Code generation for in-place stencils

Mohamed Essadki, Bertrand Michel, Bruno Maugars, Oleksandr Zinenko, Nicolas Vasilache, Albert Cohen

Code Generation for In-Place Stencils

Mohamed Essadki
ONERA
Chatillon, France
mohamed.essadki@onera.fr

Bertrand Michel
ONERA
Chatillon, France
bertrand.michel@onera.fr

Bruno Maugars
ONERA
Chatillon, France
bruno.maugars@onera.fr

Oleksandr Zinenko
Google
Paris, France
zinenko@google.com

Nicolas Vasilache
Google
Zürich, Switzerland
ntv@google.com

Albert Cohen
Google
Paris, France
albertcohen@google.com

Abstract

Numerical simulation often resorts to iterative in-place stencils such as the Gauss-Seidel or Successive Overrelaxation (SOR) methods. Writing high performance implementations of such stencils requires significant effort and time; it also involves non-local transformations beyond the stencil kernel itself. While automated code generation is a mature technology for image processing stencils, convolutions and out-of-place iterative stencils (such as the Jacobi method), the optimization of in-place stencils requires manual craftsmanship. Building on recent advances in tensor compiler construction, we propose the first domain-specific code generator for iterative in-place stencils. Starting from a generic tensor compiler implemented in the MLIR framework, tensor abstractions are incrementally refined and lowered down to parallel, tiled, fused and vectorized code. We used our generator to implement a realistic, implicit solver for structured meshes, and demonstrate results competitive with an industrial computational fluid dynamics framework. We also compare with stand-alone stencil kernels for dense tensors.

CCS Concepts: • **Software and its engineering** → **Compilers**; • **Theory of computation** → **Parallel computing models**; • **Applied computing** → *Physical sciences and engineering*.

Keywords: Computational Fluid Dynamics, Implicit Methods, Gauss-Seidel, SOR, Iterative In-place Stencils, Domain-Specific Code Generation, MLIR, Vectorization, Tiling.

ACM Reference Format:
Mohamed Essadki, Bertrand Michel, Bruno Maugars, Oleksandr Zinenko, Nicolas Vasilache, and Albert Cohen. 2023. Code Generation for In-Place Stencils. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization (CGO '23), February 25 – March 1, 2023, Montréal, QC, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3579990.3580006

1 Introduction

We are interested in the parallelization and optimization of Computational Fluid Dynamics (CFD) applications, and more specifically implicit finite-volume numerical methods to solve differential equations. This consists in discretizing the space domain into small cells representing the conservative fields of the simulation (mass density, momentum, energy, etc., where the volume value of each field is averaged over a given cell). Then, at every step of the simulation, a solver based on the implicit method (for faster convergence and scalability) proceeds by rewriting the differential equations in the form of a large and sparse linear system

$$A \cdot x = b \tag{1}$$

where A is a square matrix of size $m \times m$, x and b are two column vectors of the same size m, and x contains the numerical solution of the physical fields. An implicit CFD solver can typically be split in two main phases:

1. first compute the vector b, iterating over the faces of the cells to compute a numerical flux [34, 36] which can be considered as a function of the two solutions in adjacent cells separated by a common face;
2. then, rather than explicitly updating the fields in the cell, solve the linear system using an iterative method like Successive Overrelaxation (SOR), a variant of the Gauss-Seidel method [12, 42].

It remains an open problem to design and implement a domain-specific code generator for implicit finite-volume solvers using state-of-the-art methods like SOR. Unlike the Jacobi iterative method and all stencil codes occurring in image processing and neural networks, *SOR is an in-place stencil computation, carrying an internal data dependence over the space domain.* Because of these internal dependences, *parallelization and vectorization of in-place stencils require a wavefront schedule.* This incurs additional control flow and indexing overheads and higher complexity in modeling locality-enhancing transformations such as tiling (cache blocking) and fusion. It is important to optimize such in-place stencils, since in typical scenarios Gauss-Seidel and

SOR converge quadratically faster than the methods free of internal dependences such as Jacobi [19]. Unfortunately, none of the popular domain-specific frameworks for stencil computations and convolutions—such as Halide [31], TVM [13], polyhedral tools like Polymage [29] or Tiramisu [4], or simulation-oriented frameworks like ExaStencils [26] or Devito [27]—can model in-place stencils; until now, optimized in-place stencils relied on manual craftsmanship [37].

In the following, we consider structured meshes only, where the solution vector x can be represented by a multi-dimensional array or tensor. We propose a domain-specific code generator for in-place stencils, implementing state-of-the-art parallelization, vectorization and optimization techniques for the iterative method itself, and also enabling non-local optimization over the end-to-end CFD solver. As an additional contribution, we derive our implementation from a generic tensor compiler based on the MLIR framework [25, 38], demonstrating a high level of reuse of the existing abstractions and incremental refinement logic of the compiler, and leveraging MLIR's natural extensibility. We validate our approach by implementing a state-of-the-art implicit solver demonstrating competitive results with a manually optimized industrial CFD framework.

2 Parallelizing Iterative In-Place Stencils

We represent physical multi-fields (density, momentum, energy, etc.) with a tensor of rank $k+1$, where the leading dimensions enumerate the n_v fields[1] and the other dimensions represent discretized physical space. The in-place stencil computation of an iterative solver updates a tensor Y iteratively through the following equation, where every element of the solution tensor Y is a linear or a nonlinear combination of neighboring tensor elements at current iteration Y and previous iteration X:

$$\sum_{w=1}^{n_v} D[v,w,\vec{\imath}] \times Y[w,\vec{\imath}] = B[v,\vec{\imath}] + \sum_{\vec{r}\in L} g(Y[v,\vec{\imath}+\vec{r}], X[v,\vec{\imath}]) + \sum_{\vec{r}\in U} g(X[v,\vec{\imath}+\vec{r}], X[v,\vec{\imath}]) \tag{2}$$

In (2), X and Y are two tensors of rank $k+1$ and shape $(n_v, n_1, n_2, \ldots, n_k) \in \mathbb{N}^{k+1}$, Y represents the solution at the current iteration and X represents the solution at the previous iteration; $\vec{\imath} \in \prod_{d=1}^{k}[0, n_d - 1]$ are the space coordinates; B is a tensor with the same shape as X and Y, representing the flow across finite volume cells; D is a tensor of rank $k+2$, capturing interactions among physical fields.[2] L and U model a fixed stencil pattern, where L is a subset of $\{\{r\} \setminus r_d \in [-s_d, s_d]$ and $\vec{r} \prec \vec{0}\}$[3] modeling intra-iteration dependences, and U is a subset of $\prod_{d=1}^{k}[-s_d, s_d] \setminus L$ modeling dependences on the previous iteration. $g : \mathbb{R}^{n_v} \times \mathbb{R}^{n_v} \to \mathbb{R}^{n_v}$ is a given function which takes as argument a neighboring element $Y[v,\vec{\imath}+\vec{r}] \in \mathbb{R}^{n_v}$ or $X[v,\vec{\imath}+\vec{r}] \in \mathbb{R}^{n_v}$ and the central element $X[v,\vec{\imath}] \in \mathbb{R}^{n_v}$. The computation applies pointwise to all variables $v \in [1, n_v]$ and to all coordinate vectors $\vec{\imath}$ where the stencil does not overflow the tensor boundaries, that is $\forall l \in [1,k], s_l < i_l < n_l - s_l$. A degenerate variant of (2) applies to boundary coordinate vectors.

Practically, the tensor D can be inverted, then integrated into tensor B and function g. This transformation enables implementing the iterative computation of Y as an explicit update. We prefer the more algebraic form of (2) as it is more representative of the diversity of numerical applications considered in our domain-specific framework.

Finally, the lexicographic ordering restriction $\vec{r} \prec \vec{0}$ on neighboring tensor elements in L guarantees that the lexicographic traversal of coordinate vectors is a valid schedule. This enables systematic code generation into imperative loop nests over arrays. In the following, we consider additional transformations to improve performance.

2.1 Tiling

Loop tiling (or cache blocking) is an effective optimization that exploits temporal locality in caches [1, 41]. The sufficient validity condition for tiling is intra-iteration dependence distances being non-negative along all coordinates [21, 39], which is *not* guaranteed by the lexicographic restriction alone. For example, $(r_0, r_1) = (-1, 1) \prec (0, 0)$, but it would make the value of Y depend on "future" values when permuting the traversal order of coordinate dimensions as the result of tiling. This limits the tiling opportunities and poses the first difficulty that only few traditional (out-of-place) stencil code generators like Pluto handle it, by favoring parallelipedic tiles aligned with the (skewed) parallel wavefronts.

We address this difficulty by implementing a restriction on tile sizes, depending on the stencil pattern defined by L: for any negative dependence distance, we force the tile size along the associated dimension to be 1; this effectively prevents permuting along this particular dimension.

Like most tiling frameworks, we rely on autotuning for selecting tile sizes. We consider a range of sizes whose product, multiplied by n_v and the number of live tensors (3 in the computation above) is bounded by the size of the largest cache level private to a given core (L2 on mainstream CPUs).

2.2 Fusion

The next step in our optimization strategy is loop fusion. After choosing tile sizes, the two-step stencil computations are fused along the variable (physical fields) dimension and space coordinates to exploit temporal locality on X and Y.

Then comes the problem of accumulating the B tensor: since B is often iteration-dependent, it has to be computed and one needs to select an appropriate temporary storage and

[1] v stands for "variables", and n_v is small, e.g. $n_v = 5$ in our use case.
[2] B and D are typically dependent on X but are independent of Y.
[3] $\vec{r} \prec \vec{0}$) in lexicographical order

fusion strategy. The data reuse pattern for B is similar to out-of-place stencil computations in image processing, hinting at recomputing at the tile level as the most profitable strategy [31]. This is what we observed in practice, and we chose to compute B (and D) on the fly, storing one tensor block in a temporary buffer, tuned to remain L2-cache-local. This allows us to fuse one complete iteration of (2) at the tile level, with redundant computations of B (and D) occurring across tile boundaries due to the dependences from neighboring elements in the stencil pattern.

Such a domain-specific, combined tiling and fusion strategy is well known to CFD experts.

2.3 Sub-domain Parallelism

We are now interested in exposing thread-level parallelism. It-erative in-place stencils involve intra-iteration dependences preventing parallelization along the axes of the coordinate space. Wavefronts of independent computations can be ob-tained by partitioning the computation along some linear combination of coordinate vectors. For the Gauss-Seidel ex-ample, partitioning along $i + j$ produces a parallel wavefront. The loop transformation producing wavefronts is known as *loop skewing* [40].

Since we are dealing with a tiled coordinate space for locality, thread-level parallelization operates at or above the granularity of the cache-oriented tiles. In CFD solvers, this level is referred to as *sub-domain parallelism*. It consists in partitioning the coordinate space into (hyper-)rectangular sub-domains through another level of tiling. Based on the sub-domain sizes and on the stencil pattern, we may construct a partial execution order of sub-domains that satisfies intra-iteration dependences; this partial execution order is known as a *schedule*. A good schedule should allow independent sub-domains to be executed in parallel. Dependences across sub-domains can be derived from the L subset of the stencil pattern. Since sub-domains are rectangular, one may consider their corners only. For each corner $\vec{c}$ and $\vec{r} \in U$, we determine to which sub-domain the element $\vec{c} + \vec{r}$ belongs to; this is illustrated in Figure 1. This results in a uniform dependence graph at sub-domain-level granularity.

The next step consists in computing a schedule mapping sub-domains to natural integers. The optimal latency schedule maps a sub-domain s to the longest path leading to s from the initial sub-domain (holding space coordinates $\vec{0}$) or any sub-domain depending only on boundary conditions. By considering the lexicographic ordering restriction $\vec{r} \prec \vec{0}, \forall \vec{r} \in L$. The schedule can be expressed as:

$$\theta(\vec{\imath}_S) = \max_{\vec{r} \in L}(\theta(\vec{\imath}_S + \vec{r})) + 1 \tag{3}$$

executed in the lexicographical order of $\vec{\imath}_S$, where the index vector $\vec{\imath}_S$ is the coordinate of a sub-domain.

In general, directly computing this schedule may result in overhead, given the complexity of $O(n_{sd} \times |L|)$, where

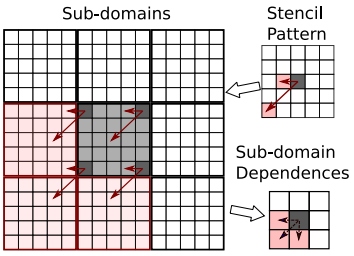

Figure 1. Derivation of sub-domain dependences from a stencil pattern: the red continuous arrows represent stencil pattern dependences and the black dashed-line arrows represent sub-domain-level dependences.

n_{sd} is the number of sub-domains. However, in practice the number of subdomains is limited ($\ll 100^3$) and the number of vectors in L is small (< 100), e.g., a $5 \times 5 \times 5$ stencil has up to $|L| \leq (5^3 - 1)/2 = 62$ vectors. Schedule computation is also performed once and reused across iterations, making its overhead acceptable for our applications.

2.4 Vectorization

Classical loop and subword-level parallelism (SLP) vectorizers typically fail to vectorize loops with data-dependences [23]. In general, they fail due to pattern matching limitations or to the lack of node splitting or variable renaming transformations [11] capable of converting problematic dependences into loop-independent ones, and to eliminate memory-based dependences. They also tend to lack the performance models to prioritize vectorization along outer loops with friendlier data access strides [14]. But more specifically, they lack the domain-specific knowledge that partial vectorization is profitable on a subset of the in-place, internally dependent stencil pattern; this is the approach we describe below.

We consider two strategies to vectorize in-place stencils. The first one vectorizes along the parallel wavefronts identified in Section 2.3. In this case, accessed elements are not contiguous in memory, resulting in scatter and gather instructions under-utilizing memory bandwidth due to access strides. The second one involves partial vectorization along the axis of contiguous memory access. Due to the dependences in the L part of the stencil pattern in (2), we can vectorize the summation with B (and its computation) as well as the summation over the U pattern, but we may only vectorize a subset of the iterations in the L pattern: $g(Y[v, \vec{i}+\vec{r}], X[v, \vec{i}])$ for $\vec{r} \neq \vec{0}$ or $i_k < -VF$ where VF is the vectorization factor. In Figure 2, we illustrate this strategy on a 5-points 2D Gauss-Seidel. We adopt the second strategy to partially vectorize iterative in-place stencils.

3 Domain-Specific Code Generation

MLIR [25] is a recent compiler infrastructure that can be seen as a conceptual successor of the vastly popular LLVM IR [24].

```
for i in range(1,Nx-1):
  for j in range(1,VF*((Ny-1)//VF),VF):
      # vectorized part
      temp = b[i,j:j+VF] + x[i,j:j+VF] + x[i+1,j:j+VF]
      temp += x[i,j+1:j+VF+1] + y[i-1,j:j+VF]
      # unrolled sequential part
      y[i,j] = d[i,j]*(temp[0] + y[i,j-1])
      y[i,j+1] = d[i,j+1]*(temp[1] + y[i,j])
          ...
      y[i,j+VF-1] = d[i,j+VF-1]*(temp[VF-1] + y[i,j+VF-2])
  #peeled loop
  for j in range(VF*((Ny-1)//VF),Ny-1):
      y[i,j] = b[i,j]+d[i,j]*(x[i,j+1]+x[i+1,j]+y[i-1,j]+y[i,j-1])
```

Figure 2. Partial vectorization of 5-points 2D Gauss-Seidel.

It offers unprecedented flexibility to define customized compiler internal representations (IR) as well as to freely mix them with other existing representations within the same module. This flexibility is achieved by adopting a few design principles partially adapted from object-oriented programming, such as separation of interfaces and implementations.

3.1 Code Generation Infrastructure

MLIR consists of multiple *dialects*, which are IR equivalents of modular dynamic libraries. Practically, dialects gather specific instantiations of core IR components: attributes, operations, and types. An *operation* is an execution primitive that has associated (operational) semantics. Operations produce typed *values* that can be used as operands by other operations while enforcing static single assignment (SSA). *Types* contain the properties of a value known at compile time, such as the value being an integer or its bit size. *Attributes* contain similar compile-time properties of an operation. The IR is structured in (basic) blocks that reside in *regions* associated with certain operations. MLIR blocks use the functional form of the SSA where PHI nodes are replaced with block *arguments* [3]. For example, a function in MLIR is merely an operation with the "name" and "signature" attributes, and a region representing its body.

The sets of attributes, operations and types are fully open and user-extensible. While the MLIR distribution includes a set of generic dialects, they have no additional capabilities compared to user-defined dialects that may be shipped as separate libraries. Reuse and adaptation of existing dialects is encouraged, but not mandatory.

This work builds on MLIR dialects designed for code generation [6, 38] in addition to designing its own dialect for in-place stencils, described in Section 3.2. It generalizes the `linalg` dialect, which defines tensor operations with a structured *static* mapping between the iteration space of the operation and the data spaces of its operands and results, by making this mapping *dynamic*. The `linalg` dialect itself supports kinds of tensor operations ranging from elementwise unary to tensor contractions, to convolutions. It operates on data that can be represented and accessed using the operations from the `tensor` dialect, which treats multidimensional arrays as immutable SSA values, or from the **`memref`** dialect, which models multidimensional arrays as in-memory buffers with a specific layout such as column- or row-major. Concrete computational payload is decoupled from the computational structure and is represented using the `arith` and `math` dialects, holding operations for simple arithmetic and for `libm`-style math functions, respectively.

The lowering process converges to the `llvm` dialect that directly maps to LLVM IR and can be translated into object files for any of the hardware platforms supported by LLVM.

3.2 CFD Dialect

We introduce a new `cfd` dialect with two operations:

- `faceIteratorOp` to perform computation over the faces;
- `stencilOp` to model an iterative in-place stencil.

`faceIteratorOp` computes the numerical flux with the finite volume method, then distributes the computed flux across adjacent cells sharing a common face. This design aims to avoid redundant computations. We use this operation to compute the tensor B in (2). Note that the existing **`linalg.`**`generic` operation can perform such computations using shifted slices of the input and output tensors; yet this solution yields redundant computations (effectively duplicated on both cells sharing a common face).

`stencilOp` implements one iteration of the in-place stencil. It takes two input operands: X, the tensor representing the solution at the previous iteration, and B the tensor resulting from `faceIteratorOp`. It produces one result: Y, the tensor that represents the solution at the current iteration Y. The subsets L and U of the stencil pattern are modeled together as one attribute named `stencil`, denoted $\tilde{s}$ below. It takes -1, 0 or 1 as element values (see Figure 4), such that:

- $\tilde{s}[\vec{r}] = -1$ means that $\vec{r} \in L$, in other words, the computation of $Y[v, \vec{i}]$ depends on $Y[v, \vec{i} + \vec{r}]$,
- $\tilde{s}[\vec{r}] = 1$ means that $\vec{r} \in U$,
- and $\tilde{s}[\vec{r}] = 0$ means that $\vec{r} \notin L \cup U$.

The stencil attribute is mainly used for parallel scheduling and vectorization. The `nbVar` attribute represents the number of multi-fields n_v. Finally, a region holding the details of the computation of $D[v, \vec{i}]$, $g(Y[v, \vec{i} + \vec{r}], X[v, \vec{i}])$ for $\vec{r} \in L$ and $g(X[v, \vec{i} + \vec{r}], X[v, \vec{i}])$ for $\vec{r} \in U$. The only block of this region has as many arguments as there are non-zero elements in the `stencil` attribute. These arguments correspond to values of X or Y (for `stencil` value -1 or 1, respectively) accessed by the stencil. The example in Figure 3 implements a simple 5-points Gauss-Seidel kernel, with $k = 2$, $n_v = 1$ and where $g(a, b) = a$ for $\forall (a, b) \in \mathbb{R}^2$.

The canonical lowering of `stencilOp` introduces nested "for" loops with a k-dimensional rectangular iteration domain derived from the input/output tensor shapes. In the body of the innermost loop, the computations in the region of `stencilOp` are inlined and the final computations to update $Y[v, \vec{i}]$ are performed as shown in Figure 5.

```
%result = cfd.stencilOp
  ins(%X: tensor<1x?x?xf64>, %B : tensor<1x?x?xf64>)
  outs(%Y : tensor<1x?x?xf64>)
  {nbVar = 1, stencil= [[0,-1,0],[-1,0,1],[0,1,0]]} {
^bb0(%wd: f64, %wl: f64, %w0: f64, %wr: f64, %wu: f64):
  /*...*/ compute %wd', %wl', %w0', %wr', %wu' /*...*/
  cfd.yield %cst, %wd', %wl', %w0', %wr', %wu'
  64, f64, f64, f64, f64, f64
}
```

Figure 3. The new MLIR `cfd.stencilOp` operation captures the static stencil pattern as an attribute, and the computation of D and g as a region.

```
for i in range(Nx)
  for j in range(Ny):
    W[i, j] = (W[i - 1, j] + W[i, j - 1] + W[i, j]
             + W[i + 1, j] + W[i, j + 1]) / 5.0
```

0	1	0
-1	1	1
0	-1	0

```
for i in range(Nx)
  for j in range(Ny):
    W[i, j] = (W[i-1, j-1] + W[i-1, j] + W[i-1, j+1]
             +W[i,   j-1] + W[i,   j] + W[i,   j+1]
             +W[i+1, j-1] + W[i+1, j] + W[i+1, j+1])/9.0
```

1	1	1
-1	1	1
-1	-1	-1

Figure 4. Stencil pattern attribute captures access directions. (Left) 5-point, (right) 9-point Gauss-Seidel in-place stencil.

```
%FULL_Y = scf.for %i = ... iter_args(%arg6 = ...) {
  %PART_Y = scf.for %j = ... iter_args(%Y = ...) {
     // we map the body arguments of stencilOp to the
     // elements in X and Y tensors
     %14 = tensor.extract %Y[0, %i - 1, %j]
     %16 = tensor.extract %Y[0, %i,     %j - 1]
     %17 = tensor.extract %X[0, %i, %j]
     %19 = tensor.extract %X[0, %i,     %j + 1]
     %21 = tensor.extract %X[0, %i + 1, %j]
     %22 = tensor.extract %B[0, %i, %j]
     // inline the computation in stencilOp body
     %33 = func.call @g(%14, %16, %17, %19, %21, %22)
     %34 = tensor.insert %33 into %Y[0, %i0, %i1]
     scf.yield %34 : tensor<1x?x?xf64>
   }
   scf.yield %PART_Y : tensor<1x?x?xf64>
}
```

Figure 5. Simplified loop-level intermediate representation lowered from `cfd.stencilOp`; `iter_args` operands are loop-carried values yielded by each iteration.

3.3 Tiling and Fusion in MLIR

An important contribution of our work is the integration of domain-specific compilation techniques into a state-of-the-art tensor compiler. We extend the *structured operation* concept of the `linalg` dialect, where a computational operation "knows" how to decompose into instances of itself operating on subsets of inputs and producing a subset of result [38]. At the time of writing, structured operations in `linalg` do not include self-dependences (aside from reductions). We design `cfg.stencilOp` to be similarly decomposable, enabling seamless integration with existing operations and transformations thereon. Specifically, we can compute the subsets of X, B and D required to compute the updated value of Y using the stencil pattern as illustrated in Figure 1.

Tiling. Similarly to other structured operations, tiling can be implemented by generating the k-deep loop nest

```
%9 = cfd.tiled_loop (%i, %j) = (0, 0) to (%max_i, %max_j)
    ins (%arg7 = %4: tensor) outs (%arg8 = %8: tensor) {
  %11 = arith.min (%tile_size_1, %max_i - %i)
  %12 = arith.min (%tile_size_2, %max_i - %i)
  // extract the slices for the tiling
  %13 = tensor.extract_slice %arg7[0,%i,%j] [1,%11,%12] [1,1,1]
  %14 = tensor.extract_slice %arg8[0,%i,%j] [1,%11,%12] [1,1,1]
  %15 = cfd.stencilOp ins(%14, %13)  outs(%14)
      {stencil = [[0,-1,0], [-1,1,1], [0,1,0]], nbVar = 1} {
  ^bb0(%wd: f64, %wl: f64, %w0: f64, %wr: f64, %wu: f64):
    /*...*/ compute %wd', %wl', %w0', %wr', %wu' /*...*/
    cfd.yield %cst, %wd', %wl', %w0', %wr', %wu'
  }
  %r = tensor.insert_slice %15 into %arg8[0,%i,%j][1,%11,%12][1,1,1]
  cfd.yield %r
}
```

Figure 6. Simplified tiled IR featuring smaller-sized instances of `cfd.stencilOp`; `extract_slice` and `insert_slice` operate on rectangular data tiles selected through per-dimension offset, size and stride tuples.

with each iteration producing a subset (tile) of the resulting Y, and then by using the operation self-decomposition to generate the body of the loop. We build on the 'tiled multi-for" construct, originally proposed[4] then retired from MLIR in favor of other composition mechanisms, to introduce `cfd.tiled_loop`. This construct represents a loop nest with explicit input and output tensors, and shoehorns the loop nest into the structured operation abstraction to enable repeated tiling. It can be lowered to classical (parallel) "for" loops after the MLIR bufferization pass that replaces immutable tensors with mutable buffers. Furthermore, we extend this construct with additional optional operands that specify the groups of iterations can be executed in parallel, which we leverage in Section 3.4. Figure 6 demonstrates the result of tiling that leverages `tensor.extract/insert_slice` operations for hyperrectangular subsetting.

Fusion after Tiling. The `cfd` dialect generalizes the optimization pattern known as "fusion after tiling" [38]. A `stencilOp` can be fused with any "structured operation" such as `cfd.faceIteratorOp` or `linalg.generic`, the main computational operations occurring in a CFD application. Fusion proceeds by constructing a DAG of dependences *between* operations, which it traverses from the leaves up. When an operand of a (tiled) structured operation is produced by another structured operation (including `cfd.tiled_loop`), the producer operation can be pulled into the loop nest surrounding the consumer operation with further adaptation to only produce the subset of result required by the consumer. This producer/consumer fusion [31] allows for rematerialization and overlapping input data tiles unlike loop fusion.

3.4 Sub-domain Parallelism

introduce We may now present our strategy to lower a (tiled and fused) DAG of structured operations into parallel computations.

[4] https://discourse.llvm.org/t/rfc-add-linalg-tileop/2833

A naïve lowering of `tiled_loop` into a nest of "parallel for" loops would not respect the intra-iteration dependences of stencils. To handle those, we introduce a utility operation:

```
%rows, %cols = cfd.get_parallel_blocks[%M, %N]
  {block_stencil = [[-1,-1,0],[-1,0,0],[ 0,0,0]]}
```

This operation accepts as operands the number of subdomains (outer `tiled_loop` iterations) along each tiled dimension. Its `block_stencil` attribute captures the subdomain dependence pattern, derived from the stencil pattern in Figure 4, similarly to `stencil` but only using 0 and -1 as values. It computes the subdomains that can be executed in parallel wavefronts and produces two values that encode sets of independent subdomains as a matrix of linearized indices in the compressed sparse row (CSR) format, with each row containing indices of one independent subset and being scheduled after all previous rows. This format can be directly consumed by our `cfd.tiled_loop` operation.

Such a `tiled_loop` with groups of independent iterations can be lowered into a sequential "for" loop iterating over groups that contains a "parallel for" loop iterating over independent iterations within a group Iterations of the outer loop serve as synchronization points between groups. Note that we always produce *one* coalesced parallel loop to better exploit the single degree of parallelism available on CPU. The CSR format allows for naturally extracting the indices of the original `tiled_loop` and the "parallel for" loop is readily convertible to OpenMP using MLIR. The `block_stencil` operation is converted into a direct implementation of (3) using MLIR's arithmetic and control flow dialects.

3.5 Vectorization

The transformations above yield a parallel loop structure containing small instances of `stencil0p`. These are amenable to vectorization, performed alongside the conversion to loops expressed in the MLIR `cfg` dialect [38], with the stride of the innermost loop set to the vectorization factor *VF*. We go over the block arguments of `stencil0p`, each of which corresponds to a value accessed by a stencil and, instead of extracting single scalars from a tensor, use MLIR's mid-level abstractions `vector.transfer_read/write` to extract *VF*-sized vectors directly. MLIR provides the lowerings from these abstractions to efficient hardware vector operations, and its arithmetic operations transparently apply to vectors elementwise, simplifying our implementation. As indicated in Section 2.4, stencil value accesses from the L subset cannot be vectorized. For these accesses, we replicate scalar operations VF times, resulting in partial loop unrolling for non-vectorizable parts only. Finally, when the last tensor dimension is not statically known to be divisble by *VF*, we peel off the trailing iterations to ensure divisibility. The result is illustrated in Figure 7.

```
%15 = scf.for %i = ... step 1 iter_args(%arg10 = ...) {
 %19 = scf.for %j = ... to %ub floordiv 8 step 8
       iter_args(%Y = %arg10) {
    %21 = arith.addi %arg9, %c-1 : index
    // map block arguments of stencil0p to vectors
    // extracted from X and Y tensors
    %22 = vector.transfer_read %Y[0,%i - 1,%j] ... : vector<8xf64>
    %23 = vector.transfer_read %X[0,%i,%j] ... : vector<8xf64>
    ....
    // inline a part of stencil0p region in a vectorized form
    %30 = vector.fma 0, %22, ... : vector<8xf64>
    %31 = vector.fma 0, %23, %30 : vector<8xf64>
    ...
    %35 = tensor.extract %Z[0, %i, %j]
    %36 = vector.extract %23[0]
    %38 = math.fma %35, %36, ... : f64
    %49 = tensor.insert %38 into %Z[0, %i, %1]
    %50 = tensor.extract %49[0, %i, %i + 1]
       ...
    // 7th unrolled scalar operation
    %75 = tensor.extract %74[0, %i, %j + 7]
    %76 = vector.extract %23[7] : vector<8xf64>
    %78 = math.fma %75, %76, ... : f64
    %80 = tensor.insert %78 into %74[0, %i, %j + 7]
    scf.yield %80
  }
  // peeled loop
  %20 = scf.for %j = %ub floordiv 8 to %ub step 1 iter_args(%Z=%19)
    %22 = tensor.extract %Y[0, %i - 1, %j]
    %24 = tensor.extract %Y[0, %i, %j - 1]
    %25 = tensor.extract %Z[0, %i, %j]
    ...
    %39 = math.fma %22, %24, %25
    %42 = tensor.insert %39 into %Y[0, %i, %j]
    scf.yield %42
  }
  scf.yield %20
}
```

Figure 7. Generated code after vectorizing `stencil0p` with a peeled scalar loop: space dimension $k = 2$, number of fields $n_v = 1$ and $VF = 8$. This generated code follow same structure as Figure 2

4 Performance Results and Analysis

Let us now evaluate the performance of our code generation strategy. All experiments are performed on a dual-socket system with two Intel Xeon 6152 CPUs @ 2.10GHz. Each CPU has two NUMA nodes, each one associated with 11 cores, for a total of 44 cores partitioned over four NUMA nodes. Each core has two AVX512 vector units, 32KB of L1D and 1MB L2 cache per core, and a common 32MB L3 cache per NUMA node. In the following, we compare with the Pluto polyhedral compiler [8], we conduct an ablation study of the different transformations, and we demonstrate the effectiveness of our code generator on a realistic CFD solver.

4.1 Comparison with a Polyhedral Compiler

We consider three in-place stencil kernels with increasingly complex stencil patterns, and one slightly more realistic (yet still static control) imperfectly nested example also featuring an in-place stencil computation:

(a) two-dimensional Gauss-Seidel 5 points stencil of order 1 (cross shape in 3×3 stencil);
(b) two-dimensional Gauss-Seidel 9 points stencil of order 1 (full 3×3 stencil);
(c) two-dimensional Gauss-Seidel 9 points stencil of order 2 (cross shape in 5×5 stencil);
(d) three-dimensional Gauss-Seidel 6 points stencil of order 1 implementing the heat equation in-place.

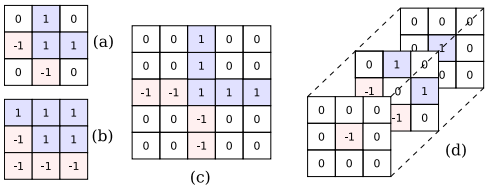

Figure 8. Stencil patterns of the four use cases.

The stencil patterns of the four use cases are given in Figure 8. The baseline implementation for each use case is written in C. The (c) use-case is the "seidel" benchmark of the PolyBench suite,[5] and the three others follow the static control coding rules of PolyBench to enable polyhedral compilation. While the first three kernels are usual suspects, Figure 9 provides the polyhedral-friendly implementation of the 3D heat equation using the Gauss-Seidel method. Every iteration starts with the computation of the laplacian of the temperature using a finite difference method, then the temperature increment is determined using the Gauss-Seidel method, and finally we update the temperature by accumulating the temperature increment.

```
for (t = 1; t <= TSTEPS; t++) {
  // Compute RHS
  for (i = 1; i < _PB_N-1; i++)
    for (j = 1; j < _PB_N-1; j++)
      for (k = 1; k < _PB_N-1; k++)
        Rhs[i][j][k] =
            (T[i+1][j][k] - 2.0 * T[i][j][k] + T[i-1][j][k])
          + (T[i][j+1][k] - 2.0 * T[i][j][k] + T[i][j-1][k])
          + (T[i][j][k+1] - 2.0 * T[i][j][k] + T[i][j][k-1]);
  // Gauss-Seidel iteration
  for (i = 1; i < _PB_N-1; i++)
    for (j = 1; j < _PB_N-1; j++)
      for (k = 1; k < _PB_N-1; k++)
        dT[i][j][k] =
            SCALAR_VAL(lambda) * (Rhs[i][j][k] +
                    (dT[i-1][j][k] + dT[i+1][j][k] + dT[i][j-1][k] +
                     dT[i][j+1][k] + dT[i][j][k-1] + dT[i][j][k+1]));
  // Update the temperature solution
  for (i = 1; i < _PB_N-1; i++)
    for (j = 1; j < _PB_N-1; j++)
      for (k = 1; k < _PB_N-1; k++)
        T[i][j][k] = T[i][j][k] + dT[i][j][k];
}
```

Figure 9. 3D heat equation implemented using Gauss-Seidel.

We compare these baseline implementations with a functionally equivalent MLIR implementation using the `cfd` dialect. Figure 10 shows a simplified MLIR implementation of the 3D heat equation introduced in Figure 9.

Pluto implements state of the art polyhedral analyses and transformations [8], including the extraction of parallel wavefronts and to tile and fuse in-place stencils. We use the options `-parallel -tile` to generate tiled OpenMP code.

Pluto analyzes the static control code region delimited by `scop` pragmas. By default, these are set to enclose the largest static control code region of the kernel, which includes the stencil iteration loop. However, we observed that this does not always yield the best performance, as it favors parallelogram tiles aligned with the (skewed) parallel wavefronts,

[5] https://sourceforge.net/projects/polybench/.

```
%9:3 = scf.for ... iter_args(%arg5, %arg6, %arg7) {
  %10 = tensor.extract_slice %arg5[...]

  // RHS computation (pointwise).
  %RHS = linalg.generic ins(...) outs(...){
  ^bb0(%arg8: f64, %arg9: f64, %arg10: f64, %arg11: f64, %arg12: f64,
       %arg13: f64, %arg14: f64, %arg15: f64):
    %27 = arith.mulf %arg8, %cst : f64
    %28 = arith.subf %arg9, %27 : f64
    %29 = arith.addf %28, %arg10 : f64
    %30 = arith.subf %arg11, %27 : f64
    %31 = arith.addf %30, %arg12 : f64
    %32 = arith.addf %29, %31 : f64
    %33 = arith.subf %arg13, %27 : f64
    %34 = arith.addf %33, %arg14 : f64
    %35 = arith.addf %32, %34 : f64
    linalg.yield %35 : f64
  }
  // Gauss-Seidel solver (stencil).
  %22 = cfd.stencilOp ins(%11, %RHS) outs(%11)
     {stencil = [[[0, 0, 0], [ 0,-1, 0], [0, 0, 0]],
                 [[0,-1, 0], [-1, 0, 1], [0, 1, 0]],
                 [[0, 0, 0], [ 0, 1, 0], [0, 0, 0]]]>} {
  ^bb0(%arg9: f64, ..., %arg14: f64):
    cfd.yield %cst, %arg9, ..., %arg14
  }
  // Temperature solution update (pointwise).
  %23 = linalg.generic ins(%22) outs(%10) {
  ^bb0(%arg8: f64, %arg9: f64):
    %27 = arith.addf %arg8, %arg9 : f64
    linalg.yield %27 : f64
  }
  %24 = tensor.insert_slice %23 into %arg5[...]
  scf.yield %24, ...
}
```

Figure 10. Pseudo-MLIR for an in-place 3D heat solver.

inducing control flow overhead. As a result, we also consider a second implementation where the `scop` pragmas enclose the (independent) spatial loops only.We refer to the default PolyBench(-like) implementation by **C+Pluto 1** and to the second implementation by **C+Pluto 2**.

The reader may consider the strategy of our MLIR code generator as a trade-off between the two extreme Pluto configurations: parallel wavefronts are derived from the complete kernel (including the iteration loop), while tiling is effective on the spatial loops only. Also, notice that fusion is only useful to the (d) use-case (the 3D heat equation), as the 3 others implement perfectly nested loops.

The size of space and the number of iterations are provided in Table 1. We have performed auto-tuning for the tile size for both MLIR and Pluto for 1, 10 and 44 threads. The chosen tile sizes are given in Table 2 and Table 3.

Figure 11 shows the speedup relative to sequential execution (without Pluto) of **C+Pluto 1**, **C+Pluto 2** and **MLIR** versions, for 1 and 10 threads. The C compiler is clang 14 with optimization flags `-O3` and `-march=native`. Every measurement corresponds to the median of 25 runs, and black bars show the confidence interval for 95%.

The MLIR-generated versions consistently outperform Pluto. This is due to automatic vectorization performing poorly on in-place stencils, aggravated by Pluto's skewed parallelogram tiles. The performance gap is not as wide in the multi-threaded case, as the kernels rapidly hit the memory bandwidth limits of a given NUMA node. The 9-points

Table 1. Gauss-Seidel kernel test case configurations.

Case	Domain size	Iterations
5-point 2D	2000×2000	500
9-point 2D	4000×4000	200
9-pt 2^{nd}-order	2000×2000	500
Heat 3D	$256 \times 256 \times 256$	50

Table 2. MLIR tile sizes configurations.

Case	Tile size 1-10 threads	Tile size 44 threads
5-point 2D	64×256	32×64
9-point 2D	1×128	1×128
9-pt 2^{nd}-order	64×256	64×128
Heat 3D	$4 \times 26 \times 256$	$4 \times 26 \times 128$

Table 3. Pluto tile sizes configurations.

Case	Tile size 1-10 threads	Tile size 44 threads
5-point 2D	16×16	16×16
9-point 2D	16×32	32×32
9-pt 2^{nd}-order	16×16	16×16
Heat 3D	$4 \times 16 \times 256$	$4 \times 16 \times 128$

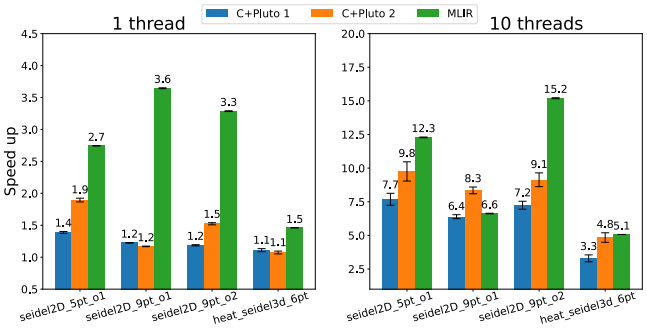

Figure 11. Optimization of four stencil kernels.

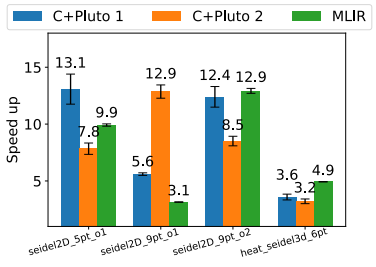

Figure 12. Autotuned speedup for 44 threads.

Gauss-Seidel use case is an exception where **C+Pluto 2** outperforms our code generator. This is due to the restriction we enforce on tile sizes in presence of a negative dependence distance; we had to choose a tile size of 1×128 as explained in Section 2.1. This time, Pluto's support for parallelogram tile shapes comes to the rescue despite hampering vectorization: it enables more parallelism and broader opportunities for tile size autotuning. Finally, for heat 3D, MLIR outperforms Pluto but only by 110% – 150%. This use case consists of three stencils, two of which are out-of-place and one is in-place. We only benefit from additionally vectorizing the in-place stencil, hence the lower overall speedup.

We have also evaluated the performance with 44 threads as shown in Figure 12. The lower performance on the 9-points Seidel is due to the constraints on subdomain tile sizes as explained previously. We also notice some NUMA effects that undo the benefits of vectorization.

For the sake of completeness, we also evaluate our method on out-of-place iterative stencils such as the Jacobi method (i.e., the L subset of the pattern is empty). Performance results are similar to Pluto: considering a 5-points Jacobi stencil, MLIR-generated code reaches about 90% of the performance of C+Pluto 1 and 110% of that of C+Pluto 2. These variations are due to parallelogram tiling: while it hampered the vectorization of in-place stencils, parallelogram tiles do not interfere with the vectorization of out-of-place stencils [7].

4.2 Ablation Study

Let us now detail the performance impact of each transformation. We focus on the MLIR implementation of the (d) use-case (3D heat equation with the Gauss-Seidel method), introduced in the the previous section, scaling its domain size to $514 \times 514 \times 514$ to improve scalability. Figure 13 shows the speedup as a function of the number of threads, considering 4 different transformations scenarios:

(Tr1) sub-domain parallelism without fusion ($6 \times 12 \times 256$): tiling ($6 \times 6 \times 128$) the 3 operations separately;
(Tr2) sub-domain parallelism with fusion: the 3 operations are tiled and fused before being parallelized;
(Tr3) sub-domain parallelism without fusion combined with vectorization;
(Tr4) sub-domain parallelism with fusion combined with vectorization.

The sequential execution of (Tr1) serves as the baseline. First of all, we see that performance scales almost linearly until 8 threads, before meeting diminishing returns. This is primarily due to NUMA effects and synchronization overheads: the total bandwidth available (to the L3 cache and memory) increases when spreading over 2 to 4 NUMA nodes, but synchronization barriers at every iteration hurt scalability when crossing NUMA node boundaries. Comparing (Tr2) and (Tr1), or (Tr4) and (Tr3), we observe that combining tiling (cache-blocking) with fusion improves scalability beyond 24 threads. On the other hand, while vectorization improves performance dramatically on lower thread counts, it is not sufficient to achieve good performance beyond 16 threads. The combination of all transformations in (Tr4) yields the best performance and continues to improve above 16 threads.

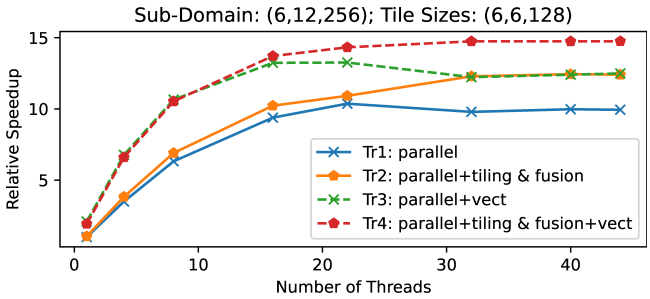

Figure 13. Impact of different optimizations applied to 3D heat Gauss-Seidel: the relative speedup in function of the number of threads using different code transformations.

4.3 Euler Equation: LU-SGS

Let us conclude this empirical evaluation with the realistic simulation of the flow of an ideal and inviscid gas. We select the 3D Euler model for this purpose, the Roe method [34] to compute the numerical flux, and the LU-Symmetric Gauss Seidel method (LU-SGS) [12] for the numerical inversion of the obtained linear system. The LU-SGS method applies a forward Gauss-Seidel sweep followed by a backward Gauss-Seidel sweep where the iteration domain is inverted. For a complete description of the numerical method, we refer to Chapter 4.2 of E. Otero's PhD thesis [30]. To model the backward Gauss-Seidel sweep with `cfd.stencil0p`, we use an optional integer attribute which takes the value 1 for the forward sweep and −1 for the backward sweep. In the case of the backward sweep, the dependences are also in the opposite direction, which means that the signs of the stencil pattern attribute must be inverted. The graph in Figure 14 summarizes the computational steps and the MLIR operations involved in the LU-SGS solver for the Euler equation.

We conduct a 3D numerical simulation using this MLIR implementation, and comparing with the elsA industrial framework. The size of the space domain is set to $512 \times 512 \times 512$, with periodic boundary conditions. For the MLIR implementation, we implement the following transformations:

- sub-domain parallelism with fusion: the sub-domain size is $8 \times 16 \times 128$;
- tiling with fusion for cache-blocking: the tile size is set to $4 \times 4 \times 128$;
- vectorization with $VF = 8$.

The elsA framework implements very similar optimization recipes, but was manually implemented in Fortran and C (sub-domain parallelism, fusion, L3 cache-blocking and vectorization). The simulation runs over 50 time iterations with the same initial conditions in both MLIR and elsA frameworks. We evaluate the average time needed to perform the computation of one cell per iteration and per thread:

$$t_{cell} = nb\_threads \times \frac{elapsed\_time}{nb\_iteration \times nb\_cells}$$

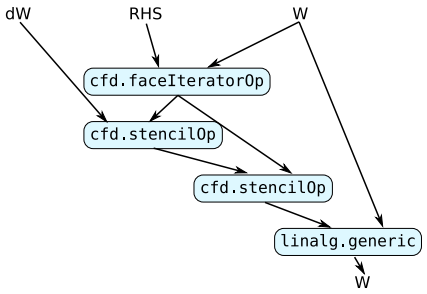

Figure 14. Computational graph of LU-SGS for the Euler equation with MLIR operations.

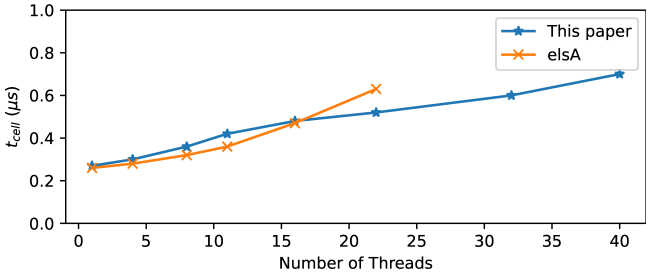

Figure 15. Cell execution time per iteration and per thread.

Figure 15 shows the average computation time t_{cell} as a function of the number of threads for both MLIR and elsA. For elsA, we only provide results up to 22 threads, which corresponds to the number of cores on one socket. Indeed, elsA's OpenMP implementation is optimized for single-socket execution, while a hybrid MPI/OpenMP strategy is used on larger NUMA systems. The performance is similar indicating that we successfully automated the generation of optimized code that replicates the manual optimizations in elsA.

5 Related Work and Design Discussion

Let us now present the most closely related work. We also take the opportunity to discuss some of our design choices.

CFD Solvers. Commercial and open-source CFD simulation environments include YALES2 [28], AVBP [35], elsA [9, 10], CEDRE [33], SU2 [16]. These applications cover a large spectrum of physical models (incompressible, compressible or weakly compressible flows, reactive flows, multiphase flows and turbulence modeling), mesh types (structured, unstructured or adaptive meshes), numerical methods (implicit or explicit time integration), upwind (like Roe and HLL/HLLC [34, 36]) or central (LAX-FRIEDRICH, Jameson) schemes, high order methods, etc. Because of this variability, CFD solvers largely rely on custom-built implementations in C, C++ or Fortran: it has been very difficult for generic packages of numerical methods such as PETSc [5] to find their way into state-of-the-art systems. This variability combined with the manual implementation of numerical methods induce high complexity and maintenance cost for CFD experts.

All these solvers rely on iterative stencil computations, both in-place and out-of-place. Thus, they often use similar optimization strategies as the ones we proposed in this paper. For example in YALES2 [28], the authors consider a double domain decomposition, the first for sub-domain parallelism and the second for cache-blocking. In [22], the formation of hierarchical wavefronts for LU-SGS enables coarse-grained parallelism and vectorization. For structured grids, such a vectorization strategy involves scatter and gather instructions, while for unstructured grids meshes may be ordered along wavefronts to benefit from contiguous memory access.

Our approach acknowledges the diversity of models and methods and that numerical libraries cannot handle such variability while offering competitive performance. We advocate for a generative approach instead, following the pioneering works of FEniCS [2] and Firedrake [32]. Yet the kernel code generators in these frameworks did not handle in-place stencils, and this limitation persists [27].

Stencil Code Generators. The experimental evaluation of stencil kernels considered Pluto as a reference. Indeed, general-purpose polyhedral compilers implement the kind of wavefront parallelization needed to optimize in-place stencils, but this is not the case of domain-specific code generators for stencil, as we have seen in the introduction [4, 13, 26, 27, 29, 31]. Note that ExaStencils [26] has been evaluated on a colored variant of the Gauss-Seidel method, but this variant is effectively an out-of-place stencil with inferior convergence guarantees.

In addition to the default parallelogram tiles, Pluto offers a diamond tiling strategy which allows to tile across stencil iterations without resorting to wavefront parallelism [7]. Overlapped tiling is another alternative with similar properties, implemented in Halide [31] and Polymage [29]. Such strategies are not applicable to in-place stencils, and when running Pluto on the out-of-place Jacobi stencil we did not activate this option for fairness reasons.

Affine Scheduling. Rather than constructing an explicit dependence graph in Section 2.3, we could have resorted to affine scheduling to derive a schedule from a reduced representation independent on the size of the coordinate space. This approach widely used in polyhedral compilation [8, 17]. In the uniform dependence case of stencil computations, with the ordering hypothesis we enforced on intra-iteration dependences, it is always possible to construct a schedule as a linear form: $\theta(\vec{\imath}) = \vec{n} \cdot \vec{\imath}$ where $\vec{n}$ is a vector of size equal to the space dimension and $\vec{\imath}$ is the coordinate of an element or sub-domain. A valid schedule satisfies $-\vec{n} \cdot \vec{r} \geq 1$ for every vector $\vec{r} \in L$. Optimizing a schedule consists in minimizing its latency $\max_{\vec{\imath},\vec{\jmath}}(\vec{n} \cdot (\vec{\imath} - \vec{\jmath}))$ which can be reduced to a linear optimization problem. While such a schedule is independent on the problem size, it is only optimal "up to a constant" [15]. Piecewise affine schedules [18] or index-set splitting [20] can overcome this problem, with extra heuristic complexity. Given that graph scheduling is practical in our context, we did not investigate affine scheduling further.

Non-rectangular Tiling. The tiling policy presented in Section 2.1 may appear overly restrictive—setting the tile size of dimensions with negative distances to 1. A more relaxed constraint—considered in advanced vectorization techniques—consists in setting the tile size of such a dimension to the minimum positive offset in the L subset [1], but we did not observe real-life scenarios of higher order stencils where this would be beneficial.

A more promising approach to lift the restriction would be to implement parallelogram tiling with "skewed" tile shapes aligned with the sub-domain schedule wavefronts, which is exactly what Pluto does [8]. The empirical results have not been encouraging however, mostly due to the loss of vectorization opportunities. Indeed, unlike rectangular tiles, parallelogram ones lead to chaotic intersections at boundaries of the coordinate space, resulting in a variety of so-called "partial tiles", where the boundary conditions prevent effective vectorization. It remains an open problem whether further autotuning and hybrid strategies could deliver better results than plain rectangular tiles.

6 Conclusion

We presented a domain-specific code generator for iterative in-place stencils, the cornerstone of computational fluid dynamics (CFD) simulations. The proposed code generation strategy and optimizations are implemented in MLIR, integrated with the generic tensor compilation abstractions and transformations provided by the framework. We evaluated the performance and expressiveness of our code generator on a realistic use case: the end-to-end implementation of an implicit CFD solver. We demonstrated its competitiveness with an industry-grade solver. Empirical evaluation on stencil kernels also outperform traditional polyhedral source-to-source parallelization and optimization.

Leveraging the MLIR framework, the next steps include portability experiments on GPU and heterogeneous platforms with hardware accelerators. This may involve further research on piecewise affine scheduling, and layout transformations to exploit fine-grained parallelism on GPUs.

Acknowledgments

We would like to express our gratitude to our colleagues S. Bourasseau, B. Berthoul, C. Content, S. Hoyer, M. Lienhart, M. Ravishankar and D. Gueyffier for valuable insights and support.

References

[1] Randy Allen and Ken Kennedy. 2001. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach.* Morgan Kaufmann, San Francisco, CA.

[2] Garth Wells Anders Logg, Kent-Andre Mardal (Ed.). 2012. *Automated Solution of Differential Equations by the Finite Element Method The FEniCS Book*. Springer.

[3] Andrew W Appel. 1998. SSA is functional programming. *ACM SIGPLAN Notices* 33, 4 (1998), 17–20.

[4] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. Association for Computing Machinery, Washington, DC, 193–205. https://doi.org/10.1109/CGO.2019.8661197

[5] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Steven Benson, Jed Brown, Peter Brune, Kris Buschelman, Emil M. Constantinescu, Lisandro Dalcin, Alp Dener, Victor Eijkhout, William D. Gropp, Václav Hapla, Tobin Isaac, Pierre Jolivet, Dmitry Karpeev, Dinesh Kaushik, Matthew G. Knepley, Fande Kong, Scott Kruger, Dave A. May, Lois Curfman McInnes, Richard Tran Mills, Lawrence Mitchell, Todd Munson, Jose E. Roman, Karl Rupp, Patrick Sanan, Jason Sarich, Barry F. Smith, Stefano Zampini, Hong Zhang, Hong Zhang, and Junchao Zhang. 2022. PETSc Web page. https://petsc.org/. https://petsc.org/

[6] Aart J.C. Bik, Penporn Koanantakool, Tatiana Shpeisman, Nicolas Vasilache, Bixia Zheng, and Fredrik Kjolstad. 2022. Compiler Support for Sparse Tensor Computations in MLIR. *ACM Transactions on Architecture and Code Optimization* 19, 4 (June 2022), 1–25. https://doi.org/10.1145/3544559

[7] Uday Bondhugula, Aravind Acharya, and Albert Cohen. 2016. The Pluto+ Algorithm: A Practical Approach for Parallelization and Locality Optimization of Affine Loop Nests. *ACM Trans. Program. Lang. Syst.* 38, 3, Article 12 (apr 2016), 32 pages. https://doi.org/10.1145/2896389

[8] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. *SIGPLAN Not.* 43, 6 (jun 2008), 101–113. https://doi.org/10.1145/1379022.1375595

[9] L. Cambier, M. Gazaix, S. Heib, S. Plot, M. Poinot, J.P. Veuillot, J.F. Boussuge, and M. Montagnac. 2011. An Overview of the Multi-Purpose elsA Flow Solver. *Aerospace Lab* 2 (March 2011), p. 1–15. https://hal.science/hal-01182452

[10] Laurent Cambier, Sébastien Heib, and Sylvie Plot. 2013. The Onera elsA CFD software: input from research and feedback from industry. *Mechanics & Industry* 14, 3 (2013), 159–174. https://doi.org/10.1051/meca/2013056

[11] Prasanth Chatarasi, Jun Shirako, Albert Cohen, and Vivek Sarkar. 2019. A Unified Approach to Variable Renaming for Enhanced Vectorization. In *Languages and Compilers for Parallel Computing*, Mary Hall and Hari Sundar (Eds.). Springer International Publishing, Cham, 1–20.

[12] RF Chen and ZJ Wang. 2000. Fast, block lower-upper symmetric Gauss-Seidel scheme for arbitrary grids. *AIAA journal* 38, 12 (2000), 2238–2245.

[13] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 579–594.

[14] Yishen Chen, Charith Mendis, Michael Carbin, and Saman Amarasinghe. 2021. VeGen: A Vectorizer Generator for SIMD and Beyond. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 902–914. https://doi.org/10.1145/3445814.3446692

[15] Alain Darte, Leonid Khachiyan, and Yves Robert. 1991. Linear scheduling is nearly optimal. *Parallel Processing Letters* 1, 02 (1991), 73–81.

[16] Thomas D. Economon, Francisco Palacios, Sean R. Copeland, Trent Lukaczyk, and Juan J. Alonso. 2016. SU2: An Open-Source Suite for Multiphysics Simulation and Design. *AIAA Journal* 54 (2016), 828–846.

[17] Paul Feautrier. 1992. Some efficient solutions to the affine scheduling problem. I. One-dimensional time. *International journal of parallel programming* 21, 5 (1992), 313–347.

[18] Paul Feautrier. 1992. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *International journal of parallel programming* 21, 6 (1992), 389–420.

[19] Anne Greenbaum. 1997. *Iterative Methods for Solving Linear Systems*. Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9781611970937

[20] Martin Griebl, Paul Feautrier, and Christian Lengauer. 2000. Index set splitting. *International Journal of Parallel Programming* 28, 6 (2000), 607–631.

[21] F. Irigoin and R. Triolet. 1988. Supernode Partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '88)*. Association for Computing Machinery, New York, NY, USA, 319–329. https://doi.org/10.1145/73560.73588

[22] Kazuhiko Komatsu, Yuta Hougi, Masayuki Sato, and Hiroaki Kobayashi. 2022. A hierarchical wavefront method for LU-SGS. *Computers & Fluids* 245 (2022), 105572. https://doi.org/10.1016/j.compfluid.2022.105572

[23] Samuel Larsen and Saman Amarasinghe. 2000. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. *ACM SIGPLAN Notices* 35, 5 (may 2000), 145–156. https://doi.org/10.1145/358438.349320

[24] Chris Lattner and Vikram Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 75–86. https://doi.org/10.1109/CGO.2004.1281665

[25] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. Association for Computing Machinery, 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[26] Christian Lengauer, Sven Apel, Matthias Bolten, Shigeru Chiba, Ulrich Rüde, Jürgen Teich, Armin Größlinger, Frank Hannig, Harald Köstler, Lisa Claus, Alexander Grebhahn, Stefan Groth, Stefan Kronawitter, Sebastian Kuckuk, Hannah Rittich, Christian Schmitt, and Jonas Schmitt. 2020. ExaStencils: Advanced Multigrid Solver Generation. In *Software for Exascale Computing - SPPEXA 2016-2019*, Hans-Joachim Bungartz, Severin Reiz, Benjamin Uekermann, Philipp Neumann, and Wolfgang E. Nagel (Eds.). Springer International Publishing, Cham, 405–452.

[27] Fabio Luporini, Mathias Louboutin, Michael Lange, Navjot Kukreja, Philipp Witte, Jan Hückelheim, Charles Yount, Paul H. J. Kelly, Felix J. Herrmann, and Gerard J. Gorman. 2020. Architecture and Performance of Devito, a System for Automated Stencil Computation. *ACM Trans. Math. Softw.* 46, 1, Article 6 (apr 2020), 28 pages. https://doi.org/10.1145/3374916

[28] Vincent Moureau, Pascale Domingo, and Luc Vervisch. 2011. Design of a massively parallel CFD code for complex geometries. *Comptes Rendus Mécanique* 339, 2 (2011), 141–148. https://doi.org/10.1016/j.crme.2010.12.001 High Performance Computing.

[29] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. 2015. PolyMage: Automatic Optimization for Image Processing Pipelines. In *ASPLOS*. Association for Computing Machinery, 429–443. https://doi.org/10.1145/2694344.2694364

[30] Evelyn Otero. 2012. *Acceleration of compressible flow simulations with edge using implicit time stepping*. Ph. D. Dissertation. KTH Royal Institute of Technology. QC 20120626, Trita-AVE, ISSN 1651-7660; 2012:25.

[31] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. *SIGPLAN Not.* 48, 6 (jun 2013), 519–530. https://doi.org/10.1145/2499370.2462176

[32] Florian Rathgeber, David A. Ham, Lawrence Mitchell, Michael Lange, Fabio Luporini, Andrew T. T. McRae, Gheorghe-Teodor Bercea, Graham R. Markall, and Paul H. J. Kelly. 2016. Firedrake: Automating the Finite Element Method by Composing Abstractions. *ACM Trans. Math. Softw.* 43, 3, Article 24 (dec 2016), 27 pages. https://doi.org/10.1145/2998441

[33] A. Refloch, B. Courbet, A. Murrone, P. Villedieu, C. Laurent, P. Gilbank, J. Troyes, L. Tessé, G. Chaineray, J.B. Dargaud, E. Quémerais, and F. Vuillot. 2011. CEDRE Software. *Aerospace Lab* 2 (March 2011), p. 1–10. https://hal.science/hal-01182463

[34] Philip L. Roe. 1981. Approximate Riemann solvers, parameter vectors, and difference schemes. *J. Comput. Phys.* 43, 2 (1981), 357–372. https://doi.org/10.1016/0021-9991(81)90128-5

[35] T. Schönfeld and Michael Rudgyard. 1999. Steady and Unsteady Flow Simulations Using the Hybrid Flow Solver AVBP. *AIAA Journal* 37 (1999), 1378–1385.

[36] Eleuterio F Toro, Michael Spruce, and William Speares. 1994. Restoration of the contact surface in the HLL-Riemann solver. *Shock waves* 4, 1 (1994), 25–34.

[37] Jan Treibig, Gerhard Wellein, and Georg Hager. 2011. Efficient multicore-aware parallelization strategies for iterative stencil computations. *Journal of Computational Science* 2, 2 (2011), 130–137. https://doi.org/10.1016/j.jocs.2011.01.010 Simulation Software for Supercomputers.

[38] Nicolas Vasilache, Oleksandr Zinenko, Aart J. C. Bik, Mahesh Ravishankar, Thomas Raoux, Alexander Belyaev, Matthias Springer, Tobias Gysi, Diego Caballero, Stephan Herhut, Stella Laurenzo, and Albert Cohen. 2022. Composable and Modular Code Generation in MLIR: A Structured and Retargetable Approach to Tensor Compiler Construction. *CoRR* abs/2202.03293 (2022), 43 pages. arXiv:2202.03293 https://arxiv.org/abs/2202.03293

[39] Michael E Wolf and Monica S Lam. 1991. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*. Association for Computing Machinery, 30–44.

[40] Michael Wolfe. 1986. Loops skewing: The wavefront method revisited. *International Journal of Parallel Programming* 15, 4 (1986), 279–293.

[41] Michael Joseph Wolfe, Carter Shanklin, and Leda Ortega. 1995. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., USA.

[42] Seokkwan Yoon and Dochan Kwak. 1994. Multigrid convergence of an implicit symmetric relaxation scheme. *AIAA journal* 32, 5 (1994), 950–955.