Design and implementation for nonblocking execution in GraphBLAS: tradeoffs and performance

ARISTEIDIS MASTORAS, Computing Systems Lab, Huawei Zurich Research Center, Switzerland
SOTIRIS ANAGNOSTIDIS*, Department of Computer Science, ETH Zurich, Switzerland
ALBERT-JAN N. YZELMAN, Computing Systems Lab, Huawei Zurich Research Center, Switzerland

GraphBLAS is a recent standard that allows the expression of graph algorithms in the language of linear algebra and enables automatic code parallelization and optimization. GraphBLAS operations are memory bound and may benefit from data locality optimizations enabled by nonblocking execution. However, nonblocking execution remains under-evaluated. In this article, we present a novel design and implementation that investigates nonblocking execution in GraphBLAS. Lazy evaluation enables runtime optimizations that improve data locality, and dynamic data dependence analysis identifies operations that may reuse data in cache. The nonblocking execution of an arbitrary number of operations results in dynamic parallelism, and the performance of the nonblocking execution depends on two parameters, which are automatically determined, at run-time, based on a proposed analytic model. The evaluation confirms the importance of nonblocking execution for various matrices of three algorithms, by showing up to 4.11× speedup over blocking execution as a result of better cache utilization. The proposed analytic model makes the nonblocking execution reach up to 5.13× speedup over the blocking execution. The fully automatic performance is very close to that obtained by using the best manual configuration for both small and large matrices. Finally, the evaluation includes a comparison with other state-of-the-art frameworks for numerical linear algebra programming that employ parallel execution and similar optimizations to those discussed in this work, and the presented nonblocking execution reaches up to 16.1× speedup over the state-of-the-art.

CCS Concepts: • **Computing methodologies** → **Shared memory algorithms**; **Massively parallel algorithms**; **Concurrent algorithms**; **Linear algebra algorithms**; • **Software and its engineering** → **API languages**; **Software performance**.

Additional Key Words and Phrases: GraphBLAS, nonblocking execution, data locality, lazy evaluation, loop fusion, loop tiling, dynamic data dependence analysis, dynamic parallelism, analytic performance model

1 INTRODUCTION

GraphBLAS defines a standard set of primitive building blocks to express graph algorithms in the language of linear algebra. Linear algebraic operations typically enable the expression of algorithms with a few lines of code that achieves high performance. The key idea that makes GraphBLAS unique amongst and more versatile than other linear algebraic frameworks is the notion of algebraic structures, i.e., monoids and semirings. Algebraic structures raise the level of abstraction and enable the expression of a wider range of algorithms as sequences of linear algebraic operations [19]. The user represents a graph as either an adjacency or incidence matrix, and linear algebraic operations over different semirings can efficiently implement graph operations.

Extension of Conference Paper: This article extends our talk [2] and short paper [22] by: (a) presenting the full design for nonblocking execution in GraphBLAS, (b) exploiting dynamic data dependence analysis to support multiple pipelines and improve the data locality optimizations, (c) describing the analytic model to automatically choose the performance parameters, and (d) evaluating the nonblocking execution for additional matrices of Pagerank and Conjugate Gradient and an additional algorithm, i.e., the Sparse Deep Neural Network single inference. In addition, this article provides a comparison of the ALP/GraphBLAS performance with that of the state-of-the-art, i.e., GSL [15], Eigen [16], and SuiteSparse:GraphBLAS [12].

*Part of this work was done while S. Anagnostidis was with the Computing Systems Lab, Huawei Zurich Research Center.

Authors' addresses: Aristeidis Mastoras, Computing Systems Lab, Huawei Zurich Research Center, Switzerland, aristeidis.mastoras@huawei.com; Sotiris Anagnostidis, Department of Computer Science, ETH Zurich, Switzerland, sotirios.anagnostidis@inf.ethz.ch; Albert-Jan N. Yzelman, Computing Systems Lab, Huawei Zurich Research Center, Switzerland, albertjan.yzelman@huawei.com.

The notion of semirings allows the user to express different algorithms with the same algebraic operations. For example, matrix–vector multiplication over a semiring of standard arithmetic allows the expression of the Conjugate Gradient algorithm, and the same algebraic operation over a min-plus semiring expresses a Shortest Path algorithm [3]. In addition, Figure 1 shows an example in the C++11 ALP/GraphBLAS [31] that uses matrix–vector multiplication over a Boolean semiring to enable the expression of an algorithm that finds all vertices within k hops from a source vertex. The algorithm uses the matrix `G` to represent the input graph, one vector to represent the starting node `source`, and another vector to store the output of the matrix–vector multiplication, $out = Gin$. Therefore, matrix–vector multiplication corresponds to an iteration of the algorithm, and each iteration computes and stores the adjacent nodes of the input vector in the output vector.

```
grb::Matrix<void> G(n, n);
grb::Vector<bool> in(n), out(n);
grb::Semiring<
  grb::operators::logical_or<bool>,
  grb::operators::logical_and<bool>,
  grb::identities::logical_false,
  grb::identities::logical_true
> booleanRing;

// graph ingestion into G is omitted

grb::setElement(out, true, source);
for (i = 0; i < k; ++i) {
  std::swap(in, out);
  grb::vxm(out, in, G, booleanRing);
}
```

Fig. 1. The matrix powers kernel over a Boolean semiring.

Applications for high performance computing, artificial intelligence, and deep learning can be expressed in GraphBLAS. Some popular examples of such applications include the Pagerank algorithm used by the Google search engine to rank web sites [20], centrality algorithms used to determine the importance of a node in a graph, and the Conjugate Gradient algorithm that solves systems of linear equations, the matrix of which is symmetric and semi-positive definite.

One of the main advantages of GraphBLAS is that a user writes serial code that achieves scalable and high performance, since algorithms expressed with linear algebraic operations enable automatic code parallelization and optimization. There exist various GraphBLAS implementations that are supported by the academia, industry, and governmental labs, and they automatically parallelize and optimize GraphBLAS programs for shared- and distributed-memory systems [4, 12, 26, 30–32]. Therefore, the low-level details of parallel and high-efficiency programming are hidden, and it is possible to map GraphBLAS programs onto different architectures without exposing this complexity to the user. Nonblocking execution may delay the execution of some GraphBLAS operations and enable additional optimizations. A preliminary evaluation of recent work by Mastoras et al. [22] shows promising results achieved via an implementation for nonblocking execution in GraphBLAS.

In this article, we extend that approach and present the novel design, implementation, and evaluation for nonblocking execution in GraphBLAS. Lazy evaluation delays the execution of GraphBLAS operations and enables nonblocking execution in a pure library implementation provided under ALP/GraphBLAS. Dynamic data dependence analysis identifies operations that may reuse data in cache, and the nonblocking execution of an arbitrary number of operations results in dynamic parallelism. The evaluation for various matrices of three algorithms confirms our expectations

about better cache utilization, and the nonblocking execution achieves up to 4.11× speedup over the blocking execution. To fully automate the nonblocking execution and achieve high performance for both small and large matrices, the performance parameters for the nonblocking execution are automatically determined based on an analytic model. The performance achieved in a fully automatic way is very close to that obtained with the best manual configuration, and the analytic model makes the nonblocking execution reach up to 5.13× speedup over the blocking execution.

Furthermore, the evaluation includes a comparison with the state-of-the-art, i.e., GSL [15], Eigen [16], and SuiteSparse:GraphBLAS [12]. SuiteSparse:GraphBLAS achieves parallel execution and Eigen additionally employs optimizations that are similar to those of the presented nonblocking execution. The evaluation confirms the efficiency of the nonblocking execution by showing up to 12.2× and 16.1× speedups over Eigen and SuiteSparse:GraphBLAS, respectively.

To summarize, the contributions of this article are:

- the design for nonblocking execution, which combines lazy evaluation, loop fusion, loop tiling, and dynamic parallelism to achieve data locality optimizations and parallel execution;
- the efficient dynamic data dependence analysis algorithm to support nonblocking execution with multiple pipelines and improve the data locality optimizations;
- the analytic performance model that fully automates the nonblocking execution;
- the implementation of the nonblocking execution in the C++11 ALP/GraphBLAS;
- the evaluation of the nonblocking execution for various matrices of three algorithms;
- the comparison of the nonblocking execution with state-of-the-art frameworks for numerical linear algebra programming; and
- the significant speedups achieved by the nonblocking execution over the blocking execution of ALP/GraphBLAS and the other state-of-the-art implementations.

2 BACKGROUND

The GraphBLAS standard defines three collections of elements, i.e., matrices, vectors, and scalars, used by linear algebraic operations, e.g., matrix–vector multiplication or the scalar dot product of two vectors. A GraphBLAS operation may receive a binary operator, a monoid, or a semiring as a parameter. For example, the matrix–vector multiplication operation receives a semiring.

2.1 Execution modes

The GraphBLAS C API specification [8] defines two execution modes, i.e., either blocking or nonblocking. The blocking execution mode guarantees that the output is computed and stored in memory when the function of the operation returns. On the other hand, the nonblocking execution mode allows a function to return although the computation may not be completed, since the nonblocking mode aims at performing code optimizations by delaying the execution of the operations. GraphBLAS operations are memory bound, and the overall performance depends on the time spent on moving data from and to the main memory. Therefore, in the case of blocking execution, data may be unnecessarily read from the main memory multiple times, instead of utilizing data locality that may lead to significant performance improvement.

2.2 Opaque data

Matrices and vectors are exposed to the user as opaque data types, which implies that their elements can be accessed only by using the GraphBLAS API. The data structure used for storing opaque data and the API may vary between different implementations of GraphBLAS. The GraphBLAS C API specification [8] initially exposed scalars as transparent data types, which precludes implementation-dependent optimizations. Based on this, the presented design for nonblocking execution assumes

non-opaque scalars, while future work in nonblocking execution may benefit from opaque scalars such as recently introduced in the second version of the C API [7].

2.3 Implementation details of ALP/GraphBLAS

While the design for the nonblocking execution presented in this work may be integrated into any GraphBLAS implementation, the remainder discussion, implementation, and evaluation shall focus on the open-source C++11 ALP/GraphBLAS [31], the source code of which is publicly available [1].

The ALP/GraphBLAS library implementation employs generic programming principles, which are compatible with the Standard Template Library (STL). ALP/GraphBLAS provides multiple template-based backends that automatically generate optimized and parallelized code. The choice of the backend is performed at compile-time, and the generated code targets shared- and/or distributed-memory systems. In particular, the main backends of ALP/GraphBLAS at present are three: one for serial execution that vectorizes operations to optimally use SIMD units, a second one for parallel execution on a shared-memory system by using OpenMP [27], and a third backend for parallel execution on a distributed-memory system by using the Lightweight Parallel Foundations (LPF) [29]. Hybrid execution is possible by combining the shared- and distributed-memory backends.

ALP/GraphBLAS provides a set of standard operators and identities, and the user may define custom operators and identities. Monoids and semirings are defined by combining operators and identities that fulfill some algebraic properties. The usage of type traits makes compile-time introspection of algebraic relation properties possible. For example, a compile-time error is reported when the user defines a monoid with a non-operator or with an operator that is not associative. ALP/GraphBLAS classifies primitive operations into four categories: primitives that operate on scalars (level-0), primitives that operate on vectors and scalars (level-1), primitives that operate on vectors and matrices (level-2), and primitives that operate on matrices (level-3).

The primitive operations of ALP/GraphBLAS handle vectors as either sparse or dense, and matrices are always handled as sparse. The elements of vectors, `grb::Vector<T>`, and matrices, `grb::Matrix<T>`, are of any Plain-Old-Data (POD) static type `T`, for which it is possible to copy the data by byte-for-byte transfers without requiring any form of marshalling or serialization.

ALP/GraphBLAS operations are amenable to vectorization even for non-standard types and operators. The templated codes employ local buffers that fit a configurable SIMD size and express operations as loops over such buffers [31]. For dense level-1 operations, this approach guides all compilers tested in generating fully vectorized binary code, which achieves performance equal to, or exceeding that of, a non-ALP/GraphBLAS vectorized baseline [31]. The same methodology applies to sparse operations, provided that the target architecture supports gather and scatter SIMD capabilities. Nevertheless, for the sparse case, whether contemporary compilers reliably generate the intended assembly instructions has not been tested thoroughly.

2.4 Problem statement

Figure 2a shows an example of three GraphBLAS operations that are executed in blocking mode: setting each vector element to some value c, an in-place application of some binary operation f, and an out-of-place application of some binary operation g. Although each of the three loops is included in a function for a GraphBLAS operation, for simplicity, we omit the invocation of the function and show directly the code executed by each function.

In the blocking mode, during the execution of the first loop, the data of the vector `x` are read from the main memory and loaded in cache. It is often the case that a vector may take a few hundred of MBs in memory and does not fit in cache. Thus, while the last iterations of the loop are executed, the data of the vector required by the first iterations of the loop are perhaps not in cache anymore. As a result of this situation, the data of `x` required by the first iterations of the second loop may

```
// data for x are loaded in cache
for (i = 0; i < N; ++i)
  x[i] = c;

// data for x and y are loaded in cache
for (i = 0; i < N; ++i)
  y[i] = f(y[i], x[i]);

// data for x, y, and z are loaded in cache
for (i = 0; i < N; ++i)
  z[i] = g(x[i], y[i]);
```

(a) Blocking execution.

```
for (i = 0; i < N; ++i) {
  x[i] = c;
  y[i] = f(y[i], x[i]);
  z[i] = g(x[i], y[i]);
}
```

(b) Nonblocking execution obtained after loop fusion.

Fig. 2. A simple example of the loop fusion optimization for three GraphBLAS operations.

have to be read again from the main memory. After the execution of the second loop that loads data in cache for both vectors x and y, the execution of the third loop will cause loading the data of x at least three times in total, and the data of y at least twice in total.

In the blocking execution mode, GraphBLAS operations are executed independently, i.e., one after another. However, reading data from the main memory is expensive and should be avoided as GraphBLAS operations are memory bound. Since the three loops of Figure 2a access shared data, it would be more efficient to load the data from the main memory only once for each vector and perform the computations of the three operations together.

The loop fusion [5] optimization provides a solution to this problem, by replacing the three loops of Figure 2a with a single loop shown in Figure 2b. Loop fusion may improve data locality and lead up to 2× speedup for this simple example, since the three operations may reuse data in cache, i.e., vectors x and y. By using loop fusion, the vectors z, y, and x may be read only once from the main memory instead of at least one, two, and three times, respectively.

Loop fusion is possible as the three loops have the same bounds and do not include cross-iteration dependences. The conforming bounds and embarrassingly parallel loops are a main contribution of GraphBLAS, since these nice properties are enabled by expressing algorithms with linear algebraic operations. In particular, programming in GraphBLAS results in operations that, in most cases, allow fully parallel execution for different elements, and these operations are performed on containers, i.e., vectors and matrices, of the same size, which result in loops with the same bounds.

Loop fusion is a well-known optimization that improves data locality and may lead to good performance for nonblocking execution in GraphBLAS. However, performing loop fusion in a fully automatic way and achieving good performance for all possible situations, e.g., for both small and large matrices, is far from obvious as it requires orchestration of different components to enable efficient execution. This is confirmed by the fact that none of the implementations of GraphBLAS supports full nonblocking execution. For example, SuiteSparse:GraphBLAS [12] is a full implementation of GraphBLAS, but it provides a very limited form of nonblocking execution discussed in Section 5. Moreover, other frameworks for numerical linear algebra programming, e.g., Eigen [16], perform loop fusion, but the evaluation in Section 4 shows that the proposed design for fully automatic nonblocking execution significantly outperforms the state-of-the-art. The main challenges in designing nonblocking execution are to fully automatically:

- decide which operations are safe to be fused together based on the data dependences;
- determine a group of operations that may improve data locality after performing loop fusion;
- combine loop fusion with other optimizations, e.g., loop tiling;
- exploit all opportunities for data locality optimizations, e.g., handling arbitrary control-flow;

- combine data locality optimizations with efficient parallel execution; and
- achieve good performance for different situations, by choosing proper parameters.

The presented design successfully addresses all these challenges. In the rest of this article, we present the necessary components for a novel design and implementation to achieve fully automatic nonblocking execution in GraphBLAS without any need for user intervention, i.e., the GraphBLAS API remains the same and the performance is automatically improved for different situations. We investigate the limits of automatic nonblocking execution, the tradeoffs, and the performance.

3 NONBLOCKING EXECUTION

We present the novel design for automatic nonblocking execution in GraphBLAS, which is provided as a new backend of the C++11 ALP/GraphBLAS [31] implementation for shared-memory systems. The design for the nonblocking execution combines:

- lazy evaluation to enable runtime optimizations;
- loop fusion to improve data locality;
- loop tiling to amortize runtime overhead;
- dynamic data dependence analysis to identify opportunities for reusing data in cache;
- dynamic parallelism to handle arbitrary control-flow and exploit a multi-core system; and
- an analytic performance model to fully automate the nonblocking execution.

3.1 Lazy evaluation

Lazy evaluation delays the execution of operations and enables the loop fusion and loop tiling optimizations at run-time. The invocation of a GraphBLAS operation does not necessarily imply its execution, as it always happens in the blocking execution mode. Instead, a lambda function is defined to execute one or more iterations of the original loop, and then it is stored for future execution. Lambda functions are an implementation decision that meshes well with template-based programming in the C++11 ALP/GraphBLAS. Other implementations of GraphBLAS may choose different mechanisms without affecting the presented design for lazy evaluation.

3.1.1 Loop fusion. Figure 3 shows the necessary modifications for implementing loop fusion for a simple program that includes three operations, i.e., `set`, `foldl`, and `apply` that corresponds to the `eWiseApply` primitive of ALP/GraphBLAS. The `set` operation initializes all the vector elements to some value. The `foldl` operation indicates a binary operator that is applied in-place, folding the output into the left-hand operand as in Figure 2. The `apply` operation performs an element-wise and out-of-place application of a binary operation. For the blocking execution, the code is executed when the function for the GraphBLAS operation is invoked as shown in Figure 3a, i.e., the function returns only once the result is computed and stored in memory. On the other hand, the nonblocking execution mode relies on lazy evaluation. To perform loop fusion, Figure 3b shows that the execution of the code is replaced with the definition of a lambda function that is stored, and the function returns without executing the lambda function.

3.1.2 Loop tiling. The loop tiling optimization partitions the iteration space into a number of tiles, and each tile consists of a number of successive iterations that is called tile size. Loop tiling is often applied by compilers to allow data accessed in a tile fit in cache and improve data locality.

Figure 4 shows a simple example for loop tiling by using a tile size equal to `T` for the loop of Figure 2b. The outer loop has a step equal to `T`, and each inner loop executes `T` iterations, except for the last iteration of the outer loop that may lead to a smaller tile of iterations. Loop tiling is used in the nonblocking execution mode to amortize the overhead for the invocation of the lambda

```
int main() {
  ...
  set(x, c);
  foldl(y, x, f);
  apply(z, x, y, g);
  ...
}

void set(Vec x, Scal c) {


  for (size_t i = 0; i < N; ++i)
    x[i] = c;


}

void foldl(Vec y, Vec x, Op f) {


  for (size_t i = 0; i < N; ++i)
    y[i] = f(y[i], x[i]);


}

void apply(Vec z, Vec x, Vec y, Op g) {


  for (size_t i = 0; i < N; ++i)
    z[i] = g(x[i], y[i]);


}
```

(a) Blocking execution.

```
int main() {
  ...
  set(x, c);
  foldl(y, x, f);
  apply(z, x, y, g);
  ...
}

void set(Vec x, Scal c) {
  auto func1 = [x, c]
                  (size_t low, size_t up) {
    for (size_t i = low; i < up; ++i)
      x[i] = c;
  }
  storeFunc(func1);
}

void foldl(Vec y, Vec x, Op f) {
  auto func2 = [y, x, f]
                  (size_t low, size_t up) {
    for (size_t i = low; i < up; ++i)
      y[i] = f(y[i], x[i]);
  }
  storeFunc(func2);
}

void apply(Vec z, Vec x, Vec y, Op g) {
  auto func3 = [z, x, y, g]
                  (size_t low, size_t up) {
    for (size_t i = low; i < up; ++i)
      z[i] = g(x[i], y[i]);
  }
  storeFunc(func3);
}
```

(b) Lazy evaluation employed by the nonblocking execution.

Fig. 3. Pseudo-code for the required modifications to enable loop fusion in the ALP/GraphBLAS library.

```
for (i = 0; i < N; i += T) {

  for (j = i; j < min(i + T, N); ++j)
    x[j] = c;

  for (j = i; j < min(i + T, N); ++j)
    y[j] = f(y[j], x[j]);

  for (j = i; j < min(i + T, N); ++j)
    z[j] = g(x[j], y[j]);
}
```

Fig. 4. Loop tiling applied to the loop of Figure 2.

functions. Figure 3b shows that each lambda function executes the code of a GraphBLAS operation for a range of iterations determined by the `low` and `up` bounds.

3.1.3 Building pipelines at run-time. Lazy evaluation expresses GraphBLAS operations as a linear sequence of stages that form a pipeline. As long as the output of an operation is a container, opaqueness guarantees that lazy evaluation is safe. Therefore, dynamic data dependence analysis

identifies a set of pipelines with which the operation shares data, and the operation is added as a new stage of a pipeline. A scalar or a container that is involved with a pipeline causes the execution of this specific pipeline in the following cases:

- the user explicitly extracts data from a container by using the GraphBLAS API, e.g., when reading the elements of a vector by using iterators in ALP/GraphBLAS;
- the user invokes the constructor of a container;
- memory is deallocated due to a destructor invocation;
- the invoked operation returns a scalar, e.g., the `dot` operation, which computes the scalar product of two vectors– here, the operation is first added into the pipeline, after which the pipeline executes immediately before returning the scalar;
- when a sparse matrix–vector multiplication (SpMV) operation is added into a pipeline with another operation that overwrites the input vector to the SpMV;
- when the user explicitly forces the execution of a pipeline via a call to `grb::wait()`.

The latter two points warrant deeper exposition, which follows.

ALP/GraphBLAS [31] stores matrices using Gustafson's format, i.e., a hybrid row- and column-major storage. However, sparse matrix–vector multiplication (SpMV), $y = Ax$, of the presented nonblocking execution relies exclusively on its row-major storage. In particular, for the i-th row of A, the computation of y_i is performed by accumulating the products of each nonzero element a_{ij} with x_j. Therefore, the computation of a single element of the output vector y potentially depends on all elements of the input vector x, and the loop fusion optimization may lead to incorrect code when the SpMV operation is added into a pipeline with another operation that overwrites the input vector of the SpMV operation. To avoid this situation and eliminate the data dependence, the pipeline that already includes one of these two operations is executed, and the other operation is added into another pipeline, i.e., into either a new pipeline or an existing pipeline. Although this work does not consider level-3 operations, a sparse matrix–sparse matrix multiplication (SpMSpM) operation implies the same constraint with SpMV, i.e., the SpMSpM operation cannot be fused together with another operation that overwrites any of the SpMSpM input matrices.

A user may explicitly enforce the execution of a pipeline, e.g., when measuring the execution time of a program, by using the GraphBLAS API. In particular, `grb::wait()` in ALP/GraphBLAS and `GrB_wait()` defined in the C API specification [8] guarantee that all pending operations are executed. By providing arguments to the wait function, the execution of pending operations may be limited to those that are required for fully computing the elements of specific output containers.

3.2 Dynamic data dependence analysis

Pipelines are built dynamically depending on the operations invoked in the program and on the data accessed by each operation. To improve data locality, the operations that are grouped together should reuse data in cache. The design for nonblocking execution maintains a set of active pipelines, and for each invoked operation, dynamic data dependence analysis identifies a subset of these pipelines that share data with the operation. If such a pipeline does not exist, the operation is added as a new stage of an empty pipeline. In the case that a single pipeline shares data with the operation, the operation is added as the last stage of this pipeline. When more than one pipelines share data with the operation, all the pipelines are merged together by maintaining the partial order of the stages, and the invoked operation is added as the last stage.

Figure 5 illustrates the pipelines that are dynamically created during the nonblocking execution of the Conjugate Gradient algorithm shown in Figure 6. In particular, the dynamic pipelines are created according to the dynamic data dependence analysis, and the invocation of a GraphBLAS operation creates a stage that may result in one or more of the following actions:

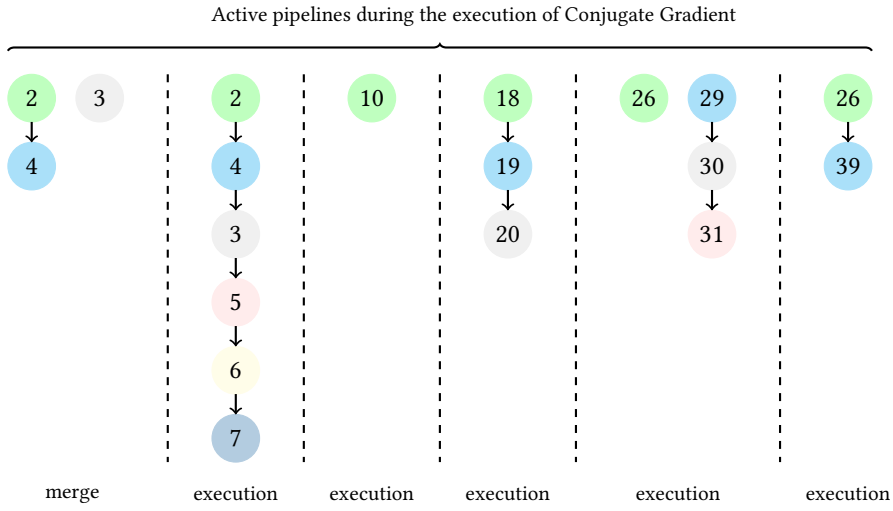

Fig. 5. Dynamic data dependence analysis for Conjugate Gradient.

- *early execution* of a pipeline with which the stage shares data, if the stage overwrites the input vector of a sparse matrix–vector multiplication operation included in the pipeline;
- *merge* of two or more pipelines;
- *addition* of the stage to an existing or a new pipeline; and
- *execution* of the pipeline in which the stage was added.

Unless another action happens in the meantime, Figure 5 shows implicitly the addition of one or more stages, by extending the corresponding pipelines. A merge or execution is explicitly shown, since the structure of the pipeline changes or the pipeline becomes empty. The action of early execution does not happen for the Conjugate Gradient algorithm. The number indicated for each stage in Figure 5 corresponds to the line of the GraphBLAS operation in Figure 6.

The first two operations access different vectors and are added into two different pipelines. The `mxv` operation in line 4 shares data with the `set` operation in line 2, and thus these two operations are grouped together. The `eWiseApply` operation shares data with both active pipelines. Therefore, the two active pipelines are merged into a single pipeline, and `eWiseApply` is added as the last stage of the merged pipeline. Both `set` and `dot` share data with the single active pipeline, and they become stages of it. The `dot` operation returns a scalar and causes the execution of the pipeline.

The `dot` operation in line 10 is added as the first stage of a new pipeline and is immediately executed. A new pipeline is created with the `set` operation in line 18. The pipeline is extended with the `mxv` and `dot` operations, and the execution is enforced after the `dot` operation. The `apply` operation in line 22 is not involved in lazy evaluation as it operates on scalars. The next pipeline is created by `eWiseMulAdd`. In line 29, the `eWiseMul` does not share data with `eWiseMulAdd`, and thus `eWiseMul` is added into a new pipeline. The next two operations are added into the same pipeline with `eWiseMul`, and the pipeline is executed after the `dot` operation.

The `apply` operation in line 35 is ignored as it operates on scalars. The `eWiseMulAdd` operation in line 39 is grouped together with `eWiseMulAdd` in line 26, since these two operations share data. The design for the nonblocking execution automatically caches the values of scalars and adds the two `eWiseMulAdd` operations into the same pipeline, although they use a different value of the `alpha` scalar, which is updated by `apply` in line 35. The execution of the pipeline that includes the two

```
1  // six-stage pipeline, vectors (temp, r, x, b, u)
2  grb::set(temp, 0);                                          // temp = 0
3  grb::set(r, 0);                                             // r = 0
4  grb::mxv(temp, A, x, ring);                                 // temp = A * x
5  grb::eWiseApply(r, b, temp, minus);                         // r = b - temp
6  grb::set(u, r);                                             // u = r
7  grb::dot(sigma, r, r, ring);                                // sigma = r' * r
8
9  // single-stage pipeline, vector (b)
10 grb::dot(bnorm, b, b, ring);
11
12 tol *= sqrt(bnorm);
13
14 size_t iter = 0;
15
16 do {
17   // three-stage pipeline, vectors (temp, u)
18   grb::set(temp, 0);                                        // temp = 0
19   grb::mxv(temp, A, u, ring);                               // temp = A * u
20   grb::dot(residual, temp, u, ring);                        // residual = u' * temp
21
22   grb::apply(alpha, sigma, residual, div);                  // alpha = sigma / residual
23
24   // part of a two-stage pipeline, vectors (x, u, r)
25   // the eWiseMulAdd at the bottom is the second stage
26   grb::eWiseMulAdd(x, alpha, u, x, ring);                   // x = x + alplha * u
27
28   // three-stage pipeline, vectors (temp, r)
29   grb::eWiseMul(temp, alpha, temp, ring);                   // temp = alplha * temp
30   grb::eWiseApply(r, r, temp, minus);                       // r = r - temp
31   grb::dot(residual, r, r, ring);                           // residual = r' * r
32
33   if (sqrt(residual) < tol) break;
34
35   grb::apply(alpha, residual, sigma, div);                  // alpha = residual / sigma
36
37   // part of a two-stage pipeline, vectors (x, u, r)
38   // the eWiseMulAdd aboce is the first stage
39   grb::eWiseMulAdd(u, alpha, u, r, ring);                   // u = r + alpha * u
40
41   sigma = residual;                                         // sigma = residual
42 } while (++iter < max_iterations);
```

Fig. 6. Conjugate Gradient in ALP/GraphBLAS.

`eWiseMulAdd` operations is enforced by the `mxv` operation in line 19, since the input vector of `mxv` is overwritten by the second `eWiseMulAdd`. This last pipeline does not include the `set` operation in line 18, since `set` does not share data with the operations in lines 26 and 39.

In the case of convergence, the break statement in line 33 interrupts the execution of the loop, and the last pipeline consists of a single stage, i.e., `eWiseMulAdd` in line 26. The execution of this pipeline happens either implicitly due to memory deallocation, implicitly by the user when reading out or further using the output vector x, or explicitly by the user when calling `grb::wait()`.

3.3 Dynamic parallelism

To achieve nonblocking execution, GraphBLAS operations are expressed as a linear sequence of stages that form a pipeline. The pipeline corresponds to the outer loop of Figure 7 obtained after loop fusion, and the stages correspond to the iterations of the inner loop, i.e., the lambda function

invocations. The pipeline has the same bounds as the initial loops of the operations, and the number of stages is determined by the inner loop, i.e., the operations that are grouped together at run-time. The parallel execution of the pipeline leads to dynamic parallelism, since the number of stages that are fused together, the tile size (T), and the number of threads (P) are all determined at run-time.

```
#pragma omp parallel for schedule(dynamic) num_threads(P)
for (i = 0; i < N; i += T)
  for (auto func = begin(); func != end(); ++func)
    // calls the stored functions
    (*func)(i, min(i + T, N));
```

Fig. 7. Execution of the lazily evaluated operations.

Operations added into the same pipeline access shared data, and in most cases, different stages of the same pipeline iteration do not allow parallel execution. Therefore, the nonblocking mode executes sequentially the inner loop in Figure 7, since each iteration of it corresponds to a different stage of the pipeline. The execution of different iterations of the pipeline, i.e., the outer loop in Figure 7, may overlap in time depending on whether the operations of GraphBLAS introduce cross-iteration dependences or not. Most operations do not introduce any cross-iteration dependences, since different loop iterations access different elements of the containers, e.g., the `set` operation.

An operation such as the `dot` introduces cross-iteration dependences, since all loop iterations accumulate the result in a shared scalar. Pipelite [23, 24] parallelizes loops with cross-iterations dependences that are expressed as dynamic linear pipelines. However, to avoid the overhead of handling cross-iteration dependences, the presented design for nonblocking execution automatically eliminates the data dependence in the `dot` operation by handling it as a reduction operation. Each thread computes its own result that is kept in thread-local storage, and the partial results of all threads are reduced into a single scalar. Therefore, the outer loop in Figure 7 does not exhibit any cross-iteration dependences and can be safely parallelized by OpenMP [27]. The dynamic scheduling policy of OpenMP is used to handle any load imbalance.

3.4 Analytic performance model

The tile size implies a clear tradeoff between overhead and parallelism. A small tile size may not amortize the overhead of the lambda functions, but it leads to a large number of iterations for the pipeline. That is, a large number of iterations creates more parallelization opportunities as it can utilize a large number of cores. On the other hand, a large tile size better amortizes the overhead implied by the lambda functions, but it leads to a small number of iterations that may limit parallelization opportunities. Therefore, the performance of the nonblocking execution depends on two important parameters: the number of threads and the tile size.

The presented design exploits an analytic model to automatically estimate, at run-time, proper values for both performance parameters. This is possible by considering the size of the L1 cache (L), the size of the vectors (V), the size of the data type (D) for the elements of the vectors, the number of the vectors (A) accessed in a pipeline, and the number of cores (C) available in the system.

The analytic model estimates the tile size based on the L1 cache size and the number of available cores. The first estimation gives an upper bound for the tile size that allows the data of size D for all accessed vectors, A, to fit in the L1 cache of size L:

$$T_{cache} = \left\lfloor \frac{L}{A \cdot D} \right\rfloor.$$

The second estimation gives an upper bound for the tile size that can utilize all available cores:

$$T_{cores} = \left\lfloor \frac{V}{C} \right\rfloor.$$

The final estimation is the minimum of T_{cache} and T_{cores},

$$T = max\{min\{T_{cache}, T_{cores}\}, MIN_{tile\_size}\},$$

while also considering a minimum tile size to successfully amortize the overhead of the lambda function invocations.

The analytic performance model chooses a number of threads:

$$P = min\left\{\left\lceil \frac{V}{T} \right\rceil, C\right\},$$

which cannot exceed the number of tiles, and thus it may be smaller than the available number of cores in the system.

The proposed analytic performance model is expected to achieve good performance on other architectures and configurations as long as the target architecture provides a private cache per core. However, the size of the private cache should be sufficiently large to successfully amortize the overhead of the lambda function invocations, i.e., at least MIN_{tile_size} elements of each accessed vector should fit in the private cache. If the private cache is small, the analytic model may be extended to target the L2 private cache, but targeting a larger private cache would affect the available bandwidth for the reused data. In the case of targeting a shared cache, the analytic model should be slightly modified to consider that the cache size is shared by a number of cores.

Architectures for non-uniform memory access (NUMA) may cause sharper effects when a large number of cores is employed by the analytic model, since additional cores used by vectors of a larger size may span new parts of the NUMA hierarchy. The analytic model does not capture such a non-linear increase in the observed cost, and thus it could be extended to consider any NUMA-related issues. Nevertheless, models should aim at being simplistic yet capturing the most important parameters. The evaluation in Section 4 confirms the effectiveness of the proposed analytic model in the presence of NUMA effects, by including experiments on a dual-socket system.

4 EVALUATION

We conducted an evaluation based on three algorithms: Pagerank, Conjugate Gradient, and a Sparse Deep Neural Network. We compare the performance of ALP/GraphBLAS by using three different implementations, i.e., serial blocking execution, parallel blocking execution, and parallel nonblocking execution. All implementations of ALP/GraphBLAS are amenable to vectorization. To provide a clear picture about the efficiency of the presented nonblocking execution in ALP/GraphBLAS, the evaluation includes a comparison of ALP/GraphBLAS v0.1 with the state-of-the-art, i.e., the GNU Scientific Library (GSL) [15] v2.7.1, Eigen [16] v3.4.0, and SuiteSparse:GraphBLAS [12] v7.2.0.

4.1 System specification

We run the experiments on a dual-socket system with two Intel Xeon Gold 6238T processors for a total of 44 cores at a clock frequency of 1.90 GHz each. We disabled both the hyper-threading and the turbo boost technology. The main memory is 192 GB, L1 cache is 32 KB, L2 cache is 1 MB, and L3 cache is 30.25 MB. There exist six memory channels per socket, and the memory speed is 2,933 MT/s for a system-wide theoretical peak bandwidth of 262.2 GB/s. Vectorization is performed with a SIMD width of 64 bytes. The operating system is Ubuntu 18.04.6 LTS with the kernel version 5.4.0-107-generic. The source code was compiled with g++ 9.2.1 and using the flags -O3 -march=native -mtune=native -funroll-loops.

4.2 Experimental methodology

The presented execution times and speedups are based on an average of at least 30 measurements for each experiment, except for a few very large matrices that are based on at least three measurements. All measurements correspond to the pure execution times of the algorithms, i.e., the time for reading the matrix, allocating memory, and so on, are excluded. For most of the experiments, the standard deviations are less than 3% of the average execution times. Despite best efforts, including re-running high variable experiments, some combinations of workloads and implementations display a high variability that we could not avoid. We present those results with underlined numbers.

We set thread-to-core affinities for all implementations. The serial execution utilizes one core, and we set 44 threads for the parallel execution of all implementations. ALP/GraphBLAS, Eigen, and SuiteSparse:GraphBLAS rely on OpenMP [27] to achieve parallel execution.

For the nonblocking execution, the number of threads and the tile size are automatically chosen according to the proposed analytic performance model described in Section 3.4. To compare the performance that is automatically achieved by the analytic model to that of manual configuration, we present results for different tile sizes, i.e., 256, 512, 1,024, 2,048, 4,096, 8,192 and 16,384. The small tile sizes show the performance when the L1 cache is not fully utilized, and the large tile sizes show the performance when the data of a tile do not fit in the L1 cache. The maximum tile size that allows the data of a tile fit in the L1 cache depends on the number of vectors accessed in a pipeline. For the experiments with manual configuration, the same values are used for the number of threads and the tile size of all pipelines regardless of the number of vectors they access. On the other hand, for the experiments with automatic configuration, the analytic model may choose a different number of threads and a tile size for the execution of different pipelines, since the dynamically created pipelines may access a different number of vectors. Therefore, this flexibility may make the automatic configuration exceed the performance of the best manual configuration.

The pipelines that are dynamically created for Pagerank and Conjugate Gradient access two to three vectors. There exist two exceptions: the first pipeline of Conjugate Gradient shown in Figure 6 that is created only once and accesses five vectors, and the first pipeline of Pagerank, which accesses a single vector. The pipelines of the Sparse Deep Neural Network access two vectors.

The three considered algorithms use vectors of doubles. The maximum number of vector elements that fit in the L1 cache of 32 KB is 4,096. For a pipeline that accesses a single vector, the maximum tile size that allows the data fit in the L1 cache is 4,096. Most of the pipelines that are dynamically created for the three considered algorithms access two or three vectors. Therefore, the maximum tile size that is automatically chosen by the analytic model is slightly smaller than 1,365 and 2,048, when two and three vectors are accessed, respectively.

4.3 Case studies

The evaluation considers three algorithms: Pagerank, Conjugate Gradient, and the Sparse Deep Neural Network single inference, and the experimental results are presented in Table 1, Table 2, and Table 3, respectively. The first two algorithms were evaluated using different matrices selected from the SuiteSparse Matrix Collection and the Stanford Network Analysis Project [14, 21]. For Pagerank, the full names of the matrices reported as coPapers and wikipedia are coPapersCiteseer and wikipedia-20070206, respectively. The Sparse Deep Neural Network algorithm was evaluated using different configurations for the Graph Challenge dataset [18]. This algorithm performs single inference and is different than the multi-inference algorithm presented by Davis et al. [13].

We report, step-by-step, the speedups achieved by the nonblocking execution. The *parallel speedup* corresponds to blocking over serial execution, i.e., the speedup is obtained from parallel execution. The speedup of nonblocking over blocking execution is obtained as a result of cache

Matrix	gyro_m	ca-AstroPh	G2_circuit	coPapers	bundle_adj	wikipedia	adaptive	road_usa	twitter7	europe_osm
Vectors size	17,361	18,772	150,102	434,102	513,351	3,566,907	6,815,744	23,947,347	41,652,230	50,912,018
serial	600	1,190	1,823	67,252	38,563	318,717	141,837	788,681	16,337,455	1,256,089
blocking	410	296	422	2,530	**2,191**	16,645	15,828	52,623	488,730	106,107
nonblocking T=256	152	172	400	2,551	3,061	13,931	14,527	48,944	455,732	114,595
nonblocking T=512	141	164	303	2,280	4,312	15,364	11,358	38,995	440,845	83,581
nonblocking T=1,024	153	185	268	**2,242**	7,047	15,855	10,063	36,677	**438,550**	73,873
nonblocking T=2,048	192	256	258	2,306	15,495	17,900	9,828	36,040	438,928	71,181
nonblocking T=4,096	270	415	291	3,670	29,495	22,699	**9,743**	35,485	439,681	70,382
nonblocking T=8,192	382	616	383	5,080	29,632	27,425	9,832	**35,260**	440,707	69,462
nonblocking T=16,384	523	863	598	7,439	29,582	45,279	10,216	35,554	441,571	**69,373**
automatic nonblocking	**126**	**141**	**257**	2,244	9,350	**12,440**	10,171	36,740	439,095	73,410
parallel speedup	1.46×	4.02×	4.32×	26.6×	17.6×	19.1×	8.96×	15.0×	33.4×	11.8×
data locality speedup	2.90×	1.80×	1.64×	1.13×	0.72×	1.19×	1.62×	1.49×	1.11×	1.53×
analytic model speedup	1.12×	1.16×	1.00×	1.00×	0.33×	1.12×	0.96×	0.96×	1.00×	0.95×

Table 1. Average execution times in microseconds and speedups for an iteration of Pagerank by using various matrices for different implementations of ALP/GraphBLAS. Bold indicates the best performance, and underline indicates six experiments with standard deviations up to 7%.

Matrix	gyro_m	vanbody	G2_circuit	bundle_adj	apache2	Emilia_923	ecology2	Serena	G3_circuit	Queen_4147
Vectors size	17,361	47,072	150,102	513,351	715,176	923,136	999,999	1,391,349	1,5785,478	4,147,110
serial	726	4,514	2,741	46,197	15,630	82,447	19,552	136,984	36,133	708,220
blocking	316	444	605	**3,638**	2,191	6,414	2,830	9,928	4,925	40,512
nonblocking T=256	163	**272**	403	4,060	1,573	5,507	2,030	9,149	3,261	40,692
nonblocking T=512	149	278	326	5,195	1,250	5,237	1,627	8,337	2,636	38,535
nonblocking T=1,024	151	314	293	8,034	1,140	**5,116**	1,479	8,069	2,490	**38,016**
nonblocking T=2,048	191	343	282	16,879	1,126	5,116	1,409	**8,049**	2,378	38,375
nonblocking T=4,096	266	534	313	30,551	**1,118**	5,323	**1,377**	8,173	**2,343**	38,124
nonblocking T=8,192	379	954	394	30,400	1,118	5,446	1,469	8,309	2,378	38,395
nonblocking T=16,384	509	1,802	590	29,465	1,174	5,998	1,583	8,520	2,558	39,027
automatic nonblocking	**130**	312	**281**	16,105	1,131	5,132	1,430	8,092	2,421	38,151
parallel speedup	2.30×	10.2×	4.53×	12.7×	7.13×	12.9×	6.91×	13.8×	7.34×	17.5×
data locality speedup	2.12×	1.63×	2.15×	0.90×	1.96×	1.25×	2.06×	1.23×	2.10×	1.07×
analytic model speedup	1.14×	0.87×	1.00×	0.25×	0.99×	1.00×	0.96×	0.99×	0.97×	1.00×

Table 2. Average execution times in microseconds and speedups for an iteration of Conjugate Gradient by using various matrices for different implementations of ALP/GraphBLAS. Bold indicates the best performance.

utilization, and the *data locality speedup* corresponds to that of the best manual configuration. We present the efficiency of the analytic performance model by reporting the *analytic model speedup* that corresponds to automatic nonblocking over the best manual nonblocking execution. Therefore, the speedup for the analytic performance model is less than 1.0×, when the automatic nonblocking execution performs worse than the nonblocking execution with the best manual configuration. On

Layers	120				480				1920			
Neurons	1,024	4,096	16,384	65,536	1,024	4,096	16,384	65,536	1,024	4,096	16,384	65,536
serial	7	28	116	528	28	112	468	2,126	112	447	1,926	10,451
blocking	24	27	21	48	98	112	86	250	400	452	351	1,022
nonblocking T=256	6	7	13	**39**	30	27	**53**	**160**	100	110	**226**	**715**
nonblocking T=512	7	9	13	41	34	34	56	174	124	147	235	787
nonblocking T=1,024	10	12	16	46	48	50	66	198	179	201	270	879
nonblocking T=2,048	10	20	24	52	48	84	98	240	177	343	392	1,028
nonblocking T=4,096	10	36	43	78	50	150	168	337	176	598	686	1,390
nonblocking T=8,192	10	37	79	120	48	149	310	499	181	609	1,259	2,030
nonblocking T=16,384	10	36	148	200	48	148	593	847	179	595	2,334	3,404
automatic nonblocking	**5**	**6**	**13**	52	**20**	**24**	55	230	**81**	**100**	230	968
parallel speedup	0.27×	1.02×	5.47×	11.0×	0.29×	1.00×	5.42×	8.50×	0.28×	0.99×	5.49×	10.2×
data locality speedup	4.07×	4.11×	1.64×	1.24×	3.28×	4.10×	1.62×	1.56×	4.01×	4.09×	1.55×	1.43×
analytic model speedup	1.26×	1.10×	0.98×	0.74×	1.51×	1.12×	0.97×	0.70×	1.24×	1.10×	0.98×	0.74×

Table 3. Total execution times in milliseconds and speedups for the Sparse Deep Neural Network single inference algorithm, by using different numbers of neurons, i.e., size of vectors, and layers, for different implementations of ALP/GraphBLAS. Bold indicates the best performance, and underline indicates five experiments with standard deviations up to 6%.

the other hand, the analytic model speedup is greater than 1.0× when the automatic nonblocking execution outperforms that of the best manual configuration.

4.3.1 Pagerank. The blocking execution outperforms the serial execution for all matrices of the Pagerank algorithm, achieving up to 33.4× parallel speedup. The performance of the nonblocking execution depends on the tile size, and a manual choice makes the nonblocking execution to achieve between 1.11× and 2.90× data locality speedups over the blocking execution. One exception is the bundle_adj matrix, for which the nonblocking execution reaches the best performance for a small tile size, and this performance is still worse than that obtained by the blocking execution. The adversarial nonzero structure of bundle_adj causes load imbalance that explains this poor performance. In particular, a small tile size results in a large number of pipeline iterations, which leads to better load balancing. By using a tile size that is smaller than 256, the performance shows a clear tradeoff implied by the tile size. Although it is possible to achieve even better load balancing, such a small tile size fails to amortize the overhead for the invocation of the lambda functions.

For most of the matrices, the nonblocking execution achieves the best performance for a tile size that allows the data fit in the L1 cache. For adaptive, the nonblocking execution achieves similar performance for a tile size between 2,048 and 8,192, and for the road_usa and europe_osm matrices, the nonblocking execution achieves slightly better performance for a tile size equal to 8,192 and 16,384, respectively. The analytic performance model leads to high performance that is very close to that obtained by the best manual configuration. For some of the small matrices, the automatic nonblocking execution outperforms the best manual configuration, since the analytic performance model chooses a number of threads that is smaller than 44. For the wikipedia-20070206 matrix, the analytic performance model chooses 44 threads and a different tile size for different pipelines, depending on whether the pipeline accesses two or three vectors. We notice that this setup makes the automatic nonblocking execution to outperform the best manual configuration.

4.3.2 Conjugate Gradient. The Conjugate Gradient algorithm requires matrices to be symmetric and semi-positive definite, and thus some of the selected matrices are different than those used for

Pagerank, and the size of the matrices is significantly smaller. The blocking execution outperforms the serial execution for all considered matrices, by reaching up to 17.5× speedup for the largest matrix. The nonblocking execution with manual configuration shows a clear tradeoff implied by the tile size. A small tile size leads to a large number of iterations for the pipeline, which is necessary to utilize a large number of cores for the small matrices, e.g., gyro_m and vanbody. However, a small tile size may fail to amortize the overhead for the lambda function invocations, and thus a large tile size leads to better performance for larger matrices.

Moreover, for all matrices, the nonblocking execution achieves the best performance for a tile size that allows the data fit in the L1 cache. The best manual configuration for the nonblocking execution leads to data locality speedups between 1.07× and 2.15× over the blocking execution. The nonblocking execution leads to performance degradation for the bundle_adj matrix for the reasons discussed earlier. Similar to Pagerank, the automatic nonblocking execution confirms that it is possible to fully automate the choice for the number of threads and the tile size, by showing experimental results that are very close to that obtained by the best manual configuration.

4.3.3 Sparse Deep Neural Network single inference. The experimental results show that the performance of the different implementations does not depend on the number of layers, which is expected as more layers result in additional iterations of the same algorithm. However, we notice that the performance depends on the number of neurons, i.e., the size of vectors. In particular, for 1,024 neurons, the blocking execution does not outperform the serial execution, since the 44 threads used by the blocking execution imply high overhead. The performance of the parallel blocking execution scales with the number of neurons, and the blocking execution achieves the highest speedup over the serial execution for 65,536 neurons.

By using the best manual configuration, the nonblocking execution outperforms both the serial and the blocking execution for all combinations of layers and neurons. One exception is for 480 layers and 1,024 neurons, for which the nonblocking execution is slightly slower than the serial execution as 44 threads imply high overhead. This situation is successfully handled by the analytic performance model that chooses a smaller number of threads, and the automatic nonblocking execution outperforms the best manual configuration for a small number of neurons, i.e., 1,024 and 4,096. The analytic performance model leads to almost the same performance with the best manual configuration for 16,384 neurons, and the performance of the automatic nonblocking execution is worse than that of the best manual configuration for 65,536 neurons. This happens as the analytic performance model chooses a tile size equal to 1,489 to combine maximum utilization of the 44 cores and as maximum utilization as possible of the L1 cache. However, the manual configuration shows that the highest speedup is achieved when the tile size is 256, since it leads to a larger number of iterations for the pipeline and may lead to better load balancing. The analytic performance model employed by the nonblocking execution outperforms the blocking execution for all combinations of layers and neurons, except for 120 layers and 65,636 neurons, which confirms that fully automatic nonblocking execution can lead to high performance for both small and large matrices.

For 120 layers and 1,024 neurons, the automatic nonblocking execution achieves 4.07× data locality speedup over the blocking execution, and the analytic model achieves an additional 1.26× speedup. These two speedups together result in a total of 5.13× speedup, which is the highest speedup achieved by the automatic nonblocking execution over the blocking execution.

The high data locality speedups observed for small matrices, e.g., about 4× for 1,024 neurons, are explained by three reasons. First, the dynamically created pipelines consist of five stages, which can justify a 4× speedup thanks to better cache utilization. Second, the effectiveness of exploiting data locality chiefly influences vector data. Since the workload also employs matrix data, the less matrix data are touched relative to vector data, such as happens, e.g., for small matrices, the higher

the expected speedup due to vector data locality improvements. Third, although we use the same number of threads to ensure a fair comparison of the blocking and the nonblocking execution with manual configuration, the observed speedup may not be only due to data locality improvements. Part of this speedup may be due to the overhead implied by the OpenMP runtime system, and in contrast to the nonblocking execution that incurs this overhead only once for each pipeline, the blocking execution incurs this overhead repeatedly.

4.4 Comparison to the state-of-the-art

The evaluation of the proposed automatic nonblocking execution includes a comparison to the state-of-the-art, and we report speedups over the ALP/GraphBLAS serial execution. In particular, we present speedups for GSL [15], which performs serial execution, for Eigen [16], which achieves parallel execution and performs optimizations such as loop fusion exploited in this work, and for SuiteSparse:GraphBLAS [12], which achieves parallel execution and exploits a limited form of lazy evaluation discussed in Section 5. To allow an easy comparison of the speedups obtained from the state-of-the-art with the speedups obtained from the automatic nonblocking execution, we present the total speedups of the nonblocking over the serial execution of ALP/GraphBLAS that includes: (a) the parallel speedup, (b) the data locality speedup, and (c) the analytic model speedup. We report this total speedup of ALP/GraphBLAS automatic nonblocking execution as *ALP/GraphBLAS nonblocking*. We also include the speedups of the ALP/GraphBLAS blocking execution, which are identical with the parallel speedups presented in Table 1, Table 2, and Table 3.

Matrix	gyro_m	ca-AstroPh	G2_circuit	coPapers	bundle_adj	wikipedia	adaptive	road_usa	twitter7	europe_osm
GSL serial	0.52×	0.91×	0.41×	0.75×	0.68×	0.82×	0.58×	0.91×	0.69×	0.68×
Eigen	2.25×	4.49×	1.33×	6.61×	1.25×	3.13×	1.14×	1.88×	7.95×	1.41×
SuiteSparse:GraphBLAS	0.78×	1.46×	0.44×	5.99×	4.13×	19.9×	6.74×	11.6×	27.7×	8.69×
ALP/GraphBLAS blocking	1.46×	4.02×	4.32×	26.6×	**17.6×**	19.1×	8.96×	15.0×	33.4×	11.8×
ALP/GraphBLAS nonblocking	**4.75×**	**8.42×**	**7.10×**	**30.0×**	4.12×	**25.6×**	**13.9×**	**21.5×**	**37.2×**	**17.1×**

Table 4. Speedups for an iteration of Pagerank by using various matrices for the state-of-the-art. Bold indicates the best performance, and underline indicates three experiments of ALP/GraphBLAS with standard deviations up to 7% and one experiment of SuiteSparse:GraphBLAS with a standard deviation of 4%.

Matrix	gyro_m	vanbody	G2_circuit	bundle_adj	apache2	Emilia_923	ecology2	Serena	G3_circuit	Queen_4147
GSL serial	0.84×	0.87×	0.95×	0.89×	0.88×	0.81×	0.91×	0.84×	0.97×	0.92×
Eigen	5.21×	11.2×	2.57×	1.61×	2.31×	5.83×	1.94×	5.80×	2.06×	9.20×
SuiteSparse:GraphBLAS	1.57×	5.02×	1.11×	5.82×	3.31×	7.37×	3.52×	9.00×	8.55×	11.6×
ALP/GraphBLAS blocking	2.30×	10.2×	4.53×	**12.7×**	7.13×	12.9×	6.91×	13.8×	7.34×	17.5×
ALP/GraphBLAS nonblocking	**5.57×**	**14.5×**	**9.75×**	2.87×	**13.8×**	**16.1×**	**13.7×**	**16.9×**	**14.9×**	**18.6×**

Table 5. Speedups for an iteration of Conjugate Gradient by using various matrices for the state-of-the-art. Bold indicates the best performance.

Layers	120				480				1920			
Neurons	1,024	4,096	16,384	65,536	1,024	4,096	16,384	65,536	1,024	4,096	16,384	65,536
GSL serial	0.36×	0.53×	0.64×	0.63×	0.54×	0.63×	0.69×	0.67×	0.64×	0.67×	0.72×	0.81×
Eigen	0.48×	2.42×	4.28×	6.33×	0.72×	2.70×	5.15×	6.76×	0.86×	3.02×	5.25×	7.91×
SuiteSparse:GraphBLAS	0.37×	0.72×	1.72×	3.34×	0.59×	0.92×	1.87×	3.49×	0.73×	1.03×	1.98×	3.98×
ALP/GraphBLAS blocking	0.27×	1.02×	5.47×	**11.0**×	0.29×	1.00×	5.42×	8.50×	0.28×	0.99×	5.49×	10.2×
ALP/GraphBLAS nonblocking	**1.40**×	**4.61**×	**8.81**×	10.1×	**1.42**×	**4.60**×	**8.50**×	**9.23**×	**1.39**×	**4.45**×	**8.39**×	**10.8**×

Table 6. Speedups for the Sparse Deep Neural Network single inference algorithm, by using different numbers of neurons, i.e., size of vectors, and layers, for the state-of-the-art. Bold indicates the best performance, and underline indicates experiments with high standard deviations: four experiments of ALP/GraphBLAS reach up to 6%, those of Eigen are between 5% and 59%, and those of SuiteSparse:GraphBLAS are between 2% and 12%, except for one experiment that is 46%.

Table 4, Table 5, and Table 6 present the experimental results for Pagerank, Conjugate Gradient, and the Sparse Deep Neural Network single inference, respectively. For all algorithms and for all matrices, the GSL serial execution is slower than the ALP/GraphBLAS serial execution.

The parallel execution of Eigen is faster than the blocking execution of ALP/GraphBLAS for the two smallest matrices of both Pagerank and Conjugate Gradient. This situation may be explained by the loop fusion optimization performed by Eigen. However, the blocking execution of ALP/GraphBLAS significantly outperforms Eigen for all the other matrices. SuiteSparse:GraphBLAS outperforms the ALP/GraphBLAS blocking execution only for the wikipedia-20070206 matrix of Pagerank and the G3_circuit matrix of Conjugate Gradient. For all the other matrices, the ALP/GraphBLAS blocking execution significantly outperforms SuiteSparse:GraphBLAS. For the Sparse Deep Neural Network single inference, Eigen and SuiteSparse:GraphBLAS are faster than the blocking execution of ALP/GraphBLAS for a small number of neurons.

The ALP/GraphBLAS nonblocking execution outperforms all the state-of-the-art implementations, for all algorithms, and for all matrices, with two exceptions. One exception is the special case of bundle_adj discussed in Section 4.3.1, for which the ALP/GraphBLAS blocking execution significantly outperforms all the other implementations, and Eigen is the only of the rest that outperforms the presented nonblocking execution. The second exception is the combination of 120 layers and 65,536 neurons used in the Sparse Deep Neural Network single inference, for which the blocking execution of ALP/GraphBLAS slightly outperforms the nonblocking execution. For Conjugate Gradient and gyro_m, the performance of the best manual configuration for nonblocking execution is slightly worse than that of Eigen as shown in Table 2, and the automatic nonblocking execution outperforms Eigen thanks to the smaller number of threads chosen by the analytic model. For the Sparse Deep Neural Network single inference, the nonblocking execution is the only one that outperforms the serial execution of ALP/GraphBLAS for 1,024 neurons, as a result of cache utilization and the automatic decisions made by the analytic performance model.

5 RELATED WORK

GraphBLAS raises the level of abstraction and enables implicit parallelism in a way similar to other frameworks for numerical linear algebra programming, e.g., Eigen [16], and the usage of skeleton functions [11]. Skeletons are basic building blocks that aim at achieving portability of parallel programs and their performance across different architectures. A program expressed with a skeleton that cannot be implemented efficiently on a given architecture may be re-expressed by using a different skeleton that provides a more efficient implementation.

Nonblocking is one of the two execution modes defined in the initial GraphBLAS C API specification [8]. SuiteSparse:GraphBLAS [12] is a full implementation of GraphBLAS, which provides a very limited form of nonblocking execution. In particular, SuiteSparse:GraphBLAS exploits the nonblocking execution mode by lazily adding nonzeroes to vectors or matrices, rather than inserting them directly into its sorted data structures. Operations that do not require sorting data can then operate separately on the unmodified sorted data and the unsorted new insertions. Only when an operation requires fully sorted data, all pending insertions are executed in bulk. This form of lazy evaluation is orthogonal to the optimizations presented in this work, and thus nonblocking execution may benefit by combining all these optimizations together.

Mattson et al. [25] propose explicit execution of lazily evaluated code by the user, for a computation that is associated with a given container. Such explicit synchronization became part of the GraphBLAS C API specification since version 1.3, by providing two variants that enforce the execution of pending operations. One variant enforces the execution of pending operations on a specific container, and the other variant enforces the execution of all pending operations due to lazy evaluation. The proposed design for nonblocking execution automatically triggers the execution of a pipeline whenever output escapes the control of ALP/GraphBLAS. Explicit execution of pending operations is possible by calling `grb::wait()`.

Similar to ALP/GraphBLAS, Eigen [16] exploits loop fusion to improve cache utilization. Our presented design for nonblocking execution builds pipelines at run-time, and thus loop fusion can cross control-flow boundaries, e.g., loops or conditional statements. Static techniques, such as compiler optimizations that Eigen induces, do not provide this additional flexibility and may miss opportunities for potential optimizations. Loop fusion is a key motivation for just-in-time (JIT) compilation. For example, LIBXSMM uses JIT to compile highly efficient code for small dense matrix computations that is able to fuse some successive operations [17]. None of these mechanisms that automatically fuse operations provide the genericity required to capture graph algorithms within a small set of primitives. On the other hand, GraphBLAS provides this genericity by explicitly exposing the notion of monoids and semirings. PyGB uses JIT to translate a Python GraphBLAS dialect into GBTL [9, 32]. The motivation is to offer the usability of a GraphBLAS Python interface with the performance of C++, but loop fusion optimizations are not included.

DESOLA [28] is an active library for dense linear algebra that delays the execution of operations and performs loop fusion and array contraction at run-time. The dynamic design of DESOLA overcomes the drawbacks of static approaches that fail to handle optimizations for arbitrary control-flow. DESOLA generates code at run-time, by relying on TaskGraph [6] that allows the user to explicitly construct the code generation and control the optimizations. TaskGraph implies significant compilation overhead that degrades the performance of DESOLA for small matrices, although this overhead is paid only once as the compiled code is cached.

Despite some similarities of our work with DESOLA, the contributions of DESOLA are significantly different than that of our work. DESOLA deals with dense matrices, and the evaluation does not show noticeable performance improvement for the array contraction optimization for any of the considered benchmarks, while the loop fusion optimization achieves significant speedups for two out of five benchmarks. For the other three benchmarks, TaskGraph generated code that performed loop fusion between vector-vector operations but not between matrix-vector operations. DESOLA does not deal with sparse matrices, and the authors state: *"Linear iterative solvers using sparse matrices have many more applications than those using dense ones, and would allow the benefits of loop fusion and array contraction to be further investigated"*. Moreover, the evaluation of DESOLA considers single-threaded solvers, and parallelization is left as future work.

On the other hand, the nonblocking execution presented in this work deals with sparse matrices and investigates the efficiency of loop fusion and loop tiling that show significant speedups

for almost all considered scenarios. The nonblocking execution exploits dynamic parallelism to utilize a shared-memory system, and our evaluation confirms the efficiency of both data locality optimizations and parallel execution. The nonblocking execution does not require code generation at run-time and avoids any dynamic compilation overhead. To achieve lazy evaluation, a specialized function is defined for each GraphBLAS primitive operation, and the overhead for the function invocation is successfully amortized by performing loop tiling. In addition, the proposed analytic performance model fully automates the nonblocking execution and leads to good performance for both small and large matrices. This is an important contribution of our work, since the evaluation using manual configuration for Pagerank, Conjugate Gradient, and the Sparse Deep Neural Network single inference shows that there is no fixed combination of a number of threads and a tile size that leads to good performance for all scenarios. In other words, we cannot replace the analytic performance model with some default values and maintain the good performance.

The TaskGraph [6] library used by DESOLA performs runtime optimizations, e.g., loop fusion and loop tiling, and implements a runtime search to find optimal performance parameters, e.g., a tile size. The analytic performance model employed by our work avoids the overhead of an exhaustive search. Moreover, the TaskGraph library performs runtime data-flow analysis, which may imply high overhead according to the authors. The dynamic data dependence analysis employed by the presented nonblocking execution exploits domain-specific knowledge to minimize this overhead. In particular, the number of input and output containers is both known and bounded for each primitive operation of GraphBLAS, and thus our dynamic data dependence analysis is tailored to the needs of GraphBLAS and avoids unnecessary steps.

Similar runtime optimizations outside the scope of linear algebraic frameworks have been proposed for an active visual effects library. Cornwall et al. [10] present a prototype component library that combines dependence metadata at run-time to build a polytope representation and perform inter-component loop fusion and array contraction, without expensive dependence analyses. The construction of components is performed by an algorithmic skeleton interface that allows the separation of kernels from the loops to enable optimizations and the collection of metadata to determine what optimizations may be performed.

To summarize, there exist different design options for nonblocking execution covered by previous work and the proposed work. On one hand, we have static approaches, e.g., Eigen, that avoid any runtime overhead but imply control-flow limitations. On the other hand, the proposed design for the nonblocking execution is fully dynamic. The proposed design is perhaps not ideal, however, the experimental results include any runtime overhead paid due to the fully dynamic design and show that our dynamic approach outperforms Eigen. Although the delayed execution exploited by DESOLA is different than a Just-in-Time (JIT) approach, both DESOLA and JIT approaches generate code at run-time and may be seen as a compromise of a fully static and a fully dynamic approach. Therefore, a JIT approach is another interesting design option for the nonblocking execution in GraphBLAS, since it can maintain the flexibility of our dynamic design and handle optimizations for arbitrary control-flow. However, it is unclear if a JIT approach can outperform the current dynamic approach due to the JIT compilation overhead.

6 CONCLUSION

We present the novel design, implementation, and evaluation for nonblocking execution mode in GraphBLAS. Lazy evaluation, dynamic data dependence analysis, and dynamic parallelism are combined to perform automatic runtime optimizations that improve data locality and achieve efficient parallel execution on shared-memory machines. The evaluation confirms the efficiency of the nonblocking execution for various matrices of three algorithms written in GraphBLAS, by showing up to 4.11× speedup over blocking execution as a result of better cache utilization.

The performance of the nonblocking execution depends on two parameters: the number of threads and the tile size. We propose an analytic performance model to automatically choose both parameters at run-time. The analytic performance model makes the nonblocking execution outperform the serial execution for all matrices, including those that consist of a small size of vectors. In most cases, the analytic performance model leads to high performance that is very close to that achieved by the best manual configuration, and in some cases, the fully automatic nonblocking execution exceeds the performance of the best manual configuration. Finally, the evaluation includes a comparison with the state-of-the-art that confirms the efficiency of the presented design for nonblocking execution in GraphBLAS.

7 FUTURE WORK

This article adds significant experience to the design, implementation, and evaluation for nonblocking execution in GraphBLAS. However, further investigation is required for a full and more advanced nonblocking execution that can efficiently handle different situations.

For example, the current design does not allow a sparse matrix–vector multiplication (SpMV) operation to be added into the same pipeline with another operation that overwrites the input vector of the SpMV. This constraint arises from the data dependences between the two operations, since the i-th element of the SpMV output vector potentially depends on all elements of the input vector. Breadth-first search (BFS) is an example of an algorithm, which does not lead to the creation of pipelines with more than one stage, since BFS consists of a sequence of SpMV operations and each operation overwrites the input vector of the other. Performing loop fusion for an arbitrary matrix is impossible. Loop fusion is a valid optimization, provided that the computation for the i-th element of the output vector does not depend on elements of the input vector generated by later iterations, i.e., the i-th element of the output vector may still depend on any a-th element, with $0 \leq a \leq i$. If a matrix allows the loop fusion optimization, then the parallel execution should respect the cross-iteration dependences by introducing some additional synchronization to ensure that all the elements of the input vector have been initialized before they are used for the computation of the output vector. Thus, loop iterations of the pipeline cannot execute in any order. A loop with cross-iteration dependences can efficiently be executed by Pipelite [24], which parallelizes dynamic linear pipelines based on user annotations and achieves load balancing by relying on a dynamic scheduler. However, it is unclear whether making the loop fusion optimization possible and introducing the necessary synchronization mechanism to ensure safe parallel execution would result in performance improvement or it would negate any benefit from nonblocking execution.

In addition, the current design for nonblocking execution relies exclusively on row-major storage. However, switching between row- and column-major storage is of special importance for the performance of other algorithms, e.g., BFS. Supporting both storage formats for the nonblocking execution will enable a more interesting evaluation with SuiteSparse:GraphBLAS and the blocking execution of ALP/GraphBLAS, since both of these implementations benefit from the combined row- and column-major storage and relevant optimizations.

8 ACKNOWLEDGMENTS

We thank the anonymous reviewers for their comments and contributions.

REFERENCES

[1] 2021. ALP/GraphBLAS. https://github.com/Algebraic-Programming/ALP Retrieved on June 22, 2022.
[2] 2022. Searchable Abstracts Document, Conference on Parallel Processing for Scientific Computing, February 23–26, 2022. https://www.siam.org/Portals/0/Conferences/PP22/PP22_ABSTRACTS.pdf Retrieved on August 17, 2022.

[3] Alfred V. Aho, John E. Hopcroft, and Jeffrey Ullman. 1974. *The Design and Analysis of Computer Algorithms*. Pearson Education.
[4] Mohsen Aznaveh, Jinhao Chen, Timothy A Davis, Bálint Hegyi, Scott P Kolodziej, Timothy G Mattson, and Gábor Szárnyas. 2020. Parallel GraphBLAS with OpenMP. In *2020 Proceedings of the SIAM Workshop on Combinatorial Scientific Computing*. 138–148.
[5] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. 1994. Compiler Transformations for High-Performance Computing. *Comput. Surveys* 26, 4 (Dec. 1994), 345–420.
[6] Olav Beckmann, Alastair Houghton, Michael Mellor, and Paul H. J. Kelly. 2004. *Runtime Code Generation in C++ as a Foundation for Domain-Specific Optimisation*. Springer Berlin Heidelberg, 291–306.
[7] Benjamin Brock, Aydın Buluç, Timothy Mattson, Scott McMillan, and Jose Moreira. 2021. The GraphBLAS C API Specification, version 2.0. http://www.graphblas.org
[8] Aydın Buluç, Timothy Mattson, Scott McMillan, Jose Moreira, and Carl Yang. 2017. The GraphBLAS C API Specification, version 1.0. http://www.graphblas.org
[9] Jesse Chamberlin, Marcin Zalewski, Scott McMillan, and Andrew Lumsdaine. 2018. PyGB: GraphBLAS DSL in Python with Dynamic Compilation Into Efficient C++. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops*. 310–319.
[10] Jay L. T. Cornwall, Paul H. J. Kelly, Phil Parsonage, and Bruno Nicoletti. 2008. Explicit Dependence Metadata in an Active Visual Effects Library. In *Languages and Compilers for Parallel Computing*. 172–186.
[11] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, Q. Wu, and R. L. While. 1993. Parallel Programming Using Skeleton Functions. In *PARLE '93 Parallel Architectures and Languages Europe*. Springer Berlin Heidelberg, 146–160.
[12] Timothy A. Davis. 2019. Algorithm 1000: SuiteSparse:GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra. *ACM Trans. Math. Software* 45, 4, Article 44 (Dec. 2019), 44:1–44:25 pages.
[13] Timothy A. Davis, Mohsen Aznaveh, and Scott Kolodziej. 2019. Write Quick, Run Fast: Sparse Deep Neural Network in 20 Minutes of Development Time via SuiteSparse:GraphBLAS. In *2019 IEEE High Performance Extreme Computing Conference*. 1–6.
[14] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Software* 38, 1, Article 1 (Nov. 2011), 1:1–1:25 pages.
[15] M. Galassi et al. 2009. *GNU Scientific Library Reference Manual (3rd Ed.)*. http://www.gnu.org/software/gsl/
[16] Gaël Guennebaud et al. 2010. Eigen v3. http://eigen.tuxfamily.org
[17] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. 2016. LIBXSMM: Accelerating Small Matrix Multiplications by Runtime Code Generation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 981–991.
[18] Jeremy Kepner, Simon Alford, Vijay Gadepally, Michael Jones, Lauren Milechin, Ryan Robinett, and Sid Samsi. 2019. Sparse Deep Neural Network Graph Challenge. In *2019 IEEE High Performance Extreme Computing Conference*. 1–7.
[19] J. Kepner and J. Gilbert. 2011. *Graph Algorithms in the Language of Linear Algebra*. Society for Industrial and Applied Mathematics.
[20] Amy N Langville and Carl D Meyer. 2011. *Google's PageRank and Beyond*. Princeton University Press.
[21] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.
[22] Aristeidis Mastoras, Sotiris Anagnostidis, and Albert-Jan N. Yzelman. 2022. Nonblocking execution in GraphBLAS. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 230–233.
[23] Aristeidis Mastoras and Thomas R. Gross. 2019. Chunking for Dynamic Linear Pipelines. *ACM Transactions on Architecture and Code Optimization* 16, 4, Article 44 (Nov. 2019), 44:1–44:25 pages.
[24] Aristeidis Mastoras and Thomas R. Gross. 2019. Efficient and Scalable Execution of Fine-Grained Dynamic Linear Pipelines. *ACM Transactions on Architecture and Code Optimization* 16, 2, Article 8 (June 2019), 8:1–8:26 pages.
[25] Timothy G Mattson, Carl Yang, Scott McMillan, Aydin Buluç, and José E Moreira. 2017. GraphBLAS C API: Ideas for future versions of the specification. In *2017 IEEE High Performance Extreme Computing Conference*. 1–6.
[26] José E Moreira, Manoj Kumar, and William P Horn. 2018. Implementing the GraphBLAS C API. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops*. 298–309.
[27] OpenMP Architecture Review Board, Version 5.2. 2021. OpenMP Application Program Interface. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf Retrieved on April 22, 2022.
[28] Francis P. Russell, Michael R. Mellor, Paul H.J. Kelly, and Olav Beckmann. 2011. DESOLA: An active linear algebra library using delayed evaluation and runtime code generation. *Science of Computer Programming* 76, 4 (2011), 227–242. Special issue on library-centric software design (LCSD 2006).
[29] Wijnand Suijlen and A. N. Yzelman. 2019. Lightweight Parallel Foundations: a model-compliant communication layer. arXiv:1906.03196 [cs.DC]

[30] Carl Yang, Aydin Buluc, and John D Owens. 2019. GraphBLAST: A High-Performance Linear Algebra-based Graph Framework on the GPU. *arXiv preprint arXiv:1908.01407* (2019).

[31] A. N. Yzelman, D. Di Nardo, J. M. Nash, and W. J. Suijlen. 2020. A C++ GraphBLAS: specification, implementation, parallelisation, and evaluation. (2020). http://albert-jan.yzelman.net/PDFs/yzelman20.pdf Preprint.

[32] Peter Zhang, Marcin Zalewski, Andrew Lumsdaine, Samantha Misurda, and Scott McMillan. 2016. GBTL-CUDA: Graph Algorithms and Primitives for GPUs. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops*. 912–920.