Hardware and software co-design for sparse linear algebra operations acceleration

Anonymous Author(s)

1 Scope of problem

Matrices and the corresponding operations are well-established components for data representation and analysis since it allows to tackle a problem with the help of a myriad of matrix accelerators. However, the data is inherently sparse in many modern applications, e.g., social graphs have far less edges than vertices [13]. 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, and then operate only on the non-zero values. 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 [10], computational biology [16] and machine learning [11].

Sparse linear algebra defines building blocks for expressing algorithms for mentioned areas in a uniform way in terms of sparse matrix and vector operations over some semiring. Once such blocks are implemented in software (or hardware) according to, e.g., *GraphBLAS* [1] standard, a plenty of expressible algorithms could be tuned and optimized at once by optimizing and tuning the building blocks. One of the key primitives is a sparse matrix-sparse matrix multiplication (spMspM) operation. It has high-performance implementations both for CPUs and GPUs. However, typical CPUs and GPUs are proven to be underutilized [5, 13, 18, 23], i.e., their computing units do not achieve peak performance, for tasks that involve sparsity, due to being too general by design and suffering from the irregularity of memory accesses incurred by sparsity. 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 1, while the peak FLOPS is less than 1% of maximum available.

To address this issue several specialized hardware accelerators have been designed for spMspM [15, 23]. However, for a sparse framework to be useful, it should incorporate not only spMspM, but also other sparse operations parameterized by a semiring, e.g., masking, which filters the matrix elements or element-wise operations needed for, e.g., PageRank and bread-first search (BFS) algorithms [21]. And when such operations are chained explicitly or implicitly, via a loop body,

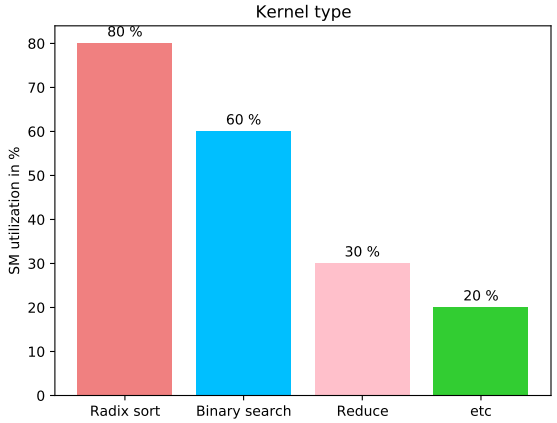

Figure 1. GPU's SM utilization for spMspM pipeline from cuSPARSE[1]

certain optimizations could be applied. However, some of such optimizations are only expressible at software level, i.e., in programming language, thus 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-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 discusses the ideas and challenges behind the design.

2 Optimizations

Since sparse linear algebra applications are mostly concerned with graph problems, some of the optimizations are graph-specific [21, 22], e.g., direction optimization. In this work, we are mostly interested in graph-agnostic optimizations, where the most considered [21, 22] one is *fusion* [2], which in essence stands for deleting intermediate data structures merging several operations into one along the way.

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

The problem of intermediate data structures is common for functional programming and there have been developed a number of optimization techniques that try reduce intermediate data structures or computations, namely partial evaluation [9], deforestation [20], supercompilation [19], and distillation [6]. All of these optimization will be reffered as fusion futher. The purpose of such optimizations is to remove intermediate structures, e.g., lists, in scenarios like

```
f :: Int → Int
f n = sum [ k * m | k ← [1..n], m ← [1..k] ]
```

or optimize recursive calls like

```
nrev :: [a] → [a]
nrev [] = []
nrev x:xs = app (nrev xs) [x]

app :: [a] → [a]
app [] xs = xs
app (x:xs) ys = x : app xs ys

-- optimizing quadratic nrev to
arev xs = arev' xs []
arev' [] ys = ys
arev' (x:xs) ys = arev' xs (x : ys)
```

For the case of sparse linear algebra such optimizations are expected to fuse operations removing intermediate matrices, e.g. eliminating `A ewiseAdd B` matrix in

```
D = A ewiseAdd B ewiseMult C
```

or fusing a mask with sparse matrix-vector multiplication reducing the number of memory accesses [21].

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. However, arbitrary functions could be fused using rules like

```
map f (map g ls) ≡ map (f ∘ g) ls
```

where the ∘ is function composition, but this would require a specialized hardware unit for each function which is impractical. Further, such optimizers produce a code that essentially represents a dataflow through constructors / destructors and corresponding semiring operations, which seems to fit dataflow processor architecture. Thus 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 dataflow DSP with enough parallelism could be a good starting point in acceleration of sparse linear algebra based programs. However there are several challenges present, some of which are incurred by the implementation-dependent nature of the optimizers.

3 Challenges

Firstly, not every operation could be fused, e.g., sorting, which is a noticeable part of sparse pipeline 1 or union / intersection, making it reasonable to implement such operations solely in hardware. Every such operation should be identified and implemented as a processing element inside the processor under design. However, the need for such operations depends on the particular compressed representation of a matrix, which strongly affects the pattern of memory accesses and henceforth the dataflow and the overall performance, e.g. compressed-row format requires sorting while quad-tree [17] representation does not.

Secondly, the matrix compression format affects the amenability for optimizations. There are many techniques for list and stream fusion [3, 12], but such data structures are not effective for irregular indexing accesses incurred by a particular compressed representation, e.g., compressed-row format. Conversely, the representation that does not incur many such accesses could be fused successfully with the known techniques but cannot be processed in parallel so well. The state-of-the-art fusion systems on top of arrays, which are effective for random accesses, presently are not smart enough to fuse something with index arithmetic [7]. Once again quad-tree representation seems to be amenable for fusion and parallelization.

Thirdly, to achieve the full benefit from fusion, the key primitives should be implemented with fusion in mind, which is not trivial and again depends on the compressed representation of choice.

Finally, the main challenge is to choose a processor architecture which would allow it to execute the fused programs most effectively. For example, the result of some of the optimizations perform poor on SIMD devices like GPUs, due to, e.g., thread divergence. Hence, the processor architecture seems to should be MIMD one and should be general enough to support arbitrary semirings and specialized enough to be effectively utilized. One prominent architecture is Transport Triggered Architecture (TTA) [8], which is simple and inherently parallel [14].

4 Evaluation

At the moment we focus on exploiting a supercompiler distillator? to perform a fusion for a subset of GraphBLAS operations, using a quad-tree representation and aim to target TTA architecture. If the approach is successful it is supposed to be compared with state-of-the-art CPUs and GPUs implementations of GraphBLAS standard [4, 21] first in simulation and then on an FPGA board. Otherwise we will explore other compressed matrix representations and / or extend the optimizers if possible.

References

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

[2] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. 2007. Stream Fusion: From Lists to Streams to Nothing at All. In *Proceedings of*

the 12th ACM SIGPLAN International Conference on Functional Programming (Freiburg, Germany) *(ICFP '07)*. Association for Computing Machinery, New York, NY, USA, 315–326. https://doi.org/10.1145/1291151.1291199

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

[4] T. A. Davis. 2018. Graph algorithms via SuiteSparse: GraphBLAS: triangle counting and K-truss. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. 1–6. https://doi.org/10.1109/HPEC.2018.8547538

[5] 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

[6] Geoff Hamilton. 2009. Extracting the Essence of Distillation. 151–164. https://doi.org/10.1007/978-3-642-11486-1_13

[7] 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

[8] Pekka Jääskeläinen, Timo Viitanen, Jarmo Takala, and Heikki Berg. 2017. *HW/SW Co-design Toolset for Customization of Exposed Datapath Processors*. Springer International Publishing, 147–164. https://doi.org/10.1007/978-3-319-49679-5_8

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

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

[11] 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

[12] 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

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

[14] Jon Mountjoy and Marcel Beemster. 1994. *Functional languages and very fine grained parallelism: Initial results*. Technical Report.

[15] S. Pal, J. Beaumont, D. Park, A. Amarnath, S. Feng, C. Chakrabarti, H. Kim, D. Blaauw, T. Mudge, and R. Dreslinski. 2018. OuterSPACE: An Outer Product Based Sparse Matrix Multiplication Accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 724–736. https://doi.org/10.1109/HPCA.2018.00067

[16] 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

[17] I. Simecek. 2009. Sparse Matrix Computations Using the Quadtree Storage Format. In *2009 11th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. 168–173. https://doi.org/10.1109/SYNASC.2009.55

[18] 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

[19] 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

[20] Philip Wadler. 1990. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science* 73, 2 (1990), 231–248. https://doi.org/10.1016/0304-3975(90)90147-A

[21] 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

[22] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. GraphIt: A High-Performance Graph DSL. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 121 (Oct. 2018), 30 pages. https://doi.org/10.1145/3276491

[23] 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)*.