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 Nikolukin
!!!
!!!

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

Nowadays high-performance processing of a huge amount of data is an actual challenge not only for scientific computing but for applied systems. Special types of hardware, such as General Purpose Graphic Processing Units (GPGPUs), Tensor Processing Units (TPUs), FPGA-based solutions, along with respective specialized software were developed to provide appropriate solutions, and the development of new solutions continues. And sparse linear algebra, particularly GraphBLAS [1], is a way to utilize all these accelerators to provide high-performance solutions in many areas including machine learning [4] and graph analysis [7].

But evaluation of expressions over matrices generates intermediate data structures similar to the well-known example is 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 can be observed in neural networks where initial data flow 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 in graph analysis [7]. It is important that not only data structure will be traversed multiple times while can be traversed only once, but the intermediate data will also be stored (to RAM) which is a big problem for real-world data analysis: the size of data is huge and memory access is one of the expensive operations. While a number of complex real-world cases including stream fusion and dense kernels fusion [8], can be successfully optimized using deforestation and other techniques, avoiding intermediate data structures in sparse data processing is still an open problem [7].

Submitted to:
VPT 2022

© A. Tyurin, E. Vinnik, et al
This work is licensed under the
Creative Commons Attribution License.

2 Proposed Solution

The goal of our research is to figure out can distillation [3] be a solution of the intermediate data structures problem in linear algebra based programs. To answer this question we developed a library of matrix operations in POT language: simple functional language used by Geoff Hamilton in his distiller.

We use a quad-tree representation [6] for matrices because it avoids indexing and natural for functional programming because can be defined as an algebraic data type. Moreover it allows one to represent both sparse and dense matrices naturally, and basic operations over such a representation can be natively expressed as recursive functions which traverse this tree-like structure. Additionally, this structure allows natively exploiting divide-and-conquer parallelism in matrices handing functions.

We selected two different target hardware platforms. The first one is a Reduceron [5] — general-purpose functional-language-specific processor. The second one 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 one may provide higher performance for specific tasks.

At the current stage, we propose to use distillation as the first step of program optimization which, we hope, should reduce memory traffic, and then compile a distilled program to two different hardware platforms using the respective compiler with platform-specific optimizations. For evaluation, we propose 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

Compositions of such basic functions, matrix-matrix elementwise operations (`mtxAdd`), matrix-scalar elementwise operation (`map`), masking (`mask`), Kronecker product (`Kron`) were used for evaluation. Namely, we use following compositions.

- `seqAdd m1 m2 m3 m4 = mtxAdd (mtxAdd (mtxAdd m1 m2) m3) m4`
- `addMask m1 m2 m3 = mask (mtxAdd m1 m2) m3`
- `kronMask m1 m2 m3 = mask (kron m1 m2) m3`
- `addMap m1 m2 = map f (mtxAdd m1 m2)`
- `kronMap m1 m2 = map f (kron m1 m2)`

We compare original versions of these functions and distilled ones in three ways: using interpreter of POT language to measure a number of reductions and memory allocation, using simulator of Reduceron, and using simulation of FHW to measure a number of clock ticks necessary to evaluate a program. We use a set of randomly generated sparse matrices of appropreate 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 !!!

4 Future Work

We show that distillation is a promising way to optimize linear algebra based programs, thus it is a promising a way to optimize machine learning and graph processing procedures.

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	10
addMask	64×64	64×64	64×64	–	2.1	1.8	1.4	10
kronMask	64×64	2×2	128×128	–	2.2	1.9	1.4	10
addMap	64×64	64×64	–	–	2.5	1.7	1.7	10
kronMap	64×64	2×2	–	–	2.9	2.2	1.8	10

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

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 will be 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.

Along with it we plan to improve both FHW and Reduceron and compilers for it in order to make them mature enough to handle real-world examples. For example, it is necessary to support 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] 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.

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

[7] Carl Yang, Aydin Buluç & John D. Owens (2019): *GraphBLAST: A High-Performance Linear Algebra-based Graph Framework on the GPU*. *CoRR* abs/1908.01407. arXiv:1908.01407.

[8] Zhen Zheng, Pengzhan Zhao, Guoping Long, Feiwen Zhu, Kai Zhu, Wenyi Zhao, Lansong Diao, Jun Yang & Wei Lin (2020): *FusionStitching: Boosting Memory Intensive Computations for Deep Learning Workloads*. *CoRR* abs/2009.10924. arXiv:2009.10924.