Saint Petersburg State University

Egor Orachev

Master's Thesis

Generalized sparse linear algebra framework for GPU computations

Education level: master

Speciality *09.04.04 «Software Engineering»*

Programme *CB.5666.2021 «Software Engineering»*

Scientific supervisor:
C.Sc., docent S.V. Grigorev

Reviewer:

Saint Petersburg
2022

Contents

Introduction

Graph model is a natural way to structure data in a number of a real practical tasks, such as databases queries evaluations [2], graph databases [27], social networks analysis [22], RDF data analysis [5], bioinformatics [26] and static code analysis [13].

In the graph model the entity is represented as a graph vertex. Relations between entities are directed labeled edge. Such a model allows to model the domain of the analysis with a little effort, saving complex relationships between objects. What is not easy and clear to do, for example, in a classic *relational* model, based on tables.

There is a number of real-world practical graphs for analysis, which have sparse structure, where the number of edges has the same order as the number of vertices. Thus, for practical analysis specialized tools are required. These tools must provide efficient data layout in memory. Since a graph can count tens and hundreds millions edges [22], such tools must provide parallel processing possibility with utilization of multiple computational units or of specialized acceleration device.

In the last decade the research community have published a number of works, which covers the topic of efficient graph data analyse. It is worth to mentions such tools as Gunrock [16], Ligra [29], GraphBLAST [31], SuiteSparse [9], etc. Existing solutions provide different execution strategies and models, use for acceleration GPUs or multi-core CPUs.

According to Yang et al. [31] and Orachev et al. [32], a graphics processing unit usage is a promising way to a high-performance graph data analysis. However, practical algorithms' implementation, as well as development of a libraries for a graph data analysis, is a non-trivial task. Reasons for this problem are the following.

- **Complex APIs**. Frameworks for GPU programming, such as Nvidia CUDA or OpenCL, have a low-level nature. These are verbose APIs, which requires a lot of auxiliary operations, as well as careful management of resources and synchronization points. What makes then inaccessible of an ordinary programmer or data analyst.

- **Different algorithms**. Graph algorithms have similar structure, but differ in details. It requires *ad hoc* tuning for each algorithm instance with utilization of local optimizations. What makes these solution less reusable and forces to implement each algorithm from scratch.
- **Workload imbalance**. Multi-core CPUs and GPUs require even work distribution for better computational blocks occupation. What is not a trivial tasks due to sparsity of analysed data.
- **Irregular access patterns**. In general, graph algorithms has high memory-access and low computational (arithmetic) intensity. Thus, the memory access can be a bottleneck. Therefore, specialized data structures must be utilized.

In order to solve the mentioned above issues, the research community suggested a promising concept, which relies on a sparse linear algebra for a graph algorithms expression. This is a GraphBLAS [19] standard. This standard provides C API, allows to express graph algorithms as set of operations over vectors and matrices. Efficient implementation of these primitives and operations allows one to get a ready for usage implementation of an algorithms, without a deep knowledge of a low-level programming.

There is a number of GraphBLAS implementations and works, inspired by its ideas. However, there is still no implementation of GraphBLAS for a wide class of graphics processing units, since existing implementations focus on CPUs or Nvidia GPUs only, missing Intel and AMD devices. Also, existing tools are a bit limited in the performance and in a customization of operations. Thus, it is an import tasks of a sparse linear algebra primitives and operations library implementation, with support for generalization of operations for used defined types, as well as with support for computations on GPUs.

1 Problem statement

The goal of this work is the implementation of the generalized sparse linear algebra primitives and operations library for GPU computations. The work can be divided into the following tasks.

- Conduct the survey of existing solutions.
- Conduct the survey of instruments and technologies for GPU programming.
- Develop the architecture of the library.
- Implement the library accordingly to the developed architecture.
- Implement a set of most common graph algorithms using library.
- Conduct experimental study of implemented artifacts.

2 Background of study

This work is related to three major topics: graph analysis, sparse linear algebra and GPU computations. This section gives an overview for this topics, as well provides a survey of existing solutions and gives a brief introduction to a GPU programming.

2.1 Graph analysis models

Large-scale graph processing frameworks for multi-core CPUs, distributed memory CPUs systems, and GPUs, accordingly to a Batari et al. [18] and Shi et al. [15] survey, can be classified into following categories.

Vertex-centric. Vertex-centric model is based on a parallel processing of a graph vertices. It is introduces in Pregel [25] framework. This framework has a similar to a *map-reduce* concept and message passing. The whole processing is divided into a number of iterative step. In the beginning, all vertices are active. The the superstep of processing occurs. After this step, the runtime collects messages from all vertices, and determines vertices for the next superstep. Supersteps are synchronized in the way, that the next step is started only after previous one is completed. The significant drawback of this model is that these synchronization points introduce GPU overhead. So, some GPU blocks can be stalled, while others are still working.

Edge-centric. Edge-centric model focuses on edges processing rather then vertices. It is first introduced in X-Stream [28] project. This model streams sequentially edges of the graphs, allows to effectively process them. However, it suffers from a random access patters, when access to the vertices is required. Sequential access pattern to edges can be important, when data is loaded from the hard or solid state driver.

Linear-algebra based. Linear algebra based frameworks for a graph processing originate from a CombinationalBLAS [3] project, which introduced primitives for a large graph data analysis for a distributed memory CPU systems.

Linear algebra approach relies on the fact, that the graph traversal can be represented as matrix-vector multiplication as shown in figure 1. The graph is stored in an adjacency matrix A. The set of active vertices, also called *frontier*, is represented as a vector v, with non-zero elements for vertices of the front. Transposed matrix A multiplied by a vector v on the right gives a new frontier with active vertices for the next iteration. In order to traverse all vertices only once, we have to store additional vector with visited vertices. This vector can be used in an inverse element-wise multiplication to filter out those vertices from the frontier, which are already visited.

This is a fundamental concept, which is lying in the most graph traversal based algorithms, such as breadth-first search or single source shortest paths. This method can be extended even further if we consider a multiple-source traversal. In this we have a number of frontier vectors, which can be stored as a matrix. It allows us use matrix-matrix product for a such task.

The graph analysis community has formalized this method in a GraphBLAS [19] standard, which provides linear algebra primitives in a form of C API. This standard has a number of implementations, including the first reference implementation SuiteSparse [9], GrapbBLAS template library GBTL [14], Huawei GraphBLAS implementation [4], etc.

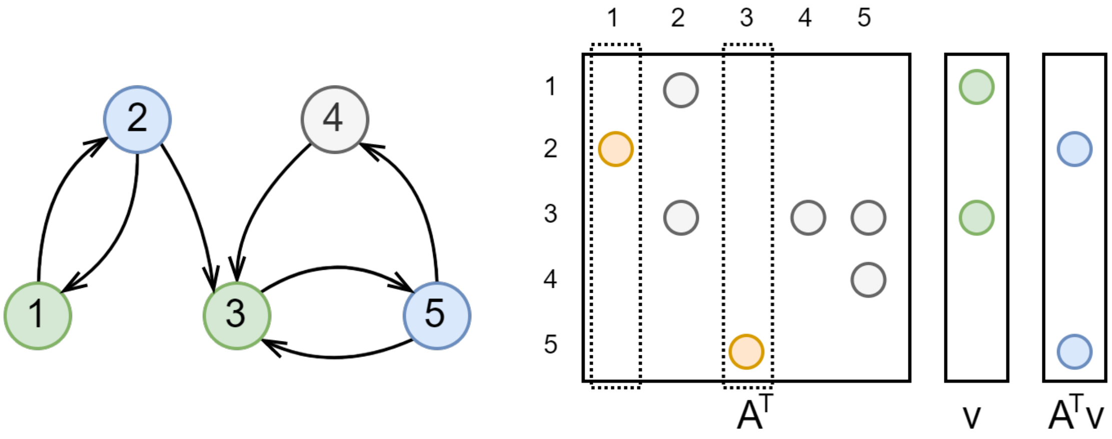

Figure 1: Graph traversal by matrix-vector product.

2.2 GraphBLAS concepts

GraphBLAS standard [19] is a mathematical notation translated into a C API. This standard provides sparse linear algebra build blocks for the implementation of graph algorithms in term of matrix and vector operations. The essential parts of this standards are the following.

Data containers. Primary data containers in this standard are general M by N matrix and M vector of values, as well as a scalar value. Containers are parameterised by the type of stored elements. The standard provides the ability to declare custom user defined types.

Matrix is used to represent the adjacency matrix of the graph. Vector is used to store a set of active vertices for traversal purposes. The scalar value is used to extract edge data from the graph or to aggregate the data across multiple edges.

Algebraic structure. Primary algebraic structures are called semiring and monoid, where two or one operation is provided respectively with some semantic requirements, such as associativity, commutativity, etc. This structures are adapted for a sparse graph analysis, so its mathematical properties differ a bit from those, which are stated in class algebra.

This structures define the element-wise operations, which work with elements in the containers. For example, they are passed as a parameters *mult* and *add* in the matrix product, where elements for row and column are multiplied, and then reduced to the final element.

There is a number of semirings, which can be used to solve different types of problems. For example, consider *MinPlus* semiring $\langle min, +, \mathbb{R} \cup \{+\infty\}, +\infty \rangle$, for a shortest path problem solving, where:

- Min used to aggregate distances and select the smallest one.
- Plus used to concatenate distances between two vertices.
- The domain is all real values with plus infinity.
- The identity element is infinity, what marks unreached vertices.

Programming constructs. GraphBLAS provides extra objects, which are required for practical algorithms implementation. One of the is a concept of the mask. Any matrix or vector can be used as a mask, which structure defines the structure of the result. It is a crucial and essential concept, since in many cases we are interested only in a partial result, not the whole matrix or vector. Mask is passed as extra argument, and implementation is free to make the fusion of the mask into the operation.

Another important construct is a descriptor. Descriptor is a set of named parameters and associated with them values. Descriptor used to tell the implementation, that, for example, mask complementary patter required, or result must not be accumulated with old content. This concept can be extended further, what is done in some GraphBLAS extensions.

Operations. GraphBLAS provides a number of commonly used linear algebra operations, such as matrix-vector and matrix-matrix products, transpose, element-wise multiplication. Also, there are some extra operations, which are more familiar for experienced developers, such as filtering, selection using predicate, reduction of matrix to vector or of vector to scalar, etc.

The important concept of the GraphBLAS operations is shown in the figure 2. For example, we can consider matrix-vector and matrix-matrix product operations, called *mxv* and *mxm* respectively. From the users perspective, they only have to use these operations, when the implementation is free to select the best algorithm, which fits the sparsity of the input arguments.

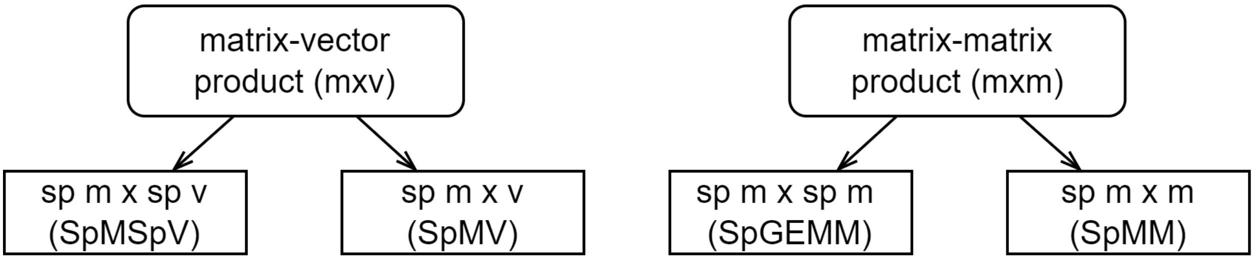

Figure 2: Key operations of the GraphBLAS standard and their implementations.

Algorithms. Using GraphBLAS constructs it is possible to write generalized graph analysis algorithms. There is a number of common and well-known graph algorithms, such as breadth-first search (BFS), single-source shortest path (SSSP), triangles counting (TC), connected components (CC), etc. which have a linear-algebra based formulation, described by Kepner et al. [17].

For example, consider a procedure with BFS algorithm in listing 1. As arguments it accepts vector v to store levels of reached vertices, adjacency matrix A, index of the start vertex s, and number of graph vertices n. Algorithm starts in lines **5 − 9** with initialization of result vector. Also, it allocates vector q, which is used as a *frontier* of currently active vertices to make a traversal step. The primary traversal loop in lines **14 − 19** of the algorithm works while the frontier has at least one active vertex. In the body, it updates current traversal level. Then, it assigns current level to currently reached vertices in the frontier in line **16** using *apply* function. Then in the line **17** it makes traversal step to find all children of current frontier vertices. Note, it uses inverted v as a mask with *GrB_DESC_RC* to filter our already visited vertices.

2.3 Existing frameworks

There is a number of libraries and frameworks for a graph data analysis on multi-core CPUs and GPUs. Some of them implement the GraphBLAS standard. Also, there are some works, which are inspired by the standard idea to utilize sparse linear algebra apparatus of data analysis. It this section the most significant contribution of the research community is covered.

Gunrock. Gunrock [16] is a state-of-the-art Nvidia GPU high level graph analytical system with support for both vertex-centric and edge-centric processing. It provides fine-grained runtime load balancing, does not requires any preprocessing of the input data, supports multi-GPU execution. Gunrock is written using Nvidia Cuda C++. It provides template based model for particular algorithms specializations.

Listing 1 Breadth-first search using GraphBLAS API

```
#include ’’GraphBLAS.h’’

GrB_Info BFS(GrB_Vector *v, GrB_Matrix A, GrB_Index s, GrB_Index n)
{
  GrB_Vector_new(v, GrB_INT32, n);

  GrB_Vector q;
  GrB_Vector_new(&q, GrB_BOOL, n);
  GrB_Vector_setElement(q, true, s);

  int32_t level = 0;
  GrB_Index nvals;

  do {
    ++level;
    GrB_apply(*v, GrB_NULL, GrB_PLUS_INT32, GrB_SECOND_INT32, q, level, GrB_NULL);
    GrB_vxm(q, *v, GrB_NULL, GrB_LOR_LAND_SEMIRING_BOOL, q, A, GrB_DESC_RC);
    GrB_Vector_nvals(&nvals, q);
  } while (nvals);

  GrB_free(&q);

  return GrB_SUCCESS;
}
```

Authors claim, that this framework is high-level from particular algorithms implementation perspective. However, it requires the knowledge of the Cuda API in order to extend this framework primitives for a new algorithm. Also, this framework utilizes a number of ad hoc optimizations. So it is limited in it a generalization.

SuiteSparse. GraphBLAS SuiteSprase [9] is a reference fully featured GraphBLAS implementation for mutli-core CPU computations. It is written using C language and OpenMP. Library is fully compatible with GraphBLAS API standard. It is available for C and C++ programs usage. Also, it provides a number of officially and unofficially supported packages, which export the functionality into other runtime, such as Java or Python (via pygraphblas [33]).

At this moment, the work is done in the project in order to support Nvidia Cuda for GPU computations. Also, the project uses a *pre-generation* approach, so it generate all possible combinations of kernels for all possible use-cases, since the is no way to provide template meta programming through raw C API.

GraphBLAST. GraphBLAST [31] library provides set of sparse linear algebra primitives and operations for computation on a single Nvidia GPU device. This project follows the GraphBLAS concepts. However, it provides C++ header-only interface and utilizes template meta programming along with Cuda C++ in order to support user types and functions customization.

Usage of a such API makes sense. It simplifies practical algorithms implementation, allows to offload the compiler with routine code generation work. However, this approach suffers from the header-only nature, since the whole library must be compiled for each executable file. What forces the end user to work with C++ compiler and actual Nvidia Cuda compiler, and recompile the whole project for each modification.

At this moment the project is in an active phase of the development. Authors of the project published the corresponding research report on the thematic conference. But, the stable and production-ready solution with full functionality is still unavailable.

Cusp. Cusp [8] library is a set of sparse linear algebra primitives with multiple backends support, such as multi-core CPU or Nvidia GPU. Library has a C++ template based interface. It utilizes meta programming for operations parametrization. It is based on a Nvidia Thrust [20], which provides realization for fundamental operations, such as *sort*, *scan*, *gather*, *reduce*, etc.

Although cusp is not inspired by GraphBLAS, it has similar to GraphBLAST interface. Also, it doesn't consider only graph analysis. Thus, it missing some important optimisations, which are done in GraphBLAST in order to speedup graph traversal. Also, library missing some important operations, such as matrix or vector reduction.

Cubool. Cubool [32] project is an attempt to customize GraphBLAS primitives to boolean algebra usage only. There is a number of algorithms, which can be efficiently implemented using sparse boolean linear algebra.

For example, it is graph reachability problem with a regular or context-free constrained path querying [1, 6, 7, 12].

The library uses Nvidia Cuda API for GPU computations. It provides a pycubool [24] package for a work in a Python runtime. This project is developed as part of this research in 2021. This library can be used as a foundation for this project. However, it has a number of fundamental limitations. So, it is not possible to generalize it for arbitrary data types.

2.4 Limitations

This section highlights the most critical limitations of existing solutions.

Imperative interface. GraphBLAS standard and other libraries have similar imperative interface. The user is supposed to define the algorithms as simple sequence of procedures calls, where operations are executed one after another. This approach simplifies user experience at the cost of the performance. Some algorithms have non-trivial data dependencies, have some steps, which can be done in parallel. Thus, the library must have enough information about algorithm structure in order to execute in the most efficient way.

Templates usage. There is a number of libraries which implement GraphBLAS in a form of C++ interface. These libraries heavily rely on a template meta programming for a generalization of a processed data. This approach simplifies implementation of the library, reduce number of auxiliary code (in this case, it is generated by the compiler). However, template based approach requires the whole project recompilation for each executable and for any change of a source code. Distribution of a such solution cannot be done in a form of binary file, since the user must compile the library locally each time for usage.

Nvidia Cuda. The most research projects rely on Nvidia Cuda for GPU computations. It provides C++ feature and has a good vendor support. However, Cuda technology is specific only for Nvidia devices. So, a number of devices from other vendors is left behind, what may be critical for users.

2.5 GPU computations

GPGPU (general-purpose computing on graphics processing units) is a technique of graphics processor utilization of a graphics card accelerator for a non-specific computations, which are typically done by a central processing unit of a computer. This technique allows to get a *significant* speedup, when the computation involve large homogeneous data processing with a fixes set of instructions.

There is a number of existing industry standards for a development of GPU programs, such as Vulkan [30], OpenGL [23], DirectX [11] for graphics and computations tasks, as well as OpenCL [23], Nvidia Cuda [21] for computational tasks only.

Existing graph analysis tools in most cases use OpenCL and Nvidia Cuda APIs for GPU work offload. The following sections provide a brief overview for each of this technologies.

Nvidia Cuda. It is an industrial proprietary technology, created by a Nvidia, which is available on graphics devices of this vendor only. This API has rich language support for C and C++ programs. It supports template meta programming, what allows to implement generalized parallel GPU algorithms, such as *sort*, *scan*, *reduce*, which are parameterized by the type of sorted element and used functions and predicates. Also, Nvidia provides a rich set of tools of debugging and profiling Cuda code. What simplifies development significantly.

OpenCL. It is an open industrial standard for a programs' development, which utilize different accelerators for parallel computations. This standard is supported on a number of platforms, such as Nvidia, AMD, Intel, Apple M1, what makes it portable for usage on a large spectrum of devices. This API is designed in a form of C interface. It doesn't have built-in support for generalized meta-programming (opposite to Cuda support). Since this is an open standard, its supports varies significantly from platform to platform. What makes the development and testing of OpenCL application as a complex tasks.

In this work the OpenCL is utilized as a API for GPU computations. This API is chosen, since its required for the project version 1.2 is supported on all actual devices. Also, this API allows dynamic code compilation in runtime for GPU execution. What makes it is usable for creating generalized library, where the user is able to implement custom primitive types and operations in a form of OpenCL code, passed as a string.

2.6 Implementation challenges on GPUs

GPU programming in a connection with a sparse linear algebra domain and large data processing introduces an number of challenges, which must be addressed by the developers of a such frameworks.

Fine-grained parallelism. The most straightforward method of a parallelism is a vertex-based parallelism. However, in many graph, particularly scale-free graphs, the number of outgoing edges per vertex may vary dramatically. In this case, the time of processing of such a vertex will vary in the same way. Thus, assigning a tread per vertex will cause a significant load imbalance in a such case.

This problem may scale to sparse linear algebra approach, where a row of a matrix can be assigned per a thread. So, it is important to dynamically define the load balance and assign different number of threads, accounting the possible amount of work to occur.

Minimizing overhead. GPU kernels running on a large load balanced dataset with a large number of computations achieve the maximum throughput. However, in some cases, the runtime may be dominated by the overhead, not by a computations. For example, GPU kernel may do not have enough work to occupy the whole computational device. In this case, many GPU processing block will be stalled and unused.

Synchronization points can also introduce additional overhead. GPU cores will finish their work and be stalled until the synchronization point is reached. Only after this point the new work will be offloaded. Also, one of the possible overheads may be introduced by the driver runtime. Dynamic JIT GPU compilation, data transfer to GPU and kernel launch may take additional time.

Computations intensity. Good GPU kernel may be characterised as highly parallel grid of threads, where each group of threads process a small portion of the data, which must fit into the on-chip L1 memory. In this case the peak performance is achieved, since the memory latency is minimized to its limits. However, it is almost newer achieved in a graph processing kernels, where the working threads have a lot of unstructured memory load operations with pure computational work, which cannot be avoided.

3 Architecture

This section covers the architecture of the developed library, primary components and modules, the sequence of operations processing on the GPUs. The library is designed in order to overcome the limitation of the existing solutions, mentioned in the previous section.

3.1 Design principles

SPLA library is designed 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.

- **OpenCL API**. The library must use the OpenCL API to accelerate computations on a single OpenCL-compatible GPU device in the system. This API is supported on a variety of graphics accelerators, and also allows you to dynamically compile GPU code at runtime, which saves the user from installing and using a third-party specialized compiler.

- **DAG-based expressions**. The user defines tasks for computations in the form of a directed acyclic graph of expressions, using the library interface to create graph nodes and dependencies between nodes in a declarative style. The user then passes the entire graph to the library for execution, and can either block or wait in a non-blocking way the result.

- **User data types**. The library provides the ability to create arbitrary custom types. The type is represented by the unique name and size of the elements in bytes. The elements of the type are POD structures, treated as regular byte sequences.

- **User functions**. The library provides the ability to create user-defined functions that can take user-defined (arbitrary) types as input, and which can be used to parameterize mathematical operations. Functions are defined as a set of sinks with OpenCL code, which allows you to create functions without using third-party tools to compile custom code.

- **Automated scheduling**. The library automatically splits the expression graph into many tasks and subtasks, which are ordered according to dependencies and distributed to the GPU device. This work is hidden from the user and does not require any action from him.

- **Automated hybrid storage format**. The library automatically formats the data, stores it in a hybrid format, and distributes it automatically between the computing device and the host. This work, storage format, distribution details are hidden from the user and do not require any action from him.

- **Exportable interface**. The library has a 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.

3.2 Execution model

The general idea of the library expression execution is depicted in the figure 3.

As an input library accepts the expression in a form of DAG. The DAG is created using library API. Vertices of the computational graph are the fundamental operations, which process primitives such as matrices or vectors. This operations can read or write data, compute product, etc. Directed edges between vertices show data dependencies between operations. If the edge is present between operations, then the next operation is executed only after the previous one is fully finished. This approach allows to specify, which operations must be ordered and which one can be executed in parallel, what allows better occupy the GPU device.

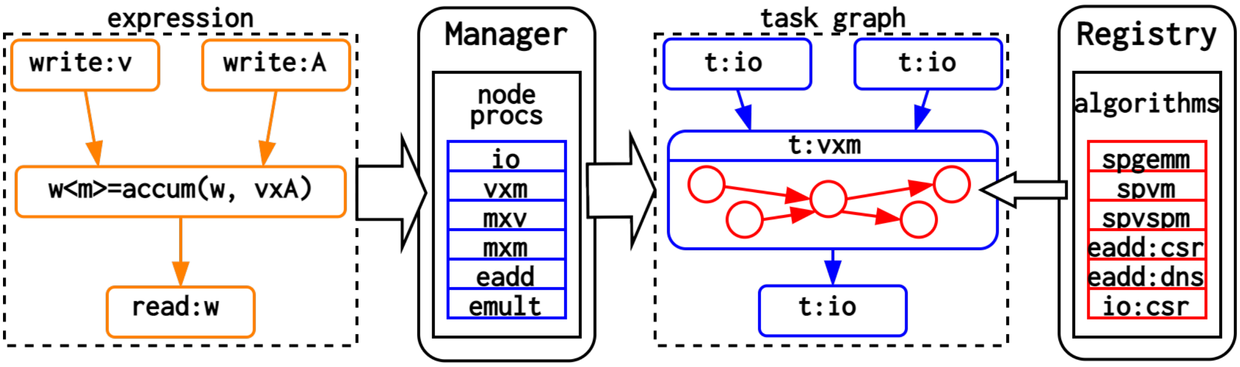

Figure 3: Library expression processing schema.

The DAG expression is submitted to the library for the execution. The expression is traversed and for each a specialized processor is found. The processor is responsible for a processing of a single node. It is possible to have multiple processors for a single type operations with specialized rules of selection. This approach allows to separate the data and the execution, as well as gives an ability to handle some edge cases and optimize operation for a particular set of input arguments.

Processors are used to construct a task graph from the expression. Each processor adds tasks and subtasks for the execution of a particular node. External dependencies between nodes are preserved and translated into dependencies between tasks. Dependencies between subtasks inside a single task are defined by a node processor.

Subtasks are spawned regardless of the storage format and parameters of a particular input arguments. For each type of the arguments there may be present a specialized algorithm in the registry. Algorithm is selected automatically at runtime using a set of rules and priorities. It is possible to specialize algorithm for any type of format and register them without processor code modification.

The task graph is scheduled a whole object to the execution on a multi-core GPU in a multi-threaded mode. CPU threads process tasks and subtasks in a safe manner, since all data dependencies are preserved due to the nature of submitted DAG expression. Each subtask uses separate GPU scheduling queue, so it offloads the GPU with a work without serialization.

Expressions are executed asynchronously. The user after submission

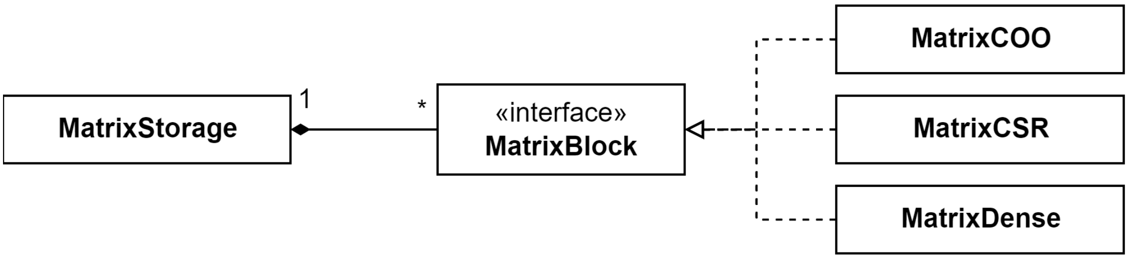

Figure 4: Class hierarchy for matrix storage.

gets a special *future* object, which allows either to block until completion or to probe the state of the expression in a non-blocking fashion.

3.3 Data Storage

Library provides hybrid blocked storage format for matrix and vector representation in CPU and device memory. The idea of the concept is depicted in the diagram 4. Blocked storage usage allows provides following benefits.

- Allows to precisely select format depending on the density of the section of the matrix.
- Allows to utilize multiple algorithms to multiply primitives.
- Allows to offload part of the matrix or vector from GPU memory to save a space.

Matrix storage class is responsible for managing the data of the matrix object. Each matrix instance has its matrix storage class. Storage represents a set of matrix blocks with unified interface, where each block may have a specialised implementation with support for a particular data format, such as *list of coordinates* (COO), *compressed sparse rows* (CSR), *dense*, etc. The list of possible formats is not limited by the design of the library.

For example purposes only, consider **3x4** matrix storage grid as depicted in the figure 5. Blocks form the one or two dimensional grid. The grid is divided into a series of square blocks of equal size except edge blocks, which

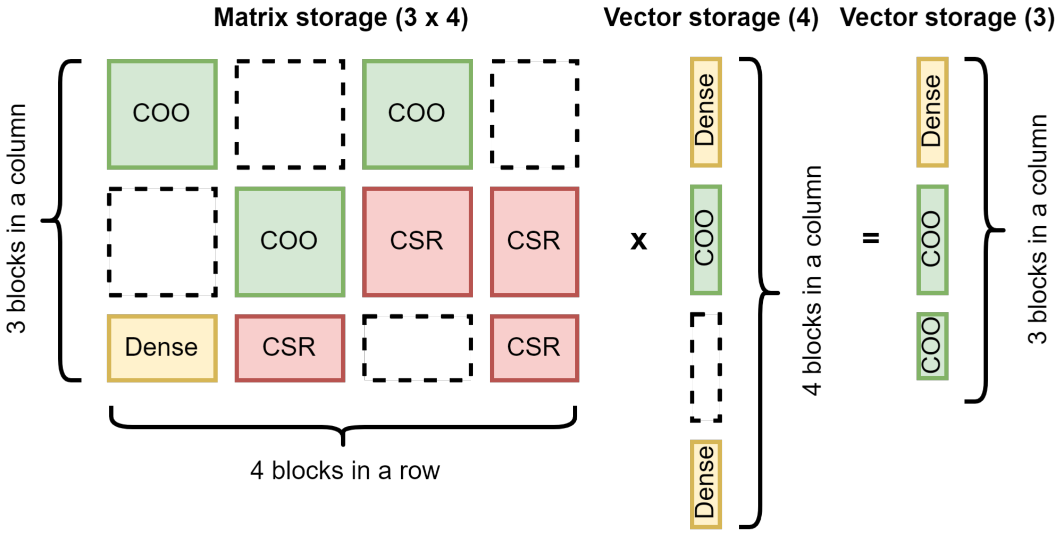

Figure 5: Blocked storage based matrix-vector product.

has clamped size to the matrix dimensions. Each block has unique storage format. Empty blocks not stored in the storage. They do not consume any resources. Size of the block is defined by a constant at library start up. In the extreme case, this constant can be set to *inf*, so the grid will always has **1x1** size.

Multiplication process of object with such a grid schema is straightforward and is done using blocked multiplication. It is guaranteed that matrix and vector with compatible dimensions will always have compatible grid properties.

3.4 Algorithms registry

As it was mentioned in the previous section, matrix and vector containers may be stored in a number of storage formats. Thus, it is important to support operations for all particular cases. Library segregates the particular algorithm invocation and its particular implementation. It provides the following features.

- Allows to invoke the operation without knowledge about underlying storage format.
- Allows to select the algorithm for a particular set of input parameters

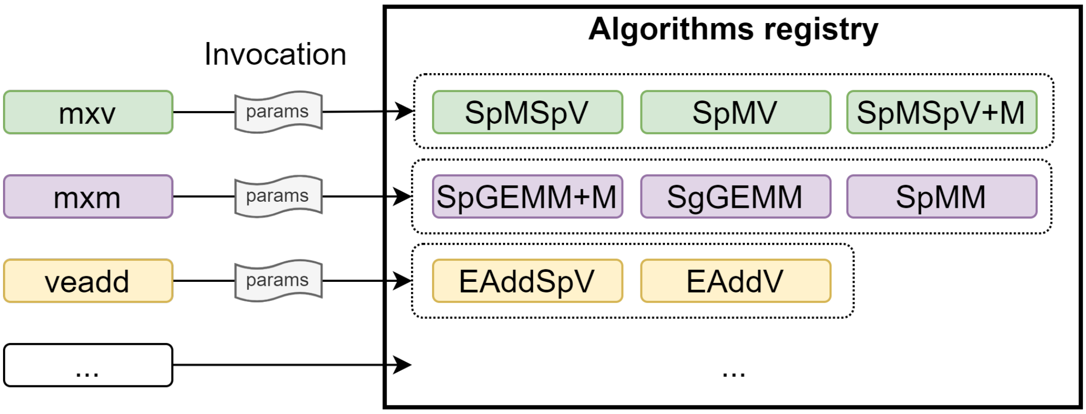

Figure 6: Idea of algorithm invocation segregation.

with a particular storage format at runtime, where extra information is available.

- Allows to extend the library algorithms and formats support without modification of the existing source code.
- Allows to treat blocks of matrix and vector grids as ordinary objects, so it is possible to write generalized blocked products.

The idea of the invocation segregation is depicted in the figure 6. For example, consider matrix-vector product. When we want to multiply some matrix block by some vector block, we use *mxv* operation. For the invocation we pack params and send them to the algorithms registry. Registry is responsible for the selection of the best suitable implementation for the evaluation.

More details about this mechanism are provided in the partial class diagram 7. For the invocation the generalized *AlgoParams* structure is provided. It stores id of the device for the execution and the descriptor with extra parameters to control evaluation. This structure is sub-classed by particular algorithms to add extra parameters, required for algorithms evaluation. The *Algo* interface provides unified functionality, required to store, select and invoke a particular algorithm.

The library relies on the C++ runtime type information, to distinguish parameters structures for different algorithms. Particular algorithms imple-

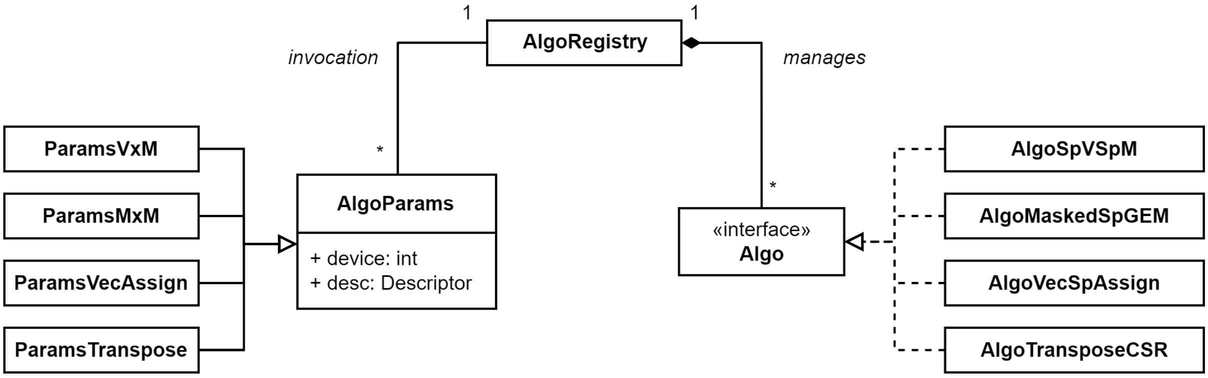

Figure 7: Class hierarchy for algorithms registry.

mentations are stored in the registry class in the order of registration. The selection is order based, so the first best fitting algorithm is selected for the execution. This mechanism can be changed in the future with modification of the *AlgoRegistry* class only.

4 Implementation details

This section covers the implementation details of the developed library. It gives an insight into the selected storage formats, algorithms for GPU processing, and chosen third-party instruments for the library foundation.

4.1 Foundation

4.2 Data Storage

4.3 Operations

4.4 Optimizations

4.5 Running example

5 Algorithms implementation

6 Evaluation

For performance analysis of the proposed solution, an evaluated is conducted of a few most common graph algorithms using real-world sparse matrix data. As a baseline for comparison we chose LAGraph [17] in connection with SuiteSparse [9] as a CPU analysis tool, Gunrock [16] and GraphBLAST [31] as a Nvidia GPU tools. Also, we tested algorithms on several devices with distinct OpenCL vendors in order to validate the portability of the proposed solution.

6.1 Experiments description

Research questions. In general, this 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?

Setup. For evaluation, we use a PC with Ubuntu 20.04 installed, which has 3.40Hz Intel Core i7-6700 4-core CPU, DDR4 64Gb RAM, Intel HD Graphics 530 integrated GPU, and Nvidia GeForce GTX 1070 dedicated GPU with 8Gb on-board VRAM. Host programs were compiled with GCC v9.3.0. Programs using CUDA were compiled with GCC v8.4.0 and Nvidia NVCC v10.1.243. Release mode and maximum optimization level were enabled for all tested programs. Data loading time, preparation, format transformations, and host-device initial communications are excluded from time measurements. All tests are averaged across 10 runs. Additional warm-up run for each test execution is excluded from measurements.

Graph algorithms. For preliminary study *breadth-first search* (BFS) and *triangles counting* (TC) algorithms were chosen, since they allow analyse the performance of *vxm* and *mxm* operations, rely heavily on *masking*,

Table 1: 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
rgg_n_2_22_s0	4.1M	60.7M	36
soc-LiveJournal	4.8M	68.9M	20,333
indochina-2004	7.5M	194.1M	256,425

and utilize *reduction* or *assignment*. BFS implementation utilizes automated vector storage sparse-to-dense switch and only *push optimization*. TC implementation uses masked *mxm* of source lower-triangular matrix multiplied by itself with second transposed argument.

Dataset. Nine matrices were selected from the Sparse Matrix Collection at University of Florida [10]. Information about graphs is summarized in Table 1. All datasets are converted to undirected graphs. Self-loops and duplicated edges are removed.

6.2 Results

Tables 2 and 3 present results of the evaluation and compare the performance of Spla (proposed library) against other tools on different execution platforms. Tools are grouped by the type of device for the execution, where either Nvidia or Intel device is used. Cell left empty if tested tool failed to analyze graph due to *out of memory* exception.

RQ1 *What is the performance of the proposed solution relative to existing tools for both CPU and GPU analysis?*

Table 2: Breadth-first search algorithm evaluation results.
Time in milliseconds (lower is better).

Dataset	Nvidia			Intel	
	GR	GB	SP	SS	SP
hollywood-2009	20.3	82.3	36.9	23.7	303.4
roadNet-CA	33.4	130.8	1456.4	168.2	965.6
soc-LiveJournal	60.9	80.6	90.6	75.2	1206.3
rgg_n_2_22_s0	98.7	414.9	4504.3	1215.7	15630.1
com-Orkut	205.2	– –	117.9	43.2	903.6
indochina-2004	32.7	– –	199.6	227.1	2704.6

Tools: Gunrock (GR), GraphBLAST (GB), SuiteSparse (SS), Spla (SP).

In general, Spla BFS shows acceptable performance, especially on graphs with large vertex degrees, such as soc-LiveJournal and com-Orkut. On graphs roadNet-CA and rgg it has a significant performance drop due to the nature of underlying algorithms and data structures. Firstly, the library utilizes immutable data buffers. Thus, iteratively updated dense vector of reached vertices must be copied for each modification, which dominates the performance of the library on a graph with a large search depth. Secondly, Spla BFS does not utilize *pull optimization*, which is critical in a graph with a relatively small search frontier and with a large number of reached vertices.

Spla TC has a good performance on GPU, which is better in all cases than reference SuiteSparse solution. But in most tests GPU competitors, especially Gunrock, show smaller processing times. GraphBLAST shows better performance as well. The library utilizes a masked SpGEMM algorithm, the same as in GraphBLAST, but without *identity* element to fill gaps. Library explicitly stores all non-zero elements, and uses mask to reduce only non-zeros while evaluating dot products of rows and columns. What causes extra divergence inside work groups.

Gunrock shows nearly the best average performance due to its specialized and optimized algorithms. Also, it has good time characteristics on a mentioned earlier roadNet-CA and rgg in BFS algorithm. GraphBLAST

Table 3: Triangles counting algorithm evaluation results.
Time in milliseconds (lower is better).

Dataset	Nvidia			Intel	
	GR	GB	SP	SS	SP
coAuthorsCiteseer	2.1	2.0	9.5	17.5	64.9
coPapersDBLP	5.7	94.4	201.9	543.1	1537.8
roadNet-CA	34.3	5.8	16.1	47.1	357.6
com-Orkut	218.1	1583.8	2407.4	23731.4	15049.5
cit-Patents	49.7	52.9	90.6	698.3	684.1
soc-LiveJournal	69.1	449.6	673.9	4002.6	3823.9

Tools: Gunrock (GR), GraphBLAST (GB), SuiteSparse (SS), Spla (SP).

follows Gunrock and shows good performance as well. But it runs out of memory on two significantly large graphs con-Orkut and indochina-2004. Spla does not rut out of memory on any test due to the simplified storage scheme.

RQ2 *What is the portability of the proposed solution with respect to various device vendors and OpenCL runtimes?*

On a Nvidia device Spla algorithms' performace in general is acceptable. But it still slower than its competitors, such as Gunrock or GraphBLAST. However, in both BFS and TC the performance gap is maintained in a predictable fashion.

Spla BFS algorithm suffers a lot on a Intel device compared to SuiteSparse implementation. This is caused due to intense data access and relatively small computations parallelism though traversal. On-chip memory has larger latency, so CPU optimized solution shows better results.

However, on Intel device Spla TC algorithm shows better performance compared to SuiteSparse on com-Orkut, cit-Patents, and soc-LiveJournal. A possible reason is the large lengths of processed rows and columns in the product of matrices. So, even embedded GPUs can improve the performance of graph analysis in some cases.

Summary. Evaluation of proposed solution for real-world graph analysis allows to conclude, that initial OpenCL-based operations implementation with a limited set of optimizations has promising performance compared to other tools and can be easily executed on devices of multiple vendors, which gives significant flexibility in a choice of HPC hardware.

7 Conclusion

The following results were achieved in this work.

- The survey of the field was conducted. Model for a graph analysis were shown. Also, the concept of the linear algebra based approach was described in a great detail with a respect to a graph traversal and existing solutions. Introduction into a GraphBLAS standard was provided. Existing implementations, frameworks and most significant contributions for a graph analytic were studied. Their limitations were highlighted.

- General-purpose GPU computations concept was covered. Different APIs for GPU programming were presented. Their advantages and disadvantages are covered. General GPU programming challenges and pitfalls are highlighted.

- The architecture of the library for a generalized sparse linear algebra for GPU computations was developed. The architecture and library design was based on a project requirements, as well as on a limitation and experience of the existing solutions.

- The implementation of the library accordingly to the developed architecture was started. The core of the library, expressions processing, foundation OpenCL functionality, common operations implementations were provided.

- Several algorithms for a graph analysis were implemented using developed library API.

- The preliminary experimental study of the proposed artifacts was conducted. Obtained results allowed to conclude, that the chosen method of the library development is a promising way to a high-performance graph analysis in terms of the linear algebra on a wide family of GPU devices.

The library source code is published on a GitHub platform. It is available at `https://github.com/JetBrains-Research/spla`.

References

[1] Azimov Rustam and Grigorev Semyon. Context-free path querying by matrix multiplication. — 2018. — 06. — P. 1–10.

[2] Barceló Baeza Pablo. Querying Graph Databases // Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems. — New York, NY, USA : Association for Computing Machinery. — 2013. — PODS '13. — P. 175–188. — Access mode: `https://doi.org/10.1145/2463664.2465216`.

[3] Buluç Aydın and Gilbert John R. The Combinatorial BLAS: Design, Implementation, and Applications // Int. J. High Perform. Comput. Appl. — 2011. — nov. — Vol. 25, no. 4. — P. 496–509. — Access mode: `https://doi.org/10.1177/1094342011403516`.

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

[5] Zhang Xiaowang, Feng Zhiyong, Wang Xin, Rao Guozheng, and Wu Wenrui. Context-Free Path Queries on RDF Graphs // CoRR. — 2015. — Vol. abs/1506.00743. — 1506.00743.

[6] Orachev Egor, Epelbaum Ilya, Azimov Rustam, and Grigorev Semyon. Context-Free Path Querying by Kronecker Product. — 2020. — 08. — P. 49–59. — ISBN: 978-3-030-54831-5.

[7] Terekhov Arseniy, Khoroshev Artyom, Azimov Rustam, and Grigorev Semyon. Context-Free Path Querying with Single-Path Semantics by Matrix Multiplication. — 2020. — 06. — P. 1–12.

[8] Dalton Steven, Bell Nathan, Olson Luke, and Garland Michael. Cusp: Generic Parallel Algorithms for Sparse Matrix and Graph Computations. — 2014. — Version 0.5.0. Access mode: `http://cusplibrary.github.io/`.

[9] Davis Timothy A. Algorithm 1000: SuiteSparse:GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra // ACM Trans. Math. Softw. — 2019. — Dec. — Vol. 45, no. 4. — Access mode: https://doi.org/10.1145/3322125.

[10] Davis Timothy A. and Hu Yifan. The University of Florida Sparse Matrix Collection // ACM Trans. Math. Softw. — 2011. — dec. — Vol. 38, no. 1. — Access mode: https://doi.org/10.1145/2049662.2049663.

[11] Direct3D 12 Graphics // Microsoft Online Documents. — 2018. — Access mode: https://docs.microsoft.com/ru-ru/windows/win32/direct3d12/direct3d-12-graphics?redirectedfrom=MSDN (online; accessed: 08.12.2020).

[12] Mishin Nikita, Sokolov Iaroslav, Spirin Egor, Kutuev Vladimir, Nemchinov Egor, Gorbatyuk Sergey, and Grigorev Semyon. Evaluation of the Context-Free Path Querying Algorithm Based on Matrix Multiplication. — 2019. — 06. — P. 1–5.

[13] Zhang Qirun, Lyu Michael R., Yuan Hao, and Su Zhendong. Fast Algorithms for Dyck-CFL-Reachability with Applications to Alias Analysis // SIGPLAN Not. — 2013. — June. — Vol. 48, no. 6. — P. 435–446. — Access mode: https://doi.org/10.1145/2499370.2462159.

[14] Zhang Peter, Zalewski Marcin, Lumsdaine Andrew, Misurda Samantha, and McMillan Scott. GBTL-CUDA: Graph Algorithms and Primitives for GPUs // 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). — 2016. — P. 912–920.

[15] Shi Xuanhua, Zheng Zhigao, Zhou Yongluan, Jin Hai, He Ligang, Liu Bo, and Hua Qiang-Sheng. Graph Processing on GPUs: A Survey // ACM Comput. Surv. — 2018. — jan. — Vol. 50, no. 6. — Access mode: https://doi.org/10.1145/3128571.

[16] Wang Yangzihao, Davidson Andrew, Pan Yuechao, Wu Yuduo, Riffel Andy, and Owens John D. Gunrock // Proceedings of the 21st

ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. — 2016. — Feb. — Access mode: http://dx.doi.org/10.1145/2851141.2851145.

[17] Szárnyas Gábor, Bader David A., Davis Timothy A., Kitchen James, Mattson Timothy G., McMillan Scott, and Welch Erik. LAGraph: Linear Algebra, Network Analysis Libraries, and the Study of Graph Algorithms. — 2021. — 2104.01661.

[18] Batarfi Omar, Shawi Radwa El, Fayoumi Ayman G., Nouri Reza, Beheshti Seyed-Mehdi-Reza, Barnawi Ahmed, and Sakr Sherif. Large Scale Graph Processing Systems: Survey and an Experimental Evaluation // Cluster Computing. — 2015. — sep. — Vol. 18, no. 3. — P. 1189–1213. — Access mode: https://doi.org/10.1007/s10586-015-0472-6.

[19] Kepner J., Aaltonen P., Bader D., Buluc A., Franchetti F., Gilbert J., Hutchison D., Kumar M., Lumsdaine A., Meyerhenke H., McMillan S., Yang C., Owens J. D., Zalewski M., Mattson T., and Moreira J. Mathematical foundations of the GraphBLAS // 2016 IEEE High Performance Extreme Computing Conference (HPEC). — 2016. — Sep. — P. 1–9.

[20] NVIDIA. CUDA Thrust // NVIDIA Developer Zone. — 2020. — Access mode: https://docs.nvidia.com/cuda/thrust/index.html (online; accessed: 16.12.2020).

[21] NVIDIA. CUDA Toolkit Documentation // NVIDIA Developer Zone. — 2020. — Access mode: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (online; accessed: 01.12.2020).

[22] Ching Avery, Edunov Sergey, Kabiljo Maja, Logothetis Dionysios, and Muthukrishnan Sambavi. One Trillion Edges: Graph Processing at Facebook-Scale // Proc. VLDB Endow. — 2015. — aug. —

Vol. 8, no. 12. — P. 1804–1815. — Access mode: https://doi.org/10.14778/2824032.2824077.

[23] OpenCL: Open Standard for Parallel Programming of Heterogeneous Systems // Khronos website. — 2020. — Access mode: https://www.khronos.org/opencl/ (online; accessed: 08.12.2020).

[24] Orachyov Egor, Alimov Pavel, and Grigorev Semyon. cuBool: sparse Boolean linear algebra for Nvidia Cuda. — 2020. — Access mode: https://pypi.org/project/pycubool/. Version 1.2.0.

[25] Malewicz Grzegorz, Austern Matthew H., Bik Aart J.C, Dehnert James C., Horn Ilan, Leiser Naty, and Czajkowski Grzegorz. Pregel: A System for Large-Scale Graph Processing // Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data. — New York, NY, USA : Association for Computing Machinery. — 2010. — SIGMOD '10. — P. 135–146. — Access mode: https://doi.org/10.1145/1807167.1807184.

[26] Anderson James, Novák Adám, Sükösd Zsuzsanna, Golden Michael, Arunapuram Preeti, Edvardsson Ingolfur, and Hein Jotun. Quantifying variances in comparative RNA secondary structure prediction // BMC bioinformatics. — 2013. — 05. — Vol. 14. — P. 149.

[27] Cailliau P., Davis T., Gadepally V., Kepner J., Lipman R., Lovitz J., and Ouaknine K. RedisGraph GraphBLAS Enabled Graph Database // 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). — 2019. — P. 285–286.

[28] Roy Amitabha, Mihailovic Ivo, and Zwaenepoel Willy. X-Stream: Edge-Centric Graph Processing Using Streaming Partitions // Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. — New York, NY, USA : Association for Computing Machinery. — 2013. — SOSP '13. — P. 472–488. — Access mode: https://doi.org/10.1145/2517349.2522740.

[29] Shun Julian and Blelloch Guy E. Ligra: A Lightweight Graph Processing Framework for Shared Memory // SIGPLAN Not. — 2013. — feb. — Vol. 48, no. 8. — P. 135–146. — Access mode: https://doi.org/10.1145/2517327.2442530.

[30] The Khronos Working Group. Vulkan 1.1 API Specification // Khronos Registry. — 2019. — Access mode: https://www.khronos.org/registry/vulkan/specs/1.1/html/vkspec.html (online; accessed: 08.12.2020).

[31] Yang Carl, Buluç Aydın, and Owens John D. GraphBLAST: A High-Performance Linear Algebra-based Graph Framework on the GPU // arXiv preprint. — 2019.

[32] cuBool: sparse linera Boolean algebra for NVIDIA CUDA // Github. — 2020. — Access mode: https://github.com/JetBrains-Research/cuBool.

[33] pygraphblas: a Python wrapper around the GraphBLAS API. — Access mode: https://github.com/Graphegon/pygraphblas (online; accessed: 28.12.2022).