

Distillation of Sparse Linear Algebra

Aleksey Tyurin

Saint Petersburg University, Russia
JetBrains Research, Russia
alekseytyurinspb@gmail.com

Ekaterina Vinnik

Saint Petersburg University, Russia
JetBrains Research, Russia
catherine.vinnik@gmail.com

Mikhail Nikoliukin

National Research University
Higher School of Economics, Russia
mnikolyukin@edu.hse.ru
michael.nik999@gmail.com

Daniil Berezun

Saint Petersburg University, Russia
JetBrains Research, Russia
d.berezun@spbu.ru
daniil.berezun@jetbrains.com

Semyon Grigorev

Saint Petersburg University, Russia
JetBrains Research, Russia
s.v.grigoriev@spbu.ru
semyon.grigorev@jetbrains.com

Geoff Hamilton

School of Computing,
Dublin City University, Ireland
geoffrey.hamilton@dcu.ie

Linear algebra is a common language for many areas, including machine learning and graph analysis, providing high-performance solutions utilizing the highly parallelizable nature of linear algebra operations. However, a variety of intermediate data structures arising during linear algebra expressions evaluation is one of the primary performance bottlenecks, especially when sparse data are used. We show that program distillation can be efficiently used to optimize linear algebra-based programs, minimizing appearance and evaluation over intermediate data structures at runtime.

1 Introduction

check the paragraph. Nowadays high-performance processing of a huge amount of data is indeed a challenge not only for scientific computing, but for applied systems as well. Special types of hardware, such as General Purpose Graphic Processing Units (GPGPUs), Tensor Processing Units (TPUs), FPGA-based solutions, along with respective specialized software have been developed to provide appropriate solutions, and the development of new solutions continues. In its turn, sparse linear algebra and Graph-BLAS [1] in particular, are a way to utilize all these accelerators to provide high-performance solutions in many areas including machine learning [4] and graph analysis [8].

TODO: connection between paragraphs?

However, evaluation of expressions over matrices generates intermediate data structures similar to the well-known example of a pipelined processing of collections: `map g (map f data)`. Suppose data is a list, then the first map produces a new list which then will be traversed by the second map. The same pattern could be observed in neural networks, where initial data flows through network layers, or in linear algebra expressions, where each subexpression produces an intermediate matrix. The last case occurs not only in scientific computations but also in graph analysis [8]. It is crucial that not only the data structures are traversed multiple times, while it is possible to traverse over them only once, but also the intermediate data populates memory (RAM). Extra memory accesses are a big problem for real-world data analysis: the size of data is huge and memory accesses are expensive operations with noticeable latency. While a number of complex real-world cases including stream fusion and dense kernels fusion [9] could be successfully optimized using deforestation [7, 5] and other techniques **add links**, avoiding intermediate data structures in sparse data processing is still an open problem [8].

2 Proposed Solution

The goal of our research is to find out if the linear algebra-based programs can be efficiently optimized by program distillation [3] by eliminating intermediate data structures and computations. To answer this question, we developed a library of matrix operations [add link to our implementation](#) in POT language: a simple functional language used by Hamilton in his distiller [add link to the Geoff's github](#).

We use a quad-tree matrix representation since it both avoids indexing and can be implemented via algebraic data types, which itself is very natural for functional programming. Besides, it provides a natural way to represent both sparse and dense matrices, as well as makes it possible to express basic operations over the representation via recursive functions traversing the tree-like structure. Finally, the quad-tree representation allows natively exploiting divide-and-conquer parallelism in matrices handling functions.

We selected two different target hardware platforms. The first one is Reduceron [6] — a general-purpose functional-language-specific processor. The second is program-specific hardware for arbitrary functional programs FHW [2] which utilizes the flexibility of FPGA to create hardware for a particular program. While the first case is more typical, the second might provide higher performance for specific tasks.

For now, we propose to use program distillation as the first step of program optimization which, we hope, should reduce memory usage, and then compile a distilled program to the two different hardware platforms by using the respective compiler with platform-specific optimizations. For evaluation, we propose [{propose or some of them are already implemented!?! =\)}](#) to create a library of linear algebraic operations, such as matrix-matrix, matrix-vector, and matrix-scalar operations. Programs of interest are compositions of these basic functions.

3 Preliminary Evaluation

For now, we have implemented some basic functions for the proposed library, which are used in the current evaluation stage: matrix-to-matrix element-wise addition (`mtxAdd`), matrix-to-scalar *apply-to-all* operation (`map`), masking (`mask`), and Kronecker product (`Kron`). The following examples which are a combination of the implemented functions are used for the evaluation.

- Sequential matrix addition:
`seqAdd m1 m2 m3 m4 = mtxAdd (mtxAdd (mtxAdd m1 m2) m3) m4`
- `todo`: `addMask m1 m2 m3 = mask (mtxAdd m1 m2) m3`
- `todo`: `kronMask m1 m2 m3 = mask (kron m1 m2) m3`
- `todo`: `addMap m1 m2 = map f (mtxAdd m1 m2)`
- `todo`: `kronMap m1 m2 = map f (kron m1 m2)`

We compare original versions of these functions and distilled ones in three ways: using interpreter of the POT language to measure the number of reductions and memory allocations, using simulator of Reduceron, and using Vivado's simulation for FHW-compiled programs to measure the number of clock ticks necessary to evaluate a program. We use a set of randomly generated sparse matrices of appropriate size as a dynamic (not known statically) input for both versions. Average results for !!! different inputs are presented in table 1.

We can see that !!! . So, we can conclude !!!

Function	Matrix size				Interpreter		Reduceron	FHW
	m1	m2	m3	m4	Red-s	Allocs	Ticks	Ticks
seqAdd	64×64	64×64	64×64	64×64	2.7	1.9	1.8	??
addMask	64×64	64×64	64×64	–	2.1	1.8	1.4	1.4
kronMask	64×64	2×2	128×128	–	2.2	1.9	1.4	2.7
addMap	64×64	64×64	–	–	2.5	1.7	1.7	1.5
kronMap	64×64	2×2	–	–	2.9	2.2	1.8	2.0

Table 1: Evaluation results: original program to distilled one ratio of measured metrics is presented

4 Future Work

We show that distillation is a promising way to optimize linear algebra-based programs, which makes it also applicable to optimize machine learning and graph processing procedures.

In the future, first of all, we should close a technical debt and make the distiller more stable to handle all important cases: current implementation can not handle such important functions as matrix-matrix multiplication. Along with it, we should improve the input language to make it more user-friendly. The main challenge here is to find the balance between language expressivity and the practicality of distillation for it. Having basic workflow implemented, we should explore how to utilize distillation in the best way for each particular platform. For example, which level of distillation is the best for our particular problem and set of functions? Can we exploit more parallelism using distillation? Can we efficiently exploit the tail-modulo-cons property of the distilled program? What are the limitations of distillation: whether all important cases can be handled?

When the language and the distiller are stable enough, we plan to implement a full-featured generic linear algebra library power enough to express basic graph analysis algorithms and to create and train neural networks. After that, a number of graph analysis algorithms and neural networks will be implemented and evaluated.

In addition to it, we plan to improve both FHW and Reduceron and compilers for them in order to make them mature enough to handle real-world examples. The most relevant improvement here, for example, is the support for out-of-chip memory.

References

- [1] Aydin Buluc, Timothy Mattson, Scott McMillan, Jose Moreira & Carl Yang (2017): *The GraphBLAS C API Specification*. GraphBLAS.org, Tech. Rep.
- [2] S. Edwards, Martha A. Kim, Richard Townsend, Kuangya Zhai & L. Lairmore (2019): *FHW Project : High-Level Hardware Synthesis from Haskell Programs*.
- [3] Geoffrey Hamilton (2021): *The Next 700 Program Transformers*. arXiv:2108.11347.
- [4] Jeremy Kepner, Manoj Kumar, Jose Moreira, Pratap Pattnaik, Mauricio Serrano & Henry Tufo (2017): *Enabling massive deep neural networks with the GraphBLAS*. 2017 IEEE High Performance Extreme Computing Conference (HPEC), doi:10.1109/hpec.2017.8091098. Available at <http://dx.doi.org/10.1109/HPEC.2017.8091098>.
- [5] Simon Marlow & Philip Wadler (1992): *Deforestation for Higher-Order Functions*. In John Launchbury & Patrick M. Sansom, editors: *Functional Programming, Glasgow 1992, Proceedings of the 1992 Glasgow Workshop on Functional Programming*, Ayr, Scotland, UK, 6-8 July 1992, Workshops in Comput-

- 104 ing, Springer, pp. 154–165, doi:10.1007/978-1-4471-3215-8_14. Available at [https://doi.org/10.1007/](https://doi.org/10.1007/978-1-4471-3215-8_14)
105 978-1-4471-3215-8_14.
- 106 [6] MATTHEW NAYLOR & COLIN RUNCIMAN (2012): *The Reduceron reconfigured and re-evaluated*. *Journal of Functional Programming* 22(4-5), p. 574–613, doi:10.1017/S0956796812000214.
- 107
- 108 [7] Philip Wadler (1988): *Deforestation: Transforming Programs to Eliminate Trees*. In Harald Ganzinger, editor:
109 *ESOP '88, 2nd European Symposium on Programming, Nancy, France, March 21-24, 1988, Proceedings*,
110 *Lecture Notes in Computer Science* 300, Springer, pp. 344–358, doi:10.1007/3-540-19027-9_23. Available at
111 https://doi.org/10.1007/3-540-19027-9_23.
- 112 [8] Carl Yang, Aydin Buluç & John D. Owens (2019): *GraphBLAST: A High-Performance Linear Algebra-based*
113 *Graph Framework on the GPU*. CoRR abs/1908.01407. arXiv:1908.01407.
- 114 [9] Zhen Zheng, Pengzhan Zhao, Guoping Long, Feiwen Zhu, Kai Zhu, Wenyi Zhao, Lansong Diao, Jun Yang &
115 Wei Lin (2020): *FusionStitching: Boosting Memory Intensive Computations for Deep Learning Workloads*.
116 CoRR abs/2009.10924. arXiv:2009.10924.