UNIVERSITÄT PASSAU

Fakultät für Informatik und Mathematik

Bachelors Thesis

Analysis and Extension of Existing Tiling Algorithms for Stencil Computations

Michael Freitag

Supervisor:	Prof. Christian Lengauer, Ph.D.
Tutor:	Dr. Armin Größlinger

July 30, 2014

Abstract

Stencil codes form the performance-critical core of a number of applications in scientific computing. The parallelization and optimization of stencil codes are subject of ongoing research. Numerous high-performance transformation frameworks for stencil computations have been developed, most of which employ iteration domain tiling as a key transformation for optimization. However, the parallel tiled code that is generated by these frameworks may suffer from a notable load imbalance, due to redundant inter-tile dependences, and thus redundant synchronization.

In this thesis, we give an overview of three selected transformation frameworks that perform optimizations based on iteration domain tiling. We then propose an approach which alleviates the load balance issues that the respective tiling algorithms may incur. A tile dependence graph is built which represents all tiles and inter-tile dependences suggested by the original algorithm. Using a polyhedral dependence analysis, redundant dependences are identified and eliminated subsequently. Finally, the actual stencil computation is performed by traversing the adjusted tile dependence graph in parallel. In doing so, parallel tiles are scheduled dynamically on the available processor cores. Experimental evaluation on a number of stencil codes and hardware platforms shows a significant performance improvement over the original tiling algorithms.

Contents

Chapter 1

Introduction

Stencil computations determine the values of points in a grid of some dimensionality by repeatedly evaluating a given function of a grid point and its neighbors. They can be found at the core of many applications in various scientific computing domains [1], where they are employed, for example, in solvers for partial differential equations, geometric modeling and image processing [2].

The parallelization and optimization of stencil computations is subject of ongoing research. The most prevalent approach today is the subdivision of the iteration domain into smaller pieces, called ***tiling***, which improves both load balance and data locality. While it is possible to optimize stencil codes manually with great success [3], many frameworks that automate this process have been developed over the years, due to the complexity of the transformations that are commonly involved. These frameworks typically achieve increases in performance of up to an order of magnitude in comparison to a naive implementation of a stencil code [1, 4–6]. Examples of such frameworks are the Pochoir stencil compiler [4], the PLUTO compiler [5,7] or the Stencil Domain Specific Language (SDSL) compiler [1,8]. But although all of these already produce high-performance code, we have found previously that Pochoir's tiling algorithm could be optimized further for one-dimensional stencil codes [9].

We developed a two-stage optimization approach, the key idea of which is to decompose the iteration domain into tiles in a separate preprocessing stage, before actually performing the stencil computation in the second stage. This allows for otherwise infeasible optimizations to be performed in both stages. In the first stage, a *tile dependence graph* that represents the tiled iteration domain is generated using a slightly modified version of Pochoir's trapezoidal algorithm [4]. In the second stage, this graph is traversed in parallel following a free schedule, which respects all dependences between tiles by definition. This is possible with minimal synchronization. Using this approach, we were able to achieve a performance gain of up to 20 % over Pochoir for a set of one-dimensional stencil benchmarks.

Therefore, as preliminary evaluation of our approach yielded such promising results, we perform further evaluation for different tiling strategies and stencil computations of higher dimensionality in this thesis. For that purpose we developed a C++ template library that allows the user to specify a stencil computation in a concise manner, similar to the Pochoir specification language [4], and that provides an interface to the implementation of our optimization approach. We implemented support for the trapezoidal tiling algorithm employed by the Pochoir compiler, the diamond tiling algorithm employed by the PLUTO compiler and the split tiling algorithms used in the SDSL compiler.

In summary, we make the following contributions. We present a strategy to perform further optimizations on existing tiling algorithms for stencil computations. Additionally, we provide an implementation of this approach for selected state-of-the-art research tiling algorithms. Finally, we empirically evaluate our technique on a number of stencil benchmarks and compare the results to those of the original implementation of the algorithms.

The rest of the thesis is arranged as follows. In the remainder of Chapter 1 we will give an introduction to stencil computations, tiling and the terminology we use. Chapter 2 provides an overview of the three selected stencil compilation frameworks introduced above. In Chapter 3 we describe our optimization approach and its implementation in detail, before performing extensive experimental evaluation thereof in Chapter 4. Finally, we draw conclusions in Chapter 5.

1.1 Stencil Computations

A ***stencil*** determines the value of a point in a d-dimensional grid as a function f of previous values of this point and its neighbors, which is generally called the ***kernel function***. A ***stencil computation*** applies the kernel function repeatedly to all points in the grid over multiple time steps, which is why an additional dimension representing time is often added to the grid (see Figure 1.1).

```
for (t = 0; t < T; t++) {
  for (i = 1; i < N-1; i++) {
    A[t+1][i] = 0.125*(A[t][i-1] - 2*A[t][i] + A[t][i+1]);
  }
}
```

Figure 1.1: A loop-based implementation of a one-dimensional heat equation stencil, using ghost cells to implement boundary conditions.

In the following, we will limit ourselves to kernel functions that only perform uniform accesses, i.e., the set of points that are referenced during the evaluation of f can be represented by a set of $(d+1)$-dimensional distance vectors $\vec{v} = (\Delta v_t, \Delta v_0, \dots, \Delta v_{d-1}) \in \mathbb{Z}^- \times \mathbb{Z}^d$ [10], which we call the ***shape*** S of the stencil. A vector $\vec{v} \in S$ indicates that, when evaluating $f(\vec{p})$ for some point $\vec{p}$ in the grid, f references the value of the point $\vec{p} + \vec{v}$. Using its shape, we can now define some more properties of a stencil. Let $i \in \{0, \dots, d-1\}$ identify a spatial dimension of the grid. Then the value $w_i := \max_{\vec{v} \in S} |\Delta v_i|$ is called the ***width*** of S in dimension i. Similarly, we define the value $\sigma_i := \max_{\vec{v} \in S} |\Delta v_i / \Delta v_t|$ to be the ***slope*** of S in dimension i. The width and slope of a stencil assist with identifying points that the evaluation of the kernel function is dependent on (see Figure 1.2). Finally, we define the ***depth*** of the stencil to be the value $\max_{\vec{v} \in S} |\Delta v_t|$.

For points that are near the spatial boundary of the grid, it can not be guaranteed that all accesses within S correspond to points within the actual grid. For that reason, a ***boundary function*** is used to provide values for such points. If the boundary function maps outside accesses to grid points on the opposite side of the grid, the stencil computation is ***periodic***, otherwise it is ***non-periodic***. Non-periodic boundary conditions are often implemented by

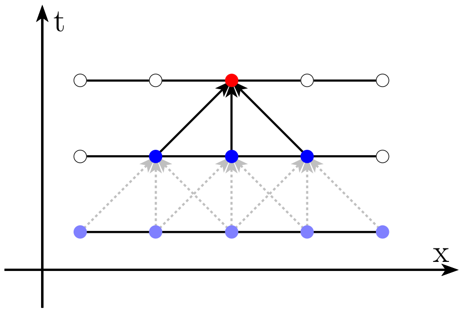


Figure 1.2: Visualization of the one-dimensional heat equation stencil depicted in Figure 1.1. The shape of this stencil is $S = \{(-1,-1),(-1,0),(-1,1)\}$, its width in dimension 0 is $w_0 = 1$ and its slope in dimension 0 is $\sigma_0 = 1$. The slope of S defines a cone that all grid points influencing the value of the red point lie within.

adding w_i ghost cells, for which the kernel function is not evaluated, on each side of the grid in the spatial dimension i. Thus, there are no more actual boundary accesses, as each former boundary access now maps to an access to a ghost cell (see Figure 1.1).

1.2 Tiling and the Polyhedral Model

The obvious way of implementing a stencil computation is to traverse the grid using nested loops, as is shown in Figure 1.1. However, such loop-based implementations generally exhibit poor performance due to insufficient data locality and parallelism [4]. A key transformation that addresses both of these problems is ***iteration space tiling*** [1,5,10–13], i.e., subdividing the iteration space into smaller ***tiles*** which can be processed atomically (see Figure 1.5). Tiling is frequently characterized by shape and size of the tiles, both of which influence the performance of tiled execution significantly [5]. Depending on which aspect of a stencil computation is intended to be optimized by tiling, different shapes and sizes may be feasible.

Tiling with data locality in mind requires tiles to be small enough to fit into faster memory, i.e. registers and on-chip caches, so that it is possible to reuse data without having to load it from the slower main memory. In order to achieve high levels of coarse-grained parallelism, tile shapes and sizes have to be found that allow for independent execution of as many tiles as possible while at the same tile minimizing the amount of communication and synchronization needed [13]. We shall elaborate on this in more detail during the presentation of different tiling algorithms in Chapter 2.

Advanced iteration space tiling is notoriously difficult to perform manually because of inter-tile dependences and possibly irregular loop structures. For this reason, the polyhedral model has proved to be a powerful tool when discussing tiling and transformations on loop nests in general [5, 13]. Within the polyhedral model, instances of a statement are viewed as integer points in a well-defined space, representing the iteration domain of the loop enclosing that statement. This allows for a precise representation of irregularly shaped tiles as well as the characterization of dependences between statements and tiles using methods from Linear Algebra and Integer Linear Programming. Some libraries supporting polyhedral operations also provide assistance in code generation, for example the integer set library [14]. We employ this library to analyze inter-tile dependences and in part for code generation.

1.2.1 Overview of the Polyhedral Model

Let us consider a loop nest of some dimensionality, as is shown, for example, in Figure 1.3(a). The loop nest is said to be ***affine***, or to be a ***static control part (SCoP)***, if all loop bounds and conditionals are affine functions of the surrounding loop iterators and parameters, i.e. constants with values unknown at compile time. Static control parts constitute the subset of general loop nests which can be represented in the polyhedral model [15]. It is possible for an affine loop nest to be imperfectly nested, or to contain multiple statements in the loop bodies. In the following, we give an introduction to the polyhedral model for perfectly nested affine loop nests that contain only a single statement S. Furthermore, we assume that there are no parameters in the loop bounds and conditionals.

Each runtime instance of a statement S contained in such a d-dimensional affine loop nest is uniquely identified by the values of the loop counters in that specific iteration. These values are aggregated in the ***iteration vector*** $\vec{x} = (x_{d-1}, \dots, x_0)^T \in \mathbb{Z}^d$ of the statement S, where x_0 is the value of the innermost loop counter, x_1 is the value of the loop counter containing the innermost loop etc. In the loop nest depicted in Figure 1.3(a), for instance, the iteration vector of S would be $\vec{x} = (i, j)^T \in \mathbb{Z}^2$. A valid value would be $\vec{x} = (0, 100)$, which identifies the first iteration of the statement. The order in which runtime instances of S are traversed by the loop nest is now specified by the lexicographic order $\prec$ on the iteration vectors.

The bounds of a loop nest can be modeled by constricting the values of the iteration vector with a set of inequalities. For example, the bounds of the outer loop in Figure 1.3(a) can be specified by the inequalities $i \geq 0$ and $i \leq 99$, or equivalently

$$\begin{aligned} 1 \cdot i + 0 \cdot j - \quad 0 &\geq 0 \\ (-1) \cdot i + 0 \cdot j - (-99) &\geq 0 \end{aligned}$$

Using vector notation, the same inequalities can be written as

$$\begin{aligned} \begin{pmatrix} 1 & 0 \end{pmatrix} \circ \begin{pmatrix} i \\ j \end{pmatrix} - \quad 0 &\geq 0 \\ \begin{pmatrix} -1 & 0 \end{pmatrix} \circ \begin{pmatrix} i \\ j \end{pmatrix} - (-99) &\geq 0 \end{aligned}$$

Each of these inequalities specifies a half-space of $\mathbb{Z}^d$, which is bounded by an affine hyperplane. Given a vector $\vec{h} \in \mathbb{Z}^d$ and $k \in \mathbb{Z}$, an ***affine hyperplane*** $H \subset \mathbb{Z}^d$ is defined by

$$H = \{\vec{x} \in \mathbb{Z}^d \mid \vec{h} \circ \vec{x} - k = 0\}$$

where the vector $\vec{h}$ is normal to the hyperplane and k specifies the offset of H from the origin. An affine hyperplane forms a $d-1$-dimensional affine subspace of $\mathbb{Z}^d$. We also identify a hyperplane with the vector $(\vec{h}, -k)$ of its normal vector and offset. Thus, the affine hyperplanes corresponding to the above inequalities are specified by $H_1 = (1,0,0)$ and

$H_2 = (-1,0,99)$. Likewise, the bounding hyperplanes for the inner loop in Figure 1.3(a) are $H_3 = (0,1,-100)$ and $H_4 = (0,-1,199)$.

Obviously, it is now equivalent to constrain the values of the iteration vector either by a set of inequalities, or by a set of bounding hyperplanes corresponding to these inequalities. Using the latter bounding hyperplanes, we can characterize a ***polyhedron*** $X \subset \mathbb{Z}^d$ that contains all valid iteration vectors for the statement S. For n such bounding hyperplanes H_i, identified by (h_i, k_i), let

$$A = \begin{pmatrix} \vec{h}_1 \\ \vdots \\ \vec{h}_n \end{pmatrix} \in \mathbb{Z}^{n \times d} \quad \text{and} \quad \vec{b} = \begin{pmatrix} -k_1 \\ \vdots \\ -k_n \end{pmatrix} \in \mathbb{Z}^n$$

Then X is defined by

$$X = \{\vec{x} \in \mathbb{Z}^d \mid A\vec{x} + \vec{b} \geq 0\}$$

where $\geq$ is evaluated component-wise. Using augmented matrices and vectors, the same polyhedron can also be defined by

$$X = \left\{ \vec{x} \in \mathbb{Z}^d \mid \begin{pmatrix} A & \vec{b} \end{pmatrix} \cdot \begin{pmatrix} \vec{x} \\ 1 \end{pmatrix} \geq 0 \right\}$$

In the following, we will use the latter definition unless stated otherwise, as it allows for a more concise representation of polyhedra and transformations using only a single matrix. As done before with hyperplanes, we will also identify a polyhedron with the matrix $(A, \vec{b})$. Concluding our running example, the polyhedron which specifies the loop nest displayed in Figure 1.3(a) is shown in Figure 1.3(b).

```
for (int i = 0; i <= 99; i++) {
  for (int j = 100; j <= 199; j++) {
    S(i, j);
  }
}
```

(a)

$$\begin{pmatrix} 1 & 0 & 0 \\ -1 & 0 & 99 \\ 0 & 1 & -100 \\ 0 & -1 & 199 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \\ 1 \end{pmatrix} \geq 0$$

(b)

Figure 1.3: A simple loop nest in (a), and its polyhedral representation in (b)

With a polyhedral representation at hand, transformations for a statement now translate to manipulations of the associated polyhedron and the iteration vector. A d-dimensional ***affine transformation*** ϕ_S for the statement S is defined by

$$\phi_S : \mathbb{Z}^{d+1} \to \mathbb{Z}^{d+1} : \begin{pmatrix} \vec{x} \\ 1 \end{pmatrix} \mapsto T \cdot \begin{pmatrix} \vec{x} \\ 1 \end{pmatrix} \quad \text{with} \quad T = \begin{pmatrix} T' & \vec{t'} \\ 0 \ldots 0 & 1 \end{pmatrix}$$

where T' is an integer matrix and $\vec{t'}$ is an integer vector. The rows of the matrix T' define affine hyperplanes, which are called ***tiling hyperplanes***. To simplify matters, we assume that T is unimodular, i.e. $T \in GL_{d+1}(\mathbb{Z})$, in the remainder of this section. There are multiple approaches that can handle non-unimodular or even non-invertible affine transformations [16–18], however these are far beyond the scope of a brief recapitulation of the polyhedral model.

Let $X = (A, \vec{b})$ now be the polyhedron associated with the statement S. An affine transformation ϕ_S maps each source iteration vector $\vec{x} \in X$ to a target iteration vector $T \cdot \vec{x} \in X'$, with

$$\begin{aligned} X' &= \{T \cdot \vec{x} \in \mathbb{Z}^{d+1} \mid \begin{pmatrix} A & \vec{b} \end{pmatrix} \cdot \vec{x} \geq 0\} \\ &= \{ \quad \vec{y} \in \mathbb{Z}^{d+1} \mid \begin{pmatrix} A & \vec{b} \end{pmatrix} \cdot T^{-1} \cdot \vec{y} \geq 0\} \end{aligned}$$

as T is unimodular. Therefore, X' is itself a polyhedron specified by a matrix $(A, \vec{b}) \cdot T^{-1}$. Unless ϕ_S is an identity, the lexicographic order on the target iteration vectors is different from that on the source iteration vectors. Accordingly, the polyhedron X' reflects a transformed loop nest, since its elements are scanned in a different order. When evaluating the statement S in the transformed loop nest, we can convert the transformed iteration vector back to the original iteration vector by applying T^{-1}.

Continuing the example depicted in Figure 1.3, let ϕ_S be an affine transformation with matrix

$$T = \begin{pmatrix} 1 & -1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \text{and} \quad T^{-1} = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Applying this transformation to the statement S leads to the transformed polyhedron displayed in Figure 1.4(b), which is associated to the loop nest shown in Figure 1.4(a). The original iteration vector $(c_0 + c_1, c_1, 1)$ was obtained by applying T^{-1} to the transformed iteration vector $(c_0, c_1, 1)$.

```
for (int c0 = -199; c0 <= -1; c0++) {
  for (int c1 =  max(-c0, 100);
           c1 <= min(-c0 + 99, 199);
           c1++) {
    S(c0 + c1, c1);
  }
}
```

(a)

$$\begin{pmatrix} 1 & 1 & 0 \\ -1 & -1 & 99 \\ 0 & 1 & -100 \\ 0 & -1 & 199 \end{pmatrix} \cdot \begin{pmatrix} c0 \\ c1 \\ 1 \end{pmatrix} \geq 0$$

(b)

Figure 1.4: The loop nest depicted in Figure 1.3 after applying an affine transformation.

1.2.2 Tiling in the Polyhedral Model

Tiling in the polyhedral model is based on ***rectangular tiling*** of the iteration domain of a statement S. Let $X = (A, \vec{b})$ be the polyhedron representing a loop nest with statement S, and let $\vec{x} = (x_{d-1}, \ldots, x_0) \in X$ be the iteration vector of this statement. Rectangular tiling of a single loop dimension x_i is performed by introducing a new tile dimension z_i as outermost dimension and bounding the loop dimension x_i by the constraint

$$z_i \cdot s \leq i \leq (z_i + 1) \cdot s - 1$$

where $s \in \mathbb{N}^+$ is the tile size. This essentially subdivides the loop dimension x_i into bands of width s, each of which is processed in one iteration of the new z_i loop. Therefore, the polyhedron X' characterizing the tiled loop nest contains the additional loop dimension z_i, which is bounded by the affine hyperplanes corresponding to the above inequality. The lower bound inequality is equivalent to $-s \cdot z_i + i \geq 0$, and thus to

$$\begin{pmatrix} -s & 0 & \dots & 0 & 1 & 0 & \dots & 0 \end{pmatrix} \circ \vec{x} - 0 \geq 0$$

On the other hand, the upper bound inequality is equivalent to $s \cdot z_i - i + (s-1) \geq 0$, i.e.,

$$\begin{pmatrix} s & 0 & \dots & 0 & -1 & 0 & \dots & 0 \end{pmatrix} \circ \vec{x} + (s-1) \geq 0$$

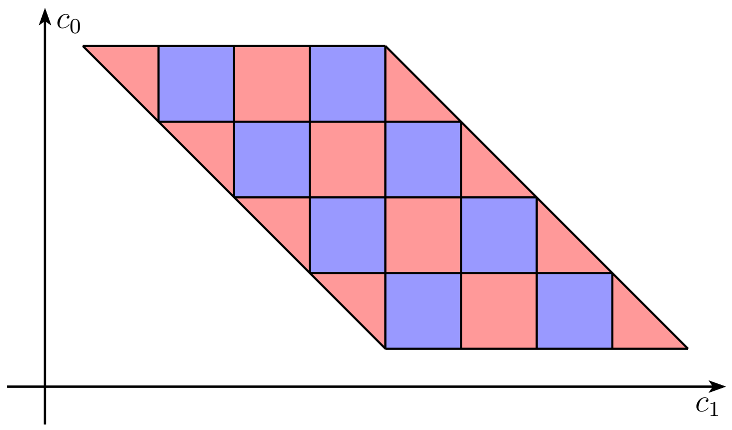
Therefore, the tiled loop nest can be represented by adding these bounding hyperplanes to the original polyhedron X', resulting in

$$X' = \begin{pmatrix} -s & 0 & \dots & 1 & \dots & 0 & 0 \\ s & 0 & \dots & -1 & \dots & 0 & s-1 \\ 0 & & & A & & & \vec{b} \end{pmatrix}$$

Rectangular tiling is always legal, regardless of tile size, if all dimensions $x_{d-1}, \dots, x_i$ up to a certain depth i are tiled, and the loops are tiled starting with the innermost loop. That is, the tile dimensions are nested in the same order as the loop dimensions and thus the tiled iteration vector is $\vec{x}' = (z_{d-1}, z_{d-2}, \dots, z_i, x_{d-1}, \dots, x_0)$.

More complex tile shapes can be achieved by first applying an affine transformation and then performing rectangular tiling of the transformed iteration space. Each tiling hyperplane of the affine transformation defines an affine hyperplane that is parallel to the boundaries of the tiles in the original iteration space. However, tiling may be illegal for specific affine transformations.

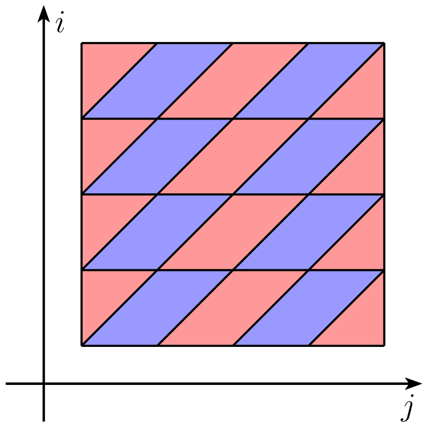
In Figure 1.5(a), rectangular tiling is performed on the loop nest displayed in Figure 1.4 with tile size 25 in both dimensions. This results in the polyhedron depicted in Figure 1.5(b), which represents parallelogram-shaped tiles in the original iteration space, as shown in Figure 1.5(c).



(a)

$$\begin{pmatrix} -25 & 0 & 1 & 0 & 0 \\ 25 & 0 & -1 & 0 & 24 \\ 0 & -25 & 0 & 1 & 0 \\ 0 & 25 & 0 & -1 & 24 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & -1 & -1 & 99 \\ 0 & 0 & 0 & 1 & -100 \\ 0 & 0 & 0 & -1 & 199 \end{pmatrix} \cdot \begin{pmatrix} z0 \\ z1 \\ c0 \\ c1 \\ 1 \end{pmatrix} \geq 0$$

(b)



(c)

Figure 1.5: Rectangular tiling of the loop nest displayed in Figure 1.4. Tiling in the transformed iteration space is shown in (a), which is reflected by the polyhedral description displayed in (b). Rectangular tiling in the transformed iteration space results in parallelogram-shaped tiles in the original iteration space, as shown in (c).

Chapter 2

Tiling Algorithms and Frameworks

The complexity found in the process of optimizing stencil computations has given rise to a number of automated optimization frameworks. In this chapter, a selection of three such frameworks will be presented, introducing both their programming interface and their optimization approach. We selected the Pochoir stencil compiler [4] and the PLUTO compiler [13, 19] for their still high relevance in current research [1, 5], where they have been used as a reference to evaluate the performance of other approaches. Representing a more recent technique, we additionally chose the Stencil Domain Specific Language (SDSL) compiler [1, 20].

2.1 The Pochoir Stencil Compiler

Pochoir provides a source-to-source compiler and a runtime library that enable the user to specify stencil computations in a domain specific language embedded in C++ [4]. The compiler then translates this specification into high-performance Cilk-code [21] that employs a cache-oblivious divide-and-conquer algorithm to perform the actual stencil computation [22]. This algorithm is based on recursive trapezoidal decompositions of the iteration domain [11, 12].

2.1.1 Programming Interface

Stencil computations are implemented using the Pochoir specification language, which consists of several language constructs that are mapped to standard C++ via preprocessor macros. As the Pochoir specification language has already been explained in detail by the developers of Pochoir in the according user manual [23], we only give an overview of programming with Pochoir.

Static information about a stencil computation of dimensionality *dim* is gathered in a central ***Pochoir object*** of type `Pochoir_`*`dim`*`D`. The user can specify grids, boundary functions and kernel functions by using the following language constructs.

- `Pochoir_Array_`*`dim`*`D`(*type*) to define a grid with elements of type *type*
- `Pochoir_Boundary_`*`dim`*`D` to define the boundary function
- `Pochoir_Kernel_`*`dim`*`D` to define the kernel function

```
 1  Pochoir_Boundary_1D(bv, array, t, i)
 2      return 0;
 3  Pochoir_Boundary_End
 4
 5  void heat(int T, int N) {
 6      Pochoir_Shape_1D shape[] = {{0, 0}, {-1, 1}, {-1, 0}, {-1,
           -1}};
 7
 8      Pochoir_Array_1D(double) array(N);
 9      array.Register_Boundary(bv);
10
11      Pochoir_1D pochoir(shape);
12      pochoir.Register_Array(array);
13
14      // ... initialize array ...
15
16      Pochoir_Kernel_1D(kernel, t, i)
17          array(t, i) = 0.125 * (array(t - 1, i + 1) - 2.0 * array
               (t - 1, i) + array(t - 1, i - 1));
18      Pochoir_Kernel_End
19
20      pochoir.Run(T, kernel);
21  }
```

Figure 2.1: Pochoir specification of the one-dimensional heat equation stencil introduced in Figure 1.1. The kernel function is specified in lines 16–18, while the boundary function is specified in lines 1–3. Additionally, the shape of the stencil has to be specified, as is done in line 6. After initializing the required data structures, the stencil computation is initiated in line 20.

The boundary and kernel function definitions can contain arbitrary C++ code, which enables the user to specify complex boundary conditions or multi-statement kernel functions with ease. Finally, this information is assembled in a central Pochoir object. Afterwards, the stencil computation can be started by invoking the `Run` method of that object with the desired number of time steps (see Figure 2.1).

Compilation of a Pochoir program is performed in two steps. In the first step, the Pochoir compiler acts as a preprocessor to the main C++ compiler, performing various transformations and optimizations on the source code (see Section 2.1.2). This optimized source code is then passed to the actual C++ compiler, which generates the final executable using the Pochoir runtime library and the Cilk multithreading extensions. There is also the option to compile a Pochoir program directly without using the Pochoir compiler. This generates an unoptimized but functionally correct executable, which can be useful for debugging purposes.

2.1.2 Optimization Approach

Pochoir is able to produce high-performance code because of two reasons. At the core of its efficiency is the cache-oblivious parallel algorithm that is used to perform a stencil computation. A further performance increase is achieved by applying several compile-time optimizations, such as base case coarsening. In the following, we provide a detailed description of the parallel algorithm as well as the compile-time optimizations.

Trapezoidal Decompositions

Before delving into the cache-oblivious parallel algorithm, it is necessary to study the trapezoidal decompositions it is based upon. Therefore, we begin with some notations and definitions, most of which are borrowed from [4,11,12].

A $(d+1)$-dimensional ***hypertrapezoid***

$$\mathscr{Z} = (t_0, t_1\ ;\ x_{0,0}, x_{1,0}, \sigma_{0,0}, \sigma_{1,0}\ ;\ \ldots;\ x_{0,d-1}, x_{1,d-1}, \sigma_{0,d-1}, \sigma_{1,d-1})$$

is defined as the set of integer points $(t, x_0, \ldots, x_{d-1}) \in \mathbb{Z}^{d+1}$ for which the following conditions hold true.

$$\begin{aligned} t_0 &\leq t < t_1 \\ \forall i \in \{0,\ldots,d-1\} : x_{0,i} + \sigma_{0,i} \cdot (t - t_0) &\leq x_i < x_{1,i} + \sigma_{1,i} \cdot (t - t_0) \end{aligned}$$

The value $\Delta t = t_1 - t_0$ is called the ***height*** of $\mathscr{Z}$, and we call the values $\sigma_{0,i}$ and $\sigma_{1,i}$ the left and right ***slopes*** of $\mathscr{Z}$ in dimension x_i, respectively.

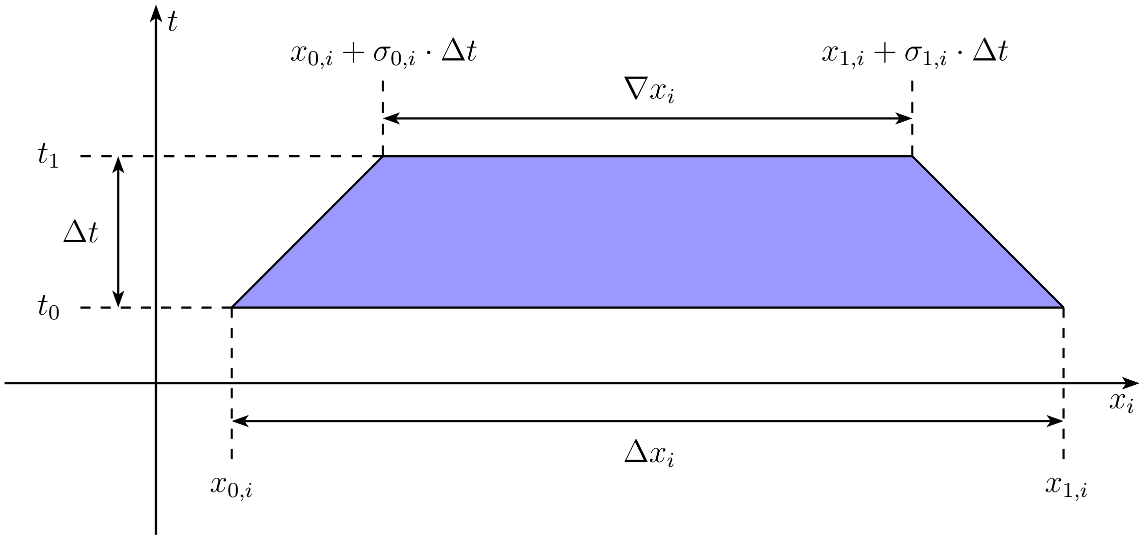
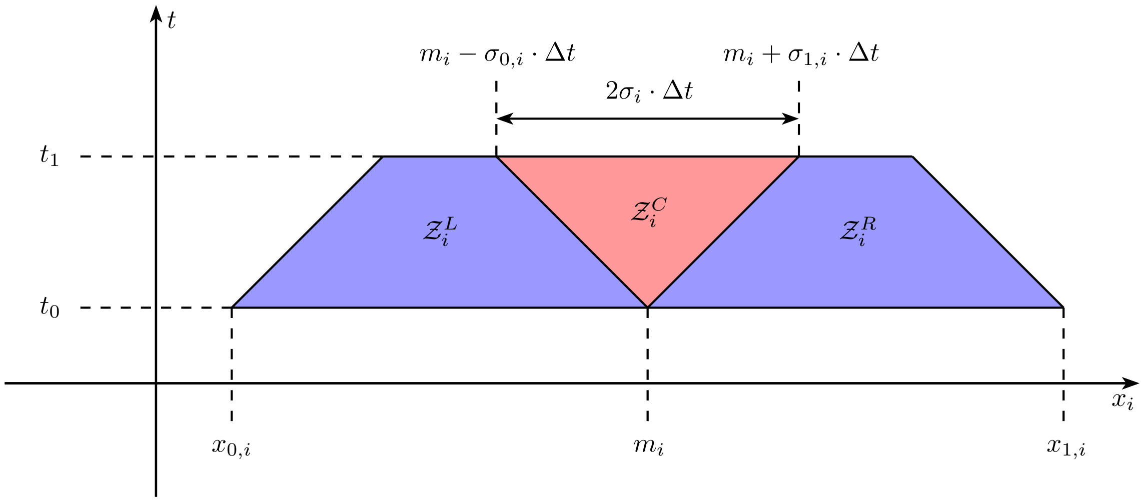

t
$x_{0,i} + \sigma_{0,i} \cdot \Delta t$
$x_{1,i} + \sigma_{1,i} \cdot \Delta t$
∇x_i
t_1
Δt
t_0
x_i
Δx_i
$x_{0,i}$
$x_{1,i}$

Figure 2.2: The (upright) projection trapezoid $\mathscr{Z}_i$ of a well-defined hypertrapezoid $\mathscr{Z}$.

A ***projection trapezoid*** $\mathscr{Z}_i$ of $\mathscr{Z}$ is obtained by projecting $\mathscr{Z}$ onto the dimensions t and x_i, resulting in a trapezoid with bases parallel to the x_i-axis (see Figure 2.2). These bases are of length $\Delta x_i = x_{1,i} - x_{0,i}$ and $\nabla x_i = (x_{1,i} + \sigma_{1,i} \cdot \Delta t) - (x_{0,i} + \sigma_{0,i} \cdot \Delta t)$ respectively. If $\Delta x_i > \nabla x_i$, i.e., the longer base is at $t = t_0$, we say that $\mathscr{Z}$ is ***upright*** in dimension x_i, otherwise $\mathscr{Z}$ is said to be ***inverted*** in dimension x_i. A hypertrapezoid $\mathscr{Z}$ is ***well-defined*** if its height is positive and the bases of all projection trapezoids have positive lengths.

A $d+1$-dimensional hypertrapezoid $\mathscr{Z}$ can be seen as a tile within the iteration space of a d-dimensional stencil computation, each point $(t, x_0, \ldots, x_{d-1})$ representing a grid point $(x_0, \ldots, x_{d-1})$ at time t. Given such a stencil, we can now introduce the concept of spacecuts and timecuts, which form the foundation of trapezoidal decompositions (see Figures 2.3–2.5). As a simplification, we assume that hypertrapezoids are symmetric, i.e., $|\sigma_{0,i}| = |\sigma_{1,i}| = \sigma_i$ with σ_i being the slope of the stencil for all dimensions i. However, this simplification is justified, as most stencils have symmetric shapes.



t
$m_i - \sigma_{0,i} \cdot \Delta t$
$m_i + \sigma_{1,i} \cdot \Delta t$
$2\sigma_i \cdot \Delta t$
t_1
$\mathcal{Z}_i^C$
$\mathcal{Z}_i^L$
$\mathcal{Z}_i^R$
t_0
x_i
$x_{0,i}$
m_i
$x_{1,i}$

Figure 2.3: Decomposing an upright projection trapezoid $\mathscr{Z}_i$ with a spacecut.

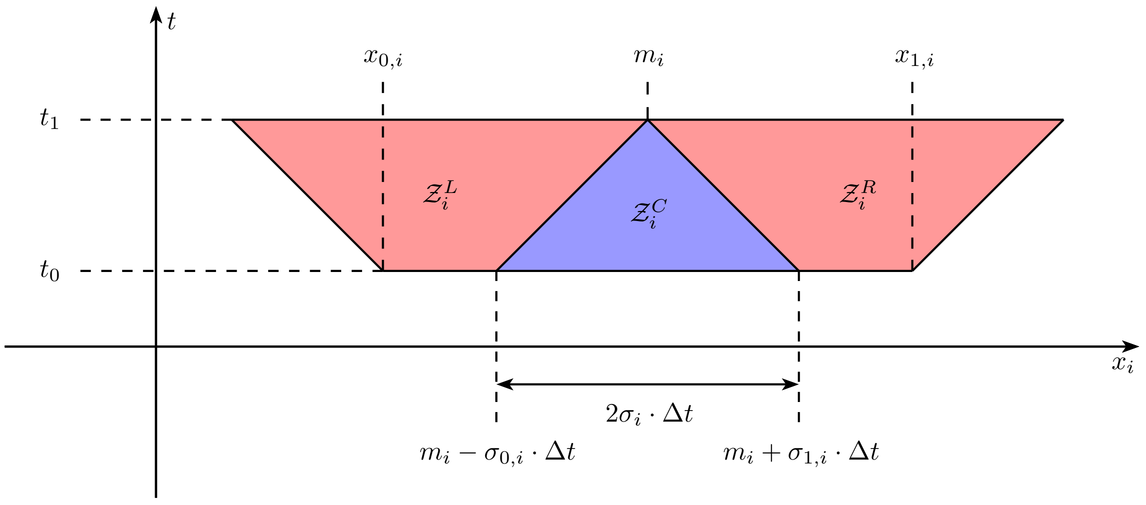
A ***spacecut*** in dimension x_i of a well-defined hypertrapezoid $\mathscr{Z}$ is performed by trisecting the corresponding projection trapezoid $\mathscr{Z}_i$. In order to do so, $\mathscr{Z}_i$ is decomposed along two lines with slopes σ_i and $-\sigma_i$ that run through the center point of its longer base. It is therefore necessary to distinguish between spacecuts of upright and inverted projection trapezoids. Let $\mathscr{Z}_i$ be upright first. Then the center point of its longer base is $m_i = (x_{1,i} - x_{0,i})/2$ and a spacecut results in the following subtrapezoids of $\mathscr{Z}_i$ (see Figure 2.3).

$$\begin{aligned}
\mathscr{Z}_i^L &= (t_0, t_1 \ ; \ x_{0,i}, m_i, \sigma_i, -\sigma_i) \\
\mathscr{Z}_i^C &= (t_0, t_1 \ ; \ m_i, m_i, -\sigma_i, \sigma_i) \\
\mathscr{Z}_i^R &= (t_0, t_1 \ ; \ m_i, x_{1,i}, \sigma_i, -\sigma_i)
\end{aligned}$$

As we chose σ_i to be the slope of the stencil, there are no interdependences between $\mathscr{Z}_i^L$ and $\mathscr{Z}_i^R$, allowing for parallel processing of the hypertrapezoids corresponding to them. The hypertrapezoid specified by $\mathscr{Z}_i^C$ is dependent on $\mathscr{Z}_i^L$ and $\mathscr{Z}_i^R$, and can therefore only be processed after them.

If $\mathscr{Z}_i$ is inverted, the center of the longer base m_i does not change due to symmetry. A spacecut then results in the following subtrapezoids (see Figure 2.4).

$$\begin{aligned}
\mathscr{Z}_i^L &= (t_0, t_1 \ ; \ x_{0,i}, m_i - 2\sigma_i, -\sigma_i, \sigma_i) \\
\mathscr{Z}_i^C &= (t_0, t_1 \ ; \ m_i - 2\sigma_i, m_i + 2\sigma_i, \sigma_i, -\sigma_i) \\
\mathscr{Z}_i^R &= (t_0, t_1 \ ; \ m_i + 2\sigma_i, x_{1,i}, -\sigma_i, \sigma_i)
\end{aligned}$$

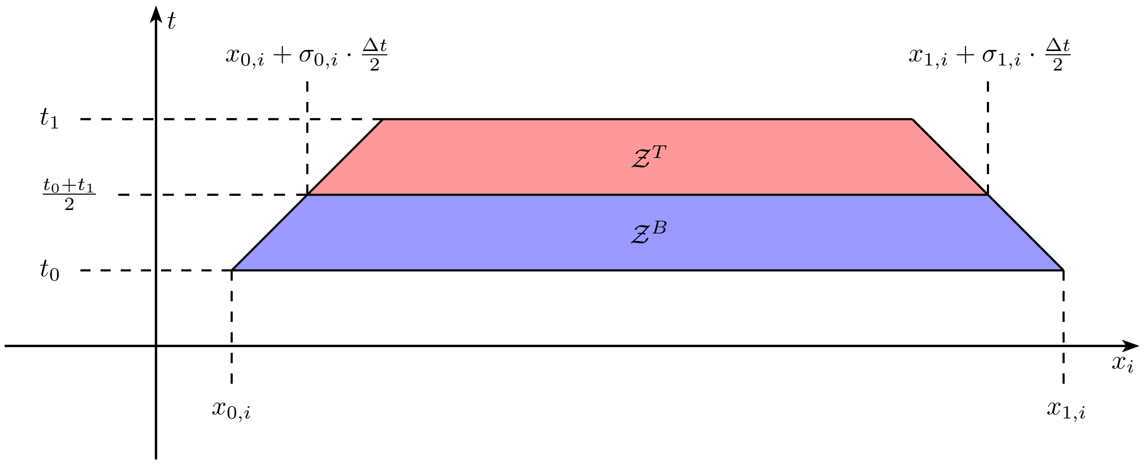


t
$x_{0,i}$
m_i
$x_{1,i}$
t_1
t_0
$\mathcal{Z}_i^L$
$\mathcal{Z}_i^C$
$\mathcal{Z}_i^R$
x_i
$2\sigma_i \cdot \Delta t$
$m_i - \sigma_{0,i} \cdot \Delta t$
$m_i + \sigma_{1,i} \cdot \Delta t$

Figure 2.4: Decomposing an inverted projection trapezoid $\mathscr{Z}_i$ with a spacecut.

In that case, $\mathscr{Z}_i^L$ and $\mathscr{Z}_i^R$ are dependent on $\mathscr{Z}_i^C$ but still independent of each other. Hence, $\mathscr{Z}_i^L$ and $\mathscr{Z}_i^R$ can still be processed in parallel, but this time after $\mathscr{Z}_i^C$ has been processed.

A spacecut is legal only if the dissection lines intersect the shorter side of the projection trapezoid. This leads to the spacecut legality constraints $\nabla x_i \geq 2\sigma_i$ for upright $\mathscr{Z}_i$ and $\Delta x_i \geq 2\sigma_i$ for inverted $\mathscr{Z}_i$. If the spacecut legality constraints are violated in all dimensions x_i, but $\Delta t \geq 2$, a timecut can still be applied.



t
$x_{0,i} + \sigma_{0,i} \cdot \frac{\Delta t}{2}$
$x_{1,i} + \sigma_{1,i} \cdot \frac{\Delta t}{2}$
t_1
$\frac{t_0+t_1}{2}$
t_0
$\mathcal{Z}^T$
$\mathcal{Z}^B$
x_i
$x_{0,i}$
$x_{1,i}$

Figure 2.5: Decomposing an inverted projection trapezoid $\mathscr{Z}_i$ with a spacecut.

A ***timecut*** is performed by splitting a hypertrapezoid $\mathscr{Z}$ in halves along dimension t, resulting in the following two subtrapezoids (see Figure 2.5).

$$\mathscr{Z}^B = (t_0, t_0 + \frac{\Delta t}{2} \ ; \ x_{0,i}, x_{1,i}, \sigma_{0,i}, \sigma_{1,i} \ ; \ \ldots)$$
$$\mathscr{Z}^T = (t_0 + \frac{\Delta t}{2}, t_1 \ ; \ x_{0,i} + \sigma_{0,i} \cdot \frac{\Delta t}{2}, x_{1,i} + \sigma_{1,i} \cdot \frac{\Delta t}{2}, \sigma_{0,i}, \sigma_{1,i} \ ; \ \ldots)$$

No parallel processing is possible in conjunction with a timecut, as $\mathscr{Z}^T$ is dependent on $\mathscr{Z}^B$.

Pochoir generally tries to apply ***hyperspace cuts***, which are performed by simultaneously applying spacecuts in as many spatial dimensions as possible. A hyperspace cut in $k \geq 1$ dimensions results in 3^k subtrapezoids, which can be processed in $k+1$ sequential steps [4]. In order to keep track of dependences between tiles, Pochoir assigns one of $k+1$ dependence levels to each tile, identifying the sequential step in which it can be processed.

Pochoir's Cache-Oblivious Parallel Algorithm

The algorithm at the heart of Pochoir is a parallel divide-and-conquer algorithm. It employs trapezoidal decompositions as introduced above to recursively decompose the iteration domain of a stencil computation into smaller tiles, before evaluating the kernel function on all points within these tiles.

Given a well-defined hypertrapezoid $\mathscr{Z}$, this algorithm proceeds as is shown in Figure 2.6. First at all, it is tried to apply a spacecut to $\mathscr{Z}$ in as many spatial dimensions as possible (lines 3 to 8). If $\mathscr{Z}$ was decomposed in $k \geq 1$ dimensions (line 9), dependence levels $0, \ldots k$ are assigned to the resulting 3^k subtrapezoids (line 10). These indicate the sequential step in which each subtrapezoid can be processed, as was explained above. Subsequently, the algorithm iterates over the $k+1$ dependence levels and recursively processes all subtrapezoids within one dependence level in parallel (lines 11 to 15).

If a hyperspace cut could not be applied to $\mathscr{Z}$ due to the tiling legality constraints being violated, but $\Delta t \geq 2$, the algorithm applies a timecut instead (lines 16 to 19). $\mathscr{Z}$ is split in half along the time dimension, resulting in the lower subtrapezoid $\mathscr{Z}^B$, which is processed first, and the upper subtrapezoid $\mathscr{Z}^T$, which is processed second. If neither a hyperspace cut nor a timecut are applicable, the base case of the recursion evaluates the kernel function for all points in $\mathscr{Z}$ (lines 20 to 23). In order to reduce overhead due to excessive recursion, in practice the base case is coarsened by choosing cut thresholds on the size of a hypertrapezoid.

Compile-Time Optimizations

The Pochoir compiler is used to transform code written in the Pochoir specification language into optimized C++ code. As explained in Section 2.1.1, code written in the Pochoir specification language can be compiled without the use of the Pochoir compiler, which results in unoptimized debug code that employs nested loops to perform a stencil computation. Thus, the first optimization performed by the compiler is to redirect calls to a Pochoir object's `Run` method to the parallel algorithm described above. Various other optimizations are performed, most of which target the user-specified kernel function.

As Pochoir allows the user to specify arbitrary boundary functions, ghost cells can not be used to speed up boundary handling. Therefore, it is necessary to check for boundary accesses when traversing the grid initially, which can easily dominate the runtime of a stencil computation [4]. The Pochoir compiler resolves this problem by generating two clones of the kernel function, a boundary clone and an interior clone. The boundary clone is used to process hypertrapezoids that contain at least one boundary access, while the interior clone is used when processing hypertrapezoids that do not contain any boundary accesses, thus eliminating the need for boundary access checks in that clone. The type of a hypertrapezoid is determined at runtime.

```
Input: A well-defined, $d+1$-dimensional hypertrapezoid $\mathscr{Z}$
procedure Trap($\mathscr{Z}$)
    k ← 0;
    for $i = 0$ to $d$ do
        if min($\Delta x_i$, $\nabla x_i$) $\geq 2 \cdot \sigma_i$ then
            Trisect $\mathscr{Z}_i$ with a hyperspace cut;
            k ← k + 1
        end
    end
    if k $> 0$ then
        Assign dependence levels $0, \ldots, k$ to subtrapezoids;
        for $i = 0$ to $d$ do
            parallel for each subtrapezoid $\mathscr{S}$ with dependence level $i$ do
                Trap($\mathscr{S}$);
            end
        end
    else if $\Delta t \geq 2$ then
        Apply timecut to $\mathscr{Z}$;
        Trap($\mathscr{Z}^B$) ;
        Trap($\mathscr{Z}^T$) ;
    else
        for $t = t_0$ to $t_1$ do
            Evaluate kernel function for all points at time $t$ within $\mathscr{Z}$;
        end
    end
end
```

Figure 2.6: Processing a well-defined hypertrapezoid $\mathscr{Z}$ using Pochoir's cache-oblivious parallel algorithm Trap

Whenever possible, the Pochoir compiler transforms accesses to Pochoir array objects in the interior clone into C-style pointer manipulations. The resulting code is usually of a type that allows Pochoir to rely on the autovectorization features of the underlying C++ compiler, instead of performing vectorization on its own. Because the Pochoir compiler does not contain a complete C++ frontend, it may, however, not be able to parse the user-specified kernel function. In that case, macro tricks are employed to eliminate boundary checks in the interior clone. Apart from that, no further transformations are performed on either the interior or boundary clone.

Finally, Pochoir reduces recursion overhead by coarsening the base case of the parallel algorithm, i.e., stopping the recursion when the size of a hypertrapezoid falls below a certain threshold. Pochoir employs some heuristics to choose reasonable threshold values, however it is possible to autotune these values according to the developers of Pochoir [4].

2.1.3 Known Limitations

Pochoir is able to generate high-performance code for a wide range of stencil computations. However, the Pochoir specification language puts limitations on the range of stencil computations that can be represented with its help. There are also some limitations on the performance Pochoir can achieve for certain types of stencil codes.

The Pochoir specification language allows the user to specify exactly one kernel function per stencil computation, which operates on only one grid. This permits multi-statement kernel functions, as they are used for example in implementations of Lattice Boltzmann Methods [24]. However, it does not allow for complex stencil codes that operate on multiple grids, possibly using multiple loops, which is for example required in the Finite Difference Time Domain method [1, 25].

Under certain circumstances, Pochoir may achieve only little speedup in comparison to a loop-based implementation. The most prominent reasons for this behavior are small spatial grids and many branch conditionals or a high rate of memory accesses to floating point operations in the kernel function [4]. In the first case, ample parallelism can not be achieved due to only very few spacecuts being possible. In the second case, evaluation of the branch conditionals and memory bandwidth constrain the runtime both for Pochoir and a loop-based implementation, so Pochoir's better cache efficiency and parallelism do not have that much of an impact on performance.

2.2 The PLUTO Compiler

PLUTO is an automatic parallelization framework which transforms C code from source to source using the polyhedral model. It focuses mainly on finding affine transformations that allow for efficient tiling and loop fusion, improving both data locality and coarse-grained parallelism [5, 13]. The user can employ compiler directives to identify loops that are amenable to parallelization, which are then automatically transformed to parallel OpenMP code by PLUTO. As a result, PLUTO is not restricted to stencil computations, but can handle any type of imperfectly nested affine loops.

2.2.1 Programming Interface

Writing source code for PLUTO is fairly straightforward. In fact, it barely differs from writing regular C code. The only difference to vanilla C are the `#pragma scop` and `#pragma endscop` compiler directives that are used to identify static control parts in a user program (see Section 1.2.1). An example of a PLUTO program can be seen in Figure 2.7.

```
 1  void heat() {
 2    int A[T][N];
 3
 4    // ... initialize A ...
 5
 6    int t, i;
 7
 8    #pragma scop
 9    for (t = 1; t < T; t++) {
10      for (i = 1; i < N-1; i++) {
11        A[t+1][i] = 0.125*(A[t][i-1] - 2*A[t][i] + A[t][i+1]);
12      }
13    }
14    #pragma endscop
15  }
```

Figure 2.7: Implementation of the one-dimensional heat equation stencil introduced in Figure 1.1 using PLUTO. Lines 8 and 14 identify the enclosed loops as a static control part, the only difference to vanilla C. Note that ghost cells are used to simulate boundary conditions.

For compilation, PLUTO is invoked on the original source code, producing parallel OpenMP code as output. Various command line options can be used in order to customize the behavior of PLUTO, by specifying tiling strategies, for example.

2.2.2 Optimization Approach

PLUTO is based on a powerful scheduling algorithm that operates on a polyhedral representation of the input program [13, 19]. This algorithm searches for affine transformations that optimize the input program with regard to parallelism and data locality. For coherence considerations, we refrain from studying the algorithm in its entirety, as it is able to handle a much wider range of input programs than we focus on in this thesis. Instead, only the special case of stencil computations as defined in Section 1.1 is elaborated on. In particular, we assume that the input program consists of exactly one perfect loop nest with constant bounds, containing exactly one statement with only uniform dependences. Such loop nests are static control parts and can always be represented in the polyhedral model.

Finding the First Tiling Hyperplane

Finding a suitable affine transformation for a given loop nest poses a number of challenges, even under simplifying assumptions. Consider, for example, the loop nest used as an example in Section 1.2.1. If the outer loop in Figure 1.3 were to be the time dimension loop of a stencil computation, many affine transformations would be illegal, like for example exchanging the inner and outer loops. Furthermore, legal affine transformations are not guaranteed to have a beneficial effect on parallelism and data locality. The scheduling algorithm used in the PLUTO compiler addresses these problems by imposing appropriate constraints on tiling hyperplanes, and by introducing a cost function that measures the quality of a tiling hyperplane [13]. In this section, we can safely assume that the offset component of tiling hyperplanes is zero, i.e., we only examine linear transformations of a statement [5].

Let $S = \{\vec{v_0}, \dots, \vec{v_n}\}$ be the shape of a d-dimensional stencil, and let ϕ_S be a one-dimensional linear transformation for that stencil. For ϕ_S to be a legal tiling hyperplane, the following constraint should be met for each $\vec{v_i} \in S$

$$-\phi_S(\vec{v_i}) \geq 0$$

The shape S of a stencil represents all dependences of that stencil, i.e., for any given iteration $\vec{x}$, the set $\{\vec{x} - \vec{v_i} \mid v_i \in S\}$ contains all iterations that depend on $\vec{x}$. Conversely, the set $\{\vec{x} + \vec{v_i} \mid v_i \in S\}$ contains all iterations that $\vec{x}$ depends on. With the above constraint, the inequality $\phi_S(\vec{x} - \vec{v_i}) \geq \phi_S(\vec{x}) \geq \phi_S(\vec{x} + \vec{v_i})$ holds true for each iteration within these sets. As ϕ_S determines at which time an iteration is executed in the transformed iteration space, $\phi_S(\vec{x} - \vec{v_i}) \geq \phi_S(\vec{x})$ implies that any iteration that is dependent on $\vec{x}$ is executed after or at the same time as $\vec{x}$. Equally, $\phi_S(\vec{x}) \geq \phi_S(\vec{x} + \vec{v_i})$ ensures that any iteration that $\vec{x}$ depends on is executed before or at the same time as $\vec{x}$.

A multidimensional linear transformation ϕ with full column rank necessarily transforms different source iterations to different target iterations. Thus, if all tiling hyperplanes of ϕ fulfill the tiling legality constraint and $\succ$ is the lexicographic order, clearly $\phi(\vec{x} - \vec{v_i}) \succ \phi(\vec{x}) \succ \phi(\vec{x} + \vec{v_i})$ holds true for any iteration $\vec{x}$ and $\vec{v_i} \in S$. Therefore, any rectangular tiling of the transformed iteration domain is valid if the tiling legality constraint is met, as all dependences cross the tiling hyperplanes in the same direction.

There are many tiling hyperplanes that fulfill the tiling legality constraint. In order to select the desired hyperplanes, a cost function is introduced. Let S and ϕ_S be defined as above, and $\vec{x}$ be an iteration. For each $\vec{v_i} \in S$, the affine form δ_{v_i} with

$$\delta_{v_i}(\vec{x}) = \phi_S(\vec{x}) - \phi_S(\vec{x} + \vec{v_i}) \geq 0$$

is used as cost function. If iterations along ϕ_S are executed sequentially, this function measures the reuse distance, i.e., how many previous iterations along ϕ_S influence $\vec{x}$. If, on the other hand, iterations along ϕ_S are executed in parallel, δ_{v_i} provides information about the degree of communication induced by dependences outside a single iteration. Minimizing this function to a constant value or even zero by choosing a suitable tiling hyperplane results in minimal or no communication along that hyperplane. Due to only uniform dependences being considered, the cost function simplifies to

$$\delta_{v_i}(\vec{x}) = -\phi_S(\vec{v_i}) \geq 0$$

which is independent of the iteration vector. Accordingly, the cost function can be bounded by a constant w for any choice of ϕ_S and v_i, which is equivalent to the constraint

$$w - \delta_{v_i}(\vec{x}) \geq 0$$

The tiling legality constraints and the cost function bounding constraints are now gathered in one constraint set, and w is appended as the outermost dimension to all tiling hyperplanes. The lexicographic minimum among the vectors $(w, \ldots, c_i, \ldots)$ that meets all constraints in that set identifies a legal tiling hyperplane for which the cost function is minimal. This lexicographic minimum can be computed, for example, by using the simplex algorithm, which is implemented in most polyhedral libraries. Actually, the lexicographically minimal solution satisfying all constraints is the zero vector, however it does not identify a feasible tiling hyperplane and is therefore excluded.

Determining further tiling hyperplanes

In order to obtain a linear transformation with full column rank, at least as many linearly independent tiling hyperplanes as the dimensionality of the loop nest need to be found. Hence, the algorithm augments the constraint set with additional constraints that ensure linear independence with all previous solutions.

Let T be the transformation matrix containing all tiling hyperplanes that were already found in the first rows and zero in the remaining rows. The rows of this matrix span a space that all linearly dependent solutions lie within. Accordingly, all linearly independent solutions lie in the subspace orthogonal to that space, which is spanned by the rows of the right kernel $\ker(T)$ of T. It would now be sufficient to add a constraint so that the next solution has a single non-zero component in that orthogonal subspace, i.e. $\ker(T)_i \circ \phi_S^T \neq 0$ for any row $\ker(T)_i$ of $\ker(T)$. However, if every possible combination is tried, this leads to a non-convex space and possibly combinatorial explosion [13]. Therefore, additional constraints are added to ensure that only solutions with a non-negative component for every row of $\ker(T)$ are considered. In summary, the following constraints are added to the constraint set each time another solution has been found

$$\forall i : \ker(T)_i \circ \phi_S^T \geq 0$$
$$\sum_i \ker(T)_i \circ \phi_S^T \geq 1$$

where the latter constraint is equivalent to $\ker(T)_i \circ \phi_S^T \neq 0$ for any row $\ker(T)_i$ of $\ker(T)$. Once T has full column rank, the algorithm stops.

Enabling Concurrent Start

The PLUTO algorithm as it was described so far calculates a linear transformation for a stencil that allows for rectangular tiling of the transformed iteration domain. Tiling hyperplanes are chosen in such way, that both communication and synchronization are optimized. However, as of now, this linear transformation commonly induces pipelined startup due to the choice of tiling hyperplanes. Therefore, the algorithm was extended to search for tiling hyperplanes that enable full or partial concurrent start along one face of the iteration domain [5].

Concurrent start along a face $\vec{f}$ of the iteration domain is possible iff $\vec{f}$ is a strict conic combination of the tiling hyperplanes $\vec{h_i}$, i.e.

$$k \cdot \vec{f} = \lambda_1 \cdot \vec{h_1} + \cdots + \lambda_n \cdot \vec{h_n}$$

where $k, \lambda_i \in \mathbb{Z}^+$ are strictly positive. This constraint is incorporated into the iterative scheme of the PLUTO algorithm. Let $n \in \mathbb{N}$ be the number of tiling hyperplanes to be found. For the first $n-1$ steps, constraints are added so that tiling hyperplanes are linearly independent of $\vec{f}$ in addition to previously found solutions. For the last hyperplane $\vec{h_n}$, the above constraint is added, ensuring that $\vec{f}$ is a strict conic combination of the tiling hyperplanes. The coefficients λ_i and k can be eliminated using the affine form of Farkas Lemma [13, 26]. An obvious choice for $\vec{f}$ would be, for example, the time dimension identified by $\vec{f} = (1, 0, \ldots, 0)$.

2.2.3 Known Limitations

It was already mentioned above that the PLUTO compiler is able to handle a much wider range of programs than just stencil computations. Nevertheless, there are some limitations on programs that can be optimized using PLUTO.

The most prominent limitation is imposed on boundary conditions in loop nests. As loop nests need to be static control parts in order to be amenable to optimization, no dedicated handling of boundary conditions is possible. This leaves the use of ghost cells as the only viable method to handle boundary accesses in loop nests. It is in theory possible to employ macro or template tricks to perform boundary checks on every array access, and perform arbitrary boundary calculations in response. However, executing boundary checks on every array access is very likely to induce so great a slowdown, that no performance benefit can be achieved.

Likewise, it is not possible to contain arbitrary C code in the loop nests, as it would be possible with Pochoir, for example. This rules out certain stencil computations like, for example, implementations of Lattice Boltzmann Methods [24], which incorporate rather complex kernel functions. A common optimization for stencil computations is to maintain only as small a number of copies of the space grid as is necessary to correctly evaluate the kernel function. This reduces the memory footprint of a stencil computation greatly. However, the version of PLUTO we used in this thesis neither performs this optimization on itself, nor is it able to parse C code with this optimization already performed. Again, this leaves only macro or template tricks as a possible solution, leading to the same problems as above.

2.3 The SDSL Compiler

The Stencil Domain Specific Language (SDSL) is a domain-specific language for specifying stencil computations. It enables the user to express a high-level specification of complex stencil computations that allows for high-performance code generation on multiple platforms [8]. Currently, the SDSL compiler can translate SDSL code embedded in C or C++ into CPU or GPU code using various techniques. Among the possible output formats are affine C code that is meant to be processed further by polyhedral optimization tools such as, for example, PLUTO, or C code that is optimized for short vector SIMD architectures [1].

2.3.1 Programming Interface

The SDSL constitutes a very powerful tool in specifying stencil computations. As such, the range of language features is much larger than we can detail reasonably in this section. Therefore, we will only cover the specification of the one-dimensional heat equation stencil using the SDSL, and refer to the SDSL user guide [8] for a complete language reference. The SDSL specification of said heat equation stencil can be found in Figure 2.8.

```
void heat() {
  int A[N];

  // ... initialize A ...

  const double ONEEIGHTH = 0.125;
  const double TWO = 2.0;

  #pragma sdsl begin
  int N;

  double ONEEIGHTH;
  double TWO;

  grid g[N];

  double griddata A on g at 0,1;

  pointfunction compute(x) {
    [1]x[0] = ONEEIGHTH * ([0]x[1] - TWO * [0]x[0] + [0]x[-1]);
  }

  iterate 50000 {
    stencil main {
      [1 : N - 2] : compute(A);
    }
  }
  #pragma sdsl end
}
```

Figure 2.8: Embedded SDSL specification of the one-dimensional heat equation stencil introduced in Figure 1.1.

The actual SDSL specification is found in lines 10–27. Before that, the input and output array has to be specified, as well as any parameters that are passed to the SDSL part. Lines 10–13 define three parameters that are used within the SDSL specification. Parameters are bound to values at the time of program execution. Line 15 defines the grid geometry used for the stencil computation, namely a one-dimensional grid of size `N`. Continuing, in line 17 a concrete instance of that grid is specified with the identifier `A`, which exists at time offsets `0` and `1`. This grid data is bound to the array `A`. Lines 19–21 define a point function with the identifier `compute`. A point function takes an arbitrary number of grid data as arguments and performs an arbitrary number of computations on that grid data. Every time a point function is invoked, it is at an arbitrary point within the grids. All grid data references are expressed using offsets from that points. In the example, `[1]x[0]` identifies the value of grid data `x` at time offset `1` and spatial offset `0`. In the context of this thesis, point functions correspond to the kernel function of a stencil computation.

The main stencil computation is specified in lines 23–27. The `iterate` construct specifies the number of time steps over which the stencil computation should be performed. Note that the number of time steps has to be hard-coded. Lines 24–26 define a stencil with the identifier `main` over a subset of the grid, omitting the boundary of the grid. A stencil can contain an arbitrary number of pairs of grid subsets and expressions. This allows, for example, the specification of complex boundary conditions.

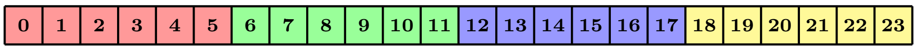
The result of the SDSL computation is copied out to the according arrays that were bound to grid data in the SDSL specification. Compilation of source code that contains embedded SDSL is done similar to Pochoir and PLUTO. The SDSL compiler acts as a preprocessor that, depending on user parameters, outputs different kinds of optimized C code. This code can then be processed further by appropriate compilers. The SDSL compiler is able to generate affine C code for use with polyhedral optimization tools, optimized C code for short vector SIMD architectures, and CUDA C code for GPUs using the OverTile backend [27].

2.3.2 Optimization Approach

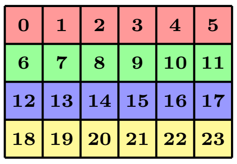
The SDSL compiler contains multiple code generation backends. In the context of this thesis, we will concentrate on code generation for short vector SIMD instruction sets, such as SSE, AVX etc. During code generation for these architectures, the compiler employs two main optimization techniques. First, a dimension-lifting-transpose (DLT) data layout transformation is performed in order to address memory alignment issues arising from arithmetic operations on physically contiguous data elements. Second, the stencil computation is tiled using an approach that is compatible with this data layout transformation [1, 20].

Data Layout Transformation

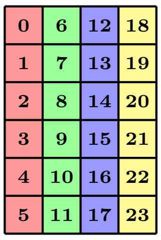
Stencil computations usually perform arithmetic operations on physically contiguous data elements. The kernel function of the one-dimensional heat equation stencil (see Figure 1.1), for instance, accesses the adjacent array elements `A[t][i - 1]`, `A[t][i]`, and `A[t][i + 1]`. When using vector loads, these physically contiguous data elements end up in different slots of the same vector register. However, efficient vector operations are possible only if these data elements are located in the same slot of different vector registers (see Figure 2.10(a)). In order to achieve this register layout, either redundant and probably unaligned vector loads, or inter- and intra-register shuffle operations are therefore required [20].



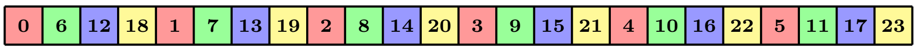
(a) Original data layout



(b) Dimension-lifted representation



(c) Dimension-lifted and transposed representation



(d) Transformed data layout

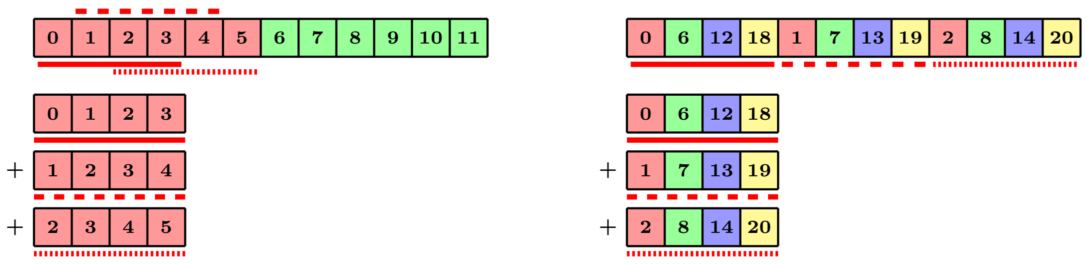
Figure 2.9: The data layout transformation employed by the SDSL compiler, applied to a one-dimensional array of length 24 for vector length 4.

The SDSL compiler overcomes this fundamental problem of stencil computations through a data layout transformation. This transformation rearranges formerly adjacent data elements, which are potentially accessed by arithmetic operations, so that they can be loaded into the same vector register slot. We detail the data layout transformation using an example borrowed from [20], assuming a vector register size of 4. Let A and B be one-dimensional arrays containing 24 double precision data elements each, with the following computation to perform

```
for (int i = 1; i < 23; i++)
    B[i] = A[i - 1] + A[i] + A[i + 1];
```

Note that this computation has the same data access layout as the one-dimensional heat equation stencil. As displayed in Figure 2.9(b), the data elements contained in A can also be viewed as a two-dimensional 4×6 matrix, which has the same data layout as A with row-major ordering. Figure 2.9(c) shows the transpose of this matrix, i.e., the dimension-lifted transpose of A, which corresponds to the one-dimensional array depicted in Figure 2.9(d). We shall refer to this array as D in the following.

There are several observations to be made now. First of all, data elements that were contiguous in A are separated in D. For instance, A[0] and A[1] are both located in column zero but different rows of the transposed matrix, mapping to the memory addresses D[0] and `D[4]`. Therefore, `A[0]` and `A[1]` are now both loaded into the first slot of a vector register, and equally `A[2]` is. This allows for the additions `(A[0] + A[1] + A[2])`, `(A[6] + A[7] + A[8])`, `(A[12] + A[13] + A[14])` and `(A[18] + A[19] + A[20])`, to be performed using three aligned vector loads and subsequently two vector additions, without any additional data movements (see Figure 2.10(b)). Furthermore, some of these vector elements can be reused when performing the next operation.



(a) Vector operations on the original data layout

(b) Vector operations on the transformed data layout

Figure 2.10: Vector operations on the original and transformed data layout. Without a data layout transformation, the data elements in vector registers overlap, as is shown in (a). No overlap is induced in (b), where a data layout transformation has been applied.

However, this data layout transformation introduces a new type of boundary case for operands that are distributed over multiple columns of the transposed matrix. The addition `(A[5] + A[6] + A[7])`, for instance, can not be performed using vector operations initially, since these elements are located in different vector slots. While it is possible to handle these boundary cases using scalar operations, a more efficient way is to employ masked vector operations. We refer to [20] for a detailed description of the latter strategy.

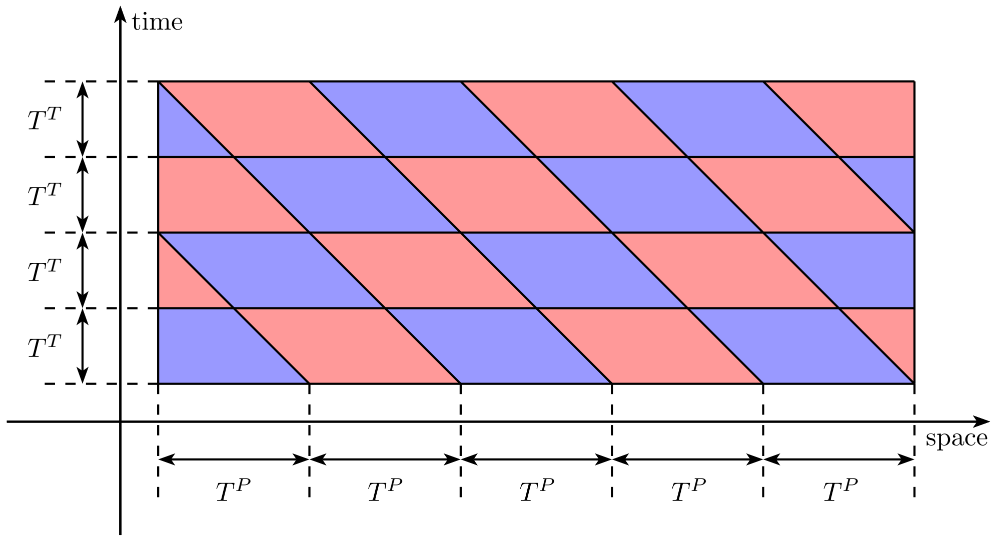


Figure 2.11: Traditional rectangular tiling as performed by the SDSL compiler.

Hybrid and Nested Split Tiling

Standard rectangular tiling is generally infeasible in conjunction with the data layout transformation elaborated on in the previous section. With standard rectangular tiling, tiles in the original iteration space are parallelogram-shaped, as displayed in Figure 2.11. As inter-tile dependences occur along both the time and spatial dimensions, tiles can not be processed concurrently along the spatial dimensions. However, this is a requirement when using the DLT data layout transformation, as spatially separated data elements are operated upon in parallel [1].

Instead of rectangular tiling, the SDSL compiler employs nested and hybrid split tiling. ***Nested split tiling*** is remotely similar to Pochoir's tiling approach, because it is based on trapezoidal tiles. Tiles are not recursively decomposed, but of a predetermined size, though. To begin with, consider a one-dimensional stencil with slope σ, and let T_T be the height of trapezoids, i.e. their size in the time dimension. Furthermore, let T_U and T_I be the base width of upright and inverted trapezoids, respectively. The iteration domain is now sliced along the time dimension to produce bands of height T_T, which have to be processed sequentially. Each of these bands is then decomposed into trapezoids with slopes $\pm\sigma$ that are alternately upright and inverted, where upright trapezoids have width T_U and inverted trapezoids have width T_I. All upright trapezoids within one band can be processed concurrently, as well as all inverted trapezoids within one band (see Figure 2.12).

Nested split tiling of a multi-dimensional stencil computation is performed by recursively split tiling each spatial dimension. This allows for parallelization of all spatial loop nests in a stencil computation. Additionally, the DLT data layout transformation can now be applied to the innermost loop dimension, because tiles can be processed concurrently along this dimension.

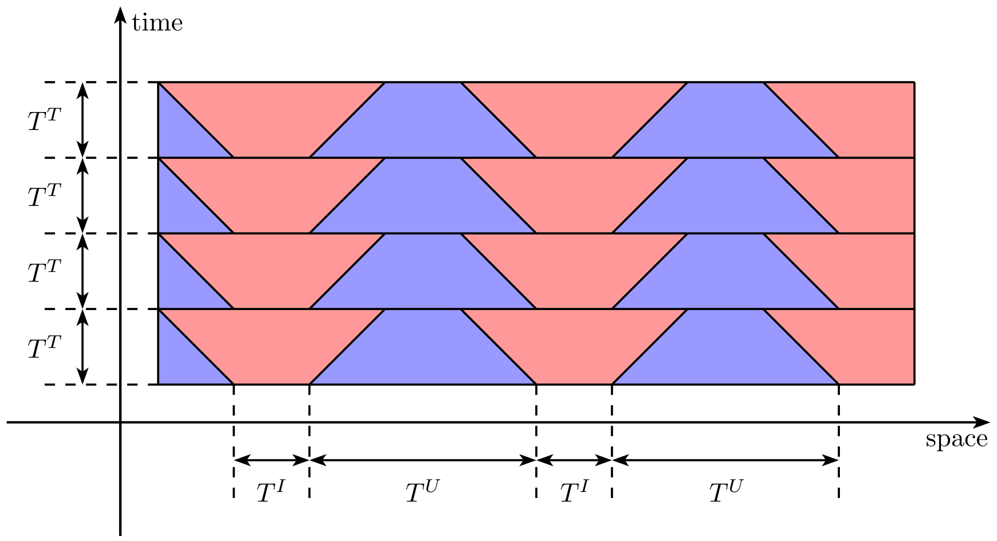


Figure 2.12: Split tiling as performed by the SDSL compiler.

There is, however, a limitation on nested split tiling especially for stencil computations of higher dimensionality. In order to retain their trapezoidal shape, tiles must meet the constraint $T_T \geq 2 \cdot \sigma \cdot T_U$ in each spatial dimension, similar to the tiling legality constraint for Pochoir tiles. This lower bound causes even small tile sizes to overflow the L1 and L2 cache very quickly on most modern CPUs. The SDSL compiler addresses the tile size constraints incurred by nested split tiling with a hybrid form of split tiling and standard rectangular tiling.

Hybrid split tiling is applicable to stencil computations with a dimensionality of 2 or higher. In hybrid split tiling, traditional rectangular tiling with size T^P is performed on the outermost spatial loop, while the remaining spatial loops are split-tiled recursively. Rectangular tiling does not induce any constraints on tile size, so tiles can be scaled down in the outermost dimension to counteract the larger tile size requirements in the split-tiled dimensions. Therefore, the memory footprint of tiles can be significantly reduced, while the DLT data layout transformation can still be applied to the innermost spatial loop.

2.3.3 Known Limitations

Most limitations of the SDSL compiler arise from the stencil domain specific language specification. For instance, only one grid may be defined per SDSL program. Additionally, grid data can only be defined at time offsets 0 and 1, excluding stencils with a depth greater than 1. Furthermore, it has already been observed in Section 2.3.1 that the number of time iterations of a stencil computation has to be hard-coded in the SDSL specification. It is not possible to pass this number as a program parameter.

If the split tiling backend of the SDSL compiler is used, the size of the innermost spatial dimension is required to be a multiple of the vector register size v and the sum of upright and inverted tile sizes, i.e., $d_{innermost} \equiv 0 \mod v \cdot (T_U + T_I)$. However, this constraint can be fulfilled easily by padding the innermost spatial dimension.

Chapter 3

Execution Order Optimization

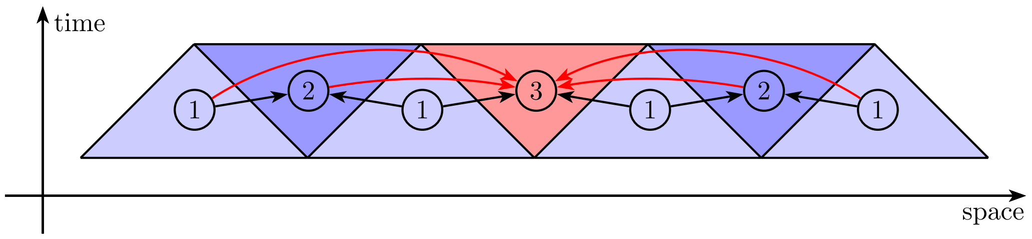
We have found previously that the trapezoidal tiling algorithm of Pochoir could be optimized further for one-dimensional stencils [9]. In the following, we extend our optimization approach to support multi-dimensional stencil computations, and apply it to the tiling algorithms of the Pochoir, PLUTO and SDSL compilers. We take a closer look on some problems experienced by these algorithms in Section 3.1, which encouraged our optimization approach. Section 3.2 provides a high-level overview of the approach, and in Section 3.3, we cover the details of our realization thereof as a C++ template library.

3.1 Motivation

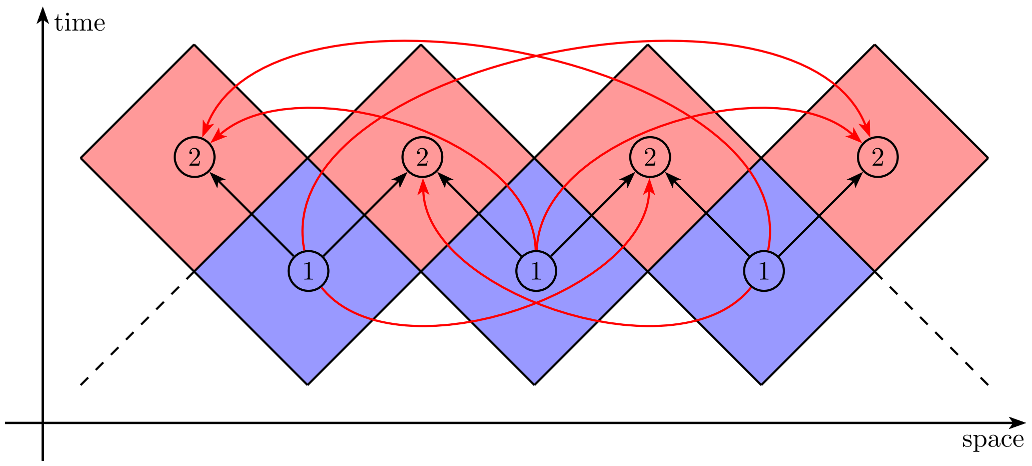
During closer investigation of the tiling algorithms presented in the previous section, we have identified the elimination of redundant dependences as a main starting point for possible optimizations.

Dependences between tiles are handled implicitly by these algorithms, i.e., an appropriate processing order is chosen that does not violate any inter-tile dependences. The PLUTO and SDSL compilers generate tiled loops that imply a valid processing order of tiles, while the Pochoir compiler employs barrier synchronization during recursive decomposition of the iteration domain in order to ensure that all dependences are met. In both cases, this introduces a number of inter-tile dependences that do not correspond to actual flow dependences (see Figures 3.1(a)–3.1(c)).

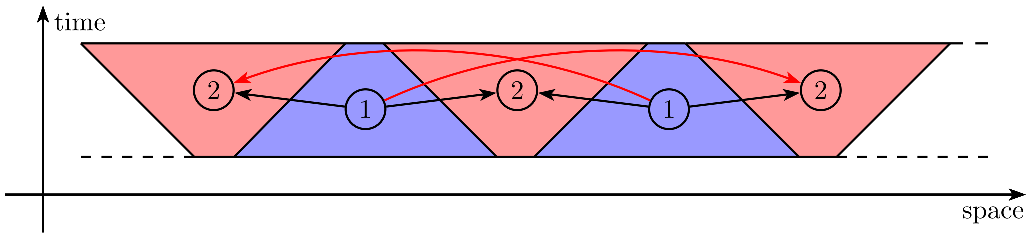
Initially, this behavior is intended and usually critical to the functionality of the respective tiling algorithm. The DLT data layout transformation employed by the SDSL compiler is, for instance, applicable only, if tiles within one time band are processed in the same parallel step along the spatial dimension. Similarly, the tile processing order contributes greatly to the cache efficiency of the Pochoir compiler. Nevertheless, these redundant dependences may still induce a notable load imbalance at run time and thus severely impact scalability on modern multi-core architectures. This was, for example, observed by the developers of the PLUTO compiler [5, 28], and by us during the implementation of our preliminary optimization approach for Pochoir's algorithm [9]. Additionally, and more severely so, inter-tile data reuse is hindered by these dependences for diamond and split tiling. Both of these tiling algorithms process all tiles within one time band in one sequential step. If the spatial dimensions of the data grid are sufficiently large, this leads to all points at the boundary of tiles being expelled from the cache before



(a) Redundant dependences for the Pochoir stencil compiler. A spacecut yields the two large upright subtrapezoids colored in blue, which are processed in all before the red subtrapezoid. Applying a second spacecut to these trapezoids thus induces the dependences colored in red.



(b) Redundant dependences for the PLUTO compiler. Tiles are processed sequentially along the time dimension, which introduces a number of additional dependences. A wider grid implies more redundant dependences.



(c) Redundant dependences for the SDSL compiler. All upright and inverted tiles that result from split tiling a time band are processed in two sequential steps, leading to the dependences colored in red. The number of these dependences increases with the number of upright and inverted tiles within a time band.

Figure 3.1: Examples of redundant inter-tile dependences that are induced by the tile processing order which the different stencil compilers choose. The numbers in circles show the logical time at which a tile is processed. This adds the redundant dependences colored in red, in addition to the actual dependences colored in black.

any dependent tile is processed. Especially for small tile sizes, as they are required for stencil computations of higher dimensionality, a large number of redundant accesses to slow main memory is therefore induced.

3.2 Overview of Approach

We developed a two-stage algorithm that aims to minimize the amount of redundant synchronization in tiling algorithms, while still maintaining a high level of data reuse. In the first stage of our algorithm, a tile dependence graph is generated, which represents tiles and their dependences like the original tiling algorithms suggest them. We then use a polyhedral dependence analysis in order to identify and eliminate redundant inter-tile dependences in this graph. In the second stage, the actual stencil computation is performed by traversing the tile dependence graph in parallel, following a dynamic schedule. This is possible with minimal synchronization and entirely without barrier synchronization.

The approach we propose is roughly equivalent to the compiler-assisted dynamic scheduling approach suggested by Baskaran et al. [28], which enables extraction of inter-tile dependences and dynamic scheduling of tiles at runtime. However, our approach is not limited to tiling algorithms that are based on the polyhedral model, and as such is applicable to a wider range of tiling algorithms. To the best of our knowledge, this is the first work to study a dynamic scheduling approach for non-polyhedral tiling algorithms, like they are employed by the Pochoir and SDSL compilers.

In the following, we give an abstract overview of our optimization approach, omitting implementation details that are, for instance, specific to tiling algorithms. We will cover all of these in Section 3.3. We assume that the shape S and the kernel function f of a stencil have been provided as input data with suitable format to our algorithm.

3.2.1 Dependence Relation

In the polyhedral model, a ***dependence relation*** d maps source iterations to the corresponding sink iterations, i.e., for any given iteration $\vec{x}$, the set of iterations that are dependent on $\vec{x}$ is specified by $d(\vec{x})$. Using the information the shape S of a stencil exposes, the dependence relation is obviously defined by

$$d(\vec{x}) = \{\vec{x} - \vec{v} \mid \vec{v} \in S\}$$

We can now use this relation to characterize a dependence check on arbitrary tiles, represented by their set of iterations T_1 and T_2. The tile represented by T_2 is dependent on the tile represented by T_1 iff the following condition holds true

$$d(T_1) \cap T_2 \neq \emptyset$$

This dependence check can be implemented easily using a polyhedral library.

3.2.2 Generation of the Tile Dependence Graph

The ***tile dependence graph (TDG)*** is a directed acyclic graph that holds all information necessary to model the decomposition of an iteration domain of fixed size. Each node in the graph represents a single tile, and each edge $V \rightarrow W$ indicates an inter-tile dependence between the tiles represented by V and W, respectively.

Given a tiling algorithm and appropriate tiling parameters, generation of the tile dependence graph is performed as follows. The iteration domain is decomposed into tiles according to the original tiling algorithm, and tiles are inserted as nodes into the TDG while doing so. Every time the original tiling algorithm suggests a dependence between two tiles, these tiles are represented as polyhedra and a dependence check is performed using the previously calculated dependence relation. An edge between two tiles in the TDG is added only, if this check confirms the suggested dependence. The actual implementation and optimization of this generation technique is dependent on the respective tiling algorithm, and is described in detail in Section 3.3.2.

3.2.3 Traversal of the Tile Dependence Graph

After the TDG has been generated, the actual stencil computation can be performed by traversing the graph in parallel. The traversal algorithm is displayed in Figure 3.2.

```
Input: A node V of the tile dependence graph
procedure Process(V)
    Evaluate kernel function $f$ for all points in the tile represented by V;
    foreach edge V $\rightarrow$ W do
        Remove edge V $\rightarrow$ W;
        if W has no more incoming edges then
            Spawn parallel task Process(W) ;
        end
    end
end
```

Figure 3.2: The recursive algorithm used for traversal of the tile dependence graph

When processing a node V during TDG traversal, the kernel function f is evaluated for all points within the tile represented by this node first. This step accounts for most of the time that is necessary to process a node. Second, all outgoing edges $V \rightarrow W$ of the node are removed, and a new parallel task to process W is spawned if W has no more incoming edges. Synchronization is required only while checking whether W has any pending dependences and, if not, while spawning a new parallel task.

Traversal of the TDG is started by spawning parallel tasks for all nodes without any incoming edges. These nodes correspond to tiles that are dependent only on the initial conditions of the stencil computation. As the TDG is a connected, acyclic and finite graph, the traversal algorithm obviously terminates after a finite number of steps, and after processing all nodes within the TDG in a topological order.

3.3 Realization Details

The algorithm as outlined above was incorporated into a C++ template library that we developed. This library offers a concise programming interface to our algorithm, allowing for a intuitive specification of a stencil computation using standard C++. Subsequently, a tile dependence graph and optimized kernel function code can be generated based on this specification, which are then used to perform the actual stencil computation. We implemented support for serialization after most of these intermediate steps, in order to facilitate the reuse of generated data. Our library relies on the integer set library (ISL) [14] for representation and manipulation of polyhedra wherever needed.

The remainder of this section is arranged as follows. Section 3.3.1 outlines the programming interface offered to the user. Generation of the TDG is described in detail in Section 3.3.2 and code generation is elaborated upon in Section 3.3.3.

3.3.1 Programming Overview

The overall work flow when using our library is depicted in Figure 3.3. There are three main components that the user normally interacts with. The precut component is responsible for generating the TDG, and the code generator component outputs optimized kernel function code. Both of these provide input for the traversal component, which performs the actual stencil computation based on the input TDG and kernel function code.

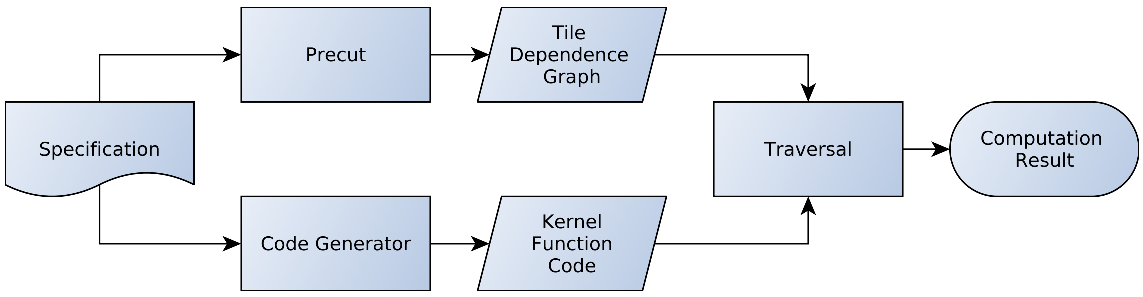


Figure 3.3: Basic work flow when using our library. The specification of a stencil computation is passed to the precut and code generation components, which results in a tile dependence graph and optimized kernel function code. These are then passed on to the traversal component, which performs the actual stencil computation and returns the results.

Continuing the running example of the previous chapters, Figure 3.4 shows the implementation of the one-dimensional heat equation stencil using our library in its most basic form. For clarity considerations, several aspects of our library have been omitted in this example, such as serialization of intermediate data or interaction with the generation process. We refer to the documentation of our header files for an in-depth description of these.

Throughout the example, the desired tiling algorithm is passed as the template parameter `STRAT` to the function, which can currently be one of either `POCHOIR`, `DIAMOND` or `SPLIT`. If applicable, a second template parameter identifying the dimensionality of the stencil computation is passed to the library. Lines 4–5 specify the kernel function and the grid size of the stencil computation. Subsequently, the tile dependence graph is generated in lines 8–11. In order to speed up the process of TDG generation, all polyhedral operations are translated to C++ code and exported as a shared library in lines 8–9, instead of directly calling ISL functions. This code expresses the respective polyhedral operations as a function on a number of integer parameters, which can be evaluated faster than the corresponding ISL function on an ISL data structure. In practice, one would only have to generate this code once per stencil computation and tiling algorithm.

In lines 14–15 the kernel function is translated to optimized C++ code and is also exported as a shared library. Again, this generation would have to be done only once

```
 1  template<Strategy STRAT>
 2  void heat(int T, int N) {
 3    // Specification
 4    AST_Node<1>& kernel = 0.125 * (Array_1D({-1, -1}) - 2.0 *
          Array_1D({-1, 0}) + Array_1D({-1, 1}));
 5    Grid_Info<1> grid_info = {T, N};
 6
 7    // TDG generation
 8    Precut_Generator<STRAT, 1>(kernel.access_shape()).
          compile_to_file("precut.tmp", true);
 9    Precut_Handle<STRAT, 1> precut_handle = Precut_Generator<STRAT
          , 1>::get_handle("precut.tmp");
10
11    Tile_Graph<STRAT, 1>* graph = Precut<STRAT, 1>(precut_handle,
          kernel.access_shape()).precut(grid_info);
12
13    // Kernel Function Code Generation
14    Kernel_Generator<STRAT, 1>(kernel).compile_to_file("kernel.tmp
          ", true);
15    Kernel_Handle<STRAT, 1> kernel_handle = Kernel_Generator<STRAT
          , 1>::get_handle("kernel.tmp");
16
17    // Traversal
18    Traversal<STRAT, 1> traversal(kernel_handle, graph);
19    traversal.run();
20
21    // Results
22    Grid<1>& result = traversal.grid();
23  }
```

Figure 3.4: Implementation of the one-dimensional heat equation stencil using our library.

per stencil computation and tiling algorithm. Finally, the kernel function code and the tile dependence graph are passed to the traversal component in line 18, and the actual stencil computation is performed in line 19. Afterwards, the results are accessed in line 22. Figure 3.5 shows how the same stencil computation can be implemented when all intermediate data has been serialized to files.

The generation of both precut and kernel function code can be customized by passing `Precut_Generator_Options` and `Kernel_Generator_Options` objects to the respective component. This allows the user to specify custom tile sizes, for example. As the generation of the tile dependence graph might take a considerably long time in some cases, a tile dependence graph can be serialized to a file for later reuse. The user can also modify the grid on which the stencil computation operates. It is possible, for instance, to specify custom boundary conditions in the form of C++ lambda expressions, and to specify initial conditions on a grid.

3.3.2 Precut Component

The precut component generates a tile dependence graph based on the specification of a stencil computation. While the internal implementation is dependent on the employed

```
template<Strategy STRAT>
void heat(int T, int N) {
  // Load TDG
  Tile_Graph<STRAT, 1>* graph = Tile_Graph<STRAT, 1>::
     deserialize("graph.tmp");

  // Load Kernel Function Code
  Kernel_Handle<STRAT, 1> kernel_handle = Kernel_Generator<STRAT
     , 1>::get_handle("kernel.tmp");

  // Traversal
  Traversal<STRAT, 1> traversal(kernel_handle, graph);
  traversal.run();

  // Results
  Grid<1>& result = traversal.grid();
}
```

Figure 3.5: Implementation of the one-dimensional heat equation stencil using our library. In this code fragment, we assume that all intermediate data has been serialized to the respective files prior to calling the `heat` function.

tiling algorithm, there is a common interface to the precut component for all tiling algorithms. The user has to provide the shape S of a stencil and the desired grid size to the component, which then returns the root node of the generated tile dependence graph. Each node in the tile dependence graph is annotated with an algorithm-dependent tile representation and an indicator whether the respective tile contains boundary accesses. Since only the shape of a stencil is necessary to build the tile dependence graph, the latter can be reused for multiple stencils with the same shape.

During the process of generation, several properties of tiles are checked by representing tiles as polyhedra and then performing some operations on these. In particular, the dependence check outlined in Section 3.2.1 is one such operation. Additionally, we employ a polyhedral tile representation to check whether a tile contains any points within the actual grid or any boundary accesses. In order to improve performance, we use polyhedral code generation to generate C++ functions that perform these operations, rather than directly calling library functions of the ISL. We transform the respective property check to an equivalent check that tests for emptiness of a polyhedron. Consider, for example, the dependence check outlined in Section 3.2.1, which is phrased as an emptiness test. Subsequently, using the polyhedral code generation capabilities of the ISL, a C++ expression is generated which performs the emptiness test in dependence of some input parameters. These input parameters specify the polyhedra that should be checked. Finally, the generated C++ code is compiled to a shared object file, which contains the appropriate functions. In the following, we refer to these functions as `dependent(V, W)`, `boundary(V)` and `domain(V)`, where `dependent(V, W)` returns whether tile `W` is dependent on tile `V`, `boundary(V)` returns whether tile `V` contains any boundary accesses, and `domain(V)` returns whether tile `V` contains any points within the actual grid.

Trapezoidal Tiling

When using Pochoir's trapezoidal tiling scheme, the tile dependence graph is built recursively. At the core of our generation algorithm is an expansion function, which employs spacecuts and timecuts as introduced in Section 2.1.2 in order to replace one node in the tile dependence graph with several nodes representing subtrapezoids. The pseudocode of this function is displayed in Figure 3.6. The generation algorithm starts off with one trapezoid which contains the entire grid that was passed to the precut component. This trapezoid is then decomposed recursively using the expansion function, until the size of all trapezoids falls below a predefined threshold.

```
Input: A node $V$ of the tile dependence graph
1 procedure Expand($V$)
2     if not Domain($V$) then
3         Delete $V$ from the graph ;
4     else if Spacecut applicable and required then
5         Expand_Spacecut($V$) ;
6     else if Timecut applicable and required then
7         Expand_Timecut($V$) ;
8     end
9 end
```

Figure 3.6: The expansion function that is used to recursively decompose nodes in the tile dependence graph when following Pochoir's trapezoidal tiling scheme. If a spacecut is used for decomposition, the `Expand_Spacecut` function displayed in Figure 3.7 is called. If, on the other hand, a timecut is employed for decomposition, the `Expand_Timecut` function displayed in Figure 3.8 is invoked.

Let V be a node of the tile dependence graph representing a trapezoid. In the following, we will identify a trapezoid with the node by which it is represented unless otherwise stated. If V does not contain any points within the actual grid of the stencil computation, the node itself and all incoming and outgoing edges are deleted from the tile dependence graph (lines 2–3). This may occur since the algorithm starts with a trapezoid that is larger than the actual grid. Otherwise, if a spacecut is applicable - see Section 2.1.2 - and required in at least one spatial dimension of V, the node is decomposed using a spacecut (lines 4–5). A spacecut is required in a spatial dimension if the size of V in that dimension exceeds a predefined threshold value. Different threshold values are maintained for tiles that do or do not contain boundary accesses, so as to enable more coarse tiling in the interior of the grid. If a spacecut is either not applicable or not required, it is then checked whether a timecut is applicable and required, and if so, the node is decomposed using a timecut (lines 6–7). A timecut is required when the height of V is larger than a predefined threshold value, which again can be chosen differently for interior and boundary tiles.

Decomposition of a node V with a spacecut is done as shown in Figure 3.7. We keep track of the subtrapezoids we obtain after applying spacecuts in the first i dimensions in a set Q_i, because all of these trapezoids have to be decomposed by any further spacecut. Initially, the set Q_0 contains only the node V (line 3). It is then iterated over all spatial

```
   Input: A node $V$ of the tile dependence graph
 1 procedure Expand_Spacecut($V$)
 2     $i \leftarrow 0$ ;
 3     $Q_0 \leftarrow \{V\}$ ;
 4     foreach Spatial dimension $d$ that needs to be cut do
 5         $Q_{i+1} \leftarrow \emptyset$ ;
 6         foreach $X \in Q_i$ do
 7             $\{X^L, X^C, X^R\} \leftarrow$ Spacecut($X$, $d$) ;
 8             if $X$ inverted in dimension $d$ then
 9                 Add edges $X^C \rightarrow X^L$ and $X^C \rightarrow X^R$ ;
10             else
11                 Add edges $X^L \rightarrow X^C$ and $X^R \rightarrow X^C$ ;
12             end
13             foreach Edge $U \rightarrow X$ and $Y \in \{X^L, X^C, X^R\}$ do
14                 if Dependent($U$, $Y$) then Add edge $U \rightarrow Y$ ;
15             end
16             foreach Edge $X \rightarrow U$ and $Y \in \{X^L, X^C, X^R\}$ do
17                 if Dependent($Y$, $U$) then Add edge $Y \rightarrow U$ ;
18             end
19             Delete $X$ from the graph;
20             $Q_{i+1} \leftarrow Q_{i+1} \cup \{X^L, X^C, X^R\}$ ;
21         end
22         $i \leftarrow i + 1$ ;
23     end
24     foreach $X \in Q_{i-1}$ do
25         Expand($X$) ;
26     end
27 end
```

Figure 3.7: The expansion function that decomposes a node in the tile dependence graph using a spacecut. Hypertrapezoids are identified with the nodes that represent them, and the notation introduced in Section 2.1.2 is used for subtrapezoids resulting from a spacecut.

dimensions that need to be decomposed (lines 4–23). Thus, let i be the number of spacecuts already applied, and let d be the dimension in which the next spacecut is applied. Furthermore, let Q_i be the set containing all 3^i subtrapezoids that result from the previous spacecuts. A spacecut in dimension d is then applied to each trapezoid $X \in Q_i$, which yields subtrapezoids X^L, X^C and X^R (line 7, see Section 2.1.2 for details). If X is inverted in dimension d, the trapezoids X^L and X^R are dependent on X^C, hence the edges $X^C \rightarrow X^L$ and $X^C \rightarrow X^R$ are added. Otherwise, X^C is dependent on X^L and X^R, so the algorithm adds the edges $X^L \rightarrow X^C$ and $X^R \rightarrow X^C$ (lines 8–12). Subsequently, all incoming edges $U \rightarrow X$ of the original node X are checked, and if the dependence relation confirms a dependence between U and one of X^L, X^C or X^R, the corresponding edge is added (lines 13–15). All outgoing edges $X \to U$ are checked in the same fashion, and edges between X^L, X^C or X^R and U are created where needed (lines 16–18). That done, the original node X and all its incoming and outgoing edges are deleted, and X^L, X^C and X^R are appended to Q_{i+1} (lines 19–20), in order to be decomposed by a further spacecut. Finally, after applying all necessary spacecuts, every newly created node is expanded recursively (lines 24–26).

```
   Input: A node $V$ of the tile dependence graph
1  procedure Expand_Timecut($V$)
2      $\{V^B, V^T\} \leftarrow$ Timecut($V$);
3      Add edge $V^B \to V^T$ ;
4      foreach Edge $U \to V$ and $Y \in \{V^B, V^T\}$ do
5          if Dependent($U$, $Y$) then Add edge $U \to Y$ ;
6      end
7      foreach Edge $V \to U$ and $Y \in \{V^B, V^T\}$ do
8          if Dependent($Y$, $U$) then Add edge $Y \to U$ ;
9      end
10     Delete $V$ from the graph;
11     Expand($V^B$) ;
12     Expand($V^T$) ;
13 end
```

Figure 3.8: The expansion function that decomposes a node in the tile dependence graph using a timecut. Hypertrapezoids are identified with the nodes that represent them, and the notation introduced in Section 2.1.2 is used for subtrapezoids resulting from a timecut.

A node V is decomposed using a timecut whenever a spacecut is either not applicable or not required in any dimension. Figure 3.8 displays the correspondent algorithm. A timecut is applied to V, resulting in subtrapezoids V^B and V^T, where V^T is dependent on V^B. Therefore, the edge $V^B \to V^T$ is created immediately (lines 2–3). The incoming and outgoing edges of the original node V are then checked in the same way as above, and dependences confirmed by the dependence relation are added as incoming or outgoing edges to V^B and V^T (lines 4–9). Eventually, the original node and all of its edges are removed, and the new nodes V^B and V^T are processed recursively (lines 10–12).

The expansion algorithm terminates, since each spacecut or timecut results in subtrapezoids that contain strictly less grid points than the original trapezoid. Thus, the number of grid points within a trapezoid strictly decreases in each recursive step. Furthermore, the expansion algorithm is correct, i.e., the grid passed to the precut component is contained in the conjunction of all trapezoids in the tile dependence graph, and all inter-trapezoid dependences are represented as edges in the tile dependence graph.

This can be proved directly by relying on the correctness of Pochoir's trapezoidal tiling algorithm [4]. We employ the same spacecuts and timecuts as Pochoir, so surely the conjunction $\bigcup_{X \in Q_{i-1}} X$ of all subtrapezoids resulting from a spacecut is equal to the original trapezoid V, and equally $V^B \cup V^T = V$ when applying a timecut. Trapezoids are deleted without being replaced by equivalent subtrapezoids only if they do not contain any points within the grid. Hence, the grid is contained in the conjunction of all trapezoids in the tile dependence graph.

When applying a spacecut or timecut in a recursive expansion step, dependences are handled in two steps. In the first step, all dependences between newly created subtrapezoids are added as edges in the tile dependence graph, just like the trapezoidal tiling algorithm of Pochoir suggests them. In the second step, all incoming and outgoing dependences of the original trapezoid are inherited by the subtrapezoids, with only redundant dependences being removed. Since the subtrapezoids contain the same grid points as the original trapezoid, no further dependences can possibly be induced by a spacecut or timecut. In summary, all actual inter-tile dependences are preserved in each recursive expansion step, and therefore the assertion follows.

Diamond Tiling

Besides using Pochoir's trapezoidal tiling algorithm, it is also possible to generate the tile dependence graph based on the diamond tiling algorithm that is employed by the PLUTO compiler. Figure 3.9 displays the respective generation algorithm. For a d-dimensional stencil, it accepts a d-dimensional linear transformation T and tile sizes $O = (s_1, \dots, s_d)$ as input, where T has been computed using PLUTO's transformation algorithm (see Section 2.2.2). A tile dependence graph is then built that reflects rectangular tiling of the grid that was passed to the precut component. The tile shape is determined by applying the transformation T to the grid, and the tile size is determined by $(s_1, \dots, s_d)$.

The algorithm begins by representing the d-dimensional grid that was passed to the precut component as a polyhedron $\mathscr{D}$ (line 2). This is just the polyhedron that represents a perfect loop nest which traverses all points in the grid. Afterwards, d additional tile dimensions $(z_1, \dots, z_d)$ are appended as outermost dimensions to the polyhedron in preparation for rectangular tiling (see Section 2.2.2). The latter is then performed in each spatial dimension, while implicitly applying the transformation T to the original grid (lines 3–5). For each spatial dimension i, this is achieved by imposing the tiling constraints

$$\begin{aligned} \phi_i \circ \vec{x} &\geq z_i \cdot s_i \\ \phi_i \circ \vec{x} &\leq (z_i + 1) \cdot s_i - 1 \end{aligned}$$

on the polyhedron $\mathscr{D}$, where $\vec{x}$ is the original iteration vector and ϕ_i is the i-th tiling hyperplane of T. We can then obtain bounds on the tiling dimensions by eliminating the original iteration vector $\vec{x}$ from $\mathscr{D}$, which is a standard operation supplied by integer programming libraries (line 6). This yields a polyhedron which contains all tile instances $(z_1, \dots, z_d)$ identifying tiles that contain at least one point of the grid. Therefore, we iterate over all of these coordinates and create the respective tile nodes (lines 7–9). Then, inter-tile dependences are added for each tile node (lines 11–16). It is sufficient to check all immediate neighbors of a tile node for dependences, as inter-tile dependences are unit vectors along the bases of the transformed iteration space [5]. Also, we only need to check dependences in one direction, as an outgoing dependence $V \rightarrow W$ for the node V is equivalent to an incoming dependences for the node W. We chose to check incoming dependences, as this allows us to identify at once tiles that are dependent only on the initial conditions of the stencil computation (line 15). After this final step, the tile dependence graph is complete, and we return its root node.

```
Input: A $d \times d$ linear transformation matrix $T$ as computed by PLUTO's
       transformation algorithm and tile sizes $O = (s_1, \dots, s_d)$
1  procedure Generate($T$, $O$)
2      Build grid polyhedron $\mathscr{D}$ that contains all grid points and add tile dimensions
       $(z_1, \dots, z_d)$ ;
3      foreach Tiling hyperplane $\phi_i$ in $T$ do
4          Add tiling constraints $\phi_i \circ \vec{x} - z_i \cdot s_i \geq 0$ and $(z_i + 1) \cdot s_i - 1 - \phi_i \circ \vec{x} \geq 0$
           where $\vec{x}$ is the original iteration vector to the grid polyhedron $\mathscr{D}$ ;
5      end
6      Eliminate the original iteration vector $\vec{x}$ from $\mathscr{D}$ to obtain bounds on the tile
       dimensions ;
7      foreach Valid tile coordinate $(z_0, \dots, z_d)$ do
8          Create tile node representing the tile with coordinates $(z_0, \dots, z_d)$ ;
9      end
10     Create root node $R$ ;
11     foreach Tile node $V$ do
12         foreach Neighboring tile node $W$ do
13             if Dependent($W$, $V$) then Add edge $W \to V$ ;
14         end
15         if $V$ has no incoming edges then Add edge $R \to V$ ;
16     end
17     return Root node $R$
18 end
```

Figure 3.9: The generation algorithm that builds a tile dependence graph based on PLUTO's diamond tiling algorithm.

The correctness of this algorithm emerges from the correctness of PLUTO's original diamond tiling algorithm [5, 13]. The linear transformation that we apply to the grid polyhedron $\mathscr{D}$ is obtained by using this algorithm. Bondhugula et al. prove that any rectangular tiling is valid in the thereby transformed iteration space, so the tile nodes we generate represent valid tiles. Furthermore, as stated above, dependences can only be present between immediately adjacent tiles, so no inter-tile dependences are lost by the algorithm.

Split Tiling

The final variation of tile dependence graph generation employs the hybrid or nested split tiling algorithms of the SDSL compiler. Both of these have been unified in one generation algorithm, which is illustrated in Figure 3.10. The tile dependence graph is built depending on the tile sizes O that are passed to the algorithm. If nested split tiling is used, tile sizes are defined by $O = (s^T, s_1^U, s_1^I, \dots, s_d^U, s_d^I)$, where s^T is the size of a tile in the time dimension, and s_i^U and s_i^I are the widths of upright and inverted tiles in the spatial dimension i, respectively. On the other hand, if hybrid split tiling is requested, tile sizes are defined by $O = (s^T, s^P, s_1^U, s_1^I, \ldots, s_{d-1}^U, s_{d-1}^I)$, where again s^T is the size of a tile in the time dimension, and s_i^U and s_i^I are the widths of upright and inverted tiles in the spatial dimension i, respectively. Additionally, s^P now identifies the width of parallelogram-shaped tiles in the outermost spatial dimension d.

To begin with, the time dimension of the grid that was handed to the precut component is tiled, i.e., subdivided into time bands of height s^T (lines 4–12). For each of these time bands, a tile node V is created and placed in a separate buffer (lines 5–6). The latter will

```
   Input: Tile sizes $O = (s^T, s_1^U, s_1^I, \ldots, s_d^U, s_d^I)$ for nested split tiling or
          $O = (s^T, s^P, s_1^U, s_1^I, \ldots, s_{d-1}^U, s_{d-1}^I)$ for hybrid split tiling
1  procedure Generate($O$)
2      Create root node $R$ ;
3      Determine the number $n$ of time bands with height $s^T$ that is necessary to
       subdivide the grid in the time dimension ;
4      for $i = 1$ to $n$ do
5          Create tile node $V$ representing the entire $i$-th time band ;
6          Create time band buffer $Q_i \leftarrow \{V\}$ ;
7          if $i = 1$ then
8              Add edge $R \rightarrow V$ ;
9          else
10             Add edge $W \rightarrow V$ for $W \in Q_{i-1}$ ;
11         end
12     end
13     foreach Spatial dimension $d$ do
14         if $d$ is outermost dimension and hybrid split tiling is performed then
15             foreach Time band $i$ do
16                 Tile_Traditional( $O, Q_i, d$ ) ;
17             end
18         else
19             foreach Time band $i$ do
20                 Tile_Split( $O, Q_i, d$ ) ;
21             end
22         end
23     end
24     return Root node $R$
25 end
```

Figure 3.10: The generation algorithm that builds a tile dependence graph based on the SDSL compiler's split tiling algorithm. For traditional tiling of a spatial dimension, the `Tile_Traditional` function displayed in Figure 3.11 is called. If split tiling is employed in a spatial dimension, the `Tile_Split` function displayed in Figure 3.12 is invoked.

```
Input: Tile sizes $O$, a time band buffer $Q_i$ and a spatial dimension $d$
1  procedure Tile_Traditional( $O, Q_i, d$ )
2      Remember the original buffer $Q_i$ as $R_i$;
3      foreach Node $X \in R_i$ do
4          Determine the number $n$ of parallelogram-shaped tiles with width $s^P$ that is
           necessary to subdivide $X$ in dimension $d$ ;
5          for $j = 1$ to $n$ do
6              Create tile node $Y_j$ representing the $j$-th tile ;
7              if $j > 1$ then
8                  Add edge $Y_{j-1} \rightarrow Y_j$ ;
9              end
10         end
11         foreach Edge $U \rightarrow X$ and $Y \in \{Y_1, \dots, Y_n\}$ do
12             if Dependent($U$, $Y$) then Add edge $U \rightarrow Y$ ;
13         end
14         foreach Edge $X \rightarrow U$ and $Y \in \{Y_1, \dots, Y_n\}$ do
15             if Dependent($Y$, $U$) then Add edge $Y \rightarrow U$ ;
16         end
17         Delete the original node $X$ from the graph and $Q_i$ ;
18         $Q_i \leftarrow Q_i \cup \{Y_1, \dots, Y_n\}$ ;
19     end
20 end
```

Figure 3.11: The algorithm used for traditional parallelogram tiling of a time band during hybrid split tiling.

be used during tiling of the spatial dimensions later, similar to how a buffer is employed in the spacecut expansion function of trapezoidal tiling. Each time band is dependent on the preceding time band in the backward direction along the time dimension, so the corresponding edge is added next. The first time band is dependent on no other tiles, so an edge to the root node of the tile dependence graph is added instead (lines 7–11).

Afterwards, each spatial dimension d is tiled one after another (lines 13–23). If d is the outermost spatial dimension, and hybrid split tiling is requested, traditional parallelogram tiling is performed in that dimension (lines 14–17). Otherwise, that dimension is split-tiled (lines 18–22).

In order to traditionally tile a time band in a spatial dimension d, the tile sizes O, the time band buffer Q_i and the dimension d are passed to the `Tile_Traditional` function (see Figure 3.11). First of all, the time band buffer is backed up, as tiling modifies the contents of Q_i (line 2). The spatial dimension d of each tile node X in the original time band buffer is then subdivided into parallelogram-shaped tiles of width s^P (lines 3–19), and a tile node is created for each of these tiles (line 6). Due to the shape of tiles, this newly created tile node is certainly dependent on the previous tile node in the backward direction along the dimension d. Accordingly, an edge is added between these nodes

```
Input: Tile sizes $O$, a time band buffer $Q_i$ and a spatial dimension $d$
procedure Tile_Split( $O, Q_i, d$ )
    Remember the original buffer $Q_i$ as $R_i$;
    foreach Node $X \in R_i$ do
        Determine the number $n$ of parallelogram-shaped tiles with width $s_d^I + s_d^U$
        that is necessary to subdivide $X$ in dimension $d$ ;
        for $j = 1$ to $n$ do
            Create tile node $Y_j^I$ for the $j$-th inverted trapezoidal tile with width $s_d^I$ ;
            Create tile node $Y_j^U$ for the $j$-th upright trapezoidal tile with width $s_d^U$ ;
            Add edge $Y_j^U \to Y_j^I$ ;
            if $j > 1$ then
                Add edge $Y_{j-1}^U \to Y_j^I$ ;
            end
        end
        foreach Edge $U \to X$ and $Y \in \{Y_1^I, Y_1^U, \dots, Y_n^I, Y_n^U\}$ do
            if Dependent($U$, $Y$) then Add edge $U \to Y$ ;
        end
        foreach Edge $X \to U$ and $Y \in \{Y_1^I, Y_1^U, \dots, Y_n^I, Y_n^U\}$ do
            if Dependent($Y$, $U$) then Add edge $Y \to U$ ;
        end
        Delete the original node $X$ from the graph and $Q_i$ ;
        $Q_i \leftarrow Q_i \cup \{Y_1^I, Y_1^U, \dots, Y_n^I, Y_n^U\}$ ;
    end
end
```

Figure 3.12: The algorithm employed for split tiling of a time band.

(lines 7–9). Subsequently, the dependences of the original node X are checked, and if the dependence relation confirms any of these dependences for one of the newly created nodes, the corresponding edge is added to the tile dependence graph (lines 11–16). Finally, the original node is deleted from the tile dependence graph and the time band buffer Q_i, in which it is replaced by the new nodes (lines 17–18).

Split tiling a time band is done quite similar to traditional tiling, as is shown in Figure 3.12. Therefore, we take a closer look only at the differences between these two algorithms, which lie with the creation of tile nodes. Instead of creating parallelogram-shaped tiles of width s^P, the algorithm initially subdivides the time band into parallelogram-shaped tiles of width $s_d^I + s_d^U$ (line 4). Each of these tiles is then composed of an inverted tile of width s_d^I and an upright tile of width s_d^U, where the inverted tile is dependent on the upright tile (lines 6–8). Additionally, the inverted tile is also dependent on the previous upright tile in the backward direction along the dimension d, if such a tile is present (lines 9–11). Processing the dependences of the original node is done in the same way as for parallelogram tiles once again.

There are several observations to be made concerning the correctness of the generation algorithm. First of all, the way in which we subdivide the grid into time bands clearly preserves all grid points and inter-band dependences. Second, traditional and split tiling of a time band can be done independently in each spatial dimension by design [1]. Therefore, it remains to prove that the `Tile_Traditional` and `Tile_Split` algorithms are correct. As we know the original tiling scheme employed by the SDSL compiler to be correct, surely all grid points are preserved when subdividing a time band. All inter-tile dependences within the time band that arise from subdivision are accounted for as well. Every inter-tile dependence that lies outside the time band is initially inherited by the newly created tile nodes, and is eliminated only if the dependence relation does not confirm this dependence.

3.3.3 Code Generation Component

When performing a stencil computation, it is necessary to evaluate the kernel function on the points of a physical data grid. For that purpose, optimized C++ code is generated by the code generation component for a given specification of a stencil computation. This C++ code is then compiled to a shared object file, which exposes several functions that take a data grid and a tile representation as input, and in turn evaluate the specified kernel function on the grid for all points within this tile. Different functions are generated for tiles that do or do not contain boundary accesses, since more optimizations are applicable when no boundary accesses have to be considered. Code generation varies slightly between tiling algorithms, due to differing tile shape and tile representation. However, the general strategy and the optimizations we perform do not change, so we refrain from discussing these variations in detail.

As an example, the structure of the code that is generated for a one-dimensional trapezoidal tile is shown in Figure 3.13. When generating code for a d-dimensional stencil computation, $d+1$ nested loops are created, where the innermost loop contains the actual kernel function evaluation. The outermost loop scans the time dimension of a tile,

```
// Loop for time dimension
for (int iter_t = param_t0; iter_t < param_t1; iter_t++) {

  // Loop for spatial dimension 0
  for (int iter_x0 = param_x0[0]; iter_x0 < param_x1[0];
    iter_x0++) {
    // Kernel function code
  }

  // Update inner loop bounds
  for (int dim = 0; dim < 1; dim++) {
    param_x0[dim] += param_slope[dim];
    param_x1[dim] -= param_slope[dim];
  }
}
```

Figure 3.13: Structure of generated code for a one-dimensional Pochoir-like tile. All variables `param_XXX` originate from the tile representation passed to the code.

while the remaining loops scan one spatial dimension each. As a result, all dependences between iterations are satisfied regardless of the actual shape of a stencil. This is valid, since all dependences have a positive component in the time dimension by definition. The spatial loops are ordered in such way, that the innermost loop scans the dimension which allows for unit-stride access of the physical data grid elements. Thus, this loop is amenable to vectorization. After the spatial dimensions have been scanned in a time step, the bounds of the spatial loops are updated depending on the tile properties.

```
// Loop for spatial dimension 0
for (int iter_x0 = param_x0[0]; iter_x0 < param_x1[0]; iter_x0++) {

  grid.interior(iter_t, iter_x0) = (0.125000 *
                  ((grid.interior(iter_t + (-1), iter_x0 + (-1))
    - (2.000000 * grid.interior(iter_t + (-1), iter_x0 + ( 0))))
                  + grid.interior(iter_t + (-1), iter_x0 + ( 1))));
}
```

(a)

```
ptr_array_0 = grid.storage_ptr(iter_t - 0, param_x0[0]);
ptr_array_1 = grid.storage_ptr(iter_t - 1, param_x0[0]);

// Loop for spatial dimension 0
for (int iter_x0 = param_x0[0]; iter_x0 < param_x1[0]; iter_x0++,
  ptr_array_0++,
  ptr_array_1++) {

  *ptr_array_0 = (0.125000 * ((ptr_array_1[(-1)]
    - (2.000000 * ptr_array_1[0])) + ptr_array_1[(1)]));
}
```

(b)

Figure 3.14: Kernel function code for the one-dimensional heat equation stencil. In (a), grid elements are accessed through the respective access function of the `grid` object, while in (b), C-style pointer manipulations are employed.

Several further optimizations are now performed on the code that is generated for interior tiles. First of all, we replace accesses to grid elements with C-style pointer manipulations, as is shown in Figure 3.14. This reduces the number of index calculations that have to be done on each access. Additionally, autovectorization is facilitated for the underlying C++ compiler. In the rare case that the compiler nevertheless fails to vectorize the kernel function code, e.g., due to overly complex kernel functions, we can also generate vectorized code ourselves for different vectorization instruction set architectures. However, we have never encountered issues concerning autovectorization during our experimental evaluation. Finally, the user can choose to unroll the second-to-innermost spatial loop, if applicable. This reduces the stress on the memory subsystem on some architectures, because data elements can be reused between iterations of the second-to-innermost spatial loop.

Chapter 4

Empirical Evaluation

Extensive experimental evaluation of our proposed approach was performed, using a number of stencil benchmarks on different hardware platforms. As a performance reference, we used the original implementations of the trapezoidal tiling scheme of Pochoir (version 0.5), the diamond tiling approach of PLUTO (version 0.10.5), and the split tiling system of the SDSL compiler (version 0.3.1) [1, 4, 5]. The hardware and benchmarks used for evaluation are presented in Section 4.1, and the results are discussed in Section 4.2.

4.1 Experimental Setup

Hardware

We performed benchmarks on two hardware platforms, one based on an AMD processor and one based on an Intel processor. The former is an AMD Phenom II X6 1045T hexa-core x86-64 processor, which is clocked at 2.7 GHz. It supports the SSE2 instruction set, and therefore exhibits a double precision peak performance of 10.8 GFLOPS per core (64.8 GFLOPS aggregate). The latter is an Intel Core i7-2630QM quad-core x86-64 processor, clocked at 2.0 GHz. The AVX instruction set is supported by this CPU model, so the double precision peak performance per core is 16.0 GFLOPS (64.0 GFLOPS aggregate). Dynamic frequency and voltage scaling features and hyper-threading, if applicable, were disabled on these processors during benchmarks, in order to acquire reproducible and comparable results. On both platforms, benchmarks were compiled using the Intel C++ Compiler v14.0, with the `-O3 -ipo -xHOST` optimization flags.

Benchmarks

A variety of stencil codes was used in our benchmarks, which we introduce by the names that later are used to refer to them. The one-dimensional three-point heat equation stencil that was used as an example throughout this thesis is referred to as heat-1d. We also implemented two- and three-dimensional variants of the heat equation stencil, which are identified by heat-2d and heat-3d, respectively. The two-dimensional variant is a five-point stencil, while the three-dimensional variant is a seven-point stencil. Furthermore, several Jacobi smoother stencils were implemented. j1d3 is a one-dimensional three-point stencil, j2d5 and j2d9 are two-dimensional five-point and nine-point stencils and j3d7 is a three-dimensional seven-point stencil.

The memory footprint of the data grid was set to 47.68 MB for one- and two-dimensional benchmarks. This was achieved by choosing spatial dimension sizes of $6.25 \cdot 10^6$ grid points for one-dimensional stencils, and 2500^2 grid points for two-dimensional stencils. Stencil computations were performed over 2000 time steps each. Three dimensional stencil computations were performed on a data grid of size 400^3 over 200 time steps, resulting in a memory footprint of 488.28 MB. Thus, the data grid is large enough to overflow the last level cache on both platforms for sure.

We employed autotuning to find near-optimal tile sizes for our implementation, the PLUTO compiler and the SDSL compiler. No autotuning was performed for the Pochoir stencil compiler, as it does not expose tile sizes to the user. In doing so, we relied on the autotuning tool distributed with the PATUS framework [6], which can be used for autotuning of arbitrary programs. Since a genetic algorithm is employed, the time needed for autotuning was reduced significantly in comparison to plain traversal of the entire parameter search space.

Split tiling can be performed either as fully nested split tiling or as hybrid split tiling, which was detailed in Section 2.3.2. Hybrid split tiling generally exhibits higher performance than nested split tiling, due to smaller tile sizes being valid in stencil computations of higher dimensionality. For that reason, we decided to evaluate only hybrid split tiling on the two- and three-dimensional benchmarks. Hybrid split tiling is not applicable to one-dimensional stencil computations, so nested split tiling was performed on the respective benchmarks.

4.2 Experimental Results

Figure 4.1 shows absolute performance for all benchmarks on the AMD-based platform, and Figure 4.2 does so for experiments on the Intel-based platform. Our optimization approach exhibits a strong performance benefit for two- and three-dimensional stencil codes on both platforms, while falling behind the SDSL compiler for several one-dimensional stencil codes.

One-Dimensional Benchmarks

Based on the preliminary evaluation of our approach for Pochoir's trapezoidal tiling algorithm [9], a notable performance improvement over Pochoir and PLUTO was expected for one-dimensional benchmarks. This expectation proved true on both platforms. On the AMD platform, optimized trapezoidal tiling outperforms Pochoir between $1.27\times$ on j1d3 and $1.59\times$, while the speedup of optimized diamond tiling over PLUTO ranges from a low of $1.50\times$ on heat-1d to a high of $1.60\times$ on j1d3. Slightly lower speedup is achieved on the Intel platform, where optimized trapezoidal tiling performs up to $1.32\times$ faster than Pochoir on heat-1d, and optimized diamond tiling performs up to $1.59\times$ faster on j1d3.

The optimized vector parallelism provided by the SDSL compiler proved to be considerably more effective than trapezoidal or diamond tiling, performing comparable to our approach on the AMD platform, and outperforming it on the Intel platform by up to $1.28\times$. Consistent with the results reported by Henretty et al., the SDSL compiler achieved a major speedup over Pochoir and PLUTO on all one-dimensional benchmarks.

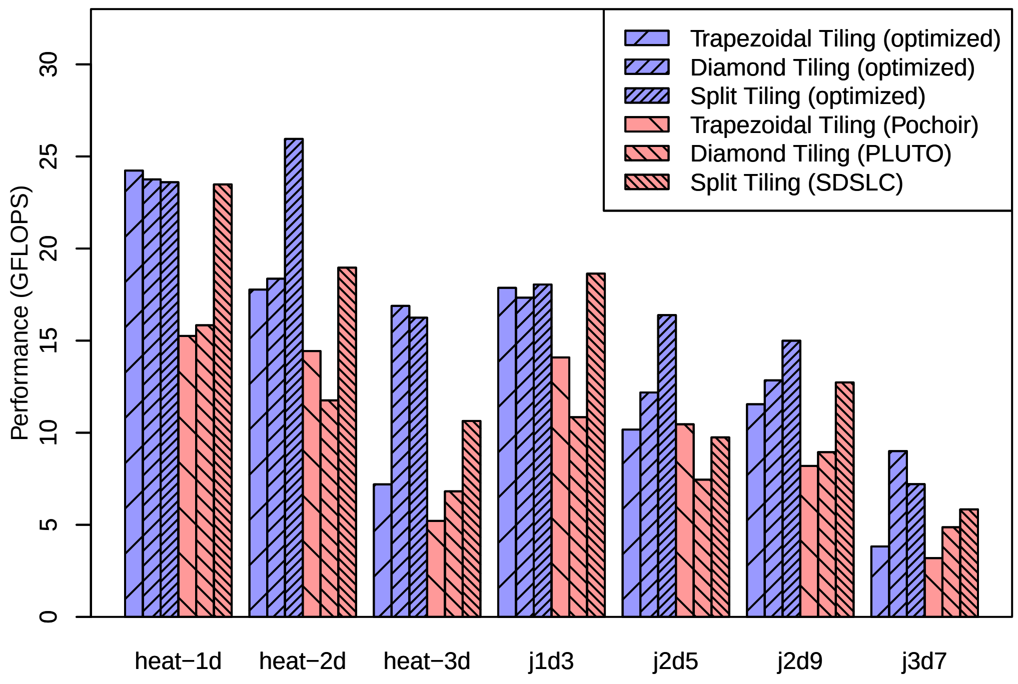


Figure 4.1: AMD Phenom II X6 1045T SSE2 Performance

Two-Dimensional Benchmarks

The performance improvement our approach achieves increases significantly for two-dimensional benchmarks. On the Intel quad core platform, optimized diamond tiling and split tiling outperform Pochoir, PLUTO and the SDSL compiler across all benchmarks. Especially hybrid split tiling was able to benefit greatly from execution order optimization, resulting in a speedup over the SDSL compiler between 1.35× on heat-2d and 2.14× on j2d9. Optimized diamond tiling achieved a speedup over the PLUTO compiler ranging from 1.37× on j2d5 to 1.89× on j2d9. Although the SDSL compiler falls behind our implementation, it still yields a considerable performance increase over Pochoir and PLUTO on most benchmarks.

On the AMD hexacore platform, optimized split tiling also exhibits high performance, while optimized diamond tiling falls behind the SDSL compiler on heat-2d. Speedup for split tiling ranges from a low of 1.18× on j2d9 to a high of 1.68× on j2d5, while optimized diamond tiling achieves a speedup over PLUTO of up to 1.64× on j2d5. On both platforms, optimized as well as unoptimized trapezoidal tiling fell behind diamond tiling and split tiling on most benchmarks, which has already been observed in previous research, and was therefore expected [1,5].

Hybrid split tiling as performed by the SDSL compiler limits parallelism to the innermost spatial dimension for two-dimensional stencil computations, because the outermost dimension is decomposed into parallelogram shaped tiles which are processed sequentially. Our optimization approach eliminates dependences that restrict parallelism to the innermost spatial dimension, thus improving parallelism. Additionally, Henretty et al. observed load balancing issues on an AMD hexacore platform for two-dimensional stencil computations, which are addressed by our dynamic scheduling approach as well.

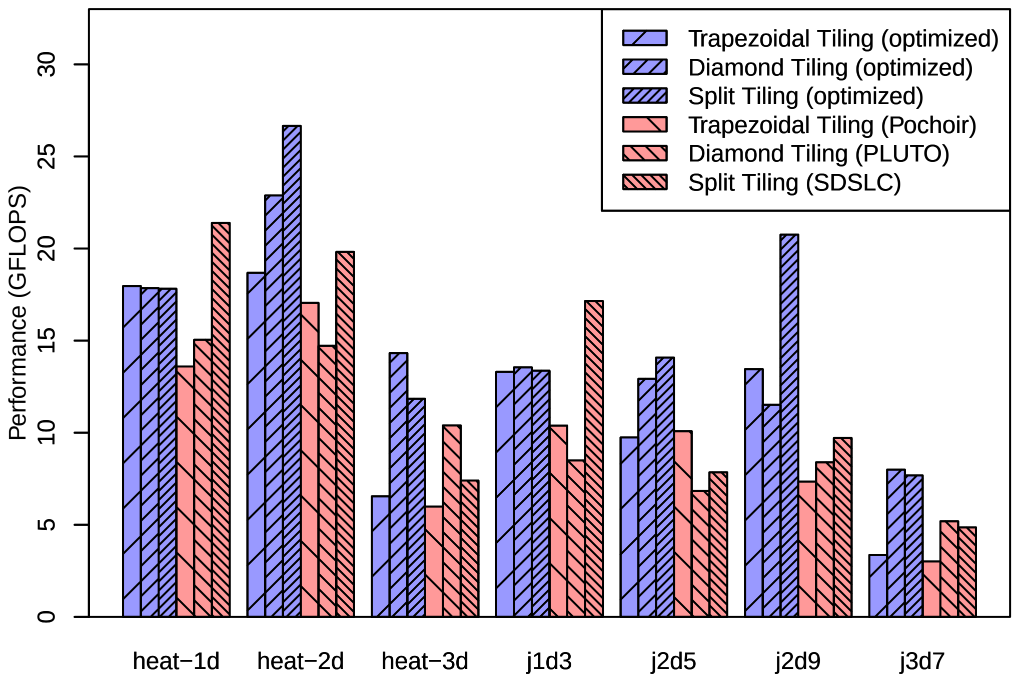


Figure 4.2: Intel Core i7-2630QM AVX Performance

Three-Dimensional Benchmarks

Since the number of redundant dependences increases with dimensionality of the stencil computation, high performance was expected of optimized diamond tiling and split tiling for three-dimensional benchmarks, and was indeed observed. Again, the performance of optimized diamond tiling and split tiling exceeds that of all remaining tiling algorithms on both platforms. However, optimized split tiling falls behind optimized diamond tiling, most likely due to the tile size requirements imposed on split tiles (see Section 2.3.2). On the AMD hexacore platform, speedup of optimized diamond tiling over PLUTO ranges from a low of 1.85× to a high of 2.48×, while optimized diamond tiling exhibits a speedup of 1.23× up to 1.53×. Benchmarks on the Intel quadcore platform yield a speedup ranging from 1.38× to 1.54× for diamond tiling and 1.60× for split tiling.

In compliance with previous research, PLUTO's diamond tiling outperforms Pochoir on all benchmarks. However, we were not able to reproduce a performance drawback of the SDSL compiler's hybrid split tiling in comparison to Pochoir or PLUTO, as it was observed by Henretty et. al. [1]. Generally, PLUTO exhibited a significantly lower than expected performance across all benchmarks on all platforms, which peaked in a 2.48× speedup of optimized diamond tiling over PLUTO for heat-3d on the AMD platform. We suspect that this performance anomaly can be traced back to PLUTO not being able to parse and process modulo operations in array subscripts. The kernel functions of all stencil computations that we used for benchmarks access only the value of grid points in the previous time step. Therefore, it is sufficient to maintain two copies of the space grid at a time, one of which contains values at time $t+1$, and the other contains values at time t. As opposed to our implementation, as well as to Pochoir and the SDSL compiler, PLUTO does not account for this during code generation. The workaround employed by the original examples shipped with the PLUTO compiler is to modify the source code accordingly after code generation through a shell script. However, we decided against this workaround, since is highly error-prone. Instead, we map array accesses to the respective grid copy using modulo operations that are performed by a wrapper template. The downside of this solution is that now modulo operations are performed on each single array access, leading to the considerable slowdown that we observed.

Chapter 5

Conclusion

In this thesis, we have proposed an approach to increase the performance of existing tiling schemes in stencil computations further. This is achieved by eliminating redundant inter-tile dependences, and by employing a dynamic schedule for tile processing, thus improving both load balance and data reuse. Inter-tile dependences correspond to either implicit or explicit synchronization, so reducing the number of dependences decreases the amount of synchronization that is necessary during a stencil computation. In particular, by dynamically scheduling tiles onto available processor cores, we were able to avoid barrier synchronization entirely. The latter is, for example, commonly present at the end of parallelized loops, as they are generated by the PLUTO and SDSL compilers. Our optimization approach follows the general idea of the compiler-assisted dynamic scheduling approach proposed by Baskaran et al. for the PLUTO compiler [28], but we have studied the applicability of dynamic scheduling techniques for a wider range of tiling algorithms.

We implemented our optimization approach as a C++ template library for the tiling algorithms employed the Pochoir stencil compiler, the PLUTO compiler, and the SDSL compiler. Experimental evaluation of the approach on a number of stencil benchmarks exhibits a strong performance increase over the original tiling schemes of the respective compilers. While we fell behind the SDSL compiler for one-dimensional stencil computations, a consistent speedup frequently exceeding $1.5\times$ was observed for two- and three-dimensional stencil computations on both an AMD-based and an Intel-based platform. Peak speedup values beyond $2.0\times$ were observed, however these were traced back to a technical limitation of the original tiling framework.

Tight restrictions were imposed on stencil codes throughout this thesis, since we examined only single-statement stencils with constant dependences. Future research is necessary in order to determine to what extent our approach can be applied to arbitrary stencil computations. Furthermore, our scheme could be enhanced further in several ways. For instance, we did not incorporate the DLT data layout transformation performed by the SDSL compiler into our implementation, leaving room for improvements concerning fine-grained vector parallelism. We also relied entirely on the OpenMP scheduler for distributing tile processing tasks, which is another starting point for additional optimizations.

Bibliography

[1] T. Henretty, R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan, “A Stencil Compiler for Short-vector SIMD Architectures,” in *Proceedings of the 27th Annual International Conference on Supercomputing (ICS).* ACM, 2013, pp. 13–24.

[2] Y. Tang, R. Chowdhury, C.-K. Luk, and C. E. Leiserson, “Coding Stencil Computations Using the Pochoir Stencil-Specification Language,” Poster session presented at the 3rd USENIX Workshop on Hot Topics in Parallelism, 2011.

[3] P. Micikevicius, “3D Finite Difference Computation on GPUs Using CUDA,” in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU-2).* New York, NY, USA: ACM, 2009, pp. 79–84.

[4] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson, “The Pochoir Stencil Compiler,” in *Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA).* ACM, 2011, pp. 117–128.

[5] V. Bandishti, I. Pananilath, and U. Bondhugula, “Tiling Stencil Computations to Maximize Parallelism,” in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC).* IEEE Computer Society Press, 2012, pp. 40:1–40:11.

[6] M. Christen, O. Schenk, and H. Burkhart, “PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures,” in *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium (IPDPS).* IEEE Computer Society Press, 2011, pp. 676–687.

[7] U. Bondhugula, J. Ramanujam, and P. Sadayappan, “PLuTo: A Practical and Fully Automatic Polyhedral Parallelizer and Locality Optimizer,” The Ohio State University, Tech. Rep. OSU-CISRC-10/07-TR70, 2007.

[8] T. Henretty, J. Holewinski, N. Sedaghati, L.-N. Pouchet, A. Rountev, and P. Sadayappan, “Stencil Domain Specific Language (SDSL) User Guide 0.2.2 draft,” The Ohio State University, Tech. Rep. OSU TR OSU-CISRC-4/13-TR09, 2013.

[9] M. Freitag, “Using a Dynamic Schedule to Increase the Performance of Tiling in Stencil Computations,” in *Proceedings of the 5th IEEE Germany Student Conference.* IEEE, 2014, pp. 45–48.

[10] M. Wolfe, “More Iteration Space Tiling,” in *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*. ACM, 1989, pp. 655–664.

[11] M. Frigo and V. Strumpen, “Cache Oblivious Stencil Computations,” in *Proceedings of the 19th Annual International Conference on Supercomputing (ICS)*. ACM, 2005, pp. 361–366.

[12] ——, “The Cache Complexity of Multithreaded Cache Oblivious Algorithms,” in *Proceedings of the 18th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 2006, pp. 271–280.

[13] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, “Automatic Transformations for Communication-minimized Parallelization and Locality Optimization in the Polyhedral Model,” in *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction (CC / ETAPS)*. Springer-Verlag, 2008, pp. 132–146.

[14] S. Verdoolaege, “Integer Set Library: Manual,” 2014. [Online]. Available: http://isl.gforge.inria.fr/manual.pdf

[15] 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 / ETAPS)*. Springer-Verlag, 2010, pp. 283–303.

[16] J. Ramanujam, “Beyond Unimodular Transformations,” *J. Supercomput.*, vol. 9, no. 4, pp. 365–389, Dec. 1995.

[17] J. Xue, “Automating Non-unimodular Loop Transformations for Massive Parallelism,” *Parallel Comput.*, vol. 20, no. 5, pp. 711–728, May 1994.

[18] C. Bastoul, “Code Generation in the Polyhedral Model Is Easier Than You Think,” in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE Computer Society, 2004, pp. 7–16.

[19] 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 (PLDI)*. ACM, 2008, pp. 101–113.

[20] T. Henretty, K. Stock, L.-N. Pouchet, F. Franchetti, J. Ramanujam, and P. Sadayappan, “Data Layout Transformation for Stencil Computations on Short-vector SIMD Architectures,” in *Proceedings of the 20th International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software (CC / ETAPS)*. Springer-Verlag, 2011, pp. 225–245.

[21] Intel Corporation, “Intel® Cilk Plus Language Specification,” 2010. [Online]. Available: http://software.intel.com/sites/products/cilk-plus/\cilk_plus_language_specification.pdf

[22] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, “Cache-Oblivious Algorithms,” in *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS).* IEEE Computer Society, 1999, pp. 285–.

[23] Y. Tang and C. E. Leiserson, “User Manual for the Pochoir Stencil Compiler.” [Online]. Available: http://people.csail.mit.edu/yuantang/pochoir_manual.pdf

[24] R. Mei, W. Shyy, D. Yu, and L.-S. Luo, “Lattice Boltzmann Method for 3-D Flows with Curved Boundary,” *J. Comput. Phys.*, vol. 161, no. 2, pp. 680–699, Jul. 2000.

[25] D. A. Orozco and G. R. Gao, “Mapping the FDTD Application to Many-Core Chip Architectures,” in *Proceedings of the 2009 International Conference on Parallel Processing (ICPP).* IEEE Computer Society, 2009, pp. 309–316.

[26] A. Schrijver, *Theory of Linear and Integer Programming*, ser. Wiley Series in Discrete Mathematics & Optimization. John Wiley & Sons, 1998.

[27] 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).* ACM, 2012, pp. 311–320.

[28] M. M. Baskaran, N. Vydyanathan, U. K. R. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan, “Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors,” in *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP).* ACM, 2009, pp. 219–228.

Statement of Authorship

I, Michael Freitag, hereby certify that this bachelors thesis has been composed by myself, and describes my own work, unless otherwise acknowledged in the text. All references and verbatim extracts have been quoted, and all sources of information have been specifically acknowledged. It has not been accepted in any previous application for a degree.

Passau, July 30, 2014

____

(Michael Freitag)