Empirical Study of Partial Evaluation of Matrix and String Algorithms

Ilya Balashov
Daniil Berezun
Semyon Grigorev

Saint Petersburg State University

SYRCoSE'21

Introduction: Partial Evaluation and Huge Programs

- Huge programs often have core parts, which are heavily loaded or employed by a large number of other parts
  - Matrix multiplication algorithms in BLAS
  - Pattern and automaton matching in programs on strings
- Optimization of core parts $\equiv$ optimization of the whole program
- It is difficult to write well-optimized programs
  - Helper tools and methods are needed
- *Partial Evaluation* — one of such a methods

In This Report...

Partial evaluation applied to a linear algebra and string algorithms.

1. Brief introduction to the background
   - Partial evaluation
   - Algorithms for the experiments
2. Algorithms implementation
3. Experimental design
4. Results
5. Related & future work

Background: Partial Evaluation

- Method of program transformation
  - Tool — Partial Evaluator
- $[\![F]\!][a, b] = [\![F_b]\!][a]$
  - F_b could be faster, smaller and easier than F
- Also used for compiler generation
  - Futamura projections
- We chose AnyDSL framework
  - DSL Impala and Artic
  - Previously applied in several areas:
    - Image processing
    - Bioinformatics
    - Ray tracing

Background: Algorithms

- Matrix algorithms
  - Matrix multiplication
  - Kronecker (tensor) product
- String processing algorithms
  - Pattern matching
  - Regular expression matching (in matrix form)
- Applied widely both in science and industry
  - GraphBLAS primitives
  - KMP-test
  - Utilities like Grep

Experimental Design: Questions

Research questions:

Q1 Does partial evaluated benefits string and matrix-based graph algorithms performance comparing to their basic versions?

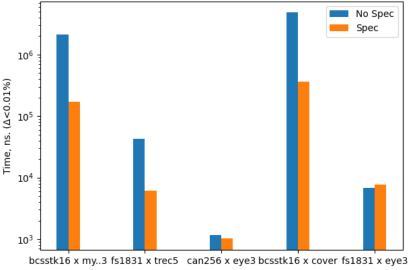
Q2 In which degree partially evaluated algorithms code performance gets closer to their state-of-art implementations?

Experimental Design: Datasets

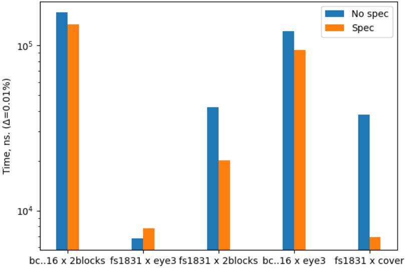
- For matrix algorithms
  - SuiteSparse matrix collection
  - Harwell-Boeing matrix collection
- For algorithms on string
  - Autogenerated strings
  - Traffic dumps
  - Regular expressions from `regexlib.com` catalog
- We chose data with the diverse configuration
  - Tried to cover more basic test cases
  - Used degenerate cases
  - A lot of different $\Longrightarrow$ less threats to validity

Results: Matrices

- Intel i5-7440HQ, 16GB RAM
- (Non)Specialized code in AnyDSL Impala
- Google Benchmark
- Only "interesting" cases are shown

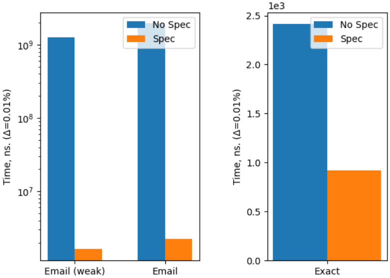


Matrix multiplication

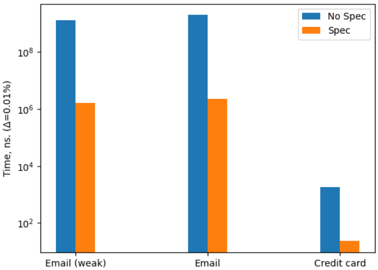


Kronecker product

Results: Strings

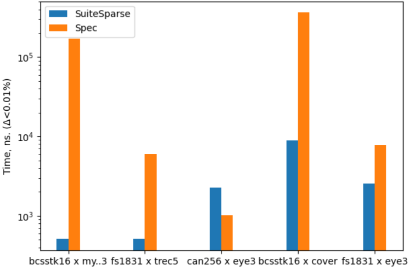


Pattern matching

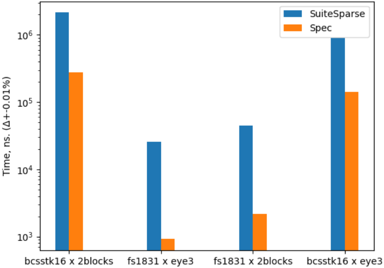


Regular expressions

Comparison with SuiteSparse GraphBLAS

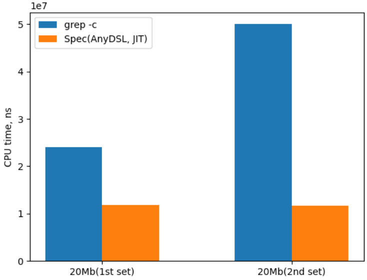


Matrix multplication

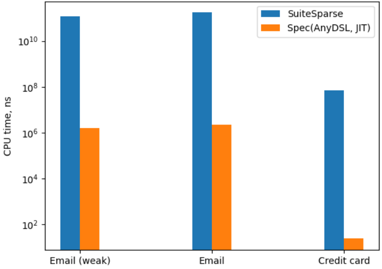


Kronecker product

Comparison with (e)Grep



Pattern matching



Regular expressions

Results: In General

- 10% to 100 times speedup for nearly all tested algorithms comparing to non-partially evaluated code
- 2–5 time to 3 orders win by time in comparison with (e)Grep
- SuiteSparse won 5 times against partially evaluated code
  - Good result for an semi-automatic optimization
  - We still won 10+ times comparing to non-evaluated code
  - The code was optimized on compile-time — no need for a heavy library

Related and Future Work

- Wide set of directions for the future
  - More complex algorithms
  - Another tools & languages (AnyDSL Artic)
  - GPGPU

- Related covers a lot of topics
  - Ray tracing/Bioinformatics/Image processing (AnyDSL team)
  - Viterbi algorithm specialization (Ivan Tyulyandin, SEIM'21)
  - CUDA specialization (Alexey Tyurin, PPoPP'20)

Appendix: Full Matrix List

	Size	Not Null	Symmetry, %	Value Type
bcsstk16	4884	147631	100	real
fs_183_1	183	1069	41.8	real
can_256	256	2916	100	binary
eye3	3	3	100	binary
2blocks	4	8	100	binary
cover	8	12	16.67	binary
mycielskian3	6	5	0	binary
trec5	8	12	0	real

Table: Matrices used in partial evaluation expriments

Appendix: Full Results of Matrix Multiplication Experiments

Time, ns. Spec/NoSpec/SP $\Delta < 0.01\%$	$\times$ *eye3*	$\times$ *2blocks*	$\times$ *cover*	$\times$ *my...3*	$\times$ *trec5*
bcsstk16 $\times$	93608 121855 2270	133434 157850 7064	364772 4842889 8559	171085 2129094 511	308535 5226893 505
fs_183_1 $\times$	7796 6752 2553	20187 42353 12310	6928 38250 9796	1358 15194 506	6078 42493 507
can_256 $\times$	1016 1177 2259	5106 38221 6549	20339 66987 9409	2561 23105 503	9548 62668 506

Table: Execution times for matrix multiplication experiments

Appendix: Full Results of Kronecker Product Experiments

Time, ns. Spec/NoSpec/SP $\Delta < 0.01\%$	$\otimes$ *eye3*	$\otimes$ *2blocks*	$\otimes$ *cover*	$\otimes$ *my...3*	$\otimes$ *trec5*
bcsstk16 $\otimes$	140628 140744 901878	276222 3032308 2145104	433397 4307538 4420688	276433 1967189 2958016	481805 4571625 1440326
fs_183_1 $\otimes$	916 934 25833	2186 21272 45159	3046 31732 88847	1838 14533 35109	3146 34356 47912
can_256 $\otimes$	1159 1069 35162	2772 30841 60600	4512 45731 130084	2736 22079 43479	4576 49512 61500

Table: Execution times for Kronecker product experiments

Appendix: Pure C Comparison