# A general compilation algorithm to parallelize and optimize counted loops with dynamic data-dependent bounds

Jie Zhao, Albert Cohen

# A general compilation algorithm to parallelize and optimize counted loops with dynamic data-dependent bounds

Jie Zhao Albert Cohen
INRIA & DI, École Normale Supérieure
45 rue d'Ulm, 75005 Paris
firstname.lastname@inria.fr

## ABSTRACT

We study the parallelizing compilation and loop nest optimization of an important class of programs where counted loops have a dynamically computed, data-dependent upper bound. Such loops are amenable to a wider set of transformations than general `while` loops with inductively defined termination conditions: for example, the substitution of closed forms for induction variables remains applicable, removing the data dependences induced by termination conditions. We propose an automatic compilation approach to parallelize and optimize dynamic counted loops. Our approach relies on affine relations only, as implemented in state-of-the-art polyhedral libraries. Revisiting a state-of-the-art framework to parallelize arbitrary `while` loops, we introduce additional control dependences on data-dependent predicates. Our method goes beyond the state of the art in fully automating the process, specializing the code generation algorithm to the case of dynamic counted loops and avoiding the introduction of spurious loop-carried dependences. We conduct experiments on representative irregular computations, from computer vision and finite element methods to sparse matrix linear algebra. We validate that the method is applicable to general affine transformations for locality optimization, vectorization and parallelization.

## Keywords

parallelizing compiler; loop nest optimization; polyhedral model; dynamic counted loop

## 1. INTRODUCTION

While a large number of computationally intensive applications spend most of their time in static control loop nests—with affine conditional expressions and array subscripts, several important algorithms do not meet such statically predictable requirements. We are interested in the class of loop nest kernels involving *dynamic counted loops*. These are regular counted loops with numerical constant strides, iterating until a dynamically computed, data-dependent upper bound. Such bounds are loop invariants, but often recomputed in the immediate vicinity of the loop they control; for example, their definition may take place in the immediately enclosing loop. Dynamic counted loops play an increasing important role in numerical solvers, media processing applications, and data analytics, as we will see in the experimental evaluation. They can be seen as a special case of `while` loop that does not involve an arbitrary, inductively defined termination condition. The ability to substitute their counter with a closed form—an affine induction variable—makes them amenable to a wider set of transformations than `while` loops. Dynamic counted loops are commonly found in sparse matrix computations, but not restricted to this class of algorithms. They are also commonly found in conjunction with statically unpredictable, non-affine array subscripts. It is important to address the performance challenge of such loops on modern architectures.

A significant amount of work aims at the parallelization of `while` loops, including [10, 9, 5, 14, 15, 16, 17] to cite polyhedral methods only. These techniques face a painful problem, solved in the better behaved case of static control loops for more than a decade: the lack of a robust, general-purpose algorithm and tool to generate imperative code after the application of an affine transformation. The state of the art approach to model `while` loops in a polyhedral framework, and in the code generator in particular, is the work of Benabderrahmane et al. [5].

This work [5] uses over-approximations to translate a `while` loop into a static control loop iterating from 0 to infinity that can be represented and optimized in the polyhedral model, and introduces exit predicates and the associated data dependences to preserve the computation of the original termination condition, and to enforce the proper termination of the generated loops the first time this condition is true. These data dependences severely restrict the application of loop transformations involving a `while` loop, since reordering of the iterations of the latter is not permitted, and loop interchange involves the removal of memory-based dependences on the exit predicates. The framework was also not fully automated at the time of its publication, leaving much room for the interpretation of its applicable cases and the space of legal transformations it effectively models. A significant effort remains to be invested in the completion of a fully operational `while` loop polyhedral framework, even a strictly conservative one (i.e., non-speculative). In this paper, we take a more pragmatic, short term direction: we

IMPACT 2017
Seventh International Workshop on Polyhedral Compilation Techniques
Jan 23, 2017, Stockholm, Sweden
In conjunction with HiPEAC 2017.
http://impact.gforge.inria.fr/impact2017

focus on the special case of dynamically counted loops where the most difficult of these problems do not occur.

There has also been a significant body of research specializing for high-performance implementations of spare matrix computations. Manually-tuned libraries [2, 4, 7, 20, 21, 27] are a commonly used approach, but it is obviously impossible to port them to each representation and various architectures. So a polyhedral framework that can handle non-affine subscripts [25] has a greater potential to to achieve transformations and optimizations on sparse matrix computations.

In this paper, we propose an automatic polyhedral compilation approach to parallelize and optimize dynamic counted loops that can express arbitrary affine transformations and achieve architecture portability. Our approach is based on the systems of affine inequalities, as implemented in state-of-the-art polyhedral libraries. Just like [23, 24], it does not resort to more expressive first-order logic with non-interpreted functions/predicates such as the advanced analyses and code generation techniques of Wonnacott et al. [28], and it does not involve speculative execution either.

Revisiting a state-of-the-art framework [5] originally designed for the polyhedral compilation of arbitrary `while` loops, we introduce exit predicates for dynamic counted loops and model the control dependence of the original loop through additional data dependences from the definition of these exit predicates to the statements in the loop body. To model fixed-size data parallelism of vector instructions and hardware accelerators such as GPUs, one needs to extend this baseline framework with an additional static analysis to compute an affine upper bound of all dynamic bounds of a given loop. And finally, to enable the generation of imperative code after the application of affine transformations, we propose code generation templates capturing the definition of the non-affine exit predicates and implementing the associated imperative control flow, with applications to shared memory multiprocessors as well as GPU accelerators.

Our method goes beyond the state of the art [5, 25] in fully automating the process, specializing the code generation algorithm to the case of dynamic counted loops and avoiding the introduction of spurious loop-carried dependences. We conduct experiments on representative irregular computations, including computer vision, finite element methods, and sparse matrix linear algebra. We validate that the method is applicable to general affine transformations for locality optimization, vectorization and parallelization.

The paper is organized as follows. We describe the background of the polyhedral model and state our motivation to parallelize dynamic counted loops in the next section. Section 3 discusses how we introduce control dependences on data-dependent predicates, and Section 4 introduces the code generation templates for different architectures. Experimental results are shown in Section 5, followed by a discussion of related work in Section 6 and concluding remarks.

## 2. BACKGROUND AND MOTIVATION

The polyhedral compilation framework is a powerful formalism to express parallelizing loop transformations and loop nest optimizations [12, 13, 6, 3]. Its application domain was traditionally constrained to static-control, regular loop nests, but the polyhedral community has been working and made progress on extending its application to more complex loops with dynamic, data-dependent behavior. The polyhedral framework abstracts computational regions of the control flow as sets and relations over statement instances. It represents a program and its semantical properties using iteration domains, access relations, dependences and schedule components. The statement instances are included in iteration domains and are mapped to the accessed array elements by access relations. Dependences relate statement instances depending on each other, while a schedule defines a partial execution order on these statement instances.

```
S0:  n = f(1);
     for (i=0; i<100; i++) {
S1:    m = g(i);
       for (j=0; j<m; j++)
         for (k=0; k<n; k++)
S2:        S(i, j, k);
S3:    n = f(i+1);
     }
```

Figure 1: An example extracted from HOG

Consider the code in Figure 1 extracted from the Histogram of Oriented Gradients (HOG) descriptor algorithm. A typical application of HOG is to detect human figures by counting occurrences of a specific gradient orientation in localized portions of an image. The upper bounds, $m$ and $n$, of the $j$-loop and $k$-loop are computed in their common enclosing loop and updated as the $i$-loop iterates. As $m$ and $n$ are updated dynamically in the $i$-loop, it is not possible to classify the whole loop nest as a static control part (SCoP), and traditional polyhedral techniques do not directly apply. And tools aiming at a greater coverage of benchmarks—such as PPCG or LLVM/Polly—will abstract the offending inner loops into a black box, greatly limiting the potential for locality-enhancing and parallelizing optimizations. Note that the lower bound of a dynamic counted loop may not be statically known either, a typical scenario of sparse matrix computations; but as a counted loop, it may be normalized to count from 0, subtracting the lower bound from its upper bound.

Statement $S_2$ does not have a data dependence on other statements. However, there are output dependences among definition statements of dynamic parameters $m$ and $n$. To faithfully capture the scheduling constraints on this example, one should also model the control dependences of $S_2$ over both headers of the enclosing dynamically counted loops. Such control dependences can be represented as data dependences between the definition statements of dynamic upper bounds and $S_2$. To establish such a dependence relation, an exit predicate may be introduced before the loop body, like in the framework of Benabderrahmane et al. [5]. The resulting dependence graph is shown in Figure 2. The solid arrows are the original dependences between definition statements of dynamic parameters, and the dashed arrows represent the introduced control-flow dependences.

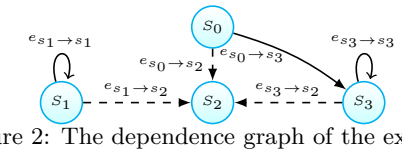


Figure 2: The dependence graph of the example

By capturing control dependences as affine relations from exit predicates to dominated statements in loop bodies, one may build a sound abstraction of the scheduling constraints for the loop nest. This technique is applicable to arbitrary

while loops, in conjunction with a suitable code generation strategy to recover the exact control flow protected by the exit predicate, and by over-approximating the loop upper bound as $+\infty$. This is the approach explored by Benabderrahmane et al., but the resulting polyhedral representation is plagued by additional spurious loop-carried dependences to update the exit predicate, removing many useful loop nest transformations from the affine scheduling space. In the more restricted context of dynamic counted loops, it is possible to eliminate those loop-carried dependences as the exit predicate only depends on loop-invariant data.

We base our formalism and experiments on the schedule tree representation [18]. A schedule tree typically comprises a domain node describing the overall extent of the statement instances, context nodes introducing symbolic parameters and constraints on those, sequence and set nodes expressing ordered or unordered branches, respectively, filter nodes selecting a subset of the statement instances as the children of a sequence or set node, and band nodes defining a partial schedule as well as permutability and/or parallelism properties on a group of statements. Band nodes are derived from tilable bands in the Pluto framework [6].

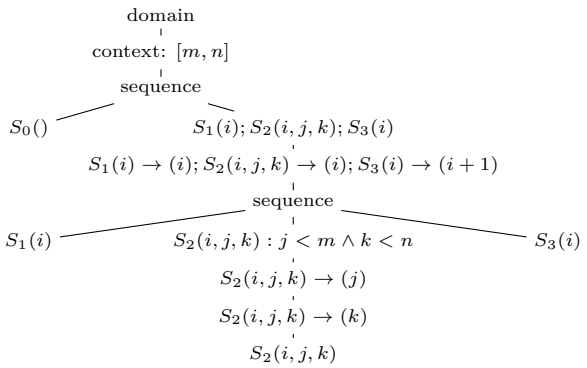


Figure 3: The schedule tree of the example

Figure 3 is the schedule tree of the example in Figure 1. A schedule tree has the same expressiveness as any affine schedule representation, but it facilitates local schedule manipulations and offers a systematic way to associate non-polyhedral semantical extensions. The domain node in Figure 3 is

$$\begin{aligned} Domain = \{S_0(); S_1(i) \mid 0 \le i < 100; S_2(i, j, k) \mid \\ 0 \le i < 100 \wedge 0 \le j \wedge 0 \le k; S_3(i) \mid 0 \le i < 100\} \end{aligned} \quad (1)$$

We will leverage this extensibility to represent non-affine loop bounds. Since $m$ and $n$ are defined in the $i$-loop, the loops they control cannot be appropriately represented in the iteration domain. Interestingly, a schedule tree allows to introduce constraints on symbolic parameters deeper than the context node introducing those parameters. These constraints, captured through filter nodes, will only apply in the branch controlled by the filter node, even if the context node introducing the parameter itself is located higher up in the tree. We may thus insert a context node right underneath the domain node, as it is recommended in the existing schedule tree implementation, but capturing constraints induced by $m$ and $n$ in the filter node of statement $S_2$. This filter node effectively splits the three-dimensional schedule into two nested bands. The resulting schedule tree may indeed be seen as a one-dimensional external domain and schedule enclosing a two-dimensional inner domain and schedule controlled by two additional parameters. In such a schedule tree, the domain and access relations of statement $S_2$ can be represented exactly and parametrically in $m$ and $n$. This representation can be used to compute the dependence relation of the whole schedule tree.

Based on this dependence information, one may derive a new schedule using the Pluto algorithm or one of its variants [6, 26], to optimize locality and extract parallelism. The final step is to generate code from the schedule tree to a high level program. The generation of the abstract syntax tree (AST) follows the approach implemented in isl [18], traversing the schedule tree and specializing the code generation algorithm to integrate the specific control flow template corresponding to the converted dynamic counted loops. Before encountering a filter node associated with a dynamic counted loop, the exit predicate and its controlled loop body is seen as a single black-box statement by the AST generation algorithm. When passing the filter node constraining the dynamic upper bound, it is necessary to complement the standard code generation procedure with a dedicated "dynamic counted loop template". This template involves the reconstruction of the exit predicate and the introduction of an early exit (`break`) instruction guarded by the predicate. Our algorithm generates code in one single traversal of the schedule tree, avoiding the need to call the AST generator multiple times for dynamically counted loops. This is another difference compared with Benabderrahmane's framework.

# 3. PROGRAM ANALYSIS

Dynamic counted loops arise frequently in irregular applications, but they sometimes are not written in a form that can be handled with our technique. We need a preprocessing step to make them amenable to our approach.

## 3.1 Preparation

The first category of dynamic counted loops is as shown in the example of Figure 1. It is referred to as the normalized format.

Sparse matrix computations constitute a second category of dynamic counted loops. They are a class of computations using compressed data layout to store nonzero elements of a matrix. Loops iterating on the compressed layout usually have a dynamic upper bound as well as a dynamic lower bound. However, these loops can be easily transformed into the normalized format a non-affine subtraction of the lower bound from the upper bound. Note that this transformation may introduce non-affine array subscripts; we assume the dependence analysis will conservatively handle such subscripts, or symbolically eliminate identical non-affine expressions on the left and right-hand side, or benefit from PENCIL annotations to refine its precision [8, 1].

Some forms of **while** loops may also be modeled, as long as an affine induction variable can be identified and assuming the variant part of the exit condition reduces to this induction variable.

## 3.2 Modeling Control Dependences

To make dynamic counted loops amenable to a polyhedral representation, we introduce additional dependences associated with exit predicates and their definition statements.

An exit predicate definition and check is inserted at the

beginning of each iteration of a dynamically counted loop. At code generation time, all statements in the body of the counted loop will have to be dominated by `break` instruction conditioned by the exit predicate. This follows the state of the art method for `while` loops [5], but without the inductive computation and loop-carried dependence on the exit predicate. Of course, we delay the introduction of `break` instructions until code generation, to keep the control flow in a statically manageable form for a polyhedral compiler. As a very simple illustrative example, the code in Figure 4(a) is preprocessed as the version in Figure 4(b) before constructing the affine representation.

```
for (j=0; j<m; j++)
  for (k=0; k<n; k++)
    S(j, k);
```

(a) Dynamic counted loops

```
for (j=0; 1; j++)
  for (k=0; 1; k++)
    if (j<m && k<n)
      S(j, k);
```

(b) if conditional

Figure 4: Conditional abstraction of dynamic counted loops

Each statement in a dynamically counted loop is associated with a list of exit predicates, the total number of which is equal to that of dynamic upper bounds. These predicates should be attached to the iteration domain of their predicated statements. As only one domain node (the root node) is allowed in the schedule tree implementation but the upper bounds are defined deeper in the nest, the constraints on the dynamic parameters and the expression of the dynamic upper bounds pose a real challenge. Still, although the upper bounds of inner counted loops vary dynamically in the outer loops, they can be viewed as static unknown loop bounds for inner loops. We may thus introduce these constraints and dynamic loop bounds by collecting all dynamic upper bounds as pseudo parameters in the context node—right underneath the domain node in a schedule tree—and also attaching the constraints introduced by the dynamic loop bounds (predicates) to the filters corresponding to the enclosed statements. This parameterization follows a similar approach to the one of fuzzy array dataflow analysis (FADA) [8], effectively decoupling the expression of parameter properties from the expression of the iteration domains.

## 3.3 Schedule Generation

To make our method applicable to general affine loop transformations, we apply a variant of the Pluto algorithm adapted to schedule trees. Let us consider again the example in Figure 1 and its schedule tree in Figure 3 for illustration purpose. We only discuss the modeling of additional dependences resulting from exit predicates.

The dynamic parameters are assigned at their definition statements, and then virtually read by statement $S_2$ implicitly guarded by the negation of the exit predicates. This can be modeled as read and write (affine) access relations:

$$\begin{aligned} R = [m,n] \to \{S_2(i,j,k) \to m[] : 0 \le i < 100 \\ \wedge 0 \le j < m \wedge 0 \le k < n; S_2(i,j,k) \to n[] : \\ 0 \le i < 100 \wedge 0 \le j < m \wedge 0 \le k < n\} \end{aligned} \tag{2}$$

and its write access relation is

$$\begin{aligned} W = [m,n] \to \{S_0() \to n[]; S_1(i) \to m[] : \\ 0 \le i < 100; S_3(i) \to n[] : 0 \le i < 100\} \end{aligned} \tag{3}$$

According to the variant of the Pluto algorithm implemented in isl, one may set the validity dependences, associated with semantics preservation, to

$$\begin{aligned} Validity = (R^{-1} \circ W' + W^{-1} \circ R' + W^{-1} \circ W') \\ \cap (Schedule \prec Schedule') \end{aligned} \tag{4}$$

and the proximity dependences, associated with locality enhancement, to

$$\begin{aligned} Proximity = (R^{-1} \circ W' + W^{-1} \circ R' + W^{-1} \circ W' \\ + R^{-1} \circ R') \cap (Schedule \prec Schedule') \end{aligned} \tag{5}$$

where *Schedule* represents the original schedule constructed from the original code according to the procedure above, and the $'$ (primed) maps distinguish iterations in dependence.

We can then compute a new schedule that applies the Pluto algorithm using

$$\begin{aligned} New\_Schedule = \text{schedule } Domain \text{ under } Schedule \\ \text{respecting } Validity \text{ and minimizing } Proximity \end{aligned} \tag{6}$$

In other words, the Pluto algorithm may safely compute a new schedule, starting from the original one, preserving all dependences and attempting to minimize the reuse distance. It may operate on the locally constrained parameters $m$ and $n$ as if they were global loop invariants, with the additional constraint that no affine set or relation operation ever combines statement instances across different instances of the filter node introducing the constraints on these parameters. Such instances would indeed belong to different iterations of the outer loop, where $m$ and $n$ take different values. By construction, such affine operations across filter nodes do not occur on schedule trees (e.g., when computing dependences or affine schedules). Besides, the ability to reconstruct the dynamically varying value of $m$ and $n$ will be recovered in the later code generation phase.

# 4. CODE GENERATION

Once a new schedule is produced, additional transformations can be applied on band nodes, to implement loop tiling or additional permutations, strip-mining for vectorization, etc. And eventually, one needs to return to imperative code through an AST generation algorithm.

As the final program effectiveness highly depends on the target code quality, AST generation is a critical step in the polyhedral framework. We thus construct a specialized code generation scheme on top of Grosser et al.'s algorithm [18], which extends the Quilleré et al. algorithm [22] and its CLooG improvement and implementation [3]. While CLooG recursively scans and generates the AST by maintaining a list of polyhedra from the outermost to the innermost loops, it does not offer much opportunity to specialize the generated control flow according to specific properties of local branches of the schedule. This complicated Benabderrahmane's approach for general `while` loops.

On the other hand, Grosser et al.'s method, implemented in isl, takes advantage of the schedule tree structure to implement such specialization along the filters nodes introducing constraints on the parameters associated with dynamically evaluated exit predicates. It performs a depth-first traversal of the schedule tree, applying a customizable form of Quilleré's algorithm on each band node. Since special care is needed to handle exit predicates and their impact on the dynamic upper bounds, the Grosser et al. algorithm is not able in its original form to generate semantically correct code for our extended schedule tree. However, it is

easy to modify it to handle the special case of exit predicates being homogeneous over all statements in a sequence or set node of the schedule tree (e.g., all statements in a band of permutable loops). Indeed, it is possible to hook the generation of custom templates for the computation of exit predicates and the insertion of early exit statements directly into the processing of filter nodes associated with dynamically updated parameters. This is facilitated by the ability to attach syntactic annotations about exit predicates using a so-called mark node of the schedule tree. A mark node is designed to attach any kind of information to a subtree. While the symbolic parameters are introduced by a context node immediately below the domain node, a mark node further down the tree can track the nesting level where the symbolic constant should be treated in a specific way, streamlining the code generation of loop nests with multiple dynamic counted loops. In particular, the AST generator can read information from a mark node to handle the generation of exit predicates.

In the following, we thus restrict ourselves to the case of homogeneous exit predicates over statements belonging to a sequence or set nodes in the schedule tree. This practically limits the applicability of the code generator to implement loop fusion across static and dynamically counted loops, but still enables tiling and unimodular transformations (interchange, skewing, reversal). The extension to loop fusion requires additional work in the separation step of the AST generation algorithm, similar to Benabderrahmane's technique, and its complete automation on schedule trees is left for future work.

## 4.1 A Static Upper Bound

A general `while` loop may be converted to an unbounded `for` loop. Yet dynamic counted loops do have an upper bound that varies dynamically. The approach to parallelize arbitrary `while` loops would waste this valuable information, introducing additional constraints on the ability to permute or distribute loops, or to move statement instances after a dynamically counted loop. To facilitate the implementation of such transformations, following Benabderrahmane's work [5], one may optionally derive a static upper bound $u$ that is greater than or equal to all bounds reached by a given dynamically counted loop.

The $u$ parameter can also be approximated statically, as the dynamic upper bounds are functions of outer enclosing loop variables: a typical solution relies on Fourier-Motzkin elimination, projecting out enclosing dimensions and eliminating non-affine constraints. The $u$ parameter can also be determined in other ways, from data structure properties or additional user-defined predicates in PENCIL [1]. For example, in sparse matrix computations, it may be computed by inspecting the maximum number of non-zero entries in a row in compressed-sparse-row (CSR) format. In many cases, a symbolic expression for the $u$ parameter can also be computed automatically, but we did not yet automate this analysis.

## 4.2 Template for Nested Bands

As there is a definition statement of the dynamic upper bound between the outer loops and a nested dynamic counted loop, and since we introduce additional dependences between this definition statement and the statements nested into dynamic counted loops, the canonically constructed schedule tree isolates two nested band nodes to represent different levels of the loop nest. Let us show a code generation template that can be applied to the customization of the control flow for each one of these bands.

Generating arbitrary conditionals for a single dynamic counted loop is straightforward, since the predicate attached in a mark node can be extracted easily. As a result, the AST generator only need to generate a conditional around the loop statements, as well as an early exit statement in the else branch.

For cases with multiple dynamic counted loops, we propose to systematically generate one conditional at the innermost level. Figure 4(b) shows an example illustrating this scenario. The predicates of both loops are included in a single conditional, and generated under the inner loop. This scheme can bring opportunities for affine loop transformations, such as loop interchange, not expressible in Benabderrahmane's framework due to the presence of spurious loop-carried dependences.

When the Grosser et al. algorithm traverses the band nodes in a schedule tree, it projects out the local schedule constraints from the domain node. As the symbol constants and their constraints would not appear in the domain node, the generated loops will iterate from 0 to $u$. It is thus necessary to emit an early exit statement, or many empty iterations will be wasted. This is straightforward in the case of a single dynamic counted loop, but in nested cases, as there will be only one conditional capturing multiple predicates, we need extract the parameters one by one from the predicate list and to generate the corresponding exit statements from the innermost outwards. The exit predicates are generated in the form of multiple conditionals rather than else branches. Figure 5 shows the result on the example of Figure 4(b).

```
for (j=0; j<u1; j++) {
  for (k=0; k<u2; k++) {
    if (j<m && k<n)
      S(j, k);
    if (k>=n)
      break;
  }
  if (j>=m)
    break;
}
```

Figure 5: Generation of exit predicates

The templates serve as a post-processing step, after code generation. A break statement with the appropriate guard is always inserted in a dynamically counted loop of the generated code. As an illustrative example, Figure 6(b) shows the result of applying loop interchange on the example of Figure 6(a). Unlike Jimborean et al.'s work [19] that handles general while loops and needs to speculate on the number of iterations, our technique always executes the same number of iterations as the original dynamic counted loops.

Loop tiling is a special case that should be taken into account. Before the generation of exit predicates, one may apply additional affine loop transformations on the schedule tree if it is possible. The typical candidate is loop tiling, which involve the insertion of an additional schedule dimension through strip-mining. When strip-mining a dynamic counted loop, there should be an exit statement at both levels. For the point loop (iterating within a tile), the common case above applies. For the tile loop (iterating among tiles),

```
for (i=0; i<N; i++) {
  m = f(i);
  for (j=0; j<m; j++)
    S(i,j);
}
```

(a) Dynamic counted loops

```
for (j=0; j<u; j++) {
  for (i=0; i<N; i++) {
    m = f(i);
    if (j<m)
      S(i,j);
  }
  if (j>=m)
    break;
}
```

(b) if conditional

Figure 6: Conditional abstraction of dynamic counted loops

we align its bounds and strides to follow the structure of the inner loop, so that its counter can also be compared systematically with the same bound. Figure 7 shows an example after the application of loop tiling on the code in Figure 5.

```
for (jj=0; jj<u1; jj++) {
  for (kk=0; kk<u2; kk++) {
    for (j=jj*BB; j<min(u1, jj*BB+BB); j++) {
      for (k=kk*CC; k<min(u2, kk*CC+CC); k++) {
        if (j<m && k<n)
          S(j, k);
        if (k>=n)
          break;
      }
      if (j>=m)
        break;
    }
    if (kk*CC>=n)
      break;
  }
  if (jj*BB>=m)
    break;
}
```

Figure 7: Generation of exit predicates in the tiling case

### 4.3 Template for Combined Parallel Bands

When the target architectures are shared memory multiprocessors, the code generation template for separate bands described above is fully applicable. However, when target GPU accelerators or producing fix-length vector code, we usually expect to combine nested bands to express parallelism at multiple levels. This motivates the exploitation of data parallelism within dynamic counted loops, in combination with other nested loops. Since dynamic counted loops result in nested bands in the schedule trees, the combined exploitation of multiple levels of parallelism including one or more dynamic counted loops requires special treatment that is not directly modeled by affine sets and relations in the tree or by band-local parallelism annotations. On GPU targets, the constraints on the grid/ND-range of multi-level data parallelism require the collection of bound information across nested bands: when launching a kernel, the parameters of the grid/ND-range must be known and may not evolve during the whole run of the kernel.

Unfortunately, the statements between nested bands that occur in dynamic counted loops are used to initialize dynamic upper bounds. Statements in the body of these dynamic counted loops depend on those definition statements, through the added dependences modeling the original dependence of the dynamic loop. Still, one may can sink these definition statements inside, within these dynamic counted loops, as a preprocessing step. As a result, the nested bounds can be easily combined together, with no intervening computation or control flow.

The inward movement of these definition statements is safe with the introduction of the upper bound $u$-parameter. Yet as a side-effect of this movement, each definition will be redundantly evaluated as many times as the number of iterations of the dynamic counted loop itself. This is the price to pay for a fixed upper bound on the number of iterations. It may be mitigated with additional strip-mining of the outer loops, to better control the value of $u$, effectively partitioning the loop nest into coarse-grain sub computations amenable to execution on a heterogeneous target.

## 5. EXPERIMENTAL EVALUATION

Our framework takes a C program with PENCIL functions as input [1]; PENCIL is intended as a target language for DSL compilers and as a high level portable implementation language to program accelerators. It allows a domain expert to express useful information for optimizing compilers like aliasing, independence, logical predicates, high level interprocedural data flow, etc., enabling more accurate static analysis and effective target-specific code generation.

We use PPCG [26], a polyhedral compiler for PENCIL that performs loop nest transformations, parallelization, data locality optimization, and generates OpenCL or CUDA code. PPCG exploits the information provided by PENCIL extensions, performs transformations and generate CUDA code automatically. In a follow-up auto-tuning step, we look for optimal parameter values for tile sizes, block sizes, grid sizes, etc. for a given application and target architecture. PPCG allows to export and import schedule trees, enabling us to prototype some of the transformation steps of our method in iscc.

We compare the performance of the code generated with our technique with that generated from the PENCIL source without our algorithm.

The experiments are conducted on a NVIDIA Quadro K4000 GPU. The CPU on the experimental platform is Intel Xeon E5-2630. The sequential code is compiled with the icc compiler from Intel Parallel Studio XE 2016, with the *-Ofast -fstrict-aliasing* optimization flags. The CUDA code is compiled with the NVIDIA CUDA 7.5 toolkit with the *-O3* optimization flag. We run each benchmark 9 times and retain the median value.

### 5.1 HOG Benchmark

The HOG benchmark is extracted from the PENCIL benchmark, a collection of applications and kernels for evaluating PENCIL compilers.

The distribution of intensity gradients or edge directions describe the local object appearance and shape within an image. When processing an image, the HOG descriptor divides it into small connected regions called cells. A histogram of gradient directions is then compiled for the pixels within each cell. The descriptor finally concatenates these histograms together. The descriptor also contrast-normalize local histograms by calculating an intensity measure across a block, a larger region of the image, and then using this value to normalize all cells within the block to improve accuracy. This normalization results in better invariance to changes in illumination and shadowing.

The kernel of the HOG descriptor contains two nested, dynamic counted loops. The upper bounds of these inner loops are defined and vary as the outermost loop iterates. The dynamic parameter is an expression of max and min

functions of the outer loop iterator and an array of constants. We derive the static upper bound parameter $u$ from the BLOCK_SIZE constant, a globally defined parameter of the program to declare the size of an image block.

Since we target a GPU architecture, we ought to extract large degrees of parallelism from multiple nested loops. As explained in the previous section, we thus sink the definition statements of dynamic parameters within inner dynamic counted loops and apply our template for a combined band. We may then generate the CUDA code with parameter values for tile sizes, block sizes, grid sizes, etc. We show performance results with and without host-device data transfer time, in Figure 8 and 9 respectively, considering multiple block sizes. The detection accuracy improves with the increase of the block size. Our algorithm achieves a promising performance improvement for each block size, and our technique can obtain a speedup ranging from 4.4× to 23.3× while the PENCIL code suffers from a degradation by about 75%.

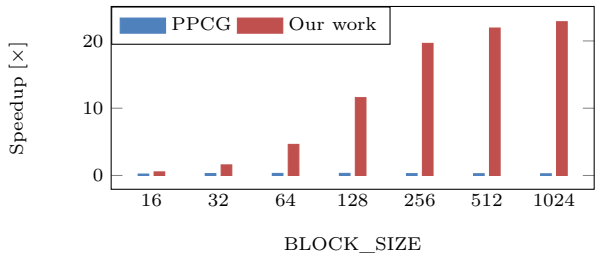


Figure 8: Performance of the HOG descriptor with data transfer time

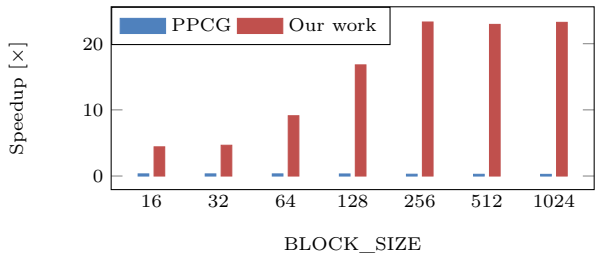


Figure 9: Performance of the HOG descriptor without data transfer time

## 5.2 SpMV Computations

Sparse matrix operations are an important class of algorithms frequently in graph applications, physical simulations to data analytics. They attracted a lot of parallelization and optimization efforts. Programmers may use different formats to store a sparse matrix, among which we consider four representations: the Compressed Sparse Row (CSR), Block CSR (BCSR), Diagonal (DIA) and ELLPACK (ELL) [27]. In their original form, loop bounds do not match our canonical structure: we apply a non-affine shift by the dynamic lower bound as discussed earlier. Our experiment in this subsection target the benchmarks presented in [25], with our own modifications to suit the syntactic constraints of our framework.

We first consider the PENCIL function for the CSR representation. All arrays are declared through the C99 variable-length array syntax with the static const restrict C99 type qualifiers/keywords, allowing PPCG to derive the size of the arrays offloaded on the accelerator despite the presence of indirect accesses (subscripts of subscripts), and that these arrays do not alias. The three other representations can modeled with a make-dense transformation, as proposed by [25], followed by a series of loop transformations. BCSR is obtained by applying a make-dense and tiling transformations, DIA can be obtained after make-dense, shift and permutation transformations, and a non-affine shift together with tiling can result in the ELL format. See Venkat et al. [25] for details.

The maximum number of non-zero entries in a row is the static upper bound and may be set as the $u$ parameter. It can be derived through an inspection. As a result, the references of indirect array subscripts can be sunk under the inner dynamic counted loop, exposing a combined band in the schedule tree. We show the performance without data transfer time in Figure 10. The input sparse matrices are obtained from the University of Florida sparse matrix collection [11]. Our technique accelerates the SpMV by 1.1× to 4.7×, which is consistent with the numbers reported by Venkat et al. [25].

BCSR is the blocked version of CSR, its parallel version is the same as that of CSR, after tiling with PPCG. We will therefore not show its performance. Similarly, we get the CUDA code with auto-tuned parameter values for the DIA and ELL formats, as shown in Figure 11 and 12. The original DIA code runs into a segmentation fault for the `mc2depi` and `pwtk` matrices, hence we will remove these two matrices from the experiments. Speedups range from 1.2× to 5.0× for DIA and from 1.9× to 7.4× for ELL.

## 5.3 Inspector/Executor Codes

We also compare the performance of the generated code when using an inspector/executor strategy to optimize the data layout of sparse matrix computations. The inspector is used to analyze memory reference patterns and to generate communication schedules, so we mainly focus on the application of our technique to the executor. Since BCSR can be obtained by applying a tiling transformation on a CSR kernel, we only take into account the executor of CSR. The executor of DIA format is not a dynamic counted loop and will not be studied.

Following the approach of Venkat et al. [25], inspector/executor schemes may cooperate with a polyhedral transformation and optimization framework. The framework uses three dedicated transformations to achieve this cooperation: make-dense, compact and compact-and-pad.

Both the compact and compact-and-pad transformations can be applied to CSR and ELL formats. The executor of the CSR format is roughly equivalent to the original CSR SpMV code, so the performance comparison is similar to that of the original CSR SpMV code, as shown in Figure 13. The ELL executor uses a transposed matrix to achieve global memory coalescing, whose efficiency depends heavily on the number of rows that have a similar number of non-zero entries. The performance result of the ELL executor is shown in Figure 14, for which we obtain a speedups from 2.9× to 10.2×.

# 6. RELATED WORK

The polyhedral framework is a powerful compilation technique to parallelize and optimize loops. It has become one of

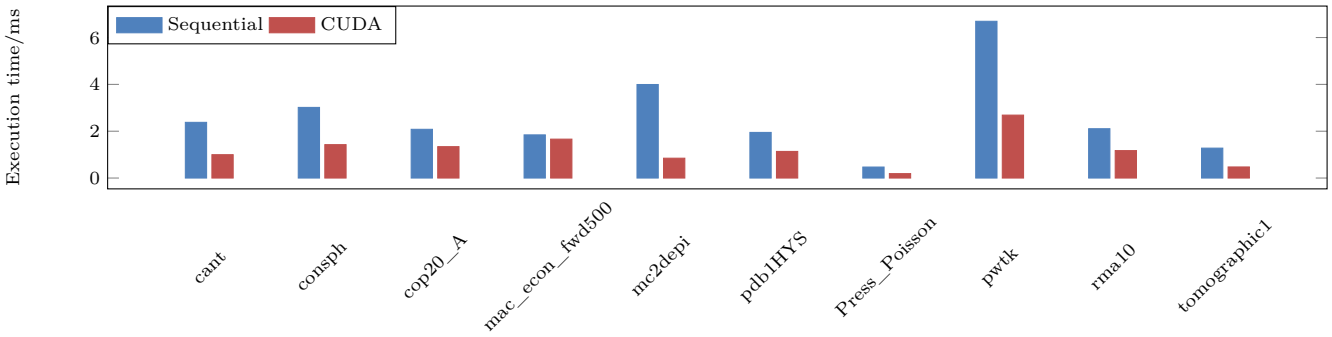


Figure 10: Execution Time comparison between sequential and CUDA CSR SpMV code

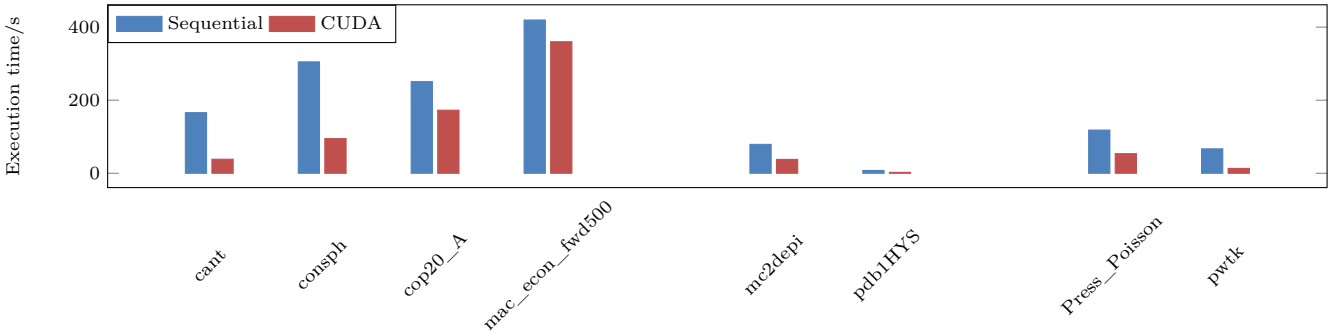


Figure 11: Execution time of the sequential and CUDA DIA SpMV code

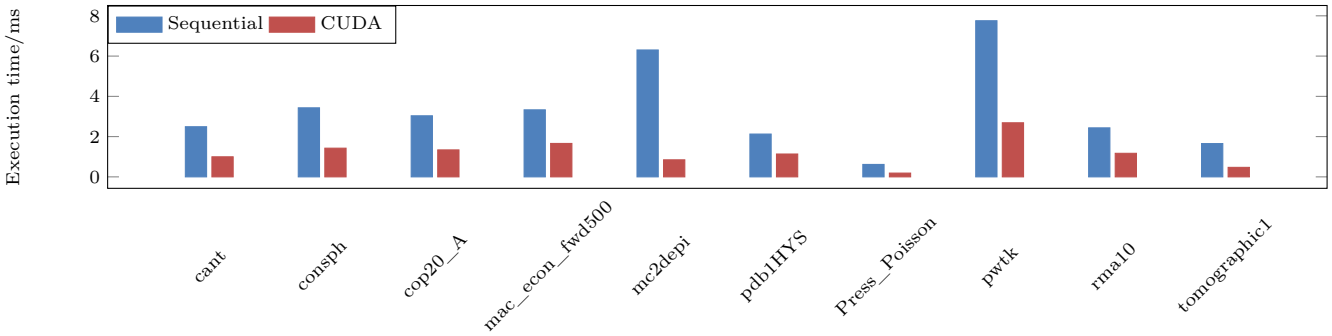


Figure 12: Execution time of the sequential and CUDA ELL SpMV code

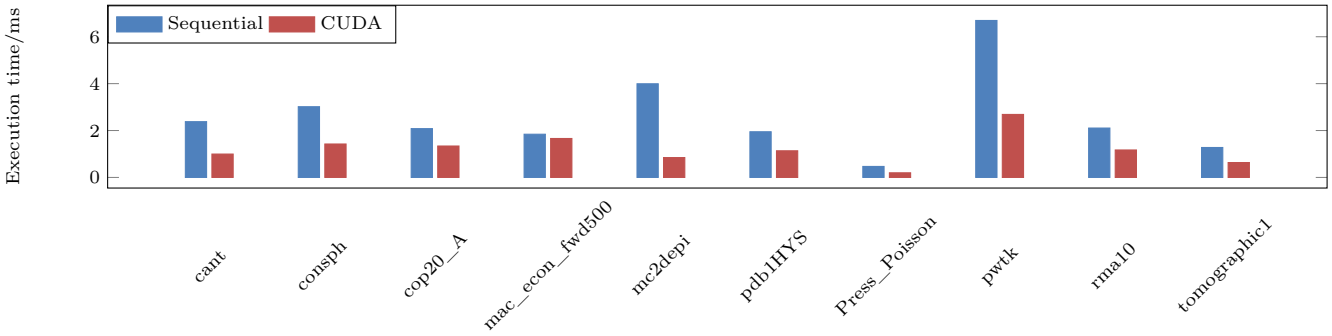


Figure 13: Execution time of the sequential executor and the CUDA CSR SpMV code

the main approach for the construction of modern parallelizing compilers. Its application domain used to be constrained to static-control, regular loop nests. But the extension of the polyhedral framework to handle irregular applications is increasingly important given the growing adoption of the technique. The polyhedral community invested significant efforts to make progress in this direction.

A representative application of irregular polyhedral tech-

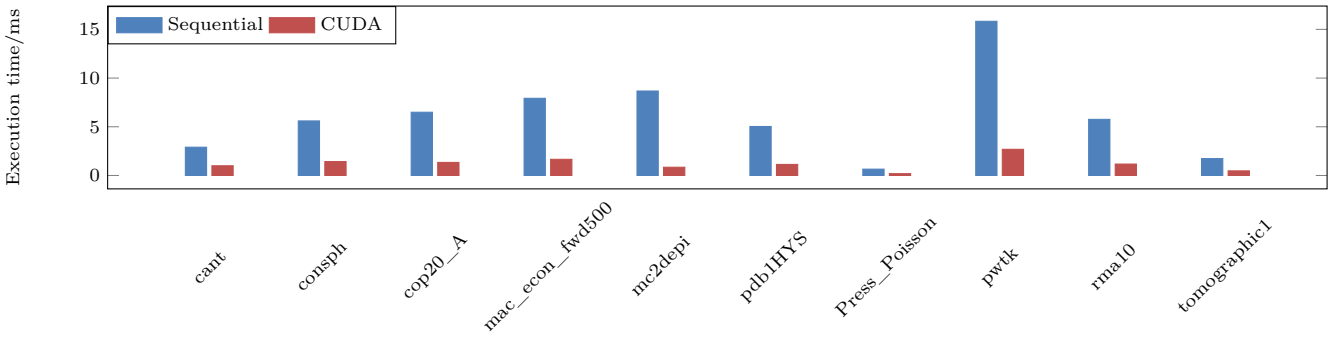


Figure 14: Execution time of the sequential executor and the CUDA ELL SpMV code

niques is the parallelization of `while` loops. The polyhedral model is expected to handle loop structures with arbitrary bounds that are typically regarded as **while** loops. Collard [10, 9] proposed a speculative approach based on the polyhedral model that extends the iteration domain of the original program and performs speculative execution on the new iteration domain. Parallelism is exposed at the expense of an invalid space-time mapping that needs to be corrected at run time. An alternative, conservative technique, consists in enumerating a super-set of the target execution space [14, 15, 16, 17], and then eliminating invalid iterations by determining termination detection on the fly. The authors present solutions for both distributed and shared memory architectures. Benabderrahmane et al. [5] introduce a general framework to parallelize and optimize arbitrary `while` loops by modeling control-flow predicates. They transform a `while` loop as a `for` loop iterating from 0 to $+\infty$. Compared to these approaches to parallelizing `while` loops in the polyhedral model, our technique relies on systems of affine inequalities only, as implemented in state-of-the-art polyhedral libraries. It does not need to resort to the first-order logic such as non-interpreted functions/predicates, it does not involve speculative execution features, and it makes dynamic counted loops amenable to a wider set of transformations than general `while` loops.

A significant body of work addressed the transformation and optimization of sparse matrix computations. The implementation of manually tuned libraries [2, 4, 7, 20, 21, 27] is the common approach to achieve high-performance, but it is difficult to port to each new representation and to different architectures. Sparse matrix compilers based on polyhedral techniques have been proposed [25], abstracting the indirect array subscripts and complex loop-bounds in a domain-specific fashion, and leveraging conventional Pluto-based optimizers on an abstracted form of the sparse matrix computation kernel. We ought to extend the applicability of polyhedra techniques one step further, considering general PENCIL code as input, and leveraging the semantical annotations expressible in PENCIL to improve the generated code efficiency and to abstract non-affine expressions.

# 7. CONCLUSION

In this paper, we studied the parallelizing compilation and optimization of an important class of loop nests where counted loops have a dynamically computed, data-dependent upper bound. Such loops are amenable to a wider set of transformations than general `while` loops with inductively defined termination conditions. To achieve this, we model control dependences on data-dependent predicates by revisiting a state-of-the-art framework to parallelize arbitrary `while` loops. We specialize this framework to facilitate its integration in schedule-tree-based affine scheduling and code generation algorithms; and we provide code generation templates for multiple scenarios, from single dynamic counted loops to nested bands case and a combined band case for fixed size multi-level data-parallel grids on GPUs. Our method relies on the systems of affine inequalities, as implemented in state-of-the-art polyhedral libraries. It takes a C program with PENCIL functions as input, covering a wide range of non-static control application encompassing a well studied class of sparse matrix computations. The experimental evaluation using the PPCG source-to-source compiler on representative irregular computations, from computer vision and finite element methods to sparse matrix linear algebra, validated the general applicability of the method and its performance benefits compared to shallower, black box approximation approaches of the control flow. Our next steps will be to fully automate and implement the algorithm in PPCG, and to conduct further experiments on CPU and GPU platforms, comparing the performance with manually-tuned libraries like OSKI and CUSP as well as reference implementations of computer vision and data analytics kernels. Full automation will also involve the static analysis of symbolic upper bounds, possibly building on additional PENCIL constructs.

## Acknowledgments

This work was partly supported by the European Commission and French Ministry of Industry through the ECSEL project COPCAMS id. 332913, and by the French ANR through the European CHIST-ERA project DIVIDEND. Our experiments with PENCIL benchmarks and PPCG benefited from the direct support of Michael Kruse, Riyadh Baghdadi and Chandan Reddy. And finally, none of this work would have been possible without the contributions of Sven Verdoolaege and Tobias Grosser on isl and PPCG.

# 8. REFERENCES

[1] Riyadh Baghdadi, Ulysse Beaugnon, Albert Cohen, Tobias Grosser, Michael Kruse, Chandan Reddy, Sven Verdoolaege, Adam Betts, Alastair F Donaldson, Jeroen Ketema, et al. PENCIL: a platform-neutral compute intermediate language for accelerator programming. In *Proceedings of the International*

*Conference on Parallel Architecture and Compilation (PACT)*, pages 138–149. IEEE Computer Society, 2015.

[2] S Balay, S Abhyankar, M Adams, J Brown, P Brune, K Buschelman, V Eijkhout, W Gropp, D Kaushik, M Knepley, et al. Petsc users manual revision 3.5. *Argonne National Laboratory (ANL)*, 2014.

[3] Cedric Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 7–16. IEEE Computer Society, 2004.

[4] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC)*, pages 18:1–18:11. ACM, 2009.

[5] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you think. In *Proceedings of 19th International Conference on Compiler Construction (CC)*, pages 283–303. Springer, 2010.

[6] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral program optimization system. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2008.

[7] Aydın Buluç and John R Gilbert. The combinatorial blas: Design, implementation, and applications. *International Journal of High Performance Computing Applications*, pages 496–509, 2011.

[8] J.-F. Collard, D. Barthou, and P. Feautrier. Fuzzy array dataflow analysis. In *ACM Symposium on Principles and Practice of Parallel Programming*, pages 92–102, Santa Barbara, CA, July 1995.

[9] Jean-François Collard. Space-time transformation of while-loops using speculative execution. In *Proceedings of the Scalable High-Performance Computing Conference 1994*, pages 429–436. IEEE Computer Society, 1994.

[10] Jean-François Collard. Automatic parallelization of while-loops using speculative execution. *International Journal of Parallel Programming*, 23(2):191–219, 1995.

[11] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1:1–1:25, 2011.

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

[13] P. Feautrier. Some efficient solutions to the affine scheduling problem, part II, multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, December 1992. See also Part I, one dimensional time, 21(5):315–348.

[14] Max Geigl, Martin Griebl, and Christian Lengauer. A scheme for detecting the termination of a parallel loop nest. *Proc. GI/ITG FG PARS*, 98, 1998.

[15] Max Geigl, Martin Griebl, and Christian Lengauer. Termination detection in parallel loop nests with while loops. *Parallel Computing*, 25(12):1489–1510, 1999.

[16] Martin Griebl and Jean-Francois Collard. Generation of synchronous code for automatic parallelization of while loops. In *Proceedings of the 1st International Euro-Par Conference on Parallel Processing*, pages 313–326. Springer, 1995.

[17] Martin Griebl and Christian Lengauer. On scanning space-time mapped while loops. In *In Proceedings of 3rd Joint International Conference on Vector and Parallel Processing (CONPAR 94-VAPP VI)*, pages 677–688. Springer, 1994.

[18] Tobias Grosser, Sven Verdoolaege, and Albert Cohen. Polyhedral ast generation is more than scanning polyhedra. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 37(4):12, 2015.

[19] Alexandra Jimborean, Philippe Clauss, Jean-François Dollinger, Vincent Loechner, and Juan Manuel Martinez Caamaño. Dynamic and speculative polyhedral parallelization using compiler-generated skeletons. *International Journal of Parallel Programming*, 42(4):529–545, 2014.

[20] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[21] John Mellor-Crummey and John Garvin. Optimizing sparse matrix–vector product computations using unroll and jam. *International Journal of High Performance Computing Applications*, 18(2):225–236, 2004.

[22] Fabien Quilleré, Sanjay Rajopadhye, and Doran Wilde. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming*, 28(5):469–498, 2000.

[23] Michelle Mills Strout, Larry Carter, and Jeanne Ferrante. Compile-time composition of run-time data and iteration reorderings. *ACM SIGPLAN Notices*, 38(5):91–102, 2003.

[24] Michelle Mills Strout, Alan LaMielle, Larry Carter, Jeanne Ferrante, Barbara Kreaseck, and Catherine Olschanowsky. An approach for code generation in the sparse polyhedral framework. *Parallel Computing*, 53:32–57, 2016.

[25] Anand Venkat, Mary Hall, and Michelle Strout. Loop and data transformations for sparse matrix code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 521–532, 2015.

[26] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for cuda. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):54, 2013.

[27] Richard Vuduc, James W Demmel, and Katherine A Yelick. Oski: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, 16(1):521, 2005.

[28] David Wonnacott and William Pugh. Nonlinear array dependence analysis. In *Proc. Third Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers*, 1995. Troy, New York.