A polyhedral compilation framework for loops with dynamic data-dependent bounds

Jie Zhao, Michael Kruse, Albert Cohen

A Polyhedral Compilation Framework for Loops with Dynamic Data-Dependent Bounds

Jie Zhao
INRIA & DI, École Normale Supérieure
Paris, France
jie.zhao@inria.fr

Michael Kruse
INRIA & DI, École Normale Supérieure
Paris, France
michael.kruse@inria.fr

Albert Cohen
INRIA & DI, École Normale Supérieure
Paris, France
albert.cohen@inria.fr

Abstract

We study the parallelizing compilation and loop nest optimization of an important class of programs where counted loops have a dynamic 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 loop-carried 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 dynamic programming, 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.

CCS Concepts • **Software and its engineering → Compilers**;

Keywords parallelizing compiler, loop nest optimization, polyhedral model, dynamic counted loop

CC'18, February 24–25, 2018, Vienna, Austria

ACM ISBN 978-1-4503-5644-2/18/02…$15.00
https://doi.org/10.1145/3178372.3179509

ACM Reference Format:
Jie Zhao, Michael Kruse, and Albert Cohen. 2018. A Polyhedral Compilation Framework for Loops with Dynamic Data-Dependent Bounds. In *Proceedings of 27th International Conference on Compiler Construction (CC'18).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3178372.3179509

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 computational 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 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 found together with statically unpredictable, non-affine array subscripts.

The polyhedral framework of compilation unifies a wide variety of loop and array transformations using affine (linear) transformations. The availability of a general-purpose method to generate imperative code after the application of such affine transformations [3, 16, 20] brought polyhedral compilers to the front scene, in the well-behaved case of static control loops. While significant amount of work targeted the affine transformation and parallelization of `while` loops [5, 8, 9, 12–15, 17], these techniques face a painful problem: the lack of a robust method to generate imperative code from the polyhedral representation. One representative 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 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. It 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 holds. 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 is also restricted. 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. Speculative approaches like the work of Jimborean et al. also addressed the issue [17], but a general "`while` loop polyhedral framework" compatible with arbitrary affine transformations has yet to emerge. In this paper, we make a more pragmatic, short term step: we focus on the special case of dynamic counted loops where the most difficult of these problems do not occur.

There has also been a significant body of research specializing on high-performance implementations of sparse matrix computations. Manually-tuned libraries [2, 4, 7, 18, 19, 27] are a commonly used approach, but it is tedious to implement and tune for each representation and target architecture. A polyhedral framework that can handle non-affine subscripts has a greater potential to achieve transformations and optimizations on sparse matrix computations, as illustrated by Venkat et al. [24].

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 performance portability. Our approach is based on systems of affine inequalities, as implemented in state-of-the-art polyhedral libraries [25]. Just like [22, 23], 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 avoids the complexity and overhead of speculative execution.

To extend the polyhedral framework to dynamic computed loops, our method relies on the computation of an affine upper bound for all dynamic trip counts that a given loop may reach, using a combination of additional static analysis and dynamic inspection. Revisiting the polyhedral compilation framework [5] of arbitrary `while` loops, we introduce exit predicates for dynamic counted loops, modeling the control dependence of the original loop through additional data dependences from the definition of these exit predicates to every statement in the loop body. We implement a schedule-tree-based algorithm [16] to enable the full automation of imperative code generation after the application of affine transformations, targetting both CPU and GPU architectures.

Our method goes beyond the state of the art [5, 17, 24] 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 or resorting to speculative execution. We conduct experiments on representative irregular computations, including dynamic programming, 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 introduce technical background and further motivate our approach to parallelize dynamic counted loops in the next section. Section 3 discusses the conversion of control dependences into data-dependent predicates. Section 4 introduces the code generation algorithm. 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 was traditionally limited to static control loop nests. It represents a program and its semantics using iteration domains, access relations, dependences and schedules. The statement instances are included in iteration domains. Access relations map statement instances to the array elements they access. Dependences capture the partial order on statement instances accessing the same array element (one of which being a write). The schedule implements a (partial or total) execution order on statement instances that is compatible with dependences.

Consider the running example in Figure 1. The upper bounds, m and n, of the j-loop and k-loop are computed in their common enclosing loop and updated dynamically as the i-loop iterates. As a result, it is not possible to classify the whole loop nest as a static control part (SCoP), and traditional polyhedral techniques do not directly apply. 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.

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

Figure 1. Example with dynamic counted loops

As an alternative, one may narrow the SCoP by only considering the j-/k-loop nest and treating the dynamic upper

bounds as symbolic parameters, enabling polyhedral transformations without problems. This, however, either introduces more frequent synchronizations by exploiting fine-grained parallelism when targeting on CPU targets, or misses the data locality along the outermost loop dimension and the opportunity to exploit full-dimensional parallelism on GPU platforms.

Statement S_2 does not have data dependences on other statements. However, there are output dependences among definition statements of dynamic parameters m and n. To faithfully capture the scheduling constraints, one should also model the control dependences of S_2 over both headers of the enclosing dynamic 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 each statement of the loop body, like in the framework of Benabderrahmane et al. [5]. The resulting dependence graph is shown in Figure 2. The solid arrows represent the original (output) dependences between definition statements of dynamic parameters, and the dashed arrows represent the data dependences converted from the exit conditions' control dependences.

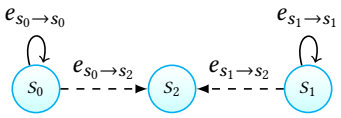

Figure 2. Dependence graph of the example

By capturing control dependences as affine relations from the definition of 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 [16]. Schedule trees can be flattened into a union of relations form, with each relation mapping the iteration domain of individual statements to a unified logical execution time space. A schedule tree typically comprises a domain node describing the overall extent of the statement instances, sequence/set nodes expressing ordered/unordered branches, filter nodes selecting a subset of the statement instances as the children of a sequence/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]. 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. We will leverage this extensibility to represent non-affine loop bounds.

Since dynamic counted loops cannot be appropriately represented in the iteration domain, a state of the art polyhedral compiler like PPCG may only model the outer loop, abstracting away the j-loop and k-loop, as the schedule tree of Figure 3. Following Benabderrahmane's work [5], we can derive two static upper bounds, u_1 and u_2, that are greater than or equal to m and n. The domain and access relations of statement S_2 can be over-approximated accordingly, and represented parametrically in u_1 and u_2. This representation can be used to compute a conservative approximation of the dependence relation for the whole schedule tree.

Based on this dependence information, one may derive a correct schedule using the Pluto algorithm or one of its variants [6, 26], to optimize locality and extract parallelism. 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, u_1 and u_2, as will be seen in Figure 5. 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` [16], traversing the schedule tree and specializing the code generation algorithm to integrate target-specific constraints, e.g., nested data parallelism and constant bounds. 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 dedicated "dynamic counted loop control flow". This involves either (on GPU targets) the reconstruction of the exit predicate and the introduction of an early exit (`goto`) instruction guarded by the predicate or (on CPU targets) the replacing the over-approximated static upper bound with the dynamic condition and the removing of the introduced control flow. Our algorithm generates code in one single traversal of the schedule tree[1].

3 Program Analysis

Dynamic counted loops arise frequently in irregular applications, but they may not be written in a form that can be

[1] Another difference with [5] where multiple traversals were needed.

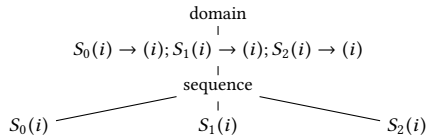

Figure 3. Original schedule tree of the example

handled with our technique. We need a preprocessing step to make them amenable to our approach.

3.1 Preparation

A dynamic counted loop with a dynamic counted upper bound and a static lower bound is referred to as the normalized format of dynamic counted loops, as shown in the example of Figure 1 is such a normalized format.

Sparse matrix computations represent an important class of dynamic counted loops. They are a class of computations using compressed data layout stores nonzero elements only. Loops iterating on the compressed layout may have dynamic lower and upper bounds. However, these loops can be easily normalized by subtracting the lower bound from the upper bound. This transformation may introduce non-affine array subscripts since the lower bound may not be affine; we assume the dependence analysis will conservatively handle such subscripts, leveraging Pencil annotations to refine its precision [1, 10]; we may also symbolically eliminate identical non-affine expressions on the left and right-hand side.

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 Deriving a Static Upper Bound

To make a dynamic counted loop amenable to a polyhedral representation, our approach assumes that a static control upper bound u on the dynamic number of iterations is available. The general idea is that a dynamic counted loop can always be converted into a static `for` loop enclosing an `if` statement whose condition checks the dynamic bound.[2]

The u parameter can 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 array size declarations or additional user-defined predicates in Pencil [1]. When such static methods fail, `MAXINT` or any type-dependent bound remains a valid approximation, but a tighter bound is preferable to avoid lifting induction variables to a wider integral type. Besides static analysis, dynamic inspection prior ahead of the loop nest of interest may be practical in some cases. For example, in sparse matrix computations, u may be computed by inspecting the maximum number of non-zero entries in a row in Compressed Sparse Row (CSR) format. All in all, affine bounds on the u parameter can generally be derived automatically, at compilation or run time, and the tightness of the approximation does not have an immediate impact on performance.

[2]This is easier than a general `while` loop, since the dynamic bound check remains continuously false after its first falsification.

3.3 Modeling Control Dependences

To model control dependences on dynamic conditions, we introduce additional data 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 dynamic counted loop. At code generation time, all statements in the body of the counted loop will have to be dominated by an early exit instruction conditioned by its predicate. This follows Benabderrahmane's 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 `goto` instructions/changing back to the dynamic conditions until code generation, to keep the control flow in a statically manageable form for a polyhedral compiler. For 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; j<u1; j++)
  for (k=0; k<u2; k++)
    if (j<m && k<n)
      S(j, k);
```

(b) if conditional

Figure 4. Conditional abstraction

The control dependences are therefore converted into data dependences between definition statements and the body of dynamic counted loops. Each statement in a dynamic counted loop is associated with a list of exit predicates. These predicates should be attached to the band node dominating the dynamic counted loop, and will be used to guard or terminate the execution within the over-approximation iteration domain bounded by the u parameters.

3.4 Scheduling

The u parameter and conversion of control dependences make it possible to approximate dynamic counted loops in the polyhedral model, at the expense of traversing a larger iteration space. We may thus apply any affine scheduling on this "approximated static control program", to safely compute a correct schedule tree preserving all dependences. Applying a variant of the Pluto algorithm attempting to minimize the reuse distance and expose tilable loops yields the schedule tree in Figure 5.

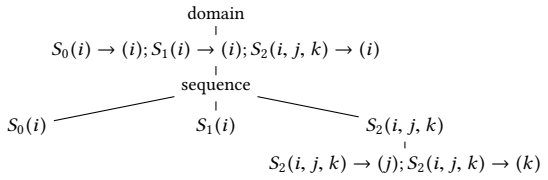

Figure 5. New schedule tree of the example

4 Generation of Imperative Code

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. Eventually, one needs to return to imperative code through a so-called code or AST generation algorithm. AST generation is a performance-critical step in any polyhedral framework. We extend the code generation scheme of Grosser et al. [16], itself derived from the algorithm by Quilleré et al. [20] and its CLooG enhancements and implementation [3].

When the Grosser et al. algorithm traverses the band nodes in a schedule, it projects out the local schedule constraints from the domain node. As the dynamic upper bounds are not modeled in the iteration domain (the domain node in the schedule tree and subsequent filter nodes), the generated loops will iterate from 0 to u. It is thus necessary to emit an early exit statement (for GPU architectures) or change the over-approximated static upper bound back to the original dynamic condition (for CPU architectures). Besides, the introduced control flow can also be removed when generating code for CPU targets, reducing the control overhead.

4.1 Extending the Schedule Tree

The Grosser et al. algorithm is not able in its original form to generate semantically correct code for our extended schedule tree. However, it can be easily modified to handle the special case of exit predicates that are homogeneous over all statements in a sequence or set node of the schedule tree (e.g., all statements in a band of permutable loops). This is facilitated through the syntactic annotation of dynamic counted loops using so-called mark nodes in the schedule tree. A mark node may attach any kind of information to a subtree; we used it here to specify which band nodes and which dimensions in those bands involve dynamic counted loops. To account for affine transformations combining static and dynamic counted loops (e.g., loop skewing), mark nodes are inserted at every dimension.

One may insert an extension node in a schedule tree to extend its iteration domain, e.g., to insert a new statement with a specific iteration domain. In our case, we replace each mark node with an extension node, inserting a guard statement with the proper exit predicate. In a first pass, all exit predicates are attached to the band node; a follow-up traversal through the predicate list lets the AST generator detect whether a dimension of the band node is a dynamic counted loop, and position early exits at the right level.

4.2 Generating Early Exits

When scanning the schedule tree to generate early exits for GPU targets, the AST generator creates a `goto` AST node for each of the above-mentioned extension nodes. All kinds of information about the early exit statement can be attached to this `goto` AST node, including (1) the iterator of the loop where the `goto` AST node is retained, (2) the depth of this loop in the nest, (3) the associated predicate list, (4) whether the loop is a dynamic counted loop, and (5) a `label` counter.

As the AST is generated in a top-down manner, it is possible to map each `goto` AST node to the loop it exits from. The predicate list is also attached to the node: one may determine whether a loop is dynamically counted by looking up for its iterator in each predicate. Finally, the `label` counter is incremented each time a dynamic counted loop is encountered, enforcing uniqueness.

4.3 Changing Back to Dynamic Conditions

When targeting on CPU architectures, it may not be allowed to jump in or out of the parallel region using an early exit statement like `goto`, but one may change the over-approximated static upper bound u back to the original dynamic condition. The information to facilitate such replacement can be attached to an AST annotation node and be the same with those of the `goto` AST node in GPU case except the `label` counter.

The upper bound of a loop can be replaced using a dynamic condition extracted from the predicate list once the loop is identified as being dynamic counted, followed by the removal of each occurrence of this dynamic condition, removing the introduced control overhead.

4.4 Code Generation for a Single Loop

The final step is converting the AST to a high level program. When a `goto` AST node of a dynamic counted loop is captured, a `goto` statement conditioned by its predicates is enforced after the loop body, as well as a `label` destination after the loop itself. The associated predicates are gathered in a conjunction and wrapped as one conditional, with loop iterators instantiated according to the loop level. A `label` is inserted after each dynamic loop as a target for a `goto` statement.

Changing back to the dynamic condition for a dynamic counted loop is straightforward, but special cares have to be taken to handle cases with multiple associated predicates. One may construct a *max* operation comprising all the associated predicates as the upper bound of a dynamic counted loop, without removing these introduced control flow since they have to be there to preserve the semantic of the code.

This schedule-tree-based code generation algorithm enables all kinds of loop transformations, the most challenging

one being loop fusion. When fusing two dynamic counted loops, the two sets of predicates are considered, and the early exit statements/*max*-operation-based dynamic upper bounds are guarded by/composed of their statementwise conjunction/them. A normal loop can be treated as a specific case of dynamic counted loop by reasoning on its static upper bound as a predicate.

Unfortunately this scheme efficiently supports a single dynamic counted loop only, and does not deal with the expression of parallelism in these loops.

4.5 Flat and Nested Parallelisms

As shown in Figure 5, the canonically constructed schedule tree isolates two nested band nodes to represent different levels of the loop nest. This works fine when the target architecture is a shared memory multiprocessor. As an illustrative example, Figure 6 is the generated code for a shared memory multiprocessor after the application of loop tiling on the code in Figure 1 with the outermost *i*-loop being parallelized. However, when targeting GPU accelerators or producing fix-length vector code, we usually expect to combine nested bands to express parallelism at multiple levels, and a constant iteration count may also be required for data-parallel dimensions. We therefore consider two cases depending on the need to extract parallelism across more than one band.

```
#pragma omp parallel for
for (i=0; i<100; i++) {
  m = f(i);
  n = g(i);
  for (jj=0; jj<m/BB+1; jj++)
    for (kk=0; kk<n/CC+1; kk++)
      for (j=0; j<min(m, jj*BB+BB); j++)
        for (k=0; k<min(n, kk*CC+CC); k++)
          S(i, jj, kk, j, k);
}
```

Figure 6. Code generation with loop tiling for CPUs

Flat parallelism within a band Let us first discuss the case of regenerating imperative code for one or more nested dynamic counted loops within a single band. As a first step, one may systematically generate conditional statements on exit predicates at the innermost level. Figure 4(b) shows an example illustrating this approach. The predicates of both loops are included in a single conditional, and generated under the inner loop. Notice that this approach is compatible with affine loop transformations such as loop interchange, not expressible in [5] due to the presence of spurious loop-carried dependences.

Yet one still needs to generate early exits in order to avoid traversing a potentially large number of empty iterations. We may extract the iterators one by one from the predicate list and generate the corresponding exit statements from the innermost outwards. The exit predicates are generated in the form of multiple conditionals rather than else branches, as shown in Figure 7. Unlike Jimborean et al. [17], we do not need speculation on the number of iterations, since we do not deal with general `while` loops; our technique always executes the same number of iterations as the original programs.

```
for (ii=0; ii<100/AA+1; ii++) {
  for (jj=0; jj<u1/BB+1; jj++) {
    for (kk=0; kk<u2/CC+1; kk++) {
      for (i=ii*AA; i<min(100, ii*AA+AA); i++) {
        for (j=jj*BB; j<min(u1, jj*BB+BB); j++) {
          for (k=kk*CC; k<min(u2, kk*CC+CC); k++) {
            m = f(i);
            n = g(i);
            if (j<m && k<n) S(j, k);
            if (k>=n) goto label0;
          } label0: ;
          if (j>=m) goto label1;
        } label1: ;
      }
      if (kk*CC>=n) goto label2;
    } label2: ;
    if (jj*BB>=m) goto label3;
  } label3: ;
}
```

Figure 7. Code generation with loop tiling for GPUs

Loop tiling is a special case that should be taken into account. Loop tiling involves the insertion of one or more additional schedule dimensions 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—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.

Nested parallelism across bands Targeting GPU accelerators or producing fix-length vector code 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 tree, 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. The constraints on the grid of multi-level data parallelism require the collection of bound information across nested bands: when launching a kernel, the parameters of the grid 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 can sink these definition statements inside, within the dynamic counted loops, as a preprocessing step. As a result, the nested bands can be combined again, with no intervening computation or control flow. Figure 7 shows an example after the application of loop tiling on the code in Figure 1.

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 iterations. Once again, this overhead 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 as input, and resorts to PENCIL [1] extensions only when dealing with indirect accesses (subscripts of subscripts), implying that all arrays are declared through the C99 variable-length array syntax with the `static const restrict` qualifiers, allowing PPCG to derive the size of the arrays offloaded on the accelerator despite the presence of indirect accesses, and telling that these arrays do not alias.

We use PPCG [26] to generate target codes, a polyhedral compiler that performs loop nest transformations, parallelization, data locality optimization, and generates OpenCL or CUDA code. The version `ppcg-0.05-197-ge774645-pencilcc` is used in our work. 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.

The experiments are conducted on a 12-core, two-socket workstation with an NVIDIA Quadro K4000 GPU. Each CPU is a 6-core Intel Xeon E5-2630 (Ivy Bridge). Sequential and OpenMP code are compiled with the icc compiler from Intel Parallel Studio XE 2017, with the flags *-Ofast -fstrict-aliasing* (*-qopenmp*). 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 Dynamic Programming

Dynamic programming is an alternative method of greedy algorithms to guarantee an optimal solution. In computer science, dynamic programming implies the optimal solution of the given optimization problem can be obtained by the combination of optimal solutions of its sub-problems, by solving the same sub-problems recursively rather generating new ones. Dynamic counted loops are usually involved in these problems. We investigate two representative dynamic programming problems—change-making and bucket sort.

Typically, the change-making problem is used to find the minimum number of coins that can add up to a certain amount W and to count how often a certain denomination is used, but it has a much wider application than just currency. The algorithm is also used to count how often a certain denomination is used.

Suppose N denominations are provided, each of which is $d_i (0 \leq i < N)$. As long as the given amount $W > d_i$, the frequency of the i-th denomination will be incremented by 1. As a result, d_i appears as a bound of the inner dynamic counted loop, enclosed by an outer loop iterating over the total number of denominations. Our technique successfully parallelizes the inner dynamic counted loop and generates the CUDA code in conjunction with a loop interchange optimization. We show the performance with different number of denominations N under different amount constraints W in Figure 8. It can be concluded from the figure that the performance improvement grows with the rise of the the number of denominations.

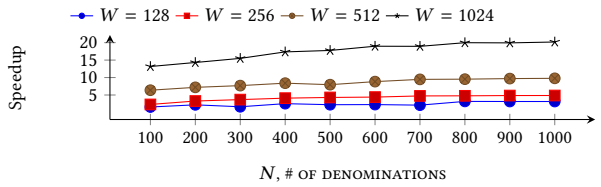

Figure 8. Performance of change-making on GPU

Bucket sort is a generalization of counting sort, sorting by first scattering the N elements of a given array into a set of M buckets, sorting each bucket individually, and finally gathering the sorted elements in each bucket in order. Due to the comparison operations, a sorting algorithm is inherently not the candidate for parallelization. However, it is possible to parallelize and optimize the gathering step of bucket sort.

We consider a uniform random distribution of elements of the input array. The algorithm has to gather $size[i]$ elements in the i-th bucket, whose static upper bound can be set as N. The dynamic counted loop controlled by the bucket size is captured by our method and parallelized in the form of CUDA code on GPUs. The performance with different array sizes N and different bucket numbers M is shown in Figure 9, indicating the speedup rises along with the increase of the number of buckets involved.

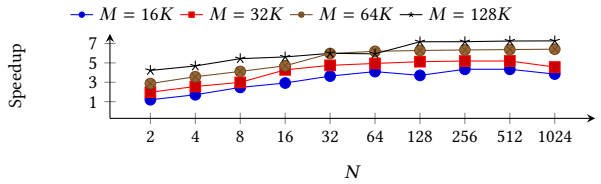

Figure 9. Performance of the bucket sort on GPU

5.2 HOG Benchmark

The HOG benchmark is extracted from the PENCIL benchmark suite.[3], a collection of applications and kernels for

[3] https://github.com/pencil-language/pencil-benchmark

evaluating PENCIL compilers. 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, resulting 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 global 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 subsection 4.5, we sink the definition statements of dynamic parameters within inner dynamic counted loops and apply our AST generation scheme for a combined band for GPU architecture. 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 10, 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%.

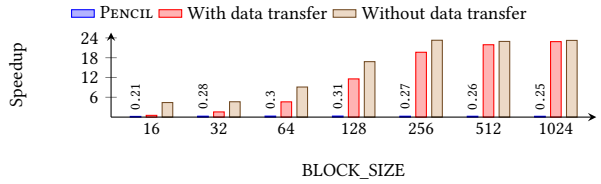

Figure 10. Performance of the HOG descriptor on GPU

5.3 Finite Element Method

equake is one of the SPEC CPU2000 benchmarks. It follows a finite element method, operating on an unstructured mesh that locally resolves wavelengths. The kernel invokes a 3-dimensional sparse matrix computation, followed by a series of perfectly nested loops. We inline the follow-up perfectly nested loops into this sparse matrix computation kernel to expose opportunities for different combinations of loop transformations.

In the 3-dimensional sparse matrix computation, a reduction array is first defined in the outer i-loop, and every element is repeatedly written by a j-loop that is enclosed by a `while` loop iterating over the sparse matrix. Finally, these reduction variables are gathered to update the global mesh. The `while` loop can be converted to a dynamic counted loop via preprocessing.

One may distribute the three components of the sparse matrix computation kernel, generating a 2-dimensional permutable bands on the dynamic counted loop in conjunction with unrolling j-loop, and fusing the gathering component with its follow-up perfectly nested loops. This case is called "2D band" in Figure 11.

One may also interchange the dynamic counted loop with its inner j-loop. As a result, all of the three components of the sparse matrix computation are fused. The loop nest is separated into two band nodes, the outer is a 2-dimensional permutable and the inner is dynamic counted loop. This is called "(2+1)D band" in the figure.

Alternatively, the three components can be distributed instead of being fused. This makes a 3-dimensional permutable band involving the dynamic counted loop, and results in the fusion of the gathering component with the follow-up perfectly nested loops. This case is called "3D band" in the figure.

We generate CUDA code for these different combinations and show the result in Figure 11, considering different input sizes. The u parameter is set to the maximum non-zero entries in a row of the sparse matrix. The baseline parallelizes the outer i-loop only, which is what PPCG does on this loop nest; we reach a speedup of 2.7× above this baseline.

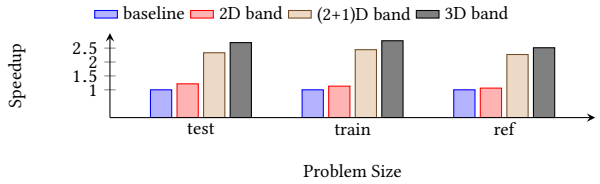

Figure 11. Performance of equake on GPU

5.4 SpMV

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: CSR, Block CSR (BCSR), Diagonal (DIA) and ELLPACK (ELL) [27]. Our experiment in this subsection target the benchmarks used in [24], with our own modifications to suit the syntactic constraints of our framework.

We first consider the CSR representation. The other three representations can be modeled with a make-dense transformation, as proposed by [24], followed by a series of loop and data transformations. 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. Note that

Venkat et al. [24] assume block sizes are divisible by loop iteration times, but our work has no such limitation. The inspector is used to analyze memory reference patterns and to generate communication schedules, so we mainly focus on comparing our technique to the executor. The executor of DIA format is not a dynamic counted loop and will not be studied.

In the original form of the CSR format, loop bounds do not match our canonical structure: we apply a non-affine shift by the dynamic lower bound as discussed earlier. 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.

Venkat et al. [24] optimize the data layout of the sparse matrix via a series of transformations including make-dense, compact and compact-and-pad, but it can only parallelize the outer loop. Our technique can identify the inner dynamic counted loop and parallelize both loops, exposing a higher degree of parallelism. We show the performance in Figure 12, using the matrices obtained from the University of Florida sparse matrix collection [11] as input. We also show the performance of a manually-tuned library–CUSP [4] in the figure. Our method beats the state-of-the-art automatic technique and manually-tuned library in most cases.

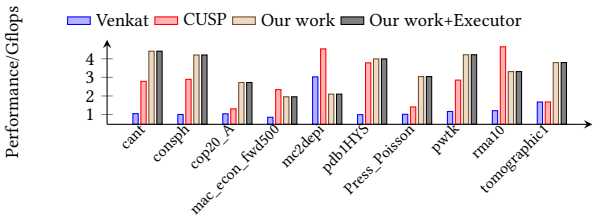

Figure 12. Performance of the CSR SpMV on GPU

In [24], the ELL format is derived from CSR by tiling the dynamic counted loop with the maximum number of nonzero entries in a row. Rows with fewer non-zeros are padded with zero values, implying there will be no early exit statements when parallelizing both loops. It makes their approach effective when most rows have a similar number of non-zeros. Our technique implements a similar idea without data transformation by extending the upper bound of the inner dynamic counted loop to the maximum number of non-zeros, and automatically emitting early exit statements when there are fewer non-zeros in a row, minimizing the number of iterations of the dynamic counted loop. The performance is shown in Figure 13 together with that of the CUSP library. A *format_conversion* exception is captured when experimenting the CUSP library with `mac_econ_fwd500`, `mc2depi`, `pwtk` and `tomographic1` while our technique remains applicable on all formats.

Although the manually-tuned library outperforms the proposed technique under three inputs, our method performs better in general. In addition, our technique provides comparable or higher performance than the inspector/executor scheme without the associated overhead.

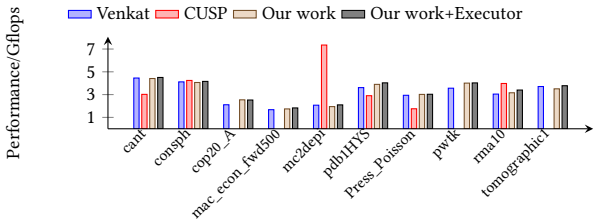

Figure 13. Performance of the ELL SpMV on GPU

5.5 Inspector/Executor

The inspector/executor strategy used in [24] obtains performance gains by optimizing the data layout. Our technique can also apply to the executor of this strategy as a complementary optimization, further improving the performance of the executor. The inspector/executor strategy, however, is not so satisfying as expected for CSR, since the CSR executor is roughly the same with the original code.

As a result, the performance of our generated code when applying our technique on the CSR executor is also roughly the same with that applying on the orignal code, as shown in Figure 12. As a complementary optimization, our technique can speedup the CSR executor by up to 4.2× (from 1.05 Gflops to 4.41 Gflops under `cant` input).

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. To get rid of this limitation, our technique may be applied to eliminate the wasted iterations by emitting early exit statements. Experimental results of the ELL executor are shown in Figure 13, for which our technique improves the performance by up to 19.7% (from 2.11 Gflops to 2.53 Gflops under `cop20_A` input).

5.6 Performance on CPU Architectures

We also evaluate our technique on CPU architectures. Unlike generating CUDA code, the original dynamic condition can be taken back when generating OpenMP code on CPU architectures, avoiding the combination of nested bands and the refactoring of the control flow.

The performance results are shown in Figures 14–17. We do not show the performance of dynamic programming examples on CPU architectures since our code generation scheme generates OpenMP code identical with the hand written one. For the remaining benchmarks, our technique enables aggressive loop transformations including tiling, interchange, etc., leading to a better performance when these

optimizations are turned on. As the CUSP library is designed for GPU architectures, we only compare the performance of the SpMV code with Venkat et al.'s [24] work.

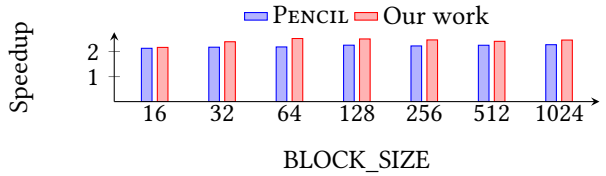

Figure 14. Performance of the HOG descriptor on CPU

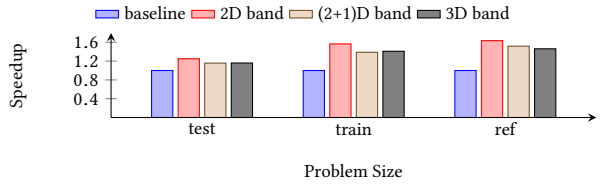

Figure 15. Performance of equake on CPU

Figure 16. Performance of the CSR SpMV on CPU

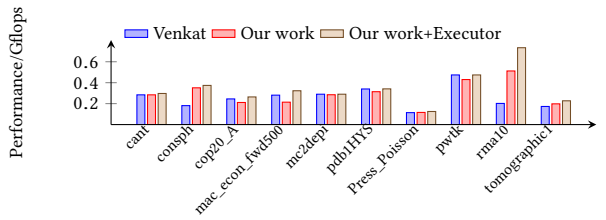

Figure 17. Performance of the ELL SpMV on CPU

6 Related Work

The polyhedral framework is a powerful compilation technique to parallelize and optimize loops. It has become one of the main approaches 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 techniques 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 [8, 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. Beyond polyhedral techniques, Rauchwerge [21] proposed a speculative code transformation and hybrid static-dynamic parallelization method for `while` loops. An alternative, conservative technique, consists in enumerating a super-set of the target execution space [12–15], 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, 18, 19, 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 [24], 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 polyhedral 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. To achieve this, we introduce a static upper bound and 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, covering

all scenarios from a single dynamic counted loop to nested parallelism across bands mapped to GPUs with fixed-size data-parallel grids. Our method relies on 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 the well studied class of sparse matrix computations. The experimental evaluation using the PPCG source-to-source compiler on representative irregular computations, from dynamic programming, computer vision and finite element methods to sparse matrix linear algebra, validated the general applicability of the method and its benefits over black-box approximations of the control flow.

Acknowledgments

This work was supported by the National Natural Science Foundation of China under Grant No. 61702546, the European Commission and French Ministry of Industry through the ECSEL project COPCAMS id. 332913, and the French ANR through the European CHIST-ERA project DIVIDEND.

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. 2015. PENCIL: a platform-neutral compute intermediate language for accelerator programming. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation*. IEEE Computer Society, 138–149.

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

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

[4] Nathan Bell and Michael Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, No. 18.

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

[6] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 101–113.

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

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

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

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

[11] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Trans. Math. Software* 38, 1 (2011), 1:1–1:25.

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

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

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

[15] Martin Griebl and Christian Lengauer. 1994. On scanning space-time mapped while loops. In *In Proceedings of 3rd Joint International Conference on Vector and Parallel Processing*. Springer, 677–688.

[16] Tobias Grosser, Sven Verdoolaege, and Albert Cohen. 2015. Polyhedral AST generation is more than scanning polyhedra. *ACM Transactions on Programming Languages and Systems* 37, 4 (2015), 12:1–12:50.

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

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

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

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

[21] L. Rauchwerger and D. Padua. 1995. Parallelizing while loops for multiprocessor systems. In *Proceedings of 9th International Parallel Processing Symposium*. 347–356.

[22] Michelle Mills Strout, Larry Carter, and Jeanne Ferrante. 2003. Compile-time Composition of Run-time Data and Iteration Reorderings. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming Language Design and Implementation*. ACM, 91–102.

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

[24] Anand Venkat, Mary Hall, and Michelle Strout. 2015. Loop and Data Transformations for Sparse Matrix Code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 521–532.

[25] Sven Verdoolaege. 2010. Isl: An Integer Set Library for the Polyhedral Model. In *Mathematical Software – ICMS 2010*. Springer Berlin Heidelberg, 299–302.

[26] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimization* 9, 4 (2013), 54:1–54:23.

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

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