

# SPLA: Portable Generic Sparse Linear Algebra Library for Multi-GPU Computations

1<sup>st</sup> Egor Orachev

*Faculty of Mathematics and Mechanics  
St. Petersburg State University,  
Programming Languages and Tools Lab  
JetBrains Research  
St. Petersburg, Russia  
egor.orachev@gmail.com  
0000-0002-0424-4059*

2<sup>nd</sup> Gleb Marin

*School of Physics, Mathematics,  
and Computer Science  
HSE University  
St. Petersburg, Russia  
glebmar2001@gmail.com  
0000-0002-0873-1647*

3<sup>rd</sup> Semyon Grigorev

*Faculty of Mathematics and Mechanics  
St. Petersburg State University,  
Programming Languages and Tools Lab  
JetBrains Research  
St. Petersburg, Russia  
s.v.grigoriev@spbu.ru  
semyon.grigorev@jetbrains.com  
0000-0002-7966-0698*

**Abstract**—Scalable high-performance graph analysis is an actual nontrivial challenge and usage of sparse linear algebra operations as building blocks for graph analysis algorithms, which is a core idea of GraphBLAS standard, is a promising way to attack it. While it is known that sparse linear algebra operations can be efficiently implemented on GPGPU, full GraphBLAS implementation on GPGPU is a nontrivial task that is almost solved by GraphBLAST project. Though it is shown by GraphBLAST that utilization of GPGPUs for GraphBLAS implementation significantly improves performance, portability and scalability problems are not solved yet: GraphBLAST uses Nvidia stack and utilizes only one GPGPU. In this work we propose an SPLA library that aimed to solve these problems: it uses OpenCL to be portable and designed to utilize multiple GPGPUs. Preliminary evaluation shows that !!!!

**Index Terms**—graphs, algorithms, graph analysis, sparse linear algebra, GraphBLAS, GPGPU, OpenCL

## I. INTRODUCTION

Scalable high-performance graph analysis is an actual challenge. There is a big number of ways to attack this challenge [1] and the first promising idea is to utilize general-purpose graphic processing units (GPGPU-s). Such existing solutions, as CuSha [2] and Gunrock [3] show that utilization of GPUs can improve the performance of graph analysis, moreover it is shown that solutions may be scaled to multi-GPU systems. But low flexibility and high complexity of API are problems of these solutions.

The second promising thing which provides a user-friendly APY for high-performance graph analysis algorithms creation is a GraphBLAS API [4] which provides linear algebra based building blocks to create graph analysis algorithms. The idea of GraphBLAS is based on is a well-known fact that linear algebra operations can be efficiently implemented on parallel hardware. Along with this, a graph can be natively represented using matrices: adjacency matrix, incidence matrix, etc. While reference CPU-based implementation of GraphBLAS, SuiteSparse:GraphBLAS [5], demonstrates good performance in real-world tasks, GPU-based implementation is challenging.

One of the challenges in this way is that real data are often sparse, thus underlying matrices and vectors are also sparse, and, as a result, classical dense data structures and respective algorithms are inefficient. So, it is necessary to use advanced data structures and procedures to implement sparse linear algebra, but the efficient implementation of them on GPU is hard due to the irregularity of workload and data access patterns. Though such well-known libraries as cuSparse show that sparse linear algebra operations can be efficiently implemented for GPGPU-s, it is not so trivial to implement GraphBLAS on GPGPU. First of all, it requires *generic* sparse linear algebra, thus it is impossible just to reuse existing libraries which are almost all specified for operations over floats. The second problem is specific optimizations, such as maskings fusion, which can not be natively implemented on top of existing kernels. Nevertheless, there is a number of implementations of GraphBLAS on GPGPU, such as GraphBLAST: [6], GBTL [7], which show that GPGPUs utilization can improve the performance of GraphBLAS-based graph analysis solutions. But these solutions are not portable because they are based on Nvidia Cuda stack. Moreover, the scalability problem is not solved: all these solutions support only single-GPU, not multi-GPU computations.

To provide portable multi-GPU implementation of GraphBLAS API we developed a *SPLA* library (sources are published on GitHub: <https://github.com/JetBrains-Research/spla>). This library utilizes OpenCL for GPGPU computing to be portable across devices of different vendors. Moreover, it is initially designed to utilize multiple GPGPUs to be scalable. To sum up, the contribution of this work is the following.

- Design of portable multi-GPU GraphBLAS implementation proposed. Additionally, proposed design is aimed to simplify library tuning and wrappers for different high-level platforms and languages creation.
- Subset of GraphBLAS API, including such operations as masking, matrix-matrix multiplication, matrix-matrix e-wise addition, is implemented. Current implementation is limited by COO matrix representation format and

uses basic algorithms for some operations, but work in progress and more data formats will be supported and advanced algorithms will be implemented in the future.

- Preliminary evaluation on such algorithms as breadth-first search (BFS) and single source shortest path (SSSP) and real-world graphs shows that !!!

## II. SOLUTION DESCRIPTION

### A. Design Principles

SPLA library is designed in the way to maximize potential library performance, simplify its implementation and extensions, and to provided the end-user verbose, but effective interface allowing customization and precise control over operations execution. These ideas are captured in the following principles.

- *DAG-based expressions.* User constructs a computational expression from basic nodes and uses oriented edges to describe data dependencies between these nodes.
- *Automated hybrid-storage format.* Library uses internally specialized preprocessing to format data and automate its sharing between computational nodes.
- *Automated multi-GPU scheduling.* Computational work is automatically scheduled between available devices for execution. Scheduling order, dependencies and granularity are defined from DAG expression, submitted by user.
- *Customization of primitive types and operations.* Underlying primitives types and functions can be customized by user. Customization process does not requires library re-compilation.
- *Exportable interface.* Library has C++ interface with an automated reference-counting and with no-templates usage. It can be wrapped by C99 compatible API and exported to other languages, for example, in a form of a Python-package.

### B. Architecture Overview

Library general execution architecture is depicted in Fig. 1. As an input library accepts expression composed in the form of a DAG. Nodes represent fundamental operations, such as matrix-matrix multiplication. Links describe dependencies between nodes. Expression execution is *asynchronous*. User can block and wait until its completion, or without blocking probe the expression until it is either *completed* or *aborted*.

Expression is transformed into a task graph. Task graph is submitted for execution to the task manager. Each task is processed by specialised *NodeProcessor*, capable of processing particular node type. Each task, when executed, is split dynamically into a set of parallel sub-tasks. Each sub-task is processed by specialized *Algorithm*, which is capable of processing input blocks of matrices or vectors in particular storage formats with specific set of options. *NodeProcessor* and *Algorithm* are selected at runtime from a registry using properties and arguments of the expression. Thus, it allows precise processing and optimization of edge-cases.

Granularity level of sub-tasks is defined by the structure of underlying processed primitives. Target device for execution

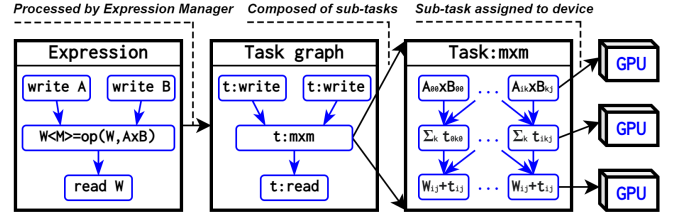


Fig. 1. Library expression processing architecture.

is automatically assigned for the sub-task based on expression and node parameters. Currently, the fixed and uniform distribution for assignment is supported.

### C. Containers

Library provides general *M-by-N Matrix*, *N Vector* and *Scalar* primitives. Underlying primitives types is specified by *Type* object. Internally primitives are stored in a hybrid storage in a form of two- or one- dimensional blocks' grid form matrices and vectors respectively. Each block is empty (not stored) or store some data in any format. Blocks are immutable, they can be safely shared across computational devices.

Currently COO/CSR blocks for matrices, and COO/Dense blocks for vectors are supported. Format choice is motivated by its simplicity and easy of implementation. Other formats, such as CSC, DCSR, ELL, etc. can be added to the library by implementation of formats conversion or by the specialization of *Algorithm* for a specific format.

### D. Algebraic Operations

Library supports all commonly used linear algebra operations, such as *mxm*, *vxm*, *eadd*, *reduce*, *transpose*. Other operations are coming soon, since library still in development. Interface of operations is designed *similar* to GraphBLAS. It supports *masking*, *accum* of the result, *add* and/or *mult* user-functions specification, and *descriptor* object for additional operation tweaking.

### E. Implementation Details

Library uses OpenCL 1.2 API as underlying compute API. Boost Compute [8] is utilized as a high-level library on top of the OpenCL functionality. It provides thread-safe kernel caching, meta-kernel programming, and a set of basic parallel primitives such as *device vector*, *sort*, *reduce*, *scan*, etc. which was extended further to meet this project requirements. Taskflow [9] is used as tasking library. It supports task-dependencies and dynamic tasking, utilized in order to create and execute sub-tasks.

User-defined *Types* are represented as POD-structures, and handled by the library as a fixed-size sequences of bytes. User-defined *Functions* are effectively textual strings with OpenCL code, injected into generalized meta-kernels. Library has a number of predefined types, such as *signed/unsigned integers*,

floating point types, and a set of common operations, such as *arithmetic*, *logic*, *first/second*, etc.

For particular SpVSpM implementation ESC algorithm [10] is employed. Masked SpGEMM is based on Carl Yang et al. work [6]. Tiled GPU merge path [11] algorithm utilized for element-wise addition and masking implementation. The code is generalized and written in a form of meta-kernels, so actual functions for elements reduction or multiplication are injected later. Kernel compilation is done on demand if no previously cached entry present.

### III. EVALUATION

For performance analysis of proposed solution we evaluated some most common graph algorithms using real-world sparse matrix data. As a baseline for comparison we chose LAGraph [12] in connection with SuiteSparse [5] as a CPU tool, Gunrock [3] and GraphBLAST [6] as a Nvidia GPU tools. Also, we tested algorithms on several devices with distinct OpenCL vendors in order to validate portability of the proposed solution. In general, these evaluation intentions are summarized in the following research questions.

- RQ1** What is the performance of the proposed solution relative to existing tools for both CPU and GPU analysis?
- RQ2** What is the portability of the proposed solution with respect to various device vendors and OpenCL runtimes?

#### A. Graph Algorithms

For preliminary study *breadth-first search* (bfs) and *triangles counting* (tc) algorithms were chosen, since they allows analyse the performance of *vxm* and *mxm* operations, rely heavily on *masking*, and utilize *reduction* or *assignment*. Algorithms are implemented following Carl Yang et al. [6] work.

#### B. Dataset

For experiments, eight graph matrices were selected from the Sparse Matrix Collection at University of Florida [13]. Information about graphs is summarized in Table I. These graphs are widely used in various sparse matrix benchmarks, and they are well suited for performance study of selected algorithms.

TABLE I  
DATASET DESCRIPTION.

Dataset	Vertices	Edges	Max Degree
coAuthorsCiteseer	227.3K	1.6M	1372
coPapersDBLP	540.4K	30.4M	3299
hollywood-2009	1.1M	113.8M	11,467
roadNet-CA	1.9M	5.5M	12
com-Orkut	3M	234M	33313
cit-Patents	3.7M	16.5M	793
soc-LiveJournal	4.8M	68.9M	20,333
indochina-2004	7.5M	194.1M	256,425

#### C. Evaluation Setup

For evaluation, we use a PC with Ubuntu 20.04 installed, which has Intel Core i7-6000 4-core CPU, 3.40Hz, DDR4 64Gb RAM, and Nvidia GeForce GTX 1070 GPU with 8Gb VRAM. Host programs were compiled with GCC 9.3.0 compiler. Programs using CUDA were compiled with GCC 8.4.0 and Nvidia NVCC 10.1.243 compiler. Release mode and maximum optimization level was enabled for all tested programs. Data loading time, preparation, format transformations and host-device initial communications are excluded from time measurements. All tests run 10 times with extra warm-up run.

#### D. Results

##### Single GPU

TABLE II  
GRAPH ALGORITHMS EVALUATION RESULTS.

Dataset	SuiteSparse	Gunrock	GraphBLAST	Spla
BFS				
hollywood-2009	0.0	0.0	0.0	0.0
roadNet-CA	0.0	0.0	0.0	0.0
com-Orkut	0.0	0.0	0.0	0.0
soc-LiveJournal	0.0	0.0	0.0	0.0
indochina-2004	0.0	0.0	0.0	0.0
TC				
coAuthorsCiteseer	0.0	0.0	0.0	0.0
coPapersDBLP	0.0	0.0	0.0	0.0
roadNet-CA	0.0	0.0	0.0	0.0
com-Orkut	0.0	0.0	0.0	0.0
soc-LiveJournal	0.0	0.0	0.0	0.0

Analysis and conclusion.

### IV. CONCLUSION

In this paper we present portable generic sparse linear algebra library for GPU computation, which provides sparse primitives and supports set of common operations. Evaluation of proposed solution for real-world graph analysis shows, that initial OpenCL-based operations implementation with limited set of operations has acceptable performance compared to other tools, but can be easily executed on devices of other vendors, what gives significant flexibility in choose of HPC hardware. Since project still in the active development, the following major and important development tasks must be highlighted.

- *Operations*. New mathematical operations must be implemented, as well as required and most used data manipulation, row/column management, filtering, selection and transformation subroutines must be supported.
- *Performance tuning*. State of the art high performance algorithms must be implemented for such operations, as SpMSpV, SpMV, SpGEMM, etc. in order to increase library efficiency. Also, different techniques, such as automated storage format transitions, pull-push direction optimization, etc. must be employed for better performance on a large range of data.
- *Graph algorithms*. Algorithms such as *single-source shortest paths*, *page rank*, *connected components*, etc.

must be implemented using library API and their performance must be analysed as well.

- *API exporting.* Library interface must be exported to other programming languages, such as C or Python. Finally, Python-package for applied graph analysis must be published.
- *Multi-GPU scheduling.* Currently library supports scheduling only on single GPU. But, its internal hybrid storage format and tasking allow multi-node execution. Thus, multi-GPU scheduling and precise data sharing must be implemented and performance analysis of the solution must be carried on.

Finally, we plan to study the characteristics of the implemented library with all algorithms and improvements, and compare its performance to existing tools using uniform benchmarking platform such as a Graphalytics [14].

## REFERENCES

- [1] M. E. Coimbra, A. P. Francisco, and L. Veiga, “An analysis of the graph processing landscape,” *Journal of Big Data*, vol. 8, no. 1, Apr. 2021. [Online]. Available: <https://doi.org/10.1186/s40537-021-00443-9>
- [2] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan, “Cusha: Vertex-centric graph processing on gpus,” in *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC ’14. New York, NY, USA: Association for Computing Machinery, 2014, p. 239–252. [Online]. Available: <https://doi.org/10.1145/2600212.2600227>
- [3] Y. Pan, Y. Wang, Y. Wu, C. Yang, and J. D. Owens, “Multi-gpu graph analytics,” in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017, pp. 479–490.
- [4] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, S. McMillan, C. Yang, J. D. Owens, M. Zalewski, T. Mattson, and J. Moreira, “Mathematical foundations of the graphblas,” in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, 2016, pp. 1–9.
- [5] T. A. Davis, “Algorithm 1000: Suitesparse:graphblas: Graph algorithms in the language of sparse linear algebra,” *ACM Trans. Math. Softw.*, vol. 45, no. 4, Dec. 2019. [Online]. Available: <https://doi.org/10.1145/3322125>
- [6] C. Yang, A. Buluc, and J. D. Owens, “Graphblast: A high-performance linear algebra-based graph framework on the gpu,” 2019.
- [7] P. Zhang, M. Zalewski, A. Lumsdaine, S. Misurda, and S. McMillan, “Gbtl-cuda: Graph algorithms and primitives for gpus,” in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016, pp. 912–920.
- [8] J. Szuppe, “Boost.compute: A parallel computing library for c++ based on opencl,” in *Proceedings of the 4th International Workshop on OpenCL*, ser. IWOCCL ’16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: <https://doi.org/10.1145/2909437.2909454>
- [9] T.-W. Huang, D.-L. Lin, C.-X. Lin, and Y. Lin, “Taskflow: A lightweight parallel and heterogeneous task graph computing system,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, pp. 1303–1320, 2022.
- [10] S. Dalton, L. Olson, and N. Bell, “Optimizing sparse matrix—matrix multiplication for the gpu,” *ACM Trans. Math. Softw.*, vol. 41, no. 4, oct 2015. [Online]. Available: <https://doi.org/10.1145/2699470>
- [11] O. Green, R. McColl, and D. A. Bader, “Gpu merge path: A gpu merging algorithm,” in *Proceedings of the 26th ACM International Conference on Supercomputing*, ser. ICS ’12. New York, NY, USA: Association for Computing Machinery, 2012, p. 331–340. [Online]. Available: <https://doi.org/10.1145/2304576.2304621>
- [12] G. Szárnyas, D. A. Bader, T. A. Davis, J. Kitchen, T. G. Mattson, S. McMillan, and E. Welch, “Lagraph: Linear algebra, network analysis libraries, and the study of graph algorithms,” 2021.
- [13] T. A. Davis and Y. Hu, “The university of florida sparse matrix collection,” *ACM Trans. Math. Softw.*, vol. 38, no. 1, dec 2011. [Online]. Available: <https://doi.org/10.1145/2049662.2049663>
- [14] A. Iosup, A. Musaafer, A. Uta, A. P. Pérez, G. Szárnyas, H. Chafi, I. G. Tănase, L. Nai, M. Anderson, M. Capotă, N. Sundaram, P. Boncz, S. Depner, S. Heldens, T. Manhardt, T. Hegeman, W. L. Ngai, and Y. Xia, “The ldbc graphalytics benchmark,” 2021.