Sparse Computation Data Dependence Simplification for Efficient Compiler-Generated Inspectors

Mahdi Soltan Mohammadi
University of Arizona
Tucson, USA
kingmahdi@cs.arizona.edu

Tomofumi Yuki
Inria, Univ Rennes, CNRS, IRISA
Rennes, France
tomofumi.yuki@inria.fr

Kazem Cheshmi
University of Toronto
Toronto, Canada
kazem@cs.toronto.edu

Eddie C. Davis
Boise State University
Boise, USA
eddiedavis@u.boisestate.edu

Mary Hall
University of Utah
Salt Lake City, USA
mhall@cs.utah.edu

Maryam Mehri Dehnavi
University of Toronto
Toronto, Canada
mmehride@cs.toronto.edu

Payal Nandy
University of Utah
Salt Lake City, USA
payalgn@cs.utah.edu

Catherine Olschanowsky
Boise State University
Boise, USA
catherineolschan@boisestate.edu

Anand Venkat
Intel Corporation
Santa Clara, USA
anand.venkat@intel.com

Michelle Mills Strout
University of Arizona
Tucson, USA
mstrout@cs.arizona.edu

Abstract

This paper presents a combined compile-time and runtime loop-carried dependence analysis of sparse matrix codes and evaluates its performance in the context of wavefront parallelism. Sparse computations incorporate indirect memory accesses such as x[col[j]] whose memory locations cannot be determined until runtime. The key contributions of this paper are two compile-time techniques for significantly reducing the overhead of runtime dependence testing: (1) identifying new equality constraints that result in more efficient runtime inspectors, and (2) identifying subset relations between dependence constraints such that one dependence test subsumes another one that is therefore eliminated. New equality constraints discovery is enabled by taking advantage of domain-specific knowledge about index arrays, such as col[j]. These simplifications lead to automatically-generated inspectors that make it practical to parallelize such computations. We analyze our simplification methods for a collection of seven sparse computations. The evaluation shows our methods reduce the complexity of the runtime inspectors significantly. Experimental results for a collection of five large matrices show parallel speedups ranging from 2x to more than 8x running on a 8-core CPU.

PLDI '19, June 22–26, 2019, Phoenix, AZ, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6712-7/19/06…$15.00
https://doi.org/10.1145/3314221.3314646

CCS Concepts • **Theory of computation → Program analysis**; • **Software and its engineering → Source code generation**.

Keywords data dependence simplification, dependence analysis, sparse matrices, inspector-executor strategies, Presburger arithmetic with uninterpreted functions, SMT solvers

ACM Reference Format:
Mahdi Soltan Mohammadi, Tomofumi Yuki, Kazem Cheshmi, Eddie C. Davis, Mary Hall, Maryam Mehri Dehnavi, Payal Nandy, Catherine Olschanowsky, Anand Venkat, and Michelle Mills Strout. 2019. Sparse Computation Data Dependence Simplification for Efficient Compiler-Generated Inspectors. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3314221.3314646

1 Introduction

Sparse matrix computations occur in many codes such as graph analysis, partial differential equations solvers, and molecular dynamics simulations. Sparse matrix representations save on storage and computation by only storing the nonzero values of the matrix. Figure 1 illustrates an example of how a forward solve computation is implemented using a common sparse matrix format, compressed sparse row (CSR).

```
// Forward Solve CSR
for(i=0; i<N; i++) {   // loop over rows
   tmp = f[i];
   for(k=rowptr[i]; k<rowptr[i+1]-1; k++){
S1:   tmp -= val[k]*u[col[k]];
   }
S2:u[i] = tmp / val[rowptr[i+1]-1];
}
```

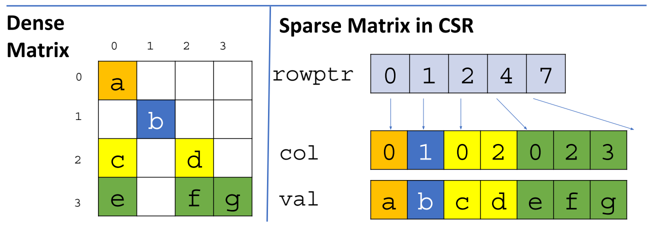

Figure 1. Compressed Sparse Row (CSR) sparse matrix format. The `val` array stores the nonzeros by packing each row in contiguous locations. The `rowptr` array points to the start of each row in the `val` array. The `col` array is parallel to `val` and maps each nonzero to the corresponding column.

CSR organizes nonzeros by row, and for each nonzero the corresponding column is stored. Sparse representations are crucial for many computations to fit into memory and execute in a reasonable time, given that fewer than 1% of values may be nonzero.

Unfortunately, the advantage sparse formats have on saving storage comes with the cost of complicating program analysis aimed at finding parallelism opportunities. Static compile-time approaches resort to conservative approximation [6, 37]. Some approaches use runtime analysis to complement compile-time analysis and discover more loops that are fully parallel [36, 45, 54]. Runtime dependence analysis is needed because of indirect accesses such as the `u[col[k]]@S1`[1] in Figure 1. The elements of the `u` array accessed by this statement are not known until runtime when the values of the index array, `col`, are available.

One advantage of sparse computations is they often exhibit partial (doacross) parallelism in loops that are fully sequential in dense codes. A *fully sequential loop* is one where each iteration depends on the previous. A *fully parallel loop* does not have any loop-carried dependences. A *partially parallel loop* occurs when any particular iteration only depends on a subset of earlier iterations. A number of previous sparse kernel implementations incorporate an inspector code that derives a graph representation of these sparse dependences, and use this graph to discover and exploit wavefront parallelism [8, 12, 39, 40, 47, 50, 53, 56, 59, 63, 66, 68]. As an example, consider that we use the data matrix in Figure 1 as an input to the forward solve code in the same figure. The

[1]Throughout the paper, we refer to an array access `A` at statement `S` as `A@S` for brevity.

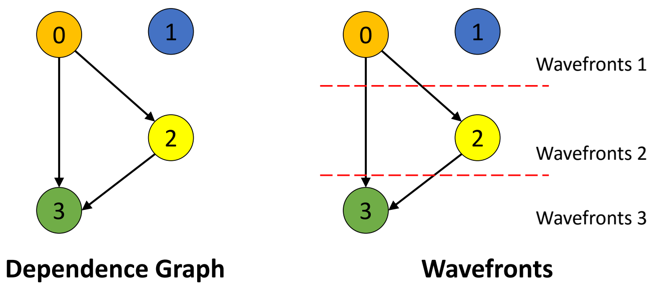

Figure 2. Dependence graph for forward solve for sparse matrix in Figure 1.

dependence graph that the runtime inspector would generate is shown in Figure 2 on the left, with waves (often called level sets) created based on that graph shown on the right.

Developers of numerical libraries typically write such inspectors by hand. Given that inspectors introduce runtime overhead, it is crucial that the inspectors have a low computational complexity (as efficient as possible, and at least no slower than the original computation). Therefore, the developers often exploit domain-specific information about properties of the index arrays to derive efficient inspectors, and in some cases over-approximate dependences to tradeoff accuracy for reduced inspection time.

Venkat et al. [63] described an automated approach to generating efficient wavefront inspectors by incorporating domain-specific information about index arrays (specifically, monotonicity), and using polyhedra scanning to generate the inspector code; this approach used overapproximation of dependences, and was applied to two benchmarks. Mohammadi et al. [34] explored how additional domain-specific properties of index arrays could eliminate more dependences, and in many cases all loop-carried dependences could be proven unsatisfiable so that some loops can be fully parallelized.

In this paper, we present algorithms for simplifying sparse dependences in partially parallel loops, thus enabling the automatic generation of efficient inspectors for creating the dependence graph[2]. We simplify data dependences by using domain-specific properties of index arrays to discover new equalities, simplifying runtime inspectors, and detecting dependence subsets that would be subsumed by inspectors for their supersets. We show the general applicability of the presented techniques by evaluating their impact on the wavefront parallelization for seven popular sparse kernels.

This paper makes the following contributions:

- An approach to dependence constraint simplification that reduces the complexity of runtime dependence testing in an inspector. The key contributions in this paper are (1) identifying equality constraints; and, (2)

[2] See Section 9.1 for details about the relationship of this work to the work in Venkat et al. [63] and Mohammadi et al. [34].

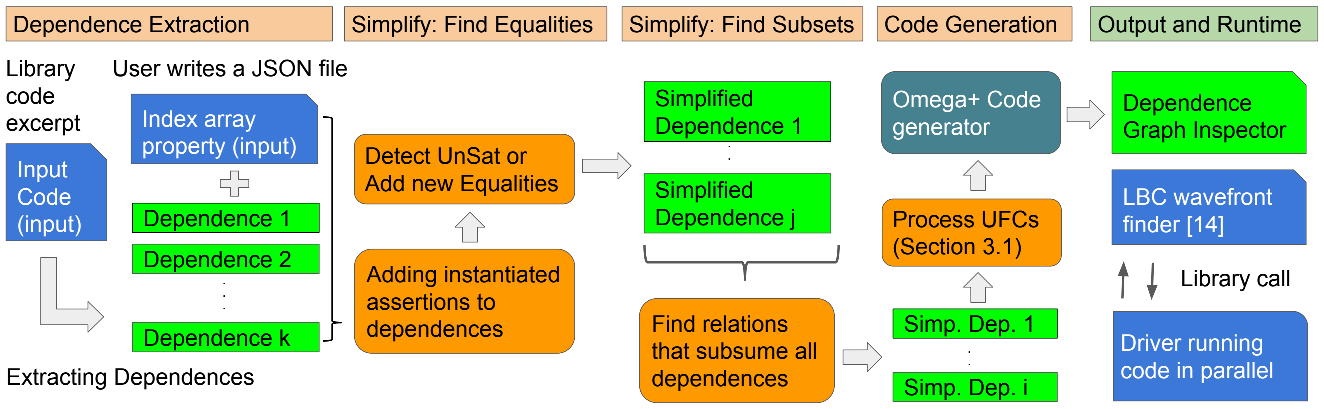

Figure 3. The overview of our approach to eliminate or simplify dependences from sparse computations utilizing domain-specific information. The blue boxes show inputs or hand written drivers that include input computation inside a C file, user-defined, domain-specific knowledge about index arrays inside a JSON file, driver for calling inspector and executor to run the input code in parallel. The green color indicates generated data, including all the dependences getting passed around, and inspector code. And, the orange color shows codes that we have implemented for our methods. A driver carries out the steps from reading input library code and user defined assertions to generating inspectors that build the dependence graph. As we simplify the dependences, their number and complexity potentially decrease, hence $k \geq j \geq i$. LBC stands for load-balanced level coarsening.

identifying subset relations between dependence constraints such that one dependence test subsumes another that can be therefore eliminated.

- An implementation of these techniques that leverages the strengths of state-of-the-art polyhedral compiler technology, but extends it for analysis and code generation of sparse computations.
- A demonstration that these simplifications lead to automatically-generated inspectors that make it practical to parallelize representative sparse computations. For a collection of seven sparse matrix computations and five large matrices consisting of millions of nonzeros, we show parallel speedups ranging from 2× to more than 8× running on 8 cores.

2 Compile-Time Dependence Analysis

Figure 3 shows an overview of our approach for automating wavefront parallelization through the simplification of sparse data dependences. First, we extract the data dependence relations for the outermost loop of an input kernel. At compile time, we use index array properties to determine which data dependences are unsatisfiable. Additionally, we simplify any remaining loop-carried dependences and generate inspector code. At runtime, the generated inspector creates a dependence graph and then sends that dependence graph to a library routine that finds parallel wavefronts. The executor code loops over wavefronts sequentially and then executes all of the iterations in a wavefront in parallel.

This section reviews the compile-time component of the data dependence analysis.

2.1 Constraint-Based Data Dependence Analysis

Data dependence analysis of a loop nest is a common analysis that is used in different applications, such as automatic parallelization [11] and data race detection [4]. A loop-carried dependence can be represented by constraints. The constraints encode when two iterations of a loop access the same memory location, where at least one of them is a write access, the loop has a loop-carried dependence [45, 46].

$$Dep : (\exists \vec{I}, \vec{I'})(\vec{I} \prec \vec{I'} \wedge F(\vec{I}) = G(\vec{I'}) \wedge Bounds(\vec{I}) \wedge Bounds(\vec{I'})),$$

where $\vec{I}$ and $\vec{I'}$ are iterations of the same loop, with $\prec$ denoting lexicographical ordering, F and G are index expressions to the same array, and $Bounds(\vec{I})$ expands to the loop nest bounds for the $\vec{I}$ iteration vector. We use the term *dependence relation* interchangeably with dependence constraints, the dependence being viewed as a relation between $\vec{I}$ and $\vec{I'}$.

As an example, consider the sparse forward solve kernel in Figure 1. Looking for wavefront parallelism in the outermost loop, there are two pairs of accesses on array `u` in `S1` and `S2` that can potentially cause loop-carried dependences. The first pair includes two identical write accesses `u[i]@S2`. Those accesses are guaranteed to be disjoint for different iterations of the `i` loop, therefore, there are no dependences based on them. Dependences can also come from `u[col[k]]@S1` (read), and `u[i]@S2` (write). The flow dependence relation for this read-write pair is:

$$\{[i] \rightarrow [i'] : \exists k' : i \neq i' \wedge i = col(k')$$
$$\wedge\ 0 \leq i, i' < N \ \wedge rowptr(i') \leq k' < rowptr(i' + 1)\}.$$

Table 1. Index array properties, and their universally quantified assertion form, discussed by Mohammadi et al. [34]. The symbol $\bowtie$ can be $<$, $\le$, $>$, and $\ge$ in the Monotonicity row and $<$ or $>$ in the Strict Monotonicity row.

Array property	Definition
Domain & Range	$(\forall x_1 : D_l \le x_1 \le D_u \Leftrightarrow R_l \le f(x_1) \le R_u)$.
Injectivity	$(\forall x_1, x_2 : f(x_1) \ne f(x_2) \Leftrightarrow x_1 \ne x_2)$.
Monotonicity	$(\forall x_1, x_2 : x_1 \bowtie x_2 \Rightarrow f(x_1) \bowtie f(x_2))$.
Strict Monotonicity	$(\forall x_1, x_2 : x_1 \bowtie x_2 \Leftrightarrow f(x_1) \bowtie f(x_2))$.
Periodic Monotonicity	$(\forall x_1, x_2, x_3 : (x_2 \bowtie x_3 \wedge g(x_1) \le x_2, x_3 < g(x_1 + 1) \Rightarrow f(x_1) \bowtie f(x_2))$.
CoMonotonicity	$(\forall x_1 = x_2 \Rightarrow f(x_1) \le g(x_2))$.
Triangularity	$(\forall x_1, x_2 : f(x_1) < x_2 \Rightarrow x_1 < g(x_2))$.

We cannot determine the pairs of i and i' that satisfy the constraints above at compile-time, since the values of the indirection arrays, `col` and `rowptr`, are not available. This is why we need runtime inspectors to find parallelism.

2.2 Disproving Dependences at Compile Time

Any dependence that can be shown to be unsatisfiable at compile time, does not need runtime analysis. There is a body of work that uses information about the index arrays, such as monotonicity, to find parallel loops by disproving data dependences that are satisfiable without domain knowledge [31, 33, 34, 36, 54, 63].

Recently, in Mohammadi et al. [34], we presented additional properties beyond commonly-used ones that can be found in sparse matrices for numerical computations. We showed that the properties can be expressed as universally quantified assertions and passed to SMT solvers along with the original dependence constraints to test for satisfiability.

Those index array properties are depicted in Table 1. Monotonicity of index arrays that compress indices of a matrix dimension is common. Triangularity happens when a numerical kernel operates only on sparse matrices that are triangular. Periodic monotonicity happens in some index arrays that are not monotonic overall, but segments of the index arrays are sorted. Correlated monotonicity happens when two index arrays index into separate but consecutive parts of same data array.

SMT solvers typically handle universal quantifiers by performing quantifier instantiation, where the main idea is that the formula can be proven unsatisfiable if some instances of the universally quantified assertions lead to contradiction. If all dependence relations are proven unsatisfiable when combined with the assertions, then the compiler can conclude that there are no loop-carried dependence at compile time.

As an example of how domain information can be used to show dependences are unsatisfiable consider the following constraints from a dependence relation:

$$\{[i] \to [i'] : \exists(m, k') : i < i' \wedge m = k' \wedge\ 0 \le i, i' < n$$
$$\wedge\ rowptr(i-1) \le m < rowptr(i)$$
$$\wedge\ rowptr(i') \le k' < rowptr(i'+1)\}$$

This relation is satisfiable depending on the content of the `rowptr` array, and hence a runtime check is necessary. The main condition is that iterations of m and k' must overlap, which can be expressed as a combination of their bounds: $rowptr(i') < rowptr(i) \wedge rowptr(i'-1) < rowptr(i+1)$.

If we know the `rowptr` is strictly monotonic:

$$(\forall x_1, x_2)(x_1 < x_2 \implies rowptr(x_1) < rowptr(x_2)),$$

then the dependence relation can be proven unsatisfiable. An instance of the assertion when $x_1 = i, x_2 = i'$ gives $rowptr(i) < rowptr(i')$ directly contradicting one of the conditions for a dependence to exist.

See Section 7 for details about how much each index array property alone and in combination with other properties results in finding unsatisfiable data dependences.

3 Runtime Data Dependence Inspector

After compile-time analysis proves as many unsatisfiable dependences as possible, the next step is to generate inspector code to perform runtime analysis on the remaining dependences (see Figure 3). Given a compile-time description of the dependence with uninterpreted functions representing index arrays, code can be generated to inspect the dependences at runtime and create a directed acyclic graph (DAG) structure where wavefronts of parallelization can be discovered.

3.1 Inspector Code Generation

The Omega+ [13] polyhedral code generation tool and IEGenLib [62], a component of the Sparse Polyhedral framework [61], are used in concert to generate the inspector code. IEGenLib supports general transformations on sets that include uninterpreted functions, and Omega+ supports code generation for those sets. Once all of the transformations have been applied and their form simplified using IEGenLib, the sets are translated to a normalized form usable by Omega+.

The Omega+ code generator has a number of syntactic limitations for handling uninterpreted function calls (UFCs). For one, the UFCs can only have loop iterators as parameters. To handle this limitation, we use macro functions that abstract away any complex expression parameters. For instance, an index array access `idx[i+1]` is represented by a macro `IDX_1(i)`. Additionally, Omega+ requires that parameters to a UFC be a prefix of input or output tuple iterators of the set that code is being generated for. Once again we use macro definition to solve this limitation. For instance,

```
  for(i = 0; i < n; i++) {
S1: val[colPtr[i]] = sqrt(val[colPtr[i]]);

    for (m = colPtr[i] + 1; m < colPtr[i+1]; m++)
S2:   val[m] = val[m] / val[colPtr[i]];

    for (m = colPtr[i] + 1; m < colPtr[i+1]; m++)
      for (k=colPtr[rowIdx[m]]; k<colPtr[rowIdx[m]+1]; k++)
        for (l = m; l < colPtr[i+1] ; l++)
          if(rowIdx[l]==rowIdx[k] && rowIdx[l+1]<=rowIdx[k])
S3:         val[k] -= val[m]*val[l];
  }
```

Figure 4. Incomplete Cholesky code from SparseLib++ [43]. Some variable names have been changed.

for the following dependence: $\{[i, j, k, l] : idx(k) < n \wedge \ldots\}$, the $idx(k)$ UF call becomes `IDX__(i,j,k)`.

3.2 Problem: Expensive Inspectors

As described in Section 2.2, the use of domain information can disprove many potential dependences at compile time. However, most of the benchmarks (all except SpMV) have remaining dependence relations necessitating runtime checks.

It is possible to employ wavefront parallelization by generating inspector code that tests all remaining dependences. However, the overhead may be prohibitively high without further optimization. In particular, three of our benchmarks would have inspectors with higher algorithmic complexity than the original computation. This makes it difficult to amortize the cost of inspection by gains from parallelization.

As an example, consider the Incomplete Cholesky kernel in Figure 4. Since `m, k, and l` loops are traversing over nonzeros when the i-loop traverses over all columns, the algorithmic complexity of the kernel is of $O(n \times (nnz/n) \times (nnz/n) \times (nnz/n)) = O(nnz \times (nnz/n)^2)$, where n is number of columns, and nnz is number of nonzeros. However, the remaining dependences for this code would require runtime inspectors with higher complexity: $O(nnz^2 \times (nnz/n)^2)$. Such a high complexity inspector cannot be amortized. This problem motivated the development of two dependence simplification methods that can result in more efficient inspectors. We describe these methods in the next two sections.

4 Simplify Dependences: Find Equalities

In this section, we discuss a dependence simplification method based on discovering additional equalities after utilizing domain information. These equalities may reduce the complexity of runtime inspectors by revealing that a dependence relation is a lower dimensional shape embedded in a higher dimensional space (e.g., $\{[i, j] : i = j\}$ is a line in a 2D space).

4.1 Usefulness of New Equality Constraints

The new equalities may be exposed by considering instances of the universally quantified assertions that represent index array properties. For instance, consider the following dependence relation, which is a simplified version of those found in some of our benchmarks:

```
for(i = 0; i <n ; i++)
 for(i'= i+1; i'<n; i'++)
  if(f(i')<=f(i)&&g(i)<=i')
   // Add i -> i'
```

(a) Inspector with the original dependence constraints.

```
for(i = 0; i <n ; i++)
  i' = g(i);
    if(f(i') <= f(i) )
    // Add i -> i'
```

(b) Inspector with an additional equality: $i' = g(i')$.

Figure 5. Inspector pseudo-code for example dependence constraints in Section 4.1, before and after utilizing index array properties to add new equalities.

$$(i < i') \wedge f(i') \le f(g(i)) \wedge g(i) \le i' \wedge (0 \le i, i' < n).$$

This is a satisfiable dependence that needs a runtime inspector with a complexity of $O(n^2)$ to traverse the space of values for i and i'; the inspector is shown in Figure 5a. Assume that we also know the following universally quantified assertion about the uninterpreted function f (strict monotonicity):

$$(\forall x_1, x_2), (x_1 < x_2) \implies (f(x_1) < f(x_2)).$$

We may also use its contrapositive:

$$(\forall x_1, x_2), (f(x_1) \ge f(x_2)) \implies (x_1 \ge x_2)$$

where its instance with $x_1 = g(i), x_2 = i'$ gives

$$f(i') \le f(g(i)) \implies i' \le g(i).$$

Since the antecedent of the instance is always true in the example relation, the consequent may be directly added. This gives $i' \le g(i) \wedge g(i) \le i' \equiv i' = g(i)$, a new equality.

Using this equality, the inspector may iterate over i and obtain values of i' from $g(i)$ without explicitly iterating over i'. The inspector after this optimization is depicted in Figure 5b. Now, utilizing the discovered equality, the runtime inspection would have complexity of only $O(n)$.

4.2 Algorithm for Discovering Equalities

If the goal is to detect unsatisfiable dependences, specifying the domain information as universally quantified assertions and using an SMT solver as presented in Mohammadi et al. [34] are sufficient. SMT solvers are specialized for solving satisfiability problems expressed as a combination of background theories. They instantiate the assertions utilizing heuristics to quickly determine unsatisfiability of a set of constraints. However, SMT solvers do not provide a way to output equality relationships that may have been inferred while answering the satisfiability question.

In contrast, libraries for manipulating integer sets, such as ISL [64], are specialized for analyzing and manipulating integer sets, providing additional intermediate constraints.

Hence, we implement a procedure to add instantiated assertions to a dependence relation and then use libraries to compute all the equalities.

Our procedure takes general assertions of the form:

$$\forall \vec{x},\ \varphi_I(\vec{x}) \implies \varphi_V(\vec{x})$$

Where $\vec{x}$ denotes vector of quantified variables, $\varphi_I(\vec{x})$ denotes antecedent of the assertion, and $\varphi_V(\vec{x})$ denotes the consequent of the assertion. Our procedure to instantiate quantified assertions is defined in the following.

Definition 1 (E) We define E to be the set of expressions used as arguments to all uninterpreted function calls in the original set of constraints. We use this set to instantiate quantified assertions.

Definition 2 (*$UNSAT_\psi$*)

1. Turn the universally quantified predicates into quantifier-free predicates using inference rule:
$$\textsc{forall}\frac{\psi[\forall \vec{x},\ \varphi_I(\vec{x}) \implies \varphi_V(\vec{x})]}{\psi[\bigwedge_{\vec{x}\in E^n}(\varphi_I(\vec{x}) \implies \varphi_V(\vec{x}))]}$$
where E^n is the set of vectors of size $n = |\vec{x}|$ produced as a Cartesian product of E.
2. Solve the quantifier-free formula ψ output of the previous step with an SMT solver that decide union of quantifier-free theories of uninterpreted functions with equality and Presburger Arithmetic.

Completeness: The instantiation procedure is similar to the one for a decidable fragment of theory with arrays [10]. For the decidable fragment, a finite number of instantiations give an equisatisfiable formula without universal quantifiers. Unfortunately, some of the assertions we use do not fit any known decidable fragment, and thus we do not have completeness. Please see Section 9.2 for discussions about techniques for quantifier elimination.

Correctness: Although the above procedure is incomplete, we do have soundness. This means if a dependence is determined unsatisfiable, it in fact is not a dependence. However, if a dependence is determined satisfiable at compile time, it could be that at runtime the actual values of index arrays lead to the dependence not being satisfiable. Since our procedure is conservatively correct, it is sound.

To show that the decidability procedure $UNSAT_\psi$ is sound, we need to show that if the original formula ψ is satisfiable, then so is the unquantified formula ψ',

$$\psi \in SAT \implies \psi' \in SAT.$$

This is equivalent to

$$\psi' \notin SAT \implies \psi \notin SAT.$$

Since universal quantification is being replaced with specific expression instantiations to create ψ', ψ' is a potentially weaker set of constraints than ψ. This means that ψ' is a conservative approximation of ψ. As such, if ψ' is not satisfiable, then ψ is not satisfiable.

```
1    for(i = 0; i < n; i++) {
2  S1: val[colPtr[i]] = sqrt(val[colPtr[i]]);
3
4      for (m = colPtr[i] + 1; m < colPtr[i+1]; m++)
5  S2:   val[m] = val[m] / val[colPtr[i]];
6
7      for (m = colPtr[i] + 1; m < colPtr[i+1]; m++)
8        for (k=colPtr[rowIdx[m]]; k<colPtr[rowIdx[m]+1]; k++)
9          for (l = m; l < colPtr[i+1] ; l++)
10           if(rowIdx[l]==rowIdx[k] && rowIdx[l+1]<=rowIdx[k])
11 S3:         val[k] -= val[m]*val[l];
12   }
```

Figure 6. Incomplete Cholesky code from SparseLib++ [43]. Same as Figure 4, repeated for reader's convenience.

5 Simplify Dependences: Find Subsets

In addition to discovering new equalities, another way to simplify runtime dependence relations is to remove tests for dependence relations that can be identified as a subset of another. This can result in faster runtime inspection when a dependence relation can be shown to be a subset of a less complex relation. Such analysis is sometimes performed by expert programmers to optimize manually written inspectors. In this section, we propose an approach to automate this process. The key challenge is to determine subset relations between two dependence relations defined with inequality constraints involving uninterpreted functions.

5.1 Intuition of the Simplification

As an intuitive example, consider the Incomplete Cholesky code shown in Figure 6. One of the dependence relations is between the write `val[k]@S3` and the read `val[m]@S3`. This test is redundant with the test between the write `val[k]@S3` and the read `val[m]@S2`, because, an iteration of the `i` loop that reads from `val[m]` in `S3` is guaranteed to access the same memory locations while executing the loop surrounding S2 as the loop bounds in lines 4 and 7 are the same. Thus, the more expensive check between accesses in `S3` can be removed. The check between two accesses in `S3` is more expensive because the dependence relation has two additional dimensions, corresponding to the `k` and `l` loops of the second instance of `S3`.

5.2 Subset Relations in the Context of Sparse Codes

It is important to clarify what it means for a dependence relation to be a subset of another in the context of sparse matrix codes, when dependence relations involve uninterpreted functions. If the compiler assumes that uninterpreted function calls (UFCs) can evaluate to any value, most of the dependence relations involving UFCs would be considered satisfiable. At runtime, the actual values of UFCs are known, making it possible to perform exact dependence analysis. For each dependence relation, there exist a set of constraints on the values of the indirection arrays, or the concrete interpretations of UFs, that characterize when the dependence manifests. These constraints on the concrete interpretations of UFs are central to the subset relationships between dependence relations. If the set of concrete interpretations for a dependence to manifest is subsumed by those for another, then it is safe to remove the test.

Subset Detection Algorithm: Recall that a dependence test for loop-carried dependence of the outermost loop takes the following form:

$$\{[i] \rightarrow [i'] : \exists \vec{u}, \vec{v} : i \neq i' \wedge \ldots\}$$

where $\vec{u}$ and $\vec{v}$ represent the indices for inner loops of two statements involved in the test. Given two dependence relations R_1 and R_2, we test for subset relation as follows:

1. Apply Ackerman's reduction to R_1 and R_2 to obtain equisatisfiable relations R_1' and R_2' without UFCs [58]. Application of Ackerman's reduction to a relation R proceeds as follows. For each pair of UFCs sharing the same UF, $f(x)$ and $f(y)$, a constraint encoding functional consistency, $A_i = (x = y \Rightarrow f(x) = f(y))$, is created. The conjunction of all such constraints becomes the antecedent of an implication where the original relation is its consequent: $M = [(A_1 \wedge A_2 \ldots) \Rightarrow R]$. Then, all UFCs in M are replaced with fresh variables. to obtain a relation R' that is UFC free.
2. Project out all the existentially quantified iterators ($\vec{u}$ and $\vec{v}$ in the above) from R_1' and R_2' to obtain R_1^* and R_2^*. These relations characterize when the dependence manifests, including constraints on the concrete interpretation of UFCs (now represented as fresh variables, or parameters when viewed as integer relations).
3. The subset relation between R_1^* and R_2^* is tested with a library for polyhedral relations.

Correctness: The latter two steps of the above are standard operations over polyhedral relations, and do not concern correctness. Ackerman's reduction gives relations without UFCs that are *effectively equivalent* to the original relation. This is a property of Ackerman's reduction that *completely replaces* UFCs with fresh variables. This means that the polyhedral subset relationship holds for the constraints involving uninterpreted functions, because the introduced F_x parameters take on any value that the corresponding UFC can hold. Let us consider a relation, $f(x) \geq 0$, and its corresponding relation after reduction, $F_x \geq 0$. These relations are technically equisatisfiable, but not logically equivalent, since models such as $f(x) = 0; F_x = -1$ and $f(x) = -1; F_x = 0$ are valid for one of the relations, but not for the other. Observe that adding $f(x) = F_x$ does not change satisfiability of the two relations in isolation, since the UFC and the replacement are never used simultaneously in a relation. Thus, the set of values that a UFC can take is exactly the same as its replacement, making them effectively equivalent.

5.3 Example

We illustrate the subset detection algorithm with an example, using Incomplete Cholesky in Figure 6. We take two dependence relations, R_1 between `val[k]@S3` and `val[m]@S2`, and R_2 between `val[k]@S3` and `val[l]@S3`:

$$\begin{gathered} R_1 = \{[i, m, k, l] \rightarrow [i', m'] : k = m' \wedge 0 \leq i < i' < n \\ \wedge\ col(i) + 1 \leq m \leq l < col(i+1) \wedge row(l+1) \leq row(k) \\ \wedge\ col(row(m)) \leq k < col(row(m)+1) \wedge\ row(l) = row(k) \\ \wedge\ col(i') + 1 \leq m' < col(i'+1)\} \end{gathered}$$

$$\begin{gathered} R_2 = \{[i, m, k, l] \rightarrow [i', m', k', l'] : k = l' \wedge 0 \leq i < i' < n \\ \wedge\ col(i) + 1 \leq m \leq l < col(i+1) \wedge row(l+1) \leq row(k) \\ \wedge\ col(row(m)) \leq k < col(row(m)+1) \wedge\ row(l) = row(k) \\ \wedge\ col(i') + 1 \leq m' \leq l' < col(i'+1) \wedge row(l'+1) \leq row(k') \\ \wedge\ col(row(m')) \leq k' < col(row(m')+1) \\ \wedge\ row(l') = row(k')\} \end{gathered}$$

where `colPtr` and `rowIdx` are represented as UFs *col* and *row*, respectively. Computing the constraints on UFCs gives:

$$\begin{gathered} R_1^* = \{0 \leq i < i' < N \wedge\ col(i) + 1 < col(i+1) \\ \wedge col(i') + 1 < col(i'+1) \wedge\ col(row(m)) < col(row(m+1)) \\ \wedge\ row(k) = row(l) \wedge row(l+1) \leq row(l) \\ \wedge\ col(i') + 1 < col(row(m+1)) \wedge\ col(row(m)) < col(i'+1)\} \end{gathered}$$

for R_1 and the following:

$$\begin{gathered} R_2^* = \{0 \leq i < i' < N \wedge\ col(i) + 1 < col(i+1) \\ \wedge col(i') + 1 < col(i'+1) \wedge\ col(row(m)) < col(row(m+1)) \\ \wedge\ \mathbf{col(row(m')) < col(row(m'+1))} \\ \wedge\ row(k) = row(l) \wedge row(l+1) \leq row(l) \\ \wedge\ \mathbf{row(k') = row(l') \wedge row(l'+1) \leq row(l')} \\ \wedge\ col(i') + 1 < col(row(m+1)) \wedge\ col(row(m)) < col(i'+1)\} \end{gathered}$$

for R_2 where the differences are highlighted in bold. These sets are computed with UFCs replaced by parameters, but we show the original UFCs for presentation reasons.

Most of these constraints (all but the last two in both R_1^* and R_2^*) represent the conditions for an iteration to be valid, coming directly from loop bounds or guards. The last two in each relation show the constraints on the interpretation of UFCs across read and write iterations participating in a dependence. These constraints can be seen as a condition for the read accesses in two dependence relations; iterations of `m` for `val[m]@S2`, and iterations of `l` for `val[l]@S3`; to overlap with the iterations of `k` for the write access: `val[k]@S3`.

One can see that R_2^* is a subset of R_1^* since all constraints in R_1^* are present in R_2^*. The subset relation can be tested with libraries for integer sets, since the relations are polyhedral after the reduction to remove uninterpreted function calls.

6 Implementation

We have automated the data dependence analysis simplification, and wavefront parallelization inspector code generation (see Figure 3 for an overview), available at: https://github.com/CompOpt4Apps/Artifact-DataDepSimplify. This section summarizes the software packages the implementation relies on, and some important optimizations to make our implementation scalable.

6.1 Software Description and Overall Flow

In our implementations, the serial computation must be provided to our driver as a C file. Additionally, user-defined domain-specific information about index arrays should accompany the code in a separate JSON file. The accompanying JSON file would also indicate the target loop to parallelize.

We use four packages to implement our approach: IEGenLib library [62], ISL library [64], CHiLL compiler framework [1], and Omega+ codegen included in the CHiLL.

CHiLL is a source-to-source compiler framework for composing and applying high-level loop transformations to improve the performance of nested loops written in C. We use CHiLL to extract the dependence relations from the benchmarks. The CHiLL compiler also includes the Omega+ library, a modified version of Omega [28], which is an integer set manipulation library with limited support for constraints that involve uninterpreted function calls. We have used Omega+'s codegen capability to generate the DAG construction portion of the wavefront inspector code.

ISL is a library for manipulating integer sets and relations that only contain affine constraints. It can act as a constraint solver by testing the emptiness of integer sets. It is also equipped with other operations on integer sets for detecting equalities and testing subset relationships. ISL does not support uninterpreted functions, and thus cannot directly represent the dependence constraints in sparse matrix code.

IEGenLib is a set manipulation library that can manipulate integer sets/relations that contain uninterpreted function symbols. It uses ISL for some of its functionalities. We implemented the detection of unsatisfiable dependences and finding the equalities utilizing the IEGenLib and ISL libraries.

The following briefly describes how our driver, illustrated in Figure 3, generates wavefront parallelization inspectors. First, the driver extracts the dependences using CHiLL, and stores them in IEGenLib data structures. The driver also reads the JSON file with user-defined, domain-specific knowledge about index arrays, and stores them in IEGenLib environment variables. Then, it makes a call to an IEGenLib function to simplify the dependences. IEGenLib instantiates universally quantified assertions using the procedure described in Section 4.2 to prove unsatisfiability and to detect equalities. The uninterpreted functions are removed by replacing each call with a fresh variable, and functional consistency is encoded with additional constraints [30, Chapter 4], before calling ISL to test for satisfiability and to expose equalities. Once the satisfiable, simplified, dependences are obtained, the driver tests each pair of the remaining dependences using IEGenLib for subsets and discards any dependence subsumed by another. The IEGenLib subset detection function implements the procedure described in Section 5.

Finally, the inspectors for the remaining dependences are generated by Omega+. Since, the outermost loop in the *inspectors* that we generate are embarrassingly parallel, the driver turns Omega+ generated code into a parallel inspector by simply adding an `omp parallel for` pragma before the outermost loop. The reason why the inspectors are obviously parallel is that each iteration of their outermost loop just connects dependence edges for the row (column) of the same iteration in the dependence graph structure.

6.2 Optimization

A straightforward approach to implementing the instantiation procedure in Section 4.2 would be to take the quantifier-free formula resulting from instantiation, replace the uninterpreted functions, and directly pass it to ISL. However, this approach does not scale to large numbers of instantiations. An instantiated assertion is encoded as a union of two constraints $(\neg p \vee q)$. Given n instantiations, this approach introduces 2^n disjunctions to the original relation, although many of the clauses may be empty. In some of our dependence relations, the value of n may exceed 1000, resulting in a prohibitively high number of disjunctions. We have observed that having more than 100 instantiations causes ISL to start having scalability problems.

We apply an optimization to avoid introducing disjunctions when possible. Given a set of instantiations, the optimization adds the instantiations to the dependence relation in two phases. The first phase only instantiates those that do not introduce disjunctions to the dependence relation. During this phase, we check if the antecedent is already part of the dependence constraint, and thus is always true. If this is the case, then q can be directly added to the dependence relation. We also perform the same for $\neg q \implies \neg p$ and add $\neg p$ to the dependence relation if $\neg q$ is always true. The second phase adds the remaining instantiations that introduce disjunctions. This optimization helps reduce the cost of dependence testing in two ways: (1) if the relation is unsatisfiable after the first phase, disjunctions are completely avoided; and (2) the second phase only instantiates the remainder, reducing the number of disjunctions.

If the dependence relation remains non-empty after the second phase, then the relation is checked at runtime. All equalities in a relation are made explicit before inspector code generation.

Table 2. The benchmark suite used in this paper. The suite includes the fundamental blocks in several applications.

Kernel	Format	Source	Index array properties
Gauss-Seidel	CSR	Intel MKL [66]	Strict Monotonicity + Periodic Monotonicity
Incomplete LU	CSR	Intel MKL [66]	Strict Monotonicity + Periodic Monotonicity + CoMonotonicity
Incomplete Chol.	CSC	SparseLib [43]	Strict Monotonicity + Periodic Monotonicity + Triangularity
Forward Solve	CSC	Sympiler [15]	Strict Monotonicity + Periodic Monotonicity + Triangularity
Forward Solve	CSR	[65]	Strict Monotonicity + Periodic Monotonicity + Triangularity
Sparse MV Mul.	CSR	Common	Strict Monotonicity + Periodic Monotonicity + Triangularity
Static Left Chol.	CSC	Sympiler [15]	Strict Monotonicity + Periodic Monotonicity + Triangularity

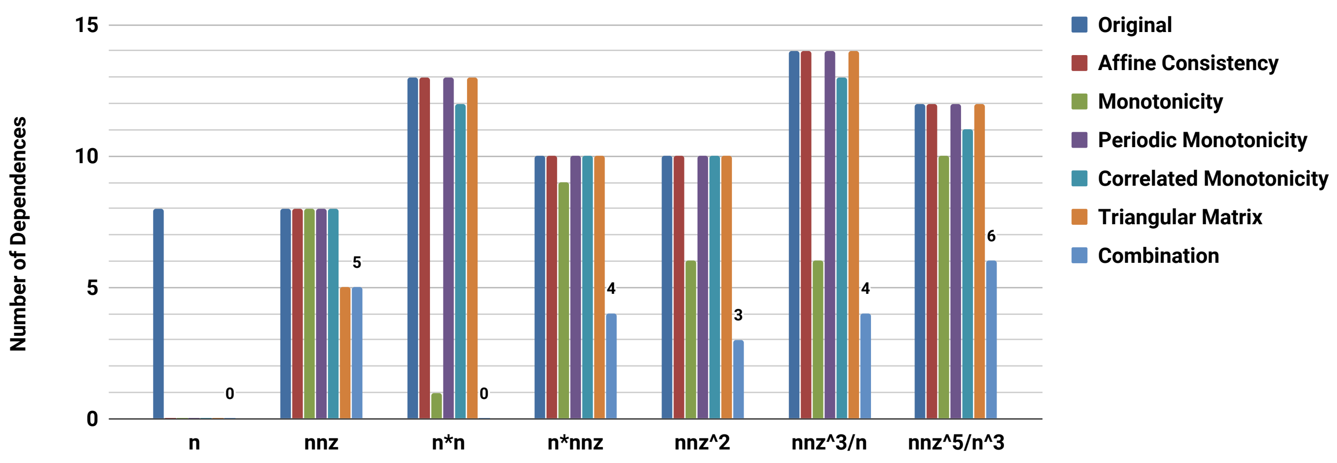

Figure 7. The number of dependences left after disproving dependences using index array properties. This is a direct application of the approach by Mohammadi et al. [34]. The complexities are calculated after applying optimizations used in previous work [63]: projecting out iterators whenever possible, and utilizing equalities already present in the dependence. Note that *nnz* is the number of non-zeros, and *n* is the number of columns/rows in a matrix. Affine Consistency means removing dependences that can be detected as unsatisfiable without utilizing any domain information, which leaves 67 out of 75 dependences. The array properties discussed in Table 1 detect 45 relations as unsatisfiable out of the remaining 67. Combination is when all properties are used together that leads to finding 12 unsatisfiable dependences that are not detectable otherwise.

7 Evaluating Simplification

In this section, we evaluate the impact of our simplification methods in terms of algorithmic complexity and number of required checks. The next section evaluates the performance impact of using our data dependence analysis approach for automatic wavefront parallelization. The benchmark suite we use to evaluate the presented data dependence simplification in this paper can be seen in Table 2.

7.1 Discarding Unsatisfiable Dependences

There are a total of 75 unique dependence relations in this suite. Of those 75, eight can be found unsatisfiable just by looking at their affine constraints, leaving us with 67 dependences to analyze. Applying the approach proposed by Mohammadi et al. [34] detects 45 out of the 67 dependences as unsatisfiable using domain knowledge, leaving us with 22 compile-time-satisfiable dependences that need runtime inspection.

Figure 7 shows how using different index array properties from Table 1 help reduce the number of dependences in different complexity classes. When applied alone, monotonocity has the highest impact helping us detect 27 dependences unsatisfiable. Co-monotonocity and triangularity each help us detect 3 dependences as unsatisfiable. Using all properties in concert detects 45 out of 67 dependences as unsatisfiable. We need the combined information of more than one property to detect 12 dependences as unsatisfiable. Consequently, the effect of using all properties is bigger than the summation of using individual properties.

The starting point of our simplification techniques is after unsatisfiable dependences are discarded using the approach we proposed in Mohammadi et al. [34]. Figure 8 summarizes the impact of simplification including statically disproving

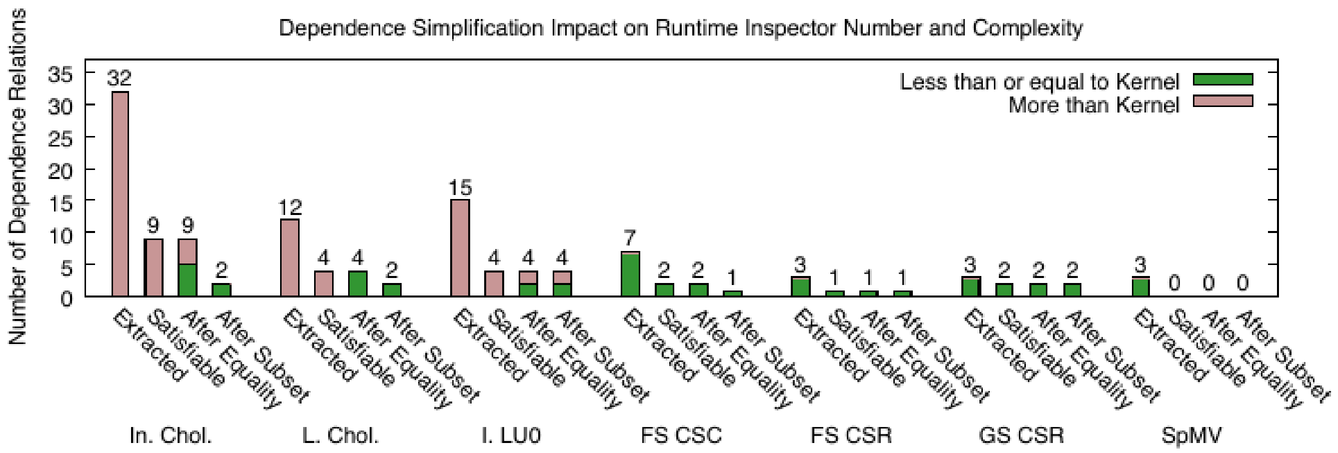

Figure 8. Impact of dependence simplification on inspector complexity. Each bar shows the required number of runtime checks and a breakdown into expensive (complexity higher than the kernel) and relatively inexpensive ones. The groups of four bars for each benchmark illustrate the impact of simplifications as they are successively applied. Satisfiable is after using domain-specific information to statically disprove a subset of dependences. After Equality is when equality detection is used on the remaining dependences, which replaces some of the expensive dependences with inexpensive ones. After Subset is when all the simplifications are combined, further reducing the number of checks using subset relationships.

dependences using that work. The x-axis shows how simplification impacts the number and complexity of the remaining compile-time dependence relations for each kernel. The first bar for each kernel is the number of remaining compile-time satisfiable dependences after disregarding all unsatisfiable ones that the approach proposed in Mohammadi et al. [34] can detect. In terms of algorithmic complexity, dependences in each bar in this figure are divided into two parts; a part that represents the number of dependences that have complexity greater than the kernel the dependences are extracted from, and a part that represents the number of dependences with complexity less than or equal to their respective kernel.

For the simpler kernels (FS CSR, GS CSR, and SpMV), finding unsatisfiable dependences is sufficient and additional simplification does not further reduce the inspector overhead. (Note that SpMV does not need domain-specific information to disprove dependences across outermost loops.)

However, the remaining kernels have expensive runtime checks that cannot be disproved even with domain-specific information. Inspectors with complexities exceeding the main computation do not finish in a reasonable amount of time to benefit from parallelism that it exposes.

7.2 Simplifying Using Equalities

After removing all unsatisfiable dependences from each code using domain-specific information, we are still left with 22 dependences that need inspectors across the 7 kernels. One of our simplifications presented in Section 4 reduces the complexity of inspectors by identifying additional equalities. The impact of this simplification is the difference between the second and third bars for each kernel in Figure 8.

An important positive effect is that a total of 11 checks:

- 5 (out of 9) expensive checks for Incomplete Cholesky,
- 2 (out of 4) expensive checks for Incomplete LU, and
- 4 (out of 4) expensive checks for Left Cholesky

are all simplified to have complexities less than or equal to the original computation.

7.3 Simplifying with Subset Relationships

Simplification using equalities leaves 11 dependences across the benchmarks that are further simplified using subset relationships as described in Section 5. The last bar for each code in Figure 8 shows how subset dependences help decrease the complexity of the overall inspectors for each code.

Interesting results are:

- The number of runtime checks for Incomplete Cholesky is reduced from 9 to 2, and all expensive checks are eliminated.
- The number of runtime checks for Left Cholesky is reduced from 4 to 1.

7.4 Overall Impact of Simplification

Table 3 summarizes the impact of our proposed approach on inspector complexity. For each code, the second column shows algorithmic complexity of inspectors that are needed for all the potential dependences, the third column shows the complexity of inspectors needed after detecting unsatisfiable dependences and applying our simplification methods, and

Table 3. The impact of simplification on inspector complexity. The baseline inspector complexity is when all possible dependences are tested at runtime, without using any of the simplifications proposed in this paper. The simplified inspector complexity reports the final cost of inspection generated by our approach. The overall complexity of inspectors decreases considerably. The complexity of the kernels are included for comparison; k and K denote constant factors where $k \ll K$.

Kernel name	Inspector complexity	Simplified inspector	Kernel complexity
Gauss-Seidel CSR	$(n) + 2(nnz)$	$2(nnz)$	$k(nnz)$
Forward solve CSR	$(n) + 2(nnz)$	nnz	$k(nnz)$
Forward solve CSC	$3(n) + 4(nnz)$	nnz	$k(nnz)$
Sparse MV Mul. CSR	$3(n)$	0	$k(nnz \times (nnz/n))$
Incomplete Cholesky CSR	$8(n^2) + 10(nnz^2) + 8(nnz^2 \times (nnz/n)) + 6(nnz^2 \times (nnz/n)^3)$	$(nnz \times (nnz/n)) + (nnz \times (nnz/n)^2)$	$K(nnz \times (nnz/n)^2)$
Left Cholesky CSC	$4(n^2) + 8(n \times nnz)$	$2(nnz)$	$K(nnz \times (nnz/n))$
Incomplete LU CSR	$(n^2) + 2(n \times nnz) + 6(nnz^2 \times (nnz/n)) + 6(nnz^2 \times (nnz/n)^3)$	$2(nnz \times (nnz/n)^2) + 2(nnz \times (nnz/n)^4)$	$K(nnz \times (nnz/n)^2)$

the fourth column states the complexity of the code itself. For all of the benchmarks except ILU, the algorithmic complexity of the inspector is less than or equal to the algorithmic complexity of the kernel. For ILU, the higher complexity can be dealt with using approximation [63].

8 Wavefront Parallelization Performance

We evaluate the pragmatic impact of the data dependence simplification approach presented in this paper on automatic wavefront parallelization with an end-to-end performance evaluation. We automatically generate inspectors based on simplified data dependences and used them to wavefront parallelize our benchmarks. Note that the inspectors are also parallelized since the outermost loops do not carry any dependence.

8.1 Methodology

We ran the parallelization experiments for 5 of the 7 benchmarks. Two kernels were excluded because (1) wavefront parallelization is not applicable to SpMV since it has a fully parallel loop and (2) inspectors for Incomplete LU are too expensive even after our simplifications as discussed in the previous section.

We ran the experiments on a machine with an Intel®Core™ i7-6900K CPU, 32GB of 3000MHz DDR4 memory, and Ubuntu 16.04 OS. The CPU has 8 cores, and thus we record performance with 8 OpenMP threads with hyper-threading disabled. We report the median of 5 runs and did not observe any significant variations between runs. All the codes are compiled with GCC 5.4.0 with -O3. Table 4 lists five matrices from the SuiteSparse Matrix Collection [17] that were used as inputs. The matrices are listed in order, by the number of nonzeros per column (row), an important factor when running numerical benchmarks in parallel since their outermost loop typically enumerates over columns (or rows). Table 5 shows the serial execution times for each kernel given the specific input matrix.

Table 4. Input Matrices for parallelized codes from [17]. Sorted in order of Number of Nonzeros per Column.

Matrix	Columns	Nonzeros	NNZ per Col.
af_shell3	504,855	17,562,051	35
msdoor	415,863	19,173,163	46
bmwcra_1	148,770	10,641,602	72
m_t1	97,578	9,753,570	100
crankseg_2	63,838	14,148,858	222

Table 5. Serial execution time (in **seconds**) for one run of each pair of kernel and input matrix.

Matrix	af_shell3	msdoor	bmwcra_1	m_t1	crankseg_2
FS CSC	0.037	0.042	0.022	0.020	0.028
FS CSR	0.039	0.046	0.025	0.022	0.033
GS CSR	0.031	0.035	0.018	0.016	0.022
In. Chol.	4.2	8.8	9.9	17.7	127.6
L. Chol.	160.1	50.6	107.8	57.5	138.3

We use the load-balanced level coarsening (LBC) wavefronts described by Cheshmi et al. [14] that create well-balanced coarsened level sets. This mitigates two issues: (i) the number of waves (levels) in wavefront parallelism increases with the DAG critical path length, leading to higher synchronization overheads; and (ii) non-uniform workloads in sparse kernels, such as Cholesky, creating imbalanced load.

8.2 Executor Speedup

Figure 9 shows the speedup of the wavefront executor for each benchmark over the library serial code listed in Table 2. All 6 executors benefit from wavefront parallelization, but less complex kernels have lower parallel efficiency since the amount of work is small.

The Left Cholesky's executor speedup is superlinear and cannot be seen in the figure; it ranges from 5 to 625. The reason for this anomaly is the dominance of memory access time in the Cholesky kernel. Left cholesky accesses several parts of the factor, during the factorization process, which is typically stored in different places in memory. The load-balanced level coarsening (LBC) greatly improves memory locality over serial code leading to super linear speedup.

8.3 Inspector Overhead

We now examine the performance of inspectors relative to their respective executors. We are unable to compare with inspectors before simplification, as their complexity makes them prohibitively expensive. For instance, Incomplete Cholesky has non-simplified dependences with algorithmic complexity of $O(nnz^2 * (nnz/n)^3)$ that would take hours to inspect, whereas its simplified dependences with algorithmic complexity of $O(nnz*(nnz/n)^2)$ can be inspected in less than 2 minutes.

It is common for the inspectors used in iterative solvers, even if they are hand-optimized, to take longer to execute than the computation itself. However, the cost of the inspection in parallel libraries is justified by the fact that the inspector is run once while the executor is repeatedly used in the iterations of the iterative solver. Iterative solvers typically execute hundreds of iterations until convergence [9, 29, 38]. For each kernel, we calculate the number of times that the parallel executor, using our inspectors, has to execute to start gaining performance over serial code. Figure 10 summarizes the break-even point (**Y-axis is in log scale.**).

For 3 codes, namely Gauss-Seidel CSR, and Forward Solve CSC and CSR, that are iterative solvers, we need to run the executor 40-60 times to amortize inspector overhead and gain performance. This is largely due to the low parallel efficiency achieved by these kernels - there is little room to further simplify the inspectors.

For more compute-intensive kernels, the cost of inspectors are much smaller than a single run of the executor, which means that the inspector-executor code is faster than the serial baseline even for one execution of the kernel.

9 Related Work

This section details our contributions over previous work. Additionally, our data dependence simplification approaches use quantifier elimination to produce more constraints. This section reviews other quantifier elimination techniques, other wavefront parallelization approaches, and algorithm-specific data dependence analysis approaches.

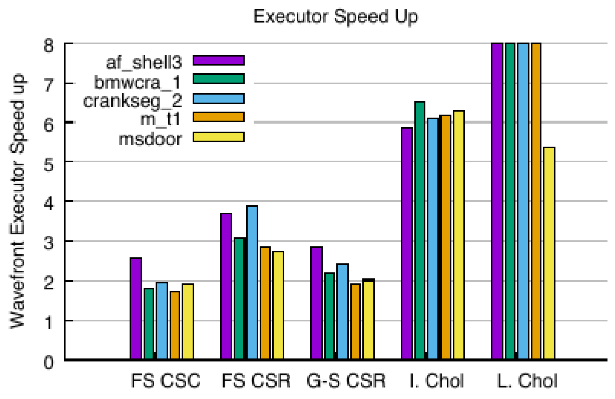

Figure 9. This figure shows wavefront parallel executor's speedup over serial code run that parallel inspector+executor is based on. The numbers for L. Chol. in order are: 8.9, 107, 625, 116, 5.3.

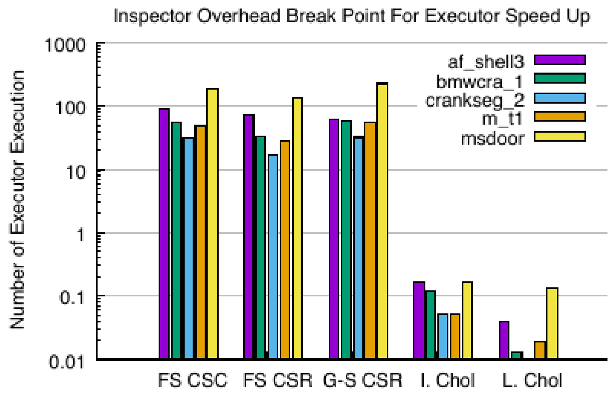

Figure 10. This figure shows how many times we need to run the executor for a benchmark given a specific input matrix in order to cover the inspector overhead and gain speedup. The X-Axis shows cluster of bars for each benchmark code, each bar being the number of needed run for a specific input matrix. The numbers in the Y-axis are $(inspector_t + executor_t)/(serial_t - executor_t)$. Please note **Y-axis is in log scale.**

9.1 Using Index Array Properties

McKinley [33] exploits user assertions about index arrays 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. [31] present a compile-time analysis for determining index array properties, such as monotonicity. They use the analysis results for parallelization of sparse matrix computations.

Venkat et al. [63] also used index array properties to simplify data dependence analysis in the context of wavefront parallelization of sparse codes. Their specification of the index array properties was specialized for two kernels, namely Gauss-Seidel and Incomplete LU0. Monotonicity and co-monotonicity index-array properties could be expressed, but periodic monotonicity and triangularity could not due to the restricted way assertions were encoded [2]. They also hard-coded which loop-carried dependences could be ignored, whereas this paper builds on the work by Mohammadi et al. [34] to automate this process by using index array properties to find unsatisfiable dependences.

Mohammadi et al. [34] focused on using index array properties automatically to find DOALL loops in sparse computations. They introduce a more expressive way of encoding index array properties than Venkat et al. [63] and use an SMT solver to detect unsatisfiable dependences. In comparison to Mohammadi et al. [34], this paper focuses on simplifying satisfiable dependences, which is needed for effectively finding DOACROSS loops that can benefit from wavefront parallelization. Additionally, Mohammadi et al. [34] use an SMT solver to find unsatisfiable dependences. Since current SMT solvers do not provide interfaces to expose implicit equalities in a given formula, we use a procedure to instantiate quantified constraints and then use ISL to detect equalities.

In summary, our contributions over the work by Mohammadi et al. [34] and Venkat et al. [63] are: (1) an instantiation procedure to expose implicit equalities with ISL; and (2) a procedure to detect when a dependence relation involving uninterpreted function calls is subsumed by another at compile time. Simplifying runtime checks for satisfiable dependences using the above provides significant reduction in overhead of automatically generated inspector.

9.2 Quantifier Elimination Techniques

The area of SMT-solving is advancing at a significant pace; the webpage for SMT-COMP[3] provides a list of virtually all actively developed solvers, and how they fared in each theory category. As these solvers are moving into a variety of domains, quantifier instantiation and elimination has become a topic of central interest. Some of the recent work in this area are: E-matching [35], Model-Based [19], Conflict-Based [52], and Counter-Example Guided [51].

These efforts make it clear that quantifier elimination is challenging, and is an area of active development. SMT solvers often rely on heuristic-driven instantiations to show unsat for difficult problems. In this context, our work can be viewed as heuristic instantiation where the heuristic is inspired by decidable fragments of the array theory.

Dependence constraints with universally quantified assertions are related to the first order theory fragments described by Bradley et al. [10] as undecidable extensions to their array theory fragment. However, Löding et al. [32] claim that the proofs for undecidability of extension theories [10] are incorrect, and declare their decidability status as an open problem. Regardless of whether the theory fragment that encompasses our dependence constraints is decidable or not the following is true: if we soundly prove that a relation is unsatisfiable just with compile-time information, the unsatisfiability applies in general, and having runtime information would not change anything. However, if a dependence is detected to be satisfiable just with compile-time information, we need to have runtime tests to see if it is actually satisfiable given runtime information, and even if it is, run time tests would determine for what values the dependence holds.

9.3 Wavefront Parallelization

For sparse codes, even when loops carry dependences, the dependences themselves may be sparse, and it may be possible to execute some iterations of the loop in parallel (previously denoted *partially parallel*). The parallelism is captured in a task graph, and typically executed as a parallel wavefront. Some prior work write specialized code to derive this task graph as part of their application [8, 39, 40, 47, 56, 68] or with kernel-specific code generators [12]. For example, Saltz and Rothbergs worked on manual parallelization of sparse triangular solver codes in the 1990s [53, 55]. There is also more recent work on optimizing sparse triangular solver NVIDIA GPUs and Intel's multi-core CPUs [50, 66]. Although manual optimizations have been successful at achieving high performance in some cases, significant programmer effort has to be invested for each of these codes, automating these strategies can significantly reduce that effort.

Other approaches automate the generation of inspectors that find task-graph, wavefront or partial parallelism. Rauchwerger et al. [48] and Huang et al. [26] have developed efficient and parallel inspectors that maintain lists of iterations that read and write each memory location. By increasing the number of dependences found unsatisfiable, the approach presented in this paper reduces the number of memory accesses that would need to be tracked. For satisfiable dependences, there is a tradeoff between inspecting iteration space dependences versus maintaining data for each memory access. That choice could be made at runtime. There are also other approaches for automatic generation of inspectors that have looked at simplifying the inspector by finding equalities, using approximation, parallelizing the inspector, and applying point-to-point synchronization to the executor [63].

9.4 Algorithm-Specific Data Dependence Analysis

An algorithm-specific approach to represent data dependences and optimize memory usage of sparse factorization algorithms such as Cholesky [42] uses an *elimination tree*, but to the best of our knowledge, this structure is not derived automatically from source code. When factorizing a column of a sparse matrix, in addition to nonzero elements of the input matrix new nonzero elements, called fill-in, might be created. Since the sparse matrices are compressed for efficiency, the

[3] http://smtcomp.sourceforge.net/2017/

additional fills during factorization make memory allocation ahead of factorization difficult. The elimination tree is used to predict the sparsity pattern of the L factor ahead of factorization so the size of the factor can be computed [16] or predicted [21, 22], and captures a potential parallel schedule of the tasks. Prior work has investigated the applicability of the elimination tree for dependence analysis for parallel implementation [20, 23–25, 27, 41, 50, 57, 67]. Some techniques use the elimination tree for static scheduling [20, 24, 41], that is while others use it for runtime scheduling.

10 Other Applications

Besides wavefront parallelism, there are many other uses for sparse data dependence analysis that can benefit from a reduction in runtime inspection overhead. Other uses include extending static race detectors [5] to incorporate runtime tests, dynamic iteration space slicing [44], runtime transformations such as sparse tiling [18, 60], and high-level synthesis (HLS) [3].

Race detection: Dynamic race detection is an essential prerequisite to the parallelization of existing sequential codes. While static analysis methods employed in race detectors [5] can often suppress race checks on provably race-free loops, they fail to do so when presented with non-affine access patterns. Consequently, race detectors would require runtime checks, that have performance overhead as well as memory overhead, often increasing memory usage by a factor of four. Our simplifications can bring down both the time and memory complexity of runtime inspections in these checkers.

Dynamic program slicing: Another application can be found in program slicing. Pugh and Rosser introduced the concept of iteration space slicing where program slicing is done on a loop iteration basis using Presburger representations [44]. Similar dynamic approaches for tiling across loops in sparse codes were presented by various groups [18, 60]. All of these techniques would require runtime dependence analysis, thus disproving dependences or reducing the complexity of runtime inspection would be applicable.

High-level synthesis: Dependence simplification can also be utilized in high-level synthesis (HLS). In HLS, it is important to pipeline the innermost loops to get efficient hardware. Alle et al. have proposed using runtime dependence checks to dynamically check if an iteration is in conflict with those currently in the pipeline, and add delays only when necessary [3].

Distributed memory parallelization: Another possible application of our work can be found in the work by Ravishankar et al. [49]. The authors produce distributed parallel code that uses MPI for loops where there might be indirect loop bounds, and/or array accesses. The read and write sets/data elements of each process are computed via an inspector where indirect accesses are involved to determine if each process is reading/writing data that is owned by other processes. Basumallik and Eigenmann use run-time inspection of data dependences to determine how to reorder a loop to perform computation and communication overlap [7].

11 Conclusion

In this paper, we present an automated approach for reducing the overhead of inspectors for wavefront parallelization of sparse code. The overhead reduction comes in three forms: statically disproving dependences, reducing the complexity by detecting equalities, and removing tests that are subsumed by another by identifying subset relationships. All of these optimizations are enabled by taking advantage of domain knowledge about index arrays.

Our approach is inspired by work in satisfiability problems and in SMT solvers. The techniques for satisfiability problems can be directly applied to disprove dependence relations involving uninterpreted function calls and universally quantified assertions. However, in the context of inspectors for sparse code, it is not sufficient to check for satisfiability. The key insight in our paper is that the techniques from satisfiability problems combined with manipulation of integer sets provide a powerful framework to reason about runtime dependence checks.

Parallelization of these sparse numerical methods is an active research area today, but one where most current approaches require *manual parallelization*. It is also worth noting that without inspector complexity reduction, most inspectors would timeout, thus underscoring the pivotal role of the work in this paper in enabling parallelization and optimization of sparse codes. Our results are established over 63 dependences extracted from 7 sparse numerical methods. A large number of the dependences can be removed and simplified to automatically produce low overhead inspectors. Our end-to-end evaluation demonstrates that these inspectors may be used to improve the performance of iterative solvers combined with executors for wavefront parallelization. An interesting direction of future work is to apply the simplification techniques in other contexts discussed in Section 10.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. NSF CCF-1563732. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation. We would also like to acknowledge Ganesh Gopalakrishnan for his contributions to an earlier version of this paper.

References

[1] 2019. Compiler Technology to Optimize Performance (CTOP) research group webpage at Utah. http://ctop.cs.utah.edu/ctop/?page_id=21
[2] 2019. IEGenLib library, SC16 artifact github repository. https://github.com/CompOpt4Apps/IEGenLib/tree/SC16_IEGenLib

[3] Mythri Alle, Antoine Morvan, and Steven Derrien. 2013. Runtime Dependency Analysis for Loop Pipelining in High-level Synthesis. In *Proceedings of the 50th Annual Design Automation Conference (DAC '13)*. ACM, New York, NY, USA, Article 51, 10 pages. https://doi.org/10.1145/2463209.2488796

[4] Simone Atzeni, Ganesh Gopalakrishnan, Zvonimir Rakamaric, Dong H. Ahn, Ignacio Laguna, Martin Schulz, Gregory L. Lee, Joachim Protze, and Matthias S. Müller. 2016. ARCHER: Effectively Spotting Data Races in Large OpenMP Applications. In *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016*. 53–62. https://doi.org/10.1109/IPDPS.2016.68

[5] S. Atzeni, G. Gopalakrishnan, Z. Rakamaric, D. H. Ahn, I. Laguna, M. Schulz, G. L. Lee, J. Protze, and M. S. MÃijller. 2016. ARCHER: Effectively Spotting Data Races in Large OpenMP Applications. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 53–62. https://doi.org/10.1109/IPDPS.2016.68

[6] Denis Barthou, Jean-François Collard, and Paul Feautrier. 1997. Fuzzy Array Dataflow Analysis. *J. Parallel and Distrib. Comput.* 40, 2 (1997), 210–226.

[7] Ayon Basumallik and Rudolf Eigenmann. 2006. Optimizing irregular shared-memory applications for distributed-memory systems. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM Press, New York, NY, USA, 119–128.

[8] Nathan Bell and Michael Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, New York, NY, USA, 1–11.

[9] Michele Benzi, Jane K Cullum, and Miroslav Tuma. 2000. Robust approximate inverse preconditioning for the conjugate gradient method. *SIAM Journal on Scientific Computing* 22, 4 (2000), 1318–1332.

[10] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. 2006. What's Decidable About Arrays?. In *Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '06)*, E. Allen Emerson and Kedar S. Namjoshi (Eds.). 427–442. https://doi.org/10.1007/11609773_28

[11] T. Brandes. 1988. The importance of direct dependences for automatic parallelism. In *Proceedings of the International Conference on Supercomputing*. ACM, New York, NY, USA, 407–417.

[12] Jong-Ho Byun, Richard Lin, Katherine A. Yelick, and James Demmel. 2012. *Autotuning Sparse Matrix-Vector Multiplication for Multicore*. Technical Report. UCB/EECS-2012-215.

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

[14] Kazem Cheshmi, Maryam Mehri Dehnavi, Shoaib Kamil, and Michelle Mills Strout. 2018. ParSy: Inspection and Transformation of Sparse Matrix Computations for Parallelism. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '18)*. ACM, New York, NY, USA.

[15] Kazem Cheshmi, Shoaib Kamil, Michelle Mills Strout, and Maryam Mehri Dehnavi. 2017. Sympiler: Transforming Sparse Matrix Codes by Decoupling Symbolic Analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*. ACM, New York, NY, USA, Article 13, 13 pages. https://doi.org/10.1145/3126908.3126936

[16] Thomas F Coleman, Anders Edenbrandt, and John R Gilbert. 1986. Predicting fill for sparse orthogonal factorization. *Journal of the ACM (JACM)* 33, 3 (1986), 517–532.

[17] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1.

[18] Craig C. Douglas, Jonathan Hu, Markus Kowarschik, Ulrich Rüde, and Christian Weiß. 2000. Cache Optimization for Structured and Unstructured Grid Multigrid. *Electronic Transaction on Numerical Analysis* (February 2000), 21–40.

[19] Yeting Ge and Leonardo de Moura. 2009. *Complete Instantiation for Quantified Formulas in Satisfiabiliby Modulo Theories.* Springer Berlin Heidelberg, Berlin, Heidelberg, 306–320.

[20] Alan George, Joseph WH Liu, and Esmond Ng. 1989. Communication results for parallel sparse Cholesky factorization on a hypercube. *Parallel Comput.* 10, 3 (1989), 287–298.

[21] John R Gilbert. 1994. Predicting structure in sparse matrix computations. *SIAM J. Matrix Anal. Appl.* 15, 1 (1994), 62–79.

[22] John R Gilbert and Esmond G Ng. 1993. Predicting structure in nonsymmetric sparse matrix factorizations. In *Graph theory and sparse matrix computation*. Springer, 107–139.

[23] John R Gilbert and Robert Schreiber. 1992. Highly parallel sparse Cholesky factorization. *SIAM J. Sci. Statist. Comput.* 13, 5 (1992), 1151–1172.

[24] Pascal Hénon, Pierre Ramet, and Jean Roman. 2002. PASTIX: a high-performance parallel direct solver for sparse symmetric positive definite systems. *Parallel Comput.* 28, 2 (2002), 301–321.

[25] Jonathan D Hogg, John K Reid, and Jennifer A Scott. 2010. Design of a multicore sparse Cholesky factorization using DAGs. *SIAM Journal on Scientific Computing* 32, 6 (2010), 3627–3649.

[26] J. Huang, T. B. Jablin, S. R. Beard, N. P. Johnson, and D. I. August. 2013. Automatically exploiting cross-invocation parallelism using runtime information. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 1–11. https://doi.org/10.1109/CGO.2013.6495001

[27] George Karypis and Vipin Kumar. 1995. A high performance sparse Cholesky factorization algorithm for scalable parallel computers. In *Frontiers of Massively Parallel Computation, 1995. Proceedings. Frontiers' 95., Fifth Symposium on the*. IEEE, 140–147.

[28] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and Dave Wonnacott. 1996. The Omega Calculator and Library, version 1.1.0.

[29] David S Kershaw. 1978. The incomplete CholeskyâĂŤconjugate gradient method for the iterative solution of systems of linear equations. *Journal of computational physics* 26, 1 (1978), 43–65.

[30] Daniel Kroening and Ofer Strichman. 2016. *Decision Procedures: An Algorithmic Point of View* (2nd ed.). Springer Berlin Heidelberg.

[31] Yuan Lin and David Padua. 2000. Compiler analysis of irregular memory accesses. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Vol. 35. ACM, New York, NY, USA, 157–168.

[32] Christof Löding, P. Madhusudan, and Lucas Peña. 2017. Foundations for Natural Proofs and Quantifier Instantiation. *Proc. ACM Program. Lang.* 2, POPL, Article 10 (Dec. 2017), 30 pages. https://doi.org/10.1145/3158098

[33] Kathryn McKinley. 1991. *Dependence Analysis of Arrays Subscriptecl by Index Arrays*. Technical Report TR91187. Rice University.

[34] Mahdi Soltan Mohammadi, Kazem Cheshmi, Maryam Mehri Dehnavi, Anand Venkat, Tomofumi Yuki, and Michelle Mills Strout. 2018. Extending Index-Array Properties for Data Dependence Analysis. In *Proceedings of The 31st International Workshop on Languages and Compilers for Parallel Computing (LCPC18)*.

[35] Leonardo Moura and Nikolaj Bjørner. 2007. Efficient E-Matching for SMT Solvers. In *Proceedings of the 21st International Conference on Automated Deduction: Automated Deduction (CADE-21)*. 183–198. https://doi.org/10.1007/978-3-540-73595-3_13

[36] Cosmin E. Oancea and Lawrence Rauchwerger. 2012. Logical inference techniques for loop parallelization. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 509–520.

[37] Yunheung Paek, Jay Hoeflinger, and David Padua. 2002. Efficient and Precise Array Access Analysis. *ACM Trans. Program. Lang. Syst.* 24, 1 (Jan. 2002), 65–109.

[38] M Papadrakakis and N Bitoulas. 1993. Accuracy and effectiveness of preconditioned conjugate gradient algorithms for large and ill-conditioned problems. *Computer methods in applied mechanics and engineering* 109, 3-4 (1993), 219–232.

[39] Jongsoo Park, Mikhail Smelyanskiy, Narayanan Sundaram, and Pradeep Dubey. 2014. Sparsifying Synchronization for High-Performance Shared-Memory Sparse Triangular Solver. In *Proceedings of the 29th International Conference on Supercomputing - Volume 8488 (ISC 2014)*. Springer-Verlag New York, Inc., New York, NY, USA, 124–140.

[40] Jongsoo Park, Mikhail Smelyanskiy, Karthikeyan Vaidyanathan, Alexander Heinecke, Dhiraj D. Kalamkar, Xing Liu, Md. Mosotofa Ali Patwary, Yutong Lu, and Pradeep Dubey. 2014. 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 (SC '14)*. IEEE Press, Piscataway, NJ, USA, 945–955.

[41] Alex Pothen and Chunguang Sun. 1993. A mapping algorithm for parallel sparse Cholesky factorization. *SIAM Journal on Scientific Computing* 14, 5 (1993), 1253–1257.

[42] Alex Pothen and Sivan Toledo. 2004. Elimination Structures in Scientific Computing.

[43] Roldan Pozo, Karin Remington, and Andrew Lumsdaine. 1996. SparseLib++ v. 1.5 Sparse Matrix Class Library reference guide. *NIST Interagency/Internal Report (NISTIR)-5861* (1996).

[44] William Pugh and Evan Rosser. 1997. Iteration space slicing and its application to communication optimization. In *Proceedings of the 11th international conference on Supercomputing*. ACM Press, 221–228.

[45] William Pugh and David Wonnacott. 1995. Nonlinear Array Dependence Analysis. In *Third Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*. Troy, New York.

[46] William Pugh and David Wonnacott. 1998. Constraint-Based Array Dependence Analysis. *ACM Transactions on Programming Languages and Systems* 20, 3 (1 May 1998), 635–678.

[47] L. Rauchwerger, N. M. Amato, and D. A. Padua. 1995. Run-Time Methods for Parallelizing Partially Parallel Loops. In *Proceedings of the ACM International Conference on Supercomputing (ICS)*. ACM, New York, NY, USA, 137–146.

[48] Lawrence Rauchwerger, Nancy M. Amato, and David A. Padua. 1995. A Scalable Method for Run-Time Loop Parallelization. *International Journal of Parallel Programming* 23, 6 (1995), 537–576.

[49] Mahesh Ravishankar, Roshan Dathathri, Venmugil Elango, Louis-Noël Pouchet, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2015. Distributed Memory Code Generation for Mixed Irregular/Regular Computations. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2015)*. ACM, New York, NY, USA, 65–75. https://doi.org/10.1145/2688500.2688515

[50] Steven C Rennich, Darko Stosic, and Timothy A Davis. 2016. Accelerating sparse Cholesky factorization on GPUs. *Parallel Comput.* 59 (2016), 140–150.

[51] Andrew Reynolds, Morgan Deters, Viktor Kuncak, Clark W. Barrett, and Cesare Tinelli. 2015. On Counterexample Guided Quantifier Instantiation for Synthesis in CVC4. *CoRR* abs/1502.04464 (2015). http://arxiv.org/abs/1502.04464

[52] Andrew Reynolds, Cesare Tinelli, and Leonardo de Moura. 2014. Finding Conflicting Instances of Quantified Formulas in SMT. In *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design (FMCAD '14)*. Article 31, 8 pages.

[53] Edward Rothberg and Anoop Gupta. 1992. Parallel ICCG on a hierarchical memory multiprocessor - Addressing the triangular solve bottleneck. *Parallel Comput.* 18, 7 (1992), 719 – 741. https://doi.org/10.1016/0167-8191(92)90041-5

[54] Silvius Rus, Jay Hoeflinger, and Lawrence Rauchwerger. 2003. Hybrid analysis: static & dynamic memory reference analysis. *International Journal Parallel Programming* 31, 4 (2003), 251–283.

[55] Joel H. Saltz. 1990. Aggregation Methods for Solving Sparse Triangular Systems on Multiprocessors. *SIAM J. Sci. Stat. Comput.* 11, 1 (Jan. 1990), 123–144. https://doi.org/10.1137/0911008

[56] Joel H. Saltz, Ravi Mirchandaney, and Kay Crowley. 1991. Run-Time Parallelization and Scheduling of Loops. *IEEE Trans. Comput.* 40, 5 (1991), 603–612.

[57] Olaf Schenk and Klaus Gärtner. 2002. Two-level dynamic scheduling in PARDISO: Improved scalability on shared memory multiprocessing systems. *Parallel Comput.* 28, 2 (2002), 187–197.

[58] Robert E. Shostak. 1979. A Practical Decision Procedure for Arithmetic with Function Symbols. *J. ACM* 26, 2 (April 1979), 351–360. https://doi.org/10.1145/322123.322137

[59] Kevin Streit, Johannes Doerfert, Clemens Hammacher, Andreas Zeller, and Sebastian Hack. 2015. Generalized Task Parallelism. *ACM Trans. Archit. Code Optim.* 12, 1, Article 8 (April 2015), 25 pages. https://doi.org/10.1145/2723164

[60] Michelle Mills Strout, Larry Carter, Jeanne Ferrante, and Barbara Kreaseck. 2004. Sparse Tiling for Stationary Iterative Methods. *International Journal of High Performance Computing Applications* 18, 1 (February 2004), 95–114.

[61] M. M. Strout, M. Hall, and C. Olschanowsky. 2018. The Sparse Polyhedral Framework: Composing Compiler-Generated Inspector-Executor Code. *Proc. IEEE* 106, 11 (Nov 2018), 1921–1934.

[62] Michelle Mills Strout, Alan LaMielle, Larry Carter, Jeanne Ferrante, Barbara Kreaseck, and Catherine Olschanowsky. 2016. An Approach for Code Generation in the Sparse Polyhedral Framework. *Parallel Comput.* 53, C (April 2016), 32–57.

[63] Anand Venkat, Mahdi Soltan Mohammadi, Jongsoo Park, Hongbo Rong, Rajkishore Barik, Michelle Mills Strout, and Mary Hall. 2016. Automating Wavefront Parallelization for Sparse Matrix Computations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*. IEEE Press, Piscataway, NJ, USA, Article 41, 12 pages. http://dl.acm.org/citation.cfm?id=3014904.3014959

[64] Sven Verdoolaege. 2010. isl: An integer set library for the polyhedral model. In *Proceedings of the 3rd International Congress on Mathematical Software (ICMS '10)*. 299–302.

[65] Richard Vuduc, Shoaib Kamil, Jen Hsu, Rajesh Nishtala, James W Demmel, and Katherine A Yelick. 2002. Automatic performance tuning and analysis of sparse triangular solve. ICS.

[66] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. 2014. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi*. Springer, 167–188.

[67] Ran Zheng, Wei Wang, Hai Jin, Song Wu, Yong Chen, and Han Jiang. 2015. GPU-based multifrontal optimizing method in sparse Cholesky factorization. In *Application-specific Systems, Architectures and Processors (ASAP), 2015 IEEE 26th International Conference on*. IEEE, 90–97.

[68] Xiaotong Zhuang, A.E. Eichenberger, Yangchun Luo, K. O'Brien, and K. O'Brien. 2009. Exploiting Parallelism with Dependence-Aware Scheduling. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE Computer Society, Los Alamitos, CA, USA, 193–202.