Automatic Code Generation for High-Performance Graph Algorithms

Zhen Peng[*], Rizwan A. Ashraf[*], Luanzheng Guo[*], Ruiqin Tian[†][§] and Gokcen Kestor[*][‡]
[*] *Pacific Northwest National Laboratory*, Richland, Washington, USA
[†] *Horizon Robotics*, Shanghai, China
[‡] *University of California, Merced*, Merced, California, USA.
Emails: {zhen.peng, rizwan.ashraf, lenny.guo, gokcen.kestor}@pnnl.gov, ruiqin.tian@horizon.ai

***Abstract*—Graph problems are common across many fields, from scientific computing to social sciences. Despite their importance and the attention received, implementing graph algorithms effectively on modern computing systems remains a challenging task that requires significant programming effort and generally results in customized implementations. Current computing and memory hierarchies are not architected for irregular computations, resulting performance that is far from the theoretical architectural peak. In this paper, we propose a compiler framework to simplify the development of graph algorihtm implementations that can achieve high performance on modern computing systems. We provide a high-level domain specific language (DSL) to represent graph algorithms through sparse linear algebra expressions and graph primitives including semiring and masking. The compiler leverages the semantics information expressed through the DSL during the optimization and code transformation passes, resulting in more efficient IR passed to the compiler backend. In particular, we introduce an Index Tree Dialect that preserves the semantic information of the graph algorithm to perform high-level, domain-specific optimizations, including workspace transformation, two-phase computation, and automatic parallelization. We demonstrate that this work outperforms state-of-the-art graph libraries LAGraph by up to $3.7\times$ speedup in semiring operations, $2.19\times$ speedup in an important sparse computational kernel, and $9.05\times$ speedup in graph processing algorithms.**

***Index Terms*—compiler, optimization, graph algorithms, triangle counting, breadth-first search, code generation, semiring, masking, sparse linear algebra**

I. Introduction

Graphs data structures are used in many domains, from computer security [1] and computational genomics [2] to network sciences [3] and scientific computing [4], to represent complex interactions and casual relationships (edges) among entities (nodes). While graphs adapt well to solve problems at different scales, real-life problems often produce graph data structures that are highly irregular and extremely large. These two factors pose challenges while implementing efficient graph algorithms on modern computer architectures, which have been developed and optimized mostly for regular computation. To achieve high-performance, developers are often forced to write ad-hoc code specifically tailored for given architectures using a fairly low-level programming language, such as C/C++ or CUDA, which, then, impedes the portability of the implementation on different systems, productivity, and reusability. With the proliferation of modern computing systems, the current practice of manually reimplementing graph algorithms for each new architecture is simply not sustainable.

In this work, we seek a solution to develop graph algorithms that provide high performance on modern computer architectures but do not hinder portability and productivity. In this endover, we identify two main challenges: 1) find the right level of abstraction to represent graph algorithms and 2) lower that abstraction to efficient machine code. The level of abstraction should be high enough to enable developers to express graph algorithms effectively and with notations that make sense in the application domain, both of which increase productivity. Hand-tuned, architecture-specific implementations (e.g., CUDA) may achieve high performance but developing such solutions is time-consuming and not portable across systems. The abstraction should carry sufficient semantics information to be used during code optimizations and machine code generation to increase performance on specific architectures. Finally, the abstraction should be architecture-independent and semantically rich to guarantee portability across different systems. In fact, it is generally easier to port high-level operations (e.g., sparse matrix-sparse matrix multiplication) than low-level constructs (e.g., nested loops) across systems. In this work, we opt for (sparse) linear algebra as a reasonable programming abstraction to develop efficient graph algorithms. Algebraic representations of graph algorithms are architecture-independent, sufficiently high-level so that users can effectively implement graph applications in their domains, and carry enough semantics information to inform the underlying system about which architecture-independent and architecture-specific optimizations should be employed. Compared to vertex-based and edge-traversal implementations, algebraic representations provide a compact and expressive way to represent graph algorithms, carrying semantic information through the characteristics of the matrix used to represent the graph [5], [6], are easier to develop, more portable, and can leverage a large body of research and optimization.

The second challenge is represented by mapping (lowering) the high-level abstraction to efficient code for specific computing systems. The inherent irregularity of graph processing

[§] Work was performed while the author was at Pacific Northwest National Laboratory algorithms and the size of real-life graphs pose considerable challenges when performing this process. The sheer size of real-life problems makes it difficult, if not impossible, to store graphs (i.e., adjacency matrix) using dense data structures. Given the intrinsic sparse nature of graph structures, storing graphs in dense format would introduce excessive pressure on the memory subsystem and unnecessary computation. Efficient graph implementations generally prefer sparse representations of the graph to reduce memory requirements and use sparse operators to increase computing efficiency by eliminating unnecessary computation. However, modern computing architectures and memory technologies have been designed and optimized for dense computation and do not perform as well for sparse computation [7]. The process of lowering high-level abstraction to efficient machine code must employ different kinds of optimizations, both architecture-independent and architecture-specific, and should be performed at all levels of the lowering process. First, the language should provide high-level, graph-oriented operators that carry enough information for efficient code generation. Second, architecture-independent, graph-specific optimizations, such as fusion and automatic parallelization, should be applied to the high-level code. Next, generic architecture-independent optimizations (loop unrollling, dead code elimination, etc.) should be considered. Finally, the resulting code should be optimized for the target architecture. This process should be automated to increase productivity and portability and, to the extend that the abstraction carries enough semantics information, should have the user out of the loop.

In this work, we propose a domain-specific compiler framework to develop graph algorithm implementations that can achieve high performance, are portable across different systems, and are easy to develop. We propose a high-level Domain-Specific Language (DSL) to represent graph algorithms through (sparse) linear algebra expressions and specific graph-oriented operators. The DSL allows users to embed domain-specific semantics that is leveraged internally during code generation through a series of optimizations and lowering steps to generate efficient Intermediate Representation (IR), such as specific graph primitives including semiring and masking. The proposed compiler is based on a multi-level IR and progressive lowering from high-level IRs (or dialects) that encapsulate the semantics of the application to low-level IRs, which are closer to the architecture. The compiler leverages the semantics information expressed through the DSL during the optimization and code transformation passes. This generally results in more efficient IR that can be passed to the compiler backend (e.g., LLVM) to generate machine code compared to general-purpose programming environments, such as C or C++. In particular, we introduce an Index Tree Dialect. This dialect preserves the semantic information of the graph algorithm to perform high-level, domain-specific optimizations. Several code optimizations and transformations are applied while lowering the index tree IR to lower-level dialects in the compilation pipeline, including optimizations specifically developed in this work: workspace transformation, two-phase computation, and automatic parallelization. Workspace transformation takes advantage of intermediate dense structure to improve the data locality and reduce computation complexity while preserving the sparse format of the resulting outputs. The two-phase computation employs symbolic computation to deduce the minimum size for the output's sparse data structure. We also introduce a novel optimization algorithm that leverages the symbolic information to perform automatic parallelization of sparse linear algebra primitives.

We show that by combining our DSL, optimizations (workspace transformation, two-phase computation, and parallelization), and efficient graph primitives (semiring and masking), we are able to outperform state-of-the-art graph libraries (e.g., LAGraph [6], which implements the GraphBLAS standard [8]) by a significant margin. We evaluate the performance of several graph primitives and graph processing algorithms. Our results show that our work outperforms LAGraph by up to $3.7\times$ for semiring operations, $2.19\times$ for SpGEMM kernel, and $9.05\times$ for graph processing algorithms Breadth First Search (BFS) and Triangle Counting (TC). In summary, this work makes the following contributions:

- a novel compiler framework and DSL, which enable users to productively develop the algebraic implementation of graph algorithms and achieve high-performance.
- important graph primitives (semirings and masking operations) and code optimizations and transformations (workspace transformation, two-phase computation, parallelization) for efficient execution;
- a performance evaluation of sparse linear algebra kernels and two prominent graph processing algorithms and comparison with LAGraph.

The rest of this work is organized as follows: Section II provides an overview of the compiler; Section III introduces the code generation optimizations of graph computations; Section IV demonstrates extended linear algebra primitives for graph algorithms; Section V provides an exhaustive performance evaluation; finally, Section VI and Section VII compare this work to other efforts and draw conclusions, respectively.

II. Overview

This work proposes a domain-specific compiler framework to develop efficient graph algorithms represented by linear algebra operations. Our work adheres to the GraphBLAS standard, which provides a comprehensive set of graph primitives for sparse matrices and vectors of various types and extends the traditional linear algebra operators with semirings and masking to achieve higher performance. The GraphBLAS-based approach provides a consistent way for graph algorithm implementation through common graph primitives that can be optimized using well-studied techniques, and it avoids the complexity of writing different ad-hoc implementations common with traditional vertex- or edge-centric approaches [9]–[11]. In most cases, the algorithmic complexity of the graph algorithms implemented using linear algebra is close to the complexity of the implementation based on vertex- or edge-transverses [12]. Given a graph $G(V, E)$, where V is the set of

N vertices (or nodes) in the graph and E is the set of M edges that connect two vertexes, such that $e_{ij} \in E$ iff there are two vertexes $v_i, v_j \in V$ and there exists an edge between the two. While graphs can be represented in various forms, such as a list of edges and the vertices they connect, a graph $G(V, E)$ in algebraic implementations of graph algorithms is represented as an adjacency matrix A of size NxN, where the elements $a_{ij} = 1$ iff there exists an edge $e_{ij} \in E$. Once a graph is represented as an adjacency matrix, the implementation of graph algorithms can leverage well-defined linear algebra operations. For example, visiting all neighbors from a source vertex v_i only requires multiplying the adjacency matrix A by a vector x that has all zero elements except x_i. Similarly, multiplying A by itself k times yields relations between vertices at distance k. For example, A^2x and A^kx return the neighbor lists that are two and k hops away, respectively. Given the nature of graphs, generally $N^2 \gg M$, hence A is very sparse. Storing A in a dense format unnecessarily increases the memory and computing requirements and results in inefficient execution. This framework employs sparse linear algebra operators to reduce both the memory and computing requirements, as only the non-zero elements of A need to be stored in memory and considered during the computation.

The proposed DSL is based on index notation (or Einstein notation), which is a concise, expressive, and widely used way to express dense and sparse tensor computations. For example, the multiplication of two matrices A and B can be expressed as $C_{ij} = A_{ik} * B_{kj}$, where i and j are the free indices that appear in the output, whereas, the remaining indices are the summation indices, k in this case. This concisely represents the following operation for each entry of the output matrix C: $\sum_{k=1}^{N} a_{ik}b_{kj}$, assuming that all matrices are of size NxN. The Einstein notation is adopted and supported in many common programming models to express tensor operations, such as the `numpy.einsum` API in NumPy [13], PyTorch (`torch.einsum`) and TensorFlow (`tf.einsum`). It is also the input language in deep learning frameworks such as Tensor Comprehension [14], and sparse tensor compilers such as TACO [15] and COMET [7], [16]. All these libraries and frameworks implement some variant of the original Einstein notation to expand the expressiveness. In this work, we adopt a consistent Einstein notation semantic as used by `numpy.einsum` and state-of-the-art compilers [7], [15]. We refer the reader to the `numpy.einsum` API page for a more comprehensive description of the notation. Note that a summation or contraction index is implied if an index variable appears on the right-hand-side tensors but not on the left-hand-side tensor. In addition, a custom function can be specified to express other types of reduction other than summation, such as $A_i = max(B_{ij})$. It is also possible to express operations such as MTTKRP (Matricized Tensor Times Khatri-Rao Product) as $A_{ir} = B_{ijk} * D_{jr} * C_{kr}$, where the index variable j and k are summed. The sparse formats of each tensor are specified as type annotations, and a compiler will automatically generate code from the expression in the back end. Also, note that for some operation sequences, the order of evaluation can result in different asymptotic time complexities, such as a chain of matrix multiplications [7]. In this work, we assume such order is already determined and do not attempt to reorder matrix multiplications.

From an implementation point of view, the proposed compiler is based on the Multi-Level Intermediate Representation (MLIR) framework and built on top of COMET [7], [16]. COMET is a dense and sparse tensor algebra compiler that targets multiple architectures. It has been extensively used to optimize dense tensor contractions within the NWChem quantum chemistry framework [7], [17]. The work introduces specific code optimizations and transformations for sparse linear algebra operators, and DSL support to implement algebraic formulations of graph algorithms. MLIR, which is part of the LLVM ecosystem, provides a solid foundation to build new compiler frameworks and a set of common optimizations and code transformation passes, such as loop unrolling, tiling, and vectorization. New optimizations and architectures added to the MLIR framework will be readily available to the proposed compiler and its users.

III. Efficient Code Generation of Graph Computation

The proposed compiler is based on the MLIR framework, which provides a multi-level IR and an infrastructure to perform progress lowering. The key insight in MLIR, hence in this work, is that different optimizations can be performed at each level of the IR stack (or dialects), from high-level, domain-specific optimizations at higher levels to architecture-specific optimizations at low levels and that optimizations and dialects can be composed to efficiently generate executable code for various target architectures.

A. Index Tree Dialect

This work introduces a new MLIR dialect, the *Index Tree dialect*, between the COMET's existing *tensor algebra dialect* and the MLIR Structured Control Flow (SCF) dialect with the specific objective of performing efficient code transformation and optimizations for sparse computation. The index tree dialect is a representation of an index tree notation, which is widely adopted by various code generation models of tensor computations [15], [18], [19]. While the COMET tensor algebra dialect is designed for traditional tensor algebra operators, the index tree dialect provides a generic and efficient representation of computing expressions (operators and their relations) based on two types of nodes, indexation, and computation nodes. The specific instantiations of the nodes are determined by the computation expressed in the tensor algebra dialect during lowering. However, the index tree nodes are not limited to the traditional tensor algebra operators and can be used to implement extended operators, such as semiring and masking, as described in Section IV. As discussed later, these extended operators are fundamental to achieving high performance with graph algorithms.

Listing 1 and Listing 2 show examples of code (matrix multiplication) represented in the tensor algebra dialect (`ta.mul`)

```
%C = "ta.mul"(%A, %B) {
  formats = ["CSR", "CSR", "CSR"],
  indexing_maps = [
    affine_map<(d0, d1, d2) -> (d0, d1)>,  //C[i,k]
    affine_map<(d0, d1, d2) -> (d1, d2)>,  //A[i,j]
    affine_map<(d0, d1, d2) -> (d0, d2)>], //B[j,k]
  semiring = "plus_times"} //matrix multiplication
  : (tensor<?x?xf64>, tensor<?x?xf64>, !ta.range, !ta.
      range) -> tensor<?x?xf64>
```

Listing 1: The sparse matrix-matrix multiplication operation represented in the tensor algebra dialect. Multi-dimensional memory references are indexed with affine maps.

```
%1 = "it.computeRHS"(%a, %b) ...{} -> tensor<*xf64>
%2 = "it.computeLHS"(%c) ...{} -> tensor<*xf64>
%3 = "it.compute"(%1, %2) {semiring = "plus_times"}
%4 = "it.indices"(%3) ...{}
%5 = "it.indices"(%4) ...{}
%6 = "it.indices"(%5) ...{}
%7 = "it.tree"(%6) ...{}
```

Listing 2: The sparse matrix-matrix multiplication operation represented in the index tree dialect highlighting the tree structure with index and compute nodes.

and lowered to index tree dialect, respectively. The code in Listing 2 can be read from bottom to top, i.e., `it.tree` operation represents the root of the index tree. The `it.indices` operations represent the various indices used in the tensor multiplication operation, which would be i, j, k. These nodes of the tree will directly map to three nested loops needed for this operation. The next three operations in the index tree dialect represent the body of the loop. In this case, `compute` operation forms a segue between index operations and the compute operations. The `compute` operation maintains the compute expression that is formed via the `computeLHS`, `computeRHS` operations. Once the lowering to SCF dialect is complete, the code is ready to be consumed within the downstream MLIR infrastructure to generate machine code.

B. *Code Optimizations and Transformations*

To provide efficient code generation for sparse kernels, this work addresses three major challenges during the code generation: 1) high insertion cost into sparse output tensors 2) the unknown size and distribution of nonzero elements in sparse output tensors, and 3) the difficulty of parallelization.

1) Workspace Transformation: In general, inserting a new element into the sparse data structure of the output tensor has a high time complexity. To eliminate this complexity, most compiler frameworks and libraries store the output of a sparse computation in a dense format. Although this approach greatly reduces the cost of computation, it may lead to "densification" of the data structures and increased memory footprints for the output results, which may result in unnecessary wasted space and an inability to execute.

To address these challenges, we developed a novel approach called *workspace transformation* to store sparse output directly in sparse formats such as CSR. The workspace transformation introduces temporary intermediate dense data structures (called the *workspace*) to avoid irregular access to the original sparse data structures. This not only simplifies the code generation algorithm, but also improves the data locality of the generated code by narrowing some irregular access to the dense data structure. Unlike previous work that [20] requires users to determine the index to apply workspace transformation, in this work, the compiler automatically identifies the index involved based on the storage format.

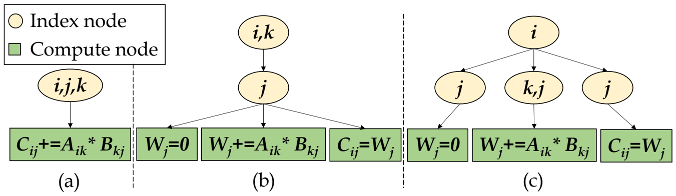


Fig. 1: The index trees (a) before and (b,c) after applying the workspace transformation on C in index j, given a matrix multiplication operation.

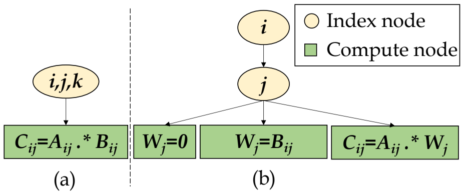


Fig. 2: The index trees (a) before and (b) after applying the workspace transformation on B in index j, given an element-wise multiplication operation.

The workspace transformation can be applied to two types of indices in a sparse-sparse expression: *input indices* associated with sparse dimensions in both input tensors and *output indices* associated with sparse dimensions in the output tensor. The workspace transformation performed depends on the indices to which it is applied.

Consider two sparse matrices A_{ik} and B_{kj} stored in the CSR format. The expression $C_{ij} = A_{ik} * B_{kj}$ produces a sparse output C_{ij}, where the index j is associated with a sparse dimension. In this case, the compiler applies the workspace transformation to the index j as *output index*. The index trees before and after workspace transformation are shown in Figure 1: Figure 1(a) shows the original index tree for a pure sparse matrix multiplication operation; Figure 1(b) shows the index tree after applying the workspace to C on index j. The data in dimension j of C is represented with a temporary dense vector W; Figure 1(c) shows the index tree after enabling loop-invariant optimizations, in which unused indices of the leaf nodes are removed. In particular, the temporary dense vector W is independent of the index k, so k can be safely removed. Before workspace transformation, the time complexity of random access to the sparse output index j is $O(\log n)$ from searching (assuming the indices are sorted), and the insertion complexity is $O(n)$ from data movement. After workspace transformation substitutes the sparse index j with the dense vector W, the complexity of both random access and insertion is reduced to $O(1)$. Therefore, the workspace transformation can provide asymptotic performance improvement for the sparse outer index.

Now consider the element-wise expression $C_{ij} = A_{ij} .{*} B_{ij}$, where index j is associated with a sparse dimension in both the input matrices A and B. In this case, we apply the workspace transformation on the input matrix B. The index trees before and after workspace transformation are shown in Figure 2, where the data in dimension j of B are temporarily preserved by a dense vector W. Before workspace transformation, iterating the sparse input index j on both the sparse matrices A and B simultaneously requires a merge `while` loop with compound conditionals. After workspace transformation, the linear traverse of index j in matrix B can be replaced by random access to the dense vector W. This can not only improve the performance asymptotically, but also substitute the merge `while` loop with a simple for loop, which enables additional optimization potentially customized for for loops.

The workspace transformation is applied on the Index Tree dialect. Compared to the kernel fusion optimization in COMET [18], which removes redundant computation and performs memory optimization for intermediate tensors created by the fusion of multiple operations, the workspace transformation increases the performance of a single operation. The workspace transformation proposed in this work and the kernel fusion [18] can potentially be composed together to result in overall higher performance. We leave this evaluation as future work.

2) Two-Phase Computation: One of the challenges of sparse computation comes from the unknown size and distribution of the output tensor. First, the number of nonzero elements in the output tensor is unknown before computation, which makes memory management very difficult. A general method is to allocate a very large chunk of memory to avoid the case where the output size exceeds the allocation, which results in redundant memory usage. Second, it is hard to know beforehand how nonzero elements are distributed among different rows (in case of row-major storage) in the output tensor. To update the output tensor in parallel, it is common to use a lock on the critical data structure, which results in high synchronization overhead.

To determine the needed size and real distribution of the output tensor, this work generates the code with two phases for sparse computation. The first phase is called the *symbolic phase*. It follows the same procedure of the given sparse computation (e.g., SpGEMM) in a "symbolic" way that it does not execute the computation, but only records the nonzero distribution of the output. After that, the symbolic phase can also determine the true number of nonzero elements in the output tensor and then allocate the sparse data structure with only the needed memory size. The second phase is called the *numeric phase*. It performs the real "numeric" computation with the prior knowledge from the symbolic phase. For some components of the sparse data structure (e.g., the index array in CSR), the output can be placed directly at the correct location that is provided by the symbolic phase. Therefore, the two-phase computation can minimize memory usage effectively.

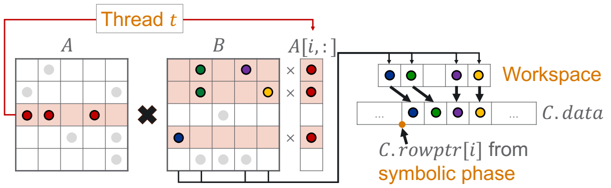


Fig. 3: An example of parallel numeric phase. After the symbolic phase, thread t knows where to insert the result into the data array using a private workspace.

3) Automatic Parallelization: Another challenge of sparse computation is the inefficient parallel execution due to the unknown distribution of the output nonzero elements. Unlike the dense matrix computation that can be parallelized by simply dividing the regular output, the sparse computation generates irregular sparse output. A naïve parallelization method would require locks on the critical sparse data structure, which may result in a high synchronization overhead.

The use of two-phase computation enables efficient parallelization without locks. First, the symbolic phase can be naturally parallelized among one dimension of the tensor. It does not have update conflicts because the symbolic phase does not perform numeric computations but only records the output distribution. Second, the numeric phase can also be parallelized according to the locations provided by the symbolic phase as shown in Figure 3. Different parts of nonzero elements can be updated simultaneously without conflicts.

Moreover, the codegen can generate a private workspace for each worker during parallel execution. In this way, multiple workers will not have writing conflicts with each other, and a worker can reuse their private workspace without unnecessary reallocation.

IV. Extended Linear Algebra Graph Primitives

As described in Section II, traditional linear algebra primitives are not sufficient to implement efficient graph algorithms. For this reason, the GraphBLAS standard introduces extended linear algebra operators that leverage the nature of graph data structure to reduce computation and increase performance. This section describes the design and implementation of such extended primitives in our work.

A. Semiring

Semiring is an algebraic structure similar to a ring, but without the requirement that each element must have an additive inverse operator. Semiring operators combine a pair of linear algebra operations into a singular one. For example, the DSL expression

$$C[i,k] = A[i,j] \ @(+,*) \ B[j,k] \tag{1}$$

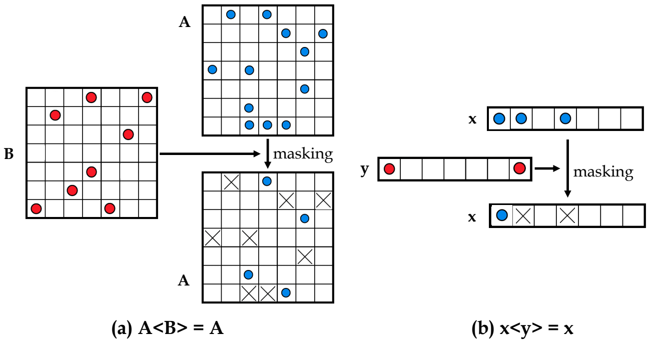
where A and B are the input matrices and C is the output matrix, contains the semiring operator $@(\cdot,\cdot)$. The semiring

```
1  def main() {
2    #IndexLabel Definition
3    IndexLabel [a] = [?];
4    IndexLabel [b] = [?];  #IndexLabel [b] = [0:?];
5
6    #Tensor Definition
7    Tensor<int> A([a,b], {CSR});  # Matrix in CSR format
8    Tensor<int> B([b], Dense);    # Dense vector
9    Tensor<int> X([b], Dense);
10   Tensor<int> M([b], Dense);    # Mask vector
11
12
13   #Tensor Readfile Operation
14   A[a,b] = comet_read(0, 1);  # read CSR matrix (the 2nd
         arg dictates how to read the matrix, e.g., lower
         or upper triangle)
15
16   #Tensor Fill Operation
17   B[a] = 1.0;
18
19   var N = 100;  # define a scalar
20
21   for index in range (N):
22     X[b]<M> = B[a] @(any, pair) A[a,b]; # semiring
           operator @
23     M[b] = X[b];
24   end
25
26   print(X);
27 }
```

Listing 3: An example program in our DSL that demonstrates the use of some programming constructs to support graph algorithms.

operator takes as input the pair of operations that needs to be combined. In this example, $@(+,*)$ combines addition and multiplication, thus it is equivalent to the standard matrix multiplication operation. Listing 3 demonstrates the use of the `any-pair` semiring between a dense vector and a sparse matrix (Line 22), where the first operator returns true if both the elements in the two input vectors are non-zero (paired); the second operator returns true if any pair was found in the output of the first operator. Our compiler also supports `plus-times`, `min-first`, `any-pair`, and `plus-pair`.

The compiler lowers the semiring operator $@(\cdot,\cdot)$ down to the tensor algebra dialect (Line 7 in Listing 1). Since semiring may contain different operations, the specific mapping of semiring to TA operators depends on the particular semiring used. For example, $@(+,*)$ maps to `ta.mul`, while $@(,*)$ maps to `ta.elews_mul`. Additionally, our compiler extends the semantics of the tensor algebra dialect operators to include the semiring type and the pair of operations required. Listing 1 shows how the `ta.mul` operator in the tensor algebra dialect is extended to represent semiring type `plus_times` (note the `semiring` attribute). In the next step, the compiler lowers the tensor algebra IR to index tree IR, where we perform most code optimizations and transformations for sparse computations. Listing 2 shows the result of this lowering step for the `plus_times` semiring described in Listing 1. Finally, the compiler lowers the index tree IR to the SCF IR and then to LLVM IR and machine code for execution.



(a) A<B> = A

(b) x<y> = x

Fig. 4: An illustrative example of the masking-based assignment operation.

B. Masking

In the GraphBLAS standard, masking is a technique to selectively apply operations to certain elements of input matrices or tensors. In our DSL, masking is represented by the operator $\langle\cdot\rangle$, which takes as input a matrix that represents the mask and has identical dimensions of the matrix to which masking is applied (hence the dimensions of the mask matrix are omitted). For example:

$$C[i,k]\langle M\rangle = A[i,j] * B[j,k] \tag{2}$$

performs a matrix-matrix multiplication between the inputs A and B. In this expression, the masking operator applied to C, $C[i,k]\langle M\rangle$, limits the scope of scalar operations to be performed and which elements of C need to be updated. The mask matrix M is a binary matrix in which $m_{ij} = 1$ iff element c_{ij} needs to be updated with the result of the computation. Figure 4(a) and 4(b) illustrate how masking works when applied to a sparse matrix and a sparse vector, respectively. In the Figures, A and x are the original structures, while B and y are the masks.

Our code generation algorithm for masking operation can be classified into two classes, depending on the sparsity structure of the masking matrix: push-based masking and pull-based masking [21]. As an example, let us consider the case $C{<}M{>} = A * B$ (spGEMM). The push-based masking algorithm is driven by the input matrices. The algorithm traverses matrix A by rows and "pushes" the non-zero elements into the corresponding rows of matrix B. First, by analyzing the nonzero elements in a row of A, the algorithm identifies the specific entries in B that contribute to the output C. Next, the compiler performs a linear combination between those elements from the row of A and corresponding rows of B. Finally, the row of A selects the row of M with the same row index, reducing the number of output elements to be computed. In contrast, the pull-based masking algorithm is driven by the matrix mask. In this case, the algorithm first traverses every non-zero element of the mask M and "pulls" the corresponding row from matrix A and the column from matrix B. First, by analyzing a non-zero element in M, the algorithm determines which row of A and which column of

Algorithm 1: TC algorithms in linear algebra.

```
Input: Graph, $A$, represented as an adjacency matrix. $U$, upper
       triangular part of $A$; $L$, lower triangular part of $A$.
Output: $ntri$, a scalar to store the resulting triangle count.
Function $Burkhardt$ ($A$, $ntri$):
    ntri=sum ((A * A) .* A) / 6 ;                // Burkhardt
Function $Cohen$ ($A$, $ntri$):
    ntri=sum ((L * U) .* A) / 2 ;                // Cohen
Function $SandiaLL$ ($A$, $ntri$):
    ntri=sum ((L * L) .* L);                     // SandiaLL
Function $SandiaUU$ ($A$, $ntri$):
    ntri=sum ((U * U) .* U);                     // SandiaUU
```

Algorithm 2: BFS expressed in linear algebra.

```
Input: Graph, $A$, represented as an adjacency matrix; $f$, the frontier
       vector. $s$, the source vertex; $n$, the number of vertices
Output: $l$, a vector of visited vertices' level.
Function $BFS$ ($A$, $s$, $n$, $l$):
    $f(s) = True$ ;                              // Initialize $s$ in $f$
    for $level$ = 0 to $n - 1$ ;                 // $level$ of the graph
    do
        $l\langle f \rangle = level$ ;           // Update the output $l$
        $f\langle \bar{l} \rangle = f * A$ ;     // compute with masking
        if $f$ is empty then
            break ;                              // earlier termination
```

B should be pulled out. Second, the algorithm does a sparse dot product between these selected vectors.

While the goal of both push-based masking and pull-based masking algorithms is to eliminate unnecessary computations, the pull-based masking approach is preferred when the matrix mask M has a very low density, as it only examines the elements in the input matrices determined by the mask. However, in this case, the input B needs to be stored in column-major storage such as CSC when performing a sparse matrix-sparse matrix operation. In contrast, the push-based masking approach is more general and allows B to be stored in a row-major format, such as CSR. Thus, this approach is preferred when the matrix mask M has a relatively high density. In general, the best approach depends on the sparsity level of the mask M: the compiler is designed to favor the approach driven by the most sparse matrix to eliminate as much unnecessary computation as possible.

Note that, in both approaches, there is an additional cost of accessing the mask matrix M and determining which elements pertain to the computation. However, the savings introduced by eliminating unnecessary computations greatly outweigh this additional cost, as shown in the next Section V.

V. Evaluation

In this section, we present the performance of automatically generated code for some of the sparse linear-algebra kernels and the graph algorithms. We compare our performance against LAGraph [6] which contains an assortment of graph algorithms implemented using linear algebra. LAGraph employs the SuiteSparse:GraphBLAS library for sparse linear algebra kernels.

A. Methodology and Benchmarks

To show the performance benefit of our work, we evaluate two sets of benchmarks: 1) simple sparse kernels commonly used in graph algorithm which consists of sparse matrix - sparse matrix multiplication (SpGEMM) and sparse matrix-sparse matrix elementwise multiplication operations, 2) two representative graph algorithms TC and BFS.

Triangle Counting (TC) algorithm counts the number of triangles given an input undirected graph G. This problem was also part of the GraphChallenge competition [22]. A triangle is defined to be a set of three mutually adjacent vertices in a graph. Generally, three vertices i, j and k form a triangle if edges (i, j), (j, k) and (k, i) are present in the graph. The naive way to count the number of triangles in a graph represented by adjacency matrix A is to perform the following operation: A^3 and take a trace of the resulting matrix. The final answer is obtained by dividing the obtained scalar by 6 to account for already counted triangles. The naive way is extremely computationally expensive since A^3 is most likely to become dense. There are various other linear algebra-based algorithms that propose better implementations as compared to the naive approach. For instance, element-wise multiplication (represented as $.*$) can be used instead of the second matrix multiplication operation in the naive approach, i.e., $(A^2). * A$. This is followed by performing a reduction operation across the matrix to obtain the final triangle count. The algorithm with element-wise operation is similar to performing the matrix multiplication with A as a mask. In this way, the calculations in A^2 that are subsequently masked out by the element-wise operation are not performed in the first place. Multiple algorithms for triangle counting exist [23] – the linear algebra formulation of the tested TC algorithms is described in Algorithm 1. These formulations include two linear algebra operations, a matrix-matrix multiplication, and an element-wise multiplication, followed by a reduction. As discussed earlier, the element-wise operation in each expression can be replaced by a masking operation. Finally, some algorithms utilize the lower and upper triangular parts of the adjacency matrix to limit the computational complexity of the problem.

Breadth-First Search (BFS) algorithm traverses nodes of the graph structure to understand a particular property, such as the level of reachability starting from a source vertex. The search starts from the source vertex and reaches all vertices at the current depth level before moving to vertices at the next level. We describe the linear algebra formulation of BFS in Algorithm 2. There are mainly two linear algebra operations: a masking operation that assigns levels to the level vector under the mask of f, the frontier vector, including visited vertices at the current level; and a sparse vector-sparse matrix multiplication operation, where the visited vertices are updated across iterations, and a masking operation, which concentrates the search on unvisited vertices.

We perform all experiments on an Intel Xeon Skylake Gold 6126 processor with 192 GB DRAM memory. We

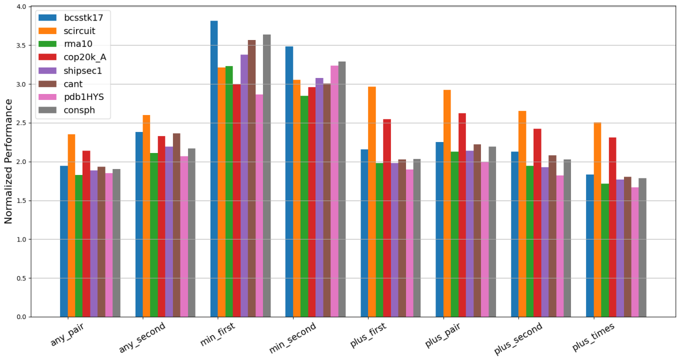


Fig. 5: Performance of semiring in our compiler as normalized to SuiteSparse:GraphBLAS when the output matrix is in a jumbled state.

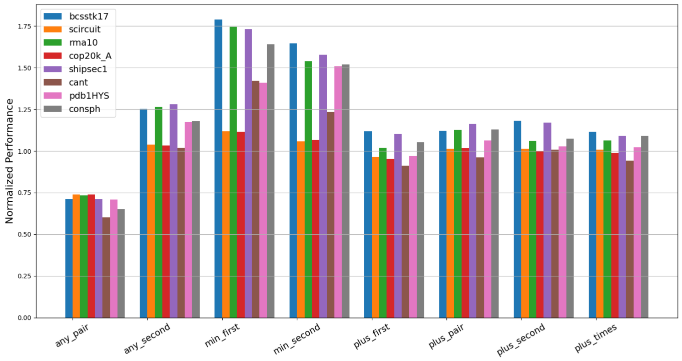


Fig. 6: Performance of semiring in our compiler as normalized to SuiteSparse:GraphBLAS when the output matrix is in an unjumbled state. (Note different y axis scale with Figure 5)

use `llvm-13` with optimization level `-O3` for compiling the LAGraph and SuiteSparse:GraphBLAS (version 7.3.2) packages. The code generated by our compiler is lowered to LLVM-IR using MLIR. The `mlir-cpu-runner` utility is used to run the LLVM-IR code on the CPU. The generated LLVM-IR code is further optimized using LLVM level `-O3` optimizations. This includes the ability of the LLVM backend to apply vectorizations when possible. That said, we do not fully explore the opportunity of MLIR to generate vectorized code using the vector dialect and this is part of future work.

The sparse inputs used for this paper are from SuiteSparse, and their characteristics are listed in Table I. The inputs `rma10` and `scircuit` are not used in the triangle counting evaluation because they are not symmetric. All the sparse inputs are stored in the CSR format. The output of this work is also produced in the CSR format. All results reported are the average of 10 runs. Unless otherwise noted, the evaluation uses sequential execution. Results for parallel execution are reported in Figure 10.

B. Performance Evaluation

We perform a detailed performance evaluation of the generated code by our compiler against the SuiteSparse:GraphBLAS library for common kernels utilized in most graph algorithms. We also evaluate various triangle counting and breadth-first search algorithms against their corresponding implementations in the LAGraph library. A detailed breakdown of performance gains obtained by our optimizations is included.

TABLE I: Sparse input matrices from SuiteSparse, ordered by density from high to low.

Name	Size	NNZ count	Density	Domain
bcsstk17	$10{,}974^2$	428,650	3.56×10^{-3}	Structural Problem
pdb1HYS	$36{,}417^2$	4,344,765	3.28×10^{-3}	Weighted Graph
rma10	$46{,}835^2$	2,329,092	1.06×10^{-3}	CFD Problem
cant	$62{,}451^2$	4,007,383	1.03×10^{-3}	2D/3D Problem
consph	$83{,}334^2$	6,010,480	8.65×10^{-4}	2D/3D Problem
shipsec1	$140{,}874^2$	3,568,176	1.80×10^{-4}	Structural Problem
cop20k_A	$121{,}192^2$	2,624,331	1.79×10^{-4}	2D/3D Problem
scircuit	$170{,}998^2$	958,936	3.28×10^{-5}	Circuit Problem
Orkut	$3{,}072{,}441^2$	234,370,166	2.48×10^{-5}	Social Network
LiveJournal	$3{,}997{,}962^2$	69,362,378	4.34×10^{-6}	Social Network

Semiring Performance. Several graph algorithms can be represented using semirings instead of traditional linear algebra operator to improve performance efficiency. To evaluate the performance of the semirings operation, we make a comparison between our compiler and SuiteSparse:GraphBLAS library for combination of SpGEMM with multiple operation pairs (semirings) and different sparse inputs. In this evaluation, the workspace transformation is applied to improve data locality and avoid irregular access to sparse data structures.

Figure 5 shows the performance of semiring in the compiler when the output is in the `jumbled` state. If the matrix is returned as jumbled, the column indices in any given row may appear out of order [8]. The sort is left pending. Some graph algorithms can tolerate jumbled matrices on input, so it is faster to generate jumbled output to be given as input to the subsequent operation. As shown in Figure 5, our work performs better than SuiteSparse:GraphBLAS for all the sparse inputs, up to 3.7× speedup. The `plus-times` semiring is another representation of sparse matrix-sparse matrix multiplication. The `plus-pair` semiring replaces the multiplication operation with `pair` operation in sparse matrix operation contributing to improved performance since the `pair` operation is a trivial operation as we operate on non-zero elements.

Figure 6 shows the performance of the same set of benchmarks while the output is in a `unjumbled` state in which the indices always appear in ascending order. The performance of sparse operations depends on the performance of the sorting algorithm if the resulting matrix must have indices sorted in each row. Our compiler currently uses the standard C++ quicksort algorithm (std::qsort) as compared to the advanced sorting algorithm implemented in SuiteSparse:GraphBLAS. Hence, the performance of the compiler significantly drops (1.75×) as compared to the jumbled case in Figure 5. We plan to improve the sorting algorithm in future work. The rest

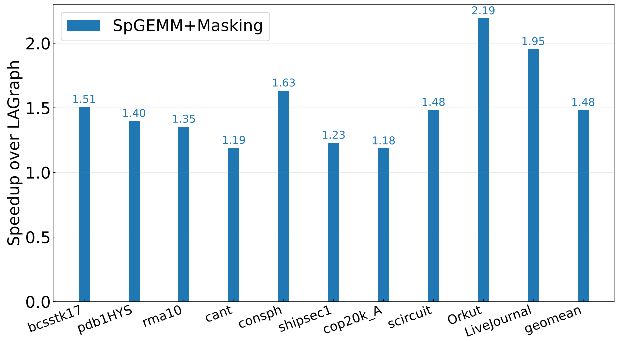


Fig. 7: Performance of masked SpGEMM (i.e., `B<A> = A*A`) as compared to SuiteSparse:GraphBLAS.

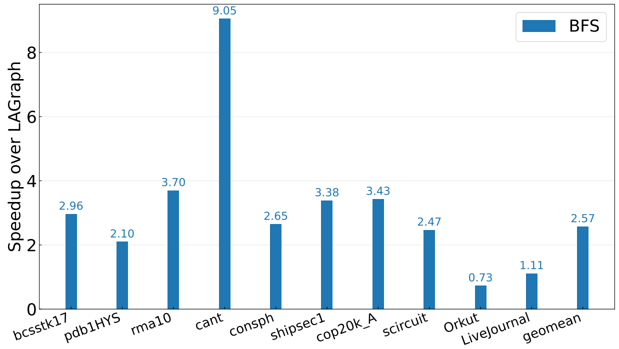


Fig. 9: Performance of the BFS algorithm with masking as compared to LAGraph.

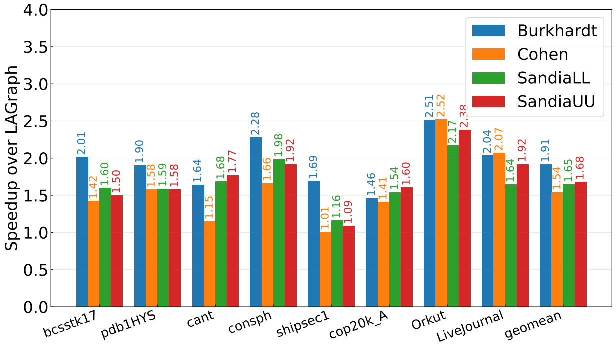


Fig. 8: Performance of the four Triangle Counting algorithms with masking as compared to LAGraph.

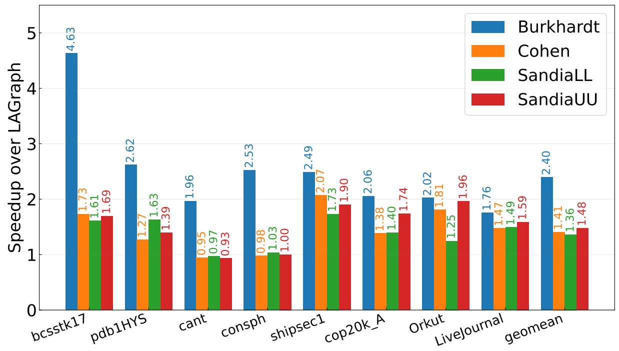


Fig. 10: Parallel performance of the four Triangle Counting algorithms with masking as compared to LAGraph.

of the paper represents the result when the output matrix is in an unjumbled state.

Overall Performance. First, we evaluate the performance of sparse matrix-sparse matrix operation with plus-times semiring (i.e., SpGEMM) using the input matrix as a mask inside both our compiler and SuiteSparse:GraphBLAS when all the optimizations are enabled in the compiler.

Figure 7 illustrates the speedup obtained by code generated by the compiler as compared to library-based realization of the same SpGEMM operations. The figure shows that the our performance is better than SuiteSparse:GraphBLAS across various inputs, and the compiler obtains up to $2.19\times$ speedup, with $1.48\times$ geometric mean speedup. Masking optimization avoids unneeded computations based on the requirements of the graph algorithms. Specifically, masking intervenes in the basic sparse vector-sparse matrix multiplication that is performed for each row of the other input matrix. At each iteration, the corresponding sparse row from the mask matrix is converted to an intermediate dense vector to support random $O(1)$ access to the elements in the mask. This accelerates the skipping of computations that do not need to be performed. The workspace transformation also provide some additional speedup as compared to SuiteSparse:GraphBLAS, which is evaluated further in this Section.

Next, we evaluate the performance of four different *Triangle Counting* algorithms implemented by our compiler. The performance is compared against LAGraph implementations of the same algorithms. The four algorithms are Burkhardt, Cohen, SandiaLL, and SandiaUU, whose linear algebra expressions are shown in Algorithm 1. In our experiments, we evaluate the implementation of these algorithms with the `plus-pair` semiring instead of SpGEMM operation (i.e., `plus-times` semiring) and with masking instead of the element-wise multiplication operation. These experiments demonstrate the benefit of all optimizations proposed in this paper. The cost to determine the strict lower and upper triangular parts of the input matrix is not included in the performance evaluations. Figure 8 shows the performance comparison of all four Triangle Counting algorithms implemented within our compiler and LAGraph with masking. It shows that our work can achieve up to $2.52\times$ speedup, and $1.91\times$, $1.54\times$, $1.65\times$, and $1.68\times$ geometric mean speedup over LAGraph for Burkhardt, Cohen, SandiaLL, and SandiaUU algorithms across all input matrices, respectively. The performance breakdown of various optimizations proposed inside the compiler is discussed later in this Section. In the results in Figure 8, when the input matrices have a relatively high density (e.g., `bcsstk17`), we do observe diminishing returns for algorithms that use sparser matrices such as lower

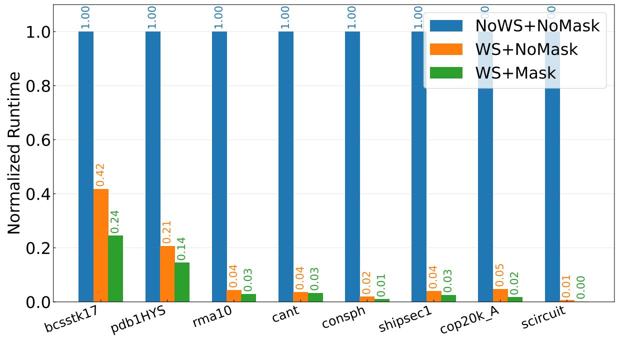


Fig. 11: A quantification of the performance gain obtained by applying our workspace and masking optimizations in a kernel that consists of the SpGEMM and element-wise multiplication operations (i.e., `(A * A) .* A`). *WS* stands for Workspace Transformation.

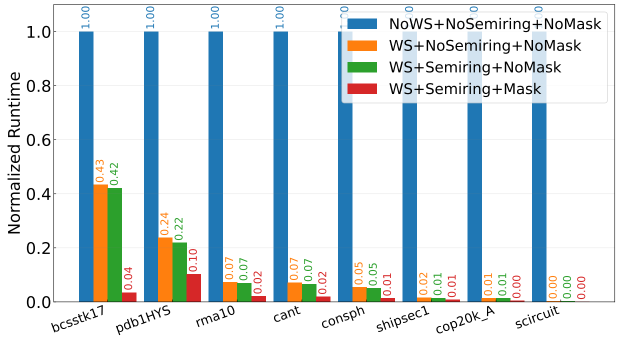


Fig. 12: Performance breakdown of triangle counting algorithm SandiaLL implemented by our compiler.

and upper triangular. We attribute this to our masking implementation that is based on the push method [21]. The push-based masking is more suitable for masks with higher density, whereas, pull-based masking is suitable for sparser masks. We plan to investigate our design choices in future work. Moreover, we expect to have better performance as we use an advanced sorting algorithm for unjumbled output matrices.

Next, Figure 9 illustrates the performance comparison of *BFS* implementation between our work and LAGraph. It shows that our work can achieve up to 9.05× speedup over LAGraph and geometric mean 2.57× speedup for all input matrices. The main speedup comes from our use of the workspace transformation. The major computation in the BFS level algorithm involves finding the next frontier in each iteration (see algorithm 2). It is achieved by performing a sparse vector-matrix multiplication with masking. Workspace transformation can avoid the expensive insertion into the middle of sparse data structures and performs asymptotically faster.

Parallel Performance. Figure 10 shows our parallel performance of four Triangle Counting algorithm with masking compared with LAGraph. All experiments use 24 threads. It shows that our work can achieve up to 4.63× speedup over LAGraph among all used input matrices, besides up to 2.02× speedup among the two large inputs `Orkut` and `LiveJournal`. Our work also achieves 2.40×, 1.41×, 1.36×, and 1.48× geometric mean speedup over LAGraph for Burkhardt, Cohen, SandiaLL, and SandiaUU algorithms among all input matrices, respectively. The results demonstrate that the compiler can achieve high-performance parallelization, thanks to the two-phase computation.

Performance Benefit Breakdown. This section discusses the performance gains obtained by each proposed optimization, including the workspace transformation, semiring, and masking.

The base case is implemented as a kernel that consists of the SpGEMM operations followed by the element-wise multiplication operation. This base version does not include any of the optimizations proposed in this paper. Then, we evaluate the performance gain of each of the optimizations. First, we apply the workspace transformation to improve data locality for sparse linear algebra operations. The masking optimization is then applied to eliminate the element-wise multiplication operation that succeeds the SpGEMM operation. The masking optimization improves the performance by skipping computations that are not needed since they will result in multiplication by zero in the element-wise multiplication operation. Figure 11 shows the performance progression as incrementally the workspace and masking optimizations are applied to the base case. The workspace transformation has 20.60× geometric mean speedup over the base case across all inputs. The masking operation can be seen to add another 1.86× speedup. It can be clearly seen that the proposed optimizations are important and lead to substantial gains compared to the base case, e.g., in most cases over 90% of the speedup is due to the workspace and masking optimizations. We highlight that the masking optimization is also important for low memory usage since we had difficulty running the relatively larger LiveJournal and Orkut inputs on our system.

We also profile the performance breakdown of the proposed optimizations for various triangle counting algorithms. Figure 12 shows the results of SandiaLL only for brevity. Other algorithms show the same trend. The base cases for these algorithms are shown in Algorithm 1, whereby a SpGEMM is followed by an element-wise multiplication operation, including reduction. As done earlier, the SpGEMM and element-wise multiplication operations can have the workspace transformation and masking optimizations applied incrementally. An additional performance advantage can be gained by replacing the SpGEMM operation with a semiring of `plus-pair`. Specifically, replacing the multiplication operation in SpGEMM with a pair operation tends to bring in a performance advantage of around 5% across all inputs for four triangle counting algorithms. Note that semiring operations are essential to support state-of-the-art graph algorithms [6], [24], [25].

Although the performance advantage is dependent on the sparsity of the input matrices, the proposed optimizations can be safely and effectively applied to achieve some benefit across multiple application domains that utilize sparse computation.

VI. Related Work

Graph Libraries. There are numerous graph libraries, such as [6], [9]–[11], [24]–[30], that aim to provide high-performance implementations of graph kernels using different sequential and parallel algorithms. LAGraph [6] is a library that contains representative graph algorithms and is based on sparse linear algebra operations from the SuiteSparse:GraphBLAS package [24]. On the other hand, NW-Graph [10] is a high-performance header-only graph library that leverages C++20 features. However, different libraries have their own approach to optimization and are tied to specific programming models. In contrast, the compiler potentially offers a unified solution for sequential and parallel code generation through the MLIR back-end while being complementary to existing library-based approaches.

Compilers for Sparse Computations. There are several domain-specific compilers designed for generating code of sparse operations in graph algorithms, including Green-Marl [31], GraphIt [32], and TACO [33]. These compilers, such as Green-Marl and TACO, perform source-to-source translation, where TACO translates its DSL operations to C++ using computational templates. However, TACO does not support parallel sparse computation (e.g., parallel SpGEMM), and its optimizations mainly focus on sequential code. In contrast, the compiler in this work proposes optimizations including two-phase computation and parallelization for sparse kernels.

Recently, MLIR infrastructure added support for sparse tensors through the sparse-tensor dialect [19]. COMET precedes this support and does not utilize the sparse-tensor dialect. As a result of MLIR's support for sparse tensors, we can expect more MLIR-based compilers to include support for graph algorithms in the future. One such example is the mlir-graphBLAS [34] effort that plans to lower to *linalg dialect*. Previously, it used to generate code at the loop level (SCF dialect) in a similar manner to this work albeit without optimizations such as workspace transforms.

VII. Conclusions

We present a compiler framework to simplify the development of graph algorithms and generate efficient code for target computing architectures. Built on top of COMET, this compiler consists of a DSL for developing graph algorithms using algebraic operations, optimized graph operators (such as semiring and masking), and various optimizations and code transformations (such as workspace transformation, two-phase computation, and automatic parallelization). We demonstrate the performance benefits of code generation through our compiler using common graph algorithms and compare it to a state-of-the-art library-based approach LAGraph. Our results show that compared to LAGraph, our compiler can achieve up to $3.7\times$ speedup in semiring operations, $2.19\times$ speedup in an important sparse computational kernel, and $9.05\times$ speedup in graph processing algorithms.

Acknowledgment

The authors thank anonymous reviewers for their constructive comments and helpful suggestions. The research described in this paper is part of the Data Model Convergence Initiative at Pacific Northwest National Laboratory (PNNL). It was conducted under the Laboratory Directed Research and Development Program at PNNL, a multi-program national laboratory operated by Battelle for the U.S. Department of Energy (DOE). This work was also supported by the U.S. DOE Office of Science, Office of Advanced Scientific Computing Research, under award 66150: "CENATE – Center for Advanced Architecture Evaluation" project. This work was also supported by the High-Performance Data Analytics (HPDA) program at PNNL. PNNL is operated by Battelle for the U.S. DOE under Contract DE-AC05-76RL01830.

APPENDIX A
ARTIFACT EVALUATION

A. Getting Started Guide

1) Availability: This artifact is available on

- Docker (https://hub.docker.com/r/cometpnnl/comet/tags),
- Zenodo (https://doi.org/10.5281/zenodo.8275066), and
- Github (https://github.com/pnnl/COMET/tree/pact23-ae).

If you got the artifact from Github, please check out the `pact23-ae` branch under the repository root by running

```
$ git checkout pact23-ae
```

2) Hardware Environment: This artifact is supposed to run on CPU machines with at least 60 GB of memory and 10 GB of free disk space.

3) Software Environment: If using Docker, please refer to Section A-A4 to prepare the container.

If not using Docker, this artifact requires:

- CMake for building the program ($\geq$ 3.25).
- Modern C++ compiler (LLVM getting started).
- Bash or Zsh for shell scripts.
- Python3 ($\geq$ 3.9) for Python scripts.
- Docker, preferred.

4) Prepare Docker: Assuming Docker is installed and running, the following command can be issued to retrieve the Docker container from Docker Hub as follows.

```
$ docker pull cometpnnl/comet:pact23
```

Approximately 1 GB of disk space will be required to download the image. The experiments will require an additional 6 GB of disk space due to the download of the SuiteSparse dataset.

The following command can be issued to run the downloaded image:

```
$ docker run -it cometpnnl/comet:pact23 /bin/bash
```

The Docker container contains all the necessary executables (COMET, LLVM) and libraries to run the experiments, as noted in sections A-B and A-C. The Dockerfile used to build the image is in the Github repository.

Once the docker container is running, the executable artifact is in the directory `AE/`.

5) Build From Sources: If using a Docker image is not preferred, please build from sources referring to `README.md`. Usually, the most time-consuming part is compiling LLVM, which could take a couple of hours depending on the machine.

Please also build LAGraph under the `AE/LAGraph`, which requires the GraphBLAS library built in advance. In general, LAGraph can be built by issuing the following commands:

```
$ cd AE/LAGraph/build/
$ GRAPHBLAS_ROOT=/graphblas/build cmake ..
$ make
```

6) Prepare Input Matrices and Environment: Under `AE/`, please run

```
$ bash scripts/sh0.install_libraries.sh
$ bash scripts/sh1.get_matrices.sh
```

These scripts will install Python3 dependencies and download all input matrices (about 6 GB) into `AE/data/`.

B. Quick Run

A quick script can run the masked SpGEMM benchmark of this work and LAGraph using the first 8 matrices in the paper. If you are in the Docker container, please run under `AE/`

```
$ bash benchmarks/run0.quick_run.sh
```

If you build the artifact from sources, please run

```
$ bash benchmarks/run0.quick_run.sh test
```

The script usually takes 10 minutes to finish and will generate `quick_run.masked_spgemm.png` in `AE/results/`. The figure will show the speedup of our compiler-generated code over LAGraph. As there are many ways to access the figure, we find it convenient to use Visual Studio Code with Docker extension to download files from running containers. Additionally, one can also use the `docker cp` command to copy the figure from the container to the host by

```
$ docker cp <container-id>:</source> </target>
```

C. Get Main Results

Similar to the quick run, there are other scripts to get the main results of the paper.

For full results of Masked SpGEMM with 10 matrices, please run (under `AE/`, use `test` if built from sources)

```
$ bash benchmarks/run1.masked_spgemm.sh [test]
```

For results of Triangle Counting, please run

```
$ bash benchmarks/run2.triangle_counting.sh [test]
```

For results of Breadth-First Search, please run

```
$ bash benchmarks/run3.bfs.sh [test]
```

They will generate figures corresponding to Figure 7, Figure 8, and Figure 9, respectively. They show the speedup of our compiler over LAGraph with different inputs.

References

[1] S. Choudhury, L. Holder, G. Chin, K. Agarwal, and J. Feo, "A selectivity based approach to continuous pattern detection in streaming graphs," 2015.

[2] S. Mallik and Z. Zhao, "Graph- and rule-based learning algorithms: a comprehensive review of their applications for cancer type classification and prognosis using genomic data," *Briefings in Bioinformatics*, vol. 21, no. 2, pp. 368–394, 01 2019. [Online]. Available: https://doi.org/10.1093/bib/bby120

[3] G. Iñiguez, F. Battiston, and M. Karsai, "Bridging the gap between graphs and networks," *Communications Physics*, vol. 3, no. 1, may 2020.

[4] S. Acer, A. Azad, E. G. Boman, A. Buluç, K. D. Devine, S. Ferdous, N. Gawande, S. Ghosh, M. Halappanavar, A. Kalyanaraman, A. Khan, M. Minutoli, A. Pothen, S. Rajamanickam, O. Selvitopi, N. R. Tallent, and A. Tumeo, "EXAGRAPH: Graph and combinatorial methods for enabling exascale applications," *The International Journal of High Performance Computing Applications*, vol. 35, no. 6, pp. 553–571, 2021.

[5] J. Kepner and J. Gilbert, *Graph Algorithms in the Language of Linear Algebra*, J. Kepner and J. Gilbert, Eds. Society for Industrial and Applied Mathematics, 2011. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9780898719918

[6] G. Szárnyas, D. A. Bader, T. A. Davis, J. Kitchen, T. G. Mattson, S. McMillan, and E. Welch, "LAGraph: Linear algebra, network analysis libraries, and the study of graph algorithms," in *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2021, pp. 243–252.

[7] E. Mutlu, R. Tian, B. Ren, S. Krishnamoorthy, R. Gioiosa, J. Pienaar, and G. Kestor, "COMET: A domain-specific compilation of high-performance computational chemistry," in *Workshop on Languages and Compilers for Parallel Computing (LCPC'20)*. Springer.

[8] A. Buluç, T. Mattson, S. McMillan, J. Moreira, and C. Yang, "Design of the GraphBLAS API for C," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2017, pp. 643–652.

[9] J. G. Siek, L.-Q. Lee, and A. Lumsdaine, *The Boost Graph Library: User Guide and Reference Manual, The*. Pearson Education, 2001.

[10] "NWGraph: Northwest graph library," https://github.com/pnnl/NWGraph, 2020.

[11] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the GPU," in *Proceedings of the 21st ACM SIGPLAN symposium on principles and practice of parallel programming*, 2016.

[12] L. Dhulipala, G. E. Blelloch, and J. Shun, "Theoretically efficient parallel graph algorithms can be fast and scalable," *ACM Trans. Parallel Comput.*, vol. 8, no. 1, apr 2021. [Online]. Available: https://doi.org/10.1145/3434393

[13] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith *et al.*, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, 2020.

[14] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, "Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2018, pp. 247–261.

[15] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 77:1–77:29, Oct. 2017. [Online]. Available: http://doi.acm.org/10.1145/3133901

[16] R. Tian, L. Guo, J. Li, B. Ren, and G. Kestor, "A high performance sparse tensor algebra compiler in MLIR," in *2021 IEEE/ACM 7th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, 2021, pp. 27–38.

[17] E. Aprá *et al.*, "NWChem: Past, present, and future," *The Journal of Chemical Physics*, vol. 152, no. 18, p. 184102, 2020.

[18] T. Zhou, R. Tian, R. A. Ashraf, G. Kestor, R. Gioiosa, , and V. Sarkar, "ReACT: Redundancy-aware code generation for tensor expressions," in *The 31st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2022.

[19] A. Bik, P. Koanantakool, T. Shpeisman, N. Vasilache, B. Zheng, and F. Kjolstad, "Compiler support for sparse tensor computations in mlir," *ACM Trans. Archit. Code Optim.*, vol. 19, no. 4, sep 2022. [Online]. Available: https://doi.org/10.1145/3544559

[20] F. Kjolstad, P. Ahrens, S. Kamil, and S. Amarasinghe, "Tensor algebra compilation with workspaces," in *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2019, pp. 180–192.

[21] S. Milaković, O. Selvitopi, I. Nisa, Z. Budimlić, and A. Buluc, "Parallel algorithms for masked sparse matrix-matrix products," 2021. [Online]. Available: https://arxiv.org/abs/2111.09947

[22] S. Samsi, V. Gadepally, M. B. Hurley, M. Jones, E. K. Kao, S. Mohindra, P. Monticciolo, A. Reuther, S. T. Smith, W. Song, D. Staheli, and J. Kepner, "Graphchallenge.org: Raising the bar on graph analytic performance," *CoRR*, vol. abs/1805.09675, 2018. [Online]. Available: http://arxiv.org/abs/1805.09675

[23] A. Azad, A. Buluç, and J. Gilbert, "Parallel triangle counting and enumeration using matrix algebra," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE, 2015.

[24] T. A. Davis, "Algorithm 1000: Suitesparse: Graphblas: Graph algorithms in the language of sparse linear algebra," *ACM Transactions on Mathematical Software (TOMS)*, 2019.

[25] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke *et al.*, "Mathematical foundations of the graphblas," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2016.

[26] A. Azad, M. M. Aznaveh, S. Beamer, M. Blanco, J. Chen, L. D'Alessandro, R. Dathathri, T. Davis, K. Deweese, J. Firoz *et al.*, "Evaluation of graph analytics frameworks using the GAP benchmark suite," in *2020 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2020.

[27] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew, "Optimistic parallelism requires abstractions," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[28] "Graph kernel collection," https://github.com/CMU-SPEED/GKC, 2013.

[29] G. Siglidis, G. Nikolentzos, S. Limnios, C. Giatsidis, K. Skianis, and M. Vazirgiannis, "GraKeL: A graph kernel library in python." *J. Mach. Learn. Res.*, vol. 21, no. 54, pp. 1–5, 2020.

[30] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2013.

[31] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun, "Green-Marl: a dsl for easy and efficient graph analysis," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[32] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe, "Graphit: A high-performance graph dsl," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, 2018.

[33] F. Kjolstad, S. Chou, D. Lugato, S. Kamil, and S. Amarasinghe, "TACO: A tool to generate tensor algebra kernels," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Oct 2017, pp. 943–948.

[34] "mlir-graphblas," https://github.com/metagraph-dev/mlir-graphblas, 2022.