Optimizing GPU Programs By Partial Evaluation

Anonymous Author(s)

Abstract

And while this approach allows speed ups to the orders of magnitude, it is often challenging to achieve maximum performance. Also, while memory optimizations are being the most significant ones, GPUs memory hierarchy implies certain limitations, thus memory should be utilized carefully. Generally, on-chip data access is to be preferred over global one.Add here the use-case/problem This paper proposes the idea of leveraging static[1] data memory management, using partial evaluation, a program transformation technique that enables the data to be embedded into the code and eventually end up directly in the registers. Generalization to runtime detection of static? An empirical evaluation of a straightforward string pattern matching algorithm implementation utilizing this technique is provided. Our approach achieves up to 6x performance gain compared to a straightforward naive CUDA C implementation.

Keywords GPU, CUDA, Partial Evaluation

1 Introduction

Performance of GPU-based solutions are critically depending on data allocation and memory management: most applications tending to bandwidth bound problem. Thus memory optimizations appear to be in a prevailing significance and addressed in huge number of research [5–7]. 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, 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 make one to 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 a 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. Finally, all these ways require to create a special nontrivial code for manual data allocation management which makes algorithms implementation harder. Moreover, such optimizations require specific knowledge from developers. One of the way to solve these problems is to automate memory management.

At the same time, the workflow of a GPU-based solution has a feature that allows one to introduce runtime optimizations which originally was 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 (in comparison with data), data is huge and thus query execution time is significant. Simple examples of such scenarios are a multiple pattern matching, database querying, convolutional filters applying. The kernel should be generic and has at least two parameters: query and data. But at the moment, when the user specifies query and host code is ready to run GPU kernel we can use the query as a static data for the kernel optimization.

There is a known program optimization technique that optimizes a given program with respect to statically known inputs, producing another program which if given only the remaining dynamic inputs will produce the same results as initial one would have produced, given both inputs. The technique is *partial evaluation*, a program transformation optimization technique that emphasizes on full automation [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 gives a more optimal function in a sense that the function needs less computations when being actually invoked.

Regarding GPU memory management partial evaluation is able to produce an optimization for memory access. Considering the problem of file carving with a known set of file headers, the result of memory access for a particular header 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. We propose to !!!!!

2 Evaluation

As an instance of problem, consider the *file carving* [4], in a field of *cyber forensics* it stands for extracting files from raw data, i.e. from lost clusters, unallocated clusters and slack

[1] Compile-time known data or data that can be determined not to be changing during runtime

PL'18, January 01–03, 2018, New York, NY, USA
2018.

space of the disk or digital media. To extract a file, we should to detect a specific file header: for a predefined set of types of files 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 [3]. 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.

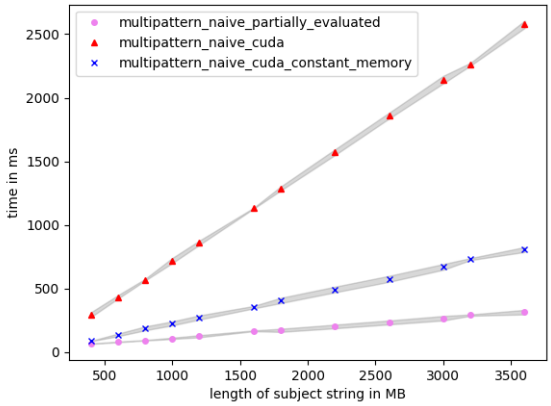

Figure 1. Multiple string pattern matching evaluation

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 from a hard drive and patterns to be searched have been taken from a taxonomy of file headers specifications [1]. The headers have been divided to groups of size 16 and run over multiple times. The results are presented in 1. The points are the average kernel running time and the gray regions are the area of standard deviation.

Since the headers could be lengthy and mismatches happen quite often, such access pattern hurts coalescing, increasing the overall number of memory transactions. Given that, the performance speed up partially evaluated algorithm achieves on a raw data piece of 4 GB size is up to $8x$ compared to CUDA C version with global memory and up to $3x$ with constant one as illustrated in 1. Namely, 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.

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 performance up to $8x$ times if compare naive implementation executed with optimization and without it. Note that optimizations do not require manual manipulation with source code.

The partial evaluator being used assumes the programs to be written with special *DSL* and up to the current level of progress requires the array-like data to be passed inplace for the *JIT* compiler to be able to extract information from it. Nevertheless, this could be rectified with symbolic computation? and the upcoming research is dedicated to the generalization of the technique so as the partial evaluation could be applied at runtime, i.e. during kernel execution.

Specialization can produce code with huge number of constants, and it can make register management harder. Can we combine our solution with advanced register spilling techniques (for example with [5])?

References

[1] [n. d.]. GCK'S FILE SIGNATURES TABLE. https://www.garykessler.net/library/file_sigs.html. Accessed: 2019-10-31.

[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] 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

[4] Bhadran V.K. Povar D. 2010. Forensic Data Carving. *Digital Forensics and Cyber Crime* (2010). https://doi.org/10.1007/978-3-642-19513-6_12

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

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

[7] 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