Enabling Task Parallelism on Hardware/Software Layers using the Polyhedral Model

DISSERTATION

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Martin Kong, B.S., M.S.

Graduate Program in Computer Science and Engineering

The Ohio State University

2016

Dissertation Committee:

Louis-Noël Pouchet, Advisor

P. Sadayappan

Gagan Agrawal
ABSTRACT

Compute intensive applications running on clusters of shared-memory computers are typically implemented using OpenMP and MPI. Applications are difficult to program, debug and maintain. Furthermore, performance portability is usually limited. Several program transformations have be to be applied at multiple levels of the software and hardware stack to expose parallelism, choose the adequate granularity and finally map it onto the target system in order to obtain high-performing code.

This thesis focuses on different aspects of the task parallel data-flow model as well as the widespread data-parallel model. We address performance issues at each of the three main levels of the cluster: intra-core, inter-core and inter-node levels. At the core level, we effectively target a hardware layer, the SIMD-vector units available on modern processors, and attempt to maximize the vectorizability opportunities of the program by aggressively transforming it using a polyhedral compilation framework.

At the node level, we address the issues of barrier synchronization and load balancing by introducing a combined compiler and run-time approach that performs automatic coarsening on a sequential code, detecting the point-to-point synchronizations required to extract atomic tasks, replacing OpenMP barriers and by providing a specific code generation step that targets a lightweight data-flow run-time, OpenStream.
At the coarsest level we consider the performance and productivity aspects of writing distributed-memory programs. Programming models such as MPI come with an inherent trade-off between productivity and high-performance. We aim to reconcile these two aspects by proposing and implementing a new data-flow language and the associated optimizing compiler capable of generating an Intel CnC C++ program from a simple textual representation of a distributed algorithm. The language offers various constructs which allow to specify distinct task and data placements, task schedules, synchronizations and communication schemes with minimal modifications to a mint program.

Then, the lack of analytical models which can accurately predict the performance of programs executing over task-parallel run-times that can automatically avoid communication when the data is known to be local to the node is addressed for the CnC run-time. Models such as these allow systematic analytical autotuning and can potentially save valuable computing hours.

Our results on a variety of scientific codes demonstrate the viability of our approaches, combining high productivity, intuitive parallel algorithm specification and high-performance execution on shared-memory and distributed-memory environments.
This work is dedicated to my late father, Maynard Kong, my dear mother, Consuelo, and my dear siblings, Maynard, Margarita and Rosa, all of who taught me the meaning of love, hard work and responsibility.
ACKNOWLEDGMENTS

Following graduate studies, and particularly PhD studies is like entering a committed relationship with due date. Motivation, endurance, a strong support system, a good adviser, a lot of hard work and a little luck are the key to success. Naturally, in a 5 year period everyone goes through hardships. Therefore these words are dedicated to all the people that helped me get through the last 5 years and 3 months of my life.

First of all, I will always be grateful to my advisor, Louis-Noël, the first person that believed in me. His way of teaching and advising was a perfect match for my personality, character and style of working. You became more than an advisor, you became a friend. Through a number of difficult times you were very supportive and understanding, to the point of forcing me take time off in order to collect myself and put my priorities straight. Thank you Louis-Noël.

I also have to thank Albert and Saday. Sort of my genealogical grand-father and step-father. Saday, even 10 minutes of your time felt like a lifetime lesson. Albert, meeting you and working with you proved to be an invaluable experience. I loved the time in ENS Paris and thanks to that I made lifetime friends.

Many of these words have to go to the people with whom I shared my student life. Starting with Justin and his ”Don’t burn my GPUs or I’ll beat you to a pulp” speech; the long(er than I would like to admit) procrastination sessions with Kevin; the work and occasional ”I need your help” conversations with Kevin, Mahesh, Naser, Prashant
and Venmugil; and during my last year, welcoming and guiding the new guys to the
group, Israt, Changwan and Wenlei. I know the lab will be in good hands with you.

From people out of my lab I would like to thank Mehmet and Emre. Mehmet, I
will always remember sitting through 756, taking our "breaks" here and there, and
the time you convinced me to have an early coffee a few days after your defense. Yes,
you were sure to convince me, and you were out waiting. Emre, the funny guy, you
always had something entertaining to say. That, in this kind of life, is priceless.

A few longer words have to go to some labmates: Kevin, Venmugil and Naser.
Kevin, you were a great fellow, we made a great team. Your caffeine "OD" for that
particular deadline was epic, to the point of patching some other guy’s python library
at 3am and doing situps. Naser, "my one year older twin". Yes, we shared many
things. Venmugil, my last standing friend. Thank you for listening and I am sorry to
have stolen so much time from you.

I have many special thanks for my non-OSU friends, who I met during the second
half of my PhD. Brandon, my "geek" buddy; Matt and Chris, that Gremlins and
algebra discussion was epic. The list is too long, but thanks to everyone at the
O’place. It was only 3 years, but it felt like a lifetime. Tobi, Kasia, Amalia, my
Parisian friends, we shared only 3 months, but you were with me through my hardest
time. Infinite thanks for being there.

Danielle, you have been and are the most supportive person I have ever met. You
always knew what to say and when to say it. You put everything into perspective.
Thank you for all your help. A piece of this work goes to you.

Last but certainly not least. I want to thank my family. My dad Maynard and
my mom Consuelo. I never had to say anything, you just knew when and how to
encourage me and push me forward. Thank you. My sister Margarita, I always felt that we had some pending years to catch up, and what do you know, we now share alma mater. Sam and Steph, I think I have permanent nerve damage due to playing Mario Kart, but it still was fun. Maynard and Rosa, despite the distance you were always there for cheering me up. Thank you all.
VITA

December 4, 1981 ..........................Born: Barquisimeto, Lara, Venezuela

Jan 2003 — Nov 2003 ........................ Intern
Business Consulting and Technology Services

May 2004 ................................. B.S.C.E.
Pontificia Universidad Catolica del Peru

Jan 2004 — Mar 2005 ...................... Teaching Assistant
Pontificia Universidad Catolica del Peru

Mar 2005 — Jul 2010 ...................... Lecturer
Pontificia Universidad Catolica del Peru

Jan 2009 ................................. M.S.C.S
Pontificia Universidad Catolica del Peru

Aug 2011 — Present ...................... Graduate Research Associate
The Ohio State University

May 2013 — Aug 2013 ...................... Research Engineer
Ecole Normale Superieure/INRIA

PUBLICATIONS

Research Publications

P. Rawat, M. Kong, T. Henretty, J. Holewinski, K. Stock, L.N Pouchet, J. Ramanujam, A. Rountev, P. Sadayappan
SDSLc: A Multi-target Domain-specific Compiler for Stencil Computations
In Proceedings of the 5th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing, November 2015
M. Kong, L.N. Pouchet and P. Sadayappan
A Roofline-Based Performance Estimator for Distributed Matrix-Multiply on Intel CnC
In *International Workshop of Automatic Performance Tuning (iWAPT/IPDPS-W)*, May 2015

M. Kong, A. Pop, R. Govindarajan, L.N. Pouchet, A. Cohen and P. Sadayappan
Compiler/Run-Time Framework for Dynamic Data-Flow Parallelization of Tile Programs
In *ACM Transactions on Architecture and Code Optimization (TACO)*, Volume 11 Issue 4, January 2015

K. Stock, M. Kong, T. Grosser, L.N. Pouchet, F. Rastello, J. Ramanujam and P. Sadayappan
A Framework for Enhancing Data Reuse via Associative Reordering
In *ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, June 2014

M. Kong, R. Veras, K. Stock, L.N. Pouchet, F. Franchetti and P. Sadayappan
When Polyhedral Transformations Meet SIMD Code Generation
In *ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, June 2013

**FIELDS OF STUDY**

Major Field: Computer Engineering
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Dedication</td>
<td>iv</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>v</td>
</tr>
<tr>
<td>Vita</td>
<td>viii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xv</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xx</td>
</tr>
<tr>
<td>List of Algorithms</td>
<td>xxii</td>
</tr>
<tr>
<td>List of Listings</td>
<td>xxiv</td>
</tr>
<tr>
<td>Chapters:</td>
<td></td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2. Background</td>
<td>7</td>
</tr>
<tr>
<td>2.1 The Polyhedral Model</td>
<td>7</td>
</tr>
<tr>
<td>2.2 Task Parallel Languages and Run-Times</td>
<td>19</td>
</tr>
<tr>
<td>2.2.1 OpenStream</td>
<td>19</td>
</tr>
<tr>
<td>2.2.2 Concurrent Collections</td>
<td>22</td>
</tr>
<tr>
<td>3. Extracting Vector Codelets</td>
<td>26</td>
</tr>
<tr>
<td>3.1 Overview</td>
<td>28</td>
</tr>
</tbody>
</table>
**LIST OF FIGURES**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Examples of iteration domain: matrix (top) and ISL set (bottom)</td>
<td>10</td>
</tr>
<tr>
<td>2.2 Iteration domain for statement S in Gauss-Seidel 1D</td>
<td>10</td>
</tr>
<tr>
<td>2.3 PolyLib [86] representation of access functions for Gauss-Seidel 1D</td>
<td>11</td>
</tr>
<tr>
<td>2.4 Map representation of array references for Gauss-Seidel 1D</td>
<td>12</td>
</tr>
<tr>
<td>2.5 Dependence polyhedron from $A[i]$ to $A[i-1]$ for $t_{dst} &gt; t_{src}$</td>
<td>13</td>
</tr>
<tr>
<td>2.6 Read after write (RAW) dependences: all (top), last-write (bottom)</td>
<td>14</td>
</tr>
<tr>
<td>2.7 Gauss-Seidel 1D dependence relations</td>
<td>14</td>
</tr>
<tr>
<td>2.8 Iteration domain after transformation $S[t, i] \rightarrow [t, t + i]$ (top); flow dependences after transformation (bottom)</td>
<td>16</td>
</tr>
<tr>
<td>2.9 OpenStream example: gemm kernel tiled and parallelized with inter-band idiom</td>
<td>21</td>
</tr>
<tr>
<td>3.1 Example of stride-0/1 constraints</td>
<td>43</td>
</tr>
<tr>
<td>4.1 Illustration of benefit of task data-flow over static affine scheduling</td>
<td>61</td>
</tr>
<tr>
<td>4.2 Compiler stages from a sequential input code to a tile level parallel version with point-to-point synchronization</td>
<td>64</td>
</tr>
<tr>
<td>4.3 Jacobi-2d tile dependences</td>
<td>69</td>
</tr>
</tbody>
</table>
8.14 PIPES Jacobi 3D - 6 ghost zones, problem size = $4 \times 384^3$, single precision, InfiniBand network ........................................ 189

9.1 Cannon’s CnC step .................................................. 194

9.2 Johnson’s CnC block-multiply step ............................... 195

9.3 Johnson’s CnC reduce-step ........................................... 196

9.4 Cannon compute time for N=2000 ............................... 204

9.5 Cannon compute time for N=8000 ............................... 204

9.6 Johnson compute time for N=2000 ............................... 205

9.7 Johnson compute time for N=8000 ............................... 205

9.8 Cannon communication time for N=2000 ....................... 207

9.9 Cannon communication time for N=8000 ....................... 208

9.10 Johnson communication time for N=2000 ..................... 208

9.11 Johnson communication time for N=8000 ..................... 209

9.12 Cannon perfbound model for N=2000 ....................... 210

9.13 Cannon perfbound model for N=8000 ....................... 211

9.14 Johnson perfbound model for N=2000 ....................... 212

9.15 Johnson perfbound model for N=8000 ....................... 213

9.16 Cannon performance bound model for N=2000 assuming a 4x step kernel speedup ............................................... 214

9.17 Cannon performance bound model for N=8000 assuming a 4x step kernel speedup ............................................... 215

9.18 Johnson performance-bound model for N=2000 assuming a 4x step kernel speedup ............................................... 215
9.19 Johnson performance-bound model for N=8000 assuming a 4x step kernel speedup

216
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 ILP statistics</td>
<td>48</td>
</tr>
<tr>
<td>3.2 Time breakdown for kernels: All Tile Codelets (ATC), All Full Tiles (AFT=ATC+Peeling) and Full Program (FP=AFT+Partial Tiles)</td>
<td>50</td>
</tr>
<tr>
<td>3.3 Single precision performance data in GFLOP/s.</td>
<td>51</td>
</tr>
<tr>
<td>3.4 Double precision performance data in GFLOP/s.</td>
<td>51</td>
</tr>
<tr>
<td>4.1 Blur-Roberts kernel performance in GFLOPS/sec for AMD Opteron 6274 and Intel Xeon E5-2650, on 1, half and all cores.</td>
<td>59</td>
</tr>
<tr>
<td>4.2 Benchmark features (I:Inter-band, DW:Dynamic Wavefront, P:Partitioning; n,t: problem sizes)</td>
<td>89</td>
</tr>
<tr>
<td>4.3 Experimental testbed</td>
<td>90</td>
</tr>
<tr>
<td>4.4 Benchmark description</td>
<td>91</td>
</tr>
<tr>
<td>8.1 Experimental setup</td>
<td>169</td>
</tr>
<tr>
<td>8.2 Communication and computation bounds</td>
<td>176</td>
</tr>
<tr>
<td>9.1 Model input parameters</td>
<td>197</td>
</tr>
<tr>
<td>9.2 General performance derived metrics</td>
<td>198</td>
</tr>
<tr>
<td>9.3 Experimental setup</td>
<td>202</td>
</tr>
<tr>
<td>9.4 Software stack flags</td>
<td>203</td>
</tr>
</tbody>
</table>
9.5 Evaluated task mappings
# LIST OF ALGORITHMS

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 PartitionDomains (G, u, v)</td>
<td>74</td>
</tr>
<tr>
<td>4.2 ConstructTaskGraph (PRDG)</td>
<td>75</td>
</tr>
<tr>
<td>4.3 SelectTaskIdiom (G)</td>
<td>79</td>
</tr>
<tr>
<td>4.4 High-level algorithm for data-flow task-level parallelization (Part I)</td>
<td>87</td>
</tr>
<tr>
<td>6.1 Compute iteration domains</td>
<td>124</td>
</tr>
<tr>
<td>6.2 Compute dependence polyhedra</td>
<td>127</td>
</tr>
<tr>
<td>6.3 Task Coalescing Algorithm</td>
<td>131</td>
</tr>
<tr>
<td>6.4 Task Affinity</td>
<td>133</td>
</tr>
<tr>
<td>6.5 Block producer affinity</td>
<td>134</td>
</tr>
<tr>
<td>6.6 Block consumer(s) affinity</td>
<td>135</td>
</tr>
<tr>
<td>6.7 Block Maxlife</td>
<td>136</td>
</tr>
<tr>
<td>6.8 PIPES Autogen</td>
<td>137</td>
</tr>
<tr>
<td>7.1 ISL-SET construction from a PIPES entity</td>
<td>146</td>
</tr>
<tr>
<td>7.2 ISL-MAP construction from a PIPES relation</td>
<td>148</td>
</tr>
<tr>
<td>7.3 Sub-SCoP extraction from main SCoP</td>
<td>152</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>------------------------------------------------------------</td>
</tr>
<tr>
<td>7.4</td>
<td>Code generation for sub-scops of type entity task</td>
</tr>
<tr>
<td>7.5</td>
<td>Code generation for sub-scops of type entity block</td>
</tr>
<tr>
<td>7.6</td>
<td>Code generation for non-affine scops</td>
</tr>
<tr>
<td>7.7</td>
<td>PIPES full-program printing</td>
</tr>
<tr>
<td>7.8</td>
<td>Printing the program context</td>
</tr>
<tr>
<td>7.9</td>
<td>PIPES driver function</td>
</tr>
<tr>
<td>7.10</td>
<td>PIPES task body printing</td>
</tr>
<tr>
<td>7.11</td>
<td>PIPES task tuner printing</td>
</tr>
<tr>
<td>7.12</td>
<td>PIPES block tuner printing</td>
</tr>
</tbody>
</table>

xxiii
# LIST OF LISTINGS

<table>
<thead>
<tr>
<th>Listing</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Gauss-Seidel 1D kernel</td>
<td>9</td>
</tr>
<tr>
<td>3.1 After transfo. for codelet exposure</td>
<td>31</td>
</tr>
<tr>
<td>3.2 After unroll-and-jam $2 \times 1$</td>
<td>31</td>
</tr>
<tr>
<td>3.3 After abstract SIMD vectorization</td>
<td>32</td>
</tr>
<tr>
<td>4.1 Blur-Roberts kernel</td>
<td>59</td>
</tr>
<tr>
<td>4.2 Blur-Roberts Kernel produced with PLuTo minfuse</td>
<td>60</td>
</tr>
<tr>
<td>4.3 Blur-Roberts Kernel produced with PLuTo smartfuse</td>
<td>60</td>
</tr>
<tr>
<td>4.4 Jacobi-2d kernel</td>
<td>68</td>
</tr>
<tr>
<td>4.5 3mm kernel (with removed initializations)</td>
<td>77</td>
</tr>
<tr>
<td>4.6 Jacobi-2d code snippet for partition with signature 6 (see Fig. 4.8)</td>
<td>85</td>
</tr>
<tr>
<td>5.1 Matrix Multiplication</td>
<td>113</td>
</tr>
<tr>
<td>5.2 A simple flat 2D topology</td>
<td>116</td>
</tr>
<tr>
<td>5.3 Mapping task instances to processors</td>
<td>117</td>
</tr>
<tr>
<td>5.4 Communication of data</td>
<td>118</td>
</tr>
<tr>
<td>5.5 Task-to-task dependence</td>
<td>119</td>
</tr>
<tr>
<td>5.6 Scanning tag sets in specific order</td>
<td>120</td>
</tr>
</tbody>
</table>
CHAPTER 1

Introduction

Compute intensive applications running on clusters of shared-memory computers are typically implemented using OpenMP [34] for shared-memory single-node parallelism, MPI [107, 7, 57, 41] for inter-node communication and process orchestration as well as low level fine-tuning of the core computations that allow to exploit the potential of the memory hierarchy and the short-vector SIMD execution units present in modern processors.

Performance portability as well as user productivity are two of the main concerns of modern compilation frameworks as severe code reorganization and restructuring are usually necessary to adapt and map applications to resources of the computer cluster.

The problems that we address in this work run vertically through the software and hardware stack of a computer cluster.

• Single node optimizations: Data locality and parallelism are critical optimization objectives for performance on modern multi-core machines. Both coarse-grain parallelism (e.g., multi-core) and fine-grain parallelism (e.g., vector SIMD)
must be effectively exploited, but despite decades of progress at both ends, current compiler optimization schemes that attempt to address data locality and both kinds of parallelism often fail at one of the three objectives.

- Data-flow and task-parallel run-times and languages are increasingly popular. Many of them provide expressive mechanisms for inter-task synchronization. For example, OpenMP 4.0 integrates data-driven execution semantics derived from the StarSs research language. Compared to the more restrictive data-parallel and fork-join concurrency models, the advanced features being introduced into task-parallel models in turn enable improved scalability through load balancing, memory latency hiding, mitigation of the pressure on memory bandwidth, and as a side effect, reduced power consumption.

- Enhancing productivity and algorithmic design in distributed-memory environments: Productivity can be vastly improved using approaches such as the Concurrent Collection framework, where the user expresses the data and control-flow relations between tasks, offering the runtime maximal freedom to place and schedule tasks. While productivity is increased, it remains a challenge to achieve high-performance execution: the implementation of parallel algorithms goes beyond the expression of inter-task parallelism and typically combines task placement and communication strategies to form an executable specification of the program on a target machine. Implementing a parallel algorithm that specifies certain placement and scheduling decisions remains difficult in CnC frameworks: one typically needs to use runtime control primitives (e.g., tuners in Intel CnC) along with a specific refactoring of the program description.
Contributions  In this work we address the aforementioned problems on three distinct levels: within a single processing core, parallelism among cores of a single node, and parallelism among different nodes in a computer cluster.

- At the finest level, we address this problem by proposing a 3-step framework, which aims for integrated data locality and SIMD execution of programs. We define the concept of vectorizable codelets, with properties tailored to achieve effective SIMD code generation for the codelets. We leverage the power of a modern high-level transformation framework to restructure a program to expose good ISA-independent vectorizable codelets, exploiting multi-dimensional data reuse. Then, we generate ISA-specific customized code for the codelets, using a collection of lower-level SIMD-focused optimizations.

- At the intermediate level, we develop a systematic approach to compile loop nests into concurrent, dynamically constructed graphs of dependent tasks. We propose a simple and effective heuristic that selects the most profitable parallelization idiom for every dependence type and communication pattern. This heuristic enables the extraction of inter-band parallelism (cross barrier parallelism) in a number of numerical computations that range from linear algebra to structured grids and image processing.

The proposed static analysis and code generation alleviates the burden of a full-blown dependence resolver to track the readiness of tasks at run time. We evaluate our approach and algorithms in the PPCG compiler, targeting OpenStream, a data-flow task-parallel language with explicit inter-task dependences.
and a lightweight runtime. Experimental results demonstrate the effectiveness of the approach.

• At the coarsest level, that is inter-node execution, we introduce a new data-flow language and the associated compiler, PIPES. The input program is a textual representation of the producer and consumer relations among data blocks and tasks, enhanced with several constructs that allow to express task affinities, data placement and scheduling of tasks and communication, thereby allowing to conduct extensive space exploration by generating various program variants. In particular, they permit to generate a class of 2.5d algorithms by performing minor modifications to the corresponding 3d variant of the algorithm. On the compiler side we leverage several recent advances in polyhedral frameworks such as the Integer Set Library [123], tilability of imperfectly nested affine loops [22, 24], code generation [14, 15] and counting integer points in parametric polytopes [125] to perform dependence analysis on the input program, extract an intermediate representation amenable to polyhedral compilation, and to compute and apply program transformations (e.g., task coarsening). The output is a C++ program that targets the Intel Concurrent Collections run-time.

• Our last contribution targets also the coarsest level. We leverage a task parallel run-time with distributed execution capabilities, Intel CnC. We model two classical distributed-memory matrix multiply algorithms, Cannon and Johnson, which encompass the two ends of the trade-off between parallelism and memory replication. Our proposed analytical models allow to estimate the execution time of both algorithms implemented as Concurrent Collection programs by
determining the data volume that is accessed both locally (i.e., resident on the current node) and externally (i.e., on a node different from the one doing the request).

**Thesis Structure**  The rest of this thesis captures the above contributions, tackling the identified issues at each level of the software and hardware stack. We first review necessary background material in Chapter 2, presenting the concepts relating to Polyhedral compilation frameworks, its abstractions as well as recalling topics such as vectorization and data-flow run-times. Next, Chapter 3 further motivates the problem of reconciling outer and inner parallelism while simultaneously exploiting locality. We design an Integer Linear Program that allows to extract maximal vectorizable codelets. For the second part of this work we move up the software stack. Chapter 4 discusses the relevance of generating point-to-point synchronizations for a lightweight streaming data-flow run-time. We propose polyhedral compiler techniques that enable the extraction of atomic execution units from an input sequential program, avoiding the usage of OpenMP barriers. The third part of this thesis is decomposed into chapters 5-8. We introduce a novel language geared towards the specification of distributed-memory algorithms containing various constructs that permit the user to perform advanced operations such as scheduling communication and allowing to specify various placement strategies for tasks and blocks. Chapter 6 presents the algorithms used to perform the complex analyses and transformations over the input program. Details of the compiler infrastructure built are then given in Chapter 7, discussing the main components of the compiler and delving into specifics of code
generation and overall program construction. Following this, our approach for generating and optimizing distributed-memory programs is validated in Chapter 8. This study is then complemented by proposing analytical performance models specific to a class of distributed linear algebra algorithms in Chapter 9. To conclude this work, Chapter 10 and Chapter 11 discuss future research directions and conclusions.
CHAPTER 2

Background

In this chapter we briefly recapture the background concepts relating to the Polyhedral Compilation Model as well as the two task-parallel run-times used throughout this work, OpenStream and Concurrent Collections (CnC). This material will be used as the building blocks in the following chapters.

Definition 1 (Polyhedron). \textit{Is the intersection of finitely many half-spaces. A polytope is a bounded polyhedron. A polyhedron can be formally defined as the set of solutions to a system of linear inequalities:}

\[ M\bar{x} \leq \bar{b} \]

where \( M \in \mathbb{Z}^{k \times d} \) is an integer matrix with \( k \) rows and \( d \) columns, \( \bar{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_d \end{bmatrix} \) and \( \bar{b} \in \mathbb{Z}^k \).

2.1 The Polyhedral Model

The polyhedral model [44, 45, 46, 47] is a powerful compilation framework that allows analyzing, reasoning and transforming programs using mathematical representations. In a nutshell, it represents loops, array references and program dependences
as polyhedra. More specifically, this model brings together three areas: i) compilers and programming languages; ii) linear algebra and iii) integer linear programming (ILP).

Programs amenable to this model are commonly called static control parts (SCoP) [47, 49] or affine programs. The polyhedral model excels in the three standard compilation stages: analysis, transformation and code generation. Analyses can be performed at the granularity of an array cell; transformation schedules respecting the program dependences can be computed for each individual program statement instance, while the code generation stage is responsible of applying the selected transformations and producing an equivalent program. Despite these advantages it also suffers from two strong restrictions: a) it uses computationally expensive algorithms such as ILP; and b) loop bounds and array references can only consist of affine expressions which are functions of outer loop iterators and scope symbolic constants, e.g. a constant N, also known as SCoP parameters.

A SCoP narrows the applicability of the model by removing data dependent control flow, i.e., a condition of the form \( \text{if } (a[i] > 0) \). The ground reason for this is the dependence analysis stage: producer/consumer instances must be computable at compilation time. This enables the compiler to determine the exact array address at which a dependence arises, as well as the loop iterator coordinates that represent the source and destination of the dependence. Therefore, pointers, array indirections (e.g. \( A[B[i]] \)) and function calls with side-effects are also forbidden. On the other hand, condition b) has the purpose of modeling program entities such as loop nests,
array references and program dependences as polyhedra. Most of the program analyses and transformations performed within the model are computed by solving integer or rational linear programming problems.

Unlike the abstract syntax trees used as internal representation in traditional compilers, polyhedral compiler frameworks internally represent imperfectly nested loops and their data dependence information as a collection of parametric polyhedra.

Programs in the polyhedral model are represented using four mathematical structures: each statement has an iteration domain, each memory reference is described by an affine access function/relation, data dependences are represented using dependence relations/polyhedra, and finally the program transformation to be applied is represented using a scheduling function or a schedule map. In the following section we will introduce the program abstractions using the two most common representations in polyhedral compilation frameworks: the earlier matrix-based and the recently reintroduced set/map notation. The former representation is broadly used in works such as [24, 80, 99, 101, 102, 120], whereas the latter was used by Pugh [104, 103] as well as in the Integer Set Library (ISL) [123].

In the following sections we will review the core polyhedral abstractions by using Listing 2.1 as a running example and showing the PolyLib [87] and ISL notations for the abstractions. A more detailed explanation of the interpretation of the Polylib format can be found in [86].

Listing 2.1: Gauss-Seidel 1D kernel

```c
for (t = 1; t <= 4; t++)
    for (i = 1; i <= 4; i++)
```
Iteration Domains  The first program abstraction to review is the iteration domain of a program statement. Its role is to identify each of the execution instances of a given statement $S$ with a $d$-dimensional tuple, where $d$ is the depth of the loop nest and the tuple is conformed of the loop iterators surrounding the statement $S$. For instance, in Listing 2.1 the run-time instances of statement $S$ are identified by the iterator vector $\vec{x} = (t, i)$. More specifically, the polyhedron that defines the iteration domain of statement $S$ can be written as:

$$
\begin{bmatrix}
t & i & 1 \\
1 & 0 & -1 \\
-1 & 0 & 4 \\
0 & 1 & -1 \\
0 & -1 & 4 \\
\end{bmatrix} 
\begin{align*}
t & \geq 1 \\
4 & \geq t \\
i & \geq 1 \\
4 & \geq i \\
\end{align*}
\{S[t, i] \in \mathbb{Z}^2 : 1 \leq t \leq 4 \wedge 1 \leq i \leq 4\}
$$

**Figure 2.1:** Examples of iteration domain: matrix (top) and ISL set (bottom)

**Figure 2.2:** Iteration domain for statement $S$ in Gauss-Seidel 1D
The iteration domain of $S$ consists of four hyperplanes, each modeled by an inequality. The four inequalities shown in Figure 2.1 restrict the values of iterator $t$ to the integer range $1 \ldots 4$ and the values of $i$ to $1 \ldots 4$. Figure 2.2 shows a graphical representation of this abstraction for the same example with $t$ on the x-axis and $i$ as the y-axis.

Access Functions/Relations Each array reference in a SCoP has an access function/relation associated to it. The role of the access function is to map points in the iteration domain of a statement $S$ to the specific memory locations of an array (e.g., $A[i-1]$) that are accessed during the statement execution. For the given example the array reference $A[i-1]$ accesses $A[0]$ for all $(t, 1)$ and $A[1]$ for all $(t, 2)$.

The array references of a program statement are typically modeled in polyhedral frameworks in one of two ways, as matrices that map an input iterator vector $\vec{x}$ to the data space of an array or as an explicit map from the space of the statement domain to the (named) space of an array, i.e. from $S$ to $A$. Examples of both representations are shown in Figures 2.3-2.4.

$$F_{A_0}^S(\vec{x}) = \begin{bmatrix} \text{ArrayID} & t & i & 1 \\ 0 & 0 & 1 & -1 \end{bmatrix} = i - 1$$

$$F_{A_1}^S(\vec{x}) = \begin{bmatrix} \text{ArrayID} & t & i & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} = i$$

$$F_{A_2}^S(\vec{x}) = \begin{bmatrix} \text{ArrayID} & t & i & 1 \\ 2 & 0 & 1 & 1 \end{bmatrix} = i + 1$$

Figure 2.3: PolyLib [86] representation of access functions for Gauss-Seidel 1D
\{S[t, i] \rightarrow A[i] : 1 \leq t \leq 4 \land 1 \leq i \leq 4\}
\{S[t, i] \rightarrow A[i - 1] : 1 \leq t \leq 4 \land 1 \leq i \leq 4\}
\{S[t, i] \rightarrow A[i + 1] : 1 \leq t \leq 4 \land 1 \leq i \leq 4\}

**Figure 2.4:** Map representation of array references for Gauss-Seidel 1D

**Dependence Polyhedra/Relations** The third abstraction we review are dependence polyhedra. The four types of data dependences meeting the Bernstein Conditions [21] can be modeled as convex polyhedra when the iteration domains and access functions are affine. One polyhedron is constructed for each pair of array references, possibly from the same syntactic statement. Then, three conditions are embedded into the polyhedron: i) the iteration domain of each statement; ii) an equality condition for the memory location; and iii) the precedence condition which states that the destination of the dependence must be executed after the source. Figure 2.5 shows a dependence polyhedron for the Gauss-Seidel 1D kernel. We model a flow dependence from the array reference \(A[i]\) being written to the read reference \(A[i-1]\). The upper part of the matrix models the iteration domain of both statements, which in this case are the same. The middle part models the condition that both array references should access the same memory location. Finally, the bottom part of the matrix models the precedence condition write \(A[i]\) followed by read \(A[i-1]\). We note that in this example there are two possible precedence conditions: \((t_{dst}, i_{dst}) - (t_{src}, i_{src}) \geq (1, 0)\) and \((t_{dst}, i_{dst}) - (t_{src}, i_{src}) \geq (0, 1)\), i.e., that the dependence exists for the same iteration of \(t_{dst} = t_{src}\) or for \(t_{dst} \geq t_{src}\). In this example we model only the latter, but a similar
polyhedron with the first precedence condition should be also constructed to account for all possible dependences. More details can be found in [15, 17].

\[
\begin{align*}
D_{S,S} = \\
\begin{bmatrix}
\ge / = & t_{dst} & i_{dst} & t_{src} & i_{src} & 1 \\
1 & 1 & 0 & 0 & 0 & -1 \\
1 & -1 & 0 & 0 & 0 & 4 \\
1 & 0 & 1 & 0 & 0 & -1 \\
1 & 0 & -1 & 0 & 0 & 4 \\
1 & 0 & 0 & 1 & 0 & -1 \\
1 & 0 & 0 & -1 & 0 & 4 \\
1 & 0 & 0 & 0 & 1 & -1 \\
1 & 0 & 0 & 0 & -1 & 4 \\
0 & 0 & 1 & 0 & -1 & 1 \\
1 & 1 & 0 & -1 & 0 & -1
\end{bmatrix}
\end{align*}
\]

**Figure 2.5:** Dependence polyhedron from $A[i]$ to $A[i-1]$ for $t_{dst} > t_{src}$

To complete the explanation of dependence polyhedra we show in Figure 2.6 the flow dependences of the program for the precedence condition $t_{dst} > t_{src}$. The top half shows all the instances of the flow dependence type, whereas the bottom half shows only the actual source of the dependence, the writes immediately preceding the read references [45] commonly known as **last write**. Finally, all dependence relations, this time in the more compact ISL format using maps, are shown in Figure 2.7. Here, the identifier preceding the input and output tuples are the namespaces used, each referring to the iteration domain of the source and target of the dependence, respectively.
Figure 2.6: Read after write (RAW) dependences: all (top), last-write (bottom)

\[
S[t_0, i_0] \rightarrow S[t_1, i_1]: t_1 \geq t_0 \land i_0 - 1 = i_1 \\
S[t_0, i_0] \rightarrow S[t_1, i_1]: t_1 \geq t_0 \land i_0 = i_1 \\
S[t_0, i_0] \rightarrow S[t_1, i_1]: t_1 \geq t_0 \land i_0 + 1 = i_1
\]

Figure 2.7: Gauss-Seidel 1D dependence relations

Schedules The power of polyhedral compilation frameworks resides in the ability to compute the loop transformation to be applied to each program statement instance. These transformations are commonly termed schedules. A single schedule
can comprise various loop transformations such as loop distribution or loop fusion, loop shifting (also known as retiming [37, 36]), loop skewing. A vast body of work has been done in this area with various distinct objectives such as maximizing or exposing parallelism [38, 37, 85, 36, 46, 47, 80], maximizing data locality and loop fusion/distribution [67, 130, 129, 132, 71, 35, 100, 99, 24, 113, 58, 23, 59] or program restructuring for execution on hardware accelerators and usage of short-vector SIMD units [55, 113, 78, 54, 63, 79].

These reordering transformations conceptually assign a logical (multi-)dimensional time-stamp (a logical execution date) to each statement instance which is later used by a polyhedral code generator [13, 14, 121, 120] to scan (or visit) each point in the iteration domain in the order specified by the schedule. In other words, points in the iteration domain of a statement $S$ will be executed in the order specified by this schedule (e.g. if $\theta^S(0,1) = (2,0)$ and $\theta^S(1,0) = (1,2)$, then $S[1,0]$ executes before $S[0,1]$).

Continuing with our running example, the relation $S[t,i] \rightarrow [t,t + i]$ (equivalently, $\theta^S(\vec{x}) = (t,t+i)$) applies a **skewing** transformation on dimension-i w.r.t to dimension-t. Figure 2.8 shows how the iteration domain $S$ of Gauss-Seidel 1D’s kernel changes after applying the transformation (top) and how the dependences are equally skewed with the given program schedule.

**Definition 2** (Affine multi-dimensional schedule). *Given a statement $R$, an affine schedule $\Theta^R$ of dimension $m$ is an affine form of the $d$ loop iterators (denoted $\vec{x}_R$) and the $p$ global parameters (denoted $\vec{n}$). $\Theta^R \in \mathbb{Z}^{m \times (d+p+1)}$ can be written as:*

$$
\Theta^S(\vec{x}_R) = \begin{pmatrix}
\theta_{1,1} & \cdots & \theta_{1,d+p+1} \\
\vdots & \ddots & \vdots \\
\theta_{m,1} & \cdots & \theta_{m,d+p+1}
\end{pmatrix} \cdot \begin{pmatrix}
\vec{x}_R \\
\vec{n} \\
1
\end{pmatrix}
$$
The scheduling function $\Theta^R$ maps each point in $D_R$ to a multidimensional time-stamp (a vector) of dimension $m$. In the transformed code, the instances of $R$ defined in $D_R$ will be executed in lexicographic order of their associated time-stamp. Multi-dimensional timestamps can be seen as logical clocks: the first dimension similar to days (most significant), the next one to hours (less significant), etc.
The optimization of a full program requires a collection of affine schedules, one for each syntactic program statement. Even seemingly non-linear transformations like loop tiling (also known as blocking) and unrolling can be modeled [49].

**Tilability of imperfectly nested loop nests: the Tiling Hyperplane method**

Bondhugula et al. [24] proposed a general algorithm to find linearly independent hyper-planes which allow the tilability of imperfectly nested affine programs. The algorithm proceeds in a level-by-level fashion, and minimizes the dependence distance between points by using a non-linear objective function, which after bounding with the set of parameter vectors, can be linearized by using the Farkas Lemma [47]. The algorithm finds one solution for each schedule dimension of each statement, and guarantees the linear independence of the results by extending the ILP system with orthogonal sub-space constraints, thereby exposing permutable dimensions whenever feasible.

**Bands of Tiles** When the computed schedule represents a tiling transformation, a band is a consecutive set of non-constant dimensions (e.g., loop levels) in time space with the property that these dimensions are permutable. A band of tiles is a band that only considers some consecutive tile dimensions. Intuitively, one can think of these dimensions as the ones that will become the tile loops after code generation. For instance, in Fig. 4.2, only the tile loops are depicted. These correspond to the dimensions of the bands of tiles and would be generated by a schedule of the form:

\[
\{ S[i,j] \to [0, ti, tj, i^2, j^2] : 1 \leq i, j \leq N - 1 \ldots \}; \quad S2[i,j] \to [1, ti, tj, i^2, j^2] : 1 \leq i, j \leq N - 1 \ldots \}
\]
The leading constant value of 0 and 1 in the range of the relations for statements $S$ and $S2$, respectively, indicate that it is a scalar dimension (e.g., not a loop) and that the generated code will consist of 2 bands of tiles, composed in both cases by the trailing tile loop dimensions $ti$ and $tj$.

**Polyhedral Reduced Dependence Graph (PRDG)** It is a multigraph where each node represents a program statement [38]. Edges in between nodes capture different dependences. Nodes are labeled with iteration domains and edges are labeled with dependence relations.

**Fusion Heuristics: minfuse and smartfuse** The loop structure produced by a transformation can be viewed as a partition of the program statements under the criterion of appearing fused in one or more outer loops. *smartfuse* is the default PluTo heuristic for tiling where statements that do not share any data reuse are placed on different partitions. *minfuse* attempts to maximize the number of partitions by setting statements into different classes, unless they are required to appear fused under some loop level (or same strongly connected component) [24].

**Macro-statements** After applying a tiling algorithm such as PLuTo to an input program, statements are naturally fused under a sequence of loops. All statements that live under a common set of loops form a *macro statement*. The set of macro statements of a tiled program depends on the tiling heuristic used (e.g., smartfuse or minfuse).
2.2 Task Parallel Languages and Run-Times

2.2.1 OpenStream

The principal motivation for research in data-flow parallelism comes from the inability of the Von Neumann architecture to exploit large amounts of parallelism, and to do so efficiently in terms of hardware complexity and power consumption. In data-flow languages and architectures, the execution is explicitly driven by data dependences rather than control flow [68]. Data-flow languages offer functional and parallel composition preserving (functional) determinism. So-called threaded or task-parallel data-flow models operate on atomic sequences of instructions, or tasks, whereas early data-flow architectures leveraged data-driven execution at the granularity of a single instruction.

We selected the task-parallel data-flow language OpenStream [96], a representative of the family of low-level (C language) task-parallel programming models with point-to-point inter-task dependences [93, 26, 29]. OpenStream stands out for its ability to materialize inter-task dependences explicitly using streams, whereas the majority of the task-parallel languages rely on some form of implicit dependence representation (e.g., induced from memory regions in StarSs or OpenMP 4 [93]). The choice of an explicit dependence representation reduces the overhead of dynamically resolving dependences. For a detailed presentation, one may refer to Pop et al. [96], and to the formal model underlying the operational semantics of OpenStream [95].

Like the majority of task-parallel languages, an OpenStream program is a dynamically built task graph. Unlike the majority of streaming languages, OpenStream

\footnotesize
1The source code repository and web site for OpenStream can be found at http://www.openstream.info.
task graphs do not need to be regular or static. They can be composed of a dynamically evolving set of tasks communicating through data-flow streams. The latter are strongly typed, first class values: they can be freely combined with recursive computations and stored in dynamic data structures. Programming abstractions and patterns allow to construct complex, fully dynamic, possibly nested task graphs with unbounded fan-in and fan-out communications. OpenStream also provides syntactic support for common operations such as broadcasts.

Programmer annotations are used to specify regions, within the control flow of a sequential program, that may be spawned as concurrent coroutines and delivered to a runtime execution environment. These control flow regions inherit the OpenMP task syntax. Without input/output stream annotations, OpenStream tasks have the same semantics as OpenMP tasks. For example, Figure 2.9 shows the OpenStream version of the first loop nest in the Blur-Roberts edge detection kernel. Each instance of the outer loop outputs a token to stream $s$. The second task, following the first loop nest, waits for $N-2$ tokens on stream $s$, and then proceeds. This pattern implements data-flow concurrency in OpenStream and avoids barrier synchronization.

While OpenStream is very expressive and can be used to express non-deterministic computations, the language comes with specific conditions under which the functional determinism of Kahn networks [69] is guaranteed by construction. These conditions enforce a precise interleaving of data in streams derived from the control flow of the program responsible for spawning tasks dynamically, hereafter called the control program. In the following, we will assume the control program is sequential, which is a sufficient condition to enforce determinism.
long band_stream_ii_size = (floor((15 + ni)/16));
int band_stream_ii[band_stream_ii_size]
  __attribute__((stream));
int read_window[W]; int write_window[W];
for (int ii = 0; ii <= floord(ni - 1, 16); ii++)
#pragma omp task
  output (band_stream_ii[ii] << write_window[W])
{
  for (int jj=0; jj <= floord(nj-1,16); jj++)
    for (int i=16*ii; i <= min(ni-1,16*ii+15); i++)
      for (int j=16*jj; j <= min(nj-1,16*jj+15); j++)
        C[i][j] *= beta;
}
for (int ii = 0; ii <= floord(ni - 1, 16); ii++)
#pragma omp task
  input (band_stream_ii[ii] >> read_window[W])
{
  for (int jj=0; jj <= floord(nj-1,16); jj++)
    for (int kk=0; kk <= floord(nk-1,16); kk++)
      for (int i=16*ii; i <= min(ni-1,16*ii+15); i++)
        for (int j=16*jj; j <= min(nj-1,16*jj+15); j++)
          for (int k=16*kk; k <= min(nk-1,16 *kk+15); k++)
            C[i][j] += ((alpha * A[i][k]) * B[k][j]);

Figure 2.9: OpenStream example: gemm kernel tiled and parallelized with inter-band idiom

OpenStream Task Idioms  In this work we leverage multiple programming patterns present in modern task-parallel languages.

1. Input and output clauses: they extend the OpenMP task syntax and allow to explicitly specify the point-to-point synchronizations between tasks via streams. By default, all streams are operated through a window of stream elements accessible to a given task. The horizon of the window is the number of stream
elements accessible through it. The burst size of a window corresponds to the number of elements read/written from/to input/output streams at a given task activation. The default burst size is 1. The input clause can be further specialized into the two following clauses.

2. Peek operation: similar to the input clause, but it does not advance the stream’s window, i.e., the window’s burst size is zero. It enables the reuse of stream elements across multiple task activations, which is particularly useful when implementing broadcast operations.

3. Tick operation: a collection of peek operations may be followed by a tick operation, to advance the window to new stream elements.

4. Soft-barrier, point-to-point barrier or data-flow barrier: it is an (empty) program statement which waits for a (fixed or parametric) number of elements from one or more streams. Once all its (input) dependences are satisfied, an element is written to each of its output streams. Then, the tasks waiting for these elements may read them with peek operations.

2.2.2 Concurrent Collections

Concurrent Collections (CnC) [26, 27, 73, 31, 72, 66, 109, 116] is a programming model that builds upon TStreams [74]. Parallelism in this model is implicit. A set of computing steps are specified and related via control and data dependences, which in turn provide the only semantic constraints.

An important feature of this programming model is that the input program must be written in **dynamic single assignment** (DSA) form [118, 119]. This property
allows for the model to be deterministic [26]. In a nutshell, the DSA form forbids the use of destructive writes (e.g. updates or memory overwrites). In other words, any data item can be defined at most once, making the producer of this data unique.

Another relevant feature of this model is the separation of concerns philosophy, which effectively divides the program specification into two parts: the domain and the tuning. The domain of the program consists of the set of computing steps and the basic relations, whereas the tuning consists of adapting and mapping the program entities to an execution platform (e.g. a concrete run-time implementation executing over a specific hardware), leveraging all the available parallelism.

Concurrent Collections has been implemented on several high-level languages, namely Java, C++, Haskell and .Net, and with several underlying communication layers (e.g. MPI, shared-memory, UPC). In this work we leverage a specific Concurrent Collections implementation, Intel CnC C++.

Program entities in CnC are grouped into five different types: tag collections for control; item collections representing data; step collections representing operations over the items; the environment or master step; and the the context object, which establishes the relations among the entities. [109].

**Tag Collections** allow to identify individual instances of an object in a CnC program. For instance \([C : i]\) denotes the \(i^{th}\) dynamic instance of the object \(C\).

**Item Collections** represent the sets of data elements manipulated by a CnC program. A CnC program can produce and consume several item collections. In pure CnC, *programs implement the dynamic single assignment rule* for item collections,
that is each tag value in a collection (e.g., \([A : i \ 1]\)) can be written only once, and read multiple times.

**Step Collections** are the compute instances. They can consume and produce both items and tags. If a program has static control flow then all tags can be produced offline before starting the CnC graph execution. Dependences between step instances can be modeled using items and tags.

**The Environment** is the master step of the program and is responsible for producing the initial data, consuming the final data as well as spawning one or more steps.

**CnC graph** describes the relation between all the 3 types of entities above in a CnC program. A graph specifies the data that a computation consumes or produces, as well as the relations identifying related instances in the different collections.

Concurrent Collections provides two basic operations for steps, **Put** and **Get**. The semantics of these depend upon the type of the collection to which they are applied, that is an item or a tag collection.

**Data transfers** CnC provides convenient accessors methods, **Get** and **Put**, which allow to receive and send data blocks (items). Each invocation of any of these methods should be accompanied by a **tag** instance that identifies the item. The DSA form is enforced by invoking a **Put** method with distinct tag values.

**Step prescription** The prescription operation of steps is the insertion of tag instances associated to a step collection into the run-time by using the **Put** method.
Once the tag of a step (instance) is written, the computing step can start its execution when all its data dependences are satisfied.

**Shared and distributed memory**  CnC implements a separation of concern between the *description* of the application as sets of tasks, data items, and producer/consumer relations between them; and *tuning annotations* that are specific to the execution context to enable higher performance for instance via explicit placement of tasks/data [109]. That is, from a high-level programmer’s point of view, only Gets and Puts are used, irrespective of the executing environment: the same program can execute on a single processor, on a single node or on multiple nodes.

**CnC performance**  Previous work showed the effectiveness of CnC in delivering high performance on distributed environments on various applications including Cholesky decomposition and unbalanced tree search [109], on dense linear algebra kernels (e.g., Asynchronous Parallel Cholesky Factorization and Generalized Symmetric Eigensolver) outperforming ScaLAPACK+MPICH2/nemesis, multi-threaded MKL and equaling PLASMA+MKL [31, 116], while incurring in a low overhead w.r.t. Intel’s TBB in single node execution [30]. This, in tandem with the CnC program semantics and its separation of concerns provide the ideal setup for high-performance tuning and space exploration.
CHAPTER 3

Extracting Vector Codelets

The increasing width of vector-SIMD instruction sets (e.g., 128 bits in SSE, to 256 bits in AVX, to 512 bits in LRBni) accentuates the importance of effective SIMD vectorization. However, despite the significant advances in compiler algorithms [70, 131, 42, 90, 82, 91] over the last decade, the performance of code vectorized by current compilers is often far below a processor’s peak.

A combination of factors must be addressed to achieve very high performance with multi-core vector-SIMD architectures: (1) effective reuse of data from cache — the aggregate bandwidth to main memory on multi-core processors in words/second is far lower than the cumulative peak flop/second; (2) exploitation of SIMD parallelism on contiguously located data; and (3) minimization of load, store and shuffle operations per vector arithmetic operation.

While hand tuned library kernels such as GotoBLAS address all the above factors to achieve over 95% of machine peak for specific computations, no vectorizing compiler today comes anywhere close. Recent advances in polyhedral compiler optimization [24, 11] have resulted in effective approaches to tiling for cache locality, even for imperfectly nested loops. However, while significant performance improvement over untiled code has been demonstrated, the absolute achieved performance
is still very far from machine peak. A significant challenge arises from the fact that polyhedral compiler transformations to produce tiled code generally require auxiliary transformations like loop skewing, causing much more complex array indexing and loop bound expressions than the original code. The resulting complex code structure often leads to ineffective vectorization by even sophisticated vectorizing compilers such as Intel ICC. Further, locality enhancing transformations, e.g. loop fusion, can result in dependences in the innermost loops, inhibiting vectorization.

In this chapter, we present a novel multi-stage approach to overcome the above challenges to effective vectorization of imperfectly nested loops. First, data locality at the cache level is addressed by generating tiled code that operates on data footprints smaller than L1 cache. Code within L1-resident tiles is then analyzed to find affine transformations that maximize stride-0 and stride-1 references in parallel loops as well as minimization of unaligned loads and stores. This results in the decomposition of a L1-resident tile into codelets. Finally, a specialized back-end codelet optimizer addresses optimization of register load/store/shuffle operations, maximization of aligned versus unaligned load/stores, as well as register level reuse. Target-specific intrinsics code is generated for the codelet by the back-end optimizer.

This work in this chapter makes the following contributions. (1) It presents a novel formalization for an affine loop transformation algorithm that is driven by the characteristics of codelets. (2) Unlike most previous approaches that focus on a single loop of a loop-nest for vectorization, it develops an approach that performs integrated analysis over a multi-dimensional iteration space for optimizing the vector-SIMD code. (3) It represents the first demonstration of how high-level polyhedral transformation
technology can be effectively integrated with back-end codelet optimization technology through a model-driven approach, with significant performance improvement over production and research compilers.

This chapter is organized as follows. Sec. 3.1 gives a high-level overview of our approach, and Sec. 3.2 defines the concept of vectorizable codelets, the interface between a high-level loop transformation engine and a low-level ISA-specific SIMD code generator. Sec. 3.3-3.6 detail our new framework for automatically extracting maximal codelets. Experimental results are presented in Sec. 3.7, and related work is discussed in Sec. 3.8.

3.1 Overview

The key to our approach to vectorization is a separation of tasks between a high-level program restructuring stage and an ISA-specific SIMD code generation stage. High-level program transformation frameworks such as the polyhedral/affine framework excel at finding loop transformation sequences to achieve high-level, cost-model-based objectives. Examples such as maximizing data locality [16], maximizing parallelism (fine-grain or coarse-grain [47, 85, 24]) and possibly balancing the two [101, 102] illustrate the ability of the polyhedral framework to aggressively restructure programs. On the other hand, SIMD-specific concerns such as ISA-specific vector instruction selection, scheduling, and register promotion have been implemented using frameworks that depart significantly from classical loop transformation engines [42, 90, 82].
In this work we formalize the interface between these two optimization stages —
the vectorizable codelet: a tile of code with specific, SIMD-friendly properties. We
develop a novel and complete system to generate and optimize vectorizable codelets.

High-level overview of algorithm  The data locality optimization algorithm in
a system like Pluto [24] is geared towards exposing parallelism and tilability for the
outer-most loops first, through aggressive loop fusion. This is critical for achieving
good L1 cache data locality through tiling, as well as coarse-grained multi-core paral-
lelization. However, for L1-resident code, those objectives are often detrimental: for
effective exploitation of vector-SIMD ISA’s, we need inner-most parallel loops, and
the addressing of stride and alignment constraints. Other work has looked at the
impact of unroll-and-jam for SIMD vectorization [32], but they do not consider the
problem of restructuring programs to maximize the applicability of unroll-and-jam,
nor any access stride constraint.

In this work, we develop an integrated system aimed at reconciling the different
considerations in optimizing for coarse-grain parallelism and cache data locality versus
effective SIMD vectorization. Our end-to-end strategy is as follows.

1. Transform the program for cache data locality (see Sec. 3.2.3), using: (1) a
model-driven loop transformation algorithm for maximizing data locality and
tilability [24]; (2) a parametric tiling method to tile all tilable loops found [11];
and (3) an algorithm to separate partial tiles and full tiles [97].

2. For each full tile, transform it to expose maximal vectorizable codelets (see
Sec. 3.3-3.6), using: (1) a new polyhedral loop transformation algorithm for
codelet extraction; (2) unroll-and-jam along permutable loops to increase data
reuse potential in the codelet(s); and (3) an algorithm for abstract SIMD vectorization, addressing hardware alignment issues.

3. For each vectorizable codelet, generate high-performance ISA-specific SIMD code using a special codelet compiler akin to FFTW’s genfft or the SPIRAL system. Further, use instruction statistics or runtime measurements to autotune the codelets if alternative implementations exist.

3.2 Vectorizable Codelets

The original program, which satisfy some specific properties that enable effective SIMD code to be synthesized for it using an ISA-specific back-end code generator. In order to illustrate the properties, we proceed backwards by first describing the process of abstract SIMD vectorization on a code satisfying those properties.

3.2.1 Abstract SIMD Vectorization

We use the term abstract SIMD vectorization for the generation of ISA-independent instructions for a vectorizable innermost loop. The properties to be satisfied by this inner-most loop are summarized in the following definition:

Definition 3 (Line codelet). A line codelet is an affine inner-most loop with constant trip count such that:

1. there is no loop-carried dependence,

2. all array references are stride-1 (fastest varying array dimension increments by one w.r.t. the innermost loop) or stride-0 (innermost loop absent from array index expressions),
3. unaligned load/stores are avoided whenever possible.

To illustrate the process of abstract SIMD vectorization, Listing 3.1 shows a code example which satisfies the above requirements. It is the transformed output of a sample full tile by our codelet exposure algorithm, to be discussed later. Listing 3.3 shows its abstract vector variant, after performing a basic unroll-and-jam as shown in Listing 3.2. The abstract vector code generation involves peeling the vectorizable loop with scalar prologue code so the first store (and all subsequent ones) will be aligned modulo \( V \), the vector length. Similarly, an epilogue with scalar code is peeled off. All arithmetic operators are replaced by equivalent vector operations, stride-0 references are replicated in a vector by “splatting”, stride-1 references are replaced by vload/vstore calls, and the vectorized loop has its stride set to the vector length \( V \). The arrays in this example, and in the remainder of this chapter, are assumed to be padded to make each row a multiple of the vector length \( V \).

**Listing 3.1:** After transfo. for codelet exposure

```plaintext
for (i = lbi; i < ubi; ++i)
  for (j = lbj; j < ubj; ++j) {
    R: A[i-1][j] = B[i-1][j];
    S: B[i][j] = C[i]*A[i-1][j];
  }
```

**Listing 3.2:** After unroll-and-jam \( 2 \times 1 \)

```plaintext
for (i = lbi; i < ubi; i += 2)
  for (j = lbj; j < ubj; ++j) {
    R: A[i-1][j] = B[i-1][j];
    S: B[i][j] = C[i]*A[i-1][j];
    R: A[i][j] = B[i][j];
    S: B[i+1][j] = C[i+1]*A[i][j];
  }
```

31
Our objective in this work is to automatically transform full tiles into code fragments satisfying Definition 3 (i.e., Listing 3.1), possibly interleaved with other code fragments not matching those requirements. Each vectorizable inner-loop is a vectorizable codelet. We show in Sec. 3.3-3.6 how to leverage the restructuring power of the polyhedral transformation framework to automatically reshape the loop nests to satisfy the requirements when possible.

### 3.2.2 Vectorizable Codelet Extraction

Efficient vector code generation requires to exploit data reuse potential. Specifically, given a program segment that is L1-resident, we should maximize the potential for register reuse while increasing the number of vector computations in the codelet. To do so we use register tiling, or unroll-and-jam, which amounts to unrolling outer loops and fusing the unrolled statements together.
Our framework is well suited to perform this optimization: the polyhedral framework can restructure a loop nest to maximize the number of permutable loops, along which unroll-and-jam can be applied. Arbitrarily complex sequences of fusion/distribution/skewing/shifting may be required to make unroll-and-jam possible, and we show how to automatically generate effective sequences for a given program to maximize the applicability of unroll-and-jam. As a result of our framework, the innermost vectorizable loops (the line codelets) will contain an increased number of vector operations and operands for the ISA-specific synthesizer to optimize. Definition 4 lists the various optimization objectives that drive the high-level transformation engine to ensure the creation of good (i.e. large) candidate vector codelets.

**Definition 4** (Maximal Codelet Extraction). *Given a program* \( P \), **maximal codelets** are obtained by applying polyhedral loop transformations on \( P \) to obtain a program \( P' \) such that:

1. the number of innermost loop(s) iterations which are parallel is maximized;
2. the number of references in \( P' \) which are not stride-0 or stride-1 is minimized;
3. the number of permutable dimensions is maximized.
4. the number of unaligned stores is minimized;
5. the number of unaligned loads is minimized;

In this work, we apply maximal codelet extraction on each full tile in the program. We then apply unroll-and-jam on all permutable loops, apply abstract SIMD vectorization on all candidate codelets, before synthesizing ISA-specific SIMD code for each of them.
3.2.3 Program Transformations for L1-resident Blocking

Tiling for locality involves grouping points in an iteration space into smaller blocks (tiles) [131], enabling reuse in multiple directions when the block fits in a faster level of the memory hierarchy (registers, L1, or L2 cache). Tiling for coarse-grained parallelism partitions the iteration space into tiles that may be executed concurrently on different processors with a reduced frequency and volume of inter-processor communication. A tile is atomically executed on a processor, with communication required only before and after execution. The tiling hyperplane method [24] is an effective approach for tiling of imperfectly nested affine loops. However, it can only generate tiled code for tile sizes that are fixed at compile-time. Alternatively, parametric tiling [58, 11], used in the present work, enables run-time selection of tile sizes. It has the advantage that tile sizes can be adapted to the problem size and machine used without recompiling the code.

The first stage of our optimization flow is as follows. Given an input program, we first apply a model-driven optimization geared towards maximizing the applicability of tiling [24]. Then parallel parametric tiling is applied, followed by full-tile separation. The process of separating full tiles leads to rectangular blocks of code whose loop bounds are functions of the tile sizes (which are run-time parameters). Each full tile is a potential candidate for effective SIMD execution using the codelet extraction and synthesis approach detailed below. We note that the codelet extraction and synthesis techniques are not limited to tilable programs. However, in our experiments in Sec. 3.7 we use benchmarks which can be processed by parallel parametric tiling.

The downside of parametric tiling is the generation of very complex loop bounds for the loops iterating on the various blocks (inter-tile loops), typically involving
complex compositions of `ceil()`, `floor()` and `round()` expressions. Current production compilers often fail to successfully analyze the dependences in such loops, and therefore are unable to automatically generate the best SIMD code for the full tiles. This leads to significant performance loss in the generated program. However, using the information extracted by the polyhedral compilation framework, full tiles can be manipulated, transformed and vectorized using our proposed approach.

3.3 Convex Set of Semantics-Preserving Transformations

A key reason for the power and effectiveness of the polyhedral transformation framework is the ability to formulate, with a single set of (affine) constraints, a set of semantics-preserving (affine) program transformations [120, 102]. An optimization problem whose solutions are subject to these constraints will necessarily lead to a semantics-preserving program transformation.

To build such a set of constraints, the reasoning is as follows. First, we observe that for all pairs of dependent instances \( \langle \vec{x}_R, \vec{x}_S \rangle \), the dependence is strongly satisfied if \( \Theta(\vec{x}_R) < \Theta(\vec{x}_S) \), that is if the producer is scheduled before the consumer. As \( \Theta \) is in practice a multi-dimensional function, we can more precisely state that the dependence is strongly satisfied if \( \Theta(\vec{x}_R) \prec \Theta(\vec{x}_S) \), where \( \prec \) is the lexicographic precedence operator. This can be rewritten as \( \Theta(\vec{x}_S) - \Theta(\vec{x}_R) \geq \delta_p \) with \( \delta_p \in \{0, 1\} \). We note that once a dependence has been strongly satisfied at a dimension \( d \), then it does not contribute to the semantics constraint and so we have instead \( \forall p > d, \Theta_p(\vec{x}_S) - \Theta_p(\vec{x}_R) \geq -\infty \) (that is, a "void" constraint with no impact on the solution space).
The constraints for semantics preservation are summarized in Def. 5, and are the starting basis for our new optimization algorithm. For each row $p$ of the scheduling matrix $\Theta$ and each dependence polyhedron $D_{R,S}$, we associate a Boolean decision variable $\delta_{D_{R,S}}^p$; these decision variables are used to encode semantics-preserving properties of a schedule so that for each pair of instances in dependence the source instance will be scheduled necessarily before the target instance. We bound the coefficients of $\Theta$ to be in some arbitrary range, so that we can find some arbitrarily large $K \in \mathbb{Z}$ such that the form $(K.\vec{n} + K)$ is an upper bound of the schedule latency [102], therefore $\Theta_p(\vec{x}_S) - \Theta_p(\vec{x}_R) \geq -(K.\vec{n} + K)$ is equivalent to $\Theta_p(\vec{x}_S) - \Theta_p(\vec{x}_R) \geq -\infty$.

**Definition 5** (Semantics-preserving affine schedules). Given a set of affine schedules $\Theta^R, \Theta^S \ldots$ of dimension $m$, the program semantics is preserved if the three following conditions hold:

(i) $\forall D_{R,S}, \forall p, \delta_{D_{R,S}}^p \in \{0, 1\}$

(ii) $\forall D_{R,S}, \sum_{p=1}^m \delta_{D_{R,S}}^p = 1$ (3.1)

(iii) $\forall D_{R,S}, \forall p \in \{1, \ldots, m\}, \forall (\vec{x}_R, \vec{x}_S) \in D_{R,S}$, (3.2)

$$
\Theta^S_p(\vec{x}_S) - \Theta^R_p(\vec{x}_R) \geq -\sum_{k=1}^{p-1} \delta_{D_{R,S}}^k (K.\vec{n} + K) + \delta_{D_{R,S}}^p
$$

As Feautrier proposed, the affine form of the Farkas lemma is used to build the set of all non-negative functions over a polyhedron [47], a straightforward process as Eq. 3.2 represents a non-negative function and $D_{R,S}$ is a polyhedron. As a result, a convex set of constraints linking the various $\theta_{i,j}$ coefficients and the $\delta$ variables is obtained, such that each point in this space is a semantics-preserving schedule [102].
3.4 Codelet-Specific Cost Functions

We now present the set of additional constraints we use to formulate our scheduling optimization problem. The complete scheduling algorithm works in two steps: first we find a schedule for the program that maximizes parallelism, the number of stride-0/1 references, and permutability. Then we apply a second algorithm for minimizing the number of unaligned load/stores on the transformed program.

3.4.1 Additional Properties on the Schedule

In the present work, we impose specific properties on Θ: θ_{i,j} ∈ \mathbb{N}. Enforcing non-negative coefficients greatly simplifies the process of finding good schedules. Large coefficients for Θ can lead to complex code being generated, containing modulo operations [14], and small values for the coefficients are usually preferred [99]. Our ILP formulation attempts to minimize the value of coefficients, therefore making them as close to 0 as possible when θ_{i,j} ≥ 0.

In addition, we use the so-called 2d+1 schedule form [49], which forces the scheduling function to be an interleaving of linear and constant dimensions. The number of rows of Θ^S is set to 2d + 1 if S is surrounded by d loops in the original sub-program, and every odd row of Θ^S is made constant (that is, θ_{i,j} = 0 for all j except j = d + n + 1). For instance, in this 2d+1 form, the original schedule of Listing 3.1 is Θ^R : (0, i, 0, j, 0) and Θ^S : (0, i, 0, j, 1). Those are computed simply by building the AST of the sub-program with loops and statements as nodes, and edges between a parent and its children in the AST. Edges are labeled by the syntactic order of appearance of the children, and to compute the original schedule one simply takes the
path from the root to the statement, collecting the node labels and the edge labels along this path.

While not technically required for this work, the 2d+1 form provides a standardized notation for the schedules, ensures that at least one schedule exists in the considered search space (the identity schedule, as shown above), and tends to simplify the generated loop bounds when implementing full distribution through the scalar dimensions.

### 3.4.2 Minimizing Dependence Distance

The first objective we encode is designed to find solutions with good data locality. It also helps with other objectives. The minimization finds a function that bounds the dependence distance [24], for each non-constant scheduling level (the even rows). This function is a form of the program parameters \( \vec{n} \), and requires additional constraints on \( \Theta \): all coefficients associated with the parameters \( \vec{n} \) are set to 0. This only removes parametric shifts, and is therefore not a limiting factor unless there are dependences of parametric distances in the program, a rare case. This bounding function is defined as follows, for each row \( k \) of the schedule:

\[
\begin{align*}
\mathbf{u}_k \cdot \vec{n} + w_k & \geq \Theta^S (\vec{x}_S) - \Theta^R (\vec{x}_R) \quad \langle \vec{x}_R, \vec{x}_S \rangle \in D_{R,S} \\
\mathbf{u}_k & \in \mathbb{N}^p, w_k \in \mathbb{N}
\end{align*}
\]

(3.3)

Intuitively, the value of \( \mathbf{u}_k \cdot \vec{n} + w_k \) is an indicator of an upper bound on the distance between dependent iterations. To minimize the distance between dependent iterations, we minimize the value of \( \mathbf{u}_k \) and \( w_k \). If, after minimization, \( \mathbf{u}_k \cdot \vec{n} + w_k = 0 \), then the dependent iterations are scheduled at exactly the same time for this loop level, i.e., the loop is parallel.
3.4.3 Maximizing Fine-grain Parallelism

Maximizing fine-grain parallelism requires that no dependences in the generated code are carried by the innermost loop(s). Feautrier addressed the problem of maximizing fine-grain parallelism through an aggressive strategy of strongly satisfying all dependences as early as possible [47]. We propose a different form: minimization of the number of dependences to be solved by the schedule dimension $2d$ (e.g., the one corresponding to the inner-most loop), together with the objective of minimizing the dependence distance at that level. We use the following optimization objective:

$$\min \sum_{DR,S} \delta_{DR,S}^2$$ (3.4)

Indeed, if this sum is equal to 0, and if $u_{2d,n} + w_{2d}$ (from the minimization of the dependence distance at that level) is also equal to 0, then the inner-most loop is parallel as no dependence is carried by this loop. Minimizing both objectives at the same time ensures that we discover inner-most parallel loops whenever possible.

3.4.4 Maximizing Stride-0/1 References

We propose a framework to embed directly, as constraints on the coefficients of $\Theta$, the maximization of the number of stride-0/1 references. It is a complex task to model such a constraint as a convex optimization problem, so that we can use standard solvers to find coefficients of $\Theta$. The reader is referred to a technical report [77] for detailed examples.

**Access functions after scheduling** In a nutshell, we aim for sufficient conditions for an array reference to be stride-0/1, when the innermost loop is parallel. These conditions come from properties of the polyhedral code generation process which
builds the expressions connecting the input iterator symbols (e.g., \( i \), \( j \)) with the new loop iterators created (e.g., \( t2 \), \( t4 \)). For instance, \( \Theta^R = (0, i + j, 0, j, 0) \) is a valid schedule for \( R \) in Listing 3.1. Two loops are created after code generation, and their schedule is given by the even dimensions of \( \Theta \): \( t2 = i + j \), \( t4 = j \). In the access \( A[i-1][j] \), after loop transformation, \( i \) and \( j \) are replaced by expressions in the form of the new loop iterators \( t2 \), \( t4 \). This process is done by constructing a system of equalities between the \( tX \) variables and the original iterators, based on the schedule, and performing Gaussian elimination. Equalities with one original iterator on the left hand side are then obtained. With the above example, we would get \( A[t2-t4-1][t4] \), which is not a stride-1 reference along \( t4 \), the inner loop.

Our approach to model sufficient conditions for the stride of a reference to be 0 or 1 is to reason about the system of equalities built to create the new references after loop transformations. Specifically, we want to ensure that the innermost loop iterator (to be vectorized) appears only in the right-most index function (the Fastest Varying Dimension, FVD) of an array reference, or does not appear at all in the reference. Otherwise, the reference is not stride-0/1.

As an illustration of a sufficient condition for stride-1: assuming the innermost non-constant schedule dimension \( p \) represents a parallel loop, then for a reference to be stride-1 it is enough to have (1) the schedule coefficients for rows less than \( p \) corresponding to the iterators present in non-FVD functions are non-zero at least once; (2) for at least one of the iterators present in the FVD function, the schedule coefficients for rows less than \( p \) corresponding to this iterator are all zero.
Convex encoding of access stride properties  To encode this kind of property, we resort to two complementary structures in the ILP. First, a new set of Boolean variables $\gamma_{i,j}$ is introduced, to help us model when a schedule coefficient $\theta_{i,j}$ is 0 or not. There are as many $\gamma_{i,j}$ variables as there are $\theta_{i,j}$ variables, and for each of them we set $\gamma_{i,j} \leq \theta_{i,j}$. Intuitively, we sum these $\gamma$ variables to count how many schedule coefficients are non-zero. For $\gamma_{i,j}$ to accurately capture when a schedule coefficient is non-zero, we maximize $\sum \gamma_{i,j}^R$ so that $\gamma_{i,j}^R$ is set to 1 as soon as $\theta_{i,j}^R$ is greater or equal to 1.\(^2\)

Second, we use an auxiliary representation of the access functions, using a normalized matrix $G^F$ for an access function $F$. Intuitively, it is used to represent which iterators appear in the non-FVD index functions, and which appear in the FVD with a coefficient of 1. $G^F$ has two rows: all iterators appearing in the non-FVD are represented in the first row, and in the second row iterators appearing in the FVD with a coefficient of 1 are represented. Precisely, for the access function matrix $F$ of dimension $l \times d + n + 1$, $G^F$ is constructed as follows. (1) $G^F$ has 2 rows and $d$ columns. (2) $\forall i \in \{1..l-1\}, \forall j \in \{1..d\}$, if $F_{i,j} \neq 0$ then $G_{1,j}^F = 1$, $G_{1,j}^F = 0$ otherwise; (3) $\forall j \in \{1..d\}$, if $F_{l,j} = 1$ then $G_{2,j}^F = 1$, $G_{2,j}^F = 0$ otherwise.

This leads to the following definition [77], wherein besides the $\gamma$ variables we also define new Boolean variables $\mu_i^F$ and $\nu_j^F$ (one pair per loop surrounding the reference $F$), and $\sigma_1^F$ and $\sigma_2^F$ to model this seemingly non-convex problem in a convex fashion.

Definition 6 (Constraint for Stride-0/1). Given a statement $R$ surrounded by $d$ loops; a legal schedule $\Theta^R$ where all loop-carried dependences have been strongly satisfied

\(^2\)As we perform lexicographic optimization, we can place this maximization as the very last optimization objective to not disrupt the actual performance-driven optimization objectives.
before level \(2d\); a memory reference \(F^A_R\) for an array \(A\) of dimension \(l\) with stride-0/1 potential; \(\mu_F^i, \nu_F^j, \sigma_F^1\) and \(\sigma_F^2\), a collection of Boolean decision variables; and \(G^F\) the normalized matrix for \(F\). When given semantics-preserving \(\Theta^R\) with parallel inner-dimension, the following constraints

\[
\left(\sum_{k=1}^{d-1} g_{1,j} \cdot \gamma_{2k,j} \geq g_{1,j} \cdot \mu_F^j \right) \land \left(\sum_{k=1}^{d-1} g_{2,j} \cdot \gamma_{2k,j} \leq (d-1) \cdot g_{2,j} \cdot \nu_F^j \right) \quad \forall j \in \{1..d\}
\]

\[
\left(\sum_{j=1}^{d} g_{1,j} \cdot \mu_F^j \geq \sum_{j=1}^{d} g_{1,j} \cdot \sigma_F^1 \right) \land \left(\sum_{j=1}^{d} g_{2,j} \cdot \nu_F^j \leq \sum_{j=1}^{d} g_{2,j} - \sigma_F^2 \right)
\]

are verified if \(\sigma_F^1 = 1\) and \(\sigma_F^2 = 1\) (stride-1), or if \(\sigma_F^1 = 1\) and \(\sigma_F^2 = 0\) (stride-0). Otherwise it is not stride-1/0.

The optimization objective is formulated as follows.

**Definition 7** (Maximization of Stride-0/1 References). The number of non-stride-0/1 references is minimized by encoding, for each memory reference \(F\) in the program, the constraints of Def. 6, and optimizing the following objectives:

\[
\max \sum_F \sigma_F^1, \max \sum_F \sigma_F^2
\]

**Example** To illustrate the above definitions, Fig. 3.1 shows how they are applied on a simple program with a single reference. The stride optimization objectives yield the optimum value of \(\sigma_1 = 1\) and \(\sigma_2 = 1\). \(\sigma_1 = 1\) implies that \(\mu^2 = 1\), which in turn makes \(\gamma_{2,2} = 1\). Thus, in the final schedule \(\theta_{2,2} \geq 1\). \(\sigma_2 = 1\) forces \(\nu^1 = 0\), which propagates and sets \(\gamma_{2,1} = 0\). In the final schedule, \(\theta_{2,1} = 0\) and \(\theta_{2,2} \geq 1\). This corresponds to making the loop \(j\) the outer loop.
for (i = 0; i < M; ++i)
for (j = 0; j < N; ++j)
A[j][i] = 0;

\[
G = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}
\]

\[
0 \cdot \mu^1 + 1 \cdot \mu^2 \geq (0 + 1) \cdot \sigma_1
\]

\[
1 \cdot \nu^1 + 0 \cdot \nu^2 \leq 1 - \sigma_2
\]

\[
0 \cdot \gamma_{2,1} \geq 0 \cdot \mu^1
\]

\[
1 \cdot \gamma_{2,2} \geq 1 \cdot \mu^2
\]

\[
1 \cdot \gamma_{2,1} \leq 1 \cdot \nu^1
\]

\[
0 \cdot \gamma_{2,2} \leq 0 \cdot \nu^2
\]

**Figure 3.1:** Example of stride-0/1 constraints

### 3.4.5 Maximizing Permutability

Our approach to maximize permutability is to add constraints to capture level-wide permutability [24], and maximizing the number of such dimensions. We first recall the definition of permutable dimensions.

**Definition 8 (Permutability condition).** *Given two statements R, S. Given the conditions for semantics-preservation as stated by Def. 5. Their schedule dimensions are permutable up to dimension k if in addition:*

\[
\forall \mathcal{D}_{R,S}, \forall p \in \{1, \ldots, k\}, \forall \langle \vec{x}_R, \vec{x}_S \rangle \in \mathcal{D}_{R,S},
\]

\[
\Theta^S_p(\vec{x}_S) - \Theta^R_p(\vec{x}_R) \geq \delta^\mathcal{D}_{R,S}_p
\]

(3.5)

Enforcing this constraint on all schedule dimensions may lead to an empty solution set, if not all dimensions are permutable. So, we need to relax this criterion in order to capture when it is true, while not removing schedules from the solution set. We start by introducing Boolean decision variables, \(\rho^\mathcal{D}_{R,S}_p\), i.e., one \(\rho\) variable per \(\delta\) variable, and set \(\delta^\mathcal{D}_{R,S}_p \geq \rho^\mathcal{D}_{R,S}_p\). These variables, when set to 1, will make Eq. (3.2) equal to Eq. (3.5), and will not impact the solution set when set to 0. This is achieved by
replacing Eq. (3.2) by:

$$\Theta_p^{S}(\vec{x}_S) - \Theta_p^{R}(\vec{x}_R) \geq - \sum_{k=1}^{p-1} (\delta_k^{D_{R,S}} - \rho_k^{D_{R,S}}) \cdot (K.\vec{n} + K) + \delta_p^{D_{R,S}}$$

When a dependence has been previously satisfied ($\delta_p^{D_{R,S}} = 1$ for some $p$), instead of nullifying the legality constraint at subsequent levels, we will attempt to maximize the number of cases where Eq. (3.5) holds, with the following optimization objective for a given dimension $p$:

$$\max \sum_{D_{R,S}} \rho_p^{D_{R,S}}$$

We note that as $\delta$ and $\rho$ variables are connected by an inequality, maximizing $\rho$ variables implicitly forces the strong satisfaction of dependences as early as possible.

### 3.5 Putting It All Together

The above constraints are all embedded into a single optimization problem, containing numerous Boolean decision variables, as well as the schedule coefficients for all statements. We combine all our optimization objectives into a single optimization problem that is solved by Integer Linear Programming, by finding the lexicographically smallest point in the space. The order of the optimization objectives determines which objective will be maximally/minimally solved first, with succeeding objectives being optimally solved given the max/min solution of the previous objectives. Our combined problem $P$ is:

$$P : \left\{ \min \sum_{D_{R,S}} \delta_{2d}^{D_{R,S}}, \min \, \mathbf{u}_{2d} \cdot \vec{n} + w_{2d}, \max \sum_{F} \sigma_1^F, \max \sum_{F} \sigma_2^F, \forall k \max \sum_{D_{R,S}} \rho_k^{D_{R,S}}, \forall k \neq 2d \min \, \mathbf{u}_k \cdot \vec{n} + w_k, \max \sum_{i,j} \gamma_{i,j} \right\}$$
There may still be multiple solutions to this optimization problem, each satisfying the criteria for extraction of maximal codelets as per Def. 4. While technically enumerating all optimal solutions should be performed for best performance, in practice we limit ourselves to the lexicographically smallest solution of the above problem. The resulting schedule is then applied to the program, polyhedral code generation is performed, and candidate codelets are detected on the generated code (all parallel innermost loops with only stride-0/1 references are candidate codelets). Those candidate codelets are the input to the second stage of our optimization framework, which performs retiming+skewing of statements to minimize the number of unaligned load/stores. In addition, permutable loops are detected through dependence analysis on the generated code, and are marked as candidate loops for unroll-and-jam.

3.6 Minimizing Unaligned Stores and Loads

The final stage of our method to extract valid candidate codelets for ISA-specific SIMD synthesis is to minimize the number of unaligned stores and loads in each candidate codelet found by the previous loop transformation step. Despite the fact that hardware alignment constraints are usually dealt with at a lower compilation level [42], this task should more naturally belong to the high-level transformation engine, where multidimensional loop transformations as well as precise array dataflow analysis can be performed.

To achieve this objective, we perform a combination of statement retiming at the codelet level, and additional loop skewing for the unroll-and-jammed loops. Intuitively, we ‘shift’ statements in the codelet such that (when possible) all array elements...
which are referenced by the vectorizable loop become aligned if the first element being referenced by the loop is aligned. Previous work by Henretty et al. [60] develops a complete framework for statement retiming so as to minimize stream alignment conflict on innermost vectorizable loops. Our candidate codelets fit their definition of vectorizable loops, and to find the relevant retiming factor for our optimization purpose, we use a slightly adapted version of that algorithm. The main difference is that we add an additional scalar quantity to the shifts found by their algorithm so that stores to different arrays need the same iteration peeling quantity to ensure alignment with the technique shown in Sec. 3.2.1. This is not strictly necessary, but simplifies the code generation and can reduce the number of peeled iterations in the scalar prologue/epilogue codes.

Finally, we need to take care of alignment properties in case of unroll-and-jam of the loops. For this purpose, we resort to skewing the unroll-and-jammable loops by a positive factor, so that the array index expression along the FVD is a multiple of the vector length. Such transformation is always legal, as a positive skew cannot change the dependence direction. As an illustration, for a Jacobi-2D example the access function after parametric tiling and full tile extraction, but before retiming is of the form \( A[-2t+i][-2t+j] \). Assuming \( V = 4 \), we apply an additional skew by 2 to unroll-and-jam along \( t \), to get \( A[-4t+i][-4t+j] \). In this manner, when unrolling by any factor along \( t \) and jamming in \( j \), references will still be vector-aligned in \( j \).

In summary, to minimize store/load alignment even in the presence of unroll-and-jam, for each candidate codelet we compute an ad-hoc polyhedral transformation made only of shifts and skews, and apply this transformation using a polyhedral code generator. The resulting codelet is then transformed to abstract vector code using
the technique depicted in Sec. 3.2.1, creating valid inputs to the next optimization stage: ISA-specific SIMD synthesis of vectorizable codelets, as detailed in Sec. ??.

3.7 Codelet Experimental Results

3.7.1 Experimental Protocol

We evaluate our framework on a collection of benchmarks where the core computational part is a SCoP. Specifically, we experiment with 5 stencil computations that arise in typical image processing and physical simulation applications, and 3 linear algebra benchmarks. Problem sizes are selected to be larger than the last level cache (LLC) size of 8 MB. Each benchmark is evaluated using single-precision (float) and double-precision (double). Experiments are performed on an Intel SandyBridge i7-2600K (3.4GHz, 4 cores, 32KB L1, 256KB L2, 8MB L3), using either the SSE or AVX vector instruction set. The testbed runs a native 64-bit Linux distribution. We use Intel ICC 13.0 with -fast -parallel -openmp, except for the sequential baseline where we use -fast. For AVX experiments, the flag -xAVX is used.

For each benchmark, we apply the following flow: (1) loop transformations for parallel parametric tiling with full-tile separation, and coarse-grain shared-memory parallel execution; (2) for each full-tile, perform loop transformation to expose maximal codelets, and perform abstract SIMD vectorization; (3) for each codelet found, perform ISA-specific SIMD synthesis. This framework is implemented using PolyOpt/C, a polyhedral compiler based on the LLNL ROSE infrastructure [2]. The SIMD codelet generator is implemented in the SPIRAL system’s back-end infrastructure. A web page and codelet generation web service with makefile tie-in can be found at [3]. On average, our end-to-end framework (not considering auto-tuning time) takes about
25 seconds to generate a program version for benchmarks like Jacobi-2d, laplacian-2d, or correlation; and about 3 minutes for benchmarks like Jacobi-3D, laplacian-3D, or doitgen.

3.7.2 Codelet Extraction Statistics

Table 3.1 reports statistics on the ILP scheduling problem developed in this work. We report for some representative benchmarks the number of dependence polyhedra for array references, the total number of schedule coefficient variables $\theta_{i,j}$, the total number of additional variables inserted to model our problem, the total number of constraints in the system to be solved (this includes the semantics preserving constraints from dependence polyhedra), and the time to optimally solve the ILP using PIPLib. More details can be found in a technical report [77].

<table>
<thead>
<tr>
<th>Benchmark</th>
<th># deps</th>
<th># refs</th>
<th># $\theta_{i,j}$</th>
<th># others</th>
<th># cst.</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>jacobi-2d</td>
<td>20</td>
<td>8</td>
<td>140</td>
<td>486</td>
<td>3559</td>
<td>3.0711</td>
</tr>
<tr>
<td>laplacian-2d</td>
<td>20</td>
<td>10</td>
<td>140</td>
<td>502</td>
<td>3591</td>
<td>3.1982</td>
</tr>
<tr>
<td>poisson-2d</td>
<td>32</td>
<td>12</td>
<td>140</td>
<td>674</td>
<td>5277</td>
<td>5.8132</td>
</tr>
<tr>
<td>correlation</td>
<td>5</td>
<td>9</td>
<td>70</td>
<td>177</td>
<td>640</td>
<td>0.0428</td>
</tr>
<tr>
<td>covariance</td>
<td>3</td>
<td>4</td>
<td>70</td>
<td>176</td>
<td>593</td>
<td>0.0415</td>
</tr>
<tr>
<td>doitgen</td>
<td>3</td>
<td>4</td>
<td>117</td>
<td>269</td>
<td>911</td>
<td>0.1383</td>
</tr>
</tbody>
</table>

Table 3.1: ILP statistics

3.7.3 Full Program Performance

One of the key aspects of our framework is the ability to perform extensive autotuning on a collection of parameters with significant impact on performance. The two categories of parameters we empirically tune are (1) the tile sizes, so as to partition
the computation in L1-resident blocks; and (2) the unroll-and-jam factor, that affects
the number of operations in a codelet. While technically the parameter space is
extremely large, it can be greatly pruned based on a number of observations. (1)
We test tile sizes whose footprint is L1-cache resident. (2) We set the tile size of
the vector dimension to be some multiple of the unrolling factor of this loop by the
vector length, so as to minimize the peeling required for alignment. (3) We keep only
tile sizes where the percentage of total computation performed by the full tiles is
above a given threshold, so as to avoid tile shapes where a significant fraction of the
computation is done by partial tiles. For 2D benchmarks we use a threshold of 80%,
and a threshold of 60% for 3D benchmarks. (4) We unroll-and-jam all permutable
dimensions, using unrolling factors of 1, 2, 4 and 8.

The auto-tuning is broken into two phases. In the first phase, approximately 100
tile sizes are tested and the 10 with the most iteration points in full tiles are selected.
This process takes 2-3 minutes. The second phase tests 6-8 codelet optimizations for
each of the 10 tile sizes, taking about 10-15 minutes.

**Time breakdown over full/partial tiles and tile codelets** To gain insights
into the effectiveness and potential limitations of our framework, we first present a
breakdown of time expended within tile codelets, within full tiles, and in partial tiles.
Table 3.2 shows the performance comparison using 3 metrics: **ATC**, the time spent
in all tile codelets; **AFT**, the time spent in all full tiles; **FP**, and the total time of
the full program. We make the following observations: Peeling is the main reason
for loss of performance when moving from ATC to AFT, and slowdowns can vary
between $1.1\times$ and $2\times$. The fraction of performance loss depends on the vector length
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>SIMD ISA</th>
<th>ATC (sec)</th>
<th>AFT (sec)</th>
<th>FP (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>jacobi-2d</td>
<td>SSE</td>
<td>0.040</td>
<td>0.046</td>
<td>0.068</td>
</tr>
<tr>
<td>jacobi-3d</td>
<td>SSE</td>
<td>0.458</td>
<td>0.514</td>
<td>1.113</td>
</tr>
<tr>
<td>jacobi-2d</td>
<td>AVX</td>
<td>0.036</td>
<td>0.054</td>
<td>0.077</td>
</tr>
<tr>
<td>jacobi-3d</td>
<td>AVX</td>
<td>0.188</td>
<td>0.354</td>
<td>1.062</td>
</tr>
<tr>
<td>laplacian-2d</td>
<td>SSE</td>
<td>0.046</td>
<td>0.056</td>
<td>0.071</td>
</tr>
<tr>
<td>laplacian-3d</td>
<td>SSE</td>
<td>0.482</td>
<td>0.538</td>
<td>1.131</td>
</tr>
<tr>
<td>laplacian-2d</td>
<td>AVX</td>
<td>0.031</td>
<td>0.042</td>
<td>0.061</td>
</tr>
<tr>
<td>laplacian-3d</td>
<td>AVX</td>
<td>0.197</td>
<td>0.373</td>
<td>0.993</td>
</tr>
<tr>
<td>poisson-2d</td>
<td>SSE</td>
<td>0.051</td>
<td>0.064</td>
<td>0.090</td>
</tr>
<tr>
<td>poisson-2d</td>
<td>AVX</td>
<td>0.029</td>
<td>0.049</td>
<td>0.075</td>
</tr>
<tr>
<td>correlation</td>
<td>SSE</td>
<td>0.840</td>
<td>0.915</td>
<td>1.179</td>
</tr>
<tr>
<td>correlation</td>
<td>AVX</td>
<td>0.752</td>
<td>0.917</td>
<td>1.139</td>
</tr>
<tr>
<td>covariance</td>
<td>SSE</td>
<td>0.846</td>
<td>0.922</td>
<td>1.159</td>
</tr>
<tr>
<td>covariance</td>
<td>AVX</td>
<td>0.733</td>
<td>0.883</td>
<td>1.121</td>
</tr>
<tr>
<td>doitgen</td>
<td>SSE</td>
<td>0.191</td>
<td>0.353</td>
<td>0.865</td>
</tr>
<tr>
<td>doitgen</td>
<td>AVX</td>
<td>0.158</td>
<td>0.308</td>
<td>0.834</td>
</tr>
</tbody>
</table>

Table 3.2: Time breakdown for kernels: All Tile Codelets (ATC), All Full Tiles (AFT=ATC+Peeling) and Full Program (FP=AFT+Partial Tiles)

and unrolling factors used in the FVD. Thus, these two parameters must be tightly coupled with the FVD tile size while remaining tile sizes must be relatively small for minimizing the peeling overhead. Also, partial tiles have a considerable impact on the FP performance. In general partial tiles can cause between 1.3× to 3× slowdown over AFT, and they are particularly detrimental for higher dimensional iteration spaces (e.g. Jacobi-3D and doitgen). In general, bigger tile sizes improve performance but only up to a certain point, as they also gradually push more iterations from full tiles into partial tiles, even reducing the number of tiles that can be executed concurrently. Finally, the accumulated effect of peeling and partial tiles execution can yield between 1.4× and 6× slowdown from ATC to FP.

**Overall performance** Finally, we report in Tables 3.3-3.4 the full-application performance for the benchmarks. For each benchmark × vector ISA × data type, we
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>PB Size</th>
<th>ICC seq</th>
<th>ICC par</th>
<th>PTile</th>
<th>Prevect</th>
<th>SIMD</th>
<th>SIMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>jacobi-2d</td>
<td>$20 \times 2000^*$</td>
<td>2.96</td>
<td>3.71</td>
<td>6.24</td>
<td>17.15</td>
<td>13.22</td>
<td>25.39</td>
</tr>
<tr>
<td>laplacian-2d</td>
<td>$20 \times 2000^2$</td>
<td>4.30</td>
<td>4.44</td>
<td>6.29</td>
<td>18.7</td>
<td>17.85</td>
<td>24.86</td>
</tr>
<tr>
<td>poisson-2d</td>
<td>$20 \times 2000^3$</td>
<td>4.23</td>
<td>6.68</td>
<td>17.47</td>
<td>19.56</td>
<td>14.75</td>
<td>31.77</td>
</tr>
<tr>
<td>jacobi-3d</td>
<td>$20 \times 256^3$</td>
<td>4.21</td>
<td>4.70</td>
<td>3.82</td>
<td>5.38</td>
<td>5.04</td>
<td>3.84</td>
</tr>
<tr>
<td>laplacian-3d</td>
<td>$20 \times 256^3$</td>
<td>4.80</td>
<td>5.44</td>
<td>4.39</td>
<td>6.13</td>
<td>5.67</td>
<td>6.99</td>
</tr>
<tr>
<td>correlation</td>
<td>$2000^3$</td>
<td>0.94</td>
<td>0.81</td>
<td>26.21</td>
<td>18.78</td>
<td>19.89</td>
<td>33.35</td>
</tr>
<tr>
<td>covariance</td>
<td>$2000^3$</td>
<td>0.97</td>
<td>0.99</td>
<td>26.53</td>
<td>18.70</td>
<td>20.08</td>
<td>34.85</td>
</tr>
<tr>
<td>doitgen</td>
<td>$256^4$</td>
<td>8.63</td>
<td>8.56</td>
<td>14.72</td>
<td>31.35</td>
<td>30.76</td>
<td>41.63</td>
</tr>
</tbody>
</table>

**Table 3.3:** Single precision performance data in GFLOP/s.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>PB Size</th>
<th>ICC seq</th>
<th>ICC par</th>
<th>PTile</th>
<th>Prevect</th>
<th>SIMD</th>
<th>SIMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>jacobi-2d</td>
<td>$20 \times 2000^4$</td>
<td>1.79</td>
<td>1.86</td>
<td>6.25</td>
<td>13.35</td>
<td>11.80</td>
<td>17.86</td>
</tr>
<tr>
<td>laplacian-2d</td>
<td>$20 \times 2000^2$</td>
<td>2.15</td>
<td>2.23</td>
<td>6.30</td>
<td>13.19</td>
<td>13.13</td>
<td>16.01</td>
</tr>
<tr>
<td>poisson-2d</td>
<td>$20 \times 2000^2$</td>
<td>3.08</td>
<td>3.36</td>
<td>16.86</td>
<td>15.19</td>
<td>11.31</td>
<td>20.67</td>
</tr>
<tr>
<td>jacobi-3d</td>
<td>$20 \times 256^3$</td>
<td>2.22</td>
<td>2.06</td>
<td>4.01</td>
<td>2.70</td>
<td>2.87</td>
<td>2.67</td>
</tr>
<tr>
<td>laplacian-3d</td>
<td>$20 \times 256^3$</td>
<td>2.55</td>
<td>2.38</td>
<td>4.58</td>
<td>3.34</td>
<td>3.17</td>
<td>4.15</td>
</tr>
<tr>
<td>correlation</td>
<td>$2000^3$</td>
<td>0.64</td>
<td>0.63</td>
<td>14.38</td>
<td>9.56</td>
<td>9.66</td>
<td>21.28</td>
</tr>
<tr>
<td>covariance</td>
<td>$2000^3$</td>
<td>0.65</td>
<td>2.16</td>
<td>13.52</td>
<td>9.66</td>
<td>9.70</td>
<td>21.88</td>
</tr>
</tbody>
</table>

**Table 3.4:** Double precision performance data in GFLOP/s.

compare our performance in GF/s to that attained by ICC on the input program (both sequentially and with automatic parallelization flags) and PTile, the performance of PTile [11], a state-of-the-art Parallel Parametric Tiling software (tile sizes have been auto-tuned). The performance achieved using only the front-end of our framework (transformation for codelet extraction) is reported in the Prevect columns,
i.e., we restructure the program to expose codelets but simply emit standard C code for the codelet body and use the C compiler’s vectorizer. **SIMD** reports the performance achieved by using our ISA-specific SIMD synthesizer on each codelet after codelet extraction.

We observe very significant performance improvements over ICC and PTile, our two reference compilers. Up to $50\times$ improvement is achieved using AVX for the covariance benchmark, for instance. We systematically outperform ICC (both with and without auto-parallelization enabled) with our complete framework using AVX (the SIMD/AVX column) since ICC fails to effectively vectorize most of the benchmarks. However, we note that due to the polyhedral model’s limitation of not allowing explicit pointer management, the stencil codes have an explicit copy-back loop nest instead of a pointer flip. We note that the stencil benchmarks are severely memory-bandwidth bounded. Although ICC is able to automatically parallelize and vectorize the benchmarks, it is unable to perform time-tiling on the benchmarks and therefore the memory bandwidth limits the speedup achieved from parallelism. We also outperform, usually very significantly, the PTile research compiler in all but two cases (Jacobi-3D and Laplacian-3D, DP). These anomalies are due to our criteria of "good tile sizes" during the auto-tuning phase: tile sizes are selected based on the percentage of points that will be executed in full tiles vs. partial tiles. Therefore, a good configuration for our line codelet is not necessarily the overall best tile configuration. As explained before, increasing tile sizes shift iterations from full to partial tiles (which are slower) and reduce the number of concurrent tiles down to possibly a single one.

We also remark that the Prevect performance does not necessarily follow the vector length (and hence the arithmetic throughput), nor does it necessarily outperform
ICC or PTile. One of the main reason is the complexity, in terms of number of basic blocks, of the codelets which are generated. By exploiting large tile sizes and unroll-and-jam, codelets with hundreds of blocks are generated and ICC simply cannot process the loop nest without splitting it, therefore negating the potential benefit of our approach. In general, we have observed that despite being in a form that satisfies the vectorizability criteria, the C code generated by our high-level framework does not lead to effectively exploited SIMD parallelism by the compiler, emphasizing the impact from an explicit coupling with a SIMD synthesizer. In some cases, ICC degrades performance with automatic parallelization, possibly due to inaccurate profitability models. PTile can also be slower than ICC parallel, due to complicated loop bound expressions as seen for single precision Jacobi and Laplacian 3D. The 3D stencils require a careful balance between maximization of the size of full tiles for high performance of the codelets, and keeping the size small enough to ensure that a high percentage of operations are in full tiles. Other tile shapes such as diamond tiling [8] may be needed for better performance.

3.8 Related Work

Automatic SIMD vectorization has been the subject of intense research in the past decades, i.e. [70, 131, 42, 90, 82]. These work are usually focusing on the back-end part, that is the actual SIMD code generation from a parallel loop [42, 90, 82], or on the high-level loop transformation angle only [70, 117, 32, 122]. To the best of our knowledge, our work is the first to address simultaneously both problems by setting a well-defined interface between a powerful polyhedral high-level transformation engine and a specialized SIMD code generator. Vasilache et al. also integrated SIMD and
contiguity constraints in a polyhedral framework, in the R-Stream compiler [122],
with similar objectives as ours. However, to the best of our knowledge, they are
not considering the coupling of this framework with a powerful back-end SIMD code
generator as we do. Other previous work considered inner- and outer-loop vector-
ization [91], our proposed work makes also a step forward by doing (tiled) loop nest
vectorization, as codelets embed in their body up to all iterations of the surrounding
loops.

Program generation (also called generative programming) has gained considerable
interest in recent years [52, 33, 18, 19, 110]. The basic goal is to reduce the devel-
opment, maintenance, and analysis of software. Among the key tools for achieving
these goals, domain-specific languages provide a compact representation that raises
the level of abstraction for specific problems and hence enables the manipulation of
programs [51, 20, 64, 115]. Our codelet generator is an example of such a program
generation system.

Automating the optimization of performance libraries is the goal in recent research
efforts on automatic performance tuning, program generation, and adaptive library
frameworks that can offer high performance with greatly reduced development time.
Examples include ATLAS [127], Bebop/Sparsity [65, 39], and FFTW [48] for FFTs.
SPIRAL [106] automatically generates highly efficient fixed-size and general-size li-
braries for signal processing algorithms across a wide range of platforms. SPIRAL
and FFTW provide automatic SIMD vector codelet generation while ATLAS utilizes
contributed hand-written SIMD vector kernels. While our transformation+synthesis
approach bears some resemblance with those work, we address a much more gen-
eral problem which requires to combine highly complex program transformations –
that can be modeled effectively only by means of the polyhedral framework – with ISA-specific code generation.
CHAPTER 4

Auto-parallelization of Sequential Programs with Streaming Data-Flow

Loop tiling and thread-level parallelization are two critical optimizations to exploit multi-processor architectures with deep memory hierarchies [6]. When exposing coarse grain parallelism, the programmer and/or parallelization tool may rely on data parallelism and barrier synchronization, such as the for worksharing directive of OpenMP, but this strategy has two significant drawbacks:

- Barriers involve a global consensus, a costly operation on non-uniform memory architectures; it is more expensive than the resolution of point-to-point dependences unless the task dependence graph has a high degree (e.g., high fan-in reductions) [25].

- To implement wavefront parallelism in polyhedral frameworks one generally resorts to a form of loop skewing, exposing data parallelism in a wavefront-based parallel schedule

An alternative is to rely on more general, task-parallel patterns. This requires additional effort, both offline and at runtime. Dedicated control flow must spawn coarse-grain tasks, that must in turn be coordinated using the target language’s constructs for the enforcement of point-to-point dependences between them. A runtime
execution environment resolves these dependences and schedules the ready tasks to worker threads [93, 26, 25, 96].

In particular, runtimes which follow the data-flow model of execution and point-to-point synchronization do not involve any of the drawbacks of barrier-based parallelization patterns: tasks can execute as soon as the data becomes available (i.e., when dependences are satisfied) and lightweight scheduling heuristics exist to improve the locality of this data in higher levels of the memory hierarchy; no global consensus is required and relaxed memory consistency can be leveraged to avoid spurious communications; loop skewing is not always required, and wavefronts can be built dynamically without the need of an outer serial loop.

Loop transformations for the automatic extraction of data parallelism have flourished. Unfortunately, the landscape is much less explored in the area of task parallelism extraction, and in particular the mapping of tiled iteration domains to dependent tasks. This chapter makes three key contributions:

- **Algorithmic:** We design a task parallelization scheme following a simple but effective heuristic to select the most profitable synchronization idiom to use. This scheme exposes concurrency and makes temporal reuse across distinct loop nests (a.k.a. dynamic fusion) possible, and further partitions the iteration domain according to the input/output signatures of dependences. Thanks to this compile-time classification, much of the runtime effort to identify dependent tasks is eliminated, allowing for a very lightweight and scalable task-parallel runtime.

- **Compiler construction:** We implement the above algorithm in a state-of-the-art framework for affine scheduling and polyhedral code generation, targeting
the OpenStream research language [96]. Unlike the majority of the task-parallel
languages, OpenStream captures point-to-point dependences between tasks ex-
licitly, reducing the work delegated to the runtime by making it independent
of the number of waiting tasks.

- **Experimental:** We demonstrate strong performance benefits of task-level au-
tomatic parallelization over state of the art data-parallelizing compilers. These
benefits derive from the elimination of synchronization barriers and from a bet-
ter exploitation of temporal locality across tiles. We further characterize these
benefits within and across tiled loop nests.

### 4.1 Motivating Example

We use the Blur-Roberts kernel performing edge detection for noisy images in
Listing 4.1 as illustrating example, with N=4000 and using double precision floating
point arithmetic. Table 4.1 shows the performance obtained when using Intel ICC
2013 as the compiler, using flags `-O3 -parallel -xhost`; PLuTo variants compiled
with `-O3 -openmp -xhost`; task variant corresponds to `task-opt` (See Sec. 4.12.3). The
original code peaks at 3.9 GFLOPS/sec on an AMD Opteron 6274 (although with half
the cores) and 8.35 GFLOPS/sec on an Intel Xeon E5-2650. (See Tab.4.4 in Sec. 4.12
for complete description of machines used). This program is memory bound but
contains significant data reuse potential, so tiling is useful to improve performance.
We leverage the power of the PLuTo compiler [24] to tile simultaneously for coarse
grained parallelism and locality, considering the two fusion heuristics `minfuse` (cut
non-strongly connected components at each level) and `smartfuse` (fuse only loops of
identical depth). For this example, the result of the third fusion heuristic of pluto,
maxfuse (fuse as much as possible) gives an identical output code than the smartfuse heuristic.

**Listing 4.1:** Blur-Roberts kernel

```c
for (i = 1; i < N - 1; i++)
    for (j = 1; j < N - 1; j++)
          A[i-1][j+1] + A[i+1][j-1] + A[i+1][j+1])/8;

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

<table>
<thead>
<tr>
<th>Processing cores</th>
<th>ref ICC</th>
<th>pluto minfuse</th>
<th>pluto smartfuse</th>
<th>our work</th>
</tr>
</thead>
<tbody>
<tr>
<td>opt-1</td>
<td>1.25</td>
<td>0.4</td>
<td>0.7</td>
<td>0.9</td>
</tr>
<tr>
<td>opt-8</td>
<td>1.25</td>
<td>2.7</td>
<td>3.9</td>
<td>4.7</td>
</tr>
<tr>
<td>opt-16</td>
<td>1.25</td>
<td>2.0</td>
<td>0.7</td>
<td>6.8</td>
</tr>
<tr>
<td>xeon-1</td>
<td>8.35</td>
<td>2.13</td>
<td>1.83</td>
<td>2.81</td>
</tr>
<tr>
<td>xeon-4</td>
<td>8.35</td>
<td>6.22</td>
<td>4.99</td>
<td>9.45</td>
</tr>
<tr>
<td>xeon-8</td>
<td>8.35</td>
<td>7.15</td>
<td>6.12</td>
<td>16.55</td>
</tr>
</tbody>
</table>

**Table 4.1:** Blur-Roberts kernel performance in GFLOPS/sec for AMD Opteron 6274 and Intel Xeon E5-2650, on 1, half and all cores.

The minfuse heuristic distributes all loops that do not live in the same strongly connected component (SCC). Thus, it tiles and parallelizes each loop nest independently, requiring 2 barriers as shown in Listing 4.2.
Listing 4.2: Blur-Roberts Kernel produced with PLuTo minfuse

```c
parfor (ti=0; ti < ubti; ti++)
    for (tj=0; tj < ubtj; tj++)
    {
        /* tile body of S1*/
    }
/* barrier */
parfor (ti=0; ti < ubti; ti++)
    for (tj=0; tj <= ubtj; tj++)
    {
        /* tile body of S2*/
    }
/* barrier */
```

Listing 4.3: Blur-Roberts Kernel produced with PLuTo smartfuse

```c
for (w = 0; w < wmax; w++)
{
    parfor (ti=f1(w); ti < f2(w); ti++)
    {
        /* tile body */
    }
    /* barrier */
}
```

**Smartfuse** performs a complex sequence of transformations such as multidimensional retiming and peeling to enable the fusion of program statements under a common loop nest, and skewing is used for the wavefront transformation to expose coarse-grain parallelism. The output exhibits an outermost sequential loop and a second outermost parallel loop followed by its implicit barrier (see sketch in Listing 4.3). The complete code of PLuTo variants can be found in [75].

As observed in Fig. 4.1, the performance does not increase linearly with the number of processors, instead it either reaches a plateau with the Intel Xeon or drastically drops as in the Opteron’s case.

Fig. 4.1 illustrates the nature of the data dependences in the Blur-Roberts code and the parallelization options for static affine scheduling, as represented by state-of-the-art polyhedral compilers such as PLuTo, versus dynamic task data-flow. The left half of the figure shows the iteration spaces for an unfused form of the code, with the left square representing the first loop nest and the right square the second loop nest,
along with 2D tiling of each loop nest, working on blocks of rows of the matrices. With PLuTo minfuse, a barrier is used between executions of tiles of the first and second loops. With smartfuse, the two loop nests are fused, but skewing is employed to expose coarse-grain parallelism using the parfor/barrier idiom supported in PluTo. After fusion, only wavefront parallelism is feasible with 2D tiling, with barriers between diagonal wavefronts in the tiled iteration space. Thus, there is a trade-off: with minfuse, the tiled execution of each loop nest is load-balanced, but inter-loop data reuse is not immediately feasible; with smartfuse, inter-loop data reuse is improved, but may lead to load imbalance due to the exploitation of wavefront parallelism. We remark that other tiling techniques such as split-tiling [61] or diamond-tiling [8] can enable better load-balance than wavefront-based rectangular tiling, it however still demands explicit barriers in the code between tile ”phases”. In this work, we focus exclusively on rectangular affine tiling.

Figure 4.1: Illustration of benefit of task data-flow over static affine scheduling
With a task data-flow model, it is possible to get the best of both worlds: increased degree of task-level parallelism as well as inter-loop data reuse. This occurs by creating 1D parallel tasks for each loop, and point-to-point task level synchronizations enable ready tasks from the second loop nest to be executed as soon as their input tasks (the ones corresponding to the same block row and the ones on either side) have completed. Thus, a “dynamic fusion” between the loop nests is achieved automatically, without the problem of load imbalance from the wavefront parallelism with a static affine tile schedule for the fused loop nests.

This problem has been recognized in the linear algebra community and specialized solutions have been designed [25]. We propose a solution for affine programs by leveraging properties of the schedule of tiles as computed by polyhedral compilers, and utilize it to determine at compile-time the inter-band (among disjoint loop nests) and intra-band (within a single loop nest) dependences. Our approach produces a fine interleaving of instances of the first and second band (outer iterations of the tiled loop nests). Unlike classical approaches in automatic parallelization, these dependences will then be instantiated at run-time, and fed to a dynamic task scheduler. The three last columns in table of Fig. 4.1 show the performance obtained in GFLOPS/sec when using two of PLuTo’s heuristics (See Fig. 4.2 and Fig. 4.3) and comparing these to our generated task-parallel code. As one can see, by using point-to-point synchronization and statically mapping tile-level dependences it is possible to greatly improve over state-of-the-art vendor and research compilers: on Intel’s Xeon we achieve nearly 2× latency improvement w.r.t. ICC and PLuTo’s best; whereas on AMD’s Opteron we obtain over 5× relative to the baseline and 1.5× over PLuTo.
Our technique can be summarized as follows. We first compute tile-level constructs which are the input to an algorithm that selects stream idioms to be used for each tile dependence or to a partition routine which splits the loop nests into classes that share identical input/output dependence patterns. This algorithm chooses when to extract parallelism across disjoint loops, while the partition routine allows to create a dynamic wavefront of tiles. Then a (static) task-graph is constructed to prune redundant dependences and to decorate it with dependence information. Finally, code is generated for the OpenStream run-time.

4.2 Barriers and Point-to-Point dependences

In this chapter [76] we introduce an approach to auto-parallelize a sequential program with a combined compiler/run-time framework. We target OpenStream [96], a light-weight task-parallel extension to the OpenMP [34] shared-memory run-time. The motivation here is the removal of data-parallel barriers which limits the amount of available parallelism, thus limiting performance. To achieve this objective we identify two distinct cases: outermost loop barriers which enforce synchronization between adjacent (disjoint) loops, and barriers present within loops and which execute a non-fixed number of synchronizations.

We differentiate these two cases due to OpenStream’s syntax and semantics. We require explicit point-to-point synchronizations because of its streaming nature. Therefore, to extract these dependences, we propose a partitioning technique to split iteration domains according to their behavior from a dependence perspective. The idea here is that each part of the partition has a different set of input and output dependences.
Once the new partitions are determined, we perform a specific code generation step that expresses the computed dependences as input and output streams. Our approach performs automatic coarsening via the PLuTo algorithm [24] implemented in the PPCG compiler [124].

4.3 Extracting Task Parallelism

The high-level flow that we follow to extract tasks at the granularity of a tile is shown in Figure 4.2. Its input is a sequential code which is first tiled with PLuTo’s algorithm. Then the tile-level polyhedral representation is extracted from the original statement representation and from the tile schedule computed in the previous stage. Next, the PRDG is obtained from this representation. The following stage partitions the tile domains according to the dependence signatures of all neighboring nodes. This produces a new set of domains for which a new PRDG is computed and input/output dependence patterns are associated to each new node/domain. Finally, code is generated and loops are decorated with OpenStream’s task syntax that describes the input and output dependence instances for each task/tile.

Figure 4.2: Compiler stages from a sequential input code to a tile level parallel version with point-to-point synchronization
The first stage of our method is to build a polyhedral representation of the *tiled* program, with one (macro-)statement per tile, each having an iteration domain (that is, capturing the set of dynamic executions of each tile). A *Tile-PRDG* is then produced, which captures data dependences between tiles. Next, we introduce an algorithm to select the most profitable task-parallel idiom, in terms of number of required synchronizations (incoming tile dependences). We present a processing step that allows to extract task parallelism from kernels that would otherwise use a (static) wavefront to expose tile-level parallelism, as well as handle cases where two disjoint loop nests have more than one dependence in common. The following step consists of building a static task-graph. Finally, we briefly discuss general details pertaining to polyhedral code generation.

### 4.4 Tile-Level Processing

The input to this compilation flow is a C program for which a tile schedule can be computed. In this work, we assume the PLuTo algorithm [24] has been used to enable tiling, however we are not limited to this particular tiling algorithm. We call *tile-level* the outermost or outermost and next outermost tile dimensions. Two tile-level abstractions must be constructed from the program statement domains, dependences and schedule: the tile domains and the tile dependences. These are determined by first projecting the intra tile dimensions $k..n$ of the schedule onto the tile dimensions $1..k - 1$ in the range of the schedule map $M^S$ of statement $S$:

$$M_{tile}^S = \text{PROJ}_{k..n}(\text{range}(M^S))$$
This yields a tile map $M_{tile}^S$ for each statement $S$, where $k$ is the starting dimension to be projected (2 for inter-band parallelism and 3 for intra-band) and $n$ is the number of output dimensions of the map. The modified map is then used to compute the tile domain by simply intersecting the map’s domain with the set representing the statement domain. The range of the resulting map is the tile domain.

Tile dependences are constructed in a similar way. Given a dependence map $M^{S\rightarrow T}$ describing a dependence from statement $S$ to statement $T$, and the tile maps $M_{tile}^S$ and $M_{tile}^T$ computed, the tile dependence $M_{tile}^{S\rightarrow T}$ is $(M_{tile}^S)^{-1} \circ M^{S\rightarrow T} \circ M_{tile}^T$, where $\circ$ is the composition of maps. At this stage, we remove existent non-forward dependences, i.e., those which have the exact same source and target tile coordinates (same tile instance). This type of dependence could emerge after the projection step, in which case the dependence was forward in some of the projected out dimensions. From the task-parallel data-flow perspective, these dependences are part of the task body of each tile instance, and do not represent any real flow of data. a domain $T.0[tt, ii]$ we produce the map $T.0[tt, ii] \rightarrow T.0[tt, ii]$) and subtracting them from the original dependence maps, prior intersection with the tile domain. dependences (due to the leading scalar dimension in the schedule). However, they do appear in single bands (e.g. seidel kernel), and must therefore be pruned.

In addition, all tile dependences that share the same source and the same target can be further simplified by: (1) combining pairs of basic maps within a single map by using the map coalescing ISL functions (this simplifies the representation); (2) if a new tile dependence is a subset of a previous dependence we do not include it; (3) conversely, if a new tile dependence is a superset of previous tile dependences, we
remove the previous ones and add only the new one; (4) if a tile dependence partially intersects with some other tile dependence we take their union.

Finally, each tile dependence represented by a map (in ISL terminology) is massaged by detecting equalities, removing redundancies and making them disjoint (basic maps that constitute a map are not guaranteed to be disjoint unless invoking an explicit ISL function) [123]. An additional pruning stage is later performed during the task graph construction to remove dependence polyhedra which are already captured through transitive dependence relations.

Here we introduce the kernel jacobi-2d (Fig. 4.4), which we use to explain the partitioning steps. On Fig. 4.3 we show 4 tile dependences computed from 9 statement dependences. These tile dependences will be further processed in order to make them uniform and proceed with the partitioning stage.

4.5 Partitioning

Domain partitioning is required in two scenarios: (1) when extracting inter-band parallelism, two nodes in the PRDG can share more than one normalized dependence (e.g. in blur-Roberts kernel), thereby needing different treatment for each dependence; (2) when the task graph consists of a single band of tiles, i.e. a single node in the PRDG. We note that when partitioning domains for the former case we only target the outermost tile dimension, whereas for the latter we work on the two outermost dimensions if dependences are uniform. Our approach for exposing task-level parallelism from a polyhedral program representation is to implement a partitioning scheme that groups tiles of a loop nest’s iteration domain into classes, where each class is associated with a single signature of incoming and outgoing, inter-tile
(or intra-tile) dependences. This single signature in turn enables the generation of
OpenStream directives and tasks with well defined input/output clauses.

**Listing 4.4:** Jacobi-2d kernel

```c
for (t = 0; t < tsteps; t++){
    for (i = 1; i <= n-2; i++)
        for (j = 1; j <= n-2; j++)
            S1: B[i][j] = (A[i][j] +
                A[i][j-1] + A[i][j+1] +
                A[i+1][j] + A[i-1][j])*
                0.2;
    for (i = 1; i <= n-2; i++)
        for (j = 1; j <= n-2; j++)
            S2: A[i][j] = B[i][j];
}
```

Our partitioning scheme takes as input the bands of permutable dimensions resulting from the PLuTo algorithm. However, any tiling algorithm that outputs a similar structure can also be used instead. We make the two following assumptions:

1. tile dependences of all statements within a permutable band must be subsumed by that of a single (macro-)statement,

2. all statements must live under a common and non-disjoint tile schedule space.

The first assumption is guaranteed by design of the PLuTo algorithm, which effectively groups statements in tiled parts of the schedule where all dependences may be collected as if the band was formed of a single, fused, macro-statement. The second one allows us to propagate the partition generated from a leading statement domain into the follower domains in the band. This is discussed next.

**Definition 9** (Leading Domain and Follower Domains). *The iteration domain $I^S$ of a statement $S$ that belongs to a tiled band is called the leading domain if*
\[ \text{tsteps, n} \rightarrow \{ \]
\[ \text{T}_0[0, 0, i2, i3] \rightarrow T_0[0, 0, o2, o3] : i2 \geq 0 \text{ and } \\
\text{n} \geq 3 \text{ and } o3 \leq 1 + i3 \text{ and } o2 \leq 1 + i2 \text{ and } 2o2 \leq i3 \]
\[ \text{and } 16o3 \leq 30 + n + 32i2 \text{ and } 16i3 \leq -5 + 2tsteps + n \]
\[ \text{and } 16o3 \leq -4 + 2tsteps + n \text{ and } 16i3 \leq 29 + n + 32i2 \]
\[ \text{and } 16o2 \leq -1 + tsteps \text{ and } 16i2 \leq -2 + tsteps \text{ and } o3 \]
\[ \geq i3 \text{ and } o2 \geq i2 \text{ and } 16o3 \leq 12 + n + 16i3 \text{ and } 32o2 \]
\[ \geq -27 - n + 16i3 \text{ and } 16o3 \leq 28 + n + 32o2; \]
\[ T_0[0, 0, i2, i3] \rightarrow T_0[0, 0, o2, o3] : 16o3 \leq -4 + 2tsteps + n \text{ and } 16i2 \leq -2 + tsteps \text{ and } 16i3 \leq 28 + n \]
\[ + 32i2 \text{ and } 16o3 \leq 30 + n + 32i2 \text{ and } i2 \geq 0 \text{ and } \\
o3 \geq 2o2 \text{ and } o2 \geq i2 \text{ and } n \geq 3 \text{ and } 32o2 \geq -26 - n + 16i3 \text{ and } o3 \geq i3 \text{ and } 16o3 \leq 28 + n + 32o2 \text{ and } 16i3 \leq \\
-6 + 2tsteps + n \text{ and } o3 \leq 1 + i3 \text{ and } o2 \leq 1 \]
\[ + i2 \text{ and } 16o2 \leq -1 + tsteps \text{ and } i3 \geq 2i2; \]
\[ T_0[0, 0, i2, i3] \rightarrow T_0[0, 0, i2, o3] : 16o3 \leq -3 + 2tsteps + n \text{ and } 16i2 \leq -1 + tsteps \text{ and } o3 \geq i3 \text{ and } 16o3 \]
\[ \leq 29 + n + 32i2 \text{ and } i2 \geq 0 \text{ and } 16i3 \leq -4 + 2tsteps + n \]
\[ \text{and } i3 \geq 2i2 \text{ and } n \geq 3 \text{ and } 16i3 \leq 28 + n + 32i2 \text{ and } \\
o3 \leq 1 + i3; \]
\[ T_0[0, 0, i2, i3] \rightarrow T_0[0, 0, o2, o3] : 16o3 \leq -3 + 2tsteps + n \text{ and } 16i2 \leq -2 + tsteps \text{ and } 16i3 \leq 29 + n + 32i2 \text{ and } 16o3 \]
\[ \leq 31 + n + 32i2 \text{ and } i2 \geq 0 \text{ and } 2o2 \leq i3 \text{ and } \\
16o3 \leq 29 + n + 32o2 \text{ and } n \geq 3 \text{ and } o2 \geq i2 \text{ and } o3 \]
\[ \geq i3 \text{ and } 32o2 \geq -27 - n + 16i3 \text{ and } 16i3 \leq -5 + 2tsteps \text{ and } \\
o3 \leq 1 + i3 \text{ and } o2 \leq 1 + i2 \text{ and } 16o2 \leq -1 + tsteps \}
\]

Figure 4.3: Jacobi-2d tile dependences

(i) the depth of this domain is maximal within the band (at the chosen granularity, 

i.e., of one dimension for inter-band and two dimensions for wavefront).

69
(ii) the domain with maximal depth is unique within the band. This implies that all tile dependences within the band are subsumed by the leading domain.

Any domain which is not the leading domain is a follower domain.

When Def. 9 does not hold, we name any statement domain with maximal (tile) depth the leading domain, but consider its tile domain as the union of all tile domains within the band that have maximal depth. In our Jacobi-2d example (Fig.4.4) both statements have the same dimensionality. Therefore, we compute the tile domain of each individual statement, take their union, and name either of the statements as the leading domain. Definition 9 combined with our assumptions on the input schedule guarantees that all dependences are subsumed by the tile domain of the leading domain. Partitioning can safely be applied to the leading domain, and then propagated onto the follower domains. In the case that the macro-statement (statements sharing a same band of tiles) has no self-dependence, then any statement can be chosen as the leading domain.

Each dependence relation is identified with a unique number 1..n, for n dependence relations. The dependence signature lists the unique identifiers of dependence relations that are either to or from the domain.

**Definition 10** (Dependence Signature). The dependence signature \( \text{Sig}^S \) of a domain \( I^S_{\text{tile}} \) is composed of two sets: the IN set and the OUT set. For each dependence relation \( k \), \( k \) is put in IN (resp. OUT) iff \( D^T \to S \) (resp. \( D^S \to T \)) has at least one destination (resp. source) in \( I^S_{\text{tile}} \).

\[
\text{Sig}^S = \{ \text{IN}^S, \text{OUT}^S \},
\]
\[ \text{IN}^S = \{k : \text{Ran}(D_{\text{tile}}^{k:T \rightarrow S}) \cap I_{\text{tile}}^S \neq \emptyset \}, \]

\[ \text{OUT}^S = \{k : \text{Dom}(D_{\text{tile}}^{k:S \rightarrow T}) \cap I_{\text{tile}}^S \neq \emptyset \} \]

It follows the definition of a disjoint partition of the iteration domain:

**Definition 11 (Domain Partition).** The partition \( P^S \) of a domain \( I^S \) is defined by the following conditions:

\[ P^S = \{P_i^S\}, \]

\[ I^S = \bigcup (P_i^S), \]

\[ P_i^S \cap P_j^S = \emptyset \land \text{Sig}(P_i^S) \neq \text{Sig}(P_j^S), i \neq j \]

where an iteration domain \( I^S \) is divided into \( P_i^S \) disjoint domains such that they all have different dependence signatures.

Fig.4.4 shows all the possible signatures that could be extracted from a 2D tile domain having as input dependences which are parallel to the axes in the transformed space. The actual signatures that arise after partitioning are a function of the tile schedule as well as the parameters. In the figure, shaded circles represent signatures never applicable to jacob-2d’s tiled code. Moreover, some partitions might not execute at runtime due to the actual parameter values (e.g., the partition associated to signature 7).

**Making dependences uniform** Fig.4.5 shows the tile dependence map for the jacobi-2d kernel before making all dependences uniform. In order to apply our partitioning algorithm we convert all the basic maps that constitute a tile dependence into their uniform shape. We do so by expanding the output dimensions that produce a
Figure 4.4: Potential signatures generated by uniform dependences (gray denotes signatures that do not appear in jacobi-2d kernel)

\[
[tsteps,n] \rightarrow \{ \\
T_0[0,0,i2,i3] \rightarrow T_0[0,0,i2+1,o3]: o3 >= i3 \\
\text{and } 16i2 <= -17 + tsteps \text{ and } 16i3 <= 29 + n \\
\quad + 32i2 \text{ and } 16o3 <= 31 + n + 32i2 \text{ and } i2 >= 0 \\
\quad \text{and } i3 >= 2 + 2i2 \text{ and } o3 <= 1 + i3; \\
T_0[0,0,i2,i3] \rightarrow T_0[0,0,o2,i3+1]: 16o2 = -1 + tsteps \text{ and } 16i2 = -1 + tsteps \text{ and } 16i3 <= -19 + 2tsteps + n \text{ and } 8i3 >= -1 + tsteps \text{ and } tsteps >= 1; \\
T_0[0,0,i2,i3] \rightarrow T_0[0,0,i2,i3+1]: 16i3 <= -19 + 2tsteps + n \text{ and } 16i2 <= -2 + tsteps \text{ and } 16i3 <= 13 + n + 32i2 \text{ and } n >= 3 \text{ and } i2 >= 0 \text{ and } i3 >= 2i2; \\
T_0[0,0,i2, 2i2+1] \rightarrow T_0[0,0,i2+1,2i2+2]: i2 >= 0 \text{ and } 16i2 <= -17 + tsteps \text{ and } n >= 3 
\}

Figure 4.5: Jacobi-2d tile dependences prior to transitive reduction pruning

range of values from the input dimensions, e.g., the first basic map with constraints \( o3 >= i3 \) and \( o3 <= 1 + i3 \) is expanded into 2 basic maps. The result of this expansion is shown in Fig.4.6. This step allows to further prune dependences by transitive reduction \cite{89} \cite{88}, remove new covered dependences, and duplicated dependences.
\[
\text{[tsteps, n] -> { }
\text{* T_0 [0,0,i2,i3] -> T_0 [0,0,i2+1,i3] ; }
\text{T_0 [0,0,i2,i3] -> T_0 [0,0,i2+1,i3+1] ; }
\text{* T_0 [0,0,i2,i3] -> T_0 [0,0,i2,i3+1] ; }
\text{T_0 [0,0,i2,i3] -> T_0 [0,0,i2+1,i3+1] ; }
\text{T_0 [0,0,i2,i3] -> T_0 [0,0,i2,i3+1] ; }
\text{T_0 [0,0,i2,2i2+1] -> T_0 [0,0,i2+1,2i2+2] ; }
\text{}}
\]

**Figure 4.6**: Jacobi-2d tile (uniform) dependences. Dependences marked with * are the non-redundant ones and used for partitioning.

At this point, all output sets are explicit functions of their input sets. This permits to replace the tile dependence constraints by the constraints of the (leader) tile domain. The net effect of this is that dependences become broader, but also enables to remove redundant communication. After this process, only two non-redundant dependences are left, each parallel to one of the outermost axis on the transformed space. The only two required dependences are marked with a *.

Algorithm 4.1 is used to perform domain partitioning (when required), based on Definition 10 and 11, during the task graph construction and after pruning transitively covered dependences. The domain of a band could be partitioned both according to the incoming dependences as well as the outgoing dependences, e.g. domain \( S \) could be partitioned first when processing dependence \( R \rightarrow S \) and again partitioned when processing dependence \( S \rightarrow T \), and also due to multiple dependences between two nodes (since the PRDG is a multigraph).
**Input:** $G$: Tile-Level PRDG; $u$: source tile domain; $v$: target tile domain

**Output:** Updated $G$ where nodes $u$ and $v$ have been decorated with their partitions

1. if $\text{parts}(u) = \emptyset$ then $\text{parts}(u) \leftarrow I^u_{\text{tile}}$
2. if $\text{parts}(v) = \emptyset$ then $\text{parts}(v) \leftarrow I^v_{\text{tile}}$
3. foreach tile dependence $d$ from $u$ to $v$ do
   - $\text{indom} \leftarrow \text{domain}(d)$
   - $\text{outdom} \leftarrow \text{range}(d)$
   - foreach $p$ in $\text{parts}(u)$ do
     - $s \leftarrow p \cap \text{indom}$
     - if $s \neq \emptyset$ then
       - $\text{diff} \leftarrow p - s$
       - $\text{SIG}(\text{diff}) \leftarrow \text{SIG}(p)$
       - $\text{SIG}(s) \leftarrow \text{SIG}(p) \cup \{\text{OUT}^s = d\}$
       - remove $p$ from $\text{parts}(u)$
       - insert $s$ and $\text{diff}$ in $\text{parts}(u)$
       - $\text{indom} \leftarrow \text{indom} - p$
     - end
   - end
   - foreach $p$ in $\text{parts}(v)$ do
     - $s \leftarrow p \cap \text{outdom}$
     - if $s \neq \emptyset$ then
       - $\text{diff} \leftarrow p - s$
       - $\text{SIG}(\text{diff}) \leftarrow \text{SIG}(p)$
       - $\text{SIG}(s) \leftarrow \text{SIG}(p) \cup \{\text{IN}^s = d\}$
       - remove $p$ from $\text{parts}(v)$
       - insert $s$ and $\text{diff}$ in $\text{parts}(v)$
       - $\text{outdom} \leftarrow \text{outdom} - p$
     - end
   - end
end

**Algorithm 4.1:** PartitionDomains $(G, u, v)$
**Input:** PRDG: Tile-Level PRDG  
**Output:** T: Task graph with signatures and partitioned domains

```plaintext
foreach written array A do
    Collect dependences on array A
    Traverse the PRDG and remove transitively covered dependences on array A
end
```

```plaintext
foreach edge u → v do
    if u or v have more than 1 dependence then
        PartitionDomain(PRDG,u,v)
    end
end
```

```plaintext
T ← ∅
foreach edge u → v ∈ PRDG do
    π_u ← Get Partitioned Domains of u
    π_v ← Get Partitioned Domains of v
    Add a node to T for each domain in π_u and π_v
    Add edges for dependences between π_u and π_v to T
end
```

return T

**Algorithm 4.2:** ConstructTaskGraph (PRDG)

Fig. 4.8 shows the partitions generated by Algorithm 4.1, the respective dependence signatures, and the distribution of the signatures around the (tile) domain. It may be surprising to see signature 7 in this figure. However, this signature could be triggered depending on the parameter values, specifically for this condition:

\[(2 \ast ((tsteps - 1)\%16) + ((14 \ast tsteps + 15 \ast n + 2)\%16) <= 13 \&\& (tsteps - 1)\%16 <= 6)\]

**Partial tiles** A band of tiles contains a partial tile when one or more of its (intra-tile) dimensions have a smaller cardinality than the selected tile size. Our algorithm does not distinguish partial tiles from full tiles (tiles which have their cardinality equal to the tile size), since we are only interested in the (outer) tile coordinates, and which are only dependent on the problem size and outer tile coordinates. Furthermore,
partial tiles already have a different dependence pattern, and only appear at the end of some domain. As such, they are naturally placed into their own (signature) class. A similar handling applies to all tiles which appear in a domain border, i.e., a first or last tile instance along some dimensions.

In the following two sections we will discuss in more detail when partitioning is required.

4.6 Task Graph Construction

The task graph construction takes as input the decorated PRDG. We contemplate a single-dimensional tile band PRDG in order to limit the number of potential synchronizations, which increases as one considers more tile dimensions by factors that depend on the problem sizes as well as the chosen tile size. This still allows a sufficiently fine interleaving between the tile instances. Thus, this stage is only performed if we have multiple loop nests after tiling.

The goal of this step is to traverse the PRDG and consider the minimum number of edges in the task graph so that all dependences are satisfied as well as being able to remove redundant synchronizations that can arise from transitively covered dependences, which often appear in PRDGs composed of several bands.

Algorithm 4.2 shows the pruning steps performed for the dependences (edges of the PRDG) on each array (scalars are 0-dimensional arrays) and the addition of the relevant edges from the PRDG into the task graph. Transitively covered dependences are found by composing the dependence relations in between PRDG nodes. For example, given nodes $R$, $S$ and $T$, and dependences $R \rightarrow S$, $S \rightarrow T$ and $R \rightarrow T$, then dependence $R \rightarrow T$ can be removed if it is equal to the composition $R \rightarrow S \circ S \rightarrow T$. 

76
These dependences can be of type RAW, WAR and WAW. Then, Algorithm 4.1 is invoked for nodes \( u \) and \( v \) of the PRDG when they have more than one dependence relation connecting them, i.e. 2 or more dependences which differ in either source or domain or both (such as in blur-Roberts for instance). Finally, the underlying task graph is constructed from the partitioned domains attached to each node of the PRDG.

4.7 Implementing Inter-Band Parallelism

We now use the 3mm benchmark from Polybench/C to illustrate the various implementations of inter-band parallelism that can be achieved. Our algorithm automatically prioritizes the possible implementations based on data dependence features, as shown later.

Listing 4.5: 3mm kernel (with removed initializations)

```c
for (i=0; i<l; i++) // <-- Band 1
    for (j=0; j<m; j++)
        for (k=0; k<q; k++)
            S1: A[i][j] += B[i][k] * C[k][j];
for (i=0; i<m; i++) // <-- Band 2
    for (j=0; j<n; j++)
        for (k=0; k<p; k++)
            S2: D[i][j] += E[i][k] * F[k][j];
for (i=0; i<l; i++) // <-- Band 3
    for (j=0; j<n; j++)
        for (k=0; k<m; k++)
            S3: G[i][j] += A[i][k] * D[k][j];
```

In the following the term *band* can be intuitively thought as a single loop nest. We need to decide whether two nodes connected by an edge in the task graph can be effectively parallelized with inter-band (across disjoint loop nests) task parallelism, if a soft barrier should be used to satisfy the dependences of the target node, or if a band should be considered as a single task. *We emphasize that this algorithm...*
is designed for a single level of intra-band parallelism. Thus, it focuses exclusively on 1-dimensional tile-level abstractions. Consider the dependence between $S1$ and $S3$ on array $A$, each $S3(i)$ depends on an $S1(i)$. Intuitively one can think of this as a "row-to-row" mapping. Consider now the dependence from $S2$ to $S3$ on array $D$. Each $S3(i)$ requires all the instances $S2(i, j, k)$, a many-to-1 relation. However, when considering all instances of $S3$ this becomes a many-to-many relation. Thus, depending on the desired granularity, this may lead to oversynchronization. At the very least each $S3(i)$ will have to wait for the $m$ instances of $S2(i)$, and could become as bad as $(l \times n \times m)$ instances of $S3$ each waiting for $(m \times n \times p)$ instances of $S2$.

This highlights that additional granularity of parallelism is not for free, as synchronization could increase in several orders of magnitude, and the additional benefits of parallelism will be mitigated. This granularity criterion also serves a second purpose, which is to exclusively restrict inter-band parallelism to the outermost dimensions of nodes/bands in a dependence relation. Finally, we systematically treat untiled bands
(for instance those that result from multi-level reduction statements) as single tasks.

The motivation comes from the exponential growth in synchronizations (a factor of tile-size for each dimension) that can arise with such loops.

**Input:** G: Task-graph after partitioning  
**Output:** G: Decorated task-graph

```plaintext
for each edge e ∈ G do  
    S ← tile-level domain of dependence source  
    T ← tile-level domain of dependence target  
    m ← dependence map S → T of edge e  
    m ← intersect domain of m with S  
    m ← intersect range of m with T  
    m ← decompose m into a union of basic maps in ISL  
    if S is an untiled band then  
        Create a new stream single  
        Output a single dependence from S  
        Input dependence of T with peek operation from S  
        Insert a tick operation after band T  
    end  
    else  
    if (Def. 12 is true for S and T) then  
        Create an array of streams D, one per distinct basic map in m  
        Decorate e with 1-to-1 inter-band parallelism on D  
    end  
    else  
        Create a stream bar of size 1 for barrier synchronization  
        Insert statement `barout` between nodes S and T  
        Output all dependences of S to bar  
        Input dependences of `barout` from S  
        Output a single dependence of `barout` to T  
        Input dependence of T from `barout` with peek operation  
        Insert statement `barin` after node T with tick operation on bar  
    end  
end
```

**Algorithm 4.3:** SelectTaskIdiom (G)

Algorithm 4.3 prioritizes the 3 types of communication from less communication (single task), to 1-to-1 mappings, to m-to-n communication patterns that require soft
barriers. Along the process nodes are decorated with the type of synchronization and the name of the stream to be used; and new nodes are inserted to handle barriers and tick operations. The latter are OpenStream operations responsible of consuming data tokens which were previously broadcasted. Fig. 4.7 shows the 1-dimensional tile-PRDG (left) and the result of applying our algorithm to it (right).

Inter-band parallelism is possible and profitable between bands B1 and B3 (akin to a row-to-row mapping). However, the algorithm deems unprofitable the number of synchronizations between bands 2 and 3. Thus the task graph is modified to reflect the insertion of a node BX which collects the output dependences from B2, and outputs a single dependence which is reused via peek operations in B3. This is OpenStream’s broadcast stream idiom. After B3 is also necessary to insert a new node BY to consume the token that has been used by all instances of B3 (a tick operation). Experimental results show that for this 3nn example, 1D inter-band parallelism (only one barrier synchronization) outperforms by 41% on AMD Opteron a 2D inter-band parallelism approach (no barriers, three 1-to-1 2D mappings and two many-to-many synchronizations with peek operations). The performance between the two schemes is comparable on Intel’s Xeon, and also in favor of 1D inter-band parallelism for AMD Phenom.

We now formalize the requirements for inter-band parallelism. Definition 12 establishes that a fine-grain interleaving of loop iterations between two bands can be legally executed. Once an iteration of the source band is executed, the dependent iteration of the target band can initiate execution.

**Definition 12** (Inter-band parallelism). *Given two distinct bands A and B. Barrier-less inter-band parallelism is exploitable if:*

1. *there exists at least one point in band B that does not depend on all the points of band A*

2. *Neither band A nor band B have dependence cycles*

Def. 12 is a general definition for inter-band parallelism. Condition 1) enforces that some subset of the target band (possibly exhibited after domain partitioning) can initiate early execution, i.e., a barrierless behavior. Condition 2) essentially states when a point-to-point synchronization is possible. Some of the cases that this definition covers are:

- when all points from band A have exactly one outgoing dependence instance, and all points from band B have exactly one incoming dependence instance (e.g. between bands B1 and B3 in Fig. 4.7);

- when the number of dependence instances outgoing from band A are bounded by some fixed number $K_A$, and the number of incoming dependence instances to band B are bounded by a fixed number $K_B$ (e.g. in blur-Roberts kernel, $K_A = K_B = 3$). Note also that not all points in both bands are required to have the same number of incoming or outgoing dependence instances;

- with no particular relation between the cardinalities of band A and B (e.g., $\text{card}(A) = \text{card}(B)$, $\text{card}(A) = 2 \times \text{card}(B)$, $\text{card}(A) = \text{card}(B) + K$, etc).

Definition 13 is used in Algorithm 4.3 to select when a soft barrier must be inserted in the task-graph. We recall that a soft barrier is a barrier with point-to-point synchronization, that is, it does not affect all bands, but only the one that indeed requires that output data of the source band.
Definition 13 (Soft barrier synchronization). Given two bands A and B, if no barrier-less inter-band parallelism can be found according to Definition 12 then a soft barrier must be inserted between both bands.

4.8 Extracting Dynamic Tile-Level Wavefronts

The (static) partitioning scheme presented in Section 4.5 allows to extract dynamic wavefronts of tiles, i.e., all inter-tile instance dependences are made explicit to the run-time, which in turn determines the actual wavefront as tiles go executing and completing. Fig. 4.8 displays the partition produced for a 2-dimensional (tiled) iteration domain corresponding to the jacobi-2d running example. The arrows depict the direction in which the data flows, in this case, one token for each tile instance. The x-axis represents space while the y-axis represents time. Partitioning yields 9 (tile) domains, each identified by its signature number. The appearance of a specific 2-dimensional signature depends directly on the domains’ shape. For jacobi-2d, the outer space dimension (tile dimension-i) is skewed w.r.t to the time dimension and has as schedule (tt,2*tt + ti), where tt is the time tile coordinate and ti is the space tile coordinate. Fig.4.4 shows the possible dependence signatures on a given 2D partitioning. The shaded ones are signatures which do not arise in jacobi-2d.

4.9 Code Generation

During polyhedral code generation, special care must be taken to replicate the structures of partitioned domains as well as pretty-printing OpenStream clauses from
the annotated information in the task graph. This includes the generation of input/output clauses from the dependence signature of each node. Streams are allocated and declared before the SCoP. The appropriate scalar values on the tile schedule must be set to represent the correct interleaving of bands.

- Separating dimensions are schedule dimensions which hold scalar values. Their purpose is to keep separated certain domains. In general, polyhedral code generation will attempt to fuse all possible dimensions allowed by the schedule. Thus, in order to keep the generated partitions separated (and their code signature unique), we insert an additional scalar dimension at position 1 (offset to 0) for each band parallelized via dynamic wavefront. In standard polyhedral techniques this is not common practice. However, since the dependences of
the outer dimensions will be dynamically resolved, inserting an arbitrary and distinct scalar value for each partition remains legal.

- Stream declarations: The current implementation of OpenStream only supports 1-dimensional arrays of streams. For each inter-band stream we compute the volume of the outermost dimension and use it as the stream array size. For dynamic-wavefront we assume a 2-dimensional linearized array of dimensions $D_0 \times D_1$, where $D_0$ and $D_1$ are the volumes obtained by projecting out all dimensions except dimension 0 for the former and dimension 1 for the latter. However, this results in a non-affine parametric product. Thus, the product is computed out of the polyhedral code generation process.

- Stream indexation: Stream arrays are indexed by the (outer) tile coordinates, which is either the source or target of the tile dependence: all input streams read from their local tile coordinate and write to the tile coordinate determined by the outset of a tile dependence. Here is where the property of having output sets that are explicit functions of the input set in a tile dependence becomes essential. In addition, since the dynamic wavefront requires a 2-dimensional access to the stream array, which commonly results in a non-affine expression (e.g., iterator $\times$ parameter), the linearization process is performed in a pragmatization pass after polyhedral code generation.

- Simplification: To avoid some corner and degenerate cases such as having only one tile instance on any dimension, we set the parameter context (e.g., problem sizes) as having at least 4 tiles along a dimension. The partition shown in Fig.4.8 as well as the code in Listing 4.6 have this simplification.
Dealing with One-Time-Loops (OTLs): OTLs are loops which execute only once. Stream indexation requires all tile coordinates in order to map tile instances to individual streams in an array. Currently, PPCG (which uses CLooG-ISL) does an (over)optimization and removes all OTLs. In this case we resort to regenerate the explicit tile coordinates. However, other code generators such as the non-ISL based CLooG [14] allow to explicitly generate OTLs. This simplifies the stream mapping process.

Finally, a global taskwait is inserted after the SCoP so that the program does not terminate early. As a note, it is particularly useful to use unscaled tile iterators during code generation in order to facilitate the mapping between tile coordinates and stream arrays. The generated code for one of the partitions is shown in Listing 4.6. More examples of the generated code can be found in [75].

**Listing 4.6:** Jacobi-2d code snippet for partition with signature 6 (see Fig.4.8)

```c
for (int tt = 1; tt < (tsteps-1) / 16; tt++)
for (int ii = 2*tt+2; ii <= 2*tt + (n-3) / 16; ii++)
#pragma omp task input (stream_tt[(tt) * S_1_3 + ii] >> token1_1,
  stream_ii[(tt) * S_1_3 + ii] >> token1_1)
  output (stream_tt[(tt+1) * S_1_3 + ii] << token1_2,
  stream_ii[(tt) * S_1_3 + 1+ii] << token2_2)
{
for (int jj=ii; jj <= ii+(n-3)/16+1; jj++)
  for (int t=max(16*tt,-n+8*jj+2); t <= 16*tt+15; t++)
    for (int i=max(16*ii,-n+16*jj+2);
         i<=min(min(16*ii+15,16*jj+14),n+2*t-1); i++)
      for (int j=max(i+1,16*jj);
           j <= min(16*jj+15,n+i-2); j++) {
        if (n+2*t >= i+2) {
          B[-2*t+i][-i+j] = 0.2*(A[-2*t+i][-i+j] + A[-2*t+i][-i+j-1]
            + A[-2*t+i][-i+j+1] + A[-2*t+i+1][-i+j] +
            A[-2*t+i-1][-i+j]);
          A[-2*t+i-1][-i+j] = B[-2*t+i-1][-i+j];
        }
      }
}
```

85
4.10 Putting It All Together

Algorithm 4.4 brings together our approach for extracting both inter-band and intra-band tile-level parallelism, which allows to describe concurrent tasks with explicit, point-to-point dependences. Based on the number of tile bands produced, one of our two main techniques is applied, i.e., inter-band parallelism in the presence of more than one band, and our partitioning scheme which enables a dynamic wavefront of tiles.

4.11 Handling complete applications

The techniques described in this chapter can be applied to a variety of real world applications. To do so, the input program needs to be affine or to have affine friendly portions. Also, a number of pre-processing transformations would normally be required to make a program tilable and to further expose parallelism (e.g., array/scalar expansion, renaming, privatization and scalar expression inlining).

Partitioning scalability  Our partitioning technique could be easily extended to handle more than the two outermost dimensions. In general, partitioning a domain according to dependence signatures yields $O(2^{2 \times \text{nbr.deps}})$ parts. This, for a 2D domain represents up to 16 parts per program statement. In practice, a few signatures are impossible to generate for a particular input and tile schedule. However, for higher dimensional domains this number grows fast. A 3D domain could potentially generate up to 64 parts and a 4D domain 256. At this point, the polyhedral code generation could become a bottleneck, especially for complete applications which can have hundreds of statements.
**Input:** Statement level PRDG and tile schedule  
**Output:** Tile level task graph and PRDG with decorated nodes and edges

```plaintext
if (nbr.bands > 1) then
  if (no dependences) then
    Bands fully parallel and independent
  else
    Tile_PRDG ← build 1-dimensional tile-level PRDG
    T ← ConstructTaskGraph (Tile_PRDG)
    T ← SelectTaskIdiom (T)
  end
else
  if (outer dimension is parallel) then
    Band is fully parallel
  else
    if (all dependences are uniform) then
      Tile_PRDG ← build 2-dimensional tile-level PRDG
      T ← ConstructTaskGraph (Tile_PRDG)
      T ← SelectTaskIdiom (T)
    end
  end
end
return {Tile_PRDG,T}
```

**Algorithm 4.4:** High-level algorithm for data-flow task-level parallelization (Part I)

### 4.12 Experimental Results

We now demonstrate the effectiveness of our approach both analytically and experimentally. In Sec. 4.12.1 we present an analysis of certain program properties allowing us to reason on performance. Sec. 4.12.2 describes our experimental setup. We conduct three sets of experiments: in Sec. 4.12.3 we first show the performance achieved by our framework and discuss its relation to the analytical features presented previously; in Sec. 4.12.4 we conduct a workload distribution study for three benchmarks while comparing it to PLuTo’s tiled variants; Sec. 4.12.5 compares our
dynamic-wavefront + partitioning technique to diamond tiling in terms of performance scalability and code features. Finally, Sec 4.12.6 discusses the profitability of tiling and fusion, and provides guidelines for the class of programs that best suit our framework.

4.12.1 Benchmark Properties

We evaluate numerous benchmarks with different reuse and barrier patterns, comparing the performance we obtain using our PPCG/OpenStream-based approach versus a Pluto/OpenMP-based one. It is well known that for certain benchmark types, typically dense linear algebra with $O(n)$ reuse and $O(1)$ barriers such as dgemm, that the Pluto/OpenMP approach is already very effective, implementing the available reuse via tiling and exposing load balancing via a simple OpenMP parallelization. For these benchmarks, our objective is to show that PPCG/OpenStream matches the performance of Pluto/OpenMP, that is we do not suffer performance degradation through our approach. On the other hand, for benchmarks with $O(1)$ data reuse and/or $O(n)$ barriers generated using Pluto/OpenMP, our technique for barrier removal and dynamic fusion has the potential to significantly outperform the Pluto/OpenMP approach, as demonstrated later in this section.

Tab. 4.2 summarizes several of the properties of the benchmarks we evaluate. The DF Par. method shows the type of dataflow parallelism implemented by our framework. The #DF barriers shows the number of “dataflow barriers” which are generated by our framework, and compares to the number of OMP barriers generated by two relevant fusion heuristics implemented in Pluto used in our experiments. Additional code features can be found in [75].
### Table 4.2: Benchmark features (I: Inter-band, DW: Dynamic Wavefront, P: Partitioning; n,t: problem sizes)

<table>
<thead>
<tr>
<th>Configuration</th>
<th>gemm</th>
<th>syrk</th>
<th>2mm</th>
<th>3mm</th>
<th>genver</th>
<th>jacobi-2d</th>
<th>fdtd-2d</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data reuse</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(t)$</td>
<td>$O(t)$</td>
</tr>
<tr>
<td>Tile footprint (KB)</td>
<td>24</td>
<td>24</td>
<td>24</td>
<td>24</td>
<td>8</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>DF Par. Method</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>P+DW</td>
<td>P+DW</td>
</tr>
<tr>
<td>#Minfuse OMP Barriers</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>4</td>
<td>$O(t+n)$</td>
<td>$O(t+n)$</td>
</tr>
<tr>
<td>#Smartfuse OMP Barriers</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>4</td>
<td>$O(t+n)$</td>
<td>$O(t+n)$</td>
</tr>
<tr>
<td>#DF Barriers</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration</th>
<th>blur-robber</th>
<th>seidel-1d</th>
<th>seidel-2d</th>
<th>correlation</th>
<th>covariance</th>
<th>segmentation</th>
<th>ExpCNS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data reuse</td>
<td>$O(1)$</td>
<td>$O(t)$</td>
<td>$O(t)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Tile footprint (KB)</td>
<td>16</td>
<td>8</td>
<td>2</td>
<td>2-8</td>
<td>2-8</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>DF Par. Method</td>
<td>P+I</td>
<td>P+DW</td>
<td>P+DW</td>
<td>P+DW</td>
<td>P+DW</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>#Minfuse OMP Barriers</td>
<td>2</td>
<td>$O(t+n)$</td>
<td>$O(t+n)$</td>
<td>$O(t+n)$</td>
<td>$O(t+n)$</td>
<td>14</td>
<td>7</td>
</tr>
<tr>
<td>#Smartfuse OMP Barriers</td>
<td>$O(n)$</td>
<td>$O(t+n)$</td>
<td>$O(t+n)$</td>
<td>$O(t+n)$</td>
<td>$O(t+n)$</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>#DF Barriers</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>4</td>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>

#### 4.12.2 Experimental Methodology

The machines used for the experiments as well as the benchmarks are described in Tab. 4.4. We test our approach on a subset of Polybench-3.2 [98] and two applications. The compilers used were GCC 4.8.1, ICC 2013-update5, PoCC-1.3. Our framework was built over PPCG [124]; experiments were performed with two OpenStream versions, the public version [96] and one in development for trace capabilities. The FLOPS for each benchmark are computed statically.

We report the performance (GFLOPS/sec) on double-precision floating point for all Polybench benchmarks and ExpCNS, and single-precision for segmentation. All available cores are used. For each benchmark we report baseline performance on GCC and ICC (-O3 -ffast-math for GCC and -O3 -parallel -xhost for ICC), PLuTo’s best performance between tile with minfuse and tile with smartfuse compiled with
both ICC and GCC (generated using PoCC’s parallel, minfuse/smartfuse, prevector, pragmatizer and vectorizer flags, compiled with -O3 -openmp -xhost), and the performance of two OpenStream variants: one, Task, exclusively compiled with OpenStream’s modified GCC (v4.7.0) and the second one, Task opt exporting the task bodies into separate compilation units which are (1) processed by PoCC using pragmatizer, vectorizer and past-hoist-lb flags to have similar intra-tile loop order as with Pluto, and (2) compiled with Intel ICC -O3 -xhost for further optimization, and then linked with OpenStream’s GCC modified compiler.

<table>
<thead>
<tr>
<th></th>
<th>AMD Opteron 6274</th>
<th>Intel Xeon E5-2650 v2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Freq</td>
<td>2.2 GHz</td>
<td>2.6 GHz</td>
</tr>
<tr>
<td>Cores</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>L1</td>
<td>16 KB</td>
<td>32 KB</td>
</tr>
<tr>
<td>L2</td>
<td>8 x 2 MB</td>
<td>256 KB</td>
</tr>
<tr>
<td>L3</td>
<td>6 MB</td>
<td>20 MB</td>
</tr>
<tr>
<td>RAM</td>
<td>32 GB</td>
<td>16 GB</td>
</tr>
</tbody>
</table>

Table 4.3: Experimental testbed

4.12.3 Performance Results

We remark that the achievable performance could vary significantly between GCC and ICC, for the same code. This is why we report numbers not only for OpenStream’s native compiler (GCC), but also for (task-opt), a hybrid compilation scheme relying on ICC for task bodies and OpenStream’s GCC for task creation and scheduling. GCC does not always implement as many optimizations as ICC for these numerical codes, implementing less effective automatic vectorization and less optimizations in
<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Category</th>
<th>Problem Size</th>
<th>Tile Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>2mm, 3mm, gemm, syrk</td>
<td>linear algebra</td>
<td>2000^3</td>
<td>32</td>
</tr>
<tr>
<td>gemver</td>
<td>linear algebra</td>
<td>8000^2</td>
<td>32</td>
</tr>
<tr>
<td>correlation, covariance</td>
<td>data mining</td>
<td>2000^2</td>
<td>32</td>
</tr>
<tr>
<td>seidel-1d</td>
<td>stencils</td>
<td>200000^2</td>
<td>1024</td>
</tr>
<tr>
<td>blur-Roberts</td>
<td>image processing</td>
<td>4000^2</td>
<td>32</td>
</tr>
<tr>
<td>fdtd-2d, jacobi-2d, seidel-2d</td>
<td>stencils</td>
<td>2000^3</td>
<td>16</td>
</tr>
<tr>
<td>segmentation</td>
<td>3D MRI processing</td>
<td>760^2 × 130</td>
<td>4x4x32</td>
</tr>
<tr>
<td>ExpCNS</td>
<td>DoE mini-app</td>
<td>32^3</td>
<td>4x4x32</td>
</tr>
</tbody>
</table>

Table 4.4: Benchmark description

the low-level generated code. In addition, the intra-tile (i.e., task) loop order with task-opt is subsequently optimized to match Pluto’s generated intra-tile loop order, leading to additional performance improvements.

**Linear algebra kernels**

2mm, 3mm, gemm, syrk and gemver fall into this group. Fig. 4.9 shows that we achieve performance comparable to PLuTo for the variants codes with O(n) reuse. These codes are already balanced in their OpenMP/PLuTo variant, and tiling achieves the O(n) reuse independently of the parallelization method. Concurrent start is possible with PLuTo’s barrier synchronization method as well as our inter-band. PLuTo is expected to perform very well on these codes, and there is no significant gain to expect with our task-based parallelization scheme.

Regarding gemver, the relative workload per barrier is much lower than for the other codes, as illustrated by the O(1) reuse, exacerbating the cost of barriers. In both Pluto and PPCG generated codes, there is one inner loop with high-stride memory access. Still, there is some reuse of data between phases of the computation, this reuse
Figure 4.9: Performance on AMD Opteron (top) and Intel Xeon (bottom) of linear algebra kernels

is fully implemented through OpenStream’s default task’s firing policy. Through trace analysis it was determined that for this benchmark, task execution followed a zig-zag pattern between tasks of adjacent bands, i.e., the first bands executes tasks (mostly) in ascending order, the second band in descending order, the third, again in ascending
order, and so on. This also enables dynamic fusion by reusing the data of the last task.

We also note that kernel 3mm exhibits an unusually high baseline. Further inspection revealed that ICC was able to pattern-match matrix-multiply kernels in the input code, replacing all three loop nests in 3mm by MKL calls, and one of the two matrix-multiply of 2mm by an MKL call. ICC was however unable to recognize and pattern-match the gemm kernel nor its occurrence in 2mm, meaning the only two kernels for which ICC was able to use MKL instead of the original code were 2mm (in part) and 3mm (in full). Regarding syrk, ICC was unable to vectorize PLuTo variants, while partially vectorizing the reference code. syrk \textit{task-opt} was vector-array scalarized, but allowing ICC to decide whether it is vectorizable or not (i.e., no SIMD/vector pragmas were inserted). Finally, no significant differences were observed between PLuTo’s fusion heuristics.

\textbf{Stencil kernels}

Here we have jacobi-2d, ftdt-2d, seidel[1-2]d and blur-roberts. All five kernels update neighboring points on a dense grid. blur-roberts performs a single sweep on the image, whereas the other are iterative stencils that are repeated \( T \) time steps, thereby showing a higher order of data reuse. These first four iterative stencils can benefit from time tiling + wavefront parallelism and show \( O(t) \) reuse, while only standard tiling is meaningful for blur-roberts. The iterative stencils have \( O(t+n) \) barriers when parallelized with PLuTo (the exact value depends on the selected tile sizes as well as the skewing factors).

The number of barriers with PLuTo could be reduced by selecting larger tile sizes, but there is a limit on the maximal tile size to preserve L1 data locality and
Figure 4.10: Performance on AMD Opteron (top) and Intel Xeon (bottom) of stencil kernels

it would also decrease the parallelism. In contrast, our framework leverages point-to-point dependences to address load imbalance, and tile sizes can be selected in a more independent fashion. In particular, smaller tile sizes benefit our framework by decreasing the task footprints, and increasing the possibility of further temporal
reuse via dynamic fusion, whereas such option will simply add overhead to static wavefronts.

Fig. 4.10 shows that our partitioning technique combined with dynamic wavefront can achieve speedups ranging from $1.1 \times$ to above $3 \times$. Moreover, for these benchmarks the performance gap between task and task-opt is negligible or favors GCC, i.e. ICC does not provide additional benefit and most performance gains are due to increased parallelism and barrier removal.

Finally, we note that PLuTo produces a 3-dimensional wavefront for jacobi-2d, fdtd-2d and seidel-2d whereas our approach exposes a 2-dimensional wavefront.

Regarding blur-roberts, partitioning allows to pipeline tasks between two bands. The reuse order is $O(1)$ and the reuse distance 1 (one stream is induced by a flow dependence). OpenStream’s firing policy favors immediate triggering of tasks, enabling dynamic fusion. PLuTo variants trade-off locality and parallelism: minfuse has 2 barrier synchronizations and achieves the steady state in $O(1)$ time, but has poor inter-band locality, whereas smartfuse enhances locality but strongly diminishes parallelism by inducing a wavefront-like parallelism ($O(n)$ barrier synchronizations). Furthermore, smartfuse also affects negatively the vectorizability of the code.

Applications

Fig. 4.11 show our results for correlation, covariance, segmentation and expcns.

**Correlation and covariance** Both applications show, as for gemver, a low ratio of workload per barrier for the most part of the application, increasing the gain in removing barriers. The $O(n)$ reuse happens in the next to last band on both applications. This avoids a potential early data flush that could hinder performance.
We also note that even when not removing all barriers (See Tab.4.2), our barriers are data-flow point-to-point barriers, e.g., tasks only wait on their specific input streams and a single task can wait on more than one data-flow barrier.
Segmentation is one of the five stages (the most time-consuming one) of the Computer-Aided Diagnostic pipeline for lung cancer screening on 3D MRI images developed by the Center for Domain-Specific Computing. It performs 3D image segmentation, using an iterative solver that typically converges in two time iterations. Only the spatial loops have a static control flow, so we limit to optimizing a single iteration of the algorithm. We evaluated multiple versions of PLuTO and different tile sizes. Tiled and untiled minfuse variants outperform their smartfuse counterparts. PLuTo minfuse outperforms inter-band by approx. 20%. This is due to the large program footprint, the DAG’s shape and the dependence distance. Essentially, this program can be divided into 2 stages: the first one, with a high number of parallel bands that have dependence distance of 1 (immediate inter-task reuse), and the second which consists of a more “ordered” stage, wherein most bands require the result of one or more of the previous bands. The tile sizes selected in the space allow for a fair number of task footprints to fit in L1. However, the ordered stage induces a complete flushing of data, killing locality. It is expected a better performance could be achieved with our framework if we had some control on the task scheduling with OpenStream.

Exp_CNS_NoSpec is a mini-application to integrate the Compressible Navier Stokes (CNS) equations, from the DoE Exact center (Center for Exascale Simulation of Combustion in Turbulence). It is written primarily in Fortran. Various C-functions are provided to replace the native hypterm and diffterm routines, which amount to 90% of the execution time. Three SCoPs are extracted from the array-based variant.
ifort (ICC v13) and gfortran (GCC 4.8.1) compiler settings were used. The default dataset was used, performing 100 time iterations on a $32^3$ grid.

Full dynamic fusion through inter-band parallelism is not achieved due to long dependence chains, but it is still able to exploit a better balance between parallelism and locality than PLuTo variants. PLuTo minfuse experiences a lack of locality, whereas smartfuse loses in terms of parallelism (complex loop structure and parallel loops at depth 1 w.r.t. to the SCoP’s root). PLuTo untiled variants were also evaluated, yielding an average 50% slowdown. An extended study of segmentation and expens can be found in [75].

4.12.4 Workload Distribution

We now analyze the degree of parallelism and load balance achieved by our method enhanced with further ICC optimization (task-opt) and contrast it to PLuTo’s heuristics, minfuse and smartfuse, which are parallelized with OpenMP’s for work-sharing construct with default scheduling policy. We focus the analysis to the AMD Opteron platform, and select 3mm, seidel-2d, covariance and gemver as case studies. For each benchmark, we decompose its execution time into 3 parts: the sequential time, the parallel balanced time and the parallel unbalanced time. The breakdown of the execution time has been obtained through the following methodology:

- we use a version of OpenStream capable of generating traces. In particular, we consider the time spent in task creation (serial), task initialization, task execution and task seeking states, of which the last three are total time across all cores. The breakdown shown was converted from cycles to seconds considering the number of executing cores and their frequency. We consider the task
execution time as parallel balanced, and the initialization and seeking times as unbalanced.

- PLuTo’s variants are compiled with ICC v11, using the same flags as in the previous section, with the exception of -openmp which is replaced by -openmp-profile. The output obtained includes sequential time and per core minimum, average and maximum parallel time, both balanced and unbalanced. Here we define $time^{unbalanced}$ as $\max(time^{unbalanced}_{core, id})$ and $time^{balanced}$ as $\max(time^{parallel}_{core, id}) - time^{unbalanced}$.

Although we use different compiler versions for this set of experiments than in the previous section, we note that no significant difference was observed in the performance.$^3$

3mm A dense linear algebra kernel such as this is not expected to benefit much from the data-flow parallel model; minfuse and smartfuse produce 6 tiled loop nests, but with different loop structure. In both cases, these are mapped to 6 parallel regions. In contrast, the inherent task parallelism is limited to the initializations of each of the 3 product matrices, followed by the computation of the left and right side matrices required for the final product. Figure 4.12 shows that ref icc and PLuTo’s variants saturate at 8 cores. This is due to the $8 \times 2$MB distributed structure of Opteron’s L2 cache, which forces many threads to access data from a distant core. Variant task, being fully compiled with GCC, is compute bound, but after further optimizing the task bodies with ICC (see the task opt graph) this is no longer the case, achieving linear scaling with the number of cores. Regarding the impact of barrier removal,

$^3$The OpenMP profiling option has disappeared from recent versions of ICC, hence the use of v11 in this specific experiment.
we show 3mm’s time breakdown in Figure 4.13. The performance achieved is slightly superior to both of PLuTO’s. *task opt* suffers from load imbalance due to initialization and task creation overhead, as these steps run on a single core. This phenomenon is visible in most of our experiments. The fraction of the unbalanced execution time for OpenStream represents approx. 8% on 8 cores and 17% on 16 cores, whereas for PLuTo it varies between 15% and 50%.

**Seidel-2d** This kernel becomes highly unbalanced when applying PLuTo’s tiling heuristics combined with the (static) wavefront technique. It consists of a single
statement and both tiling heuristics produce the same fusion structure. Fig. 4.14 shows the lack of scalability of this benchmark when compiled with GCC and ICC. PLuTo minfuse, on the other hand, scales well with GCC, but saturates at 8 cores with ICC. Further inspection revealed that task variant incurred in 50% less L1 accesses when scaling from 8 to 16 cores, whereas minfuse-ICC experienced a 15% reduction and minfuse-GCC only a 10%. L1 misses also reduced by 50% for task variant, 23% for minfuse-ICC and minfuse-GCC remained the same. Regarding the L2 cache behavior, task variant showed 25% less accesses than minfuse for ICC and GCC for 8 cores. For 16 cores, the L2 accesses difference between task variant and minfuse variant varied between 4% and 6%. L2 misses also reduced by 54% for task variant from 8 to 16 cores, but only 13% for minfuse-ICC and 30% for minfuse-GCC. When using all 16 cores the 3 variants exhibit the same number of L2 misses. Fig. 4.15 shows that the
Figure 4.14: Performance Scalability for kernel seidel-2d

unbalanced fraction of time for minfuse-ICC is approx. 50% on 8 and 16 cores. Task variant (compiled only with GCC) spends 5% of its time in an unbalanced state for 8 cores and it increases to 10% on 16, while the balanced time is reduced to half as there are twice more cores for computing.

Covariance  PLuTo’s heuristics and parallelization method limit the potential performance of this kernel. Both smartfuse and minfuse heuristics yield 7 parallel regions. While PLuTo variants saturate at 8 cores (See Figure 4.16) and reach a maximum of $4\times$ speedup, task-opt continues scaling, although at a lower rate, achieving a final speedup of $7\times$ w.r.t. single-threaded execution time. Figure 4.17 shows that approximately 96% of the total time is spent in unbalanced parallel execution with
PLuTo-generated variants, while this is reduced by a factor of 4 with task-parallel execution.

4.12.5 Dynamic Wavefront vs. Diamond Tiling

We now compare our dynamic wavefront technique to diamond tiling [8], which aims at allowing concurrent start and improving the load imbalance in stencil computations that are parallelized with doall/barriers. It allows concurrent start along (at least) one of the faces of the iteration domain. It assumes that all tile dependences (in the transformed space) are unit vectors, thus similar to our usage of uniformizing dependences prior to the domain partitioning stage. A sufficient condition for diamond tiling is that the tile schedule must have the same direction as a face of the iteration domain.

Figure 4.15: Time breakdown of seidel-2d kernel.
We compare the scalability properties as well as the code characteristics of both approaches using the jacobi-2d kernel, a 5-pt stencil which meets diamond tiling’s requirements. Diamond tiling variants were generated with PLuTo v.0.10, using flags –partlbtile (1-dimensional load balance tiling) and –parallel for OpenMP pragmatization, considering the default fusion heuristic (smartfuse). Experiments were conducted on the AMD Opteron and Intel Xeon (See Tab.4.4). The diamond tile variants were compiled with GCC 4.8 using flags -O3 -fopenmp and -ffast-math. We use the same tile sizes as the experiments reported in [8].

On the performance aspect, our dynamic wavefront outperforms diamond tiling on the two machines (Fig. 4.18). Both techniques scale very well on the Intel processor. However, as in the previous experiments, diamond tiling does not scale beyond 8 cores on the Opteron, whereas the dynamic wavefront achieves a speedup of 11× w.r.t to
its sequential performance. As the Opteron has an L1 data cache of 16 KB, a tile of $32 \times 32$ on double precision and two arrays occupies most of it. Increasing the tile size to 64 does not improve performance for the task variant, but represents a 3 GF/s increase for diamond tiling. On Intel Xeon we see a similar behavior, but without the Opteron’s plateauing, and both techniques scale almost linearly. In general, our task variants will benefit more from having tasks with tiles that are slightly smaller than the L1 cache.

Regarding the code structure, diamond tiling’s code still suffers from a large number of barriers due to having a doall loop nested within a sequential loop. Furthermore, [8] also states that, in practice, partial concurrent execution (only the outermost dimension) yields better performance than exploiting parallelism across all faces of the iteration domain due to code complexity, manifested as code explosion and numerous
modulo conditions along the code (about 8000 lines and 400 modulo conditions for full dimensional concurrent start, but about 900 lines and 7 modulo conditions when using the 1-dimensional option). Two more features hinder the performance of this
technique: conditions nested within the innermost loops and the tile shape. Both inhibit vectorization in many cases. On the contrary, partitioning for generating dynamic wavefronts produces a more compact code, about 160 lines for the same kernel and the modulo conditions appear mostly on the outermost dimensions.

Finally, we note that diamond tiling is not applicable to kernels such as seidel-2d, because of the nature of its dependences on the tiled (transformed) program. On the other hand, our framework seamlessly handles these kernels, as long as the processed tile dependences used for partitioning are uniform.

4.12.6 The Role of Tiling and Loop Fusion

While not all affine programs are tilable, a strength of the polyhedral compilation framework is to dramatically increase the applicability of tiling by automatically computing a program transformation to make the code tilable [24]. In our framework, we choose tiling as a mechanism to expose atomic tasks with a variable granularity (tile size), thereby allowing to amortize the runtime overhead and better control data locality opportunities by enforcing a partial order on the operations. We remark that while tiling is not needed for programs without significant data reuse potential, it is not a detrimental transformation either, as long as the changes needed to create a tiled implementation do not prevent optimizations that were possible on the original code, such as prefetching or SIMD vectorization.

Loop fusion is often considered in conjunction with tiling, to increase data locality and create tile bodies containing more statements. Excessive fusion can however be detrimental to performance as it may saturate prefetch streams or prevent SIMD vectorization by reducing the dependence distance. Complementary transformations
after tiling to restore SIMD vectorization capabilities, e.g. [78], were not considered in this work, nor the exploration of different coarse-grain fusion/distribution structures to form tiles [101]. As future work we will consider evaluating the impact of these task creation methods.

In this work, we restrict our analysis to tiled variants only; the complete study can be found in [75]. To conclude, the metrics displayed in Tab. 4.2 help understanding the suitability of a program for our framework, mostly to distinguish between already-efficient OpenMP implementations (large data reuse available, load balance achieved, limited number of barriers and large workload per barrier); and unbalanced/oversynchronized programs with or without data locality potential. We remark that our work precisely exposes data locality opportunities between tiles, an information that could be used to compute an affinity for the tasks. Unfortunately, the current OpenStream implementation we used does not allow to provide scheduling guidelines and therefore locality opportunities were often lost due to the task firing policy of OpenStream. It is expected that the performance of codes generated with our framework would improve further when using such affinity information.

4.13 Related Work

A number of runtimes have been proposed for dynamic, task parallelism with inter-task dependences. In particular, the DAGuE runtime [25], the U. of Delaware codelet model [114] and the related SWARM environment [43], and the T* runtime of OpenStream [96] are all “feed-forward” or “argument fetch” data-driven models. In such runtimes, tasks notify their successors upon completion. While this increases the programming or compilation burden—continuations may have to be exposed and
passed explicitly, the dependence resolution algorithm can be fully distributed and its complexity is generally independent of the number of blocked tasks [25, 96]. This contrasts with a family of runtime systems where a dynamic dependence resolver identifies the ready tasks, such as the current runtimes supporting the StarSs [93] and CnC [26] languages.

DAGuE is unique in that it uses a symbolic representation of the acyclic dependence graph to avoid building it in extension. This approach is reminiscent of context-based data-flow execution and architectures, where the iteration space structure is directly embedded into the data-driven execution mechanics [126, 81]. While such symbolic approaches are more (local-) memory-efficient than explicit graph representations à la Star-PU, the exact benefits remain largely unknown compared to the fully dynamic dependence graphs of SWARM or T* (OpenStream). Nevertheless, the evaluation is orthogonal to our work which can make use from both approaches.

Our technique relies heavily on the partitioning of the iteration domain of a loop nest. Griebl et al. proposed the technique known as Index Set Splitting (ISS) [53] and Pugh and Rosser proposed Iteration Set Slicing (also ISS) [105]. Both techniques partition iteration domains based on dependence patterns (e.g., when a dependence changes direction, or based on transitive closure computations). Their technique is applied as a pre-processing step that allows to extract more parallelism, or to apply more aggressive affine transformations. We devised a form of ISS that operates—on tile dependences—performed once the schedule has been computed.

[12] develops a framework to extract tile-level task-parallelism but with full dynamic dependence resolution. Diamond tiling [8] and split tiling [61] aim at reducing the load imbalance derived from static wavefronts, i.e., the ramp up/ramp down.
However, the wavefront itself is only the first part of the problem. The second part is the over-synchronization induced by barriers, and in that sense, both techniques still suffer from this issue. Furthermore, split tiling’s two phase execution (per dimension) increases the number of synchronizations exponentially in the number of dimensions, which are more expensive in non-uniform architectures. To conclude, our inter-band approach could also be combined with split-tiling to remove unnecessary synchronizations in between phases.
Chapter 3 and 4 introduced novel techniques useful for transforming and optimizing programs, exposing and maximizing parallelism, the first via a transformation framework geared towards maximizing vectorizability opportunities, while the second targets the removal of barriers, thereby exposing hidden parallelism and mapping it to a lightweight streaming run-time.

In each chapter we have taken one step further towards bigger execution units, first at the single core-level within a multiprocessor. Then, among tasks executing on shared-memory machines. We now introduce a novel language designed specifically for generating both shared-memory and distributed-memory programs from a data-flow representation.

In the following sections we describe the main constructs of PIPES, a polyhedral data-flow language and compiler, as well as many of its advanced features.

5.1 High-Level Macro Data-Flow Programming

From a user point of view, two elements are needed: (1) a PIPES program describing at least the task-level dataflow graph, and optionally primitives instructing about the distribution strategy; and (2) an implementation of the actual task computation,
for each task type. From this the compiler takes care of automatically generating all get/put of data before/after the task body method is called, generating the graph using Intel CnC primitives (e.g., each and every prescription and dataflow relations), and all tuners associated with the PIPES primitives. We have also provided an API to abstract away certain specifics of the runtime, with the future goal of targeting other event-driven runtimes such as the Open Community Runtime [1].

Sbirlea et al. [108] previously demonstrated the feasibility of a compiler from arbitrary DFGR programs to the Habanero-C task-parallel language [9]. Although PIPES does not share code with their project, similar ideas are used to generate code for general programs. But PIPES has a particular focus on enabling and applying polyhedral optimization on programs, and has been designed to generate *optimized code* when dataflow relations and tag sets can be modeled using the polyhedral framework. A dedicated path in the compiler therefore exists, as shown in Figure 7.1, for the “affine” parts, and is a unique feature of the PIPES framework.

Programs in our framework are specified in a high-level textual form. PIPES programs directly extend DFGR programs (the Data-Flow Graph Representation, proposed by Sbirlea et al. [108]), in that all DFGR programs are PIPES programs. The main concepts of DFGR to model task-level dataflow are recalled in Sec. 5.2. DFGR excels at representing the data and control-flow relations between tasks. But implementing a parallel *algorithm* requires to specify certain placement, communication and scheduling strategies. We present our new extensions to DFGR with numerous constructs useful for these specifications: processor topology, task and data placement, explicit communication primitives, and scheduling hints in Sec. 5.3–5.7, forming the PIPES language.
5.2 Data-Flow Graph Representation

Listing 5.1 shows a DFGR representation of a matrix-multiplication algorithm. A program in DFGR contains three core components: (1) the declaration of item collections, which are parallel containers holding data; (2) a description of the data read and written by task instances; (3) a description of the connection between the DFGR graph and the “outside world” (that is, the environment, or calling context) which is represented by the special env task.

Listing 5.1: Matrix Multiplication

```plaintext
// Parameters used to pass to context
Parameter N : 1..100000;

// Define block collections
[AA :0..N-1,0..N-1] float;
[BB :0..N-1,0..N-1] float;
[CC :0..N-1,0..N,0..N-1] float;

// Task prescriptions
env :: (MMC:0..N-1,0..N-1,0..N-1);

data-flow
env -> [AA :0..N-1,0..N-1];
env -> [BB :0..N-1,0..N-1];
env -> [CC :0..N-1,0..N-1,0];
[AA :i,k], [BB :k,j], [CC :i,j,k] -> (MMC:i,j,k) -> [CC :i,j,k+1];
[CC :0..N-1,0..N-1,N] -> env;
```

We first recall some fundamental aspects of DFGR [108]. Dynamic instances (data, operation) are uniquely identified by a tag value (e.g., (1,1)). Tag values are computed from tag functions (e.g., (i,j)) which can be arbitrary functions, including dependent on dynamic data value and/or known only at run-time. Tag functions must produce integer tuples. Item collections are arbitrary data structures, such that each entry in the collection is uniquely identified by a tag value. Hashmaps and arrays are two typical ways to implement item collections. Each tag in an item collection must be written exactly once: the program models a dynamic single assignment form. If a tag is read in a collection before being written the graph stalls; if it is written
twice the runtime asserts an error as the DSA is violated. Each step instance must be
prescribed before it can be executed. Furthermore, it will execute only when the data
it reads has been made available. The runtime takes care of starting step instances
when their data is available. The environment must write to the item collections
all the data that is input to the computation explicitly. Conversely the useful data
produced by the computation must be written back to the environment. A graph
starts as soon as some step is prescribed and its data is available; a graph terminates
as soon as all the data written to the environment has been produced.

Steps in DFGR are functional (e.g., stateless), so data accessed by steps is stored
separately. All resources internal to the step are cleared when the step instance fin-
ishes. Data read by a step may come from any source, but any tag value in a collection
is only written once. Graph items are therefore implementing Dynamic Single As-
signment (DSA) and are never collected, i.e. they are persistent. Optimizations such
as folding the data space when an ordering is imposed or setting a fixed number of
reads per item can allow items to be collected for better storage management, as dis-
cussed later. DFGR also permits arbitrary access to data outside of the graph: such
global data can be used to create a state without using an item collection. However
using global data limits the outcome of graph analysis and transformations, since this
non-graph data is not represented nor analyzed in the macro-dataflow graph.

Returning to Listing 5.1, we can now demangle the code, starting by the environ-
ment. Line 8 represents a prescription of $N \times N \times N$ instances of the step MMC. The tag
function associated to MMC is a 3-uple, each component of the tuple in the prescrip-
tion is a range, from 0 to $N - 1$. That is when using a symbol (e.g., $i$) to refer to
this component in the graph, $i$ will range from 0 to $N - 1$. Line 10–12 declare the
production by the environment of elements to the item collections \textit{AA}, \textit{BB}, \textit{CC} that is the input data. The data type for elements in this collection is \texttt{float}, each item in a collection is a tile of data (using a linearized array in this case). Lines 13–14 describes the data being read/written by step instances, using tag functions. Instance \((i, j, k)\) of the step \textit{MMC} will read item \((i, k)\) from \textit{AA}, and write item \((i, j, k + 1)\) to \textit{CC}. Indeed, as DFGR enforces that items in a collection (uniquely identified by the value of their tag) are written only once, the reduction for matrix-multiplication is modeled using a third index, \(k\), for the output matrix \textit{CC}.

**Conceptual hierarchy for tag sets**  DFGR reconciles static and dynamic macro-dataflow modeling by allowing the explicit and unique naming of each step instance and each item in a data collection. It uses a hierarchy of concepts that can be used to represent sets of tags, ranging from the simplest form of integer ranges to the most complex form of arbitrary sets. This hierarchy is (1) ranges; (2) simple polyhedron; (3) union of \(\mathbb{Z}\)-polyhedra; (4) union of arbitrary sets. In a nutshell, ranges will model rectangles and are well suited for simple, regular computations. Polyhedra, which are described using affine inequalities, enable powerful static analysis and transformation of the graph based on the affine framework [97] [124] [94]. Unions of \(\mathbb{Z}\)-polyhedra are a generalization of polyhedra, allowing for more complex geometric shapes to be modeled by using union of convex sets, and regular strides in the set are supported using affine lattices [56]. Modern polyhedral compilation frameworks fully support the complete analysis of programs described using such polyhedra, and we extensively rely on these frameworks in PIPES as shown in later. Finally, arbitrary sets are sets not belonging to any of the previous category, typically used when the set cannot
be described using strides and affine inequalities; or when the set is described using functions whose result is not statically known.

We now describe the extensions we provide in PIPES to enable the high-level description of a parallel execution strategy of DFGR programs.

### 5.3 Virtual Topology

Virtual topologies are the first element needed to easily describe placement and communication strategies for a program. PIPES supports both pre-defined topologies and user-specified ones. A PIPES virtual topology, as for instance a 2D grid of size \( P \times P \) as shown in Listing 5.2, is mapped to the MPI processor topology of the target machine using the API we designed wrapping the Intel CnC distributed runtime’s API. We remark that a topology in PIPES is essentially aimed at providing a convenient way for the user to index a particular processor when describing placement/communications, and do not require to correspond to the actual underlying physical topology.

**Listing 5.2:** A simple flat 2D topology

```plaintext
1 // 2D topology, no more than 256x256 processors
2 Parameter P : 4..256;
3 Topology TOPO = {
4   sizes=[P,P];
5   cores=[i,j] : { 0 <= i < P, 0 <= j < P};
6};
```

Topologies in PIPES can be hierarchical, that is one can represent explicitly the organization of nodes, then processors in a node, then cores in a processor. This is reminiscent to the work on hierarchical affinity groups by Knobe et al. [73]. More advanced topologies can be defined, including explicit listing of communication links,
however it remains the duty of the user to provide a mapping from the proposed processor indexing to the underlying MPI processors: we currently only provide default implementations for 1D, 2D and 3D meshes in PIPES.

5.4 Placement of Tasks and Data

Equipped with topologies, we can now provide constructs to explicitly place tasks. Listing 5.3 expands the matrix multiplication example using the topology defined in Listing 5.2, shows how to place tasks in a block-cyclic fashion (block size of 8) onto this 2D processor grid. This construct is directly translated to the `compute_on` tuner in Intel CnC, which can be defined for each `task,tag` pair and returns the processor identifier executing this task instance. The cyclic mapping is performed as part of the virtual-to-physical topology mapping.

Listing 5.3: Mapping task instances to processors

```
1 // Place the N steps (i,j,*) to Proc((i/8),(j/8))
2 (MMC:i,j,0..N-1) -> [[TOPO:(i/8),(j/8)]];
```

In PIPES, there is no primitive to explicit place data: this placement is inferred by the producer and consumer relations of the graph, and by the result of communications primitives as shown below. Indeed, for a task to execute on a processor $p$, its input data must have been transferred to $p$, and immediately after completion its output data will be on $p$. In other words, the implicit first owner of a data is its producing step instance.

5.5 Data communication

Explicit communication primitives are an essential addition in PIPES compared to previous work on CnC tuning. When specifying a parallel algorithm, users typically
write *when* a communication occurs and *what* data goes *where*. Using only a dataflow graph to specify such communication is tedious: one needs to create a new “fake” task in the graph that reads the data to move, and add “fake” dependences between this task and others to impose the desired scheduling. To hide this complexity, and allow for more efficient and customized implementations of this operation, we provide in PIPES a construct for item communication that uses tasks as referentials. An example is given in Listing 5.4, with the two communication primitives enabling to model the Cannon algorithm. Line 2 specifies the data \([\text{AA}:i,k,j]\) located on the processor where task \((\text{MMC}:i,j,k)\) is executed and which is to be sent (put) on the processor where task \((\text{MMC}:i,\text{SHIFTA}(j),k+1)\) is to be executed, immediately after task \((\text{MMC}:i,j,k)\) completed. The \text{SHIFTA} function in the destination tuple uses a parameterized region which performs the circular shift along the i-dimension.

**Listing 5.4:** Communication of data

```plaintext
1 // Region definitions for circular shifts require domain knowledge
2 \text{SHIFTA}(j) := \{
3 \quad [jj]: j - 1 >= 0 \and jj = j - 1;
4 \quad [jj]: j - 1 < 0 \and jj = N
5 \};
6 \text{SHIFTB}(i) := \{
7 \quad [ii]: i - 1 >= 0 \and ii = i - 1;
8 \quad [ii]: i - 1 < 0 \and ii = N
9 \};
10 // Circular communication pattern for Cannon algorithm
11 [\text{AA}:i,k,j]@((\text{MMC}:i,j,k) => (\text{MMC}:i,\text{SHIFTA}(j),k+1));
12 [\text{BB}:k,j,i]@((\text{MMC}:i,j,k) => (\text{MMC}:\text{SHIFTB}(i),j,k+1));
```

We chose to use a task instance (e.g., the task uniquely identified by \(\text{MMC}:i,j,k\)) as the referential in terms of scheduling and data location for the primitive. The motivation is twofold. First, forcing a communication to happen immediately before or immediately after a task executes means no additional scheduling constraint is needed in the program: we simply generate the associated **Get** or **Put** CnC command.
right before/after the user-defined task computation method is called. While this somewhat restricts what communication schedules can be implemented, it is sufficient for many practical uses and as shown below more advanced scheduling primitives are provided in PIPES if needed by the user. Second, using the task location as the referential for the source and target data placement allows to maintain the paradigm described in Sec. 5.4. However tasks must have been placed for the communication to happen in a point-to-point fashion, as otherwise if the expected location of a task cannot be determined at compile-time then our compiler must instead rely on the runtime to determine where this data is, possibly requiring broadcasts to all processors.

5.6 Scheduling Constraints

DFGR enables the explicit description of control dependences between task instances. That is, one can enforce a (partial) scheduling of tasks without using producer-consumer dependence via item collections. Listing 5.5 shows an example of such dependence between a source task and a set of tasks. This concept is supported in PIPES, and all such ordering constraints are necessarily enforced at run-time.

Listing 5.5: Task-to-task dependence

```plaintext
1 (MMC : i,j,k) => (MMC : i,j,k+1..N-1);
```

5.7 Scheduling Hints

Scheduling constraints (coming from producer-consumer or task-to-task dependences) impose an order on the execution. The concurrent collection philosophy seeks for a minimal set of such dependences being described, leaving to the tuning expert the task of finding an efficient actual schedule. Following this idea, we provide
in PIPES several constructs to represent scheduling *hints*, that is a desired order for the computation but which may not be implemented in practice by the runtime, especially if real dependences prevent this schedule to be respected. *PIPES scheduling hint cannot lead to violating real dataflow/control dependences* in the PIPES program.

Listing 5.6 shows a first example of scheduling hint. We show how to define a tag set (in this case, a simple polyhedron) and use it for the environment to prescribe \( N \times N \) instances of the task \( T \). When translating this into Intel CnC C++ code, the code scanning the tag set is generated using the polyhedral code generator CLooG [14] and by default it scans the set in lexicographic ordering. But one can specify an *order* clause which changes the order in which the set is scanned, in turn changing the order in which steps will be prescribed in this example. A traversal order can be associated with any range/polyhedron/\( \mathbb{Z} \)-polyhedra tag set.

**Listing 5.6:** Scanning tag sets in specific order

```
// Define a set with a traversal order
R := {(i,j) : 0 <= i < N, 0 <= j < N} order {[j,i]};
// Prescribe task (T:i,j), "order" property used
// for scanning the region at codegen time
env :: (T : R);
// output:
  // for (j = 0; j < N; j++)
  //   for (i = 0; i < N; i++)
  //     context.put(T, Tag(i,j));
```

Listing 5.7 shows another way to specify a scheduling hint, in a flavor closer to a task-to-task dependence. In this case, a partial order is defined between tags, and the runtime will attempt to follow this order if possible.4

**Listing 5.7:** Scheduling hint between tasks

```
// Try to order tasks serially along j
(MMC : i,j,k) ~> (MMC : i,j+1,k);
```

4The exact implementation of this primitive requires adapting the internal scheduler of Intel CnC, and is not currently supported in our prototype implementation.
In the previous chapter we introduced PIPES, a new data-flow language inspired by DFGR [108], its basic constructs as well as its advanced features such as block and task placement. In this chapter we proceed to discuss the major algorithmic details of the compiler, namely the transformations used to extract SCoPs from a PIPES input, performing optimization on the task and block (data) space, and leveraging the tuning capabilities of the CnC run-time.

6.1 SCoP Extraction

The first stage of applying polyhedral optimizations in PIPES is to translate a PIPES program (specifically the dataflow graph) into polyhedral concepts, that is (1) iteration domains for each task, properly modeling the dynamic set of executions of task instances; (2) access functions for each task to capture the data accessed by each task instance; and (3) dependence relations between task instances, to enable correct semantics-preserving transformations of the program. In this work we only require the dataflow graph (i.e., the “DFGR” subset of the PIPES language) and the task placement primitives to be modelable using affine relations and polyhedral sets. Previous work by Sbirlea et al. [108] proposed an initial translation of DFGR to the
polyhedral framework. We significantly extend their work by (1) supporting arbitrary
prescription operations; (2) capturing and analyzing task placements; (3) supporting
a wider class of integer sets statically, going beyond simple polyhedra.

We use \(\mathcal{Z}\)-polyhedra to capture sets of integer vectors, and multidimensional affine
functions to represent relations between these vectors. We remark that PIPES can
represent arbitrary (macro-)dataflow programs, therefore not all PIPES graphs can
be represented within the polyhedral framework we propose. The set of PIPES graphs
that can be optimized is driven by the restrictions we impose on each PIPES con-
structs, and the algorithms automatically reject unsupported PIPES program. For
these, no further analysis or optimization is performed, and we default to the default
non-affine translator to Intel CnC C++.

**Task Prescription** Task prescription operations form the (super-)set of all dy-
namic instances of each task. To represent the iteration domain of a task, we
leverage the fact that tag functions are restricted to producing integer tuples. The
preserve operation `::` can be between arbitrary task instances, that is the graph
may contain prescriptions like `(foo: 3*i,j)::(bar: 2*j,i)`; where a particu-
lar task instance \((i,j)\) prescribes another task instance identified by the tag function
\(f(3*i,j) = (2*j,i)\). Sets of instances may be produced, using ranges or regions [108].
To correctly model the iteration domain of `bar` in this example we need to analyze
the tag functions of both operands of the prescribe operation, as well as being able
to model the iteration domain of the left-hand side operand.

Our algorithm to compute iteration domains for tasks is as follows. It basically
iterates over all prescribe operations, propagating the left hand side iteration domain
to the right hand side if and only if the relationship between tag functions is a multidimensional affine function and if the iteration domain of the left hand side is computable and known. In addition, when regions (or ranges as 1..M above) are used to model sets of tags we require the region to be itself describing a union of Z-polyhedra, and the prescriber to have a single iteration. Affineness of a region is determined by simply ensuring all inequalities in the region are affine expressions and free of uninterpreted functions. We note \texttt{op.lhs.tf} for the tag function of the left hand side of the operator \texttt{op} (e.g., \((2*j,i)\); and \texttt{op.lhs.task} to identify the task name on the left hand side of the prescription \texttt{op}, its associated iteration domain is \texttt{task.dom}. Domains can be union of polyhedra.

In Alg. 6.1 the function \texttt{convertRegionToZPolyhedra} simply syntactically analyzes a range or region and create the associated polyhedron, and the \texttt{Image} function returns the image of a polyhedron by an affine function.

\textbf{Data and Control Dependences} Data and control dependences need to be captured using a polyhedral representation to enable optimization. We only capture scheduling constraints, and ignore scheduling hints. First, we remark that control dependences in PIPES (e.g., \((S:i) \rightarrow (T:i)\)) can be seamlessly converted to data dependences for analysis purpose by introducing fake item collections (e.g., \((S:i) \rightarrow [fk:i] \rightarrow (T:i)\)), therefore we only focus below on data dependence modeling. To properly model data dependences we need to express the data being accessed by each task instance.

\textbf{Item collections} Modeling item collections in the polyhedral representation does not need any form of treatment. We here rely on the properties that each item in a
Algorithm 6.1: Compute iteration domains

collection is uniquely identified by an integer tuple. That is, item collections can be seen as multidimensional arrays for polyhedral analysis purpose.
Access relations Access functions allow to associate to a particular task instance the data which is being read and written. To determine if an access function can be built from a PIPES read/written item collection, we simply analyze its tag function and ensure it is an affine multi-dimensional function. Similarly to prescription, ranges and regions can be used to describe a set of data being accessed by a task instance, or a set of task instances. This is why we actually use access relations instead of simple access functions.

Data dependences Equipped with iteration domains and access relations we can construct dependence polyhedra. Two task instances are in dependence if they access the same tag in the same item collection and if one of these access is a write. Dependence polyhedra are built from the set of all pairs of instances, one being the producer and the other being the consumer, such that both component of the pair access the same memory location. Classical polyhedral dataflow analysis uses information about the original program schedule to determine the producer and consumer of a memory location. This is required to model anti and output dependences, that is dependences where a write is performed after a read or write to the same location.

In PIPES, no schedule of the operations is provided as input: only the set of data being read/written by each task instance is available. This is a parallel-by-default representation, where (partial) ordering constraints are implicitly provided by modeling which task instances read/write particular tags in item collections. The lack of schedule could be problematic for dependence analysis purpose: taking for instance a write-after-write dependence, one cannot model the dependence as existing for all pairs of instances writing to the same location: it would over-constrain the
problem leading to the infeasibility of scheduling the program. Schedule information is typically used to prune this set of pairs of instances, to prune the pairs that cannot exist under the original program schedule.

Nevertheless computing polyhedral data dependences in PIPES is straightforward due to the strong property of the program being in dynamic single assignment: *one and only one instance can write a particular item*. In other words, write-after-write dependences cannot exist in a PIPES program, no write-after-read. Indeed if an item is read then it has been written before, and only one write can happen. Because of the DSA property only flow dependences can occur, and the producer instance is necessarily the one writing to an item collection. Our algorithm for building dependence polyhedra is therefore greatly simplified, as shown in Alg. 6.2. It is in essence a subset of the classical polyhedral dataflow analysis algorithm [44]. It assumes PIPES graphs containing non-affine access relations have been already rejected, and access relations \texttt{ref.accrel} are built from the tag function/region as intuitively shown above for an item collection named \texttt{ref.itemColl}.

### 6.2 Task Coarsening

One motivation for our framework is to apply coarsening of tasks, that is to reduce the total number of tasks in the graph while increasing their size. The reason is twofold. First, there is a runtime overhead associated with each task spawning and execution. In order to amortize this overhead tasks must be “long enough” in terms of computation time. Users usually find very convenient to write a fine-grain description of the program data, close to what happens at the statement level (see the example in the previous section). But in that case each statement instance translates
deppoly ← emptyList
foreach access a do
    a.accrel ← computeAccessRelation(a.tf)
end
foreach write access relation w do
    foreach read access relation r do
        if w.itemColl ≠ r.itemColl then
            continue
        end
        polydep ← CartesianProduct(w.task.dom, r.task.dom)
        polydep ← polydep ∩ w.accrel = r.accrel
        if polydep ≠ ∅ then
            deppoly ← addToList(deppoly, polydep)
        end
    end
end
return deppoly

Algorithm 6.2: Compute dependence polyhedra

to a task at runtime, literally killing the performance. Automatic PIPES transformation is motivated by the will to preserve the programmability while achieving good performance. The second reason for coarsening relates to data locality and partial scheduling. A runtime may take arbitrary scheduling and placement decisions, which can be highly detrimental to data locality. One way to address this problem is to group fine-grain tasks accessing nearby data into macro-tasks, which will then be handled by the runtime.

Task coarsening Task coarsening in PIPES is in essence classical polyhedral tiling on the polyhedral representation we extracted. Tiling achieves the coarsening and data locality objective, and as importantly, ensures by definition that tiles are atomic execution units. PIPES requires task instances to be atomic, therefore coarsening via
tiling is an effective way to preserve PIPES semantics. We implement task coarsening by using the Pluto transformation algorithm which implements the Tiling Hyperplane method [24]. This model-driven scheduler finds a multidimensional affine schedule for the task instances so that (1) data locality is maximized; (2) tilability is maximized, all under the framework restrictions of course. The transformation algorithm takes as input a list of iteration domains and dependence polyhedra, that is exactly what is produced by our algorithm in the previous section.

There is however a subtlety for true data locality optimization. In the tiling hyperplane method two conditions are enforced on the generated schedule: (1) that for all pairs of dependent instances, the producer is scheduled before the consumer (semantics preservation); and (2) that for all pairs of instances accessing the same data, the maximal distance between these accesses is minimized (data locality optimization). That second point is typically solved by considering read-after-read dependences, that is the “dependence” polyhedra which are built by looking at the various read functions only. These polyhedra need not be taken into account for semantics preservation, but they are used in the objective function

\[ \Theta_R(\vec{x}_R) - \Theta_S(\vec{x}_S) \leq \vec{u}.\vec{n} + w \]

which is modeled for each pair \( < \vec{x}_R, \vec{x}_S > \) of iterations such that they both read the same memory location. \( \vec{n} \) is the vector of program parameters, e.g. \( M \) and \( N \), and \( \vec{u} \) and \( w \) are non-negative unknowns whose value is minimized under legality constraints for the schedules \( \Theta_R \) and \( \Theta_S \) [24].

Similarly to anti and output dependence analysis, information about the original schedule is typically used to compute the exact set of read-after-read polyhedra that
are plugged in the objective function. This is however not needed and one can randomly select one of the two read reference as the “producer” and set the other as the “consumer” in the read-after-read polyhedron while simply updating the objective function above to be:

$$\left| \Theta_R(\vec{x}_R) - \Theta_S(\vec{x}_S) \right| \leq \vec{u}.\vec{n} + w$$

that is:

$$-\vec{u}.\vec{n} - w \leq \Theta_R(\vec{x}_R) - \Theta_S(\vec{x}_S) \leq \vec{u}.\vec{n} + w$$

This is the only update we implemented in Pluto, we remark it is not needed for correctness but only to achieve a potentially better scheduling.

### 6.3 Task Coalescing

Task coalescing consists on the fusion of task instances, potentially from different task collections. The benefits are various. It removes burden from the run-time by reducing the number of tasks in flight. Thus, the run-time has less book-keeping to perform. In addition, communication derived from the producer-consumer data-flow relations is effectively removed as it becomes an intra-task product. Several works such as [101, 24] have introduced complex heuristics driven by strongly connected components (SCC). Finally, polyhedral frameworks can seamlessly handle arbitrary fusion structures as they become embedded in the produced program schedule and generated by polyhedral code generators (e.g. CLooG [14]). The code generator deals with the complex task of performing potentially required peeling and loop shifting.

Although fully automatic task coalescing (fusion) is achievable by leveraging the PLuTo tiling algorithm in ISL [123], PIPES philosophy consists of allowing the user a high-level of control over the produced program. Thus, the user can decide which
tasks should be fused irrespective of the coarsening (tiling) schedule produced by ISL. It stands to reason that coalescing can violate the semantics of the program.

The default tiling scheme enabled in PIPES is PLuTo’s ISL minfuse heuristic, which is chosen to keep as much parallelism exposed. Leaving to the user the task of deciding when and how to fuse.

Task coalescing works at various levels of the compiler. It builds upon the leading scalar dimensions that are produced by polyhedral compilers. In a nutshell, every distinct integer value that appears in the front part of a multi-dimensional schedule represent separated instances. Programs in PIPES, as specified by the user explicitly define disjoint tasks. Algorithm 6.3 assumes that the main program SCoP has already been extracted, i.e., that it holds the iteration domains, data domains and access relations (both consumer and producer relations). The algorithm then starts by assigning a distinct scalar value to each declared task (collection). The value given is arbitrary, we only need them to be different for all tasks. If two tasks should be fused then this value should be the same. After this step, an explicit graph representation is constructed from the set of tasks and consumer/producer relations. The motivation for building this explicit graph is to simplify various checkings such as validating that a (instancewise) dependence cycle is not introduced. It also simplifies the job of aggregating inputs and outputs relations.

The next step of the algorithm (function RemoveIntraTaskCommunication) performs the very important role of determining when producer and/or consumer relations should be removed from the graph, and by extension, complete block collections if the intersection of the image of the producer relation with the domain of the consumer relation is equal to the block domain. Removing a relation, be it consumer or
producer, consists of subtracting the ISL maps from the main program SCoP. The
algorithm then continues by computing the tiling schedule of the complete program
with the minfuse heuristic. This leads to maximal separation among tasks. In the
case of having defined a single task, the schedule might not have the leading scalar
dimension. When this happens an arbitrary fixed integer (e.g., zero) is inserted in
the leading dimension position of the output tuple of the schedule. Naturally, at this
point the schedule produced and the fusion structure desired by the user can differ.
Thus, the last but one step consists of propagating the fusion coefficients from the
nodes in the IR to the schedule in the program.

\[
\text{Input: prog: the main program scop; root: the IR root node}
\]
\[
\text{Output: prog': the updated program scop; root': the update IR root node}
\]
\[
T \leftarrow \text{root.GetTaskEntities()};
\]
\[
\text{coef } \leftarrow 0;
\]
\[
\text{foreach } t \in T \text{ do}
\]
\[
| \quad \text{t.AssignFusionCoefficient (coef++)};
\]
\[
\text{end}
\]
\[
G \leftarrow \text{BuildExplicitGraph (root)};
\]
\[
\text{RemoveIntraTaskCommunication (prog,root,G)};
\]
\[
\text{ComputeCoarseningSchedule (prog,root,G)};
\]
\[
\text{PropagateFusionCoefficients (prog,root,G)};
\]
\[
\text{AssignTaskLeaders(root,G)};
\]

**Algorithm 6.3:** Task Coalescing Algorithm

The very last step of this algorithm defines for each groups of fused tasks, a task
leader. There can only be only task leader in each group. This flag is used to generate
(or skip) the various object and tuner declaration in the final program. For instance,
a non-leader task is not declared in the context, but it’s inputs and outputs must still
be printed when constructing the CnC graph as well as adding arguments to function calls such as the task kernels.

6.4 Leveraging CnC Tuners

Making efficient use of the CnC tuners is a critical feature of the PIPES language. CnC tuners are lightweight routines/functions that return bits of information such as assigning an execution location (a process rank) to a task.

The first three tuners that we exploit are related to the affinity of a task (i.e., on which process is each task executed), as well as the process that produces a particular block instance or consumer process(es) of a given block. These tuners have a large performance impact since it allows the run-time to directly locate tasks and blocks, therefore avoiding costly polling or querying all other processes while searching for a block. In addition, to use these tuners two requirements must be met: i) a virtual topology must have been defined; and ii) a logical-to-physical mapping must be provided.

6.4.1 Task Affinity

In order to generate code that leverages the task affinity tuner mechanism (the compute-on tuner) available in Intel CnC we construct one task scop for each individual task. This tuner allows to assign the execution of a task instance to a specific process by returning for each task-tuple an integer value representing the process’ rank. The idea here is to retrieve two maps: the coarsened schedule of the given task and the affinity map, the map from the task domain to the topology domain. If these two maps are defined, we compose them and build the proper scop fields. The resulting map is used as the access relation. As the domain of the scop we
Input: prog: the main program scop;
L ← AllocScopList();
if prog.topo is defined then
  foreach t ∈ prog.tasks do
    t_to_tt ← t.maps.schedule.coarsened;
    t_to_topo ← t.maps.affinity;
    tt_to_topo ← (t_to_tt)^{-1} ° t_to_topo;
    scop ← ScopAlloc();
    scop.dom ← domain(tt_to_topo);
    scop.accrel ← tt_to_topo;
    scop.schedule ← identity(scop.dom);
    L.append (scop);
  end
end
return L

Algorithm 6.4: Task Affinity

use the domain of the access relation, and since the schedule used is of no interest, we
use an identity schedule. Finally, the scop is appended to the scop list L. Algorithm
6.4 performs this job. The scops contained in the returned list are later used to
generate one tuner structure for each declared task.

6.4.2 Block Producer

To compute the producer of a block instance we leverage the produced-on tuner.
This tuner enables the run-time to determine for each tuple identifying a data-block
instance the rank of the producing process. More specifically, given a tuple that iden-
tifies a block instance for a particular block (item) collection, it returns the process’
rank on which the task that performed the PUT operation for this block was executed.
To generate this tuner we compose the coarsened write maps (coarsened on the block
tuple) with the task affinity maps. This yields a coarsened block affinity map. Then,
for each block definition we intersect the resulting maps (variable location) with the coarsened block, create the respective scop, and append it to the list L. Algorithm 6.5 proceeds in this manner.

**Input:** prog: the main program scop;
L ← AllocScopList();
if prog.topo is defined then
  location ← prog.writes.coarsened⁻¹ ◦ prog.affinity;
  foreach b ∈ prog.blocks.coarsened do
    bb_to_topo ← location ∩ b;
    scop ← ScopAlloc();
    scop.dom ← b;
    scop.accrel ← bb_to_topo;
    scop.schedule ← identity(scop.dom);
    L.append (scop);
  end
end
return L

**Algorithm 6.5:** Block producer affinity

### 6.4.3 Block Consumers

The algorithm to determine the process ranks that consume a particular block is very similar to Algorithm 6.5. It is used to build a block scop which is then employed to generate code for the consumed-on item tuner. This tuner returns a C++ vector of distinct process ranks associated to all the consuming tasks for a specific block tuple. The difference with the previous algorithm is that when producing the location maps, we use instead the prog.reads.coarsened, the coarsened read map. We note that each produced scop can potentially consist of two or more domains as well as access relation, since, naturally, blocks can be consumed by two or more
task instances from different task collections. Algorithm 6.6 shows how the consumer scops are constructed.

\textbf{Input}: prog: the main program scop;

L $\leftarrow$ AllocScopList();

\textbf{if} prog.topo is defined \textbf{then}

\hspace*{1em} location $\leftarrow$ prog.reads.coarsened $\circ$ prog.affinity;

\hspace*{1em} \textbf{foreach} b $\in$ prog.blocks.coarsened \textbf{do}

\hspace*{2em} bb_to_topo $\leftarrow$ location $\cap$ b;

\hspace*{2em} scop $\leftarrow$ ScopAlloc();

\hspace*{2em} scop.dom $\leftarrow$ b;

\hspace*{2em} scop.accrel $\leftarrow$ bb_to_topo;

\hspace*{2em} scop.schedule $\leftarrow$ identity(scop.dom);

\hspace*{2em} L.append (scop);

\hspace*{1em} end

\textbf{end}

\textbf{return} L

\textbf{Algorithm 6.6:} Block consumer(s) affinity

\subsection{6.4.4 Block Maxlife}

Estimating the lifetime of a block instance, i.e., the number of times it is read by any given task, is done by first fetching the coarsened read maps and intersecting this union map with each block-coarsened domain. This produces the coarsened read map restricted to the reads of a single block collection. The tasks that read the block in question are the range of this last map. Finally, for each \textbf{task} we compute its cardinality and store it. Algorithm 6.7 performs the described task to generate the \textbf{get-count} CnC tuner.
**Input**: prog: the main program scop;
Rall ← prog.reads.coarsened;
C ← ∅;

```plaintext
foreach b ∈ prog.blocks.coarsened do
    R ← Rall ∩ b;
    L ← ∅;
    foreach r ∈ R do
        task ← range(r);
        count ← card(task);
        L.append(count);
    end
    C.append(b.name,L);
end
return C
```

**Algorithm 6.7**: Block Maxlife

### 6.5 Autogen

The PIPES compiler can generate various distribution schemes from a basic input program. Distribution consists of assigning tasks to specific locations as well as varying the communication order, i.e., the order in which data is read or written, or trying distinct virtual-to-physical topology mappings (e.g., mapping a 3D-logical grid to a 2D physical grid). These degrees of freedom do not modify in any way the dependence graph of the input program.

Algorithm 6.8 creates a space of program variants by combining the logical-to-physical mappings, a list of affinity maps for each declared task, a number of code generation schemes to test for the GET and PUT operation of each block, and a list of tile sizes. The BuildX functions invoked for each of autogen parameters essentially construct the product set from a particular list type. For instance, for **commReads** a list of schedules is provided for each GET operation on each block. Finally, the
**Algorithm 6.8:** PIPES Autogen

compiler is invoked with logical-to-physical mapping, an affinity mapping set (one for each task), a set of read schedules (one schedule for each GET operation of each block), similarly for the PUT operations, and the tile sizes to be used (one for each dimension).
CHAPTER 7

The Pipes Compilation Framework

In Chapters 5-6 we introduced the PIPES language and discussed the main transformations implemented. In this chapter we describe the PIPES compiler, its internal data structures as well as implementation details. We start by giving a high-level view of the compilation process, followed by the IR and the class hierarchy used, the code generation stage and the back-end API.

7.1 Overview

The PIPES compiler takes as input a PIPES input program. The parser recognizes the program constructs (e.g., regions, task and block entities and the program relations) and extracts a PIPES-AST. Each of the PIPES-IR nodes encapsulate information specific to a program construct (e.g. domains for data-blocks or domains for tasks). Then, depending whether the specific construct fulfills the polyhedral requirements, an ISL object (i.e. an ISL-set or an ISL-map) is extracted. The PIPES program SCoP is built from these constructs. The following stage consists of transformations performed over the graph which can modify the program in one of 3 ways: a) by adding new program entities such as synchronization blocks, new relations, or
new tasks; b) alter the granularity of the tasks and blocks with coarsening (tiling) or coalescing (fusion); c) perform both modifications.

The code generation process starts after all the transformations have completed. Unlike most Polyhedral compilers which focus on optimizing identified valid SCoPs, PIPES generates an "almost complete" CnC C++ program. We say almost complete because the user has to provide a few external functions such as the task-kernel or routines that initialize and finalize the program. This means that the compiler has to generate function prototypes for all kernels and input/output routines, generate structure and function definitions for the tuners and tasks, print the CnC context, initialize all objects, build the task bodies with the correct data accesses, prescribe the tasks, generate input data, launch all the tasks, read the program results and finalize the program. Therefore, PIPES extracts various subscops from the main program SCoP for each different program function or accessor (e.g. a task-block-get or a task-prescription), generates code for each of them, and passes this result to the pretty-printer, which deals with the basic CnC/C++ program structure. The code generated consists primarily of routines and macro invocations defined in the PIPES-API. The API essentially abstracts the common functionality with other task run-times (e.g. OCR [1]) while greatly simplifying the target code and hiding low-level details from the user. Figure 7.1 explains this process.

7.2 The parser

The role of the PIPES parser is to recognize and accept valid programs. Each rule of the grammar produces a PIPES-node (e.g. a pipes-relation-consumer or a pipes-tuple-task. During this stage, scalar expressions that appear in tuples are
marked as affine or non-affine. A node is affine if all its subtrees are affine. For instance, if an expression \((T : i/N, j) \rightarrow [A : i, j]\) is found, the scalar expression \(i/N\) is marked as non-affine, as well as the source-tuple of the relation and the relation itself. This allows the compiler to decide whether the code generation should be performed using the ISL/affine back-end or if the non-affine module should be used. As this stage progresses, the parser recognizes the main program entities and relations, storing them in the PIPES-ROOT, the root node of the AST. If a syntax error is found, the compiler aborts.
7.3 PIPES-IR

The PIPES-IR is essentially an Abstract Syntax Tree (AST). The root node stores in its symbol table all the first-level constructs such as task entities, block entities, region variables, (virtual) topologies and all relations used in the program. At the same time, this node provides various search functions, with the goal of collecting types and subtypes of entities and relations.

To allow for search and collect functionalities, as well as performing various semantic checks, many IR-nodes have subtypes. For instance, a base-class pipes-entity, from which the sub-classes pipes-entity-task, pipes-entity-block and pipes-entity-topology are derived.

Figures 7.2-7.3 show two relation types, producer and prescription. The former depicts a instance \((t,i,j)\) of task \(TGG\) writing instances of block \(GG\). The specific instances being written are constrained to the \(t\) value on the input tuple, and further refined by embedding an anonymous region in the output tuple. Variables \(i\) and \(j\) are the parameters to the region and are used to constrain the output instances according to the region’s definition.

7.3.1 Tuples

Tuples are the main building block of a PIPES-AST. They are extensively used to define regions, entity domains as well as various types of relations. Each dimension of a tuple can either be a scalar expression (a scalar variable, a number, a program parameter, i.e., a fixed but unknown-value, or an arithmetic expression), a region range (1-dimensional anonymous space which can be affine or non-affine), a region
Due to the variety of dimension types, a tuple can have an iteration domain associated with it. This is the result of using region ranges or embedding previously defined region variables into it. This leads to adding the constraints of the embedded (anonymous) regions, into the tuple's domain.

This class also provides the basic unparsing functionality used to construct more complex sets and maps. This is further discussed in Sections 7.3.4-7.3.5.

### 7.3.2 Regions

Regions allow specifying a set of points, potentially living in different named sets. The main benefit of using regions is that they permit to represent complex data-flow constraints, domains, relations and provide additional information for specific program entities.
7.3.3 Named and Anonymous Regions

PIVES regions are closely related to ISL objects. ISL sets and maps allow to assign a name to sets and to the input and output tuples of a map. The role of these names is to define separate spaces, allowing to define unions of these objects. For instance, the union set \( \{ A[i] : 0 \leq i \leq 3; B[i] : 3 \leq i \leq 5 \} \) defines two sets, A and B. However, omitting these names will effectively produce an anonymous space \( \{ [i] : 0 \leq i \leq 5 \} \), assuming that the two sets are the same and fusing their domains. Furthermore, several ISL operations such as set intersection and map application are
only valid when the tuple names match. Therefore, this information is required during the definition of regions.

**Region Properties** PIPES provides a number of region properties that convey extra information to the compilation process. The three region properties currently supported are: i) **space**, used to specify the real data-space dimensions from a data-flow tuple; ii) **order**, a tuple which purpose is to define the scanning order of the associated region, i.e., a schedule for generating loops; iii) **mempool**, a tuple that defines a memory recycling mechanism by indicating a dimension and a fixed value.

**Region Ranges** Region ranges are one-dimensional anonymous sets, e.g. 1..N used to specify simple constraints within tuples. Each node of this type consists of a scalar variable that represents the dimension name and a scalar expression that bounds the dimension values.

**Region** Regions are a base class to Region Sets (non-affine sets) and Region Polyhedra (affine sets). They provide the basic mechanism for extracting the set of points as well as relations derived from specifying region properties. Every region is composed of a **pipes-tuple** and a domain, a scalar expression. The tuple used to declare the dimensionality of the region is an untyped **pipes-tuple**, but which can either be named or anonymous.

All types of regions provide unparsing functions to construct the tuple indices and the region domain.

**Union of Regions** Union of regions are essentially a list of regions, either region sets or region polyhedra. Each region in the union can have a name associated.
**Region Variables**  Are an abstraction that allow to embed regions within tuples, both typed and untyped. Region variables are first-class citizens. They can be assigned and later used to define one or more tuples in entities and relations. However, this poses a slight problem for the IR, since tuples can have regions embedded and regions require tuples to be properly defined. Thus, when instantiating a tuple which has an embedded region, we first translate the domain of the region variable to a Polylib-like format, making it independent of the tuple.

### 7.3.4 Entities

Entities are all program constructs which require a single tuple, potentially a union of regions (used for non-convex domains) and that live in a single space. Tasks, blocks and topologies are examples of entities, each embodying its own subtype. The entity sub-classes derived from the base `entity` class are the following:

- Tasks
- Blocks
- Environment
- Topology

All entity classes have the same common fields, a name, a `region_union` and an untyped-tuple. The region union and tuple fields are used to extract ISL sets that represent the iteration domain of a task, a block, a topology or the environment task. Algorithm 7.1 performs this job. The domain of an entity is built from the entity’s tuple field in full, from the `region_union` used to define the complete space of a tuple,
or from both the tuple and the region_union, in which case the region_union defines an anonymous strict sub-space of the entity’s domain.

**Input**: rel: a PIPES entity; params: program global parameter  
**Output**: set: an ISL set  

tup ← tuple.indices.getISLstring ();  
tupdom ← tuple.domain.getISLstring ();  
regdom ← region.domain.getISLstring ();  
ret ← "";  
domain ← "";  
if not params.empty () then  
  ret ← ret + params;  
end  
ret ← ret + tup;  
if not tupdom.empty() then  
  domain ← domain + tupdom;  
end  
if not regdom.empty() then  
  if not domain.empty () then  
    domain ← domain + " and ";  
  end  
  domain ← domain + regdom;  
end  
if not domain.empty () then  
  ret ← ret + domain;  
end  
set ← isl_set_read_from_str (ret);

Algorithm 7.1: ISL-SET construction from a PIPES entity

### 7.3.5 Relations

PIPES relations are classes that encapsulate mappings between a source and a destination tuple, each of which must have a specific tuple class (e.g., a tuple-task or tuple-topology). The base class `relation` provides the basic functionality for extracting ISL maps and union-maps from the specific relations.
The classes derived from a relation are the following:

- Consumer Relations
- Producer Relations
- Prescribing Relations
- Affinity Relations
- Weak Order Relations
- Strong Order Relations
- Move Relations

It is worth noting that all the relations are binary, except the move relation, which is a ternary one. Both the source and destination tuple are task-tuples, and has, in addition, a third tuple of type block-tuple, which designates the block instance(s) to be moved from the task source to the task destination.

ISL maps and union-maps are extracted from PIPES relations by translating the source and destination tuple indices, concatenating them, and extracting for both of them their domain, if any. Then the ISL function `isl_map_read_from_str` is used to construct the `isl_map`. Algorithm 7.2 shows the process of producing an `isl_map`.
**Input:** rel: a PIPES relation; params: program global parameter  
**Output:** map: an ISL map  

```plaintext
src ← rel.src.indices.getISLstring ();
dst ← rel.dst.indices.getISLstring ();
dom_src ← rel.src.domain.getISLstring ();
dom_dst ← rel.dst.domain.getISLstring ();
ret ← "";
domain ← "";
if not params.empty () then
    ret ← ret + params;
end
ret ← ret + src;
ret ← ret + " → ";
ret ← ret + dst;
if not dom_src.empty () then
    domain ← domain + dom_src;
end
if not dom_dst.empty () then
    if not domain.empty () then
        domain ← domain + " and ";
    end
    domain ← domain + dom_dst;
end
if not domain.empty () then
    ret ← ret + domain;
end
map ← isl_map_read_from_str (ret);
return map;
```

**Algorithm 7.2:** ISL-MAP construction from a PIPES relation

**Generating Real-Space Relations**  
The PIPES language design is based on data-flow relations, i.e., relations of tasks producing or consuming blocks. However, the volume (the number of integer points in the domain) of a block does not always match the data-flow relation due to the DSA property of CnC programs. The problem here is that, by default, all dimensions of a block are space dimensions. For instance, consider the relation \((TJJ:t,i,j) → [AA:t,i,j]\), of a Jacobi-like stencil. The leading
dimension in the input and output tuple of the relation represents time, i.e. it is already a data-flow dimension which guarantees the DSA property of CnC. Thus, when computing the volume of a single data-block, this dimension should not be accounted. For handling this case, the PIPES language provides the space region property, which allows to construct both the real data-space block domain and the real data-space access relations. The latter is a relation from the entity’s domain to the real data-space, whereas the former is a domain wherein the selected dimension is collapsed to size 1. In other words, the real-space relations are a mechanism to identify explicit data-flow dimensions when defining a region, avoiding to consider them when computing set volumes.

Generating Schedules for Communication and Prescriptions One of the virtues of the PIPES language is to allow specifying the order in which a particular communication operation must be performed. This order is given by the order region property. Schedules maps (maps from the entity domain to the time space) are constructed in a fashion similar to Algorithm 7.2, however, with a small variation. Instead of using the destination tuple of the relation, the tuple given in the order property is used as the output tuple. By default, all regions use an identity schedule in the 2d+1 representation.

7.4 The PIPES SCoP

The main PIPES SCoP has some substantial differences w.r.t. the traditional SCoP format used in most polyhedral compilation frameworks. The reason for this is that it relies on various types of relations to build a valid CnC program. A standard SCoP is normally comprised of domains, access functions and a schedule used
during code generation, i.e., the scanning order. In contrast, a PIPES SCoP must maintain information about prescribed tasks, task-to-topology mappings, weak and strong dependences among tasks, as well as the producer and consumer relations. Furthermore, knowledge about both the task (iteration) domain and the block domain, the data-space, is required. Finally, due to the automatic coarsening capabilities of our framework, a fine-grained and coarse-grained representation of all sets and maps are kept at all times.

7.5 Affine module

When the domains and relations of the input program are affine the PIPES compiler resorts to use the affine-module. This module is responsible of computing the block-properties (dimension sizes) of block-collections with the Barvinok library [10] [123], extracting task and block sub-scops, and generating tailored code for each of them.

7.5.1 Block Properties

Every task, including the environment task, accesses one or more block collections. One of PIPES features is to determine the number of block instances read or written for each block collection accessed from a task, as well as determining the sizes of each block. This is done by constructing a data-tile map and intersecting it with the domain of the block. Then, for each output dimension $d$ we project out all dimensions except the chosen dimension, and invoke the \texttt{isl.set.card} to compute the number of integer points. This value is then converted to a string format and stored in a dictionary indexed by the task name, the access type (PUT or GET) and the block
name. This dictionary is later queried by the code generation process and pretty-printer described in Section 7.5.3 and Section 7.7.

### 7.5.2 Subscop Extraction

The subscop extraction routine consists of building a task or block SCoP from the main program SCoP. The reason for this is threefold. First, the overall structure of a CnC C++ program, which requires following a very specific syntax. Second, we also require to generate block SCoPs and not only task-scops. The main difference between these two types of SCoPs is that the access functions of a task-scop refer to block-instances, whereas the access functions of block-scop refer to the tasks writing or reading the block. Third, we can have scops consisting exclusively of GET operations or PUT operations. For instance, when generating block-tuners that use the PULL model we use only the PUT operations of a single block collection, whereas when generating block-tuners that use the PUSH model we resort to build SCoPs in which the domain of the SCoP is the block domain while the access functions are all the GET operations performed by all tasks over the block. Algorithm 7.3 shows how a task subscop is constructed.
Input: prog: main SCoP; root: pipes IR root node
Output: S: SCoP list
FC ← CollectFusionCoefficients (prog.coarse_schedule);
foreach $c \in FC$ do
    dom ← ExtractDomains(prog.coarse_schedule, c);
    accrel ← IntersectDomainOfRelation(prog.coarse_accrel, dom);
    schedule ← IntersectDomainOfRelation(prog.codegen_schedule, dom);
    scop ← CreateNewScop (dom, accrel, schedule);
S.add_scop (scop);
end
return $S$;

Algorithm 7.3: Sub-SCoP extraction from main SCoP

7.5.3 Code Generation

Code generation in PIPES follows a layered approach. The output produced must conform to the syntax of a complete and valid C++ program, which targets the PIPES-API. The overall approach is as follows: for each task (or block) we generate a SCoP by extracting a particular set of domains and relations from the main program SCoP. In the case of a task-scop, the access relations are operations on blocks. In contrast, for a block-scop, its access relations are the tasks that either write to it or read from it. We then proceed to generate the loop structure by using ISL’s printing capabilities. However, at this point we further tailor the code generation by plugging-in different routines that extract the statement domain name and tuple, and use this information to build a specific macro-call to the PIPES-API. The code generated is then stored in a dictionary and indexed by the entity’s name and operation type (e.g., a TASK_BLOCK_GET). Listings 7.4 and 7.5 follow the depicted structure. The second argument to CodegenEntity is both the operation type and the subscop type, defined in the PIPES API.
**Input**: prog: main SCoP; root: pipes IR root node  
**Output**: codemap: dictionary indexed by entity name and operation type  

```plaintext
scops ← ExtractSubscop(prog,TASK);
foreach s ∈ scops do
  code ← CodegenEntity (s,ENV,TASK_PUT);
codemap.insert(s.name,ENV,TASK_PUT,code);
code ← CodegenEntity (s,TASK_BLOCK_GET);
codemap.insert(s.name,TASK_BLOCK_GET,code);
code ← CodegenEntity (s,TASK_BLOCK_PREPUT);
codemap.insert(s.name,TASK_BLOCK_PREPUT,code);
code ← CodegenEntity (s,TASK_BLOCK_PUT);
codemap.insert(s.name,TASK_BLOCK_PUT,code);
code ← CodegenEntity (s,TASK_BLOCK_PULL);
codemap.insert(s.name,TASK_BLOCK_PULL,code);
code ← CodegenEntity (s,TASK_BLOCK_PUSH);
codemap.insert(s.name,TASK_BLOCK_PUSH,code);
code ← CodegenEntity (s,TASK_BLOCK_MAXLIFE);
codemap.insert(s.name,TASK_BLOCK_MAXLIFE,code);
end
return codemap;
```

**Algorithm 7.4**: Code generation for sub-scops of type entity task

---

**Input**: prog: main SCoP; root: pipes IR root node  
**Output**: codemap: dictionary indexed by entity name and operation type  

```plaintext
scops ← ExtractSubscop(prog,BLOCK);
foreach s ∈ scops do
  code ← CodegenEntity (s,BLOCK_PULL);
codemap.insert(s.name,BLOCK_PULL,code);
code ← CodegenEntity (s,BLOCK_PUSH);
codemap.insert(s.name,BLOCK_PUSH,code);
code ← CodegenEntity (s,BLOCK_MAXLIFE);
codemap.insert(s.name,BLOCK_MAXLIFE,code);
end
return codemap;
```

**Algorithm 7.5**: Code generation for sub-scops of type entity block

153
7.6 Non-affine Module

If any descendant of a tuple node used in either a relation or in a region is marked as non-affine by the parser the affine-module will be incapable of proceeding with the analyses, transformations and later code generation stages. The PIPES compiler will thus fallback to the non-affine module which is capable of generating parameterized but unoptimized loops.

The overall code generation scheme is presented in Algorithm 7.6 which proceeds recursively. It starts at depth zero of the AST. At each level the function CollectScalarValues queries the scalar fusion coefficients present in the schedule, filtered by the current outer scalar dimensions. It then performs a recursive call to NonAffineCodegen for building the subtrees at the following level, and receives a forest of ASTs, which are then used as the loop bodies at the current level. Once the maximum depth of the schedule is reached, the domains are used to construct the final program statements (which target the PIPES-API). The function BuildStatementFromAccessFunctions performs this job. In addition, an specific guard for each domain is constructed.

Non-affine domains can arise in a number of ways. For instance, a loop bound could consist of (multi)-polynomial expressions of degree 2 or higher, or attempt to multiply an iterator with a program parameter. The approach we adopt to represent these domains and extract the loop bounds is similar in spirit to the Fourier-Motzkin elimination process [83], the difference being that the constraints are first sorted per dimension in an out-to-in fashion. During the sorting procedure we validate that the constraints never depend on inner iterators, and if so, the compiler aborts. If no such dependence exists the loop bounds are stored in a table indexed by domain name,
dimension and loop bound type. The functions \texttt{CollectDomainLowerBounds} and \texttt{CollectDomainUpperBounds} simply search for a particular domain and dimension, and return the associated loop bound.

When generating loops for a given level, lower and upper bounds are computed. These bounds are determined as the minimum of all lower-bounds and as the maximum of all upper-bounds for all statements, respectively. The outer scalar dimensions of the schedules enable the fusion and ordering of the statements. However, since we take this min/max approach, we also require adding \texttt{if nodes/guards} that split the next schedule dimension among the statements. Then, the function \texttt{BuildDomain-Guard} constructs the specific domain conditions to split a schedule dimension and order sibling nodes according to the value of the current scalar dimension.
Input: prog: main SCoP; root: pipes IR root node, d: current dimension; outer: list of outer dimension names; scalars: list of outer fusion coefficients

Output: L: generated loopnest or G: a guard

schedule ← root.schedule;
domains ← root.domains;
if d is the last dimension of the schedule then
    accrel ← root.accrel;
    foreach ar ∈ accrel do
        stmts ← BuildStatementFromAccessFunctions (domains, outer, ar);
        G ← BuildDomainGuard (domains, schedule, d, stmts);
    end
    return G;
end
else
    outer.append (BuildIterator(d));
    next_scalars ← CollectScalarValues (schedule, scalars, d+2);
    SortScalarsLexicographically (next_scalars);
    subtrees ← NonAffineCodegen (prog, root, d+1, outer, next_scalars);
    outer.removeLast ();
    L ← ∅;
    foreach s ∈ scalars do
        lb ← CollectDomainLowerBounds(domains, d);
        ub ← CollectDomainUpperBounds(domains, d);
        body ← ExtractBodies (subtrees, s);
        loop ← GenerateLoop(d, lb, ub, body);
        guard ← BuildDomainGuard (domains, schedule, d, loop);
        L.append (guard);
    end
    return L;
end

Algorithm 7.6: Code generation for non-affine scops

7.7 PIPES Pretty-Printer

This module is responsible for printing the general CnC C++ program structure. In essence, it prints the proper CnC header files, structure declarations for tuners and tasks, the CnC context, the program driver and the main function. In particular,
it receives code snippets generated by the affine and non-affine modules, stored in a dictionary indexed by fields entity-name and entity-role.

### 7.7.1 Full Program Structure

Algorithm 7.7 drives the generation of a complete PIPES program. It assumes that code snippets have been generated for all entities and properly stored in the codemap dictionary, and that block properties for all entity blocks have been already computed. It proceeds by printing the required header files and external functions (user provided functions), generating the program context structure (See Algorithm 7.8), generating the task bodies (See Algorithm 7.10), and finally printing the driver function (See Algorithm 7.9), which in essence is the task-body of the environment task.
**Input:** root: pipes IR root node; codemap: code generated for each subscop; props: block properties

PrintProgramDeclarations ()

```plaintext
def print_tuner(e):
    if e is entity block or e is entity task:
        print_tuner(e)
```

PrintProgramContext (root)

```plaintext
def build_header(t):
    for input_block b in inputs(t):
        block_sizes = props.find(t, b, INPUT)
        print(block_sizes)
    for output_block b in outputs(t):
        block_sizes = props.find(t, b, OUTPUT)
        print(block_sizes)
```

```plaintext
def get_code(t, task, block):
    gets = codemap.find(t, task, block, GET)
    kernel = codemap.find(t, task, block, KERNEL)
    puts = codemap.find(t, task, block, PUT)
    print(gets)
    print(kernel)
    print(puts)
```

PrintDriver ()
PrintMain ()

**Algorithm 7.7:** PIPES full-program printing

### 7.7.2 Printing the Context

The program context brings together all the task collections, block collections and associated tuners into a single CnC program. Each of these program objects must be properly declared as a field in the context and later initialized. The declaration of a task and block declaration can include a tuner of a specific type. Effectively, each
block and task collection requires the definition of a tuner type (a structure) and the instantiation of a tuner-variable that is then passed to the task/block constructor.

The body of the context constructor is responsible of establishing the relations among the different program entities: specifying what tasks produce (write) what blocks, what blocks are consumed (read) by tasks, as well has specifying the tag collections that prescribe tasks and the driver function that controls the tag collections.
**Input:** root: pipes IR root node; codemap: code generated for each subscop; props: block properties

\[
T \leftarrow \text{root.entities(TASK)};
\]

\[
B \leftarrow \text{root.entities(BLOCK)};
\]

**foreach** \(t \in T\) **do**

| PrintTagDeclaration(t); |
| PrintTaskDeclaration(t); |
| **if** \(t\text{ has tuner}\) **then** |
| PrintTunerDeclaration(t); |
| **end**

**end**

**foreach** \(b \in B\) **do**

| PrintBlockDeclaration(b); |
| **if** \(b\text{ has tuner}\) **then** |
| PrintTunerDeclaration(b); |
| **end**

**end**

**foreach** \(r \in \text{root.relations(RELATION\_PRODUCER)}\) **do**

\[
\text{src} \leftarrow r.get\_source\_name();
\]

\[
\text{dst} \leftarrow r.get\_destination\_name();
\]

Print (src + ” produces” + dst);

**end**

**foreach** \(r \in \text{root.relations(RELATION\_CONSUMER)}\) **do**

\[
\text{src} \leftarrow r.get\_source\_name();
\]

\[
\text{dst} \leftarrow r.get\_destination\_name();
\]

Print (dst + ” consumes” + src);

**end**

**foreach** \(r \in \text{root.relations(RELATION\_PRESCRIBE)}\) **do**

\[
\text{dst} \leftarrow r.get\_destination\_name();
\]

Print (” tag” + dst + ” prescribes” + dst);

Print (” driver” + controls” + ” tag” + dst);

**end**

**Algorithm 7.8:** Printing the program context
7.7.3 The Driver

Is the routine invoked from the main function. It embodies the work performed by the environment task, i.e., generating input data, prescribing tasks, waiting for task termination and reading output data.

\[\begin{align*}
\text{Input:} & \text{ root: pipes IR root node; codemap: code generated for each subscop; props:} \\
& \quad \text{block properties} \\
& E \leftarrow \text{root.environment;} \\
& \text{foreach output block } b \in E.\text{outputs do} \\
& \quad \text{block.sizes } \leftarrow \text{props.find(E,b,OUTPUT);} \\
& \quad \text{Print(block.sizes);} \\
& \text{end} \\
& \text{foreach input block } b \in E.\text{inputs do} \\
& \quad \text{block.sizes } \leftarrow \text{props.find(E,b,INPUT);} \\
& \quad \text{Print(block.sizes);} \\
& \text{end} \\
& \text{puts } \leftarrow \text{codemap.find(E,TASK\_BLOCK\_PUT);} \\
& \quad \text{Print(puts);} \\
& \text{prescriptions } \leftarrow \text{codemap.find(E,ENV\_TASK\_PUT);} \\
& \quad \text{Print(prescriptions);} \\
& \text{gets } \leftarrow \text{codemap.find(E,TASK\_BLOCK\_GET);} \\
& \quad \text{Print(gets);} \\
\end{align*}\]

**Algorithm 7.9:** PIPES driver function

7.7.4 Tasks Bodies

Algorithm 7.10 shows how the body of a single task is generated. It receives as input an entity_task IR node, the map with all the code snippets and the structure of block properties. For each input and output block collection, variables are defined with the dimension sizes. The dimension sizes are used later to declare the input and output block collections. Next, all the GET operations are listed, followed by the task PREPUT operations, the task kernel invocation and, finally, the task PUT operations.
operations. The PREPUT operation is an abstraction required by the framework and consists of pre-assigning (in an arbitrary order) a tuple (tag) to each output-block of each collection. This step is necessary to preserve the structure of the computation, since both GET and PUT operations can be aggressively reordered when scheduling communication. The assigned tuple is then used by the task-kernel to fetch the corresponding block-to-write and using it according to the nature of the computation.

**Input**: t: task to print; codemap: code generated for each subscop; props: block properties

```plaintext
foreach output block b ∈ t.outputs do
    sizes ← ∅;
    foreach dimension d of b do
        block_sizes ← props.find(t,b,d,OUTPUT);
        Print(block_sizes);
        sizes ← sizes ∪ block_sizes;
    end
    PrintBlockCollectionDeclaration (b,sizes,OUTPUT);
end

foreach input block b ∈ t.inputs do
    sizes ← ∅;
    foreach dimension d of b do
        block_sizes ← props.find(t,b,d,INPUT);
        Print(block_sizes);
        sizes ← sizes ∪ block_sizes;
    end
    PrintBlockCollectionDeclaration (b,sizes,INPUT);
end

gets ← codemap.find(t,TASK_BLOCK_GET);
Print(gets);
preputs ← codemap.find(t,TASK_BLOCK_PREPUT);
Print(preputs);
kernal ← codemap.find(t,TASK_BLOCK_KERNEL);
Print(kernal);
puts ← codemap.find(t,TASK_BLOCK_PUT);
Print(puts);
```

**Algorithm 7.10**: PIPES task body printing
The code generated for the task bodies targets the PIPES-API. Listing 7.1 shows the macros used to encapsulate the functionality of a single PUT operation. The method `set_next_tuple` of the block collection internally selects the next free (empty) block and sets its tuple to the given value. The PUT operation is responsible of fetching a block from the collection, enclosing it in a C++ `shared_ptr` and finally performing the actual block PUT to its respective collection.

**Listing 7.1: PIPES-API block operations**

```c
#define PIPES_BLOCK_COLLECTION_PRE_PUT(progname,collection,tuple) \
    PIPES_OUTPUT(collection).set_next_tuple(tuple);
#define PIPES_BLOCK_COLLECTION_PUT(progname,collection,tuple) \
    PIPES_CTX_NAME(progname).PIPS_BLOCK_NAME(collection).put(\ 
    tuple,\ 
    (PIPS_OUTPUT(collection).at(tuple)));
```

### 7.7.5 Tuners

The PIPES compiler leverages the CnC step and item tuners. By default all tuning options are disabled.

**Task Tuners**  CnC step tuners provide three essential task capabilities in PIPES: 1) they allow to associate tasks to processes, prior definition of a virtual topology, via an affinity mapping relation; 2) they allow to specify for each task instance its input dependences (block instances); 3) they allow to define a priority for each task instance. The first of these, the task affinity, is implemented via the `compute_on` tuner, input dependences are implemented with the `dependency_consumer` tuner while the task priorities are implemented with the `priority` tuner. The second of these allows to only schedule a task once all its input dependences have been satisfied, thereby avoiding a task to be scheduled, ran, and suspended if it finds that an input is not yet available.
The third tuner, priority, permits to assign an integer value to each task instance thereby favoring tasks with higher priority in the scheduling policy. Algorithm 7.11 shows how task tuners are generated once all the subscops and block properties have been computed.

**Input:** t: task to print; codemap: code generated for each subscop; props: block properties

\[\begin{align*}
\text{PrintTaskTunerHeader}(t); \\
\text{aff} & \leftarrow \text{codemap.find}(t, \text{TASK AFFINITY}); \\
\text{if} \ \text{aff} \ \text{then} & \text{PrintHeader}(t, \text{TASK AFFINITY}); \\
& \text{Print} (\text{aff}); \\
\text{depcons} & \leftarrow \text{codemap.find}(t, \text{TASK DEPENDENCY CONSUMER}); \\
\text{if} \ \text{depcons} \ \text{then} & \text{PrintHeader}(t, \text{TASK DEPENDENCY CONSUMER}); \\
& \text{Print} (\text{depcons}); \\
\text{prty} & \leftarrow \text{codemap.find}(t, \text{TASK PRIORITY}); \\
\text{if} \ \text{prty} \ \text{then} & \text{PrintHeader}(t, \text{TASK PRIORITY}); \\
& \text{Print} (\text{prty}); \\
\end{align*}\]

**Algorithm 7.11:** PIPES task tuner printing

**Block Tuners** Block tuners in PIPES leverage three tuning options from CnC: 1) the **produced on** tuner; 2) the **consumed on** tuner and 3) the **get count** tuner. The first two tuners affect how the communication is performed: a consumer task can "pull" its input blocks from the producer’s location or the producer task can "push" its written blocks towards all its consumers. In the latter case, data blocks are already local to the process whereon the task executes, once it commences execution. The
last tuning capability, allows to limit the lifetime of a block based on the number of
tasks instances that perform GET operations. Listing 7.12 shows how block tuners
are generated once all the subscops and block properties have been computed.

**Input:** b: block to print; codemap: code generated for each subscop; props: block
properties
PrintBlockTunerHeader(b);
pull ← codemap.find(b, BLOCK_PULL);
if pull then
    PrintHeader(b, BLOCK_PULL);
    Print (pull);
end
push ← codemap.find(b, BLOCK_PUSH);
if push then
    PrintHeader(b, BLOCK_PUSH);
    Print (push);
end
maxlife ← codemap.find(b, BLOCK_MAXLIFE);
if push then
    PrintHeader(b, BLOCK_MAXLIFE);
    Print (push);
end

**Algorithm 7.12:** PIPES block tuner printing

### 7.8 PIPES API

The PIPES API is designed to abstract and hide away minor syntactic differences
on how different run-times prescribe (spawn) tasks, read and write data, and bring
together the many program objects. An example of this can be found when declaring
tasks within the context: CnC requires every task (step) collection to be prescribed
by a tag collection, i.e., every task instance needs to be associated to a tag instance.
Similarly, when a (task or block) tuner is used, this must be previously declared in
order to pass the tuner type to the declaration of the task (or block). A brief example is given in listing 7.2, where a single macro is used to declare a task, its tag collection as well as to define the tuner type it uses. Listing 7.3 shows the macro expansion, which targets the CnC API.

Listing 7.2: Task declaration in program context using the PIPES-API

```
PIPES_TASK_COLLECTION_CTX_DECLARE_WITH_TUNER(TGG);
```

Listing 7.3: PIPES-API task macro declarations expanded

```
CnC::tag_collection < Tuple > TGG_tags;
struct_task_tuner_TGG task_tuner_TGG;
CnC::step_collection< struct_TGG_task, struct_task_tuner_TGG >
               task_TGG;
```

Another function of the PIPES-API is to simplify the declaration of block collections within tasks bodies (and from the environment). Each block collection used has a name, a data type, an accessor type (GET or PUT) and two lists which define the dimension sizes of the collection itself and of each block in the collection.

### 7.8.1 PIPES Block Collections

Block collections in PIPES are responsible for allocating the required memory and storing the input and output blocks. The original list of collection sizes and block sizes are passed to the macros `PIPES_BLOCK_COLLECTIONDECL_INPUT` and `PIPES_BLOCK_COLLECTIONDECL_OUTPUT`, for input and output blocks, respectively, so that collections can be resized and blocks allocated if needed. Listing 7.4 show the macros used for declaring block collections of arbitrary size as well as collections with single blocks. The latter type are used from the environment, when writing the initial data blocks and reading the final results.
7.8.2 PIPES Blocks

PIPES blocks are the structure that store the information of a single data block. Blocks are templatized, and consist of a chunk of contiguous memory, a list of dimension sizes as well as a tuple field. The tuple field is used to differentiate block instances within a collection. This is particularly needed when a task is known to read or write multiple block instances from a single collection. Furthermore, when scheduling communication (e.g., reordering PUT and GET operations) blocks can be aggressively reordered. Embedding this information into each block allows the task-kernel to retrieve them and to maintain the structure of the computation.
In this chapter we validate the effectiveness of the PIPES compilation framework. We choose two classical distributed-memory algorithms from the literature, namely the Cannon 2D algorithm [28, 84] and the Johnson 3D algorithm [5]. We also take two Jacobi stencil computations, a 2D-9pt and a 3D-27pt. In addition, we leverage PIPES’ coalescing transformation to show that Johnson’s 3D algorithm when exploring different coarsening values combined with task fusion becomes effectively a 2.5D algorithm. The results demonstrate that, contrary to the standard 2.5D algorithms, which follow a memory expansion approach, contracting data-flow dimensions also yields promising research directions.

The structure of this chapter is as follows: we first present the experimental setup used. We then briefly discuss the characteristics of the benchmarks used while performing an analytical estimation of the communication time based on the length of the critical path. Next we discuss the performance results of each benchmark. Finally, we explain some techniques that can be useful during performance debugging for programs implemented in our framework.
8.1 Experimental Setup

Our experimental test bed is described in Table 8.1. The experiments were conducted in OSU RI research cluster, using 1, 2, 4 and 8 nodes, as well as testing the effects of switching between a fast and a slow network. For the fast network we use an InfiniBand QDR (40 Gbps) whereas for the slow network we use the default, a Gigabit Ethernet. The peak performance of each node in the cluster is 160 GF/s in single precision.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>1-8</td>
</tr>
<tr>
<td>Processor</td>
<td>Intel Xeon E5630 @ 2.5 GHz</td>
</tr>
<tr>
<td>Sockets per node</td>
<td>2</td>
</tr>
<tr>
<td>Cores per socket</td>
<td>4</td>
</tr>
<tr>
<td>Intra-node bandwidth</td>
<td>25000 MB/s</td>
</tr>
<tr>
<td>10 Gigabit Ethernet (GbE) bandwidth</td>
<td>1250 MB/s</td>
</tr>
<tr>
<td>InfiniBand QDR (IB) bandwidth</td>
<td>5120 MB/s</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>32 KB per core</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>256 KB per core</td>
</tr>
<tr>
<td>L3 Cache</td>
<td>12 MB per socket</td>
</tr>
<tr>
<td>CnC</td>
<td>1.01</td>
</tr>
<tr>
<td>MPI run-time</td>
<td>Intel MPI 5.0</td>
</tr>
<tr>
<td>Compiler</td>
<td>ICPC 13</td>
</tr>
</tbody>
</table>

Table 8.1: Experimental setup

8.2 Logical to Physical Process Mappings

Before analyzing our experimental results we first explain two important aspects of the Concurrent Collections run-time. First, the task affinity feature in PIPES
translates directly to the CnC **compute-on step tuner**. When specifying a particular task affinity in a PIPES program, each task instance is set to be executed on a particular MPI process. This is the task-to-process mapping, a logical mapping. However, to have a complete specification of a distributed memory program in PIPES each MPI process must be binded to a set of cores in the cluster. This function from process-to-cores is the physical mapping. Naturally, the mapping must match the distribution of processes within the allocation of nodes in the cluster.

The second important aspect regarding the logical-to-physical process mapping is that the CnC run-time does not perform actual network transfers of data-blocks when the producer task and consumer task of the block in question are mapped to processes that execute within the same node. A special case of this occurs when the producer and consumer processes are the same. Therefore, the actual communication time could be lower than expected.

### 8.3 Benchmark Description

To evaluate our framework we implemented two distributed matrix multiply algorithms, Cannon 2D and Johnson 3D. In addition, for Johnson we produced three variants by leveraging the coalesce construct as well as different block sizes so as to constraint the available parallelism along one dimension. To complement our study, we also consider two full stencils, Jacobi-2d 9pt and a Jacobi-3d 27pt, both with ghost-zones to minimize communication.

**Cannon**  The Cannon algorithm is implemented using a single task collection and three data block collections (A, B and C). Each task M(i,j,k) consumes blocks A(i,j,k), B(i,j,k) and C(i,j,k); and produces A((i-1)%N,j,k+1), B(i,(j-1)%8,k+1) and C(i,j,k+1).
We define regions that allow to perform the shifting and circular communication patterns: leftwise for A and upwards for B. This algorithm exhibits 2D parallelism. Therefore, the tasks are mapped to a 2D virtual topology $\text{TOPO}\{[i, j] : 0 \leq i < N ∧ 0 \leq j < 8\}$, where $N$ is the number of nodes. Each task instance $M(i,j,k)$ is mapped to processor $\text{TOPO}(i\%N,j\%8)$. The environment produces the first instance of each block collection, i.e., those identified by the tuple (*,*,0).

**Johnson 3d** The Johnson algorithm is implemented using two task collections ($M$ and $R$) and four data block collections ($A$, $B$, $C$, $D$), where $A$ and $B$ are the input matrices, $C$ is the block-wise product and $D$ is the reduction result. The $M$ tasks compute block-products ($C$ blocks) while consuming blocks from $A$ and $B$. The $R$ tasks consume blocks $C$ and $D$ and output a new block $D$. The environment produces all block instances of $A$ and $B$, as well as the initial value for $D$. This algorithm exhibits 3D parallelism. Therefore, the tasks are mapped to a 3D virtual topology $\text{TOPO}\{[i, j, k] : 0 \leq i < N ∧ 0 \leq j < 2 ∧ 0 \leq k < 4\}$, where $N$ is the number of nodes. Task instances $M(i,j,k)$ and $R(i,j,k)$ are mapped to processor $\text{TOPO}(i\%N,j\%2,k\%4)$.

**Johnson 2d** Similar to Johnson 3d, but using the coalesce construct and the same topology and mapping as in Cannon.

**Johnson 2.5d** Similar to Johnson 3d, but limiting the reduction dimension to only 2 or 4 task instances. This means that each task is computing larger blocks. The topology and task mappings are identical to that of Johnson 3d.
Johnson 2.5d + Coalesce  Similar to Johnson 2.5d, but using the coalesce construct to automatically fuse tasks instances from M and R that have a producer-consumer relation. Uses the same topology and mapping as Cannon.

Jacobi-2d 9pt  A 2d full stencil. Uses 2 task collections, U (update task) and E (extract ghost), and two block collections, A and G. A represents the stencil data and G the ghost zones that must be exchanged along each spatial dimension. U, E and A are identified by 3d tuples from the set \( \{ [t, i, j] : 0 \leq t < 4 \land 0 \leq i, j < M \} \), whereas G has a fourth dimension \( g \in 0..3 \). Each task \( U(t, i, j) \) consumes a block \( A[t, i, j] \) and 2-to-4 instances of \( G[t, i, j, g] \), depending on the position of the task. i.e., whether it is a center, corner or border point in the task domain, and produces a new block, \( A[t+1, i, j] \). Each task \( E(t, i, j) \) also consumes a block instance \( A[t, i, j] \) and produces one or more ghost blocks for the current time iteration, depending on tuple identifier, which defines the logical location of the task in its domain. We use the same 2d topology as in Cannon, but with the task mapping \( \text{TOPO}(i\%N, j\%8) \).

Jacobi-3d 27pt  Similar to Jacobi-2d 9pt, but generalized to three space dimensions. It uses six ghost zones instead of four. The topology used is a 3D grid of dimensions \( M \times 2 \times 4 \), and maps the space dimensions of the tasks, e.g., \( (i, j, k) \) to \( \text{TOPO}(i\%N, j\%2, k\%4) \).

8.4 Communication and Computation Bounds

We now proceed to discuss the features of our benchmark set. Table 8.2 shows for each benchmark the problem sizes used in the experiments, the block size of the best performing variant, together with the number of processes that achieved this
performance. We also show the number of tasks launched, which depends of the
algorithmic nature as well as the problem size and the block size.

**Problem Size**  Is the problem size along a single dimension of the original PIPES
data-flow program. All distributed matrix-multiply variants use 3D domains and
the same problem size along all dimensions. Jacobi-2D and Jacobi-3D are square and
cubic on the space dimensions and execute for 4 time steps (first data-flow dimension).

**Block Size**  Is the tile size used for task coarsening. The block size shown is that
with the best achieved performance. The same block size is used for all dimensions and
all benchmarks, except for the Johnson’s 2.5d variants (with and without coalescing).
For these two variants a much larger tile size is used, such that the number of resulting
blocks along the chosen dimension is 2 or 4. For the Jacobi benchmarks, tile size of
1 is used for the leading data-flow dimension.

**Processes**  Is the total number of processes used during the execution that achieved
the best performance. Each node has two processors, and four cores per processor.
Only one process is launched per core.

**#Tasks**  Is the total number of tasks launched after applying the coarsening trans-
formation.

**#Blocks per Task**  is the number of data-blocks that are accessed using the intra-
node bandwidth, both read and written.
# Transfers is the number of inter-node data transfers performed by a single task. In the case where the program consists of 2 or more task collections, it represents the sum of data-transfers performed by the tasks with the largest number of instances.

**#Tasks per process** The number of tasks per process for each benchmark is determined by the number of processes used, the number of tasks that can execute in parallel and the longest chain of tasks in the program. In other words, it is the number of tasks that each process has to execute. Consider the Cannon algorithm first. The best performing variant uses block sizes of 1000$^3$. However, this algorithm exhibits $O(n^2)$ parallelism and uses 64 processes for executing 512 tasks. Thus, each process executes 8 tasks. Now consider the Jacobi-3D stencil. The total number of tasks launched is 1728 ($4 \times (384/64)^2 \times 2$ tasks). Thus, each process executes 54 tasks ($1728/32$).

**Giga FP per process** The total number of floating point operations per process is shown in this row. It is computed by the formula:

$$8 \times \#\text{tasks\_per\_proc} \times FP\_\text{per\_point} \times points\_per\_task/10^9$$

**Block Volume** Is the size in MB of a data block. It is computed by the formula

$$(\prod tile\_size_i) \times 4\text{bytes}/2^{20}\text{bytes\_per\_MB}.$$ 

**Inter-node Vol.** The inter-node volume (per node) is the product:

$$\#\text{tasks\_per\_proc} \times block\_volume \times \#\text{transfers} \times 8 \times proc\_per\_node$$

Here we assume that all processes contend for accessing the network device, therefore the factor 8.
**Intra-node Vol.** The intra-node volume (per node) is the product

\[
\#\text{tasks}_\text{per proc} \times \text{block volume} \times \#\text{blocks}_\text{per task} \times 8 \times \text{proc per node}
\]

Here we also assume that all processes contend for accessing the network device, therefore the factor 8.

**Operational Intensity** At this point we also compute the operational intensity, defined as the total number of flops executed by a single node divided by the number of bytes read and written for the tasks executed on the same node:

\[
giga\text{FP per node} \times 10^9 / (\text{intra node volume} \times 2^{20})
\]

**Performance per node** The maximal attainable performance in GF/s per node is computed following a Roofline bounding approach for each type of network:

\[
\min(160, \text{bandwidth} \times \text{operational intensity}/1000)
\]

**Compute Peak** shows the maximum performance achievable per node. The next three rows, that is Perf. GbE, Perf. InfiniBand QDR and Intra-node Roofline are determined by

\[
\min(160, \text{intra node vol} \times \text{operational intensity}/1000)
\]

where the communication time depends on the respective network type.
<table>
<thead>
<tr>
<th>Problem Size</th>
<th>Cannon 3D</th>
<th>Johnson 2D</th>
<th>Johnson 2.5D coalesce</th>
<th>Cannon 3D</th>
<th>Johnson 2D</th>
<th>Johnson 2.5D coalesce</th>
<th>Cannon 3D</th>
<th>Johnson 2D</th>
<th>Johnson 2.5D coalesce</th>
<th>Cannon 3D</th>
<th>Johnson 2D</th>
<th>Johnson 2.5D coalesce</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block Size</td>
<td>8000</td>
<td>8000</td>
<td>8000</td>
<td>8000</td>
<td>8000</td>
<td>8000</td>
<td>8192</td>
<td>384</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#Processes</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>2000</td>
<td>2000</td>
<td>2000</td>
<td>512</td>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#Tasks</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>32</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#Blocks per Task</td>
<td>512</td>
<td>1024</td>
<td>512</td>
<td>512</td>
<td>256</td>
<td>2048</td>
<td>1728</td>
<td>1728</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#Transfers</td>
<td>6.00</td>
<td>6.00</td>
<td>4.00</td>
<td>6.00</td>
<td>4.00</td>
<td>4.00</td>
<td>2.00</td>
<td>2.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#Tasks per Proc.</td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>12</td>
<td>6</td>
<td>64</td>
<td>54</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Giga FP per Proc.</td>
<td>192</td>
<td>192</td>
<td>96</td>
<td>288</td>
<td>288</td>
<td>6.96</td>
<td>8.25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 8.2: Communication and computation bounds

Performance all nodes (GF/s) and Peak Percentage

Next, the Machine Peak row accounts for the number of used nodes. The maximum performance is

5: The block size listed is that of the contracting dimension; the remaining two dimensions use the same tile size as Johnson, i.e., 1000.
determined by

\[ \min(\text{intra\_node\_roofline}, \text{best\_inter\_node\_perf}) \times \#\text{processes}/8 \]

The row **Measured** shows the best achieved performance and **Peak Percentage** shows the fraction of the peak achieved.

**Program Lines**  The last two lines of the table show the number of program lines of the PIPES input program and of the generated C++.

### 8.5 Results

We now proceed to discuss the results for each of the benchmarks. As previously mentioned, in every set of experiments we use all available cores. We start our analysis with the Cannon algorithm.

**Cannon Algorithm**  Figures 8.1 and 8.2 show the performance results when varying the number of nodes among 1, 2, 4 and 8 nodes (8, 16, 32 and 64 processes, respectively). Since Cannon has \(O(n^2)\) parallelism, it is natural for it to benefit more from smaller block sizes. However, larger block sizes also increase the operational intensity, the ration of flops computed per byte read. Larger blocks imply more data reuse within the kernel computation, which explains the increasing performance up to tile sizes of 2000\(^3\), at which point we start obtaining diminishing returns due to the lack of parallelism (there are only 16 tasks available for 64 processes). We also observe the effect of switching from the GbE to the IB network, particularly for the largest block size.
Figure 8.1: PIPES Cannon, problem size = 8000³, single precision, Gigabit Ethernet

Figure 8.2: PIPES Cannon, problem size = 8000³, single precision, InfiniBand network
**Johnson’s Algorithm**  The Johnson distributed-memory matrix-multiply algorithm represents the other extreme of the trade-off between memory and parallelism. Thus, in contrast to Cannon’s 2D algorithm, it benefits more from larger blocks rather than from smaller ones. The reason for this is the amount of data replication, a factor of $N/B$, where $N$ is the problem size and $B$ is the block size. As this ratio gets bigger, more parallelism is exploited at the cost of extra memory space. We can see in Figures 8.3 and 8.4 that using a block size of $500^2$ on two or less nodes produces excessively low performance. This is a direct result of a much larger critical path. Furthermore, since Johnson’s block-matrix computation is inherently parallel, small block sizes are not required to exploit sufficient parallelism. We also note that evaluating any block size smaller than $500^2$ results in many time-out executions, several of which took ten or more minutes. To conclude on Johnson, we make two final observations. First, the additional available bandwidth when using the InfiniBand network is not correctly exploited. This is expected because Johnson’s algorithm achieves the communication lower bound [112], and the only communication accounted occurs during the block-reduction phase. Second, this result is somewhat counter intuitive w.r.t. [112], but it is explainable by the fact that we use as many MPI processes as cores are available for each node. Then, the inter-node bandwidth requirement becomes 66% of Cannon’s, but grows by a factor of 2.5x for the intra-node bandwidth (which also depends on the critical path) while the computing time remains essentially the same.
Figure 8.3: PIPES Johnson-3D, problem size = 8000\(^3\), single precision, Gigabit Ethernet

Figure 8.4: PIPES Johnson-3D, problem size = 8000\(^3\), single precision, InfiniBand network
Johnson 2D  A unique feature of the PIPES compiler is to generate (arbitrary) communication schemes, task and block placements as well as automatic coarsening of tasks and blocks and coalescing (fusion) of task instances. We thus leverage the latter to generate in a completely automatic fashion a 2D variant from Johnson’s input program. This is done by adding the `coalesce` construct and listing the two tasks that are to be fused. For this set of experiments we only consider the same set of tile sizes (tasks and blocks) as for the evaluation of Cannon and Johnson. Coalescing has several benefits. First, it removes burden from the run-time by reducing the number of tasks in flight. Second, data-blocks transferred in between originally disjoint tasks are automatically removed. This is possible because the data transfers in question become intra-task dependences, i.e., they are pushed into the task body. Finally, by slightly varying the task (and block) tile sizes, interesting communication patterns can be discovered. In particular, and as shown in Table 8.2, even having different communication volumes can derive in very similar performances. Johnson’s 2D for the current set of experiments only considers the same set of block sizes as for Cannon and Johnson. However, we note that by reducing the tile size along the accumulating dimension when invoking PIPES additional parallelism is exposed. This effectively enables to create a complete class of 2.5D algorithms, but starting from a 3D description instead of a 2D algorithm (e.g. Cannon’s algorithm). This will be the goal of the following set of experiments.
Figure 8.5: PIPES Johnson 2D, problem size = $8000^3$, single precision, Gigabit Ethernet

Figure 8.6: PIPES Johnson 2D, problem size = $8000^3$, single precision, InfiniBand network
Regarding the performance of this benchmark, we can observe trends very similar to those of Cannon (See Tables 8.5-8.6). The reason for this is that by fusing both tasks of Johnson’s 3D program, the dependence of the task that reduces (accumulates) the result of block-products removes a degree of parallelism. Thus, this variant benefits more from smaller block sizes that compensate for the serial dimension (dimension k) induced by fusing both tasks.

To conclude on the analysis of this benchmark, we note that although Johnson-2D exhibits the same degree of parallelism (and performance) as Cannon’s 2D, they differ vastly in the communication scheme. We recall that a key property of the Cannon algorithm is the circular shifting for blocks A and B, leftward for the former and upward for the latter, with wrap-around. Johnson’s 2D, in contrast, benefits from the fact that all input block instances have already been distributed before the task execution phase, i.e., written by the environment and pushed onto the consumer tasks.

**Johnson 2.5D** This variant of Johnson’s algorithm leverages the ability of varying the tile sizes. The net result is a shorter critical path (w.r.t. the original Johnson program) at the cost of a larger block. In this experiment we set the tile size of the reducing dimension to 2000. This also means that all the tasks belonging to the blockwise multiplication stage will have a longer computing time (approximately 2x larger) and a higher operational intensity (again, w.r.t. to Johnson-3D). This is observable in Table 8.2. Overall, this version of Johnson-2.5D achieves up to 600 GF/s on 8 nodes (64 processes) and exhibits a trend that follows more closely Johnson’s 3D than Johnson’s 2D. Tables 8.7-8.8 show these results.
Figure 8.7: PIPES Johnson 2.5D, problem size = 8000³, single precision, Gigabit Ethernet

Figure 8.8: PIPES Johnson 2.5D, problem size = 8000³, single precision, InfiniBand network

**Johnson 2.5D Coalesced**  This variant of Johnson’s algorithm builds upon the previous one and combines it with the coalesce construct to produce a 2.5D variant
that closely resembles that reported in [112]. By coalescing the two tasks of Johnson, we effectively divide the number of tasks in flight into half, thereby reducing the critical path and further increasing the operational intensity (See Table 8.2). This produces the second smallest inter-node communication volume and time for both types of network while simultaneously achieving the second lowest intra-node communication time and blocks with the greatest operational intensity among all the matrix-multiply variants. The measured GF/s are shown in tables 8.9-8.10. The best configuration for this variant yields performance speedups of 1.2x, 2x, 1.1x and 1.5x w.r.t Cannon’s 2D best, Johnson’s 3D best, Johnson’s 2D and Johnson’s 2.5D best (without coalescing) for a maximum of 935 GF/s (72% of the theoretical peak for 8 nodes).

**Figure 8.9:** PIPES Johnson 2.5D coalesced, problem size = 8000³, single precision, Gigabit Ethernet
Figure 8.10: PIPES Johnson 2.5D coalesced, problem size = 8000³, single precision, InfiniBand network

Figure 8.11: PIPES Jacobi 2D - 4 ghost zones, problem size = 4 × 8192², single precision, Gigabit Ethernet
The last two benchmark results that we analyze are from two classical stencil computations. Jacobi-3d and Jacobi-2d. Both benchmarks are inherently bandwidth bound. The former performs 33 flops per point, whereas the latter does only 11. Both benchmarks are executed for 4 time iterations and no sort of tiling is applied to maximize locality. It is evident from Figures 8.11-8.14 that both programs achieve near linear scaling with a very low-baseline on a single node.

The large critical path induced by the number of time iterations performed and block size configuration combined with the block size and contention for network access is the main limiting resource for both benchmarks. A theoretical 18 GF/s and 21 GF/s per node can be achieved for the 2D and 3D stencil, respectively. The performance of the kernel computations are a second order factor compared to the achievable inter-node performance. (See Table 8.2).
Both benchmarks also suffer a sudden plummet in between 4 and 8 nodes, producing performance results even below that achieved when using 2 nodes, approximately a 2× slowdown for 4× the resources. Further experiments such as evaluating the run-time overhead are required to determine the low performance and lack of scaling beyond 4 nodes.

![Graph showing performance results](image)

**Figure 8.13:** PIPES Jacobi 3D - 6 ghost zones, problem size = 4 \times 384^3, single precision, Gigabit Ethernet

Despite these results, there are a number of approaches that can be taken to optimize stencil-like programs in PIPES using the CnC back-end. The first it to resort to tiling techniques that enable time tiling, for instance, overlapped tiling. This however comes at an additional cost: increasing the number of ghost regions. The program variants implemented and presented here utilize as many ghost-blocks as faces has the stencils. Nevertheless, to correctly exploit a time-tiling technique such as overlapped tiling, more and larger ghost zones are required: the ghost-zone of each face of the stencil grows by the time tile size (number of iterations intended to be performed by
a single call to the task kernel). This region growing on the direction normal to each face also induces additional communications and all surrounding stencil boxes. In other words, in all 8 directions for the 2D-case, and in all 26 directions for the 3D-case. Further experiments are required to determine if the increase in synchronization is an affordable price for maximizing locality.

A second approach worth considering is the usage of the special values for the consumed_on and produced_on tuners, specifically the CnC::CONSUMER_LOCAL and CnC::PRODUCER_LOCAL values, which indicate that the consumer (respectively producer) of a data-block are local to the same process. Leveraging these features, however, requires partitioning the coarsened block domains among tasks that are known to be mapped to the same process or to different ones. This is left as future work.
8.6 Performance Debugging

In this section we discuss a number of techniques and compiler options that can be used for performance debugging of PIPES programs.

**Program Execution Time** The execution time of a PIPES program is easily broken down into 3 parts: the startup, which consists of the environment allocating the input data and performing the initial block distribution; the main program execution, where tasks perform the actual work; and the program termination, when the environment reads the blocks produced by the executed tasks, followed by the program’s cleanup phase. These three phases are each timed by using the flag `-time`.

**Communication Only Execution Time** A useful debugging technique is to nullify the task kernels, the function executed by each task in between the set of GET and PUT operations, in order to determine the total communication time, without naturally accounting any computation. The user is normally required to provide an external function that follows the prototype defined by PIPES. However, by using the compiler flag `-no-kernel` the task kernel prototypes are changed from `extern` functions, to local functions with empty bodies.

**Computation Only Execution Time** In a similar fashion, the PIPES compiler also provides an option for performing fine grained timing of the core computations. This is enabled by using the flag `-time-task-kernel`. In this case, a timing routine is invoked before and after the execution of the task kernel. Then, for each task we print the rank of the process associated to the task, the task’s name and tuple and the time elapsed during the kernel’s computation. This information can be used to
determine if the communication scheme of the distributed execution has a negative impact in the computation time. This can occur, for instance, when the task’s kernel footprint is larger than the available last-level cache (LLC), thereby inducing several LLC misses.

**Task and Block Placement**  The next debugging feature provided by PIPES is the logging of task and block placements. It works in a slightly different manner than the **Computation Only Execution Time**. By using the options `-print-task-location` and `-print-block-location`, each task and block tuner (i.e., CnC’s compute on, produced on and consumed on tuners) outputs to standard error the rank (process ID) to which a task or block has been assigned. The objective of this feature is to validate that the communication scheme of the distributed execution matches the algorithmic description.

**DAG Execution Dump**  The final debugging feature introduced in PIPES combines several of the above features. It is enabled by the flag `-dump-graph`. This feature works in two stages. The first stage consists of logging for each task instance the start and stop date of each accessor operation (GET or PUT), together with the task’s instance tuple, block name, block tuple and the process rank associated. The second stage is performed offline. A DAG is constructed by adding edges for every block transferred, from its producer to each of the consumers. Then a tool such as Graphviz can be used to produce a visual output. Obviously, due to the number of dynamic instances, this feature is most useful for small problems.
CHAPTER 9

Performance Modeling of Matrix-Multiply Algorithms for distributed CnC

In this chapter we focus on two widely used matrix-multiply algorithms, which were efficiently implemented on top of the Intel Concurrent Collections (CnC) framework. We derived an analytical model that predicts their computation and communication times, considering both intra-node and inter-node communication. Using the roofline approach [128] we construct performance bounds taking into account machine-specific parameters (e.g., bandwidth, frequency, etc.) and application-specific ones (e.g., block size, communication expressions, etc.).

9.1 Matrix Multiply Distributed-CnC Implementations

In this section we briefly describe Cannon’s and Johnson’s CnC implementations. We assume that all the data is initially at the node with rank-0, and that the matrix decomposition into data blocks is performed before the algorithm starts its execution. Furthermore: i) we do not assume that $p$ (the number of processors/ranks) is square or cube; ii) we add an additional program parameter, `block_size`, which defines the length of the block into which the matrices are decomposed; iii) for simplicity, we assume that `block_size` divides $N$, the matrix size. The values of `block_size` and $N$
determine the number of blocks into which each matrix is decomposed. We therefore can have more matrix blocks to compute than minimally required to use all available processors, giving us a degree of freedom in terms of parallelism implementation.

9.1.1 Cannon’s Implementation

Cannon’s CnC implementation uses 3 item collections, one for each set of data blocks, one step collection and one tag collection. Each step instance is uniquely identified by a triple \{i,j,k\}. Furthermore, since CnC programs should be written in DSA form the items consumed and produced are identified by the step tag. Before initiating the actual computation, blocks of matrices A, B and C should be made available to the CnC environment via calls to the \texttt{Put} method with the proper tag value. Thus, the environment breaks matrices A, B and C into data blocks of size \emph{block\_size}, and puts blocks into the appropriate item collection (including blocks for C). Figure 9.1 shows the code that executes each Cannon step: retrieves a data-block from each matrix, performs the block matrix-multiply, and performs the data movement (left-wise for A, up-wise for B, and depth-wise for C).

9.1.2 Johnson’s implementation

Johnson’s implementation utilizes two step collections. The first one (see Figure 9.2) performs a 3D-parallel block matrix-multiply, while the second performs a reduction along dimension K. As in Cannon’s case, each step instance of each collection is uniquely identified by a triple \{i,j,k\}. The environment produces all the data blocks of matrices A and B, together with the C-data block used during the reduction step (triples \{i,j,0\}). Notice that, unlike Cannon’s step, Johnson performs only two \texttt{Get} operations since it will produce its own C-block product. Each \{i,j,k\}
i = tag.i;
j = tag.j;
k = tag.k;

ctx.mat_A_blocks.get(Triple(i,j,k), block_ik_A);
ctx.mat_B_blocks.get(Triple(i,j,k), block_kj_B);
ctx.mat_C_blocks.get(Triple(i,j,k), block_ij_C);

float * A = (* block_ik_A).addr();
float * B = (* block_kj_B).addr();
float * C = (* block_ij_C).addr();

mm_kernel (block_size, A, B, C);

int next_k = k + 1;
int next_i = i - 1;
int next_j = j - 1;

Triple nextA = SHIFTA(i,j, next_k, num_blocks);
Triple nextB = SHIFTB(i,j, next_k, num_blocks);
Triple nextC = SHIFTC(i,j, next_k, num_blocks);

ctx.mat_A_blocks.put(nextA, block_ik_A);
ctx.mat_B_blocks.put(nextB, block_kj_B);
ctx.mat_C_blocks.put(nextC, block_ij_C);

**Figure 9.1:** Cannon’s CnC step

reduction step is 2D-parallel (see Figure 9.3), and consumes two C-blocks, the C-block produced by the block-multiply step identified by the triple \{i,j,k\} and the C-block reduced by the previous step of its own collection (reduction step \{i,j,k-1\}). The final blocks of matrix C, blocks \{i,j,num_blocks\} are then consumed by the environment and reassembled to produce matrix C. We note here that, depending on the MPI settings, the data blocks of matrices A and B can be assumed to have been distributed before the actual matrix-multiply steps initiate execution. For guaranteeing this, we set the I_MPI_INTRANODE_EAGER_THRESHOLD variable to a size bigger than the largest message. For the purpose of our experiments we set this value to 20MB.
i = tag.i;
j = tag.j;
k = tag.k;

ctx.mat_A_blocks.get(Triple(i,k,j), block_ik_A);
ctx.mat_B_blocks.get(Triple(k,j,i), block_kj_B);

float * A = (*block_ik_A).addr();
float * B = (*block_kj_B).addr();
float * C = (*block_ijk_C).addr();

mm_kernel (block_size, A, B, C);

ctx.mat_C_blocks.put(Triple(i,j,k), block_ijk_C);

**Figure 9.2:** Johnson’s CnC block-multiply step

Johnson’s reduction steps in Figure 9.3 can be executed as soon as both the previously reduced C-block and a new $C_{ijk}$ block from Figure 9.2 becomes available, i.e., a data-dependence. Regarding the DSA form, we observe that each step instance of Figure 9.2 produces a fresh $C_{ijk}$ block, uniquely identified by its \{i,j,k\} tag value. Similarly, in the reduction step, these blocks are uniquely identified by the same tag and are kept in sync. Each newly reduced C-block is associated to the next value along the k-dimension, which functions as a time-dimension.

### 9.1.3 Step kernel

The matrix multiplication between blocks used both implementations is performed by a single kernel function we implemented, which is tiled for L3 (12 MB per socket) and L1 cache (32 KB per core), unrolled by $4 \times$ along the k-dimension of the L1 tile, and decorated with SIMD and vector pragmas. The L3 tile sizes used throughout the
whole set of experiments were \( \{240 \times 256 \times 128\} \), while for the L1 tiles \( \{80 \times 128 \times 16\} \) were used.

### 9.2 Analytical Model

In this section we introduce an analytical model to estimate the execution time of distributed-CnC implementations. The model predicts the computation and communication time for the previously introduced CnC matrix-multiply implementations. We first describe the general model which is algorithmic independent, and then proceed to tailor it to each individual case.

#### 9.2.1 Base Model

Table 9.1 shows the input parameters to our general model. For simplicity, we assume that both algorithms handle square matrices, use single precision (4 bytes per value), that there is no over-subscription of ranks to cores, and that each MPI process
uses a single core. Despite these assumptions, the model can be easily extended to other scenarios. We separate the parameters into two sets, the problem dependent ones (top) and the platform/network specific ones (bottom).

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>Problem size (along one dimension)</td>
</tr>
<tr>
<td>B</td>
<td>Block size (along one dimension)</td>
</tr>
<tr>
<td>M</td>
<td>Number of nodes</td>
</tr>
<tr>
<td>P</td>
<td>Number of MPI processes/ranks</td>
</tr>
<tr>
<td>$BW_{intra}$</td>
<td>Intra-node bandwidth (MB/s)</td>
</tr>
<tr>
<td>$BW_{inter}$</td>
<td>Inter-node bandwidth (MB/s)</td>
</tr>
<tr>
<td>$c_{node}$</td>
<td>Cores per node</td>
</tr>
</tbody>
</table>

Table 9.1: Model input parameters

Table 9.2 shows the derived variables used in the general model. The top set of metrics relate to the problem decomposition, the middle set are performance metrics, and the bottom ones communication variables. These metrics are still used in an algorithm independent fashion.

Equations (9.1) and (9.2) show how the problem decomposition variables are derived. In general, these are problem specific but algorithm independent. Each matrix is decomposed into $b^2$ blocks of size $B^2$. The number of compute steps is a function of $b$. We note that although we have steps to execute, the order in which these are computed are defined by the algorithm properties as well as the CnC implementation.

\[ b = \frac{N}{B} \quad (9.1) \]

\[ \text{steps} = b^3 \quad (9.2) \]
<table>
<thead>
<tr>
<th>Metric Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b$</td>
<td>Number of data blocks per dimension (algorithm independent)</td>
</tr>
<tr>
<td>steps</td>
<td>Number of steps (algorithm independent)</td>
</tr>
<tr>
<td>$msgs$</td>
<td>Number of messages per step (algorithm dependent)</td>
</tr>
<tr>
<td>$F_{par}$</td>
<td>Parallel factor (algorithm dependent)</td>
</tr>
<tr>
<td>$F_{ser}$</td>
<td>Serial factor (algorithm dependent)</td>
</tr>
<tr>
<td>$W_{intra}$</td>
<td>Intra-socket work fraction (algorithm dependent)</td>
</tr>
<tr>
<td>$W_{inter}$</td>
<td>Inter-socket work fraction (algorithm dependent)</td>
</tr>
<tr>
<td>$T$</td>
<td>Program time</td>
</tr>
<tr>
<td>$T_{comp}$</td>
<td>Compute time</td>
</tr>
<tr>
<td>$T_{comm}$</td>
<td>Communication time</td>
</tr>
<tr>
<td>$T_{step}$</td>
<td>Step compute time</td>
</tr>
<tr>
<td>$T_{intra}$</td>
<td>Intra-socket communication time</td>
</tr>
<tr>
<td>$T_{inter}$</td>
<td>Inter-socket communication time</td>
</tr>
<tr>
<td>$L$</td>
<td>Message (data block) length in MB</td>
</tr>
<tr>
<td>$V$</td>
<td>Total communication volume (MB)</td>
</tr>
<tr>
<td>$V_{intra}$</td>
<td>Intra-socket communication volume (MB)</td>
</tr>
<tr>
<td>$V_{inter}$</td>
<td>Inter-socket communication volume (MB)</td>
</tr>
</tbody>
</table>

**Table 9.2:** General performance derived metrics

Our model follows the common computation and communication split, while also taking into account if communication can be restricted to within a socket/node, or if it requires a remote access. The total compute time is estimated by the product of the number of steps executed by a task and the projected kernel time of the step $T_{step}$. The formulas used for estimating each of the performance variables are listed in Equations (9.3)-(9.8), using terms defined later on.
\[
T = T_{\text{comp}} + T_{\text{comm}} 
\]
\[
T_{\text{comp}} = F_{\text{ser}} \times T_{\text{step}} 
\]
\[
T_{\text{step}} = \text{measured on the test machine} 
\]
\[
T_{\text{comm}} = T_{\text{intra}} + T_{\text{inter}} 
\]
\[
T_{\text{intra}} = \frac{V_{\text{intra}}}{BW_{\text{intra}}} 
\]
\[
T_{\text{inter}} = \frac{V_{\text{inter}}}{BW_{\text{inter}}} 
\]

As \(T_{\text{step}}\) is the time to execute one single step on a single core, it is very practical
to measure this time on the target machine. For simplicity purpose we assume \(T_{\text{step}}\)
does not vary with the execution context (block size, etc.) which, as we show later
on, is a valid approach in our experimental setup. The general problem of predicting
DGEMM kernel performance analytically (e.g., [92]) is a difficult task, and out of the
scope of this work which focuses on proper modeling of communication times.

The communication volume is broken down into local (intra-node) and remote
(inter-node) communication. Both volumes are obtained from the total communica-
tion volume \(V\), by applying the respective work factor \(W_{\text{intra}}\) or \(W_{\text{inter}}\). These
factors are highly dependent of the algorithm and of the node and core distribution
(e.g., block, cyclic, etc). In particular, we assume that the mapping of ranks to nodes
and cores follows a block distribution. The total communication volume is the prod-
uct of the number of steps executed by a rank \(F_{\text{ser}}\), the message size \(L\) and the
number of messages per step \(\text{msgs}\). The reason to include this factor is to model
accesses of different ranks sharing a network device. Equations (9.9)-(9.12) show how
we derive the communication related variables, using terms defined later on in an
algorithmic dependent way. We remark that for utmost precision one can refine this model by considering different bandwidth for local communications between processes mapped to cores on the same socket versus on different sockets, however for simplicity purpose we assume a single intra-node bandwidth here.

\[ V_{\text{intra}} = V \times W_{\text{intra}} \]  
(9.9)

\[ V_{\text{inter}} = V \times W_{\text{inter}} \]  
(9.10)

\[ V = F_{\text{ser}} \times L \times \text{msgs} \times F_{\text{overhead}} \]  
(9.11)

\[ L = 4B^2/2^{20} \]  
(9.12)

### 9.2.2 Cannon Model Specifics

We now define the model variables specific to Cannon’s algorithm. This algorithm has \(O(b^2)\) parallelism and executes \(b\) stages. Each step performs three Get operations. Hence \(\text{msgs} = 3\). The factor \(W_{\text{intra}}\) can be interpreted as follows: if only one node is used, then the communication volume is affected by a factor equal to \(P \times C\), where \(C\) is a contention factor representing the number of cores accessing the shared memory. Otherwise, the contention will be for the network device, and the factor becomes the number of cores per node. The additional \(1/3\) factor comes from only having to access the blocks of matrix B from another node. C-blocks will always be local, and \(C_{\text{node}} - 1\) cores will access an A-block from the same node (due to the left-wise shift). For not over complicating the formula, we leave it as 1 for the A-blocks. Equations (9.13)-(9.18) define these metrics. This assumption is only valid when the distribution
of processes to cores is done in a block fashion.

\[\text{msgs} = 3\]  \hspace{1cm} (9.13)

\[F_{\text{overhead}} = 1.5\]  \hspace{1cm} (9.14)

\[F_{\text{par}} = \min(P, b^2)\]  \hspace{1cm} (9.15)

\[F_{\text{ser}} = \left\lceil \frac{\text{steps}}{F_{\text{par}}} \right\rceil\]  \hspace{1cm} (9.16)

\[W_{\text{intra}} = \min(P, c_{\text{node}}) \times (\min(P, c_{\text{node}}) - 1)\]  \hspace{1cm} (9.17)

\[W_{\text{inter}} = \begin{cases} c_{\text{node}} \frac{3}{3} & \text{if } (P > c_{\text{node}}) \\ 0 & \text{else} \end{cases}\]  \hspace{1cm} (9.18)

### 9.2.3 Johnson Model Specifics

We now proceed to define the specific variables of the Johnson CnC implementation. It consists of two step collections: the \(O(b^3)\)-parallel block matrix multiply, and the \(O(b^2)\)-parallel block reduction step, performed along dimension \(k\). We note that for simplicity purposes, the compute time of the reduction step is ignored due to its lower complexity w.r.t. the block matrix-multiply step (\(O(B^2)\) vs. \(O(B^3)\) computations). Thus, the compute time of both models will be identical for the same set of parameters. Both step collections consist of \(\text{steps}\) instances. The \(\text{min}\) function models the possibility of having fewer blocks than processes along a particular dimension. We only model the communication happening in the reduction stage, since we assumed that the block matrix-multiply step already had received all the required input blocks before they start execution. The communication distributing factors, \(W_{\text{intra}}\) and \(W_{\text{inter}}\), essentially say that if the communication is happening in a single node, then the \(W_{\text{intra}}\) of the Cannon CnC implementation applies, otherwise the original communication volume is halved, since the previously reduced C-block
will always remain in the same step. Equations (9.19)-(9.24) define the formulae used to derive these metrics.

\[ \text{msgs} = 2 \quad (9.19) \]
\[ F_{\text{overhead}} = 1 \quad (9.20) \]
\[ F_{\text{par}} = \min(P, b^3) \quad (9.21) \]
\[ F_{\text{ser}} = \lceil \frac{\text{steps}}{F_{\text{par}}} \rceil \quad (9.22) \]
\[ W_{\text{intra}} = \min(P, c_{\text{node}}) \times (\min(P, c_{\text{node}}) - 1) \quad (9.23) \]
\[ W_{\text{inter}} = \begin{cases} c_{\text{node}}/2 & \text{if } P > c_{\text{node}} \\ 0 & \text{else} \end{cases} \quad (9.24) \]

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>4</td>
</tr>
<tr>
<td>Processor</td>
<td>Intel Xeon E5630 @ 2.5 GHz</td>
</tr>
<tr>
<td>Sockets per node</td>
<td>2</td>
</tr>
<tr>
<td>Cores per socket</td>
<td>4</td>
</tr>
<tr>
<td>Intra-node bandwidth</td>
<td>25000 MB/s</td>
</tr>
<tr>
<td>Inter-node bandwidth</td>
<td>1250 MB/s</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>32 KB per core</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>256 KB per core</td>
</tr>
<tr>
<td>L3 Cache</td>
<td>12 MB per socket</td>
</tr>
<tr>
<td>CnC</td>
<td>1.01</td>
</tr>
<tr>
<td>MPI run-time</td>
<td>Intel MPI 5.0</td>
</tr>
<tr>
<td>Compiler</td>
<td>ICPC 13</td>
</tr>
</tbody>
</table>

**Table 9.3:** Experimental setup
9.3 Experimental Results

In this section we validate our analytical model by comparing the estimated compute time and communication time to the measured values the Cannon and Johnson CnC implementations.

<table>
<thead>
<tr>
<th>Software</th>
<th>Enabled options</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICPC</td>
<td>-xhost -O3 -fno-alias -std=c99</td>
</tr>
<tr>
<td>CNC</td>
<td>DIST_CNC=MPI CNC_NUM_THREADS=1 CNC_MPI_SPAWN=T</td>
</tr>
<tr>
<td>I_MPI</td>
<td>I_MPI_EAGER_THRESHOLD=20MB I_MPI_SHM_BYPASS=1 I_MPI_INTRANODE_EAGER_THRESHOLD=20MB I_MPI_DAPL_TRANSLATION_CACHE=0</td>
</tr>
<tr>
<td>Slurm</td>
<td>-nodes=M -ntasks=T -ntasks-per-socket=min(T, 4) -threads-per-core=1 -distribution=block -contiguous</td>
</tr>
</tbody>
</table>

Table 9.4: Software stack flags

<table>
<thead>
<tr>
<th>Configuration</th>
<th>M</th>
<th>P</th>
<th>Sockets</th>
</tr>
</thead>
<tbody>
<tr>
<td>map1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>map4</td>
<td>1</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>map8</td>
<td>1</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>map16</td>
<td>2</td>
<td>16</td>
<td>4</td>
</tr>
<tr>
<td>map32</td>
<td>4</td>
<td>32</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 9.5: Evaluated task mappings
Figure 9.4: Cannon compute time for N=2000

Figure 9.5: Cannon compute time for N=8000
**Figure 9.6:** Johnson compute time for N=2000

**Figure 9.7:** Johnson compute time for N=8000


9.3.1 Experimental setup

Table 9.3 summarizes the hardware setup and the software stack used to evaluate the two CnC implementations and to validate our analytical model.

Table 9.4 summarizes the non-default flags and options used for each layer of the software stack. Of special importance are the I_MPI flags that affect the choice of eager vs. rendezvous transfer protocol, and the disabling of the DAPL translation cache. The former enables the run-time to send messages as soon as they become available rather than waiting for the time at which a process requires them, i.e., these flags enable the run-time to favor a push model over a pull model. The relevance of these options is that they help in making more precise time estimations, e.g., for the Johnson implementation, where the input matrices have to be replicated and distributed, a program stage which we do not account in our experiments and modeling. Finally, the DAPL cache is disabled to avoid potential correctness errors.

For each CnC implementation we evaluate two problem sizes (N={2000,8000}), 2 block sizes (B={N/10,N/5}) and five task mappings. The task mappings are listed in Table 9.5. Lastly, throughout this section all x-labels follow the format "B[block-size]P[num-cores]".

9.3.2 Model fitting

We validate the accuracy of our model by comparing the estimated compute and communication time to the measured compute-only and communication-only execution times. The compute-only time is obtained by timing the kernel of each step
instance, adding and then averaging by the number of steps executed by each process. The communication-only time is measured when executing the original CnC program without invoking the kernel code.

Figures 9.4, 9.5, 9.6, and 9.7 show the fitting of the compute estimation time for both CnC programs. We observe that the estimated curve better matches the big problem size for both algorithms. In addition, the estimated time when using a single core is over-estimated since we do not take into account temporal data locality effects (e.g., reuse of data tiles between step instances) when the L3 does not need to be shared between processors. The curves on both implementations are identical for the same problem size because we decided to ignore the compute time of Johnson’s reduction step (it performs $O(n^2)$ operations).

![Cannon Communication Time - N = 2000](image)

**Figure 9.8:** Cannon communication time for N=2000
Figure 9.9: Cannon communication time for N=8000

Figure 9.10: Johnson communication time for N=2000

The following four figures (9.8 - 9.11) show how the estimated communication time matches the measured time. Overall we see that the model estimations follow closer
the big problem size than the small one. The reason for this is that the data block sizes (parameter B) used are for N=8000 produce bigger messages than for N=2000. This optimizes the payload-to-overhead ratio.

9.3.3 Compute and Bandwidth Bounds

We now use the previously estimated compute and communication times to observe whether we are compute or bandwidth bound. We plot both the estimated compute bandwidth bound for every work-decomposition and task-mapping configuration (i.e., the N, B, M and P parameters), since these define the compute time and the bandwidth requirements. We also plot the estimated final performance \((T = T_{\text{comp}} + T_{\text{comm}})\) from our analytical model and the measured final performance. Figures 9.12 to 9.15 show how the performance (GFLOPS) varies as one adds more compute resources.
Figure 9.12: Cannon perfbound model for N=2000

Figure 9.12 and Figure 9.14 show Cannon’s and Johnson’s performance bounds for N=2000, while Figure 9.13 and Figure 9.15 show the model for N=8000. One can observe that for \( \{N,B,P\} = \{2000,200,16-32\} \) both algorithms are close to be bandwidth bound. This happens because the ratio of operation per inter-node communication for \( B=200 \) is \( 100 = \frac{200^3 \times 2}{(200^2 \times 4)} \) as only one matrix block needs to be communicated inter-node. In addition, the inter-node bandwidth of 1250MB/s is shared between 8 cores. Thus the effective bandwidth is 156.25MB/s, leading to a bound of 16 GFLOPS per core. The theoretical machine peak assuming a vector width of 4 is about 19.2 GF/s. The estimated performance for these configurations is between 60-70 GFLOPS on 16 cores and 130-140 on 32 cores, yielding approximately 4.5 GFLOPS per core. This performance can be increased by improving the kernel
computation time, however our analysis shows it would not be useful (in terms of execution time) to improve the kernel performance beyond 16 GF/s single-core, unless incrementing the block size.

A similar reasoning can be employed to understand the performance bounds of both algorithms for N=8000 on Figures 9.13 and 9.15.

### 9.3.4 Exploratory Analysis

In this section we perform an analytical exploration to see how the performance bound of both CnC implementations vary when the kernel step would be optimized
to achieve a 4× faster execution time compared to our actual implementation used 9.3.3. In other words, we use our analytical model to determine whether it would be worth spending additional tuning time on the elementary block matrix-multiply operation, and what could be the expected gains.

Figures 9.16-9.17 show that such improvement in the kernel step time will cause the Cannon CnC implementation to be mostly bandwidth bound for N=2000, with the exceptions of P=1, and close to be bandwidth bound for N=8000 on P=16 and P=32. Regarding Johnson’s implementation, Figures 9.18-9.19 show that improving the computation bound by a factor of 4 narrows the gap w.r.t. the bandwidth curve. This program is still compute bound for N=8000, and partially bandwidth bound for N=2000, as previously discussed.
9.4 Related Work

Numerous previous works addressed the problem of minimizing and approximating the modeling of communications in distributed environments [128, 112, 28, 50, 84]. Others focused on analyzing and comparing pure MPI and hybrid MPI-OpenMP execution models, e.g. [4, 41, 107, 111, 62]. However, little attention has been given to developing analytical models that put into play intra-node communication and inter-node communication on top of a run-time with the capability of executing in pure MPI fashion, but which automatically avoids communication using shared memory when the data is local to a node such as the Intel Concurrent Collections runtime [66]. Furthermore there is a compelling need for a model with good predictive capability,
**Figure 9.16:** Cannon performance bound model for N=2000 assuming a 4x step kernel speedup that is which enables the analytical exploration of different block sizes or processor frequencies without executing power consuming runs on a large-scale cluster.
**Figure 9.17:** Cannon performance bound model for \(N=8000\) assuming a 4x step kernel speedup

**Figure 9.18:** Johnson performance-bound model for \(N=2000\) assuming a 4x step kernel speedup
Figure 9.19: Johnson performance-bound model for N=8000 assuming a 4x step kernel speedup
CHAPTER 10

Future Work

In this chapter we identify some of the potential future work both in terms of compiler infrastructure and research directions. The latter of these focuses on the three main contributions of this thesis, namely: i) the pipes compiler and distributed memory extensions; ii) converting the auto-paralllelization scheme presented in Chapter 4 to dynamic single assignment for execution on NUMA machines; and iii) formulating new constraints and performance objectives for vector codelet extraction.

10.1 Future Research

Task and Block Placements via Retiming

The first interesting research direction is to study the generation of task and block-placement schemes that minimize or avoid communication entirely while restricting the scope to stencil computations. The approach proposed in [113] can be used to explore various placing strategies via retiming operations. In this case tasks affinities (and therefore block placements too) can be fully or partially shifted along each dimension in order to minimize the producer/consumer communication volume, potentially converting inter-node accesses to intra-node accesses. The objective in this case would be to determine the family of
schedules that achieve an optimal task placement, thereby maximizing locality and avoiding communication, but that simultaneously does not hinder parallelism.

**Distributed Memory Computing Optimizations** Second, to allow more extensive analytical auto-tuning, models such as the one presented in Chapter 9 could be automatically derived once the topology, task placement and the location of produced/consumed blocks are known. Furthermore, the decoupled nature of programs written using Concurrent Collections, task and data placements can be determined by targeting and favoring a number of objectives, for instance, by minimizing communication and/or energy consumption and simultaneously maximizing parallelism.

**Graph composition and decomposition** Another interesting direction of future work is to extend the current PIPES framework so as to enable the composition of different programs. Each input program would effectively represent an operator, for instance, a matrix-multiplication, a matrix-inversion or a Gaussian filter. The presence of originally multiple disjoint programs will raise several questions such as how, where and how much to fuse tasks. On the other hand, in some cases in might be beneficial to decompose an input program into multiple graphs (or context). This could be necessary when the number of task and block instances reaches a particular threshold for the run-time, making the bookkeeping process a performance bottleneck.

**Codelet Extensions** Here we expand on two different research lines for vector codelets. First, combining the codelet extraction approach with the recent result from [113]. Stock et al. proposed an effective technique to minimize register spilling on high-order stencils. In a nutshell, a multi-dimensional retiming technique is applied
to the input program with the goal of performing less register reads but more register writes. The technique is proven effective, but extensive and systematic autotuning is required to obtain optimal performance. Thus, as a following step we propose to conduct a study to characterize the performance distribution of different stencils using different retiming configurations. This study would serve as the basis to design a new model driven approach to extract maximal vector codelets that minimize register spilling in addition to the already considered performance objectives. Second, adding new constraints and performance objectives to leverage emerging vector-SIMD instructions such as FMAs (fused-multiply adds) or gathering/scattering operations that rearrange data within a vector register, and allow strides larger than size 1.

Enabling Task Parallelism on NUMA Architectures  The work discussed in Chapter 4 focused on exposing the available parallelism and replacing OpenMP barrier synchronizations by point-to-point data-flow barriers. However, the approach taken leveraged the fact that the target machines were shared-memory non-NUMA architectures. Thus, explicit data-copying was not necessary and only streams of tokens were utilized. Recent results [40] have shown the effectiveness of dependence-aware schedulers, whereby a task is scheduled to execute in the same location (or close) to where the data is produced. The complexity here lies in generating efficient code that populates the data-stream while not violating the dynamic single assignment property of the run-time. The intra-task code must be adequately transformed to read from the incoming stream and potentially populating various output streams with the same data. Furthermore, the token-based synchronization approach described
in this work is not generalized to expose all the task parallelism available on the tile dimensions, rather at most the two outermost dimensions.

10.2 Compiler Framework

The work presented throughout this thesis spans a variety of compiler frameworks, namely the Polyhedral Compiler Collection (PoCC) [97], PolyOpt/C (a polyhedral engine for the Rose Compiler Framework) [2] and PPCG [124]. In the spirit of having a unified optimizing framework the PIPES compiler will be extended to integrate the domain partitioning and dependence signature identification that targets the OpenStream run-time. Thus, allowing creating both a CnC program from a sequential C variant or an OpenStream program from a data-flow representation. In addition, a back-end option to generate code that targets the Open Community Run-Time (OCR) is partially supported.

The work performed both on Chapters 4 and 5 have the downside that the intra-task bodies are either not generated or are unoptimized. Thus, it is necessary to integrate a intra-task optimizer such as the vector-codelet extraction to the framework or alternatively, to generate them via the fine-grained data-flow specification already at hand.
CHAPTER 11

Conclusion

The work discussed introduces various techniques to extract, express and map task-parallel and data-flow models to hardware and software layers at various levels of the stack: single-node single-core optimizations, single-node multi-core (shared-memory) and multi-node (distributed-memory) using polyhedral compilation frameworks. At the single-node single-core level the codelet extraction mechanism allows to transform the input program so that multi-dimensional reuse and inner-most loop parallelism is exploited. At the single-node multi-core level, automatic coarsening, removal of synchronization barriers and the generation of point-to-point synchronizations from a sequential program is possible and beneficial with the domain partitioning technique presented. At the multi-node level, PIPES is a new powerful and expressive language and compiler capable of generating an optimized CnC program from a fine-grained data-flow representation. The language allows to explore various communication schemes, task and data placement as well as task scheduling thereby attaining performance comparable or superior to programming models such as MPI and simultaneously augmenting productivity. The analytical models proposed allow
to perform auto-tuning in a systematic fashion for the distributed matrix-multiply algorithms implemented on Concurrent Collection (CnC) by accurately estimating the execution time while distinguishing between intra-node and inter-node data accesses.
APPENDIX A

OpenStream Generated Code

Listing A.1: OpenStream generated code for GEMM benchmark

```c
/* OSP-START */
#define min(x,y)   (((x) < (y) ? (x) : (y))
#define floord(n,d) (((n)<0)?(-(n)+(d)-1)/(d)) : (n)/(d))

int n tilesC = floord(ni - 1, 32)+1;
int s_b1b2_C[ntilesC] __attribute__((stream));
int w_b1b2_C[ntilesC];
int r_b1b2_C[ntilesC];
int a,b;
{
    for (long c1 = 0; c1 <= floord(ni - 1, 32); c1++)
        #pragma omp task output (s_b1b2_C[c1] << a)
            for (long c2 = 0; c2 <= floord(nj - 1, 32); c2++)
                for (long c3 = 32 * c1; c3 <= min(ni - 1, 32 * c1 + 31); c3++)
                    for (long c4 = 32 * c2; c4 <= min(nj - 1, 32 * c2 + 31); c4++)
                        C[c3][c4] *= beta;

    for (long c1 = 0; c1 <= floord(ni - 1, 32); c1++)
        #pragma omp task input (s_b1b2_C[c1] >> b)
            for (long c2 = 0; c2 <= floord(nj - 1, 32); c2++)
                for (long c3 = 0; c3 <= floord(nk - 1, 32); c3++)
                    for (long c4 = 32 * c1; c4 <= min(ni - 1, 32 * c1 + 31); c4++)
                        for (long c5 = 32 * c2; c5 <= min(nj - 1, 32 * c2 + 31); c5++)
                            for (long c6 = 32 * c3; c6 <= min(nk - 1, 32 * c3 + 31); c6++)
                                C[c4][c5] += ((alpha * A[c4][c6]) * B[c6][c5]);
}
#pragma omp taskwait

```
APPENDIX B

Pipes Examples

Listing B.1: PIPES program: Jacobi-3d stencil with ghost zone exchange (Part I)

```plaintext
// Parameters used to pass to context
Parameter T : 1..100;
Parameter N : 1..100000;
Parameter P : 4..256;

RA := { AA[t,i,j,k] : 1 <= i and i <= N and 1 <= j and j <= N
and 1 <= k and k <= N and 0 <= t and t <= T }
mempool {AA[0,i,j,k]};

RG := { GG[t,i,j,k,g] : 1 <= i and i <= N and 1 <= j and j <= N
and 1 <= k and k <= N and 0 <= t and t < T and 0 <= g and g < 6 }
space {GG[0,i,0,0,g]};

// 0 = -i
// 3 = +i
// 1 = -j
// 4 = +j
// 2 = -k
// 5 = +k

// Region subspace for reading GG blocks
RGJ(i,j,k) := {
[0] : i > 1;
[1] : j > 1;
[2] : k > 1;
[5] : k < N
};
```
Listing B.2: PIPES program: Jacobi-3d stencil with ghost zone exchange (Part II)

// Region subspace for producing GG blocks
RGG(i,j,k) := {
    [ii,jj,kk,3] : ii = i - 1 and i > 1 and jj = j and kk = k;
    [ii,jj,kk,0] : ii = i + 1 and i < N and jj = j and kk = k;
    [ii,jj,kk,4] : ii = i and jj = j - 1 and j > 1 and kk = k;
    [ii,jj,kk,1] : ii = i and jj = j + 1 and j < N and kk = k;
    [ii,jj,kk,5] : ii = i and jj = j and kk = k - 1 and k > 1;
    [ii,jj,kk,2] : ii = i and jj = j and kk = k + 1 and k < N
};

// Define block collections
Block [AA:RA] float;
Block [GG:RG] float;

Prescribe env :: ( TJJ :0..T-1,1..N,1..N,1..N);
Prescribe env :: ( TGG :0..T-1,1..N,1..N,1..N);

Produce env -> [AA:0,1..N,1..N,1..N];

Consume [AA:t,i,j,k] -> (TJJ:t,i,j,k);
Consume [GG:t,i,j,k,RGG(i,j,k)] -> (TJJ:t,i,j,k);
Produce (TJJ:t,i,j,k) -> [AA:t+1,i,j,k];

Consume [AA:t,i,j,k] -> (TGG:t,i,j,k);
Produce (TGG:t,i,j,k) -> [GG:t,RGG(i,j,k)];

Consume [AA:T,1..N,1..N,1..N] -> env;
Affinity (TJJ:t,i,j,k) -> [[G:i,j]];
Affinity (TGG:t,i,j,k) -> [[G:i,j]];
Listing B.3: PIPES generated tuning code for task TJJ (Part I)

// Affinity macros for task TJJ
#define PIPES_MAP_AFFINITY_TJJ_TO_G_0(N, P, T, t, i, j, k) ((i))
#define PIPES_MAP_AFFINITY_TJJ_TO_G_1(N, P, T, t, i, j, k) ((j))

PIPES_TASK_TUNER_CLASS_HEADER(TJJ) {
    int N;
    int P;
    int T;

    PIPES_TASK_TUNER_TYPE(TJJ) (int _N, int _P, int _T) {
        N = _N;
        P = _P;
        T = _T;
    }

    PIPES_TASK_TUNER_AFFINITY_HEADER(j3d_6g, TJJ) {
        // Insert LOG2PHYS function call here
        int kk0;
        int kk1;
        int kk2;
        int kk3;
        kk0 = local_tag[0];
        kk1 = local_tag[1];
        kk2 = local_tag[2];
        kk3 = local_tag[3];
        int pos0 = PIPES_MAP_AFFINITY_TJJ_TO_G_0(N, P, T, kk0, kk1, kk2, kk3);
        int pos1 = PIPES_MAP_AFFINITY_TJJ_TO_G_1(N, P, T, kk0, kk1, kk2, kk3);
        int ret;
        ret = pipes_log2phy(N, P, T, pos0, pos1);
        return ret;
    }
}
Listing B.4: PIPES generated tuning code for task TJJ (Part II)

```c
PIPES_TASK_TUNER_DEPENDENCE_HEADER(j3d_6g,TJJ) {
    // dependency consumer code
    int kk0;
    int kk1;
    int kk2;
    int kk3;
    kk0 = local_tag[0];
    kk1 = local_tag[1];
    kk2 = local_tag[2];
    kk3 = local_tag[3];
    if (T >= kk0 + 1 && kk0 >= 0 && kk1 >= 1 && N >= 64 * kk1 && kk2 >= 1 && N >= 64 * kk2 && kk3 >= 1 && N >= 64 * kk3 ) {
        if (kk0 >= 1) {
            PIPES_TASK_DEPENDENCE_PUT(PROGRAM_NAME,MP_AA,PIPES_TUPLE(-(kk0 % 2) + 1,kk1,kk2,kk3));
        }
        PIPES_TASK_DEPENDENCE_PUT(PROGRAM_NAME,AA,PIPES_TUPLE(kk0,kk1,kk2,kk3));
        if (2 * N >= 64 * kk1 + 32 * kk2 + 31 * kk3 + 2) {
            for (int c9 = max(0, -kk1 + 2); c9 <= min(1, kk2 - 1); c9 += 1) {
                PIPES_TASK_DEPENDENCE_PUT(PROGRAM_NAME,GG,PIPES_TUPLE(kk0,kk1,kk2,kk3,c9));
            }
            for (int c9 = max(2, -kk3 + 4); c9 <= min(3, N - 64 * kk1 + 2); c9 += 1) {
                PIPES_TASK_DEPENDENCE_PUT(PROGRAM_NAME,GG,PIPES_TUPLE(kk0,kk1,kk2,kk3,c9));
            }
            for (int c9 = max(4, -N + 64 * kk2 + 5); c9 <= min(5, N - 64 * kk3 + 4); c9 += 1) {
                PIPES_TASK_DEPENDENCE_PUT(PROGRAM_NAME,GG,PIPES_TUPLE(kk0,kk1,kk2,kk3,c9));
            }
        }
        if (kk0 == 0) {
            PIPES_TASK_DEPENDENCE_PUT(PROGRAM_NAME,MP_AA,PIPES_TUPLE(1,kk1,kk2,kk3));
        }
    }
}
#ifdef _DIST_
   PIPES_SERIALIZER(N&P&T)
#endif
};
```
Listing B.5: PIPES generated task code for task TJJ (Part I)

```c
// data block fetches
if (T >= kk0 + 1 && kk0 >= 0 && kk1 >= 1 && N >= 64 * kk1 && kk2 >= 1 && N >= 64 * kk2 && kk3 >= 1 && N >= 64 * kk3) {
    if (kk0 >= 1) {
        PIPES_BLOCK_COLLECTION_GET(PROGRAM_NAME, MP_AA, PIPES_TUPLE(-(kk0 % 2) + 1, kk1, kk2, kk3));
    }
    PIPES_BLOCK_COLLECTION_GET(PROGRAM_NAME, AA, PIPES_TUPLE(kk0, kk1, kk2, kk3));
    if (2 * N >= 64 * kk1 + 32 * kk2 + 31 * kk3 + 2) {
        for (int c9 = max(0, -kk1 + 2); c9 <= min(1, kk2 - 1); c9 += 1) {
            PIPES_BLOCK_COLLECTION_GET(PROGRAM_NAME, GG, PIPES_TUPLE(kk0, kk1, kk2, kk3, c9));
        }
        for (int c9 = max(2, -kk3 + 4); c9 <= min(3, N - 64 * kk1 + 2); c9 += 1) {
            PIPES_BLOCK_COLLECTION_GET(PROGRAM_NAME, GG, PIPES_TUPLE(kk0, kk1, kk2, kk3, c9));
        }
        for (int c9 = max(4, -N + 64 * kk2 + 5); c9 <= min(5, N - 64 * kk3 + 4); c9 += 1) {
            PIPES_BLOCK_COLLECTION_GET(PROGRAM_NAME, GG, PIPES_TUPLE(kk0, kk1, kk2, kk3, c9));
        }
    }
    if (kk0 == 0)
        PIPES_BLOCK_COLLECTION_GET(PROGRAM_NAME, MP_AA, PIPES_TUPLE(1, kk1, kk2, kk3));
}
```
Listing B.6: PIPES generated task code for task TJJ (Part II)

```c
// Task preputs: set tuples on freshly created blocks
if (T >= kk0 + 1 && kk0 == 0 && kk1 >= 1 && N >= 64 * kk1 && kk2 >= 1 && N >= 64 * kk2 && kk3 >= 1 && N >= 64 * kk3 ) {
    PIPES_BLOCK_COLLECTION_PRE_PUT(PROGRAM_NAME,AA,PIPES_TUPLE(kk0 + 1,kk1,kk2,kk3));
    PIPES_BLOCK_COLLECTION_MATCH_MEMPOOL(PROGRAM_NAME,AA,PIPES_TUPLE(kk0 + 1,kk1,kk2,kk3),0,2);
}

// task kernel invocation
if (N >= 1 && T >= kk0 + 1 && kk0 == 0 && kk1 >= 0 && N >= 64 * kk1 && kk2 >= 0 && N >= 64 * kk2 && kk3 >= 0 && N >= 64 * kk3 ) {
    PIPES_TASK_KERNEL_NAME(TJJ)(N,P,T,kk0,kk1,kk2,kk3,PIPES_INPUT(AA),PIPES_INPUT(GG),PIPES_OUTPUT(AA));
}

// send produced data blocks
if (T >= kk0 + 1 && kk0 == 0 && kk1 >= 1 && N >= 64 * kk1 && kk2 >= 1 && N >= 64 * kk2 && kk3 >= 1 && N >= 64 * kk3 ) {
    PIPES_BLOCK_COLLECTION_PUT(PROGRAM_NAME,AA,PIPES_TUPLE(kk0 + 1,kk1,kk2,kk3));
}
```


[52] GPCE. ACM conference on generative programming and component engineering.


[66] Intel. Intel Concurrent Collections for C++.


235


239


