Nonblocking execution in GraphBLAS

Aristeidis Mastoras
Computing Systems Laboratory
Zurich Research Center
Huawei Technologies, Switzerland

Sotiris Anagnostidis
Data Analytics Laboratory
Department of Computer Science
ETH Zurich, Switzerland

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 to improve data locality. In this paper, we present a preliminary evaluation for two algorithms, Pagerank and Conjugate Gradient, that confirms the importance of nonblocking execution, by showing promising speedups over the corresponding blocking execution.**

***Index Terms*—GraphBLAS, nonblocking execution, data locality, lazy evaluation, dynamic parallelism, loop fusion, loop tiling, analytic performance modeling**

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]. GraphBLAS represents a graph as an adjacency or incidence matrix, and it expresses graph operations as linear algebraic operations. For example, GraphBLAS represents the input graph of the Breadth-First Search (BFS) algorithm as a matrix A and the starting node s with a vector x. A BFS iteration is performed 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 min-plus semiring expresses a Shortest Path algorithm [2], while with a Boolean semiring it expresses an algorithm that finds all vertices within k hops from a source vertex. Figure 1 shows the implementation of the latter algorithm in the open-source C++11 ALP/GraphBLAS presented by Yzelman et al. [3].

GraphBLAS operations are memory bound, and the overall performance depends on the time spent on moving data to and from the main memory. The C11 GraphBLAS specification [4] 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::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.

In this paper, we evaluate the novel design of a nonblocking implementation for GraphBLAS. A preliminary evaluation for various matrices 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 matrices, we use an analytic 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 from the best manual configuration.

II. Nonblocking execution

We present the novel design and evaluation of a library-based implementation in ALP/GraphBLAS 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 fusion and loop tiling optimizations. *Dynamic parallelism* is achieved by grouping an arbitrary number of operations that reuse data in cache and by executing them in parallel.

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

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)
   for (func = begin(); func != end(); ++func)
      // calls the stored functions
      (*func)(i, min(i + tiling_size, N));
```

Fig. 2: Execution of operations by 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 containers in tiles of size `tiling_size`, and the inner loop iterates over all the stored functions. Loop tiling over sparse vectors requires keeping track of sparsity within every tile. Sparse accumulators achieve this.

B. Dynamic parallelism

Lazy evaluation expresses GraphBLAS operations as a sequence of stages that form a pipeline. 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 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. This simple design leads to good results according to the evaluation in Section III, since successive operations are usually part of the same computation.

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 initially did not define its own opaque data type for scalars. Therefore, the current design enforces the execution of the pipeline when the invoked operation returns a scalar, e.g., the dot operation which computes the scalar product of two vectors. The nonblocking execution can benefit from opaque scalars, which were added to the GraphBLAS C API specification since version 2.0.0.

C. Parallel execution of the pipeline

Stages of a pipeline often access shared data, and thus the inner loop in Figure 2 is executed sequentially. The execution of the iterations for the outer loop may overlap in time depending on whether the stages introduce cross-iteration dependences or not. Most GraphBLAS operations do not introduce any cross-iteration dependences, i.e., different iterations write on different elements of the containers, e.g., in the set operation, or any cross-iteration dependences are automatically eliminated, e.g., in the dot operation. For the two algorithms evaluated in this paper, the outer loop in Figure 2 does not exhibit any dependences and can be safely parallelized with `#pragma omp parallel for schedule(dynamic)`. 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.

D. Analytic performance model

Nonblocking execution as described thus far has two major parameters: a) the tiling size, and b) the number of threads used for parallel execution. Rather than tuning these parameters manually for different algorithms, systems, and matrices, we monitor at run-time the pipeline size, the number of containers involved, their sizes, and the size of the data type for their elements. 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 reuse private caches maximally, the resulting analytic performance model automatically derives a tiling size and a number of threads. These parameters depend on the pipeline, and thus different parameter pairs may be chosen 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 matrix, 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 utilizes 44 cores.

For the nonblocking execution, a number of threads P, with $1 \leq P \leq 44$, and a tiling size T are chosen automatically based on an analytic model, such that the L1 cache is fully utilized and every employed thread has sufficient work. To provide a clear picture about the performance of the analytic model, we experimented with 44 threads and a tiling size that varies from 256 to 16,384. Small tiling sizes show the performance when the L1 cache is not fully utilized and the overhead of invoking the stored functions may negate the performance. A tiling size larger than 4,096 shows the performance when the data do not fit in the L1 cache.

B. Case studies

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

For Pagerank, both the parallel blocking and the nonblocking execution modes significantly outperform the serial one for all matrices. The highest speedup obtained from parallel execution is observed for the largest matrix, i.e., $35.1\times$ over serial execution. For the same matrix, the nonblocking execution leads to an additional 8% improvement thanks to better cache utilization. The nonblocking execution achieves the highest speedup obtained from improved data locality for the gyro_m matrix, i.e., $2\times$ over blocking execution. Nonblocking execution achieves its highest performance with a tiling size between 256 and 4,096 for seven out of eight matrices. A larger tiling size does not allow the data to fit in the L1 cache.

For road_usa, a tiling size of 8,192 leads to the highest performance, which is close to that for a tiling size of 4,096. For the three smallest matrices, a small tiling size achieves the highest performance, since it results in a larger number of iterations for the pipeline, and more cores are utilized. For bundle_adj, the nonblocking execution achieves worse performance than the blocking one 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 that allows a more balanced distribution of the workload to threads. Although a tiling size that is smaller than 256 leads to an even more balanced distribution of the workload, the performance of the nonblocking execution does not approach that of the blocking execution. Preliminary investigations confirm that a smaller tiling size fails to amortize the invocation overhead of the stored functions.

The analytic model employed by the nonblocking execution achieves performance that is close to that of the best manual configuration, except for the bundle_adj matrix. For Pres_Poisson, the analytic model outperforms the best manual configuration, since the automatically chosen number of threads is smaller than 44 used by the manual configuration.

The results for Conjugate Gradient are similar to those for Pagerank. Both the blocking and the nonblocking execution outperform the serial one as a result of parallel execution, and the nonblocking execution outperforms the blocking one as a result of better cache utilization. Two exceptions are the bundle_adj matrix for the reasons explained earlier and the bloweybq matrix, for which 44 threads introduce high overhead and lead to performance degradation for the blocking execution. By using only 11 threads, the blocking execution outperforms the serial execution by about $2\times$. For the same matrix, the nonblocking execution outperforms both the blocking and the serial execution even for 44 threads, and the analytic model chooses a smaller number of threads and leads to even better performance. For Conjugate Gradient, the analytic model makes the nonblocking execution to automatically achieve up to $18.0\times$ and $3.93\times$ speedups over the serial and the blocking execution, respectively. Thus, the experimental results confirm that choosing the number of threads and the tiling size according to an analytic model leads to close-to-optimal performance for seven out of eight matrices.

IV. Related work

The nonblocking execution mode was first described by the initial GraphBLAS C API specification [4]. SuiteSparse:GraphBLAS exploits the nonblocking mode of GraphBLAS to delay mutations such as addition or sub-assignment 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 execution, thus allowing performance improvements to the cost of, e.g., element-wise matrix updates [8]. These optimizations are orthogonal to the presented approach. Mattson et al. [9] suggest programmers trigger part of a computation 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. The presented work executes pending operations whenever output escapes the control of ALP/GraphBLAS, as also required by the C specification [4].

Outside of GraphBLAS, loop fusion is either part of compiler optimization passes or explicitly done using template meta-programming techniques such as by Eigen [10]. The design of the presented nonblocking execution provides additional flexibility, since building pipelines dynamically, e.g., also accounts for for-loops or conditional breaks. Static techniques cannot cross such control-flow boundaries, except if explicit constructs were added to the specification.

Xiao et al. [11] present a framework for automatic parallelization of high-level programs that are partitioned into clusters and are mapped onto processing units. An optimization model is used for choosing the number of clusters and minimizing data communication. Data communication, load-balancing, and synchronization overhead are important aspects for the efficiency of automatic parallelization [12], [13]. Minimizing data communication for the nonblocking execution is simpler than for programs with arbitrary data dependences, since the pipelines built by the algorithms considered in this work do not include cross-iteration dependences. Asymmetries in the sparsity pattern of vectors and matrices cause load-imbalance, which is handled by dynamic scheduling.

Matrices	Pres_Poisson	gyro_m	vanbody	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
speedups of blocking vs serial	3.82×	2.17×	10.7×	4.63×	18.1×	10.6×	17.8×	35.1×
speedups of best manual nonblocking vs blocking	1.70×	2.00×	1.44×	1.50×	0.71×	1.35×	1.28×	1.08×
speedups of automatic 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 Pagerank and various matrices.

Matrices	bloweybq	gyro_m	vanbody	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
speedups of blocking vs serial	0.75×	2.40×	10.0×	4.43×	12.4×	6.95×	6.67×	17.1×
speedups of best manual nonblocking vs blocking	2.27×	2.61×	1.75×	2.41×	0.93×	1.90×	1.90×	1.06×
speedups of automatic 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 Conjugate Gradient and various matrices.

V. Conclusion

We present and evaluate the novel design of a nonblocking implementation for GraphBLAS. Lazy evaluation and dynamic parallelism are combined to perform automatic run-time optimizations that a) improve data locality and b) achieve efficient parallel execution. The evaluation confirms the efficiency of nonblocking execution for various matrices of two algorithms; nonblocking execution achieves up to $2.61\times$ speedup over the blocking execution as a result of better cache utilization. An analytic performance model automatically chooses the number of threads and the tiling size, making the nonblocking execution achieve up to $3.93\times$ speedup over the blocking execution. Aside from adversarial nonzero structures, the automatic performance achieved by using the presented analytic model is close to that of 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. 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

[4] A. Buluç, T. Mattson, S. McMillan, J. Moreira, and C. Yang, "The GraphBLAS C API Specification, version 1.0," 2017. [Online]. Available: http://www.graphblas.org

[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, Jun. 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. 1:1–1:25, Nov. 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, pp. 44:1–44:25, 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 *et al.*, "Eigen v3," 2010. [Online]. Available: http://eigen.tuxfamily.org

[11] Y. Xiao, S. Nazarian, and P. Bogdan, "Plasticity-on-Chip Design: Exploiting Self-Similarity for Data Communications," *IEEE Transactions on Computers*, vol. 70, no. 6, pp. 950–962, Jun. 2021.

[12] G. Ma, Y. Xiao, T. L. Willke, N. K. Ahmed, S. Nazarian, and P. Bogdan, "A Distributed Graph-Theoretic Framework for Automatic Parallelization in Multi-Core Systems," in *Proceedings of Machine Learning and Systems*, vol. 3, 2021, pp. 550–568.

[13] R. H. Bisseling, B. O. Fagginger Auer, A. N. Yzelman, T. van Leeuwen, and Ü. V. Çatalyürek, "Two-dimensional approaches to sparse matrix partitioning," in *Combinatorial Scientific Computing*, U. Naumann and O. Schenk, Eds. CRC Press, 2012, ch. 4, pp. 95–127.