Hardware and Software Co-design for Sparse Linear Algebra Operations Acceleration

Aleksey Tyurin
Saint Petersburg State University
Saint Petersburg, Russia
alekseytyurinspb@gmail.com

Daniil Berezun
Saint Petersburg State University
JetBrains Research
Saint Petersburg, Russia
daniil.berezun@jetbrains.com

Semyon Grigorev
Saint Petersburg State University
JetBrains Research
Saint Petersburg, Russia
s.v.grigoriev@spbu.ru

***Abstract*—In the era of big data, computations are expected to be faster and less power-consuming in order to become more effective and affordable.**

Introduction

Linear algebra is a great instrument for solving a wide variety of problems utilizing matrices and vectors for data representation and analysis with the help of highly optimized routines. And whilst the matrices involved in a vast diversity of modern applications, e.g., recommender systems [1], [2] and graph analysis [3], [4], consist of a large number of elements, the major part of them are zeros. Such a high sparsity incurs both computational and storage inefficiencies, requiring an unnecessarily large storage, occupied by zero elements, and a large number of operations on zeroes, where the result is obviously known beforehand. The traditional approach to address these inefficiencies is to compress the matrix and store only the non-zero elements. Thus, the effect of matrices tending to be sparse in many applications makes the techniques of matrix compressed representation and sparse linear algebra to be the effective way of tackling problems in areas including but not limited to graph analysis [5], computational biology [6] and machine learning [7].

Sparse linear algebra defines primitives for expressing algorithms for the mentioned areas in a uniform way in terms of sparse matrix and vector operations parameterized by a semiring. Such uniform representation allows to tune the whole bunch of expressible algorithms through optimizing the primitives solely. One of the most used primitive is a sparse matrix-sparse matrix multiplication (spMspM) operation. It has finely-tuned implementations for both CPU and GPU, which, however, are proven to be underutilized due to the memory-bound nature of sparse computations induced by compressed representation [8]–[11]. Further, the pipeline of spMspM is patchy, which makes some of the computational units to be idle from time to time as it could be seen in figure 1, while the peak FLOPS is less than 1% of maximum available[1].

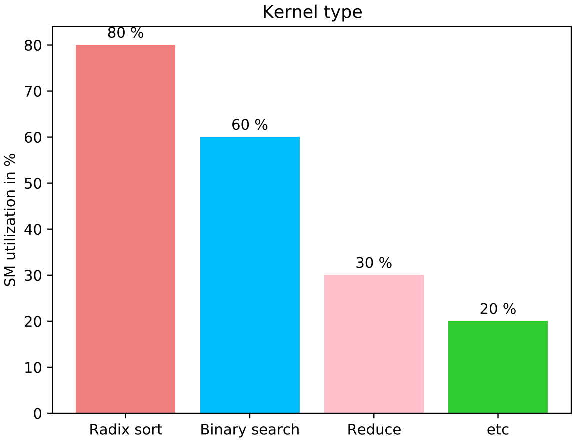

Fig. 1. GPU's SM utilization for spMspM pipeline from cuSPARSE[2]

```
-- A,B,C,D are sparse matrices
-- M is a mask
D<M> = A eWiseAdd B eWiseMult C
```

Listing 1: Sequence of sparse operations example

This makes the typical CPUs and GPUs not well-suited hardware for sparse computations and gives a rise to specialized hardware accelerators, which are primarily concerned with spMspM. However, for a sparse framework to be useful, it should incorporate not only spMspM, but also other sparse operations like in listing 1, where masking, which filters the matrix elements, and element-wise operations (possibly parameterized by a semiring) needed for, e.g., PageRank and bread-first search (BFS) algorithms [12] are presented. And when such operations are chained explicitly or implicitly, via a loop body, certain optimizations could be applied, like the one that eliminates intermediate matrices in sequence from listting 1. Unfortunately, some of such optimizations (e.g., the one mentioned) are only expressible at software level, i.e., in programming language, hence modern spMspM accelerators could be impractical for accelerating the whole program representing a linear algebra based algorithm like PageRank or BFS, due to the lack of a software part and to a too narrow hardware specialization. Thus a co-design of dedicated hardware and software components, i.e., domain-

[1] https://hanlab.mit.edu/projects/sparch/ (Accessed 09.02.2021)

[2] https://developer.nvidia.com/cusparse (Accessed 09.02.2021)

specific processor (DSP) and a corresponding domain-specific language (DSL), could provide a system which is not more effective for spMspM than present hardware accelerators, but appear to be more effective in terms of speed and power consumption for holistic pieces of program, i.e., for chained operations, than current CPUs and GPUs implementations. The ongoing work is devoted to the design of respective DSL, DSP, and an optimizing compiler, and this work in particular gives a brief overview of the filed, discusses the ideas and challenges behind the design.

I. Problem statement

Since sparse linear algebra applications are mostly concerned with graph problems, some of the optimizations are graph-specific [12], [13], e.g., direction optimization. However, in memory-bound applications optimizations that minimize data transfer are essential. For example, *kernel fusion* is a wide addressed optimization that fuses multiple operations into one, avoiding intermediate memory accesses, utilizing, e.g., registers to pass the data between the operations. In the case of sparse linear algebra frameworks kernel fusion is not yet widely implemented, but most often related to fusing chained operations like from listing 1 to avoid intermediate matrices construction and reduce memory accesses with masking [12].

The problem of intermediate data structures is common for functional programming and there have been developed a number of optimization techniques that try to reduce intermediate data structures or computations, namely partial evaluation [14], deforestation [15], supercompilation [16], and distillation [17].

These fusion family optimizations are implementation-dependent, meaning that the optimizer should have an access to the source code, which is impossible when the function is implemented solely in hardware, hence a software part is inevitable in the design of a systems that tries to put together fusion and hardware. Thus, present hardware accelerators are not general enough to implement and accelerate a whole sparse linear algebra-based program, e.g., do not provide arbitrary semiring support, and lack a software part hence leaving out essential optimizations. General purpose devices such as GPUs are hard to perform some optimizations, e.g., fusion requires two GPU kernels to have the same memory access pattern and partial evaluation could induce some thread divergence due to SIMD nature of a GPU. Further, present systems that support fusion are domain-specific[3], or perform the optimization on top of data structures that may not be suitable for sparse operations (mainly for effective compressed representation), e.g., streams and lists [18], [19], while array-based fusion systems does not perform fusion with index arithmetic [20]. Finally, some data needed for optimizations [14] is only available in runtime, thus there should be support for JIT optimizations. We propose the co-design of ??? to address the problems of optimizeability and expressibility. We believe that a functional DSL, where an arbitrary semiring could be concisely and conveniently expressed, powered by a set of optimizers and compilable to some DSP with enough parallelism could be a good starting point in acceleration of sparse linear algebra based programs. Next we discuss some possible hardware architectures and incurred challenges as well as high-level architecture of the proposed solution.

Static vs dynamic data. Specialization requires MIMD.

Representation for data (formats)

AOT vs JIT optimizations. Some data become static in running time.

II. Graph processors

Graph processing could be considered as a distinct computation paradigm and to mitigate the issues of graph processing on general purpose processors there have been emerged a number of graph processors [21]. However, many proposed accelerator models are not generic, in the sense they are optimized for specific classes of graph algorithms and lack scalability. Further, sparse linear algebra makes graph computations more amenable to effective parallelization, ease scalability and even spreads beyond graph computations [6]. Presently, there is no sparse linear algebra-based instruction set processor [10], while other sparse linear algebra accelerators [11], [22], [23] are applicable only to certain operations like spMspM. Notably, in [24] the way of compressed representation of a sparse matrix is identified as a main bottleneck and an approach for hardware acceleration for compressed representation indexing is proposed, which is able to employ existing optimizations and could be integrated in existing software frameworks. Hence, the underlying hardware should better to support the indexing needed by a compressed representation of choice to be effective. However, we believe that to fully leverage fusion the indexing should be transparent, i.e., constructive, in the software, which is not the case in [24].

III. Partial Evaluation and Supercompilation

All of the topics beneath seem to be discussed in problem statement Specialization on static data. For GPU.

Partial cases for different models (array programming, stream fusion, kernels fusion)

Easy for functional languages, hard for imperative which are widely used for HPC.

Supercompiler provides all optimizations, but too hard to implement in general case. We can use small language to design LA library.

IV. Functional language processors

There are a number of works that aim to add a hardware support for functional programming [25]–[27]. Where [25] leverages the parallelization of independent function calls, which do not appear much after supercompilation True or not? More likely False, and also provides an optimized support for lists and `map` operation specifically, which may not be the desired property due to the compressed representation of choice. Further, the general-purpose nature of such processors could hurt sparse operations performance due to non-specialized cache. They also focus on hardware support for

[3] https://www.tensorflow.org/xla (Accessed 09.02.2021)

graph reduction via parallelization, specialized memory, and pipelining, thus even more parallelism could be extracted from compressed representation. However, present functional processors could be too complex to integrate with, e.g., to add hardware support for indexing or another computation that cannot be optimized effectively in software. Instead, we plan to leverage the approach of dataflow representation of functional programs with indirect memory accesses [28], which then could be effectively represented in hardware in a highly parallel way. The approach currently generates application-specific RTL code, but we hope to adapt it for dataflow processor architecture, e.g., transport-triggered-architecture (TTA), in order to have highly scalable and parallel processor with hardware support for a compressed representation, able to run functional programs and easily extensible with custom operations.

V. Dataflow processors

Nothing except TTA :(TTA and other dataflow + MIMD

VI. Research Questions and Roadmap

As a result, we think that solution should includes simultaneous design the following components.

- Functional domain specific language to develop a library of sparse linear algebra operations. Functional programming language is well-suitable for supercompilation and for specialized hardware creation. THe language should be minimized to simplify optimizations and comilation, but expressive enough to implement almost all GraphpBLAS specification.
- Compiler form the functional DSL to the specialized hardware.
- Supercompiler for the functional DSL. Supercompiler should be designed for multistage processing. For example, first stage is a compile-time fusion of functions, and the second stage is a specialization on some data. This allows one to exploit dynamic data as static, and minimize overhead in running time.
- Library of sparse linear algebra operations (GraphBLAS implementation) in the functional DSL. We should to choose such representation of vectors and matrices, and such principles of code design, that both, required software optimizations and efficient hardware design should be possible.
- Special hardware which should be general enough to execute all code written in the functional DSL with utilization of the library of linear algebra operation.

Such the multicomponent solution with nontrivial connections between components requires deep analysis in number of areas. So, at the first stage we plan to focus on the following questions.

1) Is supercompilation suitable for required optimizations? For example, can it remove intermediate matrices creation in chained operations, like presented in listing 1.
2) Can we generate program-specific hardware from the linear algebra operation written in functional programming language such that it will be efficient in comparison with current solutions?
3) Is supercompilation suitable in functional-program-to-hardware workflow? Does supercompiled program compiles to more efficient hardware that original one?
4) Does staged supercompilation allow to minimize specialization time enough to apply specialization in running time?

As a first step we plan to write prototype of some operations like multiplication, element-wise addition, Kronecker product, and masking in minimalistic functional programming language (Haskell-core) and try to apply HOSC [**?**] to it. It allows us to estimate supercomilation effect in terms of expected optimizations like fusion, and time required for specialization. Also, at his step we should establish principles to design the library to maximize supercompilation effect.

At the same time we plan to evaluate functional language to hardware compiler [28] on the developed library to estimate performance of implemented operations and to compare basic and supercompiled versions.

As a result of this step we expect to estimate an applicability of functional programming language and related optimization techniques to crate high-performance linear algebra based solutions. As a result we should to provide a solution for program-specific accelerators creation for sparse linear algebra based subprograms.

As far as the final goal is to create not program-specific hardware accelerator, but a processor unit for high-performance linear algebra based computations, the second stage should be focused on the flowing strongly connected questions.

1) Can generalize ideas from functional languages to hardware compilation to create a specific processor unit for sparse linear algebra?
2) Can the designed language be efficiently compiled to the designed processor unit?

As a result of this stage we hope to cerate a specialized processor unit for sparse linear algebra and a compiler from the functional DSL to it.

The final stage is a creation of SDK to integrate the DSL to application. The SDK should provide tools for staged optimizations, communication with the special processor unit, interoperability with general-purpose programming languages.

References

[1] U. Gupta, C.-J. Wu, X. Wang, M. Naumov, B. Reagen, D. Brooks, B. Cottel, K. Hazelwood, B. Jia, H.-H. S. Lee, A. Malevich, D. Mudigere, M. Smelyanskiy, L. Xiong, and X. Zhang, “The architectural implications of facebook’s dnn-based personalized recommendation,” 2020.

[2] G. Linden, B. Smith, and J. York, “Amazon.com recommendations: Item-to-item collaborative filtering,” *IEEE Internet Computing*, vol. 7, p. 76–80, Jan. 2003.

[3] M. Besta, F. Marending, E. Solomonik, and T. Hoefler, “Slimsell: A vectorizable graph representation for breadth-first search,” in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 32–41, 2017.

[4] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer Networks and ISDN Systems*, vol. 30, no. 1, pp. 107 – 117, 1998. Proceedings of the Seventh International World Wide Web Conference.

[5] J. Kepner and J. Gilbert, *Graph Algorithms in the Language of Linear Algebra*. USA: Society for Industrial and Applied Mathematics, 2011.

[6] O. Selvitopi, S. Ekanayake, G. Guidi, G. Pavlopoulos, A. Azad, and A. Buluc, "Distributed many-to-many protein sequence alignment using sparse matrices," 2020.

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

[8] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, Dec. 2011.

[9] J. Leskovec and R. Sosic, "Snap: A general purpose network analysis and graph mining library," 2016.

[10] W. S. Song, V. Gleyzer, A. Lomakin, and J. Kepner, "Novel graph processor architecture, prototype system, and results," *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, Sep 2016.

[11] Z. Zhang, H. Wang, S. Han, and W. J. Dally, "Sparch: Efficient architecture for sparse matrix multiplication," in *26th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[12] C. Yang, A. Buluc, and J. D. Owens, "Graphblast: A high-performance linear algebra-based graph framework on the gpu," 2020.

[13] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe, "Graphit: A high-performance graph dsl," *Proc. ACM Program. Lang.*, vol. 2, Oct. 2018.

[14] N. D. Jones, C. K. Gomard, and P. Sestoft, *Partial Evaluation and Automatic Program Generation*. USA: Prentice-Hall, Inc., 1993.

[15] P. Wadler, "Deforestation: transforming programs to eliminate trees," *Theoretical Computer Science*, vol. 73, no. 2, pp. 231–248, 1990.

[16] M. Sørensen, R. Glück, and N. Jones, "A positive supercompiler," *Journal of Functional Programming*, vol. 6, pp. 811 – 838, 11 1996.

[17] G. Hamilton, "Extracting the essence of distillation," pp. 151–164, 06 2009.

[18] O. Kiselyov, A. Biboudis, N. Palladinos, and Y. Smaragdakis, "Stream fusion, to completeness," *SIGPLAN Not.*, vol. 52, p. 285–299, Jan. 2017.

[19] D. Coutts, R. Leshchinskiy, and D. Stewart, "Stream fusion. from lists to streams to nothing at all," vol. 42, pp. 315–326, 09 2007.

[20] 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," *SIGPLAN Not.*, vol. 52, p. 556–571, June 2017.

[21] Y. Horawalavithana, "On the design of an efficient hardware accelerator for large scale graph analytics," 2016.

[22] M. Soltaniyeh, R. P. Martin, and S. Nagarakatte, "Synergistic cpu-fpga acceleration of sparse linear algebra," 2020.

[23] S. Pal, J. Beaumont, D. Park, A. Amarnath, S. Feng, C. Chakrabarti, H. Kim, D. Blaauw, T. Mudge, and R. Dreslinski, "Outerspace: An outer product based sparse matrix multiplication accelerator," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 724–736, 2018.

[24] K. Kanellopoulos, N. Vijaykumar, C. Giannoula, R. Azizi, S. Koppula, N. M. Ghiasi, T. Shahroodi, J. G. Luna, and O. Mutlu, "Smash: Co-designing software compression and hardware-accelerated indexing for efficient sparse matrix operations," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, (New York, NY, USA), p. 600–614, Association for Computing Machinery, 2019.

[25] R. Coelho, F. Tanus, A. Moreira, and G. Nazar, "Acqua: A parallel accelerator architecture for pure functional programs," in *2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp. 346–351, 2020.

[26] M. Naylor and C. Runciman, "The reduceron: Widening the von neumann bottleneck for graph reduction using an fpga," in *Implementation and Application of Functional Languages* (O. Chitil, Z. Horváth, and V. Zsók, eds.), (Berlin, Heidelberg), pp. 129–146, Springer Berlin Heidelberg, 2008.

[27] A. Boeijink, P. K. F. Hölzenspies, and J. Kuper, "Introducing the pilgrim: A processor for executing lazy functional languages," in *Implementation and Application of Functional Languages* (J. Hage and M. T. Morazán, eds.), (Berlin, Heidelberg), pp. 54–71, Springer Berlin Heidelberg, 2011.

[28] R. Townsend, "Compiling irregular software to specialized hardware," 2019.