Linear Algebra Partial Evaluation Using AnyDSL*

*Note: NOT FINAL

Ilya Balashov
Saint Petersburg State University
7/9 Universitetskaya nab.,
St. Petersburg, 199034 Russia
i.balashov@2017.spbu.ru

Semyon Grigorev
Saint Petersburg State University
7/9 Universitetskaya nab.,
St. Petersburg, 199034 Russia
s.v.grigoriev@spbu.ru

Daniil Berezun
Saint Petersburg State University
7/9 Universitetskaya nab.,
St. Petersburg, 199034 Russia
SPBU email!

***Abstract*—This document is a model and instructions for LaTeX. This and the IEEEtran.cls file define the components of your paper [title, text, heads, etc.]. *CRITICAL: Do Not Use Symbols, Special Characters, Footnotes, or Math in Paper Title or Abstract.**

***Index Terms*—component, formatting, style, styling, insert**

I. Introduction

Recent years have seen a significant increase in sizes and complexity of programs in different areas of software engineering. A possible way of satisfying these requirements is usage of automatic optimization tools and techniques, operating with program sources.

For instance, so-called *partial evaluation* (or specialization) [1] technique is being actively used over last years as a way to optimize program execution time automatically, using known statically data. A special tool named *partial evaluator* (or specializer) analyzes data (for example, function parameters) which was provided ahead of evaluation time and applies several program optimization techniques based on the structure of this data.

One of the possible applications of partial evaluation is optimization of algorithms expressed in terms of linear algebra. It is well-known [2] that many graph algorithms could be constructed using the language of some basic operations: matrix multiplication, Kronecker (tensor) product etc. Moreover, it is possible to use matrices as a data storage in algorithms from different other area. If it is possible to build an algorithm (or representation of it's data) using some relatively small algorithmic bricks, therefore, it should be possible to utilize small and statically known bricks for optimization purposes using partial evaluation technique.

Existing results in the area of applied usage of partial evaluation for automatic linear algebra algorithms or algorithms with matrix data optimization are limited by several partially successful experiments with CUDA- and Clang-based partial evaluators [3] [my coursework???]. The main contribution of this work will be providing some experiments on linear algebra partial evaluation with AnyDSL [4] framework on CPU. We will show that partial evaluation using this specific partial evaluator gives significant increase in execution times for all tested algorithms on the most of given datasets.

II. Background

A. Partial evaluation

Let's suppose:

- P is a program, which takes values a_n $[n = 1..m]$ as an input
- mix is a program which is defined as $mix\ [P, a_1] = P_a$
- $[\![P]\!][a_1, a_2, .., a_m] = [\![P_a]\!][a_2, .., a_m]$

Then the transformation of P and a_1 to P_a using mix is called *partial evaluation* [1]. Program mix is called *partial evaluator*. In other words, partial evaluation is a technique for evaluating parts of the program ahead of compilation with the usage of static input data.

Despite partial evaluation is initially being used by Ershov [5], Jones [1] and other scientist in their work for compiler generation via Futamura projections [6], it could also be used for program optimization. For instance, partial evaluator can employ static data to unfold loops and conditional operators, propagate constants, etc [1].

However partial evaluation is a powerful method of program optimization, it is inherent in several difficulties. Firstly, partial evaluator could inflate source code size heavily because of transformation such as loop unfolding and static data substitution. Therefore, evaluation results (code structure, bottlenecks, etc.) formal assessment becomes a non-trivial problem very often. To solve issue in some degree, modern tools like AnyDSL [4] tends to translate evaluated code into some intermediate representation which is often much easier to understand and analyze. Secondly, divergent program partial evaluation with the application of average-quality tool may lead to the evaluation process divergence [1]. So, the programmer have to be very careful while using this technique for optimization purposes. Finally, partial evaluation imposes serious requirements on the programmer qualification: deep understanding of evaluation process is highly required. To solve this issue modern tools are introducing simplified language constructions, such as a special partial evaluation wrappers [4], attribute-driven evaluation [7] and many other various and creative methods.

B. Graph algorithms in the linear algebra language

It is widely known that many of graph algorithms could be explained in the language of matrices [2], [8]. Linear algebra allows constructing algorithms like Breadth-First Search or Shortest Path Search with exploitation of basic linear algebra operations: matrix multiplication, Kronecker product, etc.

For instance, one may write down Breadth-First Search in the manner of (listing). Each iteration of the algorithm represents one matrix-vector multiplication.

Formula/Code Listing

Therefore, if it was possible to speed up different matrix multiplication algorithms, it would be possible to speed up a large class of algorithms.

One of the possible basic sets of linear algebra algorithms and operations for graph algorithm construction is named *GraphBLAS* standard [8], [9]. However SuiteSparse GraphBLAS is usually considered as the state-of-art implementation of this standard [8], there are a number of custom wrappers and implementations [10], [11].

III. Algorithms implementation

All algorithms were implemented using AnyDSL Impala domain-specific language [4] for partial evaluation. AnyDSL framework was chosen due to it's Impala DSL with comparatively simple Rust-like syntax and relatively available documentation. Algorithm code is represented as computation kernels, which is further linked with Google Benchmark-based [12] benchmarking code. Each algorithm was implemented in Impala twice: with partial evaluation language constructions and without them (therefore, with no partial evaluation).

Also, every algorithm was implemented with an alternative tool or framework that is usually used in practice for algorithm implementation in the corresponding area. In details, the following programs were used:

- SuiteSparse GraphBLAS(link) — for graph algorithms in the terms of linear algebra
- Grep and eGrep — for algorithms on strings and regular expressions

All the code is placed on GitHub:

https://github.com/ibalashov24/spec_experiments

IV. Experimental design

In this section we will describe our experimental design for partial evaluation of selected algorithms using AnyDSL framework.

A. Experimental setup

Configuration of the experimental stand was:

- Intel Core i5-7440HQ (4x3.8GHz) CPU
- 16Gb RAM
- Ubuntu 20.04

Tools' versions were fixed on the following commits from their official repositories:

- Google Benchmark [12] — commit dated 22 December 2020
- AnyDSL [4] — commit dated 8 December 2020
- SuiteSparse GraphBLAS [9] — commit dated 14 July 2020

Default (e)Grep from Ubuntu 20.04 was employed.

We used Harwell-Boeing matrix collection [13] (a subset of it is also known as SuiteSparse matrix collection [14]) because it contains reasonably diverse set of matrices. COO (COOrdinate list) sparse matrix format was used.

For string algorithms, we used random strings and traffic dumps as sources and and random strings or latin words as patterns. Regular expressions (finite automata) were converted to COO sparse representation with our modification of Re2dfa tool [15].

AnyDSL partial evaluation tool was executed in JIT-mode [4], which allows to perform partial evaluation at the run time.

B. Research questions

To evaluate our approach, we design experiments to address the following research questions:

Q1: Does partial evaluated benefits string and matrix-based graph algorithms performance (execution time) comparing to their basic versions?

Q2: In which degree partially evaluated algorithms code performance gets closer to their state-of-art implementations?

Q3: How does partial evaluator influences algorithms' code size?

C. Result metrics

To evaluate the performance of partially evaluated code, we adopt the following widely used metrics for application performance:

- **Execution time** is computed by Google Benchmark tool and measured in nanoseconds. For each of algorithm the tool gives three numbers: time spent in real life, time spent on CPU and iteration number. We took *time spent in real life* in order to consider all hardware delays (for example, memory access delays).
  The smaller execution time is better.
- **Measure error** is computed by Google Benchmark tool and measured in percents.
  Numbers smaller than 0.01% are considered as good result which guarantees relatively small threat to validity.
- **Code ramification** metric consists of the number of lines of code in LLVM IR representation of algorithms generated by AnyDSL and the number of conditional jumps in this representation. The smaller code ramification is better.

V. Results

This section presents our experimental results by addressing the research questions.

Time, ns.	bcsstk16 x 2blocks	bcsstk16 x eye3	fs1831 x 2blocks	fs1831 x eye3
		Matrix-Matrix product		
No spec	3569461	152191	48082	7526
Spec	172553	94126	20487	802
SuiteSparse	5302	1825	1825	205
		Kronecker product		
No spec	186	126550	22509	966
Spec	34.5	157196	2016	893
SuiteSparse	198	199	197	199

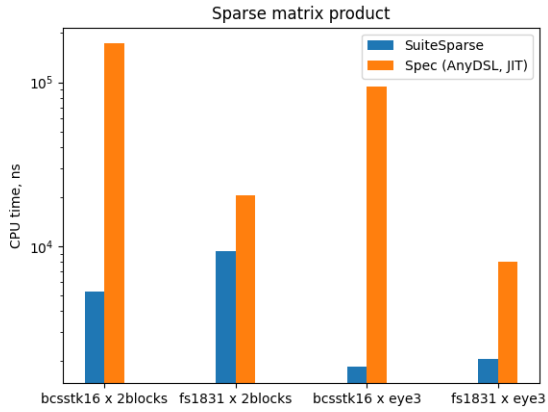

Fig. 1. Execution time of matrix multiplication algorithm comparison between SuiteSparse and partially evaluated code using AnyDSL

A. Does partial evaluation with AnyDSL benefits string and matrix-based graph algorithms performance comparing to their basic versions?

For matrix algorithms (both matrix-matrix product and Kronecker product), 4 matrices were taken from Harwell-Boeing: *bcsstk16*, *fs1831*, *2blocks* and *eye3*. As seen from [table 1], partial evaluation gives significant, more than several times, speed up on test cases involving *2blocks* as right multiplicator. It may be explained with relatively distributed structure of *2blocks* matrix non-zero elements, that allows partial evaluator to effectively perform optimizations like loop unfolding and constant propagation.
In contrast, non-zero elements of *eye3* matrix are concentrated near the main diagonal of the matrix, that leads to relatively small execution time benefit of partial evaluation — loop unfolding does not discard any empty iterations.

For string algorithms, we may observe much more noticeable execution time increase after partial evaluation than in graph algorithms. As could be seen from [table 2], the speed up lays between 10 and 100 times depending on the test. The reason of such a significant increase is that the most of iterations in classic substring search and pattern matching algorithms [16] with matrix input are not empty, like in previously discussed algorithms on sparse matrices graphs. Also, in substring search algorithm evaluation is simplified by the fact that the data is being iterated successively.
Moreover, there is absent of non-logical operations with both source and pattern data as operands in these algorithms, so the partial evaluator is able to apply constant propagation optimization heavily due to trivial data separation.

The results show that in general partial evaluation with AnyDSL benefits string and matrix-based graph algorithms execution time comparing to their basic versions.

B. In which degree partially evaluated algorithms code performance gets closer to their state-of-art implementations?

[Tables 3 and 4] show the time (in nanoseconds) of execution of matrix-based graph and string algorithms respectively.

For the string algorithms, we can see that partially evaluated code outperforms Grep (for pattern matching) and eGrep (for regular expressions matching) in several times (2 to 10000) on each of datasets. However AnyDSL beat (e)Grep in both pattern and regular expression matching problems, we could see that the latter gave by several orders of magnitude stronger results. According to our analysis, it could be the result of using COO representation for regular expression's transition graph in the experiment: linear structure of a COOrdinate list structure allows partial evaluator to use more aggressive optimizations such as vectorization or easier loop unfolding.

For graph algorithms in a matrix form (matrix multiplication and Kronecker product), we may observe that partially evaluated algorithms' code underperforms code of the same algorithms implemented with SuiteSparse GraphBLAS in 10 times in average. It could be considered as good result, since non-partially evaluated code loses 100 times in the half of cases.

To sum up, for the selected string algorithms partially evaluated code outperforms their industrial implementations by execution time in high degree; for the selected graph algorithms in matrix form partially evaluated code lags behind their state-of-art implementation by a factor of 10 (which is a good result).

C. How does partial evaluator influences algorithms' code size?

blah-blah-blah

VI. Threats to validity

A. Subject selection bias

In our research we use only AnyDSL framework for the experiments. Another partial evaluation tools may give slightly different results due more or less aggressive optimizations or different evaluation techniques.

B. Used datasets

Despite trying to run experimental code on both versatile and special datasets, we admit that partially evaluated code could give slightly different measures on some other special degenerate matrix sets.

VII. Related work

Partial evaluation of linear algebra (matrix algorithms) was studied before in several papers.

Firstly, colleagues measured [3] that partial evaluation of matrix convolution and pattern matching algorithms using AnyDSL framework and CUDA reduces execution times significantly on the most datasets.

Secondly, some research was performed on Viterbi algorithm partial evaluation. There should be the description of Ivan's work. I do not understand the topic enough at the moment

Also, AnyDSL team performed research [17] on application of partial evaluation for ray tracing purposes in their library named Rodent. It was measured that partial evaluation makes an improvement in execution time of around 25% on selected datasets.

References

[1] N. D. Jones, C. K. Gomard, and P. Sestoft, *Partial evaluation and automatic program generation*. Peter Sestoft, 1993.

[2] J. Kepner and J. Gilbert, *Graph algorithms in the language of linear algebra*. SIAM, 2011.

[3] A. Tyurin, D. Berezun, and S. Grigorev, "Optimizing gpu programs by partial evaluation," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2020, pp. 431–432.

[4] R. Leißa, K. Boesche, S. Hack, A. Pérard-Gayot, R. Membarth, P. Slusallek, A. Müller, and B. Schmidt, "Anydsl: A partial evaluation framework for programming high-performance libraries," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–30, 2018.

[5] A. P. Ershov, "Mixed computation: Potential applications and problems for study," *Theoretical Computer Science*, vol. 18, no. 1, pp. 41–67, 1982.

[6] Y. Futamura, "Partial computation of programs," in *RIMS Symposia on Software Science and Engineering*. Springer, 1983, pp. 1–35.

[7] E. Sharygin, R. Buchatskiy, R. Zhuykov, and A. Sher, "Runtime specialization of postgresql query executor," in *Perspectives of System Informatics*, A. K. Petrenko and A. Voronkov, Eds. Cham: Springer International Publishing, 2018, pp. 375–386.

[8] T. A. Davis, "Algorithm 1000: Suitesparse: Graphblas: Graph algorithms in the language of sparse linear algebra," *ACM Transactions on Mathematical Software (TOMS)*, vol. 45, no. 4, pp. 1–25, 2019.

[9] J. E. Moreira, M. Kumar, and W. P. Horn, "Implementing the graphblas c api," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2018, pp. 298–309.

[10] M. Pelletier. Pygraphblas: Graphblas for python. [Online]. Available: https://github.com/michelp/pygraphblas(accessed:28March2021)

[11] J. E. Moreira and B. Horn. Ibm grahpblas. [Online]. Available: https://github.com/IBM/ibmgraphblas(accessed:28March2021)

[12] Google. Google benchmark. [Online]. Available: https://github.com/google/benchmark(accessed:28March2021)

[13] I. S. Duff, R. G. Grimes, and J. G. Lewis, "Users' guide for the harwell-boeing sparse matrix collection (release i)," 1992.

[14] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, pp. 1–25, 2011.

[15] "re2dfa: convertor from regular expressions to graphs". [Online]. Available: https://github.com/ibalashov24/re2dfa(accessed:28March2021)

[16] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.

[17] A. Pérard-Gayot, R. Membarth, R. Leißa, S. Hack, and P. Slusallek, "Rodent: generating renderers without writing a generator," *ACM Transactions on Graphics (TOG)*, vol. 38, no. 4, pp. 1–12, 2019.