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. [] High loaded system are being employed by noticeable number of projects, and therefore, requirements for program execution time tightened strongly. []

A possible way of satisfying these requirements is usage of automatic optimisation tools and techniques, operating with program sources. For instance, so-called *partial evaluation* (or specialization) technique is being actively used over last years as a way to optimize program execution time automatically using data known statically. [AnyDSL, Alexey T.] 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. Despite being known and employed in theoretical studies for more than 40 years, partial evaluation still provides a huge space for both theoretical and practical study.

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

Existing results in the area of applied usage of partial evaluation for automatic linear algorithm optimization are limited by several partially successful experiments with CUDA- and Clang-based partial evaluators. So, the main contribution of this work will be providing some experiments on linear algebra partial evaluation with AnyDSL framework on CPU. We will show that partial evaluation using this specific partial evaluation tool gives significant increase in execution times of 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*. 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 [], Jones [] and other scientist for compiler generation via Futamura projections [], 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 [].

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 [] 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. [Jones] 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 [AnyDSL paper], attribute-driven evaluation [LLVM.mix paper?] 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. [G] 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. [G]

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. However SuiteSparse GraphBLAS is usually considered as the state-of-art implementation of this standart [GRB repo], there are a number of custom wrappers and implementations in Python [Orachev], C# [gh], ...

C. *Partial evaluation tools*

...

III. Algorithms Implementation

All algorithms were implemented using AnyDSL Impala domain-specific language [] for partial evaluation. Algorithm code is represented as computation kernels, which is further linked with Google Benchmark-based [] 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 (link).

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 []— commit dated 22 December 2020
- AnyDSL []— commit dated 8 December 2020
- SuiteSparse GraphBLAS [] — commit dated 14 July 2020

Default (e)Grep from Ubuntu 20.04 was employed.

We used Hardwell-Boeing matrix collection [] (it's subset is also known as SuiteSparse matrix collection []) because it contains reasonably diverse set of matrices. COO [] 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 [].

AnyDSL partial evaluation tool was executed in JIT-mode [], 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.

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 [Cormen] 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.

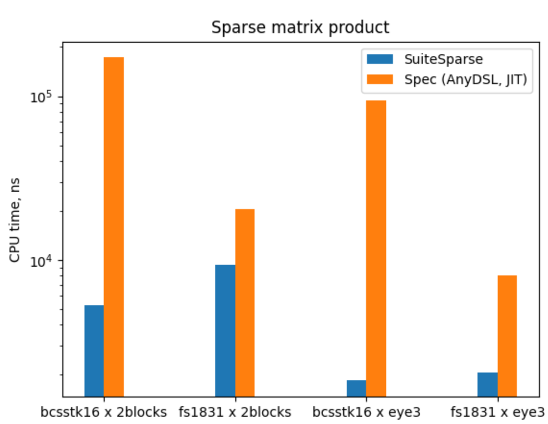

Fig. 1. Execution time of matrix multiplication algorithm comparison between SuiteSparse and partially evaluated code using AnyDSL

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 [Tyurin, 2020] that partial evaluation of matrix convolution and pattern matching algorithms using AnyDSL framework [site] and CUDA [] reduces execution times significantly on the most datasets.

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

Also, AnyDSL team provided papers [Stincillia paper] on application of partial evaluation for image processing purposes in their library named Stincillia [link to lib]. It was measured that partial evaluation speeds algorithms up to 10 times comparing to not evaluated ones on the selected datasets.

[Something more]

REFERENCES

[1] G. Eason, B. Noble, and I. N. Sneddon, "On certain integrals of Lipschitz-Hankel type involving products of Bessel functions," Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955.

[2] J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.

[3] I. S. Jacobs and C. P. Bean, "Fine particles, thin films and exchange anisotropy," in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.

[4] K. Elissa, "Title of paper if known," unpublished.

[5] R. Nicole, "Title of paper with only first word capitalized," J. Name Stand. Abbrev., in press.

[6] Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, "Electron spectroscopy studies on magneto-optical media and plastic substrate interface," IEEE Transl. J. Magn. Japan, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982].

[7] M. Young, The Technical Writer's Handbook. Mill Valley, CA: University Science, 1989.