Compas'2022 : Parallélisme / Architecture/ Système
MIS/UPJV - Amiens France, 5-8 juillet 2022

A Polyhedral Approach for Scalar Promotion

Alec Sadler, Christophe Alias, Hugo Thievenaz

CNRS, ENS-Lyon, Inria, Kalray, University of Lyon, France
First.Last@inria.fr

Résumé
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 array scalarization, 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, a general framework to design optimizing program transformations. We propose a general algorithm for array scalarization, ready to be plugged in a polyhedral compiler, among other passes. Our scalarization algorithm operates on the polyhedral intermediate representation. In particular, our scalarization algorithm is parametrized by the program schedule possibly computed by a previous compilation pass. We rely on schedule-directed array contraction and we propose a loop tiling algorithm able to reduce the footprint down to the available amount of registers on the target architecture. Experimental results confirm the effectiveness and the efficiency of our approach.

Mots-clés : compiler optimization, polyhedral model, array scalarization

---

1. Introduction

Using properly memory hierarchy until registers is of prime importance to improve the performances of a program, especially with the increasing gap between the peak rate of processing arithmetic units and the memory bandwidth. This trend in computer architecture, called the *memory wall*, boils down to the invention of memory hierarchy, and its counterpart in automatic code optimization. *Array scalarization*, or *scalar promotion*, [3, 9, 13, 4] consists in transforming an array into a group of scalar variables, to be allocated to registers. In addition to reduce the memory traffic, hence the overall performances, it generally improves the precision of compiler optimizations, as dependences resolved through a register might be finely analyzed. In particular, *register tiling* [9, 4] splits a computation into blocks where register pressure make 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.
In this paper, we focus on the *polyhedral model* [12, 11, 5, 6, 7, 8], a general framework to design loop transformations and data remapping for code optimization. Polyhedral compilers makes possible to reason about programs and their transformations thanks to a powerful geometric abstraction. We propose to 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* [10, 1] to expose array-level data reuse, and we propose an additional loop tiling algorithm to reduce the memory footprint of temporary arrays to a tunable

constant size. 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 a *general algorithm for array scalarization, 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 *reduces as much as possible the code size* for array scalarization and *exposes directly the scalar variables to be put in distinct registers*. This way, the work of the register allocator is dramatically reduced compared to seminal approaches for scalarization.

— We propose a *loop tiling algorithm able to reduce to footprint of some temporary arrays to a constant value*. This algorithm is used on demand, when required.

— We present a complete experimental validation showing the effectiveness and the efficiency of our approach.

The remainder of this paper is structured as follows. Section 2 presents the required notions in polyhedral compilation. Section 3 describes our scalarization algorithm Section 4 presents our exprimental validation. Finally, Section 5 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.

2.1. Polyhedral model

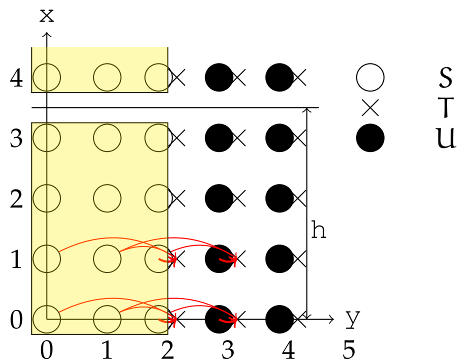
The polyhedral model [12, 11, 5, 6, 7, 8] is a general framework to design loop transformations, historically geared towards source-level automatic parallelization [8] and data locality improvement [2]. It abstracts loop iterations as a union of convex polyhedra – hence the name – and data accesses as affine functions. This way, precise – iteration-level – compiler algorithms may be designed (dependence analysis [5], scheduling [7] or loop tiling [2] to quote a few) . The polyhedral model manipulates program fragments consisting of nested `for` loops and conditionals manipulating arrays and scalar variables, such that loop bounds, conditions, and array access functions are *affine expressions* of surrounding loops counters and structure parameters (input sizes, e.g., N)). Thus, the control is static and may be analysed at compile-time. With polyhedral programs, each iteration of a loop nest 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*. The set $\mathcal{D}_S$ of iteration vectors is called the *iteration domain* of S. 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.

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* $\mathcal{D}_S$, a *schedule* θ (typically the original sequential order), an optional *tiling* ϕ (a reindexing transformation which groups iteration into tiles to be executed atomically) and an optional *array contraction function* σ (as arrays might be remapped with an allocation function $a[\vec{i}] \mapsto a_{opt}[\sigma_a(\vec{i})]$, usually with a smaller footprint) .

```
for(y=0; y<2; y++){
  for(x=0; x<N; x++) {
S:  blurx[x][y] = in[x][y]
        + in[x+1][y] + in[x+2][y];
    }
  }
for(y=2; y<N; y++)
  for(x=0; x<N; x++) {
T:  blurx[x][y] = in[x][y]
        + in[x+1][y] + in[x+2][y];
U:  out[x][y] = blurx[x][y-2] +
        blurx[x][y-1] + blurx[x][y];
    }
  }
```

a) Motivating example : 2D Blur filter



b) Iteration Domain for 2D Blur filter

FIGURE 1 – Running example : 2D Blur filter

3. Our Approach

This section outlines our approach on a running example.

3.1. Running example

We illustrate our scalarization approach on the 2D blur filter kernel depicted in Figure 1. 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`. Since `blurx` is a temporary array, it might be contracted and then *scalarized*, provided array contraction leads to a constant (non-parametrized by N) size.

We point out that the array `in` cannot be scalarized *directly* in statement S, since it is *not* a temporary array. Nonetheless, a temporary version of `in` produced by a loop at the beginning of the program could perfectly be contracted and then scalarized, with a register pressure depending on the time shift between the producer and S. This preprocessing is used on some of our experimental results.

Our scalarization algorithm is intended to be used in a polyhedral compilation chain. Hence a schedule might be imposed by the previous compilation steps. In the following, we consider two scheduling scenarios : the original execution order and a loop permutation.

Scenario 1. Original execution order

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 required three *vertical* cells of `blurx`, in particular the three first, for each x. Hence the allocation $\sigma_{\text{blurx}}(x, y) = (x \mod N, y \mod 3)$, with the non-constant (parametrized) footprint 3N. In that case, `blurx` cannot be directly scalarized. We propose to *tile* the iteration domain to limit the conflicting cells in the x direction. With that tiling, illustrated in Figure 1.(b), the footprint becomes $3h$ with h the tile size in the x direction. On a x86-64 machine with 14 general registers, we would set the tile size to $h = [14/3] = 4$.

Scenario 2. Loop permutation

We now assume that the outcome of the previous polyhedral compilation steps is a permutation of the loops x and y. This is described with the schedule $\theta_S(y, x) = (0, x, y)$, $\theta_T(y, x) = (1, x, y, 0)$, $\theta_U(y, x) = (1, x, y, 1)$. In that case, we directly have the allocation $\sigma_{\mathrm{blurx}}(x, y) = (x \mod 1, y \mod 3)$ with a constant footprint 3. Hence scalarization might be applied directly, without the need to apply further loop tiling.

3.2. Our algorithm

We now present our main algorithm (scalarization) and all its subroutines (tiling, unroll_factors and code_generation). They can be found in the appendix. Also, theorical proof of the algorithm might be found in the companion research report [14].

We input the result of the previous polyhedral compilation pass : a polyhedral IR of a program (P, θ) and an optional loop tiling ϕ. Then, we output the polyhedral IR of the scalarized program $(P_{\mathrm{out}}, \theta_{\mathrm{out}})$, which might feed the next polyhedral compilation pass until the final code generation. First, we try to contract temporary arrays with the original schedule and tiling, when it is provided (step 2). Input and output arrays are ignored, since they cannot be contracted. As mentioned in section 3.1, the only way to scalarized the references to input and output arrays is to substitute them by temporary arrays fed by an input loop (for input arrays), or read by an output loop (for output arrays) with a constant time shift. This might be addressed by a preprocessing polyhedral pass and will not be discussed further in this paper. When the contraction fails to produce only temporary arrays with constant size (step 3) and no loop tiling is imposed, we try to tile the program in such a way the footprint is reduced to a constant, non-parametrized, size (step 5, Algorithm 2). Then, the arrays are recontracted (step 6). At this point, the tile size is adjusted so the product of σ modulos fits the available amount of registers. This is simply done by iterating step 6 on tile size $\vec{S}$ from size $(1, \ldots, 1)$, incrementing each tile size component at each iteration, until the temporary arrays with constant contracted size all have a footprint (modulo product) tightly less than the available amount of registers. Arrays which still have a parametric size are skipped (step 8). When no array remains, meaning that the tiling failed to restrict at least one array to a constant size, our algorithm stops and returns the original program. Finally, we scalarize the arrays with constant size. First, we compute the unrolling factors for the loops formally described by θ (step 13, Algorithm 3). 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 14). We apply the unrolling (and our tiling ϕ when step 5 was required) with respect to θ and we generate the program statements with *scalar* variables to be allocated to registers.

Tiling Algorithm

We now describe our tiling procedure depicted in Algorithm 2. Our goal is to tile the program to bound the parametric terms of the array allocation σ. From now, we consider the running example, scenario 1. Recall that we obtained $\sigma_{\mathrm{blurx}}(x, y) = (x \mod N, y \mod 3)$, hence the need to tile the iteration domain on the x direction to restrict the number of conflicting array cells to a constant value. Actually, there is two notions of direction : a parametric direction in the *array index domain*, clearly identified : x, from which we deduce a parametric direction in the *iteration domain*, which happens to be the same, here. More precisely, given a statement S and an array reference $a[u(\vec{i})]$, we want to infer a variation $\vec{\Delta}_k$ in the iteration domain $\mathcal{D}_S$ of S which incurs a variation in the direction $\vec{\delta}_k$ (vector with 1 at position k, 0 elsewhere) in the direction k of the *array index domain* (here $k = 1, \delta_1 = (1, 0)$). If $\sigma_a(\vec{c}) = A\vec{c} \mod s(\vec{N})$, this amounts to solve :

$$A \circ u(\vec{\Delta}_k) = \vec{\delta}_k$$

This affine equation is classically solved thanks to standard linear algebra techniques (lines 8 to 15). Note that Q^{-g} denotes the generalized inverse of Q. The outcome is the set $\mathcal{P}_S$ of directions $\vec{\Delta}_k$ of the iteration domain $\mathcal{D}_S$ of statement S for which *at least* one reference $a[u(i)]$ makes a step in a parametric direction $\vec{\delta}_k$ according to σ_a. Then, a tiling is computed (line 19) using the pluto algorithm [2]. Finally, we keep only the hyperplanes going into a parametric direction.
We point out that our algorithm will lead to a contraction of temporary arrays to a constant size if hyperplanes do not cross dependences hold by those arrays. Otherwise, a copy of sources should be kept along complete slices of the iteration domain. Note that the pluto algorithm tends to avoid that pitfall by pushing the resolution of dependences to innermost hyperplanes.

4. Experimental Results

This section presents our experimental results on several polyhedral programs.

4.1. Experimental setup

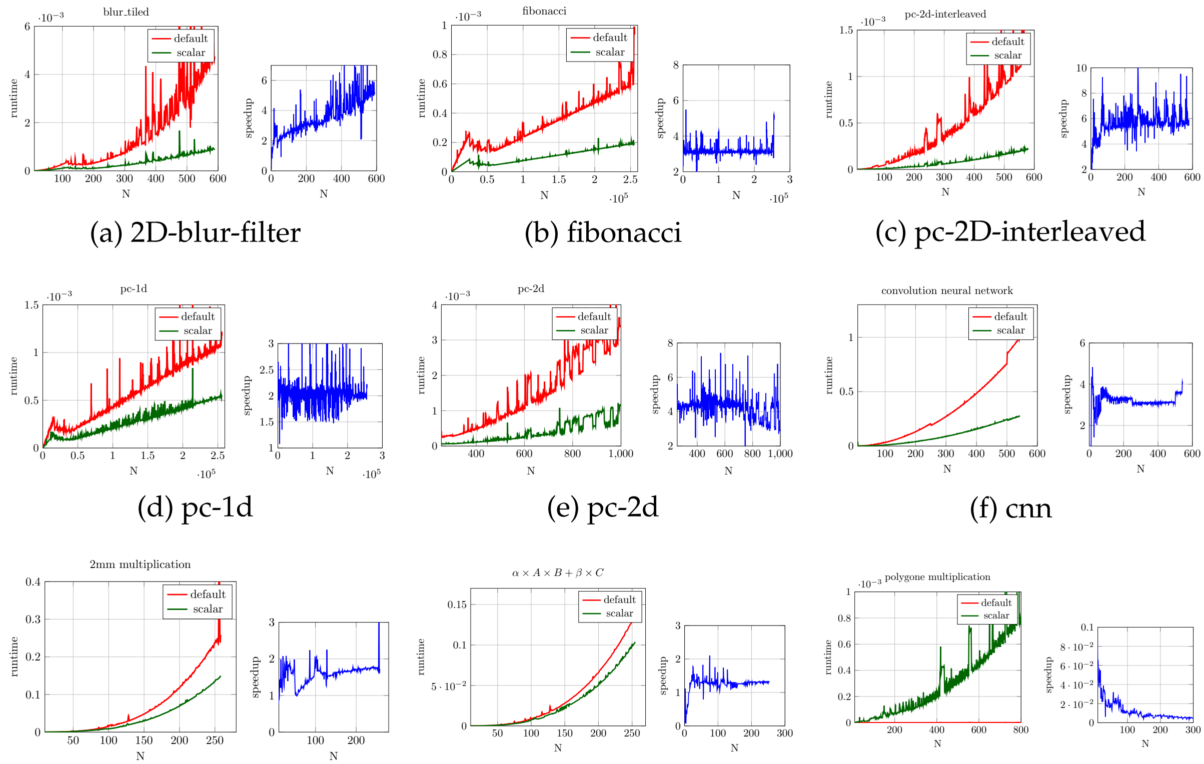
We have implemented our scalarization algorithm. The final code was generated using the iscc polyhedral code generator [15]. We have applied our algorithm to the following kernels :

- **2D-blur-filter.** Our running example, applying a 2D blur filter to an input.
- **fibonacci.** This kernel generates the N first fibonacci numbers, and returns the last one.
- **pc-2d-interleaved.** Producer/consumer throught a 2D array, where the consumer executes 2 iterations after the producer.
- **pc-1d.** Same as before for an array of one dimension
- **pc-2d.** This kernel applies a stencil pattern on a 2D array, with dependence vectors $(1, 0)$ and $(0, 1)$.
- **cnn.** Simple CNN with a convolutive layer followed a ReLU layer.
- **2mm.** Multiplication of three matrices together ($A \times B \times C$).
- **gemm.** BLAS kernel computing $C := \alpha A \times B + \beta C$. On the experiments, A and B where chosen as $N \times N$ matrices.
- **poly.** Multiplication of monovariate polynomials P and Q of degree N, represented by their array of coefficients.

Kernels *cnn*, *2mm*, *gemm* and *poly* were preprocessed to enable the contraction of input/output arrays, along the lines described in Section 3.1. Benchmarks were done by executing both the default and scalar program with different array sizes. Executions were made on a single-x86_64 intel CPU, with 14 registers. The CPU features 4 cores, with 64KB of cache L1, 512 KB of cache L2 and 4MB of cache L3. Compilation was done with GCC11 -O0 to measure exactly the impact of our optimization.

4.2. Results

Figure 2 depicts our results. Every graph shows runtimes for both default and scalar version, as well as the speed-up, for multiple array size. For every example, similar behaviours can be observed, such as cache effects when the memory footprint gets large enough. Cache memory becomes saturated, and another phase of the curve starts.



(a) 2D-blur-filter (b) fibonacci (c) pc-2D-interleaved
(d) pc-1d (e) pc-2d (f) cnn
(g) 2mm (h) gemm (i) poly

FIGURE 2 – Experimental results

For almost every example, we managed to speed up quite a lot the program. On *2D blur filter*, it is interesting to note that the scalarized version show a bigger growing rate compared to the default version, which translates to a speed-up increasing with the data size, unlike *fibonacci*, *pc-2D-interleaved*, *cnn*, *2mm*, and *gemm*, which exhibits a constant speed-up. On *pc-2d*, we observe instabilities on both curves with the ratio slighty going down. On *gemm* and *poly*, the poor performances are explained by the number of conditional branches in the target program to handle corner-cases, that we suspect to cause many branch misprediction. This is the main weakness of *direct* polyhedral code generation.

5. Conclusion

In this paper, we have proposed a complete algorithm for array scalarization as a composable pass in a polyhedral compiler. Our algorithms features a loop tiling to reschedule the input kernel so the footprint of the temporary arrays may be tuned to fit into the registers of the target architecture. We have also provided a complete correctness proof of our approach, 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 investigate how to improve the polyhedral code generation to reduce the conditional branches, which bound unexpectedly our speed-ups on some kernels.

Bibliographie

1. Bhaskaracharya (S. G.), Bondhugula (U.) et Cohen (A.). – Automatic storage optimization for arrays. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 38, n3, 2016, pp. 1–23.
2. Bondhugula (U.), Hartono (A.), Ramanujam (J.) et Sadayappan (P.). – 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*, pp. 101–113, 2008.
3. Callahan (D.), Carr (S.) et Kennedy (K.). – Improving register allocation for subscripted variables. *ACM Sigplan Notices*, vol. 25, n6, 1990, pp. 53–65.
4. Domagała (Ł.), van Amstel (D.), Rastello (F.) et Sadayappan (P.). – Register allocation and promotion through combined instruction scheduling and loop unrolling. – In *Proceedings of the 25th International Conference on Compiler Construction*, pp. 143–151, 2016.
5. Feautrier (P.). – Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, vol. 20, n1, 1991, pp. 23–53.
6. Feautrier (P.). – Some efficient solutions to the affine scheduling problem. Part I. one-dimensional time. *International Journal of Parallel Programming*, vol. 21, n5, octobre 1992, pp. 313–348.
7. Feautrier (P.). – Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *International Journal of Parallel Programming*, vol. 21, n6, décembre 1992, pp. 389–420.
8. Feautrier (P.) et Lengauer (C.). – Polyhedron model. *In : Encyclopedia of Parallel Computing*, pp. 1581–1592. – 2011.
9. Jiménez (M.), Llabería (J. M.) et Fernández (A.). – Register tiling in nonrectangular iteration spaces. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 24, n4, 2002, pp. 409–453.
10. Lefebvre (V.) et Feautrier (P.). – Automatic storage management for parallel programs. *Parallel computing*, vol. 24, n3-4, 1998, pp. 649–671.
11. Quinton (P.) et van Dongen (V.). – The mapping of linear recurrence equations on regular arrays. *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 1, n 2, 1989, pp. 95–113.
12. Rajopadhye (S. V.), Purushothaman (S.) et Fujimoto (R. M.). – On synthesizing systolic arrays from recurrence equations with linear dependencies. *In : Foundations of Software Technology and Theoretical Computer Science*, éd. par Nori (K. V.), pp. 488–503. – Springer Berlin Heidelberg, 1986.
13. Renganarayana (L.), Bondhugula (U.), Derisavi (S.), Eichenberger (A. E.) et O'Brien (K.). – Compact multi-dimensional kernel extraction for register tiling. – In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pp. 1–12. IEEE, 2009.
14. Sadler (A.), Alias (C.) et Thievenaz (H.). – *A Polyhedral Approach for Scalar Promotion*. – Research Report nRR-9437, Institut National de Recherche en Informatique et en Automatique (INRIA), novembre 2021.
15. Verdoolaege (S.). – Counting affine calculator and applications. – In *IMPACT*, 2011.

Appendices

2D blur filter, scenario 1, code generated

```
void blur_kernel(double* in, double* out, int N) {
  register double blurx_00,blurx_01,blurx_02,blurx_10,blurx_11,blurx_12,blurx_20,blurx_21,blurx_22,blurx_30,blurx_31,blurx_32;

  for (int c0 = 0; c0 <= floord(N - 1, 4); c0 += 1) {
    blurx_00 = in[4 * c0][0] + in[4 * c0+1][0] + in[4 * c0+2][0];
    if (N >= 4 * c0 + 2) {
      blurx_10 = in[4 * c0 + 1][0] + in[4 * c0 + 1 +1][0] + in[4 * c0 + 1 +2][0];
      if (N >= 4 * c0 + 3) {
        blurx_20 = in[4 * c0 + 2][0] + in[4 * c0 + 2 +1][0] + in[4 * c0 + 2 +2][0];
        if (N >= 4 * c0 + 4)
          blurx_30 = in[4 * c0 + 3][0] + in[4 * c0 + 3 +1][0] + in[4 * c0 + 3 +2][0];
      }
    }
    blurx_01 = in[4 * c0][1] + in[4 * c0+1][1] + in[4 * c0+2][1];
    if (N >= 4 * c0 + 2) {
      blurx_11 = in[4 * c0 + 1][1] + in[4 * c0 + 1 +1][1] + in[4 * c0 + 1 +2][1];
      if (N >= 4 * c0 + 3) {
        blurx_21 = in[4 * c0 + 2][1] + in[4 * c0 + 2 +1][1] + in[4 * c0 + 2 +2][1];
        if (N >= 4 * c0 + 4)
          blurx_31 = in[4 * c0 + 3][1] + in[4 * c0 + 3 +1][1] + in[4 * c0 + 3 +2][1];
      }
    }
    for (int c1 = 0; c1 <= min((N - 1) / 3, N - 3); c1 += 1) {
      if (c1 >= 1) {
        blurx_00 = in[4 * c0][3 * c1] + in[4 * c0+1][3 * c1] + in[4 * c0+2][3 * c1];
        out[4 * c0][3 * c1] = blurx_01 + blurx_02 + blurx_00;
        if (N >= 4 * c0 + 2) {
          blurx_10 = in[4 * c0 + 1][3 * c1] + in[4 * c0 + 1 +1][3 * c1] + in[4 * c0 + 1 +2][3 * c1];
          out[4 * c0 + 1][3 * c1] = blurx_11 + blurx_12 + blurx_10;
          if (N >= 4 * c0 + 3) {
            blurx_20 = in[4 * c0 + 2][3 * c1] + in[4 * c0 + 2 +1][3 * c1] + in[4 * c0 + 2 +2][3 * c1];
            out[4 * c0 + 2][3 * c1] = blurx_21 + blurx_22 + blurx_20;
            if (N >= 4 * c0 + 4) {
              blurx_30 = in[4 * c0 + 3][3 * c1] + in[4 * c0 + 3 +1][3 * c1] + in[4 * c0 + 3 +2][3 * c1];
              out[4 * c0 + 3][3 * c1] = blurx_31 + blurx_32 + blurx_30;
            }
          }
        }
        if (N >= 3 * c1 + 2) {
          blurx_01 = in[4 * c0][3 * c1 + 1] + in[4 * c0+1][3 * c1 + 1] + in[4 * c0+2][3 * c1 + 1];
          out[4 * c0][3 * c1 + 1] = blurx_02 + blurx_00 + blurx_01;
          if (N >= 4 * c0 + 2) {
            blurx_11 = in[4 * c0 + 1][3 * c1 + 1] + in[4 * c0 + 1 +1][3 * c1 + 1] + in[4 * c0 + 1 +2][3 * c1 + 1];
            out[4 * c0 + 1][3 * c1 + 1] = blurx_12 + blurx_10 + blurx_11;
            if (N >= 4 * c0 + 3) {
              blurx_21 = in[4 * c0 + 2][3 * c1 + 1] + in[4 * c0 + 2 +1][3 * c1 + 1] + in[4 * c0 + 2 +2][3 * c1 + 1];
              out[4 * c0 + 2][3 * c1 + 1] = blurx_22 + blurx_20 + blurx_21;
              if (N >= 4 * c0 + 4) {
                blurx_31 = in[4 * c0 + 3][3 * c1 + 1] + in[4 * c0 + 3 +1][3 * c1 + 1] + in[4 * c0 + 3 +2][3 * c1 + 1];
                out[4 * c0 + 3][3 * c1 + 1] = blurx_32 + blurx_30 + blurx_31;
              }
            }
          }
        }
      }
      if (N >= 3 * c1 + 3) {
        blurx_02 = in[4 * c0][3 * c1 + 2] + in[4 * c0+1][3 * c1 + 2] + in[4 * c0+2][3 * c1 + 2];
        out[4 * c0][3 * c1 + 2] = blurx_00 + blurx_01 + blurx_02;
        if (N >= 4 * c0 + 2) {
          blurx_12 = in[4 * c0 + 1][3 * c1 + 2] + in[4 * c0 + 1 +1][3 * c1 + 2] + in[4 * c0 + 1 +2][3 * c1 + 2];
          out[4 * c0 + 1][3 * c1 + 2] = blurx_10 + blurx_11 + blurx_12;
          if (N >= 4 * c0 + 3) {
            blurx_22 = in[4 * c0 + 2][3 * c1 + 2] + in[4 * c0 + 2 +1][3 * c1 + 2] + in[4 * c0 + 2 +2][3 * c1 + 2];
            out[4 * c0 + 2][3 * c1 + 2] = blurx_20 + blurx_21 + blurx_22;
            if (N >= 4 * c0 + 4) {
              blurx_32 = in[4 * c0 + 3][3 * c1 + 2] + in[4 * c0 + 3 +1][3 * c1 + 2] + in[4 * c0 + 3 +2][3 * c1 + 2];
              out[4 * c0 + 3][3 * c1 + 2] = blurx_30 + blurx_31 + blurx_32;
            }
          }
        }
      }
    }
  }
}
```

Algorithm 1: SCALARIZATION

Data: Program (P, θ), optional tiling ϕ
Result: Scalarized program (P_{out}, θ_{out})

```
begin
    From now, skip live-in and live-out arrays α σ ⟵ ARRAY_CONTRACTION(P, θ, ϕ)
    if σ has parametrized modulo then
        if no tiling is provided then
            ϕ ⟵ TILING(P, θ, σ)
            σ ⟵ ARRAY_CONTRACTION(P, θ, ϕ)
        end
        Skip arrays with parametrized modulo
        if No array remains then
            return (P, θ)
        end
    end
    𝒰 ⟵ UNROLL_FACTORS(P, θ, σ)
    (P_out, θ_out) ⟵ CODE_GENERATION(P, θ, ϕ, σ, 𝒰)
    return (P_out, θ_out)
end
```

Algorithm 2: TILING

Data: Program (P, θ), allocation σ
Result: Scalarization-aware tiling ϕ

begin
  foreach *reference* $S : \ldots a[u(\vec{i})] \ldots$ **do**
    Write $\sigma_a(\vec{c}) = A\vec{c} \mod s(\vec{N})$
    $\mathcal{P}_S \longleftarrow \emptyset$
    foreach k s.t. $s(\vec{N})[k]$ *is parametrized* **do**
      Add a basis of $\vec{\Delta}_k$ s.t. $A \circ u(\vec{\Delta}_k) = \vec{\delta}_k$:
      begin
        if u *is non-singular* **then**
          Add $\vec{\Delta}_k = u^{-1} \circ A^{-1}(\vec{\delta}_k)$ to $\mathcal{P}_S$
          continue
        end
        /* u is singular */
        Write $A \circ u(\vec{\Delta}_k) = Q\vec{\Delta}_k + \vec{r}$
        /* get a solution */
        $\vec{\Delta}_0 \longleftarrow Q^{-g}(\vec{\delta}_k - \vec{r})$
        /* add a solution basis */
        $\langle \vec{e}_1, \ldots, \vec{e}_p \rangle \longleftarrow \ker Q$
        Add each $\vec{e}_i + \vec{\Delta}_0$ to $\mathcal{P}_S$
      end
    end
  end
  $\phi \longleftarrow$ PLUTO_TILING(P)
  /* Keep hyperplanes on parametric directions */
  $\mathcal{L} \longleftarrow \emptyset$
  foreach *statement S* **do**
    Write $\phi_S(\vec{i}) = T\vec{i} + \vec{u}$
    foreach *line vector* $\vec{\ell}_j$ *of* T **do**
      if $\vec{\ell}_j \cdot \vec{\Delta} \neq 0$ *for some* $\Delta \in \mathcal{P}_S$ **then**
        Add j to $\mathcal{L}$
      end
    end
  end
  Keep only output dimensions $\mathcal{L}$ of ϕ
  return ϕ
end

Algorithm 3: UNROLL_FACTORS

Data: Program (P, θ)
Result: $\mathcal{U}$: time dimension $(\theta) \mapsto$ unroll factor

begin
  $\mathcal{U}(t_i) \longleftarrow 1$, for each time dimension t_i
  foreach *reference* $S : \ldots a[u(\vec{i})] \ldots$ **do**
    Write $\sigma_a(\vec{c}) = A\vec{c} \mod \vec{s}$
    `/* Unroll time dimensions (θ) */`
    Write $A \circ u \circ \theta_S^{-1}(\vec{t}) = (f_1(\vec{t}), \ldots, f_p(\vec{t}))$
    foreach *index dimension* $f_k(\vec{t})$ **do**
      foreach *variable* t_i *in* $f_k(\vec{t})$ **do**
        $\mathcal{U}(t_i) \longleftarrow \mathrm{lcm}(\mathcal{U}(t_i), \vec{s}_k)$
      end
    end
  end
  return $\mathcal{U}$
end

Algorithm 4: CODE_GENERATION

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

begin
  $\vec{U} \longleftarrow (\mathcal{U}(t_1), \ldots, \mathcal{U}(t_n))$
  foreach *statement* S **do**
    foreach $\vec{\pi} \in [\![0, \mathcal{U}(t_1)[\![\times \ldots \times [\![0, \mathcal{U}(t_n)[\![$ **do**
      $\mathcal{D}_{S,\vec{\pi}} \longleftarrow \{(\vec{T}, \vec{k}, \vec{i}) \mid \theta_S(\vec{i}) = \vec{k} \times \vec{U} + \vec{\pi} \wedge \text{tiling\_constraints}(\mathcal{D}_S, \phi_S, \vec{T}, \vec{i})\}$
      $\theta_{S,\vec{\pi}}(\vec{T}, \vec{k}, \vec{i}) \longleftarrow (\vec{T}, k_1, \pi_1, \ldots, k_n, \pi_n)$
      `/* final scalarization */`
      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 $\texttt{register\_a}_{\sigma_a \circ u \circ \theta_S^{-1}(\vec{\pi})}$
    end
  end
  Write P_{out} the collection domain :statement $\mathcal{D}_{S,\vec{\pi}} : S_{\vec{\pi}}$
  Write θ_{out} the collection of schedules $\theta_{s,\vec{\pi}}$
  return (P_{out}, θ_{out})
end