UNIVERSITÄT PASSAU
Fakultät für Informatik und Mathematik

AUTOMATIC PERFORMANCE OPTIMIZATION OF STENCIL CODES

STEFAN KRONAWITTER

A dissertation submitted to the faculty of computer science and mathematics in partial fulfillment of the requirements for the degree of doctor of natural sciences.

ADVISOR: Prof. Christian Lengauer, Ph.D.
EXTERNAL EXAMINER: Prof. Dr. Gerhard Wellein

Passau, June 2019

Stefan Kronawitter: *Automatic performance optimization of stencil codes,*
Dissertation, June 2019

ADVISOR: Prof. Christian Lengauer, Ph.D.
EXTERNAL EXAMINER: Prof. Dr. Gerhard Wellein

ABSTRACT

A widely used class of codes are stencil codes. Their general structure is very simple: data points in a large grid are repeatedly recomputed from neighboring values. This predefined neighborhood is the so-called stencil. Despite their very simple structure, stencil codes are hard to optimize since only few computations are performed while a comparatively large number of values have to be accessed, i. e., stencil codes usually have a very low computational intensity. Moreover, the set of optimizations and their parameters also depend on the hardware on which the code is executed.

To cut a long story short, current production compilers are not able to fully optimize this class of codes and optimizing each application by hand is not practical. As a remedy, we propose a set of optimizations and describe how they can be applied automatically by a code generator for the domain of stencil codes. A combination of a space and time tiling is able to increase the data locality, which significantly reduces the memory-bandwidth requirements: a standard three-dimensional 7-point Jacobi stencil can be accelerated by a factor of 3. This optimization can target basically any stencil code, while others are more specialized. E. g., support for arbitrary linear data layout transformations is especially beneficial for colored kernels, such as a Red-Black Gauss-Seidel smoother. On the one hand, an optimized data layout for such kernels reduces the bandwidth requirements while, on the other hand, it simplifies an explicit vectorization.

Other noticeable optimizations described in detail are redundancy elimination techniques to eliminate common subexpressions both in a sequence of statements and across loop boundaries, arithmetic simplifications and normalizations, and the vectorization mentioned previously. In combination, these optimizations are able to increase the performance not only of the model problem given by Poisson's equation, but also of real-world applications: an optical flow simulation and the simulation of a non-isothermal and non-Newtonian fluid flow.

ACKNOWLEDGMENTS

This work would not have been possible without the support of many people, for which I am deeply grateful.

First of all, I would like to thank my supervisor Christian Lengauer for the possibility to participate in project ExaStencilsand to work at his chair for several years. I am grateful to him for the continuous support, the valuable suggestions on the structure of my publications and my work in general, and the improvements of my English skills.

I also want to thank Armin Größlinger for the many fruitful discussions and the constant help in all technical belongings. He always took the time to talk about unexpected results and findings, as well as the pros and cons of different design and implementation decisions, no matter how complex they were.

I also enjoyed to work with all participants of project ExaStencils. Communication in our project has always been fast and productive, despite the fact that we were not working at the same university.

It was a pleasure to share an office with Andreas Simbürger for several years. I have to thank him and all my other friends at the University of Passau for both the serious discussions about our work and several humorous conversations in between to raise our spirits. Special thanks goes to Eva Reichhart for her constant and valuable help in all administrative tasks.

Beside all support concerning this work directly, I want to thank my family and all my friends, who provided me the distance and regeneration necessary to finish this theses.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

LIST OF LISTINGS

LIST OF ACRONYMS

AoS array-of-structs

AST abstract syntax tree

CS common subexpression

CSE common subexpression elimination

DSL domain-specific language

GVN global value numbering

LSE linear system of equations

NUMA non-uniform memory access

PDE partial differential equation

RBGS Red-Black Gauss-Seidel

SCoP static control part

SoA struct-of-arrays

1 INTRODUCTION

In November 2018, the world's fastest supercomputer according to the TOP500 list[1] is the Summit of the Oak Ridge National Laboratory (ORNL). It provides a performance of 143.5 PetaFLOP/s on the High Performance Linpack benchmark. Given this number, exascale is only a factor of 7 away. The Summit consists of 4 608 nodes, each equipped with two 22-core Power9 processors and six NVIDIA Tesla V100 accelerators. The main memory is also heterogeneous with 512 GB DDR4 and 96 GB HBM2 memory per node. The latter HBM2 memory is located on the accelerators (16 GB each), but can be accessed by the processors, too.

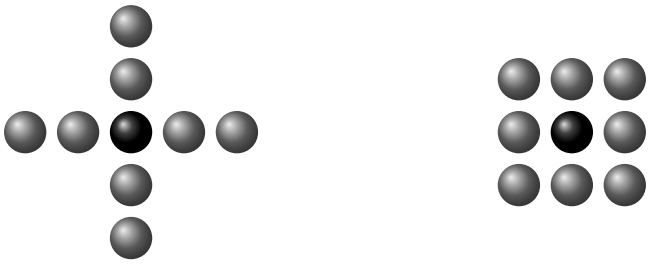
Overall, such high performance is easier to reach with a heterogeneous architecture and this trend is likely to continue. On the downside, this requires much more optimization effort for programs running on such a hardware and the traditional approaches become increasingly complex. These include the creation of highly specialized and optimized codes for specific problems running on a single machine on the one side, and the creation and maintenance of large libraries on the other side. One possibility to overcome this problem is to provide a domain-specific language (DSL) to the users and to generate an optimized target code automatically. Such a code generator can also optimize for the actual target hardware and, thus, performance portability is much easier to achieve. Supporting a new machine requires the identification of suitable optimizations and, in the worst case, the implementation of new optimizations in the code generator. Then, not only new applications but also all previously developed applications can be regenerated to benefit from these improvements.

In project ExaStencils[2], we have been working on a DSL and a code generator for the domain of stencil codes or, more specifically, the domain of multigrid methods [38, 90]. The restriction to such a narrow domain enables the automatic application of more specialized optimizations as presented later.

1.1 STENCIL CODES

Stencil codes are widely used in academia and industry alike, e. g., for solving systems arising from a discretization of partial differential equations (PDEs). The main characteristics of stencil codes is the

1 https://www.top500.org/lists/2018/11/
2 http://www.exastencils.org



(a) 2D 2nd-order **(b)** 2D 1st-order dense

Figure 1.1: 2D 9-point stencil shapes.

neighborhood relationship. It is utilized to recompute the values in a given grid. The fixed pattern specified by the neighborhood relationship is called a *stencil*. A commonly used, albeit ambiguous, naming scheme specifies the number of elements or points accessed. For example, the computation of a 2D 9-point stencil reads the center elements and either two points in all four directions or one point in each direction including the diagonals, as represented in Figure 1.1. A better alternative to name these stencil patterns is to specify the dimensionality, the radius, and whether it is dense, i. e., whether the diagonal elements are accessed. The remainder of this section presents and discusses different properties of stencil codes and their potential influence on the performance or necessary optimizations.

1.1.1 Dimensionality

In theory, stencil codes are not limited to any dimensionality and the construction of a stencil for an arbitrary dimension is straight-forward. The simplest variants of stencil codes are 1D stencils. Listing 1.1 shows an exemplary one-dimensional stencil code. Its data locality is already optimal, since only contiguous elements are read from the input array. There are some possibilities to tune 1D stencils but, since they are also not that relevant in most applications, we focus on higher-dimensional variants in this work.

2D stencils are based on the same neighborhood relationship as their 1D counterparts. However, since the address space of main memory is one-dimensional, the 2D field accesses have to be linearized and only the neighbors in one dimension are located beside each other, while the others are spaced farther apart. The distance between the two neighbors in the outer dimension is twice the extent of the inner dimension. For a field size of $32\,768^2$ elements, two lines fit

Listing 1.1: 1D stencil.

```
for (int i = 0; i < n; i++)
  out[i] = 0.8 * in[i] + 0.2 * (in[i-1] + in[i+1]);
```

easily in the processors fast on-chip cache, since they require only $2 \cdot 32\,768 \cdot 8\,\mathrm{B} = 512\,\mathrm{KB}$ for double-precision elements. Thus, data has to be loaded only once and resides in cache until its last access occurs. On the contrary, the distance between the two neighbors of the outermost dimension in a 3D stencil may already be too large for the processors cache: for as many elements as in the 2D example, i. e., for a field size of $1\,024^3$, two planes require $2 \cdot 1\,024^2 \cdot 8\,\mathrm{B} = 16\,\mathrm{MB}$. This trend continues for higher dimensionalities.

1.1.2 Radius

Another property of stencil codes that we will use is their *radius* or *order*. It specifies how many elements in each direction are accessed. A *1st-order* stencil, i. e., a stencil with radius 1, accesses only the direct neighbors in every direction. The stencil depicted in Figure 1.1(b) is a 1st-order stencil. Since the diagonal neighbors are read, too, it is the *dense* version of an ordinary 2D 1st-order stencil. Consequently, Figure 1.1(a) shows a 2nd-order stencil.

The radius has an impact on the performance and the optimizations. A larger radius increases the number of lines accessed in linear memory and, thus, the amount of data that has to be cached for later reuse to prevent a repeated load from main memory. In contrast, the dense versions do not alter the cache pressure but the number of computations. However, even for the dense versions, the arithmetic intensity, i. e., the number of computations performed relative to the amount of data accessed, is still low, which renders stencil codes memory-bandwidth bound.

1.1.3 Iteration Type

Stencil codes exhibit several types of iteration patterns.

JACOBI. The simplest one is the Jacobi iteration, which is presented in Listing 1.1. Its defining characteristics is that the write array and the (one or more) read arrays are strictly separated. There is no array that is both read and written. As a consequence, there are no data dependences in a single application of the stencil: all data elements can be computed in any order, even in parallel.

GAUSS-SEIDEL. The other extreme is the Gauss-Seidel iteration. In that type of stencil code the write array is also read. This leads to some neighbors having been updated before being read, and others not, as depicted in Figure 1.2. With x being the innermost dimension, the green points are already updated, while the red are still pending. The computation of the black point then accesses two green and two red neighbors. This obviously enforces some data dependences and

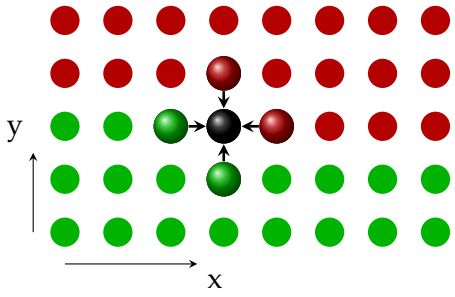


Figure 1.2: 2D grid during a Gauss-Seidel iteration.

it is no longer possible to update all data points in parallel. Actually, a naïve parallelization of a Gauss-Seidel iteration imposes several drawbacks, such as an unfavorable memory access pattern.

RED-BLACK GAUSS-SEIDEL. An algorithmic solution to the complex parallelization is to change the computation order. Instead of updating all elements in a lexicographic sequence, the set of computations is split in two: if only every other element is updated for a 1st-order stencil, the newly computed values are not accessed again, which allows half of the elements to be computed in parallel. The second half is then updated in a separate second sweep. Traditionally, the one half of the resulting checkerboard pattern gets the color red assigned, the others black and the whole iteration scheme is called Red-Black Gauss-Seidel (RBGS). Following this new iteration order, the red points only access the neighboring black points and vice versa, which effectively eliminates all data dependences inside a single sweep. Thus, each color can then be updated in parallel and only a single synchronization point is needed.

MULTI-COLOR GAUSS-SEIDEL. A splitting with two colors is only sufficient for a 1st-order stencil that is not dense. For larger stencils, more colors may be necessary. E. g., a dense 2D or 3D 1st-order stencil requires four, respectively eight colors.

1.1.4 Coefficient Type

Another variation point is the choice between constant and variable coefficients. In the former, the multiplicative weight of each neighbor accessed is a compile-time constant, i. e., a value that is independent of the location inside the grid. This enables some optimizations, such as an application of the distributive law to eliminate some multiplications if there are identical coefficients, as realized in Listing 1.1. In contrast, in the latter case, the values of the coefficients depend on the coordinate of the element to be computed. Such coefficients may either be computed explicitly during the stencil application, or they have

to be loaded from a separate memory location. The latter increases the memory and bandwidth requirements significantly since, for an n-point stencil, n different values per grid point must be stored in and loaded from main memory. However, if these coefficients are reused frequently, additional memory transfers may be faster than a repeated computation.

1.1.5 Boundary Handling

Another important aspect of stencil codes is their boundary handling. Updates of elements at the border of the grid requires special attention since not all neighbors that are specified by the stencil shape exist. Thus, either a specialized code for these computations must be executed, or one or more additional layers, the so-called *ghost layers*, have to be added. The latter obviously increases the memory footprint but it does not add any complexity to the stencil code and its control flow since every field element is updated in the exact same way. Depending on the actual problem, these ghost cells must be updated after each stencil application. However, because ghost cells are required anyway if a parallelization for distributed memory is desired, we also utilize them for boundary handling.

1.2 MULTIGRID

While stencil codes have many applications, ranging from image processing to simulations in physics and chemistry, project ExaStencils focuses on a narrower domain: the solution of PDEs via geometric multigrid solvers [52]. A multigrid solver consists of stencil computations on a hierarchy of grids with a different granularity [38, 90]. It combines two principles into an iterative solver: the *smoothing property* and the *coarse-grid principle*. The former means that, after very few steps of a classical iterative method such as a Jacobi or Gauss-Seidel iteration, the error—i. e., the difference of the actual solution to its current approximation—is smooth. Informally, smoothness is a graphical property of a grid and said methods are very ineffective in reducing the remaining smooth error. This is where the coarse-grid principle comes into play. It states that a smooth grid can be approximated sufficiently on a coarser grid with fewer discretization points.

A complete simple multigrid solver, a so-called *V-cycle*, is given in Algorithm 1.1. In pre-smoothing, the high-frequency components of the error are eliminated via ν_1 steps of a classical iterative method. Then, an approximation of the smooth error, the residual, is computed and restricted to the next coarser grid. After a solution to the coarser problem has been computed recursively, the error is prolongated back to the finer grid and then eliminated. Finally, a second smoothing

ALGORITHM 1.1: Recursive V-cycle to solve $A_l u_l = f_l$.

1 **FUNCTION** $V_l(u_l^{(k)}, A_l, f_l, \nu_1, \nu_2)$**:**
2 **if** $l > 0$ **then**
3 $\bar{u}_l^{(k)} \leftarrow S_l^{\nu_1}(u_l^{(k)}, A_l, f_l)$ { pre-smoothing }
4 $r_l \leftarrow f_l - A_l \bar{u}_l^{(k)}$ { compute residual }
5 $r_{l-1} \leftarrow R r_l$ { restrict residual }
6 $e_{l-1} \leftarrow V_{l-1}(0, A_{l-1}, r_{l-1}, \nu_1, \nu_2)$ { recursive call }
7 $e_l \leftarrow P e_{l-1}$ { prolongate error }
8 $\tilde{u}_l^{(k)} \leftarrow \bar{u}_l^{(k)} + P e_l$ { coarse grid correction }
9 $u_l^{(k+1)} \leftarrow S_l^{\nu_2}(\tilde{u}_l^{(k)}, A_l, f_l)$ { post-smoothing }
10 **else**
11 $u_l^{(k+1)} \leftarrow A_l^{-1} f_l$ { solve directly }
12 **return** $u_l^{(k+1)}$

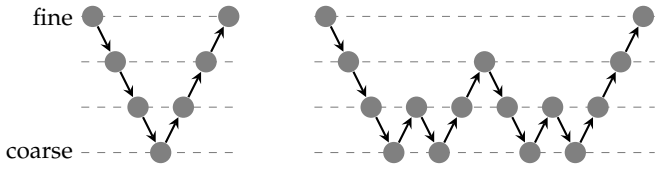
phase is conducted. The recursion terminates when the problem size is small enough to be solved directly.

A graphical representation of this method with the finer levels depicted on top of the coarser ones is shown in Figure 1.3(a). Other cycle types exist, such as a W-cycle, which is presented in Figure 1.3(b). It performs two instead of a single recursive call in line 6.

1.3 CONTRIBUTIONS

The automatic generation of a complete multigrid application is not an easy task, even if performance is neglected. In project ExaStencils, the performance of the generated code is important and, since the ExaStencils code generator has been developed from scratch, a variety of different optimization techniques is required. We make the following contributions.

- We propose a set of different code optimizations that both support the code generation process and increase the node-level performance of the generated code. The transformations described in detail are function inlining, arithmetic simplifications and nor-



(a) V-cycle **(b)** W-cycle

Figure 1.3: Different multigrid iterations.

malizations, two specialized redundancy elimination techniques of which one targets redundancies across loop boundaries, a vectorization, a space and time tiling to increase the data locality, and very versatile data layout transformations. Most of these techniques manipulate the abstract syntax tree (AST) of the generated code directly. In contrast, the time tiling to increase the data locality is based on the polyhedron model for loop transformation [29], which is a mathematical model to represent and manipulate loop nests.

- These optimizations have been implemented in the ExaStencils code generator and both their abstract idea and the concrete implementation are described in detail. This includes interactions of different techniques. Our goal is to support the implementation of this optimization capability in other code generators for similar and different domains.

- The optimizations were evaluated to demonstrate their effectiveness and impact on both the isolated smoother component of a multigrid solver and complete multigrid applications. The latter cover a well-known model problem based on Poisson's equation, an optical flow simulation, and a non-isothermal and non-Newtonian fluid flow simulation, i. e., a complex real-world application.

1.4 OUTLINE

The remainder of this thesis is structured as follows. Chapter 2 provides background information, namely details on the ExaStencils code generator and its DSL ExaSlang, as well as the polyhedron model for loop optimization. The latter is the foundation of our data locality optimizations. This is one of the techniques implemented in our code generator, which are described in detail in Chapter 3. Beside the main idea of the optimizations, their implementation is also outlined to support an integration and adaptation for other code generators and domains.

Specialized evaluations of the approaches presented are given in Chapter 4. Related work is discussed in Chapter 5, while Chapter 6 concludes and presents possible future directions.

More detail on the contents is given in the corresponding chapter's introduction.

2 BACKGROUND

This chapter provides background information on the polyhedron model for loop optimization [29] in Section 2.1 and project ExaStencils, especially its DSL and code generator, in Section 2.2.

2.1 POLYHEDRON MODEL

The source code of a loop nest can be transformed in many different ways, some of which are tiling, permutation, skewing, fusion, and distribution. Each of these transformations has the potential to increase performance. However, it is not easy to glean from the source code whether a transformation preserves the semantics and increases performance. Also the best transformation may be syntactically complex and dominated by others that are syntactically very simple. In a mathematical model, all transformations have equal complexity. The polyhedron model provides techniques and tools to perform a search across a space of all legal loop transformations. This section gives a brief overview of the polyhedron model.

The integer set library (isl) [92] is currently the most popular and advanced C library supporting the polyhedron model. Unless specified otherwise, we represent and manipulate polyhedra only with data structures and methods provided by this library.

2.1.1 Static Control Part

The polyhedron model is a quite powerful tool but it cannot be used to transform an arbitrary program. The classic model imposes some limitations to the structure of a target code. First of all, the code must have a statically analyzable control flow, i. e., the control flow must depend only on constants, constant variables, or loop iterators. Such a program region is called a static control part (SCoP). Furthermore, to keep the required computations decidable, the loop bounds, the loop stride, branch conditions, and all memory address computations, i. e., array subscripts, must be affine in the surrounding loop iterators and constant variables. Note that *constant variables* must not be constant in the whole application but only in the code region that should be transformed. Before or after such regions, updates of these variables are permitted.

Listing 2.1: Sample loop nest.

```
 for (int i = 1; i < n; i++)
   for (int j = 1; j < n; j++)
S:   A[i][j] = B[i][j] + 0.2 * (B[i-1][j] + B[i][j-1]);
 for (int i = 1; i < n; i++)
   for (int j = 1; j < n; j++)
T:   B[i][j] = A[i][j] + 0.2 * (A[i-1][j] + A[i][j-1]);
```

There have been some efforts to relax the limitations of the polyhedron model, such as permitting nonlinearity [36, 86] or including `while` loops [33]. However, these extensions have not been implemented in isl, and they are also not mandatory for the domain of stencil codes that we consider.

2.1.2 Iteration Domain

The basic element of a polyhedral representation is a *statement instance*, i. e., a single execution of a statement. A loop nest can then be viewed as a set of statement instances each of which is associated with a specific statement and specific value for each iteration variable of the surrounding loops. Given that the loop boundaries and potential conditionals are affine expressions, this set – the so-called *iteration domain* – forms a union of integer polyhedra.

Consider the loop nest in Listing 2.1. Its iteration domain can be written as follows:

```
[n] -> { S[i,j] : 1 <= i < n and 1 <= j < n;
         T[i,j] : 1 <= i < n and 1 <= j < n }
```

This notation follows the syntax of isl. The identifier list `[n]` at the beginning introduces the structural parameters of the loop nest. Structural parameters are unknown but constant values that typically correspond to the problem size. The actual polyhedron for statements `S` and `T` is specified inside the braces. Constraints must be in Presburger arithmetic.

2.1.3 Schedule

A complete specification of the loop nest requires not only the iteration domain but also the order in which its elements, the statement instances, are to be executed. This can be achieved via a schedule that assigns each instance to a point in a (possibly multi-dimensional) virtual time. The execution order can then be determined by sorting the elements of the iteration domain according to the lexicographic ordering of the associated points in time. The schedule of the loop nest in Listing 2.1 is

```
{ S[i,j] -> [0,i,j]; T[i,j] -> [1,i,j] }
```

which can alternatively be represented by the matrices

$$\Theta^S \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \\ n \\ 1 \end{pmatrix} \text{ and } \Theta^T \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \\ n \\ 1 \end{pmatrix}$$

The fist element is the constant `0` for **S** and `1` for **T**. Dimensions consisting of constant values only represent a textual order: all statement instances of **S** are executed before those of **T**. This results in two separate loop nests for **S** and **T**. The remaining two elements of both **S** and **T** are the values of the loop iterators, which means that both statements are surrounded by two loops. In this example, each statement instance is mapped to a unique point in three-dimensional time, i. e., the schedule is bijective.

We call a schedule that is bijective *complete*, otherwise *incomplete*. These terms refer to the uniqueness or ambiguity of a schedule. While a complete schedule specifies exactly in which order the statement instances are executed, a single incomplete schedule corresponds to multiple different execution orderings in a sequential environment. Even with parallelism in mind are these definitions sensible, since the number of execution units is usually less than the number of parallel statement instances by orders of magnitude and, thus, a further sequential ordering is required to prevent ambiguity.

2.1.4 Data Dependences

In addition to the iteration domain and the initial schedule, memory accesses are modeled. They are represented by a mapping from a statement instance to a memory location or array element. For example, the read and write accesses in the loop nest of Listing 2.1 are represented as follows:

```
reads: {
  S[i,j] -> B[i,j]; S[i,j] -> B[i-1,j]; S[i,j] -> B[i,j-1];
  T[i,j] -> A[i,j]; T[i,j] -> A[i-1,j]; T[i,j] -> A[i,j-1]
}
writes: { S[i,j] -> A[i,j]; T[i,j] -> B[i,j] }
```

If two statement instances access the same memory location and at least one modifies its contents, there exists a data dependence between both and the preservation of their order is a sufficient condition for a schedule to be legal. The condition may not be necessary since it may exclude some legal schedules as, e. g., in the case of five consecutive statements all of which access the same memory location and the first, third, and fifth modify its contents. The first two pairs of statements

can be interchanged, however, the classic polyhedron model prevents such a transformation. Similarly to the iteration domain, the dependences can be represented as a finite set of polyhedra, given that the memory accesses are affine expressions. If we look at the write access in **S** and one of the read accesses, e. g., `{ T[i,j] -> A[i-1,j] }`, subsequent iterations of the `i`-loop for the same value of `j` write to a common memory location. This results in the following dependences:

```
[n] -> { S[i,j] -> T[i+1,j] : 1 <= i < n-1 and 1 <= j < n }
```

Dependences according to the other pairs of reads and writes are:

```
[n] -> { S[i,j] -> T[i,j]   :   1 <= i < n and 1 <= j < n;
         S[i,j] -> T[i-1,j] : 2 <= i < n and 1 <= j < n;
         S[i,j] -> T[i,j-1] : 1 <= i < n and 2 <= j < n;
         S[i,j] -> T[i,j+1] : 1 <= i < n and 1 <= j < n-1 }
```

The constraints behind the colon specify the existence of the dependences, i. e., these include only dependences for which both the source and the target are actually executed.

The data dependences enter into constraints for a legal schedule [27, 28, 73]. For each data dependence instance, i. e., each instance of a dependence polyhedron, from $\vec{x}_S$ to $\vec{x}_T$, a legal schedule Θ must ensure a strict temporal order, i. e., an integer c between 1 and the dimensionality of the schedule such that

$$(\forall\, i < c : \Theta_i^S(\vec{x}_S) = \Theta_i^T(\vec{x}_T) \;\wedge\; \Theta_c^S(\vec{x}_S) < \Theta_c^T(\vec{x}_T))$$

Dimension c is said to *strongly satisfy* and, therefore, *carry* this dependence. If c corresponds to a loop in the target loop nest, the dependence is *loop-carried*, otherwise it is *text-carried*. For dimensions higher than c, the dependence is irrelevant.

2.1.5 Optimization

All in all, the optimization of a loop consists of the following steps:

(i) extract a polyhedral representation

(ii) compute the data dependences

(iii) find an optimal schedule

(iv) generate a target loop nest

There are libraries for extracting a polyhedral representation from a C-like source code, such as Clan [6] or pet [93]. But, if the optimization should be integrated into a compiler or code generator, an extractor based on an internal syntax tree could be more efficient. For the three remaining steps, isl provides suitable implementations of state-of-the-art algorithms.

Listing 2.2: Optimized loop nest.

```
#define S(i,j) A[i][j] = B[i][j] + 0.2*(B[i-1][j]+B[i][j-1])
#define T(i,j) B[i][j] = A[i][j] + 0.2*(A[i-1][j]+A[i][j-1])

for (int j = 1; j < n; j++)
  S(1, j);      // unrolled iteration i=1
for (int i = 2; i < n; i++) {
  S(i, 1);      // unrolled iteration j=1
  for (int j = 2; j < n; j++) {
    S(i, j);
    T(i-1, j-1);
  }
  T(i-1, n-1);  // unrolled iteration j=n
}
for (int j = 2; j <= n; j++)
  T(n-1, j-1);  // unrolled iteration i=n
```

The most interesting step is the third: the selection of an optimal schedule. The existing scheduling algorithms compute a suitable schedule by optimizing an affine objective function, such as the Feautrier scheduler [27, 28] that optimizes for parallelism or the PLuTo algorithm [8] that optimizes for both coarse-grain parallelism and data locality. The isl provides an implementation of the Feautrier scheduler and a variation of the PLuTo algorithm.

Concerning the example given in Listing 2.1, the Feautrier scheduler does not apply any transformation, since both loop nests can be executed in parallel. In contrast, the PLuTo algorithm computes the following schedule:

```
{ S[i,j] -> [i,j,0]; T[i,j] -> [i+1,j+1,1] }
```

The constant dimension is located innermost: both statements are surrounded by common loops, as evident in Listing 2.2. This reduces the amount of parallelism but increases the data locality which can result in an overall better performance.

2.2 EXASTENCILS

Project ExaStencils [52] is part of the priority program SPP 1648 "Software for Exascale Computing" (SPPEXA) funded by the German Research Foundation. We are developing a multi-layered DSL called ExaSlang (ExaStencils language) [81] for geometric multigrid solvers and a corresponding code generator that is able to produce automatically optimized target code [50] for a given execution platform. This

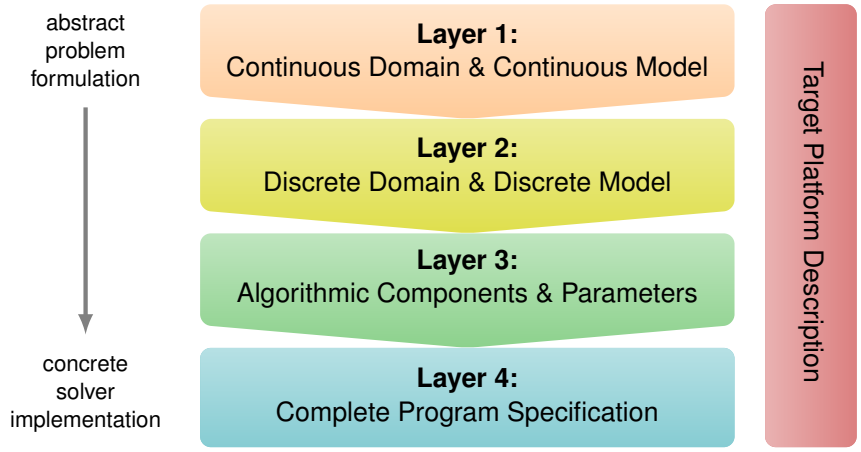


Figure 2.1: ExaSlang with its four layers of abstraction and a cross-cutting target platform description language.

section presents ExaSlang and a detailed overview of its most concrete layer, as well as the corresponding code generator.

2.2.1 ExaSlang

The input for our code generator is code in our DSL ExaSlang. It consists of four increasingly concrete layers of abstraction, ExaSlang 1 to 4, as shown in Figure 2.1. The most abstract layer is ExaSlang 1, in which the user specifies the continuous partial differential equation (PDE) to be solved, as well as related information, such as the computational domain. ExaSlang 2 allows a discretized formulation of the problem, while in ExaSlang 3 the multigrid method can be specified. The most concrete layer, ExaSlang 4, facilitates a complete specification of the final solver and further implementation details such as the parallelization, the communication pattern, and the data layout. A target platform description language orthogonal to ExaSlang 1 to 4, called ExaSlang Platform Description [79], is available, too.

All optimizations we discuss are applied either to ExaSlang 4 code or to a still more concrete internal representation of the code generator. To be able to follow requires more details on ExaSlang 4.

ExaSlang 4 is the most concrete form of ExaSlang, but it still contains language features that are abstractions of the multigrid domain. From ExaSlang 4 code, the target code is generated—normally in C++ plus MPI/OpenMP/CUDA or similar languages. Listing 2.3 shows ExaSlang 4 code for an RBGS smoother. `Solution` and `RHS` are representations of discretized variables, which are called *fields*. Similarly, `Laplace` is a representation of a discretized operator, such as a stencil or a stencil field.

Multiple copies of a field can be defined and used via *Slots*. This is useful, e. g., for the implementation of a Jacobi iteration, which needs

Listing 2.3: ExaSlang 4 code for a RBGS smoother.

```
Function Smoother@(all but coarsest) {
  color with {
    (i0+i1+i2) % 2,
    loop over Solution {
      Solution = Solution +
        1.0 / diag(Laplace) * (RHS - Laplace * Solution)
    }
  }
}
```

different memory locations for the input and output. Their usage is similar to C arrays but with three additional keywords: `active`, `previous`, and `next`. In conjunction with an `advance field` statement, these simplify the handling of different slots. For example, let's assume the `active` field contains the most up-to-date data which is the input for a Jacobi stencil. Then, the newly computed values are written to the `next` field and a call to `advance` updates `active`, `previous`, and `next` accordingly.

ExaSlang 4 allows most objects to be available at several levels of the multigrid hierarchy. Such level specifications and references are tied via the `@` operator to the objects to which they belong: function `Smoother` is available at all levels but the coarsest. The fields `Solution` and `RHS`, as well as the stencil `Laplace` in the function's body, are also level-specific. If, as here, no level is specified explicitly, the identifiers reference the objects at the level at which the function is being applied. This can be made explicit by adding the optional label `@current`. Other than `current`, the keywords `coarser` and `finer` reference objects at the next coarser or finer level and the addition or subtraction of a constant offset, such as `current-2`, is also possible. Additionally, there are `coarsest` and `finest`.

A **`color with`** block starts with a non-empty sequence of modulo operations. Their numerators are expressions that may contain field iterators (`i0`, `i1`, ...) while their divisors must be natural numbers. These expressions specify a (possibly multi-dimensional) coloring that is applied to the **`loop over`** statements. In general, the **`loop over`** construct specifies a full iteration over a (multi-dimensional) field. However, the statements inside the **`color with`** block are executed once for each color and, hence, the **`loop over`** statements are restricted to the iterations across the current color. In the example, the body of the loop is executed first for all steps for which the sum of the loop iterators `i0`, `i1`, and `i2` is even. In a second sweep, the remaining, odd steps are executed.

2.2.2 ExaStencils Code Generator

The ExaStencils code generator is based on the code generation framework Athariac [80], which has been developed as part of project ExaStencils. It is implemented in the programming language Scala [67], which is both a functional and imperative, object-oriented language running on the Java virtual machine. Athariac provides data structures and skeletons to support the development of code generation and transformation techniques.

Data Structures

The domain-specific code provided to the code generator is internally represented by an abstract syntax tree (AST). Each ExaSlang layer has its own node types, which are subclasses of a common `Node` class. This common superclass allows the framework to iterate over the whole AST and also provides a way to attach additional information to any node via an annotation mechanism. Such annotations are useful when some information specific to a part of the AST should be preserved for a later transformation or optimization. In addition to the four ExaSlang layers, we already mentioned the more concrete intermediate representation (IR), which is similar to C and is available internally. Most optimizations are applied to IR code shortly before it is pretty-printed to C++. It contains both abstract nodes representing ExaSlang 4 features, such as a specific node for a multi-dimensional **`loop over`** statement, but also nodes more closely related to C and C++, like one for a `for` loop.

Transformations

The main workhorses of the Athariac framework and the ExaStencils code generator are transformations. They are used for term rewriting either to generate new computations, or to optimize existing ones by replacing them with more efficient versions. Transformations take advantage of Scala's principle of deep pattern matching that enables the convenient search and modification of an arbitrarily large subtree as illustrated in Listing 2.4. A new transformation takes a short description and a `PartialFunction` that performs the desired modification. The description is mainly used for debugging, since it is written to the generation log. As presented, Scala provides a special syntax for a `PartialFunction`. Between the keyword **`case`** and an arrow (**`=>`**) an arbitrary pattern can be specified, as well as an optional condition. In the example, the patterns are an `IR_Subtraction` node containing either two `IR_FloatConstants` or two `IR_IntegerConstants`. If any matches, a constant with the desired value is returned, which replaces the original subtraction. The search for and replacement of any matched subtree in the complete AST is performed by the Athariac framework. Optional flags provided at the creation of a transformation

Listing 2.4: Example of a simple transformation in the ExaStencils code generator.

```
new Transformation("evaluate constant subtraction", {
  case IR_Subtraction(IR_FloatConstant(l),
                      IR_FloatConstant(r)) =>
    IR_FloatConstant(l-r)
  case IR_Subtraction(IR_IntegerConstant(l),
                      IR_IntegerConstant(r)) =>
    IR_IntegerConstant(l-r)
})
```

specify whether it should be restricted to a given subtree or applied recursively, i. e., whether the result of the transformation should be searched for matching parts, too.

Collectors

Other objects that extract and process information when traversing the AST are collectors. In contrast to transformations, these are not intended to modify the AST. Therefore, an arbitrary number of collectors can be registered to be executed in conjunction with a single transformation. A collector cannot be run in isolation, but Scala provides a partial function with an empty domain that can be used in a dummy transformation. When implementing a collector, one may provide code to be executed when a node is entered and also when it is exited in a depth-first traversal. This means that the `enter(Node)` method is called for a specific node before its subtree is traversed and the corresponding `leave(Node)` is executed after its children have been accessed. This allows the creation of, e. g., an ancestor stack, which contains all nodes of the path from the current one up to the root.

Strategies

Transformations gain their power from being composed to strategies. On the one hand, the simplest strategy is the `DefaultStrategy`, which unconditionally applies a sequence of transformations in a fixed order. It is easy to use and sufficient if only simple replacements are necessary. On the other hand, the more flexible `CustomStrategy` does not impose any fixed pattern on how transformations are applied. In fact, a `CustomStrategy` is an abstract class that provides an API to its derivatives for the direct handling of transformations and collectors. Subclasses are supposed to provide an implementation of a generic `apply()` method, which orchestrates all necessary transformations. This way, each strategy naturally encapsulates a single task.

3 OPTIMIZATIONS

The performance of a naïvely generated application in the domain of stencil codes is far from optimal. Even though the code structure of the generated application is fairly simple, the target production compilers are not fully able to optimize the code by themselves. Thus, several standard and advanced techniques have been implemented in the ExaStencils code generator to reduce the run time of the emitted application. A detailed description of the optimizations and their implementation is presented in this chapter. Even though some of these optimizations are also integrated in production compilers, it is not always possible to select the best compiler. One prominent example is the automatic vectorization performed by the Intel C/C++ compiler. While it is clearly superior compared to the vectorizing capabilities of the GNU compiler, the Intel compiler is not always available, e. g., due to architectural or license limitations. This is why we chose to implement fairly standard techniques in our code generator, too.

This chapter is organized as follows. A function inlining, arithmetic normalizations, and address precalculation—three rather supplementary optimizations—are described in Sections 3.1 to 3.3. Techniques to eliminate redundancies both in a sequence of statements and between subsequent loop iterations are detailed in Section 3.4. These optimizations heavily rely on the first two techniques, the function inlining and the arithmetic normalizations. Section 3.5 presents the vectorization capabilities integrated in the ExaStencils code generator. Standard polyhedral techniques and a polyhedral search space exploration to increase data locality are illustrated in Sections 3.6 and 3.7, while Section 3.8 describes very versatile data layout transformations in detail.

3.1 FUNCTION INLINING

One of the basic optimizations implemented is a function inlining. It replaces a call to a small function by the computations specified in the function body. For frequently invoked functions, an inlining can eliminate the function call overhead and, thus, improve performance. An inlining may also be beneficial for other optimizations for which function invocations are impenetrable barriers. For example, arithmetic simplifications cannot be applied across a function boundary. Additionally, the user can be encouraged to write small helper func-

tions wherever appropriate to achieve a modular and maintainable DSL code. The same holds for the code generation phases of our generator.

Most production compilers are capable of inlining if both caller and callee reside in the same compile module, i. e., source file. However, our code generator creates a separate file for each generated function, which effectively hampers a target compiler's inlining capability except when link-time optimizations are available and enabled.

The inlining strategy consists of three phases: a code analysis, the selection of functions to be inlined, and the actual inlining. The analysis is conducted once, while the selection of a function and its immediate inlining are repeated until all inline candidates have been processed.

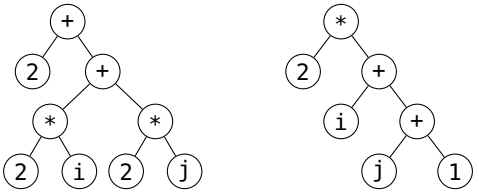
Code Analysis

Initially, an analysis of the whole abstract syntax tree (AST) is performed by a specialized collector. It creates a call graph and gathers a list of top-level variable declarations per function. Since our inlining strategy merges the scopes of the callee and the caller, these variables must be renamed if a name conflict occurs. Declarations in a nested scope must not be tracked, since such a name conflict results in a variable shadowing, which is not considered an error in C/C++. The analysis collector also creates a set of candidate functions to be inlined and removes those with more than one return path. This limitation simplifies the function inlining strategy: in case of multiple return paths, one must not only introduce a new temporary variable to store the originally returned value, but a goto may also be required to reconstruct the original control flow.

Function Selection

The set of candidate functions is refined further based on the function size, which is measured heuristically by the number of statement nodes in its body. The default threshold is 10, i. e., only functions that contain at most 10 statement nodes are considered for inlining. This threshold can be modified by the user if a more aggressive inlining is desired or if it should be disabled altogether. A very large value, for example, results in an inlining of almost an entire multigrid cycle into a single method. Next, functions that do not contain any call to an inline candidate (including the callee itself) are selected and inlined. These can be determined via the call graph, which is, in turn, updated after every step.

This order of events has several benefits. First, the only necessary modification to the call graph is to remove edges, since an inlined function does not have any (relevant) outgoing edges that must be copied. Second, it minimizes the number of inlining steps. Third, it



(a) AST for x **(b)** AST for y

Figure 3.1: ASTs for semantically equivalent expressions.

guarantees termination even if there is an indirect recursion consisting solely of inline candidates: each one contains at least one call and, thus, none is inlined.

Function Inlining

In the third phase, the actual inlining, the following is performed once for every call expression. The function body of the callee is duplicated and variables whose names are already assigned in the callers scope are renamed. For each function argument, a declaration is generated and prepended to the duplicated body. Their variables are initialized with the expressions in the function call arguments. To complete the preparation, a potential return statement is removed and the statements are inserted directly before the one that contains the call expression. The function call itself is eventually replaced by the return expression, if available; otherwise it is removed completely. An additional scope for the former method body is explicitly not created to keep the resulting code simple and to support later optimizations.

3.2 ARITHMETIC NORMALIZATIONS

Very versatile optimizations are arithmetic normalizations. They can be useful either to simplify complex computations or to improve other optimizations, e. g., to allow a redundancy elimination to detect larger redundancies. Let us take a closer look at the latter case. Since all transformations of the ExaStencils code generator are performed on an AST, the detection of redundant computations can be complex, as indicated by the ASTs in Figure 3.1 for the following two assignments:

```
x = 2 + 2*i + 2*j;
y = 2*(i + j + 1);
```

In this example, the expressions for x and y are semantically equivalent, but the corresponding ASTs are completely different. Even if the multiplication by 2 is factored out of the computation in the first line, the ASTs do not match because of the different order of the operands in the summation.

3.2.1 Commutativity and Associativity Law

For a binary addition, it is not sufficient to simply permute the children of each node, which exploits the commutativity law, but a more advanced restructuring analogously to the associativity law is required, too. To deal with this, the code generator has been equipped with a more general summation node with an arbitrary number of summands, of which a binary addition is a special case. A transformation to merge several nested additions into a single summation and to sort the summands (according to an arbitrary total ordering) is now straight-forward. The same holds for multiplications, while subtraction and division nodes must be handled with care and remain binary, since they are neither commutative nor associative. However, a subtraction can be transformed to a combination of an addition and a negation, which allows further normalizations of nested operations. Note that one must take care when dealing with matrices and vectors, as the commutativity of the multiplication only holds for scalar values.

3.2.2 Distributivity Law

A normalization according to the distributivity law is a bit more complex. The heuristics implemented focuses mainly on affine computations, which is sufficient for the computationally intense parts of the generated stencil codes.

Affine Expressions

In a first step, the AST of the input expression is analyzed bottom-up. The traversal of the expression's syntax tree proceeds explicitly via a recursive function that matches all expression nodes. Since every node that belongs to the expression has to be processed individually, a direct implementation is easier than a version that uses the generator's traversal capabilities. The latter are designed to ease the detection and manipulation of few specific nodes rather than processing them all. This analysis makes use of a special representation of a given affine expression, namely a key-value mapping: $\{k_1 : v_1, ..., k_n : v_n\}$ represents the sum expression $\sum_{i=1}^{n} v_i \cdot k_i$. The keys k_i are memory accesses and the associated constant values v_i their coefficients. For each node visited, the mapping is generated either directly in case of a constant or a memory access, or by merging the mappings of their children in an appropriate way. Such a merging is straight-forward if only affine expressions are considered.

For example, the extraction for the expression `2*(2*i+j)-(k+j+3*k)` is illustrated in Figure 3.2. Each node in the presented AST is annotated with the variable-constant mapping that represents the corresponding subtree. The multiplication `2*i` in the lower left requires the mappings $\{1 : 2\}$ and $\{i : 1\}$ to be merged. Since only a single factor

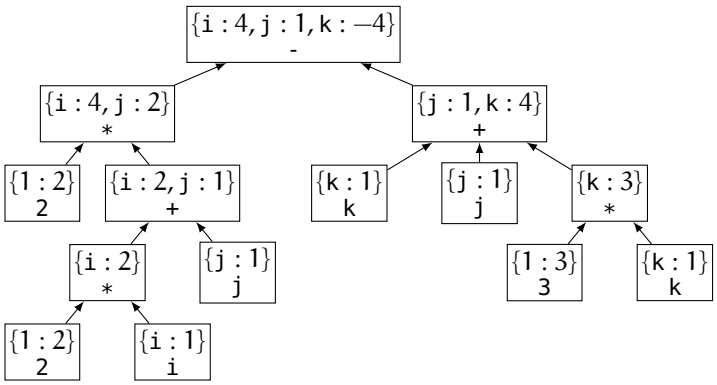


Figure 3.2: Bottom-up extraction of a sum mapping for the expression `2*(2*i+j)-(k+j+3*k)`.

contains variables, the result is its mapping in which all coefficients are scaled by the values of the constant mapping: $\{\mathtt{i} : 2\}$. Adding this to `j`, whose representation is $\{\mathtt{j} : 1\}$, creates a new mapping that contains all elements of both predecessors. If the same variable is present in multiple mappings, the respective coefficients are added. Subtractions are handled analogously with the coefficients of the right mapping being negated first. A division by a constant is treated as a multiplication with the denominator's inverse. The final mapping for the whole representation is the one at the root node: $\{\mathtt{i} : 4, \mathtt{j} : 1, \mathtt{k} : -4\}$.

AST Recreation

Such a mapping is already a normalized representation of an affine expression, so the only step remaining is the recreation of an AST. In general, one could simply generate one multiplication node per map entry and an addition to combine them. This results in the suboptimal expression `4*i + 1*j + (-4)*k` for the previous example. A better approach is to collect all variables with the same absolute coefficients and create only a single multiplication for such a group. This is incorporated by the usage of an inverse mapping, namely a mapping from the coefficient to a list of the corresponding variables. Further, the two variable lists for a positive coefficient and its negative counterpart can be combined via a subtraction before one common multiplication is generated. If one also refrains from generating a multiplication with 1 (or 0), the normalization and simplification of the upper example results in `4*(i-k) + j`.

Non-Affine Expressions

This technique becomes more complicated if non-affine expressions appear. The problem arises when two or more mappings should be merged according to a multiplication node and there are two or more non-constant mappings $m_1 ... m_n$. As a remedy, the factors of

such a multiplication are simplified in isolation and the resulting multiplication itself is treated like a normal variable access.

To start with, the coefficients of each mapping m_i are scaled down by their greatest common divisor d_i and an AST t_i is created. The new mapping for the non-affine expression then contains a single entry. Its key is a new multiplication node whose factors are the normalized expressions $t_1 ... t_n$. The coefficient for the multiplication is the product $\prod_{i=1}^{n} d_i \cdot \prod_{i=1}^{m} c_i$, with $c_1 ... c_m$ being the values of the constant factors in the original multiplication. Let us take a closer look at the expression `(3*i+6*j) * (k+k) * 5`. The mapping for the two non-constant factors are $\{$`i : 3, j : 6`$\}$ and $\{$`k : 2`$\}$. The greatest common divisors of the coefficients for both mappings, namely 3 and 2, are factored out and the new multiplication node is computed: `(i+2*j)*k`. Thus, the mapping that represents the original multiplication is $\{$`(i+2*j)*k : 30`$\}$. Note that we do not expand the non-affine multiplication since this may have a negative impact on the performance of the generated code.

There are other problematic computations, too, such as division or modulo computations with non-constant denominators. However, these did not appear in performance-critical parts and a special treatment has not been implemented yet.

Further Optimizations

The extraction process of a sum mapping provides several opportunities for different optimizations. In case of a division by a constant, a partial evaluation may be possible. For example, the floating-point computation `(2*i+j)/2` can be simplified to `i + 0.5*j` to get rid of the division. If the input is part of an integer computation, the simplified version `i + j/2` is only allowed if the result of the division is rounded towards negative infinity. A normal integer division in the generated C/C++ code truncates the result, which means that the rounding direction differs for positive and negative values. Therefore, the simplified version is only used if the code generator can prove that the sign of the numerator will not change. Such an optimization is mainly useful to simplify the memory address computation inside a loop nest. To render the required proof possible, under- and overapproximations of the lower, respectively upper loop bounds are computed in preparation.

Other optimizations are an elimination of duplicate expressions in a minimum or maximum computation, or an expansion of a power computation with a small, natural exponent. These are straight-forward and will not be discussed in detail.

3.3 ADDRESS PRECALCULATION

A fairly standard technique is the precalculation of memory addresses in nested loops [2]. Since the virtual address space is organized one-dimensionally and data structures usually represent higher-dimensional fields, a linearization of multi-dimensional accesses is necessary. Our code generator handles multi-dimensional array accesses as long as it is beneficial for the implemented optimizations, such as polyhedral techniques (Section 3.6) or data layout transformations (Section 3.8). But all accesses are eventually linearized, which reveals redundant computations and justifies this optimization. For example, a linearization of the access `a[z][y][x]` in an array with an extent of 512 elements per dimension yields `a[262144*z + 512*y + x]`. If `x` is the iterator of the innermost loop and neither `y` nor `z` are modified in the innermost loop body, the subexpression `262144*z + 512*y` need not be computed over and over again. An evaluation once prior to the `x`-loop is sufficient. Additionally, multiple accesses of neighboring elements of the same field share the same subexpression and can be optimized in conjunction. Production compilers are in theory also able to eliminate some of these redundancies. However, in conjunction with other transformations, such as a vectorization, the generated code may become too complex and the redundancies remain. Consequently, we implemented such an optimization directly to ensure it is always applied.

Collect and Analyze Accesses

The main task of this strategy is to identify loop-independent subexpressions of the array index computations. Such an analysis is performed by a specialized collector. It searches for suitable loops and collects all array accesses as well as variables that are modified or declared in the loop body or header. The latter must stay inside the loop and only subexpressions that do not contain any of these variables can be moved outside. After a loop has been traversed entirely and all array accesses have been collected, their index computations are analyzed. For each one, a sum mapping, as described in Section 3.2, is derived. Its summands are partitioned into those that can be precomputed and those that must stay inside the loop. Even though a constant summand can be added to the former group, we refrain from doing so. The reason is illustrated by the following example. A simple stencil computation accesses the center element and its direct neighbors, as depicted in Listing 3.1. Applying the described partitioning, the summands that should be precomputed are identical for all accesses, namely $\{z : 262\,144, y : 512\}$, which results in a single new base pointer for all accesses, as shown in Listing 3.2. This would not be the case if the constant summand were part of the new pointer, since it differs for all accesses.

Listing 3.1: Stencil code after array access linearization.

```
for (int z = 1; z < 511; z++)
 for (int y = 1; y < 511; y++)
  for (int x = 1; x < 511; x++)
   a[262144*z+512*y+x] +=
    .2*(a[262144*(z+1)+512*y+x] + a[262144*(z-1)+512*y+x]
      + a[262144*z+512*(y+1)+x] + a[262144*z+512*(y-1)+x]
      + a[262144*z+512*y+(x+1)] + a[262144*z+512*y+(x-1)]);
```

Integrate Changes

A separate transformation is required to incorporate the changes, because all variables written in the loop body must be collected before the redundant subexpressions can be determined. However, the preceding collector already prepares both new declarations and array accesses. The only part remaining is to prepend these declarations to the corresponding loop and replace the array accesses.

3.4 REDUNDANCY ELIMINATION

Eliminating redundant computations is a very obvious way to improve performance. There are several different situations in which redundant computations may appear. One has been illustrated in the previous section, namely a precalculation of memory address computations. The general redundancy elimination described in this section focuses on the actual computations of the generated kernels and addresses redundancies both inside a single loop iteration and between loop iterations. The latter is especially useful in the context of finite volume discretizations.

A general common subexpression elimination (CSE) [16] is frequently implemented in production compilers [2]. The basic idea is to remove repeated computations from expressions by reusing the result of the first computation. It is easy to see that this optimization can only be applied if none of the associated variables or memory regions are modified between the repeated evaluations of subexpres-

Listing 3.2: Optimized version of Listing 3.1.

```
for (int z = 1; z < 511; z++)
 for (int y = 1; y < 511; y++) {
  double *a_p = &a[262144*z+512*y];
  for (int x = 1; x < 511; x++)
   a_p[x] += .2*(a_p[x+262144] + a_p[x+512]+ a_p[x+1]
               + a_p[x-262144] + a_p[x-512]+ a_p[x-1]);
 }
```

Listing 3.3: Example of a textual CSE.

(a) input code

```
x = 2*i / j + 2*i;
x = x * 2*i;
```

(b) optimized code

```
cs = 2*i;
x = cs / j + cs;
x = x * cs;
```

sions. The drawback is that CSE potentially increases the register pressure since additional values must be preserved, which may lead to register spilling. But, in this case, the assumption is that, for larger expressions, the newly introduced memory access operations are faster than a recomputation of the expression.

3.4.1 Approaches to Common Subexpression Elimination

There are different approaches to the detection of common subexpressions (CSs). Let us introduce them in turn.

Syntactic CSE

The classic CSE searches for textual redundancies, introduces a new temporary variable that holds the value of the CS, and replaces each occurrence by an access to the new variable. Listing 3.3(a) shows a simple code snippet that contains the CS `2 * i` three times. Listing 3.3(b) shows an optimized version.

However, since redundant expressions are searched in the text, some optimization opportunities are missed. For example, there are two pairs of CSs in Listing 3.4(a). The first, `2 * i`, can be detected easily, but the other, `5 + x` respectively `5 + y`, varies in the last variable name and is therefore not detected, even though both variables have the same value. One can overcome this limitation by a repeated copy propagation and textual CSE until a fixed point is reached: first, a CSE possibly introduces name aliases, which are resolved by a copy propagation; second, since this may reveal new redundant expressions, a CSE must be reapplied. A dedicated detection of both pairs of subexpressions in this example requires a semantic equivalence test, which is provided by, e. g., global value numbering (GVN).

Listing 3.4: Example of a semantic CSE based on GVN.

(a) input code

```
x = 2 * i;
y = 2 * i;
a = 5 + x;
b = 5 + y;
```

(b) optimized code

```
x = 2 * i;
y = x;
a = 5 + x;
b = a;
```

Listing 3.5: Example in which CSE is able to remove a redundancy not recognized by GVN.

(a) input code

```
if (i > 0) {
  x = 2 * i;
  y = x * x * x;
} else {
  x = -(2 * i);
  y = x * x * x;
}
w = x * x * x;
```

(b) optimized code

```
if (i > 0) {
  x = 2 * i;
  y = x * x * x;
} else {
  x = -(2 * i);
  y = x * x * x;
}
w = y;
```

Global Value Numbering

GVN [15, 17] is an analysis based on the static single-assignment form of a program, which means that each variable is assigned exactly once. The first step of a GVN is to assign a so-called *value number* to each variable such that two variables have the same value number iff their semantic equivalence can be proved. An optimal number mapping for the example in Listing 3.4(a) would be [`i` $\rightarrow$ 1, `x` $\rightarrow$ 2, `y` $\rightarrow$ 2, `a` $\rightarrow$ 3, `b` $\rightarrow$ 3]. According to this mapping, `x` and `y`, as well as `a` and `b`, are equal, which leads to the optimized code of Listing 3.4(b).

There are cases in which GVN is not able to identify a redundant computation that can be eliminated by a textual CSE. For example, the CS `x*x*x` in Listing 3.5(a) can be detected easily by a syntactic CSE, while the value numbers of `x` in both branches must be different, since their values may differ in sign if `i` is less than or equal to `0`. This propagates to `y` and, thus, `w` cannot be statically identified with any of both.

Loop-Carried Redundancies

Another opportunity for optimization arises from CSs between subsequent iterations of a surrounding loop as depicted in Listing 3.6. The expression `exp(0.5*i + 0.25)` in iteration `i-1` evaluates to the same value as `exp(0.5*i - 0.25)` in the next iteration `i`:

```
exp(0.5*(i-1) + 0.25) ==
exp(0.5*i-0.5 + 0.25) ==
exp(0.5*i - 0.25)
```

Thus, the former can be reused. This incurs a higher detection effort, since some arithmetic conversions and simplifications are necessary due to the changing value of the loop iterator. Also, the optimization shown in Listing 3.6 requires that function `exp` is free of side-effects, so the analysis must be aware of this. One should further ensure that the CSs in different loop iterations do not overlap, i. e., do not share a

Listing 3.6: Example of a loop-carried redundancy elimination.

(a) input code

```
for (int i=0; i<n; i++) {
  A[i] = 4.2
    + exp(0.5*i - 0.25)
    + exp(0.5*i + 0.25);
}
```

(b) optimized code

```
lcs = exp(0.5*0 - 0.25);
for (int i=0; i<n; i++) {
  tcs  = exp(0.5*i + 0.25);
  A[i] = 4.2 + lcs + tcs;
  lcs  = tcs;
}
```

part of the input code. For example, the redundant expression found above could be extended to $\alpha = 4.2 + \texttt{exp(0.5*i + 0.25)}$ in iteration `i-1` and $\beta = 4.2 + \texttt{exp(0.5*i - 0.25)}$ in iteration `i`. But, since the summand `4.2` is now part of both expressions, the value of `A[i]` is $\alpha + \beta$ `- 4.2`. The additional subtraction of the shared summand `4.2` increases the complexity of the optimized code unnecessarily while reducing its benefit.

The idea of such a loop-carried CSE is not restricted to a single encasing loop but, for multiple outer loops, a separate value for each iteration of all inner loops must be remembered, which leads to a significant increase in memory consumption. Another obstacle is that the reuse of data from the previous iteration effectively sequentializes a loop or, at least, requires a special treatment for a parallel execution.

3.4.2 Preliminary Transformations

The ExaStencils code generator supports two types of redundancy elimination described in the previous subsection: a syntactic, AST-based and a loop-carried version. In order to facilitate the removal of as many and as large redundant computations as possible, a number of preliminary transformations are required. The redundancy eliminations themselves and all preparations, except for a classic, global inlining, are integrated in and orchestrated by a single strategy that makes extensive use of Athariac's capabilities to search, replace, annotate, and inspect nodes via a large number of transformations.

Global Inlining

To start with, two special inlining transformations, one global and one local, are executed. The former, which is introduced in Section 3.1, is a self-contained optimization strategy and is executed prior to the redundancy elimination by the code generator. It leads to a better starting position, since multiple calls of arbitrary functions cannot be merged in general. Inlining the body of pure functions simplifies the redundancy detection. It is then only required to recognize and deal

with calls to the standard C math library. All other function calls are rejected, i. e., they are not considered as CSs.

Local Inlining

A subsequent local inlining removes constant local variables, i. e., variables that are assigned exactly once, namely in their definition. This obviously introduces redundant computations, since every read of such a variable is replaced by the same expression. But it allows arithmetic optimizations and simplifications of the combined expressions, and the CSE applied later can potentially also detect larger redundant computations.

Arithmetic Normalizations

Arithmetic optimizations, which are detailed in Section 3.2, are the last step of the preprocessing. They are a crucial part of a syntactic redundancy elimination, as they also aim turn a semantic into a syntactic equivalence.

3.4.3 Syntactic CSE

Although the loop-carried CSE described in the next subsection is applied first, it is based in both concepts and techniques on the syntactic redundancy elimination, which justifies addressing the latter first. Section 3.4.1 introduced two different approaches to CSE, both with their own advantages and disadvantages. A value numbering would be the only transformation in the code generator that requires a static single-assignment form of the code. Therefore, a syntactic, AST-based redundancy detection has been implemented. In combination with the two inlining steps performed beforehand, most of the restrictions of this approach do not pose a hindrance. For example, both expressions for `a` and `b` from Listing 3.4(a) read `5 + 2*i` after `x` and `y` have been inlined, which can now be optimized by any approach.

Detection

The detection follows the idea that a larger CS consists solely of smaller CSs. Thus, it begins with a search of variable accesses, array accesses, and constants in the input AST, which are the smallest redundant expressions. Each instance found more than once is added to the initial set of CSs along with its ancestor stack. Note that smaller subexpressions, namely the array subscripts, are not analyzed here, since they are subject to the more specialized address precalculation introduced in Section 3.3, which is executed beforehand.

Starting with the initial list, larger CSs are detected inductively as specified in Algorithm 3.1. Each iteration of the while loop tries

ALGORITHM 3.1: Find larger CSs based on an input set of smaller ones.

INPUT: set of initial, small common subexpressions CSs
OUTPUT: set of all common subexpressions

```
1  FUNCTION Find_CSs(CSs):
2      newCSs ← CSs
3      while newCSs ≠ {} do
4          prevCSs ← newCSs
5          newCSs ← {}
6          foreach expr ∈ locations(prevCSs) do
7              parent ← parent(expr)
8              if parent is sum or product then
9                  foreach parent′ ∈ powerset_children(parent) do
10                     if children(parent′) ⊆ CSs then
11                         newCSs ← newCSs ∪ {parent′}
12             else if children(parent) ⊆ CSs then
13                 newCSs ← newCSs ∪ {parent}
14         foreach cs ∈ newCSs do
15             if | locations(cs) | = 1 then
16                 newCSs ← newCSs \ {cs}
17         CSs ← CSs ∪ newCSs
18     return CSs
```

to identify larger redundant expressions based on the results of the previous iteration. Function `locations` takes one or more CSs and returns a set of all locations in which these subexpressions occur. Thus, the loop starting in line 6 iterates over all locations in which any of the previously found CSs occur and tests whether the encasing expression (retrieved in line 7) is a potential CS, too. Since the code generator uses generalized sum and product nodes with an arbitrary number of arguments, these have to be treated specially as in lines 8 to 11. The function `powerset_children` is used to create new nodes with all possible subsets of the children of a given node. These nodes are also tested, since any combination of summands or factors can be computed repeatedly. This test, as evident twice in lines 10 and 12, simply evaluates whether all direct subexpressions, i. e., the children in the AST, are previously detected CSs. In lines 14 to 16, those CSs that occurred only once are removed again. Line 17 updates the set of all detected CS, which is returned in line 18.

Elimination

Finally, after all CSs have been identified, declarations of a new variable for each one can be inserted at the beginning of the given code

block and the old expressions are replaced by accesses to these new variables. Both tasks are straight-forward in the Athariac framework and, thus, are not discussed further.

Analysis

An analysis of the detection process reveals that each node of the input tree is added at most once to the list of potential CSs, namely as the parent of its child with largest depth. Therefore, in each step, the depth of the newly detected trees, which represent the new CSs, increases by exactly 1, so the depth of the input AST is an upper bound for the number of steps required.

After the first CS has been removed, one could either update the set of the remaining ones carefully to choose how to continue, or simply restart the whole analysis. Due to its simplicity and low performance impact, the code generator currently restarts the detection phase from scratch after each removed redundancy until either no new CS is found, or the largest one becomes too small to be profitable.

3.4.4 Loop-Carried CSE

The ExaStencils code generator additionally supports a loop-carried version of the CSE described in Section 3.4.3, which is executed first. The basic idea is to detect and eliminate redundant expressions not only in a text sequence of statements, but also between statement instances of subsequent loop iterations, as described in Section 3.4.1.

Detection

Before the actual redundancy detection is started, each node of the AST gets its own unique integer identifier assigned as a preparation for a later overlap test.

For the detection of redundancies between neighboring loop iterations, the body is duplicated and each occurrence of the loop iterator `i` is replaced by the expression `i - str(i)`, where `str` denotes the stride of the given loop. The expressions in the modified body are then simplified using the transformation described in Section 3.2. As a result, the loop bodies of two subsequent iterations of the `i`-loop are available. These two versions of the loop body then form the input for the syntactic CS detection described in the previous subsection.

As explained in Section 3.4.1, only common subtrees that do not overlap in the unprocessed source are sensible targets for an elimination. This is equivalent to the uniqueness test for the integral identifiers associated with each node among all occurrences of a CS.

From the remaining redundancies, one must select an appropriate subset to be eliminated.

Listing 3.7: Example in which eliminating a smaller CS results in a performance improvement.

(a) input code

```
for (int i=0; i<n; ++i) {
  A[i] = exp(0.5*i-0.25) + exp(0.5*i+0.25);
  B[i] = exp(0.5*i-0.25) + 4.2;
  C[i] = exp(0.5*i+0.25) + 4.2;
}
```

(b) eliminating the largest CS

```
lcs = exp(0.5*0-0.25) + 4.2;
for (int i=0; i<n; ++i) {
  tcs  = exp(0.5*i+0.25);
  A[i] = exp(0.5*i-0.25) + tcs;
  B[i] = lcs;
  C[i] = tcs + 4.2;
  lcs  = tcs + 4.2;
}
```

(c) eliminating a smaller CS

```
lcs = exp(0.5*0-0.25);
for (int i=0; i<n; ++i) {
  tcs  = exp(0.5*i+0.25);
  A[i] = lcs + tcs;
  B[i] = lcs + 4.2;
  C[i] = tcs + 4.2;
  lcs  = tcs;
}
```

Selection

The selection of subexpressions to be eliminated in this approach is worth a closer look. Choosing the largest CS is not always sufficient. Listing 3.7(a) shows a loop in which one could reuse data from the previous loop iteration. The run time of this loop is clearly dominated by the calls of `exp`. While the original code contains four calls in each iteration, a syntactic CSE can save two of them. But this code can also be optimized by a loop-carried CSE.

On the one hand, Listing 3.7(b) shows the resulting code if the largest possible redundancy, namely `exp(0.5*i-0.25) + 4.2` in iteration `i-1` and `exp(0.5*i+0.25) + 4.2` in iteration `i`, is eliminated. However, it still contains two calls of `exp`, which can only be simplified by adding another variable to carry even more data between loop iterations. In this example, it would only require one additional scalar value but the problem can also arise in situations with a higher dimensionality, which could lead to a significantly higher memory consumption.

On the other hand, starting directly with the smaller redundancy `exp(0.5*i-0.25)` and `exp(0.5*i+0.25)` results in the code shown in Listing 3.7(c), which gets along with only a single `exp` call. Therefore, the code generator takes not only the size of a CS, but also the number of its occurrences into account. The heuristics used eliminates all redundancies larger than a fixed threshold. This leads to good results in all test cases. A more advanced approach based on, e. g., the results of a roofline analysis or auto-tuning would be possible, too.

Elimination

The last phase—the elimination itself—proceeds as follows. A new array to store the values computed in the previous loop iterations must be introduced. Its extent depends on how many loops are inside the one for which the loop-carried CSE is executed. E. g., for a three-fold loop nest with iteration vector `(i,j,k)` $\in$ `{0, ..., 511}`^{3} and a redundancy between subsequent iterations of the `i`-loop, separate scalars for each of the inner $512 \cdot 512$ iterations are required. Its initialization is performed only in the first iteration of the `i`-loop using the redundant expression itself. As a side-effect during the polyhedral techniques applied later, this new condition may be removed via a partial unrolling performed in the AST recreation phase (see Section 3.6.2). The CSs are eventually replaced by an access to the introduced array at position `[j,k]`. What remains is an update of the array with a new value in the current iteration. The expression specifying the value can be generated from the CS by replacing each occurrence of `i` with `i + str(i)`. This also introduces a new textual redundancy, which is eliminated by the subsequent syntactic CSE.

This approach is not limited to subsequent iterations; it can be easily extended to any step size. In our domain of stencil computations, however, this is usually not necessary.

3.5 VECTORIZATION

Almost all modern processor architectures include vector units. Prominent examples are Intel x86's SSE and AVX, BlueGene/Q's QPX, or ARM's NEON. These support single-instruction multiple-data (SIMD) parallelism with a vector size ranging from 128 bits (SSE) to 256 bits (AVX/QPX) and even 512 bits (AVX-512). The latter is able to apply an operation to 8 double-precision or 16 single-precision values in parallel. It is easy to see that the processor's peak performance can be reached only if its vector units are charged to capacity, which is no trivial task.

3.5.1 Automatic Vectorization

Most contemporary compilers contain automatic vectorizers, which are meant to generate vectorized machine code. But the result is often far from optimal. Therefore, we designed and integrated our own vectorization strategy in the ExaStencils code generator. This prevents an implicit dependence on a small set of compilers and also provides more flexibility in how the main memory is accessed.

Listing 3.8: Trading memory accesses with in-register operations for a vector size of four elements.

(a) aligned and unaligned accesses

```
for (int x=a; x<b; x+=4) {
  vL = load_unal(&in[x-1]);
  vC = load_al(&in[x]);
  vR = load_unal(&in[x+1]);
  ...
}
```

(b) aligned accesses only

```
vecC  = load_al(&in[a-4]);
vecRa = load_al(&in[a]);
for (int x=a; x<b; x+=4) {
  vLa = vC;
  vC  = vRa;
  vRa = load_al(&in[x+4]);
  vL  = perm(vLa, vC);
  vR  = perm(vC, vRa);
  ...
}
```

Vector Load

There are basically three different options of loading data into a vector register:

(i) The most flexible but also slowest option is to load each floating-point value separately from memory and compose the vector element by element. This should be avoided whenever possible.

(ii) The fastest load instructions are usually *aligned* vector loads. A vector load instruction retrieves not only a single value but a sequence of consecutive values. Alignment means that the address of the memory location from which consecutive elements are fetched is evenly divisible by the vector size.

(iii) The third option is an *unaligned* load, i. e., a vector load without alignment restrictions. However, depending on the architecture, unaligned loads could be either slower, or even not supported at all, as is the case for IBM BlueGene/Q processors.

The same three options exist for store operations.

Potential of Aligned Loads

Current Intel processors support unaligned vector load and store operations without a performance penalty but, for some codes, a voluntary restriction to aligned accesses can result in an overall smaller number of loads from the memory hierarchy, as shown in Listing 3.8. The left code contains unaligned load instructions to fetch three partially overlapping vectors. The right version is semantically equivalent but contains only aligned load operations, which require permuting the elements of the vectors starting at positions `x-4`, `x` and `x+4`, respectively. The advantage is that one can reuse the latter two vectors in the subsequent loop iteration. This reduces the number of load

instructions per iteration at the cost of additional instructions for the permutation. However, the latter can operate on registers only, which potentially improves the performance of bandwidth-bound codes. It also requires more complex code restructuring to integrate the reuse of vectors from the preceding loop iteration, which is usually not done by production compilers.

Vector Intrinsics

To support an explicit vectorization, current compilers provide special vector intrinsics and vector types. Syntactically, the intrinsics are small functions to generate and deal with vector types. In contrast to normal functions, they instruct the compiler to generate a few specific machine instructions instead of an explicit function call. These instructions are also better integrated and optimized than inlined functions, since the compiler is fully aware of their semantics. For example, `_mm256_add_pd` tells the compiler to generate the corresponding `vaddpd` instruction, which adds two vector registers point-wise, without the need to embed assembler code in C/C++.

3.5.2 Vectorization Strategy

The vectorization strategy contains a single transformation that applies the vectorization. It searches and potentially replaces loops with the following properties:

- It must not be inside a device function, such as a CUDA function. Vectorizing a device function is currently not supported.
- It must be flagged as a parallel loop.
- It must be the innermost loop of a nest.
- It must be a traditional numeric for-loop and its counter has to be declared and modified only in the loop head.

The generator begins optimistically to vectorize such loops and backtracks if any not supported nodes or constructs appear. This strategy operates on a copy of the loop's AST, which is traversed explicitly, i. e., Athariac's traversal routines are not used since they are designed to operate only on a small set of nodes, while the vectorization has to replace or, at least, acknowledge every node. Note that the vectorization generates ExaSlang IR vector instructions that are not tied to any target architecture. They are converted to target-specific instructions in the final C++ code generation phase. However, specific properties of the target architecture are still taken into account, such as the availability of unaligned memory accesses.

Algorithm 3.2 is the vectorization procedure. The first step is to allocate a context object which handles and organizes the generated

ALGORITHM 3.2: Vectorize a loop nest.

INPUT: copy of the loop nest loop to be vectorized
OUTPUT: AST of vectorized loop nest

```
1  FUNCTION vectorize_loop(loop):
2      ctx ← generate_ctx(loop)
3      if loop performs reduction then
4          var ← get_reduction_variable(loop)
5          varV ← get_vector_temporary(var)
6          idEl ← get_reduction_identity_element(loop)
7          idElV ← broadcast_scalar(idEl)
8          ctx.pre_loop ← declare(varV, idElV)
9          ctx.post_loop ← assign(var, vector_reduction(varV))
10     if fields aligned then
11         ensure aligned write accesses
12         if avoid unaligned accesses then
13             ensure aligned read accesses
14     vstmts ← [ ]
15     foreach stmt ∈ get_body(loop) do
16         vstmts ← vstmts + vectorize_stmt(stmt, ctx)
17     new_stmts ← [ ]
18     if fields aligned then
19         new_stmts ← [generate_prolog_loop(loop)]
20     guard ← generate_condition(ctx.get_empty_test())
21     guard.true_body ← ctx.pre_loop
                       + ctx.get_vectorized_loop(vstmts)
                       + ctx.post_loop
22     new_stmts ← new_stmts + guard
23     new_stmts ← new_stmts + generate_epilog_loop(loop)
24     return new_stmts
```

nodes, such as initialization statements. Then, a special treatment for reductions is incorporated in lines 3 to 9. A reduction is divided into two phases. First, the original loop does not reduce to a scalar but to a vector, i. e., it computes multiple partial results. Second, the vector elements are reduced to a scalar after the execution of the loop. The corresponding declaration and initialization of the vector variable and the final reduction are generated and stored for later use in the ctx object. Also, a mapping from the original reduction variable to the new vector temporary is preserved as part of the `get_vector_temporary` function. It ensures that every occurrence of the former will be replaced with the latter during the vectorization.

The subsequent block (lines 10 to 13) is only executed if all fields have been aligned. In this case, the generator aligns all write accesses

Listing 3.9: Example loop to demonstrate different alignment situations.

```
for (int y = 0; y < 128; ++y)
  for (int x = 0; x < 128; ++x)
    a[128*y + x] = b[128*y + x + 1] + c[130*y + x];
```

by selection of a suitable starting value for the vectorized loop. A potential prolog loop is later inserted in line 19. The generator can also be instructed to avoid unaligned vector loads in general. In this case, a non-aligned vector load must be replaced by two aligned loads that contain all elements of the desired vector and an appropriate permutation. To account for all supported vector architectures, the permutation must be independent of any surrounding loop. Listing 3.9 presents a case in which this property does not hold for a vector size of 4 elements. Aligning the write access to array `a` does not require a prolog loop, since the index expression `128*y + x` is already a multiple of 4 for the initial values of both surrounding loops, namely `x = y = 0`. The vector for `b` can then be generated by loading the vectors starting at `b[128*y + x]` and `b[128*y + x + 4]` and selecting the last three elements of the former and the first of the latter vector. This permutation is independent of `x` and `y`. In contrast, the vector starting at `c[130*y + x]` is aligned for even values of `y` and misaligned for odd values. I. e., the permutation depends on the loop iterator and, thus, the code generator refrains from vectorizing this loop. Line 13 is a test of whether this property holds for all read accesses.

The actual vectorization of the loop body is specified in lines 14 to 16. Function `vectorize_stmt` performs an extensive matching to deal with all occurring statement types, such as assignments, and replaces all variable and array accesses, as well as operations. This also includes calls to functions of the math library, given that vectorized versions are available externally, i. e., as part of another library. If a variable or array access is encountered for the first time, a new name for the vector variable is generated. Additionally, a corresponding declaration statement and an initialization via load instructions, or a vector store statement is generated for read, respectively write accesses. If an aligned load is not possible and an unaligned load is not allowed, loads for the two adjacent aligned vectors that frame the desired one and a suitable permutation are generated. Declarations and permutations are inserted immediately before the statement to be vectorized, while a store is placed behind it. For each subsequent occurrence of the same variable or an array access with the same index expression, the vector temporary is reused.

The list *new_stmts*, declared in line 17, is the final replacement for the original loop. It starts potentially with a prolog loop, as explained earlier. Statements that are executed before or after the vectorized loop, such as the initialization of a vectorized reduction variable, must

only be executed if the vectorized loop has at least a single iteration. Therefore, these statements are governed by an according check in lines 20 to 22. An epilog loop that executes all remaining iterations (which no longer form a complete vector) is composed in line 23. The generated statements in *new_stmts* are eventually returned to replace the original loop.

3.5.3 Vector Load Optimizations

While the vectorization strategy itself does not generate obviously redundant load instructions, a simple subsequent loop unrolling might introduce some redundancies that should be dealt with. Also, there are still some improvements envisionable: it may be possible to eliminate load instructions by reusing vectors of the previous loop iteration, as discussed in Section 3.5.1.

As a remedy, a specialized strategy to identify and remove such unnecessary loads takes hold after the unrolling. It unifies syntactically identical load instructions and searches for vector declarations that, when executed in subsequent loop iterations, access the same memory location. This can be tested as described in Section 3.4.4: the loop iterator `i` in the address computation is replaced by the expression `i - str(i)` (`str` denotes the stride of the given loop) and the result is normalized with the techniques introduced in Section 3.2. If the resulting expression is then syntactically equivalent to an unmodified load, the latter is redundant and can be removed. In this case, a declaration of a vector temporary is inserted before the loop. This new temporary transports data between loop iterations, which reduces the number of transfers from main memory.

3.5.4 Interaction with Loop-Carried CSE

Another specialization targets the loop-carried CSE introduced in Section 3.4.4. One drawback of this technique is that it effectively sequentializes the corresponding loop, since data from the previous iteration is required.

On the one hand, employing multiple processor cores regardless of a previous redundancy elimination is easy if each thread executes one contiguous sequence of loop iterations. In this case, the initialization must be adapted to be executed not only in the first iteration of the loop but in the first one of each thread. Additionally, each thread must have its private buffer to carry information between different loop iterations.

On the other hand, vectorizing the innermost loop to load a single processor core to capacity is more complex, since it incurs the parallel computation of subsequent loop iterations. Excluding this loop from the loop-carried CSE is also not an option, as it is the most profitable

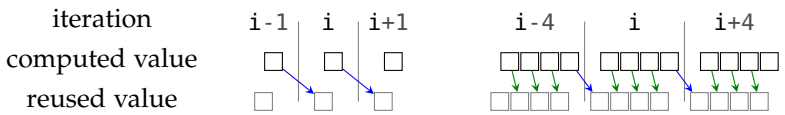


Figure 3.3: Reusing a value from the previous iteration in the scalar and vectorized case. Arrows represent data copy operations. Blue ones are across loop iterations.

one to be optimized: it requires only a single scalar to carry data between iterations. However, the newly introduced data dependences do not prevent the vectorization in general, but require a more careful selection of the generated instructions for all three accesses to the new temporary variable. First, the initialization of the temporary need not be vectorized at all, as only the initial scalar has to be computed separately. Second, vectorizing a statement, which loads the value from the previous iteration, leads to two different situations, as shown in Figure 3.3. For the first element of the vector, the required value is the one of the previous iteration stored in the temporary (blue arrow) while the values of all other elements are actually computed in the current iteration and used twice. This requires the corresponding elements of this iteration to be computed prior to the load operation and also to generate suitable data shuffling instructions. Third, the store of the newly computed value handed to the next iteration can be restricted to the last element of the computed vector. However, the code generator preserves the whole vector, since this may reduce the number of shuffle instructions. In case of a prolog or epilog loop, additional load and store instructions to save data between the different loops are inserted as well.

3.6 CLASSIC POLYHEDRAL TECHNIQUES

Stencil codes, as introduced in Section 1.1, usually have a very low computational intensity: the number of operations to be executed is low compared to the number of data elements involved. Therefore, one of the first bottlenecks that limit performance is the memory bandwidth, and optimizations that are able to reduce the bandwidth requirements are crucial. For stencil codes, an optimization of the data locality is promising. This means the computations are rearranged such that input data is fetched only once from memory and reused as long as it stays in the processor's fast on-chip cache. Two groups of tiling techniques exist that may increase locality, namely a classic space tiling and a time tiling. The polyhedral model introduced in Section 2.1 is well suited for both transformations and established tools and techniques are available.

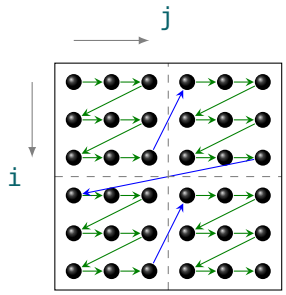


Figure 3.4: Computation ordering for a space tiling. The dashed lines represent tiles that are processed one after the other.

This section provides basic information about different tiling techniques and the integration of the polyhedron model in the ExaStencils code generator. The subsequent section then describes a polyhedral search space exploration to achieve a better result than with model-based approaches.

3.6.1 Tiling

Tiling is a generic term that covers numerous different techniques ranging from widely applicable to very specialized optimizations. There are also several stencil-specific tiling techniques, such as diamond [4, 7] or hexagonal [35] tiling. But our code generator focuses on classic techniques for now: an axis-aligned rectangular space tiling and a time tiling.

Space Tiling

Due to the neighborhood relationship of stencil codes, the computations of neighboring elements in any dimension access partially overlapping memory regions. This is not a problem for neighbors in the inner dimension or loop, since previously loaded elements are still in the processor's on-chip cache and can be reused immediately. For neighbors in outer loops, however, there are many other computations inbetween, which also load the cache and may cause data to be evicted before it can be reused. This leads to a repeated fetch of the same elements from main memory and, thus, wastes a performance-critical resource: memory bandwidth.

A remedy is a classic space tiling, also *spacial blocking*. It divides the computations, or rather the iteration domain, into several tiles, which are then processed one after the other. This reduces the number of iterations between neighbors of an outer loop and, thus, the amount of data that must fit into cache before a reuse occurs can be fine-tuned.

Figure 3.4 shows a tiled iteration domain for a two-dimensional loop nest. The original code executes all iterations row-wise, while the new, tiled iteration ordering is represented by the arrows. The green arrows

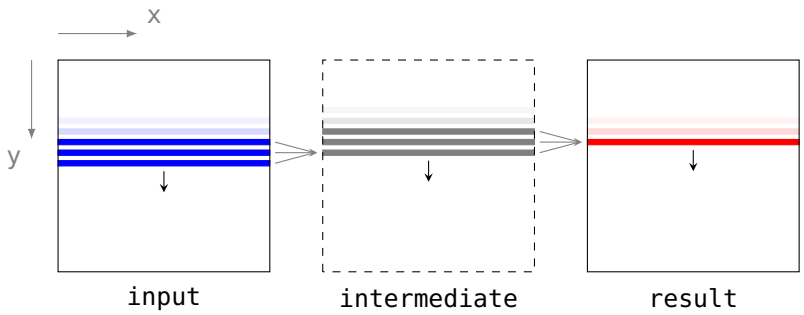


Figure 3.5: Computation ordering for a time tiling with two time steps. The field labeled `intermediate` represents the results of the first time step, `result` those of the second.

correspond to the two inner loops in the target code that enumerate all instances within the tiles, while the blue ones enumerate the tiles themselves. The tile shape can be arbitrarily complex, but this may result in a cluttered, and potentially inefficient, loop nest. Therefore, we focus on rectangular, axis-aligned tile shapes. A transformation for such a tiling has the form

$$\{\ [i,j,\ldots] \rightarrow [\mathtt{floor}(i/\beta_i),\mathtt{floor}(j/\beta_j),\ldots,\ i,j,\ldots]\ \}$$

where β_i, β_j, ... are the constant, integral tile sizes. The actual loop nest for the tiled code, including an appropriate boundary handling in case the iterations cannot be distributed evenly among the tiles, is then created by the polyhedral code generator. A brief discussion of when such a tiling is allowed can be found in Section 3.7.1.

Time Tiling

A space tiling is able to minimize the bandwidth requirements for a single, isolated stencil application, since all elements of the input array (except few at the border of the tiles) are fetched only once from main memory. But, if two or more stencil applications are executed in sequence, a further optimization opportunity emerges: the results of the first stencil application can be reused in the second application as long as they reside in cache. In this case, not a single fetch operation is issued for the input of the second stencil. This optimization is called *time tiling*, or *temporal blocking*, since subsequent applications of the same stencil can be viewed as different time steps [61, 96].

Figure 3.5 depicts a line-based version of this optimization. To compute a new line of the intermediate grid, namely the bottom gray line, the three lowest blue lines from the input grid are required. Two of them have already been accessed in the previous iterations and may still reside in cache. Only a single line from the input must be fetched from memory. Then, along with the two previously computed gray lines, the red line can be computed. If, in this example, three lines of both the input and the intermediate grid fit in the processor's cache,

data from the latter grid must never be fetched from main memory. Also, adding a space tiling on top lessens the cache space demands.

The ExaStencils code generator addresses multigrid methods. A repeated application of stencil codes occurs in the pre- and post-smoothing, as shown in Section 1.2. For example, a study by Ghysels and Vanroose [32] revealed that, for standard Jacobi, 15 iterations in the three-dimensional case and 10 iterations in the two-dimensional case exhibit best performance. Their focus was on shared-memory multicore systems. In case of a distributed memory using, e. g., MPI, an additional overhead emerges for multiple time steps, caused by the communication between nodes that compute neighboring regions of the whole field. Thus, in the case of distributed memory, it pays to keep the number of time steps even lower—typically not higher than 5. Thus, techniques that rely on hundreds or even thousands of steps may not be applicable.

Time tiling an MPI parallel stencil code requires a special treatment. To account for the distributed memory, the fields are blocked and the blocks are distributed across all nodes. Since a stencil computation accesses neighboring elements, an update of elements on the block border requires data that reside on other nodes. Thus, to prevent remote memory accesses during the stencil computations, additional ghost layers are introduced that replicate data from the neighboring blocks. However, these new layers have to be synchronized after every time step, which impairs a time tiling. One solution to this problem is to compute the new values of the ghost layers locally instead of fetching them from other nodes. This trades some communication steps with redundant computations. As the computation of a ghost layer also accesses neighboring data, additional ghost layers for each time step are necessary, which increase the memory consumption and the communication volume. But it enables a time tiling.

ExaSlang 4 provides a special loop construct for time tiling to modify the boundaries of nested **`loop over`** constructs accordingly: a "**`repeat`** ν **`times`** `with contraction` $[\delta_1, \delta_2, \delta_3]$" requests ν time steps and the three loops of each enclosed **`loop over`** are shortened after each **`loop over`** execution since said loop has been extended initially by twice the number of iterations given in the brackets. In detail, for a Jacobi stencil in the i-th time step, the lower bound of the j-th dimension of the nested **`loop over`** is increased by $\delta_j \cdot (i - 1)$ iterations, while the upper bound is reduced by the same value. For colored smoothers, the loop boundaries have to be adapted not only after every time step but also between different colors. The number ν of time steps and the contraction values δ_j must be compile-time constants since the loop is fully unrolled during the code expansion by the generator.

3.6.2 Implementation

The integration of the polyhedron model into the ExaStencils code generator necessitates a special data structure to handle polyhedral representations. It is introduced first and the optimization process follows. The latter can be divided roughly into four phases. The first phase covers the extraction of polyhedral representations for the relevant loop nests. The second phase includes all kinds of simplifications and preparations of the extracted representations, while the actual scheduling is the third phase. Finally, ASTs for the transformed representations are created that replace the original subtrees.

We integrated the isl to represent and manipulate integer polyhedra. Since the isl is a C-library and the code generator is written in Scala, Java/Scala bindings of the isl are leveraged.

Representation

A polyhedral representation of a static control part (SCoP) (see Section 2.1.1) contains at least:

- integer polyhedra for the iteration domain and the original schedule of the loop nest, as introduced in Sections 2.1.2 and 2.1.3,
- a mapping from the identifiers in the polyhedral representation to the unmodified, original subtrees for the statements.

The iteration domain specifies which statement instances a loop nest contains, i. e., for which values of the loop iterators a statement must be executed. The schedule associates every statement instance with a point in time. What remains is the actual computation a statement instance represents: the original subtrees for all statements have to be stored, too. Other data structures, such as integer polyhedra for the array access relations or the data dependences, are stored in the SCoP representation, too.

Extraction

The extraction of polyhedral representations is implemented via a specialized collector. It searches for nodes that represent a SCoP. In the code generator's current form, there is only a single node type that is of interest here, namely `IR_LoopOverDimensions`. It is a special domain-specific node that incorporates a full iteration over a multi-dimensional field. Such a node is generated for an ExaSlang 4 **`loop over`** statement (see Section 2.2.1). Therefore, for each `IR_LoopOverDimensions` node, a separate representation is created.

The basic idea of the extraction process is to generate all representation objects in a single AST traversal. The extractor maintains a simple state machine to track whether the traversed nodes are inside a SCoP or even part of a statement in a SCoP. The initial state is the search for

a SCoP. Such a state machine is easy to implement since a collector traverses the AST depth-first and every node is visited twice: once before its subtrees are processed and once immediately afterwards. Thus, the first access of an `IR_LoopOverDimensions` node initializes all data structures of the representation for the respective SCoP. At this point, the representation does not contain any statement but the loop boundaries are cached to support the creation of their iteration domain once the traversal reaches a statement. Then, the children of the loop node, i. e., the nodes inside the loop body are processed. A visited statement node extends the domain and (original) schedule of the representation and the original statements are retained. Variable and array accesses contribute to the read and write access relations, depending on where they are located in the encasing statements. If the traversal reaches code structures not supported by the polyhedron model, such as non-affine array accesses, the extractor discards all data of current representation and switches back to the initial state: the search for a new SCoP. Function calls are not supported in general, too, but calls to unproblematic pure functions of the math library, such as `fabs` or `exp`, are permitted. The extractor also provides methods to register other generated functions as unproblematic. Branches inside the loop body are supported, given that the conditions are affine expressions in the surrounding loop iterators or constants. However, to accommodate some special cases, e. g., that of a very large number of branches, the code generator coarsens the SCoP representation by treating the whole loop body as a single statement. This reduces the complexity of the extracted representation, which eases the optimization step afterwards. It also prevents the generation of too cluttered code, as conditions inside the loop body are prevented via a partial unrolling wherever possible. Eventually, the representation is finalized when the loop node itself is accessed the second time, i. e., after its body has been processed completely.

Preparation

The preparation phase can be divided further into several individual steps. The first one deals with local variable declarations inside a SCoP. A declaration of a variable inside a loop means that information can only be transferred between different statements in the same loop iteration, not between different iterations. However, this cannot be modeled satisfactorily. On the one hand, the declaration can be replaced by an equivalent assignment and a new declaration is inserted before the SCoP. This requires no special treatment when the variable is modeled but, since the same memory location is written in every iteration, the whole loop is effectively sequentialized, which prevents several further optimizations. On the other hand, the variable can be expanded to an array such that each iteration accesses a distinct, unique memory location. This is the most flexible solution, since

no unnecessary dependences are introduced. However, it increases the memory consumption significantly and the newly created array also competes for valuable cache space. Since both solutions come with serious drawbacks, we try to bypass the whole problem by a reduction of the granularity: all statements that access such a locally declared variable, as well as all statements inbetween, are scheduled alike, i. e., all of them are represented by one single statement and the locally declared variable must not be modeled at all. This reduces the scheduling flexibility but none of the other more serious drawbacks emerge.

The second step addresses the fact that the extractor creates a separate representation for each `IR_LoopOverDimensions`, even if some are adjacent in the same scope. These successive loops form a single SCoP and the different representations can be merged easily, which allows, e. g., a time tiling as introduced in Section 3.6.1. Since the extractor creates globally unique identifiers for the statements, a simple union operation suffices to merge all contents except the (unoptimized) schedule. The new schedule must then incorporate the fact that the loop nests are executed one after the other. Thus, a new constant dimension is added outermost to the individual schedules. It enumerates the different loop nests, i. e., its value for statements of the i-th loop nest is the constant i.

After everything has been merged and the domain, schedule and memory accesses have been finalized, the data dependences have to be computed. The code generator employs isl functionality for this purpose.

The flow, i. e., read-after-write dependences may then govern a polyhedral dead code elimination. However, this transformation may result in more complicated program representations which, in turn, may slow down or even preclude further optimizations. The generated code usually does not contain unnecessary computations, so this transformation may only be advisable in rare occasions. Its idea is to execute only those statement instances that are actually required to compute the final result of the SCoP. Thus, to determine the set of necessary statement instances, one can start with the (according to the schedule) last write accesses LW of all relevant memory locations. When in doubt, all written memory locations are relevant (i. e., accessed after the SCoP) and their final update must be preserved. The inverse flow dependences FD^{-1} specify a relation that maps a statement instance to those statement instances, that compute its direct input values. Then, the image of the set of the last write accesses LW determined above under the reflexive transitive closure of this relation $(FD^{-1})^*$ results in the desired statement iterations $L = (FD^{-1})^*[LW]$. No other instance except those in L need be executed and the domain can be limited accordingly.

Scheduling

The most complex phase of a polyhedral optimization is the selection of a suitable schedule that leads to a good or even optimal performance. The classic, model-driven approaches optimize some (heuristic) objective functions, which may lead to good schedules. For example, the scheduling algorithm proposed by Feautrier [27, 28] minimizes the latency, i. e., the number of time steps, given that arbitrarily many processing units are available. However, for modern processors, data locality is frequently at least as important as parallelism. The PLuTo algorithm [8] is based on this fact and optimizes for both coarse-grain parallelism and data locality.

Feautrier's algorithm and a variation of the PLuTo algorithm are implemented in isl and, thus, are available in the ExaStencils code generator. The latter is referred to as the isl scheduler. As can be seen in Section 4.3.3, the results of of the isl scheduler are poor for stencil codes and by far better schedules exist to implement a time tiling. This is why we tweaked the isl scheduler heuristically by modifying its input format. Besides the data dependences that must not be violated by a schedule (the *validity* dependences), the isl scheduler accepts a set of (*proximity*) dependences whose distance between source and target is minimized by the algorithm to increase data locality. Both sets need not be identical and we discovered that it is sometimes better to pass a specially reduced set of dependences as proximity dependences: for each source of multiple data dependences, only the one with the lexicographic smallest target is kept. This leads to a significantly better performance for some stencils—including a classic Jacobi stencil—and does not deteriorate performance for any others. Therefore, this heuristics is our default.

Alternatively, the user may explicitly provide schedules for some of the SCoPs in the generated application. In this case, one must run the code generator once using any of the above scheduling techniques. The SCoPs in the generated code are preceded with a comment containing an ID. This ID is required to instruct the code generator for which SCoP a given schedule should be used.

Another possibility to a identify even better schedules is to search the space of all legal transformations explicitly. Such a polyhedral search space exploration has been implemented and is described in Section 3.7. It can target either specific SCoPs via their IDs described above for user-provided schedules, or all applicable ones when a wildcard is provided.

AST Recreation

The final phase of a polyhedral optimization is to recreate syntax trees for the transformed SCoP representations. There are algorithms and tools for this purpose [5] and isl also provides functionality

to create a C-like syntax tree. The code generator relies on isl to create the ASTs and then transforms them to ExaSlang IR. These eventually replace the original `IR_LoopOverDimensions` nodes. The polyhedral representations are also tested for parallel dimensions and the corresponding loops are marked parallel. This allows either a vectorization, in case the innermost loop is parallel, or a subsequent parallelization using OpenMP.

3.7 POLYHEDRAL SEARCH SPACE EXPLORATION

As mentioned in the previous section, model-driven approaches do not always lead to a bearable result when a time tiling is required. Thus, we developed an exploration to evaluate a subset of all legal schedules that has the potential to contain good variants.

To illustrate the basic concepts of a polyhedral search space exploration, we start with an abstract algorithm for a naïve, exhaustive exploration. We continue with a modified exploration, guided by a more explicit representation of polyhedra and additionally powered by a set of filters tailored to the domain of stencil codes.

Both algorithms are designed to generate complete, i. e., bijective schedules only. Incomplete schedules have fewer dimensions and, thus, their generation incurs less exploration effort. But the missing (inner) dimensions still influence performance, even if all dependences are carried by the existing (outer) dimensions. For example, the incomplete schedule `{ S[i,j,k] -> [j] }` only specifies that the original `j`-loop should be the outermost loop while no information on the `i`- or `k`-loops is given. However, it is easy to see that their permutation—or a more complex transformation like a skewing—may have a serious effect on different properties, such as the memory access pattern or whether the loop nest can be vectorized efficiently.

3.7.1 Search Space

One-Dimensional Schedule Space

The search space of all one-dimensional schedules can be viewed as a multi-dimensional polyhedron of all possible coefficients for the iterators, the structural parameters, and the constants. Consider the following iteration domain:

```
[n,m] -> { S[i,j] : 0 <= i < n and 0 <= j < n;
           T[i]   : 0 <= i < m }
```

Each (affine) schedule for this iteration domain, whether legal or not, has the following form, which is called the *prototype schedule* Θ_0:

```
[n,m] -> { S[i,j] -> [iS*i + jS*j + nS*n + mS*m + cS];
           T[i]   -> [iT*i + nT*n + mT*m + cT] }
```

The set of all possible affine transformations can then be written:

$$\{\ [iS,jS,\ iT,\ nS,mS,\ nT,mT,\ cS,cT]\ \} = \mathbb{Z}^9$$

where `iS`, `jS`, and `iT` are the coefficients of the loop iterators for statement **`S`** and **`T`** respectively, `nS`, `mS`, `nT`, and `mT` are the coefficients of the structural parameters, and `cS`, `cT` are the constant parts. Following this notation, the next two lines are two different representations of the same schedule:

```
[1,1, -1, 2,0, 0,1, 3,-2]
[n,m] -> { S[i,j] -> [i+j+2n+3]; T[i] -> [-i+m-2] }
```

The former is a variant of the matrix representation presented in Section 2.1.3.

A one-dimensional schedule with only zero coefficients for all iterators of each statement is called *constant*. In the target code, it does not become a loop but a textual sequence.

Legality Constraints

The entire space of $\mathbb{Z}^9$ also contains schedules that violate some data dependences, i. e., not all elements are legal schedules. For each dependence from $\vec{x}_S$ to $\vec{x}_T$ that is not carried, constraints of the search space must be added to ensure that the inequality

$$\Theta_0^T(\vec{x}_T) - \Theta_0^S(\vec{x}_S) \geqslant 0 \tag{3.1}$$

holds. To be exact, a single dependence in this definition is a single pair of statement instances in the dependence polyhedron. However, such a fine granularity is usually not manageable and a small set of dependence polyhedra must be formed. These polyhedra are treated atomically, i. e., a dependence polyhedron is said to be carried iff all its dependence instances are carried. To create dependence polyhedra with a sufficient granularity, we start with one polyhedron per pair of statements and the following approach is used to split the polyhedra iteratively: as long as a dependence polyhedron d contains multiple instances with the same source iteration, we use isl's `lexmin` method to replace it with $d_1 = \text{lexmin}(d)$ and $d_2 = d \setminus d_1$. The method `lexmin` computes a relation that maps each element of the source to the single lexicographic minimum of the target elements.

Some of the dependence polyhedra may contain holes, e. g., if only every other statement instance is source of a dependence. Such a dependence polyhedron cannot be handled correctly in subsequent computations. A remedy is to compute the hull, which introduces new dependences that may exclude some legal transformations.

The constraints to the search space can be computed by applying an affine form of Farkas' lemma to each dependence polyhedron and

a Fourier-Motzkin elimination for the prototype schedule Θ_0^X for statement X [27, 83]. Among other libraries, isl provides an implementation for this purpose. The basic idea of Farkas' lemma is that, for any non-empty polyhedron $\mathcal{D}$ represented by k inequalities $\vec{a}_k \cdot \vec{x} + b_k \geqslant 0$, an affine function is non-negative everywhere in $\mathcal{D}$ iff it has the form

$$\lambda_0 + \sum_k \lambda_k(\vec{a}_k \cdot \vec{x} + b_k) \text{ for } \lambda_i \geqslant 0$$

Applied to the dependence polyhedra, this provides an alternative representation of the left-hand side of equation (3.1):

$$\Theta_0^T(\vec{x}_T) - \Theta_0^S(\vec{x}_S) = \lambda_0 + \sum_k \lambda_k(\vec{a}_k \cdot \vec{x} + b_k) \quad \text{for } \lambda_i \geqslant 0$$

With this representation, the coefficients of the iteration variables and the structural parameters, as well as the constants, can be gathered and equated. Projecting out the Farkas multipliers λ_i via a Fourier-Motzkin elimination results in the desired constraints for legal schedules.

Multi-Dimensional Schedules

A multi-dimensional schedule can be composed iteratively from several one-dimensional schedules, which are referred to as the dimensions of a schedule in contrast to the dimensions of the search space. There are two properties that have to be taken into account for multi-dimensional schedules. First, the satisfaction constraints for dependences carried by an outer schedule dimension are not necessary for inner dimensions and can be removed to enlarge the search space. Second, the schedule dimensions associated with a multi-dimensional schedule should be linearly independent: a dimension that is linearly dependent on outer ones will assign the same time value to statement instances that also receive the same time value in outer dimensions and, thus, can be ignored. Its generation can be prevented by adding appropriate constraints to the search space [8, 54]. The linear independence can be enforced either statement-wise, or for the whole schedule dimension, i. e., the whole vector. While the former discards some legal schedules, such a restriction may be beneficial in some cases, e. g., if a spacial tiling is desirable.

Note that equation (3.1) captures not only strong, but also weak satisfaction of the dependences. But, since we compute a complete, i. e., bijective schedule, the property $\vec{x}_S \neq \vec{x}_T \Rightarrow \Theta^S(\vec{x}_S) \neq \Theta^T(\vec{x}_T)$ holds. This implies that there is a c such that $\Theta_c^S(\vec{x}_S) \neq \Theta_c^T(\vec{x}_T)$ for each dependence from $\vec{x}_S$ to $\vec{x}_T$. That is, the dependence is strongly satisfied at some level.

Tiling

A mandatory optimization of higher-dimensional loop nests is space tiling, as presented in Section 3.6.1: the transformed iteration space

ALGORITHM 3.3: An abstract, exhaustive polyhedral search space exploration.

INPUT: set of not yet carried dependences D and an incomplete schedule s

OUTPUT: set of legal, complete schedules

```
1  FUNCTION exploration(D, s):
2      searchSpace ← search_space(D)
3      searchSpace ← searchSpace ∩ linear_independent(s)
4      if searchSpace = {} then
5          return {s}
6      Ss ← {}
7      foreach sd ∈ searchSpace do
8          s' ← add_schedule_dimension(s, sd)
9          D' ← D ∖ carried(sd, D)
10         Ss ← Ss ∪ exploration(D', s')
11     return Ss
```

is partitioned into multi-dimensional chunks that are executed atomically in sequence. Such a tiling can improve cache efficiency and is only allowed if there is an affine schedule for the tiles [43], which is the case for a sequence of dimensions if they weakly satisfy all data dependences in the considered fused loop nest [8, 34]. This can be achieved easily by selecting several linearly independent schedule dimensions from the same search space without removing the constraints for carried dependences between them.

3.7.2 Exhaustive Exploration

Based on the search space introduced in the previous subsection, Algorithm 3.3 is an abstract description of a naïve, exhaustive exploration. Its first input is the set of dependences that must not be violated. The second input represents a partial, incomplete schedule, whose outer dimensions have been set, while the inner ones are to be determined by exploration. The function starts with the search space generation (line 2): the computation of all legal one-dimensional schedules for the set of dependences as outlined in Section 3.7.1. As mentioned earlier, a strong satisfaction is not required here. Additionally, only schedule dimensions linearly independent of the previously selected ones need be explored, so appropriate constraints are added to the search space (line 3). For this algorithm, linearly independent constraints are not computed statement-wise but for the entire vector, i. e., the complete row of the schedule matrix. If this leads to an empty space, the schedule is complete and returned (lines 4 and 5). Otherwise, one has to iterate over all of its elements sd (line 7), add each one as an

Listing 3.10: 3D 1st-order Jacobi stencil.

```
for (int t = 0; t < TS; t++)
  for (int i = 1; i <= N; i++)
    for (int j = 1; j <= N; j++)
      for (int k = 1; k <= N; k++)
        A[t%2][i][j][k] = a*A[(t+1)%2][i][j][k] + b*B[i][j][k]
          + c*(A[(t+1)%2][i+1][j][k] + A[(t+1)%2][i-1][j][k]
             + A[(t+1)%2][i][j+1][k] + A[(t+1)%2][i][j-1][k]
             + A[(t+1)%2][i][j][k+1] + A[(t+1)%2][i][j][k-1]);
```

inner dimension to the schedule (line 8), remove all newly carried dependences (line 9), and issue the recursive call (line 10). Function `carried` returns only those dependences from the given set D for which the source and target are mapped to different time steps by schedule `sd`.

A severe problem of this approach is the enumeration of elements of the search space. First, iterating over an arbitrary integer polyhedron can become computationally complex. isl provides support for the enumeration of a bounded space. The set of all legal transformations is unbounded and, hence, must be restricted heuristically. However, the choice of a suitable restriction that results in a sufficiently small space to be explored completely and that contains the good schedules is very difficult. For example, consider a three-dimensional 1st-order Jacobi stencil (Listing 3.10). If all variables that define the search space are restricted to `-1`, `0` and `1`, there are 2 186 one-dimensional schedules. 755 are legal schedules. Since there is only a single statement, we can set the coefficient for the structural parameters and the constant part to `0`, which reduces the search space further to 27 elements. But, for a complete, four-dimensional schedule (three in space and one in time), there are still almost $27^4 = 531\,441$ different schedules—"almost" since, for all except the outermost dimension, the linearly dependent solutions must be ignored. Thus, in general, the search space must be restricted further for this approach to become feasible [70, 71].

3.7.3 Guided Exploration

We implemented a heuristic approach to an efficient polyhedral search space exploration tuned to stencil codes. In its current state, it cannot be applied to domains other than ours. However, we believe it could be generalized or adapted, and we point out a required modification to this end in the last paragraph of this subsection. The basic idea is to refrain from restricting the search space via additional constraints and performing a full exploration over the remaining elements, but instead to evaluate only a subset with the aid or guidance of a dual representation. Therefore, the search space we are exploring (as presented in Section 3.7.1) is larger than with other techniques such as, e. g., the

Feautrier scheduler [27, 28], which greedily carries dependences by adding appropriate constraints outermost. We use basically the same search space as the PLuTo+ algorithm [1] and the isl scheduler [100], but the former restricts the absolute values of the schedule coefficients and adds dimensions to minimize the upper bound of the reuse distances. In the isl scheduler, adding bounds to the schedule coefficients is optional but it may speed up the calculation of the schedule. Even though our search space may be larger, our search does not select larger schedule coefficients since this usually results in a schedule with poor performance.

Generator Representation

In contrast to the implicit, constraint-based representation referred to previously, polyhedra can also be represented by a set of vertices V, rays R, and lines L. Each element $\vec{x}$ inside polyhedron $\mathcal{P}$ can then be generated as follows:

$$
\begin{aligned}
&(\forall\, \vec{x} \in \mathcal{P} : \exists\, \vec{v}, \vec{r}, \vec{l} \in \mathbb{R}^d : && \vec{x} = \vec{v} + \vec{r} + \vec{l} \\
&\wedge\ (\exists\, 0 \leqslant \lambda_{1\ldots k} \leqslant 1,\ \textstyle\sum_i \lambda_i = 1 : && \vec{v} = \lambda_1 \vec{v}_1 + \ldots + \lambda_k \vec{v}_k) \\
&\wedge\ (\exists\, \mu_{1\ldots m} \geqslant 0 : && \vec{r} = \mu_1 \vec{r}_1 + \ldots + \mu_m \vec{r}_m) \\
&\wedge\ (\exists\, \nu_{1\ldots n} \in \mathbb{R} : && \vec{l} = \nu_1 \vec{l}_1 + \ldots + \nu_n \vec{l}_n))
\end{aligned}
$$

where d is the dimensionality of polyhedron $\mathcal{P}$, $V = \{\vec{v}_1, \ldots, \vec{v}_k\}$, $R = \{\vec{r}_1, \ldots, \vec{r}_m\}$, and $L = \{\vec{l}_1, \ldots, \vec{l}_n\}$. The vertices can be viewed as starting points to generate the elements of the polyhedron: exactly one point $\vec{v}$ inside the convex hull of the vertices is required. Then, any linear combination $\vec{l}$ of the lines, as well as any positive linear combination $\vec{r}$ of the rays can be added. Using Chernikova's algorithm [83, 94], one can compute such a generator representation of a polyhedron from an implicit one. A line can be converted to two rays. Thus, without loss of generality, lines receive no special treatment. Additionally, we refrain from removing the zero vector beforehand, since this allows for a much handier representation. So the set of all vertices for the corresponding polyhedron contains only a single element: the origin. In the end, the search space is described fully by a set of rays.

Search Space Generation

The new exploration technique is specified by Algorithm 3.4. Function `guided_exploration` is the entry point; it is called with the set of all dependences. The first step is to compute the search space for the given dependences (line 2) as described in Section 3.7.1. The second is to apply Chernikova's algorithm to compute the set of generators for the polyhedron (line 3). Since we are at the very beginning of the exploration and have not yet selected any schedule dimension, there are no linear independence constraints to be dealt with. Thus, `chernikova` returns only a single vertex, the origin, and a set of rays.

ALGORITHM 3.4: A guided polyhedral search space exploration for stencil codes.

INPUT: set of not yet carried dependences D
OUTPUT: set of legal, complete schedules

```
1  FUNCTION guided_exploration(D):
2      searchSpace ← search_space(D)
3      rays ← chernikova (searchSpace)
4      scheds ← combine_rays (rays)
5      scheds ← scheds ∖ constant (scheds)
6      return guided_exploration_tileable(D, {}, scheds)
7  FUNCTION guided_exploration_tileable(D, s, scheds):
8      linIndep ← linear_independent(s)
9      if linIndep = {} then
           // add any constant dimension that carries all
              remaining dependences
10         s′ ← add_cst_schedule_dimension(s, D)
11         return {s′}
12     Ss ← {}
13     foreach sd ∈ scheds do
14         if sd ∈ linIndep then
15             s′ ← add_schedule_dimension(s, sd)
16             D′ ← D ∖ carried(sd, D)
17             Ss ← Ss ∪ guided_exploration_tileable(D′, s′, scheds)
18     return Ss
```

Function `combine_rays` in line 4 computes the set of one-dimensional schedules to be considered during the exploration, by combining up to n rays. This affects seriously how many different schedules are generated. For $n = 1$, the rays are only considered individually, which results in generating only schedules at the edges of the search space. If two rays are combined, i. e., their vectors are added point-wise and the result is scaled down by the greatest common divisor of all its elements, it is either on a face of the polyhedron, or inside it. Adding combinations of three or more rays is straight-forward. However, it increases the exploration effort dramatically and results in larger schedule coefficients, which makes a poor performance more likely. Depending on the dimensionality of the stencil, we use either $n = 2$ or $n = 3$ to keep the overall number of generated schedules manageable. As demonstrated in Section 4.3, there are still some schedules remaining that exhibit good performance. From the resulting set of schedules, all constant ones are removed (line 5), since they stand in the way of a fully tileable loop nest, for which we aim. Or, at least, allowing constant schedule dimensions would result in an

undesired code structure after tiling. Next, the recursive function `guided_exploration_tileable` is called with an empty schedule.

Explore Tileable Schedule Dimensions

Function `guided_exploration_tileable` is designed to add greedily as many tileable dimensions as possible to a given incomplete schedule s. This can be achieved by simply selecting several schedule dimensions from the same search space or, in this particular case, from the same set $scheds$. Function `linear_independent` computes the space of all vectors $linIndep$ that are linearly independent to s. Note that, in contrast to the same function of Algorithm 3.3, it is computed per statement and it does not consider the coefficients of the structural parameters or the constant values, but only the coefficients of the loop iterators. This is due to the fact that we would like to exclude constant schedule dimensions in a sequence of tileable dimensions. Therefore, if $linIndep$ is empty, there could still be dependences left to be considered. These have to be carried by a single, constant dimension added in line 10. At this point, only the textual ordering inside the loops has to be determined. Then, the complete schedule is returned.

If $linIndep$ is not empty, the exploration is continued. Every schedule dimension in set $scheds$ (line 13), that is part of $linIndep$ (line 14), is considered once for expansion of the incomplete schedule s: it is added to s (line 15), carried dependences are removed (line 16), the recursive call is issued, and the complete schedules are collected (line 17) and eventually returned (line 18).

With this algorithm, only schedules that are fully tileable can be explored. An n-dimensional iteration domain is *fully tileable* if and only if n-dimensional tiles exist. This is the case for the stencil codes we consider. One can extend the algorithm to non-tileable schedules: if Ss in line 18 is empty, one can optionally add a constant dimension to perform a loop fission and start over with function `guided_exploration` for the current, incomplete schedule s rather than an empty one.

3.7.4 Exploration in ExaStencils

In our domain, the domain of multigrid methods, some of the most time-consuming parts are pre- and post-smoothing. As explained in Section 3.6.1, these rarely consist of more than a handful of stencil applications with a very low computational intensity. In order to optimize them, data locality is increased by a variant of time tiling: fully unrolling the time loop and fusing the loop nests for the time steps, i. e., the stencil applications, to a single nest. The low number of time steps justifies a complete unrolling of the time loop in the ExaStencils code generator and, thus, simplifies the border handling. The polyhedron model is well suited for this type of optimization. Different techniques exist that are easy to integrate in a code generator,

Table 3.1: Transformed access for different schedules.

	Access	Schedule	Transformed Access
S	`a[i,j,k]`	`S[i,j,k]->[x=i, y=j, z=i+j+k]`	`a[x, y, z-x-y]`
T	`a[i,j,k]`	`T[i,j,k]->[x=i+1,y=j+1,z=i+j+k+1]`	`a[x-1,y-1,z-x-y+1]`
	`a[i,j,k]`	`T[i,j,k]->[x=i+1,y=j+1,z=i+j+k+2]`	`a[x-1,y-1,z-x-y]`

such as the PLuTo algorithm, the isl scheduler, but also independent platforms such as PolyMage [59] or Polyite [31]. However, as our evaluation revealed (see Section 4.3.3), there are cases for each of the above in which they are unable to identify a good schedule. This is where our guided exploration comes in.

For a narrower domain, such as ours, it pays to analyze the explored schedules and investigate what the well, or ill performing schedules have in common. We were able to identify a set of properties on the basis of which we developed a sequence of seven filters that are employed to restrict the search space and speed up the exploration. In contrast to other work, these properties are not meant to be applicable in other domains or even for other representations of stencil codes, such as for a rolled-up time loop. Their current role is to assist our understanding of the problem domain; an in-depth comparison with other tools and techniques remains for future work.

Vector Optimizations

Some of the schedules generated during the exploration lead to vectorizable loops nests: a parallelizable inner loop and array accesses with a stride of `0` or `1`. However, not all of them can be vectorized by the code generator if unaligned memory accesses should be avoided. An access is *unaligned* if it refers to a memory location whose address is not evenly divisible by the vector size. The problem arises, for example, with two statements accessing the same array as shown in Table 3.1. In this example, `i`, `j` and `k` correspond to the loop iterators of the input program, while `x`, `y` and `z` are the loop iterators of the target code—the result of the transformation by the given schedule. Let us describe the two alternative schedules for **`T`**. Consider the schedule for **`S`** and the first schedule for **`T`**. Both statements access the same memory location `a[i, j, k]`. The resulting loop nest after transformation contains the two memory accesses `a[x, y, z-x-y]` in statement **`S`** and `a[x-1, y-1, z-x-y+1]` in statement **`T`**, both surrounded by the same loop nest. The accesses differ only in a constant of `1` in all three dimensions. While the differences in the first two dimensions are irrelevant, since the generator ensures that the extent of each dimension is a multiple of the vector size, the third prevents both accesses from being aligned simultaneously. They access adjacent elements in this dimension and, thus, cannot be evenly divisible by the vector

size at the same time. A better schedule that avoids this problem is the second one for `T`, which differs only in the constant of the third dimension. Since the ExaStencils code generator vectorizes the generated code itself, if possible, it generates, for each explored schedule that suffers from this problem, a second version by modifying only the constant parts accordingly. It is not possible (without a major data layout transformation [40]) to generate a version of a stencil code for which all read accesses are aligned simultaneously since, in a single statement, neighboring data elements are accessed. However, for write accesses only, it is possible in most cases. Similar and more advanced techniques have been presented elsewhere [47]; we focus on the additive constant only.

The basic idea of how the second schedule is computed is as follows. The first schedule,

```
T[i,j,k] -> [x=i+1, y=j+1, z=i+j+k+1]
```

can be viewed as a sequence of three equations:

$$\begin{aligned}(\mathrm{I}):\ & x = i + 1\\ (\mathrm{II}):\ & y = j + 1\\ (\mathrm{III}):\ & z = i + j + k + 1\end{aligned}$$

The innermost dimension of the original access is `k`, so the transformed access' innermost dimension is an expression based on the new iterators that has the same value as the original `k` at run time, namely $z - x - y + 1 = k$. This equation (and, therefore, the transformed array subscript) can be derived from the schedule equations: $(\mathrm{III}) - (\mathrm{I}) - (\mathrm{II}) + 1$. Here, an addition or subtraction of two equations is defined analogously to the elementary row operations used, e. g., in a Gaussian elimination: adding $\alpha_l = \alpha_r$ and $\beta_l = \beta_r$ results in the equation $\alpha_l + \beta_l = \alpha_r + \beta_r$. The summand 1 is short for the equation $1 = 1$ and gives rise to the constant part in the transformed array subscript, which is required for the right-hand-side to become k. To get rid of it, its right-hand-side can be merged with any schedule dimension, depending on which one results in a legal schedule. For example, $(\mathrm{III}) : z = i + j + k + 1$ can be replaced with $(\mathrm{III}') : z = i + j + k + 2$, which results in $(\mathrm{III}') - (\mathrm{I}) - (\mathrm{II})$ and the transformed access `z-x-y` for the innermost dimension.

This optimization is also relevant for architectures that support unaligned accesses without a performance penalty, such as current Intel processors. It can result in a smaller number of access operations to the memory hierarchy, as explained in Section 3.5.1.

Exploration Filter Levels

Even though the guided exploration leads to a manageable set of schedules for our domain, testing thousands of versions of a given

problem is not always practical. Therefore, we investigated which properties the poorly and the well-performing schedules have in common and what distinguishes them. Based on this evaluation, we created several filters to reduce the set of explored schedules while keeping the good ones. This also reduces the exploration time, since some filters can be applied early in the exploration process. For example, if we can exclude schedules based on their first, outermost dimension, we need not explore their remaining dimensions. But there are also filters that can only be applied to a complete schedule. If not specified otherwise, the filters are applied after the exploration. Note that we developed these filters unprejudiced and without any previous research in mind. I. e., on the one hand, any similarities to prior findings can be seen as a verification of previous work. On the other hand, differences may be explained reasonably by our very specialized domain and the differences in the input program.

The filtering process is structured into levels 0 to 7. Level i applies filters 1 to i. The order is immaterial, i. e., each level stands for a set, not a sequence, of filters. Level 0 does not apply any filter. Note that the filters are designed specifically for the domain of stencil codes. For other domains, other filters will apply.

LEVEL 0 This level completely disables all heuristic filters.

LEVEL 1 Dependences should be either loop-independent or carried by the outer loop of the resulting code. This can be integrated in Algorithm 3.4 by ensuring that line 17 (the recursive call) is only executed if either s is empty or $D = D'$. Thus, of the explored schedules, only the first dimension, which corresponds to the outer loop, and the last dimension, which is constant and therefore represents a textual ordering, should strongly satisfy any dependence. The semantics of this filter is that the innermost loop should be parallel, which provides sufficient parallelism for a two-dimensional case and also allows vectorization. For a three-dimensional stencil, the middle loop of the nest should also be parallel to reduce the number of synchronization points and increase the workload between synchronizations. This filter is related to the idea of the Feautrier scheduler [27, 28], which carries dependences greedily as early as possible. In contrast, we also allow dependences to be carried by the innermost, constant dimension.

LEVEL 2 Every schedule with a non-zero coefficient for the innermost loop iterator in any but the last non-constant dimension is discarded. For the codes generated, this preserves only the schedules whose innermost target loop traverses the memory linearly, since the innermost loop iterator in the original code prescribes the innermost array dimension. Since the main memory can only be accessed in larger chunks, namely cache lines, the use of all of its elements before

they are evicted is mandatory. Additionally, a non-linear memory access complicates vectorization. Even though it is achieved in a totally different way, the effect of this filter is similar to other approaches that combine the polyhedron model with vectorization [47]. It can be implemented by extending the condition in line 14 accordingly.

LEVEL 3 For some stencil codes, our exploration yields both schedules with all data dependences carried by the outer loop and others in which some dependences are not carried by any loop but are strongly satisfied by the textual ordering of the statements in the loop nest. If the latter exist, this group usually contains better schedules and, therefore, the former are discarded by this filter. This may be explained by the fact that a read-after-write dependence specifies the reuse of a data element. If such a dependence is carried by a loop, the distance between the write and the corresponding read is at least one loop iteration of the outer loop (cf. level 1) while, for dependences carried by the textual ordering, the reuse distance is much shorter and the data is still in cache.

LEVEL 4 As explained previously, preventing unaligned memory accesses—even if supported by hardware—may allow further, potentially beneficial, optimizations. Therefore, this filter discards schedules for which not all write accesses can be aligned simultaneously.

LEVEL 5 Discard schedules for which the innermost loop traverses the main memory in a non-positive direction. Our code generator currently does not support vectorization of this type of loops. Also, due to the regularity of the stencil codes in our domain, for every schedule removed by this filter, there is a counterpart that traverses the inner loop in the opposite direction and that is also valid.

The filters introduced at levels 5 to 7 can be applied even before the actual exploration starts by filtering set `scheds` appropriately before `guided_exploration_tileable` is called in line 6.

LEVEL 6 Every schedule with negative coefficients is removed. This and the next filter are designed only to reduce the number of remaining schedules.

LEVEL 7 Enforce small schedule coefficients and constants. Schedules with coefficients for loop iterators larger than 2 are excluded. The absolute values of the additive constants must not be restricted, since the data dependences may require them to be different. Thus, we limit the allowed constants to be either all 0, or increasing with a stride of at most 2.

Integration in the ExaStencils Code Generator

Our guided exploration, as well as the filter levels, have been implemented in the ExaStencils code generator. The exploration is performed semi-automatically. In its current state, the user has to provide the following:

- either a list of IDs to specify for which SCoPs the exploration should be performed, or a wildcard to select all feasible SCoPs,
- a pattern for the exploration result files, and
- optionally, a schedule identifier for each SCoP ID to select one of the explored schedules for the current generation run.

Each SCoP in a generated application is preceded with its ID. Thus, the generator must be invoked with any model-based scheduler first to determine the IDs of the SCoPs for which an exploration should be conducted. The exploration result files—one for each SCoP—contain all explored schedules along with a schedule identifier. If any of these files does not exist, the actual exploration for the SCoP is performed and the file is generated. The schedule identifiers passed to the generator then determine which of the explored schedules for a SCoP should be chosen. If there is no appropriate identifier, the generator defaults to 0, which selects the identity schedule. Thus, a single call of our generator generates only a single variant. This allows generating code for as many explored schedules concurrently as computing resources are available. An integration into a job scheduler, such as slurm[1], is straight-forward, too. The drawback is that an external, yet simple logic is required to schedule different calls to the ExaStencils code generator.

3.8 DATA LAYOUT OPTIMIZATIONS

The indices of field accesses are not stated in ExaSlang 4, which means the computation is abstracted from the actual memory layout that specifies how field elements are organized in memory. The default memory layout is a direct mapping of the loop iteration vector to the position of the element accessed inside the memory. However, for a red-black coloring as, e. g., the one in Listing 3.11, this results in an update of only every other element in both generated loop nests. In turn, it reduces the effective memory bandwidth since data can only be transferred in contiguous chunks (e. g., of 512 bit) from and to main memory on current processor architectures. This becomes even more problematic if more than two colors are required. Our solution to this and similar problems is a language extension of ExaSlang 4 that

1 https://slurm.schedmd.com/

Listing 3.11: ExaSlang 4 code for a RBGS smoother.

```
Function Smoother@(all but coarsest) {
  color with {
    (i0+i1+i2) % 2,
    loop over Solution {
      Solution = Solution +
        1.0 / diag(Laplace) *
        (RHS - Laplace * Solution)
    }
  }
}
```

provides a mechanism for the specification of arbitrary affine memory layout transformations. On the one hand, this gives users the ability to experiment with different memory layouts. On the other hand, it represents a concise interface for the automatic tuning of the memory layouts of present fields in the future.

3.8.1 New ExaSlang 4 Features

A new top-level block, **`LayoutTransformations`**, gives ExaSlang 4 programmers or generators the opportunity to add three new kinds of directives. Figure 3.6 contains a simplified grammar of this extension. Here, ⟨*X-list*⟩ specifies a repetition of the non-terminal ⟨*X*⟩, using a comma as a delimiter for all except the ⟨*layoutStmt-list*⟩, which does not require any special delimiter (other than spaces or newlines). The ⟨*field-list*⟩ and ⟨*fieldName-list*⟩ can be delimited alternatively with '`and`'. ⟨*levels?*⟩ is an optional level specification prefixed with an `@` operator as introduced in Section 2.2.1. In the absence of a level specification, all levels are affected.

Renaming

The first new statement, **`rename`**, is the simplest one. It takes the name of an existing field and a new, unused identifier. The original name is

⟨*layoutBlock*⟩ ::= '**`LayoutTransformations`** `{`' ⟨*layoutStmt-list*⟩ '`}`'

⟨*layoutStmt*⟩ ::= '**`rename`**' ⟨*field*⟩ '`to`' ⟨*field*⟩
| '**`transform`**' ⟨*field-list*⟩ '`with [`' ⟨*ident-list*⟩ '`] => [`' ⟨*expr-list*⟩ '`]`'
| '**`concat`**' ⟨*levels?*⟩ ⟨*fieldName-list*⟩ '`into`' ⟨*fieldName*⟩

⟨*field*⟩ ::= ⟨*fieldName*⟩⟨*levels?*⟩

Figure 3.6: Simplified grammar of the new **`LayoutTransformations`** block.

Listing 3.12: ExaSlang 4 code for a layout transformation.

```
LayoutTransformations {
  transform Solution and RHS
    with [x,y,z] => [x,y,z,(x+y+z)%2]
  transform Solution and RHS
    with [x,y,z,c] => [x/2,y,z,c]
}
```

then replaced at all specified levels by the new one. The sole intent of this statement is to support the linking of generated code to external or legacy code. Note that **rename** is not intended to introduce aliasing and, thus, the new name introduced cannot be used in the ExaSlang 4 code.

Transformation

An actual layout transformation can be specified with the **transform** statement. It is applied to all given fields individually and the desired transformation is specified by a linear mapping that assigns every element of the input field a new location.

For example, a color splitting of the fields `Solution` and `RHS` in Listing 3.11 is given in Listing 3.12. The first transformation adds a new dimension outermost with two possible values: one for the red and one for the black points. In ExaSlang 4, unlike in C, the elements of the leftmost dimension are stored consecutively in memory, i. e., data is organized in column-major order. This transformation results in the use of only every other element of the generated arrays. The second transformation then closes the gaps by scaling the innermost dimension down. Shrinking any other dimension would be possible as well but this would result in non-contiguous accesses. In contrast to two successive transformations for the color separation and the scaling, a single transformation specifying the composition of both is possible, too. Note that one need not specify how the extent of a dimension is affected by a **transform** statement or which extent a new dimension has. This is incorporated automatically by the code generator as described in Section 3.8.2.

Since ExaSlang 4 supports also vectors and matrices as field element types [82], dimensions induced by them may also be specified in a transformation statement. Without such a directive, the new dimensions for vectors and matrices are added outermost, i. e., the default is a struct-of-arrays (SoA) representation. However, since the additional dimensions are not treated differently from the field dimensions, they can be interchanged with the latter to achieve, e. g., an array-of-structs (AoS) layout. In the case of a 2D field and vector components, such a transformation is given by the expression `[x,y,v] => [v,x,y]`. As implied earlier, in case one wants to transform only the field dimensions,

Listing 3.13: ExaSlang 4 code to copy data from `RHS` to `RHSn`.

```
Function CopyField@all {
  loop over RHSn {
    RHSn = RHS
  }
}
```

the additional dimensions of vectors or matrices need not be stated explicitly. Thus, the left-hand side of such a transformation may read `[x,y]`.

If a layout transformation is only advisable for a part of the application, an explicit conversion between different data layouts during the execution of the generated code is also possible. One simply declares separate fields for each data layout and targets them individually by appropriate layout transformations. Finally, an explicit conversion is initiated by a simple copy loop as shown in Listing 3.13. Even though this loop looks like a naive one-to-one copy, existing data layout transformations for both fields are applied in the generated C++ code. E. g., for the layout transformation in Listing 3.12, the resulting C++ code is similar to the pseudocode in Listing 3.14.

Concatenation

The **`concat`** statement concatenates two or more fields to a new one. Only fields at the same level can be merged. Optionally, the level can be specified before the field names are listed. While the ExaSlang 4 code uses only the original, separated fields, their elements are placed in a single array in the generated C++ code. To separate the different input fields, a new dimension is added outermost whose possible values enumerate the original fields. Thus, this statement can be viewed as creating a SoA from the given fields. The extents of the inner dimensions are set to the maximum of the extents of all involved fields, which potentially introduces unused areas at the upper end of each dimension. An optional centering of a smaller field in the new, larger space can be issued via a **`transform`** statement for a simple

Listing 3.14: Pseudocode, that is semantically equivalent to the function in Listing 3.13, for the layout conversion specified in Listing 3.12.

```
void CopyField() {
  for (z = ...)
    for (y = ...)
      for (x = ...)
        RHSn[z][y][x] =
          RHS[(x+y+z)%2][z][y][x/2];
}
```

constant shift. The benefit of this statement is that the memory layout of the new, merged field can be adapted with the `transform` statement. For example, the new dimension can then be permuted innermost to create an AoS. This may be useful in situations where elements of different fields are only accessed jointly. Furthermore, the original fields can also be transformed before concatenation and the code generator takes care of the order in which the different transformation statements are applied: transformations of the original fields always take precedence over any concatenation.

3.8.2 Implementation

To apply such data layout transformations, we take advantage of isl. It was designed and implemented for polyhedral compilation, but also provides functionality that matches our requirements perfectly: it offers functionality to represent and manipulate sets and relations of integer points bounded by affine inequalities. For example, the application of a relation to a set, and the extrema computation for a set can be used to compute the new extents of a transformed data layout. And, since our code generator is capable of polyhedral optimization, as explained in Sections 3.6 and 3.7, isl is already integrated and in use.

Overview

The overall structure of our layout transformation strategy is straightforward:

(i) Collect all layout transformation statements.

(ii) Apply `transform` statements.

(iii) Perform all `rename` operations.

(iv) Create new fields for `concat` statements.

(v) If there is at least one `concat` statement, create and incorporate accesses to new fields and apply `transform` statements for them.

The first step (i) searches the entire AST for `LayoutTransformations` blocks and collects all their statements. In the second step (ii), all access expressions to fields are updated as specified by the `transform` statements. Excluded are only accesses to fields created with a `concat` statement, since they do not exist at this point but are introduced later in step (v). Additionally, the field extents are adapted to ensure that the linearization is correct and a large enough chunk of memory will be allocated. Details of how a `transform` statement is applied are presented later in this section. Step (iii) performs all requested `rename` operations. Since every field is represented by a single object and these

are not referenced by name in the AST, we can simply replace the `name` attribute of all targeted fields. The **`concat`** statements are processed in steps (iv) and (v). In the former, the new, concatenated fields are generated and some sanity checks are performed. For example, the data type of the field elements must be identical. The dimensionality of the new field is one larger than that of the original fields. The extent of this new dimension is the number of fields to be merged, while for the other dimensions, the maximum of the extents from the old fields is set. Finally, in the last step (v), all accesses to any field that is to be concatenated with others are replaced by an access to the new field and a potential transformation is applied. Additionally, all other occurrences of an original field (in internal structures) are replaced by the new one.

Access Transformation

The basic idea of the automatic data layout transformations is to modify all accesses to a field in a common way. Additionally, the layout information of the fields must be updated, i. e., the extent of each dimension must be adjusted. Algorithm 3.5 computes new field accesses for the specified transformations. Prior to the execution of this function, level specifications introduced in Section 2.2.1, are completely resolved: all objects, including the field accesses and the fields themselves, are specialized for each level. Thus, a field is identified not only by its name but by a combination of its name and level.

Initially, in line 2, the `field` that is referenced by the given `access` expression is extracted. The remainder of the function can then be divided into two parts:

(i) lines 4 to 11 update the field layout and generate a template with which the new, transformed accesses to the same field can be generated, while

(ii) lines 12 to 15 retrieve a previously computed template, specialize it to the given access and simplify the result.

On the one hand, the field layout modification of part (i) must be executed exactly once per field to ensure correctness. On the other hand, the template generation need not be repeated since its result can be cached and reused later. The `accessTemplates` mapping performs such a caching. Therefore, part (i) is executed only if there is no template available yet for the field in question (line 3). It starts by retrieving a list of **`transform`** statements for the current `field` from the `transformStmts` mapping (line 4). For each of these statements, an isl representation of the transformation expression is generated (line 7) and these individual transformations are composed to yield a single one with the same semantics (line 8). isl provides an according

ALGORITHM 3.5: Layout transformation for access expressions.

INPUT: The *access* expression to be transformed and a list of *enclosingConditions* for the current program location.

OUTPUT: The new, transformed field access expression.

DATA: *accessTemplates* is an initially empty mapping of a field to an access template whose content is preserved between different calls and *transformStmts* is a mapping of each field to a list of transformation statements for it.

```
1  FUNCTION process_access(access, enclosingConditions):
2      field ← get_field(access)
3      if field ∉ accessTemplates.keys() then
4          stmts ← transformStmts(field)
5          trafo ← identity()
6          foreach stmt ∈ stmts do
7              aff ← create_isl_multi_aff(stmt)
8              trafo ← aff ∘ trafo
9          update_layout(field, trafo)
10         accTempl ← create_access_template(trafo)
11         accessTemplates(field) ← accTempl
12     accTempl ← accessTemplates(field)
13     newAccess ← specialize(accTempl, access)
14     newAccess ← simplifyWith(newAccess,
        enclosingConditions)
15     return newAccess
```

composition operation. Updating the field layout for the transformation in line 9 also relies on methods provided by isl. In detail, the affine expression representing the layout transformation is first transformed to a relation between two integer sets. Second, an integer set containing all valid indices for the old field layout is created and, third, the previously computed relation is applied to it which results in the set of all valid indices for the new, transformed layout. Finally, it is ensured that the minimal value of every individual dimension of the new integer set (after projecting all other dimensions out) is nonnegative and the new extent of that dimension is set to its maximum value plus 1. Line 10 recreates an AST, namely the template for the transformed access, from the isl transformation expression with predefined dummies for the input variables. This new template is then stored in the *accessTemplates* mapping (line 11) for a later use and to indicate that the field layout is already modified.

In contrast to part (i), part (ii) must be executed for each access expression that has to be transformed. The access template for `field` is retrieved in line 12. This template is specialized in line 13 by replacing the i-th dummy variable with the i-th expression of the old

`access`. Line 14 performs an optimization that targets explicitly a color splitting for a colored loop nest. For example, the loop in Listing 3.11 is executed once for all loop iterations `[x,y,z]` for which the expression `(x+y+z)%2` equals to `0` and once for `1`. If additionally a color splitting, as presented in Listing 3.12, is applied, the modulo computation in the new field access expressions inside these loops can be specialized to the constant `0` or `1`, respectively. The information to which value the coloring expression is restricted in the current context is given via the `enclosingConditions` parameter. This simplification tackles the more complex memory address computations introduced by a color splitting and, thus, has the potential to increase the performance of the generated code even further. The transformed and simplified access is eventually returned in line 15.

This algorithm is likely to generate some unoptimized arithmetic expressions. However, we refrain from an explicit optimization step at this point, since this would unnecessarily increase the code generation time dramatically: all accesses are linearized in a later step of the code generation process and this linearization may reveal further optimization opportunities. Hence, the simplifications presented in Section 3.2 are applied after linearization.

4 EVALUATION

The optimizations presented in Chapter 3 have been implemented in the ExaStencils code generator and their evaluation is presented and discussed throughout this chapter. Since the set of optimizations is quite diverse and some of them target very specific aspects, they are evaluated one after the other.

Section 4.1 starts with a description of the available software and hardware for our experiments and Section 4.2 provides a brief overview over the kernels and applications. The extensive but detailed evaluation of the most complex optimizations implemented in the ExaStencils code generator are presented in Section 4.3: the polyhedral transformations and, especially, the polyhedral search space exploration. Section 4.4 targets both an address precalculation and a vectorization. The redundancy elimination approaches and data layout transformations are evaluated in Sections 4.5 and 4.6, respectively, and Section 4.7 closes this chapter with a brief summary.

A function inlining and the arithmetic normalizations are not evaluated separately, since several other code optimization and even generation strategies depend on them. E. g., the arithmetic normalizations are frequently used to simplify newly generated expressions. This reduces the complexity of several strategies and, in turn, increases their maintainability. Thus, they cannot be disabled entirely.

We would like to stress that the ExaStencils code generator was and still is under active development. Therefore, the performance numbers presented here differ in some cases from those published previously, but the conclusions drawn in previous work are still valid.

Also note that, when referring to byte quantities, we do *not* use SI prefixes: K, M, and G refer to the factors 2^{10}, 2^{20}, and 2^{30}, respectively. For example, 1 MB is exactly 1 048 576 bytes.

4.1 SETUP OF THE EXPERIMENT

This section provides information on the hardware and software setup for all experiments. This includes the hardware specifications of all machines, their system environments, the programming languages, and compilers. We also discuss some special properties of our test platforms.

Table 4.1: Machine overview.

Machine	Workstation	Chimaira	Pontipine
CPU	Core i7-4770	Xeon E5-2690 v2	Xeon E5-2620 v4
Nodes	1	17	12
Interconnect	—	10GbE	1GbE
Sockets	1	1	2
NUMA	No	No	Yes
Memory Controller	2	4	$2 \cdot 4$
RAM	32 GB	64 GB	$2 \cdot 128$ GB
	DDR4 2133	DDR3 1866	DDR4 2133
Stream Bandwidth	22.0 GB/s	45.2 GB/s	90.1 GB/s

Table 4.2: CPU architectures.

Processor	Core i7-4770	Xeon E5-2690 v2	Xeon E5-2620 v4
Architecture	Haswell	Ivy Bridge EP	Broadwell EP
Cores/Threads	4/8	10/20	8/16
Base Frequency	3.4 GHz	3.0 GHz	2.1 GHz
Turbo (all cores)	3.7 GHz	3.3 GHz	2.3 GHz
Turbo (1 core)	3.9 GHz	3.6 GHz	3.0 GHz
L3 Cache	8 MB	25 MB	20 MB
Vector Ext.	AVX2 & FMA	AVX	AVX2 & FMA
DP Peak	236.8 GFLOP/s	264.0 GFLOP/s	294.4 GFLOP/s

4.1.1 Hardware

All evaluations were conducted on one of three different platforms we can access. A short overview of the machines and their respective CPU architectures is provided in Tables 4.1 and 4.2. One machine is a simple workstation that was used primarily to generate and compile all experiments. The experiments were then conducted on two clusters named Chimaira and Pontipine.

CPUs

Chimaira consists of 17 nodes, each equipped with an Intel Xeon E5-2690 v2 CPU, 64 GB DDR3 1866 RAM and interconnected via 10 Gbit/s Ethernet. The processor is based on the Ivy Bridge EP architecture and it supports Intel's Advanced Vector Extensions (AVX) but, in contrast to the other CPUs, not the fused multiply-add (FMA) extension. I. e., each of the ten cores can issue an addition and an independent multiplication instruction per clock cycle. Their operands are 256 bit registers that can contain either four double-precision or eight single-precision values. Thus, with a clock frequency of 3.3 GHz for all ten cores, its double-precision peak performance is

$$3.3\,\text{GHz} \cdot 10 \cdot 4 \cdot 2\,\text{FLOP} = 264.0\,\text{GFLOP/s}.$$

While the double-precision peak performance (line "DP Peak") is a computed theoretical peak, the memory bandwidth (line "Stream Bandwidth") is measured with a streaming benchmark from the likwid tools [88]. A performance close to the processor's theoretical peak compute performance can be achieved by a very specialized binary, but the main-memory bandwidth is usually not as close. Therefore, we provide benchmark results instead of a theoretical value.

The more recent Intel architectures, starting with Haswell, also support fused multiply-add instructions of the form $(a \times b) + c$. Since two of these instructions can be issued per cycle, the peak floating-point performance per clock cycle and core is twice as high as with previous architectures. Both our standard workstation and the second cluster, Pontipine, support these instructions. But, while Pontipine's raw compute performance is higher than Chimaira's, the former's 12 nodes are connected via Gigabit Ethernet only. Another specialty of Pontipine is its non-uniform memory access (NUMA): each node powers two processors with independent memory controllers and memory modules that are directly connected to one processor. Thus, each processor manages its own part of the memory, which is called a *memory domain*. Software can access data in both memory domains seamlessly and independently of the CPU on which it runs. Due to the abstract virtual memory, it is even unaware of where data is located physically. However, accesses to the memory domain of another processor must be routed through that processor, which increases access times and reduces bandwidth compared to the local memory domain.

GPUs

16 Chimaira nodes are also equipped with an NVIDIA GeForce GTX Titan Black GPU. It has 2 880 cores and 6 GB of GDDR5 memory. In default mode, its base clock rate is 889 MHz, with a boost frequency of 980 MHz. However, in this operating mode, the peak performance for double-precision computations is cut down to a mere 24th fraction of the single-precision performance. As a remedy, the Titan Black features a special mode in which the ratio from single- to double-precision performance is 1:3, but the maximum clock frequency is reduced to 677 MHz. This mode is always active in our environment. Unfortunately, this GPU does not support GPUDirect[1] and, thus, data exchanges between different nodes have to be performed by the CPU. I. e., both sender and receiver must transfer the communicated data between the host and the device memory, which imposes an additional overhead.

1 https://developer.nvidia.com/gpudirect

Table 4.3: Software used in the evaluation.

Operating System	64-bit Debian 9, Linux kernel 4.9
Java Virtual Machine	OpenJDK 1.8 with Scala 2.12
C/C++ Compiler	icc 19.0 (gcc 8.3, clang 8.0)
CUDA Compiler	CUDA Toolkit 8.0 (requires gcc 4.9)
MPI Library	OpenMPI 2.1 (MPI API 3.1)

4.1.2 Software

The software environment installed on all machines for the evaluation is listed in Table 4.3. Java and Scala are required for the ExaStencils code generator but not for the emitted applications. The CPU code is C++, which is compiled with the Intel compiler version 19, if not specified otherwise. We always request the most aggressive optimizations the target compiler provides (`-O3`) and we allow it to generate specialized code for the target hardware (e. g., via `-xCORE-AVX2`). If vectorization is enabled, we configure our code generator to replace calls to math functions by their vectorized versions provided by the compiler-independent library *vecmathlib*[2].

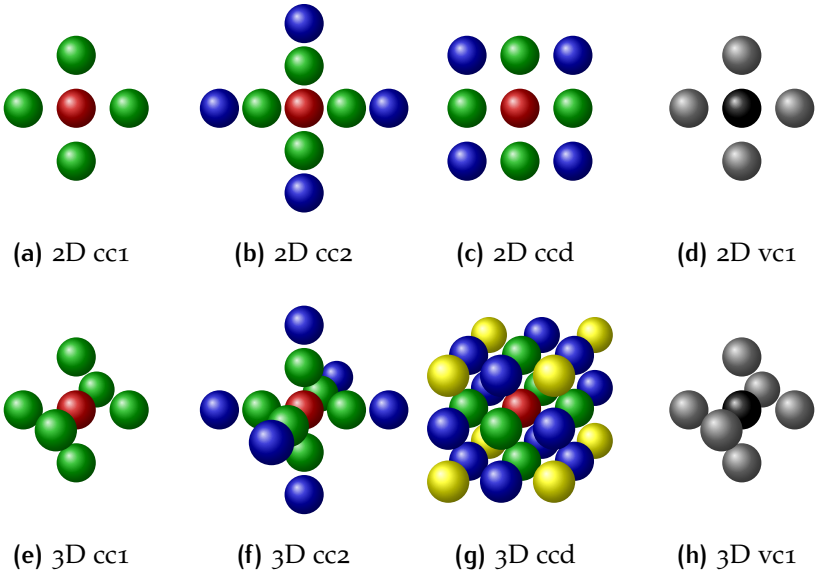
Shared-memory parallelism of multicore processors is leveraged with OpenMP while, for distributed memory, the MPI implementation OpenMPI 2.1 is employed. The GPU experiments involve CUDA code, which is passed to the CUDA compilation tools version 8.0. The host code, i. e., the CPU code, is again compiled by the Intel compiler. CUDA is a parallel computing platform and programing model specifically developed to exploit the compute capabilities of NVIDIA GPUs.

As an alternative to CUDA, there is the open standard OpenCL, developed by the Khronos Group. It is not as commonly used as CUDA, since compiler and tools for OpenCL are not as mature. However, a drawback of CUDA is that it is tied to NVIDIA GPUs only, while OpenCL is supported by many other vendors, too, such as AMD and Intel. But, according to the TOP500 list as of November 2018, NVIDIA GPUs are the most common accelerators, which is why the ExaStencils code generator has been focusing on CUDA.

4.2 EVALUATED CODES

The optimizations implemented in the ExaStencils code generator are quite diverse and, thus, different codes are needed to evaluate them. For example, the polyhedral search space exploration is meant to apply a suitable time tiling to multiple iterations of a stencil. Therefore, it is reasonable to focus on the smoothing components of a multigrid

2 https://bitbucket.org/eschnett/vecmathlib/wiki/Home



(a) 2D cc1 (b) 2D cc2 (c) 2D ccd (d) 2D vc1

(e) 3D cc1 (f) 3D cc2 (g) 3D ccd (h) 3D vc1

Figure 4.1: 2D and 3D stencil shapes.

method, as introduced in Section 1.2. But an evaluation on an entire application should not be neglected either.

4.2.1 Smoothers

The first group of codes for our evaluations are smoothers, i. e., classical iterative methods such as Jacobi or Gauss-Seidel. The shapes of the stencil codes are illustrated in Figure 4.1. Four are two-dimensional, four are three-dimensional. The colored variants represent stencils with constant coefficients, the gray ones are stencils with variable coefficients. In the latter case, each neighbor gets its own coefficient. The colors in the constant coefficients stencils correspond to the values of the coefficients in the generated code: neighbors with the same color share the same coefficient. We use 'ccX' and 'vcX' as abbreviations for constant, respectively, variable coefficient stencils, where 'X' specifies either the radius ('1' or '2') or whether the stencil is dense ('d'), in which case the radius is 1.

The ExaSlang 4 code of a Jacobi smoother is shown in Listing 4.1. The definition of the discretized operator `Laplace` determines the stencil shape. Note that not only two slots of the field `Solution` are accessed but also a field `RHS` from which one element per loop iteration is fetched.

In addition to the Jacobi smoothers, colored Gauss-Seidel smoothers were part of the experiments for all presented stencil shapes with radius 1, including the dense. The versions cc1 and vc1 get along with two colors while the two-dimensional dense version requires four and its three-dimensional equivalent eight colors. Listing 4.2 is the ExaSlang 4 code of a 3D RBGS stencil. In contrast to the Jacobi

Listing 4.1: ExaSlang 4 code for a Jacobi smoother.

```
Function Smoother {
  loop over Solution {
    Solution[next] = Solution[active] +
      0.8 / diag(Laplace) *
      (RHS - Laplace * Solution[active])
  }
  advance Solution
}
```

stencils, the coefficient of the center element for the colored stencils we evaluate simplifies to 0, i. e., the red or black marked elements in Figure 4.1 are not read during the computation. This can be seen in the ExaSlang 4 code: after factoring `Solution` out, the coefficient of its center element is

$$1.0 + 1.0/\texttt{Laplace}[0,0] \cdot (-\texttt{Laplace}[0,0]) = 0$$

where $\texttt{Laplace}[0,0]$ denotes the center element of the stencil. Thus, no load instruction for the corresponding element of the grid is generated, not even in the variable coefficients versions.

4.2.2 Applications

Besides isolated smoothers, three different multigrid applications have been selected for the evaluation of our optimizations. Their current implementation in ExaSlang 4 was developed by our colleague Sebastian Kuckuk[3] from the Friedrich-Alexander-Universität Erlangen-Nürnberg, who has been one of the main developers of the ExaStencils code generator. We made only minor modifications of his code, such as an adaption of the problem sizes.

3 https://www.cs10.tf.fau.de/person/sebastian-kuckuk/

Listing 4.2: ExaSlang 4 code for a RBGS smoother.

```
Function Smoother@(all but coarsest) {
  color with {
    (i0+i1+i2) % 2,
    loop over Solution {
      Solution = Solution +
        1.0 / diag(Laplace) * (RHS - Laplace * Solution)
    }
  }
}
```

Poisson

The first application is the well-known model problem given by Poisson's equation:

$$\begin{aligned} -\Delta u &= f \text{ in } \Omega, \\ u &= g \text{ on } \delta\Omega \end{aligned}$$

Different versions of 2D and 3D multigrid solvers for this equation were generated automatically, based on a finite-difference discretization of the equation on a d-dimensional hypercube $\Omega = (0,1)^d$. The V-cycles had three pre- and three post-smoothing RBGS steps with constant coefficients. Termination occurs when the L_2 norm of the initial residual has been reduced by a factor of 10^5.

Optical Flow

A second, more complex application is a multigrid version of an optical flow detection method [42]. Here, an approximate motion within an image sequence is computed based on gray value respectively color intensity changes. The termination criterion for the solver was the same as in the previous application. Details can be found in the literature on this application and an earlier version of the implementation [82] as well as on multigrid methods for optical flow [9, 44]. Since the motion, or flow, of a single pixel consists of multiple coordinates, namely one per dimension, the element type of the corresponding field is a vector, not a scalar. This application was chosen to evaluate the data layout transformations in Section 4.6.

Fluid Flow

The third and final application is a simulation of non-isothermal and non-Newtonian fluid flows. Such fluids usually consist of suspensions of particles or macromolecules and they can occur as gels, pastes, or foams. Relevant examples include organic fluids such as blood, food products such as fruit juice, and industrial fluids such as drilling fluids and mining pulps. These fluids are important in both academia and industry. Details on the implementation can be found elsewhere [51]. We make use of this application in Section 4.5 to demonstrate the effectiveness of our redundancy elimination techniques.

4.3 POLYHEDRAL SEARCH SPACE EXPLORATION

Let us start with an evaluation of the polyhedral optimizations. Its most complex part is the polyhedral search space exploration, introduced in Section 3.7, on which we focus in this section. A detailed evaluation of an extensive exploration is given in Sections 4.3.1 to 4.3.4.

Listing 4.3: Color splitting for the RBGS 3D cc1 cs exploration.

```
LayoutTransformations {
  transform Solution, RHS
    with [x, y, z] => [x/2, y, z, (x+y+z)%2]
}
```

Section 4.3.1 presents the setup of the experiment. Statistics of the entire exploration are given in Section 4.3.2. An overview of the exploration results and a comparison with other algorithms and tools can be found in Section 4.3.3 and Section 4.3.4 presents an in-depth analysis of the exploration results for some experiments.

Additional experiments to evaluate our exploration can be found in Sections 4.3.5 to 4.3.7. The influence of Intel's vectorizer on our exploration results is illustrated in Section 4.3.5. Section 4.3.6 analyzes the results of an exploration on a system with a NUMA architecture. The quirks of multiple memory domains and their influence on performance optimizations for bandwidth-bound codes are also portrayed. Finally, the benefits of our exploration for a complete multigrid cycle are presented in Section 4.3.7.

4.3.1 Setup of the Experiment

We evaluated our polyhedral search space exploration on twelve different stencils. Eight are the Jacobi stencils introduced in Section 4.2.1. The remaining four are the two- and three-dimensional RBGS kernels with constant (cc1) and variable coefficients (vc1). These four stencils were evaluated twice: with and without a data layout transformation to perform a color splitting. The former are labeled 'cs'. Listing 4.3 is the data layout transformation code for the 3D RBGS kernel with constant coefficients. In the variable-coefficient variant, the field holding said coefficients is transformed the same way. 2D layout transformations are analogous.

All experiments presented in Sections 4.3.3 and 4.3.4 were executed on our cluster Chimaira. The code generator was configured to add OpenMP pragmas for parallelization and to emit vectorized code for double-precision computations using AVX intrinsics. We disabled the autovectorizer of the Intel compiler explicitly in all activations of our code generator. This is due to the fact that, in some cases, it altered the execution order automatically by, e. g., reversing a loop, to be able to vectorize it, which effectively resulted in a different schedule. We did not encounter a situation in which the vectorizer of icc was able to generate a noticeably faster code than our vectorizer, which additionally justifies focusing on the latter. A more detailed comparison of the exploration results with and without icc's vectorizer are presented in Section 4.3.5. Other optimizations of our code generator, such as

Table 4.4: Exploration time and properties of all experiments.

	Jacobi 3D				RBGS 3D	
	cc1	cc2	ccd	vc1	cc1	vc1
expl. time filter lvl 0	27.0 s	23.6 s	49.2 s	15.1 s	48.5 s	20.8 s
expl. time filter lvl 7	0.64 s	0.68 s	0.75 s	0.36 s	1.15 s	0.49 s
Chernikova run time	0.22 s	0.30 s	0.24 s	0.10 s	0.52 s	0.17 s
#time steps	5	4	3	3	4	2
#search space dims	50	40	30	30	80	40
#rays	26	25	22	24	29	25
#comb. rays for lvl 0	2	2	3	2	2	2
#schedules filter lvl 0	12 048	12 048	21 872	12 048	12 048	12 048
#schedules filter lvl 7	4	4	6	4	4	4

	Jacobi 2D				RBGS 2D	
	cc1	cc2	ccd	vc1	cc1	vc1
expl. time filter lvl 0	0.55 s	0.66 s	0.72 s	0.41 s	1.01 s	0.92 s
expl. time filter lvl 7	0.31 s	0.39 s	0.40 s	0.26 s	0.59 s	0.51 s
Chernikova run time	0.14 s	0.22 s	0.19 s	0.11 s	0.38 s	0.34 s
#time steps	5	5	5	4	5	5
#search space dims	35	35	35	28	70	70
#rays	18	18	18	17	23	23
#comb. rays for lvl 0	3	3	3	3	3	3
#schedules filter lvl 0	108	108	112	108	108	108
#schedules filter lvl 7	1	1	1	1	1	1

address precalculation and a rectangular tiling, were applied to all versions generated.

We focus on CPUs only. An evaluation as part of a supervised master thesis [98] revealed that the presented time tiling techniques are not beneficial for GPUs. Custom techniques necessary for such accelerators [35] remain future work.

4.3.2 Exploration Statistics

For the three-dimensional experiments, each stencil application updates 512^3 double-precision elements, while $16\,384^2$ values are computed in the two-dimensional case. The times required to perform the search space generation and exploration, the number of time steps per experiment as well as some other statistics about the search space are presented in Table 4.4. The statistics for the color splitting version are not stated explicitly since the exploration is not affected by the layout transformation: the latter is applied after the polyhedral techniques. The exploration time of filter levels 0 and 7 contain the dependence analysis, the search space generation, the Chernikova call and the actual schedule enumeration. In the three-dimensional case without any filter applied, not more than one minute was required on our workstation, using a single thread (the exploration itself is not

parallelized). The filters are applied after Chernikova is called, so its run time does not depend on the filter level. It completes in less than one second, while the majority of the run time is spent in function `guided_exploration_tileable` of Algorithm 3.4. For filter level 7, the exploration time is significantly lower than the code generation and optimization time required for a single variant, which is usually in the range of 10 to 30 seconds for the presented experiments. In summary, for every filter level, the most time-consuming part is not the exploration itself but the code generation, compilation, and evaluation of all variants.

The maximum number of rays combined for filter level 0 is 3 for 2D and 2 for 3D. The only exception is the dense 3D stencil. According to the higher number of constraints for a legal schedule due to the stencil shape, we succeeded in the full exploration of the sets generated for up to three rays combined. For a filter level of 2 and higher, we always combined three rays. However, at level 7, all additional schedules that originated from combining three rays were removed by the filters.

4.3.3 Exploration Overview

A performance comparison of our exploration with other algorithms and tools is shown in Table 4.5. For every experiment, the absolute performance of the fastest version is presented in both million lattice updates per second (MLUP/s) and billion floating-point operations per second (GFLOP/s). The performance of the different algorithms and tools is then given as the percentage of the best variant. Remember that, when computing a new element for a stencil application, not only the neighbors, as specified by the stencil shape, are fetched from the memory but one additional value is read from a separate array, as presented in Section 4.2.1. Since this array is not modified, it does not cause any additional data dependences but it must be taken into account when comparing the performance results with other work.

Each experiment was subject to an individual tile size exploration; see Table 4.6. The leftmost entry of each list is the tile size for the outer loop, the rightmost is for the innermost loop. PolyMage autotunes the tile size automatically and RBGS stencils cannot be diamond-tiled, so they do not appear here. For each experiment, we evaluated every combination of tile sizes from the set {4, 8, 10, 16, 20, 32, 50, 64, 100, 128, 150, 200, 256, ∞} for all but the innermost dimension. "∞" indicates that a value larger than the iteration domain for this dimension was chosen. For the innermost dimension, values less than 100 were removed to keep the number of combinations manageable. Exploring the tile size and the schedule together would increase the number of tests by a factor of roughly 1 000 in 3D, so the evaluated exploration techniques use the same tile size as the isl heuristics experiment. The tile size exploration for the latter revealed that the

Table 4.5: Performance comparison for the exploration.

	Jacobi 3D				RBGS 3D			
	cc1	cc2	ccd	vc1	cc1	vc1	cc1 cs	vc1 cs
best [MLUP/s]	4 537	2 702	1 874	1 040	3 347	521	4 803	719
best [GFLOP/s]	45.4	45.9	60.0	17.7	26.8	7.3	38.4	10.1
baseline	33%	56%	81%	53%	30%	58%	32%	77%
Exploration								
filter level 0	98%	97%	96%	99%	97%	98%	100%	94%
filter level 2	98%	99%	95%	100%	75%	99%	100%	94%
filter level 7	98%	96%	95%	97%	75%	95%	100%	93%
tile opt.	100%	100%	99%	100%	100%	100%	100%	100%
isl								
simple	10%	11%	6%	31%	22%	55%	15%	44%
heuristics	95%	96%	6%	100%	22%	53%	15%	44%
PLuTo								
rectangular	69%	84%	89%	90%	8%	38%	6%	31%
unrolled	51%	38%	49%	53%	72%	67%	32%	57%
diamond	71%	84%	100%	90%	—	—	—	—
PolyMage	49%	56%	66%	48%	29%	47%	—	—
Polyite	31%	48%	67%	50%	—	—	—	—

	Jacobi 2D				RBGS 2D			
	cc1	cc2	ccd	vc1	cc1	vc1	cc1 cs	vc1 cs
best [MLUP/s]	5 997	4 816	5 327	2 126	4 709	1 249	5 283	1 838
best [GFLOP/s]	48.0	62.6	69.3	27.6	28.3	12.5	31.7	18.4
baseline	25%	31%	28%	32%	21%	30%	29%	37%
Exploration								
filter level 0	84%	91%	86%	79%	67%	99%	99%	97%
filter level 7	84%	91%	84%	79%	67%	99%	99%	97%
tile opt.	84%	91%	86%	79%	67%	100%	100%	100%
isl								
simple	8%	7%	6%	11%	15%	13%	12%	10%
heuristics	84%	91%	6%	79%	14%	13%	12%	10%
PLuTo								
rectangular	50%	63%	53%	46%	11%	16%	10%	12%
unrolled	66%	56%	52%	77%	100%	90%	66%	58%
diamond	77%	86%	83%	85%	—	—	—	—
PolyMage	100%	100%	100%	100%	56%	86%	—	—
Polyite	24%	29%	38%	32%	—	—	—	—

best tile sizes have a comparable performance with differences of only few percentage points. Thus, with regard to the exploration, we refrain from tiling the innermost and the outermost dimension. Tiling the innermost dimension potentially impairs the hardware prefetcher, which fetches data from main memory in advance to hide memory latency. Tiling the outermost dimension is not necessary either, since only few consecutive lines in 2D or planes in 3D are reused. I. e., we can simply stream through the outer dimension.

Baseline

As a baseline for our experiments, we chose the roofline performance of a single time step, i. e., without any kind of time tiling and with a barrier synchronization between stencil applications. In this case, the performance depends solely on the memory bandwidth. E. g., for all Jacobi versions with constant coefficients and a spacial tiling, one lattice update amounts to four double-precision values to be transferred. Three are required by the computation: one element is loaded from both input arrays (all others are assumed to be in the processor's cache) and one updated value is written to memory. The remaining one is due to the write-allocate for the update, since a cache line has to be loaded before it be can be modified. Therefore, the roofline for one Chimaira node is:

$$\frac{45.2 \cdot 2^{30}\,\mathrm{B/s}}{4 \cdot 8\,\mathrm{B/LUP}} \approx 1\,517\,\mathrm{MLUP/s}$$

In most cases, a performance close to the roofline without a temporal reuse between subsequent time steps can be achieved by choice of a suitable tile size.

Exploration

The two, respectively, three subsequent rows in Table 4.5 show the performance of the best among the explored schedules for the given filter level. The three-dimensional case involved, for all but the dense stencil, two exploration runs. In the first run, we configured the guided exploration to combine up to two rays only, along with a filter level of 0, i. e., no filter applied. The second run combined up to three rays with a filter level of 2. This results in larger sets of schedules before the filters are applied. Finally, we optimized the tile sizes for the best explored variants by testing all combinations mentioned previously. The performance values and tile sizes for these "tile opt." versions are given in Tables 4.5 and 4.6, respectively. For the 3D experiments, no other tool or algorithm tested was able to generate a noticeably faster code than our exploration. The same is true for the RBGS stencils with color splitting enabled. A more in-depth analysis of the most interesting results is presented in Section 4.3.4.

Table 4.6: Tile sizes for the exploration experiments.

	Jacobi 3D			
	cc1	cc2	ccd	vc1
expl. / Polyite	$\infty, 128, \infty$	$\infty, 128, \infty$	$\infty, 100, \infty$	$\infty, 100, \infty$
expl. tile opt.	$\infty, 150, \infty$	$\infty, 150, \infty$	$\infty, 200, \infty$	$\infty, 100, \infty$
isl				
simple	$200, 128, \infty$	$150, 150, \infty$	$32, 256, \infty$	$\infty, 64, \infty$
heuristics	$\infty, 128, \infty$	$\infty, 128, \infty$	$\infty, 100, \infty$	$\infty, 100, \infty$
PLuTo				
rectangular	$100, 4, \infty$	$100, 4, \infty$	$100, 10, \infty$	$50, 4, \infty$
unrolled	$16, 16, \infty$	$10, 10, \infty$	$4, 4, \infty$	$8, 8, \infty$
diamond	$20, 20, 4, 150$	$100, \infty, 4, \infty$	$8, 8, 4, \infty$	$50, \infty, 4, \infty$

	RBGS 3D			
	cc1	vc1	cc1 cs	vc1 cs
expl. / Polyite	$\infty, 150, \infty$	$\infty, 64, \infty$	$\infty, 150, \infty$	$\infty, 64, \infty$
expl. tile opt.	$\infty, 200, \infty$	$\infty, 100, \infty$	$\infty, 150, \infty$	$200, 100, \infty$
isl				
simple	$64, 100, \infty$	$\infty, 52, \infty$	$\infty, 150, \infty$	$\infty, 64, \infty$
heuristics	$\infty, 150, \infty$	$\infty, 64, \infty$	$\infty, 150, \infty$	$\infty, 64, \infty$
PLuTo				
rectangular	$10, 128, \infty$	$16, 8, \infty$	$64, 16, \infty$	$20, 4, \infty$
unrolled	$20, 20, \infty$	$8, 4, 100$	$8, 10, \infty$	$8, 8, \infty$

	Jacobi 2D			
	cc1	cc2	ccd	vc1
expl. / Polyite	∞, ∞	∞, ∞	∞, ∞	∞, ∞
expl. tile opt.	∞, ∞	$52, \infty$	$256, \infty$	$200, \infty$
isl				
simple	$150, \infty$	$8, \infty$	$16, \infty$	$150, \infty$
heuristics	∞, ∞	∞, ∞	∞, ∞	∞, ∞
PLuTo				
rectangular	$256, 200$	$256, 150$	$256, 150$	$10, \infty$
unrolled	$4, 150$	$8, 100$	$4, 150$	$4, 256$
diamond	$150, 150, 128$	$100, 100, 128$	$50, 50, 200$	$50, 50, 100$

	RBGS 2D			
	cc1	vc1	cc1 cs	vc1 cs
expl. / Polyite	∞, ∞	∞, ∞	∞, ∞	∞, ∞
expl. tile opt.	$100, \infty$	$64, \infty$	$26, \infty$	$8, \infty$
isl				
simple	$128, \infty$	∞, ∞	$52, \infty$	$13, \infty$
heuristics	∞, ∞	∞, ∞	∞, ∞	∞, ∞
PLuTo				
rectangular	$4, \infty$	$4, \infty$	∞, ∞	$16, \infty$
unrolled	$20, 128$	$150, 100$	$16, 150$	$4, 200$

isl Scheduler

Both isl versions have been integrated directly in the ExaStencils code generator and use the scheduler implemented in isl version 0.18, which is a variant of the PLuTo algorithm. The difference between experiments "isl simple" and "isl heuristics" is that the latter features our modifications to its input as described in Section 3.6.2. We did not modify any other settings of the isl scheduler, since we did not expect any improvement.

The performance of isl simple is always below the baseline, which renders it unusable for stencil codes. But, if our heuristics kicks in, which is the case for the non-dense Jacobi stencils only, the determined schedule is very promising. Otherwise, the heuristics is without effect.

For comparison, the schedule computed for the classical Jacobi 3D cc1 without the heuristics reads

```
{ S[i,j,k] -> [i,   j,   k,   0];
  T[i,j,k] -> [i+1, j+1, k+1, 1]; ... }
```

while the schedule with it is

```
{ S[i,j,k] -> [i,   i+j,   i+j+k,   0];
  T[i,j,k] -> [i+1, i+j+1, i+j+k+1, 1]; ... }
```

The best performing schedule in the exploration is similar to the one with the heuristics:

```
{ S[i,j,k] -> [j,   i+j,   i+j+k,   0];
  T[i,j,k] -> [j+1, i+j+1, i+j+k+1, 1]; ... }
```

PLuTo Algorithm

"PLuTo rectangular" and "PLuTo diamond" show the performance of the schedule detected by PLuTo version 0.11.4[4] with and without diamond tiling [4, 7] used for a rolled-up time loop, while "PLuTo unrolled" shows the performance for the time loop unrolled. Remember that the RBGS experiments exclude diamond tiling. The options enabled in all experiments are `--tile --parallel`. For the rolled-up experiments, we switched to a different frontend with the `--pet` option. This enabled us to select the input and output arrays via the modulo computations `t%2` and `(t+1)%2` for time step `t`. Diamond tiling was switched on with `--partlbtile` and, for the RBGS experiments, we also had to add the option `--lastwriter` to prevent PLuTo from crashing due to a constraint explosion. We also evaluated PLuTo+ [1], which effectively removes a restriction of the original PLuTo algorithm and allows negative schedule coefficients, but the results were no different. For a precise comparison, we gave the schedules computed

4 Git revision `8606c3d` on https://github.com/bondhugula/pluto.git

by PLuTo to our code generator and selected the exact same set of optimizations.

For an unrolled time loop, PLuTo computes the same schedules as the isl scheduler without the heuristics. The difference is that PLuTo performs an additional scheduling step after tiling. For Jacobi 3D cc1, the schedule computed by the PLuTo algorithm with a rolled-up time loop before tiling is:

```
{ S[t,i,j,k] -> [t, t+i, t+j, t+k] }
```

The time dimension is the outermost schedule dimension while, in the unrolled versions, the time steps are enumerated innermost. This is the case in all experiments. Overall, PLuTo performs quite well and, in one experiment (RBGS 2D cc1), it is even able to outperform the other tools. While the computed schedule for this experiment was also detected by our exploration, we did not perform a separate tile scheduling. However, once we activate color splitting, the best schedules explored by our code generator take the lead. The reason that diamond tiling performs worse than reported by others [7] could be the very small number of time steps, which is less than 6 in all our examples.

PolyMage

PolyMage [59] is a combination of a DSL and an optimizing code generator for image processing. Since stencil codes are also part of its domain, a comparison is relatively easy. PolyMage searches for the best tile sizes in a given set of candidates and also evaluates different grouping options for the statements. According to the tile sizes, the same set as for the other tools was supplied. The Intel compiler served to generate the final binary and perform low-level optimizations. Note that, in contrast to the PLuTo experiments, the ExaStencils code generator was not used here and PolyMage is unable to apply a color splitting automatically. Therefore, the corresponding entries in Table 4.5 are left blank. For the Jacobi 2D stencils, PolyMage is able to create faster versions than our exploration detected. This is due to the fact that PolyMage applies overlapped tiling [49], which reduces the number of synchronization points by replicating some computations for different tiles. If overlapped tiling is considered part of the schedule, the latter is no longer a function since some statement instances are mapped to multiple time steps. Such a schedule cannot be detected with our exploration by design: we only explore bijective schedules. Also, we did not implement any tiling techniques other than a rectangular tiling so far.

Polyite

Another external tool we compared against is the polyhedral search space exploration tool Polyite [31][5] built for Polly/LLVM. In contrast to our exploration, it is not domain-specific and uses a genetic algorithm to traverse the search space. We chose the same tile sizes as for our exploration. Unlike all other experiments, a different target compiler had to be used here, which may have had an impact on the performance. E. g., LLVM was not able to emit vectorized code. In principle, our code generator could be made a backend of Polyite, but this would be an extensive effort and we do not expect a genetic algorithm to perform better than our approach in the stencil domain. The reason is that, as shown in Section 4.3.4, the number of good schedules is extremely small and the majority performs equally bad. A general-purpose genetic search is likely to require significantly more exploration time to find the best variant.

The optimum found by Polyite for Jacobi 3D cc1 is the schedule

```
{ S[t,i,j,k] -> [t, 3t + i, -t + 3j, -i + k] }
```

The missing performance numbers for the RBGS experiments are due to a timeout in Polyite when the initial population is created. However, we expect a similar result as for the Jacobi stencils.

This completes our explanations of the rows in Table 4.5. We continue with further discussions of the most interesting exploration results by column.

4.3.4 Exploration Details

Jacobi 3D cc1

Let us first look at the leftmost column: an experiment with a sequence of five iterations of a 3D Jacobi stencil with radius 1 and constant coefficients. We performed two runs. In the first, we configured the guided exploration to combine up to two rays, along with a filter level of 0, i. e., no filter applied. The second combined up to three rays with a filter level of 2. Table 4.7 reveals how many schedules at each filter level were explored in the two versions.

Without any filter, the combination of two rays leads to 12 048 schedules, while only four remain at the highest filter level. Column %Opt contains the speed of the worst and the best version for the given filter level compared to the overall best version found by the exploration. In this experiment, one of the fastest schedules is still among the four remaining at the highest filter level. The performance distribution for all filter levels with two rays combined is shown in Figure 4.2. For each level, there are both a jittered scatterplot and a violin plot. A jittered scatterplot contains every data point with a

5 Git revision `fbf02eb` on https://github.com/stganser/polyite.git

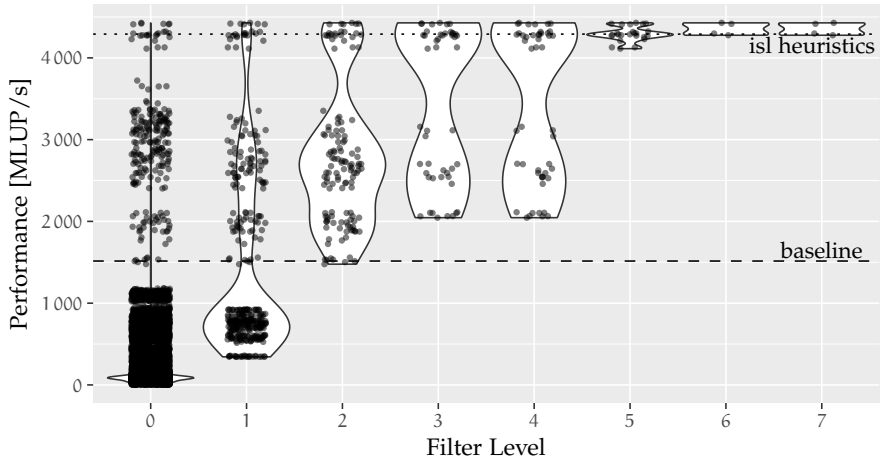


Figure 4.2: Performance distribution of all filter levels for Jacobi 3D cc1.

small, random horizontal shift to disentangle them. In a violin plot, the width of the violin represents the density at this performance value. Thus, the wider the violin at a given position, the more schedules with the respective performance were found. Additionally, the dashed line represents the baseline for a single time step, as explained earlier. The dotted line shows the performance of the schedule computed by isl with the previously mentioned heuristics. This is the best our code generator can do without an exploration. For filter level 0, the vast majority (11 515, i. e., more than 95%) of all schedules lead to a very poor performance—even worse than an untiled and unoptimized version, which achieves 975 MLUP/s. The first two filters already reduce the set of explored schedules to 128, and none of these results in a performance noticeably worse than the baseline. With all filters applied, not more than four remain. At this point, it is sufficient to select the first explored schedule and to skip the evaluation phase completely, since all perform very well.

Table 4.7: Number of schedules and their performance range for each filter level for Jacobi 3D cc1.

	combine two rays		combine three rays	
Level	#Schedules	%Opt	#Schedules	%Opt
0	12 048	0 - 100	152 592	
1	368	8 - 100	22 832	
2	128	33 - 100	496	25 - 100
3	48	46 - 100	80	46 - 100
4	48	46 - 100	80	46 - 100
5	24	92 - 100	40	92 - 100
6	4	96 - 100	6	96 - 100
7	4	96 - 100	4	96 - 100

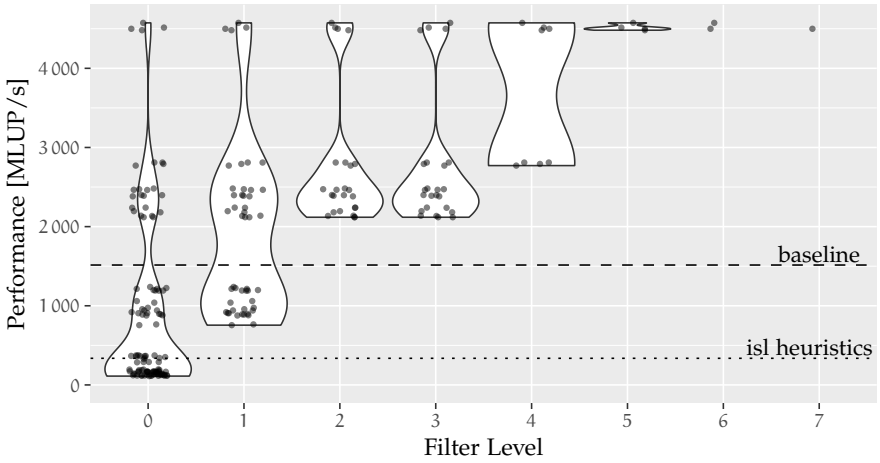


Figure 4.3: Performance distribution of all filter levels for Jacobi 2D ccd.

The combination of three rays leads to an order of magnitude more schedules at filter level 0. Evaluating all of them would have been far too time consuming, so we decided to start at filter level 2 to test if there are even better schedules. Almost four times as many schedules pass this level, but none of them is an improvement. With all filters, again, the same set of four schedules remains. This is also true for the other eleven stencils tested: at filter level 7, it does not matter whether two or three rays are combined.

Jacobi 2D ccd

This experiment covers a sequence of five iterations of a 2D dense Jacobi stencil with radius 1 and constant coefficients. For a 2D stencil, only two linearly independent schedule dimensions are required. This results in a significantly smaller number of explored schedules: for three combined rays, an exploration at filter level 0 results in only 112 schedules, as shown in Table 4.8. Thus, we skipped an exploration with only two rays combined.

Table 4.8: Number of schedules and their performance range for each filter level for Jacobi 2D ccd.

	combine three rays	
Level	#Schedules	%Opt
0	112	2 - 100
1	48	17 - 100
2	24	46 - 100
3	24	46 - 100
4	8	61 - 100
5	4	98 - 100
6	2	98 - 100
7	1	98 - 98

The performance distribution for the schedules explored at all filter levels is shown in Figure 4.3. Note that, for filter levels 6 and 7, in the violin plots, no actual violins but only the data points themselves appear, since the number of data points is too small. The distribution reveals that there is a huge gap between the four fastest schedules and the remaining 108. Those four schedules are the only ones that pass all filters up to level 5. With filter level 7, a single schedule is selected. It is not the one with the best performance, but the difference is very small.

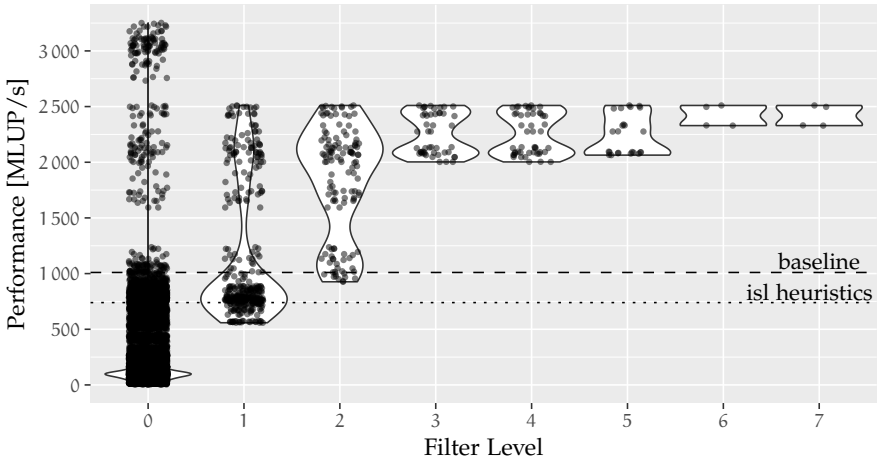
This is one of the experiments in which our heuristics for the isl scheduler fails. It selects a schedule whose performance is significantly below an unoptimized and also untiled version. Choosing the one schedule that passes all filters is as simple as an invocation of the isl scheduler but its performance is clearly superior.

RBGS 3D cc1 (cs)

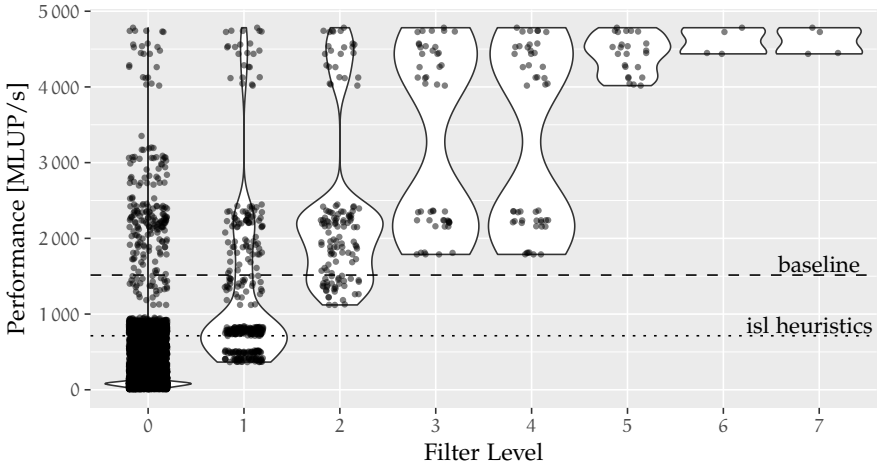
To conclude the detailed exploration evaluation, let us discuss four iterations of a 3D RBGS stencil with radius 1 and constant coefficients. This evaluation has been performed both with and without a data layout transformation that specifies a color splitting, as introduced in Section 4.3.1.

The number of explored schedules per filter level is identical to the Jacobi 3D cc1 experiment presented previously in this section and the performance distribution without a color splitting is shown in Figure 4.4(a). In this case, already the first filter removes the best schedules. This filter's purpose is to ensure that the resulting code can be vectorized. However, the code generator is not capable of vectorizing a RBGS stencil without a color splitting or, more specifically, of vectorizing the non-contiguous memory accesses induced by it. A suitable data layout transformation resolves this issue and the schedules that pass higher filter levels perform much better on the processors' vector units. Their performance is boosted by almost a factor of 2 and they outperform all other schedules, as evident in Figure 4.4(b).

It is also worth mentioning that not all schedules benefit from a color splitting, which complicates the address computations. E. g., the slowest variants that pass the first filter drop from more than 500 MLUP/s to even below 400 MLUP/s. There are also two prominent schedules that are negatively affected: the one selected by the isl scheduler—even though it gets vectorized after the layout transformation—and the schedule of the "PLuTo unrolled" experiment, whose performance is cut by one third with color splitting enabled.



(a) without color splitting



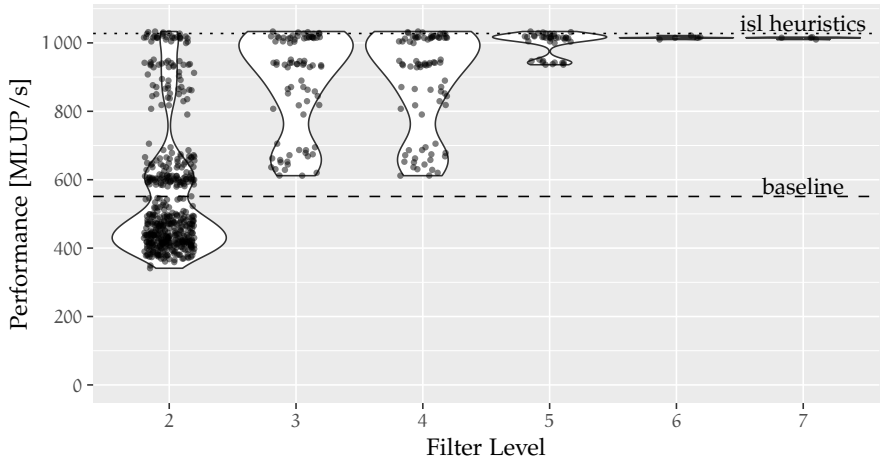
(b) with color splitting

Figure 4.4: Performance distribution of all filter levels for RBGS 3D cc1.

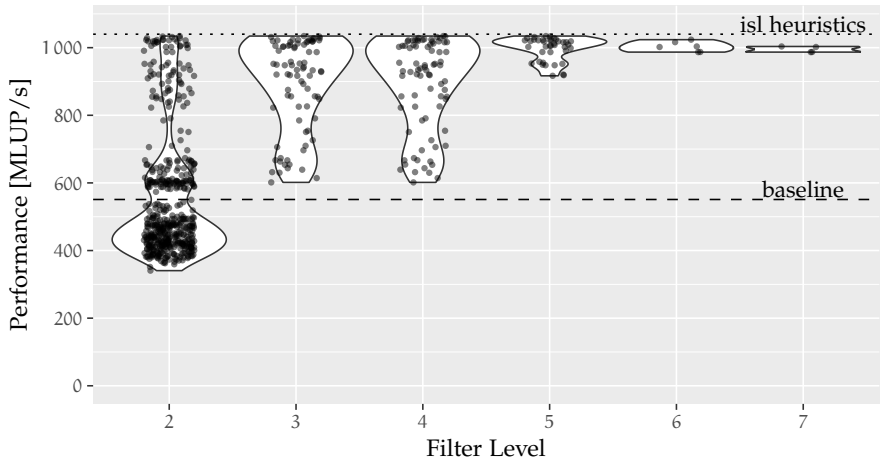
4.3.5 Intel Vectorizer

As mentioned in Section 4.3.1, we disabled Intel's auto-vectorizer in our polyhedral search space exploration since it altered the execution order, i. e., the schedule, in some cases. To strengthen our results, we conducted a quantitative analysis of the vectorizer's impact on the performance of the generated code: we repeated the evaluation with the auto-vectorizer enabled for all 2D experiments and a reduced set of the 3D experiments. We did not switch our code generator's vectorization capabilities off, i. e., vector intrinsics were generated. For the three-dimensional cases, a filter level of 2 instead of 0 was selected to reduce the evaluation time from weeks to days. However, the filtering enabled the exploration to combine up to three rays in all experiments instead of two for all 3D stencils.

Figures 4.5(a) and 4.5(b) are performance distributions for Jacobi 3D vc1 with Intel's auto-vectorizer enabled, respectively disabled. The



(a) with Intel's auto-vectorizer



(b) without Intel's auto-vectorizer

Figure 4.5: Performance distribution for Jacobi 3D vc1.

performance numbers in the former case are more clustered than in the latter but their overall distribution is very similar. There is also no noticeable difference in the best schedules. For filter level 7, the performance with Intel's vectorizer ranges from 1 010 MLUP/s to 1 017 MLUP/s, while the performance without it ranges from 987 MLUP/s to 1 004 MLUP/s.

For all other experiments, the influence of Intel's auto-vectorizer is roughly equal to or even smaller than for Jacobi 3D vc1. Thus, it is no problem to disable icc's vectorizer during an evaluation of our polyhedral search space exploration.

4.3.6 NUMA Architecture

All presented exploration experiments were run on the cluster Chimaira, which has a rather simple and pleasing architecture: each node

powers a single processor that manages a single memory domain. In order to verify that our exploration, and especially our filters, are applicable on other systems, too, a smaller exploration has been conducted on cluster Pontipine. As explained in Section 4.1.1, the main difference to Chimaira is that Pontipine has a NUMA architecture with two processors and two memory domains per node. Thus, it is advisable to keep data local to the CPU on which it is processed.

First-Touch Policy

A common strategy to select in which memory domain new data is allocated is the so-called *first-touch policy*. It means that a memory region is allocated locally to the processor that accesses it the first time.

In detail, remember that the allocation of new memory in a process, e. g., via `malloc`, only allocates virtual memory which has not yet been mapped to the physical memory. Virtual memory is divided into memory pages, which are usually 4 KB wide. The first-touch policy maps a memory page to an address in (physical) main memory only when it is accessed first, no matter whether the access is a read or a write. Which processor issues the access determines in whose memory domain the page is allocated. Therefore, in a parallel application with multiple threads distributed among both processors, the data initialization may have a huge impact on performance, especially for bandwidth-bound codes.

Adjustments of the Experiments

The generated codes for the Chimaira experiments of our search space exploration initialize all fields sequentially. This is unproblematic on Chimaira but it leads to a poor data placement on Pontipine, since all memory pages are allocated on one processor while the other accesses non-local memory only.

Our first change to deal with this problem was to simply parallelize the initialization in the same way as the computation. For the two-dimensional stencils, the inner loop is parallelized, i. e., the first half of each line is processed on one processor while the other handles the second half. The field size was $16\,384^2$ elements (excluding ghost layers). This results in roughly $16\,384 \cdot 8\,\mathrm{B} = 128\,\mathrm{KB}$ per line. Assuming a memory page size of 4 KB, a single line is distributed to at least 32 memory pages. Some of them are accessed from both processors: those in the center of each line, as illustrated in Figure 4.6, and those on the transition from one line to the next. If the others are allocated in the processor's memory that accesses them solely, a performance increase can be expected.

However, after some disappointing results, we realized that our assumption concerning the memory page size was incorrect. Since

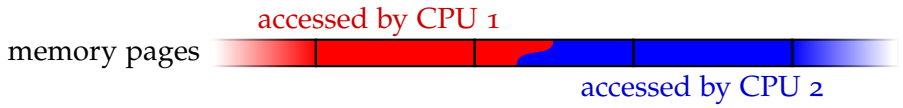


Figure 4.6: Memory pages accessed in the center of a grid line.

version 2.6.38, the Linux kernel supports so-called *huge pages* and their transparent usage[6]. I. e., the Linux kernel allocates 2 MB pages automatically without program modifications if transparent huge pages are (system-wide) enabled, which is the case on our machines. The goal of larger page sizes is to reduce the number of pages for applications with high memory requirements, which may reduce the time required to determine the physical address of a memory page. This does have a positive impact on our codes, too, but the drawbacks of the non-local memory accesses outweigh the benefits: as mentioned earlier, a single line of our fields consumes only 128 KB, which fits entirely into one huge memory page and, therefore, one processor always accesses non-local memory. One solution to this problem is to forgo huge pages on Pontipine. But, in that case, the benefit of huge pages, namely fewer memory pages to handle and a reduced overhead, vanishes, too.

Another option to optimize the data placement is to launch one MPI process per socket. This ensures that each processor accesses only local memory during the computation and explicit communication routines are triggered to transfer data between both memory domains. Even though OpenMPI detects a shared virtual memory and issues in-memory copy operations only, there is a small communication overhead. However, since this approach is much easier to realize—support for MPI communication is in any case mandatory for our generator—it is used in the following experiments.

Evaluation

The overall performance distributions of our experiments on Pontipine are quite similar to those on Chimaira. Consequently, we do not present all results but, rather, focus on two of them. Note that we did not tune the tile sizes specifically for Pontipine but stuck to the tile sizes in the previous exploration given in Table 4.6.

The first experiment we discuss consists of five iterations of a 2D Jacobi stencil with radius 1 and constant coefficients. Its performance distribution is shown in Figure 4.7. Again, baseline is the roofline of a single iteration without a time tiling. In 2D, it is easy to reach the roofline for a single time step since no space tiling is required. Measurements show a performance only 1.1% below the roofline. Concerning the exploration, most of the schedules perform very poorly:

6 see https://lwn.net/Articles/423584/

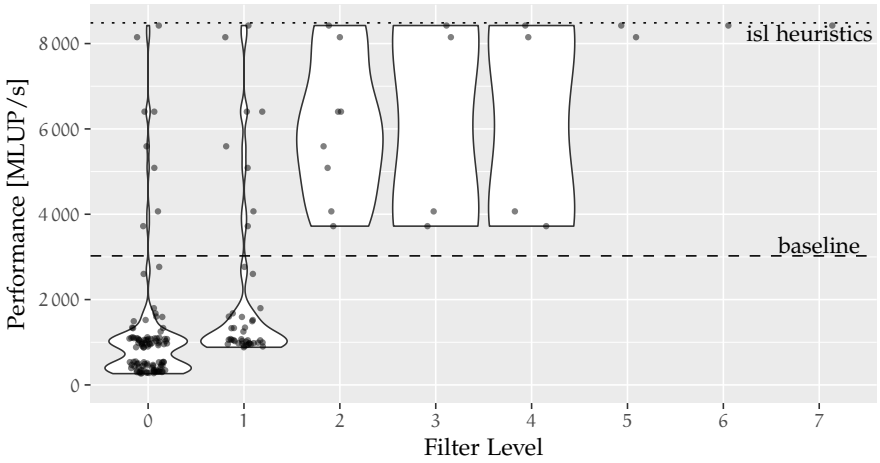


Figure 4.7: Performance distribution for Jacobi 2D cc1 on Pontipine.

100 out of the explored 108 schedules are below the baseline. Nevertheless, the first two filters are able to remove these 100 and none of the faster ones. The fastest is 2.8 times as fast as the baseline and it is also the single schedule that passes all seven filters.

The second experiment worth discussing executes three iterations of a 3D dense stencil with constant coefficients. As evident in Figure 4.8, all explored schedules are below the theoretical baseline. However, the baseline itself cannot be reached in this experiment, since we measure only 2 047 MLUP/s for a not time-tiled version. This is mainly due to the implications of a higher dimensionality and the stencil shape. E. g., the communication volume is significantly higher in 3D than in 2D: instead of a few lines, a few planes have to be exchanged. Additionally, a single lattice update accesses 28 elements, which results in as many load instructions per loop iteration. If we restrict ourselves voluntarily to aligned accesses, as explained in Sections 3.5.1 and 3.5.3, in theory 18 of them can be reused from the previous iteration. But, since there are not enough registers available, most of them are spilled to memory which cancels any benefit of this optimization. Finally, a better tile size for a baseline experiment is able to increase the performance to roughly 2 300 MLUP/s, which is still below the performance of the best explored schedules with a non-optimized tile size.

Another observation is that filter level 4 effectively removes the best performing schedules. The filter applied at level 4 is designed to remove schedules that prevent the vectorization optimization mentioned above and introduced in Section 3.5.3: it favors schedules for which aligned accesses are possible. As explained above, this optimization is ineffective here. A remedy would be to tie the filter applied at level 4 to the mentioned vectorization optimization. If the latter is not requested or not reasonable, this filter should not be applied. However, such a dependence has not been implemented yet.

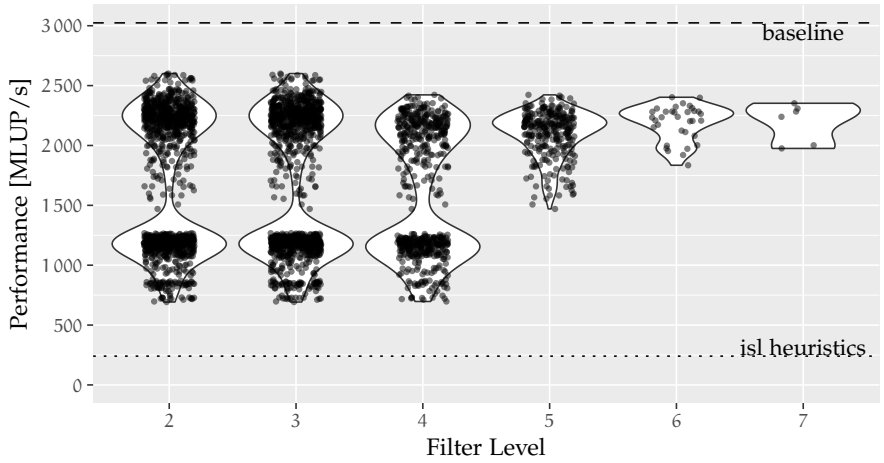


Figure 4.8: Performance distribution for Jacobi 3D ccd on Pontipine.

A dense stencil is one of those for which our heuristics to improve the isl scheduler fails and both techniques select the same schedule. With only 240 MLUP/s, its performance is even worse than that of a completely unoptimized, naïve variant.

The performance gain of our exploration for the other, not shown experiments did not always reach the same high level as for the presented Jacobi 2D cc1 experiment, but it was never as low as for the Jacobi 3D ccd either. Additionally, we encountered no other situation in which all of the best schedules have been removed by our seven filters.

4.3.7 Poisson

To conclude the polyhedral search space exploration, its influence on a complete multigrid solver for Poisson's equation was analyzed. It features a V-cycle with three pre- and post-smoothing steps using a RBGS stencil. The previous experiments focused on the smoother component only, which is the most time-consuming part of this application. No other multigrid components are currently targeted by a time tiling but some may be affected by it. E. g., in an MPI parallel version, the increased number of ghost layers consolidates communication during smoothing: the communication steps between time steps are removed but the communication volume of the rest increases. Additionally, since parts of the solution field's ghost layers are computed instead of communicated, the field for the right-hand side also requires ghost layers, which must be communicated; see Section 3.6.1. The 2D version operates on 13 multigrid levels with roughly $8\,192^2$ elements per MPI node on the finest level. In 3D, nine multigrid levels with up to 512^3 elements per node are computed. We configured our code generator to perform a time tiling on the three finest levels. The others are too small to benefit from such an optimization.

Table 4.9: Achieved speedup of a multigrid solver for Poisson's equation with time tiling.

Poisson	2D		3D	
	1 Node	12 Nodes	1 Node	12 Nodes
base [ms]	1 622	1 872	3 906	6 582
time tiling	1.60×	1.49×	1.57×	1.26×

Performance

We did not carry out a full exploration but simply selected the first schedule that passes all seven filters. This may not result in the fastest code but it also unburdens the user from the effort of a search space exploration, such as the evaluation of several variants on the target hardware. Table 4.9 presents the performance improvements for a 2D and 3D version, both with and without MPI communication, on our cluster Chimaira. The first row is the run time of the entire solver with all optimizations, including color splitting and vectorization, except a time tiling. Reducing the L_2 norm of the initial residual by a factor of 10^5 requires three V-cycles in 2D and four in 3D. Both "12 Nodes" versions are slower than their "1 Node" counterparts since they perform twelve times as many computations and an additional MPI communication. Time tiling is able to increase the performance of the smoothing component significantly and, since it is the most computationally intense component, the performance of the entire solver is noticeably reduced, too. Note that the second row shows the speedup and not the factor by which the time is multiplied. Thus, the actual run time of these versions is the base time divided by the speedup. The improvement of the MPI parallel versions is not as high, due to additional communication phases as explained above. The communication volume in 3D is higher than in 2D, which reduces the benefit of this optimization. Nevertheless, a speedup of 1.26 is still notable.

Generation Time

The code generation and compilation times for these experiments on our workstation are given in Table 4.10. The first number is how long the ExaStencils code generator takes to emit the target C++ code, while

Table 4.10: Code generation and compilation times for the applications in Table 4.9.

Poisson	2D		3D	
	1 Node	12 Nodes	1 Node	12 Nodes
base	8.3 s + 12.3 s	21.5 s + 38.2 s	18.4 s + 10.1 s	35.5 s + 30.9 s
time tiling	13.0 s + 12.9 s	27.5 s + 39.1 s	56.5 s + 24.8 s	75.6 s + 45.0 s

the second is the compilation time with four simultaneous invocations of the Intel compiler. Time tiling increases the code generation time significantly, especially for the 3D variants. A closer inspection revealed that the exploration itself is rather cheap and the integration of the explored schedule, i. e., the abstract syntax tree (AST) generation consumes a huge portion of the additional run time. In detail, isl is configured to avoid conditions inside a loop nest, which is realized by an unrolling. This dramatically increases the code volume that has to be generated and optimized by the following transformations: the number of lines of the smoother function on one of the three optimized levels increase from 82 to 3 220.

4.4 ADDRESS PRECALCULATION & VECTORIZATION

Address precalculation and vectorization, as described in Sections 3.3 and 3.5, are two rather basic and almost supplementary optimizations integrated in the ExaStencils code generator since both are supposedly applied by contemporary production compilers. However, not every compiler is capable of realizing both optimizations in all situations, especially subsequently to other more complex transformations such as time tiling. In addition, they do not require much configuration effort. Actually, the address precalculation does not have any configuration option at all and the vectorization provides a single binary option only: one can generate either unaligned or aligned vector load and store operations, given that both are supported by the target hardware.

4.4.1 Setup of the Experiment

Both the address precalculation and the vectorization compete with the corresponding techniques in production compilers. Therefore, we evaluated the optimizations integrated in the ExaStencils code generator with a variety of C++ compilers. Besides our standard compiler icc 19, binaries generated by the GNU compiler gcc version 8.3 and LLVM/clang version 8.0 were measured. All experiments ran on our cluster Chimaira. The ExaStencils code generator was configured to apply a time tiling and an OpenMP parallelization to load all ten cores. A color splitting was also performed, if appropriate. The problem size was set to 512^3 and $8\,192^2$ per MPI node in 3D, respectively 2D for all experiments.

4.4.2 Evaluation

The effect of the address precalculation and the vectorization implemented in the ExaStencils code generator was evaluated for an isolated Jacobi smoother and a multigrid solver for Poisson's equation both in

Table 4.11: Achieved speedup for 2D and 3D Jacobi kernels with address precalculation and vectorization.

Jacobi 2D cc1	1 Node			12 Nodes		
	icc	gcc	clang	icc	gcc	clang
base [MLUP/s]	2 883	1.08×	1.42×	32 592	1.09×	1.39×
address precalculation	1.03×	1.10×	1.42×	1.03×	1.11×	1.39×
vectorization (unaligned)	1.54×	1.39×	1.35×	1.54×	1.36×	1.25×
(aligned)	1.55×	1.41×	1.28×	1.50×	1.39×	1.28×
both (unaligned)	1.52×	1.45×	1.40×	1.54×	1.43×	1.33×
(aligned)	1.55×	1.50×	1.40×	1.49×	1.46×	1.38×

Jacobi 3D cc1	1 Node			12 Nodes		
	icc	gcc	clang	icc	gcc	clang
base [MLUP/s]	2 569	1.22×	0.89×	19 992	1.12×	0.92×
address precalculation	1.06×	1.26×	1.63×	1.04×	1.15×	1.36×
vectorization (unaligned)	1.78×	1.71×	1.58×	1.42×	1.36×	1.27×
(aligned)	1.76×	1.78×	1.61×	1.39×	1.39×	1.33×
both (unaligned)	1.77×	1.80×	1.79×	1.42×	1.41×	1.41×
(aligned)	1.76×	1.79×	1.80×	1.39×	1.40×	1.42×

2D and 3D. The former is a preferred target of optimizations since it consumes a significant portion of the run time in a multigrid solver. The latter is a simple but complete multigrid solver.

Jacobi Smoother

In a first step, the optimizations were evaluated for Jacobi smoothers with constant coefficients and a stencil with radius 1. It was executed either on a single node or on 12 nodes and, as mentioned previously, a time tiling with five time steps was configured. Table 4.11 contains the results. The base version was compiled by the Intel compiler icc and its performance is given in million lattice updates per second (MLUP/s), while all other variants, including the base versions of the other compilers, are shown as a speedup over icc's base.

A remarkable result is that, for the 2D stencils, the binaries compiled with clang perform exceptionally well without the address precalculation or vectorization of our code generator. A closer inspection of the generated code reveals that clang was able to vectorize the most performance critical loop automatically, despite the very complex code structure caused by time tiling. However, in 3D, clang's base version performs worse than the bases of icc and gcc and the auto-vectorizer only targets said loop if an address precalculation is enabled, which is then sufficient to outperform the other compilers, again. Another difference between the 2D and 3D variants with clang is that, in the latter case, the vectorization capabilities of the ExaStencils code generator lead to the fastest version while, in 2D, clang's vectorizer is slightly better.

Table 4.12: Achieved speedup of a 2D and 3D solver for Poisson's equation with address precalculation and vectorization.

Poisson 2D	1 Node			12 Nodes		
	icc	gcc	clang	icc	gcc	clang
base [ms]	1 119	0.98×	0.96×	1 398	1.02×	0.97×
address precalculation	1.00×	0.99×	0.97×	1.05×	1.03×	1.01×
vectorization	1.10×	1.04×	0.99×	1.08×	1.07×	1.03×
both	1.10×	1.04×	1.04×	1.11×	1.08×	1.07×

Poisson 3D	1 Node			12 Nodes		
	icc	gcc	clang	icc	gcc	clang
base [ms]	3 030	0.91×	0.92×	5 969	0.97×	0.98×
address precalculation	1.04×	1.08×	1.05×	1.06×	1.08×	1.07×
vectorization	1.18×	1.14×	1.11×	1.12×	1.11×	1.08×
both	1.22×	1.20×	1.18×	1.15×	1.15×	1.14×

The performance of the different icc and gcc versions are more predictable: while an address precalculation reduces the run time slightly, best performance is achieved with our vectorization routine. The effect of favoring aligned over unaligned fetch operations differs from case to case but has never been crucial in these experiments.

Poisson

The second application, a multigrid solver for Poisson's equation, leverages three iterations of a RBGS smoother for pre- and post-smoothing. Therefore, a color splitting at the three finest multigrid levels was specified. The performance results for the different versions are given in Table 4.12. In this experiment, we did not generate a vectorized version that focuses on aligned accesses only, since unaligned accesses usually result in a better performance for RBGS smoothers on current Intel processors. Again, absolute run time values are given for the base version compiled with icc while, for all other experiments, the relative performance improvement over icc's base is given. The actual run time of the latter can be reconstructed by a division of the corresponding base with the given speedup.

The effects of our optimizations are similar for all three compilers: vectorization is better for performance than address precalculation but, usually, both are required to achieve best performance. The improvements are not as good as for the smoother-only experiments. This is expected since there are only three consecutive smoothing steps that can be merged via a time tiling, instead of five in the previous experiments. Thus, the bandwidth requirements are still higher, which lessens the potential of a vectorization. Additionally, the other multigrid components do not benefit that much from a vectorization. Finally, the generated code has always been parallelized via OpenMP to load all ten cores. The vectorization would be much

ALGORITHM 4.1: SIMPLE algorithm.

```
1 foreach time step do
2     while not converged do
3         update physical quantities
4         set up and solve LSEs for velocity components (u, v, and w)
5         set up and solve LSE for pressure correction (pc)
6         apply pressure correction
7         set up and solve LSE for temperature (t)
```

more beneficial for a sequential code but, in practice, there is hardly any reason to forgo OpenMP parallelization, which renders sequential experiments useless.

4.5 REDUNDANCY ELIMINATION

This section evaluates the redundancy elimination techniques introduced in Section 3.4. Solving simple partial differential equations (PDEs), such as Poisson's equation, is not suitable for their evaluation since the resulting code is already quite simple. Consequently, subexpressions occur only rarely and, if they do, they are not very complex. Thus, we chose the application of simulating non-isothermal and non-Newtonian fluid flows instead.

4.5.1 Application Overview

There are many approaches to simulating such fluid flows. The implementation used for our experiments [51] is based on the SIMPLE algorithm (Semi-Implicit Method for Pressure Linked Equations). Its derivation is not of interest here and can be found elsewhere in the literature [69, 91]. A brief overview of our implementation as in Algorithm 4.1 is sufficient: in each time step, a series of linear systems of equations (LSEs) for the velocity components, the temperature, and a pressure correction is set up and solved via dedicated geometric multigrid solvers. For each of these solvers, all other components are fixed, which introduces some errors. Thus, they have to be applied repeatedly until convergence is reached.

We rely on a finite-volume approach on non-uniform staggered grids for discretization, as depicted in Figure 4.9. Details can be found elsewhere [91, 95]. In general, finite volume discretizations, especially on staggered grids, require frequent interpolation and integration of values and expressions with respect to control-volume interfaces. Usually, at least parts of these computations on interfaces are independent of the direction of the evaluation. E. g., the evaluation of a

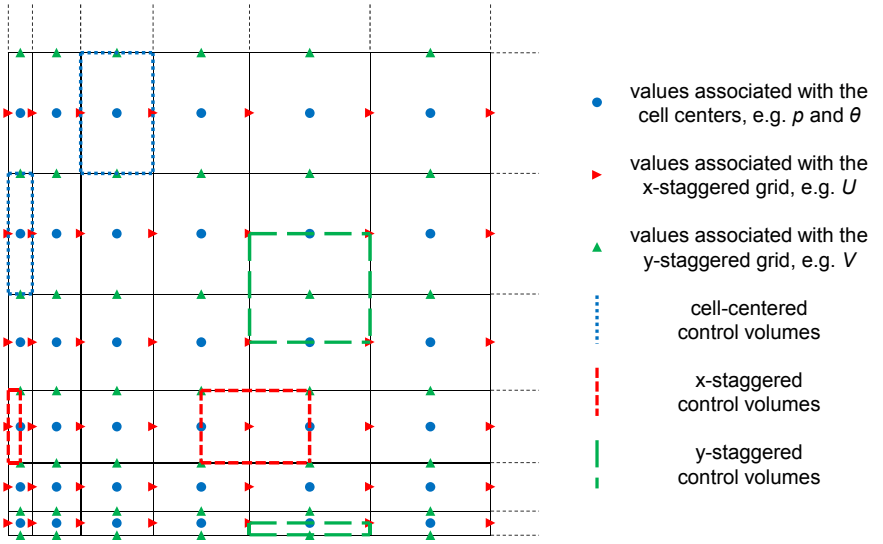


Figure 4.9: 2D illustration of the lower left part of a non-equidistant, staggered grid.

physical quantity located at the cell centers on the east/right, respectively west/left interface will be identical for neighboring cells. This translates to equivalent computations in subsequent loop iterations and, consequently, a loop-carried common subexpression elimination (CSE) has the potential to benefit most codes based on these types of discretizations.

4.5.2 Setup of the Experiment

The complete simulation computes 10 000 time steps for a grid size of 64^3 and ran on a single node of our cluster Chimaira. The binaries were created by the Intel compiler icc 19 and OpenMP is used to load all ten processor cores. Four execution times were extracted from each experiment: the time to update physical properties such as viscosity ("up. quant."), the time to set up the LSEs for all variables ("comp. LSEs") and to solve them ("solve LSEs"), as well as the total time which consists of the previously mentioned and also other factors such as convergence checks and the application of the computed pressure correction. Each of these times is the average over all 10 000 time steps for a single simulation.

4.5.3 Evaluation

Table 4.13 presents the run times and the speedups achieved by the redundancy elimination techniques in conjunction with vectorization. The proportions of the total run time are in parentheses. On the one hand, neither the update of the quantities nor the solving of the LSEs benefit from the CSE techniques. This is expected since

Table 4.13: Achieved speedup for a non-Newtonian fluid flow simulation with syntactic (syn) and loop-carried (lc) CSE.

Fluid Flow	up. quant.	set up LSEs	solve LSEs	total
base [ms]	1.91 (3%)	34.7 (63%)	14.3 (26%)	54.7
syn CSE	1.00 × (5%)	1.54 × (53%)	1.00 × (34%)	1.29×
syn & lc CSE	1.00 × (5%)	1.97 × (47%)	1.00 × (38%)	1.48×
syn CSE + vect	1.07 × (6%)	2.87 × (38%)	0.99 × (45%)	1.71×
syn & lc CSE + vect	1.07 × (6%)	3.28 × (35%)	0.99 × (47%)	1.80×

the corresponding codes do not contain any redundant computation between different loop iterations and the few ones found by the syntactic CSE are too small to influence the run time. Vectorization also does not affect the performance of solving the LSEs, since they are clearly memory-bandwidth-bound. However, it does increase the performance of the quantity update by a few percentage points. On the other hand, the compilation of the LSEs, which requires almost two thirds of the total run time in the base version, does benefit from both CSE approaches and a vectorization: their combined run time can be reduced to roughly 30% of the base. Considering only the CSE techniques, roughly half of the base run time for these parts of the application was wasted for redundant computations. In the fully optimized version, the compilation of the LSEs only demand a bit more than one third of the total run time.

The base version of this code contains several larger redundant computations, both inside the loop bodies and between subsequent loop iterations, which can be factorized out by the two CSE techniques. To quantify the effect of these optimizations further, we counted the floating-point instructions in the generated assembler codes that correspond to the innermost loop nests of the compilation of the LSEs. These numbers are given in Table 4.14. No unrolling, neither by icc nor by our code generator, was performed. On average, even the syntactic CSE is able to save half of the floating-point operations. Note that the base version was compiled with aggressive optimizations (-O3) enabled. I. e., the redundancy elimination techniques implemented in icc 19 were not able to eliminate these common expressions. The loop-carried CSE eliminates an additional 20% of the remaining operations. Note that only the operations that are executed in every loop iteration

Table 4.14: Number of floating point operations executed per loop iteration during the LSE set up phases.

set up LSE	pc	t	u	v	w	total
base	136	342	460	462	460	1 860
syn CSE	75	178	219	221	219	912
syn & lc CSE	54	133	176	186	188	737

are counted, which means that those required to initialize the new buffers are ignored here.

In total, our redundancy elimination approaches uncovered that roughly 60% of the total floating-point arithmetic instructions executed during the set up of the LSEs are redundant. Their elimination results not only in a speedup of 1.97 for these parts but also in a speedup of 1.48 for the entire application. In conjunction with a vectorization, these values increase to 3.28 and 1.80, respectively, which is very promising for a real-world application.

4.6 DATA LAYOUT TRANSFORMATIONS

This section addresses the performance influence of data layout transformations introduced in Section 3.8, both for a single smoother and for the complete multigrid code. As part of the evaluation, the corresponding layout transformation statements are provided and discussed for all experiments.

4.6.1 Setup of the Experiment

All experiments were executed on our Cluster Chimaira and we generated code for both the CPU and GPU. The code generator was configured to add OpenMP pragmas for parallelization and to emit vectorized code for double-precision computations using AVX intrinsics on the CPU. Other optimizations, such as address precalculation and common subexpression elimination, were applied to all versions generated, while a rectangular, spacial tiling was only applied to the 3D versions. If not stated otherwise, we chose a problem size of $8\,192^2$ per node in 2D and 512^3 per node in 3D. Thus, each MPI node performs roughly the same amount of work. The only difference is due to overlapping and potentially redundant computations of neighbors.

4.6.2 Colored Gauss-Seidel Smoother

We start with a performance evaluation of our layout transformations for a colored Gauss-Seidel kernel. The kernel was run in isolation, not as part of a multigrid application. As previously, the performance of the baseline experiments are presented in million lattice updates per second (MLUP/s) and those of other versions as speedups over the baseline. Both constant-coefficient (cc1) and variable-coefficient versions (vc1) were executed. The problem size of the 3D vc1 version running on the GPU had to be reduced to 256^3 to comply with the GPU's memory limit.

Listing 4.4: ExaSlang 4 code for a RBGS smoother.

```
Function Smoother@(all but coarsest) {
  color with {
    (i0+i1+i2) % 2,
    loop over Solution {
      Solution = Solution +
        1.0 / diag(Laplace) * (RHS - Laplace * Solution)
    }
  }
}
```

Two Colors

Our first experiments address the performance of RBGS kernels with constant and variable coefficients for a stencil with radius 1. The ExaSlang 4 code of the kernel in 3D without time tiling and communication is shown in Listing 4.4. The variable `Laplace` is a discretized operator and, as such, either a stencil in the cc1 version or a stencil field in the vc1 version. An appropriate layout transformation for the former is given in Listing 4.5 while, for the latter, the stencil field is transformed in the same way as the other fields. The 2D experiments are analogous. In addition to the performance of a single kernel execution, we evaluated a time-tiled version of five subsequent steps in both 2D and four, respectively, two steps subsequently in the 3D cc1 and 3D vc1 cases. Time tiling is applied by selecting the first schedule that passes all seven filters in a polyhedral search space exploration.

Table 4.15 summarizes the results of all RBGS kernel experiments. The values in parentheses for the single-node base and its color-splitting version represent the fraction of the roofline performance, which is based on the measured memory bandwidth of 45.2 GB/s. For the base cc1 versions in both 2D and 3D, the computation of one new value, i. e., one lattice update (LUP) requires six double-precision values to be transferred between the CPU and main memory. Three are required by the computation: one element is loaded from both the `Solution` and the `RHS` fields (all others are still in the processor's cache) and one updated value is written to the memory. The remaining three

Listing 4.5: Data layout transformation for a RBGS smoother.

```
LayoutTransformations {
  transform Solution, RHS
    with [x, y, z] => [x/2, y, z, (x+y+z)%2]
}
```

Table 4.15: Achieved speedup of 2D and 3D RBGS kernels with color splitting and time tiling.

RBGS	2D cc1		3D cc1	
	1 Node	12 Nodes	1 Node	12 Nodes
CPU base [MLUP/s]	965 (95%)	10 992	960 (95%)	6 900
color splitting	1.44× (91%)	1.44×	1.44× (91%)	1.38×
time tiling	3.26×	3.10×	2.83×	2.25×
both	4.69×	4.42×	4.91×	2.82×
GPU base [MLUP/s]	4 650	43 920	3 693	13 104
color splitting	1.87×	1.58×	1.75×	1.15×

RBGS	2D vc1		3D vc1	
	1 Node	12 Nodes	1 Node	12 Nodes
CPU base [MLUP/s]	378 (100%)	4 440	292 (96%)	3 084
color splitting	1.70× (95%)	1.64×	1.65× (87%)	1.48×
time tiling	3.17×	2.65×	1.75×	1.62×
both	4.41×	4.18×	2.37×	2.07×
GPU base [MLUP/s]	1 696	18 540	1 287	5 448
color splitting	2.03×	1.87×	1.98×	1.18×

are due to the inefficient memory layout: their direct neighbors have to be transferred, too. Therefore, the roofline is:

$$\frac{45.2 \cdot 2^{30}\,\mathrm{B/s}}{(3 \cdot 2) \cdot 8\,\mathrm{B/LUP}} \approx 1\,011\,\mathrm{MLUP/s}$$

A color splitting reduces the number of transferred elements to one read from both fields and one store, as well as a write-allocate for the store, which results in 1 517 MLUP/s. The write-allocate is required since the hardware has to load a cache line from the memory before it can be modified. Without color splitting, this additional load has already been performed since the direct neighbors are required to compute the new value and data is fetched in chunks of eight double-precision values.

A Gauss-Seidel kernel with variable instead of constant coefficients has an even higher requirement of memory bandwidth. Not only the field elements but also the coefficients of all neighbors must be loaded from main memory, which requires five and seven additional values from main memory in 2D and 3D. Without a layout transformation, only every other element is needed and, thus, twice as much data is fetched. The performance results confirm that a color splitting can be more effective for variable than for constant coefficients, at least if no time tiling is applied or possible.

To cut a long story short, a color splitting is able to raise the roofline and the measurements are in good agreement. Additionally, a time tiling was able to reduce the bandwidth requirements even further, which leads directly to a higher performance. However, the resulting

Listing 4.6: ExaSlang 4 code of a colored dense Gauss-Seidel kernel.

(a) Transformation for a single time step

```
LayoutTransformations {
  transform Solution and RHS
    with [x,y] => [x/2,y/2,x%2,y%2]
}
```

(b) Transformation for time tiling

```
LayoutTransformations {
  transform Solution and RHS
    with [x,y] => [x/2,y,x%2]
}
```

(c) Smoother code

```
Function Smoother {
  color with {
    i0 % 2,
    i1 % 2,
    loop over Solution {
      Solution = Solution +
        1.0 / diag(LaplaceDense) *
        (RHS - LaplaceDense * Solution)
    }
  }
}
```

code becomes very complex and we are no longer able to provide a meaningful roofline. On GPUs, a color splitting is even more effective since the accelerators are much more vulnerable to non-optimal memory accesses.

The only disappointing results are for the three-dimensional GPU kernels running on twelve nodes. Since a direct communication between GPUs—as part of NVIDIA GPUDirect—is not supported by our hardware, the MPI communication has to be handled by the CPU. Thus, a single communication consists of three sequential steps: data has to be transferred

(i) over the PCI Express bus from the GPU to the main memory,

(ii) from one node to the other over Ethernet and

(iii) back from the main memory to the GPU on the receiver node.

Such communication steps are required twice per time step, which consumes a large part of the overall run time for the larger communication volume in 3D. In fact, this high overhead makes the optimized CPU versions outperform the GPU in the 3D cc1 experiment.

Multiple Colors

Besides the sparse radius 1 stencils, we also evaluated dense constant-coefficient versions. These stencils access the neighbors not only along the axes, but also along the diagonals. This results in a 9-point stencil in 2D and a 27-point stencil in 3D, as presented in Figures 4.1(c) and 4.1(g). Listing 4.6 shows the ExaSlang 4 code of the 2D smoother and the layout transformations that employ a colored dense stencil.

Table 4.16: Achieved speedup of a 2D four-color and a 3D eight-color Gauss-Seidel kernel with color splitting and time tiling.

MCGS	2D ccd		3D ccd	
	1 Node	12 Nodes	1 Node	12 Nodes
CPU base [MLUP/s]	725 (96%)	8 112	462 (91%)	2 652
color splitting	1.30× (93%)	1.28×	1.28× (97%)	1.10×
time tiling	1.74×	1.71×	1.90×	1.68×
both	1.94×	1.89×	2.25×	1.76×
GPU base [MLUP/s]	2 844	25 932	1 039	3 336
color splitting	1.76×	1.49×	1.77×	1.17×

Two different layout transformations are given, since the one in Listing 4.6(a) excels for a single time step, while the other performs better with temporal blocking and is also superior in the 3D GPU experiments. Evaluating both requires only the adaptation of the presented transformation statement; no additional modifications are necessary. Four different colors are required here for an easy parallelization. A 3D version of a 27-point stencil with eight colors is straight-forward. Note that the actual `loop over Solution` statement is taken over from the RBGS code. Only the coloring scheme and the stencil are different. The definition of the latter (namely `LaplaceDense`) is not shown here; it is a simple list of nine mappings of the neighboring indices to the corresponding constant coefficients.

The performance results for the multi-color Gauss-Seidel experiments are shown in Table 4.16. The results are similar to the two-color versions. The performance of a single smoothing step both with and without color splitting is close to the expected performance based on the memory bandwidth. However, time tiling is not as beneficial or, at least, the chosen schedule may not be the best. But an overall speedup of roughly 2 in the CPU experiments is still satisfactory.

4.6.3 Multigrid Solvers

The previous experiments evaluated the performance impact of color splitting for the smoother component of a multigrid application. However, there are other multigrid components that are also affected by a layout transformation. To demonstrate the applicability and benefit of a color splitting for a complete multigrid application, we conducted experiments with two different applications. The first is the well-known model problem given by Poisson's equations and the second is a multigrid version of an optical flow detection method, as introduced in Section 4.2.2. For both applications, the same variants as for the smoother-only experiments were generated: a shared-memory OpenMP or CUDA version running on a single CPU or GPU and a hybrid OpenMP/MPI or CUDA/MPI parallel version for 12 nodes.

Table 4.17: Achieved speedup of a 2D and 3D solver of Poisson's equation with color splitting and time tiling.

	Poisson	2D		3D	
		1 Node	12 Nodes	1 Node	12 Nodes
CPU	base [ms]	2 201	2 410	5 275	7 902
	color splitting	1.36×	1.29×	1.35×	1.20×
	time tiling	2.11×	1.89×	1.77×	1.47×
	both	2.17×	1.92×	2.12×	1.52×
GPU	base [ms]	551	917	1541	5299
	color splitting	1.41×	1.19×	1.41×	1.10×

The memory layout changes are integrated solely via a single layout transformations block, no other changes are applied. Thus, there is no copy kernel to change the memory layout dynamically and the kernels for the restriction or interpolation are also not modified. Consequently, these kernels access elements of both colors, regardless of where they are stored.

In the case of the baseline experiments, we give the performance of the multigrid solvers as the total time required by the complete solver. Performance improvements of the layout transformations and time tiling are again stated as speedups over the baseline. Thus, the inverse of the speedup is the fraction of the execution time over the base.

Poisson

Let us start with the multigrid solvers for the Poisson equation. Its V-cycles have three pre- and three post-smoothing RBGS steps with constant coefficients and, thus, the same layout transformation as for the previous RBGS smoother experiments, i. e., a color splitting was applied to the three finest multigrid levels. At coarser levels, color splitting is not recommended: since the fields fit into cache, its benefit vanishes while the drawback of more complicated address computations still persists. In all 2D experiments, three V-cycles were executed, in 3D four.

Performance results of these experiments are given in Table 4.17. Remember that the "12 Nodes" version also computes twelve times as many unknowns. In all experiments, the storage of the red and black colored elements in distinct memory locations increased the performance. The improvements are not as high as in the smoother-only experiments in Section 4.6.2. This is due to the other multigrid components, such as the restriction or interpolation, whose performance slightly deteriorates. A time tiling is also not as effective since there are only three time steps to be merged instead of five or four in the previous experiments. Nevertheless, the performance improvements are quite good for an entire multigrid solver.

Listing 4.7: Layout transformations for the optical flow computation.

```
LayoutTransformations {
  concat @finest Ix, Iy, Iz, It into I
  concat IxIx, IxIy, IxIz, IyIy, IyIz, IzIz into II
  transform I@finest,
            II@((finest-1) to finest),
            rhs@((finest-1) to finest),
            flow@((finest-1) to finest)
    with [x,y,z] => [x/2,y,z,(x+y+z)%2]
  transform residual, cgTmp0, cgTmp1,
            II@(0 to (finest-2)),
            rhs@(0 to (finest-2)),
            flow@(0 to (finest-2))
    with [x,y,z,v] => [v,x,y,z]
}
```

Optical Flow Detection

A second, more complex multigrid solver is for an approximation of the optical flow, as introduced in Section 4.2.2. It computes an approximate motion in an image sequence. Concerning the problem size, roughly 17 million pixel per image were chosen in both 2D and 3D. The termination criterion is met after four V-cycles in the 3D version, running on a single node, and five V-cycles in the others.

The motion, or flow, of a single pixel consists of multiple components, namely one per dimension, which means that the element type of the corresponding field is a vector, not a scalar. This increases the dimensionality of the entire field but, as explained in Section 3.8.1, a layout transformation statement may ignore these additional dimensions if they should not be modified.

The layout transformations evaluated for the 3D optical flow computation are shown in Listing 4.7. It pays to concatenate some of the helper fields to enable an struct-of-arrays (SoA)-to-array-of-structs (AoS) transformation, as shown in the second **`concat`** and **`transform`** statement. The first **`concat`** does not affect the performance of the generated code, but it allows a more compact transformation statement. Its individual parts need not be listed separately to perform a color splitting. In general, the RBGS smoother allows for two mutually exclusive field layout schemes to increase the performance:

(i) split by color and place the different components of the higher-order elements in distinct memory locations,

(ii) do not split by color but store the higher-order elements together in memory.

The former increases the data locality of the smoother code and allows it to be vectorized. Other, not colored parts suffer from a more complex address computation. The latter provides a simpler memory layout

Table 4.18: Achieved speedup of a 2D and 3D solver for an optical flow simulation with color splitting.

Optical Flow Simulation		2D		3D	
		1 Node	12 Nodes	1 Node	12 Nodes
CPU	base [ms]	2 661	3 040	3 397	6 367
	layout transformations	1.36×	1.28×	1.41×	1.23×
GPU	base [ms]	691	1 286	1 022	3 795
	layout transformations	1.27×	1.10×	1.27×	1.08×

and an increased data locality for those not colored parts. For the finer levels, the performance of the smoother is the dominant factor and, hence, layout scheme (i) is advisable. At coarser levels, larger parts of the fields fit into cache and the effort of the address computation becomes more relevant: layout scheme (ii) is preferable.

The first transformation statement applies a two-color split to field `I` and to the two finest levels of fields `II`, `rhs`, and `flow`. Note that these fields refer to 4D data structures: the first three dimensions (`x`, `y`, and `z`) correspond to the problem dimensions and the fourth (not named) to the vector elements. Since the dimensions added by higher-order data types, such as vectors or matrices, are appended rightmost/outermost, this corresponds to layout scheme (i). The second transformation statement permutes the vector dimension innermost for some other fields and coarser levels of `II`, `rhs`, and `flow`, i. e., an SoA-to-AoS transformation is applied and layout scheme (ii) takes hold.

Table 4.18 presents the speedups achieved by the layout transformations of the different versions. In contrast to the previous experiments, a time tiling was not evaluated here since the Neumann boundary condition enforces the ghost elements to be updated after every time step, which is not yet supported by the code generator in conjunction with time tiling. The results are similar to those of the Poisson experiments and, thus, as expected. The layout transformations in this experiment may be more complex and not obvious but, in contrast to a manual integration, evaluating several different versions is as simple as writing very few lines that specify the transformation and execute our code generator.

4.7 SUMMARY

We conducted several different experiments to evaluate the performance of our implemented optimizations, as given in Table 4.19. These optimizations target different codes: while the polyhedral search space exploration can be used to select an efficient time tiling for the smoother, the redundancy elimination techniques mainly affect the computation of variable coefficients.

Table 4.19: List of all experiments.

Section	Experiments
Section 4.3: Polyhedral Search Space Exploration	Jacobi & RBGS smoother (also on a NUMA architecture), Poisson
Section 4.4: Address Precalculation & Vectorization	Jacobi smoother (with icc, gcc, clang), Poisson (with icc, gcc, clang)
Section 4.5: Redundancy Elimination	non-Newtonian fluid flow simulation
Section 4.6: Data Layout Transformations	colored Gauss-Seidel smoother, Poisson, optical flow simulation (also on GPU)

Address precalculation and vectorization are not so specific and target potentially all generated loops. They compete against the corresponding optimizations in production compilers but, at least in conjunction with other techniques that make the generated code more complex, our implementation outperforms the others. However, its impact is not as high as for the specialized techniques.

Our evaluation of the data layout transformations focuses mainly on color splitting, since this class of transformations has a high impact on performance. But our implementation is by no means limited to color splitting.

Each of our implemented techniques is beneficial for at least some applications and they perform well in conjunction, too. E. g., a multigrid solver for Poisson's equation can be targeted by most techniques and a speedup of more than 2 can be reached in 2D and 3D running on a single node. MPI parallel versions running on 12 nodes benefit with a speedup of almost 2 and 1.5.

5 RELATED WORK

Interest in the optimization of stencil codes is not new. In the previous decades several different techniques have been developed that target numerous aspects of this domain. While we are not able to discuss all of them, this chapter provides a brief overview of some related work: we focus on stencil DSLs and on techniques related to our most advanced optimizations.

5.1 STENCIL DSLS

To begin with, let us briefly comment on tools and frameworks that offer a DSL for stencil codes. Patus [12] is a code generation framework for the domain of stencil codes. It can generate code for both CPUs and GPUs and its strong point is the autotuning of different optimization parameters. SDSLc [77] is a compiler for the Stencil DSL (SDSL), which is embedded in C, C++ and MATLAB. The SDSL compiler is capable of generating code not only for CPUs and GPUs but also for FPGAs. Halide [75, 76] and PolyMage [59] are image processing libraries and, therefore, can be used to generate optimized codes for stencil applications, too. The latter is described in more detail in Section 4.3.3. In contrast to the custom tool chains on which all of these approaches are based, Pochoir [87] and STELLA [37] provide DSLs and C++ libraries to be used directly with a standard production compiler for C++. Pochoir programs can be translated with the regular C++ compiler, but there is also an optimizing compiler if performance is important. STELLA does not offer any custom tool chain or compiler. The optimization and code generation proceeds via C++ template metaprogramming only.

5.2 REDUNDANCY ELIMINATION

Common subexpression elimination (CSE) and global value numbering (GVN) are well-known and also well-understood compiler optimization techniques incorporated in almost all production compilers [2, 17, 58]. These implementations are suitable for removing redundancies introduced by the compiler itself, e. g., when it creates address-computation instructions from abstract array accesses. However, common subexpressions at source code level are not always identified as such, since the target compiler must make worst-case

assumptions for aliasing and other language features. More powerful CSE techniques were presented by Debray [20] and Saabas et al. [78], and elsewhere. Additionally, there exist several specialized CSE approaches [45, 91], which focus on problems in different application domains, such as digital signal processing.

Neither of these take redundancies between loop iterations into account. Hammes et al. [39] present a temporal CSE for a special type of loops from the language SA-C. For these loops, the programmer can explicitly specify a so-called window for the data structure to be traversed. This window defines how many neighboring elements are accessed per loop iteration. It is also used to identify redundancies between subsequent loop iterations, which simplifies the detection process but also limits its applicability. A more powerful approach was presented by Faber et al. [26]. It is based on the polyhedral model and can therefore detect redundancies between any loop iterations. On the downside, this approach only detects common expressions for which the non-array parts are structurally equivalent, i. e., the example of Listing 3.6 cannot be optimized. Additionally, even if a single scalar is sufficient to carry information from one loop iteration to the next, every instance of this value gets its own memory location. In contrast to the loop-carried CSE presented in Section 3.4.4, this increases memory consumption unnecessarily. Deitz et al. [21] developed their so-called array subexpression elimination. It is specialized to redundancies between subsequent iterations of the innermost loop that occur during the computation of dense stencils. However, these common expressions only consist of very few operations for the stencils in our domain. A recent and more complex redundancy elimination that also targets redundancies across loop bounds was published by Ding and Shen [22]. Their technique is based on a new symbolic notation for loop nests and it can target a broader spectrum of redundancies than other approaches, such as expressions invariant for only some of the loops in a nest.

5.3 VECTORIZATION

Vectorization is an important topic and the usage of the corresponding hardware units is crucial for optimal performance. Eichenberger et al. [25] present a transformation that is mainly focused on vectorizing codes with unaligned memory accesses if these are not directly supported by the hardware. Strides in the memory access pattern or interleaved data may also impede vectorization. The latter occurs, e. g., when dealing with a sequence or vector of complex numbers, since they are represented by a pair of floating-point values. But other techniques exist to vectorize such codes [60, 64]. Further vectorization techniques focus on outer loops [65], integrate a software prefetching

and non-temporal stores [48], or split the vectorization into an offline and an online phase to allow integration in a JIT compiler [63]. There are also attempts to use polyhedral techniques to detect vectorizable schedules [47, 84]. However, one of the most advanced automatic vectorizer in use is the one implemented in the Intel compiler. It is also regularly being improved: approaches to the optimization of data accesses for complex patterns have been published recently [3].

Much like other code generation systems, such as Spiral [74] for linear transforms or SLinGen [85] for the domain of linear algebra, the vectorization capabilities of the ExaStencils code generator are highly specialized to its domain. Also, a large portion of the implementation effort was spent to overcome problems and limitations induced by other optimizations and transformations. Therefore, other work concerning vectorization is usually more general and, to some extent, more advanced than our vectorization approach. E. g., Caballero et al. [10] present how to optimize overlapped vector loads. Such overlapped loads appear also in a very simple form in stencil codes, as shown in Section 3.5.1.

5.4 DATA LOCALITY OPTIMIZATIONS

Data locality optimizations are crucial for high performance of stencil codes, which are usually limited by the memory bandwidth. There are several different approaches; they can be divided roughly into non-polyhedral and polyhedral techniques.

5.4.1 Non-Polyhedral Techniques

Many tools and algorithms for the optimization of stencil codes come with a tiling scheme. E. g., there are different space tiling techniques, such as cache blocking and register blocking, that focus on different levels in the cache hierarchy [23]. Besides other optimizations, Kaushik Datta [18] implements and evaluates several of these approaches into an tuner. A time tiling increases the optimization potential further. Frigo et al. [30] developed a cache-oblivious tiling scheme to increase data locality between subsequent time steps. Cache obliviousness means that the actual cache sizes are not parameters of their approach. It does not use fixed block sizes but the iteration space is divided recursively. However, an evaluation of several time tiling techniques performed by Datta et al. [19] revealed that cache-aware techniques perform better than cache-oblivious ones. More advanced and also more specialized cache-aware approaches for stencil codes lead to very good results [61, 89, 96, 97], but they are also hard to integrate automatically. In contrast, polyhedral techniques benefit from a high level of automation.

5.4.2 Polyhedral Techniques

In the polyhedron model, there are several different optimization techniques and approaches not specific to stencil codes. A model-based scheduling algorithm due to Feautrier [27, 28] maximizes the parallelism for a given loop nest. However, for modern processors, data locality is frequently at least as important as parallelism. Bondhugula et al. [8] developed the PLuTo scheduling algorithm that recognizes data locality by detecting both tileable and parallel schedules automatically. Acharya et al. [1] later removed some of the restrictions imposed by the first PLuTo implementation, e. g., by allowing negative schedule coefficients. Especially for stencil computations, improved tiling mechanisms, such as split tiling [41], overlapped tiling [49], diamond tiling [4, 7], and hexagonal tiling [35] were developed. A comparison of our exploration with some of these techniques is given in Section 4.3. The scheduler by Kong et al. [47] was developed to take advantage of the vector units provided by modern processors. It computes directly a schedule for which vectorized code can be emitted. However, there is no exploration of the space of vectorizable schedules.

The polyhedron model offers a wide spectrum of transformations, which makes a search space exploration computationally complex. A genetic algorithm to explore the search space of affine schedules was implemented by Nisbet [62] in the GAPS framework. Long et al. [56] presented a polyhedral exploration to optimize Java programs. Like the GAPS framework, it uses the Unified Transformation Framework (UTF) developed by Kelly [46]. Both suffer from the fact that they generate huge search spaces with predominantly illegal transformations.

With LeTSeE, Pouchet et al. [71] provide a search space exploration for one-dimensional schedules. It searches all legal schedules with small coefficients and constants exhaustively. This is feasible but, for a multi-dimensional loop nest, a single dimension is not sufficient to specify the execution order of all statement instances. LeTSeE's extension to multi-dimensional schedules [70] imposes additional requirements, such as a (heuristically predetermined) order in which the data dependences are strongly satisfied. Since LeTSeE's exploration is designed for sequential codes, dependences are carried greedily when generating the search spaces for different dimensions. Thus, dependences are carried outermost only and schedules that pass our filter level 3, as presented in Section 3.7.4, cannot be generated: they require some dependences to be carried innermost. Our guided exploration does not impose a restriction on the dimension by which the dependences are carried. The basic difference is that LeTSeE's exploration restricts the search space as early as possible to keep it small enough for an exhaustive exploration, while our technique adds as few constraints to the search space as possible and then selects heuristically a subset for evaluation. Alternatively, LeTSeE can be

configured to leverage a genetic algorithm to search the space of legal transformations. As a follow-up, Pouchet et al. [72] combined the exploration with model-based approaches: a good loop fusion and distribution structure is searched empirically, while the resulting loop nests are optimized by the PLuTo algorithm. Ganser et al. [31] recently developed a polyhedral search space exploration, named Polyite, that is also based on the generator representation of the search space; see Section 4.3.3. In contrast to our exploration, it is not domain-specific and is based on the production compiler LLVM. It uses a genetic algorithm to traverse the search space.

5.5 DATA LAYOUT TRANSFORMATIONS

The use of data layout transformations for performance optimization is not new. O'Boyle and Knijnenburg [66] present a very detailed low-level view of layout transformations. These are represented by nonsingular matrices and, thus, enable only dimensionality-preserving transformations. This excludes the kinds of color splitting that we use. Clauss et al. [14] try to optimize the spacial locality by providing a new array reference function to the compiler. Array elements are ordered in the order in which they are accessed during the execution of the loop nest. This approach works perfectly if every element is accessed only once. In case of a later reuse, the address computation can become very complex.

Layout transformations similar to the ones in our examples are part of existing libraries and frameworks, such as Kokkos [24] or YASK [99]. The latter is a framework for stencil code generation, which supports—besides several other optimizations—a specific data layout optimization called vector folding. But, no other layout transformations are supported. In contrast, ExaStencils facilitates such extensions with the support of generic layout transformations expressed via affine transformations.

Kokkos is a C++ performance portability programming ecosystem. It relies heavily on C++ template programming to generate optimized kernels for different hardware architectures, including GPUs. Like other code generation approaches, it supports the choice of an appropriate layout for a given hardware. However, if the required layout is not yet available, a custom implementation is necessary. Moreover, Kokkos is focused on parallelization within one MPI rank and, in consequence, the synchronization of data between ranks is still the user's responsibility. Thus, when modifying the memory layout, other parts of the code might require an according adaptation, e. g., the incorporation of MPI data types. ExaStencils does not have this problem since it generates code for data exchange via MPI as well which, thus, can be adapted to the optimized layout automatically.

There are also several approaches to the automatic computation of a suitable data layout transformation, either directly [11, 14, 53, 57, 66] or in combination with a loop transformation [13, 55, 68]. We have not done so, but it would be possible to add one or more of these techniques to our code generator.

6 CONCLUSIONS

To conclude, this chapter summarizes the optimization techniques presented in Chapter 3 and the evaluation in Chapter 4. All of these techniques have been implemented in the ExaStencils code generator, which applies them to the generated code without user intervention. Some possible directions for future work are also mentioned.

6.1 SUMMARY

Chapter 3 describes several different optimizations to increase the performance of generated stencil codes, while Chapter 4 evaluates them. The inlining and arithmetic normalization approaches developed in Sections 3.1 and 3.2 are two supplementary transformations. While the former eliminates small functions to unburden other optimizations from dealing with function calls, the latter is used in numerous strategies throughout the entire code generation process.

The techniques of redundancy elimination in Section 3.4 also rely on both. A traditional, syntactic version of a common subexpression elimination (CSE) leverages them as preliminary transformations to increase the size and the number of common subexpressions found. E. g., the arithmetic simplifications prevent an interference of the commutativity and associativity law of addition and multiplication with the redundancy detection. Also in Section 3.4, a simple extension of the syntactical CSE to be applicable across loop boundaries is formulated. This allows reusing already evaluated subexpressions from the previous iteration of any surrounding loop. However, it comes at a cost since for outer loops more than a single scalar must be preserved: for each iteration of the inner loops an additional value has to be stored. We demonstrate the usefulness of the redundancy elimination techniques with a real-world application, namely a non-isothermal and non-Newtonian fluid-flow simulation in Section 4.5. The performance of the affected code parts was tripled, while a speedup of 1.8 for the entire real-world application was achieved. A text-based version can be profitable, or at least not harmful, for any stencil computation, while a loop-carried approach is especially useful for finite-volume discretizations.

Another rather standard but crucial optimization implemented in the ExaStencils code generator is an automatic vectorization of the emitted code. Its implementation is presented in Section 3.5. The vectorization process is architecture-independent and the most common

vector instruction sets are supported: Intel's SSE3, AVX, AVX2, and AVX-512, as well as IBM QPX and ARM Neon. The evaluation of the vectorization and an address precalculation presented in Section 4.4 confirmed that our implementation outperforms the corresponding techniques in current production compilers. We also describe how an additional data dependence introduced by the loop-carried CSE can be treated by the vectorizer.

The polyhedral techniques in Sections 3.6 and 3.7 tackle the very low arithmetic intensity of stencil codes via both a space and a time tiling. In this context we propose an efficient multi-dimensional polyhedral search space exploration of the unbounded set of all legal affine transformations to find the best time tiling scheme. A subset of the search space is selected using its dual representation computed by the Chernikova algorithm. This allows to control how many schedules are explored while favoring simple ones, i. e., schedules with small coefficients since the hope is that the better performing ones are simple. In principle, the exploration can target any domain. We propose a set of heuristic filters customized for the domain of stencil codes. The exploration procedure and the filters have been implemented in the ExaStencils code generator and the experimental evaluation of different codes in Section 4.3 shows promising results. While one would require to try different frameworks and libraries, such as PolyMage, isl, or PLuTo, to identify the best performing schedule, our unified approach in a single framework automatically finds a schedule that achieves 79% or more of the performance that these individual frameworks can achieve. With all filters switched on, the number of schedules to be evaluated can be reduced to at most six, and even the one with the worst performance does reasonably well. In some of the experiments, the exploration even yields exclusively much better schedules than all other tools and algorithms in our evaluation. If one does not strive for optimality, one might be happy with any one of them and save any further exploration effort. Also, the use of few rather than many filters trades compile-time optimization effort for run-time performance.

Finally, we offer a new language feature of the DSL ExaSlang and describe the corresponding implementation in Section 3.8: a set of layout transformation statements for a fast and easy modification of data layouts. With the help of these statements, the laborious and error-prone modification of every access to a data field, including its initialization, is done automatically by the ExaStencils code generator. One use case is a simple RBGS smoother, which is very profitable because of its parallelizability and numerical properties. Without any modifications, it accesses only every other data element, which complicates the use of vector units provided by modern processor architectures and also wastes memory bandwidth, which tends to be the most critical resource. In Section 4.6, we demonstrated that

these problems can be solved by adding only one single transformation statement. Even for larger applications, only few additional statements are required, which enables the testing of several different layout schemes with very little effort. Since some of the concepts required by automatic layout transformations and for polyhedral compilation are identical or at least similar, the isl, which was actually developed to support polyhedral compilation, is of great help. It is the key component in our implementation to support not only a specialized color splitting but any affine transformation.

6.2 FUTURE DIRECTIONS

The optimization techniques detailed in Chapter 3 increase the performance of generated stencil codes in various ways but they are not complete. There are still countless other stencil optimizations that can be integrated in the ExaStencils code generator, such as a hexagonal tiling for GPUs [35] or other, more complex tiling techniques [7, 41, 59]. And, even if the performance of the generated smoothers is now satisfactory, it may be worth to develop and integrate optimizations for other components of a multigrid solver, too.

Additionally, the presented techniques can be refined further. E. g., the polyhedral techniques for time tiling impose some restrictions on the loops to be applicable. Eliminating these restrictions is always an option. Also, a more in-depth evaluation of the properties of the explored schedules and the ones computed by other tools could grant a deeper insight into the requirements for stencil codes to achieve best performance. These findings may be used to refine our filters or to develop new techniques.

To sum up, the ExaStencils code generator already provides a set of powerful optimizations but there is the room and the opportunity for more. We offer it as a solid foundation for further research.

BIBLIOGRAPHY

[1] Aravind Acharya and Uday Bondhugula. "PLUTO+: Near-Complete Modeling of Affine Transformations for Parallelism and Locality." In: *Proc. 20th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*. Ed. by Albert Cohen and David Grove. ACM, 2015, pp. 54–64.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers – Principles, Techniques and Tools*. 2nd ed. Section 6.4.3. Addison-Wesley, 2007.

[3] Farhana Aleen, Vyacheslav P. Zakharin, Rakesh Krishnaiyer, Garima Gupta, David Kreitzer, and Chang-Sun Lin. "Automated Compiler Optimization of Multiple Vector Loads / Stores." In: *Int. J. Parallel Programming* 46.2 (Apr. 2018), pp. 471–503.

[4] Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. "Tiling Stencil Computations to Maximize Parallelism." In: *Proc. Int. Conf. on High Performance Computing Networking, Storage and Analysis (SC)*. IEEE Computer Society, 2012, 40:1–40:11.

[5] Cédric Bastoul. "Code Generation in the Polyhedral Model Is Easier Than You Think." In: *Proc. 13th Int. Conf. on Parallel Architecture and Compilation Techniques (PACT)*. IEEE Computer Society, Sept. 2004, pp. 7–16.

[6] Cédric Bastoul. *Clan — A Polyhedral Representation Extractor for High Level Programs*. Available at the Clan Web site. June 2014.

[7] Uday Bondhugula, Vinayaka Bandishti, and Irshad Pananilath. "Diamond Tiling: Tiling Techniques to Maximize Parallelism for Stencil Computations." In: *Trans. on Parallel and Distributed Systems (TPDS)* 28.5 (May 2017), pp. 1285–1298.

[8] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. "A Practical Automatic Polyhedral Parallelizer and Locality Optimizer." In: *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. ACM, 2008, pp. 101–113.

[9] Andrés Bruhn, Joachim Weickert, Christian Feddern, Timo Kohlberger, and Christoph Schnorr. "Variational Optical Flow Computation in Real Time." In: *IEEE Trans. on Image Processing (TIP)* 14.5 (2005), pp. 608–615.

[10] Diego Caballero, Sara Royuela, Roger Ferrer, Alejandro Duran, and Xavier Martorell. "Optimizing Overlapped Memory Accesses in User-directed Vectorization." In: *Proc. 29th Int. Conf. on Supercomputing (ICS)*. ACM, 2015, pp. 393–404.

[11] Guilin Chen, Mahmut Kandemir, and Mustafa Karakoy. "A Constraint Network-Based Approach to Memory Layout Optimization." In: *Proc. Conf. on Design, Automation and Test in Europe (DATE)*. Vol. 2. IEEE Computer Society, 2005, pp. 1156–1161.

[12] Matthias Christen, Olaf Schenk, and Helmar Burkhart. "PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures." In: *Proc. 25th Int. Parallel & Distributed Processing Symp. (IPDPS)*. IEEE Computer Society, May 2011, pp. 676–687.

[13] Philippe Clauss, Vincent Loechner, and Benoit Meister. "Minimizing Strides in Loops with Affine Array References." In: *Proc. 9th Workshop on Compilers for Parallel Computers (CPC)*. 12 pp. June 2001.

[14] Philippe Clauss and Benoit Meister. "Automatic Memory Layout Transformations to Optimize Spatial Locality in Parameterized Loop Nests." In: *SIGARCH Computer Architecture News* 28 (Mar. 2000), pp. 11–19.

[15] Cliff Click. "Global Code Motion / Global Value Numbering." In: *Proc. ACM SIGPLAN 1995 Conf. on Programming Language Design and Implementation (PLDI)*. ACM, June 1995, pp. 246–257.

[16] John Cocke. "Global Common Subexpression Elimination." In: *Proc. Symp. on Compiler Optimization*. ACM, July 1970, pp. 20–24.

[17] John Cocke and Jacob T. Schwartz. *Programming Languages and Their Compilers: Preliminary Notes*. 2nd ed. Courant Institute of Mathematical Sciences, New York University, 1970.

[18] Kaushik Datta. "Auto-Tuning Stencil Codes for Cache-Based Multicore Platforms." PhD thesis. EECS Department, University of California, Berkeley, Dec. 2009.

[19] Kaushik Datta, Shoaib Kamil, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. "Optimization and Performance Modeling of Stencil Computations on Modern Microprocessors." In: *SIAM Rev.* 51.1 (Feb. 2009), pp. 129–159.

[20] Saumya K. Debray. "Compiler Optimizations for Low-level Redundancy Elimination: An Application of Meta-level Prolog Primitives." In: *Metaprogramming in Logic*. Ed. by Alberto Pettorossi. LNCS 649. Springer, 1992, pp. 120–134.

[21] Steven J. Deitz, Bradford L. Chamberlain, and Lawrence Snyder. "Eliminating Redundancies in Sum-of-Product Array Computations." In: *Proc. 15th Int. Conf. on Supercomputing (ICS)*. ACM, 2001, pp. 65–77.

[22] Yufei Ding and Xipeng Shen. "GLORE: Generalized Loop Redundancy Elimination Upon LER-notation." In: *Proc. ACM on Programming Languages* 1 (OOPSLA Oct. 2017), 74:1–74:28.

[23] Hikmet Dursun, Ken-ichi Nomura, Weiqiang Wang, Manaschai Kunaseth, Liu Peng, Richard Seymour, Rajiv K. Kalia, Aiichiro Nakano, and Priya Vashishta. "In-Core Optimization of High-Order Stencil Computations." In: *Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA)*. CSREA Press, 2009, pp. 533–538.

[24] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns." In: *J. Parallel and Distributed Computing* 74.12 (2014), pp. 3202–3216.

[25] Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. "Vectorization for SIMD Architectures with Alignment Constraints." In: *Proc. ACM SIGPLAN 2004 Conf. on Programming Language Design and Implementation (PLDI)*. ACM, 2004, pp. 82–93.

[26] Peter Faber, Martin Griebl, and Christian Lengauer. "Loop-Carried Code Placement." In: *Euro-Par 2001: Parallel Processing*. Ed. by Rizos Sakellariou, John Gurd, Len Freeman, and John Keane. LNCS 2150. Springer, 2001, pp. 230–235.

[27] Paul Feautrier. "Some Efficient Solutions to the Affine Scheduling Problem, Part I: One-dimensional Time." In: *Int. J. Parallel Programming* 21.5 (1992), pp. 313–347.

[28] Paul Feautrier. "Some Efficient Solutions to the Affine Scheduling Problem, Part II: Multidimensional Time." In: *Int. J. Parallel Programming* 21.6 (1992), pp. 389–420.

[29] Paul Feautrier and Christian Lengauer. "Polyhedron Model." In: *Encyclopedia of Parallel Computing*. Ed. by David Padua et al. Springer, Sept. 2011, pp. 1581–1592.

[30] Matteo Frigo and Volker Strumpen. "Cache Oblivious Stencil Computations." In: *Proc. 19th Int. Conf. on Supercomputing (ICS)*. ACM, June 2005, pp. 361–366.

[31] Stefan Ganser, Armin Grösslinger, Norbert Siegmund, Sven Apel, and Christian Lengauer. "Iterative Schedule Optimization for Parallelization in the Polyhedron Model." In: *ACM Trans. on Architecture and Code Optimization (TACO)* 14.3 (Aug. 2017), 23:1–23:26.

[32] Pieter Ghysels and Wim Vanroose. "Modeling the Performance of Geometric Multigrid Stencils on Multicore Computer Architectures." In: *SIAM J. Scientific Computing* 37.2 (2015), pp. C194–C216.

[33] Martin Griebl. "The Mechanical Parallelization of Loop Nests Containing while Loops." PhD thesis. University of Passau, 1996.

[34] Martin Griebl, Paul Feautrier, and Armin Größlinger. "Forward Communication Only Placements and Their Use for Parallel Program Construction." In: *Languages and Compilers for Parallel Computing (LCPC 2002)*. Ed. by Bill Pugh and Chau-Wen Tseng. LNCS 2481. Springer, 2005, pp. 16–30.

[35] Tobias Grosser, Albert Cohen, Justin Holewinski, P. Sadayappan, and Sven Verdoolaege. "Hybrid Hexagonal / Classical Tiling for GPUs." In: *Proc. 12th Int. Symp. on Code Generation and Optimization (CGO)*. ACM, 2014.

[36] Armin Größlinger, Martin Griebl, and Christian Lengauer. "Introducing Non-linear Parameters to the Polyhedron Model." In: *Proc. 11th Workshop on Compilers for Parallel Computers (CPC)*. Ed. by Michael Gerndt and Edmond Kereku. Jan. 2004, pp. 1–12.

[37] Tobias Gysi, Carlos Osuna, Oliver Fuhrer, Mauro Bianco, and Thomas C. Schulthess. "STELLA: A Domain-specific Tool for Structured Grid Methods in Weather and Climate Models." In: *Proc. Int. Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*. ACM, 2015, 41:1–41:12.

[38] Wolfgang Hackbusch. *Multi-Grid Methods and Applications*. Springer, 1985.

[39] Jeffrey Hammes, A. P. Wim Böhm, Charlie Ross, Monica Chawathe, Bruce A. Draper, Bob Rinker, and Walid A. Najjar. "Loop Fusion and Temporal Common Subexpression Elimination in Window-based Loops." In: *Proc. 8th IPDPS Reconfigurable Architectures Workshop (RAW)*. IEEE Computer Society, Apr. 2001.

[40] Tom Henretty, Kevin Stock, Louis-Noël Pouchet, Franz Franchetti, J. Ramanujam, and P. Sadayappan. "Data Layout Transformation for Stencil Computations on Short-Vector SIMD Architectures." In: *Proc. 20th Int. Conf. on Compiler Construction (CC)*. Ed. by J. Knoop. LNCS 6601. Springer, 2011, pp. 225–245.

[41] Tom Henretty, Richard Veras, Franz Franchetti, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan. "A Stencil Compiler for Short-Vector SIMD Architectures." In: *Proc. 27th Int. Conf. on Supercomputing (ICS)*. ACM, 2013, pp. 13–24.

[42] Berthold KP Horn and Brian G Schunck. "Determining Optical Flow." In: *Artificial Intelligence* 17.1–3 (1981), pp. 185–203.

[43] François Irigoin and Remi Triolet. "Supernode Partitioning." In: *Proc. 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 1988, pp. 319–329.

[44] El Mostafa Kalmoun, Harald Köstler, and Ulrich Rüde. "3D Optical Flow Computation Using a Parallel Variational Multigrid Scheme with Application to Cardiac C-Arm CT Motion." In: *Image and Vision Computing* 25.9 (2007), pp. 1482–1494.

[45] Hassan Kamal, Joohyun Lee, and Bontae Koo. "An Improved Non-CSD 2-Bit Recursive Common Subexpression Elimination Method to Implement FIR Filter." In: *ETRI J.* 33.5 (2011), pp. 695–703.

[46] Wayne Anthony Kelly. "Optimization Within a Unified Transformation Framework." PhD thesis. Department of Computer Science, University of Maryland at College Park, 1996.

[47] Martin Kong, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and P. Sadayappan. "When Polyhedral Transformations Meet SIMD Code Generation." In: *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. ACM, 2013, pp. 127–138.

[48] Rakesh Krishnaiyer, Emre Kultursay, Pankaj Chawla, Serguei Preis, Anatoly Zvezdin, and Hideki Saito. "Compiler-Based Data Prefetching and Streaming Non-temporal Store Generation for the Intel(R) Xeon Phi(TM) Coprocessor." In: *Proc. 27th Int. Parallel and Distributed Processing Symp. Workshops & PhD Forum (IPDPSW)*. IEEE Computer Society, May 2013, pp. 1575–1586.

[49] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P. Sadayappan. "Effective Automatic Parallelization of Stencil Computations." In: *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. ACM, 2007, pp. 235–244.

[50] Stefan Kronawitter and Christian Lengauer. *Optimizations Applied by the ExaStencils Code Generator*. Tech. rep. MIP-1502. 10 pages. Faculty of Computer Science and Mathematics, University of Passau, Jan. 2015.

[51] Sebastian Kuckuk, Gundolf Haase, Diego A. Vasco, and Köstler Harald. "Towards generating Efficient Flow Solvers with the ExaStencils Approach." In: *Concurrency and Computation: Practice and Experience* 29.17 (2017). Special Issue on Advanced Stencil-Code Engineering, 4062:1–4062:17.

[52] Christian Lengauer et al. "ExaStencils: Advanced Stencil-Code Engineering." In: *Euro-Par 2014: Parallel Processing Workshops*. Ed. by Luis Lopes et al. Vol. 8806. LNCS 8806. Springer, 2014, pp. 553–564.

[53] Shun-tak Albert Leung. "Array Restructuring for Cache Locality." PhD thesis. University of Washington, Department of Computer Science & Engineering, 1996.

[54] Wei Li and Keshav Pingali. "A Singular Loop Transformation Framework Based on Non-Singular Matrices." In: *Int. J. Parallel Programming* 22.2 (Apr. 1994), pp. 183–205.

[55] Vincent Loechner, Benoit Meister, and Philippe Clauss. "Precise Data Locality Optimization of Nested Loops." In: *J. Supercomputing* 21 (Jan. 2002), pp. 37–76.

[56] Shun Long and Michael O'Boyle. "Adaptive Java Optimisation Using Instance-based Learning." In: *Proc. 18th Int. Conf. on Supercomputing (ICS)*. ACM, 2004, pp. 237–246.

[57] Qingda Lu, Xiaoyang Gao, Sriram Krishnamoorthy, Gerald Baumgartner, J. Ramanujam, and P. Sadayappan. "Empirical Performance Model-Driven Data Layout Optimization and Library Call Selection for Tensor Contraction Expressions." In: *J. Parallel and Distributed Computing (JPDC)* 72.3 (2012), pp. 338–352.

[58] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., 1997.

[59] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. "PolyMage: Automatic Optimization for Image Processing Pipelines." In: *SIGARCH Computer Architecture News* 43.1 (Mar. 2015), pp. 429–443.

[60] Dorit Naishlos, Marina Biberstein, Shay Ben-David, and Ayal Zaks. "Vectorizing for a SIMdD DSP Architecture." In: *Proc. 2003 Int. Conf. on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. ACM, 2003, pp. 2–11.

[61] Anthony D. Nguyen, Nadathur Satish, Jatin Chhugani, Changkyu Kim, and Pradeep Dubey. "3.5-D Blocking Optimization for Stencil Computations on Modern CPUs and GPUs." In: *Proc. Int. Conf. on High Performance Computing Networking, Storage and Analysis (SC)*. 13 pages. IEEE Computer Society, 2010.

[62] Andy Nisbet. "GAPS: A Compiler Framework for Genetic Algorithm (GA) Optimised Parallelisation." In: *High-Performance Computing and Networking: International Conference and Exhibition*. Ed. by Peter Sloot, Marian Bubak, and Bob Hertzberger. LNCS 1401. Springer, 1998, pp. 987–989.

[63] Dorit Nuzman, Sergei Dyshel, Erven Rohou, Ira Rosen, Kevin Williams, David Yuste, Albert Cohen, and Ayal Zaks. "Vapor SIMD: Auto-vectorize Once, Run Everywhere." In: *Proc. 9th Int. Symp. on Code Generation and Optimization (CGO)*. IEEE Computer Society, 2011, pp. 151–160.

[64] Dorit Nuzman, Ira Rosen, and Ayal Zaks. "Auto-vectorization of Interleaved Data for SIMD." In: *Proc. 27th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. ACM, 2006, pp. 132–143.

[65] Dorit Nuzman and Ayal Zaks. "Outer-loop Vectorization: Revisited for Short SIMD Architectures." In: *Proc. 17th Int. Conf. on Parallel Architecture and Compilation Techniques (PACT)*. ACM, 2008, pp. 2–11.

[66] Michael F. P. O'Boyle and Peter M. W. Knijnenburg. "Nonsingular Data Transformations: Definition, Validity, and Applications." In: *Int. J. Parallel Programming* 27.3 (June 1999), pp. 131–159.

[67] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. 2nd ed. artima, 2011.

[68] Ozcan Ozturk. "Data Locality and Parallelism Optimization using a Constraint-Based Approach." In: *J. Parallel and Distributed Computing* 71.2 (2011), pp. 280–287.

[69] S. V. Patankar and D. B. Spalding. "A Calculation Procedure for Heat, Mass and Momentum Transfer in Three-dimensional Parabolic Flows." In: *Int. J. Heat and Mass Transfer* 15.10 (1972), pp. 1787–1806.

[70] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and John Cavazos. "Iterative Optimization in the Polyhedral Model: Part II, Multidimensional Time." In: *Proc. ACM SIGPLAN 2008 Conf. on Programming Language Design and Implementation (PLDI)*. ACM, June 2008, pp. 90–100.

[71] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and Nicolas Vasilache. "Iterative Optimization in the Polyhedral Model: Part I, One-Dimensional Time." In: *Proc. 5th Int. Symp. on Code Generation and Optimization (CGO)*. IEEE Computer Society, Mar. 2007, pp. 144–156.

[72] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, and P. Sadayappan. "Combined Iterative and Model-driven Optimization in an Automatic Parallelization Framework." In: *Proc. Int. Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society, 2010, pp. 1–11.

[73] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, P. Sadayappan, and Nicolas Vasilache. "Loop Transformations: Convexity, Pruning and Optimization." In: *Proc. 38th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*. ACM, Jan. 2011, pp. 549–562.

[74] Markus Püschel, Franz Franchetti, and Yevgen Voronenko. "Spiral." In: *Encyclopedia of Parallel Computing*. Ed. by David Padua et al. Springer, 2011, pp. 1920–1933.

[75] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. "Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines." In: *ACM Trans. on Graphics (TOG)* 31.4 (July 2012), 32:1–32:12.

[76] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines." In: *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. ACM, 2013, pp. 519–530.

[77] Prashant Singh Rawat, Martin Kong, Thomas Henretty, Justin Holewinski, Kevin Stock, Louis-Noël Pouchet, J. Ramanujam, Atanas Rountev, and P. Sadayappan. "SDSLc: a multi-target domain-specific compiler for stencil computations." In: *Proc. 5th Int. Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*. ACM, 2015, 6:1–6:10.

[78] Ando Saabas and Tarmo Uustalu. "Program and proof optimizations with type systems." In: *J. Logic and Algebraic Programming* 77.1–2 (2008), pp. 131–154.

[79] Christian Schmitt, Frank Hannig, and Jürgen Teich. "A Target Platform Description Language for Parallel Code Generation." In: *31th Int. Conf. on Architecture of Computing Systems (ARCS)*. Apr. 2018, pp. 59–66.

[80] Christian Schmitt, Stefan Kronawitter, Frank Hannig, Jürgen Teich, and Christian Lengauer. "Automating the Development of High-Performance Multigrid Solvers." In: *Proc. of the IEEE* 106.11 (Nov. 2018). Special Issue on From High-Level Specification to High-Performance Code, pp. 1969–1984.

[81] Christian Schmitt, Sebastian Kuckuk, Frank Hannig, Harald Köstler, and Jürgen Teich. "ExaSlang: A Domain-Specific Language for Highly Scalable Multigrid Solvers." In: *Proc. 4th Int. Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*. ACM, Nov. 2014, pp. 42–51.

[82] Christian Schmitt, Sebastian Kuckuk, Frank Hannig, Jürgen Teich, Harald Köstler, Ulrich Rüde, and Christian Lengauer. "Systems of Partial Differential Equations in ExaSlang." In: *Software for Exascale Computing – SPPEXA 2013-2015*. Ed. by

Hans-Joachim Bungartz, Philipp Neumann, and Wolfgang E. Nagel. LNCSE 113. Springer, 2016, pp. 47–67.

[83] Alexander Schrijver. *Theory of Linear and Integer Programming*. Wiley, 1986.

[84] Jun Shirako, Louis-Noël Pouchet, and Vivek Sarkar. "Oil and Water Can Mix: An Integration of Polyhedral and AST-based Transformations." In: *Proc. Int. Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society, 2014, pp. 287–298.

[85] Daniele G. Spampinato, Diego Fabregat-Traver, Paolo Bientinesi, and Markus Püschel. "Program Generation for Small-scale Linear Algebra Applications." In: *Proc. 2018 Int. Symp. on Code Generation and Optimization (CGO)*. ACM, 2018, pp. 327–339.

[86] Aravind Sukumaran-Rajam and Philippe Clauss. "The Polyhedral Model of Nonlinear Loops." In: *ACM Trans. on Architecture and Code Optimization (TACO)* 12.4 (Dec. 2015), 48:1–48:27.

[87] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. "The Pochoir Stencil Compiler." In: *Proc. 23rd ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 2011, pp. 117–128.

[88] Jan Treibig, Georg Hager, and Gerhard Wellein. "likwid-bench: An Extensible Microbenchmarking Platform for x86 Multicore Compute Nodes." In: *Tools for High Performance Computing 2011*. Ed. by Holger Brunst, Matthias S. Müller, Wolfgang E. Nagel, and Michael M. Resch. Springer, 2012, pp. 27–36.

[89] Jan Treibig, Gerhard Wellein, and Georg Hager. "Efficient Multicore-Aware Parallelization Strategies for Iterative Stencil Computations." In: *J. Computational Science* 2.2 (May 2011), pp. 130–137.

[90] Ulrich Trottenberg, Cornelius W. Osterlee, and Anton Schüller. *Multigrid*. Academic Press, 2000.

[91] Diego A. Vasco, Nelson O. Moraga, and Gundolf Haase. "Parallel Finite Volume Method Simulation of Three-Dimensional Fluid Flow and Convective Heat Transfer for Viscoplastic Non-Newtonian Fluids." In: *Numerical Heat Transfer, Part A: Applications* 66.2 (2014), pp. 990–1019.

[92] Sven Verdoolaege. "*isl*: An Integer Set Library for the Polyhedral Model." In: *Mathematical Software (ICMS 2010)*. Ed. by Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Mobuki Takayama. LNCS 6327. Springer, 2010, pp. 299–302.

[93] Sven Verdoolaege and Tobias Grosser. "Polyhedral Extraction Tool." In: *2nd Int. Workshop on Polyhedral Compilation Techniques (IMPACT)*. http://impact.gforge.inria.fr/impact2012/. Jan. 2012.

[94] Hervé Le Verge. *A Note on Chernikova's Algorithm*. Tech. rep. RR-1662. INRIA, 1992.

[95] H. K. Versteeg and W. Malalasekera. *An Introduction to Computational Fluid Dynamics: The Finite Volume Method*. 2nd ed. Pearson Education Limited, 2007.

[96] Gerhard Wellein, Georg Hager, Thomas Zeiser, Markus Wittmann, and Holger Fehske. "Efficient Temporal Blocking for Stencil Computations by Multicore-Aware Wavefront Parallelization." In: *Proc. 33rd Ann. Computer Software and Applications Conf. (COMPSAC)* 1 (July 2009), pp. 579–586.

[97] Markus Wittmann, Georg Hager, Jan Treibig, and Gerhard Wellein. "Leveraging shared caches for parallel temporal blocking of stencil codes on multicore processors and clusters." In: *Parallel Processing Letters (PPL)* 20.4 (Dec. 2010), pp. 359–376.

[98] Christoph Woller. "Polyhedral Optimization for GPU Offloading in the ExaStencils Code Generator." MA thesis. Faculty of Computer Science and Mathematics, University of Passau, Oct. 2016.

[99] Charles Yount, Josh Tobin, Alexander Breuer, and Alejandro Duran. "YASK - Yet Another Stencil Kernel: A Framework for HPC Stencil Code-generation and Tuning." In: *Proc. 6th Int. Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*. IEEE Computer Society, 2016, pp. 30–39.

[100] Oleksandr Zinenko, Sven Verdoolaege, Chandan Reddy, Jun Shirako, Tobias Grosser, Vivek Sarkar, and Albert Cohen. "Modeling the Conflicting Demands of Parallelism and Temporal/Spatial Locality in Affine Scheduling." In: *Proc. 27th Int. Conf. on Compiler Construction (CC)*. ACM, Feb. 2018, pp. 3–13.