Distilling Sparse Linear Algebra

ALEKSEY TYURIN, Saint Petersburg State University, Russia

1 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. But in reality matrices in many applications are often sparse, incurring 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. It 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 [13], computational biology [18] and machine learning [14].

GraphBLAS [3] standard defines sparse linear algebra building blocks useful to express algorithms for already mentioned areas in a uniform way in terms of sparse matrix and vector operations over some semiring. These include matrix/vector multiplication, element-wise operations (e-wise for short), kronecker product, masking, i.e. taking a subset of elements that satisfies the mask or its complement, and are sufficient to express a lot of algorithms, e.g. *PageRank*, *Breadth-First-Search*, *Sparse Deep Neural Network* [6].

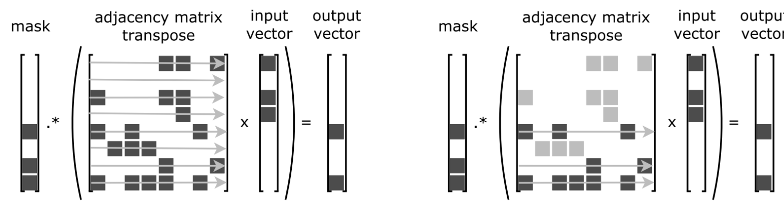
However sparse computations appear to have a low arithmetic-to-memory operations intensity, meaning that the main bottleneck of sparse-algorithms is the sparse representation itself that induces pointer-chasing. Thus, a number of optimizations have been identified [24], whose aim is to reduce the intensity of memory accesses and the one considered in this work is *fusion*. *Fusion* simply stands for removal of intermediate data structures, namely those that are first constructed and then deconstructed. There are two types of fusion that we are interested in.

Mask fusion. Ahead-of-time masking could reduce the number of memory accesses in case of, e.g., matrix-vector multiplication by taking only the elements of interest. In order to achieve such a behavior, a mask should be fused (i.e. transformed into a single operation) with the corresponding operation, for the operation to perform computations only for the elements in the mask. The effect of masking in case of sparse matrix-dense vector multiplication could be seen in figure 1. Ahead-of-time masking reduces the number of memory accesses from 8 to 3.

Kernel fusion. Kernel fusion is responsible for fusing arbitrary operations. In the case of loop-based programming fusion simply stands for joining several loops into one to increase memory locality and reduce the number of required iterations. It is a crucial technique in dense applications and is usually followed by a stage of *polyhedral analysis*. This is extensively exploited in frameworks like TensorFlow and its XLA compiler [2]. A motivating example for general fusion could be seen in listing 1[1], which is a snippet (simplified for demonstration) from Luby's maximal independent set algorithm implementation [22]. As could be seen it is a series of e-wise matrix additions: two consecutive element-wise operations could be fused into one, so `new_members` matrix creation and further iterations are reduced.

Some general-purpose solutions exists that support fusion, e.g., [10] which are based on map/reduce semantics. But in order to support sparse operations they should be able to fuse across index arithmetic, which is not the case. Also at the moment neither [5] nor [24] have adopted the fusion in

[1]The original excerpt is in C++, it is rewritten to ease the demonstration. Call-by-value is assumed.



(a) SpMV not fused with mask (20 loads and stores)

(b) SpMV fused with mask (10 loads and stores)

Fig. 1. Mask fusion example [24]

```
...
-- select node if its probability is > than all its active neighbors

-- @gt is '>' operation, @lor is logical or
let new_members = eWiseAdd gt prob neighbor_max
    iset' = eWiseAdd lor iset new_members in

...
```

Listing 1. Excerpt from Luby's maximal independent set algorithm implementation

their implementations. In this work we propose an approach to support fusion for such applications and outline the overall solution design.

2 SOLUTION

The problem of intermediate data structures is natural for functional programming and a number of approaches for fusion has been designed, namely *partial evaluation, deforestation, supercompilation, distillation* [9, 12, 21, 23]. In this work we will focus on *distillation* since it is able to produce a superlinear improvement for the program being optimized [9].

For succesful fusion the compressed representation should be fuseable, so it should avoid indexing and be natural to functional paradigm. A quad-tree representation [19] looks promising in this case. The implementation of this compressed representation as an algebraic data type could be seen in listing 2, it recursively splits a matrix into four submatrices. Turning back to the successive e-wise matrix additions example, distillation successfully fuses them into one operation where each matrix is iterated only once as also could be seen in listing 2.

If we now define the notion of **intermediate data structure** as the number of times a constructor term within `case` context is encountered during the reduction of the top-level term, i.e. the number of times something is deconstructed with pattern-matching, we could see that the fusion also produces a more effective program, as it could be seen in the table 1, where x/y are reductions, the number of steps to reduce the term to its normal form, and the number of intermediate structures respectively, the numbers at top are the orders of input matrices. Each example firstly was evaluated using the interpreter that counts the number of reductions and intermediate structures. Then it was distilled[2] and evaluated again. Matrices/masks have been taken from [7], converted to q-tree representation and embedded instead of free variables into the corresponding functions. The full list of benchmarks could be found here [1].

It could be noted that `case` $c\ e_0 \ldots e_n$, where c is a constructor, essentially performs a memory read, so the optimization reduces the number of eventual memory reads and corresponding writes

[2]The distiller from [9] was used

```
-- @QNone stands for submatrix which elements are zeroes
data QTree a = QNone
             | QVal a
             | QNode (QTree a) (QTree a) (QTree a) (QTree a)
main = ...
        let new_members = eWiseAdd gt prob neighbor_max
            iset' = eWiseAdd lor iset new_members in
       ...
--gets fused into
main = ...

    let iset' = case iset of
                ... -> case neighbor_max of
                    ... -> case prob of ...
-- @new_members has been eliminated
```

Listing 2. Fusion by means of distillation

Function	Description	# of non-zeroes		
		10^1	10^2	10^3
E-wise successive additions	Original Distilled	107 / 22 44 / 14	11293 / 1852 6129 / 1433	139851 / 20351 89215 / 15061
Kronecker with masking	Original Distilled	213 / 45 108 / 25	535125 / 92470 367868 / 67110	6968317 / 1220816 3974610 / 867137

Table 1. Distillation results

as well. Another practical example is masking of a kronecker product, since kronecker product performs more operations than matrix-vector multiplication, here it is a more representative example that shows the benefits of masking-fusion. The benefit of optimization is up to **2x** in terms of reductions and up to **1.3x** in terms of intermediate structures, hence it could be stated that distillation is applicable to optimize sparse computations and could be able to speed up practical algorithms like Luby's maximal set. The future work and overall idea behind this is described in the next section.

3 FUTURE WORK

The obvious disadvantage of this approach is that it requires a special domain-specific language amenable to distillation, so it could hardly be integrated into existing implementations like [5, 24]. However, typical CPUs and GPUs are proven to be underutilized [7, 15, 20, 25], i.e., their computing units do not achieve peak performance, suffering from the irregularity of memory accesses incurred by sparsity, so a possible direction could be to design a domain-specific co-processor that is able to execute this distillation-amenable language. Such an approach has found a successful application in image processing [16, 17], programmable networks [11] and machine learning [2, 4].

Notably, in [8] a framework is proposed that is capable of transforming arbitrary Haskell programs into hardware description. It provides datatype-specific memory spaces and divide-and-conquer optimizations (since q-tree representation is divide-and-conquer by its natures, it is a good fit). Each `case` $c\ e_0 \ldots e_n$ expression is generated as an explicit memory read of c and hence distillation is also optimizing the hardware in a sense. The resulting hardware is highly-parallel and pipelined, so it could be a good counterpart to modern CPUs and GPUs.

REFERENCES

[1] [n.d.]. Distillation benchmarks. https://github.com/YaccConstructor/Distiller. Accessed: 2021-07-01.

[2] [n.d.]. XLA: Optimizing Compiler for Machine Learning. https://www.tensorflow.org/xla?hl=en. Accessed: 2020-12-28.

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

[4] S. Cass. 2019. Taking AI to the edge: Google's TPU now comes in a maker-friendly package. *IEEE Spectrum* 56, 5 (2019), 16–17. https://doi.org/10.1109/MSPEC.2019.8701189

[5] 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

[6] Timothy A. Davis, Mohsen Aznaveh, and Scott Kolodziej. 2019. Write Quick, Run Fast: Sparse Deep Neural Network in 20 Minutes of Development Time via SuiteSparse:GraphBLAS. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–6. https://doi.org/10.1109/HPEC.2019.8916550

[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] S. Edwards, Martha A. Kim, Richard Townsend, Kuangya Zhai, and L. Lairmore. 2019. FHW Project : High-Level Hardware Synthesis from Haskell Programs.

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

[10] 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

[11] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. Netchain: Scale-free sub-rtt coordination. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 35–49.

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

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

[14] 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

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

[16] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. *SIGPLAN Not.* 48, 6 (June 2013), 519–530. https://doi.org/10.1145/2499370.2462176

[17] Jason Redgrave, Albert Meixner, Nathan Goulding-Hotta, Artem Vasilyev, and Ofer Shacham. 2018. Pixel Visual Core: Google's Fully Programmable Image Vision and AI Processor For Mobile Devices. In *Proc. IEEE Hot Chips Symp.(HCS)*. 1–18.

[18] 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:2009.14467 [cs.DC]

[19] 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

[20] 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

[21] 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

[22] USA Timothy A. Davis, Texas A&M University. [n.d.]. Algorithm 9xx: SuiteSparse:GraphBLAS: graph algorithms in the language of sparse linear algebra. https://people.engr.tamu.edu/davis/publications_files/toms_graphblas.pdf. Accessed: 2021-06-06.

[23] 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

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

[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)*.