Combining Polyhedral and AST Transformations in CHiLL

Huihui Zhang, Anand Venkat, Protonu Basu, Mary Hall
University of Utah
50 S. Central Campus Drive
Salt Lake City, UT

ABSTRACT

Polyhedral transformation and code generation frameworks are powerful tools for composing a complex sequence of loop transformations and automatically generating correct, highly-optimized code. Such frameworks are limited to applying transformations only to loop nest computations in the affine domain. For the most part, the statements of the original program remain unmodified, other than to update the array indices based on modified loop bounds. In this paper, we describe new transformations in the CHiLL compiler that modify or create new statements in the abstract syntax tree (AST) associated with the program. We show how these AST transformations can be composed with polyhedral transformations to increase the power of polyhedral frameworks. Our examples include inspector/executor optimizations for non-affine computations, stencil optimizations and parallel code generation (CUDA and OpenMP).

Categories and Subject Descriptors

D.3 [**Programming Languages**]; D.3.4 [**Processors**]: Code generation, Compiler, Optimization

General Terms

Compiler Optimization

Keywords

polyhedral transformation and code generation, abstract syntax trees

1. INTRODUCTION

Polyhedral code generation and transformation frameworks have become popular for their elegance and robustness in composing a complex sequence of transformations to loop nest computations and in generating high-performance, correct code. A polyhedral compiler typically reasons about loop nest computations by manipulating abstractions that represent the iteration spaces of individual statements in loop nests. Transformations are applied to the iteration space abstractions, and during code generation the statements from the original program are updated, with the inverse mapping applied to the array accesses. When used in this way, polyhedral compilers are typically unable to apply transformations that significantly modify the statements themselves. This paper argues that polyhedral frameworks can be far more powerful if extended to apply transformations that require significant changes to the statements to be generated, perhaps adding new data structures and replacing statements.

There has been some recent research interest in mixing abstract syntax tree (AST) and polyhedral optimizations [10, 19]. Other research [5] may rely on this mixing, particularly in the parallel code generation step [1], but quite often this is done as a post-pass outside of the polyhedral framework. The strength of a polyhedral framework lies in the ability to compose its transformations. *Therefore, if AST and polyhedral transformations are to be combined, then it is critical that the ability to compose transformations be preserved.*

This paper looks at recent research in extending the CHiLL compiler to incorporate optimization sequences that compose together AST and polyhedral transformations. This research was motivated by the need to support applications and optimizations in CHiLL that might have previously been considered beyond its reach. However, these extensions are facilitated by the data structures that comprise the framework, and in fact the first simple transformation incorporated into CHiLL that combines AST and polyhedral transformations was *unroll* [6, 12]. In this paper, we first examine optimizations for sparse matrix computations [21, 20]. Polyhedral compilers mostly operate on affine loop nests, where subscript expressions and loop bounds are linear functions of the loop indices. In contrast, sparse matrix codes exhibit non-affine indirection through index arrays (e.g., B[i] in the access expression A[B[i]] or as a loop lower or upper bound). Our research incorporates analysis, transformation and code generation in the presence of array indirection. Because some of the analysis must be deferred until run time when array indices can be resolved, our compiler employs an *inspector/executor* methodology [18, 13]. Next, we describe an optimization for high-order stencils which we call *partial sums* [2]. High-order stencils (e.g., the 27-point stencil shown in this paper) perform more computation and examine more input data compared to standard memory-bound stencils. Thus, they exhibit significant data reuse and are compute bound; application scientists are increasingly inter-

IMPACT 2016
Sixth International Workshop on Polyhedral Compilation Techniques
Jan 19, 2016, Prague, Czech Republic
In conjunction with HiPEAC 2016.

http://impact.gforge.inria.fr/impact2016

ested in high-order stencils as they are a better match for current and future architectures, where costs of data movement dominate. Partial sums optimize high-order stencils by introducing buffers that store and reuse redundant data and computation; the resulting code is more memory bound and amenable to other stencil optimizations. The inspector/executor transformations and partial sum transformation are composable with existing polyhedral transformations in CHiLL. Finally, we consider parallel code generation in both CUDA [17, 12, 21, 20] and OpenMP [3, 2]. To the best of our knowledge, this collection of AST transformations is the most extensive to be integrated into a polyhedral framework.

We observe what we view as the strengths and weaknesses of the work to date, and compare with other work [10, 19]. Our goal is to understand the requirements and challenges posed by such optimizations and their impact on the abstractions that are common to polyhedral frameworks.

The remainder of the paper presents background that describes the abstractions in CHiLL that are used in transformation and code generation, presents an overview of composing AST and polyhedral transformations, and then looks at these examples in detail. Finally, we compare to related work.

2. BACKGROUND

CHiLL harnesses both polyhedral and AST abstractions for its internal representation of a loop nest computation. The *Statement* is a central data structure to CHiLL's internal representation that achieves a clean separation between polyhedral and AST abstractions. Each statement in the code has three components:

- *IS*: The iteration space of a statement in a loop nest, expressed in relation form.
- *xform*: The transformation applied to this *IS* in relation form.
- *code*: An actual pointer to the AST segment of the code. Hence loop and conditional code constructs are created in polyhedral relation form, while the AST is encapsulated within *code* field within the statement.

In addition, a dependence graph is constructed with a mapping to the statements to aid in determining safety of transformations to be applied. In the remainder of this section, we demonstrate how these representations make it possible to combine polyhedral and AST transformations. The key point is that all the representations must remain consistent throughout to maintain composability of transformations. These points are illustrated by the simple example in Figure 1.

2.1 Dependence Graph

The construction of the dependence graph synergizes both the polyhedral and AST representations. It first extracts the array references in each statement's *code* field, and creates an affine relation corresponding to the array subscript. The underlying polyhedral library is then queried with a system of affine inequalities to check for the range of the loop index values for which the dependences exist. This process is repeated for every pair of statements and the dependence graph is constructed, with each statement represented by a vertex in the graph and edges corresponding to dependences. For instance the statement s0 in Figure 1 has a self dependence with a positive distance of $+1$ due to the array references $a[i+1]$ and $a[i]$.

2.2 Iteration Spaces

Statements within the analyzed loop nest may be at different nesting levels and enclosed by differing sets of outer loops. To present a polyhedral view of the loop nest, a unified iteration space, with an iteration space vector corresponding to each individual statement is constructed, including auxiliary dimensions in the iteration space to preserve lexicographic ordering.

An iteration space is a set of iteration vectors, which are represented as integer tuples. Given a loop nest with maximum loop depth of n, an iteration vector for each statement is defined as $i = \{c_0, l_1, c_1, l_2, ..., c_n, l_n, c_{n+1}\}$, where indices $l_{1..n}$ are used to represent the actual loop levels and indices $c_{0..n+1}$ are auxiliary loops for ensuring lexicographic order.

2.3 Transformations and Code Generation

Standard polyhedral transformations in CHiLL are implemented via linear mappings, which require affine loop bounds, conditional expressions and array subscripts. Expressing the program transformation as an affine mapping on the input code's iteration space allows composing transformations by using a transformation's output as the input of the following one, with the advantage that individual transformations need not rely too much on the input code's syntactic structure. Once all transformations are aggregated, the subscript expressions are updated with the inverse mapping of the transformation to ensure correctness as shown in Figure 1. Once all transformations have been applied, the *IS* and *xform* for each statement are input to CodeGen+ [7] which employs standard polyhedra scanning techniques and generates code with minimized control overhead.

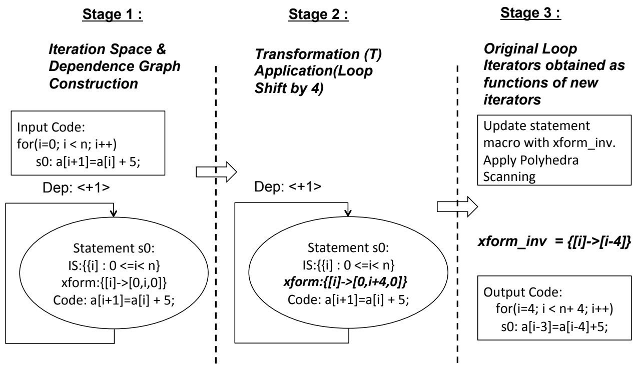


Figure 1: Polyhedral transformations and code generation.

3. OVERVIEW

Table 1 describes the combined AST and polyhedral transformations described in this paper. The columns of the table list the optimization techniques, the transformations that must be applied to the AST, the accompanying changes in the iteration spaces, and subsequent polyhedral transformations that are applied. The top part of the table focuses on transformations which are composable with subsequent polyhedral transformations. A key requirement for composability is whether the abstractions from the previous section are consistent with the current code. In particular, in the process of applying the transformations the dependence graph must be updated. The parallel code generation is not treated as a transformation and is not composable; parallel code is emitted during the polyhedra scanning. The remainder of this paper describes these combined transformations, highlighting the interplay between AST and polyhedral optimization.

4. INSPECTOR/EXECUTOR GENERATION FOR SPARSE CODES

Inspector/executor transformations are a class of run-time transformations, where the *inspector* is the code that may traverse the original code's iteration space, or analyze certain index arrays and potentially reorder or restructure the original code. Additionally it may reorganize the memory referenced by the original code. The *executor* is the modified version of the original computation that references the potentially reordered iteration space and/or memory references by the inspector code.

Within our compiler framework, we direct the automatic generation of inspector/executor code via transformation recipes. Here we provide descriptions of two types of inspector/executor transformations we have developed in CHiLL to optimize codes with indirect loop bounds and memory accesses, specifically the Sparse Matrix Vector (SpMV) kernel. In general the transformations are applicable to the class of input codes, which either do not have any loop-carried dependences or have dependences that only arise due to an associative computation such as reductions which are insensitive to iteration reordering.

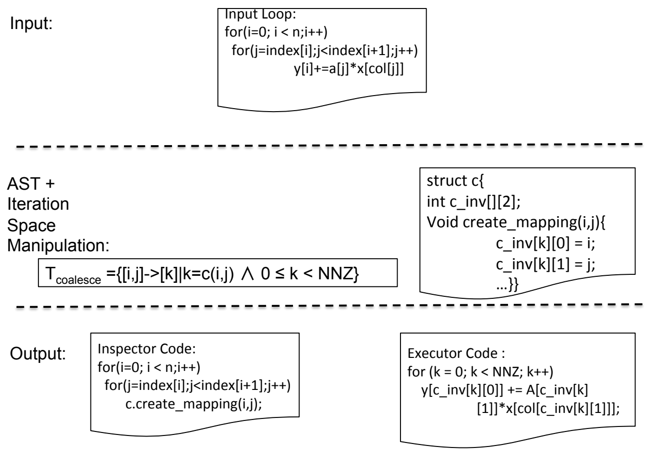
The first transformation, generalized loop coalescing, may be viewed as an iteration space restructuring transformation, while the second series of transformations *make-dense, compact and compact-and-pad* accomplish a data transformation additionally. We illustrate and highlight the interplay between polyhedral and AST code generation mechanisms employed for these transformations in this section.

4.1 Non-affine Extensions

The polyhedral transformation model was primarily designed to optimize codes with affine loop bounds and array subscript expressions. As such few polyhedral frameworks provide the capability to model non-affine code constructs such as array indirection during code generation. Furthermore, there is limited support to augment code generation with user-supplied code [10].

Within the Omega+/CodeGen+ [7] libraries used by CHiLL, the uninterpreted function symbol abstraction is provided to model functions or mappings whose exact semantics is undetermined at compile-time. This abstraction is utilized in [21] for inspector/executor transformations within CHiLL. The generalized loop coalescing transformation [21] converts a loop nest of arbitrary depth to a singly nested loop with the number of instances the body of the loop executes remaining invariant in the transformation. The transformation is modeled as an uninterpreted function that has argument arity equal to the dimension/depth of the input loop nest; it takes as input the loop indices of all levels in the input code and returns a single output loop iterator. For instance in the relation $T_{coalesce}$ shown in Figure 2, loop indices i and j are coalesced into a single output iterator k utilizing the uninterpreted function c with arity 2. The inspector code also determines the upper bound of the count of iterations of the newly coalesced loop, denoted as NNZ.

4.2 Run-time Inspector Code Generation



```
Input Loop:
for(i=0; i < n;i++)
  for(j=index[i];j<index[i+1];j++)
            y[i]+=a[j]*x[col[j]]
```

$T_{coalesce}$ ={[i,j]->[k]|k=c(i,j) ∧ 0 ≤ k < NNZ}

```
struct c{
int c_inv[][2];
Void create_mapping(i,j){
          c_inv[k][0] = i;
          c_inv[k][1] = j;
          ...}}
```

```
Inspector Code:
for(i=0; i < n;i++)
  for(j=index[i];j<index[i+1];j++)
     c.create_mapping(i,j);
```

```
Executor Code :
for (k = 0; k < NNZ; k++)
  y[c_inv[k][0]] += A[c_inv[k]
          [1]]*x[col[c_inv[k][1]]];
```

Figure 2: Compiler-generated coalescing inspector.

In typical polyhedral code generation systems [7, 22], when the iteration space of the code being transformed is constrained in the affine domain, the array subscripts in the statement macro are updated with the inverse of the specific transformation. This approach was extended for non-affine transformations in [21] where an inspector sets up the correspondence of the input iterators to the output iterator. The executor code's references to the input iterators are then updated with the values recorded by the inspector.

The functionality of the inspector is cloaked within the uninterpreted function abstraction, in other words, the inspector code generation is transparent to the CHiLL user. However the code generation backend has to instrument the functionality of the inspector code. To this end, the input code's AST is modified to set up the data structures that will be needed to flesh out the uninterpreted function relation.

The ROSE compiler infrastructure [14] is used to modify the AST, specifially to create a C++ **struct** to represent the functionality of the inspector. The struct has a field for each input iterator being coalesced as shown in Figure 2. This struct is used to flesh out the mapping corresponding to the uninterpreted function c, used in $T_{coalesce}$, hence it assumes the same name of the function.

More elaborate inspectors are constructed in [20], which introduces the following new transformations:

- *make-dense*: takes as input any set of non-affine array index expressions and introduces a guard condition and as many dense loops as necessary to replace the non-affine index expressions with an affine access.
- The *compact* and *compact-and-pad* transformations are *inspector/executor* transformations; an automatically-generated inspector gathers the iterations of a dense

Compiler Transformations			
Optimization techniques	**AST transformations**	**Polyhedral transformations**	**Composable with other optimizations**
Unroll	• Replicate AST for new unrolled statements • Adjust array index expressions by constants	• Introduce stride into iteration space	• Various • Subsequent to datacopy to registers
Inspector/executor for sparse codes	• Create linked list struct in AST • Parse *if* condition in AST and convert to relation	• Encode sparse iteration space of executor using uninterpreted function symbols • Derive closed form iterators for efficient inspector	• Datacopy, scalar expansion • Tiling and unrolling of executor
Partial sums for high-order stencils	• Create partial sum buffers • Create new statements • Delete existing statements	• Create iteration spaces for new statements • Ensure lexicographical ordering of statements • Create new dependence graph	• Fusion, distribution • Skewing • Permutation
Parallel Code Generation			
CUDA	• Eliminate block/thread loops • Rewrite statement with updated indices • Insert synchronizations • Introduce CUDA kernel and auxiliary functions	—	—
OpenMP	• Add pragmas via OMP nodes into AST • Add OMP clauses into AST • Add explicit spinlocks	—	—

Table 1: Taxonomy of CHiLL AST+Polyhedral Transformations.

loop that are actually executed and the optimized executor only visits those iterations. The executor represents the transformed code that uses the compacted loop, which can then be further optimized.

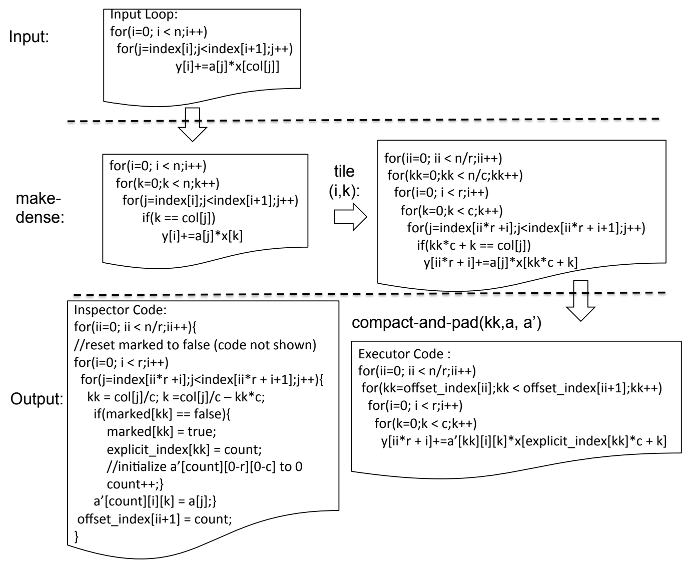
- In the *compact-and-pad* transformation, the inspector also performs a data transformation, inserting explicit zeros when necessary to correspond with the optimized executor.

These three transformations and automatic generation of inspectors combine the polyhedral manipulation of loop iteration space and constraints on statement execution with AST modifications to introduce new statements.

Importantly, linked list data structures are introduced into the inspector code's AST representation for a space-efficient implementation where the number of non-zero entries being compacted are not known a priori. The IF-condition that specifies which iterations to compact, was converted into an iteration space constraint/(in)equality on the iteration space, by using *exp2constraint* in CHiLL. The condition, once in a form amenable to polyhedral manipulation, is analyzed to identify iterations for which closed form expressions[20] may be derived from the guard condition without explicitly traversing the corresponding loop level, hence increasing the efficiency of the inspector. The *compact-and-pad* transformation derives the padding size from the maximum loop trip count of certain loops, by converting the array subscript expression to a function of the loop iterators, and then extracting the maximum footprint of the subscript function given the loop bounds. Also both the inspector and executor's iteration spaces were constructed in relation form using uninterpreted functions to represent index arrays where required. As each new transformation is applied, the dependence graph is incrementally updated to reflect the changes to the code.

4.3 Example: BCSR

The transformations *make-dense* and *compact-and-pad* facilitate the derivation of SpMV computation based on the Block Compressed Sparse Row (BCSR) representation from



```
Input Loop:
for(i=0; i < n;i++)
  for(j=index[i];j<index[i+1];j++)
          y[i]+=a[j]*x[col[j]]
```

```
for(i=0; i < n;i++)
  for(k=0;k < n;k++)
    for(j=index[i];j<index[i+1];j++)
        if(k == col[j])
            y[i]+=a[j]*x[k]
```

```
for(ii=0; ii < n/r;ii++)
 for(kk=0;kk < n/c;kk++)
   for(i=0; i < r;i++)
     for(k=0;k < c;k++)
       for(j=index[ii*r +i];j<index[ii*r + i+1];j++)
          if(kk*c + k == col[j])
            y[ii*r + i]+=a[j]*x[kk*c + k]
```

```
Inspector Code:
for(ii=0; ii < n/r;ii++){
//reset marked to false (code not shown)
for(i=0; i < r;i++)
  for(j=index[ii*r +i];j<index[ii*r + i+1];j++){
    kk = col[j]/c; k =col[j]/c – kk*c;
      if(marked[kk] == false){
          marked[kk] = true;
          explicit_index[kk] = count;
          //initialize a'[count][0-r][0-c] to 0
          count++;}
      a'[count][i][k] = a[j];}
 offset_index[ii+1] = count;
}
```

```
Executor Code :
for(ii=0; ii < n/r;ii++)
 for(kk=offset_index[ii];kk < offset_index[ii+1];kk++)
   for(i=0; i < r;i++)
     for(k=0;k < c;k++)
       y[ii*r + i]+=a'[kk][i][k]*x[explicit_index[kk]*c + k]
```

Figure 3: BCSR inspector/executor code.

the initial SpMV computation based on the Compressed Sparse Row (CSR) representation. The BCSR representation is ideal for exploiting temporal reuse of matrix column values via registers on multi cores for performance. Conceptually it tiles the matrix into small rectangular tiles of length r and width c, and stores non-zeros that fall into adjacent locations in the same tile. Since the matrix is sparse, if some locations in the tile do not have a corresponding non-zero, they are padded with a zero value. Totally empty tiles are not stored.

The BCSR example serves to highlight the composition of these inspector/executor transformations with other regular transformations, such as tiling, within CHiLL.

As shown in Figure 3, the *make-dense* is applied on the expression *col[j]* in order to expose the column dimension of the matrix as an explicit loop dimension. The transformation identifies the upper and lower bounds on the expression and introduces a loop level with the same bounds. Additionally it eliminates the indirect access due to *col[j]* with a reference to the newly introduced loop. Also it introduces a guard to check if non-zero exists at a particular value of the newly introduced loop for correctness.

This allows regular transformations such as tiling to be applied on the newly introduced loop. The loop that traverses the rows of the matrix, and the newly introduced loop that corresponds to the column dimension, are then both tiled by constant factors r and c to derive the small rectangular tiles in the BCSR representation.

Finally *compact-and-pad* is called on the loop level that iterates over the rectangular tiles. For a given iteration of this loop, if the entire tile enclosed fails to satisfy the guard, or equivalently if there is no non-zero in the region, that tile is entirely discarded from the iteration space and associated data representation. On other regions, where the guard is satisfied for a subset of iterations, the original data is copied to the corresponding new location, and on those that do not satisfy the guard a zero is inserted.

The inspector for *compact-and-pad* also populates the *offset_index* and *explicit_index* arrays. *offset_index* determines the start and end of the rectangular tiles that belong to the same set of adjacent rows of length r, while *explicit_index* records the actual values of the column offset of each block. Additionally, the code generator uses the guard condition to derive closed form expressions of certain loops, as alluded to previously, to minimize the overhead of the inspector.

5. STENCIL OPTIMIZATIONS

For almost all stencils, there is data and operation reuse between neighboring points. This reuse is more significant for high-order stencils, which examine more neighboring input points to compute each output point. We extend our compiler framework with partial sum transformation [2] to exploit this reuse to reduce loads, and remove operations that are redundant across multiple output calculations. The partial sum transformation is applicable to the class of input codes where all subscript expressions are *affine*, or linear combinations of loop indices and loop-invariant variables. Also the partial sum transformation requires that the subscript expressions are *separable*, such that each dimension references just a single loop index.

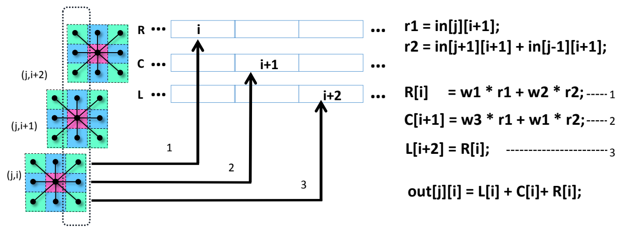
5.1 Stencil Reordering: Partial Sums

The partial sum transformation described in this paper targets constant-coefficient, out-of-place stencils. That is, in this paper, we assume a stencil value is a weighted sum of a single array, and updates are loop nest computations where the right-hand sides are read-only arrays per stencil sweep (e.g. Jacobi).

For an illustration of computing stencils via partial sums, consider the 9-point 2D stencil of Figure 4 (top). We observe three types of reuse: (1) data reused across iterations, where input points for the *right edge* of iteration $\langle j, i\rangle$, are reused as the *center* for iteration $\langle j, i+1\rangle$ and the *left edge* for iteration $\langle j, i+2\rangle$; (2) computation reuse based on symmetry of the coefficients along the $j-$axis, where entries R[i] and L[i+2] are equal; (3) data reuse based on symmetry of coefficients along the $i-$axis, where the sum of points with same coefficient are stored in $r1$ and $r2$. Therefore, the compiler constructs an array of coefficients to be used in the partial sum transformation, and computes and stores partial results in partial sums. Finally, partial sums are buffered across iterations to derive the output values.

In a similar manner, for 3D stencils we compute partial sums of 2D planes instead of 1D lines. Consider 3D 27-point stencil from Listing 1 as an example, with CHiLL transfor-

```
for (j=0; j<N; j++)
  for (i=0; i<N; i++)
    out[j][i] = w1*(
                     in[j-1][i] + in[j+1][i] +
                     in[j][i-1] + in[j][i+1]
                 ) +
                 w2*(
                     in[j-1][i-1] + in[j+1][i-1] +
                     in[j-1][i+1] + in[j+1][i+1]
                 ) +
                 w3*( in[j][i] );
```



R
C
L
i
i+1
i+2
(j,i+2)
(j,i+1)
(j,i)
1
2
3
r1 = in[j][i+1];
r2 = in[j+1][i+1] + in[j-1][i+1];
R[i] = w1 * r1 + w2 * r2; 1
C[i+1] = w3 * r1 + w1 * r2; 2
L[i+2] = R[i]; 3
out[j][i] = L[i] + C[i]+ R[i];

Figure 4: Input code for 2D 9-point stencil (top). Partial sum optimization (bottom).

```
#define N 64
void stencil(){
int k,j,i;
for (k=0; k<N; k++)
 for (j=0; j<N; j++)
  for (i=0; i<N; i++){
   out[k][j][i] =  w1*( in[k][j][i] ) +
    w2*( in[k-1][j][i] + in[k][j-1][i]
       + in[k][j+1][i] + in[k+1][j][i]
       + in[k][j][i-1] + in[k][j][i+1] ) +
    w3*( in[k-1][j][i-1] + in[k][j-1][i-1]
       + in[k][j+1][i-1] + in[k+1][j][i-1]
       + in[k-1][j-1][i] + in[k-1][j+1][i]
       + in[k+1][j-1][i] + in[k+1][j+1][i]
       + in[k-1][j][i+1] + in[k][j-1][i+1]
       + in[k][j+1][i+1] + in[k+1][j][i+1] ) +
    w4*( in[k-1][j-1][i-1] + in[k-1][j+1][i-1]
       + in[k+1][j-1][i-1] + in[k+1][j+1][i-1]
       + in[k-1][j-1][i+1] + in[k-1][j+1][i+1]
       + in[k+1][j-1][i+1] + in[k+1][j+1][i+1] );
   }}
```

Listing 1: 3D 27-point stencil.

```
original()
skew([0,1,2,3,4,5],2,[2,1])
permute([2,1,3,4])
distribute([0,1,2,3,4,5],2)
partial_sums(0)
partial_sums(5)
fuse([2,3,4,5,6,7,8,9],1)
fuse([2,3,4,5,6,7,8,9],2)
fuse([2,3,4,5,6,7,8,9],3)
fuse([2,3,4,5,6,7,8,9],4)
```

Listing 2: CHiLL script for 3D 27-point stencil.

```
// distance of farthest stencil point
// from origin per dimension
int radius = 1;
// allocate 2*radius+1 buffered partial sums,
// N is grid (box) dimension
// create (radius+1)*(radius+2)/2 temporaries
double B0[N], B1[N], B2[N];
double r1, r2, r3;

for (k=0; k<N; k++){
 for (j=0; j<N; j++){
  // preamble code sets up the pipeline
  ....
  // steady state computation
 for(i=0; i<(N-radius); i++){
  r1 = in[k][j][i+1];
  r2 = in[k+1][j  ][i+1] + in[k-1][j  ][i+1] +
       in[k  ][j-1][i+1] + in[k  ][j+1][i+1];
  r3 = in[k+1][j+1][i+1] + in[k+1][j-1][i+1] +
       in[k-1][j+1][i+1] + in[k-1][j-1][i+1];
  B2[i] = w2*r1 + w3*r2 + w4*r3;
  B1[i+1] = w1*r1 + w2*r2 + w3*r3;
  B0[i+2] = B2[i];
 }
 for(i=0; i<(N-radius); i++)
  out[k][j][i] = B0[i] + B1[i] + B2[i];
 ...
 // cleanup code to avoid extra computation
 ...
 }}
```

Listing 3: Optimized code for 3D 27-point stencil.

mation script provided in Listing 2. The simplified generated code is shown in Listing 3. Computation reuse are exploited though partial sum buffers, with three buffers allocated, for the left, center and right planes. Data reuse are exploited through scalars, three scalars are created for three unique coefficients in each 2D plane.

5.2 Compiler Abstractions and Code Generation

Abstractions of stencil points, bounding box, coefficients, and buffer for partial sums are derived automatically by our compiler and used by the code generator to produce code in Listing 3. AST transformations are applied to the statement *code*: (1) create buffer objects and scalars to hold sums for each unique coefficient in a plane; (2) create a new compound statement to compute buffers; (3) create new statement to compute output from the sum of all buffers, and replace the original statement with this statement. Subsequent polyhedral transformations are required: (1) decrease the number of iterations of IS to avoid going off the end of the buffers; (2) create a new iteration space for new statement; and, (3) peel off remaining iterations and use original statement. New nodes are created in the AST, with the AST structure reparsed and the dependence graph updated. We find that reparsing and rebuilding the dependence graph is preferable to incremental updates due to the fundamental changes to the original code.

5.3 Composing Transformations

After introducing partial sums, compute-bound kernels become more memory bound, and communication-avoiding optimizations are then used to further improve performance, such as overlapped tiling (via larger ghost zones), loop fusion and wavefronts. Overlapped tiling reduces inter-processor communication. Loop fusion and wavefront computation

```
void MM(int c[N][N], int a[N][N], int b[N][N]) {
  int i, j, k;
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      for (k = 0; k < N; k++)
        c[j][i] = c[j][i] + a[k][i] * b[j][k]; }
```

Listing 4: Matrix multiply source code (BLAS).

```
N = 1024
Ti = 128, Tj = 64
Tk = 16
Tii = 16, Tjj = 16

tile_by_index(0,{"i","j"},{Ti,Tj},
             {l1_control="ii",l2_control="jj"},
             {"ii","jj","i","j","k"})CU=1
tile_by_index(0,{"k"},{Tk},{l1_control="kk"},
             {"ii","jj","kk","i","j","k"})CU=3
tile_by_index(0,{"i","j"},{Tii,Tjj},
             {l1_control="iii",l2_control="jjj"},
             {"ii","jj","kk","i","iii","j","jjj",
                  "k"},1)CU=2
cudaize(0,"mm_GPU",{},
        {block={"ii","jj"},thread={"i","j"}},{})
copy_to_shared(0,"tx","a",-16)
copy_to_shared(0,"tx","b",-16)
copy_to_registers(0,"kk","c")
unroll_to_depth(2)
```

Listing 5: Matrix multiply CUDA-CHiLL script.

reduce communication to DRAM by fusing multiple grid sweeps into one. Wavefronts are generated by loop skewing followed by loop permutation. Skewing eliminates some dependences making permutation legal. The loop skew factor must increase with stencil radius. Wavefront exploits reuse but increases the working set, which may result in spills from faster caches. CHiLL can generate code with nested loops and OpenMP directives to reduce the working set per thread. When used together, partial sums and communication avoiding optimizations can achieve substantial performance gains for high-order stencils.

6. PARALLEL CODE GENERATION

While parallel code generation modifies the AST representation, it is done during code generation, and is therefore not composable with other polyhedral transformations. Here we describe the interplay between polyhedra scanning code generation and the AST modifications that result from generating CUDA or OpenMP code.

6.1 CUDA

The CUDA-CHiLL compiler maps from a sequential program view to the block/thread view of CUDA, as described in [16, 12]. The AST modifications are as follows: (1) eliminate loops corresponding to block and thread dimensions; (2) replace references to these indices with corresponding CUDA block and thread indices; (3) introduce datacopy and synchronization for GPU shared memory; and, (4) construct function calls for kernel launch, device and host data movement. We now consider matrix multiply as an example, from Listing 4. With transformation script in Listing 5, generated code in Listing 6.

```
  ...
  cudaMalloc
  cudaMemcpy
  dim3 dimGrid0 = dim3(8,16);
  dim3 dimBlock0 = dim3(16,16);
  mm_GPU<<<dimGrid0,dimBlock0>>>(...);
  cudaMemcpy
  cudaFree
  ...
__global__ void mm_GPU(...)
{
  bx = blockIdx.x; by = blockIdx.y;
  tx = threadIdx.x; ty = threadIdx.y;
  __device__ __shared__ int _P1[128][17];
  __device__ __shared__ int _P2[16][65];
  int _P3[4][8];

  _P3[0][0] = c[64*by+ty][128*bx+tx];
  _P3[1][0] = c[64*by+ty+16][128*bx+tx];
  _P3[2][0] = c[64*by+ty+32][128*bx+tx];
  _P3[3][0] = c[64*by+ty+48][128*bx+tx];
  ...
  for (kk = 0; kk <= 63; kk += 1) {
    _P1[tx][ty] = a[ty+16*kk][tx+128*bx];
    _P1[tx+16][ty] = a[ty+16*kk][tx+128*bx+16];
    ...
    _P1[tx+112][ty] = a[16*kk+ty][128*bx+tx+112];
    __syncthreads();
    _P2[tx][ty] = b[64*by+ty][16*kk+tx];
    _P2[tx][ty+16] = b[64*by+(ty+16)][tx+16*kk];
    _P2[tx][ty+32] = b[64*by+(ty+32)][tx+16*kk];
    _P2[tx][ty+48] = b[64*by+(ty+48)][16*kk+tx];
    __syncthreads();
    for (k = 0; k <= 15; k += 1) {
      _P3[0][0] = _P3[0][0]+_P1[tx][k]*_P2[k][ty];
      _P3[1][0] = _P3[1][0]+_P1[tx][k]*_P2[k][ty+16];
      _P3[2][0] = _P3[2][0]+_P1[tx][k]*_P2[k][ty+32];
      _P3[3][0] = _P3[3][0]+_P1[tx][k]*_P2[k][ty+48];
      ...
      __syncthreads();
    }
    __syncthreads();
  }
  c[64*by+ty][128*bx+tx] = _P3[0][0];
  c[64*by+ty+16][128*bx+tx] = _P3[1][0];
  c[64*by+ty+32][128*bx+tx] = _P3[2][0];
  c[64*by+ty+48][128*bx+tx] = _P3[3][0];
  ...
}
```

Listing 6: Generated CUDA kernel code from script.

6.1.1 *Computation Partitioning*

Computation partitioning into CUDA blocks and threads involves tiling and permutation through polyhedral loop transformations. In CUDA-CHiLL, loops corresponding to parallel block and thread dimensions are effectively removed from the generated code; AST annotation of block and thread loops result from using *cudaize* as in Listing 5. CUDA-CHiLL passes the iteration space and mapping to CodeGen+ and receives the AST of the loop code from CodeGen+. CUDA-CHiLL then extracts the annotation inserted by CodeGen+ to identify parallel loops and reduces the loops.

Consider line 6 from Listing 5, where loop `i` is tiled by size `Ti`, resulting tile controlling loop `ii`, and tile loop `i`. Loop `ii` maps exactly to the block `x` dimension. All block and thread loops are eliminated similarly. As a result, only loops {`kk,iii,jjj,k`} remain in the generated CUDA kernel, which is further optimized with GPU memory hierarchy optimizations. Thus, the loops are marked for elimination, but their polyhedral and AST abstractions remain until code generation.

```
//Laplacian(phi) = b div beta grad phi
for (k=0;j<N;k++)
 for (j=0;j<N;j++)
  for (i=0;i<N;i++)
  /* statement S0 */
  temp[k][j][i] = b*h2inv*(
  beta_i[k][j][i+1]*(phi[k][j][i+1]-phi[k][j][i])
  -beta_i[k][j][i]*(phi[k][j][i]-phi[k][j][i-1])
  +beta_j[k][j+1][i]*(phi[k][j+1][i]-phi[k][j][i])
  -beta_j[k][j][i]*(phi[k][j][i]-phi[k][j-1][i])
  +beta_k[k+1][j][i]*(phi[k+1][j][i]-phi[k][j][i])
  -beta_k[k][j][i]*(phi[k][j][i]-phi[k-1][j][i]));

//Helmholtz(phi) = (a alpha I - laplacian)*phi
for (k=0;j<N;k++)
 for (j=0;j<N;j++)
  for (i=0;i<N;i++)
  /* statement S1 */
  temp[k][j][i] = a*alpha[k][j][i]*phi[k][j][i]-
                  temp[k][j][i];

//GSRB relaxation: phi = phi - lambda(helmholtz-rhs)
for (k=0;j<N;k++)
 for (j=0;j<N;j++)
  for (i=0;i<N;i++){
  if ((i+j+k+color)%2==0)
  /* color is 0 for Red pass, 1 for black */
  /* statement S2 */
  phi[k][j][i] = phi[k][j][i]-lambda[k][j][i]*
                 (temp[k][j][i]-rhs[k][j][i]);}
```

Listing 7: Smooth operator with GSRB.

Similarly, CodeGen+ annotates loop levels with preferred index names as comments to the AST of loop structures. CUDA-CHiLL then extracts the annotation inserted by CodeGen+ to recursively replace block and thread indices in the statement code. For example, loop iterators of block control loop `ii`, `jj` are replaced with `blockIdx.x` and `blockIdx.y`.

6.1.2 *Datacopy and Synchronization*

Data destined for shared memory must be explicitly copied to/from the device global memory, as shown in lines 16-18 in the transformation script from Listing 5. Therefore, we couple tiling with explicit copying of data and associated synchronization to perform the data staging into shared memory. Synchronization (i.e., calls to *__syncthreads()*) are added in conjunction with datacopy. These calls are represented as annotations in the loop's AST structure.

6.1.3 *Outlining Kernel Function*

After CUDA-CHiLL obtains the loop nest AST from CodeGen+, the transformed loop nest is outlined as a CUDA kernel. The original function body is replaced with a function call to this CUDA kernel. Auxiliary functions are added to marshal inputs and outputs.

6.2 OpenMP

CHiLL supports OpenMP code generation using OpenMP pragmas for parallel regions (`#pragma omp parallel`) and parallel fors (`#pragma omp for`). In this discussion, we limit

```
for (k=-3; k<=66; k++)
  for (t=0; t<=min(3,intFloor(t+3,2)); t++) {
    for (j=t-3; j<=-t+66; j++)
      for (i=t-3+intMod(-k-color-j-(t-3),2); i<=-
          t+66; i+=2) {
        S0(t,k-t,j,i); /* Laplacian */
        S1(t,k-t,j,i); /* Helhmoltz */
        S2(t,k-t,j,i); /* GSRB      */
  }}
```

Listing 8: CHiLL generated wavefront for a GSRB stencil computation.

```
#pragma omp parallel private (...) num_threads(y)
{
tid=omp_get_thread_num();
for (k=-3; k<=66; k++) {
 for (t=0; t<=min(3,intFloor(t+3,2)); t++) {
  for (j=6*tid-3; j<=min(6*tid+2,66); j++) {
   for (i=t-3+intMod(-k-color-j-(t-3),2); i<=-t
        +66; i+=2) {
      S0(t,k-t,j,i); /* Laplacian */
      S1(t,k-t,j,i); /* Helhmoltz */
      S2(t,k-t,j,i); /* GSRB      */
    }}}
   //Explicit Spin Lock
   //Can also use OMP_Barrier
  zplanes[tid] = t2;
  if (left != tid)
  {while(zplanes[left] < t2)
    { _mm_pause();}} else{}
  if (right != tid)
  {while(zplanes[right] < t2)
    {_mm_pause();}}
 }//end k
}
```

Listing 9: CHiLL generated threaded wavefront using OpenMP and explicit spin locks for point-to-point synchronization.

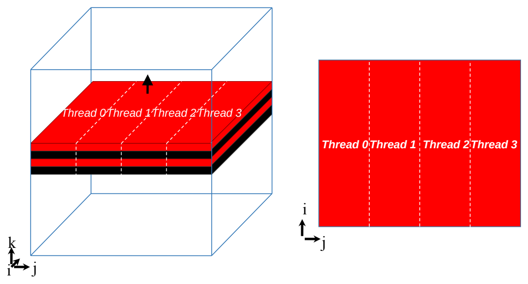


Figure 5: Parallel threaded wavefront computation. (left) Four OMP threads processing a grid and synchronizing after four output planes are computed. (right) Each 2D plane is portioned amongst four threads.

ourselves to OMP parallel region, which is the more challenging of the two, and resembles the approach taken with CUDA code generation.

OpenMP code generation is explained in the context of a wavefront computation. 3D stencil computations involve a triply-nested loop which sweeps through 3D grids (arrays). If the stencil computation is applied multiple times to the grids, there is an outer, fourth loop. This is often called a time step loop, and it creates multiple stencil sweeps of the grids. Generating wavefronts is a known technique that fused the multiple grid sweeps caused by the time step loop into a single sweep. Wavefronts are created by loop skewing followed by loop permutation. Listing 8 illustrates the loop structure of a wavefront for stencil computation involving a 7-point stencil and Gauss-Seidel Red-Black (GSRB) updates, with placeholders for the statements corresponding to Listing 7. This stencil computation prior to wavefront has a loop order {`t`,`k`,`j`,`i`}; `i` being the dimension of unit stride, `k` has the largest stride, and `t` is the time-step loop. The loop order after creating a wavefront is {`k`,`t`,`j`,`i`}. To legally permute `k` and `t`, the compiler first has to skew `k` against `t`. Unfortunately, creating wavefront computations increase the working set.

To ensure the working set doesn't spill from the fast L1/L2 caches, we use OpenMP to perform thread blocking. We strip mine (tile) the `j` loop, and assign each strip to an OpenMP thread. The `j` loop in the loop nest is tiled using CHiLL's tiling capabilities. This decomposes it into two, loop `j` and the loop controlling `jj` loop. The tile controlling loop is then hoisted outside the time step loop (this is legal for the Gauss-Seidel Red-Black updates, as it doesn't break data dependencies), and the final loop order is {`k`,`jj`,`t`,`j`,`i`}. The `j` loop has updated loop bounds which are functions of `jj`. CHiLL's scripting language interface is then used to mark the `jj` loop to be assigned to OpenMP threads. This is very similar in spirit to CUDA code generation, where loops were tiled, and then certain loop levels were marked as threads and thread blocks.

Once all loop transformations have been applied, CHiLL scans the polyhedra to create the AST for the output code. While creating the AST, if the OpenMP code generation flag is set, CHiLL wraps the entire loop in an AST node associated with `#pragma omp parallel`. It then creates and adds another AST node for the OMP private clause. The loop level (`jj`), that was marked as OpenMP thread, is then removed from the AST representation of the loop nest. Furthermore, all reference to the loop index `jj` in the body of the loop are replaced by the variable `tid`. This means that bounds for loop `j` are now functions of `tid`. A declaration for `tid`, and a statement to set it's value via an OpenMP call is added to the function body. When creating the AST, as loops are being added from outer to the inner one, the indices for loops nested inside the parallel loop `jj` are added to the omp private clause.

To ensure correctness, thread synchronization is appended to body of the loop that surrounds the loop level marked to omp threads, analogous to adding *syncthreads* in CUDA code generation. In this case, it is appended to the body of loop `k`. To reduce the overhead of an OpenMP barrier, code generation was extended to add nodes to AST to support point-to-point synchronization between neighboring threads via spinlocks. The threaded wavefront code for the GSRB stencil computation is illustrated in Listing 9, and visualized in Figure 5. The code shows a threaded wavefront for a GSRB stencil computation on a $(64 + 8)^3$ grid with 6 threads, and the code snippet after the stencil computation implements a spinlock using the shared volatile array `zplanes`.

7. RELATED WORK

Polyhedral frameworks are often restricted to affine computations and employ only rewriting of the iteration space and array subscripts. Other authors have pointed out that the following transformations are difficult in polyhedral frameworks: index set splitting [8], piecewise schedules [9], time-tiling of periodic stencils [4], register tiling (unroll-and-jam plus scalar promotion) [15]. For more complex optimizations, AST transformations are often introduced as post-processing outside of polyhedral framework [11, 5]. In the CHiLL compiler, since AST structure is encapsulated within the polyhedral abstractions, the dependence graph is constructed from both the polyhedral and AST representations, and shared among polyhedral and AST transformations.

Shirako et al. [19] proposed a decoupled framework, separating polyhedral transformations and AST optimizations into different stages. However, dependences need to be extracted from the polyhedral framework, to perform legality analysis for AST transformations. The authors argued that separate AST transformations are necessary to detect proper parallelism, pipeline parallelism for example, which is typically implemented as inefficient wavefront schedules in polyhedral framework. However, we showed that pipeline parallelism in OpenMP can be implemented in CHiLL because of the modifications to the AST, resulting in a coarse-grain threaded wavefront, with global synchronizations optimized with point-to-point synchronizations via spinlocks, and start-up/draining overhead reduced. Moreover, the polyhedral phase proposed by this paper requires loops to have affine controls, leaving non-affine transformations to AST phases. We showed that we can extend the polyhedral framework with support of non-affine constraints, enabling a wider range of applications.

Grosser et al. [10] presented an integrated AST generation approach, that enables AST optimizations through arbitrary user-provided AST expressions, within the process of scanning polyhedra. User-supplied AST expressions enable optimizations of modulo operations, piecewise schedules, memory layout transformations, and multiple AST generation strategies for different AST subtrees. However, this AST optimization approach is limited. Optimizing through AST expressions may fail to see the subtle interactions between a suite of transformations. Moreover, for more complex optimizations, such as generating inspector/executor for irregular applications, complicated data structures are hard to express through user-defined AST expressions.

8. CONCLUSION

In this paper, we present case studies that demonstrate that the composability of polyhedral framework is preserved from our polyhedral and AST abstractions. Mixing such AST transformations within the polyhedral model avoids the effort of switching back to the AST and harnesses the power of composability of polyhedral frameworks. The future of such technology will demand that it can adapt to the needs of a broad class of applications, and a contribution of this paper is pointing to existing support for challenging applications.

9. REFERENCES

[1] C. 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*, pages 7–16. IEEE Computer Society, 2004.

[2] P. Basu, M. Hall, S. Williams, B. V. Straalen, L. Oliker, and P. Colella. Compiler-directed transformation for higher-order stencils. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 313–323. IEEE, 2015.

[3] P. Basu, A. Venkat, M. Hall, S. Williams, B. Van Straalen, and L. Oliker. Compiler generation and autotuning of communication-avoiding operators for geometric multigrid. In *High Performance Computing (HiPC), 2013 20th International Conference on*, pages 452–461. IEEE, 2013.

[4] U. Bondhugula, V. Bandishti, A. Cohen, G. Potron, and N. Vasilache. Tiling and optimizing time-iterated computations on periodic domains. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 39–50, New York, NY, USA, 2014. ACM.

[5] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. Pluto: A practical and fully automatic polyhedral program optimization system. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008)*. Citeseer, 2008.

[6] C. Chen. *Model-guided empirical optimization for memory hierarchy*. PhD thesis, University of Southern California, 2007.

[7] C. Chen. Polyhedra scanning revisited. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 499–508, New York, NY, USA, 2012. ACM.

[8] M. Griebl, P. Feautrier, and C. Lengauer. Index set splitting. *Int. J. Parallel Program.*, 28(6):607–631, Dec. 2000.

[9] T. Grosser, A. Cohen, J. Holewinski, P. Sadayappan, and S. Verdoolaege. Hybrid hexagonal/classical tiling for gpus. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 66:66–66:75, New York, NY, USA, 2014. ACM.

[10] T. Grosser, S. Verdoolaege, and A. Cohen. Polyhedral ast generation is more than scanning polyhedra. *ACM Trans. Program. Lang. Syst.*, 37(4):12:1–12:50, July 2015.

[11] A. Hartono, M. M. Baskaran, C. Bastoul, A. Cohen, S. Krishnamoorthy, B. Norris, J. Ramanujam, and P. Sadayappan. Parametric multi-level tiling of imperfectly nested loops. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 147–157, New York, NY, USA, 2009. ACM.

[12] M. Khan, P. Basu, G. Rudy, M. Hall, C. Chen, and J. Chame. A script-based autotuning compiler system to generate high-performance cuda code. *ACM Trans. Archit. Code Optim.*, 9(4):31:1–31:25, Jan. 2013.

[13] R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nico, and K. Crowley. Principles of runtime support for parallel processors. In *Proceedings of the 2nd International Conference on Supercomputing*, pages 140–152, 1988.

[14] D. J. Quinlan. ROSE: compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(2/3):215–226, 2000.

[15] L. Renganarayana, U. Bondhugula, S. Derisavi, A. E. Eichenberger, and K. O'Brien. Compact multi-dimensional kernel extraction for register tiling. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 45. ACM, 2009.

[16] G. Rudy. *CUDA-CHiLL: A programming language interface for GPGPU optimizations and code generation*. PhD thesis, The University of Utah, 2010.

[17] G. Rudy, M. M. Khan, M. Hall, C. Chen, and J. Chame. A programming language interface to describe transformations and code generation. In *Proceedings of the 23rd International Conference on Languages and Compilers for Parallel Computing*, LCPC'10, pages 136–150, Berlin, Heidelberg, 2011. Springer-Verlag.

[18] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *J. Parallel Distrib. Comput.*, 8(4):303–312, Apr. 1990.

[19] J. Shirako, L.-N. Pouchet, and V. Sarkar. Oil and water can mix: An integration of polyhedral and ast-based transformations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 287–298, Piscataway, NJ, USA, 2014. IEEE Press.

[20] A. Venkat, M. Hall, and M. Strout. Loop and data transformations for sparse matrix code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 521–532. ACM, 2015.

[21] A. Venkat, M. Shantharam, M. Hall, and M. M. Strout. Non-affine extensions to polyhedral code generation. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 185:185–185:194, New York, NY, USA, 2014. ACM.

[22] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor. Polyhedral parallel code generation for cuda. *ACM Trans. Archit. Code Optim.*, 9(4):54:1–54:23, Jan. 2013.