THESE DE DOCTORAT DE
L'UNIVERSITE PIERRE ET MARIE CURIE

École Doctorale Informatique, Télécommunications et Électronique de paris

Improving Tiling, Reducing Compilation Time, and Extending the Scope of Polyhedral Compilation

Présentée par Riyadh BAGHDADI
Codirigée par Albert COHEN et Sven VERDOOLAEGE

soutenue le 25/09/2015

devant le jury composé de :

Rapporteurs	M. Cédric Bastoul	Professor, University of Strasbourg
	M. Paul H J Kelly	Professor, Imperial College London
Examinateurs	M. Stef Graillat	Professor, University Pierre et Marie Curie (Paris 6)
	M. Cédric Bastoul	Professor, University of Strasbourg
	M. Paul H J Kelly	Professor, Imperial College London
	M. J. Ramanujam	Professor, Louisiana State University
	M. Albert Cohen	Senior Research Scientist, INRIA, France
	M. Sven Verdoolaege	Associate Researcher, Polly Labs and KU Leuven

Contents

List of Figures

List of Tables

Nomenclature

Roman Symbols

3AC	Three-Address Code
ABF	Adaptive Beamformer
APIs	Application Program Interface
AST	Abstract Syntax Tree
BFS	Breadth-First Search
C11	C Standard 2011
C99	C Standard 1999
CUDA	Compute Unified Device Architecture
DSL	Domain-Specific Language
EBNF	Extended Backus-Naur Form
GCC	GNU Compiler Collection
GFLOPS	Giga FLoating-point Operations Per Second
GPU	Graphics Processing Unit
HMPP	Hybrid Multicore Parallel Programming
isl	Integer Set Library
OpenACC	Open Accelerators
OpenCL	Open Computing Language

OpenMP	Open Multi-Processing
pet	Polyhedral Extraction Tool
PGI	Portland Group, Inc.
PPCG	Polyhedral Parallel Code Generator
PRE	Partial Redundancy Elimination
PTX	Parallel Thread Execution
SESE	Single-Entry Single-Exit
SHOC	Scalable HeterOgeneous Computing
SSA	Static Single Assignment
VOBLA	Vehicle for Optimized Basic Linear Algebra

Chapter 1

Introduction and Background

1.1 Introduction

There is an increasing demand for computing power in many applications, ranging from scientific computing to gaming and embedded systems. This is leading the hardware manufacturers to design architectures with increasing degrees of parallelism and more complex memory hierarchies. A recent graphics card such as the Nvidia GeForce GTX Titan Z, for example, has a theoretical peak performance of 2707 GFLOPS (Giga FLoating-point Operations Per Second) in double precision, such a graphics card is higher in performance than the most powerful supercomputer listed on the TOP500 list of June 2000 [71], which had a peak performance of 2379 GFLOPS and is 4$\times$ higher in performance than the most powerful supercomputer listed on the TOP500 list of June 1997 [33].

With this increasing degree of parallelism and complex memory hierarchies, writing and debugging parallel programs is becoming more and more challenging. First, because thinking about the execution of parallel threads is not intuitive, which makes the process of writing parallel programs difficult and error prone. Second, getting good performance is not easy, especially when it comes to writing a program for an architecture that has a complex memory hierarchy and strict resource limits. Third, the problem of portability: a parallel program written and optimized for a specific architecture, cannot, in general, take advantage of the characteristics of another architecture without being modified (a different implementation of the program should be provided for each target architecture). These problems make parallel programming and code maintenance quite expensive in time and in resources.

An emerging solution to these problems is the use of higher level programming languages and the use of compilers to efficiently compile them into parallel and highly optimized low level code. There are many advantages of using a high level language. First, such a language is easy to use and more intuitive. Second, the parallel code is generated automatically by the

optimizing compiler which reduces the amount of bugs in the generated parallel code. Third, a high level language is architecture independent in general and thus it does not suffer from the problem of portability. Leaving the tedious, low level and architecture-specific optimizations to the compiler enables programmers to focus on higher level optimizations, leading to reduced development costs, shorter time-to-market and better optimized code.

Many approaches have been proposed to address the problem of lowering (compiling) a high level code into a parallel and highly optimized low level code. Most of the existing approaches focus on the optimization of loops (loop nests) since these are, in general, the most time consuming parts of a program. Early work in this area focused on the problem of data dependence analysis [15, 98], it was followed by the introduction of many loop transformations designed to enhance program performance, examples of these transformations include loop blocking (also known as tiling), which enhances data locality [12, 87, 108], vectorization, which transforms scalar operations into vector operations [1, 2], and automatic parallelization [16, 32, 34].

Although classical loop transformations are well understood in general, the design of a model that allows an effective application of a large set of loop transformations is not an easy task. Finding the right sequence of transformations that should be applied is even more challenging. To address these problems, researchers have investigated the possibility of using more expressive models and the possibility of deriving combinations of classical loop transformations using a single unified loop transformation framework. As a result of this research the polyhedral framework evolved [37, 38]. The polyhedral framework is an algebraic program representation and a set of analyses, transformations and code generation algorithms mainly dedicated to the optimization and automatic parallelization of loop nests. It enables a compiler to reason about advanced optimizations and program transformations addressing the parallelism and locality-enhancing challenges in a unified loop transformation framework. The polyhedral framework relies mainly on static analysis, although some efforts are being made to add support for dynamic analysis to the polyhedral model [5, 53, 93]. Static analysis is the process of extracting semantic information about a program at compile time whereas dynamic (or runtime) analysis extracts information at runtime. One classical example of static analysis is deciding whether two pointers *may-alias*: names a and b are said to *may-alias* each other at a program point if there exists a path P from the program entry to that program point, such that a and b refer to the same location after execution along path P.

In polyhedral compilation, an abstract mathematical representation is first extracted from the program, then data dependences are computed based on this representation. Loop transformations satisfying these dependences can be applied, and finally code is generated from the transformed polyhedral representation. Significant advances were made in the past two

decades on dependence analysis [36, 82], automatic loop transformation [40, 64, 65] and code generation [11, 54] in the polyhedral model. Examples of recent advances include automatic loop transformations and code generation for shared memory [18], distributed memory [17] and for complex architectures such as GPGPUs (General Purpose Graphics Processing Units) [10, 104].

Among the strengths of the polyhedral framework:

- It allows precise data dependence analysis and fine grain scheduling i.e., the ability to change the order of execution of only one or a few execution instances of a statement. An execution instance or an iteration of the statement `S` (the statement labeled with `S`) in the following loop nest

```
  for (int i = 1; i < n; i++)
S: A[i] = 0;
```

  is an execution of the statement in one of the iterations of the loop. Each iteration of the loop executes one instance of the statement.

- The polyhedral framework provides a unified framework in which a large set of transformations can be modeled and in which a compiler can reason about complex loop transformations.

- The algebraic representation of a program in the polyhedral model captures the semantics of the program. Loop transformations operate on this representation instead of operating on the syntax (code) of the program, and thus loop transformations are robust and would operate in the same way on programs that have the same semantics but are written differently.

Among the limitations of the polyhedral framework:

- Limitations that are common to all static analysis frameworks:

  – Static analysis frameworks do not accurately model programs with dynamic control. For example a conditional such as `if (A[i])` cannot be represented accurately and cannot be analyzed at compile time since the value of `A[i]` is only known at runtime. Many extensions to the classical polyhedral model were developed in order to overcome this limitation, some of these extensions use exit/control predicates and over-approximations to the iteration domain (the set of the execution instances of a statement) [14, 41], and other extensions encapsulate the dynamic control in macro-statements [102].

  - Automatic optimization and parallelization frameworks that rely on static analysis have limited ability in performing accurate dependence analysis and in optimizing code that manipulates dynamic data structures such as lists, graphs and trees (which all heavily use pointers). This is mainly because an effective optimization requires an accurate dependence analysis which in turn requires an accurate alias analysis, but alias analysis is undecidable in languages that permit recursive data structures (multilevel pointer indirection) [58, 86], thus the dependence analysis is not accurate in such a context which reduces the effectiveness of automatic optimization and parallelization.

- Limitations that are specific to the polyhedral framework:

  - Many algorithms in the polyhedral model are computationally expensive as they rely on the use of integer linear programming (existing algorithms to solve ILP problem have an exponential complexity in the worst case).

Although the polyhedral model has some weaknesses, it is showing promising results in many contexts and is being gradually included in industrial compilers such as IBM XL [19] and R-Stream [70]. The key to take full benefit from the polyhedral model is to use it in the right context. In this dissertation, we consider the following context: we do not address the problem of optimizing legacy code and we assume that the optimized code does not use pointers except in very few cases that we present in Chapter 4. We do not use the polyhedral model to optimize code that manipulates graphs, lists or trees, we rather use it in code that mainly manipulates arrays.

In this dissertation, we aim to widen the scope in which polyhedral loop optimizations can be used effectively. Our work was motivated by the use of the polyhedral model in two contexts:

- The use of the polyhedral model in the GCC compiler (the GRAPHITE polyhedral framework [81, 95]): we particularly address a classical technical problem in applying tiling in this context (tiling is an optimization that enhances data locality).

- The use of the polyhedral framework in DSL (Domain Specific Language) compilers: we particularly address two classical limitations (mentioned above) that became apparent in this context.

The work presented in this dissertation is an attempt to address these three problems. This work is often the result of collaborative research, development and experiments. We will acknowledge the key contributors and their respective roles in each technical chapter. In the

rest of this chapter we will introduce some key concepts and then we will provide an outline of the dissertation and a high level overview of the three problems that we address.

1.2 Background and Notations

Let us now introduce the basic concepts and notations used throughout the rest of the document.

1.2.1 Maps and Sets

The notations used in this dissertation for sets and maps are those used by ISL (Integer Set Library) [101] (these notations are also close to those used in the Omega project [109]).

i. Set

An integer *set* as defined in ISL is a set of integer tuples from $\mathbb{Z}^d$ that can be described using Presburger formulas [89] (presented later). d is the dimensionality of the set (the number of integers in each tuple). An n-tuple is represented in ISL with $[a_1, a_2, \ldots, a_n]$.

Supposing that we have a set of objects $\beta_1, \beta_2, \ldots, \beta_n$. We represent this set as follows: $\{\beta_1; \beta_2; \ldots; \beta_n\}$. An example of a set of integer tuples is:

$$\{[1,1]; [2,1]; [3,1]; [1,2]; [2,2]; [3,2]\}$$

Instead of listing all the integer tuples of the set, the integer tuples can be described using Presburger formulas (presented later). The two-dimensional set can be written as follows

$$\{S[i,j] : 1 \leq i \leq 3 \wedge 1 \leq j \leq 2\}$$

i and j are the the dimensions of the set. The tuples of a set can optionally have a common name: S in the example. Figure 1.1 shows a graphical representation of the set.

In general, an integer set has the form

$$S = \{N[\vec{s}] | f(\vec{s}, \vec{p})\}$$

with $\vec{s}$ representing the integer tuples of the integer set ($\vec{s} \in \mathbb{Z}^d$), N, a common name for all the tuples $\vec{s}$, d the dimensionality of the set, $\vec{p} \in \mathbb{Z}^e$ a vector of e parameters and $f(\vec{s}, \vec{p})$ a Presburger formula that evaluates to true, if and only if $\vec{s}$ is an element of S for the given parameters $\vec{p}$.

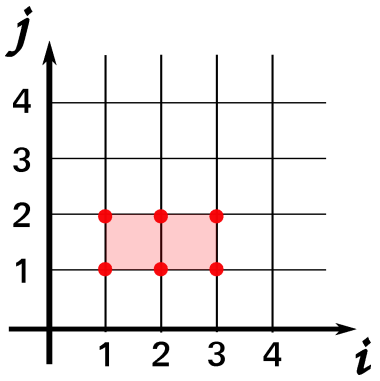

Figure 1.1 Graphical representation of a set

Presburger formula We define a Presburger formula using an EBNF (Extended Backus-Naur Form) grammar [106].

$$
\begin{array}{lcl}
\langle\text{formula}\rangle & \leftarrow & \langle\text{formula}\rangle \wedge \langle\text{formula}\rangle \mid \langle\text{formula}\rangle \vee \langle\text{formula}\rangle \\
 & & \mid \neg\langle\text{formula}\rangle \mid \exists\langle\text{var}\rangle.\langle\text{formula}\rangle \\
 & & \mid \forall\langle\text{var}\rangle.\langle\text{formula}\rangle \mid \langle\text{atom}\rangle \\
\langle\text{atom}\rangle & \leftarrow & \langle\text{term}\rangle\langle\text{relop}\rangle\langle\text{term}\rangle \\
\langle\text{term}\rangle & \leftarrow & \langle\text{numeral}\rangle \mid \langle\text{term}\rangle + \langle\text{term}\rangle \mid -\langle\text{term}\rangle \\
 & & \mid \langle\text{numeral}\rangle * \langle\text{term}\rangle \mid \langle\text{var}\rangle \\
\langle\text{relop}\rangle & \leftarrow & < \mid \leq \mid = \mid > \mid \geq \\
\langle\text{var}\rangle & \leftarrow & x \mid y \mid z \mid \ldots \\
\langle\text{numeral}\rangle & \leftarrow & 0 \mid 1 \mid 2 \mid \ldots
\end{array}
$$

$\langle\text{numeral}\rangle * \langle\text{term}\rangle$ is not a general multiplication operator; it is a short-cut for $\langle\text{term}\rangle + \cdots + \langle\text{term}\rangle$.

Presburger arithmetic (in which Presburger formulas are used) is used mainly because it is a decidable arithmetic. That is, there exists an algorithm which decides whether an arbitrary Presburger formula is true (valid) or not. The best known upper bound on the complexity of such an algorithm is $\mathcal{O}(2^{2^{2^{pn}}})$ where p is a positive constant and n is the length of the formula [80]. The length of a formula is the number of its atoms. An atom is an expression of one of the forms $(t_1 < t_2)$, $(t_1 \leq t_2)$, $(t_1 = t_2)$, $(t_1 > t_2)$ or $(t_1 \geq t_2)$ where t_1 and t_2 are terms.

Operations on sets. Examples of operations that can be done on sets include checking whether a set is empty, computing the union and the intersection of two sets and checking whether two sets are equal.

ii. Map

A map is a relation between two sets. For example

$$M = \{S1[i,j] \rightarrow S1[i+2, j+2] : 1 \leq i \leq 3 \wedge 1 \leq j \leq 2\}$$

represents a relation between two sets, the first is called the *domain* or the *source* and the second is called the *range* or the *sink*.

Figure 1.2 shows a graphical representation of the map M.

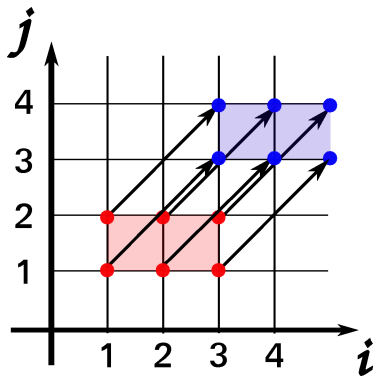

Figure 1.2 Graphical representation of a map

In general, a map has the form

$$M = \{A[\vec{s}] \rightarrow B[\vec{o}], (\vec{s}, \vec{o}) \in \mathbb{Z}^{d_1} \times \mathbb{Z}^{d_2} | f(\vec{s}, \vec{o}, \vec{p})\}$$

where $A[\vec{s}]$ represents the domain or the source and $B[\vec{o}]$ represents the range or the sink. d_1 and d_2 are the dimensionalities of $\vec{s}$ and $\vec{o}$, $\vec{p} \in \mathbb{Z}^e$ is a vector of e parameters and $f(\vec{s}, \vec{o}, \vec{p})$ is a Presburger formula that evaluates to true, if and only if there is a relation from $\vec{s}$ to $\vec{o}$ in M for the given parameters $\vec{p}$.

Among the operations on maps and sets, there is the operation of *application* of a map to a set. The application of the map `map1` to the set `set1` is written as `map1(set1)`. The function calls `Source(map1)` and `Sink(map1)` return respectively the source and sink of the map `map1`.

1.2.2 Additional Definitions

Lexicographic Order An iteration vector $[i_1, \ldots, i_k, \ldots, i_n]$ precedes another iteration vector $[i'_1, \ldots, i'_k, \ldots, i'_n]$ if and only if $\exists k \in \mathbb{N}$ such that $i_1 = i'_1 \wedge i_2 = i'_2 \wedge \cdots \wedge i_k < i'_k$. This is denoted as follows

$$[i_1, i_2, \ldots, i_k, \ldots, i_n] \ll [i'_1, i'_2, \ldots, i'_k, \ldots, i'_n]$$

Quasi-affine expressions A quasi-affine expression is a formula in Presburger arithmetic that cannot have quantifiers and that has the operators `==`, `!=`, `/`, `%` and the ternary `?:` operator in addition to the classical operations supported in Presburger arithmetic.

More explicitly, a quasi-affine expression is any expression over integer values and integer variables involving only the operators `+`, `-` (both unary and binary), `*`, `/`, `%`, `&&`, `||`, `<`, `<=`, `>`, `>=`, `==`, `!=` or the ternary `?:` operator. The second argument of the `/` and the `%` operators is required to be a (positive) integer literal, while at least one of the arguments of the `*` operator is required to be piecewise constant expression. An example of a quasi-affine expression is: `15*i+12*j+10`.

```
for (int i = 1; i < n; i++) {
  A[10*i+20] = 0;     // The subscript of A[] is quasi-affine
  A[i*n] = 0;         // The subscript of A[] is not quasi-affine
  B[i*i] = 0;         // The subscript of B[] is not quasi-affine
  C[t[i]] = 0;        // The subscript of C[] is not quasi-affine
  D[foo(i)] = 0;      // The subscript of D[] is not quasi-affine
}
```

Static-affine control In order to be able to extract the polyhedral representation, all loop bounds and conditions need to be quasi-affine expressions with respect to the loop iterators and a fixed set of *symbolic constants*, i.e., variables that have an unknown but fixed value throughout execution. A code is *static-affine* if it only contains control that respects this condition. In Chapter 2 we assume that the code that we treat is static-affine, but we do not assume this for Chapters 3 and 4.

1.2.3 Polyhedral Representation of Programs

In polyhedral compilation, an abstract mathematical representation is used to model programs, each statement in the program is represented using three pieces of information: an *iteration domain*, *access relations* and a *schedule*. This representation is first extracted from the program AST (Abstract Syntax Tree), it is then analyzed and transformed (this is the step where loop transformations are performed). Finally the polyhedral representation is converted back into an AST.

i. Iteration Domains

The *iteration domain* of a statement is a set that contains all the execution instances of that statement. Each execution instance (or dynamic execution) of the statement in the loop is

represented individually by an identifier for the statement (the name of the statement) and an *iteration vector*, which is tuple of integers that uniquely identifies the execution instance.

```
for (i=1; i<=2; ++i)
  for (j=1; j<=2; ++j)
S:    a[i] = 0;
```

Figure 1.3 Example of a kernel

The iteration domain for the statement S, called D_s, is the following set

$$D_s = \{S[1,1], S[1,2], S[2,1], S[2,2]\},$$

where each tuple in the set represents a possible value for the loop iterators i and j. The iteration domain for the statement S can also be written as follows

$$D_s = \{S[i,j] : 1 \leq i \leq 2 \wedge 1 \leq j \leq 2\}$$

We use D_s to refer to the iteration domain of a statement S.

Figure 1.4 shows a graphical representation of the iteration domain of the statement $S1$ in the following loop .

```
for (i=1; i<=n; ++i)
. for (j=1; j<=n; ++j)
. . if (i<=n-j+2)
S1:  . s[i] = ...
```

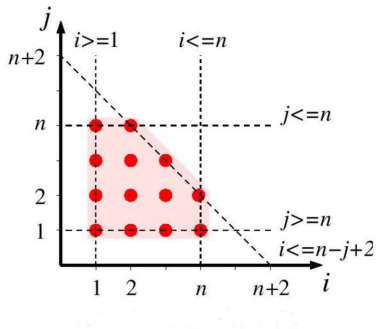

Figure 1.4 Iteration domain of `S1`

ii. Access Relations

Access relations are a set of read, write and may-write access relations that capture memory locations on which statements operate. They map statement execution instances to the array

elements that are read or written by those instances. They are used mainly in dependence analysis.

```
for (int i=1; i<=99; i++)
S:  A[i] = B[i];
```

In the previous example, the set of read access relations is

$$R = \{S[i] \rightarrow B[i] : i \geq 1 \wedge i \leq 99\}$$

which means that the statement S in iteration i reads the array element $B[i]$ and that this is true for each $i \geq 1 \wedge i \leq 99$.

The set of write access relations is

$$W = \{S[i] \rightarrow A[i] : i \geq 1 \wedge i \leq 99\}$$

and the set of may-write access relations (i.e., writes that may happen) is equal to the set of writes.

iii. Dependence Relations

Dependence. A statement S' *depends* on a statement S (we write $\delta_{S \rightarrow S'}$) if there exists a statement instance $S'(\vec{i'})$ and a statement instance $S(\vec{i})$ and a memory location m such that:

- $S'(\vec{i'})$ and $S(\vec{i})$ (may) access the same memory location m and at least one of them is a write access;
- $\vec{i'}$ and $\vec{i}$ respectively belong to the iteration domains of S and S';
- in the original sequential order, $S(\vec{i})$ is executed before $S'(\vec{i'})$.

Dependence Relation. Many formalisms can be used to represent dependences, some of these formalisms are precise and some formalisms are imprecise. Dependence relations are an example of precise formalisms and are the formalism that we use throughout this dissertation. They map statement instances (called *source statements*) to statement instances that depend on them for their execution (which are called *sink statements*). These dependence relations are derived from the access relations and the original execution order. A dependence relation is represented using a map and thus it has the following general form (the same general form of a map):

$$\delta_{S\to S'} = \{S[\vec{i}] \to S'[\vec{i'}], (\vec{i},\vec{i'}) \in \mathbb{Z}^{d_1} \times \mathbb{Z}^{d_2} | f(\vec{i},\vec{i'},\vec{p})\}$$

where $S[\vec{i}]$ represents the source statement and $S'[\vec{i'}]$ represents the sink statement. $\vec{p} \in \mathbb{Z}^e$ is a vector of e parameters and $f(\vec{i},\vec{i'},\vec{p})$ is a Presburger formula that evaluates to true if and only if $S'[\vec{i'}]$ depends on $S[\vec{i}]$ for the given parameters $\vec{p}$.

In the following example

```
for (int i=1; i<=99; i++)
S:  A[i] = A[i-1];
```

the value written in `A[i]` in the iteration (`i`) is read in the following iteration (`i+1`). This dependence can be represented using the following map:

$$\delta_{S\to S} = \{S[i] \to S[i+1] : i \geq 1 \wedge i \leq 98\}$$

Computing Dependences. Many tests have been designed for dependence checking between statement instances. Some of these tests give an exact solution and some of them only provide approximative but conservative solutions (they overestimate data dependences). Examples of these tests include the GCD-test [8] and the Banerjee test [9] which provide approximate solutions. The Omega-test [83] on the opposite is an example of tests that provide exact solutions.

Dependence Graph. The dependence graph $\Delta = (V,E)$ is a directed graph where the vertices V are the statements of the program and where the edges E are the dependence relations (maps) between these statements.

Classification of Dependences Program dependences can be classified into:

- Flow dependences (also known as true dependences and as read after write dependences), in which the write to a memory cell occurs before the read.
- Output dependences (also known as write after write dependences) in which both accesses are writes.
- Anti-dependences (also known as write after read dependences) in which the read occurs before the write.
- In some contexts (e.g., when discussing data locality), one may consider read after read dependences, which do not constrain parallelization.

Some flow dependences are spurious in general. Let us consider a memory cell and an execution instance of a statement that reads it (let us call it `S'`). There may be many writes to this cell. The only one on which `S'` really depends is the last one `S` which is executed before `S'`. The dependence from `S` to `S'` is a direct dependence. If the source program is regular, direct dependences can be computed using a technique such as array dataflow analysis [36].

iv. Schedule

While iteration domains define the set of all dynamic instances of an instruction, they do not describe the execution order of these instances. In order to define the execution order we need to assign an execution time (date) to each dynamic instance. This is done by constructing a *schedule* representing the relation between iteration vectors and time stamp vectors. The *schedule* determines the relative execution order of the statement instances. Program transformations are performed through modifications to the schedule.

A one-dimensional affine schedule for a statement S is an affine function defined by:

$$\theta_S(\vec{i}_s) = (c^s_1, c^s_2, \ldots, c^s_{m_s})(\vec{i}_s) + c^s_0 \tag{1.1}$$

where $c^s_0, c^s_1, c^s_2, \ldots, c^s_{m_s} \in \mathbb{Z}$ are called schedule coefficients and $\vec{i}_s$ is the iteration vector.

A multi-dimensional affine schedule can be represented as a sequence of such θ's for each schedule dimension. We use a superscript to denote the scheduling dimension. θ^k_s represents the schedule at the schedule dimension k for the statement S. If $1 \leq k \leq d$, all the θ^k_s can be represented by a single d-dimensional affine function T_s given by

$$T_s(\vec{i}_s) = M_s\vec{i}_s + \vec{t}_s,$$

where $M_s \in \mathbb{Z}^{d \times m_s}, \vec{t}_s \in \mathbb{Z}$.

$$T_s(\vec{i}) = \begin{pmatrix} \theta^1_s \\ \theta^2_s \\ \theta^3_s \\ \cdots \\ \theta^d_s \end{pmatrix} = \begin{pmatrix} c^s_{11} & c^s_{12} & \cdots & c^s_{1m_s} \\ c^s_{21} & c^s_{22} & \cdots & c^s_{2m_s} \\ c^s_{31} & c^s_{32} & \cdots & c^s_{3m_s} \\ \cdots & & & \\ c^s_{d1} & c^s_{d2} & \cdots & c^s_{dm_s} \end{pmatrix} \vec{i}_s + \begin{pmatrix} c^s_{10} \\ c^s_{20} \\ c^s_{30} \\ \cdots \\ c^s_{d0} \end{pmatrix}$$

The Pluto algorithm [18] is an example of an algorithm designed to find, for each statement in the program, an affine schedule that maximizes parallelism and data locality. It is an example of affine scheduling algorithms.

Static dimensions. If m_s is the dimensionality of the statement S and d is the dimensionality of T_s, it is possible in general that d is greater than m_s as some rows in T_s serve the purpose of representing partially fused or unfused dimensions at a level. Such a row has $(c_1, \ldots, c_{m_s}) = \vec{0}$, and a constant value for c_0. We call such a dimension a *static dimension*.

Dynamic dimensions. Any schedule dimension that is not a static dimension is a *dynamic dimension*.

1.2.4 Data Locality

The performance of a computer highly depends on the ability of the memory system to provide data to the processing unit at the required speed. Unfortunately the speed of microprocessors (processing units) is increasing much faster than the speed of memories, which creates a gap between the two. This is known as the *memory gap*. One solution to avoid the memory gap is the use of *cache memories* which are intermediate memories smaller and faster than the main memory. The idea is to take advantage of the speed of the cache memory by putting the most frequently used data in the cache: the frequently used data is loaded from the main memory to the cache, it is used many times and then it is copied back to the main memory.

During a given period of time, a given program in general accesses only a small fraction of their address space. This is called the principle of data locality. Data locality can be either:

- temporal locality: once a given data is accessed, it will tend to be accessed again during the next clock cycles;
- spatial locality: once a given data is accessed, nearby data in the address space will tend to be accessed during the next clock cycles.

Two main families of techniques can be used to improve data locality:

- data transformations: these techniques aim at changing the data layout;
- program transformations: these techniques aim at finding a better execution order for statements by modifying control statements. The goal of program transformations is to maximize temporal and/or spatial locality. Tiling is one of the main methods used in this context.

In the remaining of this section we will introduce tiling since this concept is used in Chapter 3.

Loop Tiling Tiling is a transformation that partitions a loop into blocks of computations in order to enhance data locality. The following code fragments show an example of tiling.

```
// Original code
for (i=0; i<1024; i+=1)
  for (j=0; j<1024; j+=1)
    c[i] = c[i] + a[i][j] * b[j];
```

```
// Tiled code
for (ti=0; ti<1024; ti+=4)
  for (tj=0; tj<1024; tj+=4)
    for (i=tj; i<ti+4; i+=1)
      for (j=tj; j<tj+4; j+=1)
        c[i] = c[i] + a[i][j] * b[j];
```

In the original code (left), each iteration of the outermost loop accesses the whole array `b[]` and one row of the array `a[][]`, this means that the whole array `b[]` and a row of the array `a[][]` will be loaded gradually from the main memory into the cache memory. Since the whole array `b[]` is read in each iteration of the outermost loop, it would be useful to keep such array in cache memory. If the size of the array `a[][]` is quite large, then loading this array into cache memory may overwrite the array `b[]` and thus `b[]` will need to be loaded for each iteration which is not efficient. To avoid this problem, one can apply tiling. In tiling (code on the right), only a small block (tile) of the array `b[]` is loaded into cache memory and is reused multiple times before the next block (tile) is loaded. This new access has the advantage of reusing data in the cache memory and thus it is much more efficient than the first access.

The classical criterion that a compiler must check before being able to apply tiling is the following: a *band* of consecutively nested loops is tilable if it has *forward* dependences only for every level within the band [18, 107]. In chapter 3 we extend this criterion so that we can tile codes that contain false dependences (dependences induced by memory reuse) and which cannot be tiled with the classical criterion.

1.2.5 Loop Transformations

Loop transformations are performed by changing the order of execution of statement instances. The goal is to transform the loops, while preserving the semantics of the original program, into a form that minimizes the execution time of the program or that minimizes the energy consumption, or to minimize/maximize any other objective function. In this dissertation, the main goal of the loop transformations that we apply is to minimize the execution time of the optimized program. Examples of loop transformations include the following.

Loop Distribution/Fusion. Loop distribution splits a single loop into multiple loops by distributing the statements of one loop across multiple loops. Each of the new loops has the same iteration domain as the original, but contains a subset of the statements of the original loop. This transformation is also known as `loop fission` or `loop splitting`.

```
// Original loops
for (i=0; i<N; i+=1)
  for (j=0; j<M; j+=1)
  {
    S1;
    S2;
  }
```

```
// Loops after distribution
for (i=0; i<N; i+=1)
  for (j=0; j<M; j+=1)
    S1;

for (i=0; i<N; i+=1)
  for (j=0; j<M; j+=1)
    S2;
```

Loop fusion is the opposite transformation. It fuses multiple loops into one loop.

Loop Interchange. Also known as permutation. It is a transformation that exchanges two loops in a loop nest.

```
// Original loops
for (i=0; i<N; i+=1)
  for (j=0; j<M; j+=1)
    S1;
```

```
// Loops after loop interchange
for (j=0; j<M; j+=1)
  for (i=0; i<N; i+=1)
    S1;
```

Loop Reversal. It is a transformation that reverses the order of execution of the iterations of a loop nest.

```
// Original loop
for (i=0; i<N; i+=1)
    S1;
```

```
// Loop after loop reversal
for (i=N-1; i>=0; i--)
    S1;
```

Loop Peeling. It is a transformation that peels some iterations of the loop and executes them separately.

```
// Original loop
for (i=0; i<=N; i+=1)
    S1[i]= 0;
```

```
// Loop after loop peeling
S1[0] = 0;
for (i=1; i<=N-1; i+=1)
  S1[i]= 0;
S1[N] = 0;
```

1.3 Outline

In this dissertation we address three problems:

- The first problem (Chapter 2) is related to the ability to apply loop tiling on code that has false dependences. The application of tiling is severely limited if the code contains false dependences. False dependences (dependences induced by the reuse of a memory

location multiple times) are frequently found in programs that are transformed by the compiler into three address code (3AC). Three-address code is an intermediate representation used by compilers to aid in the implementation of optimizations. An example of a polyhedral framework that operates on three-address code is the GRAPHITE framework used in GCC. While a transformation such as array expansion, which transforms program scalars into arrays and arrays into higher dimensional arrays, removes all false dependences and thus enables tiling, the overhead of such a transformation on memory and the detrimental impact on register-level reuse can be catastrophic. We propose and evaluate a technique that enables a compiler to safely ignore a large number of the false dependences of the program in order to enable loop tiling in the polyhedral model. This technique is based on a criterion that allows a compiler to identify which false dependences can be safely ignored during tiling and which cannot, and it does not incur any scalar or array expansion.

- The second problem (Chapter 3) is related to the long compilation time that one may experience when using polyhedral tools to optimize a program. Particularly, the long execution time of the Pluto scheduling algorithm [18], a widely used algorithm for automatic parallelization and locality optimization. We propose and evaluate a technique called *offline statement clustering*, a practical technique designed to reduce the execution time of the Pluto scheduling algorithm without much loss in optimization opportunities. Using this technique, the statements of the program are clustered into macro-statements, the Pluto affine scheduling algorithm is then used to schedule the macro-statements instead of scheduling the original statements of the program. Since the number of macro-statements is less than the number of statements in the original program, scheduling the macro-statements is, in general, faster than scheduling the original statements of the program.

- The third problem (Chapter 4) is related to two limitations:

  - The limited ability to optimize programs that contain dynamic control and which cannot be represented accurately in the polyhedral model.
  - The limited ability of the tools to generate efficient code and to do some optimizations due to the lack of some information. Deducing such information automatically by the compiler is difficult in general and thus such information needs to be provided by the programmer.

  To address these two limitations, we present PENCIL (Platform-Neutral Compute Intermediate Language), a language designed to simplify code analysis and code optimiza-

tion for the optimizing compiler. It is a rigorously-defined subset of GNU C99 [50] with specific programming rules and few extensions. Adherence to this subset and the use of these extensions enable compilers to exploit parallelism and to better optimize code when targeting accelerators such as GPGPUs. We intend PENCIL both as a language to program accelerators, and as an intermediate language for DSL (Domain Specific Language) compilers.

Chapter 2

Tiling Code With Memory-Based Dependences

2.1 Introduction and related work

To guarantee that loop transformations are correct, the compiler needs to preserve data-dependences between statements. Two types of data dependences exist, true (also known as data-flow) and false (also know as memory-based) dependences [55]. False dependences have two types: output-dependences and anti-dependences. They are induced by the reuse of temporary variables across statement instances. In the presence of these false dependences compilers may conservatively avoid the application of some loop transformations such as tiling. Tiling is possible when a *band* [1] of consecutively nested loops is fully permutable. A band is permutable if it is possible to permute two loop levels with each other in that band. This is possible if all the dependences within that band of consecutive loop nests are *forward* dependences only, i.e., dependences with non-negative distance for every loop level within the band [18, 107]. The presence of *backward* false dependences, i.e., dependences with a negative distance in some loop levels, prevents tiling. Backward false dependences are actually very common, especially in code that defines and uses scalar variables or array elements as in the example in Figure 2.1 where backward dependences are induced by the reads and writes to `s[t]`.

In this work, we introduce a new validity criterion for tiling that allows the tiling of loop nests even in the presence of some patterns of false dependences as in the example of Figure 2.1.

Scalar and array expansion [35] is a classical solution that can be used to remove false dependences and enable tiling. Expansion transforms scalars into arrays and transforms arrays

[1] The term band in the literature refers to a set of consecutive loop levels in a loop nest.

```
for (t = 0; t < nt; t++)
  for (i = 0; i < ni; i++)
    for (j = 0; j < nj; j++)
    {
S1:   s[t] = alpha * A[t][i][j] + beta;
S2:   B[t][i][j] = s[t];
    }
```

Figure 2.1 A code with false dependences that prevent tiling

into higher dimensional arrays and thus eliminates memory reuse. The terms expansion and privatization are sometimes used by the community interchangeably to refer to expansion. In this work we make a clear distinction between the two terms. In privatization [45, 63, 97], a private copy of the variables is created for each thread executing some iterations of the loop in parallel. In expansion, a private copy is created for each loop iteration [35]. This usage is consistent with the use of the two terms privatization and expansion in [68, 72]. In particular, privatization creates, for each thread cooperating on the execution of the loop, private copies of scalars (or arrays) [45, 63, 97]. The number of private copies is equal to the number of parallel threads. Although privatization is required to enable parallelism, it is not sufficient to enable tiling, because privatization only creates private copies of scalars and arrays for each thread, eliminating false dependences crossing the boundary of data-parallel blocks of iterations. Classical tiling techniques require the elimination of all (false) negative-distance dependences [18, 40, 49, 55], which is the domain of scalar or array expansion.

Although scalar and array expansion eliminates false dependences and enables loop tiling, it incurs high footprint on memory and degrades temporal locality [94]. Scalar expansion is particularly harmful as it converts register operations into memory operations [21]. A family of array contraction techniques attempts to reduce the memory footprint without constraining loop transformations [31, 61, 84]: the compiler performs a maximal expansion, applies loop transformations including tiling, and then attempts to contract the arrays. By performing a maximal expansion, false dependences are eliminated, but when unrestricted loop transformations are applied after expansion, contraction is not always possible as the set of simultaneously live values may effectively require high-dimensional arrays to store them. Loop distribution is an example of a widely applied loop transformation that prevents array contraction and thus disables the effectiveness of this technique as we show in an example in Section 2.7.

We propose and evaluate a technique allowing compilers to tile and parallelize loop nests, even in the presence of false dependences that would violate existing validity criteria for loop tiling. The technique enables the compiler to decide which false dependences can be safely ignored in the context of loop tiling. The proposed technique does not incur any costs of scalar or array expansion.

The main contributions are the following:

- We propose a relaxed permutability criterion that allows loop tiling to be applied without the need for any expansion or privatization, the impact on the memory footprint is minimized, and the overheads of array expansion are avoided.
- Our criterion allows the tiling of programs that privatization does not allow.

Section 2.2 presents the possible sources of false dependences. Section 2.3 shows that existing tiling algorithms are too restrictive and miss safe tiling opportunities. Section 2.4 introduces a new tiling test that avoids this problem and allows the tiling of kernels with false dependences. Section 2.5 proves that, by ignoring false dependences, tiling before 3AC transformation is equivalent to tiling after 3AC transformation. Sections 2.6 and 2.7 show an evaluation of the proposed technique.

2.2 Sources of False Dependences

One common source of false dependences is found in temporary variables introduced by programmers in the body of a loop.

A second source of false dependences is the compiler itself. Even in source programs that do not initially contain scalar variables, compiler passes may introduce false dependences when upstream transformations introduce scalar variables. This is a practical compiler construction issue, and a high priority for the effectiveness of loop optimization frameworks implemented in production compilers [96]. Among upstream compiler passes generating false dependences, we will concentrate on the most critical ones, as identified by ongoing development on optimization frameworks such as the GRAPHITE polyhedral loop optimization framework in GCC [81, 95]:

- Transformation to Three-Address Code (3AC). Three-address code is an intermediate code used by compilers to aid in the implementation of optimizations. Each 3AC instruction has at most three operands and is typically a combination of assignment and a binary operator. For example: `temp_1 := temp_2 + temp_3;`

  In 3AC transformation, each instruction that has more than three operands is transformed into a sequence of instructions, and temporary scalar variables are used to hold the values calculated by new instructions. The GRAPHITE polyhedral pass in GCC is an example of a loop optimization framework operating on low level three-address instructions. It is affected by the false dependences that result from a conversion to three-address code.

- Partial Redundancy Elimination (PRE) This is a transformation that can be applied to array operations to remove invariant loads and stores, and transform array accesses into scalar accesses [57, 74]. Figures 2.2 and 2.3 show an example of this optimization.
- Loop-invariant code motion is a common compiler optimization that moves loop-invariant code outside the loop body eliminating redundant calculations. An example is shown in Figures 2.4 and 2.5 where tiling of the outer two loops is inhibited by the extra dependences on `t` after the application of loop-invariant code motion.

```
for (i = 0; i < ni; i++)
  for (j = 0; j < nj; j++) {
S1:  C[i][j] = C[i][j] * beta;
      for (k = 0; k < nk; ++k)
S2:     C[i][j] += alpha * A[i][k] * B[k][j];
  }
```

Figure 2.2 *Gemm* kernel

```
for (i = 0; i < ni; i++)
  for (j = 0; j < nj; j++) {
S1:  t = C[i][j] * beta;
     for (k = 0; k < nk; k++)
S2:    t += alpha * A[i][k] * B[k][j];
S3:  C[i][j] = t;
    }
```

Figure 2.3 *Gemm* kernel after applying PRE

```
for (i = 0; i < ni; i++)
  for (j = 0; j < nj; j++)
    for (k = 0; k < nk; k++)
      A[i][j][k] = B[i][j] * C[i][j] * D[i][j];
```

Figure 2.4 Three nested loops with loop-invariant code

```
for (i = 0; i < ni; i++)
  for (j = 0; j < nj; j++) {
    t = B[i][j] * C[i][j] * D[i][j];
    for (k = 0; k < nk; k++)
      A[i][j][k] = t;
  }
```

Figure 2.5 Three nested loops after loop-invariant code motion

In the case of GCC and LLVM, performing loop optimizations directly on three-address code provides many benefits. Mainly because at this level also the representation is in Static Single Assignment form (SSA), where each variable is assigned exactly once, and every variable is defined before it is used. Many compiler optimizations such as constant propagation, dead code elimination and partial redundancy elimination are performed on the code when it is in SSA form. The main advantage of this is the improved performance of the algorithms [55]. Loop transformations on three-address code also enable a tight integration of loop optimization techniques with the other compilation passes, including automatic vectorization, parallelization and memory optimizations. The ideal case for an optimization framework is to operate on a representation that is low enough so that this optimization framework benefits from all the passes operating on three-address code, and at the same time the optimization framework should be able to ignore spurious false dependences generated by these passes. Note that transforming the code into SSA form removes some false dependences, but does not remove all the false dependences in the program.

Loop tiling is of utmost importance to exploit temporal locality [18]. Unfortunately, this transformation is also highly sensitive to the presence of false dependences. Although it is possible to perform PRE and loop-invariant code motion after tiling in some compiler frameworks such as LLVM. Transformation to 3AC must happen before any loop optimization in order to benefit from all the passes operating on three-address code, and this limits the chances to apply tiling.

Finally, enabling tiling on 3AC automatically extends its applicability to "native 3AC languages" and environments, such as Java bytecode and general-purpose accelerator languages such as PTX (which is a low-level parallel instruction set proposed by Nvidia). This may contribute to offering some level of performance portability to GPGPU programming and would have higher impact when integrated in the just-in-time compilers of languages such as PTX and Java bytecode [28].

Our goal is to propose and evaluate a technique allowing compilers to escape the problems described previously, providing a more relaxed criterion for loop tiling in the presence of false dependences. Such a validity criterion for tiling does not require a priori expansion of specific variables.

2.3 Motivating Example

The usual criterion to enable tiling is that none of the loops involved should carry a backward dependence (a dependence with a negative distance). The kernel presented in Figure 2.3 has backward dependences. These dependences stem from the fact that the same scalar `t` is over-

written in each iteration of the loop and enforce a sequential execution. In this example, the *forward* dependence condition prohibits the application of tiling, although the transformation is clearly valid as we show in Figure 2.6.

```
#define B 32
for (t1=0; t1<=floor((ni-1)/B); t1++)
  for (t2=0; t2<=floor((nj-1)/B); t2++)
    for (t3=B*t1; t3<=min(ni-1,B*t1+B-1); t3++)
      for (t4=B*t2; t4<=min(nj-1,B*t2+B-1); t4++)
      {
   S1:  t = C[t3][t4] * beta;
        for (t6=0; t6<nk; t6++)
   S2:    t = t + alpha * A[t3][t6] * B[t6][t4];
   S3:  C[t3][t4] = t;
      }
```

Figure 2.6 Tiled version of the *gemm* kernel.

Our goal is to enable the compiler to perform tiling on code that contains false dependences, by ignoring false dependences when this is possible, and not by eliminating them through array and scalar expansion. In the next section, we introduce the concept of live range non-interference. We use this concept later to justify the correctness of the proposed technique and to decide when is it safe to ignore false dependences.

2.4 Live Range Non-Interference

In this section we present the concept of *live range non-interference*, which is then used to establish a relaxed validity criterion for tiling.

2.4.1 Live ranges

We borrow some terminology from the register allocation literature where live ranges are widely studied [20, 24, 46], but we extend the definition of *live ranges* to capture the execution instances of statements instead of the statements in the program. In a given sequential program, a value is said to be *live* between its definition instance and its last use instance. Since tiling may change the order of statement instances, we conservatively consider live ranges between a definition and *any* use, which includes the last use under any reordering. The definition instance is called the *source* of the live range, marking its beginning, and the use is called the *sink*, marking the end of the live range.

$\{S_k(\vec{I}) \rightarrow S_{k'}(\vec{I'})\}$ defines an individual *live range* beginning with an instance of the write statement S_k and ending with an instance of the read statement $S_{k'}$. $\vec{I}$ and $\vec{I'}$ are iteration vectors

```
 for (i = 0; i < n; i++)
   for (j = 0; j < n; j++) {
S1:  t = A[i][j] * y_1[j];
S2:  x1[i] = x1[i] + t;
   }
```

Figure 2.7 One loop from the *mvt* kernel

identifying the specific instances of the two statements. For example, the scalar `t` of the *mvt* kernel shown in Figure 2.7 induces several live ranges, including

$$\{S_1[0,0] \to S_2[0,0]\}$$

This live range begins in the iteration $i = 0$, $j = 0$ and terminates in the same iteration.

Since the execution of a given program generally gives rise to a statically non-determinable number of live range instances, we cannot enumerate them individually, but rather we provide "classes" of live ranges defined with affine constraints. For example, the live range above is an instance of the following live range class

$$\{S_1[i,j] \to S_2[i,j] : 0 \leq i \leq n \wedge 0 \leq j \leq n\},$$

where $0 \leq i \leq n \wedge 0 \leq j \leq n$ are affine constraints defining the domain of i and j. For each iteration of i and j, there is a live range that begins with the write statement S_1 and terminates with the read statement S_2. To be able to describe such classes using affine constraints, we only consider loop nests with static-affine control.

2.4.2 Construction of live ranges

Live ranges form a subset of the read-after-write dependences. In the following, we assume that each sink in a dependence has only one source. We construct the live ranges using the array data-flow analysis described in [36] and implemented in the `isl` library [101] using parametric integer linear programming. Array data-flow analysis answers the following question: given a value v that is read from a memory cell m at a statement instance r, compute the statement instance w that is the source for the value v. The result is a (possibly non-convex) affine relation mapping read accesses to their source. The live ranges are then computed by reversing this relation, mapping sources to all their reads.

This systematic construction needs to be completed with the special treatment of live-in and live-out array elements. Array data-flow analysis provides the necessary information for live-in ranges, but inter-procedural analysis of array regions accessed outside the scope of the code being optimized is needed for live-out properties. This is an orthogonal problem, and for

the moment we conservatively assume that the program is manually annotated with live-out arrays (the live-in and live-out scalars are explicit from the SSA form of the loop).

2.4.3 Live range non-interference

Any loop transformation is correct if it respects dataflow dependences and does not lead to live range interference (no two live ranges overlap) [95, 96, 99]. If we want to guarantee the non-interference of two live ranges, we have to make sure that the first live range is scheduled either before or after the second live range. Figure 2.8 shows two examples where pairs of live ranges do not interfere. Figure 2.9 shows two examples where these pairs of live ranges do interfere. We will now take a closer look at the conditions for preserving non-interference across loop tiling.

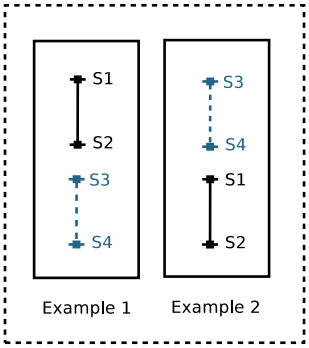

Figure 2.8 Examples where $\{S_1 \rightarrow S_2\}$ and $\{S_3 \rightarrow S_4\}$ do not interfere

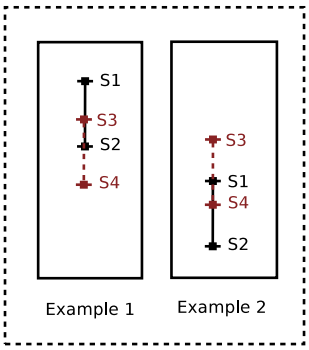

Figure 2.9 Examples where $\{S_1 \rightarrow S_2\}$ and $\{S_3 \rightarrow S_4\}$ interfere

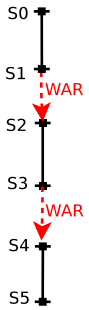

Figure 2.10 An example illustrating the adjacency concept

2.4.4 Live range non-interference and tiling: a relaxed permutability criterion

Tiling is possible when a *band* of consecutively nested loops is tilable. Using state-of-the-art tiling algorithms, the compiler looks for a band with *forward* dependences only for every level within the band [18, 107]. This can be done by calculating dependence directions for all dependences and for each loop level. A tilable loop band is composed of consecutive loop levels with no negative loop dependence. When a loop band is identified, dependences that are strongly satisfied by this band are dropped before starting the construction of a new inner loop band (a dependence is strongly satisfied if the sink of the dependence is scheduled strictly after the source of the dependence in one of the loop levels that are a part of that band). Loop levels that belong to the identified loop bands are tilable.

The main contribution of our work is a relaxation of this tilability criterion, based on a careful examination of which false dependences can be ignored. In particular, our criterion ignores anti-dependences (write-after-read dependences) that are *adjacent* to *iteration-private* live ranges. We say that a live range is *iteration-private at level k* if it begins and ends in the same iteration of a loop at level k.

We say that an anti-dependence and a live range are *adjacent* if the source of one of them is equal to the sink of the other and if the anti-dependence and live range derive from the same memory location. For example, in Figure 2.10, the dependence $\delta_{S_1 \to S_2}$ is adjacent to the live ranges $\{S_0 \to S_1\}$ and $\{S_2 \to S_3\}$ but is not adjacent to $\{S_4 \to S_5\}$. The dependence $\delta_{S_3 \to S_4}$ is adjacent to $\{S_2 \to S_3\}$ and to $\{S_4 \to S_5\}$ but is not adjacent to $\{S_0 \to S_1\}$.

Definition of the relaxed permutability criterion A band of loops is fully permutable if the relaxed permutability criterion holds on that band. The relaxed permutability criterion holds on a band if and only if for every dependence in the band one (or several) of the following conditions hold(s):

1. the dependence is forward for each level within the band;

2. it is an output-dependence between statement instances that only define values that are local to the region of code being analyzed, in the sense that the values defined *are* used *inside* the region and are *not* used *after* the region;

3. it is an anti-dependence, and it is adjacent only to live ranges that are iteration private within the band.

The first condition is inherited from the classical tiling criterion. The writes involved in the output dependences eliminated by the second condition are also involved in live ranges and are therefore guaranteed not to interfere with each other because of the third condition, as explained below. Moreover, the output dependences that are needed to ensure that values assigned by the last write to a given location (live-out values) will not be overwritten are *not* eliminated by the second condition. The third condition stems from the following observations:

1. Any affine transformation—including tiling—is valid if it preserves data flow dependences and if it does not introduce live range interferences.

2. Tiling is composed of two basic transformations: loop strip-mining and loop interchange.

   - Strip-mining is always valid because it does not change the execution order of statement instances and thus it can be applied unconditionally.
   - Loop interchange changes the order of iterations, hence may introduce live range interferences. But a closer look shows that it never changes the order of statement instances within one iteration of the loop body.

3. If live ranges are private to one iteration of a given loop (i.e., live ranges begin and terminate in the same iteration of that loop), changing the order of iterations of the present loop or any outer loop preserves the non-interference of live ranges.

4. We can ignore an anti-dependence only when it is adjacent to iteration-private live ranges (i.e., not adjacent to any non-iteration-private live range). This is correct for the following reason: an anti-dependence δ between two live ranges (S_1, S_2) and (S_3, S_4) is used to guarantee the non-interference of the two live ranges during loop transformations. If the two live ranges (S_1, S_2) and (S_3, S_4) are iteration-private, then they will not interfere when tiling is applied, regardless of the presence of the anti-dependence δ. In other words, δ is useless in this case. And thus ignoring this anti-dependence during

tiling is safe as long as all statements in this live range are subject to the same affine transformation. If one of the live ranges is non-iteration-private, then there is no guarantee that they will not interfere during tiling, and thus, conservatively, we do not ignore the anti-dependence between the two live ranges in this case.

Comparison with the criterion for privatization Reformulated in our terminology, a variable can be privatized in a loop if all the live ranges of this variable are iteration private within that loop[68]. This condition is the same as the third condition of our relaxed permutability criterion, except that we only impose this condition on live ranges that are adjacent to backward anti-dependences. Our relaxed permutability criterion can therefore be considered as a hybrid of the classical permutability criterion and the privatization criterion. In particular, our criterion will allow tiling if either of these two criteria would allow tiling or privatization, but also if some dependences are covered by one criterion and the other dependences are covered by the other criterion.

In particular, the privatization criterion cannot privatize the scalar `t` in the following example and thus some false dependences cannot be ignored and thus tiling cannot be applied.

```
for (i = 0; i < ni; i++)
  for (j = 0; j < nj; j++)
  {
   if (!(j == 0 && i == 0))
      t = alpha * A[i][j];
    B[i][j] = t;
  }
```

With our criterion, the false dependences can be ignored and thus tiling can be applied.

The importance of adjacency Note that it is not safe to ignore an anti-dependence that is adjacent to a tile-private live range but that is also adjacent to a non-tile-private live range. Tiling is not valid in this case.

2.4.5 Illustrative examples

i. The *mvt* and *gemm* kernels

We consider again the *mvt* kernel of Figure 2.7. The false dependences created by the `t` scalar inhibit the compiler from applying tiling, although tiling is valid for *mvt*.

Figure 2.11 represents the live ranges of `t` and `x[i]` for a subset of the iterations. The figure shows the statements that are executed in each iteration (S_1 and S_2) and the live ranges

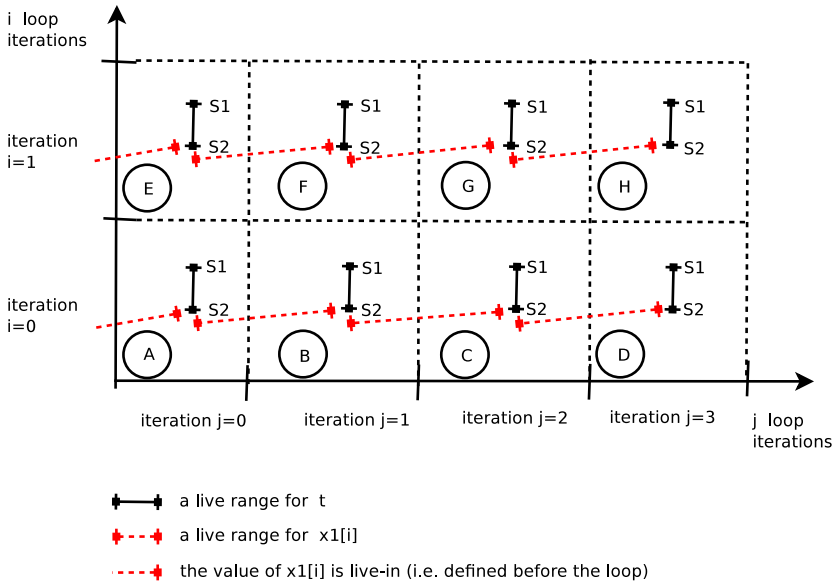

Figure 2.11 Live ranges for `t` and `x1[i]` in *mvt*

generated by these statements. The original execution order is as follows: live range Ⓐ of `t` is executed first (this is the live range between S_1 and S_2 in the iteration $i = 0$, $j = 0$), then live range Ⓑ is executed (this is the live range for iteration $i = 0$, $j = 1$), then Ⓒ, Ⓓ, Ⓔ, Ⓕ, Ⓖ, etc.

We notice that each live range of `t` is iteration-private. Each of these live ranges starts and terminates in the same iteration (no live range spans more than one iteration). The order of execution after applying a 2×2 tiling is as follows: (Ⓐ, Ⓑ, Ⓔ Ⓕ), (Ⓒ, Ⓓ, Ⓖ, Ⓗ). Tiling changes the order of execution of live ranges, but it does not break any live range of `t`.

The false dependences generated by the array access `x1[i]` do not inhibit loop tiling as all of these false dependences are forward on both levels. The live ranges of `x1[i]` satisfy the first condition of the relaxed tilability criterion (they are all forward) and therefore also do not inhibit tiling.

Although tiling is valid in the case of *mvt*, the classical tiling algorithm would fail to tile this code because of the false dependences induced by `t`. A similar reasoning applies to tiling of the *gemm* kernel shown in Figure 2.3: tiling is valid because the live ranges generated by the scalar `t` are iteration-private.

ii. An example where tiling is not valid

Consider the example in Figure 2.12. Similarly to the previous examples, the false dependences created by the scalar `t` on the j loop prevent the compiler from applying loop tiling. But is it correct to ignore the false dependences?

Figure 2.13 shows the live ranges of `t`. The original execution order is: Ⓐ, Ⓑ, Ⓒ, Ⓓ, Ⓔ, Ⓕ, Ⓖ, Ⓗ. After a 2×2 tiling, the new execution order is: Ⓐ, Ⓑ, Ⓔ, Ⓕ, etc. This means that the value written in Ⓑ on the scalar `t` will be overwritten by the write statement in Ⓔ. This tiling breaks the live range $\{S_2(\vec{I}) \rightarrow S_2(\vec{I'})\}$.

Tiling in this case is not valid because the live ranges are not iteration-private and thus we should not ignore the false dependences on `t`.

```
for (i = 0; i <= 1; i++)
  for (j = 0; j <= 3; j++) {
    if (j==0)
 S1:  t = 0;
    if (j>0 && j<=2)
 S2:  t = t + 1;
    if (j==3)
 S3:  A[i] = t;
  }
```

Figure 2.12 Example where tiling is not possible

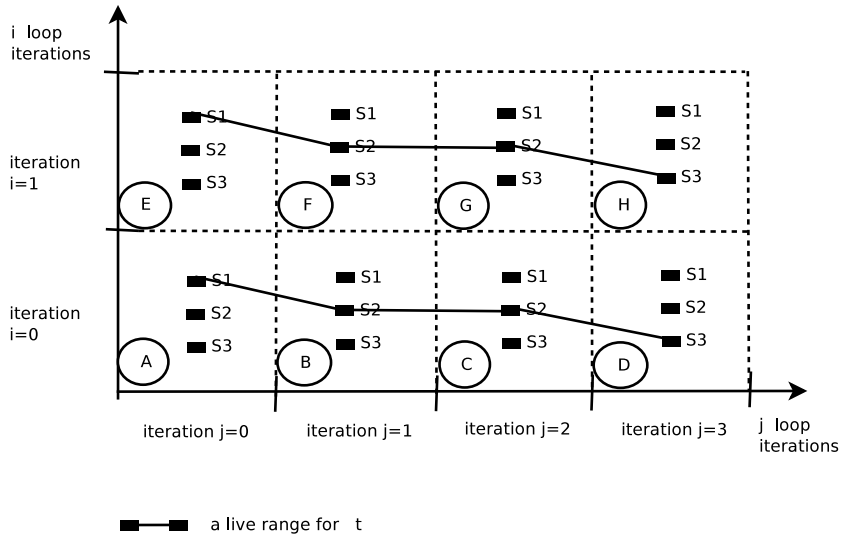

Figure 2.13 Live ranges for the example in Figure 2.12

2.4.6 Parallelism

Besides tiling, we are also interested in the extraction of data parallelism and in generating OpenMP directives for parallelism. In order to parallelize a loop, the variables inducing false dependences in this loop need to be privatized.

The privatization of variables for parallelism is much less intrusive than actual scalar or array expansion. The former only makes variables thread-private, not impacting address com-

putation and code generation inside the loop body, while the latter involves generating new array data declarations, rewriting array subscripts, and recovering the data flow across renamed privatized structures [36]. In addition, marking a scalar variable as thread-private does *not* result in the conversion of register operands into memory accesses, unlike the expansion of a scalar variable along an enclosing loop. In practice, support for privatization is implemented as a slight extension to the state-of-the-art violated dependence analysis [96, 100].

Figure 2.14 compares between the number of private copies that have to be created for a given scalar (or array) in order to enable parallelization, to enable tiling or to enable both. N is the number of loop iterations and T is the number of parallel threads that execute the loop. In general T is significantly smaller than N.

	number of private scalar and arrays	
	when expansion is applied	when false dependences are ignored
parallelization only	N	T
tiling only	N	0
tiling + parallelization	N	T

Figure 2.14 Comparing the number of private copies of scalars and arrays created to enable parallelization, tiling or both

Interestingly, the transformation of source code to three address code does not impact parallelism detection as it only induces intra-block live ranges. Section 2.5 proves a similar result about the impact of our technique on the applicability of loop tiling.

2.5 Effect of 3AC on the Tilability of a Loop

This section investigates the effect of 3AC transformation on the tilability of a loop (i.e., on whether we can tile it or not). Can the transformation of a code into 3AC from result in dependences that prevent tiling?

We use our relaxed permutability criterion to show that, in fact, the transformation of code into 3AC form does not have any effect on the tilability of a loop. In other words, whether tiling is applied before or after three-address lowering, we always get the same result. Consider the following example:

```
for (i1=...; i1<n1; i1++) {
  for (i2=...; i2<n2; i2++) {
    ...
    for (im=...; im<nm; im++) {
```

```
      ...
 S1:  x = expr_1 + expr_2 + expr_3;
      ...
    }
  }
}
```

Let Δ be the set of all dependences induced by S_1. After transforming this statement to 3AC, we get two new statements: S_2 and S_3.

```
for (i1=...; i1<n1; i1++) {
  for (i2=...; i2<n2; i2++) {
    ...
    for (im=...; im<nm; im++) {
      ...
 S2:  x1 = expr_2 + expr_3;
 S3:  x = expr_1 + x1;
      ...
    }
  }
}
```

Variable `x1` introduced by the three-address transformation is a new variable and thus it does not have any dependence with the other variables of the program (the variables that were already in the program before three-address transformation), but it induces three new dependences between the two newly introduced statements. These dependences are:

- A flow dependence: $\delta_{S_2 \to S_3}$;
- A write-after-write dependence: $\delta_{S_2 \to S_2}$;
- A write-after-read dependence: $\delta_{S_3 \to S_2}$;

Before three-address transformation the set of dependences was Δ. After three-address transformation the set of dependences is $\Delta \cup \delta_{S_2 \to S_3} \cup \delta_{S_2 \to S_2} \cup \delta_{S_3 \to S_2}$.

The flow dependence $\delta_{S_2 \to S_3}$ constitutes a set of live ranges that are iteration-private (they begin and terminate in the same iteration) and thus, by applying our relaxed permutability criterion, we can ignore the dependences $\delta_{S_2 \to S_2}$ and $\delta_{S_3 \to S_2}$ during tiling. The dependence that remains is $\Delta \cup \delta_{S_2 \to S_3}$. Since this dependence is iteration-private (because its dependence

distance is zero), it will have no effect on tiling[2], and thus applying tiling on the transformed code is exactly equivalent to applying tiling on the original source-level code (since in both cases the ability to apply tiling will depend on the set of dependences Δ).

This rationale can be generalized to the case where the right side of a statement has more than 3 expressions.

2.6 Experimental Evaluation

We implemented the relaxed permutability criterion on top of the Pluto polyhedral source to source compiler [18] (version 0.6) and evaluated the proposed criterion on a modified version of the Polybench 3.2 benchmark suite[3].

The original Polybench suite was not initially designed to expose the problem of false dependences. Among all of the benchmarks of the suite, scalar variables are used in only 5 benchmarks, the remaining benchmarks do not use scalars.

To stress the proposed technique on realistic false dependences, we chose to introduce scalar variables in each one of the kernels of Polybench instead of designing a new benchmark suite. These scalar variables were introduced either by transforming the source program into Three-Address Code (3AC) or by applying Partial Redundancy Elimination (PRE). Both transformations (3AC and PRE) have been manually applied to Polybench. Note that Pluto is a source-to-source compiler and that it does not convert the code to three address form internally, it does not apply PRE, nor does it privatize or rename variables to eliminate false dependences.

In order to transform the benchmarks into 3AC, each instruction that has more than three operands is transformed into a sequence of instructions, where each one of these new instructions has at most three operands. Temporary scalar variables are used to hold the values calculated by new instructions, each temporary scalar is assigned only once. Figures 2.15 and 2.16 show an example of transforming the *gesummv* kernel into 3AC.

The second transformation is PRE. We apply PRE to eliminate expressions that are redundant. We apply it whenever it is possible, ignoring any cost/benefit analysis to decide whether applying PRE is profitable for performance or not. Note that we do not apply 3AC when we apply PRE. Figures 2.17 and 2.18 show an example of applying PRE on the *gemm* kernel.

[2] Because tiling is possible in two cases: either if the dependence distance is zero (weakly satisfied) or if it is positive (strongly satisfied). The dependence distance in this case is zero so the tiling possibility does not depend on $\delta_{S_2 \to S_3}$ but on other dependences in the kernel

[3] http://www.cse.ohio-state.edu/~pouchet/software/polybench

Throughout the following sections, Polybench-3AC is used to refer to Polybench in three-address form, Polybench-PRE to refer to Polybench after applying PRE and Polybench-original to refer to the original Polybench.

To compare the proposed technique with expansion, we apply expansion on Polybench-3AC manually. This manual expansion is performed as follows: each scalar in the benchmarks of Polybench-3AC is transformed into an array. The dimension of the resulting array is the minimal dimension that makes tiling possible. Figures 2.19 and 2.20 show an example of applying expansion on the three-address form of the *matmul* kernel. We call the resulting benchmark Polybench-expanded. Expansion is not applied on Polybench-PRE, because after expanding the Polybench-PRE benchmark we get Polybench-original.

The manually transformed benchmarks (Polybench-3AC, Polybench-PRE and Polybench-expanded) are publicly available for reference.[4]

```
for (i = 0; i < N; i++) {
  tmp[i] = 0;
  y[i] = 0;
  for (j = 0; j < N; j++) {
    tmp[i] = A[i][j] * x[j] + tmp[i]
    y[i] = B[i][j] * x[j] + y[i];
  }
  y[i] = alpha * tmp[i] + beta * y[i
}
```

Figure 2.15 Original *gesummv* kernel.

```
for (i = 0; i < N; i++) {
  tmp[i] = 0;
  y[i] = 0;
  for (j = 0; j < N; j++) {
    temp1 = A[i][j] * x[j];
    tmp[i] = temp1 + tmp[i];
    temp2 = B[i][j] * x[j];
    y[i] = temp2 + y[i];
  }
  temp3 = alpha * tmp[i];
  temp4 = beta * y[i];
  y[i] = temp3 + temp4;
}
```

Figure 2.16 *gesummv* in 3AC form.

```
/* C := alpha*A*B + beta*C */
for (i = 0; i < NI; i++)
  for (j = 0; j < NJ; j++) {
    C[i][j] = C[i][j] * beta;
    for (k = 0; k < NK; ++k)
      C[i][j] += alpha*A[i][k]*B[k][
  }
```

Figure 2.17 Original *gemm* kernel.

```
/* C := alpha*A*B + beta*C */
for (i = 0; i < NI; i++)
  for (j = 0; j < NJ; j++) {
    temp0 = C[i][j];
    temp0 = temp0 * beta;
    for (k = 0; k < NK; ++k)
      temp0 += alpha*A[i][k]*B[k][j];
    C[i][j] = temp0;
  }
```

Figure 2.18 *gemm* after applying PRE.

[4] https://github.com/rbaghdadi/polybench-3.2-3AC

```
/* Scalar version of 2mm */
/* D := alpha*A*B*C + beta*D */
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++) {
    tmp0 = 0;
    for (k = 0; k < N; k++) {
      tmp1 = alpha*A[i][k];
      tmp2 = tmp1*B[k][j];
      tmp0 += tmp2;
    }
    E[i][j]=tmp0;
  }
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++) {
    D[i][j] *= beta;
    for (k = 0; k < N; k++) {
      tmp3 = E[i][k]*C[k][j];
      D[i][j] += tmp3;
    }
  }
```

Figure 2.19 *2mm*-3AC (*2mm* in 3AC form).

```
/* Expanded version of 2mm */
/* D := alpha*A*B*C + beta*D */
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++) {
    tmp0[i][j] = 0;
    for (k = 0; k < N; k++) {
      tmp1[i][j] = alpha*A[i][k];
      tmp2[i][j] = tmp1[i][j]*B[k][j];
      tmp0[i][j] += tmp2[i][j];
    }
    E[i][j]=tmp0[i][j];
  }
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++) {
    D[i][j] *= beta;
    for (k = 0; k < N; k++) {
      tmp3[i][j] = E[i][k]*C[k][j];
      D[i][j] += tmp3[i][j];
    }
  }
```

Figure 2.20 Expanded version of *2mm*-3AC.

2.7 Experimental Results

```
for (i = 0; i < _PB_N; i++) {
  temp0 = 0;
  temp1 = 0;
  for (j = 0; j < _PB_N; j++) {
    temp0 = temp0 + A[i][j] * x[j];
    temp1 = temp1 + B[i][j] * x[j];
  }
  y[i] = alpha * temp0 + beta * temp1;
}
```

Figure 2.21 *gesummv* kernel after applying PRE.

The experiments were performed on a dual-socket AMD Opteron (Magny-Cours) blade with 2×12 cores at 1.7GHz, 2×12MB L3 cache and 16GB of RAM running with Linux kernel 2.6.32. The baseline is compiled with GCC 4.4, with optimization flags `-O3 -ffast-math`. This version of GCC performs no further loop optimization on these benchmarks, and does not perform any tiling, but it succeeds in automatically vectorizing the generated code in many cases. We use OpenMP as the target of automatic transformations. We compare the original

	pluto –tile –parallel			
	Polybench-original	Polybench-3AC	Polybench-3AC –ignore-false-deps	Polybench-expanded
2mm	yes	no	yes	yes
3mm	yes	no	yes	yes
gemm	yes	no	yes	yes
gemver	yes	no	yes	yes
gesummv	yes	no	yes	yes
mvt	yes	no	yes	yes
gramschmidt	yes	no	yes	yes
dynprog	yes	no	yes	yes
fdtd-2d	yes	no	yes	yes
trisolv	yes	no	yes	yes
trmm	yes	no	yes	yes
syr2k	yes	no	yes	yes
syrk	yes	no	yes	yes
covariance	yes	no	yes	yes
correlation	yes	no	yes	yes
atax	yes	no	yes	yes
fdtd-apml	yes	no	yes	yes
bicg	yes	no	yes	yes
lu	yes	no	no	yes
jacobi-2d-imper	yes	no	no	yes
jacobi-1d-imper	yes	no	no	yes
seidel	yes	no	no	yes
symm	no	no	yes	yes
cholesky	no	no	yes	yes
ludcmp	no	no	no	yes
durbin	no	no	no	no

Table 2.1 A table showing which 3AC kernels were tiled

	pluto –tile –parallel		
	Polybench-original	Polybench-PRE	Polybench-PRE
	-	-	–ignore-false-deps
2mm	yes	no	yes
3mm	yes	no	yes
gemm	yes	no	yes
gemver	yes	no	no
gesummv	yes	no	no
mvt	yes	no	no
syr2k	yes	no	yes
syrk	yes	no	yes
correlation	yes	no	no

Table 2.2 A table showing which PRE kernels were tiled.

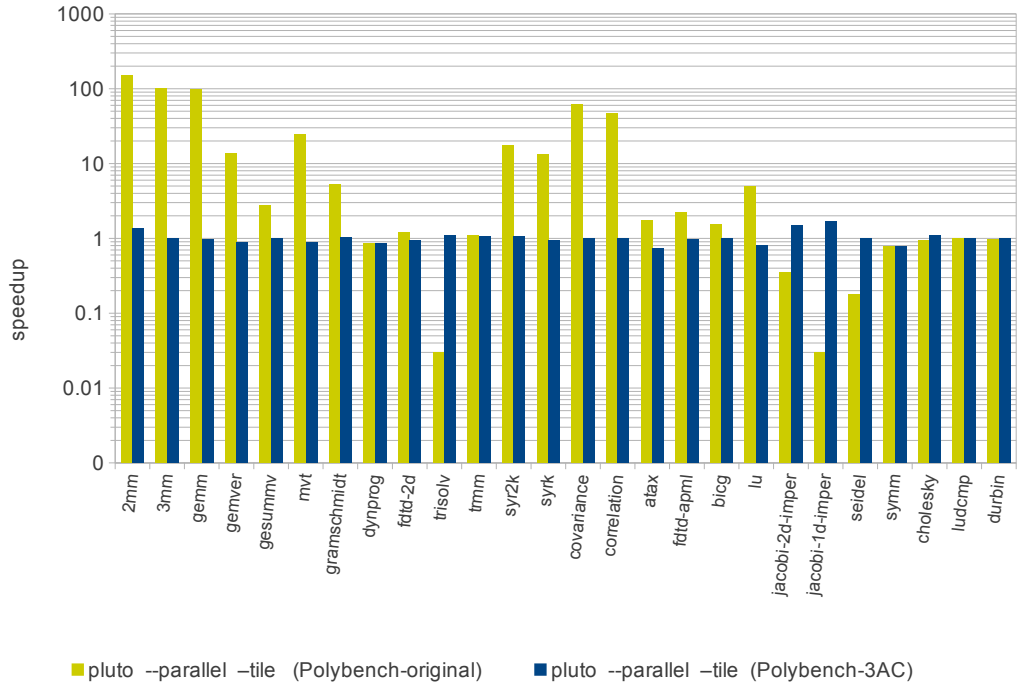

Figure 2.22 The effect of 3AC transformation on tiling

Pluto implementation with our modified version, reporting the median of the speedups obtained after 30 runs of each benchmark. The pass that implements the relaxed permutability criterion is activated in the modified Pluto through the option `--ignore-false-dependences`. The options `--tile` and `--parallel` are used to enable tiling and parallelization in Pluto (OpenMP code generation). All the tests are performed using the *big* datasets in Polybench.

2.7.1 Evaluating the Effect of Ignoring False Dependences on Tiling

To evaluate the proposed technique, we perform a set of experiments. The results of these experiments are summarized in Table 2.1. We report for each experiment whether Pluto succeeded to perform tiling or not. The first column lists the different benchmarks, classified into 4 classes. The second column shows the result of optimizing Polybench-original with Pluto (`pluto --tile --parallel`). It shows that all the kernels are tiled, except 4 kernels (*symm*, *cholesky*, *ludcmp* and *durbin*). *symm*, *cholesky* and *ludcmp* are not tiled because the original version of the benchmarks already contains scalar variables that induce false depen-

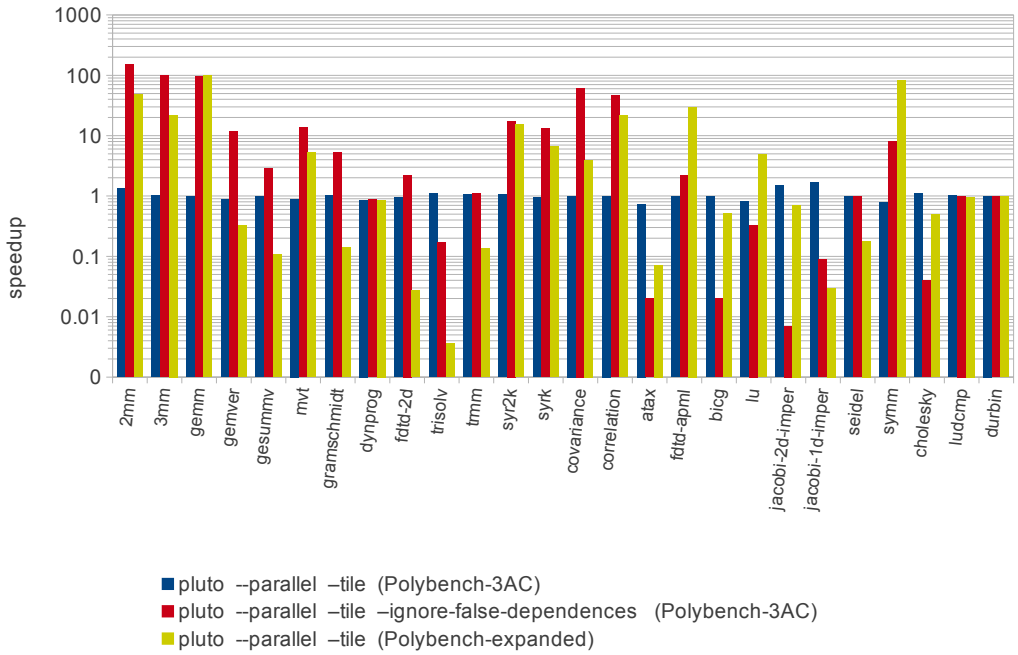

Figure 2.23 Speedup against non-tiled sequential code for Polybench—3AC

dences and prevent tiling. These benchmarks are examples where the original code written by the programmer contains scalar variables even before any 3AC or PRE code transformations. In the third column, we show the results of optimizing Polybench-3AC with Pluto (`pluto --tile --parallel`). In this experiment, Pluto systematically reports the presence of negative dependences, due to the extra scalar variables introduced by 3AC transformation, and fails to tile the Polybench-3AC benchmarks. None of these benchmarks is tiled.

When we repeat the previous experiment but add the option `--ignore-false-dependences` together with the options `--tile --parallel`, Pluto ignores the false dependences introduced by the transformation to 3AC, recovering the ability to tile Polybench-3AC (the results are presented in the fourth column). Most of the benchmarks are tiled except *lu*, *jacobi-2d-imper*, *jacobi-1d-imper*, *seidel*, *ludcmp* and *durbin* (the explanation comes later). In the last experiment (fifth column), Polybench-expanded is compiled with `pluto --tile --parallel`. The expansion succeeded in enabling the tiling of all the kernels, except the *durbin* kernel. Tiling for *durbin* is not possible, because it would lead to an incorrect schedule.

```
#define B 32
#pragma omp parallel for private(t3,t4,t5,t8)
for (t2=0; t2<=floor((N-1)/B);t2++)
  for (t3=0;t3<=floor((N-1)/B);t3++)
    for (t4=B*t2;t4<=min(N-1,B*t2+B-1);t4++)
      for (t5=B*t3;t5<=min(N-1,B*t3+B-1);t5++) {
        tmp0=0;
        for (t8=0;t8<=N-1;t8++) {
          tmp1=alpha*A[t4][t8];
          tmp2=tmp1*B[t8][t5];
          tmp0+=tmp2;
        }
      E[t4][t5]=tmp0;
    }

#pragma omp parallel for private(t3,t4,t5,t8)
for (t2=0; t2<=floor((N-1)/B);t2++)
  for (t3=0;t3<=floor((N-1)/B);t3++)
    for (t4=B*t2;t4<=min(N-1,B*t2+B-1);t4++)
      for (t5=B*t3;t5<=min(N-1,B*t3+B-1);t5++)
        for (t8=0;t8<=N-1;t8++) {
          tmp3=E[t4][t5]*C[t5][t8];
          D[t4][t8]+=tmp3;
        }
```

Figure 2.24 2mm-3AC optimized while false dependences are ignored.

```
#define B 32
#pragma omp parallel for private(t3,t4,t5)
for (t2=0; t2<=floor((N-1)/B);t2++)
  for (t3=0;t3<=floor((N-1)/B);t3++)
    for (t4=B*t2;t4<=min(N-1,B*t2+B-1);t4++)
      for (t5=B*t3;t5<=min(N-1,B*t3+B-1);t5++) {
        D[t4][t5]*=beta;
        tmp0[t4][t5]=0;
      }
#pragma omp parallel for private(t3,t4,t5,t8)
for (t2=0; t2<=floor((N-1)/B);t2++)
  for (t3=0;t3<=floor((N-1)/B);t3++)
    for (t4=B*t2;t4<=min(N-1,B*t2+B-1);t4++)
      for (t5=B*t3;t5<=min(N-1,B*t3+B-1);t5++) {
        for (t8=0;t8<=N-1;t8++) {
          tmp1[t4][t5]=alpha*A[t4][t8];
          tmp2[t4][t5]=tmp1[t4][t5]*B[t8][t5];
          tmp0[t4][t5]+=tmp2[t4][t5];
        }
        E[t4][t5]=tmp0[t4][t5];
        for (t8=0;t8<=N-1;t8++)
          tmp3[t4][t8]=E[t4][t5]*C[t5][t8];
        for (t8=0;t8<=N-1;t8++)
          D[t4][t8]+=tmp3[t4][t8];
      }
```

Figure 2.25 Optimizing the expanded version of 2mm-3AC.

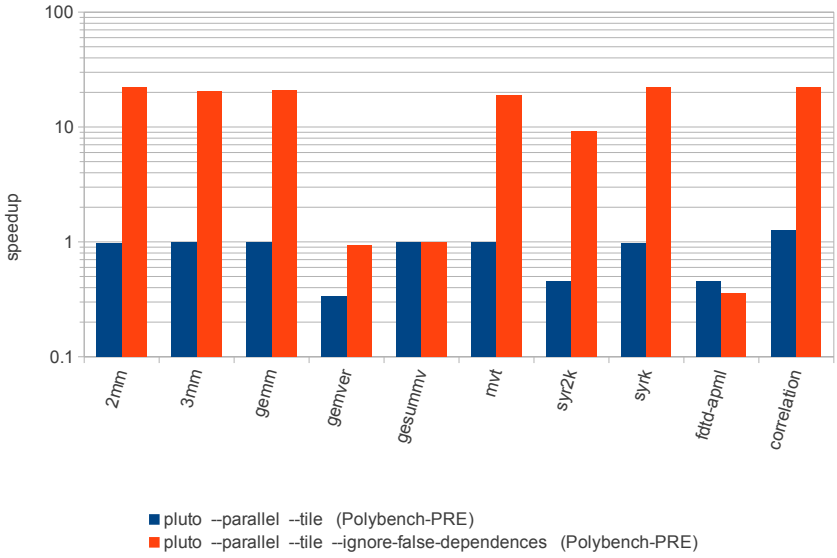

Figure 2.26 Speedup against non-tiled sequential code for Polybench—PRE

We repeat later another set of experiments where we focus on performance and show that although expansion enables tiling, it leads in many cases to worse performance due to its high memory footprint compared to the technique of ignoring false dependences.

Similar results are found when the same experiments are applied on Polybench-PRE (Table 2.2). Since PRE is only effective for some loop nests, this figure does not show all Polybench kernels. The original Pluto fails to tile the kernels. When the `--ignore-false-dependences` is used, Pluto succeeds in tiling 5 kernels out of 9. *Gemm* is a representative example of the kernels that were tiled and is presented in Figure 2.18.

In the other kernels that were not tiled (*gemver*, *gesummv*, *mvt* and *correlation*) the original loop depth is exactly 2 and applying the PRE optimization introduces a temporary scalar variable (to hold invariant data across the iterations of the inner loop); this variable induces false dependences that cannot be ignored by our criterion their adjacent live ranges are not iteration-private. This prevents Pluto from applying tiling. *Gesummv*, a representative example of kernels with a loop depth equal to 2, is presented in Figure 2.21.

2.7.2 Performance evaluation for Polybench-3AC

The following experiments focus on performance. In all of the experiments, the benchmark is first compiled with the Pluto source to source compiler, and then the resulting optimized code is compiled with GCC. The figures show different speedups compared to the baseline. Each speedup is the execution time of the kernel being tested normalized to the execution time of the baseline. The baseline is compiled with GCC 4.4, with optimization flags `-O3 -ffast-math`. Values greater than 1 indicate speedups, values smaller than 1 indicate slowdowns (the speedup of the baseline is 1).

Figure 2.22 shows the difference in performance between the optimization of Polybench-original with `pluto --parallel --tile` and the optimization of Polybench-3AC with the same options. As presented in Table 2.1, Pluto fails to tile Polybench after three-address code transformation.

Figure 2.23 shows the difference in performance between optimizing Polybench-3AC with and without the `--ignore-false-dependences` option. The figures compare also between ignoring false dependences and applying array and scalar expansion.

The results shown in Table 2.1 are important to understand these figures. Four classes of kernels can be identified by analyzing the figures. While some of the kernels show slowdowns, many kernels show a speedup. This speedup is obtained only when false dependences are ignored or when expansion is applied. The explanation for the slowdowns is presented in Section 2.7.3.

1. The first and largest class is represented by *2mm*, *3mm*, *gemm*, *gemver*, *gesummv*, *mvt*, *covariance*, *correlation*, *syrk* and *syr2k*. The `--ignore-false-dependences` option, applied to 3AC, restores the full tiling potential of Pluto. While expansion was effective in enabling tiling, the performance obtained with expansion is lower compared to the performance obtained when false dependences are ignored. The performance when expansion is applied is lower, because, with expansion, Pluto gets more freedom in applying loop transformations, and thus it applies transformations that on one hand enable better vectorization and parallelism but on the other hand prevent array contraction.[5] We explain this in the example presented in Figures 2.19, 2.20, 2.24 and 2.25. Figure 2.19 shows *2mm* in 3AC form, Figure 2.20 shows *2mm* after expansion (expansion is applied on the *2mm* 3AC code by transforming the scalars `tmp0, tmp1, tmp2` and `tmp3` into arrays). Figures 2.24 and 2.25 show *2mm* after applying loop transformations with Pluto. To optimize the expanded *2mm*, Pluto chose to apply loop distribution and to reschedule

[5] Contraction is used to transform the expanded arrays back into scalars. This transformation eliminates the memory overhead created by expansion but is not always possible.

statements moving the statement `tmp0` into a separate loop. Although loop distribution enables better vectorization and parallelism in general, in this example, it prevents the compiler to contract the `tmp0` array (i.e., transform the `tmp0` array into an array with fewer dimensions or into a scalar) and thus the memory overhead created by expansion is not eliminated. Loop distribution was also applied on the last loop of *2mm* leading to a scheduling of the statements manipulating `tmp3` into two separate loops which prevents the contraction of `tmp3`.

2. The second class contains 7 kernels (see Table 2.1): *atax*, *fdtd-apml*, *bicg*, *lu*, *jacobi-2d-imper*, *jacobi-1d-imper* and *seidel*. The kernels in this class are either kernels where we have better performance for expansion over ignoring false dependences (*atax*, *fdtd-apml* and *bicg*) or where ignoring false dependences does not enable tiling (*lu*, *jacobi-2d-imper*, *jacobi-1d-imper*, *seidel*). These kernels are a part of the same class because in all of these kernels, the Pluto algorithm which is supposed to perform a set of loop transformations that enable parallelism and enable tiling is not able to do so due to the false dependences in these kernels. Skewing is an example of these code transformations that need to be applied before the compiler can perform tiling. Since loop skewing is performed during the Pluto algorithm, and since we do not ignore false dependences during the Pluto algorithm, skewing is not performed and thus tiling is not possible. The expanded code has no false dependences which gives much freedom for the Pluto algorithm to perform the adequate affine transformations including skewing. The application of skewing in this case enables tiling.

   jacobi-2d-imper is a good representative of the kernels where false dependences limit the ability of the Pluto algorithm to perform loop transformations. To be able to tile the code and to extract outermost parallelism in *jacobi-2d-imper*, the Pluto algorithm has to perform loop skewing. Due to false dependences in three-address code, the Pluto algorithm could not apply skewing and thus tiling was not applied and only innermost loops could be parallelized which led to a loss in performance. This class of benchmarks motivates future work towards the integration of non-interference constraints into the Pluto algorithm itself.

3. In the third class, represented by *cholesky* and *symm*, the original code contains scalar variables, and thus Pluto fails to find tilable loop bands (Table 2.1). Ignoring false dependences enables Pluto to find tilable bands, resulting in performance improvements. Although the proposed technique enables loop tiling in *symm*, the performance obtained with expansion is better than the performance obtained with ignoring false dependences, because expansion enables the compiler to perform loop distribution which helps Pluto

to find more profitable, outermost parallelism on the newly created loops instead of a less profitable, innermost parallelism on the original loop. Finding outermost parallelism in the case of *symm* is not possible without expansion. This is an interesting example that shows that expansion can, in fact, enable aggressive loop transformations and that the gain from these transformations can bypass the loss of performance induced by memory footprint after expansion. This benchmark motivates future work towards studying the synergy between using expansion and ignoring false dependences to reach higher orders of performance gains.

4. In the fourth class, represented by *ludcmp*, and *durbin*, tiling is not possible as tiling is not a valid transformation for these kernels (tiling will break live ranges). The original code contains negative dependences that cannot be ignored even when our relaxed permutability criterion is used since these false dependences are adjacent to live ranges that are not iteration private (in both, Polybench-original and in Polybench-3AC). In *durbin*, even after expansion, the application of tiling remains impossible since the code contains false dependences that cannot be removed even by the application of expansion. Such a code cannot be tiled.

2.7.3 Reasons for slowdowns

In kernels such as *atax*, *fdtd-2d*, *trisolv*, *bicg*, *lu*, *jacobi-2d-imper*, *jacobi-1d-imper*, *seidel*, and *cholesky*, the overhead of parallelization is higher than the benefit of parallelization. In *trmm*, for example (presented in Figure 2.27), the overhead of synchronization at the end of each iteration of the loop `t2` (the loop with induction variable `t2`) is higher than the benefit of the parallel execution especially in the case of a small number of iterations in the inner loop. Although the parallelization is not profitable, Pluto parallelized the inner loop because it lacks an adequate profitability model. In some other kernels, the Pluto algorithm performs partial loop distribution creating many parallel inner loops nested in an outer sequential loop. Parallelizing many inner loops impacts performance negatively, and leads to more slowdowns. Having a model that helps Pluto to decide when the parallelization is profitable and when it is not would enable Pluto to avoid such slowdowns, but this is orthogonal to our contributions.

2.7.4 Performance evaluation for Polybench-PRE

Figure 2.26 shows the effect of optimizing the Polybench-PRE with and without the option `--ignore-false-dependences`. Tiling for the kernels *gemver*, *gesummv* and *fdtd-apml* is not valid, consequently these kernels do not show any speedup. For the other kernels, the tests

```
for (t1=1;t1<=NI-1;t1++)
  for (t2=ceil((t1-31)/32);t2<=floor((t1+NI-2)/16);t2++) {
    lb1=max(ceil(t2/2),ceil((t1-31)/32));
    ub1=min(floor((t1+NI-1)/32),floor((t1+32*t2+31)/64));
    #pragma omp parallel for shared(t1,t2,lb1,ub1) private(t3,t4,t5,t6)
    for (t3=lb1; t3<=ub1; t3++)
      for(t4=max(-t1+32*t3,32*t2-32*t3);
          t4<=min(min(32*t3+30,t1+NI-2),32*t2-32*t3+31);
          t4++)
        for(t5=max(max(t1,32*t3),t4+1);
            t5<=min(min(t1+t4,32*t3+31),t1+NI-1);
            t5++) {
          temp1[t1][-t1+t5]=A[t1][t1+t4-t5]*B[-t1+t5][t1+t4-t5];;
          B[t1][-t1+t5]+=alpha*temp1[t1][-t1+t5];;
        }
  }
```

Figure 2.27 Innermost parallelism in the expanded version of *trmm*.

show that the use of the option `--ignore-false-dependences` restores Pluto's ability to identify tilable bands despite the application of PRE.

2.8 Conclusion and Future Work

Loop tiling is one of the most profitable loop transformations. Due to its wide applicability, any relaxation on the safety criterion for tiling will impact a wide range of programs. The proposed technique allows the compiler to ignore false dependences between iteration-private live ranges, allowing the compiler to discover larger bands of tilable loops. Using our technique, loop tiling can be applied without any expansion or privatization, apart from the necessary marking of thread-private data for parallelization. The impact on the memory footprint is minimized, and the overheads of array expansion are avoided.

We have shown that ignoring false dependences for iteration-private live ranges is particularly effective to enable tiling on three-address code, or after applying scalar optimizations such as partial redundancy elimination and loop invariant code motion. Ignoring false dependences for tile-private live ranges is also an effective technique but such a criterion is beneficial only in a very small number of kernels.

Investigating how to integrate non-interference constraints into the Pluto algorithm itself is done in a separate work [102]. The goal here is to ignore false dependences on a more general class of affine transformations and avoid the performance degradation observed when converting a few benchmarks (such as *lu* and *jacobi-2d*) to three-address code. We are also

interested in combining this technique with on-demand array expansion, to enable the maximal extraction of tilable parallel loops with a minimal memory footprint.

Chapter 3

Scheduling Large Programs

3.1 Introduction

The polyhedral framework has shown an interesting success in modeling and applying complex loop transformations for complex architectures. As different polyhedral frameworks are becoming more and more mature, and as they are being used more and more to optimize large programs, there is a need to explore ways to reduce the compilation time of such large programs.

The Pluto algorithm [18] for affine scheduling is an example of a widely used scheduling algorithm. It is used in the `PPCG` [104] source to source polyhedral compiler, in the Pluto [18] source to source compiler and in Polly [42], a pass for polyhedral compilation in the LLVM compiler infrastructure [59]. This is an algorithm that modifies the schedules of statements (order of execution of statements) in order to apply loop transformations and enhance the performance of the generated code. Reducing the execution time of this algorithm is an important step towards reducing the overall compilation time in many polyhedral compilers.

In this work, we introduce a technique called *offline statement clustering*. This technique is motivated by a practical study of 27 numerical kernels from the *polybench*[1] benchmark suite: we noticed that statements that are a part of the same SCC (Strongly Connected Components) in the dependence graph in a given loop level are very likely to have almost the same schedule for that loop level (the only difference between the schedules is in the lexicographical order of these statements in the innermost loop level). A natural question to ask is whether we can take advantage of this remark in order to reduce the execution time of the affine scheduling step. If the schedules of some statements are usually almost equal, then there is no point in computing a separate schedule for each one of these statements (since all of them will have almost the

[1] http://web.cse.ohio-state.edu/~pouchet/software/polybench

same schedule anyway). It would be less expensive to represent all of these statements as one statement (we call this statement a *macro-statement*) and only schedule that macro-statement. This is what we call *statement clustering*.

In statement clustering, the original polyhedral representation of a program is transformed into a new representation where some statements are clustered together into macro-statements. Deciding which statements should be clustered together is done through a clustering heuristic. Then, instead of scheduling the original statements of the program, we schedule the macro-statements. Assuming that D is the set of the program statements, the Pluto affine scheduling algorithm is estimated to have an average complexity of $\mathcal{O}(|D|^5)$ [69], i.e., a quintic complexity in the number of statements on average. Since, in general, the number of macro-statements is less than the number of the original statements of the program, scheduling the macro-statements should take less time than scheduling the original statements of the program. This can be explained in more details as follows: the Pluto affine scheduling algorithm relies on creating an ILP (Integer Linear Programming) problem and on solving that ILP problem using an ILP solver in order to find new schedules for the statements. The number of variables in the ILP problem depends on the number of statements of the program being optimized and the number of constraints depends on the number of dependences of the program. Using statement clustering, we create a new representation of the program that has fewer statements and fewer dependences which reduces the total number of variables and the total number of constraints in the ILP problem. This simplification leads to a much smaller constraint system, that in general, is more likely to be solved more quickly by the ILP solver.

Note that in this work, we only focus on reducing the time for the step of affine scheduling. Reducing the time of the other steps is not in the scope of this work. Note also that we do not change the complexity of the Pluto affine scheduling algorithm, we rather suggest a practical technique to reduce its execution time. Although reducing the size of the input of the scheduling algorithm reduces the scheduling time in general (as we show experimentally), there is no theoretical guarantee about this. The reason is that the time needed by the ILP solver to find new schedules does not depend only on the number of variables and on the number of constraints in the constraint system but depends also on other factors.

The contributions of this chapter are the following:

- We present an algorithm for statement clustering. We make a clear distinction between the clustering algorithm and the clustering heuristics (used to decide which statements should be clustered together).
- We present two clustering heuristics: the basic-block heuristic and the SCC heuristic, and present a criterion to decide about the validity (correctness) of a given clustering decision (i.e., the decision of which statements should be clustered together).

- We show that the only optimization that cannot be done when SCC clustering is used is the ability to shift some statements in the SCC without shifting the other statements.
- We show experimentally that statement clustering helps in reducing the affine scheduling time by a factor of 6 (median) without a significant loss in optimization opportunities.

3.2 Definitions

Macro-statement A macro-statement is a virtual statement used to represent a cluster or a group of statements. A macro-statement is represented using an iteration domain. We say that a statement S corresponds to a macro-statement M if the macro-statement M was created by clustering S and probably some other statements in the same macro-statement. We need to keep track of which statements correspond to which macro-statements as this information is needed later.

Clustering Decision A clustering heuristic decides which statements should be clustered or grouped together. The output of the heuristic algorithm is a *clustering decision*. It is a set of clusters, where each cluster represents a macro-statement. Each cluster is a set of iteration domains (i.e., a set of statements).

3.3 Example of Statement Clustering

We illustrate the idea of statement clustering on the code in Figure 3.1.

```
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++) {
S1:   temp1 = A[i][j] * B[i][j];
S2:   C[i][j] = temp1;

S3:   temp2 = A[i][j] * C[i][j];
S4:   D[i][j] = temp2;
    }
```

Figure 3.1 Illustrative example for statement clustering

This program has 4 statements and 7 dependence relations. The dependence graph of the program is presented in Figure 3.2. To apply statement clustering on this program, first we need to use a clustering heuristic which produces a clustering decision. Let us suppose that

the heuristic suggests the following clustering decision:

$$\{\{D_{S1};D_{S2}\};\{D_{S3};D_{S4}\}\}$$

This clustering decision indicates that the two statement domains D_{S1} and D_{S2} should be clustered together and that the two statement domains D_{S3} and D_{S4} should be clustered together.

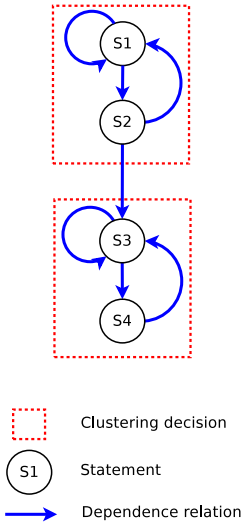

Figure 3.2 Original program dependence graph

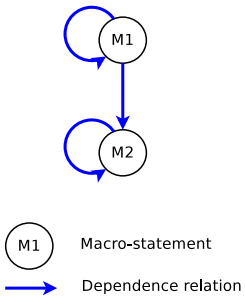

Figure 3.3 Dependence graph after clustering

If we apply this clustering decision, we get new iteration domains and a new dependence graph for the program that we present in Figure 3.3. The old dependence graph is transformed into a new dependence graph where the vertices are the macro-statements and the edges are the dependences between these macro-statements. In the new dependence graph, we have 3 dependence relations and 2 macro-statements: $M1$ which represents the clustering of D_{S1} and D_{S2} and $M2$ which represents the clustering of D_{S3} and D_{S4}.

3.4 Clustering Algorithm

The goal of the clustering algorithm is to construct a set of macro-statements and a dependence graph between these macro-statements. The clustering algorithm does not decide which statements should be clustered together, such a decision is taken by a separate heuristic that we describe in Section 3.6. Note that the clustering algorithm clusters only statements that are a part of the same basic-block and does not cluster statements that belong to different basic-blocks together.

The clustering algorithm takes as input:

- The set of statement iteration domains of the original program.
- The graph of dependences between these statements.
- A clustering decision: a set of clusters of statements.

The clustering algorithm returns:

- The set of macro-statement domains.
- The graph of dependences between the macro-statements.

In this work we only consider the clustering of statements that are a part of the same basic-block as the algorithms in this case are simpler. Thus we assume that all the statements that are clustered together have the same number of dimensions. A more generic clustering that handles statements that are not a part of the same basic-block is left for a future work.

The clustering algorithm is described in Algorithm 1. It has three main steps:

- First, we create a mapping, M, that maps the iteration domain of each statement in the program into the iteration domain of the corresponding macro-statement. This mapping will be used to transform the statements into macro-statements. It is created as follows: for each cluster in the clustering decision, we start by generating a new name for the macro-statement (let us assume that N is the generated name), for each statement in the cluster, we create a map from the iteration domain of that statement into a new iteration domain that has the same dimensions but has N as a name. This is the iteration domain of the corresponding macro-statement.
- In the second step, D', the set of iteration domains of the macro-statements is created. This is done by applying the mapping M to the iteration domains of the program statements. D' is the union of all the resulting dependences.

- In the third step, Δ', the set of dependences between the macro-statements is created. This is done by applying the mapping M to each one of the dependences of the program. Δ' is the union of all the resulting dependences. The number of dependences in Δ' is less than the number of dependences in the original dependence graph because after transforming the original dependences of the program, we may get some redundant dependences.

3.5 Using Offline Clustering with the Pluto Affine Scheduling Algorithm

No modification to the Pluto affine scheduling algorithm is needed to take advantage of offline clustering. We only need to apply offline clustering before calling the Pluto affine scheduling algorithm. The general work-flow is as follows:

1. Call the clustering heuristic. This heuristic creates a clustering decision. The output of this step is passed as an argument to the clustering algorithm.

2. Perform offline clustering to create a new dependence graph, and a set of macro-statements.

3. Call the Pluto affine scheduling algorithm with the new macro-statements and dependence graph as inputs. The Pluto affine scheduling algorithm returns a new schedule for the macro-statements.

4. We deduce the schedule of the original statements of the program as follows:

   - For each macro-statement `M`:
     - For each statement `S` that corresponds to `M`, the schedule of `S` is equal to the schedule of `M` except that a static dimension is added to the schedule of `S`. This static dimension is used to order the statements of the same macro-statement among themselves lexicographically in the same basic-block according to their original lexicographical order (or in any order that satisfies the dependences between these statements).

5. The original representation along with the newly computed schedule of the program are used in the rest of the polyhedral compilation flow.

Input:
D: a set of the iteration domains of the program statements.
Δ: a set of the dependences between the program statements.
ClusteringDecision: a set of sets of iteration domains.
Output:
D': a set of the macro-statement iteration domains.
Δ': a set of the dependences between the macro-statements.

```
// Step 1:  create a mapping
```
foreach *cluster* $\in$ *ClusteringDecision* **do**
```
    // Generate a name (identifier) for the new macro-statement
```
    N $\leftarrow$ GenerateName();
    $M \leftarrow \{\}$;
    foreach $D_S \in$ *cluster* **do**
```
        // We assume that S is the name of the statement represented
           by D_S, and n is the number of dimensions of D_S
        // Create a map M that maps the iteration domain of S into a
           new iteration domain (macro-statement's iteration domain)
```
        $m \leftarrow \{S[i_1,\ldots,i_n] \rightarrow N[i_1,\ldots,i_n]\}$;
        $M \leftarrow M \cup m$;
    end
end

```
// Step 2:  create macro-statements
```
foreach *cluster* $\in$ *ClusteringDecision* **do**
    $D' \leftarrow \{\}$;
    foreach $D_S \in$ *cluster* **do**
```
        // apply M to D_S
```
        $D'_S \leftarrow M(D_S)$;
        $D' \leftarrow D' \cup D'_S$;
    end
end

```
// Step 3:  transform Δ from a graph of dependences between
   statements into a graph of dependences between macro-statements
```
$\Delta' \leftarrow \{\}$;
foreach $\delta \in \Delta$ **do**
    $\delta' \leftarrow$ apply M to the source and sink of δ; // this returns a map from $M(Source(\delta))$ to $M(Sink(\delta))$
    $\Delta' \leftarrow \Delta' \cup \delta'$;
end

Algorithm 1: High Level Overview of the Clustering Algorithm

3.5.1 Correctness of Loop Transformations after Clustering

A loop transformation is correct if it does not violate the order of execution imposed by the dependences of the program. Let us assume that a statement S_1 is part of a macro-statement M_1 and that the statement S_1 is the source (or the sink) of a dependence δ in the original program dependence graph. If we apply clustering, the source of the dependence δ (or its sink) becomes M_1 and the schedule of M_1 will be constrained by the dependence δ which means that the schedules of all the statements that correspond to M_1 are constrained also by the dependence δ. This means that statement clustering, in fact, only adds additional constraints on the schedules but does not remove any existing constraint. In any case, a statement is never going to be less constrained than it should be and thus a correct transformation is still assured after statement clustering.

3.6 Clustering Heuristics

The goal of a clustering heuristic is to decide which statements should be clustered together in the same macro-statement. The clustering heuristic only clusters statements that belong to the same basic-block together and thus it needs to know which statement belongs to which basic-block. This information is not encoded in the iteration domains of the statements but is extracted from the original AST of the program.

3.6.1 Clustering Decision Validity

Clustering a given set of statements together is not always valid. In this section we show when is it valid to cluster a given set of statements in the same macro-statement. Although the clustering algorithm is defined only for the case where statements are a part of the same basic-block, the validity criterion is defined for the general case (i.e., when statements to be clustered together are not necessarily in the same basic-block).

To reason about the clustering validity criterion, let us consider the following example:

```
for (i = 0; i < N; i++)
S1:   A[i] = B[i] + 1;

for (i = 0; i < N; i++)
S2:   C[i] = A[i] + C[i - 1];

for (i = 0; i < N; i++)
S3:   D[i] = A[i] + C[i];
```

Figure 3.4 shows the flow dependences between the statements `S1`, `S2` and `S3`.

If the two statements `S1` and `S3` are clustered together, there are two possibilities regarding the order of execution of `S2` with regard to the two statements `S1` and `S3`: `S2` may be scheduled to be executed before the two statements or after them. Either ordering violates a dependence and thus clustering `S1` and `S3` is not valid in this case.

In general, given a dependence graph, it is not possible to cluster two statements if there is a path of dependences, in the dependence graph, between these two statements that passes by a third statement that is not being clustered with them.

This criterion needs always to be verified to check whether a given clustering decision is valid unless the clustering heuristic is guaranteed always to produce valid clustering decisions (such as the SCC clustering heuristic).

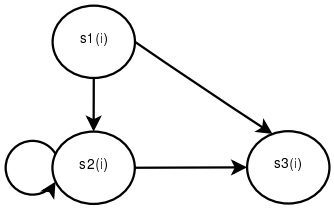

Figure 3.4 Flow dependence graph

In the following two sections we present examples of clustering heuristics.

3.6.2 SCC Clustering

Let $\Delta_k = (V, E)$ be the dependence graph of the program restricted to the statements that belong to *one basic-block* (let us call it BB1) and restricted to the loop depth k (k is between 1 and the maximal depth of the loop). The vertices V of Δ_k are the statements of BB1 and the edges E are the dependence relations between these statements projected on the k dimension.

Let $\Delta'_k = (V', E')$ be a dependence restricted also to the statements of BB1 and to the loop level k. This graph is the graph of dependences between the *execution instances* of the statements of BB1 (unlike Δ_k which is the graph of dependences between the *statements* of BB1). The vertices V' of Δ'_k are the execution instances of the statements of BB1 and the edges E' are the dependences between these execution instances.

Let us assume that Δ_k has an SCC that we call ω. Let V_ω be the set of vertices (statements) of this SCC. We say that ω is *dense* if for each two statements in V_ω, for each two instances of these two statements, there is a directed path in Δ'_k between these two instances.

To perform SCC based clustering, we compute Ω, the set of strongly connected components (SCCs) in the dependence graph Δ_k. For each $\omega \in \Omega$ (i.e., for each SCC in Ω), if ω is *dense*, we mark the statements of ω as candidates for clustering together at depth k. If a set

of statements are marked as candidates for clustering together at all the loop depths of a given loop, then those statements are marked by the heuristic to be actually clustered together. We do the previous procedure for all the basic-blocks of the program.

i. Validity of SCC Clustering

The SCC clustering heuristic always produces valid clustering decisions. To verify this, we need to verify that for any two statement instances $S(\vec{I})$ and $S'(\vec{I'})$ that are a part of an SCC, any third statement instance $S''(\vec{I''})$ that is a part of a path that goes from $S(\vec{I})$ to $S'(\vec{I'})$ must also be clustered with $S(\vec{I})$ and $S'(\vec{I'})$. Indeed, this is the case because if there is a path that goes from $S(\vec{I})$ to $S'(\vec{I'})$ that includes $S''(\vec{I''})$ then $S''(\vec{I''})$ is also a part of the SCC and thus $S''(\vec{I''})$ is also clustered with $S(\vec{I})$ and $S'(\vec{I'})$.

ii. Characterization of Possible Losses in Optimization Opportunities

Without statement clustering, the scheduling algorithm is free to assign any valid schedule to the statements of the program. But with statement clustering, this is not the case anymore. The use of statement clustering implies that all the statements that are a part of the same cluster will have the same schedule (more precisely, these statements will have the same schedule coefficients for the dynamic schedule dimensions). This means that we may lose some optimization opportunities. An important question, that arises here, is how much optimization opportunities we may lose.

Let us consider the following program:

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

The dependence graph of this program, restricted on the two statements $S1$ and $S2$ at the loop level 1, has a dense SCC. Since the two statements $S1$ and $S2$ are part of the SCC, and since the SCC is dense, then there exists $\delta_{S_1(\vec{I})\rightarrow S_2(\vec{I'})}$, a dependence (or a path of dependences) from $S1(\vec{I})$ to $S2(\vec{I'})$ and there exists $\delta_{S_2(\vec{I'})\rightarrow S_1(\vec{I''})}$, another dependence (or a path of dependences) from $S2(\vec{I'})$ to $S1(\vec{I''})$. This is true for any $S1(\vec{I}) \in D_{S1}$, $S2(\vec{I'}) \in D_{S2}$ and $S1(\vec{I''}) \in D_{S1}$ such that $S1(\vec{I}) \ll S2(\vec{I'}) \ll S1(\vec{I''})$. This means that $S2(\vec{I'})$ has to be executed between $S1(\vec{I})$ and $S1(\vec{I''})$. Any schedule for $S1$ and $S2$ must preserve this property.

Supposing that statement clustering is not used, the only possible transformation that the Pluto algorithm can do in the previous example, without violation the previous property, is to shift $S2$ one or more iterations as in the following example

```
for (i = 0; i < 2*N; i++) {
    if (i%2 == 0)
S1:    t = A[i];
    else
S2:    B[i] = t;
}
```

Such an optimization is not possible if statement clustering is applied.

In fact, the only optimization opportunity that we may lose if we apply SCC clustering is the ability to shift some of the statements that are a part of the SCC without shifting all the other statements. In practice, this is not a harmful restriction as we show in section 3.7.

Note that, when we apply SCC clustering, we do not lose the ability to distribute statements into separate loops, since it is not possible to distribute statements that are a part of the same SCC into two separate loops anyway [55].

3.6.3 Basic-block Clustering

In this clustering heuristic, all the statements that are a part of the same basic-block are clustered together in one macro-statement. We abbreviate the basic-block clustering with BB clustering.

Let us consider the example in Figure 3.5, applying the basic-block clustering heuristic on this code gives the following clustering: $\{\{D_{S0}, D_{S1}\}, \{D_{S2}, D_{S3}\}, \{D_{S4}\}\}$

```
for (i=0; i<N; i++) {
  S0;
  S1;
  for (j=0; j<N; j++) {
        S2;
        S3;
        if (l>0)
          S4;
  }
}
```

Figure 3.5 Illustrative example for basic-block clustering

i. Possible Losses in Optimization Opportunities

Basic-block clustering prevents the scheduler from assigning different schedule coefficients to the statements that are a part of the same basic-block. The same schedule coefficients (for

the dynamic schedule dimensions) must be assigned to all the statements of a basic-block. Loop transformations such as loop distribution, loop shifting of only some statements of the basic-block are no more possible when this clustering is applied. Although this may seem to be restrictive, the experimental section (Section 3.7) shows that there is no significant loss in performance due to these restrictions.

3.7 Experiments

We implemented the algorithm of statement clustering on top of the `PPCG` [104] source-to-source polyhedral compiler (version 0.02). `PPCG` takes C code as input and generates optimized C+OpenMP or CUDA [76] (Compute Unified Device Architecture) code or OpenCL [92] (Open Computing Language) code. It relies on the `pet` library [103] to extract the iteration domains and the access relations while the dependence analysis is performed by the `isl` library [101]. A new schedule is computed by `isl` using a variant of the Pluto affine scheduling algorithm [18]. Code is generated from the polyhedral representation using an algorithm derived from the ClooG algorithm [11]. The algorithm is described in detail in [44].

We use a command line argument to our tool to specify which statements should be clustered together. The tool reports the scheduling time, the original number of statements and dependences of the program and the number of macro-statements and dependences after the application of statement clustering. We evaluate statement clustering on a set of benchmarks that represent a variety of applications including linear algebra, image processing, stencils ...For each benchmark we measure the performance of the generated code when statement clustering is used and when statement clustering is not used in order to assess how much optimization opportunities may have been lost due to statement clustering. We use the `PPCG` options `--target=c --openmp` to generate C code annotated with OpenMP directives for parallelism.

We use the following benchmarks to evaluate statement clustering:

Image processing benchmark suite. This is a set of benchmarks that includes 7 image processing kernels (*color conversion*, *dilate*, *2D convolution*, *gaussian smoothing*, *basic histogram*, *resize* and *affine warping*). This suite covers the computationally intensive parts of a computer vision stack used by *Realeyes*, a leader in recognizing facial emotions (http://www.realeyesit.com).

Polybench-3AC benchmark suite. When a program is transformed into 3AC (Three Address Code), the number of statement increases considerably. It would be interesting to assess how much statement clustering would help in such a context. Since `PPCG` is a source-to-source compiler that does not convert its input code into three address form internally, and since no

automatic source-to-source tool, that converts C code into 3AC form, is available to us, we chose to convert the Polybench benchmark suite 3.2 into 3AC code manually. The goal is to assess how would statement clustering help in reducing the scheduling time when it is applied on a three-address-code.

This is the same benchmark used in Chapter 2. The process of converting Polybench into 3AC is described in Section 2.6. [2]

Swim benchmark. A benchmark extracted from the SPEC CPU2000 [47] benchmark rewritten in C (the original benchmark is written in Fortran).

Table 3.1 shows the amount of reduction of the number of statements of the program after statement clustering (SCC and BB clustering). A value such as 4 for the *trmm* kernel (SCC-clustering) indicates that the number of statements of the program is reduced by a factor of 4, i.e., for each 4 statements of the original program, one macro-statement is created. A high value in the table indicates an aggressive reduction in the number of statements of the original program after statement clustering.

In median, the number of macro-statements is 2.5× less than the original number of statements when SCC clustering is enabled and is 3× less than the original number of statements for BB clustering. The number of dependences is 3.67× less than the original number of dependences for SCC clustering and is 4× less than the original number of dependences for BB clustering. Statement clustering reduces the number of statements by a factor reaching 17× and 25× in *resize* and *affine warping*.

The numbers in Table 3.1 confirm the fact that BB clustering is more aggressive than SCC clustering since the SCC heuristic only considers clustering SCCs that are a part of the same basic-block and in general one basic-block may contain multiple SCCs, thus it is natural that the number of statements when BB clustering is applied is smaller than the number of statements when SCC clustering is applied (i.e., BB clustering is more aggressive than SCC clustering).

In Table 3.1, for the image processing benchmarks, the amount of reduction in the number of statements after SCC and BB statement clustering is identical. In fact both heuristics cluster statements exactly in the same way and yield exactly the same macro-statements in the image processing benchmarks. This is because in each one of these benchmarks, all the statements of a basic-block are a part of the same SCC for all the loop levels.

In *color conversion*, statement clustering does not reduce the original number of statements and dependences because this code is composed of only one statement. In *affine warping*, statement clustering reduces the number of statements from 25 statements into only 1 macro-statement and reduces the number of dependences from 72 into 1. In *swim*, which is a time-

[2] Polybench-3AC is publicly available on https://github.com/rbaghdadi/polybench-3.2-3AC

	SCC-clustering		BB-clustering	
Code	macro statements	dependences	macro statements	dependences
2mm	3.5	5.86	3.5	5.86
3mm	3.33	6.33	3.33	6.33
atax	3.5	8.2	3.5	8.2
bicg	3.5	10	4.67	13.33
cholesky	1.83	1.59	1.83	1.59
doitgen	1.67	3.67	1.67	3.67
gemm	2.5	6	2.5	6
gemver	3.75	5	3.75	5
gesummv	2.2	4.43	2.2	4.43
mvt	3	7	3	7
symm	3	3.18	4	3.89
syr2k	3.5	7.5	3.5	7.5
syrk	3	5	3	5
trisolv	2	2.83	2	2.83
trmm	4	12	4	12
durbin	2.14	2.6	2.5	2.6
dynprog	1.6	1.56	1.6	1.56
gramschmidt	1.86	2.06	1.86	2.06
lu	2.5	3.5	2.5	3.5
ludcmp	1.5	1.42	1.5	1.42
correlation	1.57	2.13	1.83	2.33
fwarshall	2	4	2	4
reg_detect	1.33	1.27	1.33	1.27
fdtd-2d	3.5	4.6	3.5	4.6
fdtd-apml	12.5	12.1	18.75	22
jacobi-1d-imper	2	3.5	2	3.5
jacobi-2d-imper	3	5.5	3	5.5
seidel-2d	9	35	9	35
color conversion	1	1	1	1
dilate	3.33	2	3.33	2
2D convolution	1.67	1.86	1.67	1.86
gaussian smoothing	1.33	1.33	1.33	1.33
basic histogram	1.5	2.5	1.5	2.5
resize	17	64	17	64
affine warping	25	72	25	72
Swim	1	1	17.33	84.44
Median	2.5	3.67	3	4
Average	3.83	8.5	4.6	11.22

Table 3.1 The factor of reduction in the number of statements and dependences after applying statement clustering

iterated stencil code, SCC clustering does not help in reducing the number of statements since one of the conditions of applying the SCC heuristic does not apply in this code: only the dependence graph that is restricted to the outermost loop level (time loop level) has SCCs, the other graphs restricted to the inner loop levels do not have any SCC (in the inner loop levels each SCC has only one statement).

In these experiments we use a 10 minutes time out for `PPCG`. If `PPCG` takes more than 10 minutes while scheduling a code, the `PPCG` process is killed. The default `PPCG` fails to compile

the *Swim* benchmark before the 10 minutes time-out. While enabling SCC statement clustering in PPCG does not solve the problem (since SCC clustering does not have any effect on the *Swim* benchmark), BB statement clustering succeeds in reducing the number of statements in the *Swim* benchmark by a factor of 17.33× which enables PPCG to compile the benchmark in 25 seconds only.

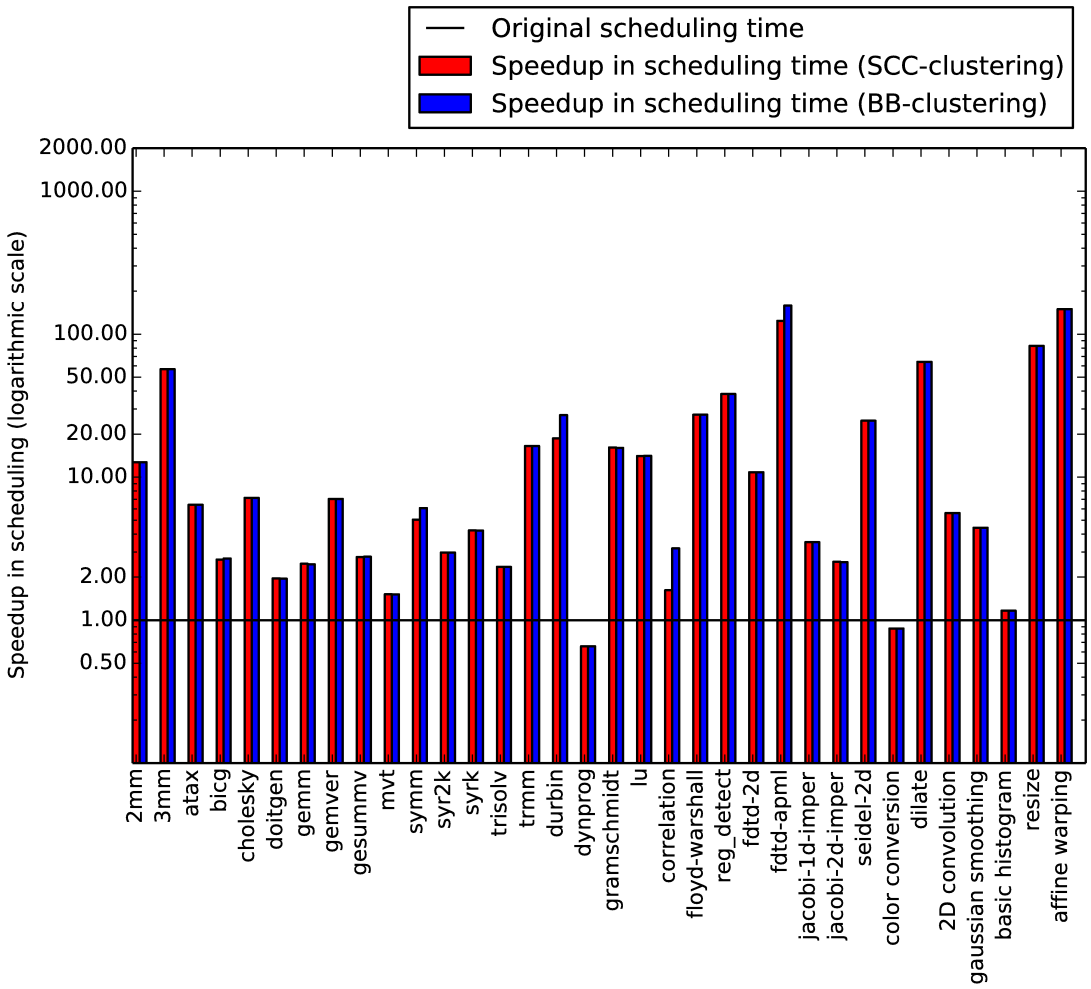

Figure 3.6 Speedup in scheduling time obtained after applying statement clustering

Figure 3.6 shows the speedups of the scheduling when statement clustering is applied over the case when statement clustering is not applied. We take in count the overhead of the clustering algorithm when we measure the scheduling time in the case of statement clustering (i.e., we sum the scheduling time and the clustering time when statement clustering is applied). The baseline is the scheduling time without clustering. Speedups higher than 1 indicate that statement clustering succeeded in reducing the scheduling time of the program. The experiments

were conducted on a system with a 64 bit Intel Core i5 CPU U470 at 1.33 GHz (4 cores) and 2 GB of RAM.

In the image processing benchmarks (Figure 3.6), the scheduling time for the BB heuristic is identical to the scheduling time for the SCC heuristic. This is in line with the previous results indicating that the two heuristics (SCC and BB) yield exactly the same clustering decisions in the image processing benchmarks. Statement clustering for *resize* reduces the scheduling time by a factor reaching 82×, and a factor reaching 149× for *affine warping* and by a median factor of 5.61× for the whole set of the image processing benchmarks. The speedups for *affine warping* and *resize* are due to the fact that statement clustering in these two cases reduces the number of statements by a factor of 25× and 16× respectively which is a quite high reduction in the number of statements compared to a factor of 1.16× for *basic histogram* for example.

In the Polybench-3AC benchmark suite (Figure 3.6), the speedups when the BB heuristic is applied are identical to the speedups when the SCC heuristic is applied. This is true for all the kernels except for 4 kernels: *symm*, *durbin*, *correlation* and *fdtd-apml* where the speedups for BB clustering are higher than the speedups for SCC clustering because BB clustering is more aggressive than SCC clustering in reducing the number of the statements of the program.

In *dynprog*, although statement clustering reduces the number of statements by a factor of 1.6×, the scheduling of the clustered program is slower than the scheduling of the original program. This is a case that confirms that the scheduling time does not depend only on the number of statements and on the number of constraints but also depends on other factors and thus reducing the number of statements does not guarantee a reduction in the scheduling time (although it is the case in general).

In all the kernels, the benefit from statement clustering is higher than the overhead of the clustering algorithm (clustering time). The only exception to this is *color conversion*, where statement clustering does not reduce the number of statements of the original program since the original program has only one statement. There is not benefit in applying statement clustering in this case while there is an overhead in applying statement clustering and this is why there is a slowdown in *color conversion*.

Figure 3.7 measures the effect of statement clustering on the quality of the `PPCG` generated code (we generate C + OpenMP using `PPCG`). The figure compares the execution time of the `PPCG` generated code when statement clustering is enabled with the execution time of the same code when statement clustering is not enabled. The baseline is the execution time of the `PPCG` generated code when statement clustering is not enabled. Values that are equal to 1 indicate that statement clustering does not deteriorate the quality of the generate code.

The figure indicates that the performance of the code generated when statement clustering is applied is very close (almost equal) to the performance of the code generated when state-

Figure 3.7 Execution time when clustering is enabled normalized to the execution time when clustering is not enabled (lower means faster)

ment clustering is not used. In a few cases, there is a small difference between the two, but this difference is not due to a loss in optimization opportunities. In fact, there is no difference in the schedule coefficients for the dynamic schedule dimensions for each one of these benchmarks. The difference is in the schedule coefficients for the static schedule dimensions. This difference is orthogonal to the application of statement clustering because the choice of the values of the static schedule dimensions by `PPCG` is arbitrary as long as the values chosen do not lead to a violation of the program dependences. In some kernels, the values of the static schedule dimensions, when clustering is enabled and when it is not, happen to be different and thus there is a small difference in the performance of the generated code. Our algorithm chooses values for the static schedule dimensions that preserve the original order of statements in the basic-block (since we only cluster statements that are part of the same basic-block). This order is valid because the original order is valid.

3.8 Related Work

Many alternative solutions can be applied to enhance the affine scheduling time in polyhedral compilation.

The first technique is to avoid the use of a general purpose affine scheduling algorithm that uses ILP (such as the Pluto [18] algorithm) and to apply instead a predefined set of loop transformations. An example of a compiler that used this technique is PolyMage [75]. A polyhedral compiler for an image processing DSL. In this compiler, two algorithms for loop transformations are used: the first algorithm applies overlapped tiling [48] (if possible) and the second algorithm groups different stages of a pipeline together to enhance data locality. No other loop transformation (loop interchange, loop shifting, ...) is applied. This approach relies on algorithms that were designed for image processing applications and in the context of DSL compilation where there is no need to discover parallelism since the information about parallelism comes from the DSL language itself. Although the algorithm used in the PolyMage compiler scales better than the Pluto algorithm, it suffers from being non generic, unlike the Pluto algorithm which can be applied in a variety of contexts.

The GRAPHITE polyhedral pass in GCC [95] and the Polly polyhedral pass in LLVM [42] both treat each basic-block in the program as a macro-statement and schedule the basic-blocks instead of scheduling the statements individually, but they do not use any clustering algorithm or clustering heuristic. The macro-statements are simply created during the creation of the polyhedral representation of the program. In a similar way, Mehta et al. [69] also represent basic-blocks as macro-statements (they call them super-statements), and propose an algorithm for reducing the number of statements and dependences but their algorithm only performs a clustering that is equivalent to our basic-block clustering. They do not make a distinction between the clustering algorithm and clustering heuristics. Our approach can be seen as a generalization of both approaches. Unlike [42, 95], we suggest an algorithm to perform clustering. Unlike [69] we make a clear distinction between the clustering algorithm and the heuristics that decide which statements should be clustered together, we suggest examples of heuristics and provide a criterion to decide whether a given clustering decision is correct or not (previous work did not need such a criterion since it focused only on basic-block clustering).

Feautrier proposes the use of Communicating Regular Processes (CRP) [39]. He uses a C-like specification language, in which smaller modules, comparable to functions in C, could be defined. The modules communicate with each other through communication channels, and communication channels between two modules are multi-dimensional arrays. Feautrier's technique sees processes as gray boxes, exposing constraints for coarse grain scheduling of the CRP. These constraints are much simpler than the original problem for the full program: coefficients and Farkas multipliers internal to the processes have been eliminated. But the constraints on the communication channels remain. The modules in CRP could be scheduled independently from each other, leading to smaller size linear programming problems and hence better compilation times.

3.9 Conclusion and Future Work

We presented the basic idea of statement clustering along with an algorithm to perform statement clustering and two clustering heuristics. We proposed a criterion that can be used to decide whether a given clustering decision is valid or not. The advantage of applying statement clustering is that it integrates well and transparently with the widely-used Pluto affine scheduling algorithm and that it can be combined with other techniques. We suggest this method as a practical, easy to implement solution that enables polyhedral tools to schedule programs more efficiently.

We provide an experimental evaluation on a variety of benchmarks and show experimentally that statement clustering succeeds in reducing the scheduling time by more than $10\times$ in many of the benchmarks and by a median factor of $6\times$ across all the benchmarks. In practice, the technique does not lead to any significant loss in optimization opportunities. We were able to match the performance of the code generated by PPCG for all the 34 benchmarks that we used to evaluate statement clustering.

A possible improvement for this work would be to extend the current clustering algorithm to enable the clustering of statements that are a part of different basic-blocks. Another improvement would be to explore new clustering heuristics, and to try to use the idea of statement clustering during the steps of code generation and dependence analysis in a polyhedral model (if statement clustering is to be used before the step of dependence analysis, the clustering heuristic should not rely on the use of dependences but on other means).

Chapter 4

The PENCIL Language

4.1 Introduction

Many systems — from supercomputer installations to embedded systems-on-chip — benefit from using special-purpose *accelerators* which can significantly outperform general-purpose processors in terms of energy efficiency as well as execution speed. Software for such systems, however, is currently written using low-level APIs such as OpenCL (Open Computing Language) [92] and CUDA (Compute Unified Device Architecture) [76], which increases the cost of its development and maintenance. A compelling alternative for developers is to work with higher-level programming languages, and to leverage compilation technology to automatically generate efficient low level code.

For general-purpose languages in the C family, this approach is hindered by the difficulty of static analysis in the presence of pointer aliasing, data-dependent array accesses and dynamic control. For example, the possibility of aliasing often forces a parallelizing compiler to assume that it is *not* safe to parallelize a region of source code, even though aliasing might not actually occur at runtime.

Domain-specific languages (DSLs) can circumvent this problem: it is often clear how parallelism can be exploited given high-level knowledge about standard operations in a given domain such as linear algebra [13], image processing [85] or partial differential equations [3]. The drawback of the DSL approach is the significant effort required to lower code all the way from the DSL level to highly optimized OpenCL or CUDA. The effort involved is even more significant if optimization is required for multiple platforms. Given typical budget constraints, the DSL implementers will likely limit their efforts to a set of techniques useful for a small number of target platforms, thus compromising on performance portability. Moreover, the implementers of different DSLs will likely spend their efforts on implementing an overlapping set of techniques. The existence of a common intermediate language serving as a target for

DSL compilers would reduce considerably the development costs. In addition, if this language is a high level language, it can also be used directly by domain experts to target accelerators and the advantage is doubled.

We present the design and implementation of PENCIL, a platform-neutral compute intermediate language. PENCIL aims to serve both as a portable implementation language to facilitate the acceleration of new and legacy applications on modern accelerators, and as an intermediate language for DSL compilers.

The contributions of this chapter are the following:

- the design of PENCIL, a platform-neutral compute intermediate language for direct accelerator programming and DSL compilation; thanks to the extensions of PENCIL, it is possible for a polyhedral compilation framework to generate more efficient code and to handle applications that do not fit in the classical restrictions of the polyhedral model;
- the evaluation of PENCIL on multiple GPUs and on several real-world, non-static-control applications that were previously out of scope for polyhedral compilation.

4.2 PENCIL Language

4.2.1 Overview of PENCIL

PENCIL is a rigorously-defined subset of C99 [50]. It enforces a set of coding rules principally related to restricting the manner in which pointers can be manipulated. These restrictions make PENCIL code "static analysis-friendly": the rules are designed to enable a compiler to perform better optimization and parallelization when translating PENCIL to a lower-level formalism such as OpenCL. PENCIL is also equipped with specific language constructs, including *assume predicates* and *side effect summaries* for functions, that enable communication of domain-specific information and static properties to the PENCIL compiler, to be used for parallelization and optimization. These specific constructs provide information that is difficult for a compiler to extract from arbitrary code but that can be easily captured from a DSL, or expressed by a programmer. Although the target platforms are highly parallel, PENCIL deliberately has sequential C99 semantics in order to simplify DSL compiler development and the work of a domain expert directly developing in PENCIL, and more importantly, to avoid committing to any particular pattern(s) of parallelism.

Where necessary, PENCIL exploits the flexibility of non-C99 extensions, and particularly GNU C extensions [105] such as type attributes. A design goal was to avoid pragma-based directives as directives are still not considered to be first class citizens by many compilers.

However, in very few cases, PENCIL relies on directives to attach properties to a control flow region of the code, no better C-compatible alternative being available. These directives have been inspired by standard directives used for vectorization and thread-level parallelism, but retain a strictly sequential semantics in PENCIL.

Because it is based on C, the learning curve for PENCIL is gentle. By design, PENCIL interfaces with C code, so that legacy C applications can be incrementally ported to PENCIL. From the point of view of DSL compilation, PENCIL offers an easy-to-target intermediate language because all a DSL-to-PENCIL compiler must do is faithfully encode the semantics of the input DSL program into PENCIL; auto-parallelization and optimization for multiple accelerator targets is then taken care of by the downstream PENCIL compiler. Because DSL-to-PENCIL compilers have tight control over the code they generate, such compilers can aid the effectiveness of the downstream PENCIL compiler by careful generation of code, and by communicating domain-specific information via the language constructs PENCIL provides for this purpose.

Design considerations The design of PENCIL is guided by the following considerations:

- PENCIL should have sequential semantics to facilitate the design and implementation of domain-specific compilers targeting PENCIL, to ease the work of PENCIL programmers, and to avoid committing early to target-specific patterns of parallelism.
- PENCIL should simplify static code analysis for the optimizing compiler. For example, the use of pointers is disallowed, except in specific cases, relieving the compiler from issues related to aliasing.
- PENCIL should provide facilities that allow a DSL-to-PENCIL compiler to convey, in the PENCIL code that it generates, domain-specific information that can be exploited by the compiler to perform better optimizations. For example, PENCIL should allow the user to indicate that the size of an array does not exceed a specific size to enable the compiler to place that array in the shared memory of a GPU (Graphics Processing Unit).
- For compatibility reasons, a standard C99 compiler that supports GNU C attributes [105] should be able to compile PENCIL, this allows for greater portability and makes the debugging of PENCIL code easier.
- The subset of C99 that constitutes PENCIL should be as large as the above design considerations permit. A very small and restrictive subset of C99 limits the reuse and modularity of PENCIL code, and makes PENCIL less attractive to programmers and DSL compiler-writers.

- Language extensions (compared to C99) should be minimized. Too many extensions make it harder for compilers to support PENCIL.
- PENCIL code should be able to interface with non-PENCIL code and external library functions.

Three possible scenarios of using PENCIL are possible:

1. PENCIL code is generated by a DSL compiler;
2. PENCIL code is written by a programmer;
3. a mixture of both scenarios.

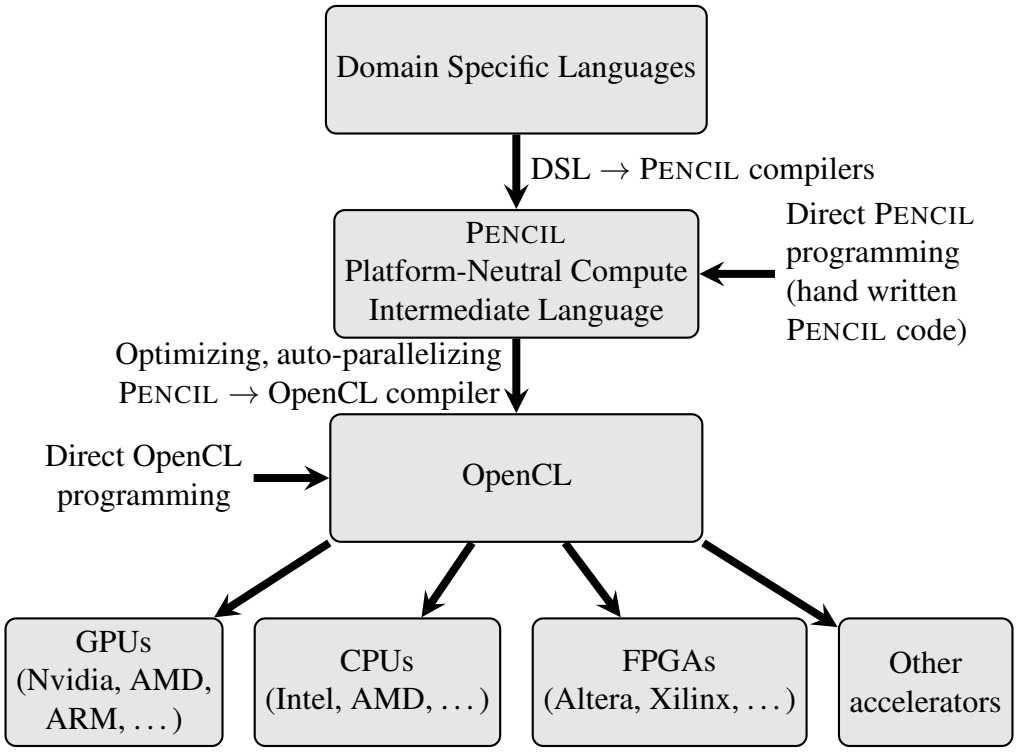

Figure 4.1 High level overview of our vision for the use of PENCIL

Figure 4.1 shows a high level overview of how we envision PENCIL to be used. First, a program written in a domain specific language is compiled into PENCIL. Domain specific optimizations are applied during this translation, and the DSL compiler may add domain specific information during this step through specific PENCIL language constructs. Second, the generated PENCIL code is combined with hand-written PENCIL code that implements specific library functions. PENCIL is used here as a standalone language. The combination of the two

pieces of code is then optimized and parallelized. Finally, highly specialized low-level code is generated. In Figure 4.1, we illustrate the case where the generated low-level code is OpenCL, allowing the compiled code to run across a range of OpenCL-compliant devices. We hereafter assume that OpenCL is the target language for PENCIL compilation, but in principle PENCIL can target any suitable low-level representation.

The design of the extensions to C99 that are a part of PENCIL went through two steps. First numerous DSLs (and benchmarks) were analyzed, and based on this analysis, a list of the properties that are expressed in these DSLs was created. This list was then filtered and only few properties were kept. Deciding which properties were supposed to be expressed in PENCIL was guided by the following design choice: all domain-specific optimizations should be performed at the DSL compiler level, while the PENCIL compiler should be responsible only for parallelization, data locality optimization, loop transformations, and mapping to OpenCL. This separation meant that, in PENCIL, only the properties that are necessary to enhance static-analysis and enable mapping to accelerator platforms are needed. This choice has the advantage of keeping PENCIL general-purpose, semantically sequential and lightweight.

This chapter is mainly based on a research report written in collaboration with other partners in the PENCIL project [7]. Section 4.2.2 presents the subset of C99 that defines the core of PENCIL, while Section 4.2.3 presents the extensions to C99 that are a part of PENCIL. The language syntax is defined in detail in Appendix A as an EBNF (Extended Backus-Naur Form) grammar [106].

4.2.2 PENCIL Definition as a Subset of C99

i. Program Scope

The following C99 [50] global definitions and declarations are allowed inside a PENCIL program: type definition, function declaration and definition, and constant declaration. PENCIL allows users to declare global constants as in C99, but it does not allow users to declare non-constant variables as global variables. This restriction enables PENCIL compilers to assume that PENCIL code does not have any side effect on global variables.

ii. Expressions

PENCIL supports a strict subset of C99 expressions: arithmetic, logical, bit and comparison operations, array and member operators (`[]` , `.`), ternary conditional (`cond?op1:op2`), `sizeof(arg)`, scalar conversion (`(type)arg`), PENCIL function calls, and C function calls (when a summary function is provided, details in Section 4.3.2).

iii. Types

Unlike C99, unions and bitfields are not supported in PENCIL. Only the following C99 types are allowed.

- Scalar types: scalar data types in PENCIL are the same as in C99, with the addition of the optional `half` float type (PENCIL compilers are not required to support the `half` float type).
- Structural types (same as in C99)
- Array types
  - Arrays (including array function arguments) must be declared using the C99 variable-length array syntax [50]. For example an array should be declared with `int A[N]` instead of being declared with `int *A`.
  - Array function arguments must be declared using the `static` keyword and the `const` and `restrict` type qualifiers described later (the macro `pencil_array` can be used to abbreviate `static const restrict`).
  - Array accesses should not be linearized, as this tends to obfuscate affine subscript expressions and may reduce the quality of the data dependence analysis. Multidimensional C99 arrays (C99 variable-length arrays syntax) should be used instead.
  - In order to have a precise dependence analysis, it is recommended to use quasi-affine array subscripts whenever this is possible.
- Pointer types
  - The declaration of pointers is allowed in PENCIL. The main motivation is to provide PENCIL users with the ability to use non-PENCIL libraries in a PENCIL code. Forbidding pointer declarations makes the use of non-PENCIL libraries difficult since the header files of such libraries may contain pointer declarations.
  - Pointer manipulation (including pointer arithmetic and reading or writing to a pointer) is not allowed. There is one exception to this restriction, reading an array reference is allowed when the reference is passed in a function call (e.g., `mat_add(C, A, B)`).

    The main motivation is to guarantee that the following property is preserved: throughout the life of a PENCIL program, separate array references never alias for write accesses and remain constant. Preserving this property is necessary to avoid the

need for an advanced pointer analysis in PENCIL compilers. Passing an array reference to a function is allowed in PENCIL as it does not violate the previous property. The property is not violated because function arguments in PENCIL are required to be qualified with `restrict` and `const`: if two separate arrays are passed to a function and if they are qualified with the `restrict` type qualifier then they are guaranteed not to alias for write accesses withing that function. Moreover, the `const` type qualifier guarantees that those array references remain constant within that function.

– Pointer dereferencing is not allowed. The only exception to this is accessing arrays using array subscripts (e.g., `A[i][j]`).

  Forbidding pointer arithmetic and forbidding pointer dereferencing (except dereferencing through array subscripts) makes the use of array subscripts to access an array element the unique way to do so which simplifies compiler analyses.

The restricted use of pointers is important for dependence analysis and for moving data between different address spaces of hardware accelerators, as it essentially eliminates aliasing problems.

Scalar type conversion and type definition (through `typedef`) in PENCIL both follow the same rules as in C99.

iv. Functions

Function definition and declaration in PENCIL follow the same rules as in C99. A function defined in PENCIL can be called from PENCIL or from C99 code. Function recursion is not allowed in PENCIL since OpenCL does not support recursion. Function overloading is also not allowed as the C99 standard does not allow it.

v. Statements

Unlike C99, `goto` and `switch` statements are not allowed in PENCIL as these two constructs make code analyses unnecessarily complicated without any benefit in expressiveness. Only the following C99 statements are allowed.

Assignment Statement Both, basic assignment (`=`) and compound assignment (`+=`, `-=`, `*=`, `/=`, `%=`, `|=`, `&=`, `^=`, `<<=`, `>>=`) are supported in PENCIL.

For Loop A PENCIL for loop must have a single iterator, an invariant start value, an invariant stop value and a literal constant increment (step). Invariant in this context means that the value does not change within the loop body. The iteration variable must not be visible (defined) outside the loop and cannot be modified inside the loop body.

```
for (type iter = init; iter [<|<=|>|>=] bound; iter[+|-]=step)
{
  //Body
}
```

By precisely specifying the loop format, we avoid the need for a sophisticated induction variable analysis. Such an analysis is not only complex to implement, but more importantly results in compiler analyses succeeding or failing under conditions unpredictable to the user.

While Loop The same as in C99.

If Statement The same as in C99.

Break and Continue Statements The same as in C99.

Return Statement The same as in C99 except that it can be used only at the end of a function (i.e., as the last statement in a function body). This allows PENCIL compilers to assume that they work on SESE (Single-Entry Single-Exit) regions of the control flow.

Call Statement The same as in C99 except that

- Calling a non-PENCIL function from PENCIL is allowed only if its summary function is provided (details about summary functions are presented in Section 4.3.2).
- Recursive function calls are not allowed.

vi. Identifiers, Declarations and Initializations

The naming of identifiers, the scope of identifiers, identifier declaration and initialization, in PENCIL, all follow the same rules as in C99.

vii. Preprocessor

PENCIL provides preprocessing directives equivalent to the C99 preprocessing directives. A normal C99 preprocessor can be used to preprocess PENCIL code.

4.2.3 PENCIL Extensions to C99

This section provides a list of PENCIL extensions. A detailed description of these extensions is provided in Section 4.3.

Directives

- `#pragma pencil independent [reduction(op: scal_1, ..., scal_n)]*`
- `#pragma pencil ivdep`
- `#pragma pencil region`

Function Attributes

`__attribute__((pencil_access()))`, `__attribute__((const))`, and `__attribute__((pencil))`

Builtin Functions

`__pencil_kill`, `__pencil_use`, `__pencil_def`, `__pencil_maybe`, `__pencil_assume`, `__pencil_assert`, and PENCIL math, common and integer builtin functions.

4.3 Detailed Description

4.3.1 `static`, `const` and `restrict`

i. `static` Keyword

The `static` keyword is used when declaring an array argument of a PENCIL function. It has the same semantics as the `static` keyword in C99. All array arguments of a PENCIL function must be declared using this keyword. The use of this keyword is important to implement array expansion or data transfers when subscripts are not affine.[1]

ii. `const` Type Qualifier

The `const` type qualifier is used when declaring an array argument of a PENCIL function. It has the same semantics as the `const` type qualifier in C99. All array arguments of a PENCIL function must be declared using this type qualifier. It is used to make sure that array arguments

[1] An array expansion maps an array to a larger array, typically by adding extra dimensions. The mapping may depend on the statement instance and can be used to remove some memory reuse.

behave as closely as possible to array variables, and to forbid that array arguments (the base pointer, not individual elements) occur at the left-hand side of an expression. This rule eliminates the risk of inducing array aliasing through the assignment of an arbitrary base address to an array argument.

iii. `restrict` Type Qualifier

The `restrict` type qualifier is used when declaring an array argument of a PENCIL function. It has the same syntax and semantics as in C99. All array arguments of a PENCIL function must be declared using this type qualifier. The use of `restrict` guarantees that the array arguments of the function may only alias if they are being used for read only. This allows PENCIL compilers to perform more precise dependence analysis.

Note that making the use of `restrict` mandatory in PENCIL requires upstream versioning of functions: if two arguments of a function are known to alias, then the two arguments must be transformed into one argument to avoid the aliasing problem.

Examples

Here is a correct PENCIL function declaration and call.

```
/* PENCIL code.
 * a and b are restricted arrays (analogous to restricted pointers in C99).
 */
void foo(int n, float a[static const restrict n]
                float b[static const restrict n]);


void bar()
{
  int n = 42;
  float pa[n], dc[n];

  foo(n, pa, dc);
}
```

`pencil_array` is a macro that abbreviates `static const restrict`.

4.3.2 Description of the Memory Accesses of Functions

The effect of a function call on its array arguments is derived from an analysis of the called function. In some cases, the results of this analysis may be too inaccurate. In the extreme case,

no code may be available for the function and the compiler can then only assume that every element of the passed arrays is accessed. In order to obtain more accurate information on memory accesses, the user may tell the compiler to derive the memory accesses not from the actual function body, but from some other function with the same signature. Such a function is called a *summary function*.

Summary functions are used to:

- Describe the memory accesses of library functions called from PENCIL code — as library functions cannot be analyzed at compile time.
- Describe the memory accesses of non-PENCIL functions called from PENCIL code — as they are difficult to analyze.

The use of summary functions in these cases enables more precise static analysis. To indicate the summary function of a function `foo()`, one uses the attribute `pencil_access(summary)`, where `summary` is the name of the summary function that describes the memory accesses in `foo()`.

A summary function is not meant to be executed, and is instead only used for the analysis of memory footprints. It has the same arguments as the qualified function. Each and every array element accessed in a function should be accessed in its summary. Yet a summary is generally simpler than the function it summarizes: it only captures sets of accesses, not their ordering and number of occurrences.

The polymorphic builtin functions `__pencil_use` and `__pencil_def` must be used in summary functions to mark memory access information (and to protect them from aggressive, PENCIL-agnostic upstream passes). The builtin function `__pencil_use`, abbreviated with `USE`, annotates read accesses, while `__pencil_def`, abbreviated with `DEF`, annotates (must-)write accesses. The polymorphic argument of these builtin functions may be a scalar, dereferenced pointer argument, or array element. It may also be a complete array when the dimension and size of the array are statically known: e.g., `__pencil_use(A)` marks the use of the complete array A, alleviating the need to list every element.

A summary function can contain calls to other functions, indicating that the corresponding calls are present on the original function. Example:

```
void foo(int N, int A[pencil_array N]);

void bar_summary(int N, int A[pencil_array N])
{
  foo(N, A);
```

```
  for (int i = 0; i < N; i ++)
  {
    USE(A[i]);
    DEF(A[i]);
  }
}

void bar(int N, int A[pencil_array N])
  __attribute__((pencil_access(bar_summary)))
  {
    foo(N, A);

    for (int i = 0; i < N; i ++)
    {
      int t = A[i];
      t = t + 1;
      A[i] = t;
    }
  }
```

To express may-write accesses, the boolean builtin `__pencil_maybe` should be used to guard these accesses in an `if (__pencil_maybe())` conditional. Example:

```
if (__pencil_maybe()) __pencil_def(v))
```

Summary functions were chosen over other formalisms (e.g., C++11 lambda functions [52], the use of directives, etc.) to express the memory accesses of a function because summary functions are based on the C syntax and thus can be parsed and analyzed easily using existing C compiler APIs. Moreover, summary functions are expressive and allow a fine-grained specification of memory accesses. In addition, summary functions are intuitive to C programmers, who do not have to learn a dedicated specification syntax or sub-language.

A summary function should be PENCIL compliant so that PENCIL tools can analyze it. Writing the summaries of library functions is the library developer's responsibility. Such summary functions should be provided in the library's header files and are used directly by the DSL compilers or PENCIL programmers. This is the most common case. In other less common cases, summary functions are either written by the PENCIL programmer himself or generated automatically by the DSL compiler (only the sets of read and written elements for each function argument need to be provided in this case).

The attribute `pencil_access` is abbreviated with the `ACCESS` macro.

Example 1

```
__attribute__((pencil_access(summary_fft32)))
void fft32(int i, int j, int n,
           float in[pencil_attributes n][n][n]);

int ABF(int n, float in[pencil_attributes n][n][n])
{
  // ...
  for (int i = 0; i < n; i++)
  {
    // ...
    for (int j = 0; j < n; j++)
      fft32(i, j, n, in);
  }
  // ...
}

void summary_fft32(int i, int j, int n,
           float in[pencil_attributes n][n][n]);
{
  for (int k = 0; k < 32; k++)
    __pencil_use(in[i][j][k]);
  for (int k = 0; k < 32; k++)
    __pencil_def(in[i][j][k]);
}
```

This example shows a loop extracted from *ABF* (Adaptive Beamformer), a signal processing kernel used in radar systems. The code calls the function `fft32` (Fast Fourier Transform) which only reads and modifies (in place) 32 elements of its input array `in`, rather than modifying the whole input array. Such a function is not analyzed by the PENCIL compiler as it is not a PENCIL function. Without a summary function, the compiler assumes conservatively that the whole array passed to `fft32` is accessed for read and for write. Such a conservative assumption prevents the parallelization of the code. The use of a summary function in this case indicates to the compiler that each iteration of the loop reads and writes 32 elements of the input array. This information allows the compiler to parallelize and optimize the loop.

Example 2

```
struct complex {
  int image;
  int real;
```

```
};

typedef struct complex Cplx;

void foo_summary(int n, int A[pencil_array n],
                     Cplx d[pencil_array n])
{
  for (int i = 0; i < n; i++)
    USE(A[i]);

  /* Note that i starts from 10. */
  for (int i = 10; i < n; i++)
    if (__pencil_maybe())
      DEF(d[i].real);

  DEF(A[15]);
}

void foo(int n, int A[pencil_array n],
           Cplx d[pencil_array n])
           ACCESS(foo_summary(n,A,Cplx))
{
  int t;

  for (int i = 0; i < n; i++)
    printf("Value=%d", A[i]);

  for (int i = 0; i < n; i++)
  {
    if (A[i])
      printf("%d", i);

    t += A[i];
  }

  for (int i = 0; i < n; i++)
    if (A[i] && i>10)
      d[i].real = t;
```

```
  A[15] = 0;
}
```

In the above example, the summary function `foo_summary` indicates that the function `foo` reads the values of `A[i]` for i from 0 to $n-1$ and writes to `A[15]`. `foo` may write to `d[i].real` for i from 10 to $n-1$.

4.3.3 `const` Function Attribute

The `const` attribute used in the GCC compiler is allowed in PENCIL for compatibility with existing `const`-annotated library functions (e.g., `cos`, `sin`, `min`, `max`, etc.). It indicates that the function does not read any value except its arguments, and does not have any effects except its return value [105]. The `CONST` macro can be used to abbreviate `__attribute__((const))`.

4.3.4 Pragma Directives

PENCIL defines several directives inspired by OpenMP [79] and OpenACC [22] and commonly found in advanced vectorizing compilers.

i. `independent` Directive

The `independent` directive is used to annotate loops. It has the following form:

```
#pragma pencil independent [reduction(op: scal_1, ..., scal_n)]*
```

The directive indicates that the desired result of the annotated loop does not depend upon the execution order of data accesses from different iterations. The definition of data accesses encompasses all memory accesses enclosed by the loop, either directly or indirectly through calls to PENCIL or non-PENCIL functions. In particular, data accesses from different iterations may be executed simultaneously. The definition of desired result is algorithm- and application-dependent.

The independent directive currently has informal semantics only. There are plans for more formal semantics in future revisions of PENCIL. External non-PENCIL functions called from the annotated loops may employ target-dependent constructs to protect the atomicity of their data access sequences, or to refine their parallel semantics regarding relaxed memory ordering, sound implementation of benign races, etc. Reductions implemented as atomic regions in the generated code are one typical example. Low-level atomics in C11 [51] or OpenCL 2.0 are another one (e.g., to give semantics to benign races). Such an approach is currently necessary to allow benign races (when the same value is written by multiple threads), to parallelize asso-

ciative and commutative operations, and more generally for any parallel algorithm tolerating the unordered execution of intermediate steps.

The `independent` directive has an effect only on the marked loop and does not have an effect on any other loop. It is typically used to indicate that the annotated loop does not carry any dependence. It allows the compiler to ignore all loop carried dependences of the annotated loop, including those that may be introduced by the compiler due to conservative assumptions (for example, if the code contains non affine write accesses). Note that no implicit privatization of scalars and arrays is assumed when the `independent` directive is used. Instead, scalars and arrays that are privatizable should be declared as local variables within the scope of the annotated loop. This may sound overly restrictive, but it ensures the portability of PENCIL when targeting languages and architectures where races have undefined semantics (C11, OpenCL 2.0).

Example 1 The following example shows a code fragment of a PENCIL implementation of the breadth-first search algorithm. This algorithm computes the minimal distance from a given source node to each node of the input graph. The algorithm maintains a frontier and computes the next frontier by examining all unvisited nodes adjacent to the nodes of the current frontier. All nodes in a frontier have the same distance from the source node.

The `for` loop shown in the example can be parallelized since each node of the current frontier can be processed independently. This creates a possible race condition on the `cost` and `next_frontier` arrays. The race condition can be ignored, however, because each conflicting thread will write the same values to the arrays. By specifying the `independent` pragma, the programmer guarantees that the race condition is benign, which enables a PENCIL compiler to parallelize the loop.

We plan also to add a built-in (`__pencil_store_atomic`) for concurrent writes. This would improve portability of concurrent writes, matching the design of standards like OpenCL 2.0 [60].

```
/* Examine nodes adjacent to current frontier */
#pragma pencil independent
for (int i = 0; i < n_nodes; i++) {
  if (frontier[i] == 1) {
    frontier[i] = 0;
    /* For each adjacent edge j */
    for (int j = edge_idx[i];
             j < edge_idx[i] + edge_cnt[i]; j++) {
      int dst_node = dst_node_index[j];
      if (visited[dst_node] == 0) {
```

```
        cost[dst_node] = cost[i] + 1;
        next_frontier[dst_node] = 1;
      }
    }
  }
}
```

Example 2 The following example illustrates the use of the directive in a typical context where the code executed in the loop body executes target-dependent, non-PENCIL code, e.g., code with atomic execution constraints that are currently not expressible in PENCIL.

```
void inc_summary(float A[256], int c)
{
  if (MAYBE)
  {
    USE(A);
    DEF(A);
  }
}

void inc(float A[pencil_array 256], int c) ACCESS(inc_summary);

void foo(int N, float A[pencil_array 256], int t[pencil_array N])
{
  #pragma pencil independent
  for (int i = 0; i < N; i++)
    inc(A, t[i]);
}
```

The following non-PENCIL C code implements `inc`, and is provided in a different file and compiled separately.

```
void inc(float A[256], int c)
{
  atomic_inc(&A[c]);
}
```

In this case, the user needs to provide an OpenCL implementation for `inc`.

Example 3 In the following example, the writes to `t` induce loop carried dependences and thus the use of `independent` is incorrect because there are at least two iterations that may write different values to `t`.

```
int t;
#pragma pencil independent
for (int i = 0; i < N; i++) {
  t = foo(i);
  A[B[i]] = t;
}
```

To be able to use `independent` in the previous example, the scalar `t` has to be declared in the loop body. This way, each iteration will have its own copy of `t` and thus the writes to `t` by different iterations will not induce any loop carried dependence. This is possible because `t` is privatizable (i.e., can be made private) [68]. The following example is correct (assuming that the elements of `B` are all different).

```
#pragma pencil independent
for (int i = 0; i < N; i++) {
  int t = foo(i);
  A[B[i]] = t;
}
```

`reduction` ***Clause*** Adding the `reduction` clause to the `independent` directive restricts the execution order of data accesses with respect to `independent` alone: considering the execution order of data accesses to the reduction variables (`scal_1, ..., scal_n`), the compiler must preserve the atomicity of side effects on these variables within a given loop iteration. This in turn widens the applicability of the directive, compared to `independent` alone. The reduction operator (`op`) itself is only useful to indicate how partial results on a reduction variable resulting from any interleaving should be combined.

Aside from extending the applicability of the independent directive, one motivation for introducing the `reduction` clause in PENCIL is to eliminate the need for having a sophisticated analysis to detect reductions in PENCIL compilers.

In order to simplify code generation for PENCIL compilers, only scalar variables can be used as reduction variables. Multiple reduction clauses can be used for different reduction operators. The syntax of this clause is equivalent to the syntax of the reduction clause defined in OpenMP. As in OpenMP, the reduction variables should not be used anywhere outside the reduction statement. Example:

```
#pragma pencil independent reduction(+: result)
for (int i = 0; i < n; i++) {
```

```
  B[T[i]] = foo(i);
  result += A[i];
}
```

In the above example, the compiler will ignore all the loop carried dependences of the loop except the loop carried dependences induced by the reduction variable `result` in the second statement.

The following reduction operators are supported: `+`, `-`, `*`, `min`, `max`.

ii. `ivdep` Directive

The `ivdep` directive is used to annotate innermost loops that are candidates for vectorization. If applied on a loop nest, it is effective only on the innermost loops of the nest.

It has the following form:

```
#pragma pencil ivdep
```

It allows the compiler to ignore all the loop-carried dependences in the loop marked with the directive from one statement to a textually earlier one (Cray semantics for ivdep [29]). This is generally sufficient to enable the vectorization of loops when the compiler is taking conservative assumptions.

The `independent` directive is stronger than the `ivdep` directive, as the latter only guarantees the correctness of a lock-step parallel execution (i.e., an implicit synchronization barrier in between every pair of statements in the loop body).

Example 1

```
#pragma pencil ivdep
for (int i = 0; i < m; i++)
{
  float t = a[i + k] * c;
  a[i] = t;
}
```

In this example, vectorization would be invalid if $k < 0$. The `ivdep` directive allows the compiler to ignore the assumed loop carried dependences that may exist if $k < 0$.[2]

[2] Ignore possible out-of-bound errors in this example.

iii. PENCIL Region Directive

The PENCIL region directive is used to delimit PENCIL code within C code. It is used for compatibility with legacy C code. The PENCIL compiler in this case is supposed to only optimize the region delimited with the `pencil region` directive.

Any array used within a PENCIL region should be declared using the same rules used to declare an array in PENCIL (i.e. the C99 VLA syntax and the macro `pencil_array` should be used).

Example

```
int foo(int m, int a[pencil_array m],
              int b[pencil_array m])
{
  /* Non PENCIL code. */
  int *c = malloc (100*sizeof(int));
  /* ... */

  /* PENCIL code. */
  #pragma pencil region
  {
    for (int i = 0; i < m; i++)
      a[i] = a[i] * c;

    for (int i = 0; i < m; i++)
      b[i] = b[i] + 1;
  }
}
```

4.3.5 Builtin Functions

In this section, `T` represents any valid PENCIL type.

i. `__pencil_use`, `__pencil_def` and `__pencil_maybe` Builtins

These builtins should be used in summary functions (a detailed description is provided in Section 4.3.2). They have the following prototypes:

```
void __pencil_use(T location);
void __pencil_def(T location);
int __pencil_maybe();
```

ii. `__pencil_kill` Builtin

The `__pencil_kill` builtin function has the following prototype:

```
void __pencil_kill(T location);
```

It allows the user to refine dataflow information within and across any control flow region in the program. It is a polymorphic function that signifies that its argument (a variable or an array element) is dead at the program point where `__pencil_kill` is inserted, meaning that no data flows from any statement instance executed before the kill to any statement instance executed after.

This information is used in two ways.

- In eliminating dataflow dependences within the control flow region.

- To determine which array elements may have their contents preserved by the region. In particular, when the region is mapped to a device kernel, then data that may be written inside the region and is possibly needed afterwards has to be copied back from the device to the host. The region of the array that is copied back to the host may however be larger than the set of elements that are known to be written by the region, either because some elements may only be written under certain circumstances or because the region that is copied back is an over-approximation. In such cases, the region first needs to be copied into the device to ensure that the elements within the array region that are not actually written by the code region preserve their original values. This latter step is not needed if these values were not preserved by the original code region. Such information can be passed to the compiler using `__pencil_kill`.

Example In the following code, the elements of `A` may be written inside the loop. This means that if the loop is mapped to a device kernel, then this array needs to be copied out from the device to the host. Since not all elements may be written by the loop, the array would in principle also need to be copied in first. The `__pencil_kill(A)` statement is used to indicate that the data is not expected to be preserved by the region and that therefore this copy-in can be omitted.

```
__pencil_kill(A);
for (int i = 0; i < n; i++) {
  if (B[i] > 0)
    A[i] = B[i];
}
```

iii. `__pencil_assume` Builtin

The `__pencil_assume` builtin function has the following prototype:

```
void __pencil_assume(int expression);
```

The intrinsic allows the optimizer to assume that its argument (which is a logical expression) is true whenever the control flow reaches the intrinsic call. No code is generated for this intrinsic. If the condition is violated during execution, the behavior is undefined. `__pencil_assume(exp)` does not instruct the compiler to check at run time whether `exp` is actually true or not. One may use `__pencil_assert(exp)` for that purpose instead. In the context of DSL compilation, an assume statement allows a DSL-to-PENCIL compiler to communicate high level facts in the generated code.

The argument of `__pencil_assume` does not need to be a quasi-affine expression. But as of today, our PENCIL compiler will not attempt to exploit `__pencil_assume unless` its argument is a quasi-affine expression.

Example 1 The code in Figure 4.2 (a general 2D convolution) is a good example where the assume builtin can be used. It is an image processing kernel that computes the weighted sum of the area around each pixel using a kernel matrix for weights. In this code, it is sufficient to consider that the size of the array `kern_mat` does not exceed 15×15 (such information is used implicitly in the OpenCV OpenCL image processing library for example). In fact, most convolutions do not exceed a kernel matrix size of 5×5

While this information is well known for an image processing expert, the compiler does not have this knowledge and must assume that the kernel matrix can be arbitrarily large. When compiling for a GPU target, the compiler must thus allocate the kernel matrix in GPU global memory, rather than fast shared memory, or must generate multiple variants of the kernel — one to handle large kernel matrix sizes and another optimized for smaller kernel matrix sizes — selecting between variants at runtime. The use of `__pencil_assume` in this case tells the compiler about the limits on the size of the array, allowing it to store the whole array in shared memory.

Example 2

```
void foo(int n, int m, int S, int D[pencil_array S])
{
        __pencil_assume(m > n);
        for (int i = 0; i < n; i++) {
                D[i] = D[i+m];
        }
}
```

```
#define clampi(val, min, max) \
  (val < min) ? (min) : (val > max ) ? (max):(val)

__pencil_assume(ker_mat_rows <= 15);
__pencil_assume(ker_mat_cols <= 15);

for (int i = 0; i < rows; i++)
  for (int j = 0; j < cols; j++) {
    float prod = 0.;
    for (int e = 0; e < ker_mat_rows; e++)
      for (int r = 0; r < ker_mat_cols; r++) {
        row = clampi(i+e-ker_mat_rows/2, 0, rows-1);
        col = clampi(j+r-ker_mat_cols/2, 0, cols-1);
        prod += src[row][col] * kern_mat[e][r];
      }
    conv[i][j] = prod;
  }
```

Figure 4.2 General 2D convolution

The loop above cannot be parallelized since it might have loop carried dependences (if the step `m` is less than the number of iterations `n`). The `__pencil_assume` builtin can be used to inform the compiler that `m` is greater than `n`, which allows the compiler to parallelize the loop. An alternative solution in this particular case is to explicitly mark the loop as independent.

```
void foo(int n, int m, int S, int D[pencil_array S])
{
       #pragma pencil independent
       for (int i = 0; i < n; i++) {
              D[i] = D[i+m];
       }
}
```

iv. `__pencil_assert` Builtin

The `__pencil_assert` builtin function has the following prototype:

```
void __pencil_assert(int expression);
```

It instructs the compiler to insert a run-time check of whether its argument (which is a boolean expression) is true at the program point where `__pencil_assert` is inserted. If the target architecture does not support the assert builtin (OpenCL for example), the PENCIL compiler should emit a warning message.

v. PENCIL Math, Common and Integer Builtin Functions

PENCIL supports a set of the OpenCL builtin functions:

- all OpenCL integer functions (`abs`, `clz`, `popcount`, ...);
- all OpenCL common functions (`min`, `max`, `clamp`, `sign`, ...);
- all OpenCL math functions (`sin`, `exp`, `cos`, `log`, ...).

The full list of OpenCL integer, common and math builtin functions is available in [56].

As PENCIL does not support function overloading (to be consistent with C99), every OpenCL builtin integer, common or math function has multiple equivalent functions in PENCIL with prefixes and suffixes for the different argument types. For example, the OpenCL *max* function has the following equivalent PENCIL functions

- *max*: used for int arguments;
- *smax*: used for short arguments;
- *bmax*: used for char arguments;
- *fmax*: used for double arguments;
- *lmax*: used for long arguments;
- *fmaxf*: used for float arguments;
- unsigned versions have a *u* prefix (*ubmax*, *usmax*, *umax*, *ulmax*).

PENCIL includes scalar builtin functions to help vectorize specific idioms. In particular, saturated and clamped arithmetic, absolute value, min, max, etc. Developers and DSL front-ends can use these functions and rely on the PENCIL compiler to generate a vectorized version of these functions.

Floating point operations are considered associative by default in PENCIL. PENCIL builtin functions do not have side-effects (on `errno`[3]) and floating point operations do not trigger exceptions (note that these assumptions match the effects of the `-fno-math-errno -fno-signaling-nans` GCC flags [105]).

[3] `errno` is an integer variable set by system calls and some library functions in case of an error to indicate what went wrong.

4.4 Examples of PENCIL and non-PENCIL Code

4.4.1 BFS Example

The following example is a parallel breadth-first search implementation in PENCIL. The program takes as input a graph and a start node and performs a breadth-first search. The order in which the nodes are visited (BFS order) is stored in the array `cost`. The algorithm has two steps which repeat one after the other until all nodes have been explored.

In the first step, each node n which is on the frontier (array `front` in the code) inspects its neighbors. The neighbor of n, if not visited previously, is put on `updating_front`, which is the set of nodes that will be on the next front. The cost of the neighbor is the cost of the node n plus one. Each node can perform this operation independently. It is possible that two nodes, both on the frontier, have an edge to the same neighbor m. In that case, both update the cost of this neighbor m (Line 41 in the example). This is a race (hazard) but it is a benign one since both will attempt to write the same value.

In the second step, the nodes which have been put on `updating_front` are moved to the frontier (array `front`). Each node can perform this step independently from the others.

One must note that the PENCIL code below is much easier to read and understand than the equivalent OpenCL code.

```
1
2  void parallel_version(int no_of_nodes,
3    int edge_start_no[pencil_array no_of_nodes],
4    int edge_count[pencil_array no_of_nodes],
5    int no_of_edges,
6    int dst_node_index[pencil_array no_of_edges],
7    int cost[pencil_array no_of_edges],
8    char front[pencil_array no_of_nodes],
9    char updating_front[pencil_array no_of_nodes],
10   char visited[pencil_array no_of_nodes],
11   int src_index, int quiet)
12 {
13   unsigned int carry_on = 1;
14
15   for (int i = 0; i < no_of_nodes; i++)
16   {
17     front[i]= 0;
18     updating_front[i] = 0;
19     visited[i] = 0;
```

```
20      cost[i] = -1;
21    }
22
23    front[src_index] = 1;
24    visited[src_index] = 1;
25    cost[src_index] = 0;
26
27    #pragma pencil region
28    {
29      while (carry_on == 1)
30      {
31        carry_on = 0;
32
33        #pragma pencil independent
34        for (int i = 0; i < no_of_nodes; i++) {
35          if(front[i] == 1) {
36            front[i] = 0;
37            for (int j = edge_start_no[i];
38                j < (edge_start_no[i] + edge_count[i]); j++) {
39              int dst_node = dst_node_index[j];
40              if (visited[dst_node] == 0) {
41                cost[dst_node] = cost[i] + 1;
42                updating_front[dst_node] = 1;
43              }
44            }
45          }
46        }
47
48        for (int i = 0; i < no_of_nodes; i++) {
49          if (updating_front[i] == 1) {
50            front[i] = 1;
51            visited[i] = 1;
52            carry_on = 1;
53            updating_front[i] = 0;
54          }
55        }
56      } //while.
57    } //#pragma pencil region
58  }
```

4.4.2 Image Resizing Example

The following example implements an image resizing kernel, a common image processing kernel.

```
1  #include <pencil.h>
2
3  #define bilinear(A00, A01, A11, A10, r, c) \
4    ((1-c) * ((1-r) * A00 + r*A10) + c*((1-r)*A01 + r*A11))
5
6  static void resize(const int rows,
7    const int cols,
8    const int step,
9    const unsigned char original[pencil_array rows][step],
10   const int r_rows,
11   const int r_cols,
12   const int r_step,
13   unsigned char resampled[pencil_array r_rows][r_step])
14 {
15     __pencil_assume(rows > 0);
16     __pencil_assume(cols > 0);
17     __pencil_assume(step >= cols);
18     __pencil_assume(r_rows > 0);
19     __pencil_assume(r_cols > 0);
20     __pencil_assume(r_step >= r_cols);
21
22     __pencil_kill(resampled);
23
24     int o_h = rows;
25     int o_w = cols;
26     int n_h = r_rows;
27     int n_w = r_cols;
28
29     for (int n_r = 0; n_r < r_rows; n_r++)
30     {
31       for (int n_c = 0; n_c < r_cols; n_c++)
32       {
33         float o_r = (n_r + 0.5) * (o_h) / (n_h) - 0.5;
```

```
34        float o_c = (n_c + 0.5) * (o_w) / (n_w) - 0.5;
35
36        float r = o_r - floor(o_r);
37        float c = o_c - floor(o_c);
38
39        int coord_00_r = clamp((int) floor(o_r), 0, o_h - 1);
40        int coord_00_c = clamp((int) floor(o_c), 0, o_w - 1);
41
42        int coord_01_r = coord_00_r;
43        int coord_01_c = clamp(coord_00_c + 1, 0, o_w - 1);
44
45        int coord_10_r = clamp(coord_00_r + 1, 0, o_h - 1);
46        int coord_10_c = coord_00_c;
47
48        int coord_11_r = clamp(coord_00_r + 1, 0, o_h - 1);
49        int coord_11_c = clamp(coord_00_c + 1, 0, o_w - 1);
50
51        unsigned char A00 = original[coord_00_r][coord_00_c];
52        unsigned char A10 = original[coord_10_r][coord_10_c];
53        unsigned char A01 = original[coord_01_r][coord_01_c];
54        unsigned char A11 = original[coord_11_r][coord_11_c];
55
56        resampled[n_r][n_c] = bilinear(A00, A01, A11, A10, r, c);
57      }
58    }
59    __pencil_kill(original);
60 }
```

4.4.3 Recursive Data Structures and Recursive Function Calls

The following code is not valid PENCIL code. The problems in this code are the following.

- PENCIL does not allow recursive function calls;
- pointer dereferencing is not allowed in PENCIL (except in a few cases cited in Section 4.2.2.iii.);
- in PENCIL, the return statement should be used only at the end of the function.

```
struct node
{
  int value;
  struct node* left;
  struct node* right;
};

struct node* find(struct node* node, int value)
{
  if (!node)
    return NULL;
  if (node->value == value)
    return node;
  if(value > node->value)
    return find(node->left);
  else
    return find(node->right);
}
```

4.5 Polyhedral Compilation of PENCIL code

The PENCIL language design was validated through the `PPCG` compiler (Polyhedral Parallel Code Generator) [104]. `PPCG` is a source-to-source polyhedral compiler that takes C code as input and generates OpenCL, CUDA and C code. `PPCG` transforms a program through the following steps.

1. **Model extraction**. This step takes C code as input and produces a polyhedral model consisting of an iteration domain, access relations and a schedule. The extraction of this model is performed using the `pet` library (Polyhedral Extraction Tool) [103].

2. **Dependence analysis**. This step takes an iteration domain, access relations and the schedule produced in Step 1 as input and produces the dependences between statement instances (i.e., which statement instances depend on which other statement instances). This step is performed using the `isl` library (Integer Set Library) [101].

3. **Scheduling**. In this step a new schedule that exposes parallelism and tiling opportunities is constructed by `isl` using a variant of the Pluto algorithm [18]. This is followed by the actual application of tiling and mapping to blocks and threads.

4. **Memory management**. It includes transfer to/from GPU and allocation to registers and shared memory.

5. **Code generation**. Code generation takes a schedule and generates code that visits each element in the iteration domain in the order specified by the schedule. The code generator used in `PPCG` is part of the `isl` library and is similar to CLooG [11].

The rest of this chapter shows how `PPCG` was modified to support PENCIL. Only a very short description of these modifications is provided here, since PENCIL support in `PPCG` was implemented by Sven Verdoolaege. For a detailed description of these changes we refer to [102]. The solutions proposed in this chapter for handling dynamic control (and especially `while` loops) are basic solutions. More sophisticated solutions can be developed and implemented but this is not the goal of this work.

Assume builtin. The `pet` library already keeps track of constraints on the symbolic constants of the program (variables that have an unknown but fixed value throughout the execution). These constraints are automatically derived from array declarations and index expressions. In particular, the values of the symbolic constants that necessarily result in negative array sizes or negative array indices are excluded (negative indices are not allowed because they could result in aliasing within an array).

`__pencil_assume` allows the user to provide additional constraints on the symbolic constants of the program that cannot be derived automatically from the code, For example, Line 4, and Line 5 in Figure 4.2 provide additional constraints on the symbolic constants `ker_mat_rows` and `ker_mat_cols` that are used throughout code generation whenever needed.

Kill builtin. A kill statement in `pet` represents the fact that no dataflow on the killed data elements can pass through any instance of the kill statement. This information can be used during dataflow analysis to stop the search for potential sources on data elements killed by the statement.

Whenever `pet` comes across a variable declaration, two kill statements that kill the entire array are introduced, one at the location of the variable declaration and one at the end of the block that contains the variable declaration. A user can introduce explicit kills by adding a `__pencil_kill`.

Non static-affine array accesses. In order to handle accesses that may not be static-affine, `pet` has been modified to make a distinction between *may*-writes and *must*-writes. Any index expression that cannot be statically analyzed or that is not affine, is treated as *possibly* accessing any index. This typically results in more dependences as more pairs of statement instances may possibly access the same array element.

Non static-affine conditionals and loop bounds, while loops, break and continue. A non static-affine conditional or a loop with non static-affine loop bounds is treated by PPCG as follows: the statement and its body are all treated as one macro-statement (i.e., as one statement that encapsulates the control statement and its body). Any write inside this macro-statement is treated as a may-write. For example, the statement in Line 2 of Figure 4.3 is governed by the condition in Line 1 which cannot be analyzed. The if-statement and its body therefore are considered as one macro-statement and the access to sup is treated as a may-write.

```
1  if (se[e][r] != 0)
2    sup = max(sup, img[cand_row][cand_col]);
```

Figure 4.3 Code extracted from *Dilate* (image processing)

While-loops and loops containing a break or a continue statement are treated similarly: the loop and its body are treated as one macro-statement (one single statement). For example, due to the break statement in Line 7 in Figure 4.4, the whole loop in Line 3 is treated by PPCG as a single statement. This means that PPCG can schedule (i.e., change the order of execution of) the loop in Line 3 and its body as a whole, but cannot schedule the statements in the body individually.

```
1  for (int i = 0; i < N; i++)
2    for (int j = 0; j < M; j++)
3      for (int k = 0; k < M; k++) {
4        B[i][j][k] = 0;
5
6        if (A[i][j][k] == 0)
7          break;
8      }
```

Figure 4.4 Example of code containing a break statement

Handling while loops is currently very basic in PPCG. The current work does not have any contribution in that direction.

Independent directive. When the independent directive is used to annotate a loop, the iterations of that loop may be freely reordered with respect to each other, including reorderings that result in (partial) overlaps of distinct iterations. In particular, the user asserts through this directive that no dependences need to be introduced to prevent such reorderings. The directive assumes that a variable that is declared inside the loop is considered private to any given iteration. pet currently handles the independent directive by building a relation between the statement instances that are excluded from depending on each other, as well as the set of variables that are local to the marked loop. This set of local variables is used by PPCG to

ensure that their live ranges do not overlap in any affine transformation, and to privatize them if needed when generating parallel code.

Summary functions. `pet` has been modified to extract access information from called functions. Whenever a summary function is provided, this information is extracted from the summary function instead of the actually called function.

4.6 Related Work

PENCIL language constructs such as the `independent` directive are inspired from directive-based languages such as OpenMP [79] and OpenACC [22], but unlike those, PENCIL has sequential semantics. In PENCIL, the `independent` directive describes the absence of loop carried dependences and such information can be used to enable a range of loop transformations rather than enabling loop parallelization alone. A semantically similar directive, also called `independent`, has been part of High Performance Fortran [66]. Despite also being defined with parallelism in mind, its definition allows compilers to derive information that enables additional sequential loop transformations.

The `__pencil_assume` construct, not defined in OpenMP or in OpenACC, allows the compiler to receive additional information from the DSL (or directly from a programmer), and to exploit this information to enable further optimizations. Microsoft Visual C supports a proprietary `__assume` statement and a `__builtin_assume` statement has been introduced in clang 3.6. Such builtins have semantics identical to `__pencil_assume` and could be used as substitutes if available. As a subset of C, PENCIL is designed to allow advanced compilers to perform better static analysis, enabling automatic parallelization which is not addressed by OpenMP and OpenACC. Certain C compilers (gcc, clang, icc) support builtins such as `__builtin_unreachable` which can be used to communicate assumptions that the compiler can exploit in a best-effort way. However, supporting these builtins in the PENCIL compiler is undesirable as it requires more pattern matching, which means it is more fragile and difficult to define formally.

DSL compilers in general map the DSL code directly to GPU relying on parallelism information provided by the domain specific language constructs that express parallelism. Using such an approach, DSL compilers like Halide [85] and Diderot [26], designed for image processing, and OoLaLa [67], designed for linear algebra, show promising results. Our goal is complementary, as we aim to build a more generic and reusable framework and intermediate language that can be used for different domain specific optimizers.

Delite [23] is a more generic DSL framework and run-time designed to simplify building DSL compilers. It uses Lightweight Modular Staging (LMS) [88], a compiler framework designed for embedding DSLs into Scala. Delite relies on information from the DSL to

decide whether a loop is parallel and does not use any framework for advanced loop transformations. We believe that the use of PENCIL and the polyhedral framework with generic DSL frameworks like Delite can leverage automatic parallelism detection and more complex loop transformations. One example of such complex loop transformations is the use of hybrid hexagonal/classical tiling [43] in stencils. A transformation that favors data reuse in local/shared-memory, avoidance of thread divergence, concurrency, and combines hexagonal tile shapes along the time and one spatial dimension with classical tiling along the other spatial dimensions.

In the next chapter, we evaluate PENCIL on code with irregular, data-dependent control including a set of standard benchmark suites (Rodinia and SHOC), image processing kernels, and a DSL compiler for linear algebra (BLAS). To assess performance portability, we perform the experiments on four GPU platforms: AMD Radeon HD 5670 and Radeon R9 285, Nvidia GTX470, and ARM Mali-T604 GPU.

Chapter 5

Evaluating PENCIL

5.1 Introduction

We evaluate the performance of OpenCL code generated from PENCIL using a development version of PPCG [104]. To show that PENCIL can be used as a standalone language as well as an intermediate language for DSL compilers, we present experimental results covering benchmark suites (written in PENCIL) and code generated automatically by a DSL compiler. The benchmark suites written in PENCIL are the RealEyes image processing benchmark suite (Section 5.2.1), and a selected set of benchmarks from Rodinia and SHOC (Section 5.2.2). The DSL compiler is the VOBLA DSL compiler (Section 5.2.3). This chapter is mainly based on a paper written in collaboration with other partners of the PENCIL project [6]. A part of the image processing benchmark was developed by Robert David from RealEyes, a company working in automatic emotion detection. Partners from ARM did the evaluation of PENCIL on the Rodinia/SHOC benchmarks and developed the VOBLA DSL compiler. The experiments on the VOBLA DSL were performed in collaboration with these partners. Michael Kruse did many experiments on the SpearDE DSL compiler (a DSL compiler for signal processing applications), the results of that evaluation are not presented in this manuscript but are available in [6].

The experiments evaluate whether PENCIL enables the parallelization (mapping to OpenCL) of kernels that cannot be parallelized with the current state-of-the-art polyhedral compilers such as the Pluto compiler [18]. They also evaluate whether PENCIL enables the generation of more efficient code and how close the performance of the automatically generated code is to hand-crafted code. The experiments evaluate the whole PENCIL framework on a relatively large set of real world applications and test platforms.

We developed an autotuning compiler framework to facilitate the retargeting of our framework to very different GPU architectures. We only apply autotuning to the PPCG-generated

code. For one, autotuning the reference code (which is mostly implemented as libraries) does not make sense because library code is not designed to be autotuned (for example, work-group sizes are hard-coded in the libraries, the use of shared and private memory requires manual code modification of the kernels, etc.). Moreover, BLAS libraries (clMath [27] and cuBlas [77]) do not require autotuning because these libraries are already configured with a set of optimal parameters for their target architectures. Our autotuning framework searches for the most appropriate optimizations (compiler flags) by generating many different code variants and executing each of them on the target hardware. It searches through combinations of PPCG's compiler flags that include the work group sizes, tile sizes, whether to use shared memory, whether to use private memory and which loop distribution heuristic to use (out of two possible heuristics). The autotuning of each benchmark suite takes several hours (except for the six kernels generated from VOBLA, which altogether take up to two days as the search space is larger).

The code generated by PPCG for a given kernel is optionally instrumented to measure the wall clock execution time of that kernel. This time includes kernel execution, data copy (between host memory and GPU device memory), and any kernel code executed on the host CPU. It does not include device initialization and release, nor kernel compilation time. This measured wall clock execution time is the time we report below. In order to exclude compilation time, we either invoke a dry-run computation that is not timed beforehand (caching the compiled kernels), or subtract the compilation time from the total duration, depending on how the reference benchmark compiles and invokes its kernels. We use OpenCL profiling tools to further analyze the performance of the reference code and the PPCG generated code (to get the number of cache misses, the number of device global memory accesses, the GPU occupancy, etc.). Each test is run 30 times and the median of the speedups over the reference benchmark is reported.

We use four GPU platforms for the experimental evaluation: an Nvidia GTX470 GPU (with AMD Opteron Magny-Cours, 2×12 cores and 16GB of RAM), an ARM Mali T604 GPU (with dual-core ARM Cortex-A15 CPU and 2 GB of RAM), an AMD Radeon HD 5670 GPU (with Intel Core2 Quad CPU Q6700 and 8 GB RAM) and an AMD Radeon R9 285 GPU (with Intel Xeon CPU E5-2640, 8 cores and 32 GB of RAM).

5.2 Benchmarks

5.2.1 Image Processing Benchmark Suite

We studied a set of image processing kernels covering computationally intensive parts of a computer vision stack of RealEyes. The benchmark suite includes simple image filters as well as composite image processing algorithms. This is the same benchmark used in the evaluation section in Chapter 3. For each kernel in the benchmark suite, we compare a straightforward PENCIL implementation of the kernel (without any optimization), with a call to the equivalent kernel in the OpenCL implementation of the OpenCV image processing library [78].

The benchmark suite contains 7 image processing kernels:

Color conversion. A pixel-wise operation that changes the color representation of each pixel in an image.

Image resize. A common image manipulation step used to resize images.

General 2D convolution. A two-dimensional convolution of the image implemented by calculating the weighted sum of the area around the pixel using a kernel matrix for weights. It is used to perform a wide range of image processing operations.

Gaussian smoothing. A process of blurring an image by a Gaussian function. It is often used before edge filtering as it helps in a better detection of the more prominent edges on the image.

Affine warping. Affine transformations in two dimensions include translation, scaling, rotation, reflection. This operation is often used to revert the effects of an unwanted transformation, such as image stabilization of shaky videos. In this transformation, each pixel of the source image is moved to a position on the destination image, determined by the transformation matrix, then using interpolation the pixel values are calculated on the target image.

Dilate. A morphological operator where the output at a given pixel position is the maximum value within the neighborhood. It is used to extend or add a border to white areas on the image.

Basic image histogram. A widely used kernel that computes the tonal distribution in a digital image.

One important characteristic of image processing kernels is that they contain non static-affine code: non static-affine array accesses, non static-affine if conditionals and non static-affine loop bounds that a classical polyhedral compiler does not handle efficiently since they do not fit the traditional restrictions of the polyhedral model. The conditional `if (se[e][r] != 0)` in Figure 4.3 is an example of such non static-affine code.

Benchmark	Non static-affine conditionals	Non static-affine read accesses	Non static-affine write accesses
resize	6	4	0
dilate	3	0	0
color conversion	0	0	0
affine warping	16	4	0
2D convolution	4	1	0
gaussian smoothing	8	2	0
basic histogram	0	1	1

Table 5.1 Statistics about patterns of non static-affine code in the image processing benchmark

Table 5.1 shows statistics about the patterns of non static-affine code in the image processing benchmark.

The benchmark exhibits many patterns of non static-affine code: 5 out of 7 kernels have non static-affine conditionals, 5 out of 7 kernels have non static-affine read accesses, 1 kernel has non static-affine write accesses. To be able to efficiently handle these kernels, a polyhedral compiler needs to be able to handle not only the non static-affine conditionals, and the non static-affine read accesses, but also the non static-affine write accesses. Write array accesses are more difficult to handle because they prevent the compiler, in general, from determining whether the loop is parallel or not.

Benchmark	support for non static-affine code	independent	assume	kill
resize	required	-	-	33% ↑
dilate	required	-	-	10% ↑
color conversion	-	-	-	34% ↑
affine warping	required	-	-	23% ↑
2D convolution	required	-	20% ↑	21% ↑
gaussian smoothing	required	-	-	47% ↑
basic histogram	-	required	-	-

Table 5.2 Effect of enabling support for individual PENCIL features on the ability to generate code and on gains in speedups

The kernels in this benchmark require the following PENCIL features: support for non static-affine code, `independent` directive, and the `__pencil_assume` and `__pencil_kill` builtins. Table 5.2 shows the list of PENCIL features that were useful in the image processing benchmark suite. It shows whether a given PENCIL feature is required for OpenCL code generation

and shows the gain in speedup obtained when support for that feature is enabled (compared to the case where support for that feature is disabled). We only show the effect on performance on one test platform (Nvidia GTX), the effect on the other platforms is similar. The symbol "-" indicates that the absence of the feature does not have any effect on the generated code.

The table shows that support for non static-affine code is required to be able to generate OpenCL code for 5 out of 7 kernels in the benchmark. In *basic histogram*, the use of the `independent` directive enables the parallelization and OpenCL code generation for the kernel which is difficult otherwise. In *dilate*, assuming that the size of the structuring element (the array that represents the neighborhood used to compute each pixel) is less than 16×16 enables `PPCG` to map that array to shared memory. This assumption allows `PPCG` to generate code that is 20% faster compared to the case where the assumption is not used. The `__pencil_kill` builtin allows `PPCG` to generate code that is 28% faster compared to the case where the `kill` builtin is not used. `__pencil_kill` in this case mainly eliminates extra data copies that `PPCG` generates to move data between host and device memories.

Table 5.3 shows the speedups of the `PPCG` generated OpenCL code over the baseline OpenCV 2.4.10 OpenCL implementation. We use the same image to evaluate all the kernels (2880×1607, 1.5 MB image).

Benchmark	Nvidia GTX 470	ARM Mali T604	AMD Radeon HD 5670	AMD Radeon R9 285
resize	1.00	1.25	2.47	8.09
dilate	0.59	0.32	0.25	2.91
color conversion	1.32	2.37	1.56	1.11
affine warping	1.06	1.93	2.44	2.85
2D convolution	0.91	-	0.95	2.53
gaussian smoothing	0.92	0.97	0.51	1.61
basic histogram	0.45	0.42	0.16	4.34

Table 5.3 Speedups of the OpenCL code generated by `PPCG` over OpenCV

`__pencil_kill` helps `PPCG` to eliminate spurious data copies but no other optimization on the data copies is applied by `PPCG`. In all the kernels, the amount of data copied by the `PPCG` generated code (when `__pencil_kill` is used) is exactly equal to the amount of data copied by the reference benchmark code. As a consequence, the speedups (or slowdowns) listed in the table are due to faster (or slower) kernel executions and not to a difference in data copy time. This is true for all the test platforms except for the AMD Radeon R9 285 platform. The particularity of this platform is discussed later in this section.

The speedup in *color conversion* and in *resize* for the Nvidia, ARM and AMD Radeon HD 5670 test platforms is due to the tiling of the 2D loop in each of these two kernels which enhances data locality considerably (up to 56% less L1 cache misses on the Nvidia platform for *color conversion*). In *affine warping*, the speedup is due to two optimizations: thread

coarsening where multiple work-items are merged together leading to less redundant computations and tiling which enhances data locality (up to 65% less L1 cache misses on the Nvidia platform).

In the *basic histogram* kernel, the code automatically generated by `PPCG` still does not meet the performance of the hand optimized OpenCV implementation of the histogram for all the test platforms. The OpenCV code is faster because each workgroup computes its own local histogram placed in shared memory, and then the different local histograms are combined into one final histogram (a reduction). Automatic generation of OpenCL code that exploits this kind of reductions is not yet supported by `PPCG`.

For *dilate*, the OpenCV code is vectorized while the current `PPCG` OpenCL backend still does not support the generation of vectorized code. The lack of vectorization in the `PPCG` generated code affects the performance more on the AMD and the ARM test platforms. Moreover, in the OpenCV code for *dilate*, the input image array is mapped into shared memory while `PPCG`'s shared memory heuristic decides not to map this array into shared memory. As a consequence, the `PPCG` generated code accesses global GPU memory 175× more often compared to the OpenCV code, which leads to a decrease in performance. The same problem in the shared memory heuristic applies to *gaussian smoothing*.

While `PPCG` can generate code for *2D convolution*, the OpenCV reference implementation for *2D convolution* could not be run on the ARM Mali GPU, as it uses hardcoded shared memory and workgroup sizes that both exceed the limits of ARM Mali.

On the AMD Radeon R9 285 platform, the speedups of the `PPCG` generated kernels over the OpenCV kernels are due to the slow data copies that OpenCV performs. The data copies are slower because OpenCV, on this platform, decides to add padding to the input image to get aligned memory accesses. In order to do that, OpenCV uses the OpenCL `clEnqueueWriteBufferRect` function which copies data from host to device memory and adds padding at the same time. `PPCG` in contrast uses the `clEnqueueWriteBuffer` OpenCL function which only copies data from host to device memory. Using `clEnqueueWriteBufferRect` is 7× slower than the use of `clEnqueueWriteBuffer`. This difference explains the high speedups that were obtained for the `PPCG` generated code on this platform. Other than this difference in data copies, there is no other significant difference between the speedups obtained on the AMD Radeon R9 285 platform and the AMD Radeon HD 5670 platform. Note that, although the use of `clEnqueueWriteBufferRect` may be less efficient in these tests, it may be more efficient in other cases where only one data copy is performed and many filters are applied on the same input image.

5.2.2 Rodinia and SHOC Benchmark Suites

When choosing benchmarks from the Rodinia [25] and SHOC [30] benchmark suites for writing in PENCIL (reverse-engineering from OpenCL to PENCIL), we decided to focus our resources on a selection of benchmarks that offer diversity (cover different Berkeley "motifs" [4] such as dense and sparse linear algebra, structured grids, and graph traversal), and pose a challenge to traditional polyhedral compilers arising from non static-affine code. We chose five benchmarks (presented in Table 5.4). Three of these benchmarks are particularly challenging for a polyhedral compiler as they exhibit patterns of non static-affine code. We show the benefit of using PENCIL to implement these benchmarks and compare the performance of the `PPCG` generated code for each benchmark with the reference Rodinia/SHOC implementations.

Benchmark	Suite	dataset sizes	Description / notes
2D Stencil	SHOC	100 iterations, 4096×4096 grid	On structured grid
Gauss. Elim.	Rodinia	1024×1024 matrix	Dense matrix
SRAD	Rodinia	100 iterations, 502×458 image	Image enhancement
SpMV	SHOC	16384 rows	Sparse matrix-vector multiplication
BFS	Rodinia	4 million nodes	Breadth-first search on a graph

Table 5.4 Selected benchmarks from Rodinia and SHOC

Benchmark	support for non static-affine code	independent
2D Stencil	-	-
Gauss. Elim.	-	-
SRAD	required	-
SpMV	required	-
BFS	required	required

Table 5.5 PENCIL Features that are useful for SHOC and Rodinia benchmarks

The benchmarks require support for non static-affine code and the use of the `independent` directive. Table 5.5 shows the effect of these features on the ability of `PPCG` to generate OpenCL code. Other PENCIL features do not have any effect on these benchmarks.

The table shows that supporting non static-affine code is required for OpenCL code generation in 3 out of 5 benchmarks. The non static-affine code patterns in these benchmarks include non-affine read accesses, non-affine conditionals and non-affine write accesses. The non-affine write accesses in *BFS* are particularly difficult to handle since they prevent the compiler from parallelizing the code, requiring the use of the `independent` directive.

Table 5.6 shows speedups. The speedups in *Stencil* and *Gaussian* are mainly due to tiling which enhances data locality and reduces cache misses ($4\times$ less L1 cache misses for *Stencil* on Nvidia GTX 470). For *SRAD*, the PENCIL-generated OpenCL code is significantly slower than the reference benchmark, mainly because `PPCG` did not map a reduction in *SRAD* to OpenCL

(as PPCG does not support the generation of parallel reductions yet). This leads to unnecessary data transfers between the part of *SRAD* mapped to host and the part of *SRAD* mapped to device hence the slowdown. In *BFS*, the generated OpenCL code is slightly slower than the reference code also due to unnecessary data transfers that PPCG generates. This happens because PPCG does not handle while loops yet (currently only the while loop body is mapped to GPU, which makes PPCG generate a data copy between host and device at the beginning and end of each while loop iteration).

Benchmark	Nvidia GTX 470	ARM Mali T604	AMD Radeon HD 5670	AMD Radeon R9 285
2D Stencil	3.44	3.04	2.68	5.76
Gauss. Elim.	0.67	1.54	4.39	2.58
SRAD	0.22	0.34	0.43	0.56
SpMV	1.17	1.67	1.04	1.08
BFS	0.65	0.78	0.43	0.72

Table 5.6 Speedups for the OpenCL code generated by PPCG for the selected Rodinia and SHOC benchmarks

Rodinia and SHOC as well as the previously analyzed image processing benchmarks are examples of the use of PENCIL as a standalone language. In the next section, we show an example where PENCIL is used as an intermediate language for DSL compilers.

5.2.3 VOBLA DSL for Linear Algebra

VOBLA is a domain specific language designed by ARM for implementing linear algebra algorithms [13]. It provides a compact and generic representation for linear algebra algorithms using an imperative programming style.

The main idea behind VOBLA is to decouple the description of algorithms from the implementation details. VOBLA programs are compiled into PENCIL and this PENCIL code is compiled to OpenCL code for a wide range of platforms. The generated OpenCL code is competitive with hand-optimized OpenCL code while it requires much less effort from the programmer. A template system allows to define generic functions on generic types and access patterns. Each instance of a template function specifies the types to be used (single or double precision, real or complex numbers) and how arrays are stored (row or column major order, sparse or dense matrix). This makes the code more generic. Any parallelism or possible optimization inherent to the algorithm is not obscured by implementation details.

The main control flow operators defined in VOBLA include `if`, `for`, `forall` and `while`. The `if` and the `while` operators have semantics similar to their counterparts in C and PENCIL. The `for` and `forall` operators are used to iterate over a scalar range (using a multi-dimensional iteration space) or over an array. `forall` loops are used to indicate that the iterations of a loop can be executed in any order.

To access arrays with a common interface, VOBLA mainly uses array iterators. They enumerate the elements or sub-structures (e.g., the rows of a matrix) of an array. A template function only sees the iterators available for a particular array but not the implementations of these iterators. The implementation is specified in the layout description and can be tuned separately for each storage format. Some storage formats also provide a way to access an element by its coordinates, allowing more complicated algorithms to be expressed. Array iterators benefit both the compactness and the clarity of the code, providing an easy, compact and standardized way to iterate over arrays.

For very regular code, the programmer can also use array operators. They provide basic operations on arrays (addition, multiplication, etc) in an even more compact and clean way. The compiler automatically chooses the best access pattern to implement them.

The following VOBLA example first defines a template function that iterates over the non-zero elements of the `X` matrix and adds each `X[i][j]` element to `Y[i][j]`. The template function is then instantiated and named `add_mat_csr`.

```
// A simple VOBLA template function
function add_mat(in X: Sparse<Value>[n][n],
                 out Y: Value[n][n]) {
  // Iterate over the non-zero elements of X
  // and add each X[i][j] to Y[i][j]
  Y[i][j] += Xij forall i, j, Xij in X.sparse;
}

// Instantiate the template
export add_mat<Float>(
  X is Csr, // X: a sparse matrix in CSR format
  Y is Transposed Array) // Y: a transposed matrix
  as add_M_csr; //Name of PENCIL generated function
```

We only provide a brief description of VOBLA and its compilation into PENCIL. A detailed description is available in [13].

The VOBLA-to-PENCIL compiler is quite simple. It does not perform any sophisticated analyses or optimizations. Advanced loop optimizations and the mapping to OpenCL code are all handled by the PENCIL compiler. The VOBLA-to-PENCIL compiler generates only the following PENCIL features: the `independent` directive, the `assume` builtin and the `restrict` type qualifier. No other PENCIL feature needs to be generated. The `kill` builtin is only useful to eliminate spurious data copies in non-static control code and is not needed for the purely

static control code generated by VOBLA. Summary functions are only needed when library functions are called from PENCIL which is not the case in the VOBLA generated code.

The `independent` directive, the `assume` builtin and the `restrict` type qualifier are generated in the following way:

- `forall` VOBLA operators are translated into `for` loops that are annotated with the `independent` directive.

- `assume` builtins are inferred from the relations between array sizes in the VOBLA program. Such information allows the PENCIL compiler to simplify the generated code and avoid unnecessary checks.

  For example, thanks to the statement `C = A + B`, the compiler can infer that the sizes of `A` and `B` are equal and can generate a `__pencil_assume` to indicate that. This information enables the PENCIL compiler to avoid handling the case where `(size_A != size_B)`. This information is useful for instance if the PENCIL compiler decides to fuse two loops over `A` and `B`.

- The VOBLA compiler annotates all the arrays with the `restrict` type qualifier in the PENCIL generated code. This is correct because of the way VOBLA is designed. In VOBLA, the only way to create aliasing is by calling a function and passing the same array to that function more than once in the same call. But even in this case, there is no aliasing in the generated pencil function because the compiler creates a new version of that function where the different arrays that actually represent the same array are merged into a single array.

We used VOBLA to implement a set of linear algebra kernels including *gemver* (vector multiplication and matrix addition), *2mm* (2 matrix multiplications), *3mm* (3 matrix multiplications), *gemm* (general matrix multiplication), *atax* (matrix transpose and vector multiplication) and *gesummv* (scalar, vector and matrix multiplication). Many of these kernels are a sequence of BLAS function calls.

We compare the code generated from `PPCG` for these kernels with equivalent code that calls BLAS library functions. The VOBLA implementation is first compiled to PENCIL using the VOBLA-to-PENCIL compiler and then PENCIL is mapped to the GPU using `PPCG`. We compare the generated code with two highly optimized BLAS library implementations.

- We use the clMath 2.2.0 [27] BLAS library provided by AMD for comparison on AMD platforms.

- We use the cuBlas 5.5 [77] BLAS library provided by Nvidia for comparison on the Nvidia platform. In this case we use `PPCG` to generate CUDA code instead of OpenCL code.

We do not provide a comparison on the Mali GPU as no reference BLAS library is available for Mali to this date. We use 4096×4096 as a matrix size for all the benchmarks.

The only PENCIL features that are beneficial to the VOBLA generated code are the `assume` builtin and the `restrict` type qualifier. The `independent` directive is not needed as the kernels do not contain any non-affine write accesses. The `restrict` type qualifier is mandatory to eliminate aliasing problems.

Writing a hand-tuned kernel in OpenCL is a complicated task which requires deep knowledge of the target architecture and requires significant time dedicated by a specialized engineer. When the kernels are written in VOBLA, no OpenCL knowledge is necessary, which, in our experience, notably simplifies linear algebra code writing and reduces development time.

Table 5.7 shows the gains in speedups obtained when support for the `assume` builtin is enabled (measured on the Nvidia platform). When the `assume` builtin is used, the generated code is significantly faster. For example, the generated code for *gemm*, when `assume` is used, is 71% faster than the generated code without `assume`. This happens because `PPCG` simplifies the control flow in the generated kernels using the information provided through the `assume` builtin.

Benchmark	assume
gemver	06% ↑
2mm	84% ↑
3mm	91% ↑
gemm	71% ↑
atax	13% ↑
gesummv	02% ↑

Table 5.7 Gains in performance obtained when support for the `assume` builtin is enabled in VOBLA

Benchmark	Nvidia GTX 470	AMD Radeon HD 5670	AMD Radeon R9 285
gemver	1.17	2.14	0.39
2mm	0.91	0.62	0.14
3mm	0.87	0.66	0.12
gemm	1.09	0.69	0.19
atax	0.88	1.79	0.37
gesummv	1.03	1.83	0.33

Table 5.8 Speedups obtained with `PPCG` over highly optimized BLAS libraries

Table 5.8 shows the speedups of the kernels generated by `PPCG` over BLAS libraries. The `PPCG` generated kernels for the Nvidia and the AMD HD 5670 platforms were close in perfor-

mance to the highly optimized BLAS library calls for *2mm*, *3mm*, *atax* and *gemm* (e.g., 0.69× for *gemm* on the AMD platform). The main optimizations applied in these kernels are loop fusion, tiling, and the use of shared and private memories. The BLAS code still outperforms the `PPCG` generated code as it implements many other optimizations including vectorization (clMath) and the use of register tiling (cuBlas) which are not yet supported by `PPCG`. The speedups for *gesummv* and *gemver* are due to loop fusion and tiling that are performed across different library calls. The *gemver* kernel, for example, is a sequence of 6 BLAS library calls. Although the individual BLAS library functions are highly optimized, better performance can be obtained by fusing and tiling across function calls. `PPCG` is able to perform these optimizations, and thus outperforms the sequence of BLAS library calls by a factor of 2.14× on AMD Radeon. clMath is highly vectorized and tuned for the AMD Radeon R9 285, since `PPCG` still does not support vectorization it fails to reach the performance levels for clMath on this platform.

5.2.4 SpearDE DSL for Data-Streaming Applications

SpearDE [62] is a domain-specific modeling and programming framework for signal processing applications, designed by *Thales Research and Technology*. An evaluation of PENCIL on two representative SpearDE applications is provided in [6]. The two applications are: Space-Time Adaptive Processing (*STAP*) and Adaptive Beamformer (*ABF*). Both are common signal processing applications for radar systems.

5.3 Discussion of the Results

The evaluation section shows how PENCIL features improve the ability of `PPCG` to generate OpenCL code in two ways:

- By enabling the generation of OpenCL through features such as the `independent` directive, and summary functions and through support for non static-affine code. Support for non static-affine code was required in 52% of the benchmarks. The `independent` directive was required in *basic histogram* and *bfs* while summary functions were required in *ABF* (from SpearDE, which is not discussed in detail in this dissertation).

- By enhancing the quality of the generated code through features such as the `assume` and `kill` builtins. The `assume` builtin improved performance in 33% of the benchmarks while the `kill` builtin improved performance in 38% of the benchmarks.

The experiments also provide an assessment about the efficiency of the generated code compared to a large set of highly optimized reference codes: 47% of the generated kernels

outperform the reference implementations. Yet, the evaluation also shows some limitations in the current tools including limitations in generating parallel reductions, loop fusion heuristics, in handling while loops and the lack of vectorization and register tiling. This motivates more work on the tool chain itself to address these issues.

Currently, our framework uses autotuning to select the most appropriate work group sizes, tile sizes, to decide whether to use shared memory, whether to use private memory and which loop distribution heuristic to use. Although exhaustive search using this framework may be suitable for small applications, it is not suitable for larger applications as the exploration of all the possible optimizations for each kernel would create a large search space (since there are many kernels in each one of those applications). This motivates the integration of better heuristics and the integration of parametric tiling in the compilation flow in order to reduce the size and the overall time of the autotuning.

5.4 Conclusion and Future Work

We presented PENCIL, an intermediate language for DSL compilers and for accelerator programming, designed to simplify static code analysis. The design of PENCIL is unique in its combination of sequential semantics, strict compliance with the syntax and semantics of C, and a rich set of static analysis helpers through attributes and pragmas. It makes many forms of non static-affine code and access patterns amenable to advanced loop transformation and parallelization techniques based on the polyhedral framework. We ported a representative set of benchmarks to PENCIL, some of which written in a DSL and compiled to PENCIL. We parallelized these applications automatically on GPU platforms, demonstrating unprecedented expressiveness capabilities for a polyhedral framework. Our experiments validate the use of PENCIL together with an optimizing compiler as building blocks for the implementation of languages and compilers aiming for performance portability.

Chapter 6

Conclusions and Perspectives

This dissertation presented three limitations in polyhedral compilation along with a practical solution to each one of these.

6.1 Contributions

- We proposed a relaxed permutability criterion that allows a compiler to ignore false dependences between iteration-private live ranges, allowing the compiler to discover larger bands of tilable loops. We presented the rationale that led to the design of this criterion along with an experimental evaluation. The main contributions of this work are the following: (1) using our relaxed permutability criterion, loop tiling can be applied without the need for any expansion or privatization, the impact on the memory footprint is minimized, and the overheads of array expansion are avoided, (2) our criterion allows the tiling of programs that privatization does not allow.

- We presented a statement clustering algorithm and two clustering heuristics. The experimental evaluation showed that statement clustering succeeded in reducing the scheduling time by a median factor of $8\times$ across all the benchmarks and by more than $1000\times$ for some benchmarks. Moreover, the evaluation showed that the technique does not lead to any significant loss in optimization opportunities. The main contributions of this work are the following: (1) we proposed an algorithm that makes a clear distinction between the clustering itself and the clustering heuristic, and (2) we proposed an algorithm to decide about the validity of a given clustering decision.

- We presented PENCIL, an intermediate language for DSL compilers and for accelerator programming. PENCIL is a subset of the C99 language carefully designed to capture static properties essential to enable more precise data dependence analysis. It provides

also a set of language constructs that includes the `independent` directive, the `assume` and `kill` builtins and summary functions. These language constructs enable parallelizing compilers to generate more efficient code. We evaluated PENCIL on a representative set of benchmarks: 47% of the generated kernels outperform the reference implementations. Our experiments validate the use of PENCIL together with an optimizing compiler as building blocks for the implementation of languages and compilers aiming for performance portability.

This work was partly supported by the European FP7 project CARP id. 287767, and many contributions resulted from fruitful discussions and close collaboration among the CARP project partners.

6.2 Future Directions

Currently, statement-clustering is used only to speed up the affine scheduling step. An interesting extension of this work is to use the idea of statement clustering to speed up the steps of dependence analysis and code generation in a polyhedral compiler. To do this, the macro-statements need to be constructed during the extraction of the polyhedral representation of the program. The SCC clustering heuristic cannot be used in this case since it relies on the use of the dependence graph which is not available during the step of the extraction of the polyhedral representation.

In addition to the SCC and the BB heuristics, another heuristic that can be used is a heuristic that groups similar statements into one macro-statement. Two statements are similar if the only difference between them is in their access relations (array subscripts). The following example shows a set of statements that are similar.

```
for (i=0; i<N; i++)
{
    A[i][0] = B[i][0];
    A[i][1] = B[i][1];
    A[i][2] = B[i][2];
    A[i][3] = B[i][3];
    A[i][4] = B[i][4];
    A[i][5] = B[i][5];
}
```

Regarding the PENCIL tool chain, the evaluation showed some limitations, including limitations in the support for parallel reductions, loop fusion heuristics, in handling while loops

and the lack of vectorization. This motivates more work on the tool chain to address these issues.

An autotuning framework that uses heuristics instead of exhaustive search is also needed in order to be able to handle large applications (applications with multiple kernels). The use of such an autotuning framework will make autotuning more usable in the context of large applications.

Many extensions to the PENCIL language are being considered. Among these extensions:

- A builtin function to mark a given array element as being unused after some PENCIL regions and as being used only after a specific PENCIL region. This information allows the PENCIL compiler to avoid copying the array into host memory after the end of the PENCIL region (performing a copy back to the host memory is the default behavior since the compiler cannot figure out whether an array element is unused after the PENCIL region as the compiler cannot analyze non PENCIL code).

- A builtin to indicate that an operation is atomic (atomic add, atomic write, ...). The semantics of such an operation are not yet defined, since the concept of "atomicity" is only defined for languages with parallel semantics which is not the case of PENCIL. Yet, we want to enable the user to indicate that a given operation should be implemented as an atomic operation if the PENCIL code gets parallelized. We are investigating how this can be done.

Apart from these short-term perspectives, we believe that the polyhedral framework is very promising for many reasons. It provides powerful algorithms to reason about and for implementing loop nest transformations. Thanks to its fine grain code representation, exact dependence analysis and fine grain scheduling are possible. In addition to this, during the last few years, many tools around polyhedral compilation have been developed by the research community which facilitates research in the polyhedral model. In our experience, the proposed extensions that enable the polyhedral frameworks to handle non static-affine code [14, 41], seem to address their goals reasonably successfully, thus we do not consider this problem as a high priority.

Although polyhedral compilation is showing good results, more work is still needed to make use of its full potential in production compilers. One of the most urgent problems is the problem of scalability. This problem mainly appears in the step of scheduling as we have shown in Chapter 3. The solution we proposed can be combined with a stack of other practical solutions to reduce the scheduling time of programs. More ambitious work should focus on developing new scheduling algorithms that inherently do not suffer from this scalability problem (probably such algorithms will not rely on ILP – Integer Linear Programming).

One other urgent issue in the current polyhedral compilers is related to the need of heuristics that guide the search for optimizations. Manually designing a heuristic is a tedious task. Such a heuristic is in general designed for a specific architecture and new heuristics need to be crafted continuously for new architectures. Developing automatic methods for the creation of heuristics that extend the existing techniques such as the use of machine learning [73, 90, 91] is a possible solution to this problem.

Bibliography

[1] J. R. Allen and K. Kennedy. *PFC: A program to convert Fortran to parallel form*. Rice University. Department of Mathematical Sciences, 1982.

[2] R. Allen and K. Kennedy. Automatic translation of fortran programs to vector form. *ACM Trans. Program. Lang. Syst.*, 9(4):491–542, Oct. 1987. ISSN 0164-0925. doi: 10.1145/29873.29875. URL http://doi.acm.org/10.1145/29873.29875.

[3] M. S. Alnæs, A. Logg, K. B. Ølgaard, M. E. Rognes, and G. N. Wells. Unified form language: A domain-specific language for weak formulations of partial differential equations. *ACM Trans. Math. Softw.*, 40(2):9, 2014.

[4] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, University of California, Berkeley, Dec 2006.

[5] R. Baghdadi, A. Cohen, C. Bastoul, L.-N. Pouchet, and L. Rauchwerger. The potential of synergistic static, dynamic and speculative loop nest optimizations for automatic parallelization. In *Workshop on Parallel Execution of Sequential Programs on Multi-core Architectures (PESPMA'10)*, Saint-Malo, France, June 2010.

[6] R. Baghdadi, U. Beaugnon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, J. Absar, S. v. Haastregt, A. Kravets, A. Lokhmotov, A. Betts, J. Ketema, A. F. Donaldson, R. David, and E. Hajiyev. Pencil: a platform-neutral compute intermediate language for accelerator programming. In *under review*, 2015. URL http://www.di.ens.fr/~baghdadi/public/papers/pencil.pdf.

[7] R. Baghdadi, A. Cohen, T. Grosser, S. Verdoolaege, A. Lokhmotov, J. Absar, S. Van Haastregt, A. Kravets, and A. Donaldson. PENCIL Language Specification. Research Report RR-8706, INRIA, May 2015. URL https://hal.inria.fr/hal-01154812.

[8] U. Banerjee. Data dependence in ordinary programs. Thése de master, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Nov. 1976.

[9] U. K. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Norwell, MA, USA, 1988. ISBN 0898382890.

[10] M. M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic c-to-cuda code generation for affine programs. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*,

CC'10/ETAPS'10, pages 244–263, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-11969-7, 978-3-642-11969-9. doi: 10.1007/978-3-642-11970-5_14. URL http://dx.doi.org/10.1007/978-3-642-11970-5_14.

[11] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *IEEE Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'04)*, pages 7–16, Juan-les-Pins, France, Sept. 2004.

[12] C. Bastoul and P. Feautrier. Improving data locality by chunking. In *Proceedings of the 12th International Conference on Compiler Construction*, CC'03, pages 320–334, Berlin, Heidelberg, 2003. Springer-Verlag. ISBN 3-540-00904-3. URL http://dl.acm.org/citation.cfm?id=1765931.1765962.

[13] U. Beaugnon, A. Kravets, S. van Haastregt, R. Baghdadi, D. Tweed, J. Absar, and A. Lokhmotov. VOBLA: A vehicle for optimized basic linear algebra. In *LCTES*, pages 115–124, 2014.

[14] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. The polyhedral model is more widely applicable than you think. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, CC'10/ETAPS'10, pages 283–303, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-11969-7, 978-3-642-11969-9. doi: 10.1007/978-3-642-11970-5_16. URL http://dx.doi.org/10.1007/978-3-642-11970-5_16.

[15] A. Bernstein. Analysis of programs for parallel processing. *Electronic Computers, IEEE Transactions on*, EC-15(5):757–763, Oct 1966. ISSN 0367-7508. doi: 10.1109/PGEC.1966.264565.

[16] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, B. Pottenger, L. Rauchwerger, P. Tu, et al. Polaris: The next generation in parallelizing compilers. In *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*, pages 141–154, 1994.

[17] U. Bondhugula. Compiling affine loop nests for distributed-memory parallel architectures. In *High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for*, pages 1–12. IEEE, 2013.

[18] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 101–113, Tucson, AZ, USA, 2008. ACM. ISBN 978-1-59593-860-2. doi: 10.1145/1375581.1375595. URL http://portal.acm.org/citation.cfm?id=1375581.1375595.

[19] U. Bondhugula, O. Gunluk, S. Dash, and L. Renganarayanan. A model for fusion and code motion in an automatic parallelizing compiler. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 343–352, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0178-7. doi: 10.1145/1854273.1854317. URL http://doi.acm.org/10.1145/1854273.1854317.

[20] F. Bouchez, A. Darte, C. Guillon, and F. Rastello. Register allocation: What does the NP-completeness proof of Chaitin et al. really prove? or revisiting register allocation: Why and how. In *LCPC'06*, LNCS, New Orleans, Louisiana, 2006. Springer Verlag.

[21] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Symp. on Programming Language Design and Implementation (PLDI'90)*, White Plains, New York, June 1990.

[22] CAPS Enterprise, Cray Inc., Nvidia, and the Portland Group. The OpenACC application programming interface, v1.0, Nov. 2011.

[23] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. A domain-specific approach to heterogeneous parallelism. In *PPoPP*, pages 35–46, 2011.

[24] G. J. Chaitin, M. A. Auslander, A. K. C. J. Cocke, M. E. Hopkins, and P. W.Markstein. Register allocation via coloring. *Computer languages*, 6:47–57, 1981.

[25] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, pages 44–54, 2009.

[26] C. Chiw, G. Kindlmann, J. Reppy, L. Samuels, and N. Seltzer. Diderot: A parallel DSL for image analysis and visualization. In *PLDI*, pages 111–120, 2012.

[27] clMath Developers Team. OpenCL math library, 2013. URL https://github.com/clMathLibraries.

[28] A. Cohen and E. Rohou. Processor virtualization and split compilation for heterogeneous multicore embedded systems. In *Design Automation Conference (DAC'10)*. IEEE Computer Society, June 2010. 6 pages. Special session on Embedded Virtualization.

[29] Cray Inc. Cray standard c/c++ reference manual. Technical Report S–2179–81, 2012.

[30] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The scalable heterogeneous computing (SHOC) benchmark suite. In *GPGPU*, pages 63–74, 2010.

[31] A. Darte and G. Huard. New complexity results on array contraction and related problems. *J. VLSI Signal Process. Syst.*, 40(1):35–55, May 2005. ISSN 0922-5773. doi: 10.1007/s11265-005-4937-3. URL http://dx.doi.org/10.1007/s11265-005-4937-3.

[32] A. Darte, Y. Robert, and F. Vivien. *Scheduling and Automatic Parallelization*. Birkhauser Boston, 1st edition, 2000. ISBN 0817641491.

[33] J. J. Dongarra, H. W. Meuer, E. Strohmaier, et al. Top500 supercomputer sites. *Supercomputer*, 13:89–111, 1997.

[34] R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua. *Experience in the automatic parallelization of four Perfect-Benchmark programs*. Springer, 1992.

[35] P. Feautrier. Array expansion. In *Proceedings of the 2nd international conference on Supercomputing*, pages 429–441, St. Malo, France, 1988. ACM. ISBN 0-89791-272-1. doi: 10.1145/55364.55406. URL http://portal.acm.org/citation.cfm?id=55406.

[36] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, Feb. 1991. ISSN 0885-7458. doi: 10.1007/BF01407931. URL http://www.springerlink.com/content/k6253346u547n15v/.

[37] P. Feautrier. Some efficient solutions to the affine scheduling problem. i. one-dimensional time. *International journal of parallel programming*, 21(5):313–347, 1992.

[38] P. Feautrier. Some efficient solutions to the affine scheduling problem. part ii. multidimensional time. *International journal of parallel programming*, 21(6):389–420, 1992.

[39] P. Feautrier. Scalable and structured scheduling. *International Journal of Parallel Programming*, 34(5):459–487, 2006.

[40] M. Griebl. Automatic parallelization of loop programs for distributed memory architectures. Habilitation thesis. Facultät für Mathematik und Informatik, Universität Passau, 2004.

[41] M. Griebl and J.-F. Collard. Generation of synchronous code for automatic parallelization of while loops. In *EURO-PAR'95 Parallel Processing*, pages 313–326. Springer, 1995.

[42] T. GROSSER, A. GROESSLINGER, and C. LENGAUER. Polly — performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010, 2012. doi: 10.1142/S0129626412500107. URL http://www.worldscientific.com/doi/abs/10.1142/S0129626412500107.

[43] T. Grosser, A. Cohen, J. Holewinski, P. Sadayappan, and S. Verdoolaege. Hybrid hexagonal/classical tiling for GPUs. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 66:66–66:75, New York, NY, USA, 2014. ACM.

[44] T. Grosser, S. Verdoolaege, and A. Cohen. Polyhedral ast generation is more than scanning polyhedra. *ACM Transactions on Programming Languages and Systems*, 2015.

[45] M. Gupta. On privatization of variables for data-parallel execution. In *Parallel Processing Symposium, 1997. Proceedings., 11th International*, pages 533–541. IEEE, 1997.

[46] S. Hack, D. Grund, and G. Goos. Register allocation for programs in SSA-form. In *CC'06*, pages 247–262, 2006.

[47] J. L. Henning. Spec cpu2000: Measuring cpu performance in the new millennium. *Computer*, 33(7):28–35, July 2000. ISSN 0018-9162. doi: 10.1109/2.869367. URL http://dx.doi.org/10.1109/2.869367.

[48] J. Holewinski, L.-N. Pouchet, and P. Sadayappan. High-performance code generation for stencil computations on gpu architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 311–320, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1316-2. doi: 10.1145/2304576.2304619. URL http://doi.acm.org/10.1145/2304576.2304619.

[49] F. Irigoin and R. Triolet. Supernode partitioning. In *Symp. on Principles of Programming Languages (POPL'88)*, pages 319–328, San Diego, CA, Jan. 1988.

[50] ISO. The ANSI C standard (C99). Technical Report WG14 N1124, ISO/IEC, 1999.

[51] ISO. The ANSI C standard (C11). Technical Report WG14 N1570, ISO/IEC, 2011. URL http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf.

[52] ISO. ISO/IEC 14882:2011 – information technology – programming languages – c++. Technical report, 2011.

[53] A. Jimborean, P. Clauss, J.-F. Dollinger, V. Loechner, and J. M. M. Caamaño. Dynamic and speculative polyhedral parallelization using compiler-generated skeletons. *International Journal of Parallel Programming*, 42(4):529–545, 2014.

[54] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation (Frontiers'95)*, FRONTIERS '95, pages 332–, Washington, DC, USA, 1995. IEEE Computer Society. ISBN 0-8186-6965-9. URL http://dl.acm.org/citation.cfm?id=528717.796653.

[55] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., 2002. ISBN 1-55860-286-0. URL http://portal.acm.org/citation.cfm?id=502981.

[56] Khronos Group. Opencl 1.2 specification. http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/, 2011.

[57] J. Knoop, O. Rüthing, and B. Steffen. Optimal code motion: Theory and practice. *ACM Trans. on Programming Languages and Systems*, 16(4):1117–1155, 1994.

[58] W. Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, Dec. 1992. ISSN 1057-4514. doi: 10.1145/161494.161501. URL http://doi.acm.org/10.1145/161494.161501.

[59] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. pages 75–88, San Jose, CA, USA, Mar 2004.

[60] H. Lee and M. Aaftab. The OpenCL specification 2.0. 2015.

[61] V. Lefebvre and P. Feautrier. Automatic storage management for parallel programs. *Parallel Computing*, 24:649–671, 1998. ISSN 01678191. doi: 10.1016/S0167-8191(98)00029-5.

[62] E. Lenormand and G. Edelin. An industrial perspective: A pragmatic high end signal processing design environment at Thales. In *SAMOS*, pages 52–57, 2003.

[63] Z. Li. Array privatization for parallel execution of loops. In *Proceedings of the 6th international conference on Supercomputing*, pages 313–322, Washington, D. C., United States, 1992. ACM. ISBN 0-89791-485-6. doi: 10.1145/143369.143426. URL http://portal.acm.org/citation.cfm?id=143369.143426.

[64] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine partitions. *Parallel Comput.*, 24(3-4):445–475, May 1998. ISSN 0167-8191. doi: 10.1016/S0167-8191(98)00021-0. URL http://dx.doi.org/10.1016/S0167-8191(98)00021-0.

[65] A. W. Lim, G. I. Cheong, and M. S. Lam. An affine partitioning algorithm to maximize parallelism and minimize communication. In *Proceedings of the 13th International Conference on Supercomputing*, ICS '99, pages 228–237, New York, NY, USA, 1999. ACM. ISBN 1-58113-164-X. doi: 10.1145/305138.305197. URL http://doi.acm.org/10.1145/305138.305197.

[66] D. B. Loveman. High performance Fortran. *Parallel & Distributed Technology: Systems & Applications*, 1(1):25–42, 1993.

[67] M. Luján, T. L. Freeman, and J. R. Gurd. Oolala: An object oriented analysis and design of numerical linear algebra. In *OOPSLA*, pages 229–252, 2000.

[68] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. Array-data flow analysis and its use in array privatization. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '93*, pages 2–15, Charleston, South Carolina, United States, 1993. doi: 10.1145/158511.158515. URL http://dl.acm.org/citation.cfm?id=158515.

[69] S. Mehta and P.-C. Yew. Improving compiler scalability: optimizing large programs at small price. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 143–152. ACM, 2015.

[70] B. Meister, N. Vasilache, D. Wohlford, M. M. Baskaran, A. Leung, and R. Lethin. R-stream compiler. In D. A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1756–1765. Springer, 2011. ISBN 978-0-387-09765-7. URL http://dx.doi.org/10.1007/978-0-387-09766-4_515.

[71] H. W. Meuer, E. Strohmaier, J. J. Dongarra, and H. D. Simon. Top500 supercomputer sites 06/2000. *Lawrence Berkeley National Laboratory*, 2000.

[72] S. Midkiff. *Automatic Parallelization: An Overview of Fundamental Compiler Techniques*. Morgan & Claypool Publishers, Feb. 2012. ISBN 9781608458417.

[73] A. Monsifrot, F. Bodin, and R. Quiniou. A machine learning approach to automatic production of compiler heuristics. In *Proceedings of the 10th International Conference on Artificial Intelligence: Methodology, Systems, and Applications*, AIMSA '02, pages 41–50, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-44127-1. URL http://dl.acm.org/citation.cfm?id=646053.677574.

[74] S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann, 1997.

[75] R. T. Mullapudi, V. Vasista, and U. Bondhugula. Polymage: Automatic optimization for image processing pipelines. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*,

ASPLOS '15, pages 429–443, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2835-7. doi: 10.1145/2694344.2694364. URL http://doi.acm.org/10.1145/2694344.2694364.

[76] Nvidia. Nvidia CUDA programming guide 4.0, 2011.

[77] Nvidia. *cuBLAS Library User Guide*, Oct. 2012.

[78] OpenCV Developers Team. Open source computer vision library, 2002. URL http://opencv.org/.

[79] OpenMP Architecture Review Board. OpenMP application program interface, v3.0, May 2008.

[80] D. C. Oppen. A 2 22pn upper bound on the complexity of presburger arithmetic. *Journal of Computer and System Sciences*, 16(3):323–332, 1978.

[81] S. Pop, A. Cohen, C. Bastoul, S. Girbal, G.-a. Silber, and N. Vasilache. GRAPHITE: polyhedral analyses and optimizations for GCC. *In proceedings of the 2006 GCC developers summit*, page 2006, 2006. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.77.9149.

[82] W. Pugh. The omega test: A fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 4–13, New York, NY, USA, 1991. ACM. ISBN 0-89791-459-7. doi: 10.1145/125826.125848. URL http://doi.acm.org/10.1145/125826.125848.

[83] W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 4–13. ACM, 1991.

[84] F. Quilleré and S. Rajopadhye. Optimizing memory usage in the polyhedral model. *ACM Trans. on Programming Languages and Systems*, 22(5):773–815, Sept. 2000.

[85] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. P. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *PLDI*, pages 519–530, 2013.

[86] G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471, Sept. 1994. ISSN 0164-0925. doi: 10.1145/186025.186041. URL http://doi.acm.org/10.1145/186025.186041.

[87] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for non-shared memory machines. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 111–120, New York, NY, USA, 1991. ACM. ISBN 0-89791-459-7. doi: 10.1145/125826.125893. URL http://doi.acm.org/10.1145/125826.125893.

[88] T. Rompf and M. Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled dsls. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, GPCE '10, pages 127–136, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0154-1. doi: 10.1145/1868294.1868314.

[89] R. Stansifer. Presburger's article on integer airthmetic: Remarks and translation. Technical Report TR84-639, Cornell University, Computer Science Department, September 1984. URL http://techreports.library.cornell.edu:8081/Dienst/UI/1.0/Display/cul.cs/TR84-639.

[90] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly. Meta optimization: improving compiler heuristics with machine learning. In *ACM SIGPLAN Notices*, volume 38, pages 77–90. ACM, 2003.

[91] M. W. Stephenson. *Automating the construction of compiler heuristics using machine learning*. PhD thesis, Massachusetts Institute of Technology, 2006.

[92] J. E. Stone, D. Gohara, and G. Shi. OpenCL: a parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, 12(3):66–73, 2010.

[93] A. Sukumaran-Rajam, J. M. M. Caamano, W. Wolff, A. Jimborean, and P. Clauss. Speculative program parallelization with scalable and decentralized runtime verification. In *Runtime Verification*, pages 124–139. Springer, 2014.

[94] W. Thies, F. Vivien, J. Sheldon, and S. Amarasinghe. A unified framework for schedule and storage optimization. In *Proc. of the 2001 PLDI Conf.*, 2001.

[95] K. Trifunovic, A. Cohen, D. Edelsohn, F. Li, T. Grosser, H. Jagasia, R. Ladelsky, S. Pop, J. Sjodin, and R. Upadrasta. GRAPHITE two years after: First lessons learned from Real-World polyhedral compilation, Jan. 2010.

[96] K. Trifunovic, A. Cohen, R. Ladelski, and F. Li. Elimination of Memory-Based dependences for Loop-Nest optimization and parallelization. In *3rd GCC Research Opportunities Workshop*, Chamonix, France, 2011.

[97] P. Tu and D. Padua. Automatic array privatization. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, volume 768 of *Lecture Notes in Computer Science*, pages 500–521. Springer Berlin / Heidelberg, 1994.

[98] U. Banerjee. Data dependence in ordinary programs. Master thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Nov. 1976.

[99] N. Vasilache. *Scalable Program Optimization Techniques in the Polyhedral Model*. PhD thesis, University of Paris-Sud, INRIA, Sept. 2007.

[100] N. Vasilache, C. Bastoul, A. Cohen, and S. Girbal. Violated dependence analysis. In *Proceedings of the 20th annual international conference on Supercomputing*, pages 335–344, Cairns, Queensland, Australia, 2006. ACM. ISBN 1-59593-282-8. doi: 10.1145/1183401.1183448. URL http://portal.acm.org/citation.cfm?id=1183401.1183448.

[101] S. Verdoolaege. isl: An integer set library for the polyhedral model. In J. v. d. Hoeven, M. Joswig, N. Takayama, and K. Fukuda, editors, *Mathematical Software - ICMS 2010*, volume 6327, pages 299–302. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-15581-9. URL http://www.springerlink.com/content/p4968q65x854510t/.

[102] S. Verdoolaege. Pencil support in pet and PPCG. Technical Report RT-457, INRIA Paris-Rocquencourt, Mar. 2015.

[103] S. Verdoolaege and T. Grosser. Polyhedral extraction tool. In *IMPACT*, 2012.

[104] S. Verdoolaege, J. C. Juega, A. Cohen, J. I. Gómez, C. Tenllado, and F. Catthoor. Polyhedral parallel code generation for CUDA. *ACM Trans. Archit. Code Optim.*, 9(4), Jan. 2013.

[105] W. Von Hagen. *The definitive guide to GCC*, volume 2. Springer, 2006.

[106] N. Wirth. Extended Backus-Naur Form Syntaxt Specification. *ISO/IEC 14977*, 1996.

[107] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. on Parallel and Distributed Systems*, 2(4):452–471, 1991.

[108] M. Wolfe. Iteration space tiling for memory hierarchies. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, pages 357–361, Philadelphia, PA, USA, 1989. Society for Industrial and Applied Mathematics. ISBN 0-89871-228-9. URL http://dl.acm.org/citation.cfm?id=645818.669220.

[109] D. Wonnacott. Omega calculator. In D. A. Padua, editor, *Encyclopedia of Parallel Computing*. Springer, 2011. ISBN 978-0-387-09765-7. URL http://dx.doi.org/10.1007/978-0-387-09766-4_515.

Appendix A

EBNF Grammar for PENCIL

The reserved words are in bold.

<pencil>	←	<top level definition>∗
<top level definition>	←	<function> | <type definition> | <global constant>
<global constant>	←	<variable declaration>**'='**<init expression>**';'**
<type definition>	←	(<typedef> | <struct definition>)**';'**
<typedef>	←	**typedef**<base type><name><array suffix>∗
<struct definition>	←	**struct**<name>**'{'**(<variable declaration>**';'**)∗**'}'**
<array suffix>	←	**'['**(<array attribute>∗)<expression>**']'**
<array attribute>	←	**const** | **restrict** | **static**
<pointer>	←	**'*'**
<declarator>	←	(<pointer>∗)<direct declarator>
<direct declarator>	←	<name>(<array suffix>∗) | **'('**<declarator>**')'**<array suffix>∗
<base type>	←	<scalar type fragment>+ | <type attribute>∗**struct**?<name>
<scalar type fragment>	←	<type specifier> | <type attribute>
<type attribute>	←	**const**
<type specifier>	←	**bool** | **_Bool** | **char** | **short** | **int** | **long** | **float** | **half** | **double** | **signed** | **unsigned**
<variable declaration>	←	<base type><declarator>
<init expression>	←	<expression> | <array init expression>
<array init expression>	←	**'{'**<constant>(**','**<constant>)∗**'}'**
<function>	←	**static**?<function type> <name>**'('**<function args>**')'**<attribute>∗ <function body>
<function type>	←	<base type> | **void**

Figure A.1 PENCIL syntax as an EBNF.

```
<function body>            ← <block> | ';'
<attribute>                ← __attribute__'('('<attr>')')'
<attr>                     ← const | pencil | pencil_access'('<name>')'
<function args>            ← | <variable declaration>(','<variable declaration>)*
<block>                    ← '{'<statement>*'}'
<statement>                ← <assignment>';' | <for> | <while> | <if> | <block> | <return>
                             | <block variable declaration> | <call statement>
                             | break';' | continue';'
<assignment>               ← <lvalue>
                             ('=' | '+=' | '-=' | '%=' | '*=' | '/=' | '^=' |
                              '&=' | '|=' | '»=' | '«=')
                             <expression>
                             | <lvalue>'++' | <lvalue>'--'
                             | '++'<lvalue> | '--'<lvalue>
<while>                    ← while'('<expression>')'<block>
<if>                       ← if'('<expression>')'<block>(else<block>)?
<return>                   ← return<expression>?';'
<block variable declaration> ← <variable declaration>('='<init expression>)?';'
<call statement>           ← <call expression>';'
<for directive>            ← '#'pragma pencil(ivdep | <independent>)
<independent>              ← independent(<reduction>)*
<name list>                ← <name>(','<name>)*
<reduction>                ← reduction'('('+' | '-' | '*' | min | max)':'<name list>')'
<for step>                 ← '++'<name> | '--'<name> | <name>'++' | <name>'--'
                             | <name>'+='<constant> | <name>'-='<constant>
<for>                      ← <for directive>*
                             for'('<base type>?<name>'='<expression>';'
                               <name>('>' | '<' | '>=' | '<=')<expression>';'
                               <for step>')'
                             <block>
<expression>               ← <ternary expression>
<ternary expression>       ← <LOR expression>('?'<expression>':'<ternary expression>)?
<LOR expression>           ← <LAND expression>('||'<LAND expression>)*
<LAND expression>          ← <BitOR expression>('&&'<BitOR expression>)*
<BitOR expression>         ← <BitXOR expression>('|'<BitXOR expression>)*
<BitXOR expression>        ← <BitAND expression>('^'<BitAND expression>)*
```

Figure A.1 PENCIL syntax as an EBNF continued overleaf.

<BitAND expression> ← <EQ expression>(**'&'**<EQ expression>)∗
<EQ expression> ← <CMP expression>((**'='** | **'!='**)<CMP expression>)∗
<CMP expression> ← <shift expression>((**'>'** | **'<'** | **'>='** | **'<='**)<shift expression>)∗
<shift expression> ← <plus expression>((**'«'** | **'»'**)<plus expression>)∗
<plus expression> ← <mult expression>((**'+'** | **'-'**)<mult expression>)∗
<mult expression> ← <cast expression>((**'*'** | **'/'** | **'%'**)<cast expression>)∗
<cast expression> ← (**'('**<scalar type fragment>+**')'**)∗<unary expression>
<unary expression> ← **' '**<cast expression> | **'-'**<cast expression>
| **'+'**<cast expression> | **'!'**<cast expression>
| <sizeof expression> | <postfix expression>
<lvalue> ← <subscription>
<postfix expression> ← <call expression> | <subscription>
<call expression> ← <name>**'('**(<expression>(**','**<expression>)∗)?**')'**
<subscription> ← <term>(**'['**<expression>**']'** | **'.'**<name>)∗
<term> ← <name> | <constant> | **'('**<expression>**')'**
<constant> ← <HEX humber> | <DEC number> | <OCT number>
| <Floating point number>

Figure A.1 PENCIL syntax as an EBNF continued overleaf.

Appendix B

Personal Publications

The following is a list of publications I wrote or contributed to while working towards my doctoral degree.

- R. Baghdadi, U. Beaugnon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, J. Absar, S. v. Haastregt, A. Kravets, A. Lokhmotov, A. Betts, J. Ketema, A. F. Donaldson, R. David, E. Hajiyev. "PENCIL: a Platform-Neutral Compute Intermediate Language for Accelerator Programming", The 24th International Conference on Parallel Architectures and Compilation Techniques (PACT 2015), San Francisco, CA, USA.
- R. Baghdadi, A. Cohen, S. Verdoolaege, T. Grosser, J. Absar, S. v. Haastregt, A. Kravets, A. Lokhmotov, A. F. Donaldson. "PENCIL Language Specification". Research Report RT-8706, INRIA, Paris-Rocquencourt, May. 2015.
- U. Beaugnon, A. Kravets, S. V. Haastregt, R. Baghdadi, D. Tweed, J. Absar, A. Lokhmotov, "VOBLA: A Vehicle for Optimized Basic Linear Algebra", LCTES'14, Edinburgh, UK, 2014.
- R. Baghdadi, A. Cohen, S. Verdoolaege, K. Trifunovic, "Improved Loop Tiling Based on the Removal of Spurious False Dependences", in ACM Transactions on Architecture and Code Optimization (TACO), 2013.
- R. Baghdadi, A. Cohen, S. Guelton, S. Verdoolaege, J. Inoue, T. Grosser, G. Kouveli, A. Kravets, A. Lokhmotov, C. Nugteren, F. Waters, A. F. Donaldson, "Pencil: Towards a Platform-Neutral Compute Intermediate Language for DSLs". WOLFHPC'12, Salte Lake, 2012.

Index