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

GPUs performance could be optimized via memory, instructions and configuration. Any statistics that memory optimizations are needed more often? However, with most applications tending to be bandwidth bound, memory optimizations appear to be in a prevailing significance. 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 order 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 order also should be kept in mind. Shared memory is to 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 to 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.

Constant memory is rather generic in a sense it only requires the data to be read only and does not make a distinction whether the data is statically or dynamically known. However, there are applications, where some pieces of data are known statically. For instance, consider a problem of *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 space of the disk or digital media. To extract a file, the file itself should provide a header, thus for a predefined set of files the headers to search for are known beforehand and commonly are relatively short.

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.

The partial evaluator being used is one developed as part of *AnyDSL* framework [3]. For the fiel carving probleam the partially evaluated version of algorithm achieves up to 6x better performance compared to straightforward implementations with *CUDA C*.

2 Performance Evaluation

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

[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.

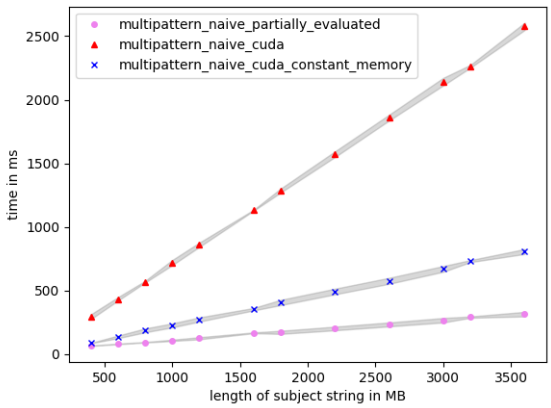

Figure 1. Multiple string pattern matching evaluation

from a hard drive and patterns to be searched have been taken from a taxonomy of file headers specifications [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.

The evaluation compares 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. 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

The work proposes the idea of applying partial evaluation to optimize GPU programs. The program transformation moves data from arrays directly into the code, thus enhancing performance due to memory access transactions number reduction. In the context of file carving problem evaluation it has achieved up to $8x$ performance gain.

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.

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