New Insights on Scalar Promotion with the Polyhedral Model

Alec Sadler
Inria, CNRS, ENS de Lyon, UCBL
Lyon, France
alec.sadler@inria.fr

Nathan Chandanson
Inria, CNRS, ENS de Lyon, UCBL
Lyon, France
nathan.chandanson@ensta.org

Hugo Thievenaz
CEA List
Saclay, France
hugo.thievenaz@cea.fr

Christophe Alias
Inria, CNRS, ENS de Lyon, UCBL
Lyon, France
christophe.alias@inria.fr

Abstract

Memory accesses are a well known bottleneck whose impact might be mitigated by using properly the memory hierarchy until registers. In this paper, we address scalar promotion, a technique to turn temporary arrays into a collection of scalar variables to be allocated to registers. We revisit array scalarization in the light of the recent advances of the polyhedral model. We propose a general algorithm for array scalarization and we show a scalarization of stencil computations thanks to a preliminary preprocessing. Our scalarization algorithm operates on the polyhedral intermediate representation and could be plugged in a polyhedral compiler among other passes. In particular, our scalarization algorithm is parametrized by the program schedule, possibly computed by a previous compilation pass. We present a preliminary experimental validation with promising results.

Keywords: Compiler Optimization, Polyhedral Model, Scalar Promotion, Code Generation

ACM Reference Format:
Alec Sadler, Nathan Chandanson, Hugo Thievenaz, and Christophe Alias. 2026. New Insights on Scalar Promotion with the Polyhedral Model. In *Proceedings of International Workshop on Polyhedral Compilation Techniques (IMPACT).* ACM, New York, NY, USA, 11 pages.

1 Introduction

Minimizing the memory transfers by using properly the memory hierarchy until registers is of prime importance to improve the performances of a program. Many compiler optimizations were designed to improve data locality by reorganizing the computation and the data layout. In particular, *scalar promotion*, or *array scalarization*, [7, 9, 14, 19] consists in transforming an array into a group of scalar variables, to be allocated to *registers*. In addition to improve the memory traffic, hence the overall performances, it generally enable more compiler optimizations, as dependences resolved through registers might be finely analyzed. In particular, *register tiling* [9, 14] splits a computation into blocks where register pressure makes possible scalar promotion. Most of these approaches are monolithic, they are designed as end-to-end optimizations without taking account of the scheduling constraints induced by previous compilation passes. General purpose compilers enable such optimizations through a series of symbolic analysis [2], but they have inherent limitations when applied to variables with loop-carried dependences.

We focus on the *polyhedral model* [10–13, 17, 18], a general framework to design loop transformations and data remapping for code optimization. Polyhedral analysis makes possible to reason about programs transformations, and how they affect schedule and dependences properties.

In this paper, we rephrase array scalarization as a *generic* polyhedral compilation pass, parametrized by an *input schedule* – the result of a previous polyhedral compilation pass. We exploit *array contraction* [5, 15] to expose array-level data reuse. Also, we propose a preliminary preprocessing to *enable the scalarization of stencil computations* by exploiting reuse between read points, as well as a postprocessing to *mitigate the complexity of the generated code*. At the end, we expose a *minimum amount of scalar variables* ready to be assigned a register. Specifically, we make the following contributions:

- We propose an *algorithm for array scalarization* based on *array contraction*, *ready to be plugged in a polyhedral compiler*. In particular, our algorithm is parametrized by the program schedule which might be the result of a previous polyhedral pass.
- Our transformation *exposes directly the scalar variables to be put in distinct registers*. This way, the work of the register allocator is reduced compared to the other approaches for scalarization.
- We propose a preliminary *preprocessing to enable scalarization of stencil computations* by exploiting reuse across stencil reads. This preprocessing is not inherently bound to stencil computations and could be applied to any computation where the same data is read by different instances of the same statement.

IMPACT, Kraków, Poland
2026.

- We propose a *post-processing* on the scalarized polyhedral intermediate representation to *mitigate the control complexity* on the final code.
- We present a *preliminary experimental evaluation* of our approach. In particular, we analyze the effect of our control improvement post-processing on the final code performance.

The remainder of this paper is structured as follows. Section 2 presents the required notions in polyhedral compilation. Section 3 presents related work. Section 5 describes our scalarization algorithm. Then, Section 6 presents our preprocessing for scalarizing stencil computations and Section 7 presents our postprocessing for mitigating code complexity. Section 8 presents our experimental validation. Finally, Section 9 concludes this paper and draws research perpectives.

2 Preliminaries

This section outlines the concepts of polyhedral compilation used in this paper. In particular, we define the polyhedral intermediate representation of a program. Then, we outline array contraction.

2.1 Polyhedral model

The polyhedral model [10–13, 17, 18] is a general framework to design loop transformations, historically geared towards source-level automatic parallelization [13] and data locality improvement [6]. It abstracts loop iterations as a union of convex polyhedra – hence the name – and data accesses as affine functions. This way, precise – iteration-level – compiler algorithms may be designed (dependence analysis [10], scheduling [12] or loop tiling [6] to quote a few) .

2.2 Polyhedral intermediate representation (IR)

In polyhedral compilers, the intermediate representation (IR) usually consists of a *program* P summarized as a set of *statements* S and their *iteration domains* $\mathcal{D}_S$. In a polyhedral statement, each of its iteration is uniquely represented by the vector of enclosing loop counters $\vec{i}$. The execution of a program statement S at iteration $\vec{i}$ is denoted by $\langle S, \vec{i} \rangle$ and is called an *operation* or an *execution instance*. Figure 1.(b) provides the iteration domains $\mathcal{D}_S = \{(y,x) \mid 0 \leq y < 2 \wedge 0 \leq x < N\}$, $\mathcal{D}_T = \mathcal{D}_U = \{(y,x) \mid 2 \leq y < N \wedge 0 \leq x < N\}$ for the 2D blur filter presented later. The iteration domains might be parametrized (here by N). Statements are accompanied with a *schedule* θ (typically the original sequential order), an optional *tiling* ϕ and in our case an optional *array contraction function* σ.

2.3 Polyhedral transformations

To introduce our algorithm, we first need to list needed loop transformations, and how they apply to a polyhedral representation.

Scheduling. A *schedule* θ_S assigns each operation $\langle S, \vec{i} \rangle$ with a timestamp $\theta_S(\vec{i}) \in (\mathbb{Z}^d, \ll)$. Intuitively, $\theta_S(\vec{i})$ is the iteration of $\langle S, \vec{i} \rangle$ in the transformed program. A schedule is *correct* if $\langle S, \vec{i} \rangle \rightarrow \langle T, \vec{j} \rangle \Rightarrow \theta_S(\vec{i}) \ll \theta_T(\vec{j})$, where $\rightarrow$ denotes the *dependence relation* between operations. The lexicographic order ensures that the dependence is preserved. In the polyhedral model, we focus on *affine schedules* $\theta_S(\vec{i}) = A\vec{i} + \vec{b}$. If f is an affine mapping $f(\vec{i}) = A\vec{i} + \vec{b}$, the *linear part* of f, $\operatorname{lin} f$, is such that $(\operatorname{lin} f)(\vec{i}) = A\vec{i}$.

Tiling. *Tiling* is a reindexing transformation which groups iteration into tiles to be executed atomically. There are many variants of this transformation. *Rectangular tiling* reindexes any iteration $\vec{i} \in \mathcal{D}_S$ to an iteration $(\vec{i}_{block}, \vec{i}_{local})$ such that $\vec{i} = \mathcal{T}_S(\vec{i}_{block}, \vec{i}_{local})$, with $\mathcal{T}_S(\vec{i}_{block}, \vec{i}_{local}) = (\operatorname{diag} \vec{s})\ \vec{i}_{block} + \vec{i}_{local}$, $0 \leq \vec{i}_{local} < \vec{s}$ where $\vec{s}$ is a vector collecting the tile size across each dimension of the iteration domain. $\vec{i}_{block}$ is called the *outer* tile iterator and $\vec{i}_{local}$ is called the *inner* tile iterator. The companion schedule associated to the tiling $\theta_S(\vec{i}_{block}, \vec{i}_{local})$ orders $\vec{i}_{block}$ first to to ensure the execution tile by tile. Figure 1.(b) gives an example of rectangular tiling with $\vec{s} = (4)$. To enforce the atomicity (avoid cross dependences between two tiles), it is sometimes desirable to precede the tiling by an injective affine mapping ϕ_S. The coordinates of $\phi_S(\vec{i})$, for $\vec{i} \in \mathcal{D}_S$ are usually called *tiling hyperplanes*. In that case, the transformation $\mathcal{T}_S^{-1} \circ \phi_S$ for some statements S is called an *affine tiling*. Note that rectangular tiling is a particular case of affine tiling where ϕ_S is the identity mapping.

Array contraction. Arrays might be remapped with an *allocation function* $a[\vec{i}] \mapsto a_{opt}[\sigma_a(\vec{i})]$, usually with a smaller footprint. In the polyhedral model, we focus on mappings $\sigma_a : \vec{i} \mapsto A\vec{i} \mod \vec{s}(\vec{N})$, where A is an integral matrix and $\vec{s}$ is an affine function applied on program parameters $\vec{N}$. Several algorithms were designed to derive such allocations, they are briefly discussed in the next section.

3 Related Work

This section outlines the related work on *scalar promotion*, *register tiling* and *array contraction*, which all closely relate to our paper. We also discuss *memory reuse* optimizations for stencils, which relates to our preprocessing step.

Scalar promotion. *Scalar promotion* or *register promotion* is the storage of a dependency (a value produced to be stored for later) in registry instead of memory. As register access is way faster than memory access, this effectively removes the loading time, but register entries usually are of very limited quantity. Since the seminal work of Callahan, Carr and Kennedy [7], many approaches were developed for scalar promotion. Most of these are coupled with a *loop tiling* and refered to as *register tiling*, as described in the next paragraph.

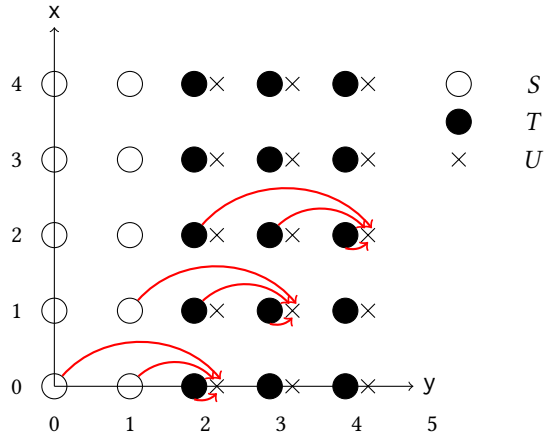
In LLVM, passes like *Scalar Replacement of Aggregate*, coupled with symbolic expressions analysis from the ScalarEvolution framework [2], break complex memory objects and loop-invariant variables into individual scalars, which can then be promoted as registers with the *mem2reg* transformation pass. While highly effective, these passes have inherent limitations when applied to loop-carried dependences. Thus, they conservatively restrict register promotion for variables involved in such loop.

Register tiling. *Register tiling* uses the loop tiling transformation to exploit data locality at the register level. Loop unrolling and tiling made its debut in the domain of parallelism, as a way to expose parallelization opportunities, by assigning each tile to a computing unit in order to parallelize their computation, assuming no dependency is broken. The notion of tiling is a general method to circumvent the limited number of computing resources, by cutting the iteration domain of the program into tiles that fit into the target memory level (register, cache, memory...). Jiménez et al. [14] present an approach to the problem of scalar promotion for non-rectangular perfect loop nests by tiling the iteration space appropriately. They detail a source-to-source transformation of the program, the locality analysis, where they perform a reuse analysis to search for the candidates for promotion with the highest temporal reuse, and use their heuristic to determine the tiling parameters. Then, Renganarayana et al. [19] presented a technique to extract an unrollable kernel from an imperfect loop nest, allowing the optimization to work on more complex program inputs yet again. More recently, Domagala et al. [9] presented a novel approach to register tiling, by using innermost-loop scheduling to expose data reuse. The scheduling is done *ad-hoc* by unrolling and rescheduling the innermost-loop under dependence constraints, and then tiling the iteration space resulting of the statement order and innermost iteration index dimensions. Therefore, the order of the statements within the loop is considered as a dimension too, which brings a new perspective to the problem. However, their method is restricted to perfect loops and focuses only on the deepest index space to promote from.

Array contraction and memory reuse. *Array contraction* [1, 5, 15, 20] consists in finding a storage function that maps elements of an array to their storage location, such that the resulting storage requirement is minimized similarly to register allocation. Array liveness is exploited to maximize *memory reuse*, and thereby to reduce the overall footprint. *Memory reuse* is also exploited to design efficient hardware for stencil computations [8, 23], using FIFO lines to convey the data to the different read points of the stencil. There has been some work to automate the synthesis of such designs [4], particularly in the context of polyhedral process networks [21], a dataflow model of computation for heterogeneous multiprocessor platforms and hardware synthesis.

```
1  for(y=0; y<2; y++)
2    for(x=0; x<N; x++) {
3  S:  blurx[x,y] = in[x,y]
4          + in[x+1,y] + in[x+2,y];
5      }
6  for(y=2; y<N; y++)
7    for(x=0; x<N; x++) {
8  T:  blurx[x,y] = in[x,y]
9          + in[x+1,y] + in[x+2,y];
10 U:  out[x,y] = blurx[x,y-2] +
11       blurx[x,y-1] + blurx[x,y];
12     }
```

(a) 2D Blur filter



(b) Iteration domains and dependences

Figure 1. Motivating example 1: 2D Blur filter

All these approaches are complementary to ours, we address the unification of memory reuse and scalar promotion for software optimization.

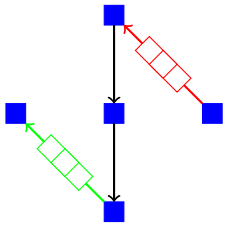
4 Motivating Examples

We illustrate our scalarization approach on two examples: the 2D blur filter kernel depicted in Figure 1 – used as running example in Section 5 – and the 2D Jacobi stencil depicted in Figure 2 – used as running example to describe our data systolization preprocessing in Section 6.

Example 1: 2D Blur filter. The computation is divided into two steps. First, an *horizontal filter* (statements S and T) is applied to the input picture `in` and stores the result into the array `blurx`. Then, a *vertical filter* (statement U) is applied to `blurx` and stores the final result to the array `out`. The whole might be seen as a producer/consumer through the temporary array `blurx`. So of the two arrays `in` and `blurx`, only the latter might be contracted and then *scalarized*; provided array contraction leads to a constant (non-parametrized by N) size.

```
  for(t=0; t<T; i++) {
    for(i=1; i<N-1; i++) {
      for(j=1; j<N-1; j++) {
S:      out[i,j] = in[i-1,j] + in[i+1,j] +
          in[i,j] + in[i,j-1] + in[i,j+1];
    }}
    for(i=1; i<N-1; i++) {
      for(j=1; j<N-1; j++) {
T:      in[i,j] = out[i,j];
    }}
  }
```

(a) 2D Jacobi kernel



(b) Data pipelining across S reads while executing $\langle S, t, i, j\rangle$ for a fixed t.

Figure 2. Motivating example 2: 2D Jacobi stencil

With the original schedule $\theta_S(y,x) = (0,y,x)$, $\theta_T(y,x) = (1,y,x,0)$, $\theta_U(y,x) = (1,y,x,1)$, 3 iterations of x must be completed before the execution of U. Indeed, the second filter applied by U would require three *vertical* cells of `blurx` (as shown in Figure 1.b), in particular the three first, for each x. Hence the allocation $\sigma_{blurx}(x,y) = (x \bmod N, y \bmod 3)$, with the non-constant (parametrized) footprint $3N$. Hence loop transformations are required to lower the liveness of `blurx` to a constant level. This is done with the schedule $\theta_S(y,x) = (0,x,y), \theta_T(y,x) = (1,x,y,0), \theta_U(y,x) = (1,x,y,1)$, which leads to the allocation $\sigma_{blurx}(x,y) = (x \bmod 1, y \bmod 3)$ with a constant footprint 3 where scalarization might now be applied. Sometimes, affine schedules are not sufficient, and tiling may be required. At worst, arrays might be privatized per tile and our algorithm would reproduce the results of a *register tiling*.

Our algorithm will try to scalarize *provided an input schedule*. How to compute a schedule leading to scalarization is *not* the scope of this paper. In general, schedules obtained by minimizing the dependence distance [6] are relevant candidates.

Example 2: 2D Jacobi stencil. On this example, each output point `out[i,j]` is computed from a fixed neighbourhood of `in[i,j]`. At first glance, there is no possible scalarization, since the arrays are either live-in or live-out. However, we observe that each *datum circulates in a pipelined fashion between each read* of `in` during the execution. Indeed, with the sequential schedule $\theta_S(t,i,j) = (t,0,i,j)$ and $\theta_S(t,i,j) = (t,1,i,j)$ the data `in[i+1,j]` read at iteration (t,i,j) will be read as `in[i,j+1]` at iteration $(t, i+1, j-1)$, then as `in[i,j]` at iteration $(t, i+1, j)$ and so on until the last read as `in[i-1,j]` at iteration $(t, i+2, j)$. This pipeline, illustrated on Figure 2.(b), is called ***data systolization***.

In this paper, we show how to produce a scalarized program *emulating data systolization with register variables*. Section 6 will show how to preprocess the program so our polyhedral scalarisation algorithm presented in the next section produces such a transformation.

5 Scalarization

This section presents our scalarization algorithm. This algorithm may be composed with our stencil specific preprocessing (Section 6) and our code complexity mitigation postprocessing (Section 7).

At a glance. First, we illustrate the basic principle of our algorithm on Example 1. Provided the loop permutation schedule $y \leftrightarrow x$ and the allocation $\sigma_{blurx}(x,y) = (x \bmod 1, y \bmod 3)$, each read `blurx[x,y]` may be rephrased

$$\texttt{blurx\_c}[\sigma_{blurx}(x,y)]$$

where the array `blurx_c` is the *contracted version* of `blurx`, with a size 1×3. Applying an unrolling of 3 on the y loop enclosing U, the loop body would have the statements:

```
out[x,y]   = blurx_c[0,y-2 mod 3] +
             blurx_c[0,y-1 mod 3] +
             blurx_c[0,y mod 3];
out[x,y+1] = blurx_c[x,y-1 mod 3] +
             blurx_c[0,y mod 3] +
             blurx_c[0,y+1 mod 3];
out[x,y+2] = blurx_c[x,y mod 3] +
             blurx_c[0,y+1 mod 3] +
             blurx_c[0,y+2 mod 3];
```

Now, since the y loop is unrolled with a factor 3, $y \bmod 3$ is ***constant***, so are $y-1 \bmod 3$, $y-2 \bmod 3$, $y+1 \bmod 3$ and $y+2 \bmod 3$. *Hence, all the reads to blurx might replaced by a register*. Two challenges must be addressed:

- Compute the unroll factors when several arrays are contracted with different allocations σ.
- We do not consider loops, but *affine schedules*. This is somehow equivalent, as schedule dimensions are virtually loop counters. However, it complicates the whole procedure.

Main algorithm. Algorithm 1 depicts the main algorithm. We input the result of the previous polyhedral compilation pass: a polyhedral IR of a program (P, θ) with an array allocation σ and an optional loop tiling ϕ. Both scheduling and tiling might be either imposed by a previous polyhedral pass, or required to bound the liveness of some temporary arrays to a constant size. This latter point will be discussed further in Section 6. Then, we output the polyhedral IR of the scalarized program (P_{out}, θ_{out}), which might feed the next polyhedral compilation pass until the final code generation.

Algorithm 1: Scalarization

```
Data: Program (P, θ), allocation σ, optional tiling φ
Result: Scalarized program (P_out, θ_out)
1  begin
2      Skip arrays with parametrized modulo
3      if No array remains then
4          return (P_out, θ_out)
5      end
6      U ⟵ Unroll_Factors(P, θ, σ)
7      (P_out, θ_out) ⟵ Code_Generation(P, θ, φ, σ, U)
8      return (P_out, θ_out)
9  end
```

Arrays which still have a parametric size are skipped (step 2). When no array remains, our algorithm stops and returns the original program (step 4). Then, we scalarize the arrays with constant size. First, we compute the unrolling factors for the loops formally described by θ (step 6, Algorithm 2). These are the loops produced after the final polyhedral code generation for P under the scheduling constraint θ. Of course, we *do not have* syntactically these loops at this point of the polyhedral compilation, and we have to reason directly on θ. Then, we produce the polyhedral IR for the final scalarized program (step 7). We apply the unrolling with respect to θ and we generate the program statements with *scalar* variables to be allocated to registers.

The two next subsections present our algorithms Unroll_Factors and Code_Generation.

5.1 Unrolling the Loops

The main challenge is to find out the minimum unroll factors to expose constant array indices (after contraction), so they might be subsequently substituted by a scalar variable. Consider Algorithm 2. We first rephrase array indices to use the loop counters prescribed by θ (time counters t_i). If $\sigma_a(\vec{c}) = A\vec{c} \bmod \vec{s}$, the index $u(\vec{i})$ for the reference $a[\sigma_a(u(\vec{i}))]$ of statement S is rephrased $A \circ u \circ \theta_S^{-1}(\vec{t}) \bmod \vec{s}$, since $\theta_S(\vec{i}) = \vec{t}$ (definition of θ_S). The remainder focuses on the affine part $A \circ u \circ \theta_S^{-1}(\vec{t})$.

Algorithm 2: Unroll_Factors

```
Data: Program (P, θ), allocation σ
Result: U: time dimension (θ) ↦ unroll factor
1  begin
2      U(t_i) ⟵ 1, for each time dimension t_i
3      foreach reference S : ... a[u(i⃗)] ... do
4          Write σ_a(c⃗) = Ac⃗ mod s⃗
           /* Unroll time dimensions (θ)        */
5          Let (f_1(t⃗), ..., f_p(t⃗)) = A ∘ u ∘ θ_S^{-1}(t⃗)
6          foreach index dimension f_k(t⃗) do
7              foreach variable t_i in f_k(t⃗) do
8                  U(t_i) ⟵ lcm(U(t_i), s⃗_k)
9              end
10         end
11     end
12     return U
13 end
```

The following lemma proves that the obtained unroll factors expose constant array indices. We denote by $\vec{u} \times \vec{v}$ the element-wise multiplication of vectors: $(u_1, \ldots, u_n) \times (v_1, \ldots, v_n) = (u_1 \times v_1, \ldots, u_n \times v_n)$.

Lemma 5.1. *Let $\vec{U} = (\mathcal{U}(t_1), \ldots, \mathcal{U}(t_n))$ and $S : \ldots a[u(\vec{i})] \ldots$ a reference to a contracted array in P. Then, with unroll factors $\mathcal{U}$, the reference is constant (the same cell $a[\vec{c}_0]$ at each iteration):*

$$\exists \vec{c}_0 \in \mathbb{Z}^p : \forall \vec{k} \in \mathbb{Z}^n : \ \sigma_a \circ u \circ \theta_S^{-1}(\vec{k} \times \vec{U}) = \vec{c}_0$$

Proof. We use the notations defined in Algorithm 2. The k-th dimension of $\sigma_a \circ u \circ \theta_S^{-1}(\vec{k} \times \vec{U})$ may be written: $(\alpha_k(\vec{k} \times \vec{U}) + \beta_k) \bmod s_k$, which develops to: $(\alpha_k(\vec{k} \times \vec{U}) \bmod s_k + \beta_k \bmod s_k) \bmod s_k$ (since $\mathbb{Z} \to \mathbb{Z}/s_k\mathbb{Z}$ is a ring morphism) Now, each non-null U_i in the expression $\alpha_k(\vec{U})$ is a multiple of s_k (line 8), so is $\alpha_k(\vec{k} \times \vec{U})$. Hence $\alpha_k(\vec{k} \times \vec{U}) \bmod s_k = 0$. Therefore, the k-th dimension of $\sigma_a \circ u \circ \theta_S^{-1}(\vec{k} \times \vec{U})$ is the constant $\vec{c}_{0_k} = \beta_k \bmod s_k$. □

This shows the correctness of our unroll factors. We point out that our unroll factors are minimal, as we use an lcm (step 8).

Running example (cont'd). Recall the schedules $\theta_S(y, x) = (0, x, y, 0)$, $\theta_T(y, x) = (1, x, y, 0)$, $\theta_U(y, x) = (1, x, y, 1)$. Hence $\theta_S^{-1} = \theta_T^{-1} = \theta_U^{-1} = (t_1, t_2, t_3, t_4) \mapsto (t_3, t_2)$. Also, we have $\sigma_{blurx}(x, y) = (x \bmod 1, y \bmod 3)$, hence A is the rank 2 identity matrix. The references to $blurx$, written as $A \circ u \circ \theta_S^{-1}(\vec{t})$ are (step 5): $blurx[x, y] \mapsto blurx[t_2, t_3]$, $blurx[x, y-2] \mapsto blurx[t_2, t_3 - 2]$, $blurx[x, y-1] \mapsto blurx[t_2, t_3 - 1]$. Finally, we obtain: $\mathcal{U} : t_1 \mapsto 1, t_2 \mapsto 1, t_3 \mapsto 3, t_4 \mapsto 1$. □

5.2 Code Generation

We generate the polyhedral representation of the scalarized program with Algorithm 3. For each statement S, each loop $t_j$is unrolled by a factor $\mathcal{U}(t_j) = U_j$ (step 5). This is expressed by an euclidian division: $t_j = \theta_S(\vec{i})_j = k_j \times U_j + \pi_j$ with $0 \leq \pi_j < U_j$ (euclidian division), k_j being the counter of the unrolled loop for t_j and π_j being the unroll offset in that loop. The tiling constraints are [6]: $\vec{i} \in \mathcal{D}_S \wedge \vec{T} = \phi_S(\vec{i})/\vec{S}$, where / denotes the element-wise euclidian division and $\vec{S}$ is the tile size along each hyperplane.

Algorithm 3: Code_Generation

Data: Program (P, θ), tiling ϕ, allocation σ, unroll factors $\mathcal{U}$
Result: Scalarized program (P_{out}, θ_{out})

1 **begin**
2 $\quad \vec{U} \longleftarrow (\mathcal{U}(t_1), \ldots, \mathcal{U}(t_n))$
3 $\quad$ **foreach** *statement* S **do**
4 $\quad\quad$ **foreach** $\vec{\pi} \in [\![0, \mathcal{U}(t_1)[\![\times \ldots \times [\![0, \mathcal{U}(t_n)[\![$ **do**
5 $\quad\quad\quad \mathcal{D}_{S,\vec{\pi}} \longleftarrow \{(\vec{T}, \vec{k}, \vec{i}) \mid \theta_S(\vec{i}) = \vec{k} \times \vec{U} + \vec{\pi} \wedge$ tiling_constraints$(\mathcal{D}_S, \phi_S, \vec{T}, \vec{i})\}$
6 $\quad\quad\quad \theta_{S,\vec{\pi}}(\vec{T}, \vec{k}, \vec{i}) \longleftarrow (\vec{T}, k_1, \pi_1, \ldots, k_n, \pi_n)$ /* final scalarization */
7 $\quad\quad\quad$ Set a new statement $S_{\vec{\pi}}(\vec{T}, \vec{k}, \vec{i})$ from $S(\vec{i})$ by substituting each reference $a[u(\vec{i})]$ by register_a$_{\sigma_a \circ u \circ \theta_S^{-1}(\vec{\pi})}$
8 $\quad\quad$ **end**
9 $\quad$ **end**
10 $\quad$ Write P_{out} the collection domain:statement $\mathcal{D}_{S,\vec{\pi}} : S_{\vec{\pi}}$
11 $\quad$ Write θ_{out} the collection of schedules $\theta_{s,\vec{\pi}}$
12 $\quad$ **return** (P_{out}, θ_{out})
13 **end**

The following theorem proves the correctness of the schedule computed at step 6.

Theorem 5.2. *If θ is correct, Then: $\theta_{S,\vec{\pi}}$ is a correct schedule over $\mathcal{D}_{S,\pi}$, for any $\vec{\pi}$ enumerated in Algorithm 3.*

Proof. Since $\theta_S(\vec{i}) = \vec{k} \times \vec{U} + \vec{\pi}$ (element-wise euclidian division), the lexicographic order of $(k_1, \pi_1, \ldots, k_n, \pi_n)$ is the same as $\theta_S(\vec{i})$. Hence the correctness of $\theta_{S,\vec{\pi}} : (\vec{T}, \vec{k}, \vec{i}) \mapsto (\vec{T}, k_1, \pi_1, \ldots, k_n, \pi_n)$. □

The following lemma proves correctness of the register naming at step 7.

Lemma 5.3. *For any $(\vec{T}, \vec{k}, \vec{i}) \in \mathcal{D}_{S,\vec{\pi}}$, the index function of a, $\sigma_a \circ u \circ \theta_S^{-1}(\vec{t})$ depends only on $\vec{\pi}$ and is equal to $\left(\vec{c}_0 + \text{lin}\left(A \circ u \circ \theta_S^{-1}(\vec{\pi})\right)\right) \bmod \vec{s}$ for some constant vector $\vec{c}_0 \in \mathbb{Z}^n$, where $\vec{t} = \theta_S(\vec{i}) = \vec{k} \times \vec{U} + \vec{\pi}$ (step 5).*

Proof. With the notations of the algorithm,
$\sigma_a \circ u \circ \theta_S^{-1}(\vec{k} \times \vec{U} + \vec{\pi})$
becomes after affine decomposition:
$\left(A \circ u \circ \theta_S^{-1}(\vec{k} \times \vec{U}) + \text{lin}\left(A \circ u \circ \theta_S^{-1}\right)(\vec{\pi})\right) \bmod \vec{s}$.
From then, we distribute the modulos to get:
$\left(\sigma_a \circ u \circ \theta_S^{-1}(\vec{k} \times \vec{U}) + \text{lin}\left(A \circ u \circ \theta_S^{-1}\right)(\vec{\pi}) \bmod \vec{s}\right) \bmod \vec{s}$.
which further simplifies (Lemma 5.1) to:
$\left(\vec{c}_0 + \text{lin}\left(A \circ u \circ \theta_S^{-1}\right)(\vec{\pi})\right) \bmod \vec{s}$.
This final equation depends only on $\vec{\pi}$. □

Adding a constant vector to the left hand side of $\left(\vec{c}_0 + \text{lin}\left(A \circ u \circ \theta_S^{-1}\right)(\vec{\pi})\right) \bmod \vec{s}$ does not change its exclusive dependence on π, hence we may safely use $A \circ u \circ \theta_S^{-1}(\vec{\pi}) \bmod \vec{s} = \sigma_a \circ u \circ \theta_S^{-1}(\vec{\pi})$ instead to name the register (step 7).

Running example (cont'd). For each statement S, T, U, we enumerate all the values of $\vec{\pi} \in \{0\} \times \{0\} \times [\![0, 2]\!] \times \{0\}$. For instance, for S and the first combination $\vec{\pi} = (0, 0, 0, 0)$, we generate:

- $\mathcal{D}_{S,(0,0,0,0)} = \{(k_1, k_2, k_3, k_4, y, x) \mid \theta_S(y, x) = (0, x, y, 0) = (k_1.1+0, k_2.1+0, k_3.3+0, k_4.1+0) \wedge 0 \leq x < N \wedge 0 \leq y < 2\}$
- $\theta_{S,(0,0,0,0)}(k_1, k_2, k_3, k_4, y, x) = (k_1, 0, k_2, 0, k_3, 0, k_4, 0)$
- $S_{(0,0,0,0)} : register\_blurx_{(0,0)} = in[x, y] + in[x+1, y] + in[x+2, y]$;

The 11 other combinations for $\vec{\pi}$ are processed in the same way. Given to a polyhedral code generator, this produces the desired scalarized program. □

6 Exposing Data Systolization

This section outlines a preprocessing to enable scalarization of stencil kernels. After this preprocessing, our scalarization algorithm will naturally emulate a data systolization pipeline as depicted on Figure 2. We discuss the principle of our preprocessing on two examples: a 1D convolution and the 2D jacobi stencil.

A simple 1D-convolution. For the sake of clarty, and without loss of generality, we present our ideas on a simple 1D convolution:

```
1 for(i=1; i<N-1; i++) {
2   out[i] = in[i] + in[i+1]; //S
3 }
```

Both arrays are either live-in or live-out, hence no direct scalarization is possible. However, the data is *pipelined across the reads*: read `a[i+1]` at iteration i is read as `a[i]` at iteration $i+1$. Consequently, each iteration should require only one memory access (a[i+1]) the other one (a[i]) being obtained through some *pipeline registers*. However, this would work only for $i \geq 2$, once the pipeline is in *steady state*. Hence

the need to add *initialization iterations* or *ghost iterations* to have a steady state pipeline for any iteration $i \geq 1$:

```
for(i=1-δ_{i,ℓ}; i<N-1+δ_{i,u}; i++) {
  pipeline[i+1] = in[i+1]; //P
  if(1 ≤ i < N-1)
    out[i] = pipeline[i] +
        pipeline[i+1]; //S
}
```

Now, the shifts $\delta_{i,\ell}$ and $\delta_{i,u}$ must be tuned so *each read of* `pipeline` *by S is defined be some instance of P*. This might be infered from an *array dataflow analysis* of `pipeline` reads. The operation producing `pipeline[i+1]` (read 2 of $\langle S, i\rangle$) is:

$$h_{S,2}(i) = \langle P, i\rangle$$

Hence it is always defined whatever δ_i. The operation producing `pipeline[i]` is:

$$h_{S,1}(i) = \text{if}(i + \delta_{i,\ell} \geq 2) \text{ then } \langle P, i\rangle \text{ else } \bot$$

Hence, the read `pipeline[i]` by $\langle S, i\rangle$ is defined provided $i + \delta_{i,\ell} \geq 2$ for any $i \in \mathcal{D}_S$. To enforce a definition for $i \geq 1$, we must have $2 - \delta_i \leq 1$ or equivalently $\delta_i \geq 1$. We take the smallest solution to minimize the amount of ghost iterations: $\delta_{i,\ell} = 1$. Any value for $\delta_{i,u}$ work, hence we can avoid ghost iterations at the end of the loop: $\delta_{i,u} = 0$.

Finally, an array contraction on the resulting program gives $\sigma_{pipeline}(i) = i \bmod 2$. From this input, our scalarization algorithm would deduce an unroll factor of 2 and produce the following code:

```
for(i=0; i<N-1; i+=2) {
  register1 = in[i+1];
  if(i ≥ 1)
    out[i] = register0 + register1;
  register0 = in[i+2];
  if(i+1 ≥ 1)
    out[i+1] = register1 + register0;
}
```

which emulates the pipeline through the register variables `register0` and `register1`. We now show how the same code may be produced for a 2D stencil.

2D Jacobi stencil. Consider the 2D Jacobi stencil depicted in Figure 2.(a). As discussed in section 4, the reads of statement S might be systolized. To simplify the presentation, we consider the execution of S for a given outer iteration t. The code described thereafter will be executed for each iteration t. Our preprocessing gives the program:

```
for(i=1-δ_{i,ℓ}; i<N-1+δ_{i,u}; i++) {
  for(j=1-δ_{j,ℓ}; j<N-1+δ_{j,u}; j++) {
    pipeline[i+1,j] = in[i+1,j] //P
    if(1 ≤ i,j < N-1)
      out[i,j] = pipeline[i-1,j] + ...
          + pipeline[i+1,j]; //S
}}
```

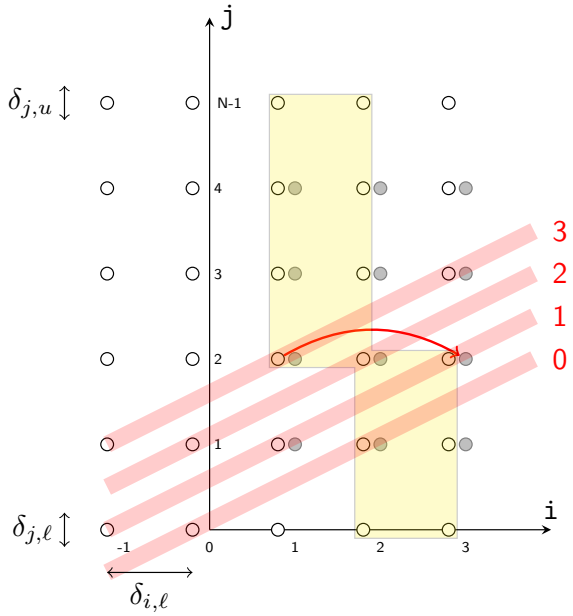


Figure 3. 2D Jacobi stencil: liveness and array contraction of the *pipeline* variable with $\sigma_{pipeline}(i, j) = -i + 2j \bmod 2N + 1$

Similarly, an array dataflow analysis infers that `pipeline` reads by $\langle S, i, j\rangle$ are defined by some instance of P providing $i \geq 3 - \delta_{i,\ell}$, $j \geq 2 - \delta_{j,\ell}$ and $j \leq N - 3 + \delta_{j,u}$. Hence, the optimal shifts are $\delta_{i,\ell} = 2$, $\delta_{i,u} = 0$, $\delta_{j,\ell} = \delta_{j,u} = 1$. Applying an array contraction on the resulting program, we obtain $\sigma_{pipeline}(i, j) = -i+2j \bmod 2N+1$, hence a final size of $2N+1$ for `pipeline`.

Figure 3 depicts the iteration domain of P (∘) and S (•) as well as the liveness range of `pipeline[2][2]` (in yellow) from its definition ($\langle P, 1, 2\rangle$) to its last use ($\langle S, 3, 2\rangle$). Thick red lines show `pipeline` array cells mapped to the same location after array contraction by $\sigma_{pipeline}$. For instance, the thick red line labelled by 3 indicates that pipeline cells written by $\langle P, -1, 1\rangle$, $\langle P, 1, 2\rangle$ and $\langle P, 3, 3\rangle$ are all mapped to array cell 3 after contraction. In particular, when $\langle S, 3, 2\rangle$ is executed, the contracted array cell 3 is freed and then reused by $\langle P, 3, 3\rangle$.

Since the size of `pipeline` after contraction is parametrized, our method cannot be directly applied. This is resolved by cutting the iterations in the j direction with a *tiling hyperplane*. Each tile will have to execute ghost iterations with the same δ parameters to ensure the correctness of the code. Hence, if the tile size across j is b, the footprint of `pipeline` would be $2(b + \delta_{j,\ell} + \delta_{j,u}) + 1 = 2b + 5$. For the scalarization to saturate the 16 registers of an x86-64 processor, we need $2b + 5 \leq 16$, that is $b \leq 5$. In our experiments, we take $b = 4$.

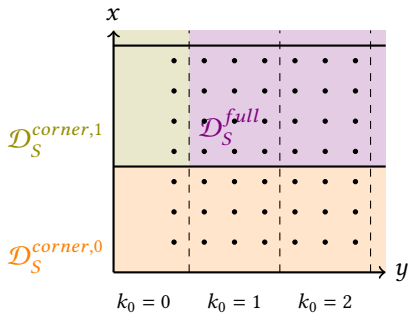


Figure 4. Separation obtained on the 2D Blur filter kernel

7 Mitigating Control Complexity

This section presents a post-processing of the polyhedral IR produced by our scalarization algorithm to reduce the control complexity after code generation by a polyhedral code generator [3, 22].

At a glance. Consider Example 1 and assume a tiling $\phi_S(y,x) = \phi_T(y,x) = x$ with tile size $b = 4$. The tiling is partially depicted on Figure 4 (points denote instances of U, full lines represents tile hyperplanes, dashed lines separates each iteration k_0 of the loop unrolling along y). Also, assume the schedule $\theta_S(T_1,y,x) = (T_1,0,y,x)$, $\theta_T(T_1,y,x) = (T_1,1,y,x,0)$, $\theta_U(T_1,y,x) = (T_1,1,y,x,1)$, which reproduces the original sequential order in each tile. This example will be referred to as *2D blur tiled* in the following. Array contraction gives $\sigma_{blurx}(x,y) = (x \bmod 3, y \bmod 3)$. Then, our scalarization algorithm find the following unroll factors $\mathcal{U}$: $y \mapsto 3, x \mapsto 4$. While the code's runtime is greatly improved (as discussed in Section 8), we also observe that the ISCC code generator introduces a *significant number of branching in the generated code to check corner cases*. This increase in branch instructions has a certain impact on the binary size, which also impacts performance and branch-miss at runtime. To fix this, we simply separate the *steady cases*: the iterations $\vec{k}$ of unrolled loops executing *all* the unrolled instances $\vec{\pi}$; and the *corner cases* where some unrolled instance $\vec{\pi}$ are not executed. This is illustrated in Figure 4. This way, steady iterations (most of the iterations) will be free of corner case conditionals.

Our algorithm. Our code complexity mitigation post-processing is depiceted on Algorithm 4. First, we list the origin vertices with $k_i = 0$ for all i. On our example with $k_0 = k_1 = 0$, we would have $C_{\mathcal{U}} = \{(0,0),(0,3),(2,0),(2,3)\}$. From there, we compute loop iterations that contain each vertex s_i, that we store in each $\mathcal{D}_S^{\vec{s_i}}$. Finally, intersecting all $\mathcal{D}_S^{\vec{s_i}}$ gives $\mathcal{D}_S^{full}$ the set of iterations containing all vertices. Finally, the complementary of $\mathcal{D}_S^{full}$ in $\mathcal{D}_S^{all}$ gives the $\mathcal{D}_S^{corner,i}$ that we add to the final polyhedral IR. This separation eliminates most of the conditional in loops, thus lowers branch-instruction at runtime.

Algorithm 4: $\mathcal{D}_S^{full}$ and $\mathcal{D}_S^{corner,i}$ separation

Data: program(P_{out}, θ_{out}), Unroll-factors $\mathcal{U} \sim \vec{U}$
Result: domain-separated program (P_{sep}, θ_{sep})

begin
  $C_{\mathcal{U}} \longleftarrow \{\vec{s} \in \{(0,\mathcal{U}_0-1),\dots,(0,\mathcal{U}_n-1)\}$;
  foreach *Statement* S **do**
    Get each domain $\mathcal{D}_S^{\vec{s_i}}$ containing $\vec{s_i}$;
    $\mathcal{D}_S^{\vec{s_i}} \longleftarrow \left\{(\vec{T},\vec{k},\vec{i}) \mid \theta_S(\vec{i}) = \vec{k}\times\vec{U} + \vec{s_i}\right\}$;
    Get full domain $\mathcal{D}_S^{full}$;
    $\mathcal{D}_S^{full} \longleftarrow \bigcap_{\mathcal{D}_S^{\vec{s_i}}} \text{projection}_{/(\vec{T},\vec{k})}\left(\mathcal{D}_S^{\vec{s_i}}\right)$;
    Using $\vec{\Pi} \in [\![0,\mathcal{U}(t_1)[\![\times\cdots\times[\![0,\mathcal{U}(t_n)[\![$;
    $\mathcal{D}_S^{all} \longleftarrow \text{proj}_{/(\vec{T},\vec{k})}\left\{(\vec{T},\vec{k},\vec{i}) \mid \theta_S(\vec{i}) = \vec{k}\times\vec{U} + \vec{\Pi}\right\}$;
    $\mathcal{D}_S^{corner} \longleftarrow \mathcal{D}_S^{all} \setminus \mathcal{D}_S^{full}$;
    foreach *Statement* $S_{\vec{\pi}}$ **do**
      $\mathcal{D}_{S,\vec{\pi}}^{full} \leftarrow \mathcal{D}_{S,\vec{\pi}} \cap \mathcal{D}_S^{full}$;
      $\mathcal{D}_{S,\vec{\pi}}^{corner} \leftarrow \mathcal{D}_{S,\vec{\pi}} \cap \mathcal{D}_S^{corner}$;
      $P_{sep} \longleftarrow$ add statement $(S_{\vec{\pi}}^{full}, \mathcal{D}_{S,\vec{\pi}}^{full})$;
      $P_{sep} \longleftarrow$ add statement $(S_{\vec{\pi}}^{corner}, \mathcal{D}_{S,\vec{\pi}}^{corner})$;
      $\theta_{sep} \longleftarrow$ add schedule $\theta_{S,\vec{\pi}}$;
    end
  end
  return (P_{sep}, θ_{sep}) ;
end

8 Experimental Results

This section presents a preliminary experimental evaluation of our approach.

8.1 Experimental setup

We have implemented our scalarization algorithm and our separation post-processing using the PoCo polyhedral compilation framework. The scalarized code was generated using the polyhedral code generator of ISL [22]. The systolization was applied by hand with a slighly different preprocessing: the pipeline array was explicitely contracted and shifted each iteration, similarly to a shift register; which adds an extra penalty. We evaluate our approach on the following kernels of the Polybench suite [16]: *gemm*, *jacobi-1d*, *jacobi-2d*, *2mm*, *gesummv*, *symm*, *correlation*, *covariance*, and *atax*. We also added *2D-blur-filter*, *2D-blur-filter tiled* (see Section

7) and the *fibonacci* kernel with dynamic programming in the experiments.

The kernel *gemm* could not be directly scalarized as no array is temporary. Hence it was preprocessed with a tiling such that each tile loads input regions of A and B into private arrays which are then scalarized. All the kernels use the `double` type for its data, except for *fibonacci* that used `int`. All the kernels were compiled using `g++ 15.1.1` with the `-O3` optimisation flag. The benchmarks were all executed on an 8 cores x86-64 AMD processor, that features 16 `%xmmi` vector/floating point registers, 64KB of L1 and 512KB of L2 cache per core, and a shared L3 cache of 16MB.

The kernels *jacobi-1d* and *jacobi-2d* were preprocessed to use explicitely a pipeline array. The pipeline array was explicitely contracted and shifted each iteration, which is slightly less general and efficient than the preprocessing described on Section 6, but nonetheless lead to interesting speed-ups. For *jacobi-1d*, this implementation is quite straightforward, as there is only one dimension, and the calculation only needs three contiguous points. However, for *jacobi-2d*, to keep the size of the systolic array constant, one dimension was tiled to make the size of the systolic array parametrized by the tile size and not by the problem size.

Runtimes events and performances were registered on both default and scalarized programs using the `perf` linux profiling utility. We only monitor events inside the kernel's body. Each kernel was run a hundred times for evenly distributed problem sizes ; for each size N, the kernel was run 10 times and the results were averaged.

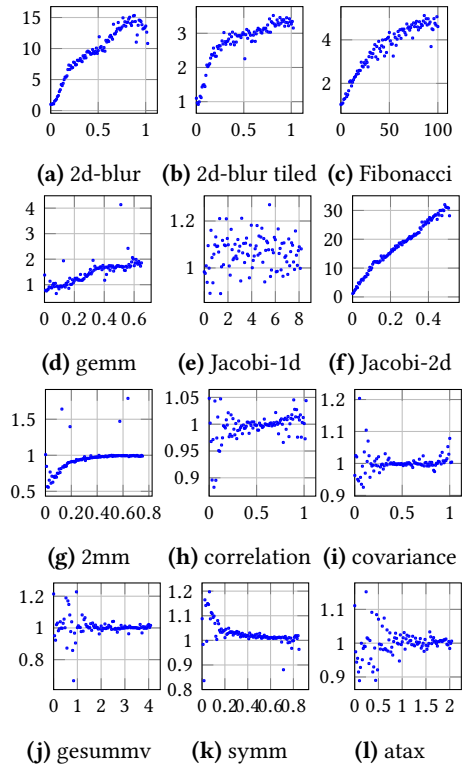


Figure 5. Speedup of each kernels. Speedups on the Y-axis are plotted over the input size N on the X-axis, scaled at 10^3.

8.2 Results

8.2.1 Execution time. We first discuss execution time speedups of the kernels. The results are depicted in Figure 5.

For the examples where no significant improvements were observed, there are different reasons. First, for the kernels *correlation* (resp. *covariance*), the scalarized arrays are the mean and stddev (resp. mean) arrays, which are used in the first part of the kernel to center and normalize (resp. center) the input data. However, for these two kernels, there is a second part of the code, that computes the correlation (resp. covariance) matrix. From the measures we took, this second part is more compute-intensive and takes a lot more time to complete. Thus, the gains obtained by scalarizing these temporary arrays are almost insignificant compared to the time the rest of the kernel takes to complete.

Then, the kernels *gesummv* and *symm* had their temporary arrays reduced to a single scalar by our algorithm. However, when looking at the GIMPLE representation of the optimized default kernels we saw that the compiler, when given the `-O3` flag, was able to see that each datum from the temporary arrays was only used in a single iteration, and that there was no dependances inter-iteration for the outermost loops. Thus, the values were computed using a single scalar value, and then stored inside the array, therefore eliminating all the loads and array manipulations to keep only one store at the end of each outer iteration. This is probably possible because the sequential schedule is straightforward and that it can be easily spotted that it is not needed to compute the values inside the arrays. Thus, our algorithm seems to make no difference, as the compiler performs similar optimizations to our when optimizing agressively. This is for a similar reason that the kernels *jacobi-1d* and *atax* saw no significant improvements : these examples are simple enough for the compiler to efficiently optimize them. It is important to note that even if the runtime results are similar, the compiler keeps the array whereas our method only uses a few scalars, thus also reducing the memory footprint taken by temporary values compared to the compiler-optimized code, which can be an important factor in memory-constrained architectures such as embedded ones.

Finally, for the kernel *2mm*, the issue is not the access to data, but rather the access to the program. Indeed, the measured number of cache references and misses decreased when scalarizing, even though the runtime is longer. After investigation, we saw that there were more branch instructions taken on the scalarized version, and even though the number of branch mispredictions was lower, these branches made the instruction pointer "jump" all around the program. These branches resulted in a significant increase in cache

misses of the instruction cache, therefore increasing the number of `stalled-cycles-frontend`, meaning the processor could not fetch the instructions rapidly enough to correctly pipeline the program. It happens because the generated code is quite complex, and has a lot of added control `if` statement.

On the other examples, we can observe speedups showing the effectiveness of our method. On some of these examples, the speedup obtained by scalarizing can be substantial, as shown for the examples *fibonacci* and *2d blur filter - Tiled* where the scalarized code ran up to 3 and 5 times faster than the default one, and even 15 and 30 times faster for *2d blur filter - Running example* and *jacobi-2d*. The reason for such an increase in performances for *jacobi-2d* and the fact that the speedup behaves linearly is that the systolisation array for the default code is parametrized by the size of the data and thus has a footprint of $O(N)$. overall, our method has significantly improved execution time for non-reduction programs.

8.2.2 Data systolization performances. We now discuss the systolized jacobi speedups described in Figure 5.e and 5.f. The scalarized 1D version has barely any differences as the initial code is already quite simple. For the 2D version, the classical code was first pipelined to produce the baseline code. Then, it was tiled in the vertical direction using a tile size of $b = 4$, to bound the `pipeline` array to a constant size ; before being scalarized to produce the optimized code. That way, the `pipeline` array has a size of $2b + 1$ instead of $2N + 1$, which allows the storage of its values in the registers. This is also the reason why the speedup is so important : the `pipeline` array in the non-scalarized code has a footprint of $O(N)$ which leads to more cache misses as the size of the array increases.

	2d-blur-filter			gemm		
	scalar	sep	sepM	scalar	sep	sepM
cpu-cycles	966k	712k	608k	32,7M	33M	32,5M
instructions	3,2M	2,7M	1,7M	124M	117M	124M
branch-instructions	208k	85k	22k	2,6M	2,6M	2,6M
branch-misses	136	143	145	5,6k	5,8k	5,6k
cache-references	200k	169k	202k	2,3M	2,3M	2,3M
cache-misses	26k	16k	25k	360k	386k	371k
time-elapsed	758µs	583µs	507µs	24,6ms	24,8ms	24,4ms

Table 1. Results from iteration separation

8.2.3 Post-processing performances. We finally discuss the behavior of the iteration separation algorithm on runtime events. Table 1 shows how Algorithm 4 discussed in Section 7 impacts runtime performances. We compare from `scalar` two versions: *sep* only separates *full* and *corner* iterations, while *sepM* also separates the different *corner* iterations in distincts loops. We observed two different behavior with this optimization. For blur-2D, separation greatly lowered the number of instructions at runtime: *sepM* dropped the amount of branch-instructions by a factor of 10. This also affects runtime, with a $1.3x$ speedup. On the other hand, certain kernels like *gemm* which initially contain few indirections find no real improvment with the algorithm. We observed behavior between *sep* and *sepM* for every kernel, and we found little to no differences in runtime events.

9 Conclusion

In this paper, we have proposed a complete algorithm for array scalarization as a composable pass in a polyhedral compiler. We also proposed preliminary preprocessing and post-processing steps to enable data systolization and to mitigate the control complexity of the final optimized code. We have also provided a complete correctness proof of our scalarization algorithm, completed with an experimental validation on a set of representative polyhedral kernels used in linear algebra and signal processing applications.

In the future, we would like to generalize and to automate our preliminary data systolization preprocessing and to investigate the interplay of our algorithm with other polyhedral optimization passes.

Acknowledgments

As part of the “France 2030” initiative, this work has benefited from a national grant managed by the French National Research Agency (Agence Nationale de la Recherche) attributed to the Exa-MA project of the NumPEx PEPR program, under the reference ANR-22-EXNU-0003.

References

[1] Christophe Alias, Fabrice Baray, and Alain Darte. 2007. Bee+Cl@k : An Implementation of Lattice-Based Array Contraction in the Source-to-Source Translator ROSE. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'07)*.

[2] Olaf Bachmann, Paul S. Wang, and Eugene V. Zima. 1994. Chains of recurrences—a method to expedite the evaluation of closed-form functions. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation* (Oxford, United Kingdom) *(ISSAC '94)*. Association for Computing Machinery, New York, NY, USA, 242–249. doi:10.1145/190347.190423

[3] Cédric Bastoul. 2003. Efficient Code Generation for Automatic Parallelization and Optimization. In *2nd International Symposium on Parallel and Distributed Computing (ISPDC 2003), 13-14 October 2003, Ljubljana, Slovenia*. 23–30.

[4] Protonu Basu, Mary Hall, Samuel Williams, Brian Van Straalen, Leonid Oliker, and Phillip Colella. 2015. Compiler-directed transformation for higher-order stencils. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE, 313–323.

[5] Somashekaracharya G Bhaskaracharya, Uday Bondhugula, and Albert Cohen. 2016. Automatic storage optimization for arrays. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 38, 3 (2016), 1–23.

[6] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the ACM SIGPLAN 2008 Conference on*

Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008. 101–113. doi:10.1145/1375581.1375595

[7] David Callahan, Steve Carr, and Ken Kennedy. 1990. Improving register allocation for subscripted variables. *ACM Sigplan Notices* 25, 6 (1990), 53–65.

[8] Riccardo Cattaneo, Giuseppe Natale, Carlo Sicignano, Donatella Sciuto, and Marco Domenico Santambrogio. 2015. On How to Accelerate Iterative Stencil Loops: A Scalable Streaming-Based Approach. *ACM Trans. Archit. Code Optim.* 12, 4, Article 53 (Dec. 2015), 26 pages. doi:10.1145/2842615

[9] Łukasz Domagała, Duco van Amstel, Fabrice Rastello, and Ponuswamy Sadayappan. 2016. Register allocation and promotion through combined instruction scheduling and loop unrolling. In *Proceedings of the 25th International Conference on Compiler Construction*. 143–151.

[10] Paul Feautrier. 1991. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming* 20, 1 (1991), 23–53. doi:10.1007/BF01407931

[11] Paul Feautrier. 1992. Some Efficient Solutions to the Affine Scheduling Problem. Part I. One-dimensional Time. *International Journal of Parallel Programming* 21, 5 (Oct. 1992), 313–348. doi:10.1007/BF01407835

[12] Paul Feautrier. 1992. Some Efficient Solutions to the Affine Scheduling Problem. Part II. Multidimensional Time. *International Journal of Parallel Programming* 21, 6 (Dec. 1992), 389–420. doi:10.1007/BF01379404

[13] Paul Feautrier and Christian Lengauer. 2011. Polyhedron Model. In *Encyclopedia of Parallel Computing*. 1581–1592.

[14] Marta Jiménez, José M Llabería, and Agustín Fernández. 2002. Register tiling in nonrectangular iteration spaces. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 24, 4 (2002), 409–453.

[15] Vincent Lefebvre and Paul Feautrier. 1998. Automatic storage management for parallel programs. *Parallel computing* 24, 3-4 (1998), 649–671.

[16] Louis-Noël Pouchet. 2012. Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/˜ pouchet/software/polybench/[cited July,]* (2012).

[17] Patrice Quinton and Vincent van Dongen. 1989. The mapping of linear recurrence equations on regular arrays. *Journal of VLSI signal processing systems for signal, image and video technology* 1, 2 (1989), 95–113.

[18] Sanjay V. Rajopadhye, S. Purushothaman, and Richard M. Fujimoto. 1986. On synthesizing systolic arrays from Recurrence Equations with Linear Dependencies. In *Foundations of Software Technology and Theoretical Computer Science*, Kesav V. Nori (Ed.). Lecture Notes in Computer Science, Vol. 241. Springer Berlin Heidelberg, 488–503.

[19] Lakshminarayanan Renganarayana, Uday Bondhugula, Salem Derisavi, Alexandre E Eichenberger, and Kevin O'Brien. 2009. Compact multi-dimensional kernel extraction for register tiling. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. IEEE, 1–12.

[20] Hugo Thievenaz, Keiji Kimura, and Christophe Alias. 2022. Lightweight Array Contraction by Trace-Based Polyhedral Analysis. In *C3PO 2022 - International Workshop on Compiler-assisted Correctness Checking and Performance Optimization for HPC*. Hamburg, Germany. https://inria.hal.science/hal-03862219

[21] Sven Verdoolaege. 2010. *Polyhedral Process Networks*. 931–965.

[22] Sven Verdoolaege. 2011. Counting Affine Calculator and Applications. In *IMPACT*.

[23] Hasitha Muthumala Waidyasooriya and Masanori Hariyama. 2019. Multi-FPGA accelerator architecture for stencil computation exploiting spacial and temporal scalability. *IEEE Access* 7 (2019), 53188–53201.