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
mnnikolyukin@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 by 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 structures are used. We show that program distillation can be efficiently used to optimize linear algebra-based programs by minimizing occurrence and evaluation of intermediate data structures at runtime.

Keywords— fusion, high-level synthesis, sparse data processing, linear algebra

1 Introduction

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 GraphBLAS [2] in particular, are a way to utilize all these accelerators to provide high-performance solutions in many areas including machine learning [9] and graph analysis [17].

Unfortunately, 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 [17]. 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. It was shown that CPUs and GPUs are uberutilized in sparse data processing cases [18, 15]: compute units do not achieve peak performance due to memory traffic problems.

While a number of complex real-world cases including stream fusion and dense kernels fusion [19] could be successfully optimized using deforestation [16, 11] and other techniques [7, 10], avoiding intermediate data structures in sparse data processing is still an open problem [17]. Existing works show that kernels fusion improves performance in such cases as graph neural networks [8], scientific computations [1], graph processing [3], but neither of them does not proposes systematic general solution for sparse kernels fusion.

Submitted to:
VPT 2022

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

Listing 1: Quad-tree compressed representation

2 Proposed Solution

The goal of our research is to find out if linear algebra-based programs can be efficiently optimized by program distillation [6] through eliminating intermediate data structures and computations. To answer this question, we have developed a library of matrix operations[1] in POT language: a simple functional language used by Hamilton in his distiller[2].

We use a quad-tree matrix representation [14] since it both avoids indexing and can be implemented via algebraic data type as depicted in listing 1, which itself is very natural for functional programming. Besides, it provides similar compression rate to widely adopted CSR and COO, and 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 to natively exploit divide-and-conquer parallelism in matrices handling functions.

Since general purpose devices like CPUs and GPUs appear to be heavily underutilized when executing sparse routines due to low arithmetic intensity and memory boundness, custom hardware seems to be a promising solution towards mitigating these issues. Distillation provides all the needed optimizations for free for a functional language, so what remains is to translate a functional program into custom hardware. We have opted for two solutions here. The first one is Reduceron [12] — a processor, designed to perform highly performant reductions. The second one is FHW [5] — a project, which is constituted of several compilers that help to translate an arbitrary Haskell progam into System Verilog to eventually provide a bitstream for a custom hardware. It leverages a parallel and pipelined dataflow representation, which is abstracted over, e.g., nodes for memory operations, which makes it possible to come up with a specialized memory for our data structures. 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 other unnecessary computations, and then compile a distilled program to the two different hardware platforms by using the respective compiler with platform-specific optimizations.

For evaluation, we have been implementing a library of a subset of linear algebraic routines, only matrix-matrix operations for now. All operations are simplified, for example, they do not check matrices' size correctness, handling of incorrect input not provided. Programs of interest are compositions of these basic functions.

3 Implementation

Initially neither Reduceron nor FHW were stable enough to run our sparse routines. Reduceron supported only limited number of arguments for each function, while FHW had support only for algebraic data types with less than 64 number of fields which was not the case when translating our benchmarks. So both issues have been fixed. On top of that, appropriate initial values were set for System Verilog signals in FHW to make it possible to utilize Vivado to handle further development.

FHW relies on External Core feature of GHC as a frontend, which has been removed since GHC $>$ 7.6.3. To mitigate this dependency, to make the code base more maintainable, and to overcome other issues, like overflows during CPS-transformation and support for partially applied functions, we have opted for GRIN [13] as an

[1]Linear algebra operations in POT language: `https://github.com/YaccConstructor/Distiller/blob/master/inputs/LinearAlgebra.pot`

[2]Distiller implementation on GitHub: `https://github.com/poitin/Distiller`

intermediate representation between POT language and dataflow representation of FHW. GRIN makes defunctionalization, which is essential for hardware generation, more convenient and provides extensive points-to analysis. Finally, FHW assumes the presence of a hardware garbage collector, but does not implement it. We also have not implemented this feature yet.

The distiller at the moment produces function duplicates during residualization, which is non-trivial to resolve. Such duplicates increase the consumption of logic blocks in hardware, so we remove them before translation. The usage of De Bruijn indexes makes it possible to rename only function names to determine whether two functions are duplicates.

Both Reduceron and FHW do not support external memory at the moment, and thus we store all the date inside our programs. It makes FHW-generated hardware larger, introduces code size overhead and unnecessary reductions.

4 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`), which takes a subset of matrix elements, and Kronecker product (`Kron`). The following examples, which are a combination of the implemented functions, are used for the evaluation. The examples are fairly practical, for example, one could see a sequence of element-wise additions in a Luby's maximal independent set algorithm, and masking is a one of key operation in GraphBLAS standard [2].

- Sequential addition of four matrices:
  `seqAdd m1 m2 m3 m4 = mtxAdd (mtxAdd (mtxAdd m1 m2) m3) m4`
- Masking of two matrices addition:
  `addMask m1 m2 m3 = mask (mtxAdd m1 m2) m3`
- Masking of two matrices Kronecker product:
  `kronMask m1 m2 m3 = mask (kron m1 m2) m3`
- Element-wise processing of two matrices addition:
  `addMap m1 m2 = map f (mtxAdd m1 m2)`
- Element-wise processing of two matrices Kronecker product:
  `kronMap m1 m2 = map f (kron m1 m2)`

We compare original versions of these functions and distilled ones in three ways. First, we use the interpreter of the POT language to measure the number of reductions and memory reads inside `case` expressions. We use the simulator shipped with Reduceron to measure the number of clock ticks necessary to evaluate a program, and Vivado's simulator for FHW-compiled programs to measure the number of both clock ticks and memory writes that a program produces. It is worth noting that Reduceron has somewhat fixed clock frequency, while frequency for FHW-generated hardware varies depending on a particular program. Since we do not have external memory at the moment, and all the data lives inside the generated scheme, the logic is not synthesizable for reasonably sized matrices in the case of FHW. We get similar clock frequencies for distilled and non-distilled programs for inputs with smaller matrices and hence assume that clock frequencies are also similar below. Thus, we provide only the number of ticks instead of time.

A set of sparse matrices of appropriate sizes provided at [4] is used. The matrices are converted into boolean ones since POT language lacks the needed primitives at the moment. Average results for several hundreds of different inputs are presented in table 1.

We can see that on average distillation provides up to 3 and 2 times improvement in terms of reductions and memory reads respectively for the interpreter. The number of reductions is also considerably reduced for hardware benchmarks. The lack of matches between ticks for FHW and Reduceron is justified by architecture distinction. Finally, from the last column one could see memory consumption reduction, which supports our approach. All this hopefully makes the proposed solution viable, and we look forward to coming up with full-fledged experiments that would target real hardware and real life competitors like C++ implementations.

Function	Matrix size				Interpreter		Reduceron	FHW	
	m1	m2	m3	m4	Red-s	Reads	Ticks	Ticks	Writes
seqAdd	64×64	64×64	64×64	64×64	2.7	1.9	1.8	1.4	1.1
addMask	64×64	64×64	64×64	–	2.1	1.8	1.4	1.4	1.1
kronMask	64×64	2×2	128×128	–	2.2	1.9	1.4	2.7	2.5
addMap	64×64	64×64	–	–	2.5	1.7	1.7	1.5	1
kronMap	64×64	2×2	–	–	2.9	2.2	1.8	2.0	1

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

5 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, we should close a technical debt and make the distiller more stable to handle all the 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.

Finally, a comprehensive evaluation of proposed solutions should be done using FPGA, not hardware simulators. The main challenge here is to utilize the resources of modern FPGA accelerators, such as HBM, to achieve the maximal performance of our solution.

References

[1] José I. Aliaga, Joaquín Pérez & Enrique S. Quintana-Ortí (2015): *Systematic Fusion of CUDA Kernels for Iterative Sparse Linear System Solvers*. In Jesper Larsson Träff, Sascha Hunold & Francesco Versaci, editors: *Euro-Par 2015: Parallel Processing*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 675–686.

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

[3] Xuhao Chen, Pingfan Li, Jianbin Fang, Tao Tang, Zhiying Wang & Canqun Yang (2017): *Efficient and high-quality sparse graph coloring on GPUs*. *Concurrency and Computation: Practice and Experience* 29(10), p. e4064, doi:https://doi.org/10.1002/cpe.4064. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.4064. E4064 cpe.4064.

[4] Timothy A. Davis & Yifan Hu (2011): *The University of Florida Sparse Matrix Collection*. *ACM Trans. Math. Softw.* 38(1), doi:10.1145/2049662.2049663. Available at `https://doi.org/10.1145/2049662.2049663`.

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

[6] Geoffrey Hamilton (2021): *The Next 700 Program Transformers*. arXiv:2108.11347.

[7] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein & Cosmin E. Oancea (2017): *Futhark: Purely Functional GPU-Programming with Nested Parallelism and in-Place Array Updates*. *SIGPLAN Not.* 52(6), p. 556–571, doi:10.1145/3140587.3062354. Available at `https://doi.org/10.1145/3140587.3062354`.

[8] Mert Hidayetoglu, Carl Pearson, Vikram Sharma Mailthody, Eiman Ebrahimi, Jinjun Xiong, Rakesh Nagi & Wen mei Hwu (2020): *At-Scale Sparse Deep Neural Network Inference With Efficient GPU Implementation*. In: *2020 IEEE High Performance Extreme Computing Conference, HPEC 2020, Waltham, MA, USA, September 22-24, 2020*, IEEE, pp. 1–7, doi:10.1109/HPEC43674.2020.9286206. Available at `https://doi.org/10.1109/HPEC43674.2020.9286206`.

[9] 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`.

[10] Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos & Yannis Smaragdakis (2016): *Stream Fusion, to Completeness*. *CoRR* abs/1612.06668. arXiv:1612.06668.

[11] 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 Computing, Springer, pp. 154–165, doi:10.1007/978-1-4471-3215-8_14. Available at `https://doi.org/10.1007/978-1-4471-3215-8_14`.

[12] 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.

[13] Peter Podlovics, Csaba Hruska & Andor Pénzes (2021): *A Modern Look at GRIN, an Optimizing Functional Language Back End*. *Acta Cybernetica*, doi:10.14232/actacyb.282969.

[14] 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.

[15] William S. Song, Vitaliy Gleyzer, Alexei Lomakin & Jeremy Kepner (2016): *Novel graph processor architecture, prototype system, and results*. In: *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7, doi:10.1109/HPEC.2016.7761635.

[16] Philip Wadler (1988): *Deforestation: Transforming Programs to Eliminate Trees*. In Harald Ganzinger, editor: *ESOP '88, 2nd European Symposium on Programming, Nancy, France, March 21-24, 1988, Proceedings*, *Lecture Notes in Computer Science* 300, Springer, pp. 344–358, doi:10.1007/3-540-19027-9_23. Available at `https://doi.org/10.1007/3-540-19027-9_23`.

[17] 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.

[18] Zhekai Zhang, Hanrui Wang, Song Han & William J. Dally (2020): *SpArch: Efficient Architecture for Sparse Matrix Multiplication*. *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 261–274.

[19] 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.