

Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 21.M07-мм

*Орачев Егор Станиславович*

# Разработка библиотеки обобщенной разреженной линейной алгебры для вычислений на GPU

Отчёт по технологической практике

Научный руководитель:  
Доцент кафедры информатики, к. ф.-м. н. С. В. Григорьев

Санкт-Петербург  
2022

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

<b>Introduction</b>	<b>5</b>
<b>1. Problem statement</b>	<b>7</b>
<b>2. Background of study</b>	<b>8</b>
2.1. Graph analysis approach . . . . .	8
2.2. GraphBLAS concepts . . . . .	9
2.3. Existing frameworks . . . . .	12
2.4. Known limitations . . . . .	15
2.5. GPU computations . . . . .	17
2.6. GPU architecture . . . . .	19
2.7. OpenCL concepts . . . . .	21
2.8. Implementation challenges on GPUs . . . . .	23
<b>3. Architecture</b>	<b>25</b>
3.1. Design principles . . . . .	25
3.2. Design overview . . . . .	26
3.3. Execution model . . . . .	28
3.4. Data Storage . . . . .	29
3.5. Algorithms registry . . . . .	32
<b>4. Implementation details</b>	<b>35</b>
4.1. General . . . . .	35
4.2. Dependencies . . . . .	36
4.3. Automation . . . . .	37
4.4. Interface . . . . .	39
4.5. Data storage . . . . .	42
4.6. Kernel management . . . . .	44
4.7. Running example . . . . .	45
<b>5. Evaluation</b>	<b>47</b>
5.1. Experiments description . . . . .	47
5.2. Results . . . . .	48

<b>6. Results</b>	<b>52</b>
<b>References</b>	<b>54</b>

# Introduction

Graph model is a natural way to structure data in a number of a real practical tasks, such as graph databases [2,25], social networks analysis [21], RDF data analysis [5], bioinformatics [24] and static code analysis [12].

In the graph model the entity is represented as a graph vertex. Relations between entities are directed labeled edge. This notation 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 domains with graphs for analysis having 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 [21], 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 [27], GraphBLAST [28], SuiteSparse [9], etc. Existing solutions provide different execution strategies and models, use for acceleration GPUs or multi-core CPUs.

According to Yang et al. [28] and Orachev et al. [29], 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 to implement a sparse linear algebra primitives and operations library, 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 approach

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 [23] 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. Then 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 than vertices. It is first introduced in X-Stream [26] 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 drive.

**Linear-algebra based.** Linear algebra based frameworks for a graph processing originate from a CombinationalBLAS [3] project, which intro-



duced 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 one 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 [13], Huawei GraphBLAS implementation [4], etc.

## 2.2 GraphBLAS concepts

GraphBLAS standard [19] is a mathematical notation translated into a C API. This standard provides sparse linear algebra building blocks for the implementation of graph algorithms in terms of operations over matrices and vectors. 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

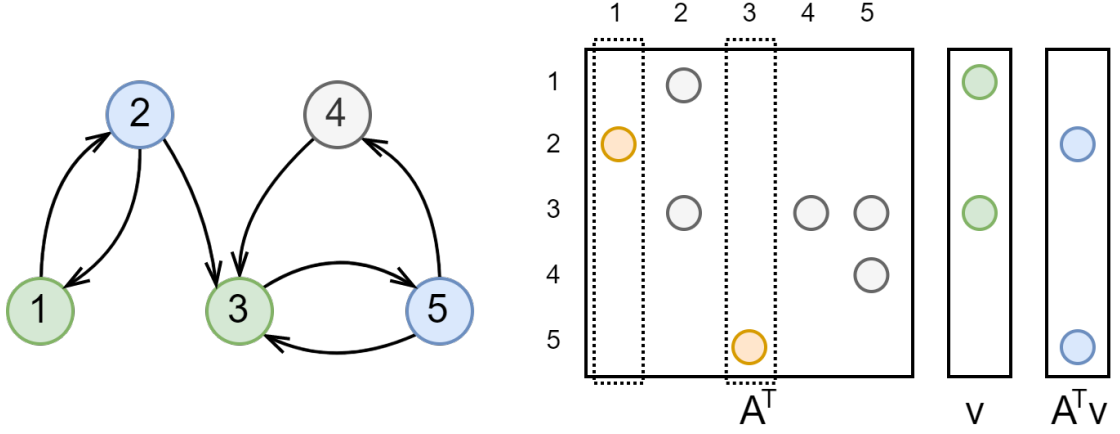


Figure 1: Graph traversal by matrix-vector product

are parameterised by the type of stored elements. The standard provides a set of predefined commonly used types, as well as, 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. These structures are adapted for a sparse graph analysis, so its mathematical properties differ a bit from those, which are stated in classical algebra.

These structures define the element-wise operations, which work with elements in the containers. For example, they are passed as a parameters *multiply* 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 unreachable vertices.

**Programming constructs.** GraphBLAS provides extra objects, which are required for practical algorithms implementation. One of these programming features 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 pattern 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.

**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

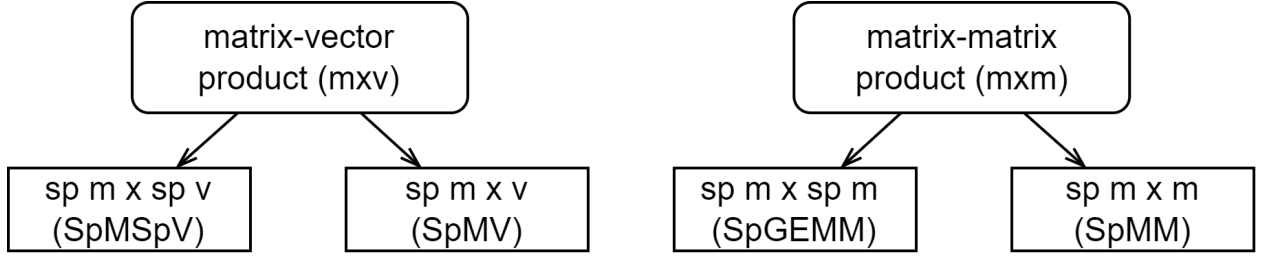


Figure 2: Key operations of the GraphBLAS standard and their implementations

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 out 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. In 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-

---

**Listing 1** Breadth-first search using GraphBLAS API

---

```
1 #include 'GraphBLAS.h'
2
3 GrB_Info BFS(GrB_Vector *v, GrB_Matrix A, GrB_Index s, GrB_Index n)
4 {
5     GrB_Vector_new(v, GrB_INT32, n);
6
7     GrB_Vector q;
8     GrB_Vector_new(&q, GrB_BOOL, n);
9     GrB_Vector_setElement(q, true, s);
10
11     int32_t level = 0;
12     GrB_Index nvals;
13
14     do {
15         ++level;
16         GrB_apply(*v, GrB_NULL, GrB_PLUS_INT32, GrB_SECOND_INT32, q, level, GrB_NULL);
17         GrB_vxm(q, *v, GrB_NULL, GrB_LOR_LAND_SEMIRING_BOOL, q, A, GrB_DESC_RC);
18         GrB_Vector_nvals(&nvals, q);
19     } while (nvals);
20
21     GrB_free(&q);
22
23     return GrB_SUCCESS;
24 }
```

---

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.

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 [30]).

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 there is no way to provide template meta programming through raw C API.

**GraphBLAST.** GraphBLAST [28] 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 [29] 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, 11].

The library uses Nvidia Cuda API for GPU computations. It provides a pycubool [22] 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 Known limitations

This section highlights the most critical limitations and notable drawback of the GraphBLAS standard, existing implementations, and other relative solutions. These issues aggregation is based on an end-user and a developer perspective. It is important to list and address this limitations, in order to better understand scope of this work. The majority of issues is discussed in a talk given by Gilbert [14].

- **Complicated API.** GraphBLAS standard declares a very verbose and complex API for implementation. This API consists of a C header file, which counts more than 12 thousand lines of a code, which must be implemented and tested. Implementation of a such API is a challenging activity. Declared API has a lot of hidden states, invariants, C-specifics, what makes the efficient implementation complex and error prone.
- **C-oriented.** GraphBLAS standard initially oriented on a C99 compatible API due to some technical and performance considerations. Standard does not considers any high-level build-in package for a more streamlined work with API. It makes the standard inaccessible and less popular then, for instance, Sci-Py or NumPy packages for python.

- **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.
- **Lack of interoperability.** GraphBLAS declares opaque objects with hidden from the user structure. It is not possible to some-how extend or interact with an existing standard implementation. However, some practical tasks may required integration of existing formats, storage into a library for practical tasks solving. For example, it can be use full to integrate NumPy arrays into library in order to avoid extra copy operations and reduce marshaling overhead between execution environments.
- **Little introspection.** GraphBLAS declares a very limited functionality to inspect structure, state, type, behaviour, performance, correctness, progress of library primitives and operations. It is not feasible to build production-ready data-analysis platform without featured introspection, which is a port of all modern DBMS.
- **Implicit zeros.** GraphBLAS standard tries to use a mix of math and engineering concepts to address the values storage model. As the result, this model is to complex and not obvious for both mathematicians and programmers. GraphBLAS has a know issue with a storage of implicit zeroes or identity elements in memory. Inaccurate storage manipulations may cause a sufficient memory usage increase in your application even if you correctly follow the standard.
- **Inflexible masking.** GraphBLAS standard provides an ability to apply a mask to filter out result matrices or vectors. However, rules for selecting values from a mask are implicit and rely on selecting raw



zero values, like in a C program. This mechanism is not configurable. An alternative for that is the ability to select mask values using user-provided predicate.

- **No GPU support.** GraphBLAS has no fully-featured implementation with GPU backed support for computations. The primary reason for this is the complexity of the standard. Also, standard is not designed with idea GPU support. Thus, there are a lot of another open problems to solve.
- **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.

Summarizing, the GraphBLAS standard, its known implementations and nearby analogues have a number of significant and critical limitations, which are addressed in this work.

## 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, OpenGL, DirectX for graphics and computations tasks, as well as OpenCL, Nvidia Cuda 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 GPU architecture

This section gives an overview for a typical GPU architecture. Understanding of a target hardware is a key to performance optimizations, required to speed up developed GPU library.

The particular GPU architecture is a very controversial topic for a discussion. Implementation details of a given GPU depend on a number of factors, such a GPU's vendor, family, generation, etc. Thus, writing an efficient GPU code forces a programmer to learn a particular details of a target processor for execution. For the sake of brevity, let's focus on an existing, open-source and well-document state-of-the-art modern GPU microarchitecture such as AMD's RDNA.

Radeon DNA (RDNA) is a GPU microarchitecture and accompanying instructions set architecture developed by AMD. The block diagram of Radeon RX 5700 XT GPU having RDNA architecture is depicted in a figure 3. The GPU itself is placed on a infinity fabric surface for a fast data interconnect between units. PCIe controller allows communication of a GPU with the system RAM.

The GPU is composed of two *shader engines*. Each engine has a number of *shader arrays*. Each array possess own L1 cache and a set of actual *dual compute units*, which responsible for computation work. Shader engines surrounded by a L2 shared cache. Any of L2 banks can be accessed by shader array during the computations.

Programs for the execution are structured in a form of *kernels*. Kernel is a single stream of instructions that operate on large number of data parallel work items. The work items organized into a work groups, which can communicate explicitly through local memory. The shader compiler subdivides the work groups further into micro-architecture specific wavefronts

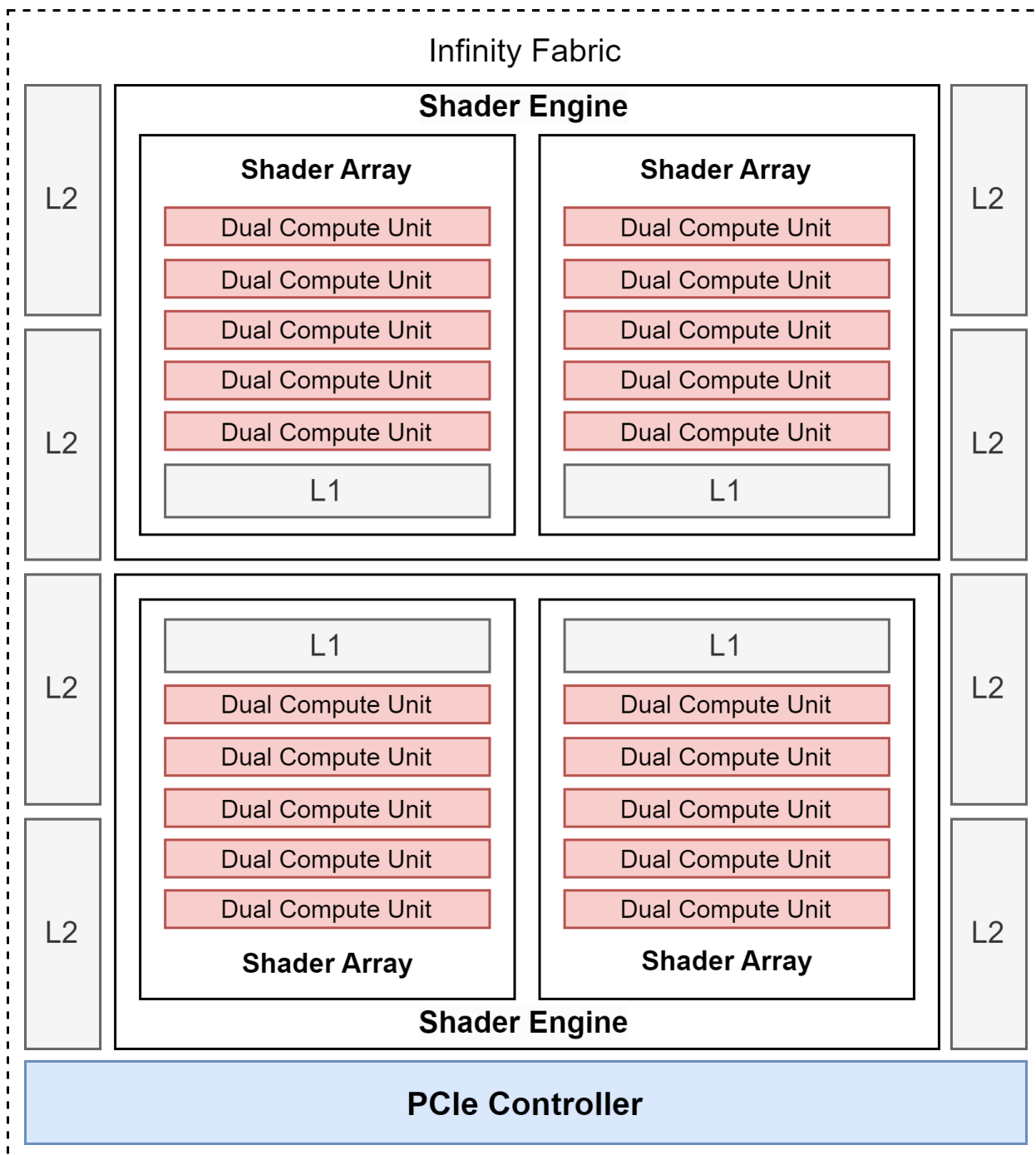


Figure 3: The block diagram of a Radeon RX 5700 XT GPU powered by the RDNA architecture.

that are scheduled for parallel execution on a compute unit.

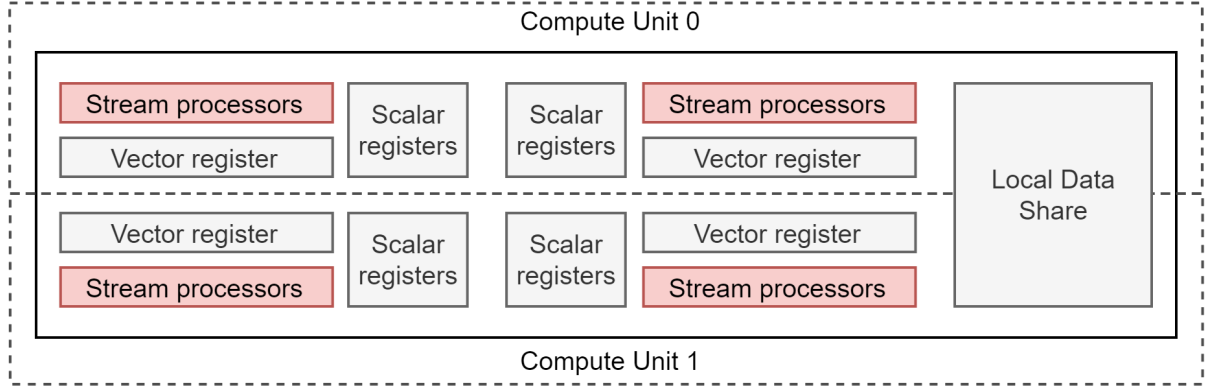


Figure 4: AMD Radeon DNA dual compute unit. Compute unit consist of a number of SIMD processors. Each processor has independent registers set. All processors share local memory, called local data share in AMD terms.

Dual compute unit in AMD architecture is depicted in a figure 4. Compiler crates wavefronts of a size 32 (wave32). Every item inside a wavefront is executing the same instruction (SIMD). Each compute unit (CU) includes four SIMD units. Each SIMD unit has 32 ALUs, has 32-wide vector registers and scalar registers. SIMD executes full wavefront instruction over single clock cycle.

Thus, writing an OpenCL kernel requires saturation of CU with works as well as keeping all slots of a SIMD processor active. Elimination of some of these features causes inefficiency and, as the consequences, performance drop of a target application.

## 2.7 OpenCL concepts

This section gives a brief introduction to the OpenCL standard. This section covers platform, execution and memory model. It introduces essential programming constructs and gives an understanding on how a typical OpenCL program is written.

**Platform.** OpenCL exists in a context of a platform. Platform in OpenCL terms is vendor, or organization, which provides OpenCL imple-

mentation for a target machine. Several platforms on single machine may be available. It depends on the installed CPU, GPU or FPGA.

Typical providers of OpenCL implementations are Intel, Nvidia, AMD and Apple. For computations only single platform can be selected. Thus, resources and features are not shared between different implementations.

OpenCL is an open API. Its implementation and support is optional. So, the presence of the OpenCL, actual version, set of features and extensions is a subject, which varies a lot from one system to another.

**Execution.** Execution of an OpenCL program takes within an execution context. Context is an environment, which is created using platform and list of devices, which must be used for computations. Context keeps track of all resources, manages global execution state.

Device is an logical unit, which performs operations. Several devices may be available in a single platform. Typical devices are integrated Intel GPUs, discrete Nvidia or AMD video adapters.

Device consists of a set of compute units. Compute unit is a small processor with its own instruction and data caches, registers, controllers. Distinct compute units can run distinct programs. However, single compute unit can run only single program at given time.

Work for a compute unit is structured as work items. Single work item inherently is a thread, which executes instructions stream and has own registers.

**Memory.** The whole available memory for an OpenCL program is divided into three parts. Global memory is available across all devices within a context. In most cases, it is a dedicated VRAM with L2 cache of the GPU. Local memory is a memory available only within single compute unit. This memory is visible only inside this unit. It is not persistent, and exist only in time or program execution. Local memory is registers, available for a single work item.

**Programming.** OpenCL provides C-compatible API for applications

development. From a user point of view an OpenCL program is a set of kernel invocations with some resources, bound to the kernel.

Resources are different memory buffers or textures, which can be consumed on a GPU for read/write operations. These resources created from CPU side using specialized C API. The data inside buffers can be access using copy commands or specialized map/unmap functions. Actual location for a storage is hidden for the user. However, it is possible to hint storage properties using some flags.

Kernels are scheduled for the execution using command queues. Command queue is an logical abstraction, which control the order of execution of different kernels. User can create multiple command queues and synchronize them using specialized events. Commands queues mapped to hardware queues automatically by the OpenCL driver.

Kernel is a special function, written using C-language extension and compiled using OpenCL built-in compiler. This function is invoked for each work item to perform some meaningful work.

## 2.8 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. JIT GPU kernels 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 never 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 sparse linear algebra (Spla for short) 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

The 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.

- **Optional acceleration.** Library is designed in a way, that GPU acceleration is fully optional part. Library can perform computations using standard CPU pipeline. If GPU is supported, library can offload a part of a work for accelerator.
- **Manual scheduling.** The user defines tasks for computations in the form of a schedule object. Schedule consists of a series of dependent step. Each step has a list of prioritised independent tasks. The uses assembles schedule using The user then passes the entire schedule to the library for execution, and can either block or wait in a non-blocking way the result.
- **Predefined types.** The library provides the ability to parameterize operations using a set of predefined 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. Standard types are supported.
- **Rich functions set.** The library provides the ability to parameterize operations using predefined functions, declared over supported stan-

dard types. Functions are built-in. They have a C++ and OpenCL analogues integrated into library core.

- **Explicit execution.** 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.
- **Multiple storage formats.** The library provides a generalized mechanism to store a data in a number of different formats. It is done in a form of decorators. A decorator stores data in a single format. Decorators can be created on demand by different parts of the library. Transformation rules used to synchronize data between decorators.
- **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 Design overview

The general library design idea is depicted in a figure 5. The most import part of the design is that the GPU acceleration is a fully optional part of the library. Thus, the library can function in any environment even without GPU. This design is dictated by the following considerations.

- **CPU and GPU asymmetry.** CPU and GPU are not equal devices for programming and execution. The GPU device is always in a satellite mode relatively to the CPU. CPU defines the control sequence for a GPU. CPU and GPU also capable of doing a bit different things. Thus, CPU must remain the main part of the design library in order to reflect this peculiarities.
- **GPU complexity.** GPU is a very complex thing to program. GPU requires additional algorithms and data structures to work properly.

Thus, this complexity must be isolated in a form of fully optional module.

- **GPU specifics.** GPU has specialized requirements for data allocation and layout. Thus, CPU data must be duplicated, transformed and processed into specialized acceleration structures, stored on a GPU. It introduces duplication of data. But, it can be neglected. RAM has order of magnitude bigger size then the bleeding edge VRAM of a modern video card.
- **User experience.** Data analysis involves a lot of tasks. Not all tasks can be leveraged to GPU. It involves complex data processing, statics, read queries, etc. Thus, some work must be done on a CPU.
- **Modularity.** Modular design allows replacement of computations accelerators on fly. Thus, it can be possible to use multi-core CPUs, different GPUs APIs, and even multi-GPUs as acceleration backends without any existing code modifications.

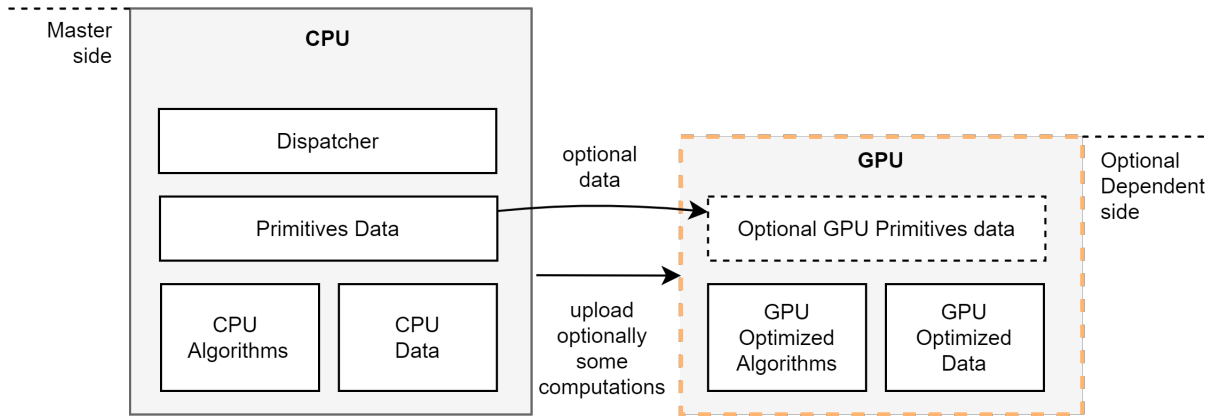


Figure 5: Library primary design idea. The CPU side of the library is a master. It is fully featured to perform computations. GPU is an optional acceleration. It can be used to offload some work. Data and algorithms support is optional.

### 3.3 Execution model

The general idea of the library schedule execution is depicted in the figure 6.

As an input library accepts the expression in a form of a schedule. The schedule is created using library API. The schedule is built from a number of sequential steps. Steps are ordered. The next step starts only when the previous step is fully finished. Each step is composed from a sorted by a priority list of tasks. Single task represents a fundamental operation, which processes matrices or vectors. This operation can be product, assignment, transposition, etc. Tasks will be started independently in the order, defined by priorities.

Number of parallel tasks can be limited by a number of factors. Number of available CPU threads as well as number of GPU queues defines how many tasks run in parallel. GPU has a limited number of hardware queues for submitting commands. In most cases it is limited up to only a single queue. Thus, on some GPUs it is not possible to get any parallelism from parallel tasks.

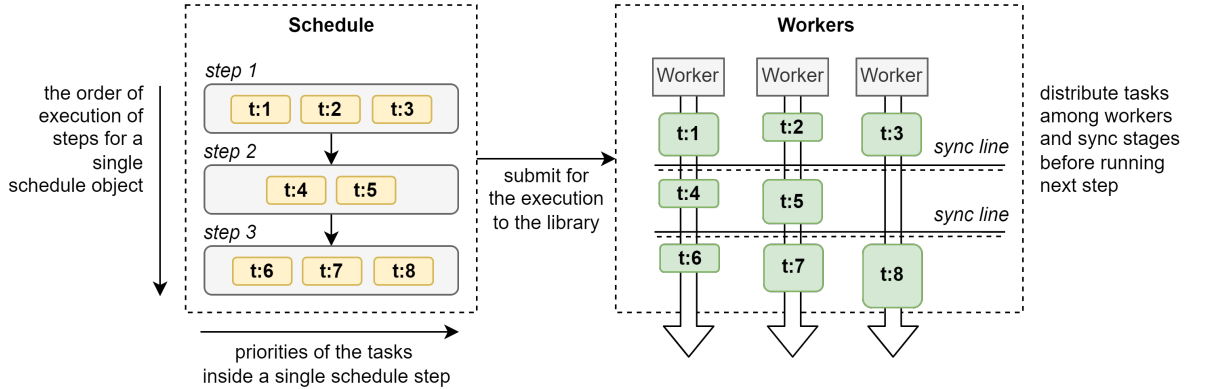


Figure 6: Library schedule execution idea.

The schedule is submitted to the library for the execution. The schedule is traversed and for each task in a step the algorithm for execution is found. The algorithm is responsible for a processing of a single task. It is possible to have multiple task 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

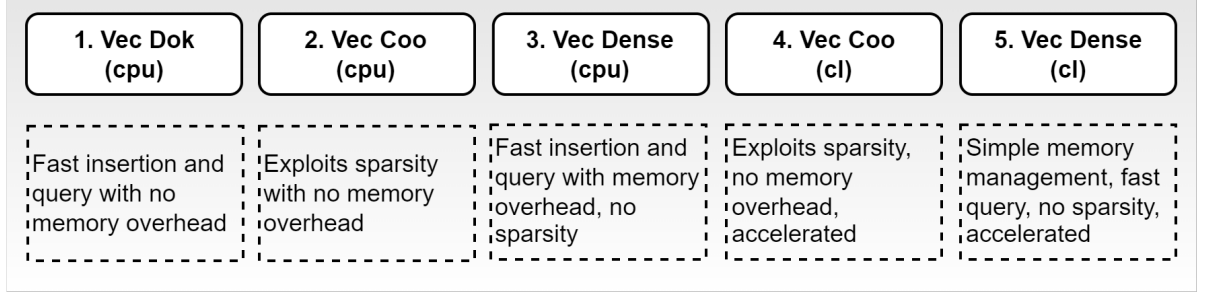


Figure 7: Vector decorations storage. Storage provides slots for different representations. Representation choice depends on a task currently being solved.

for a particular set of input arguments.

Each algorithm is responsible for the evaluation of a single task. Task is passed as an argument. Algorithm obtains all necessary parameters through the task using runtime type system.

Schedules are executed asynchronously. The user after submission gets a special *future* object, which allows either to block until completion or to probe the state of the schedule in a non-blocking fashion.

For the sake of simplicity, each operation can be run in an procedural fashion without a schedule. In this case, dummy schedule is constructed under the hood, submitted and immediately executed. It is convenient for simple user algorithms without complex scheduling potential.

### 3.4 Data Storage

Library provides flexible and extensible hybrid decorations' based storage format. The idea of the storage is depicted in a figure 7. This storage model aims to solve the following problems.

- **Format flexibility.** There is no the silver bullet storage for allow tasks. Thus, different formats must be employed for distinct tasks solving. What is more important if we talking about GPUs acceleration. GPU has distinct memory space and unique memory layout requirements. Also, layout of the data may vary depending on type

of operations, which reads, modifies, or entirely updates the content of the vector or matrix on a GPU.

- **Data synchronization.** The presence of multiple valid decorations with vector or matrix data causes data transformations. Thus, data for one target decoration can be actualized from another source decoration. This data transformation must be formalize in terms of different rules, so it is possible add new formats to the storage.
- **Code complexity.** Formalization of storage mechanism in a form of programmable set of rules is required. The complexity of convertation has quadratic nature if we want to support N different storage formats. Thus, formalization must be don prior to standardize the way of how new format is added to the library.
- **Generalization.** This storage mechanism can be used for both vector and matrix storage. It makes it usable and very powerful thing. Potentially, it allows adding new primitives to the library reusing existing storage schema. Decorations can be create to hold any data useful for the application.
- **Interoperability.** Library storage schema also must solve the interoperability issues. Decorations can hold any data. It allows to support NumPy and Sci-Py based primitives without modification of any existing source code.

The idea of storage transformation graph is depicted in a figure 8. Transformations graph defined using rules as oriented edges between formats. The presence of an arrows shows, that here is a rule to convert one decorator data into other decorator data. These rules can be used to synchronize decorations data on demand.

For an instance, consider a task of obtaining a decoration in a dictionary of keys (dok cpu) format for user read operations in figure 9. The storage keeps tracking of valid decorations with valid data. In our example currently valid decoration is list of coordinates decoration for OpenCL device (vec coo

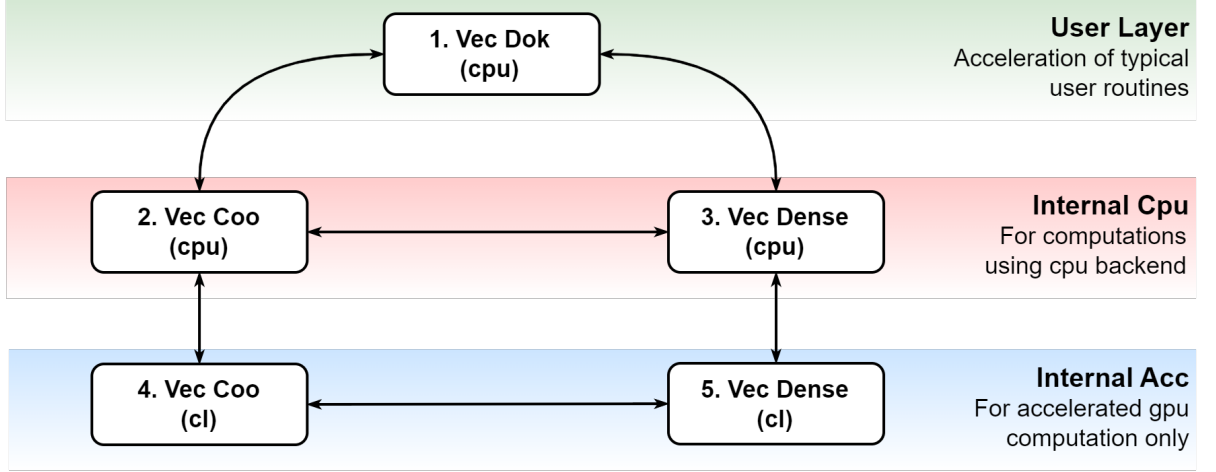


Figure 8: Vector decorations storage transformations graph. Graph declares transformation rules between different decoration formats. Existence of a path in this graph allows to convert one decoration into another automatically.

cl). The transformation graph searches for the shortest path between node 4 and 1 in a rules graph. Then it issues conversions. The first conversion will copy OpenCL data to CPU data into vector list of coordinates for CPU side. Then, this data will be converted further into a dictionary of keys. After that, the data stored on GPU can be used as a dok on a CPU for fast queries and modifications.

The transformation process might be costly. It can involve several transformation stages. Also, its complexity depends on the amount of data to convert. Thus, the transformation must be avoided as much as it can. In this case, typical strategy for a data processing is following.

- User uploads all the necessary data into matrices and vectors.
- User issues a preparation the data. Internally it will be converted into suitable formats.
- Then user performs and number of computations states keeping data on the accelerator side without unnecessary read-backs.
- Finally, user copies result of the computations back to the CPU to analyse achieved results.

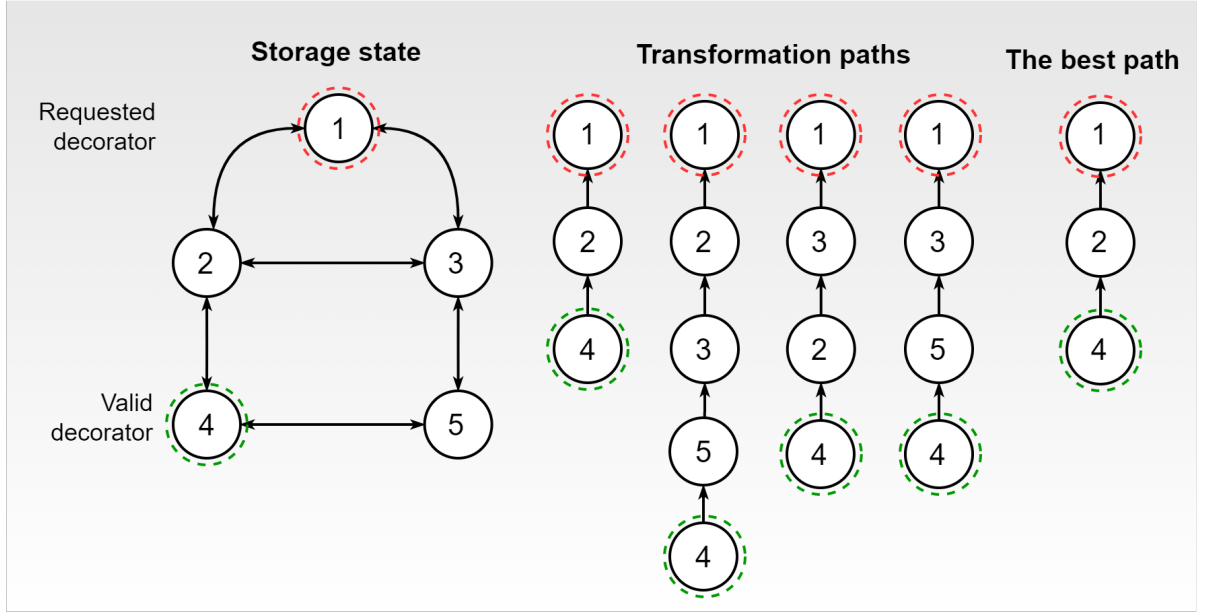


Figure 9: Vector storage transformation process. Target format is 1 (red circle). Source format is 4 (green circle). The shortest path is the best path for conversion.

### 3.5 Algorithms registry

This section describes how different algorithms implementations stored in the library how it is found on a request. Algorithms registry aims to solve the following tasks.

- Registry segregates algorithms declarations and its particular invocations. It allows to dispatch algorithms dynamically at runtime what gives flexibility for a configuration.
- Registry allows to query for supported and not supported algorithm. It gives an ability to select algorithm at runtime. If algorithm is not presented, then we can fallback to other less optimized version.
- Registry gives an ability to store multiple versions of the same algorithm for different platforms and accelerations backends.

Idea of the registry storage is shown in a figure 10. Registry is a key value storage like a dictionary. As a key strings with special formatting are employed. This strings are constructed from an operation description



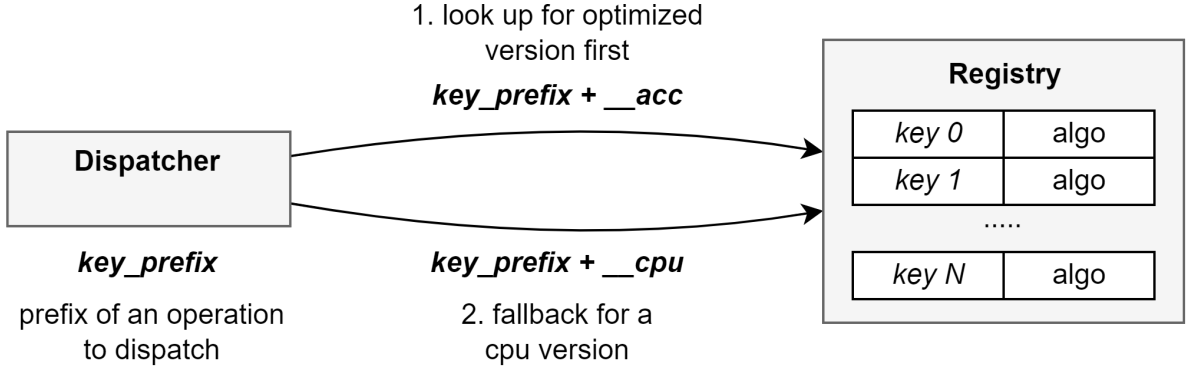


Figure 10: Registry of algorithms. Dispatcher looks up for optimized algorithms first. As a fallback it uses cpu suffix to get default algorithm implementation without an acceleration.

which must be evaluated. Dispatcher appends the suffix of a target device or backend for computations. If key is presented, then the algorithm is used. Otherwise some fallback implementation is utilized.

The structure of keys is depicted in a figure 11. The key is effectively a string literal, which is constructed using specialized rules. The prefix of the key is the name of the operation which must be evaluated. As a name for operation actual mathematical name can be used, such as matrix-vector product, matrix-matrix product, etc.

The name is followed by the name of used functions and their type codes. Mathematical functions must be parameterized by scalar functions. Each function has a name and a set of type codes for each type of the argument. Scalar multiplication and addition functions has three opcodes. Since each function is a binary operator of type  $A \times B \rightarrow C$ .

Binary functions are followed by a selection operation. Selection is an unary function with the signature  $A \rightarrow bool$ . Selection operator is used to filter final results using masking. Typically, we are not interested in a whole result. So the mask is provided. Type of mask values used to parameterize select operation. Selection can be any unary predicate. In most cases, greater than, equals or not equals zero are the most used one.

Finally, the key prefix is appended with a code of a backend for computations. Different accelerators, including cpu fallback, may be supported for

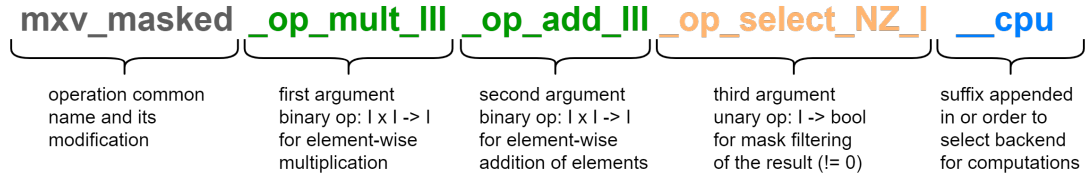


Figure 11: The structure of an algorithm key. The is a string literal composed from several parts. Prefix shows the algorithm name and its parametrization by operations. The suffix of the key shows which backed or accelerator to use for the evaluation of the algorithm.

computations. Using accelerator suffix allows to switch between backend at runtime and select the most optimized algorithm.

It is possible that there is no algorithms for a given key. Thus, the fallback version must be utilized. In order to do that, key prefix may be concatenated with a `cpu` suffix. All algorithms have a `cpu` analogues in a registry.

Keys in a form of a string solve two major problems. Firstly, it gives a readable and human understandable representation of an operation. Secondly, it allows to actually segregate an operation with its arguments and particular algorithm instance. Algorithm is an object with its own state and unified interface. It allows to maximize the performance and reuse artifacts, which may appear between algorithm invocations, such as GPU kernels, acceleration structures, etc.














## 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 General

Developed *spla* library is written using C++17 language and standard library. CMake 3.17 is used as build configuration tool. Ninja library is used to generate platform specific build files. Library supports build on Linux (tested on Ubuntu 20.04), Windows (tested on 10) and macOS (tested on Catalina). Git used as version control system. The source code of the project is hosted on a GitHub page. Library is compiled into shared executable object with respect to the target platform naming convention and object extension.

Project directory has the following structure.

-  *include*. Public library interface files in *.hpp* format.
  -  *spla*. Public library C++ interface header files.
  -  *spla-c*. Public library C99-compatible interface header files.
-  *src*. Source files, compiled into shared executable object.
-  *deps*. Third-party project dependencies stored as a source code.
-  *tests*. Directory with unit tests files for Google Tests.
-  *examples*. Example applications for graph algorithms.
-  *python*. Source code for a python package for *spla* library.
-  *CMakeLists.txt*. Root cmake file of the project.
-  *build.py*. Python script to build library artifacts.
-  *generate.py*. Python script to generate *.hpp* from *.cl* files.
-  *run\_tests.py*. Python script to run unit tests.
-  *bump\_version.py*. Python script to upgrade package version.

## 4.2 Dependencies

This section briefly covers third-party libraries and projects, utilized by the `spla` library.

**Khronos OpenCL headers.** C-compatible OpenCL header files library developed and maintained by the Khronos Consortium. Since the OpenCL is an public API declared as a standard, its support is optional for operating systems and programming environments. In order to access OpenCL functions the respective header files with OpenCL functions declarations, signatures, constants, defines and other symbols must be manually used by a project. An alternative is to install this headers to a computer manually. But this step is error prone and less flexible.

**Khronos OpenCL hpp headers.** C++-compatible OpenCL header files library developed and maintained by the Khronos Consortium. Since the `spla` project is written using modern C++ standard, safe C++ bindings for an OpenCL code must be used.

OpenCL C++ bindings provide a memory and exceptions safe, object-oriented API, which relies on a standard containers and data structures. It allows to automate and simplify objects lifetime management.

**Khronos OpenCL ICD loader.** OpenCL installable client driver (ICD) library developed and maintained by the Khronos Consortium. It provides a mechanism to allow developers to build applications against an ICD loader rather than linking their applications against a specific OpenCL implementation.

The ICD loader is responsible for: exporting OpenCL API entry points, enumerating OpenCL implementations, forwarding OpenCL API calls to the correct implementation.

The ICD mechanism is required in order to load dynamically particular OpenCL implementation at runtime. The motivation for that is the vast variety of different OpenCL implementations. Each implementation can be

shipped with a GPU driver. The system can have a number of different GPUs with distinct drivers and vendors. Thus, it is not possible to know a target implementation a priori.

The ICD loader is bundled inside the `spla` dynamic library during build process. Loader is used on a library startup. It quires available OpenCL drivers in the system. Then it selects one to use in the application. The selection is based on a user parameters. Then it loads OpenCL symbols through shared library mechanism and initializes global OpenCL state.

**GTest.** GTest is an open source unit-testing library for C++ projects. This library is developed and maintained by a Google company. The library provides flexible macro system for declaring unit tests and assertions. This library is used extensively for testing a `spla` functionality. Project units testes with `gtest` are stored in a `tests` directory.

**Cxxopts.** Cxxopts is an open-source command-line arguments parsing library for C++ projects. This library automates processing of executable arguments, passed in a classic *argc* & *argv* fashion. Library is used a an auxilary tool for example applications, built using library API. Example applications used for a benchmarking of the library performance.

## 4.3 Automation

The library source code is hosted on a GitHub platform. This platform provides a convenient *actions* mechanism, also called *workflows*. It allows to automate the process of a continuous project changes integration and continuous delivery of the project artifacts to potential users. The GitHub repository is configure with the following list of scripts for automation.

- **build.** The build action, which compiles the source code of the library, executable examples and test for three target platform: Windows 10, Ubuntu (20.04) and macOS (for x64 and arm architectures). The artifacts of a build process are published automatically in a GitHub

repository. These artifacts are reused in a later step, when the python package is assembled to be pushed to either test or retail index repository.

- **clang-format.** The formatting script which automatically checks the conformance of the library header and source files. The project uses a clang-format tool to check the code style of the project automatically. Definition of a code style is stored in repository as a configuration file in special format.
- **deploy.** Deployment script is responsible for an automated publishing of a spla python package to the python package index (PyPI) repository. This action assembles a bundle, which stores python sources as well as artifacts for all platforms from *build* action. Action automatically pushes package to the PyPI using credentials, stored in a repository. The action is triggered automatically on commits to special *release* branch. This is supposed to happen on a major and minor library versions' releases.
- **deploy-test.** Deployment script similar to *deploy* action. The difference is that this script pushes package to the Test PyPI repository for testing purposes. The action is triggered automatically on commits to special *pre-release* branch.
- **docs-cpp.** Script which assembles C/C++ library documentation using Oxygen format. The documentation is represented by a set of html pages. These pages are deployed automatically to the project website page.
- **docs-pythos.** Script which assembles python package documentation using python docs library. The documentation is represented by a set of html pages. These pages are deployed automatically to the project website page.

Table 1: A list of the supported operations to access vector and matrix containers.

Method	Description
<i>Matrix</i>	Matrix constructor from type T and dimensions
<i>Vector</i>	Vector constructor from type T and dimension
<i>clear</i>	Empty vector or matrix
<i>get_nrows</i>	Query number of rows for a matrix or vector
<i>get_ncols</i>	Query number of columns for a matrix or vector
<i>get_type</i>	Query the type T of elements
<i>set_&lt;T&gt;</i>	Set element of type T at index $\langle i, j \rangle$
<i>get_&lt;T&gt;</i>	Get element of type T at index $\langle i, j \rangle$

## 4.4 Interface

This section the technical details of the library public interface are covered. Interface includes matrix, vector and scalar containers for a typed data storage, operations for the execution, algebraic functions for operations customization, etc.

**Containers.** Library provides *vector*, *matrix* and *scalar* data containers. Each container can be parameterised with a type of stored values. The actual storage mechanism is automated and is hidden from a user. Matrix and vector containers can store data in multiple formats at the same time. In order to access them, the library provides opaque interface, which allows to incrementally build containers, query elements, inspect its properties and state. List of supported operations to access containers is shown in a table 1.

**Operations.** An expression for the execution is constructed as a schedule object using library API. The primitive unit of the schedule is a single task. Task represents an operation over matrices, vectors and scalars. Library provides a number of common and widely used operations for evaluation. List of supported operations provided in the table 2.

Table 2: A list of the *spla* mathematical operations for computations.

Operation	Math equivalent	Description
<i>masked vxm</i>	$r_i = (vM)_i, \forall i : f(m_i)$	Masked vector-matrix product
<i>masked mxv</i>	$r_i = (Mv)_i, \forall i : f(m_i)$	Masked matrix-vector product
<i>masked assign</i>	$r_i = s, \forall i : f(m_i)$	Masked vector scalar assignment
<i>reduce</i>	$s = \Sigma v_i$	Vector reduce to scalar
<i>select count</i>	$s =  \{v_i : f(v_i)\} $	Vector select count

**Element-wise functions.** The core feature of the library is the ability to parameterise math operations mentioned above with arbitrary algebraic element-wise binary and unary functions. The list of build-in functions, which supported for both CPU and GPU computations, is depicted in the table 3. The operations can be used for any of build type such int, uint and float values. The only exception is bit-wise operations, which can be applied only to integral types.

**Signatures.** Library heavily relies on a built-in mechanism of automated reference counting of objects. Each object has an atomic counter, which tracks number of references. When the counter reaches the zero, it frees up the object. This mechanism used for safe arguments passing around library and for safe marshaling of objects through C and python APIs.

Library employs explicit operations signatures. All arguments and parameters must passed by the user through operation interface. If operation has variations or provides tweaking, all parameters must be specified.

As an example, consider the signature of the masked matrix-vector product operation in a listing 2. This is a procedure, which can be loaded from a dynamic or shared library. As the result of an invocation it returns special *Status* enumeration value. Library uses no exceptions. Thus, error codes are employed. It is standard practice for libraries with C-compatible API.

The procedure takes nine input in lines **1** – **9** and one optional output argument in line **10**. The *r* is a vector where to store result of operation execution. The *mask* is a vector of the same dimension as *r*, which is used



Table 3: A list of the spla element-wise mathematical functions to parameterise operations.

Function	Equivalent	Type	Description
<i>plus</i>	$r = a + b$	function	Sum of two elements
<i>minus</i>	$r = a - b$	function	Difference of two elements
<i>mult</i>	$r = a * b$	function	Product of two elements
<i>div</i>	$r = a / b$	function	Division of two elements
<i>min</i>	$r = \min(a, b)$	function	Minimum value
<i>max</i>	$r = \max(a, b)$	function	Maximum value
<i>first</i>	$r = a$	function	First argument of function
<i>second</i>	$r = b$	function	Second argument of function
<i>one</i>	$r = 1$	function	Identity element
<i>and</i>	$r = a \wedge b$	function	Bit-wise product
<i>or</i>	$r = a \vee b$	function	Bit-wise sum
<i>xor</i>	$r = a \oplus b$	function	Bit-wise exclusive sum
<i>eqzero</i>	$a == 0$	predicate	Check equals zero
<i>neqzero</i>	$a \neq 0$	predicate	Check not-equals zero

to update only selected entries of the vector  $r$ . Matrix  $M$  and vector  $v$  are the actual primitives to multiply. Actual algebraic element-wise functions from multiplication and addition are passed as *op\_multiply* and *op\_add*. The predicate to filter result by a mask passed as a *op\_select*. Note, that functions in the library are first-class objects, which can be manipulated as any other library object. The identity element for product evaluation is passed as *init* scalar.

An optional parameter is a Descriptor *desc* object. It has the same usage as in a GraphBLAS standard. Descriptor stores additional parameters, which configure actual execution of the operation, occupation, preferred device, mode, etc. It can be used to optimize the execution of particular operations for edge cases.

Finally, the procedure accepts an optional output argument. It is a pointer to the handle of the schedule task object. If pointer is null, then the procedure executes the operation in an imperative fashion. If this pointer is

---

**Listing 2** The C++ signature of the spla masked matrix-vector product.

---

```
1 SPLA_API Status exec_mxv_masked(/* in */ ref_ptr<Vector>      r,  
2                                /* in */ ref_ptr<Vector>      mask,  
3                                /* in */ ref_ptr<Matrix>       M,  
4                                /* in */ ref_ptr<Vector>       v,  
5                                /* in */ ref_ptr<OpBinary>     op_multiply,  
6                                /* in */ ref_ptr<OpBinary>     op_add,  
7                                /* in */ ref_ptr<OpSelect>     op_select,  
8                                /* in */ ref_ptr<Scalar>       init,  
9                                /* in */ ref_ptr<Descriptor>   desc = nullptr,  
10                               /* out */ ref_ptr<ScheduleTask>* task_hnd = nullptr);
```

---

not null, then the implementation creates a deferred task for the execution, stores reference to this task in provided pointer and returns. In this case, the returned task can be used to construct schedule object, which can be submitted at once as a whole. The scheduling mechanism implemented as previously described in architecture section.

## 4.5 Data storage

The library is implemented following storage schema, introduced and described in a previous section. In this schema each data container, such as a matrix or vector has a number of decorations or formats, which can be simultaneously assigned to the container. It introduces duplication. However, it gives the flexibility for the choice of target device and particular implementation algorithm for the execution.

The same container can have a number of formats allocated at the same time. The storage manager automatically controls the validity of the data. This mechanism allows to cache the same data in both RAM and VRAM memory. Since the RAM in most cases has a order of magnitude larger size, the duplication is negligible.

The vector storage container supports following formats.

- **CPU dictionary of keys (DoK).** The format of the vector, where non-zero entries stored as a dictionary. Storage space is proportional to a number of values. It gives fast query and insertion operations at cost of inefficient memory layout. This format is used for incremental builds of container on a CPU side.

- **CPU list of coordinates (COO).** This format used for sparse vector representation. Data in this format stored as a list of indices and as a list of values. Memory space proportional to the number of non-zero entries. Used to prepare data for GPU source & target transfer.
- **CPU dense vector.** This format used for a dense vector representation. Data in this format stored as a large array of values with the same size as vector dimension. Memory space proportional to the vector dimension. Memory consumption can be excessive on vectors with over 1M elements. Used to prepare data for GPU source & target transfer.
- **OpenCL list of coordinates (COO).** It is GPU representation of a COO format using OpenCL API. Used for sparse vector manipulation on GPU side.
- **OpenCL dense vector.** It is GPU representation of a COO format using OpenCL API. Used for a dense vector manipulation on GPU side.

The matrix storage container supports following formats.

- **CPU dictionary of keys (Dok).** This format is used for an incremental matrix building on a CPU side as it is done for a vector. The memory space used by this format is proportional to the number of non-zero entries. This format provides fast query and insertion operations at cost of inefficient memory layout.
- **CPU compressed sparse row (CSR).** Compressed sparse row is one of the most common formats for a sparse matrix representation. The data is stored in a form of three arrays: rows offsets, column indices and column values. Rows values are packed. They are stored continuously. Column indices of each value in a row and each value are packed together in index and value arrays. Offsets to the start of a particular row are stored in offsets buffer. Total memory cost of the

storage proportional to the number of entries in sparse arrays. Offsets buffers always allocated using total number of rows of a matrix.

- **OpenCL compressed sparse row (CSR).** This is a CSR format implementation for a GPU computations using OpenCL. Data to this storage format is transferred from a CPU CSR format representation. GPU CSR allows fast access to a random matrix row. However, the access on a particular value in a row is linear and requires consecutive reads. It is more suitable for GPUs processing, where each row can be processed in parallel by separate SIMD processors or compute units.

## 4.6 Kernel management

OpenCL provides a flexible but yet complex way to manage GPU executable code. The OpenCL program is text file written using some sort of a C-language with special extensions. The program must be compiled at runtime before the actual usage. The compilation process may take a couple of seconds. It is not feasible to compile a program for each invocation of an OpenCL operation. Thus, the management mechanism is required.

The library implements this mechanism in a form of a runtime program cache. Main entities of this mechanism are listed below.

- **CLProgram.** Is a wrapper for the OpenCL program. It stores compiled program as well as used definitions, symbols and cached kernels for the invocation.
- **CLProgramBuilder.** Program builder provides a mechanism for a flexible runtime program construction. It allows to parameterise source code of the program with particular types, functions, constants, etc. It is required since programs are generalized and unaware of particular types of processed elements such as int, float, uint. The builder automatically checks if program already compiled and cached. It avoids unnecessary compilation.
- **CLProgramCache.** Program cache is a global runtime storage of all

compiled programs. It uses source code of programs as keys. Thus, any variation of a particular algorithm implementation can be cached and reused as many times as needed.

## 4.7 Running example

---

**Listing 3** Breadth-first search algorithm implementation using Spla API

---

```

1 SPLA_API Status spla::bfs(const ref_ptr<Vector>&      v,
2                          const ref_ptr<Matrix>&      A,
3                          uint                        s,
4                          const ref_ptr<Descriptor>& desc) {
5     const auto N = v → get_n_rows();
6
7     ref_ptr<Vector> front_prev = make_vector(N, INT);
8     ref_ptr<Vector> front     = make_vector(N, INT);
9     ref_ptr<Scalar> front_size = make_int(1);
10    ref_ptr<Scalar> depth      = make_int(1);
11    ref_ptr<Scalar> zero       = make_int(0);
12    int            current_level = 1;
13    int            front_size    = -1;
14    int            discovered    = 1;
15    bool           front_empty = false;
16
17    front_prev → set_int(s, 1);
18
19    while (!front_empty) {
20        depth → set_int(current_level);
21        exec_v_assign_masked(v, front_prev, depth, SECOND_INT, NQZERO_INT, desc);
22        exec_vxm_masked(front, v, front_prev, A, BAND_INT, BOR_INT, EQZERO_INT, zero, desc);
23        exec_v_reduce(front_size, zero, front, PLUS_INT, desc);
24
25        front_size → get_int(front_size);
26        front_empty = front_size == 0;
27        discovered += front_size;
28        current_level += 1;
29
30        std::swap(front_prev, front);
31    }
32
33    return Status::Ok;
34 }

```

---

As an example of the developed spla library usage consider breadth-first search algorithm implementation shown in the code listing 3.

Algorithm procedure is declared in the *spla* namespace in the public interface file. The implementation of the algorithm is defined in the private cpp source file, compiled into shared object library. Procedure expects as

an input reference to the result vector  $v$  where to store reached depths of vertices, adjacency matrix of the undirected graph  $A$ , index of the start vertex  $s$  and an optional descriptor to tweak algorithm execution.

Before actual execution, the graph size saved as  $N$  in line **5**. Then in lines **7** – **11** data containers required for the algorithm execution are allocated. The front of the search is created in lines **7** – **8**. Two instances are used, since the update of the front for the new iteration requires the previous version of the front. In order to avoid unintentional costly GPU memory allocations, these fronts allocated explicitly and kept until the end of the procedure. The front size scalar is created in line **9**. Scalar holding current depths and scalar with identity elements are created in lines **10** – **11**. A number state tracking variables are created in lines **12** – **15**. They are used to track current state of the search.

Initial frontier start vertex is set in line **20**. The starting depths of the source vertex is 1. Yet unreached vertices or unreachable vertices have a depth equal to 0 by default.

The algorithm iterates in the *while loop* in lines **22** – **34** until frontier of vertices to visit is not empty. Iteration operations executed in lines **23** – **26**. Firstly, the scalar is updated with new depth value. Then, this depth is assigned to the result vector using frontier from the previous iteration as a mask. After assignment new frontier is obtained as a single search step from front vertices to next vertices through vector-matrix product. Note, that vector  $v$  used as mask to filter all already visited vertices. The special predicated for that is used. Only mask values which equal to zero will be touched. It implies to the update only of yet unreached vertices.

After the execution the vector  $v$  for each graph vertex stores either the depth of the vertex or zero in the case if this vertex is not reachable from the BFS source vertex  $s$ . Since library API relies on C++ RAII mechanism, no explicit resources cleanup is required after the execution.

## 5 Evaluation

For performance analysis of the proposed solution, an evaluation 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 [28] 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.

### 5.1 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 4: 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  $m \times m$  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 4. All datasets are converted to undirected graphs. Self-loops and duplicated edges are removed.

## 5.2 Results

Tables 5 and 6 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 5: 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 6: 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 run 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’ performance in general is acceptable. But it is 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 through 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.

## 6 Results

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 following tasks must be done to complete this work.

- Extend a set of available linear algebra operations, implemented in the library.
- Implement a set of a common graph analysis algorithms utilising library primitives and operations, as well as introducing some optimizations for this algorithms.
- Conduct a complete experimental study of the set of common graph analysis algorithms. Extend the dataset and study the edge cases of library workarounds.

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] 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.
- [12] 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>.
- [13] 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.
- [14] Gilbert John R. What did the GraphBLAS get wrong? // HPEC GraphBLAS BoF. — 2022. — Access mode: <https://sites.cs.ucsb.edu/~gilbert/talks/talks.htm>.
- [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, Rif- fel 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] 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>.
- [22] 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.



- [23] 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>.
- [24] 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.
- [25] 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.
- [26] 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>.
- [27] 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>.
- [28] Yang Carl, Buluç Aydın, and Owens John D. GraphBLAST: A High-Performance Linear Algebra-based Graph Framework on the GPU // arXiv preprint. — 2019.
- [29] cuBool: sparse linear Boolean algebra for NVIDIA CUDA //

Github. — 2020. — Access mode: <https://github.com/JetBrains-Research/cuBool>.

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