Automating Wavefront Parallelization for Sparse Matrix Computations

Anand Venkat*, Mahdi Soltan Mohammadi†, Jongsoo Park*, Hongbo Rong*,
Rajkishore Barik*, Michelle Mills Strout† and Mary Hall‡
*Parallel Computing Laboratory, Intel Corporation; Email: {anand.venkat, jongsoo.park, hongbo.rong, rajkishore.barik}@intel.com
†Department of Computer Science, University of Arizona; Email: {kingmahdi, mstrout}@email.arizona.edu
‡School of Computing, University of Utah; Email: mhall@cs.utah.edu

***Abstract*—This paper presents a compiler and runtime framework for parallelizing sparse matrix computations that have loop-carried dependences. Our approach automatically generates a runtime inspector to collect data dependence information and achieves wavefront parallelization of the computation, where iterations within a wavefront execute in parallel, and synchronization is required across wavefronts. A key contribution of this paper involves dependence simplification, which reduces the time and space overhead of the inspector. This is implemented within a polyhedral compiler framework, extended for sparse matrix codes. Results demonstrate the feasibility of using automatically-generated inspectors and executors to optimize ILU factorization and symmetric Gauss-Seidel relaxations, which are part of the Preconditioned Conjugate Gradient (PCG) computation. Our implementation achieves a median speedup of 2.97× on 12 cores over the reference sequential PCG implementation, significantly outperforms PCG parallelized using Intel's Math Kernel Library (MKL), and is within 6% of the median performance of manually-parallelized PCG.**

I. Introduction

Sparse matrix computations represent an important class of algorithms that arise frequently in numerical simulation and graph analytics. Sparse matrix computations have been considered mostly beyond the reach of parallelizing compilers due to information required for optimization only becoming available at execution time. For example, static analysis lacks the necessary information to analyze access patterns within the context of indirect array accesses, e.g., A[B[i]], where the contents of B are determined dynamically. Such index expressions are termed *non-affine*, since they are not linear functions of loop index variables. Earlier work has addressed this challenge by incorporating the *inspector/executor* approach, where an inspector determines memory access patterns at runtime and transforms the performance bottleneck loops into executors that use optimizations that rely on the groundwork and information gathering of the inspectors [1]–[9].

There has been much prior work on using inspector/executor for sparse matrix computation that focuses on primitives like sparse matrix-vector multiply (SpMV) [10]–[15]. For SpMV, parallelization is straightforward, so the optimization focus is on managing locality and load balance, often using specialized sparse matrix representations. However, there are more complex sparse matrix algorithms where loop-carried data dependences make efficient parallelization challenging; examples include sparse triangular solver, Gauss-Seidel relaxation, and Incomplete LU factorization with zero level of fill-in (ILU0) [16]. Many of these appear as preconditioners of iterative algorithms that accelerate the convergence. These operations account for a large fraction of total execution time of many sparse iterative linear algorithms, and due to the challenges in their parallelization, their proportion will only increase as the trend of increasing number of cores continues. For example, the High Performance Conjugate Gradient (HPCG) benchmark was recently proposed as an alternative to HPL for ranking high performance computing systems [17]; about 2/3 of its total execution time is spent on a symmetric Gauss-Seidel preconditioner with loop carried dependences [18].

Consider the following code that implements a forward Gauss-Seidel relaxation, with the system $Ay = b$.

```
1 for (i=0; i < N; i++)  {
2   y[i] = b[i];
3   for (j=rowptr[i];j<rowptr[i + 1];j++) {
4     y[i] -= values[j]*y[colidx[j]];     }
5   y[i] = y[i]*idiag[i]; }
```

Listing 1. Gauss Seidel Code.

There are data dependences between reads and writes of vector `y`, and these cannot be determined until runtime when the values of `colidx[j]` are known. One way to parallelize this computation is to inspect at runtime the values of `colidx[j]` to identify the dependent iterations of `i` such that `i==colidx[j]`. Dependences are represented by directed acyclic graphs (DAGs), which are traversed in a topological order. This traversal produces a list of *level sets*; a level set consists of iterations that can execute in parallel, and the ordering in the list captures dependences among level sets. The approach of parallelizing within level sets, with synchronization between level sets, is *wavefront parallelism*.

There has been a long tradition of the compiler-generated inspector/executor approach that optimizes recording memory accesses to derive the data dependences [2], [19]–[21]. Nonetheless, many wavefront parallelizations of Gauss-Seidel and related algorithms employ manually written inspectors and executors [22]–[25]. The hand-optimized inspectors reduce the inspector time and storage by iterating over the dependences

SC16; Salt Lake City, Utah, USA; November 2016
978-1-4673-8815-3/16/$31.00

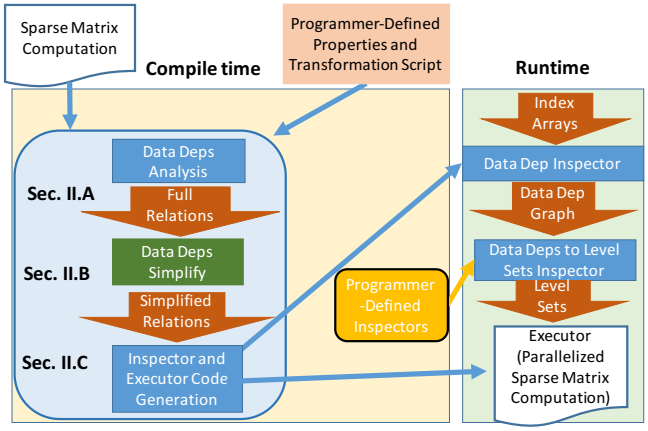

Fig. 1. Overview of the compilation system and the generated inspector and executor code. The compiler performs data dependence analysis, applies a simplification algorithm to the data dependence relations, and then generates the inspector and executor code including calls to a programmer-defined inspector that converts the runtime data dependence graph to level sets. The executor, a transformed version of the input code, contains an outer serial loop and then an inner parallel loop for each level set.

in the loop directly at runtime. This avoids the need to store information about which iterations access each data element, and usually results in an inspector loop that uses space and time comparable to the sparse matrix size.

In this paper we present an algorithm for automatically simplifying the compile-time specification of the data dependences so that the compiler-generated inspector uses time and space comparable to the sparse matrix size. This paper makes the following contributions: (1) it performs compile-time dependence simplification for a class of sparse matrix computations that allows reduced overhead for both inspector and executor; (2) it automatically generates the inspector for dependence checking and integrates this with a library implementation for deriving level sets, and these are integrated with a compiler-generated executor; (3) it demonstrates the effectiveness of parallelization over sequential execution for a preconditioned conjugate gradient algorithm (PCG); and, (4) it demonstrates the efficiency and accuracy of the compiler-generated inspector.

II. Simplifying Data Dependences for Efficient Inspectors and Executors

The goal of this paper is to automate the generation of inspectors and executors for wavefront parallelism and make that code as performant as hand-tuned code. Our approach involves the following steps, which are illustrated in Figure 1.

1) Perform data dependence analysis to derive data dependence relations.
2) Simplify the data dependence relations by projecting out as many iterators as possible.
3) Generate optimized inspector code that enumerates the simplified data dependence relations to create the data dependence graph at runtime.
4) Generate the executor code that utilizes the runtime dependence information to implement the wavefront parallelization.

A. Data Dependence Analysis

Compile-time data dependence analysis statically determines constraints on reordering memory accesses. More concretely, a dependence exists between statement S1 at iteration $I = [i_1, i_2, \ldots, i_N]$ and statement S2 in a subsequent iteration $I' = [i'_1, i'_2, \ldots, i'_N]$ (i.e., $I < I'$) if both of the iterations potentially access the same memory location, at least one of those accesses is a write, and the iterations lie within the loop bounds. Dependence testing involves introducing loop bound constraints and testing whether there is a possible solution to $I = I'$.

When the array indexing functions are affine, it is possible for the compiler to prove whether or not the dependence relation is satisfiable. However, when the array indexing functions are non-affine determining the dependences at compile time is not possible. For instance, consider the Gauss-Seidel code in Listing 1. The data dependence relation between the write `y[i]` and the read `y[colidx[j]]` on line 4 is

$$\begin{aligned}\{[i,j] \rightarrow [i',j'] \quad | \quad & (i < i' \vee (i = i' \wedge j < j')) \\ & \wedge\ 0 \leq i, i' < N \\ & \wedge\ rowptr(i) \leq j < rowptr(i+1) \\ & \wedge\ rowptr(i') \leq j' < rowptr(i'+1) \\ & \wedge\ i = colidx(j')\} \\ \cup\{[i',j'] \rightarrow [i,j] \quad | \quad & (i' < i \vee (i = i' \wedge j' < j)) \\ & \text{same as above}\}.\end{aligned}$$

In this paper, we automatically generate a runtime inspector to perform dependence testing involving such indirection. During compile-time dependence testing, the compiler uses uninterpreted functions to represent information that will not be known until runtime. For example, the uninterpreted function $colidx()$ represents the index array `colidx[]` for the Gauss-Seidel code in Listing 1.

In general, we assume that the outermost loop level is chosen for wavefront parallelization. We specify this in general by forcing inequality on the outermost iterators i_1 and i'_1 and having the inner loop iterators be existentially quantified. In the Gauss-Seidel example this leads to a rewrite of the data dependence relation where the focus is the loop-carried dependences for the shared outer `i` loop and the inner loop iterators become existentially quantified as shown in Figure 2.

At compile time, we generate inspector code that iterates over all pairs in the data dependence relation for the loop-carried dependences. At runtime, the inspector explicitly constructs the dependence graph to connect all such pairs of iterations where a dependence exists. For Gauss-Seidel, the runtime inspector that would be generated naively from this dependence relation would result in two separate loop nests (one for each conjunction) and each loop nest would be four deep (i', j', i, j), as shown in Listing 2. Sections II-B and II-C

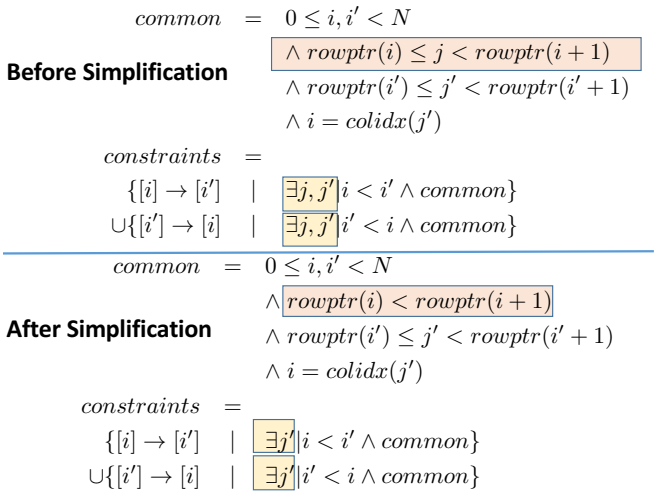

Fig. 2. Data dependences for outer loop of Gauss-Seidel from Listing 1 before and after simplification.

present how the data dependence relations can be simplified to result in less deeply nested loop nests and code generation can reduce the number of loops needed.

B. Data Dependence Simplification via Projection

To reduce inspector overhead, we simplify the data dependence relations by eliminating the existentially quantified inner loop iterators. It also determines equalities that can aid in such projections. For the Gauss-Seidel example, it is possible to project out the j iterator leading to the After Simplification dependence relation shown in Figure 2. Each iterator that can be projected out results in the inspector code being one less loop level deep. For the Gauss-Seidel example, this reduces the iteration space by a multiplicative factor equal to the average number of nonzeros per row in a matrix.

Any iterator that does not have a dependent uninterpreted function call (i.e., the iterator is not passed to an uninterpreted function call in all of the constraints) can be projected out of the constraints. However, the problem is that there is no straightforward method to project out variables from an integer space with uninterpreted functions. To solve this problem, the simplification algorithm replaces all uninterpreted function call terms with fresh symbolic constants (also called parameters in polyhedral code generators). This replacement is similar to Akermann's reduction, which is used to reduce satisfiability problems with uninterpreted functions to equality logic [26]. One difference is that our simplification algorithm uses required domain and range information about each uninterpreted function to bound the actual parameter expressions to the function calls and the values of the function calls.

By replacing the uninterpreted function calls with symbolic constants and inserting bound and range constraints, we create an affine relation that is a superset of the non-affine relation containing uninterpreted function calls. Next, the simplification algorithm uses a polyhedral algorithm to project out the targeted iterators from the affine superset relation. The final step of the simplification algorithm is to reverse the substitution.

```
1  // Naive Inspector Code
2  //   Input: Dependence relations
3  //   Output: Dependence Graph(A)
4  for(ip=0; ip<=m-1; ip++){
5   for(jp=rowptr[ip]; jp<rowptr[ip+1];jp++){
6    for(i=0; i<=m-1;i++){
7     for(j=rowptr[i]; j<rowptr[i+1];j++){
8      if(col[jp] == i){
9       if(i < ip){connectTo(A,i,ip);}
10      else if(ip < i){connectFrom(A,ip,i);}
11           }}}}}
```

Listing 2. Naive Inspector Code for Gauss-Seidel.

C. Optimizations for Generated Inspector Code

To generate the inspector code, we utilize polyhedra scanning to enumerate the dependences between iterations of the outermost loop. When a dependence is found between two iterations of the outer loop, those iterations are connected in the dependence graph with an edge (note the `connectFrom()` and `connectTo()` functions in Listing 3). We use the CodeGen+ tool for code generation because it can generate code for polyhedra [27] and is also capable of handling uninterpreted function constraints [28] and loop bounds [29][1].

Listing 3 illustrates the inspector and executor code when the input to the wavefront parallelization compiler transformation is the Gauss-Seidel example from Listing 1. The `connectFrom()` and `connectTo()` functions populate the dependence graph, `A`, with edges. The dependence graph is represented as each iteration having a set of incoming and outgoing dependence edges and then another pass creates an adjacency list representation with just incoming edges.

The code generator is able to improve the generated inspector code by performing the following three optimizations.

1) Unnecessary loop removal optimization: Data dependence analysis creates data dependence relations where array index expressions must be equal for a dependence to exist between two iterations of the loop being wavefront parallelized (e.g., the equality constraint $i = col(j')$ for the Gauss-Seidel example). Utilizing polyhedra scanning as the underlying methodology for enumerating dependences has a major advantage that any time an iterator can be expressed as a function of other iterators, the code generator will *not* generate a loop for that iterator (e.g., the iterator i in the Gauss-Seidel example). For the Gauss-Seidel example, this reduces the need for a three deep nested loop after simplification to a two-deep nested loop only containing the iterators i' and j', eliminating the i loop.

[1]One limitation in the code generator that we work around is that each uninterpreted function call must be made to accept a prefix of the iterators for the set. For example in the Gauss-Seidel code specification $i = col(j')$ is converted to $i = col(i', j')$.

```
1  // Inspector Code
2  //   Input: Dependence relations
3  //   Output: Dependence Graph(A)
4  #pragma omp parallel for private(ip,jp,i)
5  for(ip=0; ip<=m-1; ip++){
6   for(jp=rowptr[ip]; jp<rowptr[ip+1]; jp++){
7     i = colidx[jp];
8     if(i < ip)      { connectTo   (A,i,ip); }
9     else if(ip < i) { connectFrom(A,ip,i); }
10 }}
11
12 // Level Set Construction (Inspector Code)
13 //   Input: Dependence Graph(A)
14 //   Output: Level Sets(levelSetBoundaries)
15 deriveLevelSets(A, &levelSetBoundaries,
16                 &num_levels);
17
18 // Executor Code After
19 //  Reordering(Parallelized)
20 for(i=0; i<num_levels; i++){
21  #pragma omp parallel for private(j,k,sum)
22  for(j=levelSetBoundaries[i];
23      j<levelSetBoundaries[i+1]; j++){
24   sum = b[j];
25   for (k=rowptr[j]; k<rowptr[j+1]; k++){
26    sum -= values[k] * y[colidx[k]];
27   }
28   y[j] = (sum * idiag[j]);
29 }}
```

Listing 3. Example of compiler output.

For the ILU0 example, which is described in Section III, more advanced features of the simplification algorithm result in a useful equality constraint being derived even though it was not in the original data dependence relation.

2) Loop fusion: In general as well as in the example in Figure 2, a data dependence relation for the outermost loop will contain two conjunctions: one for the case when $i_1 < i'_1$ and another for $i'_1 < i'_1$. This would typically lead to two loop nests in the inspector, one for each conjunction. It is possible to fuse the loops and place those ordering constraints in the loop body to select which edge to insert into the graph (see lines 8 and 9 in Listing 3).

3) Parallelization: Since it is critical that the generated inspector is efficient to reduce runtime overhead, the code generator parallelizes the outer loop of the inspector. This is possible because of what is known about the structure of the data dependence relations and the inspector code after loop fusion. Because the outer loop in the inspector will either be i_1 or i'_1 from the dependence relation, we can parallelize that outer loop by ensuring that the `connectFrom()` call always has the outer inspector loop iterator passed in as the source of the data flow dependence edge and the `connectTo()` passes it as the target. This results in the outgoing and incoming edges data structure only being updated based on the outer loop iterator and thus being parallelizable.

D. Parallelized Executors

At run-time, once the level-sets are constructed, all the iterations belonging to a level set can be scheduled in parallel without any dependence violation, but synchronization between level sets is required to ensure correctness (see Listing 3 for barrier synchronization variant). In Listing 3 `levelSetBoundaries[]` denotes the set of tasks that belong to each level set, or wavefront.

```
1  for(i=0; i < m; i++) {
2   for(k= rowptr[i]; k < diagptr[i]; k++){
3    values[k] = values[k]/
4          values[diagptr[col[k]]];
5    j1 = k + 1;
6    j2 = diagptr[col[k]] + 1;
7    while (j1 <  rowptr[i+1] &&
8            j2<rowptr[col[k]+1]) {
9     if(col[j1] == col[j2] {
10     values[j1] -= values[k]*values[j2];
11     ++j1; ++j2;
12    }
13    else if (col[j1] < col[j2]) { ++j1; }
14    else { ++j2; }
15 }}}
```

Listing 4. ILU0 Code

III. Advanced Simplification Based on Additional Information

In this section, we present techniques for improving our simplification algorithm especially for more complicated examples. First, we present an approximation heuristic that removes some of the constraints based on user input and enables the simplification algorithm to project out more iterators. Next, we discuss techniques for discovering equalities between iterators and expressions so that such iterators will not require a loop to be generated for them in the inspector. The additional information that enables this process is knowledge about whether index arrays are monotonically increasing and relationships between index arrays used to represent sparse matrices (e.g., $\forall e, rowptr(e) \leq diagptr(e)$).

A. Incomplete LU Factorization Example

We use an Incomplete LU (ILU0) code throughout this section as a case study (see Listing 4). Dependence analysis of this code would identify eight pairs of read/write or write/write data accesses that may be data dependences[2]. Each pair produces two distinct conjunctions considering the ordering of accesses. Overall there are 16 distinct conjunctions for the complete dependence relations for ILU0. One dependence occurs due to the write to the array `values[j1]` in line 10 of Listing 4, and read from the same array using iterator $j2$. Figure 3 shows the constraints for this flow dependence (read after write). Another data dependence again in line 10 would be writing to array values with $j1$, and reading with k.

The naive inspector code for ILU0 without any simplification contains a loop with a nesting depth of eight. This inspector does not even finish after one day for some of the big matrices with which we experimented. The simplification algorithm as described in Section II cannot project out any

[2]Using typical compiler analyses the lower bounds of $j1$ and $j2$ can be determined as well as the fact that they increase by at most one each iteration of the while loop.

iterators from the sets of constraints listed in Figure 3. The only simplification for ILU0's inspector based on projection is that the code generator can remove one loop because of the equality $j1 = j2$.

With the approximation techniques described in this section, we are able to project out four iterators. After approximation has occurred, using information about the uninterpreted functions enables the derivation of the equality $i = col(j')$, which enables the code generator to avoid creating a loop for the iterator i. Our simplification algorithm results showed that of the 16 conjunctions for ILU0's dependence relations, 12 are not satisfiable and the remaining four simplify to two distinct conjunctions.

B. Approximation heuristic for removing expensive iterators

Here we present a heuristic that improves our simplification algorithm utilizing the concept of conservative approximation. An iterator is expensive if: (1) it is not one of the iterators in the outer loop we are attempting to wavefront parallelize; and, (2) it is an argument to some uninterpreted function call and therefore cannot be directly projected out. It is desirable to project out expensive iterators; however, if we project out a variable that is part of an actual argument in an uninterpreted function call, the dependence relation will become an approximation. The approximation is conservative; if any of the constraints in a data dependence are removed then the set of data dependences is a superset of the actual dependences. In the extreme, the data dependence relation includes all pairs of iterations in the outer loop. However, this would lead to losing all of the parallelism by producing a huge over-approximation of dependences. Instead, our experience shows there are dependence relations from which removing a nominal set of constraints enables us to project out more iterators. This leads to significant performance gains for the inspector, and an inconsequential loss of parallelism in the executor.

Consider the ILU0 example, where the data dependence relation contains eight iterators: $i, k, j1, j2, i', k', j1', j2'$. Hence, a naive inspector for this code is an eight-deep nested loop that only terminates for small-sized matrices and is up to 6 orders of magnitude slower than an inspector that does an approximate dependence test. There are two constraints in ILU0's dependence relation that prevent projecting out 4 of the expensive iterators: $col(j1) = col(j2)$ and $col(j1') = col(j2')$. These constraints are keeping us from projecting out four iterators: $j1$, $j2$, $j1'$, $j2'$. In our experiments, we removed these two constraints, and by doing so we were able to project out four more iterators. In theory, the resulting parallel schedule could have lost a certain amount of parallelism. However, taking out these two constraints had an insignificant impact on parallelism while decreasing execution time of the inspector by an order of magnitude.

Therefore, in addition to the simplification algorithm, we provide an approximation heuristic that enables a user to specify how many constraints to remove from the original set of constraints in the data dependence relation. By exposing

$$i < i' \tag{1}$$
$$0 \leq i, i' < n \tag{2}$$
$$rowptr(i) \leq k < diagptr(i) \tag{3}$$
$$rowptr(i') \leq k' < diagptr(i') \tag{4}$$
$$k + 1 \leq j1 < rowptr(i + 1) \tag{5}$$
$$k' + 1 \leq j1' < rowptr(i' + 1) \tag{6}$$
$$diagptr(col(k)) + 1 \leq j2 < rowptr(col(k) + 1) \tag{7}$$
$$diagptr(col(k')) + 1 \leq j2' < rowptr(col(k') + 1) \tag{8}$$
$$col(j1) = col(j2) \tag{9}$$
$$col(j1') = col(j2') \tag{10}$$
$$j1 = j2' \tag{11}$$

Fig. 3. Data dependence constraints for Incomplete LU (ILU0) Factorization on a compressed sparse matrix.

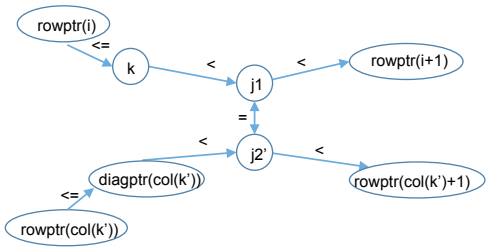

Fig. 4. Partial ordering of terms that leads to the constraint $rowptr(i) < rowptr(col(k') + 1)$.

the number of constraints to remove as a parameter, it could later be exposed to an autotuner. For this work, we set the parameter manually for each test case based on the inspectors that had been written manually in the past.

The approximation heuristic loops over the iterators from innermost to outermost, where the iterators for ILU0 are ordered outermost to innermost as follows: $i, i', k, k', j1, j1', j2, j2'$. For each iterator, the algorithm determines how many constraints exist where that iterator is part of an actual parameter to an uninterpreted function call. As long as that number of constraints does not move the heuristic above the user-specified limit, the constraints are removed and the iterator is projected out of the data dependence constraints.

C. Using Information about Uninterpreted Functions

We find opportunities for specifying one iterator in a dependence in terms of another iterator (i.e., an equality the code generator can remove) by using information about the uninterpreted functions such as their domain and range, whether the uninterpreted function is monotonic, and additional constraints that a programmer might be able to provide such as relationships between various uninterpreted functions.

1) Domain and Range Information: The index arrays used in the compact representation of sparse matrix nonzeros have well-defined domains and ranges. For example, in the Gauss-

Seidel example (Listing 1), the domain of `colidx[]` is from 0 to $nnz - 1$, where nnz is the number of nonzeros in the sparse matrix. Our simplification algorithm includes a step where all of the parameters to uninterpreted function calls and the calls themselves are provided bounds due to the domain and range information associated with each index array. Our compiler determines the domains and ranges of the index arrays involved in the computation based on declarations of the index arrays and how such index arrays are accessed.

2) User-Defined Constraints: User-defined constraints involving the index arrays can be specified to generate additional inequalities as well. This information is specific to the sparse matrix data structure itself. As an example, for the `rowptr[]` and `diagptr[]` index arrays used in the ILU0 example, it is the case that $\forall e, rowptr(e) \leq diagptr(e)$, which indicates that the start of the non-zeros for any particular row will start before or at the same time as the index to the diagonal for a row. The algorithm instantiates the user-defined constraint for every expression that is passed to either of the uninterpreted functions involved in the constraint. For the ILU0 example in Figure 3 and the user-defined constraint that $\forall e, rowptr(e) \leq diagptr(e)$, some of the constraints that are added include $rowptr(i) \leq diagptr(i)$ and $rowptr(col(k') + 1) \leq diagptr(col(k') + 1)$.

3) Monotonicity Constraints: Programmers can also indicate when an uninterpreted function is monotonically nondecreasing, increasing, decreasing, or nonincreasing[3]. We use the monotonicity rules to leverage that partial ordering and create more constraints on expressions being passed into uninterpreted function calls. Specifically, if a function f is monotonically non-decreasing, then the following two properties hold:

$$f(e_1) \leq f(e_2) \Leftrightarrow e_1 \leq e_2$$
$$f(e_1) < f(e_2) \Rightarrow e_1 < e_2.$$

The first property would enable the derivation of inequalities between uninterpreted function calls from inequalities between expressions, but that process could infinitely recurse so our current implementation does not utilize that direction of the equivalence.

4) Using New Constraints to Derive Equalities: To discover equalities if possible, the simplification algorithm collects partial orderings between terms in the constraints. Figure 4 illustrates some of the partial ordering that exists for the ILU0 example after the constraints due to the user-defined property $\forall e, rowptr(e) \leq diagptr(e)$ have been added. This subset of the partial ordering highlights that we can derive the constraints $rowptr(i) < rowptr(col(k') + 1)$ and $rowptr(col(k')) < rowptr(i + 1)$. Using the fact that $rowptr()$ is monotonically non-decreasing, these result in the constraints $i < col(k') + 1$ and $col(k') < i + 1$, which lead to the equality constraint $i = col(k')$.

In summary, for the example ILU0 computation on a compressed sparse row matrix with an index array indicating diagonals, the constraints in the dependence relation are shown in Figure 3. After the approximation heuristic described in Section III-B has been applied with the tuning parameter set to remove two constraints, constraints (9) and (10) in Figure 3 will be removed. By using information available about the uninterpreted functions and the remaining constraints after approximation, it is possible to determine that $i = col(j')$. This equality removes the need for the code generator to generate a loop for the i iterator.

[3]Compiler analyses can determine monotonicity in some cases [30].

IV. Implementation

Implementation of the automatic wavefront parallelization of sparse codes uses a polyhedral transformation and code generation framework comprised of CHiLL [31], IEGenLib [32], Omega+ [28] and Codegen+ [27]. The generated code is integrated with the SpMP [33] open source library for parallelizing sparse computations.

CHiLL is the polyhedral transformation framework that represents the input code's iteration space as a set and the transformations as mappings/functions on sets. Omega+ is the underlying polyhedral library that provides these set and relation abstractions. CHiLL extracts the dependence constraints from the input code by parsing array access expressions and loop bounds. These constraints may involve uninterpreted functions symbols and therefore could be non-affine. IEGenLib simplifies these constraints according to the algorithms outlined in Sections II and III. CodeGen+ utilizes the simplified constraints generated by IEGenLib to generate the parallel inspector code that constructs the dependence graph outlined in Section II-C. CodeGen+ also utilizes Omega+ for integer hull computation and standard affine projection operations that are used in polyhedra scanning

The SpMP library constructs the level sets using a modified breadth-first search algorithm. CHiLL interfaces with SpMP to insert wrappers around SpMP's sparse matrix and dependence graph data structures which are exported to CHiLL via SpMP header files parsed by the ROSE [34] compiler infrastructure. CHiLL and CodeGen+ also use ROSE to manipulate the C/C++ code AST.

A new transformation *sparse_partition* was developed in CHiLL to partition a particular loop level, typically the outermost one, into wavefronts of irregular length. The start and end of each irregular partition is represented using index arrays, which are represented using uninterpreted functions in CHiLL. The transformations prepare the iteration space of the executor code. For the barrier synchronization variant of the code, we use CHiLL's OpenMP code generation capabilities [35]. For the point-to-point synchronization, the compiler generates SpMP's specialized post-and-wait synchronization primitives.

V. Experimental Results

In these experiments, we evaluate the performance of the generated code for the Preconditioned Conjugate Gradient (PCG) that has been used in a prior study [25]. We evaluate 2 variants of the PCG benchmark, (i) With a symmetric Gauss Seidel preconditioner and (ii) An Incomplete LU with

zero fill-ins (ILU0) preconditioner. Both preconditioners are expressed as products of a lower and upper triangular matrix. The symmetric Gauss Seidel preconditioner is expressed as (L+D)*D^{-1}*(U+D), where D is the diagonal, and L and U are the lower and upper triangular portions of the input matrix A. ILU0 factorization is computed once to find L and U matrices such that L*U is close to A [36], where L and U are identically defined as in the symmetric Gauss Seidel preconditioner. Both variants of the solver use forward and backward Gauss-Seidel relaxations that execute each iteration to solve for the sparse lower and upper triangular systems of equations, respectively. ILU0 preconditioning in general leads to faster convergence than symmetric Gauss Seidel preconditioning.

The compiler generates two distinct implementations of the synchronization: (1) an OpenMP barrier between level sets; or, (2) a point-to-point synchronization between level sets (Post and Wait in SpMP). We use as input a representative set of matrices from the University of Florida sparse matrix collection [37], listed in Table I.

We measure performance gains on a compute node of Edison at NERSC. The compute node is equipped with a dual-socket Intel Ivy Bridge generation Xeon E5-2695 v2 running at 2.4 GHz and with 64 GB of DDR3 1866 MHz main memory. Each socket has 12 cores and a shared LLC with capacity 30 MB. Each core has a private 32 KB L1 data cache and a private 256 KB L2 cache. For our experiments, we use only 12 cores in one of the two sockets, thus avoiding NUMA effects and focusing the results on the efficiency of the generated code. We use one thread per core because hyperthreading does not speed up the sparse matrix operations we evaluate.

The resulting parallel code is compiled with Intel C compiler (icc) version 17.0.0, and we compare our performance results with Intel Math Kernel Library (MKL) version 2017.

In this section, we first show the profitability due to parallelism of the generated ILU0 factorization in Section V-A. Section V-B examines the performance of the generated Gauss-Seidel executor, and compares against sequential performance and Intel's MKL library. Additionally we look at the overhead of the inspection time for Gauss-Seidel. In Section V-C, we compare overall parallel performance of the generated PCG to that of the Intel MKL library and the manually-tuned code from [25].

A. Optimizing ILU0 Preconditioner

We first consider the impact of parallelizing the ILU0 factorization, which is used to improve the convergence behavior of the PCG algorithm. Since ILU0 is called only once, it is critical that the generated inspector be very efficient so that the overhead is far below the gains from the parallel executor. Figure 5 compares the combined parallelized inspector and executor time for ILU0 (using point-to-point synchronization), normalized to the execution time of the sequential ILU0 implementation. We show the individual contributions to ILU0 execution time: the parallelized inspector that constructs the dependence graph, the parallelized inspector that constructs the level set, and the parallelized executor.

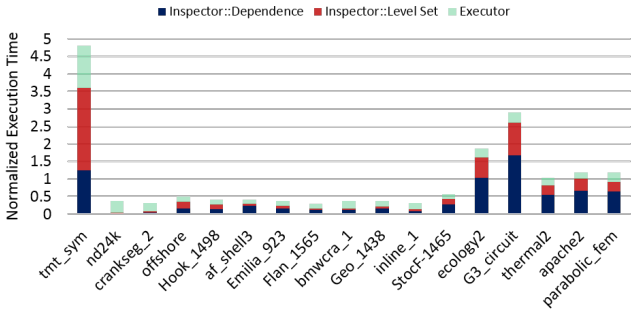

Fig. 5. ILU0 Factorization Performance. The total inspection and execution time for parallel ILU0 is normalized to the sequential ILU0 execution time, and this value is plotted on the y-axis.

TABLE I
INPUT MATRICES SORTED IN ORDER OF INCREASING PARALLELISM

Matrix	Rows	Average number of nonzeros/row	Parallelism	Overall serial PCG time(s)
tmt_sym	726,713	6.99	1.00	22.09
nd24k	72,000	398.82	6.28	31.27
crankseg_2	63,838	221.64	14.53	8.33
offshore	259,789	16.33	75.28	2.49
Hook_1498	1,498,023	39.64	95.92	173.09
af_shell3	504,855	34.79	135.57	1.07
Emilia_923	923,136	43.74	176.17	29.96
Flan_1565	1,564,794	72.96	200.49	524.26
bmwcra_1	148,770	71.53	204.35	23.79
Geo_1438	1,437,960	41.89	246.99	49.59
inline_1	503,712	73.09	287.67	557.71
StocF-1465	1,465,137	14.34	487.89	139.14
ecology2	999,999	5.00	500.50	32.12
G3_circuit	1,585,478	4.83	611.45	23.84
thermal2	1,228,045	6.99	992.96	67.16
apache2	715,176	6.74	1,078.70	9.93
parabolic_fem	525,825	6.98	87,637.50	6.48

In Figure 5, the input matrices are sorted in increasing order according to their average parallelism, measured as the total number of nonzeros in the matrix divided by the total number of nonzeros in the rows along the chain of dependent rows on the critical path. This is an approximate indicator of the available parallelism, and is shown in the *Parallelism* column of Table I.

On most matrices (11 out of 17), parallelization yields a speedup, as the combined inspector and executor time is less than the sequential time. The best speedup is roughly $3\times$ for *crankseg_2*. The six matrices for which parallelization was not beneficial have the lowest number of average nonzeros per row and have relatively fast sequential implementations. Executor time has quadratic complexity with respect to the average number of nonzeros per row: i.e., $O(n(m/n)^2)$ where n is the number of rows and m is the number of non-zeros. In contrast,

inspection time heavily depends on the number of vertices in the dependence graph, which equals the number of rows of the matrix. Therefore, for matrices with fewer nonzeros per row, we tend to have a higher inspection time relative to execution time.

Here we would like to quantify the effect that approximation of dependences outlined in Section III for ILU0 factorization has on overall performance. For all input matrices we did not observe any difference in measured parallelism for the precise and approximate versions of the test.

B. Gauss-Seidel Inspector and Executor Performance

As opposed to the inspection for ILU0 factorization, the parallel wavefront schedules computed for the Gauss-Seidel relaxations are reused multiple times. In Table II, we characterize the inspector overhead, which includes the total time to construct the dependence graph as well as the modified breadth-first search to derive the wavefronts. For reference, the second column is the number of iterations to convergence for each matrix. The third column shows the inspector time in relation to the time taken per iteration of the preconditioned conjugate gradient solver, which includes a forward and backward Gauss-Seidel relaxation. Then, the fourth column is the inspector time as a percentage of the total time for all iterations required for convergence. Evidently, the average inspection overhead is approximately 3.68% and the maximum is 32.33%.

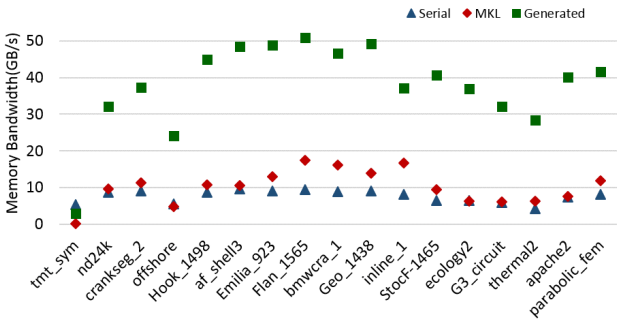

Fig. 6. Performance of Parallel Gauss-Seidel Executor.

More than two thirds of the total execution time of PCG is spent in the forward and backward Gauss-Seidel relaxations. We report the performance of the parallelized Gauss-Seidel executor code in Figure 6, and compare with the serial implementation, and Intel's MKL library. The y-axis shows effective memory bandwidth (GB/s), calculated by dividing the size of matrix and input/output vectors by the time taken for the symmetric Gauss Seidel relaxations. Since sparse matrix operations are memory bandwidth bound due to their low arithmetic intensity, this is a useful metric to quantify how close the performance is to the machine peak [38]. Speedups of the compiler-generated code over the serial and Intel MKL versions, calculated as a median across the matrices, are $5.26\times$ and $3.95\times$, respectively.

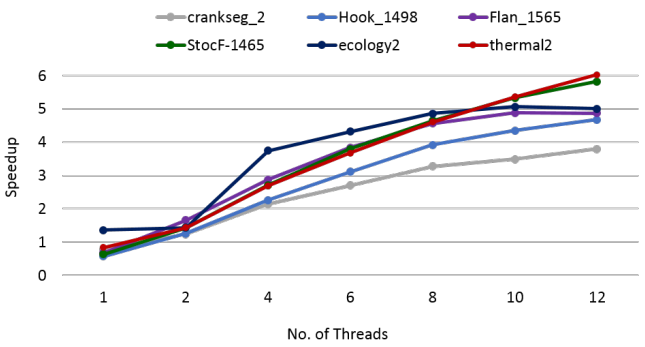

Fig. 7. Scaling Behavior of Parallel Gauss-Seidel Executor.

In addition, we consider absolute performance based on the bandwidth measurements on the y-axis. As described in [25] the maximum parallel performance attainable by the Gauss-Seidel relaxation is capped by the performance of Sparse Matrix Vector Multiply (SpMV) on the corresponding matrix as they have the same number of floating point operations per nonzero and SpMV is more easily parallelized. On average the SpMV performance measured for our configuration was approximately 50 GB/s, again matching the bandwidth on the same configuration as measured by the STREAM benchmark [39], [40]. We achieve on average 74.9% of the maximum STREAM bandwidth for the Gauss-Seidel Relaxations. Except for *tmt_sym*, which has virtually no parallelism as indicated by Table I, we observe a speedup due to parallelization for the compiler generated code on all inputs.

In Figure 7 we present the scaling of the multi-threaded Gauss Seidel executor for 6 representative matrices. We measure the speedup of the multithreaded executors over the sequential implementation. Both the amount of average parallelism and the average number of nonzeros per row determine the scaling of the executors. The available work is a complex function of both the number of rows and average nonzeros per row as the wavefronts are partitioned by rows. The matrices *thermal2* and *StocF-1465* have relatively high parallelism and available work and thus achieve the highest speedups for 12 threads. The matrix *ecology2* is representative of those with relatively high parallelism but low work and they do not scale well beyond 10 threads as there is not enough work to keep all cores busy. *Flan_1565* and *Hook_1498* have sufficient work but their moderate parallelism prevents scaling beyond 10 threads. *crankseg_2* has both modest parallelism and work and so scales worse than the others.

C. Overall Performance of PCG

Here we quantify the performance impact of all our optimizations for parallel PCG. For all input matrices, we demonstrate the speedup obtained over the reference sequential implementation, and compare against the Intel MKL library and manually-optimized code from [25].

We compute execution time for parallel PCG by summing the execution time of the ILU0 inspector and executor, Gauss-

TABLE II
INSPECTOR OVERHEAD MEASURED AS % OF TOTAL ITERATIONS TO CONVERGENCE.

Matrix	Convergence Iterations	Inspector Overhead #	Inspector Overhead %
tmt_sym	1176	3.48	0.30
nd24k	381	22.49	5.90
crankseg_2	177	25.20	14.24
offshore	186	4.71	2.53
Hook_1498	1580	7.53	0.48
af_shell3	22	7.11	32.33
Emilia_923	411	7.75	1.89
Flan_1565	2817	8.86	0.32
bmwcra_1	1300	8.70	0.67
Geo_1438	443	7.63	1.72
inline_1	8399	15.73	0.19
StocF-1465	2493	4.56	0.18
ecology2	1791	2.92	0.16
G3_circuit	755	3.00	0.40
thermal2	1657	3.25	0.20
apache2	723	3.60	0.50
parabolic_fem	678	4.04	0.60
		Average	3.68

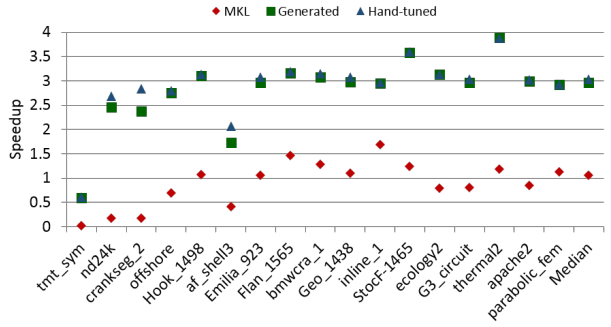

Fig. 8. Speedup of Parallel PCG over Sequential PCG.

Seidel inspector and overall solver time. This is compared with the time for a PCG implementation utilizing sequential ILU0 factorization and Gauss Seidel relaxations. For reference, the serial time is shown in Table I. The results in Figure 8 show the speedup of the parallel PCG kernel over the serial PCG implementation taking into account both the total inspector overhead and the executor times. We observe a median speedup of 2.97× over the serial implementation, for the point-to-point version. Although not shown here, the barrier version achieves a slightly more modest median speedup of 2.29×. The manually-tuned code and Intel MKL achieve a median speedup of 3.03× and 1.05× over the serial implementation. The manually tuned code has a slightly lower inspection overhead as it does not construct the dependence graph explicitly but uses the input matrix directly as the dependence graph, since the structure of the sparse matrix is equivalent to the dependence graph for the purposes of Gauss Seidel relaxations. Its inspector overhead only involves constructing the wavefronts from the input matrix.

Overall, these results demonstrate the efficacy of an Inspector/Executor approach to parallelize PCG. The compiler-generated code is competitive with manually-tuned code.

VI. RELATED WORK

This section broadly classifies related research into three categories and describes them in details below.

A. Compiler-Based Approaches

There have been several efforts in wavefront parallelization of loop nests that contain statically unanalyzable access patterns. One of the most widely used techniques is the runtime approach via inspector-executor [41]. Our paper uses a similar approach but with some key differences. Although they parallelize the inspector like ours, their inspector code still retains the original structure of the loop nest, which can give rise to increased time and space complexity. For example, in the ILU0 example, since we remove some constraints thus approximate the dependences, our inspector loop is less deeply nested than their approach. Moreover, during inspection they explicitly build a dependence graph for each memory location, which as they show benefits reduction and array privatization, but results in increased space consumption for inspection. In contrast, our approach simplifies the inspector via assertions [42] and results in simplified and faster inspector code.

Previous compiler-based research includes many techniques for reducing the inspector overhead by parallelizing the inspector [21], [41], overlapping inspectors with executors [21], [24], and using a GPU to perform inspection [8], [24]. One of the compilation approaches by Zhuang et al. [21] incorporated the techniques into the IBM XL compiler and observed speedups for some benchmarks from SPEC2000, SPEC2006, and SEQUOIA on a 16 core Power 5 machine. Their approach is similar to ours in that they use approximation, although not one based on the data dependence relations. Their approach is also complementary in that it would be possible to use their technique of overlapping the generated inspector and executor in the code we generate. Performance improvements in executors are possible by using a finer-grained, run-time dependence analysis that enables some overlapping of dependent iterations in the executor [43].

Ravishankar et al. [44] present compiler algorithms for generating inspectors and executors that parallelize producer/consumer loop sequences when there are indirect memory accesses. This is a different parallelization than wavefront parallelism, which is within a single loop instead of across multiple loops. Our technique for simplifying data dependences at compile-time could also help reduce the overhead of the inspectors in this work.

Vanroose et al. [45] propose to optimize pipelined Krylov solvers on multiple nodes. Specifically they target the multiple preconditioned SpMV operations applied per time step. They reformulate the SpMV operations as stencil accumulations and then extract wavefront computations by applying compiler-based time-skewing. This would improve the arithmetic intensity of the SpMV computations. The skewing compiler

transformation can extract wavefronts as the dependence distance is constant, i.e. the values computed in each time step is dependent on those from the previous one. In the case of the sparse matrix computations we optimize, the dependence distance, in terms of iterations of the outermost loop are not constant and hence necessitate the use of an inspector/executor method to extract dependences at run-time and reorganize the computation into wavefronts. Further, SpMV operations can be rewritten as stencil applications on a grid only if the matrix has certain structure, eg. diagonal or banded, and our evaluation includes unstructured matrices as well, hence necessitating sparse matrix formats

In the HELIX project [46] speedups on the SPEC CPU200 benchmarks were observed on an Intel i7 with 6 cores. Their technique involves a profiling step to feed in information to a compiler algorithm that determines which loops to parallelize. Inter-iteration dependences within those loops are handled by injected code that performs synchronization between dependent iterations [47]. In [46], a number of optimizations are presented to make this process as efficient as possible.

Other compile-time and run-time techniques use as much compile-time information as possible to generate efficient run-time checks to determine if a loop is fully parallelizable [48], [49]. Loops that only admit wavefront parallelization will be determined not parallelizable by these approaches, but the inspector overhead is significantly reduced.

The work in [42] takes a different approach than runtime techniques: it uses user assertions about index arrays (which may otherwise be statically unanalyzable) to increase the precision of dependence testing. The assertions certify common properties of index arrays, e.g., an index array can be a permutation array, monotonically increasing, and monotonically decreasing. Our paper also uses these assertions, but it uses them to improve the performance of runtime techniques, i.e., it reduces the complexity of inspector code. Other data dependence analysis techniques that seek to find more parallelizable loops at compile-time include [30], [50]–[52].

B. Hand-crafted implementations for Sparse Matrix Kernels

There have been efforts on manually optimizing sparse matrix kernels with loop carried dependences, from Saltz and Rothbergs work on sparse triangular solver in 1990s [53], [54] to work on optimizing sparse triangular solver for recent NVIDIA GPUs and Intel's multi-core CPUs [18], [23], [25]. Even though these manual optimizations have been successful at achieving high performance in some cases, the significant software engineering efforts involved can be affordable only for a few high profile software projects with manageable code complexity. This software engineering challenge is also indicated by the fact that Intel MKL started incorporating inspector-executor optimization only very recently in their 11.3 beta version released around April 2015 [55]. One of the biggest challenges is composing the manual parallelization with other optimizations. Parallelization of kernels like sparse triangular solve needs to be composed with other optimizations such as reordering to realize its full potential as shown in [18]. Various preconditioners with different dependency pattern can be composed with numerous choices of iterative solver algorithms [16]. Our work addresses this composability issue with a compiler.

C. Polyhedral Compiler Optimization of Sparse Matrices

Strout et al. [56] first demonstrated the feasibility of automatically generating inspectors and composing separately specified inspectors using the sparse polyhedral framework; the inspector-executor generator prototype composed inspectors by representing them using a data structure called an inspector dependence graph. Subsequently, Venkat et al. [29], [57] developed automatically-generated inspectors for non-affine transformations and for data transformations to reorganize sparse matrices into different representations that exploit the structure of their nonzeros. Both these works demonstrated that the performance of the *executors* generated by this approach match and sometimes exceed performance of manually tuned libraries such as OSKI and CUSP [29], [57]; further, the automatically generated *inspectors* match and sometimes exceed the performance of preprocessing used by these libraries to convert into new matrix representations [57]. The data dependence simplification algorithm in this work could be used by compilers to reduce the overhead of the generated inspectors when the transformation in question needs to determine data dependences.

VII. Conclusions

In this paper we have demonstrated how a compiler dependence analysis and parallel code generation framework can be utilized to automatically generate wavefront parallelization of sparse matrix computations. The compiler simplifies loop-carried dependence constraints that arise in sparse computations with index array indirection and automatically generates optimized parallel inspectors and executors. The compile-time simplification reduces the time complexity of the inspectors so that it uses time and space comparable to the sparse matrix size. Our compiler-generated inspector and executor codes outperform the corresponding computations implemented with the Inspector/Executor interface of the Intel Math Kernel Library (MKL) and nearly match that of hand-tuned code.

Acknowledgments

Partial support for this work was provided through Scientific Discovery through Advanced Computing (SciDAC) program funded by U.S. Department of Energy Office of Advanced Scientific Computing Research under award number DE-SC0006947, and by NSF awards CNS-1302663 and CCF-1564074.

References

[1] J. H. Saltz, R. Mirchandaney, and K. Crowley, "Run-time parallelization and scheduling of loops," *IEEE Transactions on Computers*, vol. 40, no. 5, pp. 603–612, 1991.

[2] L. Rauchwerger, N. M. Amato, and D. A. Padua, "Runtime methods for parallelizing partially parallel loops," in *Proc. 9th Int. Conf. Supercomputing*, ser. ICS '95, 1995, pp. 137–146.

[3] 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*, vol. 45, pp. 105–153, 1997.

[4] C. Ding and K. Kennedy, "Improving cache performance in dynamic applications through data and computation reorganization at run time," in *Proc. ACM SIGPLAN 1999 Conf. Programming Language Design and Implementation*, ser. PLDI '99, 1999, pp. 229–241.

[5] N. Mitchell, L. Carter, and J. Ferrante, "Localizing non-affine array references," in *Proc. 1999 Int. Conf. Parallel Architectures and Compilation Techniques*, ser. PACT '99, 1999, pp. 192–.

[6] J. Mellor-Crummey, D. Whalley, and K. Kennedy, "Improving memory hierarchy performance for irregular applications using data and computation reorderings," *Int. J. Parallel Program.*, vol. 29, no. 3, pp. 217–247, Jun. 2001.

[7] H. Han and C.-W. Tseng, "Exploiting locality for irregular scientific codes," *IEEE Trans. Parallel Distrib. Syst.*, vol. 17, no. 7, pp. 606–618, Jul. 2006.

[8] B. Wu, Z. Zhao, E. Z. Zhang, Y. Jiang, and X. Shen, "Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on gpu," in *Proc. 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '13, 2013, pp. 57–68.

[9] A. Basumallik and R. Eigenmann, "Optimizing irregular shared-memory applications for distributed-memory systems," in *Proc. Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '06, 2006, pp. 119–128.

[10] J.-H. Byun, R. Lin, K. A. Yelick, and J. Demmel, "Autotuning sparse matrix-vector multiplication for multicore," UCB/EECS-2012-215, Tech. Rep., 2012.

[11] T. Gkountouvas, V. Karakasis, K. Kourtis, G. Goumas, and N. Koziris, "Improving the performance of the symmetric sparse matrix-vector multiplication in multicore," in *IPDPS*, 2013.

[12] E. Saule, K. Kaya, and Ü. V. Çatalyürek, "Performance evaluation of sparse matrix multiplication kernels on Intel Xeon Phi," in *Proc of the 10th Int. Conf. Parallel Processing and Applied Mathematics (PPAM)*, Sep. 2013, p. 10.

[13] A. Buttari, V. Eijkhout, J. Langou, and S. Filippone, "Performance optimization and modeling of blocked sparse kernels," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 4, pp. 467–484, Nov. 2007.

[14] R. Vuduc, J. W. Demmel, K. A. Yelick, S. Kamil, R. Nishtala, and B. Lee, "Performance optimizations and bounds for sparse matrix-vector multiply," in *Proc. ACM/IEEE Conf. Supercomputing*, 2002, pp. 1–35.

[15] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on CUDA," NVIDIA Corporation, NVIDIA Technical Report NVR-2008-004, Dec. 2008.

[16] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd. 2003.

[17] J. Dongarra and M. A. Heroux, "Toward a new metric for ranking high performance computing systems," Sandia National Laboratories, Tech. Rep. 4744, 2013.

[18] J. Park, M. Smelyanskiy, K. Vaidyanathan, A. Heinecke, D. D. Kalamkar, X. Liu, M. M. A. Patwary, Y. Lu, and P. Dubey, "Efficient shared-memory implementation of high-performance conjugate gradient benchmark and its application to unstructured matrices," in *Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14, 2014, pp. 945–955.

[19] C.-Q. Zhu and P.-C. Yew, "A scheme to enforce data dependence on large multiprocessor systems," *IEEE Trans. Softw. Eng.*, vol. 13, no. 6, pp. 726–739, Jun. 1987.

[20] S.-T. Leung and J. Zahorjan, "Improving the performance of runtime parallelization," in *Proc. Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP '93, 1993, pp. 83–91.

[21] X. Zhuang, A. E. Eichenberger, Y. Luo, K. O'Brien, and K. O'Brien, "Exploiting parallelism with dependence-aware scheduling," in *Proc. 2009 18th Int. Conf. Parallel Architectures and Compilation Techniques*, ser. PACT '09, 2009, pp. 193–202.

[22] M. M. Strout, L. Carter, J. Ferrante, J. Freeman, and B. Kreaseck, "Combining performance aspects of irregular Gauss-Seidel via sparse tiling," in *Proc. 15th Workshop Languages and Compilers for Parallel Computing (LCPC)*, 2002.

[23] M. Naumov, "Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the gpu," NVIDIA Corporation, Tech. Rep. 001, 2011.

[24] R. Govindarajan and J. Anantpur, "Runtime dependence computation and execution of loops on heterogeneous systems," in *Proc. 2013 IEEE/ACM Int. Symposium on Code Generation and Optimization (CGO)*, ser. CGO '13, 2013, pp. 1–10.

[25] J. Park, M. Smelyanskiy, N. Sundaram, and P. Dubey, "Sparsifying synchronization for high-performance shared-memory sparse triangular solver," in *Int. Supercomputing Conf. (ISC)*, 2014.

[26] D. Kroening and O. Strichman, *Decision Procedures: An Algorithmic Point of View*, 1st ed. 2008.

[27] C. Chen, "Polyhedra scanning revisited," in *Proc. 33rd ACM SIGPLAN Conf. Programming Language Design and Implementation*, ser. PLDI '12, 2012, pp. 499–508.

[28] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott, "The Omega Library interface

guide," University of Maryland at College Park, Technical Report CS-TR-3445, Mar. 1995.

[29] A. Venkat, M. Shantharam, M. Hall, and M. M. Strout, "Non-affine extensions to polyhedral code generation," in *Proc. of Annual IEEE/ACM Int. Symposium on Code Generation and Optimization*, ser. CGO '14, 2014, 185:185–185:194.

[30] Y. Lin and D. Padua, "Compiler analysis of irregular memory accesses," in *Proc. ACM SIGPLAN 2000 Conf. Programming Language Design and Implementation*, ser. PLDI '00, 2000, pp. 157–168.

[31] C. Chen, J. Chame, and M. Hall, "Chill: A framework for composing high-level loop transformations," University of Southern California, Technical Report 08-897, Jun. 2008.

[32] M. M. Strout, G. George, and C. Olschanowsky, "Set and relation manipulation for the sparse polyhedral framework," in *Proc. 25th Int. Workshop Languages and Compilers for Parallel Computing (LCPC)*, 2012.

[33] J. Park, https://github.com/jspark1105/SpMP.

[34] http://www.rosecompiler.org.

[35] P. Basu, A. Venkat, M. W. Hall, S. W. Williams, B. van Straalen, and L. Oliker, "Compiler generation and autotuning of communication-avoiding operators for geometric multigrid," in *20th Annual Int. Conf. High Performance Computing, HiPC 2013*, 2013, pp. 452–461.

[36] Y. Saad, *Iterative Methods for Sparse Linear Systems*. 2003.

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

[38] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009.

[39] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.

[40] ——, "Stream: Sustainable memory bandwidth in high performance computers," University of Virginia, Tech. Rep., 1991-2007.

[41] L. Rauchwerger, N. Amato, and D. Padua, "A scalable method for run-time loop parallelization," English, *Int. Journal of Parallel Programming*, vol. 23, no. 6, pp. 537–576, 1995.

[42] K. McKinley, "Dependence analysis of arrays subscripted by index arrays," Rice University, Tech. Rep. TR91187, 1991.

[43] D. K. Chen, J. Torrellas, and P. C. Yew, "An efficient algorithm for the run-time parallelization of doacross loops," in *Proc. 1994 ACM/IEEE Conf. Supercomputing*, ser. Supercomputing '94, 1994, pp. 518–527.

[44] 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 *Proc. Int. Conf. High Performance Computing, Networking, Storage and Analysis*, ser. SC '12, 2012, 72:1–72:11.

[45] W. Vanroose, P. Ghysels, K. Meerbergen, and D. Roose, "Hiding global communication latency and increasing the arithmetic intensity in extreme-scale Krylov solvers," Aug. 2013.

[46] S. Campanoni, T. Jones, G. Holloway, V. J. Reddi, G.-Y. Wei, and D. Brooks, "Helix: Automatic parallelization of irregular programs for chip multiprocessing," in *Proc. Tenth Int. Symposium on Code Generation and Optimization*, ser. CGO '12, 2012, pp. 84–93.

[47] S. P. Midkiff and D. A. Padua, "Compiler algorithms for synchronization," *IEEE Trans. Comput.*, vol. 36, no. 12, pp. 1485–1495, Dec. 1987.

[48] B. Pugh and D. Wonnacott, "Nonlinear array dependence analysis," Dept. of Computer Science, Univ. of Maryland, Tech. Rep. CS-TR-3372, 1994.

[49] C. E. Oancea and L. Rauchwerger, "Logical inference techniques for loop parallelization," in *Proc. 33rd ACM SIGPLAN Conf. Programming Language Design and Implementation*, ser. PLDI '12, 2012, pp. 509–520.

[50] D. E. Maydan, J. L. Hennessy, and M. S. Lam, "Efficient and exact data dependence analysis," in *Proc. ACM SIGPLAN 1991 Conf. Programming Language Design and Implementation*, ser. PLDI '91, 1991, pp. 1–14.

[51] W. Pugh and D. Wonnacott, "An exact method for analysis of value-based array data dependences," K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama, Eds., ser. Lecture Notes in Computer Science, 1994, pp. 546–566.

[52] J.-F. Collard, D. Barthou, and P. Feautrier, "Fuzzy array dataflow analysis," *SIGPLAN Not.*, vol. 30, no. 8, pp. 92–101, Aug. 1995.

[53] J. H. Saltz, "Aggregation methods for solving sparse triangular systems on multiprocessors," *SIAM J. Sci. Stat. Comput.*, vol. 11, no. 1, pp. 123–144, Jan. 1990.

[54] E. Rothberg and A. Gupta, "Parallel ICCG on a hierarchical memory multiprocessor - addressing the triangular solve bottleneck," *Parallel Computing*, vol. 18, no. 7, pp. 719–741, 1992.

[55] *Intel math kernel library inspector-executor sparse blas routines*, https://software.intel.com/en-us/articles/intel-math-kernel-library-inspector-executor-sparse-blas-routines.

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

[57] A. Venkat, M. Hall, and M. Strout, "Loop and data transformations for sparse matrix code," in *Proc. 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2015, 2015, pp. 521–532.