Nonblocking execution in GraphBLAS

Aristeidis Mastoras
Computing Systems Laboratory
Zurich Research Center
Huawei Technologies Switzerland

Sotiris Anagnostidis
Department of Computer Science
ETH Zurich

Albert-Jan N. Yzelman
Computing Systems Laboratory
Zurich Research Center
Huawei Technologies Switzerland

***Abstract*—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 executed either in blocking or in nonblocking mode. Although there exist multiple implementations of GraphBLAS for efficient blocking execution on both shared and distributed memory systems, none of these implementations supports full nonblocking execution. In this paper, we present a preliminary evaluation that confirms the importance of nonblocking execution in GraphBLAS, by showing promising speedups over the corresponding blocking execution for two algorithms, Pagerank and Conjugate Gradient.**

***Index Terms*—GraphBLAS, nonblocking execution, data locality, lazy evaluation, dynamic parallelism, loop fusion, loop tiling, analytical modeling, automatic runtime system configuration**

I. Introduction

GraphBLAS provides an elegant way to express graph algorithms in the language of linear algebra, typically resulting in a few lines of code, yet with performant results [1]. As the key idea, GraphBLAS represents a graph as an adjacency or incidence matrix, and it expresses graph operations as linear algebraic operations. For example, the algorithm for Breadth-First Search (BFS) in GraphBLAS represents the input graph as a matrix A and the starting node s with a vector x. It performs a BFS iteration by the matrix-vector multiplication, $y = Ax$, which computes and stores the adjacent nodes of s in the output vector y.

The notion of semirings in GraphBLAS enables expressing different algorithms with the same algebraic operations. For example, we can express the Conjugate Gradient algorithm by using a semiring of standard arithmetic for the matrix-vector multiplication. The same operation with a semiring corresponding to the min-plus algebra allows us to express a Shortest Path algorithm [2], and finding all vertices within k hops from a source vertex (k-hop BFS) can be implemented using a boolean semiring. Figure 1 shows the implementation of the k-hop BFS algorithm in ALP/GraphBLAS.

GraphBLAS operations are memory bound, and the overall performance depends on the time spent on moving data to and from the main memory. The ANSI C11 GraphBLAS specification [3] defines two execution modes, i.e., either blocking or nonblocking. The blocking mode guarantees that the output is computed when the function of the operation returns. This situation implies unnecessary data movement and precludes utilizing data locality that may lead to significant performance improvement. On the other hand, the nonblocking execution allows a function to return although the computation may not be completed. Delaying the execution of operations allows for additional optimizations, which for current GraphBLAS implementations remains under-evaluated.

```
grb::Matrix< void > G( n, n );
grb::Vector< bool > in( n ), out( n );

grb::setElement(out, true, source);

// graph ingestion into G omitted

grb::Semiring<
   grb::operators::logical_or< bool >,
   grb::operators::logical_and< bool >,
   grb::identities::logical_false,
   grb::identities::logical_true
> booleanRing;

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

Fig. 1: The k-hop BFS algorithm in ALP/GraphBLAS.

In this paper, we evaluate the novel design of a nonblocking implementation provided under the C++11 ALP/GraphBLAS dialect presented by Yzelman et al. [4]. A preliminary evaluation for various datasets of two algorithms confirms our expectations about better cache utilization. To fully automate the nonblocking execution and achieve high performance for both small and large datasets, we use an analytical model to automatically determine the performance parameters for the execution of GraphBLAS programs. The evaluation shows that the performance achieved in a fully automatic way is very close to that obtained with the best manual configuration.

II. Nonblocking execution

We present the novel design and evaluation of a library-based implementation that automatically performs nonblocking execution for shared memory systems. The GraphBLAS API remains the same and the nonblocking execution achieves better performance, as a result of better cache utilization. The key ideas in the design of the nonblocking execution are *lazy evaluation* and *dynamic parallelism*. Lazy evaluation delays the execution of GraphBLAS operations and enables efficient nonblocking execution, by performing, at run-time, the loop

This work was done while Sotiris Anagnostidis was with the Computing Systems Laboratory, Zurich Research Center, Huawei Technologies Switzerland.

fusion and loop tiling optimizations. Dynamic parallelism is achieved by grouping an arbitrary number of operations that reuse data in the cache and by executing them in parallel.

A. Lazy evaluation

The invocation of a GraphBLAS operation does not necessarily imply its execution, as it always happens in the blocking execution mode. Instead, a function that corresponds to the GraphBLAS operation is defined, and the function is stored for future execution. The lazily evaluated code is executed when the delayed computation is necessary to ensure the sound execution of the program. Since the invocation of a function may imply high overhead, the loop tiling optimization is used to amortize this overhead and improve the performance.

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

   lower = i;
   upper = getUpperBound(N, lower, tiling_size);

   for (func = begin(); func != end(); ++func)
      (*func)(lower, upper);  // call stored function
}
```

Fig. 2: Execution of operations using lazy evaluation.

Figure 2 shows the code for the execution of the lazily evaluated operations. The outer loop iterates over all the elements of the accessed vectors and matrices in tiles of size `tiling_size`, and the inner loop iterates over all the stored functions that are fused together at run-time.

B. Dynamic parallelism

The lazy evaluation strategy enables GraphBLAS operations to be expressed as a sequence of stages that form a pipeline. In particular, the pipeline corresponds to the outer loop of Figure 2 obtained after loop fusion, and the stages correspond to the iterations of the inner loop, i.e., the invocation of the stored functions. The structure of the pipeline is dynamic as the number of stages is determined at run-time.

The current design maintains a single pipeline. All invoked operations are added into the same pipeline regardless of whether they share data or not. This simple solution leads to good results according to the evaluation in Section III, which is expected as successive operations of a program are usually part of the same computation and operate on shared data.

Building pipelines dynamically also accounts for for-loops or conditional breaks. Static techniques cannot cross such control flow boundaries, except if explicit GraphBLAS constructs in their support were added to the specification.

GraphBLAS containers, i.e., vectors and matrices, are defined as opaque data types; they can be accessed exclusively by using the GraphBLAS API. Therefore, as long as the output of an operation is a container, opaqueness guarantees that lazy evaluation is safe and the operation is added into the pipeline. Pipelines always execute when the user extracts data from a container that is involved with the current pipeline.

GraphBLAS does not define its own opaque data type for scalars. Therefore, the pipeline must execute when the invoked operation returns a scalar, e.g., the `dot` operation which computes the scalar product of two vectors.

C. Parallel execution of the pipeline

Stages of the same pipeline do not allow parallel execution, and thus the inner loop in Figure 2 is executed sequentially. The execution of different iterations of the pipeline, i.e., the outer loop in Figure 2, 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 iterations write on different elements of the vectors, e.g., the `set` operation. For the two algorithms evaluated in this paper, there are no cross-iteration dependences between iterations, and the outer loop in Figure 2 is annotated with `#pragma omp parallel for`. Loops with cross-iteration dependences can be efficiently handled by Pipelite [5], which parallelizes dynamic linear pipelines based on compiler directives and performs dynamic scheduling to achieve load-balancing.

D. Analytic performance model

Nonblocking execution as described thus far has two major parameters: 1) the tiling size, and 2) the number of threads used for parallel execution. Rather than tuning these parameters manually for different algorithms, systems, and datasets, we monitor, at run-time, the pipeline size, the number of containers involved, and their sizes. On triggering a pipeline, these data are compared versus the machine parameters such as the total number of cores available and the size of the smallest private cache to each core, i.e., the L1 cache size. On the basis that we should employ and re-use private caches maximally, the resulting analytic performance model thus automatically derives a tiling size and a number of threads. These parameters depend on the pipeline and as such different parameter pairs may be selected between or even within algorithms.

III. Evaluation

We conducted a preliminary evaluation by comparing the performance of three shared-memory implementations, i.e., serial blocking execution, parallel blocking execution, and parallel nonblocking execution. The code for all implementations is vectorized, and the parallel code for both blocking and nonblocking execution is parallelized by OpenMP.

A. System specification and experimental methodology

We conducted 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. The source code was compiled with g++ 9.2.1 (-O3) on Ubuntu 18.04.6 LTS. Vectorization is performed with a SIMD width of 64 bytes.

The presented execution times are in milliseconds and are based on an average of 30 measurements for each experiment. The measurements correspond to the pure execution times of

the algorithms, i.e., the time for reading the dataset, allocating memory, and so on, are excluded. We set thread-to-core affinities for OpenMP for both blocking and nonblocking parallel execution. The serial execution utilizes one core, and the parallel blocking execution is set to utilize 44 cores.

For the nonblocking execution, the number of threads and the tiling size are chosen automatically based on an analytical model. To provide a clear picture about automatic configuration, we experimented with different tiling sizes and 44 threads. The tiling size varies from 256 to 16,384. The small tiling sizes are used to show the performance when the L1 cache is not fully utilized and the overhead of invoking the stored functions negates the performance. The tiling sizes of 8,192 and 16,384 show the performance when the data do not fit in the L1 cache. Automatic configuration by the analytical model may employ any number of threads between 1 and 44.

The pipelines that are dynamically created access a different number of vectors, and the highest possible performance may be achieved by using a different tiling size for each pipeline. For the experiments with manual configuration, the same values are used for the number of threads and the tiling size of all pipelines regardless of the number of vectors they access. When decisions are made automatically, different pipelines are executed using a different number of threads and tiling size.

B. Case studies

The evaluation considers two algorithms: Pagerank and Conjugate Gradient (CG), and their results are presented in Table I and Table II, respectively. Both algorithms were evaluated using different datasets that consist of vectors of a small and a large size, using matrices selected from the SuiteSparse Matrix Collection and the Stanford Network Analysis Project [6], [7].

Experimental results for the Pagerank algorithm show that both the parallel blocking and nonblocking modes significantly outperform serial execution for all datasets. The highest speedup for parallel execution is observed on the largest matrix in the dataset, at approximately $35-37\times$. Nonblocking execution achieves its highest performance with a tiling size between 256 and 4'096 for seven out of eight datasets. A larger tiling size does not allow the data to fit in the L1 cache. For road_usa, the highest performance is achieved for a tiling size of 8'192, though close to the 4'096 tiling size result. The highest speedups achieved by the nonblocking execution are $37.85\times$ and $2.43\times$ versus the serial and the blocking execution, respectively.

For the three smallest datasets, the highest performance is achieved for a small tiling size, since it results in a sufficiently large number of iterations for the pipeline that can utilize all available cores. For bundle_adj, the nonblocking execution achieves worse performance than the blocking execution as a result of load-imbalance caused by the structure of the matrix. A small tiling size leads to the highest performance as it results in a larger number of iterations and allows a more balanced distribution of the workload to threads.

Disregarding adverse nonzero structures such as in the case of bundle_adj, the nonblocking execution significantly outperforms the blocking execution, showing that automatic configuration of the number of threads and the tiling size achieves good performance that is close to the best manual configuration. For some datasets, e.g., Pres_Poisson, automatic configuration leads to better performance than the best manual configuration, since the automatic method may select a number of threads smaller than 44 if this predicts better performance.

The results for the Conjugate Gradient algorithm are similar to those obtained for the Pagerank algorithm. Both the parallel blocking and nonblocking execution outperform the serial execution. One exception is the blocking execution on the smallest dataset bloweybq, for which the 44 threads introduce high overhead and lead to performance degradation. In particular, the blocking execution outperforms the serial execution by about $2\times$ when only 11 threads are used. The nonblocking execution outperforms both the blocking and the serial execution even for 44 threads thanks to better data locality. The nonblocking execution with automatic configuration, by virtue of the performance model, chooses a smaller number of threads and achieves even better performance as shown in Table II. The experiments using the manual configuration for nonblocking execution confirm that the highest performance is achieved for a tiling size between 256 and 4,096. The results using automatic configuration achieve speedups up to $18.0\times$ and $3.93\times$ versus the serial and the blocking execution, respectively. Slowdown versus blocking execution again occurs only for bundle_adj, and otherwise re-confirm that choosing the number of threads and the tiling size according to a performance model leads to close-to-optimal performance compared to manual configuration.

IV. Related work

SuiteSparse:GraphBLAS exploits the nonblocking mode of GraphBLAS to delay mutations such as additions or subassignment to vectors and matrices. Operations that can cope with pending mutations by processing them separately will do so. Other operations will apply the mutations in bulk before performing the operation, thus allowing performance improvements to the cost of, e.g., element-wise matrix updates [8]. Such optimizations are orthogonal to the presented approach. Both are made possible by the nonblocking mode that allows delaying the execution of operations until after they are called. The nonblocking mode was first described by the initial GraphBLAS C specification [3].

The presented work executes a full pipeline of pending operations whenever output escapes the control of the GraphBLAS library, as also required by the C specification [3]. Mattson et al. [9] suggest programmers trigger part of a pipeline by explicitly synchronizing on a target container. This was incorporated into the C specification since version 1.3 and may be automated by a data dependence analysis phase.

Outside of GraphBLAS, loop fusion is a common technique, either part of compiler optimization passes or explicitly programmed using template meta-programming techniques such as done by Eigen [10]. Different from such static compiler techniques, the presented nonblocking implementation builds

Dataset	Pres_Poisson	gyro_m	van_body	G2_circuit	bundle_adj	adaptive	road_usa	twitter7
Vectors size	14,822	17,361	47,072	150,102	513,351	6,815,744	23,947,347	41,652,230
serial	65	39	245	111	2,716	5,758	71,896	1,088,195
blocking	17	18	23	24	150	542	4,030	30,994
nonblocking - $P = 44, T = 256$	11	9	16	23	211	608	4,650	30,109
nonblocking - $P = 44, T = 512$	10	9	16	18	297	452	3,547	28,859
nonblocking - $P = 44, T = 1,024$	12	10	19	17	487	418	3,302	28,750
nonblocking - $P = 44, T = 2,048$	16	12	21	16	1,096	402	3,206	28,778
nonblocking - $P = 44, T = 4,096$	26	17	32	18	2,098	401	3,167	28,825
nonblocking - $P = 44, T = 8,192$	46	25	57	24	2,151	405	3,155	28,872
nonblocking - $P = 44, T = 16,384$	85	34	108	37	2,154	419	3,170	28,888
automatic nonblocking	7	9	19	16	655	425	3,345	28,800
nonblocking vs blocking	2.43×	2.00×	1.21×	1.50×	0.23×	1.28×	1.20×	1.08×

TABLE I: Experimental results in milliseconds for various datasets of Pagerank.

Dataset	bloweybq	gyro_m	van_body	G2_circuit	bundle_adj	apache2	ecology2	Queen_4147
Vectors size	10,001	17,361	47,072	150,102	513,351	715,176	999,999	4,147,110
serial	2,111	5,257	44,834	24,295	453,714	61,991	104,960	6,968,331
blocking	2,817	2,188	4,483	5,479	36,581	8,917	15,730	408,478
nonblocking - $P = 44, T = 256$	1,241	934	2,567	2,979	39,473	6,264	10,885	392,962
nonblocking - $P = 44, T = 512$	1,268	839	2,707	2,518	52,035	4,950	8,971	384,850
nonblocking - $P = 44, T = 1,024$	1,323	988	3,100	2,270	80,834	4,767	8,444	385,204
nonblocking - $P = 44, T = 2,048$	1,450	1,297	3,360	2,368	167,721	4,714	8,288	388,442
nonblocking - $P = 44, T = 4,096$	1,793	1,883	5,387	2,723	307,226	4,690	8,294	384,979
nonblocking - $P = 44, T = 8,192$	2,196	2,779	9,597	3,631	302,168	4,788	8,753	386,000
nonblocking - $P = 44, T = 16,384$	3,444	3,742	18,112	5,342	308,869	5,028	10,197	389,181
automatic nonblocking	716	786	3,092	2,337	160,750	4,772	8,445	387,547
nonblocking vs blocking	3.93×	2.78×	1.45×	2.34×	0.23×	1.87×	1.86×	1.05×

TABLE II: Experimental results in milliseconds for various datasets of Conjugate Gradient.

pipelines dynamically and is able to construct pipelines that cross control flow statements.

V. Conclusion

We present a novel design and implementation for nonblocking execution in GraphBLAS. Lazy evaluation and dynamic parallelism are combined to perform automatic run-time 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 datasets of two algorithms written in GraphBLAS, by showing up to 3.93× speedup over blocking execution as a result of better cache utilization. The experimental results show that an analytical model provides a very good estimation for the number of threads and the tiling size, automatically achieving performance close to, or exceeding, those obtained with the best manual configuration.

References

[1] T. A. Davis, M. Aznaveh, and S. Kolodziej, "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*, 2019, pp. 1–6.

[2] A. V. Aho, J. E. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*. Pearson Education, 1974.

[3] A. Buluç, T. Mattson, S. McMillan, J. Moreira, and C. Yang, "The GraphBLAS C API Specification, v1.0," 2017. [Online]. Available: http://www.graphblas.org

[4] A. N. Yzelman, D. Di Nardo, J. M. Nash, and W. J. Suijlen, "A C++ GraphBLAS: specification, implementation, parallelisation, and evaluation," 2020, preprint. [Online]. Available: http://albert-jan.yzelman.net/PDFs/yzelman20.pdf

[5] A. Mastoras and T. R. Gross, "Efficient and Scalable Execution of Fine-Grained Dynamic Linear Pipelines," *ACM Transactions on Architecture and Code Optimization*, vol. 16, no. 2, pp. 8:1–8:26, 2019.

[6] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Transactions on Mathematical Software*, vol. 38, no. 1, pp. 1–25, Dec. 2011.

[7] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford Large Network Dataset Collection," http://snap.stanford.edu/data, jun 2014.

[8] T. A. Davis, "Algorithm 1000: SuiteSparse:GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra," *ACM Transactions on Mathematical Software*, vol. 45, no. 4, Dec. 2019.

[9] T. G. Mattson, C. Yang, S. McMillan, A. Buluç, and J. E. Moreira, "GraphBLAS C API: Ideas for future versions of the specification," in *2017 IEEE High Performance Extreme Computing Conference*, 2017, pp. 1–6.

[10] G. Guennebaud, B. Jacob *et al.*, "Eigen v3," http://eigen.tuxfamily.org, 2010.