A Polyhedral Approach for Scalar Promotion

Alec Sadler, Christophe Alias, Hugo Thievenaz

A Polyhedral Approach for Scalar Promotion

Alec Sadler, Christophe Alias, Hugo Thievenaz

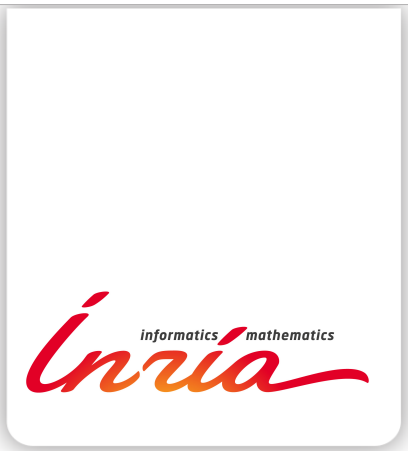
RESEARCH REPORT

N° 9437

November 2021

Project-Team Cash

ISSN 0249-6399 ISRN INRIA/RR--9437--FR+ENG



A Polyhedral Approach for Scalar Promotion

Alec Sadler*, Christophe Alias†, Hugo Thievenaz‡

Project-Team Cash

Research Report n° 9437 — November 2021 — 14 pages

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 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.

Key-words: Compilation, scalar promotion, automatic parallelization, polyhedral model

* Inria / ENS de Lyon
† Inria / ENS de Lyon
‡ Inria / ENS de Lyon

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Une approche polyédrique pour la promotion scalaire

Résumé : Les accès mémoires sont un goulot d'étranglement bien connu, dont l'impact peut être limité en utilisant correctement la hiérarchie mémoire jusqu'aux registres. Dans ce rapport, nous étudions la scalaration de tableaux, une technique pour transformer un tableau en une collection de variables scalaires à allouer à des registres. Nous revisitons la scalarisation de tableau à la lumière des avancées récentes du modèle polyédrique, un cadre général pour construire des transformations de programme. Nous proposons un algorithme pour la scalarisation de tableaux, directement intégrable comme passe d'un compilateur polyédrique. Notre algorithme de scalarisation opére sur la représentation intermédiaire polyédrique. En particulier, notre algorithme est paramétré par un ordonnancement possiblement calculé par une passe précédente. Nous utilisons la contraction de tableaux sous contrainte d'ordonnancemnet et nous proposons un nouvel algorithme de tuilage de boucles pour régler l'empreinte mémoire et ainsi utiliser le bon nombre de registres. Les résultats expérimentaux confirment l'intérêt et l'efficacité de notre approche.

Mots-clés : Compilation, promotion scalaire, parallélisation automatique, modèle polyédrique

A Polyhedral Approach for Scalar Promotion

Alec Sadler
Inria
Lyon, FRANCE
Alec.Sadler@inria.fr

Christophe Alias
Inria
Lyon, FRANCE
Christophe.Alias@inria.fr

Hugo Thievenaz
Inria
Lyon, FRANCE
Hugo.Thievenaz@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 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.**

***Index Terms*—compiler optimization, polyhedral model, array scalarization**

I. 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], [4], [9], [14] 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* [4], [9] 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* [5]–[8], [12], [13], 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* [1], [10] 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 II presents the required notions in polyhedral compilation. Section III presents related work Section IV describes our scalarization algorithm Section V presents our exprimental validation Finally, Section VI concludes this paper and draws research perpectives.

II. 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 recall the polyhedral approaches for array contraction.

A. Polyhedral model

The polyhedral model [5]–[8], [12], [13] 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.

B. Polyhedral transformations

a) 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.

b) 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.

c) Array contraction: Arrays might be remapped with an allocation (or contraction) 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 a vector of affine forms on program parameters [11] (with A the identity matrix), [1].

C. 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 and an optional *array contraction function* σ.

III. Related Work

This section goes over the notions presented by other works on register allocation, more precisely *scalar promotion*, *register tiling* and *array contraction*, which all closely relate to our paper.

a) 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. Callahan, Carr and Kennedy [3] show one approach to the implementation of this optimization, but the scope is on rectangular loop nests, with no conditionals. Their method is to search for potential candidates to promotion by analysing the dependencies, then promoting those candidates, and they schedule the iterations so as to reduce the resulting register pressure.

As the work of Jiménez et al. [9] describes, the main problem of scalar promotion for non-rectangular loop nests at the registry level, is that loops need to be fully unrolled for addresses to be exposed, as registers are addressed using absolute addresses. This unroll is non-trivial for non-rectangular loop nests.

b) 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...). The work of Jiménez et al. [9] 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. [14] 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, the work of Domagala et al. [4] 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.

c) Array contraction: The problem of *array contraction* consists in finding a storage function, that maps elements of an array to their storage location, such that the resulting storage requirement is minimized. Works such as [1], [10]

present array contraction methods that infer mappings such that multidimensional arrays on some benchmarks are truncated by a parameter factor. This problem has a strong bond with the register allocation problem, as we seek to fit tiles of parametrized size into a register file whose entry count is limited.

IV. Our Approach

This section presents our scalarization algorithm. First, section IV-A outlines our running example. Then, section IV describes our algorithm and provides a proof of correctness.

A. 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.

a) 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 $\sigma_{blurx}(x,y) = (x \mod N, y \mod 3)$, 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 $h = [14/3] = 4$. The final scalarized code is depicted in Figure 3.

b) 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 $\sigma_{blurx}(x,y) = (x \mod 1, y \mod 3)$ with a constant footprint 3. Hence scalarization might be applied directly, without the need to apply further loop tiling.

B. Our algorithm

We now present our main algorithm (Algorithm 1) and its subroutines (Algorithms 2, 3 and 4).

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_{out},θ_{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 IV-A, 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.

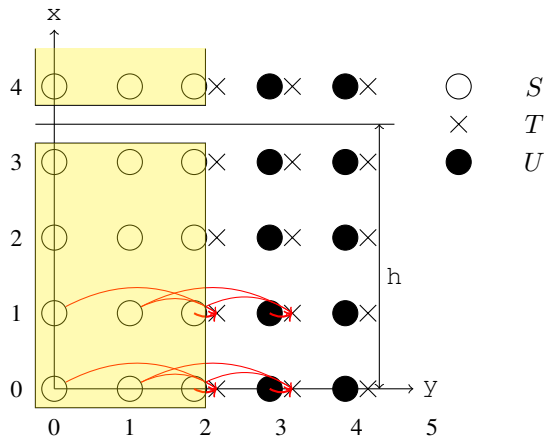
Algorithm 1: Scalarization

```
   Data: Program (P, θ), optional tiling φ
   Result: Scalarized program (P_out, θ_out)
1  begin
2      From now, skip live-in and live-out arrays a
       σ ⟵ Array_Contraction(P, θ, φ)
3      if σ has parametrized modulo then
4          if no tiling is provided then
5              φ ⟵ Tiling(P, θ, σ)
6              σ ⟵ Array_Contraction(P, θ, φ)
7          end
8          Skip arrays with parametrized modulo
9          if No array remains then
10             return (P, θ)
11         end
12     end
13     U ⟵ Unroll_Factors(P, θ, σ)
14     (P_out, θ_out) ⟵ Code_Generation(P, θ, φ, σ, U)
15     return (P_out, θ_out)
16 end
```

Finally, we scalarize the arrays with constant size. First, we

```
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

Fig. 1: Running example: 2D Blur filter

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.

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 $\sigma_{blurx}(x,y) = (x \mod N, y \mod 3)$, 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.

Running example. We would get $\mathcal{P}_S = \mathcal{P}_T = \mathcal{P}_U = \{(0,1)\}$, then the tiling $\phi_S = \phi_T = \phi_U = (y,x) \mapsto (x,y) = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} y \\ x \end{pmatrix}$. Finally we would keep the first hyperplane (first line) as $(0,1) \cdot (0,1) = 1 \neq 0$, and reject the second hyperplane (second line) as $(0,1) \cdot (1,0) = 0$. Hence the final tiling $\phi_S = \phi_T = \phi_U = (y,x) \mapsto (x)$, as depicted on Figure 1.(b). □

Once the program is properly tiled and some temporary arrays are reduced to a constant size thanks to array contraction, we apply our scalarization algorithm. First, we compute unroll factors on the loops abstractly described by the input schedule θ (Algorithm 3). Then, we apply the unrolling and we scalarized those temporary arrays (Algorithm 4).

Consider Algorithm 3. 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. We first rephrase array indices to use the loop counters prescribed by θ (time counters t_i). If $\sigma_a(\vec{c}) = A\vec{c} \mod \vec{s}$, the index $u(\vec{i})$ for the reference $a[\sigma_a(u(\vec{i}))]$ of statement S is rephrased $A \circ u \circ \theta_S^{-1}(\vec{t}) \mod \vec{s}$, since $\theta_S(\vec{i}) = \vec{t}$ (definition of θ_S). The remainder focuses on the affine part $A \circ u \circ \theta_S^{-1}(\vec{t})$.

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: $(u_1, \ldots, u_n) \times (v_1, \ldots, v_n) = (u_1 \times v_1, \ldots, u_n \times v_n)$.

Algorithm 2: TILING

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

1 **begin**
2 **foreach** *reference* $S : \dots a[u(\vec{i})] \dots$ **do**
3 Write $\sigma_a(\vec{c}) = A\vec{c} \mod s(\vec{N})$
4 $\mathcal{P}_S \longleftarrow \emptyset$
5 **foreach** k s.t. $s(\vec{N})[k]$ *is parametrized* **do**
6 Add a basis of $\vec{\Delta}_k$ s.t. $A \circ u(\vec{\Delta}_k) = \vec{\delta}_k$:
7 **begin**
8 **if** u *is non-singular* **then**
9 Add $\vec{\Delta}_k = u^{-1} \circ A^{-1}(\vec{\delta}_k)$ to $\mathcal{P}_S$
10 **continue**
11 **end**
 /* u is singular */
12 Write $A \circ u(\vec{\Delta}_k) = Q\vec{\Delta}_k + \vec{r}$
 /* get a solution */
13 $\vec{\Delta}_0 \longleftarrow Q^{-g}(\vec{\delta}_k - \vec{r})$
 /* add a solution basis */
14 $\langle \vec{e}_1, \dots, \vec{e}_p \rangle \longleftarrow \ker Q$
15 Add each $\vec{e}_i + \vec{\Delta}_0$ to $\mathcal{P}_S$
16 **end**
17 **end**
18 **end**
19 $\phi \longleftarrow$ PLUTO_TILING(P)
 /* Keep hyperplanes on parametric directions */
20 $\mathcal{L} \longleftarrow \emptyset$
21 **foreach** *statement* S **do**
22 Write $\phi_S(\vec{i}) = T\vec{i} + \vec{u}$
23 **foreach** *line vector* $\vec{\ell}_j$ *of* T **do**
24 **if** $\vec{\ell}_j \cdot \vec{\Delta} \neq 0$ *for some* $\Delta \in \mathcal{P}_S$ **then**
25 Add j to $\mathcal{L}$
26 **end**
27 **end**
28 **end**
29 Keep only output dimensions $\mathcal{L}$ of ϕ
30 **return** ϕ
31 **end**

Algorithm 3: UNROLL_FACTORS

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

1 **begin**
2 $\mathcal{U}(t_i) \longleftarrow 1$, for each time dimension t_i
3 **foreach** *reference* $S : \dots a[u(\vec{i})] \dots$ **do**
4 Write $\sigma_a(\vec{c}) = A\vec{c} \mod \vec{s}$
 /* Unroll time dimensions (θ) */
5 Write $A \circ u \circ \theta_S^{-1}(\vec{t}) = (f_1(\vec{t}), \dots, f_p(\vec{t}))$
6 **foreach** *index dimension* $f_k(\vec{t})$ **do**
7 **foreach** *variable* t_i *in* $f_k(\vec{t})$ **do**
8 $\mathcal{U}(t_i) \longleftarrow \mathrm{lcm}(\mathcal{U}(t_i), \vec{s}_k)$
9 **end**
10 **end**
11 **end**
12 **return** $\mathcal{U}$
13 **end**

Lemma 4.1: Let $\vec{U} = (\mathcal{U}(t_1), \dots, \mathcal{U}(t_n))$ and $S : \dots a[u(\vec{i})] \dots$ 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 3. The k-th dimension of $\sigma_a \circ u \circ \theta_S^{-1}(\vec{k} \times \vec{U})$ may be written: $(\alpha_k(\vec{k} \times \vec{U}) + \beta_k) \mod s_k$, which develops to: $(\alpha_k(\vec{k} \times \vec{U}) \mod s_k + \beta_k \mod s_k) \mod s_k$ (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 $\alpha_k(\vec{k} \times \vec{U}) \mod s_k = 0$. Therefore, the k-th dimension of $\sigma_a \circ u \circ \theta_S^{-1}(\vec{k} \times \vec{U})$ is the constant $\vec{c}_{0_k} = \beta_k \mod s_k$. ■

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 $\theta_S(y, x) = (0, y, x, 0)$, $\theta_T(y, x) = (1, y, x, 0)$, $\theta_U(y, x) = (1, y, x, 1)$. Hence $\theta_S^{-1} = \theta_T^{-1} = \theta_U^{-1} = (t_1, t_2, t_3, t_4) \mapsto (t_2, t_3)$. After tiling, we have $\sigma_{blurx}(x, y) = (x \mod 4, y \mod 3)$ (Algorithm 1, step 6), 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): $blurx[x][y] \mapsto blurx[t_3][t_2]$, $blurx[x][y-2] \mapsto blurx[t_3][t_2 - 2]$, $blurx[x][y-1] \mapsto blurx[t_3][t_2 - 1]$. Finally, we obtain: $\mathcal{U} : t_1 \mapsto 1, t_2 \mapsto 3, t_3 \mapsto 4, t_4 \mapsto 1$. □

Finally, Algorithm 4 generates the polyhedral representation of the scalarized program. 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 [2]: $\vec{i} \in \mathcal{D}_S \wedge \vec{T} = \phi_S(\vec{i})/\vec{S}$, where $/$ denotes the element-wise euclidian division and $\vec{S}$ is the tile size along each hyperplane.

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

Theorem 4.2: 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 4.

Proof. The tiling found by Algorithm 2, step 19 is correct [2]. Then any subset of hyperplanes, as the result ϕ, is a correct tiling, so is the schedule $(\vec{T}, \vec{i}) \mapsto (\vec{T}, \theta_S(\vec{i}))$. Since $\theta_S(\vec{i}) = \vec{k} \times \vec{U} + \vec{\pi}$ (element-wise euclidian division), the lexicographic order of $(k_1, \pi_1, \dots, k_n, \pi_n)$ is the same as $\theta_S(\vec{i})$. Hence the correctness of $\theta_{S,\vec{\pi}} : (\vec{T}, \vec{k}, \vec{i}) \mapsto (\vec{T}, k_1, \pi_1, \dots, k_n, \pi_n)$ ■

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

Algorithm 4: 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**

Lemma 4.3: 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 $\left(\vec{c}_0 + \lin\left(A \circ u \circ \theta_S^{-1}(\vec{\pi})\right)\right) \mod \vec{s}$ 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, we have: $\sigma_a \circ u \circ \theta_S^{-1}(\vec{k} \times \vec{U} + \vec{\pi})$ equals to $\left(A \circ u \circ \theta_S^{-1}(\vec{k} \times \vec{U}) + \lin\left(A \circ u \circ \theta_S^{-1}\right)(\vec{\pi})\right) \mod \vec{s}$ (affine decomposition).
By distributing the modulo, this is equals to $\left(\sigma_a \circ u \circ \theta_S^{-1}(\vec{k} \times \vec{U}) + \lin\left(A \circ u \circ \theta_S^{-1}\right)(\vec{\pi}) \mod \vec{s}\right)$ $\mod \vec{s}$, which simplifies (Lemma 4.1) to: $\left(\vec{c}_0 + \lin\left(A \circ u \circ \theta_S^{-1}\right)(\vec{\pi})\right) \mod \vec{s}$ which depends only on $\vec{\pi}$. ■

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

Running example (cont'd). For each statement S, T, U, we enumerate all the values of $\vec{\pi} \in \{0\} \times [\![0, 2]\!] \times [\![0, 3]\!] \times \{0\}$. For instance, for S and the first combination $\vec{\pi} = (0, 0, 0, 0)$, we generate:

- $\mathcal{D}_{S,(0,0,0,0)} = \{(T_1, k_1, k_2, k_3, k_4, y, x) \mid \theta_S(y, x) = (0, y, x, 0) = (k_1.1 + 0, k_2.2 + 0, k_3.3 + 0, k_4.1 + 0) \wedge 0 \leq x < N \wedge 0 \leq y < 2 \wedge 4T_1 \leq x < 4(T_1 + 1)\}$
- $\theta_{S,(0,0,0,0)}(T_1, k_1, k_2, k_3, k_4, y, x) = (T_1, k_1, 0, k_2, 0, k_3, 0, k_4, 0)$
- $S_{(0,0,0,0)} : register\_blurx_{(0,0)} = in[x][y] + in[x+1][y] + in[x+2][y]$;

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, as depicted in Figure 3. Note that each register variable $register\_blurx\_*$ as been renamed $blurx\_*$ instead, for the sake of clarty. □

V. EXPERIMENTAL RESULTS

This section presents our experimental results on several polyhedral programs, ranging from linear algebra to signal processing kernels.

A. 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 picture of size N.
- **fibonacci.** This kernel generates the N first fibonacci numbers, before returning the last one generated.
- **pc-2d-interleaved.** Producer/consumer throught a 2D array $N \times N$, where the consumer executes exactly 2 iterations after the producer.
- **pc-1d.** Producer/consumer through an array of size N. Values are consumed 2 iterations after their production.
- **pc-2d.** This kernel applies a stencil pattern on a 2D array $N \times N$, with dependence vectors $(1, 0)$ and $(0, 1)$.
- **cnn.** Simple CNN with a convolutive layer followed a ReLU layer. The layers are of size $N \times N$.
- **2mm.** Multiplication of three matrices $N \times N$ 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 IV-A. 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.

B. 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.

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.

VI. 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.

References

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

[2] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 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*, pages 101–113, 2008.

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

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

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

[6] Paul Feautrier. Some efficient solutions to the affine scheduling problem. Part I. one-dimensional time. *International Journal of Parallel Programming*, 21(5):313–348, October 1992.

[7] Paul Feautrier. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, December 1992.

[8] Paul Feautrier and Christian Lengauer. Polyhedron model. In *Encyclopedia of Parallel Computing*, pages 1581–1592. 2011.

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

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

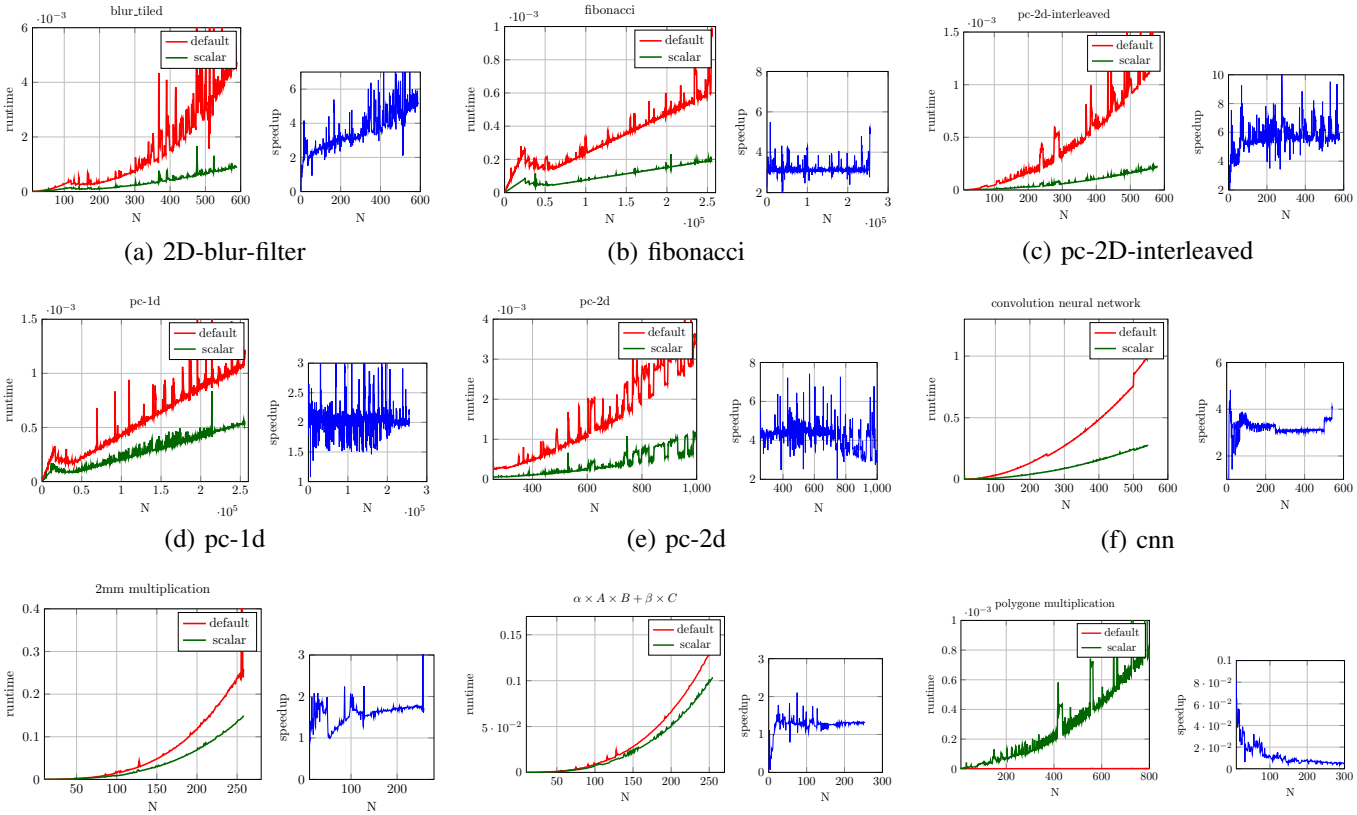
[11] Vincent Lefebvre and Paul Feautrier. Automatic storage management for parallel programs. *Parallel Computing*, 24:649–671, 1998.

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

[13] Sanjay V. Rajopadhye, S. Purushothaman, and Richard M. Fujimoto. On synthesizing systolic arrays from recurrence equations with linear dependencies. In Kesav V. Nori, editor, *Foundations of Software Technology and Theoretical Computer Science*, volume 241 of *Lecture Notes in Computer Science*, pages 488–503. Springer Berlin Heidelberg, 1986.

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

[15] Sven Verdoolaege. Counting affine calculator and applications. In *IMPACT*, 2011.



(a) 2D-blur-filter (b) fibonacci (c) pc-2D-interleaved

(d) pc-1d (e) pc-2d (f) cnn

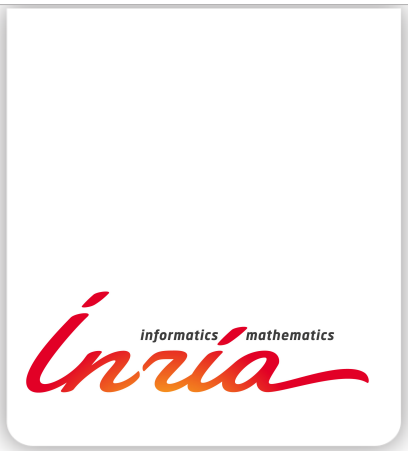
(g) 2mm (h) gemm (i) poly

Fig. 2: Experimental results

```
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;
            }
          }
        }
      }
    }
  }
}
```

Fig. 3: 2D blur filter, scenario 1, code generated



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399