High-Performance GraphBLAS 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 [1] 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 [2] and machine learning [3].

There are a number of GraphBLAS API implementations, such as SuiteSarse:GraphBLAS [4] and CombBLAS [5], but all of them do not utilize the power of GPGPU, except GraphBLAST [6], 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 [6]. 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] [7], Accelerate[2] [8], AnyDSL[3] [9].

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 [10], [11]. 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

In this work we are focused on making development process easier and safer by using !!!. Automate optimization. Accurate type-level encoding of domain: monoids, semirings.

Monoids and semirings are closed under operations. Thus, in contrast with GraphBLAS API, all operations in semirings and monoids have the following type: $t \to t \to t$ (instead of $t_1 \to t_2 \to t_3$ as proposed in the GraphBLAS specification). It makes our definition less flexible, but allows one to generalize some operations, such as closure of relation. We realize, that in some cases such restrictive constrains are not required. Namely, definition of matrix multiplication does not requires a semiring, it just requires two operations $\oplus$ and $\otimes$ with following types: $\otimes : t_1 \to t_2 \to t_3$, $\oplus : t_3 \to t_3 \to t_3$. But formally, a set with such operations is not a semiring. We think that such case should be investigated separately from semirings, because additional guaranties provided by semirings may be used for code simplification and optimization. For example, it may help to solve a problem with explicit zeros[6] because we should explicitly specify conversion from one semiring to another if required.

[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.

```
1  type RInfinity = R of float | Infinity
2
3  [<Struct>]
4  type MinPlusSemiring =
5     MinPlusSemiring of RInfinity
6  with
7     static member Zero = MinPlusSemiring Infinity
8     static member (+)
9        (MinPlusSemiring x, MinPlusSemiring y) =
10           match x, y with
11           | R x, R y -> System.Math.Min(x,y) |> R
12           | _        -> Infinity
13           |> MinPlusSemiring
14    static member (*)
15       (MinPlusSemiring x, MinPlusSemiring y) =
16           match x, y with
17           | R x, R y -> x + y |> R
18           | _        -> Infinity
19           |> MinPlusSemiring
20    static member op_Implicit (MinPlusSemiring src) =
21       src
```

Listing 1: Example om `Min-Plus` semiring definition

Matrices and vectors are equipped with monoid or semiring. Explicit type conversions. Can be automatically removed in some cases during translation time.

We propose to generate OpenCL c code in running time as a way to solve problems with generics: with strong typing all type information become known and can be used to generate kernels for specific types. Moreover, running time code generation is a way to apply advances optimization techniques, such as partial evaluation (or code specialization), which can improve performance of generated code when part of input parameter of kernel becomes known prior its generation [12].

The example of `Min-Plus` semiring definition is provided in listing 1. Type is defined using discriminated unions (line 1): new set can contains both floats, marked with `R` and a special value `Infinity`. Thus floats are extended with infinity as required for accurate definition of `Min-Plus` semiring. Semiring definition (lines 3–21) includes definition of zero (idenity), operations $\oplus$ (lines 8–13) and $\otimes$ (lines 14–19), !!!!

III. Implementation Details

To evaluate ideas described above we start a development of library named GraphBLAS#[7].

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.

[6]Discussion on zeros removing !!!. Access date: 12.01.2021.

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

TABLE I
RESULTS OF ELEMENT-WISE ADDITION EVALUATION

Matrix			SuiteSparse	Math.NET	GraphBLAS#	
Name	Rows	NNZ			CPU	GPGPU
m1	10	9	8	3	2	1
m2	10	9	8	3	2	1
m3	10	9	8	3	2	1
m4	10	9	8	3	2	1
m5	10	9	8	3	2	1

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 of GraphBLAS API is on very early stage, we cannot evaluate it on well-known linear algebra based algorithms. But in order to !!! Elementwise addition.

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 as a ...
- Math.NET Numerics[8]
- GraphBLAST

Dataset description. Matrices form SuiteSparse collection[9]

For each matrix !!!!. For .NET-based implementations *BenchmarkDotNet*[10] is used. Results of performance evaluation are presented in table I. Time is measured in !!!

We can see, that !!!! results analysis and conclusion.

V. CONCLUSION

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 !!!

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[11] 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 1.

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.

[8]Library which provides numerical computations primitives for .NET: https://numerics.mathdotnet.com/. Access date: 12.01.2021.

[9]!!!

[10]*BenchmarkDotNet* allows one to automate benchmarking process for .NET platform. Project web page: https://benchmarkdotnet.org/. Access date: 12.01.2021.

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

REFERENCES

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

[2] O. Selvitopi, S. Ekanayake, G. Guidi, G. A. Pavlopoulos, A. Azad, and A. Buluç, "Distributed many-to-many protein sequence alignment using sparse matrices," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '20. IEEE Press, 2020.

[3] J. Kepner, M. Kumar, J. Moreira, P. Pattnaik, M. Serrano, and H. Tufo, "Enabling massive deep neural networks with the graphblas," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, 2017, pp. 1–10.

[4] T. A. Davis, "Algorithm 1000: Suitesparse:graphblas: Graph algorithms in the language of sparse linear algebra," *ACM Trans. Math. Softw.*, vol. 45, no. 4, Dec. 2019. [Online]. Available: https://doi.org/10.1145/3322125

[5] A. Buluç and J. R. Gilbert, "The combinatorial blas: Design, implementation, and applications," *Int. J. High Perform. Comput. Appl.*, vol. 25, no. 4, pp. 496–509, Nov. 2011. [Online]. Available: https://doi.org/10.1177/1094342011403516

[6] C. Yang, A. Buluç, and J. D. Owens, "GraphBLAST: A high-performance linear algebra-based graph framework on the GPU," *arXiv preprint*, 2019.

[7] T. Henriksen, N. G. W. Serup, M. Elsman, F. Henglein, and C. E. Oancea, "Futhark: Purely functional gpu-programming with nested parallelism and in-place array updates," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: ACM, 2017, pp. 556–571. [Online]. Available: http://doi.acm.org/10.1145/3062341.3062354

[8] T. L. McDonell, M. M. Chakravarty, G. Keller, and B. Lippmeier, "Optimising purely functional gpu programs," *SIGPLAN Not.*, vol. 48, no. 9, pp. 49–60, Sep. 2013. [Online]. Available: https://doi.org/10.1145/2544174.2500595

[9] R. Leißa, K. Boesche, S. Hack, A. Pérard-Gayot, R. Membarth, P. Slusallek, A. Müller, and B. Schmidt, "Anydsl: A partial evaluation framework for programming high-performance libraries," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, Oct. 2018. [Online]. Available: https://doi.org/10.1145/3276489

[10] T. Kenter, "Invited tutorial: Opencl design flows for intel and xilinx fpgas: Using common design patterns and dealing with vendor-specific differences," in *FSP Workshop 2019; Sixth International Workshop on FPGAs for Software Programmers*. VDE, 2019, pp. 1–8.

[11] K. Shagrithaya, K. Kepa, and P. Athanas, "Enabling development of opencl applications on fpga platforms," in *2013 IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors*, 2013, pp. 26–30.

[12] A. Tyurin, D. Berezun, and S. Grigorev, "Optimizing gpu programs by partial evaluation," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 431–432. [Online]. Available: https://doi.org/10.1145/3332466.3374507