Distilled Sparse Linear Algebra in Hardware

Aleksey Tyurin[1,3], Ekaterina Vinnik[1,3], Daniil Berezun[1,2,3], and Semyon Grigorev[1,2,3]

[1]Saint Petersburg State University, Saint Petersburg, Russia
[2]JetBrains Research, Saint Petersburg, Russia
[3]alekseytyurinspb@gmail.com, catherine.vinnik@gmail.com, d.berezun@2009.spbu.ru, s.v.grigoriev@spbu.ru

***Abstract*—Sparse linear algebra is widely used in graph processing, machine learning, and computational biology. And whilst in dense applications fusion optimization is extensively used to reduce the number of kernel calls and memory accesses it is hard to support in case of sparse applications due to the considerably irregular nature of the latter. Fusion is also a well-studied optimization in the world of functional programming where it is often needed to remove intermediate data structures to achieve near imperative languages performance. We propose the usage of functional compressed representation for sparse data and a functional language to express sparse linear algebra algorithms to be able to utilize distillation: the technique which provides fusion and partial evaluation, enhances asymptotic time and guarantees a specific structure of the output programs. Next, we suggest using hardware code generation to convert our optimized programs into highly parallel and pipelined hardware to increase performance. The work presents some intermediate evaluation of this approach.**

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 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 an effective way of tackling problems in areas including but not limited to graph analysis [1], computational biology [2] and machine learning [3].

GraphBLAS [4] 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, for instance, 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, etc., and are sufficient to express a lot of algorithms, e.g. *PageRank*, *Breadth-First-Search*, *Sparse Deep Neural Network* [5].

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 and presents irregularity of memory accesses. Thus, a number of optimizations have been identified [6], whose aim is to reduce the intensity of memory accesses and the one considered in this work is *fusion*. *Fusion* simply stands for gluing several functions into one to remove intermediate data structures, namely those that are first constructed and then deconstructed. There are two types of fusion that we are interested in.

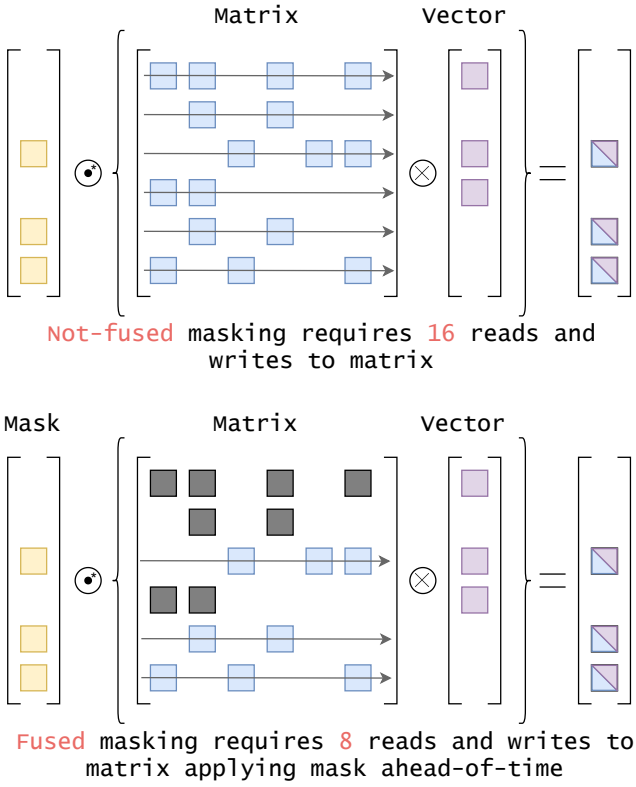

Fig. 1: Mask fusion

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 the case of sparse matrix-dense vector multiplication could be seen in figure 1. Ahead-of-time masking reduces the number of memory accesses from 16 to 8.

Kernel fusion: it is responsible for fusing arbitrary operations to explicitly remove intermediate data structures as could

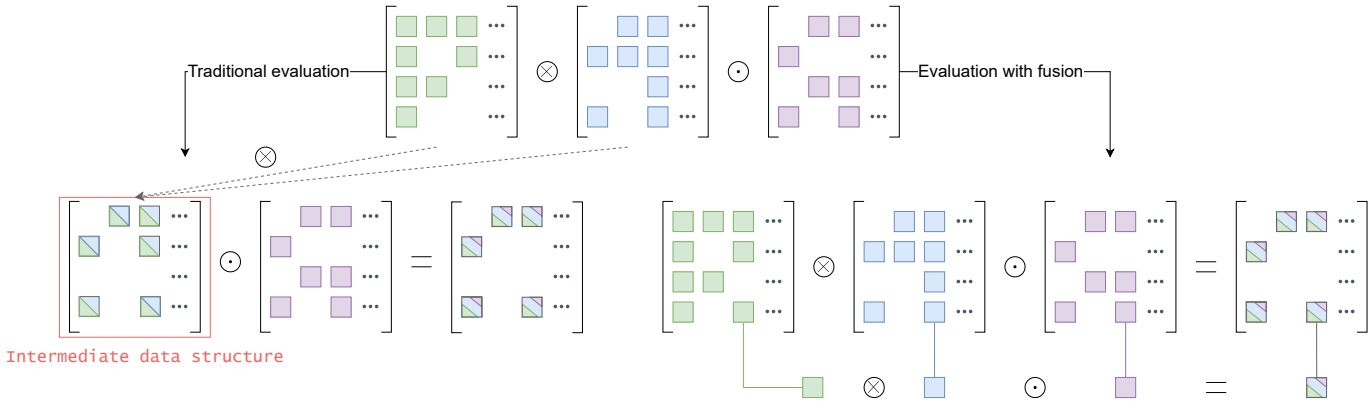

Fig. 2: Kernel fusion

be seen from figure 2. In a traditional way of evaluation having consecutive operations leads to the creation of intermediate data structures, which first should be constructed and then deconstructed to create another data structure, fusion is in charge of removing them.

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 [7], where fusion is used to merge several GPU kernels into one to reduce both memory transfer and kernel start overheads.

Some general-purpose solutions exist that support fusion, e.g., [8] 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 [9] nor [6] have adopted the fusion in their implementations. In this work, we propose an approach to support fusion for such applications and outline the overall solution design as well as present some intermediate evaluations of the approach.

I. Solution

The problem of intermediate data structures is natural for functional programming and a number of approaches for fusion have been designed, namely *partial evaluation, deforestation, supercompilation, distillation* [10]–[13]. In this work, we will focus on *distillation* since it is capable of the same as other approaches but is able to produce a superlinear improvement for the program being optimized [13] and imposes a specific structure of the output programs, namely tail modulo cons where each function call is at most tail-recursive wrapped with a constructor application.

For successful fusion the compressed representation should be fuseable, i.e., it should avoid indexing and be natural to the functional paradigm. A quad-tree representation [14] looks promising in this case. The implementation of this compressed representation as an algebraic data type is straightforward and

```
data QTree a = QNone
             | QVal a
             | QNode (QTree a) (QTree a)
                     (QTree a) (QTree a)
```

Listing 1: Quad-tree compressed representation

could be seen in listing 1, it recursively splits a matrix into four submatrices.

What is more important is that sparse linear algebra routines based on this representation are in a divide-and-conquer form and thus are parallelizable. To increase the performance of function programs we aim to generate an FPGA-kernels from them and at the moment we use FHW project [15] for this, which generates parallel and pipelined dataflow hardware from arbitrary Haskell programs. This approach is also aimed to bridge the gap between our solution and existing sparse linear algebra frameworks: widely used compressed representations could be first converted into quad-tree representation and then our kernels could be called from existing frameworks written in C/C++.

II. Distillation

Distillation [13] is the evolution of the deforestation approach, which was designed to eliminate intermediate trees and lists in functional programs making them as performant as their imperative counterparts. Distillation is a transformation algorithm, which is able to produce superlinear improvement in the runtime of programs. Basically, it glues a sequence of several possibly recursive functions into one possibly recursive function. It is doing so by symbolically applying normal order reduction rules to all possible branches of a program, which is called *driving*, generating a process tree where each node is an expression. The topmost node represents the sequence of functions we want to fuse, and when a similar node is encountered at the bottom of the tree the driving is stoped and a residual program is generated from the tree where such similar nodes represent a new recursive function call, the result of fusion.

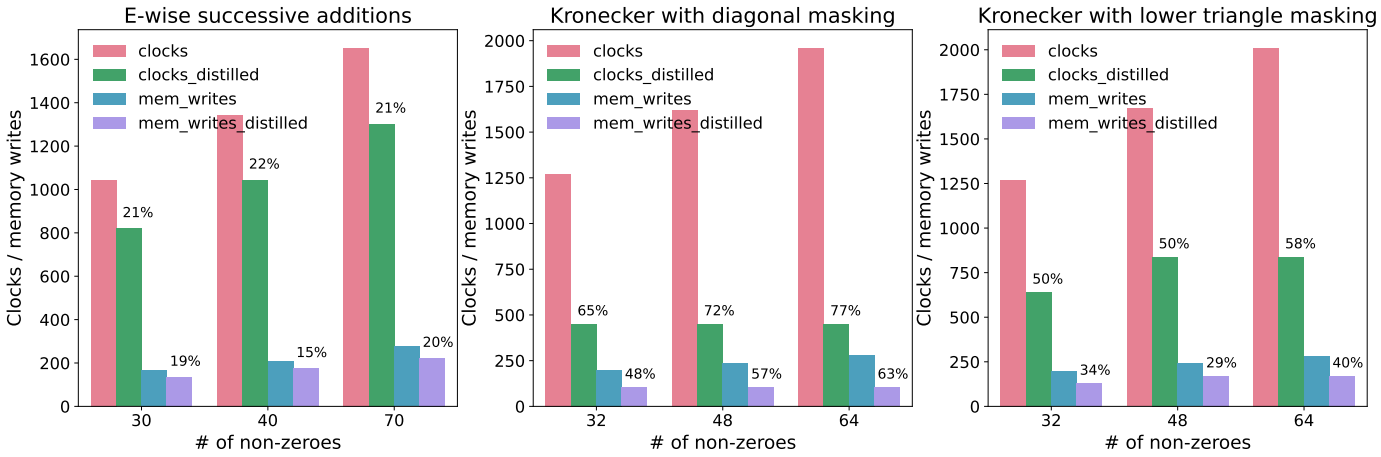

Fig. 3: Hardware evaluation

Function	Matrix size	Reductions		Memory reads		Ratio	
		Original	Distilled	Original	Distilled	Reductions	Memory reads
E-wise additions	64	20317	11459	3170	2372	56%	74%
E-wise additions	2048	139851	81907	20351	15056	58%	73%
Kronecker with triangle masking	64	535125	367868	92470	67110	68%	72%
Kronecker with triangle masking	2048	1215051	827020	212133	151601	68%	71%

TABLE I: Software evaluation

Also, the algorithm imposes a tail modulo cons structure on the output program, which eases the translation into hardware and mitigates the requirements for garbage collection, which is not implemented yet in the hardware compiler that we use [15]. Both these properties are useful but are not used at the moment.

III. FHW PROJECT

FHW [15] is a Haskell-to-System-Verilog compiler that supports functions with arbitrary recursion, which is a problem for most modern high-level synthesis tools. It encodes algebraic data types in hardware and provides a distinct memory space for each. It also provides some optimizations for divide-and-conquer programs which is just the case for our compressed representation of choice. The exact implementation of memory for each type is fully customizable, so the memory for quad-trees could be implemented in any way, e.g. supporting multiple parallel reads/writes for subtrees.

IV. EVALUATION

The distillation is supported for a toy functional language shipped with the distiller, so we implement a set of sparse linear algebra routines in this language leaving everything which is not expressible in the language as free variables, e.g., integers and primitive operations for them. Then the program in this language is distilled and translated into Haskell. Then free variables are substituted with corresponding values and the program is translated into System Verilog using FHW. We use Vivado to simulate generated programs and count the number of clock cycles and memory writes performed during the simulation. The result of evaluation could be seen in figure 3 where we compare distilled and non-distilled programs that were translated into hardware.

The leftmost example is the distillation of a sequence of element-wise additions, which appear for example in Luby's maximal independent set algorithm and is the case of kernel fusion. It could be seen that the fused program is up to 20% more efficient in terms of both clock cycles and memory writes. The number of non-zeroes is the sum of the number of non-zeroes for each matrix (there are three 8x8 matrices). The number of non-zeroes is small due to the fact that the compiler does not support external memory at the moment, thus the construction of matrices should be part of the program itself and the simulator is not comfortable with huge input programs. Also, it is easier to manually count the number of memory writes when the order of matrices is small.

The other two examples are masked Kronecker products, i.e. the case of mask fusion. Here the number of non-zeroes is the number of non-zeroes in the result of Kronecker product before masking, which is also a 8x8 matrix. Since the mask fusion allows to perform evaluation only for the needed elements, fusion provided up to 77% improvement in clock cycles and up to 63% in memory writes for diagonal mask and up to 58% fewer clock cycles and 40% less memory writes for lower triangle masking.

We also evaluated the enhancement brought by fusion using the interpreter of the toy language using the corresponding interpreter and boolean matrices with the structure from [16]. The results of this evaluation could be seen in the table I. Reductions are the number of reductions rules applied during interpretation while memory reads is the number of `case e` contexts encountered where expression `e` is deconstructed.

```
eWiseAdd g m1 m2 =
  case m1 of{
    QNone -> m2;
    QVal v1 -> case m2 of
          QNone -> m1;
          QVal v -> QNode (g v1 v);
          QNode t1 t2 t3 t4 -> error
    QNode q1 q2 q3 q4 -> case m2 of
          QNone -> m1
          QVal v -> error
          QNode t1 t2 t3 t4 -> QNode
                               (eWiseAdd g q1 t1)
                               (eWiseAdd g q2 t2)
                               (eWiseAdd g q3 t3)
                               (eWiseAdd g q4 t4)}

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

The improvement is up to 70% in terms of both reductions and memory reads.

The exact transformation performed by the distiller could be seen in listing 2, which is a simplified excerpt from Luby's maximal independent set algorithm. The series of e-wise additions create an intermediate `new_members` matrix which is eliminated after fusion where all three matrices are destructed in one top-level case expression.

The evaluation shows that sparse linear algebra routines are suitable for fusion with distillation giving noticeable performance both in software and hardware. The hardware part is prominent since it allows to join fusion and sparse hardware where memory could be implemented in the most effective way, unlike modern CPUs and GPUs which perform poorly in sparse applications. The software and hardware benchmarks are available at [17] and [18] respectively.

V. Conclusion

In this work, we proposed the approach of supporting fusion for sparse linear algebra applications using distillation and functional programming. To achieve better performance of functional programs we translate them into System Verilog using a hardware compiler. We presented the evaluation of the approach both in software and hardware which showed prominent results. Future work includes the improvement of the hardware compiler. We want to evaluate divide-and-conquer hardware optimizations as well as to design an efficient memory for quad-tree compressed representation. Also, we want to support external memory to be able to evaluate large real-world examples. Finally, the overall solution could be shipped as a library of OpenCL-like FPGA kernels to be able to utilize it in current workflows.

References

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

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

[3] 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. [Online]. Available: http://dx.doi.org/10.1109/HPEC.2017.8091098

[4] A. Buluc, T. Mattson, S. McMillan, J. Moreira, and C. Yang, "The graphblas c api specification," *GraphBLAS. org, Tech. Rep.*, 2017.

[5] T. A. Davis, M. Aznaveh, and S. Kolodziej, "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)*, 2019, pp. 1–6.

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

[7] "Xla: Optimizing compiler for machine learning," https://www.tensorflow.org/xla?hl=en, accessed: 2020-12-28.

[8] 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, no. 6, p. 556–571, Jun. 2017. [Online]. Available: https://doi.org/10.1145/3140587.3062354

[9] T. A. Davis, "Graph algorithms via suitesparse: Graphblas: triangle counting and k-truss," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, 2018, pp. 1–6.

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

[11] P. Wadler, "Deforestation: transforming programs to eliminate trees," *Theoretical Computer Science*, vol. 73, no. 2, pp. 231–248, 1990. [Online]. Available: https://www.sciencedirect.com/science/article/pii/030439759090147A

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

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

[14] I. Simecek, "Sparse matrix computations using the quadtree storage format," in *2009 11th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, 2009, pp. 168–173.

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

[16] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011. [Online]. Available: https://doi.org/10.1145/2049662.2049663

[17] "Distillation benchmarks," https://github.com/YaccConstructor/Distiller, accessed: 2021-09-08.

[18] "Fhw project," https://github.com/sedwards-lab/fhw, accessed: 2021-09-08.