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

Anonymous Author(s)

1 Scope of problem

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 [8, 17] and graph analysis [1, 3], 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 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 [12], computational biology [20], and machine learning [13].

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 tunning 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 [7, 16, 21, 25]. Further, the pipeline of spMspM is patchy, which makes some of the computational units to be idle from time to time, while the peak FLOPS is less than 1% of maximum available[1].

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 [24], are presented. Such operations are assembled in *GraphBLAS* [4] specification and have found their usage as building blocks for algorithms even far beyond graph processing [12, 13, 20].

```
-- 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

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 listing 1. Unfortunately, some of such optimizations (e.g., the one mentioned) are only expressible at a software level, i.e., in a programming language, hence modern spMspM accelerators could be impractical for accelerating the whole program representing a sparse linear algebra-based algorithm like PageRank or BFS, due to the lack of a software part and to a too narrow hardware specialization.

For example, in memory-bound applications, like sparse linear algebra one, optimizations that minimize data transfer are essential. Namely, *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. Fusion is also wide addressed in functional programming by techniques like supercompilation [22] and serves to remove intermediate computations induced by the paradigm. In the case of sparse linear algebra frameworks, it 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 [24]. 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 system that tries to put together fusion and hardware. Present systems that support fusion are either domain-specific[2], 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 [6, 14], while array-based fusion systems do not perform fusion with index arithmetic [9].

We propose a co-design of dedicated hardware and software components, i.e., domain-specific processor (DSP) and a corresponding domain-specific language (DSL), to provide a system that is not more efficient for spMspM than present hardware accelerators, but appear to be more efficient in

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

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

terms of speed and power consumption for holistic pieces of program, i.e., for chained operations, than current CPUs and GPUs implementations. The hardware should natively support the compressed representation of choice, while the latter should provide enough capabilities for successful fusion. We hope to exploit a functional programming language as a DSL, since they have a support for such optimizations [15].

2 Solution

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 towards the problem. Next, we briefly elaborate on the software and hardware parts of the proposed co-design.

Software. We intend to implement a DSL for a reasonable subset of GraphBLAS specification in a small functional language supported by a supercompiler [15] with the focus on a more amenable to fusion quadtree compressed representation, with some custom fusion rules added if needed. If some functions are not fused well it is reasonable to implement them in hardware. Another feature we want to have is to generate code in runtime for some data which is fixed, e.g., between different loop iterations. It is referred to as partial evaluation [10].

Hardware. Any program representing a sparse linear algebra computation written in the DSL should be compiled to the DSP. There are a number of state-of-the-art processors [2, 5, 19], developed to add hardware support for functional languages where the main computation is tree reduction. They support tempting features like parallel execution of independent function calls, pipelining, extended support for lists and `map` function. However, they seem to be too complex to integrate with, e.g., to add hardware support for indexing or another computation, which cannot be optimized at software level. Instead, we plan to opt for a more flexible solution and want to leverage the approach of dataflow representation of functional programs with indirect memory accesses [23], 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 a highly scalable and parallel processor with hardware support for a compressed representation, able to run functional programs and easily extensible with custom operations. Finally, the processor should exploit MIMD parallelism, since, e.g., a huge bunch of `case` statements possibly induced by supercompilation makes SIMD ineffective due to execution paths divergence.

Mainly the following challenges would be addressed in our hardware design

1. Is it possible to add effective hardware support for quadtree compressed representation? It seems more profitable due to better fusion support and divide-and-conquer nature, hence parallelism could be explored by the compiler [23]. The idea of hardware support for indexing originates from [11], but instead of making indexing as hardware intrinsics, we want to keep the indexing fully transparent for the supercompiler and support the indexing with, e.g., cache mechanism that would prefetch sub-trees.
2. Is it possible to generalize the dataflow approach of compiling a functional program to RTL description [23] to compiling a functional program to a dataflow processor, e.g., to TTA?

As a full-fledged prototype, we want to use this hardware as a co-processor, i.e., the prototype should be a platform where FPGA and CPU share the memory, supported by SDK for interoperability with general-purpose programming languages.

3 Evaluation

The intend of the presented optimizations is to enhance the performance in terms of, e.g., clock cycles in case of simulation, rather than to reduce power consumption, which, nevertheless, is an important aspect especially when compared to GPUs. So we will focus on execution time in our evaluation.

Our first experiments will be aimed at evaluating HLS of fused and non-fused programs implemented in our functional language of choice via simulation. This would tell us whether the whole hypothesis of fusion is true and, in particular, the usefulness of a supercompiler in this scenario. Then we will try to integrate hardware support for quadtrees into these implementations. Once we have our dataflow processor ready, we plan to perform an evaluation of GraphBLAS algorithms implemented in our DSL and state-of-the-art implementations from [18, 24] using a CPU/FPGA board, while potentially targeting ASIC board in the future if successful.

References

[1] M. Besta, F. Marending, E. Solomonik, and T. Hoefler. 2017. SlimSell: A Vectorizable Graph Representation for Breadth-First Search. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 32–41. https://doi.org/10.1109/IPDPS.2017.93

[2] Arjan Boeijink, Philip K. F. Hölzenspies, and Jan Kuper. 2011. Introducing the PilGRIM: A Processor for Executing Lazy Functional Languages. In *Implementation and Application of Functional Languages*, Jurriaan Hage and Marco T. Morazán (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 54–71.

[3] Sergey Brin and Lawrence Page. 1998. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems* 30, 1 (1998), 107 – 117. https://doi.org/10.1016/S0169-7552(98)00110-X Proceedings of the Seventh International World Wide Web Conference.

[4] Aydin Buluc, Timothy Mattson, Scott McMillan, Jose Moreira, and Carl Yang. 2017. The GraphBLAS C API Specification. *GraphBLAS. org, Tech. Rep.* (2017).

[5] R. Coelho, F. Tanus, A. Moreira, and G. Nazar. 2020. ACQuA: A Parallel Accelerator Architecture for Pure Functional Programs. In *2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 346–351. https://doi.org/10.1109/ISVLSI49217.2020.00070

[6] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. 2007. Stream Fusion. From Lists to Streams to Nothing at All. *Sigplan Notices - SIGPLAN* 42, 315–326.

[7] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (Dec. 2011), 25 pages. https://doi.org/10.1145/2049662.2049663

[8] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim Hazelwood, Bill Jia, Hsien-Hsin S. Lee, Andrey Malevich, Dheevatsa Mudigere, Mikhail Smelyanskiy, Liang Xiong, and Xuan Zhang. 2020. The Architectural Implications of Facebook's DNN-based Personalized Recommendation. arXiv:cs.DC/1906.03109

[9] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: Purely Functional GPU-Programming with Nested Parallelism and in-Place Array Updates. *SIGPLAN Not.* 52, 6 (June 2017), 556–571. https://doi.org/10.1145/3140587.3062354

[10] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, Inc., USA.

[11] Konstantinos Kanellopoulos, Nandita Vijaykumar, Christina Giannoula, Roknoddin Azizi, Skanda Koppula, Nika Mansouri Ghiasi, Taha Shahroodi, Juan Gomez Luna, and Onur Mutlu. 2019. 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* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 600–614. https://doi.org/10.1145/3352460.3358286

[12] Jeremy Kepner and John Gilbert. 2011. *Graph Algorithms in the Language of Linear Algebra.* Society for Industrial and Applied Mathematics, USA.

[13] Jeremy Kepner, Manoj Kumar, Jose Moreira, Pratap Pattnaik, Mauricio Serrano, and Henry Tufo. 2017. Enabling massive deep neural networks with the GraphBLAS. *2017 IEEE High Performance Extreme Computing Conference (HPEC)* (Sep 2017). https://doi.org/10.1109/hpec.2017.8091098

[14] Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. 2017. Stream Fusion, to Completeness. *SIGPLAN Not.* 52, 1 (Jan. 2017), 285–299. https://doi.org/10.1145/3093333.3009880

[15] Ilya Klyuchnikov. 2010. Supercompiler HOSC 1.5: homeomorphic embedding and generalization in a higher-order setting. (01 2010).

[16] Jure Leskovec and Rok Sosic. 2016. SNAP: A General Purpose Network Analysis and Graph Mining Library. arXiv:cs.SI/1606.07550

[17] Greg Linden, Brent Smith, and Jeremy York. 2003. Amazon.Com Recommendations: Item-to-Item Collaborative Filtering. *IEEE Internet Computing* 7, 1 (Jan. 2003), 76–80. https://doi.org/10.1109/MIC.2003.1167344

[18] T. Mattson, T. A. Davis, M. Kumar, A. Buluc, S. McMillan, J. Moreira, and C. Yang. 2019. LAGraph: A Community Effort to Collect Graph Algorithms Built on Top of the GraphBLAS. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 276–284. https://doi.org/10.1109/IPDPSW.2019.00053

[19] Matthew Naylor and Colin Runciman. 2008. The Reduceron: Widening the von Neumann Bottleneck for Graph Reduction Using an FPGA. In *Implementation and Application of Functional Languages*, Olaf Chitil, Zoltán Horváth, and Viktória Zsók (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 129–146.

[20] Oguz Selvitopi, Saliya Ekanayake, Giulia Guidi, Georgios Pavlopoulos, Ariful Azad, and Aydin Buluc. 2020. Distributed Many-to-Many Protein Sequence Alignment using Sparse Matrices. arXiv:cs.DC/2009.14467

[21] William S. Song, Vitaliy Gleyzer, Alexei Lomakin, and Jeremy Kepner. 2016. Novel graph processor architecture, prototype system, and results. *2016 IEEE High Performance Extreme Computing Conference (HPEC)* (Sep 2016). https://doi.org/10.1109/hpec.2016.7761635

[22] Morten Sørensen, R. Glück, and Neil Jones. 1996. A positive supercompiler. *Journal of Functional Programming* 6 (11 1996), 811 – 838. https://doi.org/10.1017/S0956796800002008

[23] Richard Townsend. 2019. Compiling Irregular Software to Specialized Hardware.

[24] Carl Yang, Aydin Buluc, and John D. Owens. 2020. GraphBLAST: A High-Performance Linear Algebra-based Graph Framework on the GPU. arXiv:cs.DC/1908.01407

[25] Zhekai Zhang, Hanrui Wang, Song Han, and William J. Dally. 2020. SpArch: Efficient Architecture for Sparse Matrix Multiplication. In *26th IEEE International Symposium on High Performance Computer Architecture (HPCA)*.