GraphBLAS-like API Design in Functional Style

Kirill Garbar*, Igor Erin†, Artyom Chernikov‡, Dmitriy Panfilyonok§, Semyon Grigorev¶
St Petersburg University, St. Petersburg, Russia
Email: *st087492@student.spbu.ru, †igor.erin.a@gmail.com, ‡artem_chernikov00@list.ru, §dmitriy.panfilyonok@gmail.com, ¶s.v.grigoriev@spbu.ru

***Abstract*—GraphBLAS API standard describes linear algebra based blocks to build parallel graph analysis algorithms. While it is a promising way to high-performance graph analysis, there are a number of drawbacks such as complicated API, hardness of implementation for GPGPU, and explicit zeroes problem. We show that the utilization of techniques from functional programming can help to solve some GraphBLAS design problems.**

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

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 makes it useful in practice. GraphBLAS API introduces appropriate algebraic structures (monoid, semiring), objects (scalar, vector, matrix), and operations over them to provide 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 in itself because it introduces heterogeneous computational devices, special programming models, 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 solve 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 extend $\mathbb{R}$ 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 the 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 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].

This research is sponsored by Huawei.

[1]Futhark is a purely functional statically typed programming language for GPGPU. Project web page: https://futhark-lang.org/. Access date: 08.01.2023.

[2]Accelerate: GPGPU programming with Haskell. Project web page:https://www.acceleratehs.org/. Access date: 08.01.2023.

[3]AnyDSL is a partial evaluation framework for parallel programming. Project web page: https://anydsl.github.io/. Access date: 08.01.2023.

[4]Brahma.FSharp project on GitHub: https://github.com/YaccConstructor/Brahma.FSharp. Access date: 08.01.2023.

[5]F# code quotations is a runtime 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: 08.01.2023.

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. OpenCL C as a target language provides high portability: it is possible to run OpenCL C code on multi-thread CPU, on different GPGPUs (not only Nvidia but also Intel and AMD).

To summarize, we make the following contribution.

1) We propose a typing scheme for sparse matrices and show how types can naturally solve the *problem of explicit and implicit zeroes* which actively discussed in community[6].
2) We show that utilization of generic high-order functions can simplify API by kernels unification.
3) We implemented generic kernels for matrix-matrix element-wise functions using Brahma.FSharp. Evaluation of implemented kernels shows that it is possible to create performant flexible linear algebraic operations using functional programming features and metaprogramming techniques.

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 tor Graphs, Matrices, and Operations

Suppose one has an edge-labeled graph G where labels have type $\texttt{T}_{\texttt{lbl}}$. Suppose also one declares 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, Swift and other popular languages. 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 1.

```
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 cells which should be stored and which do not: for cells with value `Some(x) x` should be stored, and for cells with value `None` should not be stored. Note that there is no storage and data transfer overhead in this solution: one can use any format for sparse matrices and store only $\texttt{T}_{\texttt{lbl}}$ values as usual.

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

```
None op None = None
```

for any operation `op`.

To solve this problem one can introduce additional type-level constraints, but such constraints can not be expressed in F#. Actually, one needs more power type system which supports dependent types. An alternative solution is to introduce a type $\texttt{AtLeastOne}\langle T_1, T_2\rangle$ as presented in listing 2. This type is less flexible (for example it disallows one to apply operation partially) but it 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 a non-zero value from two zeroes, and explicitly shows whether the result should be stored or not. Thus, the proposed typing scheme solves the *problem of explicit and implicit zeroes*. Moreover it allows one to generalize element-wise operations. For example, binary operations for element-wise addition, element-wise multiplication, and even for masking can be described as presented in listings 3, 5, 6 respectively.

```
let op_int_add args =
    match args with
    | Both (x, y) ->
        let 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 which explicitly filter zeroes out

[6]Discussions related to *explicit zeroes problem*: https://github.com/lessthanoptimal/ejml/pull/145#issuecomment-888293732, https://github.com/GraphBLAS/LAGraph/issues/28. Access date: 08.01.2023.

```
let op_int_add args =
    match args with
    | Both (x, y) -> Some (x + y)
    | Left x -> Some x
    | Right y -> Some y
```

Listing 4: An example of element-wise addition operation which preserves zeroes

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

Listing 5: An example of element-wise multiplication operation definition

B. Operations Over Matrices and Vectors

GraphBLAS API introduces **monoid** and **semiring** abstraction to specify element-wise operations for functions over matrices and vectors. We propose to use binary operations instead as parameters for functions over matrices and vectors. Using proposed types we always know that identity is always `None`, so we do not need to specify identity separately as a part of semiring or monoid. Additionally, applications often do not require operations that actually satisfy semiring or monoid properties, so usage of correctly typed functions should be more clear and less confusing than usage of mathematical objects in non-convenient settings.

For example, function for element-wise matrix-matrix operations[7] should has the following type:

$$
\begin{aligned}
&\texttt{map2}: \\
&\quad \texttt{op: AtLeastOne}\langle T_1, T_2\rangle \to \texttt{Option}\langle T_3\rangle \\
&\to \quad m_1 : \texttt{Matrix}\langle\texttt{Option}\langle T_1\rangle\rangle \\
&\to \quad m_2 : \texttt{Matrix}\langle\texttt{Option}\langle T_2\rangle\rangle \\
&\to \quad \texttt{result} : \texttt{Matrix}\langle\texttt{Option}\langle T_3\rangle\rangle
\end{aligned}
$$

In these settings we can predefine a set of widely used binary operations, and allow users to specify their own ones, and combine all of them freely and safely. Moreover, this way allows one to introduce monoids and semirings as an additional level of abstraction. This way we can simplify core API: one should implement a relatively small set of high-order generic functions that unify functions from existing API.

C. Fusion

Kernels fusion is a way to reduce memory allocation for intermediate data structures and it is an important part of GraphBLAS-inspired way to high-performance graph analysis [**?**]. 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.

[7]We name it `map2` to be consistent with similar functions over standard collections.

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

Listing 6: An example of masking operation definition

While it is unlikely possible to implement general fusion, it should be possible to provide fusion for a fixed set of operations. This way it is important to minimize API which can be done using high-level abstractions as shown before. This allows us to define **mask** as a partial case of generic element-wise function (respective operation is presented in listing 6), not an optional parameter of other functions. This makes API more homogeneous and clear.

III. Evaluation

While our project[8] is in a very early stage we implemented and evaluated only generic matrix-matrix element-wise function `map2` to demonstrate that the proposed solution may provide not only a way to an expressive compact high-level API, but also a way to its high-performance implementation. This function is implemented for matrices in CSR format. We use Brahma.FSharp to support GPGPUs.

We evaluated `map2` function parameterized by two operations: `op_int_add` (listing 3) to get element-wise addition in terms of GraphBLAS API, and `op_int_mult` (listing 5) to get element-wise multiplication.

A. Environment

We perform our experiments on the PC with Ubuntu 20.04 installed and with the following hardware configuration: Intel Core i7–4790 CPU, 3.60GHz, 32GB DDR4 RAM with GeForce GTX 2070, 8GB GDDR6, 1.41GHz.

For comparison we choose a SuiteSparse:GraphBLAS as a reference CPU implementation of GraphBLAS API, and CUSP[9] as a most stable GPGPU implementation of generic sparse linear algebra.

TABLE I
Matrices for evaluation

Matrix	Size	NNZ	Squared matrix NNZ
wing	62 032	243 088	714 200
luxembourg_osm	114 599	119 666	393 261
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	29 710 164
webbase-1M	1 000 005	3 105 536	51 111 996
cit-Patents	3 774 768	16 518 948	469

[8]GraphBLAS# project page: https://github.com/YaccConstructor/GraphBLAS-sharp. Access date: 08.01.2023.

[9]Cusp is a CUDA-based library for sparse linear algebra and graph computations: https://cusplibrary.github.io/. Access date: 08.01.2023.

B. Dataset

For evaluation we selected a set of matrices from SuiteSparse matrix collection[10] To simplify the evaluation of element-wise operations over matrices with different structures we precomputed the square of each matrix. Characteristics of selected matrices are presented in table I.

C. Evaluation Results

To benchmark a .NET-based implementation we use *BenchmarkDotNet*[11] which allows one to automate benchmarking process for .NET platform. We run each function 100 times, separated warm up runs were added for our implementation. Time is measured in milliseconds. Time to prepare data and initially transfer it to GPU is not included.

Results of performance evaluation are presented in table II. For element-wise addition, our implementation is slightly slower than SuiteSparse:GraphBLAS for small matrices (**wing, luxembourg_osm**) and up to 7 times faster for big matrices (1.5 times in median). At the same time our implementation is 2.5 times slower than CUSP-based. Note that CUSP is based on the highly-optimized Thrust library. For element-wise multiplication comparison with SuiteSparse:GraphBLAS shows almost similar results except matrix **webbase-1M** for which our implementation is slower than SuiteSparse:GraphBLAS.

Comparison between original element-wise addition over primitive types without `AtLeasOne` and generalized version which uses `AtLeastOne` type is also presented in table II. We can see that more complex data types and element-wise operations do not poor performance of matrix-matrix operations because data transfer dominates arithmetic computations for sparse matrix processing, and the proposed abstraction does not increase the memory footprint.

IV. Conclusion and Future Work

We present a work in progress that demonstrates a way to utilize both the power of high-level languages and performance of GPGPUs to implement GraphBLAS-like API. Our preliminary evaluation shows that the proposed way is promising: high-level abstractions do not poor performance, allows one to create compact expressive API, and metaprogramming techniques provide a way to utilize GPUs and achieve reasonable performance.

In the future, first of all, we should extend our library to provide an API which is at least as powerful as GraphBLAS API. Known drawbacks of GraphBLAS design[12] should be analyzed to find a way to fix them.

The next step is evaluation of the solution on real-world cases and comparison with other implementations of GraphBLAS API on different devices and different algorithms.

Finally, we should implement high-level optimizations, like kernels fusion and specialization to achieve high performance.

[10]SuiteSparse matrix collection: https://sparse.tamu.edu/. Access date: 08.01.2023.

[11]*BenchmarkDotNet*: https://benchmarkdotnet.org/. Access date: 08.01.2023.

[12]"What did GraphBLAS Get Wrong?", John Gilbert, UC Santa Barbara, GraphBLAS BoF at HPEC, September 2022

TABLE II
EVALUATION RESULTS FOR ELEMENT-WISE OPERATIONS, TIME IN MS

Matrix	Elemint-wise addition				Elemint-wise multiplication	
	GraphBLAS-sharp		SuiteSparse	CUSP	GraphBLAS-sharp	SuiteSparse
	No `AtLeastOne`	`AtLeastOne`				
wing	$4,3 \pm 0,8$	$4,3 \pm 0,6$	$2,7 \pm 0,9$	$1,5 \pm 0,0$	$3,7 \pm 0,5$	$3,5 \pm 0,4$
luxembourg_osm	$4,9 \pm 0,7$	$4,1 \pm 0,5$	$3,0 \pm 1,1$	$1,2 \pm 0,1$	$3,8 \pm 0,6$	$3,0 \pm 0,6$
amazon0312	$22,3 \pm 1,3$	$22,1 \pm 1,3$	$33,4 \pm 0,8$	$11,0 \pm 1,4$	$18,7 \pm 0,9$	$35,7 \pm 1,4$
amazon-2008	$38,7 \pm 0,8$	$39,0 \pm 1,0$	$55,9 \pm 1,0$	$19,1 \pm 1,4$	$34,5 \pm 1,0$	$58,9 \pm 1,9$
web-Google	$43,4 \pm 0,8$	$43,4 \pm 1,1$	$67,2 \pm 7,5$	$21,3 \pm 1,3$	$39,0 \pm 1,2$	$66,2 \pm 0,4$
webbase-1M	$63,6 \pm 1,1$	$63,7 \pm 1,3$	$86,5 \pm 2,0$	$24,3 \pm 1,3$	$54,5 \pm 0,7$	$37,6 \pm 5,6$
cit-Patents	$26,9 \pm 0,7$	$26,0 \pm 0,7$	$183,4 \pm 5,4$	$10,8 \pm 0,6$	$24,3 \pm 0,7$	$162,2 \pm 1,7$