POSTER: Optimizing GPU Programs By Partial Evaluation

Aleksey Tyurin
Position1
Department1
Institution1
City1, State1, Country1
first1.last1@inst1.edu

Daniil Berezun
Position2a
Department2a
Institution2a
City2a, State2a, Country2a
first2.last2@inst2a.com

Semyon Grigorev
Associate Professor
Saint Petersburg State University
St. Petersburg, Russia
s.v.grigoriev@spbu.ru
Researcher
JetBrains Research
St. Petersburg, Russia
semyon.grigorev@jetbrains.com

Abstract

While GPU utilization allows one to speed up computations to the orders of magnitude, it is often challenging to achieve maximum performance. Notably, memory optimizations are being the most significant problem: GPUs memory hierarchy implies certain limitations, thus making data memory allocation management nontrivial and memory to be utilized carefully. In the paper we propose to automate data memory management leveraging partial evaluation, a program transformation technique that enables the data to be embedded into the code, optimized, and eventually end up directly in the registers. 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 depend on data allocation and memory management, since most applications tend to be bandwidth bound problem. Thus memory optimizations appear to be in a prevailing significance and addressed in a huge amount of research [7–9]. The GPUs memory access latency varies between different memory types, from hundreds of cycles for global memory to just a few for shared and register memory. Moreover, the latency could be aggravated by wrong access patterns or misaligned accesses and the possibility of proper access patterns could depend on the domain of the problem being solved. For example, the global memory access pattern could be not clear, thus preventing GPU from efficient coalescing. It imposes a burden of memory management to a programmer or makes one rely on caching mechanisms.

In order to achieve the fastest memory access constant, shared or registers memory should be utilized. However, constant memory lacks flexibility in the sense that the size of data should be known beforehand and access pattern also should be kept in mind. Shared memory should be used carefully due to considerations of synchronization and bank conflicts, while register allocation is managed by the compiler and explicit storing of data to them is difficult. E.g. small arrays could be stored in registers, but only if the compiler is able to figure out that arrays indexing is static and if it does not, the array would end up in local memory. Moreover, all these approaches require to create a special nontrivial code for manual data allocation management which makes algorithm implementation harder. Finally, such optimizations require specific knowledge from developers. One of the ways to solve these problems is to automate memory management.

At the same time, a common workflow of a GPU-based solution has a feature that allows one to introduce runtime optimizations which originally are static. Suppose the next scenario. We have created an interactive solution for huge data analysis. The user can sequentially write queries to a dataset, and GPU kernel is used for query processing. Suppose that query is relatively small (compared to data), data is huge and thus query execution time is significant. Simple examples of such scenarios are multiple pattern matching, database querying, convolutional filters applying. The kernel should be generic and have at least two parameters: query and data. But at the moment, when the user specifies a query and the host code is ready to run GPU kernel, the query could be used as a static data for the kernel optimization.

PPoPP '20, February 22–26, 2020, San Diego, CA, USA
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-6818-6/20/02.
https://doi.org/10.1145/3332466.3374507

There is a known program optimization technique that optimizes a given program with respect to statically known inputs, producing another, optimized, program which if given only the remaining dynamic inputs yields the same results as the initial one being executed given both inputs. The technique is *partial evaluation* (aka program specialization) [1, 2]. Basically, given a function f of n arguments with some of them being static, denoted with k, partial evaluator evaluates or *specializes* those parts of the function depending only on static arguments, producing a residual function f' of $(n - k)$ arguments. Thus it produces a more optimal function in a sense that the specialized function being invoked makes fewer computations than the original one.

Application of specialization for one of the described scenarios, namely database querying, have been known to significantly improve query execution performance [7]. Given that regarding GPU memory management partial evaluation is able to produce an optimization for memory access, could similar results be achieved for GPU-based applications? Particularly, considering the problem of multiple patterns matching with a known set of patterns, the result of memory access for a particular pattern could be embedded into the code during compilation, rather than being compiled to load instructions for different memory spaces. More precisely, partial evaluation results in data being accessed through instruction cache. In this work, we propose to apply *partial evaluation* for GPU code optimization.

2 Evaluation

As an instance of a problem, consider the *file carving* [5], in a field of *cyber forensics* it stands for extracting files from raw data, i.e. from lost clusters, unallocated clusters and slack space of the disk or digital media. To extract a file, we should detect a specific file header: for a predefined set of file types, the headers to search for are known beforehand and commonly are relatively short.

The partial evaluator being used is one developed as part of *AnyDSL* framework [4]. So, we compare AnyDSL framework implementation leveraging partial evaluation with respect to the file headers against two base-line implementations 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.

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.

To estimate the performance gain brought by partial evaluation the problem of file carving has been evaluated. For the evaluation, the piece of data of 4 GB size has been taken

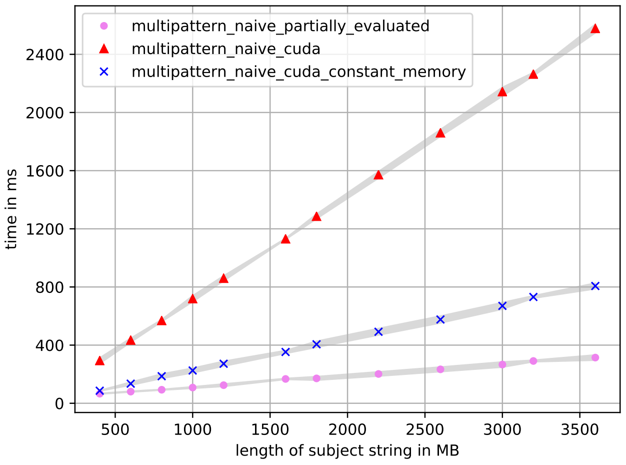

Figure 1. Multiple string pattern matching evaluation

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.

Since the headers could be lengthy and mismatches happen quite often, such an 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 stuff for memory management.

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.

Specialization can produce code with a huge number of variables, and it can make register management harder. Could our partial evaluation be combined with advanced register spilling techniques (for example with [6]) to improve performance?

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