Санкт-Петербургский государственный университет

Программная инженерия

Системное программирование

Тюрин Алексей Валерьевич

Экспериментальное исследование специализатора GPGPU-программ AnyDSL

Выпускная квалификационная работа бакалавра

Научный руководитель:
к. ф.-м. н., доцент С. В. Григорьев

Консультант:
программист ООО «Интеллиджей Лабс»
к. ф.-м. н., Д. А. Березун

Рецензент:
стажёр-исследователь «ИСП им. В. П. Иванникова РАН»
Е. Ю. Шарыгин

Санкт-Петербург
2020

SAINT PETERSBURG STATE UNIVERSITY

Software Engineering

Aleksey Tyurin

Practical study of AnyDSL GPGPU program partial evaluator

Bachelor's Thesis

Scientific supervisor:
Associate professor, Ph. D. Semyon Grigorev

Adviser:
IntelliJ Labs Co. Ltd developer
Ph. D. Daniil Berezun

Reviewer:
Research assistant at Ivannikov Institute for System Programming of the RAS
Eugene Sharygin

Saint Petersburg
2020

Contents

Introduction **4**

Problem statement **6**

1. Related work & background **7**

1.1. Nvidia CUDA 7
1.2. Memory optimizations 10
1.3. Partial evaluation 11
1.4. AnyDSL framework 15

2. Experimental setup **18**

2.1. Runtime partial evaluation 18

3. Evaluation **20**

3.1. String matching 20
3.1.1. Naìve single substring matching 20
3.1.2. Data embedding 22
3.1.3. Naìve multiple substring matching 25
3.1.4. Aho-Corasick matching 26
3.2. Convolutional filtering 28
3.2.1. Separable convolution 29
3.3. Discussion 30

Conclusion **31**

References **32**

Introduction

In the era of big data and high load computations, graphical processing units (GPUs) are used extensively for data processing. There have even been designed embedded devices with the support of GPUs for general purpose computations, like image recognition on mobile robots [17]. However, since many GPU-based applications tend to be bandwidth bound, meaning that data allocation and access are the main bottlenecks, memory optimizations appear to be in a prevailing significance and are addressed in a huge amount of research.

While the problem of available GPU memory could be tackled with sophisticated memory pooling techniques through memory swapping and sharing [7], memory architecture of a GPU often requires more sophisticated optimizations. Given that a typical GPU incorporates several memory types, each having different capacity and throughput, a few memory management automation techniques have been introduced. They could leverage more effective memory spaces to handle register spilling and systematically consider the performance benefit achievable through a specific allocation of shared memory to save global memory transactions [25, 21]. Further, the diversity of memory types imposes that each memory access should satisfy memory type specific patterns, to be most effective. Under the non-fulfillment of these patterns the data throughput of an application is aggravated due to the increase in the number of required memory transactions. Considering the following typical scenario of a GPU-accelerated application, another runtime memory optimization could be proposed.

To facilitate the data processing a GPU routine is executed by multiple threads simultaneously with different pieces of data, often exceeding the maximal number of threads that could be executed simultaneously resulting in blocks of threads being executed iteratively. Being that the input data often exceeds the available GPU memory, the routine could not be applied at once, and there is a need to split the data into chunks and process them iteratively by the routine. It happens that some relatively small properties within a processing routine are maintained between the iterations and could be considered static in that sense. For example, if the application is a GPU-accelerated data processing engine that allows one to write queries to data and typically these queries remain small if compared to data and take significant execution time, the query, once specified by the user, can be used as a static, i.e. already known and constant, data for the query execution kernel runtime optimization since it remains unchanged during the host code execution. This observation allows to apply a *partial evaluation* technique to optimize such routines in runtime.

Partial evaluation or program *specialization* [11, 10] is a program transformation and 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. Basically, a partial evaluator performs an aggressive unfolding/unrolling, inlining, and constant propagation. The application of partial evaluation at runtime has recently shown a significant performance improvement of query execution for CPU-based database querying [20].

Regarding memory optimizations of GPU kernels, partial evaluation is able to embed static data memory accesses into the code, i.e. place it directly into registers, once a kernel is properly written, which could result in a better performance compared to non-embedded access since memory transactions would be replaced by accesses to the instruction cache. Thus, the aim of this work is to provide an empirical evaluation of an existing partial evaluator AnyDSL [4] that is capable of producing CUDA [1] code for NVIDIA GPUs to investigate the effects that appear after partially evaluating GPU applications and what aspects of GPU architecture affect the desired result and whether any significant performance improvements could be achieved at all.

[1] Programming and hardware model by NVIDIA.

Problem statement

The aim of the work is the practical evaluation of whether any performance enhancement could be brought by partially evaluating memory accesses through the utilization of AnyDSL framework, compared to CUDA implementations, considering GPU microarchitecture details that affect the result. In order to achieve the aim, the following objectives have been set.

- Implement experimental scenarios in both AnyDSL and CUDA.
- Collect relevant datasets for the evaluation to be more practical.
- Perform the evaluation and analyze the results.

More specifically, the work performs the evaluation on string matching and convolutional filtering scenarios, providing some relevant CUDA assembly examples to ground the effects being observed.

1 Related work & background

This section gives the necessary insights about a typical GPU architecture, performance-related considerations, and differences between compute capabilities from CUDA perspective, as well as known approaches for memory optimizations, while also providing the insights about partial evaluation and its known practical applications.

1.1 Nvidia CUDA

Modern GPUs are highly parallel computational devices equipped with a very high-bandwidth memory, designed to speed up general-purpose computations. CUDA defines a specific programming model[2] and architecture for NVIDIA GPUs. These details are provided considering Nvidia Tesla T4 GPU.

Hardware architecture Nvidia Tesla T4[3] GPU is of Turing architecture and constitutes of five Graphics Processing Clusters (i.e. self-contained GPUs), each including four Texture Processing Clusters that incorporates two Streaming Multiprocessors (SM) each. Each SM is built up of four processing blocks, each including a warp[4] scheduler, dispatch unit, and units for a memory fetch, integer, and floating-point operations, latter being called CUDA-cores. The GPU includes 2560 of such cores. Further, Tesla is supplied with 16GB GDDR6 memory that supports throughput up to 320GB/s. Unlike previous architectures it incorporates shared memory in L1 cache, increasing the bandwidth up to 2x, and adds an independent integer datapath, enabling concurrent execution of integer and floating-point operations. Finally, Tensor Cores has been introduced, being specialized execution units specifically for performing the tensor/matrix operations.

[2] `https://docs.nvidia.com/pdf/CUDA_C_Programming_Guide.pdf` (last accessed date: 30.05.2020)

[3] `https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf` (last accessed date: 30.05.2020)

[4] A batch of 32 CUDA threads

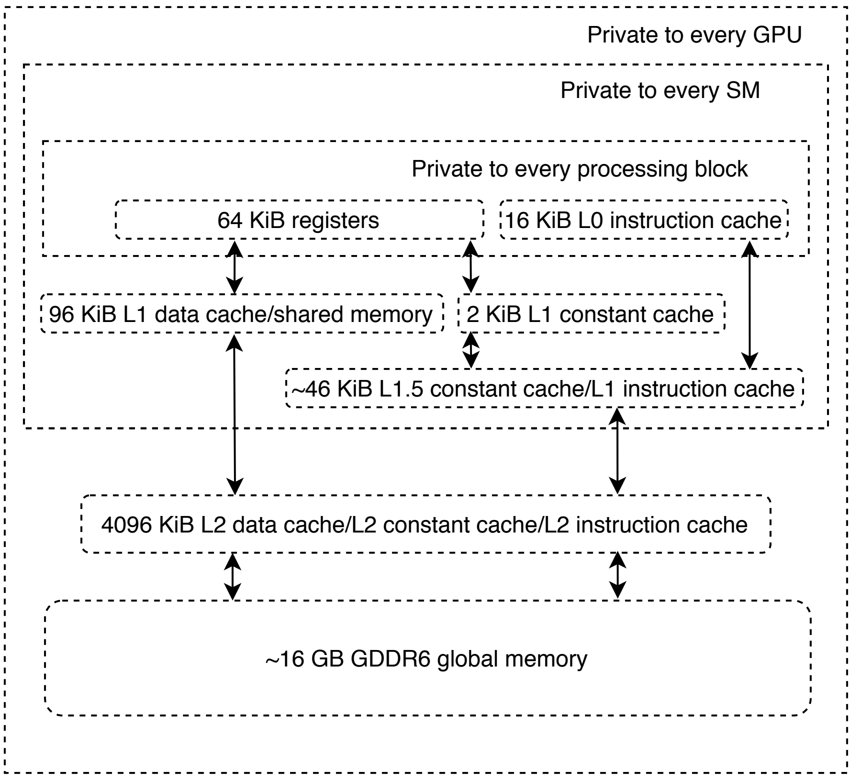
Private to every GPU
Private to every SM
Private to every processing block
64 KiB registers
16 KiB L0 instruction cache
96 KiB L1 data cache/shared memory
2 KiB L1 constant cache
~46 KiB L1.5 constant cache/L1 instruction cache
4096 KiB L2 data cache/L2 constant cache/L2 instruction cache
~16 GB GDDR6 global memory

Figure 1: Nvidia Tesla T4 memory hierarchy [6]

Programing model CUDA implies Single Instruction Multiple Threads architecture, where each instruction is concurrently executed by multiple threads, that are combined in blocks, that populates a grid. Threads in a block are split into warps, which are distributed between warp schedulers on a single SM, such schedulers assign per-thread instructions to the available computation units. Hence, in general, threads in a warp should execute the same instruction to achieve the best performance, in case of different instructions, caused e.g. by an if-statement, the execution is serialized, this is called a *thread-divergence.* A piece of a program that is intended to be executed on a GPU is called a device kernel and usually is implemented in CUDA C, where grid and block sizes also could be specified.

Memory hierarchy Any GPU incorporates several memory types, that serve different purposes and differ in access latency, size, and bandwidth. The memory architecture of Tesla T4 is depicted in 1.

Global memory is the main resource to transfer data between host and device. It is the largest and the slowest memory space, cacheable in L1 and

L2 caches, that have 32 B and 64 B lines respectively. Global memory loads and stores by threads of a warp are served by the device with 32 B transactions[5]. Basically, concurrent accesses of the threads in a warp will coalesce into a number of such transactions necessary to service all the threads in the warp. So in order to keep the number of such transactions to a minimum, threads in a warp should access adjacent segments of memory, aligned with the transaction size, avoiding strided accesses. Such a requirement could not always be satisfied in practice, thus GPU resources being used not to its maximum. This memory space is accessible and allocated from the host, and visible to all the threads in a grid.

Shared memory is on-chip memory (hence it has lower latency and higher bandwidth than global memory) used to optimize frequent accesses to the same elements in global memory. It is fast as long as bank conflicts do not occur. The whole shared memory space is divided into 32 banks — 4 B memory elements, with successive 4 B words belonging to successive banks. Any bank has a bandwidth of 4 B per clock cycle. Therefore, any memory load or store of n addresses, belonging to n distinct memory banks could be serviced simultaneously with the bandwidth of n times the bandwidth of a single bank. However, accesses to memory addresses from the same bank are serialized, decreasing the effective bandwidth in a magnitude of the number of conflict-free accesses. When multiple threads of a warp access the same shared memory, a broadcast occurs. Several such broadcasts could coalesce into a single multicast. This memory space has a visibility scope of a single thread block and is not accessible from the host.

Constant memory is a cacheable off-chip memory, that is read-only from the device. It is a 64 KB part of global memory, that has a specific path for caching. As a result, on a cache miss, a constant memory read costs one memory read from global memory; otherwise, it is as fast as register access. When the threads in a warp access different constant memory addresses, these accesses are serialized. Thus the constant cache is best when the threads in warp access the same or very few distinct addresses, in the first

[5]`https://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf` (last accessed date: 30.05.2020)

case the access is broadcasted, and the latency is the same as registers one.

Register memory is on-chip memory, consuming zero extra clock cycles per instruction, as long as there is no register read-after-write dependencies and bank conflicts. Is has a scope of a single thread and is the fastest memory type available.

1.2 Memory optimizations

Given such a sophisticated memory hierarchy that should be managed by a programmer, memory optimizations are in a great interest. Only automatic memory optimizations would be considered, avoiding the guidelines about how to rearrange existing access patterns and speed-up the transferring between host and device.

In cases when abundant data parallelism is hard to achieve, e.g. when utilizing concurrent data structures and related synchronization primitives that arbitrate the accesses, dynamic memory allocation, i.e. per-thread allocation inside the device code, for such data structures becomes a bottleneck, hurting the throughput. In [9] a novel approach for dynamic memory allocation is proposed. It is based on shared data structures, supporting fine-grained mutual exclusion regions, cooperative synchronization primitives allowing to allocate memory concurrently between the threads, and a policy of execution delegation. The proposed allocator has a much higher allocation throughput compared to the one, deployed with CUDA.

Since the register memory is bounded in size, excessive registers[6] could be spilled into the global memory, introducing higher latency. Contrary to the default CUDA compiler approach, that handles excessive registers via re-materialization[7] as long as possible before spilling, an approach from [21] advocates the usage of underutilized shared memory to spill the excessive registers. The approach is implemented as an extra binary translation pass over CUDA assembly. Under certain conditions, such optimization is worse than CUDA default approach, thus the authors also provide a compile-time

[6]When the compiler needs more registers, than available on SM.

[7]Recomputation of a value instead of loading and storing. It produces the code with lower efficiency, however with lower register usage as well.

performance gain estimator, based on collected either explicit or implicit[8] instruction stalls, to compare the default code produced by the compiler and an optimized one, to decide which one is better. On average, this technique achieves 10% better performance on the selected benchmarks.

Another work is focused at the automatic allocation of shared memory to reduce the number of global memory transactions [25] for automatically C-to-CUDA generated programs. Authors propose a performance model, designed to choose the best configuration for shared memory allocation, considering estimated execution time, memory metrics, and the number of reduced global memory transactions. Once the best configuration is found, it is applied to the original C code by adding specific OpenACC[9] pragmas.

To tackle the problem of the lack of available GPU memory, a domain-oriented memory pooling and swapping is proposed in [7]. Variables not in use are swapped to host and swapped back before any access, while variables that have non-overlapping lifetime could be allocated to the same memory space by the heuristic-driven memory pool. The heuristic is based on the iterative nature of the deep learning training algorithms to derive the lifetime and read/write order of all variables.

Despite such a variety of different memory optimization approaches, none of the presented works exploits possible static nature of device kernel parameters in any way. Thus, the application of partial evaluation technique to GPU kernels could provide another optimization approach, oriented at static data management.

1.3 Partial evaluation

Partial evaluation is a program transformation and optimization technique, also known as program *specialization*, first formulated in [13]. A partial evaluator is an algorithm, that takes a program and some of its known inputs called *static* and produces another residual program, one yielding the same result given the remaining *dynamic* inputs as the original

[8]Some stalls are labeled by the compiler, others are deduced, based e.g. on latencies for the memory type being accessed by the instruction.

[9]A set of directives for heterogeneous code parallelization

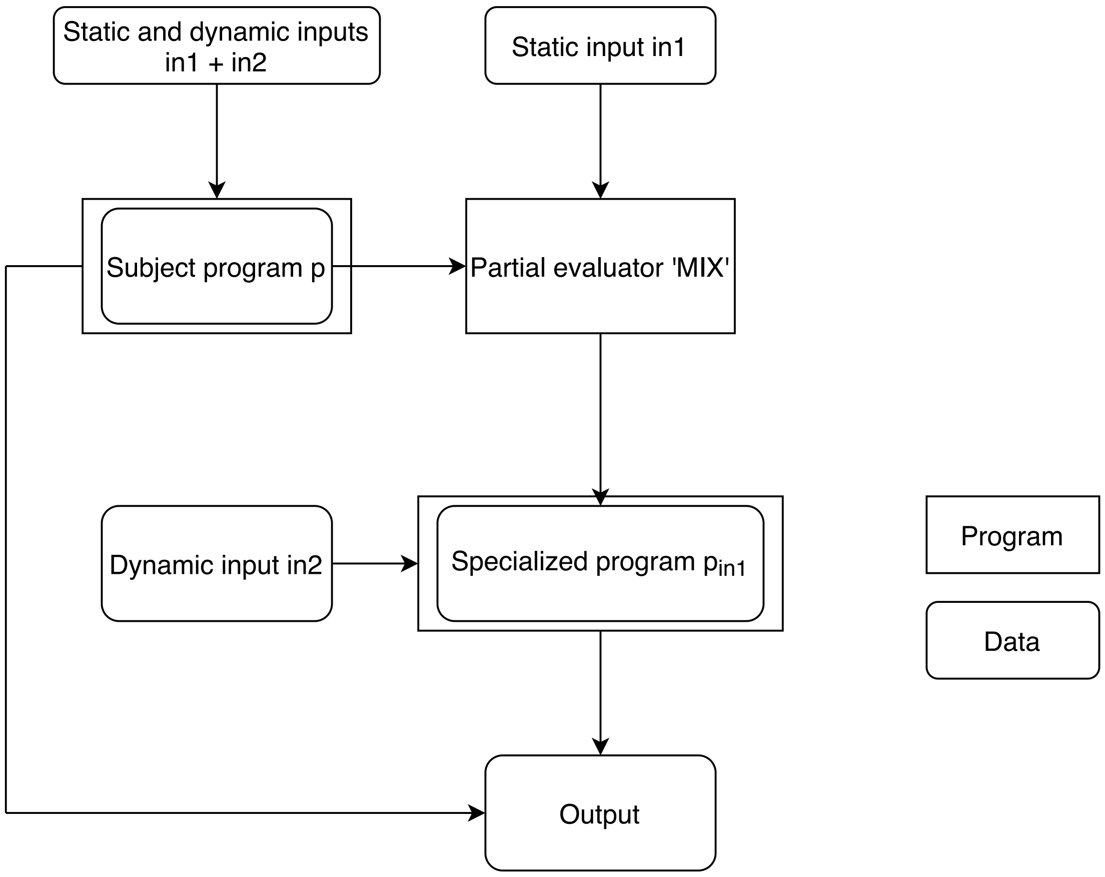

Figure 2: Partial evaluation pipeline

program would have produced given all the inputs at once as illustrated in commutative diagram 2. A partial evaluator performs a mixture of code generation and execution: it reduces those parts of program **p**, depending on **in1**, and generates code for calculations for yet unavailable **in2**. Generally, it performs symbolic computations, unfolding function calls, and replacement of function calls to their specialized versions. For example, a partial evaluator for the program in listing 1 reduces the number of static conditional statements driven by static parameter.

The key idea is that a specialized program requires fewer computations (through moving them to compile or specialization time), and thus is intended to be more effective. A notable feature of specialization is the ability to generate a compiled program, once given an interpreter and a source program, also known as the first Futamura projection [8]. And while many practical applications benefit from partial evaluation through reducing the interpretation overhead (either by generating a compiler or exploiting interpretative nature of a program), there are scenarios, when a program takes more than one input, and one of the inputs varies more slowly than the others, thus specialization with respect to less-variable input could produce a more effective program. Significant speed-up results were reported for pattern recognition, ray tracing, database querying, and scientific computing [11].

```
int pow(int x,int n){
    if (n == 0) return 1;
    else if (n \% 2 == 0) {
        return pow2(f(x,n/2));
    }
    else return x * f(x,n-1);
}
//partially evaluating pow(x,5) would yield:
int pow(int x){
    return x * pow2(pow2(x));
}
```

Listing 1: Pow function partial evaluation example

In [22] an annotation-driven partial evaluator for Java programming language is introduced. The goal is to specialize framework configurations and quite expensive reflection calls, which are widespread in object-oriented systems. The replacement of reflective method calls with partially evaluated ordinary calls allows to speed-up a dynamic pricing system by a power of six.

In [16] an approach for automatic virtual machine construction is proposed, which allows new languages to be implemented through the specification of abstract syntax tree interpreter. Partial evaluation is used to compile hot and stable parts of the already specialized[10] interpreter.

Another usage of partial evaluation as a means for compilation has been described in [20]. The PostgreSQL query interpreter, compiled at first to LLVM IR, which the designed partial evaluator works with, has been specialized in runtime with respect to the query, i.e. the query is compiled. The approach results in rather significant performance improvement, requires fewer efforts compared to just-in-time compiler development and is

[10]Supplied with runtime information such as types.

fully automatic.

Despite such a broad variety of partial evaluation- related research, at the moment of writing, there is a lack of research combining GPGPU and partial evaluation. In [12] the approach from [16] was adopted to compile code from *R* language to *OpenCL*, to make it possible to write GPU-accelerated applications with dynamic interpreted languages, popular in big data, ignoring third-party libraries and relatively low-level programming languages for GPU. A metaprogramming system for writing shaders is presented in [15], which supports partial evaluation via currying. However, the system is rather outdated and has no relation to modern GPGPU programming. Furthermore, there are no works dedicated to what benefits partial evaluation could provide once applied directly to GPU program, or whether it could provide any at all.

```
handleData (filterParams, data)
{
  res = new List()
  for d in data
     for e in filterParams
        if d % e == 0
        then res.Add(d)
  return res
}
```

Listing 2: A typical GPGPU kernel

```
handleData (data)
{
  res = new List()
  for d in data
    if d % 2 == 0 ||
       d % 3 == 0
    then res.Add(d)
  return res
}
```

Listing 3: A typical GPGPU kernel specialization with respect to [2;3]

However, in practice, it appears that many GPGPU scenarios could have static parameters in a sense, theoretically appropriate for specialization. A typical GPGPU kernel is depicted in listing 2. It often represents some kind of a filter, that is applied to different pieces of the data in parallel by a huge number of threads. The execution time primarily depends on the size of the data, which often exceeds the available memory of GPU, therefore the device kernel is run multiple times on different chunks of the data, resulting in significant execution time. Hence, the filter varies less frequently than the data and could be considered as a static input and subjected to specialization. However, the filter becomes known at runtime, before the kernel actually runs, thus the specialization should be performed at runtime. The overhead of the partial evaluation could be hidden by the gained speed-up across the long run of the kernel. Moreover, since one filter could be applied to different data pieces, the specialized version could be cached and reused instead of specializing again. The effect of specialization could be seen in listing 3. The inner cycle with accesses to the memory space of the filter has been reduced with the filter parameters being placed directly into the instructions. Since memory access operations have been proven to be the most expensive, such a replacement could provide a benefit of accessing the filter from the registers, which is the fastest memory type, rather than from any other memory space. Furthermore, a partial evaluator could be able to reduce those parts of the kernel, that depend on static filter parameters made available.

1.4 AnyDSL framework

To the date of writing, there is no partial evaluator known that works directly with CUDA C or any of CUDA intermediate representations. As mentioned, the partial evaluator from [20] works with LLVM IR, and CUDA has an appropriate LLVM frontend, however, the partial evaluator leverages special attributes in IR, that conflicts with the ones of CUDA itself during LLVM JIT compilation. AnyDSL [4] is a framework for the development of domain-specific libraries that could utilize different backends, including the one of CUDA. The framework includes a partial evaluator that works with a specific intermediate representation, supporting CUDA C code generation. The framework requires the programs to be developed in a special DSL named *Impala*, which resembles C language with functions being the first-class objects. Since the DSL is very CUDA C alike, the framework has been chosen as a means for partial evaluation of GPU programs.

The impala compiler translates the code into a functional graph-based intermediate representation similar to typed lambda calculus with continuations. Such a representation allows Impala to achieve near C performance, despite higher-order functions support [14]. Partial evaluation is performed on this level, while the representation could target different hardware architectures utilizing LLVM and compiler intrinsics. An example of such an intrinsic is given in listing 4. The whole function would be first converted into the intermediate representation, with the parts of the function labeled with `@` partially evaluated, then the device-independent parts of the code would be translated into LLVM and device code would be translated into device-specific code, e.g. the code between lines 4-8 would be translated into CUDA C, which then would be included in LLVM code with a call to an external function that loads and executes the generated device code, e.g. the external function for mentioned lines would call NVIDIA CUDA compiler in runtime and launch the compiled kernel.

```
1 fn iterate(fld: Field , @body: fn(int, int) -> ()) -> () {
2     let grid = (fld.cols , fld.rows , 1);
3     let block = (128 , 1, 1);
4     cuda(grid , block , || {
5         let x = tid_x() + blockDim_x() * blockId_x();
6         let y = tid_y() + blockDim_y () * blockId_y ();
7         body(x, y);
8     });
9 }
```

Listing 4: Impala GPU-accelerated loop

The authors of the framework evaluated the effectiveness of partial evaluation targeting different backends, including CUDA [4, 23]. The results show similar performance with hand-tuned third-party implementations. However, the authors focused on compile-time partial evaluation of kernels and have not provided any GPU-specific details that affect the success of the result as well as what particular optimization a partial evaluator performs with the device code. In contrast, this work is aimed at revealing GPU architecture details that affect the success of partial evaluation, focused at runtime partial evaluation and selects different experimental scenarios.

2 Experimental setup

In this section, the scheme of runtime partial evaluation is presented as well as the evaluation configuration.

2.1 Runtime partial evaluation

In practice, it is infeasible to compile a new kernel for each static input value, which is often known in runtime. Thus the partial evaluation of the kernel should be performed in runtime as well as the kernel compilation to a specific GPU target. Each specialized benchmark scenario corresponds to the sequence in figure 3. The device kernel in Impala is included in the target application using `xxd` tool during the compilation. When static data becomes known at runtime, the kernel wrapper is constructed, that supplies the static data to the included kernel, creating partially applied kernel. Then AnyDSL JIT compiler is invoked, which specializes the kernel according to the annotations provided, and static arguments supplied, gen-

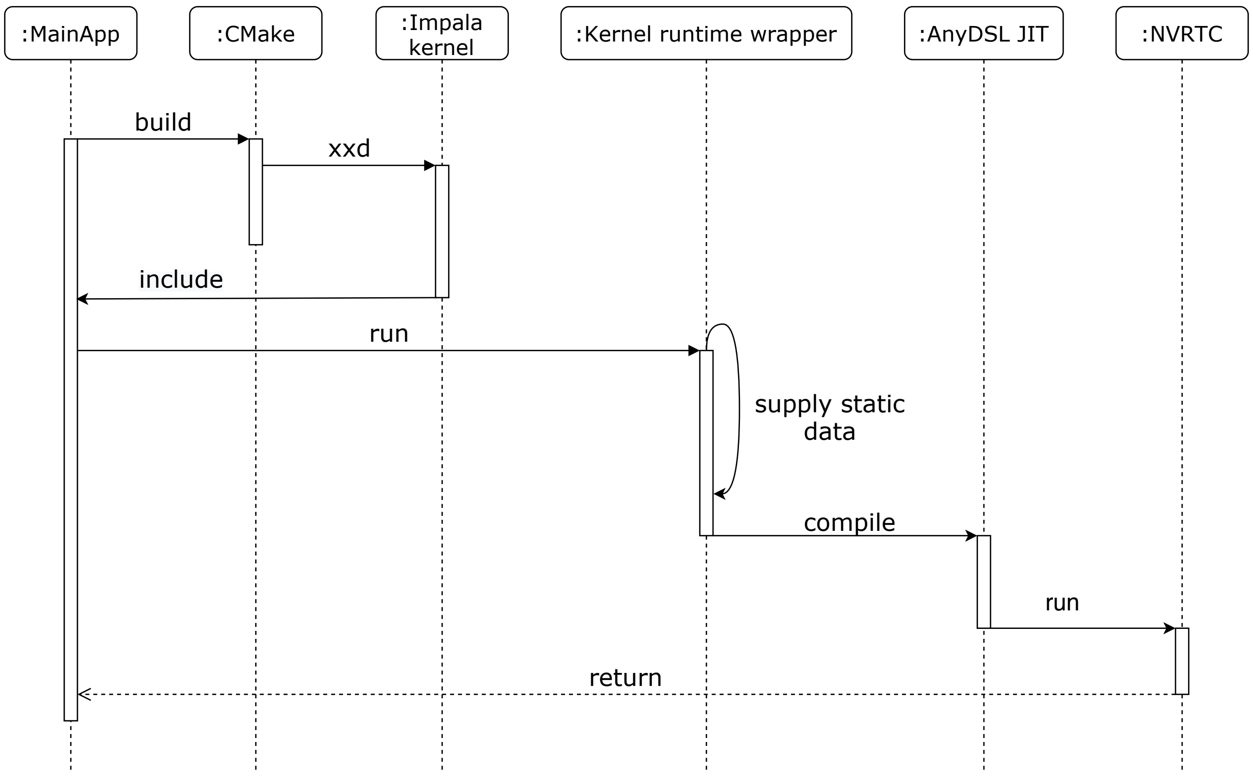

Figure 3: Runtime partial evaluation diagram

erating CUDA C code, which is then passed to NVRTC[11] and got eventually compiled to GPU assembly and invoked.

The evaluation aim is to show whether a device kernel could benefit from data embedding performed by partial evaluation and possible reduction of static computations. For the data to be embedded, the accesses should be static, corresponding to constant memory to be a good fit in CUDA code, thus constant memory being a baseline in several scenarios. Further, only the execution time of a device kernel should be measured, since, for example, overhead for partial evaluation and JIT compilation for a device could be hidden by GPU data transferring or other workarounds.

Since NVRTC is internal to AnyDSL framework, the benchmarking results could be obtained via nvprof[12] or by utilizing a specially recompiled version of the framework runtime with CUDA events. The latter option is used since it allows to perform warm-up runs of the kernel to make the benchmarking more reliable. The whole system is implemented in C++ and Python, and enclosed into a Docker container with the datasets indexed in Git LFS[13] for benchmarks to be easily built and run on any system with NVIDIA GPU[14]. The following GPGPU scenarios have been implemented, which are fit under the described in 1.3 pipeline.

- Naíve single substring matching.
- Naíve multiple substring matching.
- Aho–Corasick matching.
- 2-D convolution filter.

The following system has been used to run the benchmarks: Ubuntu 18.04 with CUDA Toolkit 10.2. hosted in Google Cloud bundled with 4 cores of Intel Xeon and NVIDIA Tesla T4 GPU.

[11]`https://docs.nvidia.com/cuda/nvrtc/index.html` (last accessed date: 30.05.2020)

[12]`https://docs.nvidia.com/cuda/profiler-users-guide/index.html` (last accessed date: 30.05.2020)

[13]`https://git-lfs.github.com/` (last accessed date: 30.05.2020)

[14]`https://github.com/Tiltedprogrammer/spec` (last accessed date: 30.05.2020)

3 Evaluation

In this section, the implementation details of the benchmarks are provided, as well as the selected datasets, and GPU architecture details that affect the result.

3.1 String matching

GPU-accelerated string matching frequently appears in GPU-based databases, file carving [19, 5] that stands for extracting files from raw data in a field of cyber forensics, and intrusion detection [24]. Thus such a problem has a huge practical interest. Substrings could be considered static and subjected to partial evaluation.

3.1.1 Naìve single substring matching

Naìve algorithm operates on a *subject* string and a *pattern*, iteratively comparing each symbol of the pattern with the symbols of each substring of the subject string of size equal to the size of the pattern. The algorithm is inherently data-parallel: such substrings could be traversed separately, each in their thread. Further, such a traversal provides a rather optimal global memory access pattern, since adjacent threads would access addresses of adjacent substrings. There exist an opportunity for optimization, since at one moment all threads in a warp access the same symbol of the pattern (and distinct symbols of the subject string), thus the pattern could be placed in constant memory to speed up the performance.

There is a *KMP* test for optimizers like partial evaluator, intended to check whether they correctly reduce static computations. Partial evaluation is able to reduce a naìve single substring matching, though properly rewritten, to something very like Knuth–Morris–Pratt algorithm [2], which uses a prefix function to recover from a mismatch, thus being linear with respect to the subject string. When a mismatch occurs, it is known that all previous symbols of the pattern match the ones of the substring of the subject string up to the point of mismatch. In order to simulate a prefix function at the mismatch point, the pattern could be matched against itself up to this point, searching for the largest prefix being a suffix as well. Since the pattern is static and matched against itself, this computation is fully static and could be performed at specialization time. Porting this approach to GPU and Impala, and applying partial evaluation at runtime indeed produces a KMP-like program[15]. Thus, making the used partial evaluator sound in a sense. The KMP algorithm itself is hard to partially evaluate since the already matched piece of the pattern that drives the search through the backtracking table is fully dynamic.

However, such an approach is far less data parallel as well as KMP. The subject string should be divided into interleaved chunks, each assigned to a different thread. Such parallelization has a far worse access pattern since each thread accesses strided addresses of the subject string. However, in practice, such an access pattern occurs when the search is performed across different subject strings, where each thread operates on a specific string[16]

A naìve single substring matching could be also specialized by simply unrolling the traversal, that does not hurt the parallelization. The evaluation of these approaches is presented in figure 4.

The current recursive implementation of the partial evaluator does not handle well the KMP test with patterns of more than 16 bytes in length. Thus, for the evaluation, a set of uniformly randomized 50 patterns[17] from two-characters alphabet has been taken, satisfying that restriction. The subject string has been also randomly generated from the same alphabet. The standard deviation is shown in gray regions. The KMP algorithm obtained by partial evaluation (*kmp_ pe*), KMP algorithm implemented with CUDA, utilizing constant memory to store the backtrack table and the pattern (*kmp_ cuda_ const*), naìve algorithm in CUDA with chunk-based parallelization (*naive_ cuda_ const*), naiveìve algorithm in CUDA with char-based parallelization and its partially evaluated version, that embeds the pattern into the code through unrolling the traversal, are compared. The

[15]`https://github.com/Tiltedprogrammer/spec/blob/master/kmp.dump` (last accessed date: 30.05.2020)

[16]`https://github.com/NVIDIA/nvstrings` (last accessed date: 30.05.2020)

[17]Enough for the divergence not to affect deviation so much.

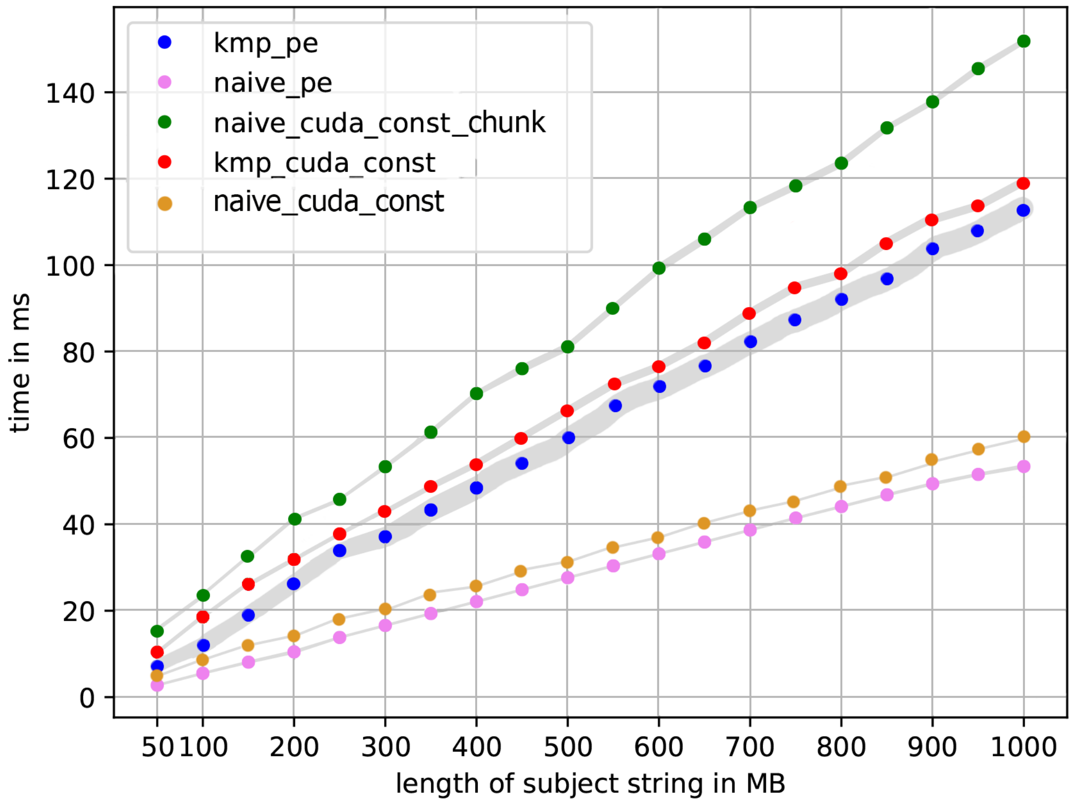

Figure 4: Naìve single substring matching

figure grounds the success of the KMP test and shows that data embedding does not give any significant improvements: the difference between embedded and not embedded versions are mainly due to the reduced amount of address computations and loop overhead. The char-based parallel version is at the bottom due to a better memory access policy.

3.1.2 Data embedding

Partial evaluation for the scenario above performed the transformation similar to the one in listing 6. While the load/store speed comparison for memory spaces is well described [6], e.g. non-conflicting constant load is faster than L1 cache hit, the behavior of such embedded data is unspecified, but could be deduced as follows. All the arguments of assembly instructions should be put into registers, even the embedded ones, e.g. NVIDIA profiler[18] is able to tell how much of registers is required to perform an

[18]`https://developer.nvidia.com/nsight-visual-studio-edition` (last accessed date: 30.05.2020)

instruction: `MOV R1 0x62` requires two registers, thus embedded data is eventually ending up in them. Yet, it is unclear from what memory space the embedded data is being put. The mini benchmark from listing 5 measures the number of cycles required to perform an addition operation `add.u32` with one of the arguments passed via embedded value or via constant memory, lines 9 and 13. Given such a benchmark, the version with the value embedded performs 10 times faster: 42 cycles versus 430 on a constant cache miss. If the constant cache is firstly warmed up, the latency becomes 42 cycles and is the same for both instructions. Thus, embedded values are more likely to be accessed via instruction memory, since the instructions are prefetched and would outperform constant memory access under cache misses. Notably, when embedding, a partial evaluator is able to generate a more effective instruction, e.g. shift instead of division, which takes more than 10 instructions on GPU, while shift requires only one.

```
 1 __constant__ int mini_array [2];
 2
 3 __global__ void dummy_kernel(int* dst,int* clocks){
 4
 5     int i;
 6     int start,stop;
 7
 8     asm volatile(”mov.u32 %0, %%clock;”: ”=r”(start) :: ”memory”);
 9     asm volatile(
10                 ”add.u32 %0, %1, 12;\n\t”
11                 :”=r”(i) :”r”(i): ”memory”);
12     // vs
13     asm volatile(
14                 ”add.u32 %0, %1, %2;\n\t”
15                 :”=r”(i) :”r”(i),”r”(mini_array[0]) : ”memory”);
16     asm volatile(”mov.u32 %0, %%clock;”: ”=r”(stop) :: ”memory”);
17 }
```

Listing 5: Memory benchmark

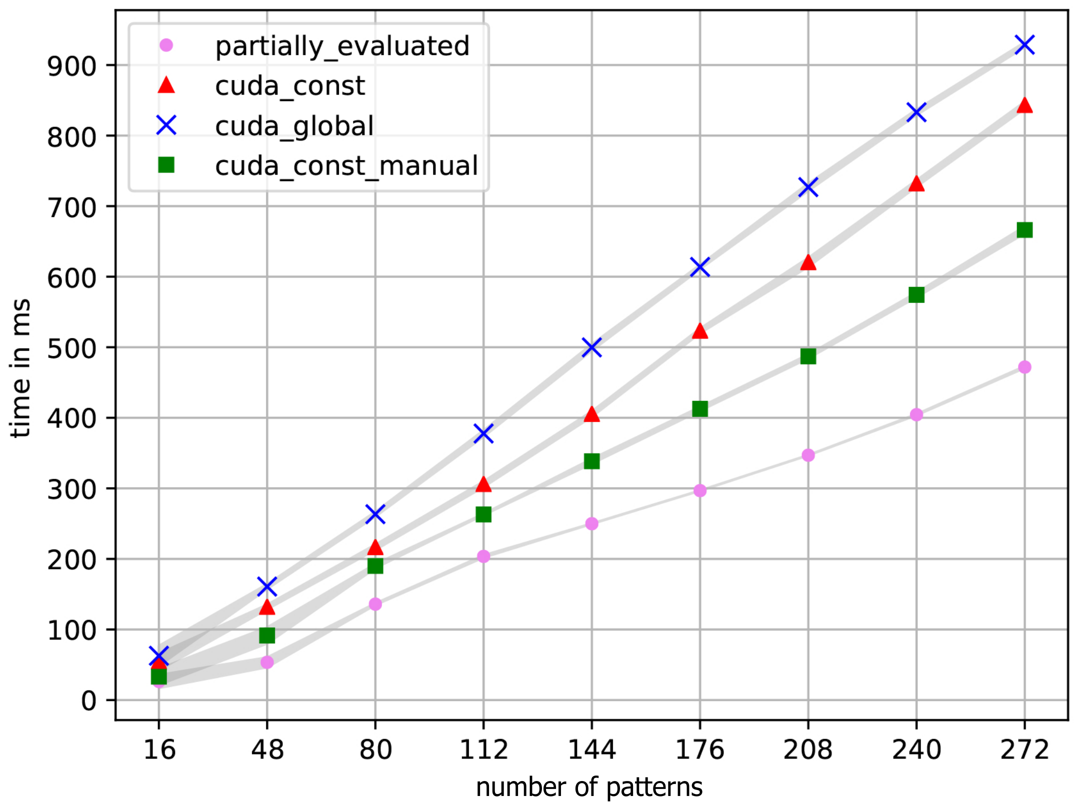

Figure 5: Naïve multiple substring matching

```
    //kmp_cuda_const
LDC R0, c[0x3][R0] //loads pattern's character from constant memory into
    register
    //...//
LDG R12, [R2] //loads subject's character from global memory
ISETP.NE.AND P0, R0, R12 // compares
    //..//
LDC R4, c[0x3][R2] // in case of mismatch goto position in backtrack table

    //kmp_pe
LDG R12, [R2] //loads subject's character from global memory
    //    ...
ISETP.NE.AND P0, R12, 0x61 // pattern's char is put right into the
    instruction
```

Listing 6: KMP partial evaluation

3.1.3 Naìve multiple substring matching

The core of the algorithm is the same as the one of 3.1.1 with the addition that a set of patterns is traversed against the substring. Since a set of patterns is traversed, their sizes are needed to be accessed, in order to be able to determine the border between the patterns. Constant memory could be utilized to store the sizes and the patterns, since threads of a warp access the same size and the same symbol of the pattern. Partially evaluating the algorithm with respect to the patterns, allows to fully reduce all the memory accesses to the sizes, which are numerous when the subject string is huge. Partial evaluation is achieved through loop unrolling for the sizes and the patterns. The results of this benchmark are presented in figure 5.

The dataset is from the intrusion detection area, which is common for multiple pattern matching problems [3]. The subject string is 500MB *tcp-dump* from *Botnet* dataset [27]. The patterns are extracted from *Snort V3*[19] rules, which are the patterns containing malicious traffic. The same set of patterns has been run over 30 times taking the average. The gray area represents the standard deviation.

Cuda_ global is the implementation with global memory for storing the sizes and the patterns, while *cuda_ const* uses constant memory for this. As it could be seen, the partially evaluated version is at the very bottom being 2x faster than the version with constant memory. *cuda_ const_ manual* is the implementation, where all size accesses have been reduced manually, using a CUDA JIT compiler[20]. It is slower than the partially evaluated version due to the reasons related to CUDA compiler implementation. In this case, provided the patterns embedded into the code, the compiler is able to produce a more efficient code from listing 7, considering the number of instructions. It could be tempting to embed data of size exceeding a constant memory pool of 64 KB into code, provided that such access could be even faster. First, CUDA limits the maximum number of instructions that could be put into a module by 512 million. Second, compilation time

[19]`https://www.snort.org/downloads` (last accessed date: 30.05.2020)

[20]https://github.com/NVIDIA/jitify

begins to matter for relatively huge embedded data, e.g. 272 patterns are 5~KB in size and compilation took several minutes.

```
    //Cuda_const_manual

IMAD.MOV.U32 R8, RZ, RZ, R4
ULDC.64 UR4, c[0x0][0x160]
IMAD.MOV.U32 R9, RZ, RZ, R3
LDG.E.U8.SYS R8, [R8.64+UR4+0x1] // load subject's character
ULDC.U8 UR4, c[0x3][0x1] //load pattern's character
ULOP UR4, UR4, 0xff, URZ, 0xc0
IMAD.U32 R11, RZ, RZ, UR4
PRMT R11, R11, 0x9910,RZ
PRMT R12, R8, 0x9910,RZ
ISETP.NE.AND P0, PT, R12, R11, PT //compare

    //partially_evaluated

IMAD.MOV.U32 R6, RZ, RZ, R2
ULDC.64 UR6, c[0x0][0x160]
IMAD.MOV U32 R7, RZ, RZ, R5
LDG.E.U8.SYS R6, [R6.64 + UR6+0x1] // load subject's character
PRMT R8, R6, 0x9910, RZ
ISETP.NE.AND P0, PT, R8, 0x42, PT //compare
```

Listing 7: Cuda_const_manual vs partially_evaluated

3.1.4 Aho-Corasick matching

Aho-Corasick is a time efficient algorithm for multiple string pattern matching [3] and based on suffix tree and a failure transition table. However, a failure transition table becomes redundant by switching to GPU, since a simple suffix tree traversal per thread for each position in the subject string appear to be more efficient [1]. The parallel Aho-Corasick first construct a transition table where final states have numbers less than the starting state, to reduce memory accesses for checking whether the state is final or not. The table is stored in global memory and each thread traverses the table taking a subject string character from shared memory. The benchmark and the implemented algorithms are presented in figure 6. The dataset used is the same as in 3.1.3. *Cuda_corasick_lib* is the implementation of this algorithm[21] taken from [1]. The transition table is often sparse and could be embedded into the code during partial evaluation using static binding of dynamic variables [11] avoiding empty entries, *corasick_pe* is the implementation of this approach. *Naive_pe* is the algorithm from 3.1.3 utilizing shared memory. *Cuda_corasick_pe* is the implementation, where a pattern set specific interpreter for the transition table is generated. The interpreter represents a sequence of code-embedded conditional statements to traverse the suffix tree.

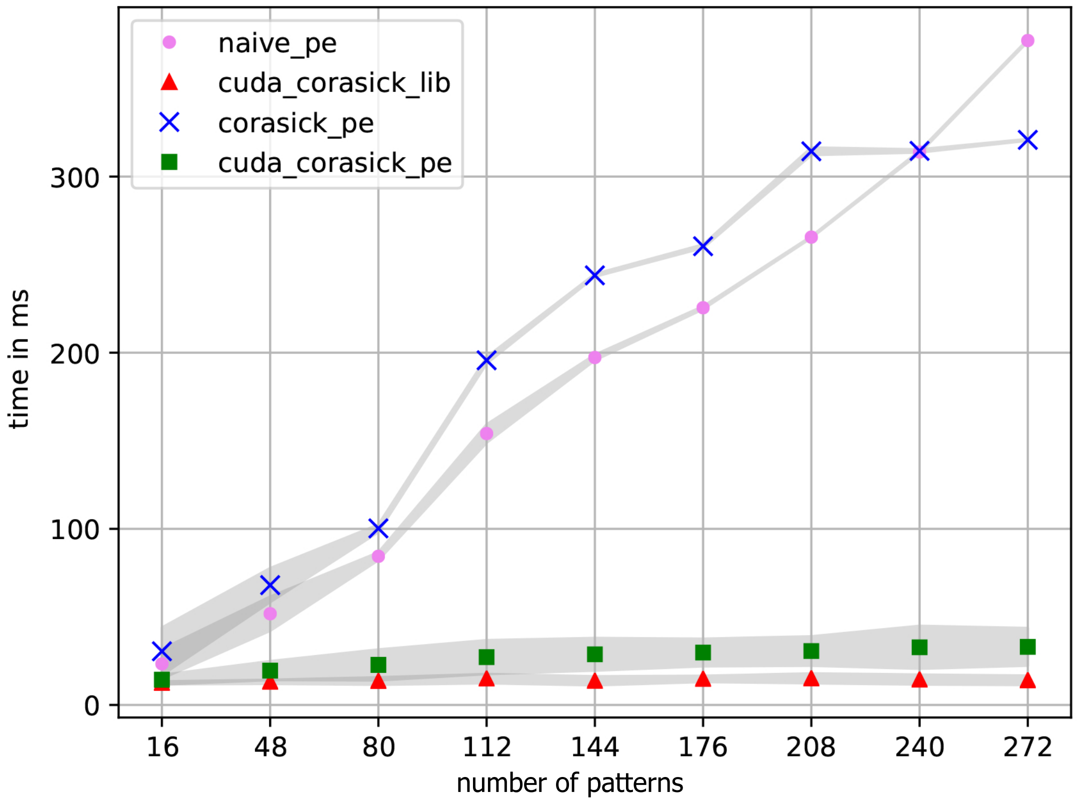

Figure 6: Aho-Corasick matching

Static binding is a traversal of statically known possible values of a dynamic parameter in a bunch of condition statements. In this case, such an approach induces more thread divergence, e.g. 6 times if compared to *cuda_corasick_lib*, which drastically decreases the performance of a GPU application. The interpreter approach has better divergence behavior but

[21]https://github.com/pfac-lib/PFAC

still diverges twice as much as *cuda_corasick_lib*. So, given that global memory access latency even on L1 cache hit is more than the one for embedded data, the static binding approach that could work well on a CPU has a poor performance when a GPU is used, and even if interpretative implementation is near, such an embedding also does not provide any performance benefit.

3.2 Convolutional filtering

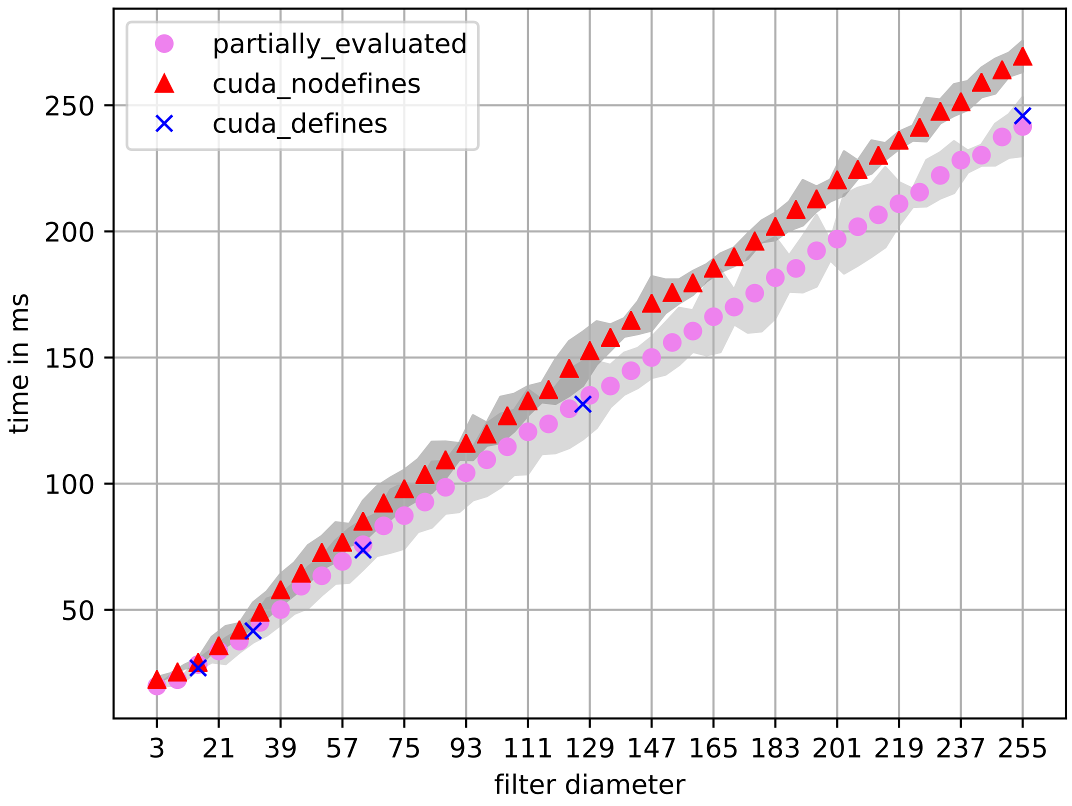

Figure 7: Separable convolution

Convolutional filtering is a matrix dot product frequently used in image processing [26]. Basically, there are a huge subject matrix and a small filter matrix, and each submatrix of the subject matrix is dot producted with the filter matrix. Since the filter is small and is static, the convolution operation could be partially evaluated with respect to the filter. A partial evaluation for a 2-D separable[22] convolutional filtering has been performed in [23], targeting different hardware, however, the described filter is a rewritten modification of the reference filter, thus the obtained speed-up is not achieved by means of partial evaluator solely, and is used to demonstrate the facilities of the Impala language.

3.2.1 Separable convolution

A separable convolutional filter has been implemented in CUDA and Impala [18]. It operates on a 2-D array in global memory with threads organized in $32x16$ blocks, where each thread convolves 8 elements. Shared memory is utilized to store the big area for convolution with a block and the required borders. Since the filter is read-only and accessed equally by all the threads, it is stored in constant memory. Elements that fall away the borders of the 2-D subject array are assigned zeroes. There are two device kernels to perform a convolution: one convolves the rows and the other convolves the columns with the shared memory padded enough to not cause bank conflicts when accessing column elements.

```
//cuda_defines
LDS.U R16, [R2 + 0x38] //load from shared
FFMA R17, R10, c[0x3][0x1dc] R17 //float multiply add

//partially_evaluated
LDS.U R16, [R2 + 0x38] //load from shared
FFMA R17, R10, 42 R17 //float multiply add
```

Listing 8: Convolution partial evaluation

The convolution itself is a dot product of two vectors of filter size. Since the size is known, this cycle could be unrolled with the filter being embedded. Such an unroll could be also performed by means of C++ templates or macros, by creating a dedicated device kernel for a specific filter size and dynamically dispatching the appropriate kernel for input data. The results of the benchmark are depicted in figure 7. The subject 2-D array is 1GB randomly generated, the filters are randomly generated for each diameter. The average has been taken for each filter size from 30 runs. The gray

[22]1-D filter could be applied first to rows and then to columns.

area is the standard deviation. *Cuda_ nodefines* is the implementation with the dot product not unrolled, *cuda_ defines* leverages macros to unroll the product, *partially_ evaluated* leverages partial evaluation. Unrolled versions are at the bottom since they reduce the loop overhead. Unrolled version perform equally due to the code generated by the compiler as in listing 8 which has been shown to be executed in the same number of cycles in 3.1.2.

3.3 Discussion

While data embedding by means of partial evaluation achieves the same performance as constant memory access, given that no access instructions had been reduced, there are cases when it outperforms due to the reasons internal to the compiler. Thus, partial evaluation could be a way to automate constant memory management and even speed up the performance. Since the performance is compiler specific, which is further device-specific, there is a need to investigate the compiler which is publicly unspecified, and at the moment of writing there is a lack of tools for that. Also, partial evaluation performance is scenario- specific, hence other scenarios should be investigated as well. Finally, guidelines of developing a program that is amenable to partial evaluation [11] could be extended to explicitly multithreaded programs such as CUDA kernels, or the semantics of the intermediate representation could consider the multithreaded environment. However, these questions are beyond the scope of the work.

Conclusion

The following results have been obtained in the course of the work.

- Experimental scenarios have been implemented[23] in CUDA and AnyDSL.
- Experimental datasets have been gathered.
- The evaluation has been performed and the results have been analyzed.
  - GPU architecture-specific details that affect the result have been identified.
- A poster with a part of this work has been published[24] in the proceedings of *PPOPP-2020*.

A partial evaluation has been shown to speed up memory operations in certain situations.

[23] `https://github.com/Tiltedprogrammer/spec` (last accessed date: 30.05.2020)
[24] `https://dl.acm.org/doi/abs/10.1145/3332466.3374507` (last accessed date: 30.05.2020)

References

[1] Accelerating Pattern Matching Using a Novel Parallel Algorithm on GPUs / C. Lin, C. Liu, L. Chien, S. Chang // IEEE Transactions on Computers. — 2013. — Vol. 62, no. 10. — P. 1906–1916.

[2] Ager Mads Sig, Danvy Olivier, Rohde Henning Korsholm. On Obtaining Knuth, Morris, and Pratt's String Matcher by Partial Evaluation // Proceedings of the ASIAN Symposium on Partial Evaluation and Semantics-based Program Manipulation. — ASIA-PEPM '02. — New York, NY, USA : ACM, 2002. — P. 32–46. — Access mode: `http://doi.acm.org/10.1145/568173.568177`.

[3] Aho Alfred V., Corasick Margaret J. Efficient String Matching: An Aid to Bibliographic Search // Commun. ACM. — 1975. — Jun. — Vol. 18, no. 6. — P. 333–340. — Access mode: `https://doi.org/10.1145/360825.360855`.

[4] AnyDSL: A Partial Evaluation Framework for Programming High-performance Libraries / Roland Leissa, Klaas Boesche, Sebastian Hack et al. // Proc. ACM Program. Lang. — 2018. — Oct. — Vol. 2, no. OOPSLA. — P. 119:1–119:30. — Access mode: `http://doi.acm.org/10.1145/3276489`.

[5] Bayne E., Ferguson R.I., Sampson A.T. OpenForensics: A digital forensics GPU pattern matching approach for the 21st century // Digital Investigation. — 2018. — Vol. 24. — P. S29 – S37. — Access mode: `http://www.sciencedirect.com/science/article/pii/S1742287618300379`.

[6] Jia Zhe, Maggioni Marco, Smith Jeffrey, Scarpazza Daniele Paolo. Dissecting the NVidia Turing T4 GPU via Microbenchmarking. — 2019. — 1903.07486.

[7] Zhang Junzhe, Yeung Sai Ho, Shu Yao et al. Efficient Memory Management for GPU-based Deep Learning Systems. — 2019. — 1903.06631.

[8] Futamura Yoshihiko. Partial Evaluation of Computation Process–An Approach to a Compiler-Compiler // Higher-Order and Symbolic Computation. — 1999. — Dec. — Vol. 12, no. 4. — P. 381–391. — Access mode: `https://doi.org/10.1023/A:1010095604496`.

[9] Gelado Isaac, Garland Michael. Throughput-Oriented GPU Memory Allocation // Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming. — PPoPP '19. — New York, NY, USA : Association for Computing Machinery, 2019. — P. 27–37. — Access mode: `https://doi.org/10.1145/3293883.3295727`.

[10] Jones Neil D. An Introduction to Partial Evaluation // ACM Comput. Surv. — 1996. — Vol. 28, no. 3. — P. 480–503. — Access mode: `https://doi.org/10.1145/243439.243447`.

[11] Jones Neil D., Gomard Carsten K., Sestoft Peter. Partial Evaluation and Automatic Program Generation. — Upper Saddle River, NJ, USA : Prentice-Hall, Inc., 1993. — ISBN: 0-13-020249-5.

[12] Just-In-Time GPU Compilation for Interpreted Languages with Partial Evaluation / Juan Fumero, Michel Steuwer, Lukas Stadler, Christophe Dubach // SIGPLAN Not. — 2017. — Apr. — Vol. 52, no. 7. — P. 60–73. — Access mode: `https://doi.org/10.1145/3140607.3050761`.

[13] Kleene Stephen Cole. Introduction to Metamathematics. — North Holland, 1952.

[14] Leißa R., Köster M., Hack S. A graph-based higher-order intermediate representation // 2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). — 2015. — P. 202–212.

[15] McCool Michael, Du Toit Stefanus. Metaprogramming GPUs with Sh. — CRC Press, 2009.

[16] One VM to Rule Them All / Thomas Würthinger, Christian Wimmer, Andreas Wöß et al. // Proceedings of the 2013 ACM International

Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software. — Onward! 2013. — New York, NY, USA : Association for Computing Machinery, 2013. — P. 187–204. — Access mode: `https://doi.org/10.1145/2509578.2509581`.

[17] Performance of the NVIDIA Jetson TK1 in HPC / Y. Ukidave, D. Kaeli, U. Gupta, K. Keville. // 2015 IEEE International Conference on Cluster Computing. — 2015. — P. 533–534.

[18] Podlozhnyuk Victor. Image convolution with CUDA // NVIDIA Corporation white paper, June. — 2007. — Vol. 2097, no. 3.

[19] Povar Digambar, Bhadran V. K. Forensic Data Carving // Digital Forensics and Cyber Crime / Ed. by Ibrahim Baggili. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2011. — P. 137–148.

[20] Runtime Specialization of PostgreSQL Query Executor / Eugene Sharygin, Ruben Buchatskiy, Roman Zhuykov, Arseny Sher // Perspectives of System Informatics / Ed. by Alexander K. Petrenko, Andrei Voronkov. — Cham : Springer International Publishing, 2018. — P. 375–386.

[21] Sakdhnagool Putt, Sabne Amit, Eigenmann Rudolf. RegDem: Increasing GPU Performance via Shared Memory Register Spilling // CoRR. — 2019. — Vol. abs/1907.02894. — 1907.02894.

[22] Shali Amin, Cook William R. Hybrid Partial Evaluation // SIGPLAN Not. — 2011. — Oct. — Vol. 46, no. 10. — P. 375–390. — Access mode: `https://doi.org/10.1145/2076021.2048098`.

[23] Shallow embedding of DSLs via online partial evaluation / Roland Leißa, Klaas Boesche, Sebastian Hack et al. — 2015. — 10. — P. 11–20.

[24] Tumeo Antonino, Villa Oreste, Sciuto Donatella. Efficient Pattern Matching on GPUs for Intrusion Detection Systems // Proceedings of the 7th ACM International Conference on Computing Frontiers. —

CF ’10. — New York, NY, USA : Association for Computing Machinery, 2010. — P. 87–88. — Access mode: https://doi.org/10.1145/1787275.1787296.

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

[26] Chetlur Sharan, Woolley Cliff, Vandermersch Philippe et al. cuDNN: Efficient Primitives for Deep Learning. — 2014. — 1410.0759.

[27] A survey of network-based intrusion detection data sets / Markus Ring, Sarah Wunderlich, Deniz Scheuring et al. // Computers & Security. — 2019. — Sep. — Vol. 86. — P. 147–167. — Access mode: http://dx.doi.org/10.1016/j.cose.2019.06.005.