POSTER: Optimizing GPU Programs by Partial Evaluation

Aleksey Tyurin
Saint Petersburg State University
alekseytyurinspb@gmail.com

Daniil Berezun
JetBrains Research
daniil.berezun@jetbrains.com

Semyon Grigorev*
Saint Petersburg State University
s.v.grigoriev@spbu.ru

Abstract

While GPU utilization allows one to speed up computations to the orders of magnitude, memory management remains the bottleneck making it often a challenge to achieve desired performance. Thus, different memory optimizations are leveraged to reduce the number of memory transactions and make memory being used more effectively. In the paper we propose an approach automating memory management utilizing partial evaluation, a program transformation technique that enables the data accesses to be pre-computed, optimized, and embedded into the code, saving memory transactions. As an empirical evaluation of our approach we applied the technique to a straightforward CUDA C naïve string pattern matching algorithm implementation. Our experiments show that the transformed program is up to 8 times as efficient as the original one.

CCS Concepts • **Software and its engineering** → **General programming languages**; • **Social and professional topics** → *History of programming languages*;

Keywords GPU, CUDA, Partial Evaluation

1 Introduction

Performance of GPU-based solutions critically depends on memory management, since most applications tend to be bandwidth bound, requiring the data to be allocated and accessed most effectively. Thereby, memory optimizations appear to be in a prevailing significance and addressed in a huge number of research.

To address issues related to the lack of available GPU memory, sophisticated memory pooling techniques are used leveraging memory swapping and sharing [9]. Further, since GPU memory access latency varies between various memory types, techniques like shared memory register spilling or automatic shared memory allocation are utilized to save global memory transactions and reduce cache contention [6, 8]. However, memory access latency could be aggravated by improper access patterns that entail the increase of the total number of memory transactions making the memory type being used much less effectively. The proposed approach is a runtime memory optimization addressing the mentioned issues that reduces the overall memory transactions number. The approach is based on the following typical feature for GPU-based solutions common workflow.

Suppose we have created an interactive solution for huge data analysis allowing the end-user to sequentially write GPU kernel proccessed queries to a dataset. Let a query be relatively small and data be large, resulting in execution time being signigicant. Simple examples of such scenarios are multiple patterns matching, database querying, convolutional filtering. The GPU kernel should be generic and have at least two parameters: a query and data. But at the moment the user issues a query and the host code is ready to run the kernel the query could be used as a static data for the kernel runtime optimization, since the query remains the same during the execution. Specifically, the kernel could be *partially evaluated* in runtime with respect to the query being issued.

Partial evaluation or *specialization* is a well-known optimization technique that given a program and part of its input data, called *static*, specializes the program with respect to the data, producing another, optimized, program which if given only the remaining part of input data, called *dynamic*, yields the same results as the original program would have produced being executed given both parts of the input data [1, 2].

Application of specialization for one of the described scenarios, namely database querying, has been recently known to significantly improve CPU-based queries execution performance [7]. It appears that partial evaluation is able to achieve similar results for GPU-based applications, optimizing memory accesses. Particularly, considering the problem of multiple patterns matching, where the patterns are *static* during the execution, the result of the memory access for a particular pattern could be evaluated in specialization time and embedded as a value into the code during compilation, rather than being compiled to a load instruction for some memory type. More precisely, partial evaluation lets the values to be accessed thought instructions cache, avoiding

*Also with JetBrains Research.

PPoPP '20, February 22–26, 2020, San Diego, CA, USA

ACM ISBN 978-1-4503-6818-6/20/02.
https://doi.org/10.1145/3332466.3374507

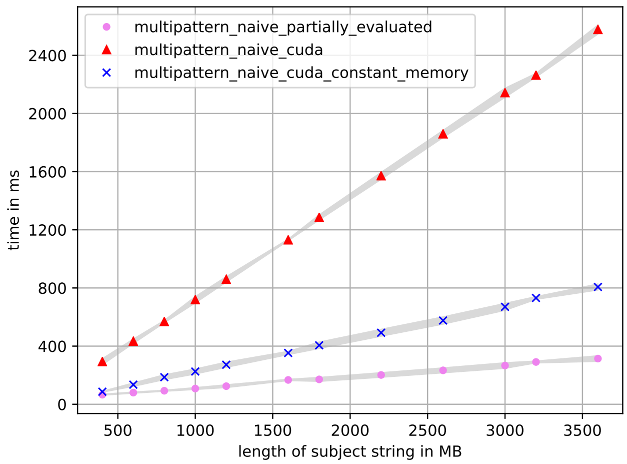

Figure 1. Multiple string pattern matching evaluation

memory load transactions. Thus, in this work we propose to apply partial evaluation for GPU code optimization.

2 Evaluation

The approach has been evaluated considering the problem of *file carving* [5] that stands for extracting files from raw data in a fied of *cyber forensics*. To extract a file, a specific file header should be detected, and the set of relatively short file headers becomes static at the search query executing time. Hence, the query could be specialized with respect to the headers.

The partial evaluator being used is one developed as part of *AnyDSL* framework [4]. Thus, we compare[1] naïve multiple string matching algorithm implementations: one utilizing AnyDSL framework, leveraging partial evaluation with respect to the file headers, and two base-line ones in CUDA C with global and constant memory for header access respectively. All implementations invoke the algorithm in a separate thread for each position in the subject string. The headers are stored as a single char-array and accessed via offsets. The algorithm simply iterates over all headers searching for a match, if it encounters a mismatch, it jumps to the next header forward through the array. Basically, during specialization the partial evaluator performs loop unrolling and evaluates unrolled memory access instructions for the headers.

For the evaluation, the piece of data of 4 GB size has been taken from a hard drive and patterns to be searched have been taken from a taxonomy of file headers specifications [3]. The headers have been divided into groups of size 16 and run over multiple times. The results are presented in Figure 1. The points are the average kernel running time and the grey regions are the area of standard deviation.

[1]The approach has been evaluated on Ubuntu 18.04 system with *Intel Core i7-6700* processor, 8GB of RAM and *Pascal*-based *GeForce GTX 1070* GPU with 8GB device memory.

Since mismatches happen quite often inducing thread divergence, such access pattern hurts coalescing, increasing the overall number of memory transactions. Given that, the performance speedup of a partially evaluated algorithm achieves on a raw data piece of 4 GB size is up to 8 compared to CUDA C version with global memory and up to 3 with constant one as illustrated in Figure 1. Namely, the partially evaluated version spends about 300 ms for searching while global and constant memory CUDA C versions making it in 800 ms and 2500 ms respectively. Note that specialization time is 2 sec, so in case of analysing 1 Tb storage, performance improvement would be significant.

3 Conclusion

In this work, we apply partial evaluation to optimize GPU programs. We show that this optimization technique in the context of file carving problem can improve the performance up to 8 times if compare naïve implementation executed with optimization and without it. Note that optimizations do not require manual manipulation with source code and implementation of additional memory management routines.

The partial evaluator being used assumes the programs to be written with special *DSL*. Nevertheless, partial evaluation could be applied to general-purpose languages, and CUDA C, and the upcoming research is dedicated to the generalization of the technique so as the partial evaluation could be applied at runtime, during GPU-based application execution. Moreover, the performance of partial evaluator should be improved in order to decrease specialization overhead.

Finally, specialization could produce code with a huge number of variables, making the compiler to spill excessive registers. A workaround would be a pipelining of specialization with other optimization techniques, e.g. with shared memory register spilling [6].

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. nnnnnnn and Grant No. mmmmmmm. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

References

[1] Neil D. Jones. 1996. An Introduction to Partial Evaluation. *ACM Comput. Surv.* 28, 3 (1996), 480–503. https://doi.org/10.1145/243439.243447

[2] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[3] Gary Kessler. 2019. GCK'S FILE SIGNATURES TABLE. https://www.garykessler.net/library/file_sigs.html. Accessed: 2019-10-31.

[4] Roland Leissa, Klaas Boesche, Sebastian Hack, Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, André Müller, and Bertil Schmidt. 2018. AnyDSL: A Partial Evaluation Framework for Programming High-performance Libraries. *Proc. ACM Program. Lang.* 2, OOPSLA, Article

119 (Oct. 2018), 30 pages. https://doi.org/10.1145/3276489

[5] Digambar Povar and V. K. Bhadran. 2011. Forensic Data Carving. In *Digital Forensics and Cyber Crime*, Ibrahim Baggili (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 137–148.

[6] Putt Sakdhnagool, Amit Sabne, and Rudolf Eigenmann. 2019. RegDem: Increasing GPU Performance via Shared Memory Register Spilling. *ArXiv* abs/1907.02894 (2019).

[7] Eugene Sharygin, Ruben Buchatskiy, Roman Zhuykov, and Arseny Sher. 2018. Runtime Specialization of PostgreSQL Query Executor. In *Perspectives of System Informatics*, Alexander K. Petrenko and Andrei Voronkov (Eds.). Springer International Publishing, Cham, 375–386.

[8] Xinfeng Xie, Jason Cong, and Yun Liang. 2018. ICCAD : U : Optimizing GPU Shared Memory Allocation in Automated Cto-CUDA Compilation.

[9] Junzhe Zhang, Sai Ho Yeung, Yao Shu, Bingsheng He, and Wei Wang. 2019. Efficient Memory Management for GPU-based Deep Learning Systems. arXiv:cs.DC/1903.06631