Loop and Data Transformations for Sparse Matrix Code

Anand Venkat Mary Hall
University of Utah, USA
{anandv,mhall}@cs.utah.edu

Michelle Strout
Colorado State University, USA
mstrout@cs.colostate.edu

Abstract

This paper introduces three new compiler transformations for representing and transforming sparse matrix computations and their data representations. In cooperation with run-time inspection, our compiler derives transformed matrix representations and associated transformed code to implement a variety of representations targeting different architecture platforms. This systematic approach to combining code and data transformations on sparse computations, which extends a polyhedral transformation and code generation framework, permits the compiler to compose these transformations with other transformations to generate code that is on average within 5% and often exceeds manually-tuned, high-performance sparse matrix libraries CUSP and OSKI. Additionally, the compiler-generated inspector codes are on average 1.5× faster than OSKI and perform comparably to CUSP, respectively.

Categories and Subject Descriptors D.3 [*Programming Languages*]; D.3.4 [*Processors*]: Code generation, Compilers, Optimization

General Terms Languages, Performance

Keywords sparse matrices, non-affine, inspector/executor, polyhedral model, loop transformations

1. Introduction

Sparse matrix computations represent an important class of algorithms that arise frequently in numerical simulation and graph analytics. To reduce computation and storage requirements, sparse matrix representations attempt to store only the nonzero data elements, with indirection matrices to make it possible to determine the location of the nonzero in the corresponding dense matrix. Because of irregular and unpredictable memory access patterns, sparse matrix computations are notoriously memory bound, and this problem is getting worse on modern architectures where cost of data movement dwarfs computation. To address these performance challenges, in recent years there has been a significant body of research on new sparse matrix representations and their implementations that specialize for particular application domains and new architectures [7, 8, 25, 27, 29, 40, 45, 48].

PLDI'15, June 13–17, 2015, Portland, OR, USA
© 2015 ACM. 978-1-4503-3468-6/15/06...$15.00
http://dx.doi.org/10.1145/2737924.2738003

A common approach to encapsulate high-performance implementations of new sparse matrix representations and convert to them from standard representations involves using a manually-tuned library. Sparse matrix and graph libraries employ a variety of sparse matrix representations to exploit whatever structure is present in the nonzeros of the input matrix [5, 8, 14, 26, 29, 46]. For example, the specific representations BCSR, DIA and ELL considered in this paper, all widely used in libraries, insert a small number of zero-valued elements into the sparse matrix representation, which has the effect of increasing computation but making memory access patterns and generated code more regular and efficient. A purely library approach has several weaknesses: (1) there are dozens of sparse matrix representations in common use, and many new ones being developed, and the code for each representation must be manually optimized; (2) libraries must be manually ported to new architectures; and, (3) a library encapsulates individual functions that cannot be composed in an application (e.g., only performs a single sparse matrix-vector multiply or computation of similar scope).

While ideally a compiler can be used to provide generality, architecture portability, and composability with other transformations, compilers have been severely limited in their ability to optimize sparse matrix computations due to the indirection that arises in indexing and looping over just the nonzero elements. This indirection gives rise to *non-affine* subscript expressions and loop bounds; i.e., array subscripts and loop bounds are no longer linear expressions of loop indices. A common way of expressing such indirection is through *index arrays* such as, for example, array `B` in the expression `A[B[i]]`. Code generators based on polyhedra scanning are particularly restricted in the presence of non-affine loop bounds or subscripts [3, 15, 21, 35, 43]. As a consequence, most parallelizing compilers either give up on optimizing such computations, or apply optimizations very conservatively.

In this paper, we develop and evaluate novel compiler transformations and automatically-generated run-time inspectors that make it possible to integrate loop and data transformations on sparse matrices into a polyhedral transformation and code generation framework. Some compiler approaches begin with a dense abstraction of a sparse matrix computation; these compilers then generate sparse data representations during code generation, placing a significant burden on the compiler to optimize away the sometimes orders of magnitude difference in performance between dense and sparse implementations [10, 28, 34]. To our knowledge, the only prior compiler approach that starts with a sparse computation and derives new sparse matrix representations is that of Wijshoff et al. [42]. They convert code with indirect array accesses and loop bounds into dense loops that can then be converted into sparse matrix code using the MT1 compiler [9, 12]. Sublimation is more restrictive, and their work does not compare performance attained with manually-tuned library implementations of either inspector or executor for specific sparse matrix representations.

Other closely related work combines run-time inspection with non-affine representations in a polyhedral transformation and code generation framework, making possible non-affine transformations and composability with standard transformations [44]. Our work relies on the same techniques to extend polyhedral compiler technology, but additionally it systematically introduces new data representations for the sparse matrices and the associated code transformations to use the new representations.

For this purpose, we introduce three key transformations: *make-dense* derives an equivalent computation that mimics the iteration space of a dense computation, permitting downstream transformations to be applied; *compact* converts a dense iteration space back to a non-affine, sparse one, after the desired transformations have been performed; and, *compact-and-pad* additionally transforms the data representation. A run-time inspector is used to locate nonzeros and optionally perform the data transformation. A key technical challenge in this work is to automatically generate inspectors that do not introduce significant run-time overhead.

This paper makes the following contributions: (1) develops new compiler transformations for codes with indirection through index arrays that facilitates code and data transformations; (2) incorporates these into a polyhedral transformation and code generation framework so that these transformations can be composed with other standard ones; and, (3) demonstrates that the performance of the compiler-generated inspector and executor code performs comparably to a manually-tuned specialized library: CUSP for GPUs [8], and OSKI [46] for multicore processors.

The remainder of this paper is organized into five sections and conclusion. We describe the transformations and inspector in the next section, followed by a description of the compiler implementation. Section 5 applies the system to derive three common sparse matrix representations and their associated code, and the performance of the new representation is presented in the subsequent section. Section 7 describes prior work in more detail.

2. Background and Motivation

Transformations on sparse codes with indirect array accesses require the generation of inspector code. In this section we describe inspector/executor paradigms and how they can be combined into polyhedral frameworks. We review the sparse matrix formats used in this paper, and motivate the development of a small number of loop transformations that can be combined with polyhedral transformations to create optimized implementations for a number of sparse matrix formats.

2.1 Inspector/Executor

A general technique to analyze data accesses through index arrays and consequently reschedule or reorder data at run time employs an *inspector/executor* paradigm whereby the compiler generates *inspector* code to be executed at run-time that can collect the index expressions and then an *executor* employs specific optimizations that incorporate the run-time information [6, 31, 36, 37]. These inspector/executor optimizations have targeted parallelization and communication [36, 39] and data reorganization [17, 20, 30, 32, 50].

In this paper, we use the inspector for two purposes: to identify the locations of the nonzero entries in the sparse matrix, and optionally, to perform the data transformation. The executor represents the optimized computation on the transformed data.

2.2 Incorporating into a Polyhedral Framework

Polyhedral frameworks describe the iteration space for each statement in a loop nest as a set of lattice points of a polyhedron. Loop transformations can then be viewed as mapping functions

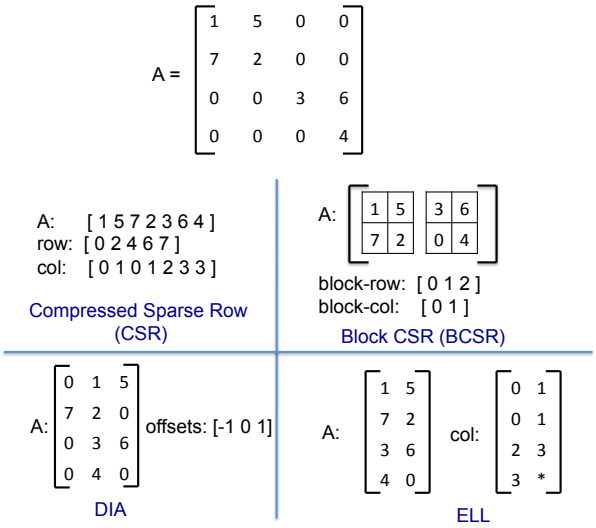


Figure 1: Sparse matrix representations.

that convert the original iteration space to a transformed iteration space providing the compiler a powerful abstraction to transform a loop nest without being restricted to the original loop structure [18]. A separate dependence graph abstraction is used to prevent applying individual transformations that do not preserve the dependences in the program. The code corresponding to the transformed iteration space may then be generated by polyhedra scanning [3, 15, 21, 35, 43].

This transformation and code generation approach, widely used in existing tools, is limited by the fact that the iteration space sets and transformation relations must all have affine constraints. We rely on the prior work in [44], which extends a polyhedral framework to tolerate and manipulate non-affine loop bounds and array access expressions using the abstraction of *uninterpreted function symbols*, expanding on their use in Omega [21].

Strout et al. [41] extend the polyhedral model to represent data and iteration reordering transformations on sparse codes, and they present a prototype code generator for composed inspectors and executors. These inspectors are more general but presumably less performant than the specific inspector/executor transformations we incorporate into an existing polyhedral framework in this work.

2.3 Sparse Matrix Formats

This paper primarily considers four sparse matrix representations, as captured in Figure 1, and defined as follows:

- *Compressed Sparse Row (CSR):* CSR, which is used by the code in Listing 1, represents only the nonzero elements in flattened matrix `A`. The `col` array indicates the column corresponding to each nonzero, but the `row` array has one element per row, indicating the location of the first element for that row.
- *Block CSR (BCSR):* In BCSR format, the nonzero elements are represented by a collection of small dense blocks, and the blocks are padded where necessary with zero values. An auxiliary array tracks the row and column of the upper left element of each nonzero block. The resulting interior computation can be performed on a small dense array, for which high-performance implementations are more easily obtained.
- *DIA:* The DIA format captures only the diagonals that have nonzero elements. The `offset` auxiliary array represents the offset from the main diagonal. It is well-suited for representing banded matrices.

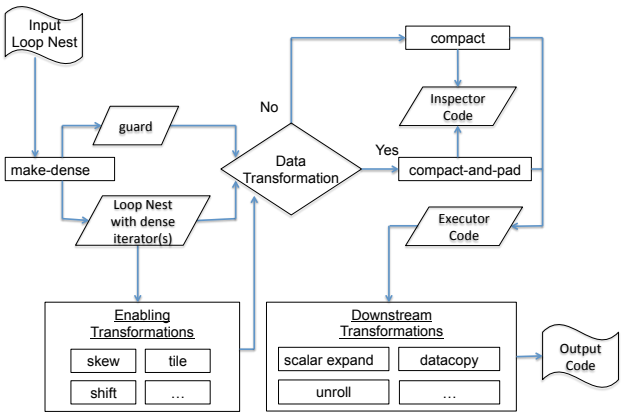


Figure 2: Overview of approach, showing how transformations are incorporated.

- *ELL:* The ELL format uses a 2-dimensional matrix with a fixed number of nonzeros per row, and rows with fewer nonzeros are padded with zero values. An auxiliary `col` matrix tracks the columns for the nonzeros as in CSR. When most rows have a similar number of nonzeros, ELL leads to more efficient code because of a fixed number of iterations and no indirection in the loop bounds.

We will use CSR as the default matrix representation, and BCSR, DIA and ELL as exemplars of the range of optimized sparse matrix representations. We focus our comparison in this paper with two manually-optimized sparse matrix libraries. OSKI tunes sparse matrix computation automatically for a particular architecture, focusing on identifying the best matrix representation given the architecture and input matrix [46]. The CUSP library roughly matches the manually-optimized parallel implementations of SpMV for GPUs described in [8].

3. Overview of Approach

This section describes the new transformations *make-dense*, *compact*, and *compact-and-pad*. Figure 2 illustrates how the new transformations interact with each other and other transformations.

- First, *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 *make-dense* transformation enables further loop transformations such as tiling.
- The *compact* and *compact-and-pad* transformations are *inspector-executor* transformations; an automatically-generated inspector gathers the iterations of a dense 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 an inspector combine polyhedral manipulation of loop iteration spaces and constraints on statement execution with modification of the underlying abstract syntax tree to introduce new statements.

We implemented the new transformations and inspector within the CHiLL and CUDA-CHiLL frameworks [38], which rely internally on CodeGen+ and Omega+ for polyhedral code generation. For the purposes of this paper, both CHiLL and CUDA-CHiLL take as input sequential C code and produce as output optimized C or CUDA code. A script called a *transformation recipe* describes the transformations to be applied to the code [19], produced either by the compiler or by domain experts. Example scripts are shown in the next two sections.

3.1 make-dense

Figure 3(a) presents before and after code templates for *make-dense*, illustrating its effects on the control flow, loop bounds, and array index expressions. The arguments of command *make-dense*($s0$,[$idx_1, \ldots, idx_m$]) identify the statement and the index expressions to which the transformation is to be applied. Each input index expression is replaced with a corresponding dense loop iterator, $I_{x_1}, \ldots, I_{x_m}$. These dense loops iterate over the range of their corresponding index expression and are placed immediately outside some loop I_k, where I_k is the innermost loop upon which any of the m index expressions depend. As the set of index expression values may not be a continuous sequence, and as loop iterators are continuous integer sequences, a guard surrounds the loop body to compare the new dense iterators to the associated index functions. Finally the non-affine index expressions are replaced by a reference to the new loop iterator.

Although the transformed code after the *make-dense* transformation is executable unlike the sublimation approach in [42], typically we would not want to execute the code because it is not efficient. Specifically, the entire dense range for each non-affine index expression passed to *make-dense* is now being visited although the guard ensures only the iterations from the original loop are executed. However, *make-dense* is still useful as an enabling transformation because it enables tiling over the range of index expressions, register tiling, and scalar replacement. In essence, *make-dense* results in a loop nest with more affine loop bounds and affine array index expressions.

Safety Test: A conservative safety test for *make-dense* requires that the only dependences carried by loops $I_1, \ldots, I_k$ are due to reduction computations [2]. This restriction is because the new dense loops will iterate over the range of possible index expression values in order, whereas the original loop potentially employs a different order (e.g., if the nonzeros in each row `i` in Listing 1 are not stored in order by column). If it is possible to prove that the non-affine array accesses are monotonically non-decreasing [24, 33], and therefore the iterations are not reordered, then this restriction is not needed.

There are other requirements on the array index expressions and the placement of the guard introduced by *make-dense*. Any index expression where it is possible to compute its range is allowed as input to *make-dense*. Additionally, the guard depends on the loops $I_1, \ldots, I_k$ and $I_{x_1}, \ldots, I_{x_m}$ and therefore must be nested within those loops and should surround the whole loop body.

3.2 compact

The *compact* transformation replaces a dense loop containing a conditional guarding execution of its body with a sparse loop that only visits the iterations where the condition is true. The *compact* command takes as arguments the statement corresponding to the loop body, and I_k, the loop level whose iterations are to be evaluated against a guard. The transformed original code is called the executor, as illustrated in the before-and-after code template in Figure 3(b). The transformation also generates an inspector to pack the iteration values that satisfy the condition into array `explicit_index`, shown in Figure 4(a). The *compact* transformation is similar to guard encapsulation [9].

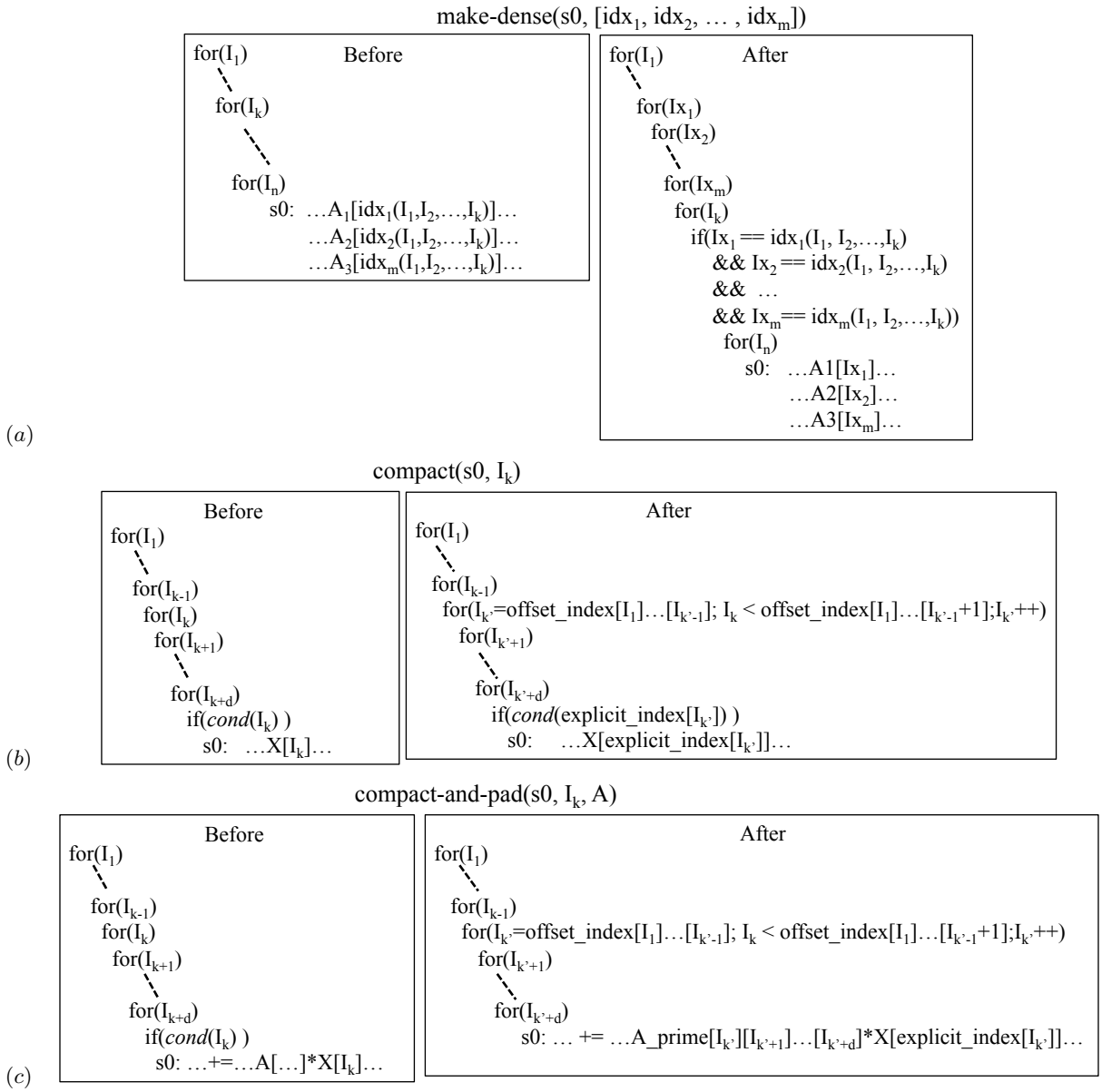


Figure 3: Templates for (a) make-dense, (b) compact and (c) compact-and-pad transformations.

Each of the outer loops of I_k: $I_1, \ldots, I_{k-1}$, are represented by dimensions in `offset_index`. On iterations that satisfy the guard, `explicit_index` records the original value of I_k, and is used in place of references to I_k in the executor. Since the loop being compacted, I_k may have inner loops (e.g., loops I_{k+1} through I_{k+d} in Figure 3(b)), the inspector needs to ensure that it only stores a specific value of I_k once. The *marked* variable flags the presence of a compacted loop iteration that satisfies the guard and ensures that each such iteration is counted only once. After *compact* has been applied the resulting code will have more non-affine loop bounds and array index expressions.

An example where the *compact* transformation could be used is in the construction of unaligned block compressed sparse row (UBCSR) [47], GCSR [49], and doubly compressed sparse columns (DCSC) [13], where only non-empty rows or columns are stored for blocks of nonzeros in a sparse matrix.

Safety Test: The *compact* transformation is always legal because it does not change the ordering of the iterations in the loop. It merely replaces a dense loop iterator with a sparse one that has non-affine loop bounds. While *compact* is safe, further transformations that could have been applied to the previously affine loop bounds may no longer be applicable to that loop level.

3.3 compact-and-pad

The *compact-and-pad* transformation generates a similar inspector and executor to *compact* but additionally performs a data transformation. It takes as arguments the statement corresponding to the loop body, and I_k, the loop level whose iterations are to be evaluated against a guard, and a single array `A` to which a data trans-

formation is to be applied. The before-and-after executor code and inspector code are shown in Figures 3(c) and 4(b), respectively. Inspector generation relies on the following definition:

Definition 1. *For a given set of constraints S on a set of variables $I_1, \ldots, I_n$, Project(S, I_j) is defined as the set of reduced constraints obtained from S by eliminating every occurrence of variable I_j from all (in)equalities in S using Fourier-Motzkin elimination, i.e. every pair of inequalities of the form $lb \leq c_1 I_j, c_2 I_j \leq ub$ is replaced by a new inequality $c_2 lb \leq c_1 ub$.*

Conceptually, the *compact-and-pad* inspector copies the footprint of the compacted iterations associated with a specified array `A` into a transformed array `A_prime` that will be referenced in the optimized executor code. The count of the compacted loop's iterations that satisfy the guard assumes the leading dimension of the newly reorganized array, `A_prime`. When the compacted loop level is not innermost, we use the inner loops' $(I_{k+1}, \ldots, I_{k+d})$ iteration space to derive the size of `A_prime`. For each loop level j nested inside I_k, the size of the dimension corresponding to that level is computed as $ub_j - lb_j + 1$, where $lb_j \leq I_j \leq ub_j$ and lb_j and ub_j are the respective lower and upper bounds on the loop level I_j in the set of constraints obtained from *Project*(S,I_k),. . .,*Project*(S,I_{j-1}). That is, the outer loop constraints are projected into each inner loop to derive the inner loops' maximum loop bounds. These bounds are then used to allocate the corresponding array dimension.

Additionally *compact-and-pad* might pad data elements with an identity value (0 in the template) in `A_prime` to effect redundant but harmless computation for associative operators. The inspector can insert an identity value, particular to the type of operation, into `A_prime` even on iterations that do not satisfy the guard. This allows *compact-and-pad* to eliminate the guard condition in the executor where *compact* would not, as illustrated in Figures 3(b) and (c). The `marked` variable in *compact-and-pad* serves an identical function as in *compact*.

Safety Test: Eliminating the guard condition in *compact-and-pad* is unsafe if two distinct input iteration space vectors evaluate to the same location in the array being transformed. Thus, *compact-and-pad* aborts in the inspector if two distinct data entries being inspected are mapped to the same array location. This is the injectivity criterion. Any affine transformations that manipulate the loop indices in the guard such that the injectivity criterion is violated will cause *compact-and-pad* to fail. Further, padding and eliminating the guard relies on updates involving `A_prime` to be associative and to have an identity value.

3.4 Example: CSR SpMV

```
1  for(i = 0; i < N; i++)
2   for(j = index[i];j < index[i+1]; j++)
3    y[i] += A[j]*x[col[j]];
```

Listing 1: CSR SpMV code.

```
1  for(i = 0; i < N; i++)
2   for(k = 0; k < N; k++)
3    for(j = index[i];j < index[i+1]; j++)
4     if(k == col[j])
5      y[i] += A[j]*x[k];
```

Listing 2: CSR SpMV after make-dense.

```
1  for(i = 0; i < N; i++)
2   for(k = offset_index[i]; k < offset_index[i+1]; k++)
3    y[i] += A_prime[k]*x[explicit_index[k]];
```

Listing 3: Executor for CSR SpMV.

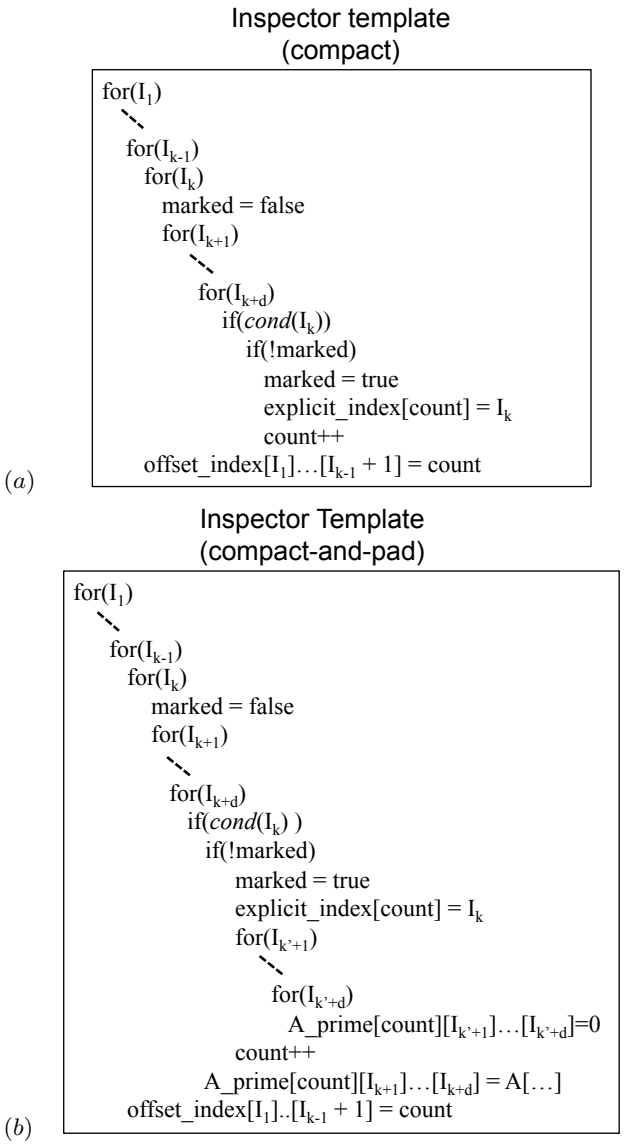


Figure 4: Templates for the run-time inspector for (a) compact and (b) compact-and-pad (before optimizations in Section 4).

Applying *make-dense* and a subsequent *compact-and-pad* on the SpMV code in Listing 1, based on the templates presented in Figure 3, results in the executor code shown in Listing 3. This code is roughly equivalent to the original CSR SpMV code.

3.5 Example: Traversing an Unstructured Mesh

The example input code in Listing 4 traverses over triangles in an unstructured mesh and performs various operations by accessing values associated with the three nodes of each triangular element. The code has three distinct non-affine index expressions `n1`, `n2` and `n3`, and illustrates how *make-dense* and *compact* could be called on multiple array index expressions and loop levels simultaneously.

Three outer loops are inserted by *make-dense*, as shown in Listing 5, with one corresponding to each indirect reference. The execution is guarded with a logical conjunction over the conditions corresponding to each loop level. Then, *compact* is subsequently called with all three outer loops simultaneously specified to pro-

```
for (e=0; e<numelem; e++){
 ... data[ n1[e] ] ...
 ... data[ n2[e] ] ...
 ... data[ n3[e] ] ...
}
```

Listing 4: Triangle code.

duce the inspector in Listing 6. Since all loop levels for *compact* are continuous, they are treated as a single logical entity. The inspector code records the value of each iterator that satisfies the guard, one per compacted loop level. The inspector code in Listing 6 is further optimized as outlined in Section 4.2. The optimized executor code appears in Listing 7.

```
for(n3i=0; n1i < N: n3i++)
 for(n2i=0; n2i < N: n2i++)
  for(n1i=0; n1i < N: n1i++)
   for (e=0; e<numelem; e++)
    if(n1i==n1[e] && n2i==n2[e] && n3i==n3[e]){
     ... data[ n1i ] ...
     ... data[ n2i ] ...
     ... data[ n3i ] ...
    }
```

Listing 5: Triangle after make-dense.

```
count=0
for(n3i=0; n1i < N: n3i++){
 marked_3 = false;
 for(n2i=0; n2i < N: n2i++){
  marked_2 = false;
  for(n1i=0; n1i < N: n1i++){
   marked_1 = false;
   for (e=0; e<numelem; e++)
    if(n1i==n1[e] && n2i==n2[e] && n3i==n3[e])
     if(!(marked_3 && marked_2 && marked_1)){
      marked_3  = true;
      marked_2  = true;
      marked_1  = true;
      explicit_index_1[count] = n1i;
      explicit_index_2[count] = n2i;
      explicit_index_3[count] = n3i;
      count++;
     }
  }
 }
}
```

Listing 6: Inspector resulting from compact for Triangle.

```
for(i=0; i < count;i++){
    ... data[ explicit_index_1[i] ] ...
    ... data[ explicit_index_2[i] ] ...
    ... data[ explicit_index_3[i] ] ...
}
```

Listing 7: Executor resulting from compact for Triangle.

3.6 Example: BCSR SpMV

Now let us examine how to use these transformations to modify the matrix representation. The inspector copies from `A` to `A_prime` to introduce additional zero-valued elements. We first apply *make-dense* to the CSR input of Listing 1 and produce the code in Listing 2. To derive dense blocks of constant size $R \times C$, we can tile the output of *make-dense*, both the `i` and `k` loops, corresponding to the rows and columns of the sparse matrix. The tiles are then the inner loops (with `j` temporarily remaining as innermost loop), and the tile controlling loops `ii` and `kk` permuted to the outermost positions. This tiling is proven safe as it does not modify the direction of the dependence on `y`. The associated transformation relation is:

$$T = \{[i,k,j] \rightarrow [ii,kk,i,k,j] | \\ C * kk + k < N \;\&\&\; R * ii + i < N\}$$

A subsequent *compact-and-pad* at loop level `kk` produces the BCSR executor shown in Listing 9. The corresponding inspector and CHiLL script for BCSR are shown in Listing 10 and Figure 5, respectively.

```
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];
```

Listing 8: CSR SpMV after make-dense and tiling.

```
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_prime[kk][i][k]*x[C*explicit_index
     [kk] + k];
```

Listing 9: Executor for BCSR SpMV.

```
make_dense(stmt,"j","k")

tile(stmt,"i", R, 1, counted)
tile(stmt,"k", C, 1, counted)

compact-and-pad(stmt,"kk","a", "a_prime")

--downstream transformations
--copy to temporary storage and
--fully unroll inner loops

datacopy(executor_stmt,"k", x)
datacopy(executor_stmt, "i", y)
unroll(executor_stmt,"k", C)
unroll(executor_stmt, "i", R)
```

Figure 5: CHiLL script for BCSR.

4. Optimization of Inspector and Executor

A key consideration in automating an inspector/executor approach for sparse matrices, and a distinguishing feature of our work, is to derive high-performance inspectors that perform comparably to those used in manually-tuned libraries. In particular, we want to avoid the inspector having to make several passes over the sparse matrix or introduce significant additional work not present in the original code. This section describes code generation and optimization details for both the inspector and executor.

```
1  struct list_item{
2    float data[R][C];
3    int col;
4    struct list_item *next;
5  };
6  struct mk{
7    struct list_item *list_ptr;
8  };
9  offset_index[0] = 0;
10 count = 0;
11 struct mk marked[];
12 struct list_item *list=NULL;
13 for(ii = 0; ii < N/R; ii++){
14  for(i = 0; i < R; i++)
15   for(j = index[ii*R + i]; j < index[ii*R+i+1] ; j++){
16    kk = col[j]/C;
17    marked[kk].list_ptr = NULL;
18   }
19  for(i = 0; i < R; i++)
20   for(j = index[ii*R + i]; j < index[ii*R+i+1] ; j++){
21    kk = col[j]/C;
22    k = col[j] - kk*C;
23    if(marked[kk].list_ptr == NULL){
24     struct list_item *new_entry =
25     malloc(sizeof(struct list_item));
26     for(i_ = 0; i_ < R; i_++)
27      for(k_ = 0; k_ < C; k_++)
28       new_entry->data[i_][k_] = 0;
29     new_entry->col = kk;
30     new_entry->next = list;
31     list = new_entry;
32     marked[kk].list_ptr = new_entry;
33     count++;
34    }
35    marked[kk].list_ptr->data[i][k] = A[j];
36   }
37   offset_index[ii+1] = count;
38 }
```

Listing 10: Optimized inspector for BCSR SpMV.

4.1 Dynamic Memory Allocation and Reduced Traversals (Inspector)

The size of the new matrix representation cannot be determined statically. However, traversing the input to compute the size, as is done in OSKI, is expensive as it requires two passes over the input: one for initialization of an auxiliary data structure, and another to traverse and count the number of nonzeros.

To minimize the number of traversals over the input, we utilize dynamic memory allocation to create a linked list of nonzero elements or blocks when they are discovered by the inspector or use static memory allocation when the size is known a priori, such as in ELL. This allows us to copy the data associated with the nonzeros while counting them in a single effective pass over the input matrix. Despite the overhead of subsequently copying the linked list into an array, this approach is faster than using two separate passes, as will be shown in Section 6.

4.2 Derivation of Closed-Form Iterators (Inspector)

The unoptimized inspector code resulting from the *compact* and *compact-and-pad* transformations retains the loops and the guard from the input code. Since the *make-dense* command introduces additional loops to the executor and *compact* or *compact-and-pad* are typically applied after *make-dense* and other enabling transformations, the inspector's traversal over the resulting iteration space can be expensive. To reduce the overhead associated with the inspector, we replace these loop iterators with closed-form expressions wherever possible. The main idea behind this optimization is to compute the maximal set of loop iterators that can be derived from other iterators based on the guard condition for *compact*.

We utilize the rich set of polyhedral functions, such as variable projection and elimination, provided by the Omega+ library to accomplish this. The guard conditions (typically those introduced by *make-dense*) are encoded as equality constraints involving loop index variables, and all possible orderings of the loop index variables involved in the guard condition are considered to determine how to eliminate the maximum number of inner loops in the order. The guard condition is progressively updated with the eliminated variables being replaced with the corresponding expressions. A variable can be eliminated if present in the set of equalities of the current guard.

One of the patterns our algorithm detects is variables that are multiplied by some constant and then summed with a variable whose constant range is defined by the same constant (e.g., the `kk*C + k == col[j]` condition in Listing 8). Generally, if we have the constraints $v * c + m = e$ and $0 \leq m < c$, where c is a constant, v and m are iterators, and e is some expression, then a closed-form expression for v is $v = \lfloor e/c \rfloor$ used in conjunction with terms being multiplied by that constant range.

For example, consider the sample guard condition `kk*C + k == col[j]`. The bounds on the `k` loop are $0 \leq k < C$ and if the `kk` loop is the candidate for elimination, the `k` loop is replaced as an existential leading to the constraints below:

$$I = \{[j, kk] \mid (\exists k : C * kk + k = col(j) \wedge 0 \leq k < C)\}$$

We observe the existence of a closed form solution for `kk` is $\lfloor col[j]/C \rfloor$ and eliminate the `kk` loop in the inspector (see lines 28 and 33 in Listing 10). Once `kk` is derived, its solution can be substituted into the set of constraints to yield further loop iterators such as `k`, where `k` is `col[j]-C*kk`, which would be uncovered by checking for equalities in the modified guard constraint. Hence both these loops may be eliminated from the inspector code. The optimized code with the loops and guard condition eliminated and replaced with assignments as functions of the sparse iterator is shown in Listing 10.

4.3 Elimination of Loops (Executor)

The additional loops introduced by the *make-dense* transformation make it imperative for the *compact* and *compact-and-pad* transformations to eliminate the additional loop(s) and/or guard introduced to minimize the run time of the executor code.

Redundant loops in the executor can be identified from considering the guard condition and the loop(s) being compacted. Let us assume that the input code for *compact* (or equivalently, *compact-and-pad*) is an `n`-deep loop nest of the form $I_1, \ldots, I_n$ and a particular loop level I_j is a candidate for compaction and elimination from the executor. The I_j loop may be eliminated if: (1) the guard is satisfied on all iterations of I_j; and, (2) an injective function F exists such that on each iteration of I_j that satisfies the guard, F maps that iteration to a unique tuple formed from the other iterators. This function is explicitly constructed in the inspector code for *compact*.

```
1  for(i = 0; i < N; i++)
2   for(k = offset_index[i]; k < offset_index[i+1]; k++)
3    y[i] += A[col_inv[k]]*x[explicit_index[k]];
```

Listing 11: Executor code for CSR SpMV from compact

In Listing 11, the optimized executor code is shown with the `j` loop removed. The reference to the `j` loop in the array `A`, is now derived from `k`, using `col_inv`, which represents the injective function F. The redundant loops being eliminated from the executor

are distinct from the required loops whose closed form expressions are being derived in the inspector.

5. Parallel GPU Implementations

We now describe two parallel implementations targeting Nvidia GPUs, which extend the work of Venkat et al. to examine implementations that require new matrix representations and will be used to compare with manually-tuned CUSP [44].

5.1 DIA

To uncover the diagonals that are implicit in the SpMV CSR format, the *make-dense* command is used to convert the iteration space of the sparse computation to its corresponding dense one, as we did for BCSR. After this transformation we skew the resulting dense loop `k` by `k=k-i` and permute the `i` and `k` loops to obtain the code in Listing 12. The transformation relations are as follows:

$$T = \{[i,k,j] \rightarrow [i,k-i,j]\}$$
$$T = \{[i,k,j] \rightarrow [k,i,j]\}$$

The outer `k` loop iterates over the diagonals that are numbered from 0 to 2*N-1, while the inner `i` loop gives the maximum possible count of the number of elements in each diagonal. However, since the matrix is sparse, the additional guard `(k+i-(N-1) == col[j])` checks for the presence of the diagonal entry. Now the *compact* command is called on the `k` loop, with the `A` matrix as argument to eliminate diagonals with nonzeros. The final executor code is shown in Listing 13.

```
1  for(k = 0; k<= 2*N-2; k++)
2   for(i = max(0,N-1-k); i <= min(N-1,2*N-2-k); i++)
3    for(j = index[i];j < index[i+1]; j++)
4     if(k+i-(N-1) == col[j])
5      y[i] += A[j]*x[k+i-(N-1)];
```

Listing 12: CSR SpMV after make-dense, skew and permute.

```
1  for(k=0; k < ub; k++)
2   for(i = 0; i <= N-1; i++)
3     y[i] += A_prime[k*N + i]*x[explicit_index[k]+i -(N
        -1)];
```

Listing 13: Executor for DIA SpMV.

The CUDA-CHiLL script for DIA is shown in Figure 6. Following the *make-dense* and *compact-and-pad* commands, the inner i loop is parallelized for threaded execution on a GPU. The *copy_to_shared* command copies the diagonal offset matrix or the explicit_index matrix into shared memory to exploit its reuse across multiple threads, while the *copy_to_registers* command copies the output y vector to an intermediate register to accumulate the rows' dot products.

5.2 ELL

The ELL data format relies on determining the maximum number of nonzero entries in a row for a sparse 2-D matrix and then extending the other rows to that length. The number of nonzeros per row in the initial SpMV CSR code is implicit in the loop bounds. To make it explicit, the inner `j` loop is normalized to give an exact count of the nonzeros in each row, using a non-affine shifting transformation. The maximum row length `M` must be supplied, or can be derived by additional inspection, and is used as a tile size for the `j` loop. After *compact-and-pad*, the ELL executor is

```
make_dense(0,2,"k")
--enabling transformations
skew(stmt,"k",{-1,1})
permute(stmt,{"k","i","j"})

compact-and-pad(stmt,"k","a","a_prime")

--downstream transformations
permute(executor_stmt,{"i","k"})
tile_by_index(executor_stmt,{"i"},{Ti},{l1_control="ii"},
{"ii","i","k"})
tile_by_index(executor_stmt,{"k"},{Ti},{l1_control="kk"},
{"ii","i","kk","k"})

cudaize(executor_stmt,"spmv_diag_GPU",{x=N,y=N},
{block={"ii"}, thread={"i"}}, {"a_prime", "_P_DATA2"})

--shared memory and register copy --optimizations for
GPU
copy_to_shared(executor_stmt,"k","_P_DATA2",-16)
copy_to_registers(executor_stmt, "kk", "y")
```

Figure 6: CUDA-CHiLL script for DIA.

parallelized similarly to DIA, using a transposed matrix to achieve global memory coalescing and a register copy to accumulate the output results. As the non-affine shift is somewhat unique, we include the transformation relations as follows.

$$T = \{[i,j] \rightarrow [i, j - index(i)]\}$$
$$\{[i,j] \rightarrow [i,jj,j] \mid M * jj + j < index(i+1) - index(i)\}$$

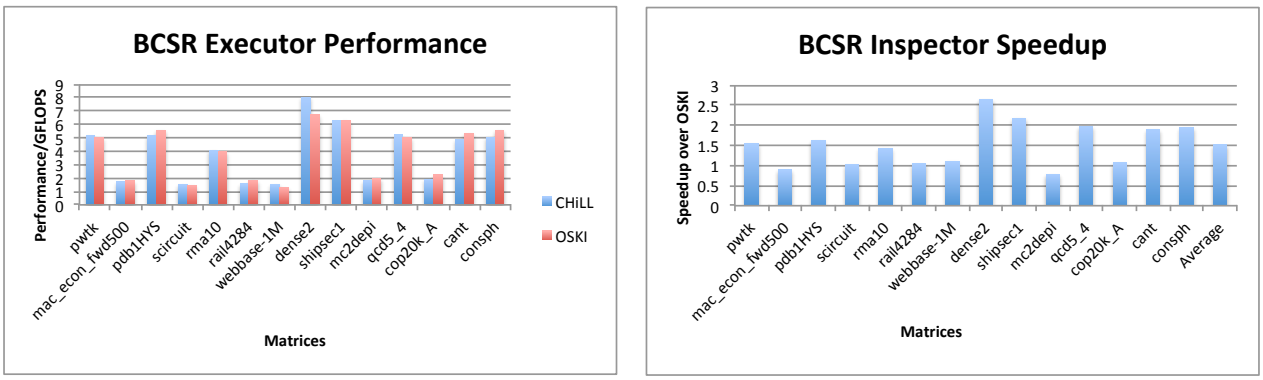
6. Performance Results

The transformations described in Section 3 are used in the transformation recipes such as in Figures 5 and 6 to derive optimized BCSR, DIA and ELL matrix representations. These recipes include further downstream transformations as were described in Sections 3 and 5 to derive the final corresponding optimized code variants.

We use two target platforms for this experiment. The BCSR comparison focuses on optimizing for the memory hierarchy of a node in a conventional multi-core architecture. For this purpose, we use an Intel i7-4770 (Haswell) CPU with 256KB L1 cache, 1 MB L2 cache, and 8MB L3 cache size. The system has 32GB of DRAM memory. The DIA and ELL comparisons look at performance on an Nvidia Tesla K20c Kepler GPU. The K20c has 13 Streaming Multiprocessors with 192 cores per SM. It has 4800 MB of global memory and a 64KB register file per streaming multiprocessor. All CPU code is compiled with the icc compiler, version 15.0.0, and all GPU code uses the nvcc compiler, version 6.5.

We compare BCSR performance to OSKI version 1.0.1h for the same set of 14 matrices used by Williams et al.[48] and obtained from the University of Florida Sparse Matrix Collection[16]. We compare DIA and ELL performance to CUSP version 0.4.0 for the same sparse matrices as in [8], structured matrices resulting from standard discretizations of the Laplacian operator. The sizes of these matrices range from 958,936 to 11.6 million nonzeros, therefore sufficiently large to exceed the size of the last level cache of the underlying machine.

We report the absolute performance for all implementations in terms of GFlops. As sparse matrix libraries that convert to optimized matrix representations also incorporate an inspector to derive the new representation, we also compare against inspector times for OSKI and CUSP. As is done in CUSP, the compiler-generated DIA and ELL inspector code is executed sequentially on the CPU.



(a) (b)

Figure 7: Performance comparison of BCSR executor and inspector code with respect to OSKI.

6.1 BCSR

A performance comparison of the compiler-generated BCSR code with OSKI is shown in Figure 7(a). We ran all configurations for block size $R \times C$ (see Listing 9) in the range of 1 to 8 and report the best-performing configuration for both OSKI and our compiler-generated code. The compiler-generated codes are within 1% of the performance of OSKI for the BCSR executor.

The inspector speedup, shown in Figure 7(b), compares the overhead of converting from CSR to the BCSR representation. On average the compiler-generated inspector is 1.5x faster than OSKI's. This is because OSKI does effectively four sweeps over the input matrix: two to compute the number of nonzero $R \times C$ blocks to allocate the memory accordingly and two more to actually fill the nonzeros in the appropriate locations of the reorganized matrix. We eliminate the first two sweeps over the input matrix by simultaneously allocating memory dynamically as a linked list of nonzero blocks and filling in the nonzeros. An additional traversal is done to reset the data structure. Finally a sweep over the nonzero blocks stored as a linked list is done to copy the data into an array layout. Hence we accomplish the data reorganization in three sweeps as opposed to four by OSKI.

For input matrices mc2depi and mac_econ_fwd500, BCSR does not achieve any advantage over the CSR format for both OSKI and our compiler. The extra costs associated with dynamic memory allocation for every nonzero in the matrix is not amortized even by the lesser number of traversals in our inspector code compared to OSKI.

Comparing the inspector time for a particular matrix is fair only when the same $R \times C$ configuration is picked across both as larger $R \times C$ sizes lead to faster inspector times for a given matrix. In cases where the best performing configurations of the executor for both OSKI and ours were identical, such as for cant, consph, qcd5_4, and shipsec1, we observe that our inspector is uniformly faster than OSKI's, due to fewer traversals over the input matrix.

6.2 DIA and ELL

Figures 8(a) and (b) compare performance of the DIA and ELL executor code against that of the CUSP library. On average the compiler-generated code is within 5% of the performance of CUSP for both representations. The best-performing compiler-generated versions are up to 6% faster than CUSP.

We observed that as the size of the stencil increases from a 3- to 27-point stencil, the performance of the CUDA-CHiLL code variant relative to its CUSP counterpart improves. CUSP outperforms CUDA-CHiLL marginally for the 3- and 5-point stencils. This is due to a subtle difference in the two implementations. In CUSP the output vector storing the result of the matrix-vector multiply is assumed to be zero; the inner product of each vector is accumulated in a register, pre-initialized to zero, and then copied to the output vector. The code generated by CUDA-CHiLL has an additional global memory read to initialize the result vector. This read overhead is noticeable when the work per thread is relatively small, such as for the low-order 3- and 5-point stencils. The performance advantage of the code generated by CUDA-CHiLL on the other stencils is a result of an IF-condition in the CUSP code checking if the column entries are valid in the innermost loop. CUDA-CHiLL avoids this inner conditional check by virtue of the inspector adding zeros into the data representation. DIA outperforms ELL significantly, up to a factor of 2×, because the DIA implementation reuses the elements of the `x` vector and the `offset` vector, whereas ELL cannot.

Figures 8(c) presents the speedup for the automatically-generated DIA inspector over CUSP, which is on average 1.27× faster. The CUSP inspector initializes the entire DIA matrix in a single pass prior to copying the nonzero entries in a separate pass, whereas the CUDA-CHiLL inspector initializes and copies the data in a single pass. Initializing and copying the data in a single pass is beneficial to exploit temporal reuse of the initialized diagonal, if it is copied into subsequently. This is the case for all matrices except the 27-point stencil where the size of the last level cache is insufficient to exploit this temporal reuse. In the case of the 27-point stencil, some of the initialized diagonals are flushed without being reused for the data copy of the nonzeros. This result suggests that we might improve inspector time for large matrices with numerous diagonals if we could generate both versions of the inspector, and use techniques such as autotuning, learning or additional inspectors to decide which inspector is best for the input matrix.

Figure 8(d) examines the performance of the ELL inspector. We show two bars, labeled Inspector and Inspector+Transpose. To achieve global memory coalescing in the executor (i.e., adjacent threads accessing adjacent elements in memory), the CUSP library performs the computation on a transposed ELL matrix by declaring the ELL matrix to be in column major order. Our compiler-generated implementation must perform an additional transpose to achieve this same representation; alternatively, the generated code would be equivalent if the programmer could provide the designation of column major order to the compiler. Without the cost of the transpose, the red columns in Figure 8(d), the compiler-generated inspector achieves a speedup ranging from 0.52× to 1.26×. With the additional overhead of the transpose, the automatically-generated inspector code is always slower than CUSP, as the blue columns show speedups between 0.26× and

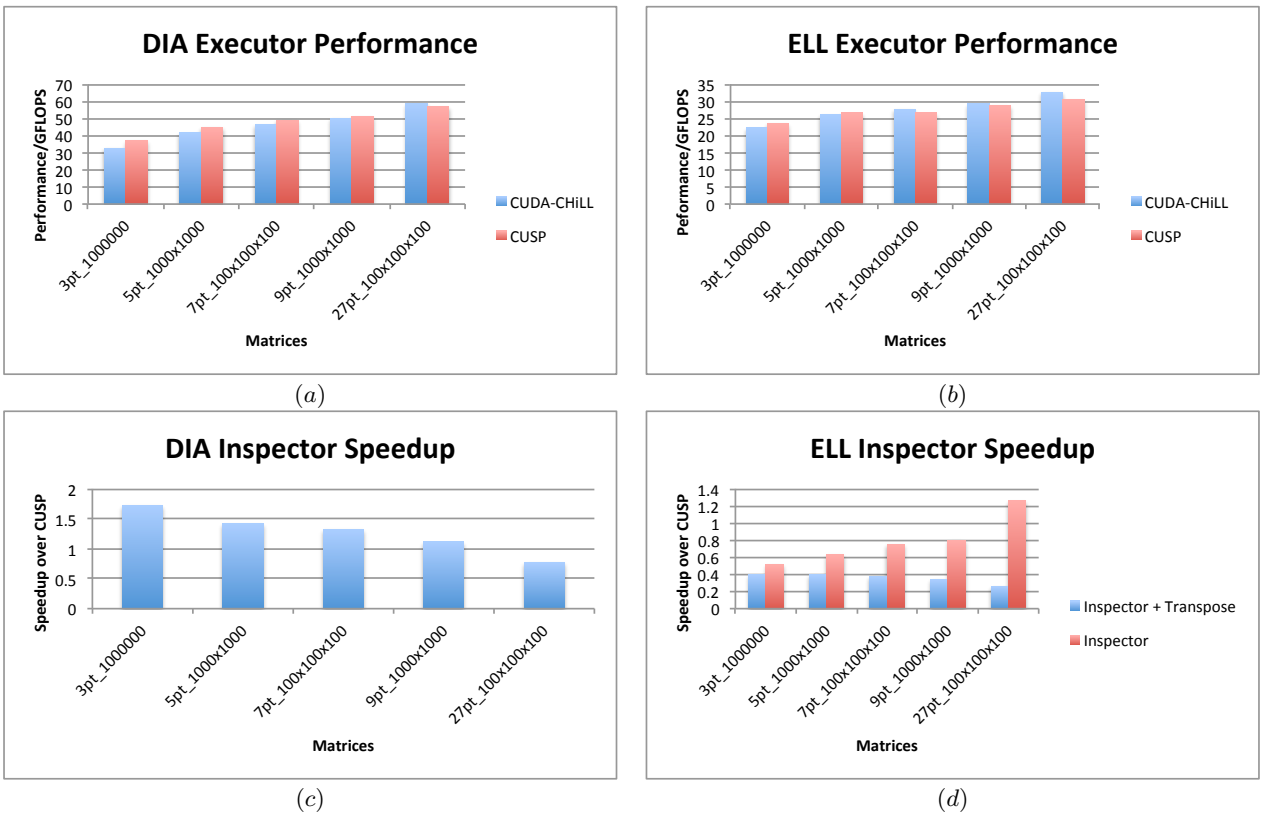


Figure 8: Performance comparison of DIA and ELL inspector and executor code with respect to CUSP.

0.40×. As the size of the matrix increases, the inspector performance relative to CUSP improves because the difference in layout is less significant for large matrices. We have identified additional optimizations to improve ELL inspector time that could be the target of future work. First, we could combine the transpose with the data layout transformation in a single sweep by permuting the loops in the CSR code prior to *compact-and-pad*, incorporating the knowledge of the fixed upper bound for the inner loop into account to make this safe. Second, we can reduce loop overhead arising from tiling by the fixed row width.

7. Related Work

This section expands on the discussions of prior work in Sections 1 and 2 to focus on the most closely-related compilers targeting sparse matrix codes.

7.1 Sparse Matrix Compilers

Previous work has developed compiler optimizations for sparse matrices beginning with a dense abstraction of a sparse matrix computation, as optimizations for dense matrix computations are well understood; these compilers generate sparse data representations during code generation [4, 10, 28, 34]. These compilers either incorporate a small, fixed set of matrix representations for which code generation is straightforward or rely on the user to provide implementations for accessing data in sparse formats for operations such as searching, enumeration and de-referencing. Shpeisman and Pugh [34] specify an intermediate program representation for transforming sparse matrix codes. The specification directs an underlying C++ library for efficient enumeration, permutation and scatter-gather access of nonzeros stored according to some compressed stripe storage. The Bernoulli compiler permits extension to new formats by abstracting sparse matrix computations into relational expressions that describe constraints on the iteration space and predicates to identify nonzero elements [22, 23, 28]. Using an approach similar to optimizing relational database queries, the compiler derives a plan for efficiently evaluating the relational expressions, and generates corresponding code. Gilad et al. [4] use the LL functional language for expressing and verifying sparse matrix codes with their dense analogs, under the assumption that the matrix is dense initially.

In contrast to these compilers specialized for sparse matrix computations, we have developed code and data transformations applicable to non-affine loop bounds and subscript expressions within a polyhedral framework, in conjunction with an automatically-generated inspector. Existing non-affine code can be optimized in this way with our compiler, and new matrix formats can be supported by applying the appropriate sequence of transformations. Our work is also distinguished in demonstrating that the inspection time is comparable to manually-tuned libraries.

7.2 Sublimation and Guard Encapsulation

The make-dense transformation is similar to sublimation presented by van der Spek et al. [42]. Sublimation requires analysis or pragmas to determine injectivity properties of the access functions so that the sublimation transformation can replace an existing loop with irregular bounds (like the inner loop in SpMV) with the dense loop. Additionally, a related idea of expanding a sparse vector into a dense vector is called *access pattern expansion* [11].

The *compact* transformation we present is related to guard encapsulation [9], which moves tests for elements into the loop

Table 1: A list of transformations performed for each variant.

	Enabling Transformations							Downstream Transformations			
	make-dense	permute	skew	shift	tile	compact	compact-and-pad	datacopy	scalar-expand	unroll	coalesce
ELL				✓	✓		✓		✓		
DIA	✓	✓	✓				✓				
BCSR	✓	✓			✓		✓	✓		✓	
GCSR	✓					✓					
TRI	✓					✓					
CSB	✓	✓			✓		✓				✓
S-DIA	✓	✓	✓		✓		✓	✓		✓	

bounds; in addition, *compact-and-pad* rewrites the matrix into a new representation and performs optimizations on the inspector.

Further, we have incorporated our transformations into CHiLL, which enables compositions with other polyhedral transformations and compiler-based auto-tuning within a broader context. We have designed these transformations to also generate optimized inspectors that match or beat existing hand-written inspectors in libraries.

8. Conclusions and Future Work

This paper has presented three new transformations and an automatically-generated inspector that can be used to transform sparse matrix computations and their data representations. The compiler-generated inspector and executor code achieves performance that is comparable and sometimes exceeds the performance of popular manually-tuned sparse matrix libraries OSKI and CUSP. We see this work as an important step towards a general framework for automating transformation and data representation selection for the domain of sparse matrix computations.

To clarify the scope of the effort, and the power of incorporating these transformations into an existing compiler framework, Table 1 shows sparse matrix formats that can be derived using our framework, with BCSR, DIA and ELL the subject of this paper. We see that a rich set of transformations are needed, both to enable the restructuring of the code and matrix representation following *make-dense*, and to generate optimized code for our two target architectures.

The formats highlighted in grey, GCSR and TRI from Section 3, Compressed Sparse Block (CSB) [1] and S-DIA [27] can also be derived using the transformations we introduced. CSB [1] relies on blocking the dense matrix into square tiles and determining the nonzeros that fall within each tile. The nonzeros are then stored in a Coordinate (COO) format as in [44], where the row and column offsets within the tile are stored explicitly. S-DIA [27] relies on storing blocked diagonals and may be conceptualized as a hybrid between BCSR and DIA, and is derived similarly to DIA, with the addition of tiling.

Many different sparse matrix representations have been developed recently to exploit structural properties of the matrices whenever possible to improve code performance. In the future, we see the need to extend the inspector in our framework to support two new capabilities to be able to implement new sparse matrix representations. First, a number of representations require reorganizing the sparse matrix by sorting of the rows and/or columns of the input matrix, usually to expose locality [25, 40]. Other representations split the matrix into multiple parts with different characteristics, and use a different implementation for each part [7, 29, 45]. Incorporating sorting and splitting to implement hybrid schemes is the subject of future work.

Acknowledgments

The work at University of Utah was partially supported by the National Science Foundation award CCF-1018881 and by the Scientific Discovery through Advanced Computing (SciDAC) program funded by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research under award number DE-FG02-11ER26053. The work at Colorado State University was supported by a Department of Energy Early Career Grant DE-SC0003956.

References

[1] H. M. Aktulga, A. Buluç, S. Williams, and C. Yang. Optimizing sparse matrix-multiple vectors multiplication for nuclear configuration interaction calculations. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, 2014.

[2] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers, 2002.

[3] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Apr. 1991.

[4] G. Arnold, J. Hölzl, A. S. Köksal, R. Bodík, and M. Sagiv. Specifying and verifying sparse matrix codes. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, ICFP '10, pages 249–260, New York, NY, USA, 2010. ACM.

[5] S. Balay, J. Brown, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.1, Argonne National Laboratory, 2010.

[6] A. Basumallik and R. Eigenmann. Optimizing irregular shared-memory applications for distributed-memory systems. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, 2006.

[7] M. Belgin, G. Back, and C. J. Ribbens. Pattern-based Sparse Matrix Representation for Memory-Efficient SMVM Kernels. In *ICS '09*, pages 100–109, 2009.

[8] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of SC '09*, Nov. 2009.

[9] A. Bik and H. Wijshoff. On automatic data structure selection and code generation for sparse computations. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, volume 768 of *Lecture Notes in Computer Science*, pages 57–75. Springer Berlin Heidelberg, 1994.

[10] A. Bik and H. A. Wijshoff. Advanced compiler optimizations for sparse computations. In *Supercomputing '93 Proceedings*, pages 430–439, Nov 1993.

[11] A. J. C. Bik. *Compiler Support for Sparse Matrix Computations*. PhD thesis, Leiden University, May 1996.

[12] A. J. C. Bik and H. A. G. Wijshoff. Automatic data structure selection and transformation for sparse matrix computations. *IEEE Trans. Parallel Distrib. Syst.*, 7(2):109–126, 1996.

[13] A. Buluç and J. R. Gilbert. Highly parallel sparse matrix-matrix multiplication. *CoRR*, abs/1006.2183, 2010.

[14] A. Buluç and J. R. Gilbert. The combinatorial blas: Design, implementation, and applications, 2010.

[15] C. Chen. Polyhedra scanning revisited. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 499–508, June 2012.

[16] T. Davis. The University of Florida Sparse Matrix Collection. *NA Digest*, 97, 1997.

[17] C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 229–241, New York, NY, USA, May 1999. ACM.

[18] P. Feautrier. Automatic parallelization in the polytope model. In *The Data Parallel Programming Model*, pages 79–103, 1996.

[19] M. Hall, J. Chame, J. Shin, C. Chen, G. be Rudy, and M. M. Khan. Loop transformation recipes for code generation and auto-tuning. In *LCPC*, October, 2009.

[20] H. Han and C.-W. Tseng. Exploiting locality for irregular scientific codes. *IEEE Transactions on Parallel and Distributed Systems*, 17(7): 606–618, 2006.

[21] W. A. Kelly. *Optimization within a Unified Transformation Framework*. PhD thesis, University of Maryland, Dec. 1996.

[22] V. Kotlyar and K. Pingali. Sparse code generation for imperfectly nested loops with dependences. In *Proceedings of the 11th International Conference on Supercomputing*, ICS '97, 1997.

[23] V. Kotlyar, K. Pingali, and P. Stodghill. A relational approach to the compilation of sparse matrix programs. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *Euro-Par'97 Parallel Processing*, volume 1300 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1997.

[24] Y. Lin and D. Padua. Compiler analysis of irregular memory accesses. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, May 2000.

[25] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey. Efficient sparse matrix-vector multiplication on x86-based many-core processors. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 273–282, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2130-3.

[26] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, Apr. 2012. ISSN 2150-8097.

[27] D. Lowell, J. Godwin, J. Holewinski, D. Karthik, C. Choudary, A. Mametjanov, B. Norris, G. Sabin, P. Sadayappan, and J. Sarich. Stencil-aware gpu optimization of iterative solvers. *SIAM J. Scientific Computing*, pages –1–1, 2013.

[28] N. Mateev, K. Pingali, P. Stodghill, and V. Kotlyar. Next-generation generic programming and its application to sparse matrix computations. In *Proceedings of the 14th International Conference on Supercomputing*, pages 88–99, Santa Fe, New Mexico, USA, May 2000.

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

[30] J. Mellor-Crummey, D. Whalley, and K. Kennedy. Improving memory hierarchy performance for irregular applications using data and computation reorderings. *International Journal of Parallel Programming*, 29(3):217–247, 2001.

[31] 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.

[32] N. Mitchell, L. Carter, and J. Ferrante. Localizing non-affine array references. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 192–202, October 1999.

[33] C. Oancea and L. Rauchwerger. A hybrid approach to proving memory reference monotonicity. In S. Rajopadhye and M. Mills Strout, editors, *Languages and Compilers for Parallel Computing*, volume 7146 of *Lecture Notes in Computer Science*, pages 61–75. Springer Berlin Heidelberg, 2013.

[34] W. Pugh and T. Shpeisman. Sipr: A new framework for generating efficient code for sparse matrix computations. In *Proceedings of the Eleventh International Workshop on Languages and Compilers for Parallel Computing*, Chapel Hill, North Carolina, August 1998.

[35] F. Quilleré and S. Rajopadhye. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming*, 28 (5):469–498, Oct. 2000.

[36] L. Rauchwerger and D. Padua. The lrpd test: speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, PLDI '95, 1995.

[37] M. Ravishankar, J. Eisenlohr, L.-N. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan. Code generation for parallel execution of a class of irregular loops on distributed memory systems. In *Proceedings of SC'12*, November 2012.

[38] G. Rudy, M. M. Khan, M. Hall, C. Chen, and C. Jacqueline. A programming language interface to describe transformations and code generation. In *Proceedings of the 23rd international conference on Languages and com pilers for parallel computing*, LCPC'10, pages 136–150, Berlin, Heidelberg, 2011. Springer-Verlag.

[39] J. Saltz, C. Chang, G. Edjlali, Y.-S. Hwang, B. Moon, R. Ponnusamy, S. Sharma, A. Sussman, M. Uysal, G. Agrawal, R. Das, and P. Havlak. Programming irregular applications: Runtime support, compilation and tools. *Advances in Computers*, 45:105–153, 1997.

[40] M. Shantharam, A. Chatterjee, and P. Raghavan. Exploiting dense substructures for fast sparse matrix vector multiplication. *Int. J. High Perform. Comput. Appl.*, 25(3):328–341, Aug. 2011.

[41] M. M. Strout, A. LaMielle, L. Carter, J. Ferrante, B. Kreaseck, and C. Olschanowsky. An approach for code generation in the sparse polyhedral framework. Technical Report CS-13-109, Colorado State University, December 2013.

[42] H. van der Spek and H. Wijshoff. Sublimation: Expanding data structures to enable data instance specific optimizations. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Lecture Notes in Computer Science, pages 106–120. Springer Berlin / Heidelberg, 2010.

[43] N. Vasilache, C. Bastoul, and A. Cohen. Polyhedral code generation in the real world. In *Proceedings of the 15th International Conference on Compiler Construction*, Mar. 2006.

[44] 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, 2014.

[45] R. Vuduc and H. Moon. Fast Sparse Matrix-Vector Multiplication by Exploiting Variable Block Structure. In *Proceedings of the High Performance Computing and Communications, volume 3726 of LNCS*, pages 807–816. Springer, 2005. ISBN 978-3-540-29031-5.

[46] R. Vuduc, J. W. Demmel, and K. A. Yelick. Oski: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, 16(1):521–530, 2005.

[47] R. W. Vuduc. *Automatic Performance Tuning of Sparse Matrix Kernels*. PhD thesis, University of California, Berkeley, 2003.

[48] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Computing*, 35(3):178 – 194, 2009.

[49] S. Williams, N. Bell, J. Choi, M. Garland, L. Oliker, and R. Vuduc. Sparse matrix vector multiplication on multicore and accelerator systems. In J. Kurzak, D. A. Bader, and J. Dongarra, editors, *Scientific Computing with Multicore Processors and Accelerators*. CRC Press, 2010.

[50] B. Wu, Z. Zhao, E. Z. Zhang, Y. Jiang, and X. Shen. Complexity analysis and algorithm design for reorganizing data to minimize noncoalesced memory accesses on gpu. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '13, 2013.