Polyhedral Search Space Exploration in the ExaStencils Code Generator

STEFAN KRONAWITTER and CHRISTIAN LENGAUER, University of Passau, Germany

Performance optimization of stencil codes requires data locality improvements. The polyhedron model for loop transformation is well suited for such optimizations with established techniques, such as the PLuTo algorithm and diamond tiling. However, in the domain of our project ExaStencils, stencil codes, it fails to yield optimal results. As an alternative, we propose a new, optimized, multi-dimensional polyhedral search space exploration and demonstrate its effectiveness: we obtain better results than existing approaches in several cases. We also propose how to specialize the search for the domain of stencil codes, which dramatically reduces the exploration effort without significantly impairing performance.

CCS Concepts: • **Software and its engineering** → **Software performance**; *Source code generation*; • **Computing methodologies** → *Discrete space search*; • **Mathematics of computing** → Combinatorial optimization;

Additional Key Words and Phrases: ExaStencils, polyhedron model, polyhedral search space exploration, stencils

ACM Reference Format:
Stefan Kronawitter and Christian Lengauer. 2018. Polyhedral Search Space Exploration in the ExaStencils Code Generator. *ACM Trans. Arch. Code Optim.* 1, 1 (August 2018), 25 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The polyhedron model [13] is a powerful algebraic representation of loop nests in which any combination of affine transformations can be represented by a single function. The main challenge is the automatic selection of a suitable function that leads to the best performance. In the classic polyhedron model there are basically two different approaches:

(1) Use a heuristics to identify directly an optimizing transformation of the loop nest.
(2) Generate the space of all legal transformations that the model covers and search it for the one with the best performance on the given hardware.

Examples of the former are Feautrier's scheduling algorithm [11, 12] and the PLuTo algorithm [6]. This approach has the advantage of being comparatively cheap and not requiring experiments on the targeted hardware during optimization and compilation. Examples of the latter are LeTSeE [33, 34] and Polyite [15]. The exploration in the alternate approach requires the generation, compilation and actual execution of several variants on the targeted hardware. But the results are potentially of higher quality. A focus on a narrow domain mitigates the search effort.

We present both a new exploration technique and a set of additional filters that enable a low-overhead search for an optimal transformation in the domain of stencil codes. The exploration

Authors' address: Stefan Kronawitter, stefan.kronawitter@uni-passau.de; Christian Lengauer, University of Passau, Chair of Programming, Innstraße 33, 94032, Passau, Germany, christian.lengauer@uni-passau.de.

XXXX-XXXX/2018/8-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

itself is not domain-specific and traverses only a subset of the search space — in contrast to other exploration techniques [15, 33, 34] that either generate a restricted but manageable search space, by removing potentially bad schedules, and traverse this space exhaustively, or use a genetic algorithm to traverse parts of the search space. To select efficiently the subset to be traversed, we employ a dual representation of the search space to be discussed later. The size of the subset is controlled via a single parameter and even small values provide a good spread of possible transformations.

The set of filters must be specific to the domain of stencil codes. Stencil codes are widely used in a variety of scientific application areas. The solution of discretized partial differential equations (PDEs) is one example. Since their composition and performance tuning can become quite complex, we are developing in project ExaStencils[1] [27] a multi-layered domain-specific language (DSL), called ExaSlang [39], for the specification of geometric multigrid solvers and a corresponding automatic generator of optimized target code [26]. Among the most time-consuming phases of multigrid solvers are pre- and post-smoothing, which consist of few stencil applications, typically not more than five, with very low computational intensity (i.e., ratio of the number of computations to the number of memory accesses). This clearly renders their application memory-bandwidth-bound. In order to optimize these phases, data locality is increased by combining successive stencil applications to a single loop nest. The polyhedron model is well suited for this type of optimization. However, the identification of the transformation that leads to the best performance is not trivial.

We make the following contributions:

- a new technique of polyhedral search space exploration that selects efficiently a good subset of all legal affine transformations,
- a specialization of the search space exploration to the domain of stencil codes to minimize the exploration effort,
- a demonstration of its effectiveness and
- a comparison with some other optimization techniques on a set of a dozen stencil codes.

The rest of the paper is organized as follows. Section 2 provides a brief overview of the polyhedron model. The search space for a polyhedral exploration is described in Section 3.1. The basic concepts of an abstract exhaustive exploration and of our new guided exploration follow in Sections 3.2 and 3.3 respectively. Section 3.4 discusses the implementation of the exploration in the ExaStencils code generator, and introduces a set of heuristic filters, tailored to the domain of stencil codes, to reduce the exploration effort. An evaluation and comparison of the various techniques for different 2D and 3D Jacobi and Red-Black Gauss-Seidel (RBGS) codes follow in Section 4, while Section 5 discusses related work and Section 6 concludes.

2 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 and all concepts required to perform a search space exploration.

[1] http://www.exastencils.org

```
  for (int i = 0; i < n; i++)
    for (int j = 0; j < m; j++) {
S:    A[i][j] = A[i][j] + 0.2 * (A[i-1][j] +
                  A[i+1][j] + A[i][j-1] + A[i][j+1]);
    }
```

Fig. 1. Sample loop nest.

The integer set library (isl) [43] 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 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 that are distinguishable by the statement to which an instance belongs and the values of the iteration variables 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 Figure 1. Its iteration domain can be written as follows:

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

This notation follows the syntax of isl. The identifier list `[n,m]` 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 statement `S` is specified inside the curly braces. Constraints must be in Presburger arithmetic.

2.2 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 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 time. The schedule of the loop nest in Figure 1 is

```
{ S[i,j] -> [i,j] }
```

which can alternatively be represented by the matrix

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

The two elements of `S` are the values of the loop iterators, which means that this statement is surrounded by two loops. In this example, each statement instance is mapped to a unique point in a three-dimensional time, i.e., the schedule is bijective. We call a schedule that is bijective *complete*, otherwise it is *incomplete*.

2.3 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 or array location. For example, the read and write accesses in the loop nest of Figure 1 are represented as follows:

```
reads:  { S[i,j] -> A[i,j];
          S[i,j] -> A[i-1,j]; S[i,j] -> A[i,j-1];
          S[i,j] -> A[i+1,j]; S[i,j] -> A[i,j+1] }
writes: { S[i,j] -> A[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. 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 and one of the read accesses, e.g., `{ S[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,m] -> { S[i,j] -> S[i+1,j] : 0 <= i < n-1 and 0 <= j < m }
```

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 [11, 12, 35]. 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 that there is 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 called *loop-carried*, otherwise it is called *loop-independent* or also *text-carried*. For dimensions higher than c, the dependence is irrelevant.

2.4 Optimization

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

(1) extract a polyhedral representation
(2) compute the data dependences
(3) find an optimal schedule
(4) generate a target loop nest

There are libraries for extracting a polyhedral representation from a C-like source code, such as Clan [4] or pet [44]. 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. 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 PLuTo algorithm [6] that optimizes for coarse-grain parallelism and data locality.

3 POLYHEDRAL SCHEDULE EXPLORATION

To illustrate the basic concepts, we introduce first an abstract algorithm for a naïve, exhaustive exploration. Subsequently, we present our restricted exploration, additionally powered by a set of filters tailored to the domain of stencil codes.

Both algorithms are designed to generate complete schedules only. The omited schedule dimensions of an incomplete schedule still have an influence on the performance. They may affect tiling, vectorization, the memory access pattern, and other potentially less obvious properties. On the downside, generating a complete schedule increases the exploration effort.

3.1 Search Space

3.1.1 One-Dimensional Transformation 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] -> { S1[i,j] : 0 <= i,j < n; S2[i] : 0 <= i < m }
```[2]

Each schedule for this iteration domain has the following form, which is called the *prototype schedule* Θ_0:

```
[n,m] -> { S1[i,j] -> [i1*i + j1*j + n1*n + m1*m + c1];
           S2[i]   -> [i2*i + n2*n + m2*m + c2] }
```

The set of all possible affine transformations can then be written

```
{ [i1,j1, i2, n1,m1, n2,m2, c1,c2] } = ℤ^9
```

where `i1`, `j1`, and `i2` are the coefficients of the loop iterators for statement `S1` and `S2` respectively, `n1`, `m1`, `n2`, and `m2` are the coefficients of the structural parameters, and `c1`, `c2` are the constant parts. Following this notation, the next two lines are different representations of the same schedule:

```
                [1,1, -1, 2,0, 0,1, 3,-2]
[n,m] -> { S1[i,j] -> [i+j+2n+3]; S2[i] -> [-i+m-2] }
```

This is a variant of the matrix representation presented in Section 2.2.

One-dimensional schedules with only zero coefficients for all iterators of each statement are called *constant*. These do not correspond to a loop in the target code but they do represent a textual ordering.

3.1.2 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

$$\Theta_0^T(\vec{x}_T) - \Theta_0^S(\vec{x}_S) \geq 0 \tag{1}$$

holds for every pair of statement instances in the dependence polyhedron. These constraints can be computed by applying an affine form of the Farkas Lemma and a Fourier-Motzkin elimination for the prototype schedule Θ_0^X for statement X [11, 40]. Among other libraries, isl provides an implementation for this purpose, which is used in our work. The basic idea of the Farkas Lemma is that, for any non-empty polyhedron $\mathcal{D}$ represented by k inequalities $\vec{a}_k \cdot \vec{x} + b_k \geq 0$, an affine function is non-negative everywhere in $\mathcal{D}$ iff it has the form

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

This provides an alternative representation of the left-hand side of Equation (1):

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

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.

[2] `0 <= i,j < n` is short for `0 <= i < n and 0 <= j < n`

ALGORITHM 1: An abstract, exhaustive polyhedral search space exploration.

Input: A set of not yet carried dependences D and an incomplete schedule s
Output: A list 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
```

3.1.3 Multi-dimensional Transformations. 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. 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 [6, 28]. The satisfaction constraints for carried dependences are no longer necessary and can be removed for further inner dimensions to enlarge the search space.

Note that Equation (1) captures both weak and strong satisfaction of the dependences. But, since we compute a complete 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 $\Theta^S_c(\vec{x}_S) \neq \Theta^T_c(\vec{x}_T)$ holds for each dependence from $\vec{x}_S$ to $\vec{x}_T$. That is, the dependence has been strongly satisfied at some level.

3.1.4 Tiling. A mandatory optimization of higher-dimensional loop nests is tiling: the transformed iteration space 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 [22], which is the case for subsequent dimensions if they weakly satisfy all data dependences in the considered fused loop nest [6, 17]. 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.

3.2 Exhaustive Exploration

Based on the search space introduced in the previous subsection, Algorithm 1 is an abstract description of an 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 3.1.2. As mentioned earlier, a strong satisfaction is not required here. Additionally, only schedule dimensions linearly independent with the previously selected ones need to be explored, so appropriate constraints are added to the search space (line 3).

```
for (int t = 0; t < T; 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][j+1][k] + A[(t+1)%2][i][j][k+1]
             + A[(t+1)%2][i-1][j][k] + A[(t+1)%2][i][j-1][k] + A[(t+1)%2][i][j][k-1]);
```

Fig. 2. 3D 1st-order Jacobi stencil.

For this algorithm, linearly independent constraints are not computed statementwise 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 4–5). Otherwise, one has to iterate over all of its elements *sd* (line 7), add each one as an 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 sufficently 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 (Figure 2). If all variables that define the search space are restricted to `-1`, `0` and `1`, there are 2186 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 $27^4 = 531441$ 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 [33, 34].

3.3 Guided Exploration

We pursue a heuristic approach to an efficient polyhedral search space exploration tuned to stencil codes. It is specific to stencil codes; in its current state, it cannot be applied to other programs. However, we believe it could be generalized or adapted to other domains, and we provide some required modifications to this end later in this section. 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 of a dual representation. Therefore, the search space we are exploring (as presented in Section 3.1.2) is larger than with other techniques such as, e.g., the Feautrier scheduler [11, 12], which greedily carries dependences by adding appropriate constraints outermost. We use basically the same search space as the PLuTo+ algorithm and the isl scheduler, but the former restricts the absolute values of the schedule coefficients and adds dimensions to minimize the upper bound of the reuse distances [1]. 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. Our guided exploration is implemented in the ExaStencils code generator.

ALGORITHM 2: A guided polyhedral search space exploration for stencil codes.

Input: A set of not yet carried dependences D
Output: A list 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
```

3.3.1 Generator Representation. In contrast to the implicit, constraint-based representation referred to in the previous sections, 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:

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

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 [40, 45], 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.

3.3.2 Search Space Generation. The new exploration technique is specified by Algorithm 2. 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 3.1.2,

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.

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 pointwise 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 straightforward. 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, 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. Next, the recursive function `guided_exploration_tileable` is called with an empty schedule.

3.3.3 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 1, 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.

3.4 Exploration in ExaStencils

The ExaStencils code generator addresses multigrid methods. To their most time-consuming parts belong pre- and post-smoothing, which consist of few stencil applications. For example, a study by Ghysels and Vanroose [16] 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, an additional overhead emerges in multiple smoothing steps, caused by the communication between nodes that compute

Table 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]`	`S: 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]`	`T: a[x-1,y-1,z-x-y+1]`
`T: a[i,j,k]`	`T[i,j,k] -> [x=i+1,y=j+1,z=i+j+k+2]`	`T: a[x-1,y-1,z-x-y]`

neighboring regions of the whole field. Thus, in the case of distributed memory, it pays to keep the number of smoothing steps low — typically not higher than 5, which justifies a complete unrolling of the time loop in the ExaStencils code generator and, thus, simplifies the border handling. Its very low computational intensity renders smoothing memory-bandwidth bound. In order to optimize it, data locality must be increased, e.g., by a technique similar to temporal blocking: fully unrolling the time loop and fusing the time steps, i.e., the stencil applications, to a single loop nest. The polyhedron model is well suited for this type of optimization. Different techniques exist that are easy to integrate in a code generator, such as the PLuTo algorithm, the isl scheduler, but also independent platforms such as PolyMage [30] or Polyite [15]. However, as our evaluation revealed (see Section 4), cases remain in which none of the above is able 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 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 time. 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.

3.4.1 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 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 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`, 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. 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 [20]) 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

```
for (int x=start; x<end; x+=4) {
  vecL = load_unaligned(&in[x-1]);
  vecC = load_aligned(&in[x]);
  vecR = load_unaligned(&in[x+1]);
  ...
```

```
vecC  = load_aligned(&in[start-4]);
vecRR = load_aligned(&in[start]);
for (int x=start; x<end; x+=4) {
  vecLL = vecC;
  vecC  = vecRR;
  vecRR = load_aligned(&in[x+4]);
  vecL  = permute(vecLL, vecC);
  vecR  = permute(vecC,  vecRR);
  ...
```

Fig. 3. Trading memory accesses with in-register operations for a vector size of four elements.

possible in most cases. Similar and more advanced techniques have been presented elsewhere [24]; 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:

$$(I): \ x = i + 1$$
$$(II): \ y = j + 1$$
$$(III): \ z = i + j + k + 1$$

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: $(III) - (I) - (II) + 1$. The constant summand +1 gives rise to the constant part in the transformed array subscript. To get rid of it, it can be merged with any schedule dimension, depending on which one results in a legal schedule. For example, $(III) : z = i + j + k + 1$ can be replaced with $(III') : z = i + j + k + 2$, which results in the expression $(III') - (I) - (II)$ 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 shown in Figure 3. The left code example contains unaligned load instructions to fetch three partially overlapping vectors. The right version is semantically equivalent but contains only aligned load operations, which requires 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 to perform the permutation. However, the latter can operate on registers only, which increases the performance of bandwidth-bound codes.

3.4.2 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 good performing schedules have in common and what distinguishes them from the others. 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.

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 2 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 idea behind 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 [11, 12], which carries dependences greedily as early as possible. In contrast, we also allow dependences to be carried by the innermost, constant dimension.

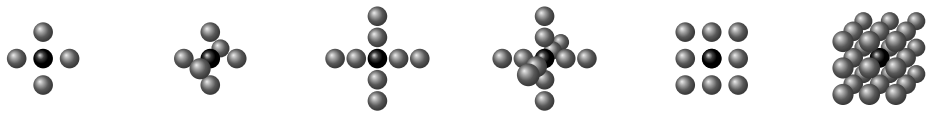
Level 2. Every schedule with a non-zero coefficient for the innermost loop iterator in any but the last non-constant dimension is discarded. For the codes generated, this preserves only the schedules whose innermost target loop traverses the memory linearly, since the innermost loop iterator in the original 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 [24] that combine the polyhedron model with vectorization. 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 that have some dependences 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 can 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 in this section, 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 supports vectorization of this type of loops only. 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.



(a) 2D 1st-order (b) 3D 1st-order (c) 2D 2nd-order (d) 3D 2nd-order (e) 2D dense (f) 3D dense

Fig. 4. 2D and 3D stencil shapes.

Level 7. Enforce small schedule coefficients and constants. 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.

3.4.3 Integration in our Code Generator. The presented guided exploration, as well as the filter levels were implemented in the ExaStencils code generator. The exploration is performed semi-automatically. In the current state, the user has to specify both the path to a configuration file containing all explored schedules and an identifier for the schedule to use in the current run. If the configuration file does not exist, the actual exploration is performed and the file is generated. 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[3], is straightforward, too. The drawback is that an external logic is required to schedule the different calls to the ExaStencils code generator.

4 EVALUATION

The guided exploration and all filter levels have been implemented in the ExaStencils code generator. In total, twelve different stencil codes were evaluated, six two- and six three-dimensional ones.

The ExaStencils code generator, its input code for this work and the results of the exploration can be found on a supplementary Web page[4].

4.1 Experiment Description

Six of the twelve stencils are Jacobi versions with constant coefficients, i.e., the coefficients of the center element and the neighbors are compile-time constants. Their stencil shapes are shown in Figure 4. A Jacobi stencil accesses separate memory locations for reading and writing, which are interchanged after each time step. For both 1st-order stencil shapes, we also evaluated a variable-coefficient version, for which the coefficient of each neighbor depends on the array index of the data element. The actual values are precomputed and stored in separate arrays, which increases the amount of data that must be loaded from memory. Finally, RBGS versions with both constant and variable coefficients of the 1st-order stencils were evaluated. RBGS stencils read from and write to the same array and are applied in two sweeps that update alternately every other array element.

The structure of the smoother codes in our experiments is similar to Figure 2 with constants in place of `a`, `b` and `c`. As mentioned previously, our code generator has the property of unrolling the `t`-loop and applying additional optimizations such as a vectorization. For PLuTo, we evaluated both the versions with and without the time loop unrolled.

In the rest of the evaluation, 'ccX' and 'vcX' are abbreviations for constant, respectively variable coefficients, where 'X' specifies the stencil shape: '1' or '2' for 1st or 2nd order and 'd' for dense.

[3]https://slurm.schedmd.com/

[4]https://www.fim.uni-passau.de/cl/staff/kronawitter/taco18/

Table 2. Exploration time and properties of all twelve experiments. The exploration time for filter levels 0 and 7 contain the dependence analysis, the search space generation, the Chernikova call and the actual schedule enumeration. The filters are applied after Chernikova is called, so its run time does not depend on the filter level. The number of rays combined for level 0 is listed; for level 7, it was always 3.

	Jacobi 3D				RBGS 3D	
	cc1	cc2	ccd	vc1	cc1	vc1
expl. time filter level 0 [s]	26.67	23.03	48.59	14.85	52.70	21.85
expl. time filter level 7 [s]	0.71	0.84	0.85	0.39	1.96	0.69
Chernikova run time [s]	0.24	0.31	0.25	0.09	0.90	0.17
#time steps	5	4	3	3	4	2
#search space dimensions	50	40	30	30	80	40
#rays	26	25	22	24	29	25
#comb. rays for level 0	2	2	3	2	2	2
#schedules filter level 0	12048	12048	21872	12048	12048	12048
#schedules filter level 7	4	4	6	4	4	4

	Jacobi 2D				RBGS 2D	
	cc1	cc2	ccd	vc1	cc1	vc1
expl. time filter level 0 [s]	0.57	0.68	0.75	0.44	1.40	0.71
expl. time filter level 7 [s]	0.36	0.42	0.46	0.27	0.99	0.45
Chernikova run time [s]	0.11	0.16	0.17	0.09	0.59	0.20
#time steps	5	5	5	4	5	3
#search space dimensions	35	35	35	28	70	42
#rays	18	18	18	17	23	19
#comb. rays for level 0	3	3	3	3	3	3
#schedules filter level 0	108	108	112	108	108	108
#schedules filter level 7	1	1	1	1	1	1

4.2 Experiment Setup

All experiments were executed on an Intel Xeon E5-2690 v2 processor. It consist of 10 Ivy Bridge EP cores, each running at 3.3 GHz when fully loaded and with Intel Turbo Boost enabled. The generated C++ code was compiled with Intel's icc compiler, version 17, using aggressive optimizations (-O3). 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 auto-vectorizer 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 also not encounter a situation in which the vectorizer of icc was able to generate a faster code than our vectorizer, which additionally justifies focusing on the latter. Other optimizations, such as address precalculation and a rectangular tiling, were applied to all versions generated. The former precomputes a maximally sized part of the linearized array index expression outside the innermost loop. This is a standard compiler optimization implemented in production compilers [2], but not always applied by them. Our code generator applies it in case the compiler does not. The rectangular tiling is realized with a transformation applied by the isl that adds the desired dimensions outermost, e.g., `{ S[i,j] -> [floor(i/32),floor(j/128),i,j] }`.

4.3 Exploration Statistics

For the three-dimensional experiments, each stencil application updates 512^3 double-precision elements, while 16384^2 values are computed in the two-dimensional case. To prevent specialized border-handling code, so-called ghost layers are added to the arrays: elements before and after the ones to be updated are inserted in every dimension to ensure the presence of neighbors to access. 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 2. In the three-dimensional case without any filter applied, not more than one minute was required on a standard workstation with an Intel Core i7-6700, using a single thread (the exploration itself was not parallelized). The Chernikova algorithm completes in less than one second, while the majority of the run time is spent in function `guided_exploration_tileable` of Algorithm 2. 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.

4.4 Performance Evaluation

A performance comparison of our exploration with other algorithms and tools is shown in Table 3. 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. Note that, when computing a new element for a stencil application, not only the neighbors, as specified by the stencil shape, are read from the memory but one additional value is read from a separate array, as depicted in Figure 2. However, as 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. For each experiment, we evaluated every combination of tile sizes from the set $\{4, 8, 10, 16, 20, 32, 50, 64, 100, 128, 150, 200, 256, \infty\}$[5] for all but the innermost dimension. For the latter, 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 1000 in 3D, so our exploration and Polyite both use the tile size of the isl heuristics experiment. The only exception is RBGS 2D vc1, for which tiling of the inner dimension does not lead to good results. Therefore, the exploration was repeated with no tiling at all.

4.4.1 Baseline. As a baseline for our experiments, we choose the roofline performance of a single time step, i.e., without any kind of temporal blocking 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, 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 still in the processor's cache) and one updated value is written to memory.

[5] "∞" means a value larger than the iteration domain for this dimension is chosen.

Table 3. Performance comparison of all experiments. The performance of the fastest version per stencil is given in both million lattice updates per second (MLUP/s) and billion floating-point operations per second (GFLOP/s). The performance results of all experiments are given as the percentage of the best one.

		Jacobi 3D				RBGS 3D	
		cc1	cc2	ccd	vc1	cc1	vc1
	best [MLUP/s]	4494	2682	1862	1038	3176	536
	best [GFLOP/s]	44.9	45.6	59.6	17.6	31.8	9.1
1.	baseline	34%	56%	81%	53%	32%	57%
2.	Exploration						
	filter level 0	99%	99%	98%	100%	90%	96%
	filter level 2	99%	99%	98%	100%	72%	96%
	filter level 7	99%	99%	96%	99%	72%	94%
	tile opt.	100%	100%	99%	100%	100%	100%
3.	isl						
	simple	11%	12%	6%	31%	20%	54%
	heuristics	96%	96%	6%	100%	22%	55%
4.	PLuTo						
	rectangular	68%	85%	87%	90%	8%	41%
	unrolled	50%	36%	49%	53%	72%	63%
	diamond	72%	85%	100%	91%	—	—
5.	PolyMage	49%	57%	70%	47%	31%	44%
6.	Polyite	34%	50%	53%	65%	—	—

		Jacobi 2D				RBGS 2D	
		cc1	cc2	ccd	vc1	cc1	vc1
	best [MLUP/s]	5973	4929	5060	2110	3784	1250
	best [GFLOP/s]	47.8	64.1	65.8	27.4	30.3	16.3
1.	baseline	25%	31%	30%	32%	27%	30%
2.	Exploration						
	filter level 0	82%	88%	90%	82%	73%	100%
	filter level 7	82%	88%	88%	76%	72%	100%
	tile opt.	83%	89%	90%	83%	73%	100%
3.	isl						
	simple	8%	7%	7%	12%	16%	11%
	heuristics	82%	88%	7%	77%	16%	10%
4.	PLuTo						
	rectangular	50%	61%	55%	47%	13%	16%
	unrolled	62%	44%	48%	70%	100%	81%
	diamond	75%	83%	87%	86%	—	—
5.	PolyMage	100%	100%	100%	100%	64%	85%
6.	Polyite	23%	29%	33%	31%	—	—

The remaining one is due to the write-allocate for the update, since a cache-line has to loaded before it be can be modified. Therefore, the roofline is: (48500 MB/s)/(4 · 8 B/LUP) = 1516 MLUP/s. A performance close to the roofline without a temporal reuse between subsequent time steps can be achieved by choice of a suitable tile size.

Table 4. Tile sizes used in the experiments. The leftmost entry is the tile size for the outer loop, the rightmost is for the innermost loop. PolyMage autotunes the tile size automatically, so it does not appear here. RBGS stencils cannot be diamond-tiled.

		expl. / Polyite	expl. tile opt.	isl simple	isl heuristics	PLuTo rectangular	PLuTo unrolled	PLuTo diamond
Jacobi 3D	cc1	∞, 200, ∞	∞, 150, ∞	100, 128, ∞	∞, 200, ∞	∞, 150, 4, ∞	16, 20, ∞	20, 20, 4, 150
	cc2	∞, 128, ∞	∞, 150, ∞	150, 50, ∞	∞, 128, ∞	∞, 100, 4, ∞	50, 10, ∞	100, ∞, 4, ∞
	ccd	∞, 100, ∞	200, 150, ∞	16, 256, ∞	∞, 100, ∞	∞, 200, 100, ∞	4, 4, ∞	8, 8, 4, ∞
	vc1	∞, 100, ∞	∞, 50, ∞	∞, 50, ∞	∞, 100, ∞	∞, 50, 4, 200	8, 8, ∞	50, ∞, 4, ∞
RBGS 3D	cc1	150, 100, ∞	∞, 200, ∞	150, 150, ∞	150, 100, ∞	∞, 8, 100, ∞	20, 20, ∞	
	vc1	∞, 64, ∞	∞, 100, ∞	128, 50, ∞	∞, 64, ∞	∞, 16, 8, ∞	8, 8, ∞	
Jacobi 2D	cc1	16, ∞	128, ∞	128, ∞	16, ∞	∞, 256, 200	4, 150	150, 150, 100
	cc2	256, ∞	10, ∞	32, ∞	256, ∞	∞, 256, 150	10, 128	100, 100, 128
	ccd	32, ∞	150, ∞	100, ∞	32, ∞	∞, 256, 150	4, 150	64, 64, 150
	vc1	150, ∞	8, ∞	150, ∞	150, ∞	∞, 256, 100	4, 256	50, 50, 100
RBGS 2D	cc1	256, ∞	200, ∞	100, ∞	256, ∞	∞, 4, ∞	16, 100	
	vc1	∞, ∞	4, ∞	150, 100	128, 64	∞, 4, ∞	4, 128	

4.4.2 Exploration Overview. The two, respectively, three subsequent rows 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 runs, as explained earlier. 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 in Section 4.2. The tile sizes and performance values for these "tile opt." versions are given in Tables 3 and 4, respectively. Except for RBGS 3D cc1, the initial tile sizes used for our exploration are almost optimal. For the 3D experiments, no other tool or algorithm tested was able to generate a faster code than our exploration. The same is true for the RBGS 2D vc1 stencil. A more in-depth analysis of the most interesting results from the exploration is presented later in this section.

4.4.3 isl Scheduler. Both isl versions have been implemented 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 isl scheduler expects three sets of dependences as input:

validity These dependences are respected by the generated schedule. The source of each of these dependences is always executed before the corresponding target.

proximity The scheduling algorithm tries to minimize the distance between the source and target of these dependences.

coincidence If possible, source and target of these dependences are executed in the same iteration of the resulting loop nest.

In experiment "isl simple", we run the isl scheduler by passing all dependences (see Section 2.3) for validity, proximity, and coincidence. In experiment "isl heuristics", we pass a reduced proximity set. The reduction heuristics is that, for each source of multiple data dependences, only the one with the lexicographic smallest target is kept. Since, in most cases, this heuristics has a dramatically positive impact on the performance, we make it the default in our code generator when no exploration is

performed. We did not modify any other settings of the isl scheduler, since we did not expect any improvement.

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

```
{ S0[i,j,k] -> [i,   j,   k,   0];
  S1[i,j,k] -> [i+1, j+1, k+1, 1]; ... },
```

while the schedule with it is

```
{ S0[i,j,k] -> [i,   i+j,   i+k,   0];
  S1[i,j,k] -> [i+1, i+j+1, i+k+1, 1]; ... }.
```

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

```
{ S0[i,j,k] -> [j,   i+j,   i+j+k,   0];
  S1[i,j,k] -> [j+1, i+j+1, i+j+k+1, 1]; ... }
```

4.4.4 PLuTo Algorithm. PLuTo with rectangular and with diamond tiling show the performance of the schedule detected by PLuTo version 0.11.4 with and without diamond tiling [3, 5] 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 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. The reason that diamond tiling performs worse than reported by others [5] could be the very small number of time steps, which is less than 6 in all our examples.

4.4.5 PolyMage. PolyMage [30] 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 from 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 and described in Section 4.2 was supplied. The Intel compiler served to generate the final binary and perform low-level optimizations. 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 [25], 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.

Table 5. Number of schedules and their performance range for each filter level for Jacobi 3D cc1. The %Opt range specifies the performance of the fastest and slowest variant for the given filter level as percentage of the fastest one of both explorations.

	combine two rays		combine three rays	
Level	#Schedules	%Opt	#Schedules	%Opt
0	12048	0.2 - 100.0	152592	
1	368	6.7 - 100.0	22832	
2	128	29.0 - 100.0	496	22.8 - 100.0
3	48	47.4 - 100.0	80	47.4 - 100.0
4	48	47.4 - 100.0	80	47.4 - 100.0
5	24	92.6 - 100.0	40	92.6 - 100.0
6	4	96.1 - 100.0	6	96.1 - 100.0
7	4	96.1 - 100.0	4	96.1 - 100.0

4.4.6 Polyite. Polyite [15] is a polyhedral search space exploration tool built for Polly/LLVM. 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 later in this section, 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] -> [3t + 2i + 2j - 3, 15t + 10j, t - 2i, 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 stencil.

This completes our explanations of the rows in Table 3. We continue with further discussions of the most interesting exploration results by column.

4.4.7 Exploration: Jacobi 3D cc1. Let us first look at the leftmost column: an experiment with a sequence of five iterations of a 3D 1st-order Jacobi stencil with 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 5 reveals how many schedules at each filter level were explored in the two versions.

Without any filter, the combination of two rays leads to 12048 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, the fastest schedule 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 5. For each level, there are both a jittered scatterplot and a violin plot. A jittered scatterplot contains every data point with a 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

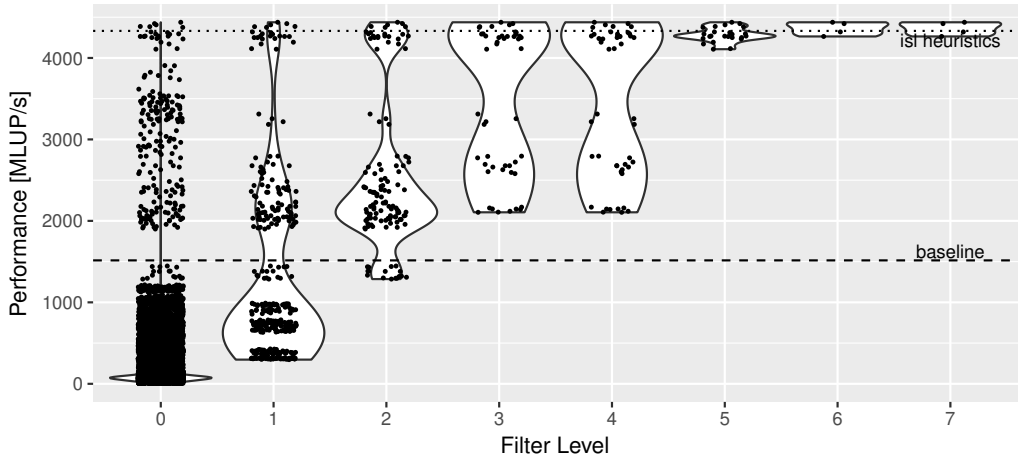


Fig. 5. Performance distribution of all filter levels for Jacobi 3D cc1.

best ourcode generator can do without an exploration. For filter level 0, the vast majority (more than 95%) of all schedules lead to a very poor performance — even worse than the unoptimized version. The first two filters already reduce the set of explored schedules to 128, and each of these results in a performance significantly better than the baseline. With all filters applied, not more than four remain.

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.

4.4.8 Exploration: Jacobi 2D vc1. This experiment covers a sequence of four iterations of a 2D 1st-order Jacobi stencil with variable 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 108 schedules, as shown in Table 6. Thus, we skipped an exploration with two rays combined.

Filter level 6 is sufficient to select a single schedule, however, not the one with the best performance. Filter level 5 is the highest level that the best schedule passes — and one other schedule that has worse performance, unclear why. The only difference between the two is that the stride of the outer loop is −1 for the best and +1 for the other schedule. Both schedules share the structure of the loop nest, the number of iterations per loop and also the stride of the inner loop. Even an examination of the assembler instructions emitted by the Intel compiler for both versions did not clarify why both perform so very differently.

The performance distribution for the schedules explored at all filter levels is shown in Figure 6. Note that, for filter levels 5 to 7, in the violin plots, no actual violins but only the data points themselves appear, since the number of data points is too small.

4.4.9 Exploration: RBGS 3D cc1. Our third experiment concerns the sequence of four iterations of a 3D 1st-order RBGS stencil with constant coefficients; see Table 7 and Figure 7.

Table 6. Number of schedules and their performance range for each filter level for Jacobi 2D vc1. The %Opt range specifies the performance of the fastest and slowest variant for the given filter level as percentage of the fastest one for level 0.

	combine three rays	
Level	#Schedules	%Opt
0	108	2.4 - 100.0
1	44	12.9 - 100.0
2	8	68.9 - 100.0
3	4	79.9 - 100.0
4	4	79.9 - 100.0
5	2	91.9 - 100.0
6	1	91.9 - 91.9
7	1	91.9 - 91.9

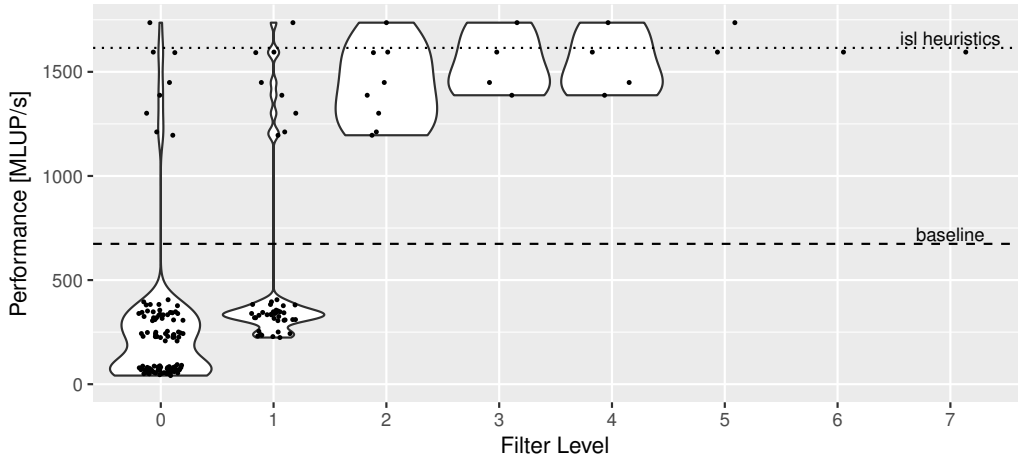


Fig. 6. Performance distribution of all filter levels for Jacobi 2D vc1.

Already the first filter removes the best schedules for this stencil. Its purpose is to ensure that the resulting code can be vectorized. However, the code generator is currently not yet capable of vectorizing a RBGS stencil or, more specifically, of vectorizing the non-contiguous memory accesses induced by it. When this problem is solved, we expect the schedules that pass higher filter levels to perform much better on the processors' vector units. This should lead to a performance distribution similar to the one of the Jacobi 3D cc1 experiment.

This is one of the stencil codes we encountered for which the isl scheduler does not return a good variant. Its performance is even below the baseline for a single time step, which can be reached with tiling alone.

5 RELATED WORK

Stencil codes are targeted by a wide range of tools and algorithms. Many come with a tiling scheme but without a DSL [8–10, 14, 31, 42]. Let us briefly comment on the ones that offer a DSL. Patus [7] is a code generation framework for the domain of stencil codes. It can generate code for

Table 7. Number of schedules and their performance range for each filter level for RBGS 3D cc1. The %Opt range specifies the performance of the fastest and slowest variant for the given filter level as percentage of the fastest one of both explorations.

	combine two rays		combine three rays	
Level	#Schedules	%Opt	#Schedules	%Opt
0	12048	0.0 - 100.0	152592	
1	368	24.3 - 80.6	22832	
2	128	35.0 - 80.6	496	19.7 - 80.6
3	48	67.0 - 80.6	80	67.0 - 80.6
4	48	67.0 - 80.6	80	67.0 - 80.6
5	24	67.2 - 80.4	40	67.2 - 80.6
6	4	76.0 - 80.2	6	76.0 - 80.5
7	4	76.0 - 80.2	4	76.0 - 80.2

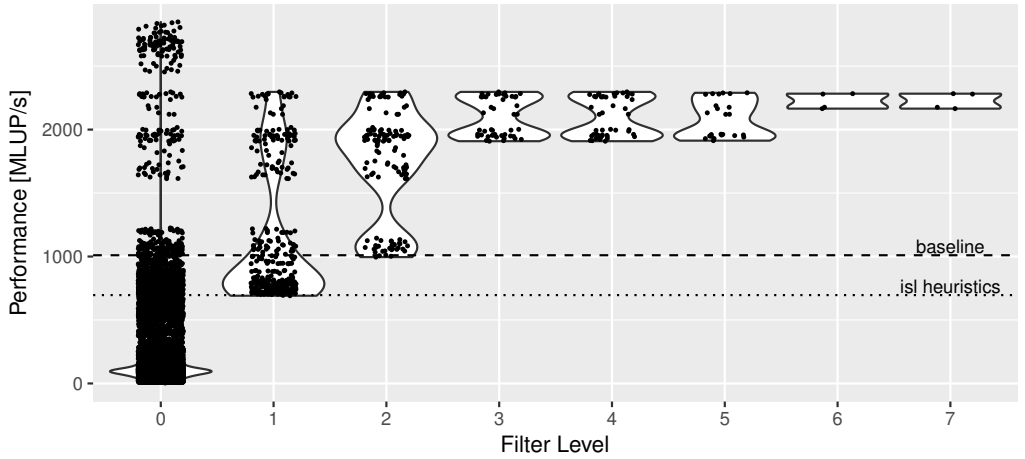


Fig. 7. Performance distribution of all filter levels for RBGS 3D cc1.

both CPUs and GPUs and its strong point is the autotuning of different optimization parameters. SDSLc [38] 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 [36, 37] and PolyMage [30] are image processing libraries and, therefore, can be used to generate optimized codes for stencil applications, too. In contrast to the custom tool chains on which all of these approaches are based, Pochoir [41] and STELLA [19] 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 meta-programming only.

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 [11, 12] 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. [6] 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 [21], overlapped tiling [25], diamond tiling [3, 5], and hexagonal tiling [18] were developed. The scheduler by Kong et al. [24] was developed to take advantage of the vector units provided by modern processors. It directly computes 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 [32] in the GAPS framework. Long et al. [29] presented a polyhedral exploration to optimize Java programs. Like the GAPS framework, it uses the Unified Transformation Framework (UTF) developed by Kelly [23]. Both suffer from the fact that they generate huge search spaces with predominantly illegal transformations.

With LeTSeE [34], Pouchet et al. 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 [33] 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 3.4, 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. Ganser et al. [15] recently developed a polyhedral search space exploration, named Polyite, that is also based on the generator representation of the search space; see Subsection 4.4.6. 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.

6 CONCLUSIONS AND FUTURE WORK

We propose an efficient multi-dimensional polyhedral search space exploration of the unbounded set of all legal affine transformations. A subset of the search space is selected using its dual representation computed by the Chernikova algorithm. This allows controlling 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 more likely among them.

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 an experimental evaluation of twelve different codes shows promising results. While one would require to try different frameworks, 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 73% 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 for all other tools and algorithms evaluated. 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.

There is potential for a further tuning of vectorizing RBGS stencils. At present, the schedules that pass the highest filter level perform quite well, but they are still not optimal. With an enhancement of our techniques for vectorization, exploration experiments on other architectures, including GPUs, will be a useful and interesting enterprise. Additionally, 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.

ACKNOWLEDGMENTS

Thanks to our colleague Armin Größlinger for providing a Chernikova implementation and his assistance throughout the course of this research. We are impressed by and sincerely grateful for the level of competence and involvement of the reviewers. This work is supported by the German Research Foundation (DFG), as part of Priority Programme 1648 "Software for Exascale Computing" in project ExaStencils under contract LE 912/15.

REFERENCES

[1] A. Acharya and U. Bondhugula. 2015. PLUTO+: Near-Complete Modeling of Affine Transformations for Parallelism and Locality. In *PPoPP 2015*. ACM, 54–64.

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. 2007. *Compilers – Principles, Techniques and Tools* (2nd ed.). Addison-Wesley. Section 6.4.3.

[3] V. Bandishti, I. Pananilath, and U. Bondhugula. 2012. Tiling Stencil Computations to Maximize Parallelism. In *SC 2012*. IEEE Computer Society, Article 40, 11 pages.

[4] C. Bastoul. 2014. Clan — A Polyhedral Representation Extractor for High Level Programs. Available at the Clan Web site.

[5] U. Bondhugula, V. Bandishti, and I. Pananilath. 2017. Diamond Tiling: Tiling Techniques to Maximize Parallelism for Stencil Computations. *IEEE TPDS* 28, 5 (May 2017), 1285–1298.

[6] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *PLDI 2008*. ACM, 101–113.

[7] M. Christen, O. Schenk, and H. Burkhart. 2011. PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures. In *IPDPS 2011*. IEEE, 676–687.

[8] K. Datta. 2009. *Auto-Tuning Stencil Codes for Cache-Based Multicore Platforms*. Ph.D. Dissertation. EECS Department, University of California, Berkeley.

[9] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick. 2009. Optimization and Performance Modeling of Stencil Computations on Modern Microprocessors. *SIAM Rev.* 51, 1 (Feb. 2009), 129–159.

[10] H. Dursun, K. Nomura, W. Wang, M. Kunaseth, L. Peng, R. Seymour, R. K. Kalia, A. Nakano, and P. Vashishta. 2009. In-Core Optimization of High-Order Stencil Computations. In *PDPTA 2009*. CSREA Press, 533–538.

[11] P. Feautrier. 1992. Some Efficient Solutions to the Affine Scheduling Problem, Part I: One-dimensional Time. *IJPP* 21, 5 (1992), 313–347.

[12] P. Feautrier. 1992. Some Efficient Solutions to the Affine Scheduling Problem, Part II: Multidimensional Time. *IJPP* 21, 6 (1992), 389–420.

[13] P. Feautrier and C. Lengauer. 2011. Polyhedron Model. In *Encyclopedia of Parallel Computing*, D. Padua et al. (Eds.). Springer, 1581–1592.

[14] M. Frigo and V. Strumpen. 2005. Cache Oblivious Stencil Computations. In *ICS 2005*. ACM, 361–366.

[15] S. Ganser, A. Grösslinger, N. Siegmund, S. Apel, and C. Lengauer. 2017. Iterative Schedule Optimization for Parallelization in the Polyhedron Model. *ACM TACO* 14, 3, Article 23 (Aug. 2017), 26 pages.

[16] P. Ghysels and W. Vanroose. 2015. Modeling the Performance of Geometric Multigrid Stencils on Multicore Computer Architectures. *SISC* 37, 2 (2015), C194–C216.

[17] M. Griebl, P. Feautrier, and A. Größlinger. 2005. Forward Communication Only Placements and Their Use for Parallel Program Construction. In *LCPC 2002 (LNCS 2481)*, W. Pugh and C.-W. Tseng (Eds.). Springer, 16–30.

[18] T. Grosser, A. Cohen, J. Holewinski, P. Sadayappan, and S. Verdoolaege. 2014. Hybrid Hexagonal / Classical Tiling for GPUs. In *CGO 2014*. ACM, Article 66.

[19] T. Gysi, C. Osuna, O. Fuhrer, M. Bianco, and T. C. Schulthess. 2015. STELLA: A Domain-specific Tool for Structured Grid Methods in Weather and Climate Models. In *SC 2015*. ACM, Article 41, 12 pages.

[20] T. Henretty, K. Stock, L.-N. Pouchet, F. Franchetti, J. Ramanujam, and P. Sadayappan. 2011. Data Layout Transformation for Stencil Computations on Short-Vector SIMD Architectures. In *CC 2011 (LNCS 6601)*, J. Knoop (Ed.). Springer, 225–245.
[21] T. Henretty, R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan. 2013. A Stencil Compiler for Short-Vector SIMD Architectures. In *ICS 2013*. ACM, 13–24.
[22] F. Irigoin and R. Triolet. 1988. Supernode Partitioning. In *POPL 1988*. ACM, 319–329.
[23] W. A. Kelly. 1996. *Optimization Within a Unified Transformation Framework*. Ph.D. Dissertation. Department of Computer Science, University of Maryland at College Park.
[24] M. Kong, R. Veras, K. Stock, F. Franchetti, L.-N. Pouchet, and P. Sadayappan. 2013. When Polyhedral Transformations Meet SIMD Code Generation. In *PLDI 2013*. ACM, 127–138.
[25] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. 2007. Effective Automatic Parallelization of Stencil Computations. In *PLDI 2007*. ACM, 235–244.
[26] S. Kronawitter and C. Lengauer. 2015. *Optimizations Applied by the ExaStencils Code Generator*. Technical Report MIP-1502. Faculty of Computer Science and Mathematics, University of Passau. 10 pages.
[27] C. Lengauer, S. Apel, M. Bolten, A. Größlinger, F. Hannig, H. Köstler, U. Rüde, J. Teich, A. Grebhahn, S. Kronawitter, S. Kuckuk, H. Rittich, and C. Schmitt. 2014. ExaStencils: Advanced Stencil-Code Engineering. In *Euro-Par 2014: Parallel Processing Workshops (LNCS 8806)*, L. Lopes et al. (Eds.). Springer, 553–564.
[28] W. Li and K. Pingali. 1994. A Singular Loop Transformation Framework Based on Non-Singular Matrices. *IJPP* 22, 2 (April 1994), 183–205.
[29] S. Long and M. O'Boyle. 2004. Adaptive Java Optimisation Using Instance-based Learning. In *ICS 2004*. ACM, 237–246.
[30] R. T. Mullapudi, V. Vasista, and U. Bondhugula. 2015. PolyMage: Automatic Optimization for Image Processing Pipelines. *SIGARCH Computer Architecture News* 43, 1 (March 2015), 429–443.
[31] A. D. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey. 2010. 3.5-D Blocking Optimization for Stencil Computations on Modern CPUs and GPUs. In *SC 2010*. IEEE. 13 pages.
[32] A. Nisbet. 1998. GAPS: A Compiler Framework for Genetic Algorithm (GA) Optimised Parallelisation. In *High-Performance Computing and Networking: International Conference and Exhibition (LNCS 1401)*, P. Sloot, M. Bubak, and B. Hertzberger (Eds.). Springer, 987–989.
[33] L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos. 2008. Iterative Optimization in the Polyhedral Model: Part II, Multidimensional Time. In *PLDI 2008*. ACM, 90–100.
[34] L.-N. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache. 2007. Iterative Optimization in the Polyhedral Model: Part I, One-Dimensional Time. In *CGO 2007*. IEEE Computer Society, 144–156.
[35] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, P. Sadayappan, and N. Vasilache. 2011. Loop Transformations: Convexity, Pruning and Optimization. In *POPL 2011*. ACM, 549–562.
[36] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand. 2012. Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines. *ACM TOG* 31, 4, Article 32 (July 2012), 12 pages.
[37] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *PLDI 2013*. ACM, 519–530.
[38] P. S. Rawat, M. Kong, T. Henretty, J. Holewinski, K. Stock, L.-N. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan. 2015. SDSLc: a multi-target domain-specific compiler for stencil computations. In *WOLFHPC 2015*. ACM, 6:1–6:10.
[39] C. Schmitt, S. Kuckuk, F. Hannig, H. Köstler, and J. Teich. 2014. ExaSlang: A Domain-Specific Language for Highly Scalable Multigrid Solvers. In *WOLFHPC 2014*. ACM, 42–51.
[40] A. Schrijver. 1986. *Theory of Linear and Integer Programming*. Wiley.
[41] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. 2011. The Pochoir Stencil Compiler. In *SPAA 2011*. ACM, 117–128.
[42] J. Treibig, G. Wellein, and G. Hager. 2011. Efficient Multicore-Aware Parallelization Strategies for Iterative Stencil Computations. *J. Computational Science* 2, 2 (May 2011), 130–137.
[43] S. Verdoolaege. 2010. *isl*: An Integer Set Library for the Polyhedral Model. In *ICMS 2010 (LNCS 6327)*, K. Fukuda, J. van der Hoeven, M. Joswig, and M. Takayama (Eds.). Springer, 299–302.
[44] S. Verdoolaege and T. Grosser. 2012. Polyhedral Extraction Tool. In *IMPACT 2012*. http://impact.gforge.inria.fr/impact2012/.
[45] H. Le Verge. 1994. *A Note on Chernikova's Algorithm*. Technical Report RR-1662. INRIA.