Sparso: Context-driven Optimizations of Sparse Linear Algebra

Hongbo Rong Jongsoo Park Lingxiang Xiang
Todd A. Anderson Mikhail Smelyanskiy

Parallel Computing Lab, Intel Corporation
{hongbo.rong,jongsoo.park,lingxiang.xiang,todd.a.anderson,mikhail.smelyanskiy}@intel.com

ABSTRACT

The sparse matrix is a key data structure in various domains such as high-performance computing, machine learning, and graph analytics. To maximize performance of sparse matrix operations, it is especially important to optimize across the operations and not just within individual operations. While a straightforward per-operation mapping to library routines misses optimization opportunities, manually optimizing across the boundary of library routines is time-consuming and error-prone, sacrificing productivity.

This paper introduces Sparso, a framework that automates such optimizations, enabling both high performance *and* high productivity. In Sparso, a compiler and sparse linear algebra libraries collaboratively discover and exploit *context*, which we define as the invariant properties of matrices and relationships between them in a program. We present compiler analyses, namely *collective reordering analysis* and *matrix property discovery*, to discover the context. The context discovered from these analyses drives key optimizations across library routines and matrices.

We have implemented Sparso with the Julia language, Intel MKL and SpMP libraries. We evaluate our context-driven optimizations in 6 representative sparse linear algebra algorithms. Compared with a baseline that invokes high-performance libraries without context optimizations, Sparso results in 1.2∼17× (average 5.7×) speedups. Our approach of compiler-library collaboration and context-driven optimizations should be also applicable to other productivity languages such as Matlab, Python, and R.

1. INTRODUCTION

Exploiting sparse connectivity is increasingly important with bigger data, as exemplified by recent work on sparse linear algebra in areas of high-performance computing [18], machine learning [50], and graph analytics [7]. However, harnessing the full potential of modern computing systems for sparse matrix operations poses unique challenges. In dense linear algebra, BLAS and LAPACK interfaces effectively abstract the complexity of highly-optimized library implementations, achieving both performance *and* productivity. In sparse linear algebra, however, a similar approach misses important optimization opportunities that take advantage of the specific sparsity structure of input matrices and whose scope goes beyond individual kernels. We call these optimizations *context-driven optimizations*. Manually applying these optimizations is time-consuming and error-prone even for expert programmers [45] and thus hampers productivity.

See the skeleton code below that is extracted from pre-conditioned conjugate gradient (PCG), a widely used algorithm for solving linear systems.

```
1: L = ichol(A)        # Incomplete Cholesky factorization
2: while !converged
3:     ... = A·p       # Sparse-matrix vector multiplication
4:     z=L\r           # Forward triangular solver
5:     z=L'\z          # Backward triangular solver
6:     ...
```

A programmer attempting to optimize this code can easily map key operations like sparse-matrix vector multiplication (SpMV, $A \cdot p$) and sparse triangular solver ($L \backslash r$ [1]) to high-performance library routines. However, such a straightforward per-operation mapping is 5.6× slower than code with context-driven optimizations, as will be shown in Section 7.

With the above PCG example, let us briefly introduce context-driven optimizations, which will be detailed later. Sparse triangular solver has data-dependent parallelism [44], so we use the *inspector-executor* approach [39, 49], a context-driven optimization. We *inspect* the input matrix at run-time to arrange triangular solver's computation as a task dependency graph (TDG). Then, we *execute* the tasks in parallel using the TDG. Since the inspection involves non-trivial overhead, we reuse the inspection result to amortize the overhead. However, manually reusing the inspection results adds a burden to programmers; they have to reason about code regions where matrices preserve constant sparsity structures.

Another context-driven optimization, matrix reordering [17, 20, 42, 51], can improve the cache locality of sparse matrix operations. However, this optimization needs to be propagated to other related matrices and vectors. In the PCG code, if we reorder matrix L, we also need to reorder vectors r and z that are used in the same expressions with L. We also need to coordinate among multiple kernels because each may prefer different reordering. For example, in the PCG code, SpMV and triangular solvers may prefer different reordering.

Still, additional context-driven optimizations that will be described in Section 3 are needed for the best performance of PCG. In general, implementing these context-driven optimizations for any application like those in Table 1 needs a considerable amount of effort.

This paper introduces Sparso, a framework that automates context-driven optimizations for sparse matrix applications via

[1] $z = L \backslash r$ is a computation such that $L \cdot z = r$.

PACT '16, September 11-15, 2016, Haifa, Israel

 ISBN 978-1-4503-4121-9/16/09…$15.00
DOI: http://dx.doi.org/10.1145/2967938.2967943

Table 1: Evaluated applications with their brief descriptions, including optimization opportunities

Application	Domain	Description	Source of Matrices	Optimization Opportunities: Exploit Matrix Properties	Optimization Opportunities: Collective Reordering
`PCG`	Operations research, HPC	A representative of sparse iterative solvers	UF collection [14]	✓	✓
`IPM` [37]	Operations research	Interior-point method for convex optimization problems	UF collection [14]	✓	
`L-BFGS` [35]	Machine learning	Optimization algorithm popular for parameter estimation	libSVM [8, 34]	✓	✓
`CoSP2` [41]	HPC	Calculate density matrix in electronic structure theory	CoSP2 website [41]	✓	
`Adiabatic`	HPC	Validate annealing speed of quantum adiabatic simulation		✓	
`PageRank` [6]	Graph analytics	Rank web pages according to their popularity	UF collection [14]	✓	✓

collaboration between a compiler and libraries. Programmers write straightforward code, and Sparso transparently generates code with context-driven optimizations. The framework centers around the concept of *context*, which we define as the invariant properties of matrices and relationships between them in a code region (like a function or a loop). *Static context* is discovered by compiler analyses, and *run-time context* is discovered by run-time inspections in libraries.

As shown in Figure 1, static context is discovered by key compiler analyses, including *collective reordering analysis* and *matrix property discovery*, whose results are populated in two tables, *reordering control table (RCT)* and *matrix property table (MPT)*, respectively. The tables are shared with a context-sensitive wrapper that is a thin layer around existing libraries like Intel math kernel library (MKL). With the static context in the tables, the wrapper invokes the libraries to inspect matrices, and caches the inspection results back to the tables as run-time context. Sparso generates code such that the run-time context can be reused by subsequent invocations of library routines.

In the PCG code above, the compiler can discover static context like "A is symmetric" and "L has the same sparsity structure as the lower triangular part of A". Such information can be exploited by libraries, as will be described in Section 3. Library routines inspect sparse matrices and generate run-time context like TDG, reordering permutation, and different sparse matrix storage formats.

This paper makes the following contributions:

1. *Sparso framework for context-driven optimizations.* We demonstrate that compiler-library collaboration can effectively automate context-driven optimizations, achieving performance *and* productivity for sparse linear algebra algorithms. Even though inspector-executor and reordering are known ideas, Sparso integrates and automates them to be easily usable by non-expert programmers in productivity languages like Julia [4], Matlab, Python, and R.
2. *Compiler analyses for matrix property discovery and reordering.* We propose a set of dataflow analyses for discovering properties of matrices and relationships among them. We also propose collective reordering, which automates matrix reordering with minimal overhead. The outcome of these analyses drives library optimizations and achieves higher performance.
3. *Performance comparable to manually-optimized implementations.* Compared to a baseline that invokes high-performance library kernels without context, Sparso yields 1.2-17× (average 5.7×) speedups, on a 14-core Intel Haswell Xeon processor. This evaluation is with the 6 representative sparse linear algebra algorithms from various domains shown in Table 1. Sparso's PCG performance corresponds to 67% of the underlying hardware's capabilities and is close to the top results in recent HPCG benchmark lists [18][2]. Performance on other algorithms are also comparable to manually optimized results [41, 46, 54, 55].

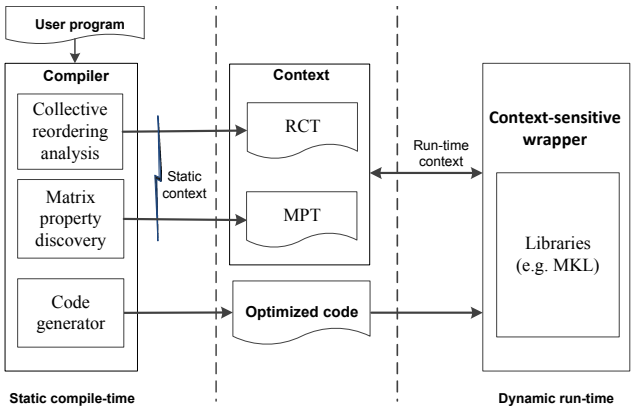


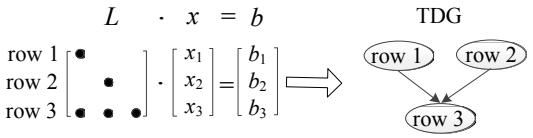
Figure 1: Sparso Work-flow: A compiler discovers static context and generates optimized code. When executed, the optimized code invokes context-sensitive routines in a wrapper interface around existing libraries. Guided by the static context, the wrapper routines generate reusable run-time context, and invoke the best specialized library routines. Both kinds of context are stored in two tables, RCT and MPT.

The rest of this paper is organized as follows: Section 2 defines the notations and reviews preliminaries, which can be skipped by those who are familiar with sparse linear algebra. Section 3 overviews Sparso framework using PCG as an example. Section 4 presents two compiler analyses, matrix property discovery and collective reordering, that derive context-driven optimizations. Section 5 shows examples on how compiler and library can collaboratively optimize sparse linear algebra algorithms by communicating context. Section 6 briefly describes our specific implementation of Sparso framework in Julia, which is used for evaluation presented in Section 7. The readers are encouraged to refer to Appendix B for our open source package on Github (https://github.com/IntelLabs/Sparso.git).

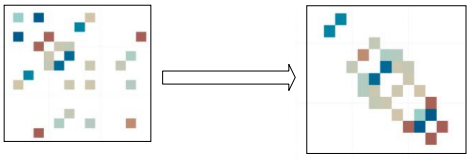
2. SPARSE LINEAR ALGEBRA PRELIMINARIES

In this paper, an *array* refers to either a column vector (denoted as lower-case letters like x) or a matrix (denoted as capital letters like A). We denote scalars as Greek letters like α. In sparse matrices, we store non-zeros only, which are located by *(sparsity) structure* of the matrix [25]. For example, if a matrix is $\left[\begin{smallmatrix}1&0&0\\2&4&5\\3&0&0\end{smallmatrix}\right]$, its structure can be figuratively shown as $\left[\begin{smallmatrix}\bullet&\cdot&\cdot\\\bullet&\bullet&\bullet\\\bullet&\cdot&\cdot\end{smallmatrix}\right]$. Structures are

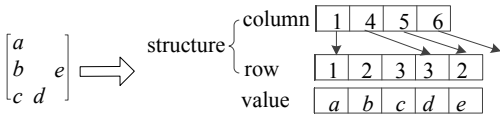
[2] HPCG (high-performance conjugate gradient) is a recent benchmark ranking supercomputers by the performance of solving sparse linear systems. The top result was achieved with manually applying various context-driven optimizations that has taken considerable amount of effort [45]. Note that the effort devoted to a high-profile benchmark like HPCG is rarely affordable for general applications, where automation in Sparso is of great value.



(a) Task dependence graph



(b) Matrix reordering



(c) CSC storage format

Figure 2: Sparse linear algebra preliminaries: TDG, reordering, and storage format

of particular interest because it is often the structure, not the actual values, that determines parallelism and data locality.

For example, Figure 2a shows how structure determines parallelism for the forward triangular solver, which solves $L \cdot x = b$ for the unknown x, where L is lower triangular. Because of the non-zeros' positions, the equations for rows 1 and 2 involve only x_1 and x_2, respectively. Thus, we can solve the two equations in parallel. However, the equation for row 3 involves all unknowns and cannot be solved until the equations of rows 1 and 2 are solved. Therefore, from the structure of matrix L only, we can build a *task dependence graph (TDG)*, as shown to the right of the figure.

Structure also determines data locality. *Matrix reordering* permutes the rows and columns of a sparse matrix to improve locality. Figure 2b shows a matrix before and after being reordered with reverse Cuthill-Mckee (RCM) [11], a well-known reordering algorithm. As illustrated, RCM clusters the non-zeros more closely. In $A \cdot x$ for example, this improves temporal locality of vector x.

A sparse matrix can be stored in various formats. Depending on the sparsity structures of matrices and the computation to perform, one format can outperform the others. Figure 2c shows a common storage format, *compressed sparse columns* (CSC), where non-zeros are stored column by column, along with the corresponding row numbers, and each column has an index pointing to the first non-zero in the column. In CSC, column indices and row numbers collectively represent structure, and, as long as they are constant, inspection results can be often reused. Another common format, *compressed sparse rows* (CSR), is similar to CSC but stores non-zeros row-by-row instead of by columns.

3. OVERVIEW OF SPARSO FRAMEWORK

This section overviews Sparso using PCG as an example. Subsequent sections describe compiler analyses (Section 4) and the compiler-library interface (Section 5) in more detail.

Figure 3a shows a straightforward implementation of PCG, which solves $A \cdot x = b$ for the unknown x. Understanding the details of the code is unnecessary to grasp the optimization opportunities. We can make several observations:

1. *Iterative code structure*: PCG is a typical sparse linear algebra algorithm that repeats key kernels such as SpMV ($A \cdot p$), forward triangular solver ($L \backslash r$), and backward triangular solver ($L^\intercal \backslash z$) in lines 8-20.
2. *Matrices with constant values and structures*: A and L are never updated, once generated.
3. *Matrices with special structures and relations among them*: Incomplete Cholesky factorization with zero fill-ins (`ichol` in line 1) requires the input A to be symmetric in value, and returns L whose structure is same as the lower triangular part of A.
4. *Storage formats affect performance*: When parallelizing SpMV ($A \cdot p$ in line 10), CSR is preferred to CSC for A to avoid write sharing of the output vector.
5. *Reordering can help performance*: Kernels here can benefit from matrix reordering. For SpMV ($A \cdot p$ in line 10), reordering A improves temporal locality of vector p (recall Figure 2b). Similarly, for the triangular solvers ($z = L \backslash r$ and $z = L^\intercal \backslash z$ in lines 17 and 18), reordering L reduces false sharing in vector z [45].

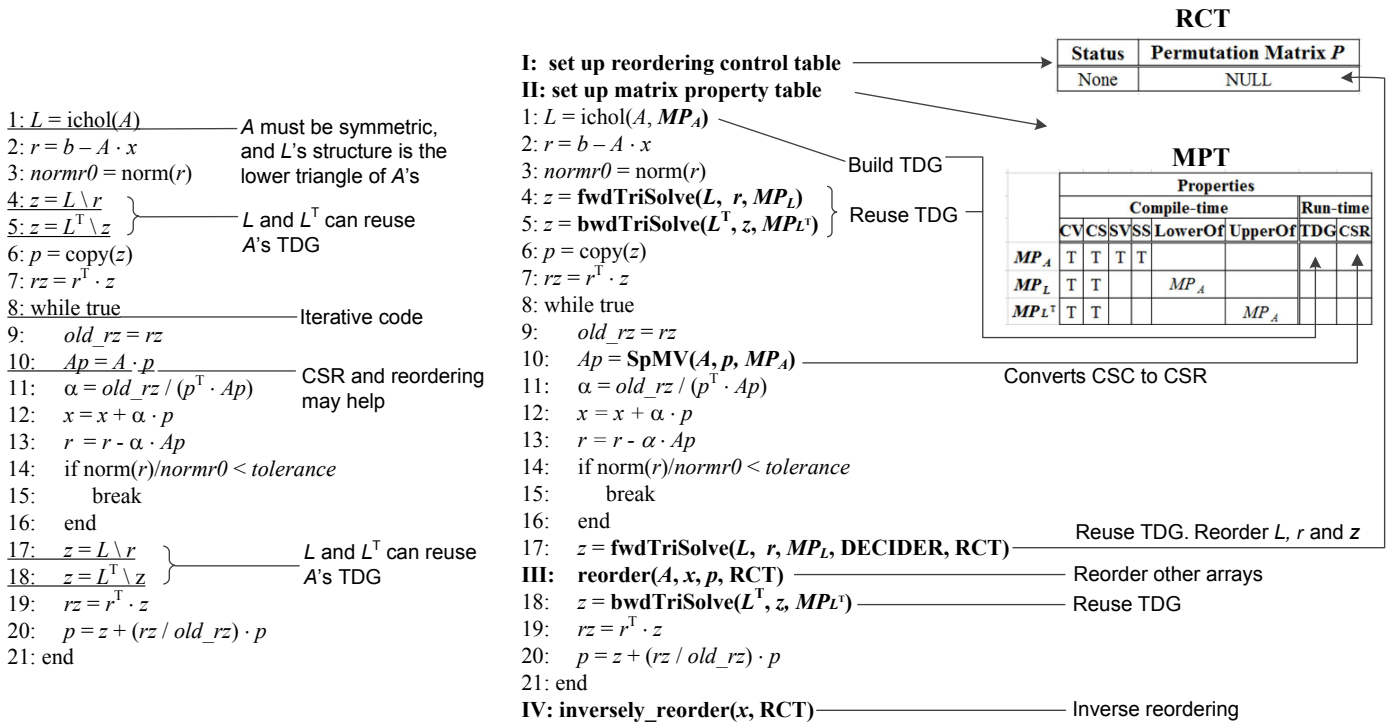
Our compiler discovers these facts from the original code shown in Figure 3a, and produces the optimized code shown in Figure 3b. The differences are highlighted in bold face, and new statements are numbered with Roman numerals.

The optimized code in Figure 3b first sets up RCT and MPT (lines I and II). RCT represents the run-time state of collective reordering with a status and a permutation matrix field. MPT has entries for the 3 matrices A, L, and $L^\intercal$, denoted as MP_A, MP_L, and $MP_{L^\intercal}$, respectively. Each entry has both static and run-time context. *Static context* includes 4 booleans, `CV` (constant-valued), `CS` (constant-structured), `SV` (symmetric-valued), and `SS` (symmetric-structured). Note that all 3 matrices are constant in value and structure, and A is symmetric in value and structure. Static context also includes two relations, `isLowerOf` and `isUpperOf`, between the matrices' structures. The table shows that the structures of L and $L^\intercal$ are the same as those of the lower and upper triangular parts of A, respectively. *Run-time context* includes a `TDG` and a storage format (`CSR`) field, for this example.

In line 1, the library kernel `ichol` is called with context, MP_A. The kernel may inspect A to build a TDG once (recall Figure 2a) and save the result in MP_A's `TDG` field. This TDG is reused by the triangular solvers in lines 4, 5, 17, and 18: When a triangular solver is invoked, the solver checks MPT, finds that its input matrix's structure is constant and is the lower or upper triangular part of the symmetric matrix A, and thus it can reuse the TDG.

Suppose A is in CSC format, the default format in Julia. In line 10, SpMV may transparently convert A into CSR for more efficient parallelization, and saves the result in MP_A's `CSR` field. Such a conversion happens only once: the next time SpMV is invoked, SpMV can directly access the CSR matrix, because A has not been changed (MP_A`.CV`=true). In fact, in this specific case, the conversion is unnecessary even once, because A is symmetric (MP_A`.SV`=true), which makes A in CSC and CSR identical.

The kernels operating with matrices here may benefit from reordering. However, each may prefer a different permutation. We appoint one kernel as the *decider of reordering*, based on an experiential estimate how much the kernel may benefit from reordering. The kernel that is estimated to benefit most from reordering becomes the decider. Forward triangular solver in line 17 typically has the biggest benefit from reordering, hence it will be chosen as the decider (indicated by the `DECIDER` parameter). When executed, the solver will find a permutation that improves its performance and reorder its inputs L and r, and consequently the output z. Then, the solver will save the permutation, which is represented by a *permutation matrix*, into RCT, and set `RCT.status` to `Init`, indicating that an initial set of arrays have been reordered.



(a) Original PCG code with annotations for optimization opportunities

(b) Optimized code and its execution

Figure 3: An illustration of representative code structure and optimization opportunities in sparse linear algebra algorithms.

After the initial set of arrays have been reordered by the decider, the other arrays that will directly or indirectly take part in a computation with these reordered arrays have to be reordered as well. Here, arrays A, x, and p are related to the already reordered arrays (L,r, and z). Therefore, in line III, `reorder(`A, x, p, `RCT)` is executed. It checks if `RCT.status` equals `Init`, and if so, reorders A, x, and p, and sets `RCT.status` to `Done`, meaning that no more reordering is needed afterwards.

After the loop, live arrays have to be inversely reordered. In our example, x is the only live array to be returned from PCG function. Therefore, in line IV, `inversely_reorder(`x, `RCT)` is executed. It checks if `RCT.status` equals `Done`, and, if so, it inversely reorders x.

Note that we only show the fields of RCT and MPT needed for PCG in Figure 3b. RCT and MPT can have other fields. In general, every permutation matrix can have a field in RCT, and any matrix property can have a field in MPT (e.g., `isDiagonal`, `isTriangular`, etc.), as can be seen later in Sections 4 and 5.

4. COMPILER ANALYSES

We first describe how to discover matrix properties (Section 4.1), and then collective reordering (Section 4.2). The outcome of these compiler analyses is communicated with libraries as will be described in Section 5.

4.1 Matrix Property Discovery

A set of dataflow analyses discover the following matrix properties. First, we analyze the program to identify arrays that are of *constant size, constant value*, and/or *constant structure* in any single run of the program. Note that these compiler analyses do not inspect the actual data of the matrices, which are typically unavailable at compile-time. This distinguishes matrix property discovery from run-time inspection. These analyses extend classical constant propagation algorithm [31, 60], in a way similar to symbolic constant propagation [10]. In interior-point method (IPM), for example, based on that A and D have constant structures, we derive that $B=A\cdot D\cdot A^\intercal$ also has a constant structure.

Table 2: Example rules for the compiler to derive matrix properties

1.	isDiagonal(D) → isSymmetric($A\cdot D\cdot A^\intercal$)
2.	isSymmetric(A), isSymmetric(B) → isSymmetric($\alpha\cdot A+\beta\cdot B$)
3.	B=tril(A) → isLowerTriangular(B)
4.	B=diag(x) → isDiagonal(B)
5.	B=spones(A) → $S_B=S_A$, isUnweighted(B)
6.	$B=\alpha\cdot A$ → $S_B=S_A$
7.	isDiagonal(D) → $S_{D\cdot A}=S_{A\cdot D}=S_A$
8.	isDiagonal(D) → $S_{A\cdot D\cdot B}=S_{A\cdot B}$
9.	L=ichol(A) → isSymmetric(A), isLowerTriangular(L), $S_L=S_{\text{lowerTriangleOf}(A)}$

Note:
S_A: structure of A.
isDiagonal(D): D is diagonal.
isUnweighted(B): B is unweighted.
isSymmetric(A): A is symmetric in value (and thus also in structure).
isLowerTriangular(L): L is lower triangular.
lowerTriangleOf(A): the lower triangle of A.
spones(A): a Julia/Matlab operation that creates a sparse matrix with the same structure as that of A, but with 1 in every non-zero position.
tril(A) : a Julia/Matlab operation returning the lower triangle of A.
diag(x) : a Julia/Matlab operation that creates a sparse matrix with the elements of vector x on the main diagonal.

Second, we analyze the program for special properties of matrices such as *symmetric value, symmetric structure, triangular structure, diagonal structure, and unweighted*[3], using basic prop-

[3] Unweighted sparse matrices represent unweighted graphs, where we only specify existence of connection, not the weight, between a pair of nodes. All non-zero values are 1, thus we only need to store structures of unweighted matrices, not their values.

erties in linear algebra like the first five rules in Table 2. This approach is similar to Rose's static structural inference [16].

Finally, we analyze the program for relations between the matrices. Example relations are: A and B have the same structure, B is the transpose of A, or L has the same structure as the lower triangular part of A. Rules 5–9 in Table 2 show example rules that we use to discover the relations.

The time complexity of these analyses is $O(EV^2)$, where E is the number of edges in the control flow graph, and V is the number of arrays. This is the same complexity as Kildall's constant propagation algorithm [31], which has been extended symbolically for most of the above analyses.

4.2 Collective Reordering

Reordering and the associated concepts of distributivity and inter-dependence are described in Section 4.2.1. Generating reordering code is then described in Section 4.2.2. Additional supportive details for these topics are contained in the Appendix A.

4.2.1 Reordering, Distributivity, and Inter-dependence

Reordering R is defined as follows:

$$R(A) = P_A \cdot A \cdot Q_A^\intercal \quad (1)$$

$$R(x) = P_x \cdot x \quad (2)$$

$$R(\alpha) = \alpha, \quad (3)$$

where A, x and α represent a matrix, a vector, and a scalar, respectively, as said in Section 2; P_A is a row permutation matrix for A and Q_A is a column permutation matrix for A.

Any permutation matrix P has a useful property:

$$P \cdot P^\intercal = I, \quad \text{the identity matrix.} \quad (4)$$

With this property, the *inverse reordering* can be easily derived[4].

Reordering R is *distributive* over function f if the function's semantics remains the same whether its output is reordered, or its inputs are reordered. That is,

$$R(f(i_1,i_2,...,i_n)) = f(R(i_1),R(i_2),...,R(i_n)), \quad (5)$$

where i_1,i_2,...,i_n are the inputs of f.

Reordering is usually distributive over array expressions. For example, it is distributive over SpMV, because

$$R(A \cdot x) = P \cdot (A \cdot x) = (P \cdot A \cdot Q^\intercal) \cdot (Q \cdot x) = R(A) \cdot R(x).$$

Similarly, reordering is distributive over $A \cdot B$, $A \pm B$, $A \backslash x$, $x \pm y$, $dot(x, y)$, $\alpha \cdot A$, and $\alpha \cdot x$. A few non-distributive cases are discussed in Appendix A.1.

Inter-dependent arrays are a set of arrays that, if any one of them is reordered, all the others have to be reordered. Suppose A is reordered in expression, $y = A \cdot x$. Since the columns of A are permuted, the corresponding rows of x need to be permuted in the same way, i.e. $Q_A = P_x$. Also, since the rows of A are permuted, the corresponding rows of y need to be permuted in the same way, i.e. $P_A = P_y$. Similarly, if x (or y) is reordered, the columns of A (or rows of A) have to be reordered, respectively. In short, x is inter-dependent with the columns of A, and y is inter-dependent with the rows of A. For arrays inter-dependent with each other, the same permutation matrix is used for reordering. How to identify inter-dependent arrays is described in Appendix A.2.

4.2.2 A Code Generation Scheme with Reordering

The significance of distributivity and inter-dependence is that they enable reordering within an arbitrary piece of code without changing its semantics. Intuitively, if a reordering R is distributive over all the expressions in a region of code, we can insert code before that region to reorder the arrays that are inputs to that code region. Likewise, at the end of the code region, inverse reordering is applied to the arrays that are outputs of the code region and are inter-dependent with the inputs, so as to restore the original semantics of the code.

[4] For example, from Equation 1, we have $P_A^\intercal \cdot R(A) \cdot Q_A = A$. The left side of this equation tells how to recover the original matrix A from a reordered matrix $R(A)$.

THEOREM 4.1 (COLLECTIVE REORDERING). *Given a reordering R, if R is distributive over every expression in an arbitrary piece of code except for array element accesses, then it can be applied to the code in the following way, without changing its semantics:*

Action 1 *At each entry of the code, reorder every array that is live into the code at that entry. Let S be the set of arrays that are inter-dependent with any of these reordered arrays.*

Action 2 *For every element access to an array in S, replace the indices of the access with new indices in the way shown in Appendix A.1.*

Action 3 *At each exit of the code, inversely reorder every array that is in S and is live out of the code.*

For example, in the PCG code in Figure 3a, consider the loop in lines 8–21. Reordering is distributive over every expression in this loop. Usually, the program point right before the loop is regarded as an entry to the loop. However, as we explained in Section 3, considering the fact that there are multiple kernels in the loop that may benefit from reordering, but each might prefer a different reordering, we have to choose one kernel to decide a unique reordering for the loop. Therefore, for the purpose of reordering, the program point right before the decider should be treated as the entry to the loop. When executed, this decider will determine if it is beneficial to perform reordering, and if so, the decider will reorder its input arrays, and consequently, its output array as well. For the other live, unreordered arrays at that entry, the compiler inserts code at the program point after the decider to reorder them explicitly. That accomplishes Action 1.

In this particular example of Figure 3a, we pick the forward triangular solver at line 17, $L \backslash r$ (i.e. `fwdTriSolve`), as the decider of reordering. The loop is viewed to have 1 entry, which is the program point right before the decider (i.e. before line 17), and 1 exit, the program point right after the loop (i.e. after line 21). Since L, r, z, A, x, and p are live at that entry, we need to reorder them there. This is achieved partly by the decider function: L and r are reordered inside the decider function, and z is reordered as a natural consequence of that. After that, A, x, and p are explicitly reordered by a compiler-inserted call to *reorder* (See line III in Figure 3b), a routine in the wrapper interface, as explained before in Section 3. This finishes Action 1. There is no array element access in the loop, and thus Action 2 is unnecessary. For Action 3, since x is an inter-dependent array with the reordered arrays, and is live out at the exit, we inversely reorder it there (See line IV in Figure 3b). A proof of the theorem 4.1 is left to Appendix A.3.

5. COMPILER-LIBRARY INTERACTION

In Sparso, the compiler and libraries interact through context. Context is stored in RCT and MPT, which are shared between the compiler and a wrapper interface around existing libraries such as Intel MKL (recall Figure 1). When the compiler sees a sparse-matrix operation f, it replaces f with a call to its wrapper interface. The compiler passes additional parameters to the wrapper interface for communicating context. The context allows the wrapper to apply various optimizations such as dispatching to a version of f that follows a specialized code path.

Table 3 show examples on how compiler and library can collaboratively optimize sparse linear algebra operations by communicating context. Here, we explain two additional examples in more details:

Example 1. memoizing storage format conversion: Assume that f prefers a certain matrix storage format (e.g., SpMV

Table 3: Examples of matrix properties and relations discovered by the compiler that can be used for optimizing sparse linear algebra operations.

Operation	Matrix properties	Implementation in the wrapper interface of libraries	Applications
SpMV $A \cdot x$ (Assume A is in CSC, the default format in Julia)	Default	Call SpMV in CSC.	
	isConstantStruct(A)	Convert A to CSR, save it as run-time context. Call SpMV in CSR thereafter.	
	isSymmetric(A)	Call SpMV in CSR as if A is in CSR.	PCG, Adiabatic
	isUnweighted(A)	Call SpMV optimized for unweighted matrices.	Adiabatic, PageRank
	$B = A^\intercal$	Call SpMV in CSR with B.	L-BFGS
fwd/bwdTriSolve(L, x)	Default	Call inspector-executor pair every time	
	isConstantStruct(L)	Save inspection result (a TDG) as run-time context and reuse thereafter.	PCG, IPM
	$S_L = S_{\text{lowerTriangleOf}(A)}$	Reuse the TDG of A, if available.	PCG
$x = A \backslash b$ (solve $A \cdot x = b$)	Default	Sparse LU decomposition + triangular solves	
	isConstantStruct(A)	Save the inspection result of LU decomposition and reuse.	IPM
	isTriangular(A)	Call triangular solve.	PCG
$A = f(...)$ (a function generating a sparse matrix)	Default	Allocate space for A every time.	
	isConstantStruct(A) $\bigwedge$ hasDedicatedSpace(A)	Allocate space for A, save it as run-time context, and reuse thereafter.	IPM

Note:
isConstantStruct(A): A is constant in structure.
isTriangular(A): A is triangular.
hasDedicatedSpace(A): A has a pre-allocated space for its exclusive use.

prefers CSR to CSC for efficient parallelization). The wrapper function of f can first check if the run-time context in MPT already has a preferred format. If not, the wrapper may convert the input matrix from its current format to the preferred, and memoize the new format in MPT. The compiler also provides context information such as if the matrix has constant structure so that the wrapper function can tell if the conversion overhead can be amortized. Finally, the wrapper invokes an existing library routine for f.

Example 2. collective reordering: Assume that f is chosen by the compiler as the decider of reordering that potentially benefits from reordering. When first called, the wrapper function of f generates a permutation that optimizes the locality for f (e.g., RCM for SpMV and level-set ordering for triangular solvers [44]), reorders the input matrix using the permutation, saves the permutation in RCT as a run-time context, and marks a flag to record that a permutation has been created. Later, Sparso-generated code, `reorder` and `inverse_reorder`, checks the flag and reorders other inter-dependent arrays as in lines III and IV of Figure 3b.

6. A SPARSO IMPLEMENTATION IN JULIA

We have implemented the Sparso framework as a package of Julia, a new scripting language for technical computing [4]. In the current implementation, user specifies which function to optimize. Then, Sparso applies context-driven optimizations only to the loops in the function, as the loops usually dominate the execution time of the function. However, this is a specific implementation choice. In general, our approach can be applied to an arbitrary code region, including both loops and functions.

The compiler analyses in Section 4 are static, thus sometimes it can be difficult or impossible for the compiler to discover certain matrix properties, before actually looking at concrete input matrices at run-time. To address this situation, Sparso allows annotations in a user code. In our experiments with the 6 algorithms in Table 1, user annotations are needed only in a few places, and only 3 kinds of annotations are needed: isSymmetric, isUnweighted, and hasDedicatedSpace. The meaning of the first two annotations has been introduced before in Table 2. The other annotation, hasDedicatedSpace(x), tells the compiler that x has been pre-allocated a space, and that space is not shared with any other variable. Thus, for an expression like $x = f(...)$ inside the loop, the compiler will generate a call to routine f that reuses the allocated space for output in every loop iteration.

7. EVALUATION

We evaluate the performance of Sparso framework on a Haswell generation of Intel Xeon machine, with the details listed in Table 4. The peak memory bandwidth of the machine is measured by the STREAM benchmark [38], which is often the limiting factor of sparse matrix operations' performance due to their low arithmetic intensity [26, 61].

Our experiments use 6 representative sparse linear algebra algorithms listed in Table 1. Not only are these algorithms important in their own right, but also they capture common compute patterns of sparse linear algebra in various domains. We use double precision for floating point numbers. We use various sparse matrices, which is important for evaluating the performance of sparse linear algebra operations, as they heavily depend on specific sparsity structures: e.g., from the University of Florida matrix collection [14], we use all the symmetric positive definite matrices with more than 50 MB for `PCG`, and all the social network and road network matrices with more than 5 MB for `PageRank`.

We test each application with 4 different optimization configurations:

1. `Julia-as-is`: The applications are run in the original Julia, with neither our compiler nor libraries. Note that key linear algebra operations in Julia already call optimized C/Fortran library routines in LAPACK [1] and SuiteSparse [9].
2. `Baseline` (`Call-repl`): We replace all time-consuming linear algebra operations in Julia with calls to Intel MKL [30] and SpMP [43] library routines. Since MKL and SpMP are generally faster than the open source LAPACK and SuiteSparse on the tested machine, we obtain a reasonably high-performance baseline for comparison. Note that `Baseline` runs the library routines already in parallel with OpenMP.
3. `+Matrix-properties`: In addition to `Call-repl`, enable

Table 4: Evaluation Settings

Processor	14-core Xeon E5-2697 v3 @ 2.6 GHz 35MB on-chip L3$
Memory	64 GB DDR with 54 GB/s peak bandwidth
Julia	Version 0.4.6
Libraries	MKL 13.0 SpMP compiled with Intel compiler 15.0.3
Parallelization	OpenMP with options `OMP_NUM_THREADS=14` and `KMP_AFFINITY=granularity=fine,compact,1` (sparse matrix computations generally perform the best with 1 thread per core)

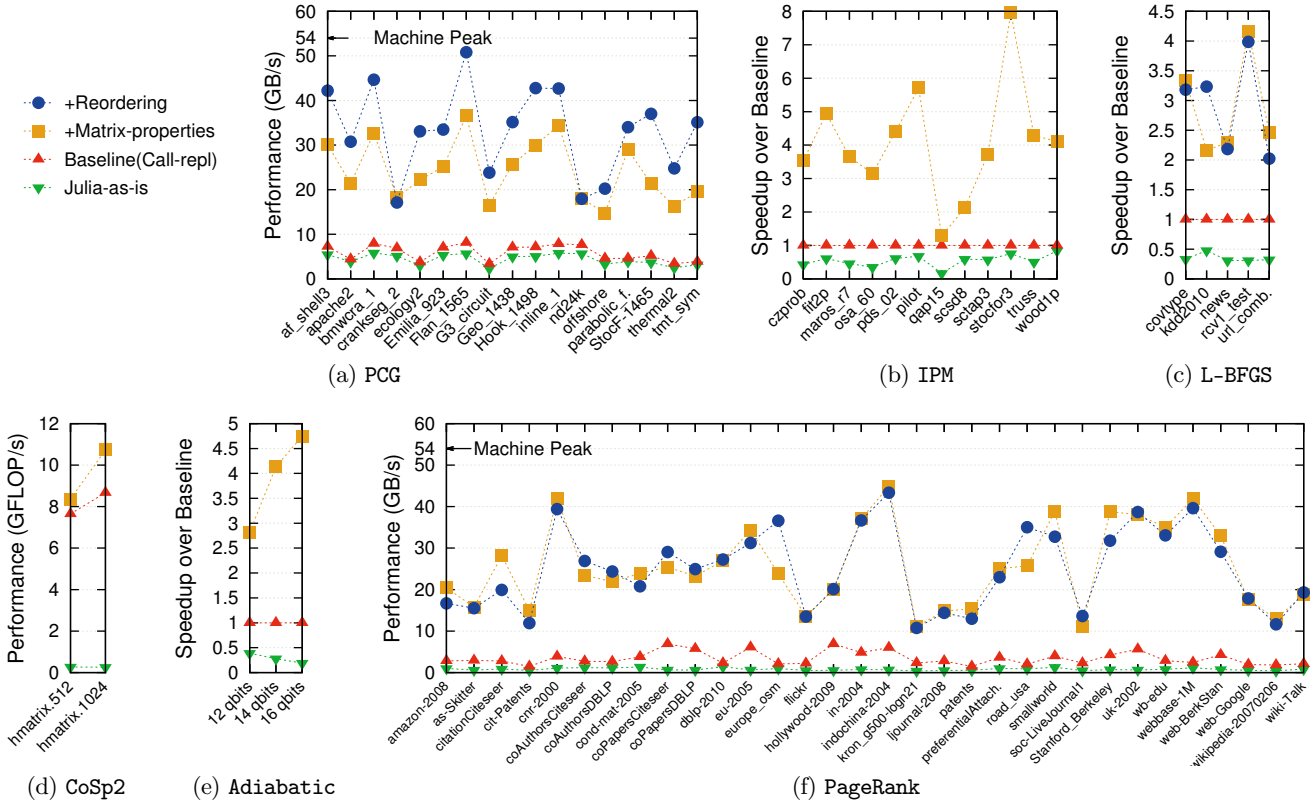


Figure 4: Performance of Sparso with different optimization options.

all the context-driven optimizations *except* collective reordering.

4. `+Reordering`: In addition to `+Matrix-properties`, enable collective reordering as well.

7.1 Overall Performance

Figure 4 shows the performance of the 6 applications with various input matrices. We report the results following a common practice in the high performance computing community. For memory bandwidth-bound applications, including `PCG` (only up to 1 floating-point operation per 6 bytes) and `PageRank` (only up to 1 floating-point operation per 2 bytes), we report performance in GB/s and also compare against the machine peak bandwidth; for the other applications that are neither memory- nor computation-bound, we report GFlop/s when it can be easily computed (`CoSP2`), or otherwise report the relative speedups over `Baseline` (`IPM`, `L_BFGS`, and `Adiabatic`).

With all Sparso optimizations enabled (`+Reordering`), significant speedups are observed for most test cases. The average (harmonic mean) speedups over `Baseline` across all inputs are 5.6×, 3.7×, 2.8×, 1.2×, 3.8×, and 17×, for `PCG`, `IPM`, `L-BFGS`, `CoSp2`, `Adiabatic`, and `PageRank`, respectively.

Noticeably, in `PCG` and `PageRank`, the performance of `+Reordering` often closely approaches the peak capability provided by the evaluated hardware. The performance of `PCG` and `PageRank` is bound by the memory bandwidth, and we utilize on average 67% and 50% of the peak memory bandwidth, respectively. This more-than 50% bandwidth efficiency in sparse matrix computations is hard to achieve even by highly optimized C/C++ implementations, as demonstrated by recent work such as [44, 45].

Comparing `Julia-as-is` and `Baseline`, we observe that *optimized library routines alone, without context information, provide limited speedups*. This result also quantitatively demonstrates the limitation of library-only optimizations for sparse matrix applications — the cost of optimization setup such as matrix inspection cannot be effectively amortized without context information. Even though key routines in `Julia-as-is` and `Baseline` are already parallelized, speedups from parallelization are quite limited. For example, `PCG` and `IPM` obtain only 1.4× and 1.5× speedups by using 14 cores on average, respectively, mainly because the inspection overhead for parallelization is not amortized. On the other hand, with context-driven optimizations, average parallelization speedups over `Julia-as-is` are 7.6× and 5.6×, respectively.

Collective reordering (`+Reordering` vs. `+Matrix-properties`) contributes significant speedups to `PCG`, making it approaching the peak capability of the hardware in several matrices, but exhibits marginal impact in `PageRank`, as some social network matrices used by `PageRank` are already reordered. From the software engineering point of view, Sparso allows programmers to quickly try out reordering without worrying about correctness and restructuring the code.

7.2 Performance of Individual Applications

PCG: Figure 4a shows the performance of `PCG` with incomplete LU with zero fill-ins pre-conditioner until it reaches 10^{-7} relative residual. The evaluated `PCG` implementation is similar to high performance conjugate gradient (HPCG)[5], a recently proposed benchmark for ranking supercomputers [18], and captures characteristics of a wide range of iterative sparse linear solvers

[5]One difference is that our PCG implementation does not use multi-grid method, but key context-driven optimizations are similar.

such as GMRES [53] and AMG [27]. We utilize on average 67% of the machine peak bandwidth, when all optimizations are applied (denoted as `+Reordering`). This corresponds to on average 5.6× higher performance than `Baseline`, and 7.6× higher performance than `Julia-as-is`.

The bulk of speedup comes from exploiting context information. Based on the matrix properties, we apply the following three optimizations: (1) We reuse the result of inspection, a task dependency graph, across the invocations of forward triangular solver. Similar reuse happens for backward triangular solver as well[6]. (2) We dispatch the Julia expression $A \cdot p$ to the SpMV implementation in the SpMP library with A in CSR format, utilizing the property that matrix A is symmetric in value, and thus avoiding unnecessary format conversion (parallel SpMV performs better with CSR than CSC, the default Julia representation). (3) We reorder the matrices and related vectors so that, in the triangular solvers, each thread accesses matrix rows in contiguous manner, reducing false sharing and improving spatial locality.

Relatively small speedups in `Baseline` compared to `Julia-as-is` demonstrates that invoking optimized library kernels without context information is insufficient. For example, even with an efficient parallel sparse triangular solver, the cost of constructing a task dependency graph by inspecting the input matrix every time negates most of parallelization speedups.

IPM: Figure 4b shows the performance of interior-point method for optimizing linear programming problems until it reaches 10^{-7} relative residual. Sparso with context-sensitive optimizations (`+Matrix-properties`) obtains on average 3.7× speedup over `Baseline`, and 5.6× speedup over `Julia-as-is`. In both `Baseline` and `+Matrix-properties`, the SpGEMM used in `IPM` is replaced with a call to an SpGEMM implementation in SpMP library that follows optimizations described in a recent work [46]. The sparse direct solver used in `IPM` is also replaced with a call to MKL PARDISO [30]. In `+Matrix-properties`, the compiler discovers that the output matrix of SpGEMM has a constant structure, which allows reusing the memory allocated for the output matrix. We also reuse the inspection result of the sparse direct solver by exploiting that the input matrices have constant structures. As a result of eliminating most of the redundancies in other parts of `IPM` computation, `+Matrix-properties` spends on average more than 45% of total time in the execution step of MKL sparse direct solver, similar to the manually optimized IPM implementation presented in [55].

L-BFGS: Figure 4c shows the performance of limited-memory Broyden-Fletcher-Goldfarb-Shanno [35], a quasi-Newton optimization method popular for parameter estimation in machine learning, until it reaches 10^{-10} relative residual. `L-BFGS` spends significant time in two SpMV calls: $A^\intercal \cdot x$ and $A \cdot x$. In `+Matrix-properties`, by utilizing the relationship between matrices (i.e. the first call already provides matrix A's transpose in CSC format, which is equivalent to matrix A in CSR format), the second call can be performed directly in CSR without the additional overhead of format conversion.

Compared to `+Matrix-properties`, collective reordering further speeds up `L-BFGS` by about 1.5× with one matrix, and has no significant slowdown with the other matrices. In contrast to `PCG` and `PageRank` where matrices are square, the sparse matrices handled by collective reordering in `L-BFGS` are rectangular: rows correspond to observations and columns correspond to features.

CoSP2: Sparso achieves a 1.1× speedup for `hmatrix.512`, and a 1.2× speedup for the larger `hmatrix.1024` matrix over `Baseline`, which are 31× and 42× speedup over `Julia-as-is`. `CoSP2` spends most of its time in $X \cdot X^\intercal$, an SpGEMM operation. In `Baseline` and `+Matrix-properties`, we replace the SpGEMM operation with an implementation that follows the optimizations described in [46]. `+Matrix-properties` exploits the fact discovered by the compiler that X stays symmetric, thus avoids transposing X. Sparso's 8.4 and 11 GFlop/s CoSP2 performance in double precision is comparable to that of highly optimized SpGEMM implementation in [46] that ranges up to 18 GFlop/s in single precision. Sparso actually outperforms the C implementation from [41] by 1.4 and 1.3× for `hmatrix.512` and `hmatrix.1024`, respectively, due to its more efficient SpGEMM implementation.

Adiabatic: Figure 4e shows the performance of validating annealing speed of adiabatic quantum computation [23] with varying number of qbits. Such validation in classical computation is necessary because adiabatic quantum computation relies on appropriate annealing speed. `Adiabatic` spends a significant fraction of its time in Lanczos algorithm that represents a wide range of iterative eigenvalue solvers and used in other scientific problems such as solving Schrödinger's equation [52]. In `+Matrix-properties`, a matrix is annotated as unweighted, so that we can use an SpMV implementation in the SpMP library specialized for unweighted matrices in CSR. In this version of SpMV, all non-zeros are assumed to be 1, so we skip reading non-zero values, saving memory bandwidth.

PageRank: Figure 4f shows the performance of running 100 iterations of PageRank. The evaluated matrices are mostly social network graphs and a few road network graphs. We realize on average 50% bandwidth efficiency when all optimizations are applied (denoted as `+Reordering`). This corresponds to on average 17× speedup over `Baseline`, and 31× speedup over `Julia-as-is`. Sparso's PageRank performance is also comparable to that of manually optimized implementation reported in [54]. In `+Matrix-properties`, compiler discovers that a matrix is unweighted so that we can use a specialized SpMV in SpMP.

8. RELATED WORK

This section compares Sparso with related work in four categories: sparse linear algebra libraries, frameworks for easier programming of sparse matrix computation, data reordering, and matrix property discovery.

8.1 High-performance Sparse Linear Algebra Libraries

There are many libraries for sparse matrix computations [2, 7, 12, 13, 19, 21, 22, 24, 28, 29, 32, 33, 36, 40, 43, 59]. Basic interfaces provided by these libraries are easy to use but do not allow context-driven optimizations. As an advanced interface, libraries can expose sub-steps of context-driven optimizations for higher performance, but they are not easy to be used by average programmers. For example, a triangular solver can have a 2-step interface with one function for inspection and another for execution as in the latest version of Intel MKL library. However, effectively using such advanced interface requires expertise. This explains why MKL has not provided the inspector-executor interface until very recently [29]. Sparso optimizations enable non-expert programmers to take advantage of such advanced library features without even knowing their existence.

Another example of libraries with such advanced interface is OSKI [59] that can auto-tune sparse matrix kernels, including

[6]As explained in Section 6, the current implementation of the Sparso framework applies `+Matrix-properties` and `+Reordering` to loops only. So for PCG, the current implementation generates optimized code similar to Figure 3b, except that `+Matrix-properties` and `+Reordering` are applied only to the loop between lines 8–21, which contains the forward and backward triangular solver. This still keeps the essence of our approach as well as the performance of the application, as the loop dominates the execution time of PCG.

SpMV and sparse triangular solver. In fact, OSKI can be a good target library of Sparso because it provides various optimization opportunities like matrix storage format selection that can be taken advantage by Sparso. Sparso can complement OSKI by hiding the complexities associated with the additional parameters for auto-tuning and by providing context that spans multiple kernels and matrices.

8.2 Frameworks for Easier Programming of Sparse Matrix Computation

There have been many efforts to alleviate programming burden in sparse matrix computation, automatically or with programmer intervention [3, 5, 17, 47, 48, 56–58]. They often focus on optimizing individual sparse matrix kernels, and cannot optimize across different kernels and different matrices. In more sophisticated approaches, a compiler generates an inspector/executor and hoists the inspector out of a loop [3, 5, 17, 47, 48, 56–58].

In comparison, Sparso reasons about higher level constructs like matrices and vectors, and deduces matrix properties useful for performance optimizations using basic rules of sparse linear algebra (recall Table 2 for example). The matrix properties found are useful contextual information that can be communicated with the libraries, allowing them to follow the path optimized best for the given context (recall Table 3 for example).

8.3 Data Reordering

Given a loop accessing a matrix, either the matrix can be reordered (*data reordering*) or loop iterations can be reordered (*iteration reordering*) to achieve better spatial or temporal locality. This paper focuses on data reordering. In typical linear algebra libraries, data reordering routines find a permutation for a specific kernel invoked with a specific matrix [2, 28, 43]. As in MKL and OSKI, it is often programmers' responsibility to compose with other kernels and propagate the reordering to inter-dependent arrays. Sparso automatically handles these composability issues with collective reordering.

Iteration reordering [17, 57] is a compiler transformation that reorders the iterations of a loop accessing a sparse matrix, so as to improve temporal locality. Compared to data reordering, composability can be a lesser concern because iteration reordering often does not change the data layout that may affect other parts of the program. However, changing the data layout together sometimes results in higher performance due to better spatial locality [45].

8.4 Matrix Property Discovery

The constant size, constant value, and constant structure analyses are extensions of Kildall's classic constant propagation algorithm [31, 60], similar to the static size inference in Falcon [15] and symbolic constant propagation [10].

9. CONCLUSION AND FUTURE WORK

Sparse matrix applications are pervasive, yet difficult to optimize. We present Sparso, a new framework to speed them up by automating experts' manual context-driven optimizations. The key idea is to use context information to drive optimizations across kernels and matrices transparently, providing both performance *and* productivity. Sparso is evaluated with several representative sparse linear algebra algorithms, obtaining performance comparable to the limit bound by machine peak bandwidth for PCG and PageRank, which is hard to achieve even by highly optimized C/C++ implementations. Sparso is open-sourced at https://github.com/IntelLabs/Sparso.git.

We can extend the system to handle contexts that are not constant, but are changing slowly. We can also automate more context-driven optimizations, and port Sparso to other languages like Python and R.

Acknowledgments

We appreciate the valuable feedback of the anonymous reviewers, Jed Brown, Karl Rupp, Brian Lewis, Lindsey Kuper, Michelle Strout, Xipeng Shen, and Jiwon Seo, and the strong support from Tatiana Shpeisman, Christopher Hughes, Gilles Pokam, Rajkishore Barik, Pradeep Dubey, Paul Petersen, Youfeng Wu, and Geoff Lowney.

References

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[2] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient Management of Parallelism in Object Oriented Numerical Software Libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[3] S. Benkner. VFC: The Vienna Fortran Compiler. *Scientific Programming*, 7(1):67–81, Jan. 1999.

[4] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman. Julia: A Fast Dynamic Language for Technical Computing. *Computing Research Repository*, abs/1209.5145, 2012.

[5] T. Brandes and F. Zimmermann. Adaptor — A Transformation Tool for HPF Programs. In *Programming Environments for Massively Parallel Distributed Systems: Working Conference of the IFIP WG 10.3*, pages 91–96. Birkhäuser Basel, 1994.

[6] S. Brin and L. Page. The Anatomy of a Large-scale Hypertextual Web Search Engine. In *Proceedings of the Seventh International Conference on World Wide Web*, WWW7, pages 107–117. Elsevier Science Publishers B. V., 1998.

[7] A. Buluç and J. R. Gilbert. The Combinatorial BLAS: Design, Implementation, and Applications. *International Journal of High Performance Computing Applications*, 25(4):496–509, Nov. 2011.

[8] C.-C. Chang and C.-J. Lin. LIBSVM: A Library for Support Vector Machines. *ACM Transactions on Intelligent Systems and Technology*, 2(3):27:1–27:27, May 2011.

[9] Y. Chen, T. A. Davis, W. W. Hager, and S. Rajamanickam. Algorithm 887: CHOLMOD, Supernodal Sparse Cholesky Factorization and Update/Downdate. *ACM Transactions on Mathematical Software (TOMS)*, 35(3):22:1–22:14, Oct. 2008.

[10] C. P. Colby. *Semantics-based Program Analysis via Symbolic Composition of Transfer Relations*. PhD thesis, Carnegie Mellon University, 1996. AAI9813822.

[11] E. Cuthill and J. McKee. Reducing the Bandwidth of Sparse Symmetric Matrices. In *Proceedings of the 1969 24th National Conference*, pages 157–172. ACM, 1969.

[12] S. Dalton, N. Bell, L. Olson, and M. Garland. Cusp: Generic Parallel Algorithms for Sparse Matrix and Graph Computations. http://cusplibrary.github.io, 2014. Version 0.5.0.

[13] T. A. Davis. *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2)*. Society for Industrial and Applied Mathematics, 2006.

[14] T. A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1:1–1:25, Dec. 2011.

[15] L. De Rose and D. Padua. Techniques for the Translation of MATLAB Programs into Fortran 90. *ACM Transactions on Programming Languages and Systems*, 21(2):286–323, Mar. 1999.

[16] L. A. De Rose. *Compiler Techniques for Matlab Programs*. PhD thesis, University of Illinois at Urbana-Champaign, 1996. AAI9712420.

[17] C. Ding and K. Kennedy. Improving Cache Performance in Dynamic Applications Through Data and Computation Reorganization at Run Time. In *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 229–241, 1999.

[18] J. Dongarra and M. A. Heroux. Toward a New Metric for Ranking High Performance Computing Systems. Technical Report 4744, Sandia National Laboratories, 2013.

[19] J. Dongarra, A. Lumsdaine, X. Niu, R. Pozo, and K. Remington. LAPACK Working Note 74: A Sparse Matrix Library in C++ for High Performance Architectures. Technical Report CS-94-236, University of Tennessee, 1994.

[20] I. S. Duff. On Algorithms for Obtaining a Maximum Transversal. *ACM Transactions on Mathematical Software (TOMS)*, 7(3):315–330, 1981.

[21] I. S. Duff, M. A. Heroux, and R. Pozo. An Overview of the Sparse Basic Linear Algebra Subprograms: The New Standard from the BLAS Technical Forum. *ACM Transactions on Mathematical Software (TOMS)*, 28(2):239–267, June 2002.

[22] R. D. Falgout, J. E. Jones, and U. M. Yang. Pursuing Scalability for Hypre's Conceptual Interfaces. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):326–350, Sept. 2005.

[23] E. Farhi, J. Goldstone, S. Gutmann, and M. Sipser. Quantum Computation by Adiabatic Evolution. Technical Report MIT-CTP-2936, Massachusetts Institute of Technology, 2000.

[24] S. Filippone and M. Colajanni. PSBLAS: A Library for Parallel Linear Algebra Computation on Sparse Matrices. *ACM Transactions on Mathematical Software (TOMS)*, 26(4):527–550, 2000.

[25] J. R. Gilbert, C. Moler, and R. Schreiber. Sparse Matrices in MATLAB: Design and Implementation. *SIAM Journal on Matrix Analysis and Applications*, 13(1):333–356, 1992.

[26] W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith. Towards Realistic Performance Bounds for Implicit CFD Codes. In *Parallel Computational Fluid Dynamics: towards Teraflops, Optimization, and Novel Formulations*, pages 241–248. Elsevier, 1999.

[27] V. E. Henson and U. M. Yang. BoomerAMG: a Parallel Algebraic Multigrid Solver and Preconditioner. *Applied Numerical Mathematics*, 41:155–177, 2000.

[28] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley. An Overview of the Trilinos Project. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):397–423, Sept. 2005.

[29] Intel® . Math Kernel Library Inspector-executor Sparse BLAS Routines. https://software.intel.com/sites/default/files/managed/0a/81/Documentation_inspector_executor_sparse_blas_mkl113b.pdf, 2015.

[30] Intel® . MKL PARDISO - Parallel Direct Sparse Solver Interface. https://software.intel.com/en-us/node/470282, 2015.

[31] G. A. Kildall. A Unified Approach to Global Program Optimization. In *Proceedings of ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 194–206, 1973.

[32] A. R. Krommer. Parallel Sparse Matrix Computations in the Industrial Strength PINEAPL Library. In *Proceedings of the 4th International Workshop on Applied Parallel Computing, Large Scale Scientific and Industrial Problems*, pages 281–285. Springer-Verlag, 1998.

[33] X. S. Li. An Overview of SuperLU: Algorithms, Implementation, and User Interface. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):302–325, 2005.

[34] M. Lichman. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml, 2013.

[35] D. C. Liu and J. Nocedal. On the Limited Memory BFGS Method for Large Scale Optimization. *Mathematical Programming*, 45(1), 1989.

[36] A. Lugowski, A. Buluc, J. Gilbert, and S. Reinhardt. Scalable complex graph analysis with the knowledge discovery toolbox. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5345–5348, March 2012.

[37] I. J. Lustig and E. Rothberg. Gigaflops in Linear Programming. *Operations Research Letters*, 18(4):157–165, Feb. 1996.

[38] J. D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, Dec. 1995.

[39] R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nico, and K. Crowley. Principles of Runtime Support for Parallel Processors. In *Proceedings of the International Conference on Supercomputing (ICS)*, pages 140–152. ACM, 1988.

[40] M. Naumov. Incomplete-LU and Cholesky Preconditioned Iterative Methods Using cuSPARSE and cuBLAS. https://developer.nvidia.com/cusparse.

[41] A. M. Niklasson. Expansion Algorithm for the Density Matrix. *Physical Review B*, 66(15):155115, Oct. 2002.

[42] P. K. S. Norman E. Gibbs, William G. Poole. An Algorithm for Reducing the Bandwidth and Profile of a Sparse Matrix. *SIAM Journal on Numerical Analysis*, 13(2):236–250, 1976.

[43] J. Park. Sparse Matrix Preprocessing Library. https://github.com/IntelLabs/SpMP.

[44] J. Park, M. Smelyanskiy, N. Sundaram, and P. Dubey. Sparsifying Synchronization for High-Performance Shared-Memory Sparse Triangular Solver. In *Proceedings of International Supercomputing Conference (ISC)*, pages 124–140. Springer International Publishing, 2014.

[45] 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 *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 945–955, 2014.

[46] M. M. A. Patwary, N. R. Satish, N. Sundaram, J. Park, M. J. Anderson, S. G. Vadlamudi, D. Das, S. G. Pudov, V. O. Pirogov, and P. Dubey. Parallel Efficient Sparse Matrix-Matrix Multiplication on Multicore Platforms. In *Proceedings of International Supercomputing Conference (ISC)*, pages 48–57. Springer International Publishing, 2015.

[47] R. Ponnusamy, J. Saltz, and A. Choudhary. Runtime Compilation Techniques for Data Partitioning and Communication Schedule Reuse. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 361–370. ACM, 1993.

[48] W. Pugh and T. Shpeisman. SIPR: A New Framework for Generating Efficient Code for Sparse Matrix Computations. In *Proceedings of the 11th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '98, pages 213–229. Springer-Verlag, 1999.

[49] L. Rauchwerger. Run-time Parallelization: Its Time Has Come. *Parallel Computing - Special issues on languages and compilers for parallel computers*, 24(3-4):527–556, May 1998.

[50] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, pages 693–701. Curran Associates, Inc., 2011.

[51] J. K. Reid and J. A. Scott. Reducing the Total Bandwidth of a Sparse Unsymmetric Matrix. *SIAM Journal on Matrix Analysis and Applications*, 28(3):805–821, Aug. 2006.

[52] Y. Saad. *Numerical Method for Large Eigenvalue Problems*. Society for Industrial and Applied Mathematics, 2011.

[53] Y. Saad and M. H. Schultz. GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems. *SIAM Journal on Scientific and Statistical Computing*, 7(3):856–869, July 1986.

[54] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey. Navigating the Maze of Graph Analytics Frameworks Using Massive Graph Datasets. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 979–990. ACM, 2014.

[55] M. Smelyanskiy, V. W. Lee, D. Kim, A. D. Nguyen, and P. Dubey. Scaling Performance of Interior-point Method on Large-scale Chip Multiprocessor System. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 22:1–22:11. ACM, 2007.

[56] M. M. Strout, G. Georg, and C. Olschanowsky. Set and Relation Manipulation for the Sparse Polyhedral Framework. In *Proceedings of Languages and Compilers for Parallel Computing: 25th International Workshop, LCPC, Tokyo, Japan, September 11-13, Revised Selected Papers*, pages 61–75. Springer Berlin Heidelberg, 2013.

[57] M. M. Strout, A. LaMielle, L. Carter, J. Ferrante, B. Kreaseck, and C. Olschanowsky. An Approach for Code Generation in the Sparse Polyhedral Framework. Technical Report CS-13-109, Colorado State University, December 2013.

[58] A. Venkat, M. Hall, and M. Strout. Loop and Data Transformations for Sparse Matrix Code. In *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 521–532, 2015.

[59] R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A Library of Automatically Tuned Sparse Matrix Kernels. *Journal of Physics: Conference Series*, 16(1):521, 2005.

[60] M. N. Wegman and F. K. Zadeck. Constant Propagation with Conditional Branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, Apr. 1991.

[61] S. Williams, A. Waterman, and D. Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Communications of the ACM*, 52(4):65–76, Apr. 2009.

A. APPENDIX

A.1 Distributivity and Inter-dependence

By definition, reordering R is distributive over any function whose inputs and output are all scalars, such as `if(condition)` or `goto(label)`, treating both as accepting a scalar input and returning the same scalar as output. Similar conclusion can be extended to higher-level language constructs like loops (`for`, `while`, etc.).

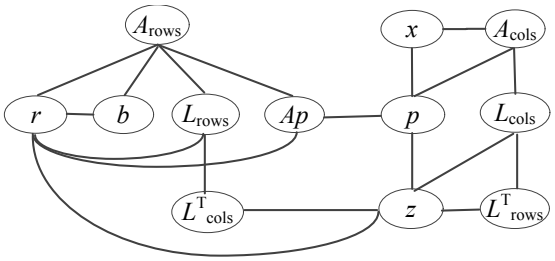
However, there are functions over which reordering is not distributive:

Case 1. Array element accesses: Reordering changes an array, but not an array index, which is a scalar. An array access $A[i,j]$ or $x[i]$, after A or x being reordered, has to be replaced with $A[i',j']$ or $x[i']$. From Equation 1 and 2, we can derive that i' and j' are such that $P_A[i',i]=1$ and $Q_A[j',j]=1$ for matrix A, and similarly, $P_x[i',i]=1$ for vector x.

Case 2. A function whose inputs are scalars but result is an array: An array creation function whose input is a size is such an example. Reordering the size has no effect, but reordering the output changes the result. Reordering is distributive over the function only when the created array is considered the same with and without reordering, i.e. reordering has

Expression	Inter-dependences	
L=ichol(A)	L_{rows}—A_{rows}	L_{cols}—A_{cols}
$r=b-A\cdot x$	r—b	r—A_{rows}
	b—A_{rows}	A_{cols}—x
$z=L\backslash r$	L_{rows}—r	L_{cols}—z
$L^\intercal$	L_{rows}—$L^\intercal_{\text{cols}}$	L_{cols}—$L^\intercal_{\text{rows}}$
$z=L^\intercal\backslash z$	$L^\intercal_{\text{rows}}$—$z$	$L^\intercal_{\text{cols}}$—$z$
$p=copy(z)$	p—z	
$r^\intercal\cdot z$	r—z	
$Ap=A\cdot p$	Ap—A_{rows}	A_{cols}—p
$p^\intercal\cdot Ap$	p—Ap	
$x=x+\alpha\cdot p$	x—p	
$r=r-\alpha\cdot Ap$	r—Ap	
$p=z+(rz/old\_rz)\cdot p$	p—z	

(a) Inter-dependences for all the expressions.



(b) Inter-dependence graph.

Figure 5: Building inter-dependences for the PCG example in Figure 3a.

no impact on the created array. This happens, for example, when the array is initialized to 0 or uninitialized.

By the contraposition of the logic in the case 2 above, we reach the following conclusion:

FACT A.1 (DISTRIBUTIVITY AND INTER-DEPENDENCE). *If reordering is distributive over a function that returns an array, then (1) the function must have at least one input that is an array and is inter-dependent with the returned array, or (2) reordering has no impact on the returned array.*

A.2 Identifying Inter-dependent Arrays

For basic operations, meta-data are required to describe the inter-dependences between their operands. Based on that, any user code can be analyzed for inter-dependent arrays: visit every expression of the code and build up the inter-dependences between the operands.

For example, consider the expression $r=b-A\cdot x$ of PCG code in Figure 3a. With the meta-data of the basic operation =, we know r's rows must be permuted in the same way as the rows of the right hand side $b-A\cdot x$. Let us denote the inter-dependence as

$$r \quad \text{———} \quad (b-A\cdot x)$$

For the same reason, we can find the following inter-dependences:

$$\begin{aligned} (b-A\cdot x) &\quad \text{———} \quad b \\ (b-A\cdot x) &\quad \text{———} \quad (A\cdot x) \\ (A\cdot x) &\quad \text{———} \quad A_{\text{rows}} \\ A_{\text{cols}} &\quad \text{———} \quad x \end{aligned}$$

where A_{rows} and A_{cols} refer to the rows and columns of A, respectively. Since inter-dependence is transitive, we can remove the intermediate expressions like $(b-A\cdot x)$ from the above relations, and get the following inter-dependences between operands:

$$\begin{aligned} r &\quad \text{———} \quad b \\ r &\quad \text{———} \quad A_{\text{rows}} \\ b &\quad \text{———} \quad A_{\text{rows}} \\ A_{\text{cols}} &\quad \text{———} \quad x \end{aligned}$$

Figure 5a shows all the inter-dependences for the PCG example.

With them, an *inter-dependence graph* can be built (see Figure 5b), where a vertex represents a vector, a matrix's rows, or a matrix's columns, and an edge represents an inter-dependence between two vertices. All vertices in the same connected component in such a graph are inter-dependent with each other, and thus have the same permutation matrix. For this specific graph, there is only one connected component. Therefore, all vertices have the same permutation matrix. For example, both A_{rows} and A_{cols} are in the component, which means A is reordered by:

$$R(A)=P_A\cdot A\cdot P_A^\intercal \tag{6}$$

This is exactly how a square matrix is usually reordered, even though our process never assumes A is square.

Note that the concrete content of a permutation matrix like P_A here is not determined by the compiler, but by the decider of reordering (a wrapper routine) at run-time. The compiler only identifies *which* arrays are inter-dependent and thus should be reordered with the same permutation matrix. The decider, when executed at run-time, constructs the concrete permutation matrix, uses the permutation matrix to reorder its input arrays, and memoizes the permutation matrix in RCT. RCT is then used by `reorder()` and `inversely_reorder()` (see Figure 3b) to reorder and inversely reorder other inter-dependent arrays.

Although this example is for PCG, which involves only square matrices, the same process can be applied to any code and rectangular matrices. For example, Sparso automatically reorders rectangular matrices in L-BFGS. The worst case complexity of the process is $O(V^2)$, where V is number of arrays in the user program.

A.3 A Proof of Theorem 4.1

Perform Actions 1 and 2 listed in Theorem 4.1 and prove Action 3 is necessary and sufficient, no matter what path the code has followed in the execution. Note that the code may contain arbitrary conditional branches, and there may be exponential number of possible execution paths.

Assume array Γ is in set S and is live out at an exit. We prove that after Actions 1 and 2, Γ at that exit must equal $R(\Gamma_0)$, where Γ_0 refers to the Γ at the same exit in the original code (with no reordering).

Case 1 Γ has never been defined during the execution of the code. Then, Γ must have been live into the code at the entry that starts the execution, reordered per Action 1.

Case 2 Γ has been defined during the execution, but reordering has no impact on it. Then, it can be considered as reordered.

Case 3 Otherwise, Γ has been defined by an expression $\Gamma = f(\Lambda,...)$ during the execution, where Λ is an array and is inter-dependent with Γ, per Fact A.1. Apply similar reasoning to Λ. Eventually, we must arrive at either Case 1 or 2. □

B. ARTIFACT DESCRIPTION

B.1 Abstract

The artifact contains the latest version of Sparso on github. It automatically downloads the public data we have used, and produces the results data used for Figure 4.

B.2 Description

1. **Algorithm:** PCG, IPM, L-BFGS, CoSP2, Adiabatic, PageRank
2. **Program:** Julia code
3. **Compilation:** Julia 0.4.6
4. **Transformations:** call replacement, context-sensitive optimizations, and collective reordering
5. **Data set:** Public available matrix market files
6. **Run-time environment:** Ubuntu 16.04 x86-64 desktop
7. **Hardware:** 14-core Xeon E5-2697 v3 @ 2.6 GHz, 35MB on-chip L3, 64 GB DDR

8. **Output:** Execution time, GB/s, and/or GFLOP/s
9. **Experiment workflow:** Install Sparso and related libraries; run the test scripts; observe the results
10. **Publicly available?:** Yes

B.2.1 How delivered

Sparso is an open source package to the Julia language and is hosted with code and build instructions on Github.

B.2.2 Hardware dependencies

A modern x86 machine with more than 10 cores is preferred to run Sparso for performance testing.

B.2.3 Software dependencies

Sparso has been tested on Ubuntu 16.04, although it is a source-level package to Julia and thus is independent of OS and should be runnable on any OS where Julia can run.

Sparso requires MKL and SpMP libraries.

Scripts for validating correctness and performance are provided. For correctness validation, pcregrep is required.

B.2.4 Datasets

A script is provided to automatically download all the datasets that are publically available and in matrix market format. Users are welcome to try other datasets. We currently only support matrix market format files as input.

B.3 Installation

Follow the installation instruction on Sparso's github page (https://github.com/IntelLabs/Sparso.git).

B.4 Experiment workflow

Follow the experiment instruction on Sparso's github page (https://github.com/IntelLabs/Sparso.git). It allows users to validate both correctness and performance.

B.5 Evaluation and expected result

Correctness-wise, a set of tests will be run and should all pass. Log files will be produced, including the original Julia code and the code after being optimized by Sparso.

Performance-wise, expected results include GB/s for PCG and PageRank, execution time for IPM, L-BFGS, and Adiabatic, and GFLOP/s for CoSP2.

B.6 Notes

To know more about Sparso, send feedback, or file issues, please visit our github page (https://github.com/IntelLabs/Sparso.git).