High-Performance GraphBLAS-like API Implementation in Functional Style

1st Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

2nd Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

3rd Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

Semyon Grigorev
Saint Petersburg State University,
JetBrains Research,
St. Petersburg, Russia
s.v.grigoriev@spbu.ru,
semyon.grigorev@jetbrains.com

***Abstract*—Abstract is very abstract. Abstract is very abstract.**

***Index Terms*—graph analysis, sparse linear algebra, GraphBLAS API, GPGPU, parallel programming, functional programming, .NET, OpenCL**

I. Introduction

One of the promising ways to high-performance graph analysis is based on the utilization of linear algebra: operations over vectors and matrices can be efficiently implemented on modern parallel hardware, and once we reduce the given graph analysis problem to the composition of such operations, we get a high-performance solution for our problem. A well-known example of such reduction is a reduction of all-pairs shortest path (APSP) problem to matrix multiplication over appropriate *semiring*. GraphBLAS API standard [**?**] provides formalization and generalization of this observation and make it useful in practice. GraphBLAS API introduces appropriate algebraic structures (monoid, semiring), objects (scalar, vector, matrix), and operations over them to provides building blocks to create graph analysis algorithms. It was shown, that sparse linear algebra over specific semirings is useful not only for graph analysis, but also in other areas, such as computational biology [**?**] and machine learning [**?**].

There are a number of GraphBLAS API implementations, such as SuiteSarse:GraphBLAS [**?**] and CombBLAS [**?**], but all of them do not utilize the power of GPGPU, except GraphBLAST [**?**], while GPGPU utilization for linear algebra is a common practice today. GPGPU development is difficult itself because it introduces heterogeneous computational device, special programming model, and specific optimizations. Implementation of GraphBLAS API even more challenging, because it means the processing of irregular data, and the creation of generic (polymorphic) functions to declare and use user-defined semirings which is hard to express in low-level programming languages like CUDA C or OpenCL C which are usually used for GPGPU programming. Moreover, it is necessary to use high-level optimizations, like kernel fusion or elimination of unnecessary computations to improve the performance of end-user solutions based on the provided API implementation. But such high-level optimizations are too hard to automate for C-like languages.

Functional programming can help to solves these problems. First of all, native support functions as parameters simplify semirings descriptions and implementation of functions parametrized with semirings. Moreover, a powerful type system allows one to describe abstract (generic) functions which simplifies the development and usage of abstract linear algebra operations. Even more, such native features of functional programming languages, like discriminated unions (union types) and strong static typing allows one to create more robust code. For example, discriminated unions allows one naturally express `Min-Plus` semiring, where we should equip $\mathbb{R}$ with special element ∞ (infinity, namely identity element for $\oplus$), so we cannot use predefined types like `float` or `double`. Another area where functional programming can be useful is automatic code optimization. A big number of nontrivial optimizations for functional languages for GPGPU were developed, such as specialization, deforestation, and kernels fusion, one of the actively discussed optimizations in GraphBLAS community [**?**]. These techniques make programs in high-level programming languages competitive in terms of performance with solutions written in CUDA or OpenCL C. For more details one can look at such languages and

Identify applicable funding agency here. If none, delete this.

frameworks as Futhark[1] [**?**], Accelerate[2] [**?**], AnyDSL[3] [**?**].

In this work we discuss a way to implement GraphBLAS API which combines high-performance computations on GPGPU and the power of high-level programming languages in both application development and possible code optimizations. Our solution is based on metaprogramming techniques: we propose to generate code for GPGPU from a high-level programming language. Namely, we plan to generate OpenCL C from a subset of F# programming language. To translate F# to OpenCL C we use a Brahma.FSharp[4] which is based on F# quotations metaprogramming techniques[5]. Usage of F# simplifies both implementation of GraphBLAS API, making features of functional programming available, and its utilization in application development with high-level programming language on .NET platform. Moreover, as far as F# is a functional-first programming language, it should make it possible to use advanced optimization techniques and power of type system. Choice of OpenCL C as a target language is motivated by its portability: it is possible to run OpenCL C code on multi-thread CPU, on different GPGPUs (not only Nvidia), and even on FPGA [**?**], [**?**]. The utilization of FPGAs may open a way to hardware acceleration of sparse linear algebra and, as a result, of many solutions in different areas such as graph analysis, computational biology, machine learning.

This work in progress, so only tiny not optimized prototype is implemented, but our preliminary evaluation shows that !!!

II. Design Principles

Basic principles of proposed design described in this section. Here we will use .NET-like style for generic types: $\texttt{Type}_1\langle\texttt{Type}_2\rangle$ means that the type $\texttt{Type}_1$ is generic and $\texttt{Type}_2$ is a type parameter. Also we use F#-like type notations and syntax in our examples.

A. Types of Graphs, Matrices, and Operations

Suppose one have an edge-labelled graph G where labels have type $\texttt{T}_{\texttt{lbl}}$. Suppose also one declare a generic type $\texttt{Matrix}\langle\texttt{T}\rangle$ to use this type for graph representation where type parameter `T` is a type of matrix cell. It is obvious that type of cell of adjacency matrix of graph G should a special type which has only two values: some value of type $\texttt{T}_{\texttt{lbl}}$ or special value `Nothing`. This idea can be naturally expressed using discriminated unions (or sum types) which actively used not only in functional languages such as F#, OCalm, or Haskell, but also in TypeScript etc. Moreover, the described case is widely used and there is a standard type in almost all languages which supports discriminated unions: $\texttt{Option}\langle\texttt{T}\rangle$ in F# or OCaml, or $\texttt{Maybe}\langle\texttt{T}\rangle$ in Haskell. In F# this type defined as presented in listing **??**.

```
type Option<T> =
| None
| Some of T
```

Listing 1: `Option` type definition

Thus, to represent the graph G as a matrix one should use an instance of $\texttt{Matrix}\langle\texttt{Option}\langle\texttt{T}_{\texttt{lbl}}\rangle\rangle$ of generic type $\texttt{Matrix}\langle\texttt{T}\rangle$. This way we can explicitly separate non-zero and zero cells in terms of sparse matrix: non-zero cells are cells with value `Some(x)` for which `x` should be stored, and zero cells are cells with value `None`.

In these settings, natural type for binary operation is

$$\texttt{Option}\langle T_1\rangle \to \texttt{Option}\langle T_2\rangle \to \texttt{Option}\langle T_3\rangle.$$

But this type is not restrictive enough: it allows one to define operation which returns some non-zero (`Some(x)`) value for two zeroes (`None`-s), while we expect that

$$\texttt{None op None = None}$$

for any operation `op`.

To solve this problem one can introduce additional constraints, but such constraints can not be expressed in F#. An alternative solution is to introduce a type $\texttt{AtLeastOne}\langle T_1, T_2\rangle$ as presented in listing **??**. This type is less flexible (for example it disallows one to apply operation partially) but is explicitly shows that we expect that at least one argument of operation should be non-zero.

```
type AtLeastOne<T1,T2> =
| Both of T1 * T2
| Left of T1
| Right of T2
```

Listing 2: `AtLeastOne` type definition

Finally in this settings binary operations should have the following type:

$$\texttt{AtLeastOne}\langle T_1, T_2\rangle \to \texttt{Option}\langle T_3\rangle.$$

This type disallows one to build non-zero value from two zeroes, and explicitly shows whether result should be stored or not. Thus, proposed typing scheme solves problem of explicit and implicit zeroes. Moreover it allows to generalize element-wise operations. For example, binary operations for element-wise addition, element-wise multiplication, and even for masking can be specified as presented in listings **??**, **??**, **??** respectively.

B. Operations Over Matrices and Vectors

GraphBLAS API introduces **monoid** and **semiring** abstraction to specify element-wise operations for functions over matrices and vector. We propose to use binary operations

[1]Futhark is a purely functional statically typed programming language for GPGPU. Project web page: https://futhark-lang.org/. Access date: 12.01.2021.

[2]Accelerate: GPGPU programming with Haskell. Project web page:https://www.acceleratehs.org/. Access date: 12.01.2021.

[3]AnyDSL is a partial evaluation framework for parallel programming. Project web page: https://anydsl.github.io/. Access date: 12.01.2021.

[4]Brahma.FSharp project on GitHub: https://github.com/YaccConstructor/Brahma.FSharp. Access date: 12.01.2021.

[5]F# code quotations is a run time metaprogramming technique which allows one to transform written F# code during program execution. Official documentation: https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/code-quotations. Access date: 12.01.2021.

```
let op_int_add args =
match args with
| Both (x, y) ->
    ler res = x + y
    if res = 0
    then None
    else Some res
| Left x -> Some x
| Right y -> Some y
```

Listing 3: An example of element-wise addition operation definition

```
let op_int_mult args =
match args with
| Both (x, y) ->
    ler res = x * y
    if res = 0
    then None
    else Some res
| Left x -> None
| Right y -> None
```

Listing 4: An example of element-wise multiplication operation definition

instead as a parameters for functions over matrices and vectors. Using proposed types we always know that identity is always `None`, so we do not need specify identity separately as a part of semiring or monoid. Additionally, we often do not require operations actually satisfy semiring or monoid properties, so usage of correctly typed functions should be more clear and less confusing than usage of mathematical object in non-convenient settings.

So, for example function for matrix-matrix multiplication should has the following type:

$$\begin{aligned}
&\texttt{mXm}:\\
&\quad \texttt{mlt}:\ \texttt{AtLeastOne}\langle T_1, T_2\rangle \to \texttt{Option}\langle T_3\rangle\\
&\to \texttt{add}:\ \texttt{AtLeastOne}\langle T_3, T_3\rangle \to \texttt{Option}\langle T_3\rangle\\
&\to \qquad\qquad \texttt{m}_1 : \texttt{Matrix}\langle\texttt{Option}\langle T_1\rangle\rangle\\
&\to \qquad\qquad \texttt{m}_2 : \texttt{Matrix}\langle\texttt{Option}\langle T_2\rangle\rangle\\
&\to \qquad\qquad \texttt{result} : \texttt{Matrix}\langle\texttt{Option}\langle T_3\rangle\rangle
\end{aligned}$$

In this settings we can predefine a set of widely used binary operations, and allows user to specify own ones, and combine all of them freely and safely.

C. Fusion [in progress]

Runtime metaprogramming allows us to implement runtime fusion for kernels: having an expression over matrices and vectors we can build an optimized F# function from predefined building blocs, and after that translate this optimized version to OpenCL C.

This allows us to define **mask** as a partial case of generic element-wise function (respective operation is presented in listing **??**), not an optional parameter of other functions. This makes API more homogeneous and clear.

```
let op_mask args =
match args with
| Both (x, y) -> Some x
| Left x -> None
| Right y -> None
```

Listing 5: An example of masking operation definition

III. Implementation Details

To evaluate ideas described above we start a development of library named GraphBLAS#[6].

We use a Brahma.FSharp library for running time translation of F# code to OpenCL C, and for translated kernels execution. Brahma.FSharp is based on code quotations, thus utilizes strong typing to provide more static code checks, and polymorphic first class functions for general highly abstract code creation. Additionally, Brahma.FSharp provides special workflow builder to simplify heterogeneous programming and automate resource management.

Abstraction layers which hides details of matrix representation and operations implementation. Currently we are working on COO and CSR formats and respective operations.

IV. Evaluation

While our implementation is on very early stage, we cannot evaluate it on well-known linear algebra based algorithms. But in order to demonstrate applicability of proposed solutions we evaluate element-wise matrix-matrix operations.

We perform our experiments on the PC with Ubuntu 18.04 installed and with the following hardware configuration: !!! CPU, !!! RAM, !!!GPGPU with !!!!.

our solution on CPU and GPGPU. For comparison we choose the following libraries.

- SuiteSparse:GraphBLAS as a reference CPU implementation of GraphBLAS API.
- GraphBLAST as a most stable GPGPU implementation of GraphBLAS API.

A. Dataset

For evaluation we select a set of matrices from SuiteSparse matrix collection collection[7] To simplify evaluation of element-wise operations over matrices with different structure we precompute square of each matrix. Characteristics of selected matrices are presented in table **??**.

B. Evaluation Results

To benchmark .NET-based implementation we use *BenchmarkDotNet*[8] which allows one to automate benchmarking process for .NET platform. We run each function XXX times, !!! Time is measured in milliseconds.

Results of performance evaluation are presented in tables **??** and **??**.

[6]Sources of GraphBLAS# on GiHub: https://github.com/YaccConstructor/GraphBLAS-sharp. Access date: 12.01.2021.

[7]SuiteSparse matrix collection: https://sparse.tamu.edu/.

[8]*BenchmarkDotNet*: https://benchmarkdotnet.org/. Access date: 12.06.2022.

TABLE I
MATRICES FOR EVALUATION

Name	Size	NNZ	NNZ in square
wing	62 032	243 088	714,200
luxembourg_osm	114 599	119 666	4 582
amazon0312	400 727	3 200 440	14 390 544
amazon-2008	735 323	5 158 388	25 366 745
web-Google	916 428	5 105 039	30 811 855
webbase-1M	1 000 005	3 105 536	51 111 996
cit-Patents	3 774 768	16 518 948	469

TABLE II
EVALUATION RESULTS FOR CSR, GTX 2070, TIME IN MS

	GraphBLAS-sharp	SuiteSparse	CUSP
wing	$1,8 \pm 0,1$	$1,9 \pm 0,1$	$0,5 \pm 0,2$
luxembourg_osm	$2,9 \pm 0,3$	$1.9 \pm 0,5$	$0,5 \pm 0,1$
amazon0312	$17,0 \pm 0,8$	$28,9 \pm 0,2$	$2,8 \pm 0,1$
amazon-2008	$12,2 \pm 0,8$	$50,1 \pm 2,4$	$3,5 \pm 0,1$
web-Google	$18,4 \pm 0,6$	$58.8 \pm 0,7$	$3,6 \pm 0,1$
webbase-1M	$70,7 \pm 1,0$	$72,9 \pm 0,4$	$24,6 \pm 2,1$
cit-Patents	$54,6 \pm 1,2$	$157,4 \pm 1,2$	$8,5 \pm 1,2$

We can see, that for element-wise addition our implementation slower than SuiteSparse:GraphBLAS for small matrices (**luxembourg_osm**) and up to 4 times faster for big matrices. For element-wise multiplication results are almost similar except matrix **webbase-1M** for which our implementation slower than SuiteSparse:GraphBLAS while this matrix contains big number of non-zero values.

V. CONCLUSION AND FUTURE WORK

We present a work in progress that demonstrates a way to utilize both a power of high-level languages and performance of GPGPUs to implement GraphBLAS API. Our preliminary evaluation shows that !!!

Matrix-matrix multiplication

Matrix-vector multiplication (both sparse and dense)

Iterators: map, iter, filter.

Fusion for element-wise operations and iterators. Metaprogramming techniques.

Evaluation.

In the future, first of all, we should extend our library up to full GraphBLAS API implementation. Moreover, it may be useful for community to implement an analog of LAGraph[9] algorithms collection for .NET on the top of our GraphBLAS API implementation.

The next step is evaluation of the solution on real-world cases and comparison with other implementations of GraphPLAS API on different devices and different algorithms. Additionally, it may be interesting to compare our solution with graph analysis libraries and with linear algebra libraries for .NET platform.

Another direction of future work is Brahma.FSharp improvements. First of all, it is necessary to support discriminated unions to make it possible to express custom semirings such as `Min-Plus`, as presented in listing **??**.

[9]LAGraph is a collection of algorithms implemented using GraphBLAS. Project sources on GitHub: https://github.com/GraphBLAS/LAGraph. Access date: 12.01.2021.

TABLE III
EVALUATION RESULTS FOR ELEMENT-WISE MULTIPLICATION, GTX 2070, TIME IN MS

	GraphBLAS-sharp	SuiteSparse
wing	$2,5 \pm 0,4$	$1,0 \pm 0,1$
luxembourg_osm	$2,6 \pm 0,3$	$1,4 \pm 0,3$
amazon0312	$13,0 \pm 1,0$	$23,0 \pm 0,9$
amazon-2008	$9,1 \pm 0,8$	$35,2 \pm 4,0$
web-Google	$14,7 \pm 0,8$	$43,9 \pm 0,2$
webbase-1M	$55,4 \pm 1,2$	$31,0 \pm 1,6$
cit-Patents	$47,9 \pm 0,9$	$107,9 \pm 0,4$

Also, it is necessary to add high-level abstractions for both asynchronous programming and for multi-GPU programming. Such mechanisms can be naturally expressed in F# with native primitives for asynchronous programming, and by using high-level abstractions for multiple GPUs management.

Finally, we plan to implement high-level optimizations, like fusion and specialization in Brahma.FSharp.