

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 производственной практике

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

Санкт-Петербург
2023

Saint Petersburg State University

Egor Orachev

Master's Thesis

Generalized sparse linear algebra library with vendor-agnostic GPUs accelerated 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:
Expert, Huawei S. V. Moiseev

Saint Petersburg
2023

Contents

Introduction	5
1. Problem statement	7
2. Background of study	8
2.1. Related Work	8
2.2. GraphBLAS concepts	11
2.3. GraphBLAS limitations	14
2.4. GPU computations	16
2.5. GPU architecture	18
2.6. OpenCL concepts	20
2.7. Implementation challenges on GPUs	22
3. Proposed solution description	24
3.1. Design Principles	24
3.2. Architecture overview	25
3.3. Data Containers	26
3.4. Algebraic Operations	27
3.5. Differences with GraphBLAS standard	29
4. Implementation details	31
4.1. Project structure	31
4.2. Compile-time dependencies	32
4.3. Development automation	33
4.4. Library interface	35
4.5. Algorithms registry	38
4.6. Storage formats	41
4.7. OpenCL backend	43
4.8. Linear Algebra Operations	44
4.9. Graph Algorithms	45
4.10. Running example	46

5. Evaluation	48
5.1. Research questions	48
5.2. Evaluation setup	48
5.3. Graph algorithms	49
5.4. Dataset	49
5.5. Results Summary	50
6. Results	58
References	60

Introduction

Graph model is a natural way to structure data in a number of a real practical tasks, such as graph queries [1], graph databases [24], social networks analysis [22], RDF data analysis [6], bioinformatics [23] and static code analysis [11]. 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.

Practical real-world graph data tends to be sparse and counts dozens of millions of vertices and billions of edges [22]. Thus, scalable high-performance graph analysis is demanding area and an actual challenge for a research. There is a big number of ways to attack this challenge [5] and the first promising idea is to utilize general-purpose graphic processing units. Such existing solutions, as CuSha [7] and Gunrock [16] show that utilization of GPUs can improve the performance of graph analysis, moreover it is shown that solutions may be scaled to multi-GPU systems. However, low flexibility and high complexity of API are problems of these solutions.

The second promising thing which provides a user-friendly API for high-performance graph analysis algorithms creation is a GraphBLAS API [20], which provides linear algebra based building blocks to create graph analysis algorithms. The idea of GraphBLAS is based on a well-known fact that linear algebra operations can be efficiently implemented on parallel hardware. Along with that, a graph can be natively represented using matrices: adjacency matrix, incidence matrix, etc. While reference CPU-based implementation of GraphBLAS, SuiteSparse [9], demonstrates good performance in real-world tasks, GPU-based implementation is challenging.

One of the challenges in this way is that real data are often sparse, thus underlying matrices and vectors are also sparse, and, as a result, classical data structures and respective algorithms are inefficient. So, it is necessary to use advanced data structures and procedures to implement sparse linear algebra, but the efficient implementation of them on GPUs is hard due to

the irregularity of workload and data access patterns. Though such well-known libraries as cuSPARSE [25], clSPARSE [4], bhSPARSE [19], Cusp [8] show that sparse linear algebra operations can be efficiently implemented for GPUs, it is not so trivial to implement GraphBLAS on GPU. First of all, it requires *generalized* sparse linear algebra, thus it is impossible just to reuse existing libraries which are almost all specified for operations over floats with classical element-wise functions. The second problem is specific optimizations, such as masking fusion, which can not be natively implemented on top of existing kernels. Nevertheless, there is a number of implementations of GraphBLAS on GPU, such as GraphBLAST [28], GBTL [13], which show that GPUs utilization can improve the performance of GraphBLAS-based graph analysis solutions. But these solutions are not portable across different device vendors because they are based on Nvidia Cuda stack.

Although GraphBLAS is solid and mature standard with a number of implementation, it has limitations and shortcomings discussed in a talk given by John R. Gilbert [14]. Some of them are lack of interoperability and introspection, what is an obstacle on the way of GraphBLAS integration into real-world data analysis pipelines. Implicit zeroes mechanism and masking, which uses mix of engineering and math, leads to unpredictable memory usage in some cases, keeping API complex for both implementation and usage.

Summarizing, there is still no portable and high-performance library, which provides generalized linear-algebra based building blocks for real-world large sparse graph analysis problems. Such a library can potentially solve a number of problems, such as complex user API for particular graph algorithms implementation, GPUs performance and portability issues. Although GraphBLAS standard is a good reference point for implementation, it is still possible design a bit different API. The solution can solve some of technical GraphBLAS limitations while still being coherent with the standard for further co-operation.

1 Problem statement

The goal of this work is the implementation of the generalized sparse linear algebra primitives and operations library with portable vendor-agnostic yet high-performance GPUs accelerated computations. The work can be divided into the following tasks.

- Conduct the survey of existing solutions, focusing on design principles and programming model, overview technologies and tools for programming GPU computations and highlight challenges of GPU programming.
- Develop the architecture of the library. Design the high-level library structure, execution model, storage scheme, GPUs backend for vendor-agnostic and portable computations acceleration.
- Implement the library according to the developed architecture, including library core, backend for GPUs accelerated computations, some GPU optimizations in order to speedup computations, and a set of common graph algorithms.
- Conduct the preliminary experimental study of implemented artifacts. Analyse the performance of the proposed solution compared to existing tools, test the portability and scalability of the developed library on GPUs of different device vendors.

2 Background of study

This section provides a brief overview of existing solutions for graph analysis on GPU. Also, it describes the concepts of the GraphBLAS standard, highlights some of its shortcomings and limitations on the way to a full-fledged GPU implementation. Finally, this section gives a brief introduction to a GPU programming.

2.1 Related Work

There is a number of graph processing frameworks for a both CPU and GPU analysis. A great survey of such frameworks is done by Batari et al. [18] and Shi et al. [15]. Problems, addressed by those graph processing frameworks on a GPUs, can be categorized into the following major aspects: data layout, memory access pattern, workload mapping and graph programming model. While all of them are important for a high-performance analysis, the latter is what the user directly encounters when solving applied problems. A flexible, expressive, and at the same time efficient for implementation graph programming model is one of the determining factors for the widespread use of the framework.

Existing GPU-based frameworks typically adopt vertex-centric model, where computation is defined as a series of user functions, executed over vertices in some parallel fashion. Thus, this model falls into two variations further: gather-apply-scatter (GAS) and bulk synchronous parallel (BSP).

GAS model. Such frameworks as CuSha [7], MapGraph [12] adopt GAS model. The computation in GAS model consists of three phases, where each phase performs some vertex processing by user-defined functions, while the framework controls the overall phases execution. This model allows to abstract the need of explicit synchronizations, what simplifies analysis and ensures correctness. However, this approach suffers from an extra GPU overhead. High-level steps introduce explicit synchronization, which must be done by the framework. Extra synchronization reduces the device utilization and introduces extra time delays.

BSP model. Medusa [30], Gunrock [16] use BSP model. In this model the computation is divided into a series of super steps, where local computation occurs within each step with message passing. This model allows local computations, local memory usage, reduces synchronization and kernel launch overhead, but may suffer from workload imbalance among threads in super step.

Gunrock, one of the fastest programmable frameworks for GPU graphs analysis [15], has solved this issue introducing several workload mapping techniques. This improvement allows to achieve great speedup in almost all algorithms. However, Gunrock is only Cuda-oriented framework with relatively low-level API, which requires a significant programming effort to implement a particular algorithm for analysis. Also, this framework utilizes a number of ad hoc optimizations. So it is limited in its generalization.

Linear algebra based model. This model was pioneered by Buluç et al. [2] in CombinationalBLAS, which introduced primitives for a large graph data analysis for a distributed memory CPU systems. This model allows one to define graph algorithms using linear algebra operations over matrices and vectors with some custom user-defined element-wise operations. This allows one to express complex computations in few lines of code without significant performance sacrifice. What makes it promising for implementation.

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 traver-

sal 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 research community formalized linear algebra based model in a form of GraphBLAS standard [20], which has a number of implementations for CPUs, such as high-performance SuiteSparse library [9] or Huawei GraphBLAS implementation [3], and some adaptations for a GPUs analysis.

GraphBLAS SuiteSparse [9] is a fully featured reference GraphBLAS implementation for multi-core CPUs 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 [31]). At this moment, the work is done in the project in order to support Nvidia Cuda for GPU computations.

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. 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.

GBTL [13] is a GraphBLAS-like framework for Cuda GPUs focused on programming language research, API formalization and correctness rather than high performance.

It is worth to mention a number of high-performance sparse linear algebra libraries, such as proprietary cuSPARSE [25] sparse math library for Nvidia Cuda platform, open-source clSPARSE [4] sparse math library for

OpenCL platform, open-source bhSPARSE [19] sparse math library for heterogeneous computations which is still in development, Cusp [8] template-based sparse math library with support for Nvidia Cuda, etc. These libraries are focused on numerical computations over floating-point values with standard element-wise addition and multiplication. Some of these libraries support only Nvidia Cuda devices. Other libraries use cross-platform technologies, but still missing some features required for graph analysis, such as user-defined functions, masking, etc. Thus, they are not fully suitable for needs of a GPU graph analysis.

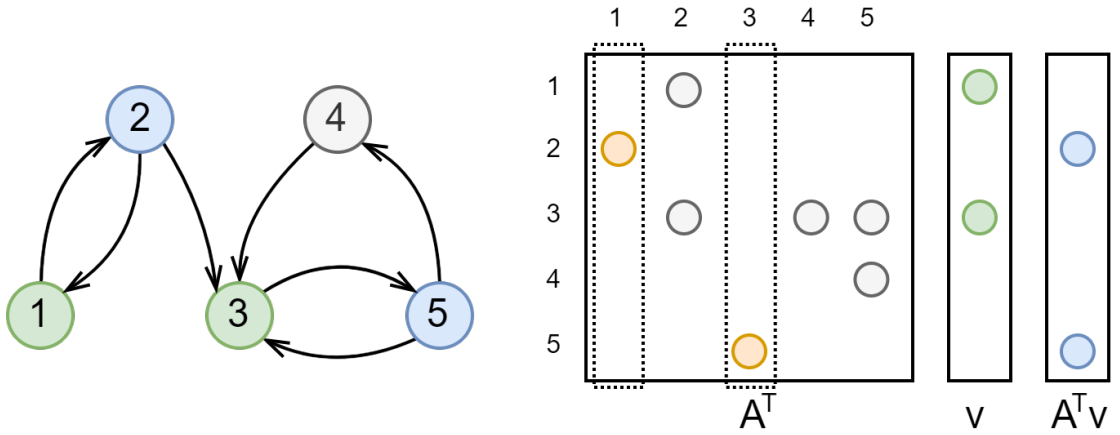


Figure 1: Graph traversal by matrix-vector product

2.2 GraphBLAS concepts

GraphBLAS standard [20] 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 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 values associated with them. 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.

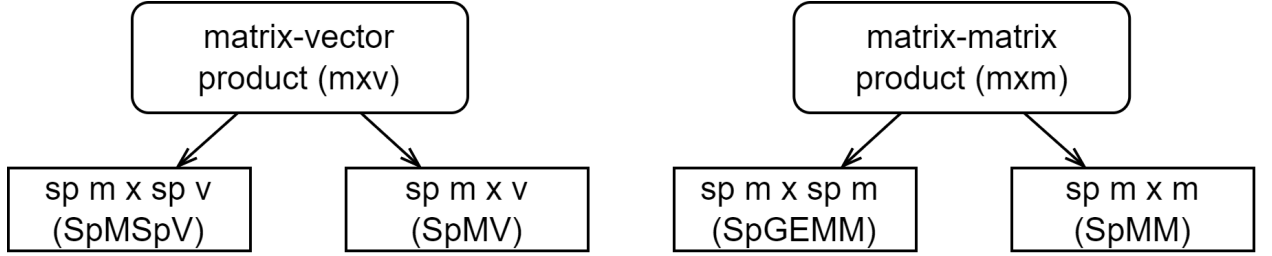


Figure 2: Key operations of the GraphBLAS standard and their implementations

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 out already visited vertices.

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 }

```

2.3 GraphBLAS limitations

Although GraphBLAS is a mature standard with a number of implementations, it has some limitations and shortcomings, discussed in a talk

given by John R. Gilbert [14]. Some of them are explained in the next paragraphs.

1. **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.
2. **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.
3. **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.
4. **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.
5. **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.

6. **GPU support.** GraphBLAS has no fully-featured implementation with GPU support. The primary reason for this is the complexity of the standard. There is a number of attempts to adopt GraphBLAS for a GPU analysis. But, most of them are focused only on Nvidia platform, what limits the portability of the potential solution.

Summarizing, the GraphBLAS is solid and mature standard with a number of some conceptual and technical shortcomings. Thus, it is important to overcome these technical limitations and address some conceptual problems designing similar yet distinct API, which can be wrapped with GraphBLAS API if required.

2.4 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, Metal for graphics and computations tasks, as well as OpenCL, SyCL, 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.

SyCL. It is a higher-level programming model to improve programming productivity on various hardware accelerators. It is a single-source embedded domain-specific language based on pure C++ standard language. The SyCL is a open-source standard maintained by a Khronos Group. SyCL allows to program heterogeneous computations within single C++ application, while the compiler infrastructure is used to translate SyCL specifics into specific computations backend. This is a new and promising technology, while its support across different compilers, devices and operating systems is still debatable.

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. OpenCL API is low-level. It gives precise control over execution and resources management. 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.

2.5 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

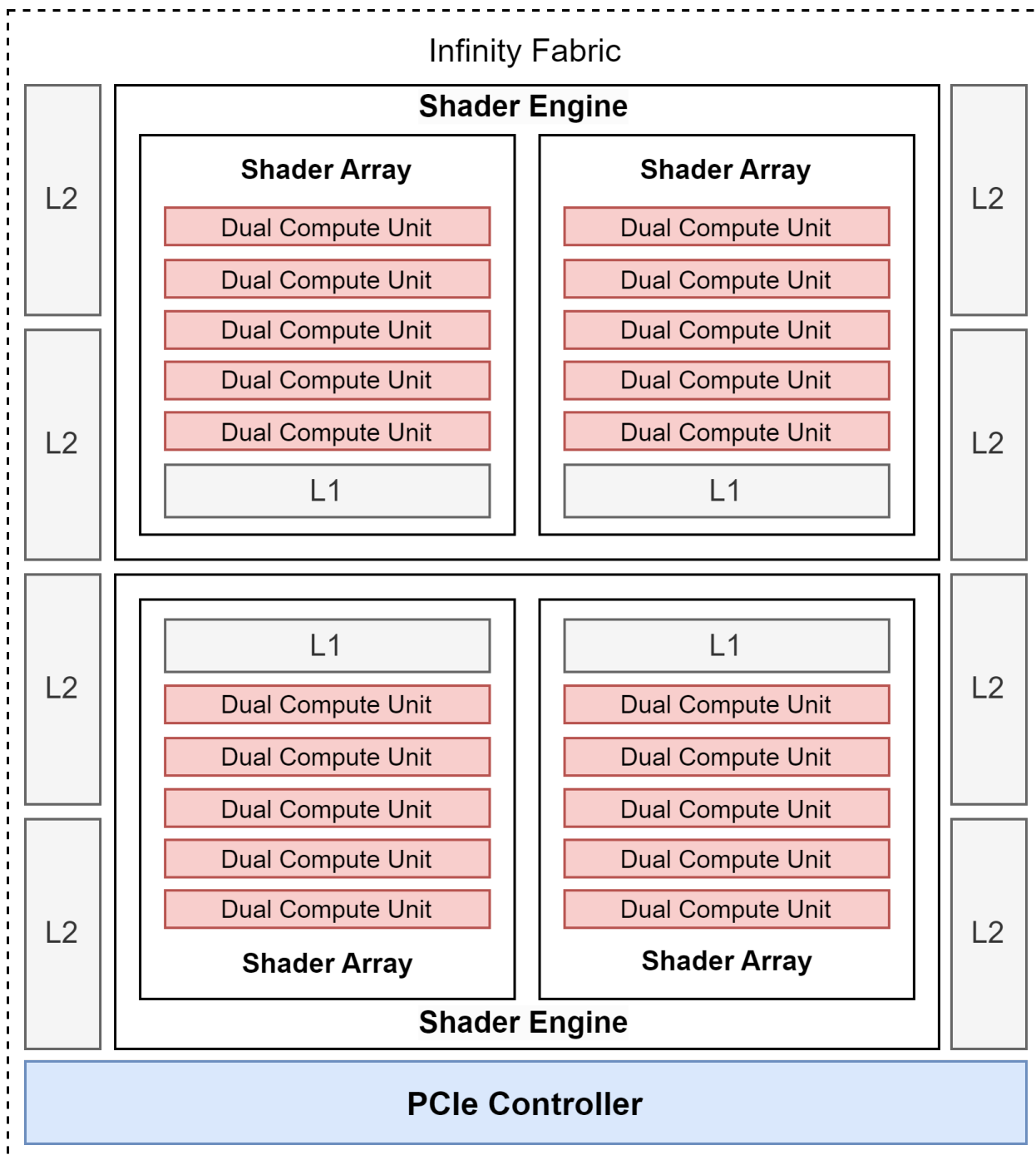


Figure 3: The block diagram of a Radeon RX 5700 XT GPU powered by the RDNA architecture.

communicate explicitly through local memory. The shader compiler subdivides the work groups further into micro-architecture specific wavefronts that are scheduled for parallel execution on a compute unit.

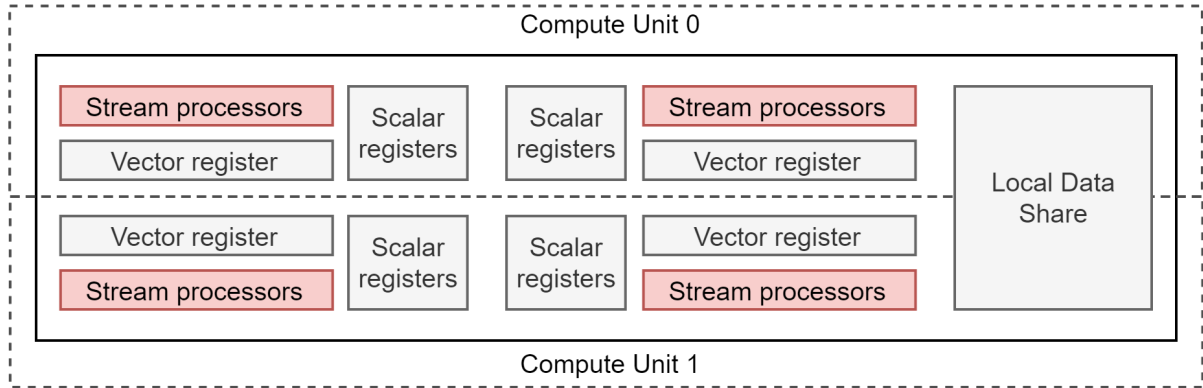


Figure 4: AMD Radeon DNA dual compute unit. Compute unit consist of a number of SIMD processors. Each processor has independent registers set. All processors share local memory, called local data share in AMD terms.

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.

2.6 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 implementation 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 a 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.

2.7 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 Proposed solution description

This section describes the high-level details of the proposed solution. It highlights the design principles, high-level architecture of the solution, data storage representation, operations, and also shows differences from the GraphBLAS API.

3.1 Design Principles

Spla library aims to address some of GraphBLAS standard limitations. It is designed the way to maximize potential library performance, to simplify its implementation and extension, and to provide the end-user verbose, but expressive 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 pluggable and optional part. Library can perform computations using standard CPU pipeline. If GPU acceleration is presented, library can offload a part of a work for it. It allows both non-trivial processing of the data on the CPU only, as well as possibility to integrate different backends in the future.
- **User-defined functions.** The user can create custom element-wise functions to parameterize operations. Custom functions can be used for both CPU and GPU execution.
- **Predefined scalar data types.** The library provides a set of built-in scalar data types that have a natural one-to-one relationship with native GPU built-in types. Data storage is transparent. The library interprets the data as POD-structures. The user can interpret individual elements as a sequence of bytes of a fixed size.
- **Hybrid-storage format.** The library automates the process of data storage and preprocessing. It supports several data formats, chooses the best one depending on the situation.

- **Exportable interface.** The library has a C++ interface with an automated reference-counting and with no-templates usage. It is compiled into a shared library. The interface is wrapped by C99 compatible API and exported to other languages, for example, in a form of a Python package.
- **Introspection.** Each library class instantiates into a first-class object. Such objects can be captured, manipulated, passed as arguments and returned as function results. Parameterization types of containers can be inspected, as well as declared user functions.

3.2 Architecture overview

The general idea of the proposed solution is depicted in Fig. 5. The core of the library and its main part is the CPU, which is the master node which controls all computations. It is responsible for storing data, maintaining a registry with algorithms, and scheduling operations to perform. In this paradigm, the GPU is an optional backend for acceleration, implemented through a special interface. It can optionally store data in a specific format. The CPU can offload the calculation of a part of the operations to the GPU, if the corresponding operation is supported by the given accelerator.

The reason for this is that the CPU and GPU are inherently asymmetric in nature. The end-user uses CPU side API. Thus, some preprocessing on the CPU side must be always done in the majority of cases.

In addition, access to data on the GPU and their storage is carried out differently due to the peculiarities of the execution of kernels. Also, VRAM is more expensive and has less capacity than RAM. Therefore, RAM is a cache for VRAM, and data duplication can be neglected.

In the end, the explicit separation of the CPU side from the GPU backend gives the modularity. This can be used not only to support different GPU technologies, but also to integrate multiple GPUs or distributed processing in the future.

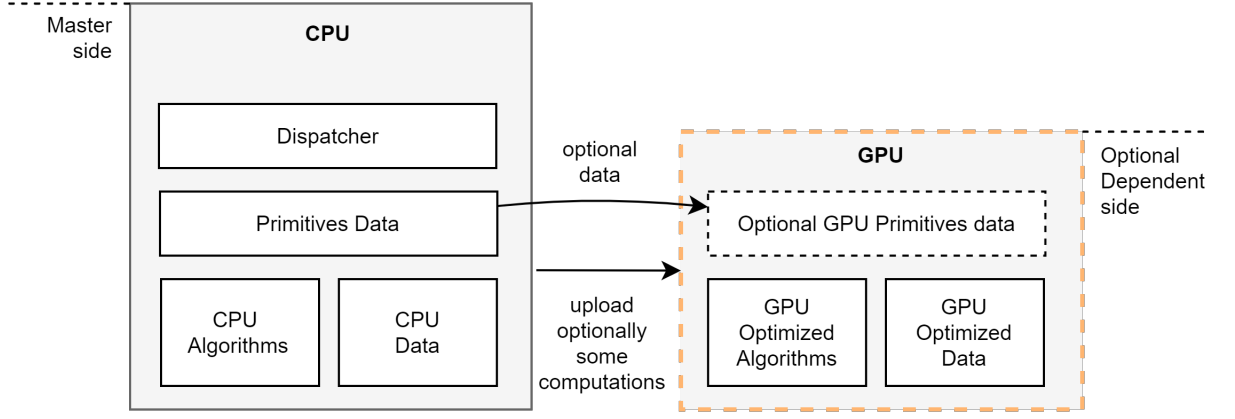


Figure 5: Library primary design idea. The CPU side of the library is a master. It is fully featured to perform computations. GPU is an optional acceleration. It can be used to offload some work. Data and algorithms support is optional.

3.3 Data Containers

Library provides general *M-by-N Matrix*, *N Vector*, *Scalar* and *N Array* data containers. Underlying primitive value types are specified by *Type* object. Single vector or matrix data is stored in specialized multi-format storage container. An example of the single vector storage is depicted in Fig. 6.

The storage is responsible for keeping data in multiple different formats at the same time. Each format is best suited for a specific type of task and requested on demand. Key-value dictionary suites well frequent insertion, query or deletion operations, when memory usage and response time are critical. Mathematical operations perform better with compacted sequential lists of values since they have more friendly cache behaviour. GPU operations require separate format with a copy of the data resident in VRAM.

The storage and particular format can be inspected using array primitive. It allows one to get the view of an existing CPU or GPU buffer without actual copy, or initialize matrix or vector in particular format from existing arrays, which may be created and filled by user code. Also array gives an ability to acquire raw pointer to memory or GPU buffer handler, what can be used for interoperability and seamless integration into user data pipeline.

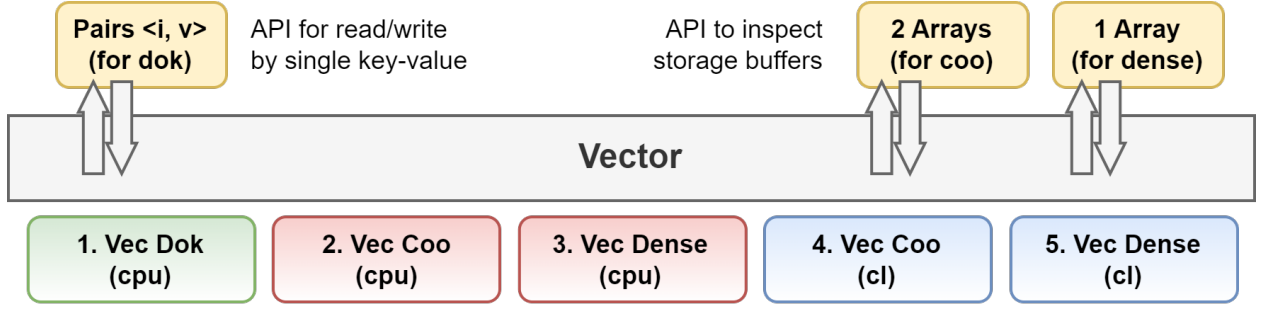


Figure 6: Vector decorations storage. Storage provides slots for different representations. Representation choice depends on a task currently being solved.

Data transformation from one format to another is carried out using a special rules graph. Example graph for a vector storage is shown in Fig. 7. The directed edges in this graph indicate conversion rules. The graph must be the single strongly connected component. An example of the data transformation process is depicted in Fig. 8. For a requested format the best path of convertation is obtained. Currently, the shortest one is used. Weight assignment to rules can potentially be used to prioritize convertations for some formats.

Currently, several storage formats are supported. There is dictionary of keys for vector and matrix (DoK), list of coordinates (COO), dense vector, list of lists (LIL) and compressed sparse rows (CSR) matrix formats. Other formats, such as CSC, DCSR, ELL, etc., can be added to the library by the implementation of formats conversion and by the specialization of operations for a specific format.

3.4 Algebraic Operations

Library provides a number of commonly used operations, such as *vxm*, *mxv*, *mxmT*, *element-wise add*, *assign*, *map*, *reduce*, etc. Other operations can be added on demand. Interface of operations is inspired by GraphBLAS standard. It supports *masking*, parametrization by *binary mult* and *binary add* functions, *select* for filtering and mask application, *unary op* for values transformation, and *descriptor* object for additional operation tweaking.

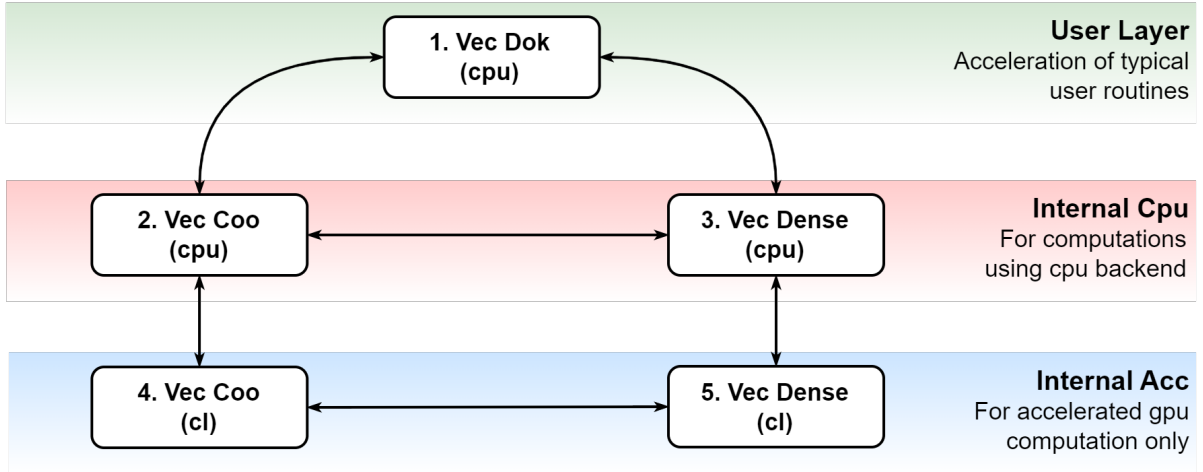


Figure 7: Vector decorations storage transformations graph. Graph declares transformation rules between different decoration formats. Existence of a path in this graph allows to convert one decoration into another automatically.

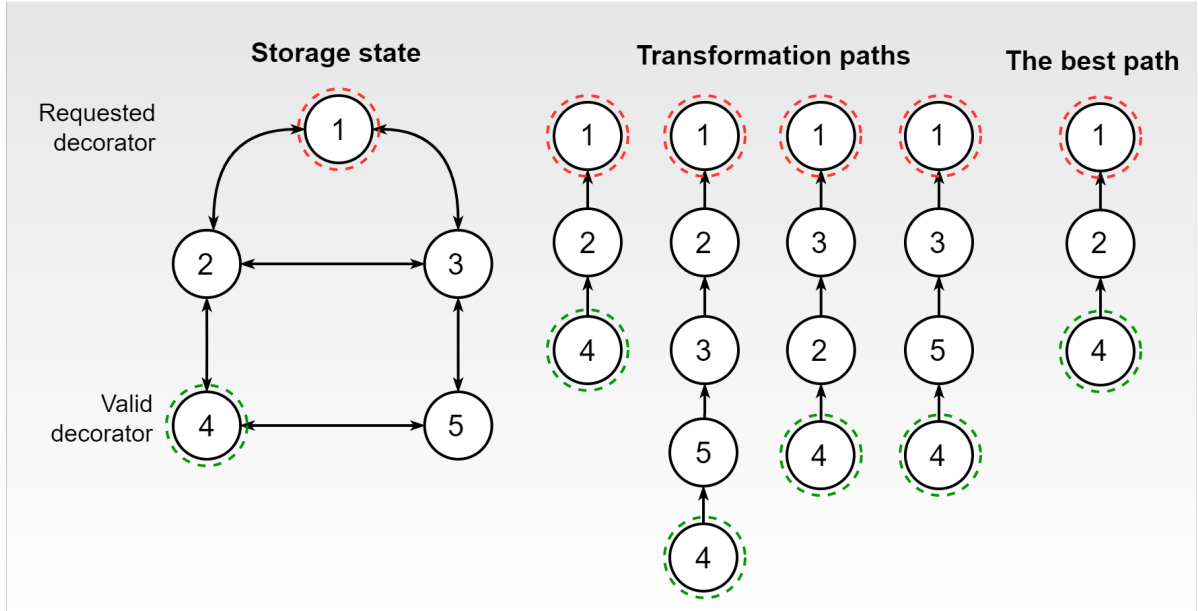


Figure 8: Vector storage transformation process. Target format is 1 (red circle). Source format is 4 (green circle). The shortest path is the best path for conversion.

3.5 Differences with GraphBLAS standard

To be clear, the proposed solution is not an implementation of GraphBLAS C or C++ API. The design of the library uses only the concepts described by the standard. Thus, the signatures and semantics of some of the operations have been changed in the proposed solution. The API has been made more verbose and explicit. In particular, the handling of *zero* elements and *masking* are made cleaner for the end user.

- **Interoperability (issue 1).** The library provides additional Array primitive for storage inspection. Array gives an ability to acquire a raw CPU data pointer or a GPU buffer pointer, which can be retained and used in a user-side applications without additional heavy and expensive copy operations.
- **Introspection (issue 2).** The library is designed with reflection and extra type information embedded into library objects. It allows user to inspect any created primitive and gives more runtime information.
- **Explicit zeros (issue 3).** The library interprets data simply as collections of bytes, without mathematical semantics. Identity element must be explicitly passed by the user where required. Special fill value used for sparse-dense convertations. It allows to make the memory usage predictable and the result of each operation clear to the end user without internal implicit storage manipulations.
- **Configurable masking (issue 4).** Mask applied using separate user-defined predicate for selection. Predicate explicitly selects arbitrary elements, not only zeros and not-zeros. It gives more flexibility than GraphBLAS concept and removes an ambiguity of a result and its memory consumption.
- **No templates usage (issue 5).** The API of the library is designed without C++ templates meta-programming. It allows to compile library once into a shared executable object, distribute it and use without extra manipulations.

- **GPU support (issue 6).** The library is designed in a modular fashion with a backend for GPU computations kept in mind. The API allows to express computations over matrices and vectors with user-defined functions, what can be accelerated using GPU as evaluation backend.












4 Implementation details

This section describes implementation details of the proposed solution. It highlights key aspects of the core implementation, OpenCL specifics, optimization of particular operations, and high-level optimizations of graph algorithms. It gives an insight into the selected storage formats, algorithms for GPU processing, and chosen third-party instruments for the library foundation.

4.1 Project structure

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* and *.h* format.
-  *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 Compile-time 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 Development 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 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_<T></i>	Set element of type T at index $\langle i, j \rangle$
<i>get_<T></i>	Get element of type T at index $\langle i, j \rangle$

4.4 Library 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 mxmT</i>	$R_{i,j} = (AB^T)_{i,j}, \forall i, j : f(M_{i,j})$	Masked matrix-matrix transposed product
<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>reduce by row</i>	$r_i = \Sigma M_{i,j}$	Matrix reduce by row to column vector
<i>ewise add</i>	$r_i = v_i + u_i$	Vector element-wise addition
<i>masked assign</i>	$r_i = s, \forall i : f(m_i)$	Masked vector scalar assignment
<i>map</i>	$r_i = g(v_i)$	Vector map to vector using unary function
<i>reduce</i>	$s = \Sigma M_{i,j}$	Matrix reduce to scalar
<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 be 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.

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
<i>gtzero</i>	$a > 0$	predicate	Greater than zero
<i>ltzero</i>	$a < 0$	predicate	Less than zero
<i>geero</i>	$a \geq 0$	predicate	Greater equals zero
<i>lezero</i>	$a \leq 0$	predicate	Less equals zero

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 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.

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);
```

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 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 Algorithms registry

The implemented library uses the concept of a registry to find operations as shown in Fig.9. A call to a particular operation is stored as a command to be executed later by *Dispatcher*. For each command the special lightweight string key is built depending on type of the operation and arguments passed. This key is used as a regex to get the required implementation of the requested operation. The advantages of the proposed approach are listed below.

- **Late binding.** The operation call becomes a command. The processing of such a command can be configured at run time. Changing the acceleration backend can be done without recompilation. Moreover, several backends can be transparently used within a single application.
- **Optionality of accelerator.** The acceleration backend is free to

support only those operations that require it. Fallback implementations will be used automatically for the rest of the operations.

- **Performance tuning.** The key of the command reflects operation type, arguments types, passed user functions types, etc. It can be used for *ad-hoc* optimizations. Custom operation implementation with a verbose key can be also stored in the registry. If several operations match the key, the longest key is used, since it is more specific for a particular operation.
- **Scheduling.** The full list of submitted commands for execution can be examined at runtime. This opens up the possibility for the fusion of some operations, sorting, rearrangement, and any other high-level optimizations that require introspection.

The structure of keys is depicted in a figure 10. 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 computations. Using accelerator suffix allows to switch between backend at runtime and select the most optimized algorithm.

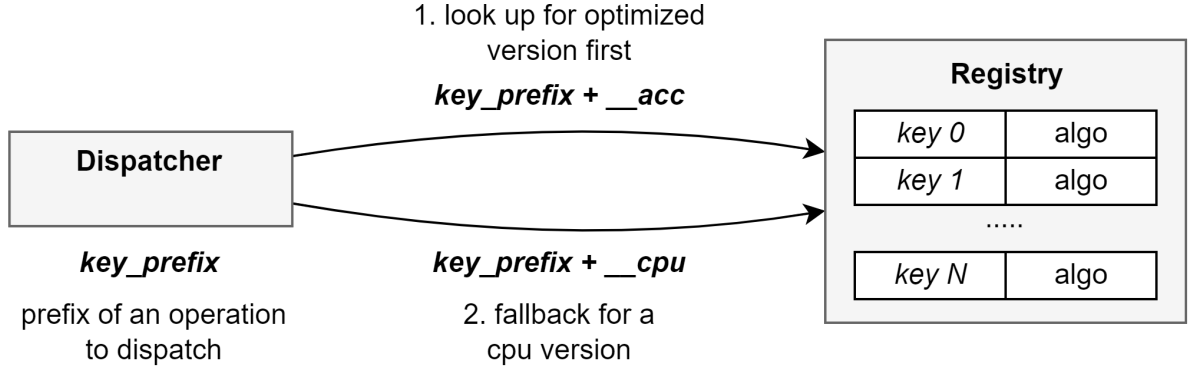


Figure 9: Registry of algorithms. Dispatcher looks up for optimized algorithms first. As a fallback it uses cpu suffix to get default algorithm implementation without an acceleration.

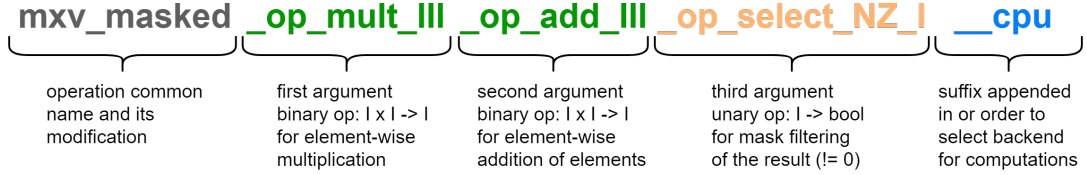


Figure 10: The structure of an algorithm key. The is a string literal composed from several parts. Prefix shows the algorithm name and its parametrization by operations. The suffix of the key shows which backed or accelerator to use for the evaluation of the algorithm.

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.

4.6 Storage formats

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 list of lists (LiL).** This format is used for an incremental matrix building on a CPU side as well as DoK. The memory space used by this format is proportional to the number of non-zero entries. This format provides more efficient memory layout at the cost of slower search, insert and deletion operations. Thus, LiL is better suited for sequential ordered construction and for some operations on a CPU side, such as reduction, product, map, etc.
- **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.

4.7 OpenCL backend

OpenCL 1.2 is used as the primary API for backend GPU implementation. Header files with C and C++ definitions are supplied with the source code of the project. Official Khronos installable client driver (ICD) loader bundled within a library to load at runtime particular OpenCL implementation depending on running OS and GPU vendor.

Implementation of sparse linear algebra algorithms for a GPU requires auxiliary libraries for memory management, sorting, reducing, merging, scanning, etc. Nvidia Cuda platform features libraries such as Thrust and Cub. OpenCL lacks such support. All primitives for this project are implemented from a scratch in most cases. What is an extra challenge. Third-party library, such as Boost Compute [26], cannot be used, since it has significant runtime overhead, portability and performance issues, and lack of long term support.

User-defined functions for GPU usage are represented as strings with additional metadata, such as type of parameters, return types, unique id, etc. Source code of particular operations stored in a form of .cl files. Operations implemented with generalization for parameters types and user functions. Their definitions obtained later at runtime in a compilation step through the text pre-processing.

Compilation of actual OpenCL kernels is done on demand. All compiled kernels are stored in a cache. Cache key is composed from types of kernel parameters, defines, etc., which identify uniquely a particular variation of a kernel. Key composition is done in $O(1)$.

In-place allocation is utilized for a key builder to avoid global heap usage. In order to reduce CPU overhead and keep access to the cache fast, library uses robin hood hashing based hash map.

Custom linear memory allocator implemented in order to reduce the overhead of frequent and small buffer allocations, arising in a time of execution of some operations. Allocator uses sub-buffer mechanism and serves request typically less than 1 MB of size. Otherwise, the general GPU heap is used.

4.8 Linear Algebra Operations

The following primitives are the core of computations: *masked sparse-vector sparse-matrix product*, *masked sparse-matrix dense-vector product* and *masked sparse-matrix sparse-matrix product*. Efficient implementation and load balancing of those operations dominate the performance of particular algorithms. The following paragraphs give an insight into these operations implementation in the library.

Masked sparse-vector sparse-matrix product. The implementation is based on the algorithm proposed by Yang et al. [29]. It is a k -way merge based algorithm which suites well for sparse vectors. Our implementation uses custom gather to collect temporary products. Radix sort used to sort products for further reduction. Reduction by key uses parallel prefix scan to carry out final destination of reduced values.

Masked sparse-matrix dense-vector product. The implementation of this operation relies on a classic row-based parallel algorithm. Both scalar and vector versions are implemented to fit better relatively sparse and dense matrix rows.

Masked sparse-matrix sparse-matrix product. The implementation of this algorithm uses the approach proposed by Yang et al. [28]. It is straightforward single-pass row-major and column-major matrix product. Mask is used to estimate the size of the final result to filter out some result of the product.

4.9 Graph Algorithms

The advantage of the linear algebra approach is that graph algorithms can be easily composed of primitive operations using a few lines of code. For preliminary study breadth-first search (BFS), single-source shortest paths (SSSP), page rank (PR) and triangles counting (TC) algorithms were chosen. These are the most commonly evaluated graph algorithms. They allow one to test basic operations and key aspects of graph frameworks performance. Implementation details for chosen algorithms are given below.

BFS. It utilizes a number of optimizations described by Yang et al. [27]. It uses masking to filter out already reached vertices, change of direction (push-pull) to switch from sparse from to dense and vice versa, and early exit in *mxv* operation.

SSSP. This algorithm uses change of direction as well. Also, it employs filtering of unproductive vertices according to Yang et al. [28]. Vertices which do not relax their distance in current iteration are removed from a front of the search. It keeps workload moderate.

PR. This algorithm assigns numerical weights to objects in the network depending on their relative relevance. As a key operation it uses *mxv* operation with a dense vector. For error estimation it uses custom element-wise function with a fusion of subtraction and square operations.

TC. Triangles counting uses masked sparse matrix product [28] and reduction. As an input algorithm accepts a lower triangular component L of an adjacency matrix of the source graph. The result is a count of non-zero values from $B = LL^T.*L$, where $.*$ used for the masking. The second argument is not actually transposed, since row-column based product gives exactly the required effect.

4.10 Running example

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.

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
14    front_prev → set_int(s, 1);
15
16    while (!(front_size → as_int() == 0)) {
17        depth → set_int(current_level);
18
19        exec_v_assign_masked(v, front_prev, depth, SECOND_INT, NQZERO_INT, desc);
20        exec_vxm_masked(front, v, front_prev, A, BAND_INT, BOR_INT, EQZERO_INT, zero, desc);
21        exec_v_reduce(front_size, zero, front, PLUS_INT, desc);
22
23        current_level += 1;
24
25        std::swap(front_prev, front);
26    }
27
28    return Status::Ok;
29 }

```

5 Evaluation

For performance analysis of the proposed solution, a few most common graph algorithms were evaluated using real-world sparse matrix data. The following tools for comparison were chosen: LaGraph [17] in connection with SuiteSparse [9] as a baseline CPU tool, Gunrock [16] and GraphBLAST [28] as a Nvidia GPU tools. Also, algorithms were tested on several devices with distinct OpenCL vendors in order to validate the portability of the proposed solution.

5.1 Research questions

In general, these evaluation intentions are summarized in the following research questions.

- RQ1** What is the performance of the proposed solution relative to existing tools for GPU analysis?
- RQ2** What is the performance of the proposed solution on various devices vendors and OpenCL runtimes?
- RQ3** What is the performance of the proposed solution on integrated GPU compared to existing CPU tool for analysis?

5.2 Evaluation setup

For evaluation of RQ1, a PC with Ubuntu 20.04 installed used, 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.

For evaluation of RQ2, a PC with Ubuntu 22.04 installed used, which has 4.70Hz AMD Ryzen 9 7900x 12-core CPU, DDR4 128 GB RAM, AMD GFX1036 integrated GPU, and either Intel Arc A770 flux dedicated GPU with 8GB on-board VRAM or AMD Radeon Vega Frontier Edition dedicated GPU with 16GB on-board VRAM.

For evaluation of RQ3, the first PC with Intel CPU and integrated GPU and the second PC with AMD CPU and integrated GPU are used.

Spla and LaGraph were compiled with GCC v9.4. Gunrock and GraphBLAST were compiled with GCC v8.4 and Nvidia NVCC v10.1. Release mode and maximum optimizations level were enabled for all tested programs.

All tests are averaged across 10 runs. The deviation of measurements does not exceed the threshold of 10 percent. Additional warm-up run for each test execution is excluded from measurements. Only actual execution time of algorithms is measured. Data loading time, preparation, format transformations, and host-device initial communications are excluded from time measurements. For measurements standard official benchmarks are used, which provided by compared tools developers.

5.3 Graph algorithms

For preliminary study *breadth-first search* (BFS), *single-source shortest paths* (SSSP), *page rank* (PR) and *triangles counting* (TC) algorithms were chosen. Implementation of those algorithms for competitors is used from official source code repositories with default parameters. Compared tools are allowed to make any optimizations as long as the result remains correct. The graph vertex with index 1 is set as the initial traversal vertex in the algorithms BFS and SSSP for all tested instruments and all tested devices.

5.4 Dataset

Thirteen matrices with graph data were selected from the Sparse Matrix Collection at University of Florida [10]. Information about graphs is summarized in Table 4. This is a common graphs collection used for sparse linear algebra and graph algorithms benchmarks in other works in the domain. These graphs represents typical analysed data structure, sparsity, distribution, and allows one to study common behaviour and performance characteristics of developed libraries.

Table 4: Dataset description.

Graph	Vertices	Edges	Out Degree		
			Avg	Sd	Max
coAuthorsCit	227.3K	1.6M	7.2	10.6	1.4K
coPapersDBLP	540.5K	30.5M	56.4	66.2	3.3K
amazon2008	735.3K	7.0M	9.6	7.6	1.1K
hollywood2009	1.1M	112.8M	98.9	271.9	11.5K
comOrkut	3.1M	234.4M	76.3	154.8	33.3K
citPatents	3.8M	33.0M	8.8	10.5	793.0
socLiveJournal	4.8M	85.7M	17.7	52.0	20.3K
indochina2004	7.4M	302.0M	40.7	329.6	256.4K
belgiumosm	1.4M	3.1M	2.2	0.5	10.0
roadNetCA	2.0M	5.5M	2.8	1.0	12.0
rggn222s0	4.2M	60.7M	14.5	3.8	36.0
rggn223s0	8.4M	127.0M	15.1	3.9	40.0
roadcentral	14.1M	33.9M	2.4	0.9	8.0

The dataset is converted to undirected graphs. Self-loops and duplicated edges are removed. Average, sd and max metrics relate to out degree property of the vertices. For SSSP weights are initialized using pseudo-random generator with uniform $[0, 1]$ distribution of floating-point values.

Graphs are roughly divided into two groups. The first group represents relatively dense graphs, where the number of edges per node is sufficient on average to effectively load the GPU with useful work. The second group represents relatively sparse graphs, where the average vertex degree is below the typical GPU vector register size, and the search depth reaches hundreds of hoops. Graphs are sorted in ascending order by the number of vertices within each group.

5.5 Results Summary

Fig. 11 presents results of the evaluation and compares the performance of Spla against other Nvidia GPU tools and uses as a baseline LaGraph CPU tool. Fig. 12 presents result of the portability analysis of the proposed

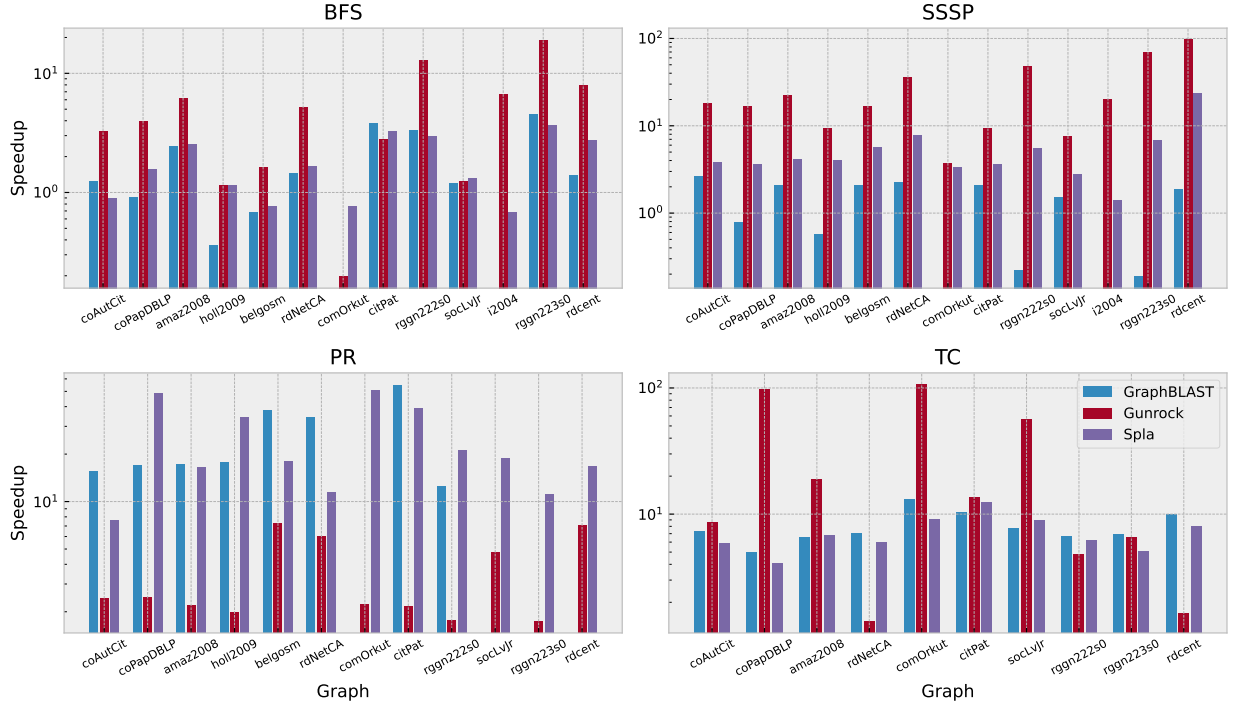


Figure 11: Performance of Spla library and GPU tools on the same device compared to LaGraph. Logarithmic scale is used.

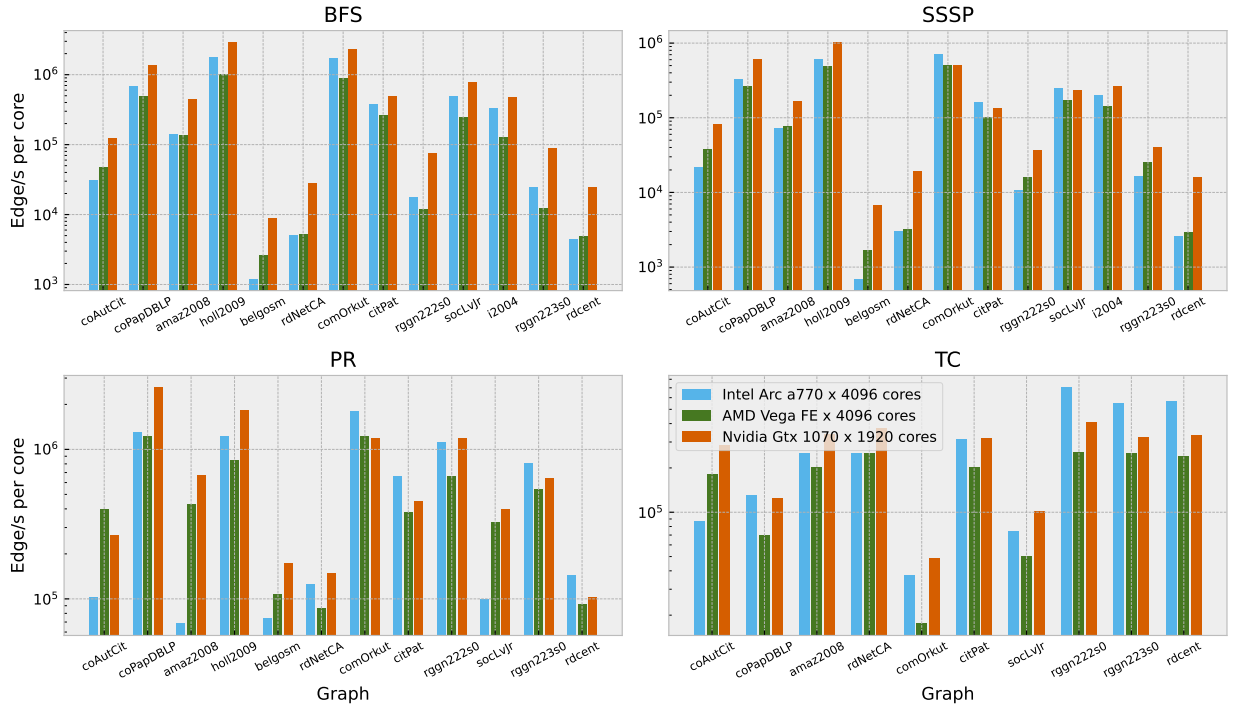


Figure 12: Performance of Spla library on different devices relative to the number of compute cores. Logarithmic scale is used.

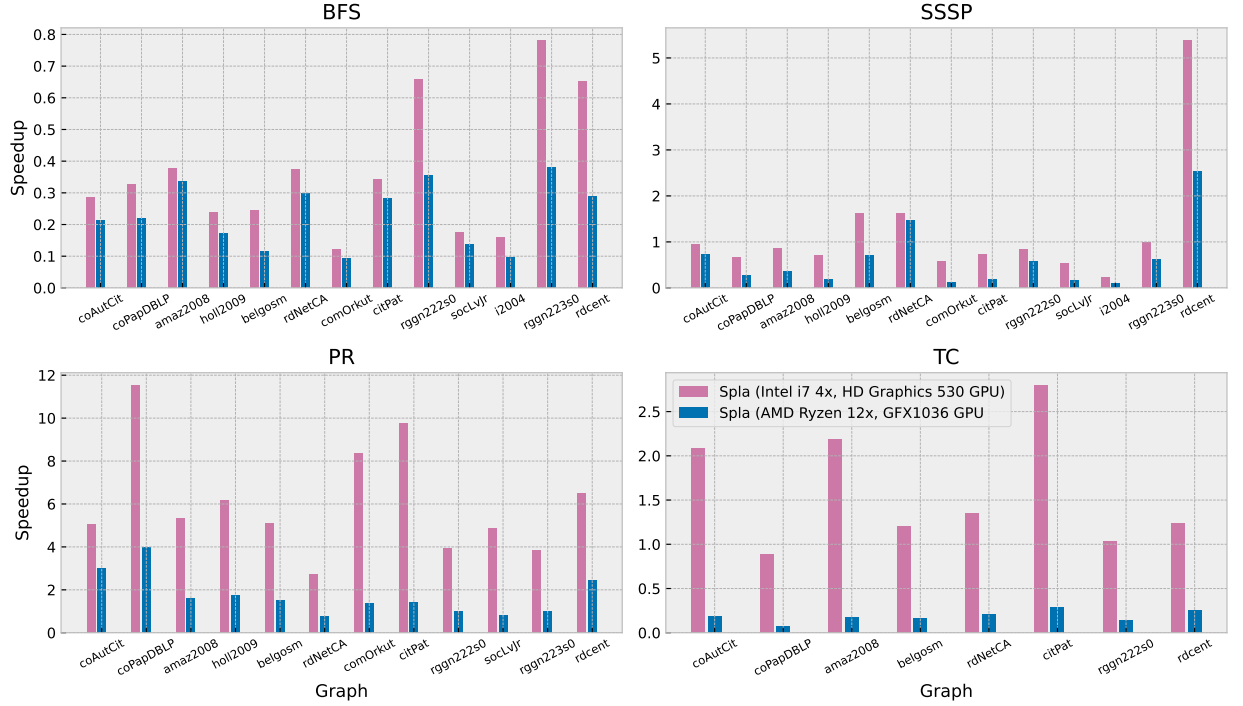


Figure 13: Performance of Spla library on integrated GPU compared to LaGraph on the same chip.

solution. It shows performance of the proposed solution on discrete GPUs of distinct vendors. Fig. 13 present result of per-device comparison of Spla library running on integrated GPU and CPU LaGraph running on the same chip.

The absolute results of the performance study are available in the Table 5, Table 6 and Table 7 for each stated research question. Cells left with *none* if 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 GPU analysis? In general, Spla shows very acceptable performance in all algorithms, running with comparable speed to its nearest competitor, GraphBLAST. Also proposed library does not suffer from memory issues on some large graphs such as *indochina*, *orkut* and *rggn23*. Spla is consistently several times faster than LaGraph, overcoming it up to $25\times$ in some cases. Gunrock is the fastest GPU framework for analysis. It dominates the overall performance and only suffers in a PR algorithm.

Taking a closer look at Fig. 11, Spla-based BFS shows comparable to

GraphBLAST performance in most cases. Spla has good speed at relatively dense graphs with high vertex degree and small depth of the search, what allows to saturate GPU with a work better. However, the performance degrades in network and road graphs with small front of the search and large diameter, what cause a lot of iterations. Thus, both Spla and GraphBLAST suffer from the overhead of kernel launches and relatively small amount of the work for a GPU. SSSP shares with BFS the same picture in general. However, Spla behaves here slightly better than GraphBLAST, running up to $36\times$ faster at some extreme cases.

For PR, Spla and GraphBLAST show the best performance, except cases with GraphBLAST memory issues. Both tools are faster than Gunrock in average reaching up to $20\times$ and more relative speedup. This performance can be motivated by the usage of *mxv* operation as a core primitive for pull-updates, which is computationally intensive and has good work load balance compared to Gunrock push-updates. However, Spla suffers a bit in case of lower-degree graphs due to lack of more precise balance for small matrix rows.

Finally, Gunrock dominates performance in TC as well, except two sparse road graphs where it has significant performance drop down. Spla and GraphBLAST have comparable results. However, GraphBLAST slightly faster nearly in all runs. Both tools use the same approach for *mxm* implementation. However, Spla may encounter some OpenCL overhead or lack of precise performance tuning.

RQ2. What is the performance of the proposed solution on various devices vendors and OpenCL runtimes? Spla successfully launches and workes on the GPU of distinct vendors, including Intel, AMD and Nvidia. It shows promising performance and demonstrated scalability in relation to the number of computing cores. Fig. 12 depicts the edge/s throughput per a GPU core for all devices. This metric is quite predictable for the same graphs. This can be seen if one takes into account the overall shape of the figures for BFS, SSSP and PR as a whole.

In general, Spla on Nvidia shows better average performance, especially

for sparser graphs with relatively small degree per row. Nvidia OpenCL driver features faster memory allocations and has less overhead on a frequent kernel launches. Spla on Intel runtime lags a bit behind Nvidia, but performs better in some PR and TC cases. Spla performance on AMD is acceptable. However, better tuning and further polishing are required.

RQ3. What is the performance of the proposed solution on integrated GPU compared to existing CPU tool for analysis? Result of detailed comparison are shown in Fig. 13. This figure depict Spla relative to LaGraph speedup on the same chip, where Spla is running on integrated GPU part and LaGraph is running on multi-core CPU part.

In general, LaGraph shows better performance for both CPUs, especially on a new powerful AMD Ryzen with 12 cores. The difference in a speed is extremely dramatic in BFS and SSSP algorithms. For a PR algorithm the picture is slightly better. Spla shows up to $10\times$ speedup. PR algorithm tends to be more computationally intensive, so difference to BFS and SSSP is reasonable. For TC Spla performs better only for Intel device, having in some cases conservative $2\times$ speedup.

Summarizing, the evaluation of the proposed solution for some real-world graph data in four different algorithms shows, that OpenCL-based solution has a promising performance, comparable to analogs, has acceptable scalability on devices of different GPU vendors, and, surprisingly, has a speedup in some cases when compared with highly-optimized CPU library on some integrated GPUs. However, there are still a plenty of research questions and directions for improvement.

Table 5: RQ1. Performance comparison of the proposed solution. Time in milliseconds (lower is better).

Dataset	GraphBLAST	Gunrock	LaGraph	Spla (proposed)
BFS				
coAuthorsCit	5.0	1.9	6.3	6.9
coPapersDBLP	19.9	4.5	18.0	11.5
amazon2008	8.3	3.3	20.4	8.1
hollywood2009	64.3	20.3	23.4	20.3
belgiumosm	200.6	84.4	138.0	181.2
roadNetCA	116.3	32.4	168.2	101.7
comOrkut	none	205.0	40.6	53.2
citPatents	30.6	41.3	115.9	35.1
rggn222s0	367.3	95.9	1228.1	415.3
socLiveJournal	63.1	61.0	75.5	57.1
indochina2004	none	33.3	224.6	328.7
rggn223s0	615.3	146.2	2790.0	754.9
roadcentral	1383.4	243.8	1951.0	710.2
SSSP				
coAuthorsCit	14.7	2.1	38.9	10.3
coPapersDBLP	118.6	5.6	92.2	25.7
amazon2008	43.4	4.0	90.0	21.7
hollywood2009	404.3	24.6	227.7	57.5
belgiumosm	650.2	81.1	1359.8	240.9
roadNetCA	509.7	32.4	1149.3	147.9
comOrkut	none	219.0	806.5	241.0
citPatents	226.9	49.8	468.5	129.3
rggn222s0	21737.8	101.9	4808.8	865.4
socLiveJournal	346.4	69.2	518.0	189.5
indochina2004	none	40.8	821.9	596.6
rggn223s0	59015.7	161.1	11149.9	1654.8
roadcentral	13724.8	267.0	25703.4	1094.3
PR				
coAuthorsCit	1.6	10.0	24.3	3.2
coPapersDBLP	17.6	120.2	297.6	6.1
amazon2008	5.2	40.6	89.8	5.5
hollywood2009	62.9	559.5	1111.2	32.4
belgiumosm	4.4	22.9	167.6	9.4
roadNetCA	6.6	37.7	225.8	19.6
comOrkut	none	2333.6	5239.0	103.3
citPatents	27.0	686.1	1487.0	38.3
rggn222s0	45.2	320.0	563.5	26.6
socLiveJournal	none	445.9	2122.5	112.0
rggn223s0	none	662.7	1155.6	103.4
roadcentral	none	408.8	2899.9	172.0
TC				
coAuthorsCit	2.3	2.0	17.3	3.0
coPapersDBLP	105.2	5.3	520.8	128.4
amazon2008	11.2	3.9	73.9	10.8
roadNetCA	6.5	32.4	46.0	7.7
comOrkut	1776.9	218.0	23103.8	2522.0
citPatents	65.5	49.7	675.0	54.5
socLiveJournal	504.3	69.2	3886.7	437.8
rggn222s0	73.2	101.3	484.5	77.7
rggn223s0	151.4	158.9	1040.1	204.2
roadcentral	42.6	259.3	425.3	52.7

Table 6: RQ2. Portability of the proposed solution. Time in milliseconds (lower is better).

Dataset	Intel Arc A770	AMD Vega Frnt. Edt.	Nvidia Gtx 1070
BFS			
coAuthorsCit	12.8	8.3	6.9
coPapersDBLP	10.8	14.9	11.5
amazon2008	12.3	12.6	8.1
hollywood2009	15.3	26.7	20.3
belgiumosm	627.5	292.4	181.2
roadNetCA	265.5	259.8	101.7
comOrkut	33.2	63.6	53.2
citPatents	21.0	30.3	35.1
rggn222s0	825.3	1259.7	415.3
socLiveJournal	43.0	85.8	57.1
indochina2004	220.6	573.4	328.7
rggn223s0	1245.5	2519.6	754.9
roadcentral	1864.9	1680.8	710.2
SSSP			
coAuthorsCit	18.3	10.4	10.3
coPapersDBLP	22.9	27.7	25.7
amazon2008	23.4	22.2	21.7
hollywood2009	44.6	56.2	57.5
belgiumosm	1085.9	454.8	240.9
roadNetCA	447.3	422.5	147.9
comOrkut	79.7	111.5	241.0
citPatents	49.8	78.4	129.3
rggn222s0	1378.8	924.3	865.4
socLiveJournal	82.7	120.7	189.5
indochina2004	366.2	519.0	596.6
rggn223s0	1880.2	1201.4	1654.8
roadcentral	3176.3	2848.8	1094.3
PR			
coAuthorsCit	3.9	1.0	3.2
coPapersDBLP	5.7	6.1	6.1
amazon2008	25.2	4.0	5.5
hollywood2009	22.6	32.4	32.4
belgiumosm	10.2	7.1	9.4
roadNetCA	10.8	15.7	19.6
comOrkut	31.9	46.6	103.3
citPatents	12.3	21.3	38.3
rggn222s0	13.4	22.4	26.6
socLiveJournal	210.0	64.2	112.0
rggn223s0	38.6	57.2	103.4
roadcentral	57.9	89.6	172.0
TC			
coAuthorsCit	4.6	2.2	3.0
coPapersDBLP	57.6	106.2	128.4
amazon2008	6.9	8.5	10.8
roadNetCA	5.4	5.4	7.7
comOrkut	1533.5	3267.6	2522.0
citPatents	25.9	39.8	54.5
socLiveJournal	280.6	420.3	437.8
rggn222s0	21.0	57.8	77.7
rggn223s0	56.7	123.2	204.2
roadcentral	14.5	34.6	52.7

Table 7: RQ3. Integrated GPU mode performance comparison of the proposed solution. Time in milliseconds (lower is better).

Dataset	Intel i7, HD Graphics 530		AMD Ryzen 9, GFX1036	
	LaGraph	Spla (proposed)	LaGraph	Spla (proposed)
BFS				
coAuthorsCit	7.5	26.3	3.9	18.2
coPapersDBLP	18.7	57.3	12.0	54.9
amazon2008	24.6	65.0	13.5	40.0
hollywood2009	23.8	100.1	14.8	86.6
belgiumosm	131.4	536.0	60.0	527.6
roadNetCA	173.2	461.8	100.8	339.7
comOrkut	41.6	341.4	25.2	269.4
citPatents	126.9	371.6	61.3	217.7
rggn222s0	1288.0	1959.9	644.6	1821.7
socLiveJournal	75.0	429.8	41.6	301.6
indochina2004	228.5	1424.8	137.0	1445.1
rggn223s0	2850.8	3647.2	1403.9	3701.3
roadcentral	2087.8	3196.3	767.2	2670.3
SSSP				
coAuthorsCit	40.5	42.5	29.2	40.5
coPapersDBLP	92.9	141.8	48.9	181.6
amazon2008	97.4	114.4	48.3	131.3
hollywood2009	236.7	337.9	93.8	507.4
belgiumosm	1383.2	854.3	588.9	845.7
roadNetCA	1174.2	721.7	712.7	482.9
comOrkut	822.9	1420.5	214.8	1699.5
citPatents	488.3	669.4	171.4	897.3
rggn222s0	4919.1	5928.3	2845.6	4952.9
socLiveJournal	534.7	1007.7	185.3	1205.1
indochina2004	837.1	3708.3	345.5	3971.8
rggn223s0	11375.6	11567.8	6099.6	9899.7
roadcentral	26314.1	4887.0	7867.2	3102.0
PR				
coAuthorsCit	25.3	5.0	17.6	5.9
coPapersDBLP	302.3	26.2	154.5	39.0
amazon2008	93.0	17.5	36.0	22.4
hollywood2009	1109.8	179.9	531.7	300.7
belgiumosm	178.9	35.0	45.1	29.4
roadNetCA	236.9	86.9	67.6	86.2
comOrkut	4458.5	531.9	959.6	701.4
citPatents	1559.9	159.8	277.4	195.7
rggn222s0	576.7	145.9	275.1	270.2
socLiveJournal	2181.0	449.7	520.5	630.9
rggn223s0	1187.0	309.3	617.2	605.3
roadcentral	2995.8	461.4	993.7	409.8
TC				
coAuthorsCit	17.3	8.3	5.2	28.3
coPapersDBLP	534.1	604.2	129.4	1682.3
amazon2008	75.4	34.5	22.2	126.6
belgiumosm	28.1	23.4	11.3	67.8
roadNetCA	47.7	35.2	21.5	105.6
citPatents	693.1	247.6	170.5	589.3
rggn222s0	495.2	481.3	177.7	1218.1
roadcentral	438.8	355.8	176.6	679.7

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 were covered. General GPU programming challenges and pitfalls were 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 differences with GraphBLAS standard were stated.
- **The library was implemented accordingly to the developed architecture.** The core of the library, interface, data containers, built-in scalar data types, built-in element-wise functions, expressions processing, OpenCL backend functionality, common linear algebra operations implementations were covered. Several graph 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 proposed solution showed comparable performance to existing state-of-the-art solutions. The developed library showed a scalability on different device vendors GPUs. Also,

the proposed solutions got an acceptable performance on integrated GPUs in some cases even if compared with highly-optimized multi-core CPU frameworks.

All in all, there are still a plenty of research questions and directions for improvement. Some of them are listed bellow.

- **Performance tuning.** There is a still space for optimizations. Better workload balancing must be done. Performance must be improved on AMD and Intel devices. More optimized algorithms must be implemented, such as SpGEMM algorithm proposed by Nagasaka et al. [21] for general $m \times m$ operation.
- **Operations.** Additional linear algebra operations must be implemented as well as useful subroutines for filtering, joining, loading, saving data, and other manipulations involved in typical graphs analysis.
- **Graph streaming.** The next important direction of the study is streaming of data from CPU to GPU for out-of-memory graphs analysis. CuSha adopt data partitioning techniques for graphs processing which do not fit single GPU. Modern GPUs have a limited VRAM. Even high-end devices allow only a moderate portion of the memory to be addressed by the kernel at the same time. Thus, manual streaming of the data from CPU to GPU is required in order to support analysis of extremely large graphs, which count billions of edges to process.
- **Multi-GPU.** Finally, scaling of the library to multiple GPUs must be implemented. Gunrock shows, that such approach can increase overall throughput and speedup processing of really dense graph. In connection with a streaming, it can be an ultimate solution for a large real-world graphs analysis.

The library source code is published on a GitHub platform. It is available at <https://github.com/SparseLinearAlgebra/spla>.

References

- [1] 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>.
- [2] 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>.
- [3] 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>.
- [4] Greathouse Joseph L., Knox Kent, Pola Jakub, Varaganti Kiran, and Daga Mayank. ClSPARSE: A Vendor-Optimized Open-Source Sparse BLAS Library // Proceedings of the 4th International Workshop on OpenCL. — New York, NY, USA : Association for Computing Machinery. — 2016. — IWOCL '16. — Access mode: <https://doi.org/10.1145/2909437.2909442>.
- [5] Coimbra Miguel E., Francisco Alexandre P., and Veiga Luís. An analysis of the graph processing landscape // Journal of Big Data. — 2021. — Apr. — Vol. 8, no. 1. — Access mode: <https://doi.org/10.1186/s40537-021-00443-9>.
- [6] 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.
- [7] Khorasani Farzad, Vora Keval, Gupta Rajiv, and Bhuyan Laxmi N. CuSha: Vertex-Centric Graph Processing on GPUs // Proceedings

of the 23rd International Symposium on High-Performance Parallel and Distributed Computing. — New York, NY, USA : Association for Computing Machinery. — 2014. — HPDC '14. — P. 239–252. — Access mode: <https://doi.org/10.1145/2600212.2600227>.

- [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] 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>.
- [12] Fu Zhisong, Personick Michael, and Thompson Bryan. MapGraph: A High Level API for Fast Development of High Performance Graph Analytics on GPUs // Proceedings of Workshop on GRaph Data Management Experiences and Systems. — New York, NY, USA : Association for Computing Machinery. — 2014. — GRADES'14. — P. 1–6. — Access mode: <https://doi.org/10.1145/2621934.2621936>.
- [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] Liu Weifeng and Vinter Brian. A Framework for General Sparse Matrix-Matrix Multiplication on GPUs and Heterogeneous Processors // J. Parallel Distrib. Comput. — 2015. — Nov. — Vol. 85, no. C. — P. 47–61. — Access mode: <https://doi.org/10.1016/j.jpdc.2015.06.010>.
- [20] 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.
- [21] Nagasaka Yusuke, Nukada Akira, and Matsuoka Satoshi. High-Performance and Memory-Saving Sparse General Matrix-Matrix Multiplication for NVIDIA Pascal GPU. — 2017. — 08. — P. 101–110.
- [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] 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.
- [24] 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.
- [25] Sparse matrix library (in Cuda). — Access mode: <https://docs.nvidia.com/cuda/cusparse/> (online; accessed: 16.04.2021).
- [26] Szuppe Jakub. Boost.Compute: A Parallel Computing Library for C++ Based on OpenCL // Proceedings of the 4th International Workshop on OpenCL. — New York, NY, USA : Association for Computing Machinery. — 2016. — IWOCL '16. — Access mode: <https://doi.org/10.1145/2909437.2909454>.
- [27] Yang Carl, Buluc Aydin, and Owens John D. Implementing Push-Pull Efficiently in GraphBLAS. — 2018. — Access mode: <https://arxiv.org/abs/1804.03327>.

- [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] Yang Carl, Wang Yangzihao, and Owens John D. Fast Sparse Matrix and Sparse Vector Multiplication Algorithm on the GPU // 2015 IEEE International Parallel and Distributed Processing Symposium Workshop. — 2015. — P. 841–847.
- [30] Zhong Jianlong and He Bingsheng. Medusa: Simplified Graph Processing on GPUs // IEEE Transactions on Parallel and Distributed Systems. — 2014. — Vol. 25, no. 6. — P. 1543–1552.
- [31] pygraphblas: a Python wrapper around the GraphBLAS API. — Access mode: <https://github.com/Graphegon/pygraphblas> (online; accessed: 28.12.2022).