88] Luporini, F., Varbanescu, A. L., Rathgeber, F., Bercea, G., Ramanujam, J., Ham, D. A., and Kelly, P. H. J. COFFEE: an optimizing compiler for finite element local assembly. *CoRR abs/1407.0904* (2014).

[89] Majo, Z., and Gross, T. R. Matching Memory Access Patterns and Data Placement for NUMA Systems. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization* (New York, NY, USA, 2012), CGO '12, ACM, pp. 230–241.

[90] Majo, Z., and Gross, T. R. A Library for Portable and Composable Data Locality Optimizations for NUMA Systems. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2015), PPoPP 2015, ACM, pp. 227–238.

[91] Meister, B., Vasilache, N., Wohlford, D., Baskaran, M. M., Leung, A., and Lethin, R. *R-Stream Compiler.* Springer US, Boston, MA, 2011, pp. 1756–1765.

[92] Miranda, C. *Erbium: Reconciling Languages, Runtimes, Compilation and Optimizations for Streaming Applications.* PhD thesis, Université de Paris-Sud, 2013.

[93] Miranda, C., Pop, A., Dumont, P., Cohen, A., and Duranton, M. Erbium: A Deterministic, Concurrent Intermediate Representation to Map Data-flow Tasks to Scalable, Persistent Streaming Processes. In *Proceedings of the 2010 International Conference on Compilers, Architectures and Synthesis for Embedded Systems* (New York, NY, USA, 2010), CASES '10, ACM, pp. 11–20.

[94] MUDDUKRISHNA, A., JONSSON, P. A., AND BRORSSON, M. Locality-Aware Task Scheduling and Data Distribution for OpenMP Programs on NUMA Systems and Manycore Processors. *Scientific Programming 2015* (2015).

[95] MÜLLER-PFEFFERKORN, R., NAGEL, W. E., AND TRENKLER, B. Optimizing cache access: A tool for source-to-source transformations and real-life compiler tests. In *Euro-Par 2004 Parallel Processing* (Berlin, Heidelberg, 2004), M. Danelutto, M. Vanneschi, and D. Laforenza, Eds., Springer Berlin Heidelberg, pp. 72–81.

[96] NANDY, P., HALL, M., DAVIS, E. C., OLSCHANOWSKY, C., MOHAMMADI, M. S., HE, W., AND STROUT, M. Abstractions for Specifying Sparse Matrix Data Transformations. In *Proceedings of the Eighth International Workshop on Polyhedral Compilation Techniques* (2018), IMPACT '18.

[97] NECULA, G. C., MCPEAK, S., RAHUL, S. P., AND WEIMER, W. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of the 11th International Conference on Compiler Construction* (London, UK, UK, 2002), CC '02, Springer-Verlag, pp. 213–228.

[98] NOVILLO, D., UNRAU, R. C., AND SCHAEFFER, J. Concurrent SSA Form in the Presence of Mutual Exclusion. In *Proceedings of the 1998 International Conference on Parallel Processing* (Washington, DC, USA, 1998), ICPP '98, IEEE Computer Society, pp. 356–.

[99] PAI, S., GOVINDARAJAN, R., AND THAZHUTHAVEETIL, M. J. PLASMA: Portable Programming for SIMD Heterogeneous Accelerators. In *Proceedings of the Workshop on Language, Compiler, and Architecture Support for GPGPU* (Bangalore, India, 2010), held in conjunction with HPCA/PPoPP 2010.

[100] PELLEGRINI, S. *On Simplifying and Optimizing Message Passing Programs: a Compiler and Runtime-Based Approach.* PhD thesis, University of Innsbruck, 2011.

[101] PELLEGRINI, S., HOEFLER, T., AND FAHRINGER, T. Exact dependence analysis for increased communication overlap. In *Proceedings of the 19th European Conference on Recent Advances in the Message Passing Interface* (Berlin, Heidelberg, 2012), EuroMPI'12, Springer-Verlag, pp. 89–99.

[102] PÉRACHE, M., JOURDREN, H., AND NAMYST, R. MPC: A Unified Parallel Runtime for Clusters of NUMA Machines. In *Proceedings of the 14th International Euro-Par Conference on Parallel Processing* (Berlin, Heidelberg, 2008), Euro-Par '08, Springer-Verlag, pp. 78–88.

[103] PLESCO, A. *Program Transformations and Memory Architecture Optimizations for High-Level Synthesis of Hardware Accelerators.* PhD thesis, Ecole Normale Supérieure de Lyon, 2010.

[104] Pop, A., and Cohen, A. Preserving High-Level Semantics of Parallel Programming Annotations Through the Compilation Flow of Optimizing Compilers. In *Proceedings of the 15th Workshop on Compilers for Parallel Computers* (2010), CPC'10.

[105] Pop, A., and Cohen, A. OpenStream: Expressiveness and Data-flow Compilation of OpenMP Streaming Programs. *ACM Transactions on Architecture and Code Optimization (TACO) 9*, 4 (2013), 53:1—-53:25.

[106] Pop, S., Cohen, A., Bastoul, C., Girbal, S., Silber, G.-A., and Vasilache, N. GRAPHITE: Polyhedral Analyses and Optimizations for GCC. In *Proceedings of the 2006 GCC Developers Summit* (2006), GCC Summit '06.

[107] Pousa Ribeiro, C., Castro, M., Méhaut, J.-F., and Carissimi, A. *High Performance Computing for Computational Science – VECPAR 2010: 9th International conference, Berkeley, CA, USA, June 22-25, 2010, Revised Selected Papers.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, ch. Improving Memory Affinity of Geophysics Applications on NUMA Platforms Using Minas, pp. 279–292.

[108] Pradelle, B., Meister, B., Baskaran, M., Springer, J., and Lethin, R. Polyhedral optimization of tensorflow computation graphs. In *Proceedings of the 6th Workshop on Extreme-scale Programming Tools at The International Conference for High Performance Computing, Networking, Storage and Analysis* (2017), EPST @SC '17.

[109] Qu, Z.-Q. *Static Condensation.* Springer London, London, 2004, pp. 47–70.

[110] Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., and Amarasinghe, S. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2013), PLDI '13, ACM, pp. 519–530.

[111] Rink, N. A. Modeling of languages for tensor manipulation. *CoRR abs/1801.08771* (2018).

[112] Rink, N. A., Huismann, I., Susungi, A., Castrillon, J., Stiller, J., Fröhlich, J., and Tadonki, C. Cfdlang: High-level code generation for high-order methods in fluid dynamics. In *Proceedings of the Real World Domain Specific Languages Workshop 2018* (New York, NY, USA, 2018), RWDSL2018, ACM, pp. 5:1–5:10.

[113] Rudy, G., Khan, M. M., Hall, M., Chen, C., and Chame, J. *A Programming Language Interface to Describe Transformations and Code Generation.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 136–150.

[114] Rus, S., He, G., Alias, C., and Rauchwerger, L. Region Array SSA. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques* (New York, NY, USA, 2006), PACT '06, ACM, pp. 43–52.

[115] Sarkar, V. Analysis and optimization of explicitly parallel programs using the parallel program graph representation. In *Proceedings of the 10th International Workshop on Languages and Compilers for Parallel Computing* (London, UK, UK, 1998), LCPC '97, Springer-Verlag, pp. 94–113.

[116] Sarkar, V., and Simons, B. Parallel Program Graphs and Their Classification. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing* (London, UK, UK, 1994), Springer-Verlag, pp. 633–655.

[117] Schardl, T. B., Moses, W. S., and Leiserson, C. E. Tapir: Embedding fork-join parallelism into llvm's intermediate representation. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2017), PPoPP '17, ACM, pp. 249–265.

[118] Shirako, J., Zhao, J. M., Nandivada, V. K., and Sarkar, V. N. Chunking Parallel Loops in the Presence of Synchronization. In *Proceedings of the 23rd International Conference on Supercomputing* (New York, NY, USA, 2009), ICS '09, ACM, pp. 181–192.

[119] Shires, D. R., and Pollock, L. Program Flow Graph Construction for Static Analysis of Explicitly Parallel Message-Passing Programs. Tech. rep., Army Research Laboratory, 2000.

[120] Spampinato, D. G., Fabregat-Traver, D., Bientinesi, P., and Püschel, M. Program generation for small-scale linear algebra applications. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization* (New York, NY, USA, 2018), CGO 2018, ACM, pp. 327–339.

[121] Spampinato, D. G., and Püschel, M. A basic linear algebra compiler. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (New York, NY, USA, 2014), CGO '14, ACM, pp. 23:23–23:32.

[122] Spampinato, D. G., and Püschel, M. A basic linear algebra compiler for structured matrices. In *International Symposium on Code Generation and Optimization (CGO)* (2016), pp. 117–127.

[123] Springer, P., Sankaran, A., and Bientinesi, P. TTC: A tensor transposition compiler for multiple architectures. *CoRR abs/1607.01249* (2016).

[124] Srinivasan, H., Hook, J., and Wolfe, M. Static Single Assignment for Explicitly Parallel Programs. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1993), POPL '93, ACM, pp. 260–272.

[125] Srinivasan, H., and Wolfe, M. Analyzing Programs with Explicit Parallelism. In *Languages and Compilers for Parallel Computing*, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Eds., vol. 589 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1992, pp. 405–419.

[126] Stanier, J., and Watson, D. Intermediate Representations in Imperative Compilers: A Survey. *ACM Computing Surveys 45*, 3 (July 2013), 26:1–26:27.

[127] Steuwer, M., Remmelg, T., and Dubach, C. Lift: A functional data-parallel ir for high-performance gpu code generation. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization* (Piscataway, NJ, USA, 2017), CGO '17, IEEE Press, pp. 74–85.

[128] Stoltz, E., Gerlek, M. P., and Wolfe, M. Extended ssa with factored use-def chains to support optimization and parallelism. In *System Sciences, 1994. Proceedings of the Twenty-Seventh Hawaii International Conference on* (Jan 1994), vol. 2, pp. 43–52.

[129] Strout, M. M., Kreaseck, B., and Hovland, P. D. Data-flow Analysis for MPI Programs. In *Internationl Conference on Parallel Processing, 2006* (2006), ICPP '06, IEEE, pp. 175–184.

[130] Susungi, A., Cohen, A., and Tadonki, C. More data locality for static control programs on numa architectures. In *Proceedings of the 7th International Workshop on Polyhedral Compilation Techniques* (2017), IMPACT '17.

[131] Thoman, P. *Insieme-RS: A Compiler-supported Parallel Runtime System.* PhD thesis, University of Innsbruck, 2013.

[132] Valiev, M., Bylaska, E., Govind, N., Kowalski, K., Straatsma, T., Dam, H. V., Wang, D., Nieplocha, J., Apra, E., Windus, T., and de Jong, W. Nwchem: A comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications 181*, 9 (2010), 1477 – 1489.

[133] Vasilache, N., Zinenko, O., Theodoridis, T., Goyal, P., DeVito, Z., Moses, W. S., Verdoolaege, S., Adams, A., and Cohen, A. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *CoRR abs/1802.04730* (2018).

[134] Verdoolaege, S., and Grosser, T. Polyhedral Extraction Tool. In *Proceedings of the Sixth International Workshop on Polyhedral Compilation Techniques* (2012), IMPACT '12.

[135] Verdooleage, S. isl: An Integer Set Library for the Polyhedral Model. In *Mathematical Software (ICMS'10)* (2010), K. Fukuda, J. Hoeven, M. Joswig, and N. Takayama, Eds., LNCS 6327, Springer-Verlag, pp. 299–302.

[136] Verdooleage, S. Counting Affine Calculator and Applications. In *Proceedings of the Sixth International Workshop on Polyhedral Compilation Techniques* (2011), IMPACT '11.

[137] Verdooleage, S., Juega, J. C., Cohen, A., Gómez, J. I., Tenllado, C., and Catthoor, F. Polyhedral Parallel Code Generation for CUDA. *ACM Trans. Archit. Code Optim. 9*, 4 (2013), 54:1—-54:23.

[138] Xue, J. *Loop Tiling for Parallelism.* Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[139] Yi, Q., Seymour, K., You, H., Vuduc, R., and Quinlan, D. Poet: Parameterized optimizations for empirical tuning. In *2007 IEEE International Parallel and Distributed Processing Symposium* (March 2007), pp. 1–8.

[140] Yuki, T., Feautrier, P., Rajopadhye, S., and Saraswat, V. Array Dataflow Analysis for Polyhedral X10 Programs. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2013), PPoPP '13, ACM, pp. 23–34.

[141] Zhao, J., and Sarkar, V. Intermediate Language Extensions for Parallelism. In *Proceedings of the Compilation of the Co-located Workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11* (New York, NY, USA, 2011), SPLASH '11 Workshops, ACM, pp. 329–340.

[142] Zory, J., and Coelho, F. Using algebraic transformations to optimize expression evaluation in scientific code. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques* (Washington, DC, USA, 1998), PACT '98, IEEE Computer Society, pp. 376–.

Appendix A

ICC Optimizations Reports

A.1 Naive Interpolation

```
1  Intel(R) Advisor can now assist with vectorization and show optimization
2  report messages with your source code.
3  See "https://software.intel.com/en-us/intel-advisor-xe" for details.
4
5  Intel(R) C Intel(R) 64 Compiler for applications running on Intel(R) 64, Version
   ↪ 18.0.2.199 Build 20180210
6
7  Compiler options: -qopenmp -xHost -O3 -o interpol_naive -g -qopt-report=5
   ↪ -qopt-report-phase=vec,openmp,loop,par
8
9  Begin optimization report for: main(int *, char **)
10
11 Report from: Loop nest, Vector & Auto-parallelization optimizations [loop, vec, par]
12
13
14
15 Intel(R) Advisor can now assist with vectorization and show optimization
16 report messages with your source code.
17 See "https://software.intel.com/en-us/intel-advisor-xe" for details.
18
19 Intel(R) C Intel(R) 64 Compiler for applications running on Intel(R) 64, Version
   ↪ 18.0.2.199 Build 20180210
20
21 Compiler options: -qopenmp -xHost -O3 -o interpol_naive -g -qopt-report=5
   ↪ -qopt-report-phase=vec,openmp,loop,par
22
23 Begin optimization report for: main(int *, char **)
24
25 Report from: Loop nest, Vector & Auto-parallelization optimizations [loop, vec, par]
26
27
28 LOOP BEGIN at interpol_naive.c(66,3)
29 remark #25444: Loopnest Interchanged: ( 1 2 3 4 5 6 7 ) --> ( 1 2 3 5 6 4 7 )
```

```
30  remark #25440: unrolled and jammed by 4   (pre-vector)
31  remark #15541: outer loop was not auto-vectorized: consider using SIMD directive   [
    ↪  interpol_naive.c(72,8) ]
32
33  LOOP BEGIN at interpol_naive.c(67,5)
34  remark #25440: unrolled and jammed by 4   (pre-vector)
35  remark #15423: loop was not vectorized: has only one iteration
36
37  LOOP BEGIN at interpol_naive.c(68,7)
38  remark #15541: outer loop was not auto-vectorized: consider using SIMD directive   [
    ↪  interpol_naive.c(72,8) ]
39
40  LOOP BEGIN at interpol_naive.c(70,4)
41  remark #15344: loop was not vectorized: vector dependence prevents vectorization
42  remark #15346: vector dependence: assumed ANTI dependence between v[e][i1][i2][i3]
    ↪  (73:3) and v[e][i1][i2][i3] (73:3)
43  remark #15346: vector dependence: assumed FLOW dependence between v[e][i1][i2][i3]
    ↪  (73:3) and v[e][i1][i2][i3] (73:3)
44
45  LOOP BEGIN at interpol_naive.c(71,6)
46  remark #15344: loop was not vectorized: vector dependence prevents vectorization
47  remark #15346: vector dependence: assumed ANTI dependence between v[e][i1][i2][i3]
    ↪  (73:3) and v[e][i1][i2][i3] (73:3)
48  remark #15346: vector dependence: assumed FLOW dependence between v[e][i1][i2][i3]
    ↪  (73:3) and v[e][i1][i2][i3] (73:3)
49
50  LOOP BEGIN at interpol_naive.c(69,2)
51  remark #15541: outer loop was not auto-vectorized: consider using SIMD directive   [
    ↪  interpol_naive.c(72,8) ]
52
53  LOOP BEGIN at interpol_naive.c(72,8)
54  remark #25085: Preprocess Loopnests: Moving Out Load and Store    [
    ↪  interpol_naive.c(73,3) ]
55  ...
56  remark #25085: Preprocess Loopnests: Moving Out Load and Store    [
    ↪  interpol_naive.c(73,3) ]
57  remark #15389: vectorization support: reference A[i3][i6] has unaligned access   [
    ↪  interpol_naive.c(73,47) ]
58  remark #15389: vectorization support: reference u[e][i4][i5][i6] has unaligned access
    ↪  [ interpol_naive.c(73,59) ]
59  ...
60  remark #15381: vectorization support: unaligned access used inside loop body
61  remark #15335: loop was not vectorized: vectorization possible but seems inefficient.
    ↪  Use vector always directive or -vec-threshold0 to override
62  remark #15305: vectorization support: vector length 4
63  remark #15309: vectorization support: normalized vectorization overhead 1.971
64  remark #15450: unmasked unaligned unit stride loads: 5
65  remark #15475: --- begin vector cost summary ---
66  remark #15476: scalar cost: 200
67  remark #15477: vector cost: 42.500
68  remark #15478: estimated potential speedup: 1.260
```

```
69  remark #15488: --- end vector cost summary ---
70  remark #25436: completely unrolled by 7
71  LOOP END
72  ...
73
74  LOOP BEGIN at interpol_naive.c(67,5)
75  <Remainder>
76  remark #15541: outer loop was not auto-vectorized: consider using SIMD directive   [
    ↪  interpol_naive.c(72,8) ]
77
78  LOOP BEGIN at interpol_naive.c(68,7)
79  remark #15541: outer loop was not auto-vectorized: consider using SIMD directive   [
    ↪  interpol_naive.c(72,8) ]
80
81  LOOP BEGIN at interpol_naive.c(70,4)
82  remark #15344: loop was not vectorized: vector dependence prevents vectorization
83  remark #15346: vector dependence: assumed ANTI dependence between v[e][i1][i2][i3]
    ↪  (73:3) and v[e][i1][i2][i3] (73:3)
84  remark #15346: vector dependence: assumed FLOW dependence between v[e][i1][i2][i3]
    ↪  (73:3) and v[e][i1][i2][i3] (73:3)
85
86  LOOP BEGIN at interpol_naive.c(71,6)
87  remark #15344: loop was not vectorized: vector dependence prevents vectorization
88  remark #15346: vector dependence: assumed ANTI dependence between v[e][i1][i2][i3]
    ↪  (73:3) and v[e][i1][i2][i3] (73:3)
89  remark #15346: vector dependence: assumed FLOW dependence between v[e][i1][i2][i3]
    ↪  (73:3) and v[e][i1][i2][i3] (73:3)
90
91  LOOP BEGIN at interpol_naive.c(69,2)
92  remark #15541: outer loop was not auto-vectorized: consider using SIMD directive   [
    ↪  interpol_naive.c(72,8) ]
93  remark #25436: completely unrolled by 7
94
95  LOOP BEGIN at interpol_naive.c(72,8)
96  remark #25085: Preprocess Loopnests: Moving Out Load and Store    [
    ↪  interpol_naive.c(73,3) ]
97  remark #25085: Preprocess Loopnests: Moving Out Load and Store    [
    ↪  interpol_naive.c(73,3) ]
98  remark #25085: Preprocess Loopnests: Moving Out Load and Store    [
    ↪  interpol_naive.c(73,3) ]
99  remark #25085: Preprocess Loopnests: Moving Out Load and Store    [
    ↪  interpol_naive.c(73,3) ]
100 remark #15389: vectorization support: reference A[i3][i6] has unaligned access   [
    ↪  interpol_naive.c(73,47) ]
101 remark #15389: vectorization support: reference u[e][i4][i5][i6] has unaligned access
    ↪  [ interpol_naive.c(73,59) ]
102 remark #15389: vectorization support: reference A[i3][i6] has unaligned access   [
    ↪  interpol_naive.c(73,47) ]
103 remark #15389: vectorization support: reference u[e][i4][i5][i6] has unaligned access
    ↪  [ interpol_naive.c(73,59) ]
```

```
104 remark #15389: vectorization support: reference A[i3][i6] has unaligned access   [
    ↪  interpol_naive.c(73,47) ]
105 remark #15389: vectorization support: reference u[e][i4][i5][i6] has unaligned access
    ↪  [ interpol_naive.c(73,59) ]
106 remark #15389: vectorization support: reference A[i3][i6] has unaligned access   [
    ↪  interpol_naive.c(73,47) ]
107 remark #15389: vectorization support: reference u[e][i4][i5][i6] has unaligned access
    ↪  [ interpol_naive.c(73,59) ]
108 remark #15381: vectorization support: unaligned access used inside loop body
109 remark #15335: loop was not vectorized: vectorization possible but seems inefficient.
    ↪  Use vector always directive or -vec-threshold0 to override
110 remark #15305: vectorization support: vector length 4
111 remark #15309: vectorization support: normalized vectorization overhead 2.065
112 remark #15450: unmasked unaligned unit stride loads: 5
113 remark #15475: --- begin vector cost summary ---
114 remark #15476: scalar cost: 50
115 remark #15477: vector cost: 11.500
116 remark #15478: estimated potential speedup: 1.200
117 remark #15488: --- end vector cost summary ---
118 remark #25436: completely unrolled by 7
119 LOOP END
120
121 LOOP BEGIN at interpol_naive.c(72,8)
122 LOOP END
123
124 ...
125
126 LOOP BEGIN at interpol_naive.c(66,3)
127 <Remainder>
128 remark #15541: outer loop was not auto-vectorized: consider using SIMD directive   [
    ↪  interpol_naive.c(72,8) ]
129
130 LOOP BEGIN at interpol_naive.c(67,5)
131 remark #15541: outer loop was not auto-vectorized: consider using SIMD directive   [
    ↪  interpol_naive.c(72,8) ]
132
133 LOOP BEGIN at interpol_naive.c(68,7)
134 remark #15541: outer loop was not auto-vectorized: consider using SIMD directive   [
    ↪  interpol_naive.c(72,8) ]
135
136 LOOP BEGIN at interpol_naive.c(70,4)
137 remark #15541: outer loop was not auto-vectorized: consider using SIMD directive   [
    ↪  interpol_naive.c(72,8) ]
138
139 LOOP BEGIN at interpol_naive.c(71,6)
140 remark #15541: outer loop was not auto-vectorized: consider using SIMD directive   [
    ↪  interpol_naive.c(72,8) ]
141 remark #25436: completely unrolled by 7
142
143 LOOP BEGIN at interpol_naive.c(69,2)
```

```
144 remark #15541: outer loop was not auto-vectorized: consider using SIMD directive   [
    ↪  interpol_naive.c(72,8) ]
145 remark #25436: completely unrolled by 7
146
147 LOOP BEGIN at interpol_naive.c(72,8)
148 remark #25085: Preprocess Loopnests: Moving Out Load and Store    [
    ↪  interpol_naive.c(73,3) ]
149 remark #15389: vectorization support: reference A[i3][i6] has unaligned access   [
    ↪  interpol_naive.c(73,47) ]
150 remark #15389: vectorization support: reference u[e][i4][i5][i6] has unaligned access
    ↪  [ interpol_naive.c(73,59) ]
151 remark #15381: vectorization support: unaligned access used inside loop body
152 remark #15335: loop was not vectorized: vectorization possible but seems inefficient.
    ↪  Use vector always directive or -vec-threshold0 to override
153 remark #15305: vectorization support: vector length 4
154 remark #15309: vectorization support: normalized vectorization overhead 2.000
155 remark #15450: unmasked unaligned unit stride loads: 2
156 remark #15475: --- begin vector cost summary ---
157 remark #15476: scalar cost: 13
158 remark #15477: vector cost: 3.250
159 remark #15478: estimated potential speedup: 1.160
160 remark #15488: --- end vector cost summary ---
161 remark #25436: completely unrolled by 7
162 LOOP END
163
164 LOOP BEGIN at interpol_naive.c(72,8)
165 LOOP END
166
167 ...
168 ===========================================================================
```

A.2 Naive Interpolation with Loop-invariant Code Motion

```
1  LOOP BEGIN at interpol_hoist.c(50,3) inlined into interpol_hoist.c(96,3)
2  remark #25445: Loop Interchange not done due to: Data Dependencies
3  remark #25446: Dependencies found between following statements:  [From_Line# ->
   ↪  (Dependency Type) To_Line#]
4  remark #25447: Dependence found between following statements:  [55 -> (Output) 55]
5  remark #25447: Dependence found between following statements:  [57 -> (Output) 57]
6  remark #25447: Dependence found between following statements:  [61 -> (Anti) 57]
7  remark #25447: Dependence found between following statements:  [61 -> (Anti) 55]
8  remark #25451: Advice: Loop Interchange, if possible, might help loopnest. Suggested
   ↪  Permutation : ( 1 2 3 4 5 ) --> ( 2 1 3 5 4 )
9  remark #15542: loop was not vectorized: inner loop was already vectorized
10
11 LOOP BEGIN at interpol_hoist.c(51,5) inlined into interpol_hoist.c(96,3)
12 remark #15542: loop was not vectorized: inner loop was already vectorized
13
14 LOOP BEGIN at interpol_hoist.c(52,7) inlined into interpol_hoist.c(96,3)
15 remark #15542: loop was not vectorized: inner loop was already vectorized
```

```
16
17 LOOP BEGIN at interpol_hoist.c(53,4) inlined into interpol_hoist.c(96,3)
18 remark #15542: loop was not vectorized: inner loop was already vectorized
19
20 LOOP BEGIN at interpol_hoist.c(54,6) inlined into interpol_hoist.c(96,3)
21 remark #15542: loop was not vectorized: inner loop was already vectorized
22
23 LOOP BEGIN at interpol_hoist.c(56,8) inlined into interpol_hoist.c(96,3)
24 remark #15542: loop was not vectorized: inner loop was already vectorized
25
26 LOOP BEGIN at interpol_hoist.c(60,8) inlined into interpol_hoist.c(96,3)
27 remark #25085: Preprocess Loopnests: Moving Out Load and Store    [
   ↪  interpol_hoist.c(61,5) ]
28 remark #15389: vectorization support: reference A has unaligned access   [
   ↪  interpol_hoist.c(61,33) ]
29 remark #15389: vectorization support: reference u has unaligned access   [
   ↪  interpol_hoist.c(61,45) ]
30 remark #15381: vectorization support: unaligned access used inside loop body
31 remark #15305: vectorization support: vector length 2
32 remark #15399: vectorization support: unroll factor set to 3
33 remark #15309: vectorization support: normalized vectorization overhead 0.513
34 remark #15417: vectorization support: number of FP up converts: single precision to
   ↪  double precision 3   [ interpol_hoist.c(61,5) ]
35 remark #15418: vectorization support: number of FP down converts: double precision to
   ↪  single precision 1   [ interpol_hoist.c(61,5) ]
36 remark #15300: LOOP WAS VECTORIZED
37 remark #15450: unmasked unaligned unit stride loads: 2
38 remark #15475: --- begin vector cost summary ---
39 remark #15476: scalar cost: 15
40 remark #15477: vector cost: 6.500
41 remark #15478: estimated potential speedup: 1.410
42 remark #15487: type converts: 4
43 remark #15488: --- end vector cost summary ---
44 remark #25015: Estimate of max trip count of loop=1
45 LOOP END
46
47 LOOP BEGIN at interpol_hoist.c(60,8) inlined into interpol_hoist.c(96,3)
48 <Remainder loop for vectorization>
49 remark #25436: completely unrolled by 1
50 ...
```

RÉSUMÉ

La compilation traditionnelle est confrontée à de nombreux défis face aux besoins d'optimisations de programmes pour architectures parallèles. Un défi particulier est la conception de langages et représentations intermédiaires (RIs) appropriés. Bien que différentes RIs aient été proposés pour repousser les limites de la compilation traditionnelle, la plupart ne sont toujours pas adaptées pour appliquer des transformations de programmes pertinentes. Différentes alternatives sont donc de plus en plus exploitées, telles que l'*autotuning* ou la compilation interactive. Ces dernières nécessitent l'usage de langages intermédiaires fondamentalement différents, par exemple, les méta-langages pour la transformation de programmes. Dans cette thèse, centrée sur les besoins en applications numériques, nous étudions ce type de meta-langages; nous addressons particulièrement quatre questions: (i) Comment introduire une expressivité spécifique à un domaine? (ii) Comment repenser leur conception pour améliorer leur flexibilité dans la composition de transformations et la génération de plusieurs variantes de programmes? (iii) Jusqu'où pouvons-nous introduire du support pour le NUMA (Non-Uniform Memory Access)? (iv) En tant que nouvelle classe de méta-langages, comment formaliser leur sémantique? Nous répondons à ces questions au travers de la conception et sémantique de TeML, un méta-langage pour l'optimisation d'applications tensorielles.

MOTS CLÉS

compilation, optimization, transformation de programme, architectures parallèles, meta-langages, langages dédiés, sémantique

ABSTRACT

Traditional compilation faces numerous challenges with program optimizations for parallel architectures. A particular challenge is the design of proper intermediate languages and representations to enable the application of relevant optimization techniques. Various parallel intermediate representations and languages have been proposed. However, they still fall short of being adequate for nowadays applications and target architectures. To overcome this issue, different alternatives are more and more exploited such as empirical autotuning or interactive compilation. Such alternatives require fundamentally different types of intermediate languages such as transformation meta-languages. In this thesis, we study transformation meta-languages for numerical applications; we particularly address four questions: (i) How do we introduce domain-specific expressiveness? (ii) How do we rethink their design to enhance their flexibility in composing optimizations paths and generating multiple program variants? (iii) How far can we introduce NUMA (Non-Uniform Memory Access) awareness? (iv) As a new class of meta-languages, how do we formalize their semantics? We answer these questions through the design and semantics of TeML, a tensor optimizations meta-language.

KEYWORDS

compilation, optimization, program transformation, parallel architectures, meta-languages, domain-specific languages, semantics