Optimizing GPU Programs By Partial Evaluation*

Subtitle†

Anonymous Author(s)

Abstract

GPUs have been known to be used to facilitate computationally-intensive problems solving. 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. 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.

Given that caches tend to undergo misses and random access could hurt coalescing or broadcasting, we exploit partial evaluation techniques to specialize a GPU program on static data in such a way to move the data straight into the code rather than to any memory space.

Partial evaluation is a program transformation optimization technique that emphasizes on full automation.[1] 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 of memory access. Consider the snippet of code from **Listing 1**. Suppose the array named *template* and its size *template_size* are statically known. The array *ibuffer* is dynamic. Given that memory access indexing and the content of the array are static, the snippet could be transformed to *.ptx* instructions during the compilation as presented in **Listing 2**. The thing is the content of *template* array has been embedded into the comparison instructions rather than to be globally loaded multiple times as in **Listing 3**.

The partial evaluator being used is one developed as part of *AnyDSL* framework. The specialized achieves got up to

```
/*
*   assume  template is ['\x49','\x44',...]
*/

 for i in unroll(0,template_size) {
    //global/constant memory access
    if ibuffer(t_id + i as i64) != template(i) {
        //... Do something ... //
    }
 }
```

Listing 1. Partial evaluation example

*Title note
†Subtitle note
[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.

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 searching multiple string occurrences in some subject string has been considered. The problem arises in many areas like computer security or bioinformatics where the patterns are often known beforehand, rather small and to be searched in huge arrays of data.

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. The patterns have been divided to 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 gray regions are the area of standard deviation.

The evaluation compares AnyDSL framework implementation leveraging partial evaluation with respect to the patterns against two base-line implementations in CUDA C with global and constant memory for patterns storing respectively. All implementations invoke the algorithm in a separate thread for each position in the subject string.

The patterns are stored as a single char-array and accessed via offsets. The algorithm simply iterates over all patterns searching for a match, if it encounters a mismatch, it jumps to the next pattern forward through the array. Since the patterns 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 is up to 6x compared to CUDA C version with global memory and up to 3x with constant one as illustrated in **Figure 1**.

```
    ...
LDG.E.U8 R0, [R4] ; //load from global memory, i.
    e. ibuffer(...)
BFE R6, R0, 0x1000 ;
ISETP.NE.AND P0, PT, R6, 0x49, PT ; //0x49 got
    extracted, so we have avoided global memory
    access!
@P0 SYNC ;
LDG.E.U8 R0, [R4+0x1] ;
BFE R0, R0, 0x1000 ;
ISETP.NE.AND P0, PT, R0, 0x44, PT ; //extracted
    again!
    //and so on
    ...
```

Listing 2. Code after partial evaluation

```
LDG.E.U8 R8, [R8] //load global
LDG.E.U8 R6, [R6] ; // load global
BFE R13, R8, 0x1000 ;
BFE R12, R6, 0x1000 ;
ISETP.NE.AND P0, PT, R13, R12, PT ; //compare pre
    -loaded registers
```

Listing 3. Code without partial evaluation

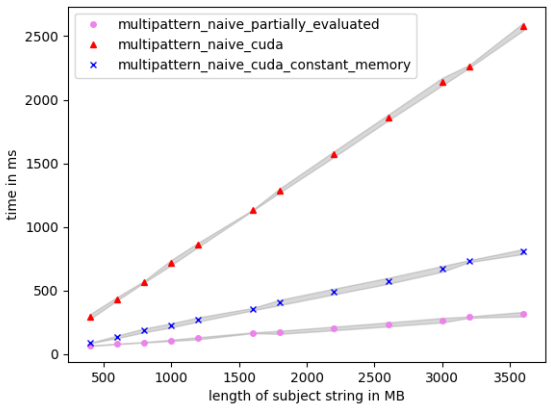

Figure 1. Multiple string pattern matching evaluation.

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. It results in a noticeable performance gain but not always.

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 *JIT* compiler to be able to extract information from it. + maybe add a remark that PE could enhance programs asymptotically. Nevertheless, this could be rectified with symbolic computation? and we strongly believe that GPU specialization is possible in runtime, i.e. during kernel execution.

References

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