Static Analysis of GPU Kernel Performance Hyperproperties

Bachelor's Thesis

Mathias Blarer

blarerm@student.ethz.ch

Programming Methodology Group
Institute for Programming Languages and Systems
Department of Computer Science
ETH Zurich, Switzerland

Supervisors:
Marco Eilers, Jérôme Dohrau
Prof. Dr. Peter Müller

September 14, 2019

Contents

Chapter 1

Introduction

Graphical processing units (GPUs) are highly-parallel computing devices, commonly built into today's computers and smartphones. They are important for the speed-up of many different applications, reaching from graphic-intense software like games to fast numerical computation tools like MATLAB. Programs written for the execution on GPUs are called *kernels*. In order to achieve optimal GPU kernel performance, various aspects need to be considered when writing kernel code. The goal of this thesis is to design a static analysis of GPU kernels that analyzes properties that have an effect on the performance of the kernel execution. Such a static analysis can point out possible code inefficiencies and, thus, help a programmer to write faster kernels. We focus on three properties, which are

1. whether the kernel contains control flow statements that lead to *branch divergence* (which is bad for instruction throughput),
2. whether the kernel contains accesses to shared memory that lead to *shared memory bank conflicts* (which is bad for memory throughput), and
3. whether the kernel contains accesses to global memory that allow for *global memory coalescing* (which increases the memory throughput).

These three properties belong to a class of program properties called *hyperproperties* which are properties that relate multiple program executions.

Related and Previous Work

The GPUVerify project [2] is a project dedicated to the analysis of GPU kernel correctness through invariant guessing. GPUVerify does not analyze hyperproperties related to kernel performance but proves the absence of data races and barrier divergence, which are correctness properties.

In 2018, Eilers et al. [7] have proposed k-modular product programs as a means to express hyperproperties as properties of a single program execution trace. Thus, using product programs, it is possible to analyze hyperproperties with off-the-shelf verifiers.

Building upon product programs, Knabenhans [9] has implemented a hyperproperty analysis framework in the Sample static analyzer[1] as part of the Viper verification infrastructure [10]. This framework performs abstract interpretation on product programs with standard relational abstract domains like Octagons or Polyhedra and uses a technique called *trace*

[1] `https://www.pm.inf.ethz.ch/research/sample.html`

partitioning. We extend this framework with the ability to analyze GPU kernels regarding performance hyperproperties.

Contributions and Overview

In this thesis, we make the following contributions:

- We introduce a so-called *m-alignment transformation*, a program transformation that allows to leverage standard numerical static analyses to capture division and modulo constraints.
- We use the above m-alignment transformation to develop a static analysis of GPU kernel performance hyperproperties, thus proving m-aligned programs useful.
- We implement this static analysis in the Sample static analyzer and show that the analysis is able to reason about GPU kernel performance hyperproperties in interesting kernels.

Chapter 2 contains all the preliminaries for the rest of this thesis that can be skipped if the reader is familiar with the individual topics. In Chapter 3, we formalize the aforementioned GPU kernel performance hyperproperties and explain our approach for statically analyzing them. We describe the m-alignment transformation which simplifies our task significantly. More details on the implementation of our static analysis are given in Chapter 4 and evaluation results are shown in Chapter 5. We summarize all findings and propose future work in the conclusion in Chapter 6. The appendix contains a list of all m-alignment transformations on one hand and the listing of all GPU kernels used in the evaluation on the other hand.

Acknowledgements

First of all, I want to thank my supervisors Marco Eilers and Jérôme Dohrau for their great support, their numerous ideas and their suggestions for the improvement of this thesis. I also want to thank Prof. Dr. Peter Müller, head of the Programming Methodology Group at ETH Zurich, for giving me the opportunity to do my Bachelor's thesis in his research group. Many thanks to Caroline Pilat for proofreading this document, and, last but not least, I want to thank my family, friends, and my wife Pauline for supporting me the whole duration of my thesis.

Chapter 2

Preliminaries

2.1 Language Syntax

Since this thesis contains many definitions and explanations that contain programming language constructs like statements and expressions, we define a language syntax below, which we will be using in the following. It is similar to the syntax used in [7]. $\mathbb{V}$ denotes the set of program variables.

(Programs)	Prog	$::=$	$\texttt{procedure}\ \mathit{main}(\overline{\texttt{x}})\ \texttt{returns}\ (\overline{\texttt{y}})\ \{s\} \quad \mid \quad \mathit{Proc} :: \mathit{Prog}$
(Procedures)	Proc	$::=$	$\texttt{procedure}\ f(\overline{\texttt{x}})\ \texttt{returns}\ (\overline{\texttt{y}})\ \{s\}$
(Statements)	s	$::=$	$\texttt{skip} \quad \mid \quad \texttt{x} := e \quad \mid \quad s; s \quad \mid \quad \texttt{if}\ (b)\ \texttt{then}\ \{s\}\ \texttt{else}\ \{s\} \quad \mid$
			$\texttt{while}\ (b)\ \texttt{do}\ \{s\} \quad \mid \quad \overline{\texttt{y}} := \texttt{call}\ f(\overline{e})$
(BoolExps)	b	$::=$	$\mathit{true} \quad \mid \quad \mathit{false} \quad \mid \quad \neg b \quad \mid \quad b \wedge b \quad \mid \quad b \vee b \quad \mid$
			$e \circ e$, where $\circ \in \{=, \neq, <, >, \leq, \geq\}$
(IntExps)	e	$::=$	$\texttt{x} \in \mathbb{V} \quad \mid \quad c \in \mathbb{Z} \quad \mid \quad e \star e$, where $\star \in \{+, -, *, /, \%\}$

Some remarks on notation:

- We write $\overline{\texttt{x}}$ to denote zero or more variables, i.e., $\overline{\texttt{x}} = \texttt{x}_1, \ldots, \texttt{x}_\texttt{n}$ for $n \geq 0$. Similarly, we write $\overline{e}$ to denote zero or more expressions.
- If the precedence of operations in some expression is clear using the usual rules (e.g., $*, /$, and $\%$ have higher precedence than $+$ and $-$), then we do not insert parentheses.
- Instead of nesting `if` statements, we use the common notation `else if` to state multiple alternative conditional clauses.
- The unary negation expression $-e$ is syntactic sugar for the subtraction $0 - e$. For simplicity, we do not explicitly include negation expressions in our language syntax.
- Similarly, we treat the implication expression $b_1 \Rightarrow b_2$ as syntactic sugar for $\neg b_1 \vee b_2$.
- $/$ denotes C-like integer division with truncation (rounding towards zero). If in some formula, we want to denote integer division with rounding towards negative infinity instead, we write $\lfloor \dot{\div} \rfloor$.
- Analogously, $\%$ denotes the C-like modulo operation, which is defined with respect to $/$ such that $n = (n\ /\ m) * m + n\ \%\ m = n$. In particular, if $n < 0$ above, then $n\ \%\ m \leq 0$ for all integers $m \neq 0$. If we want to refer to the mathematical remainder, which is always non-negative, we write $n \bmod m$ instead.

2.2 Abstract Interpretation

Abstract interpretation is a theory formalized by Patrick and Radhia Cousot [5] that is used in static analysis for the sound approximation of the semantics of a program. A concrete program trace is a (possibly infinite) sequence of program states where a state can be seen as a function that assigns values to all program variables. Without actually executing a given program on any inputs, abstract interpretation can provide an overapproximation of all possible concrete program traces using abstract program states. We do not go into all the details of abstract interpretation in static analysis here, more information can be found in [4].

It is important to note that while concrete program states live in a concrete universe of integers, pointers etc., the abstract states that approximate the concrete states live in an abstract universe called an *abstract domain.* These abstract domains are usually based on lattices and are part of the abstract interpretation configuration which is chosen appropriately for a specific problem. For our purposes, we consider numerical abstract domains which are relational, i.e., which approximate the values of numerical variables and the relations between them. Specifically, we use one of the two following abstract domains:

- *Octagons.*
  The Octagons domain can store inequalities of the form

  $$\pm\,\mathtt{x} \pm \mathtt{y} \leq c,$$

  where `x` and `y` are any two numerical program variables and c is a constant number.

- *Polyhedra.*
  The Polyhedra domain is more powerful than the Octagons domain since it can store inequalities of the form

  $$a_1\mathtt{x_1} + a_2\mathtt{x_2} + \ldots + a_n\mathtt{x_n} \leq c,$$

  where $\mathtt{x_1}$ through $\mathtt{x_n}$ are some n numerical program variables, a_1 through a_n are constant coefficients, and c is a constant number.

In contrast to the Octagons domain, the Polyhedra domain can represent relations between more than two variables as well as weighted sums and differences using the coefficients a_i for $i = 1, \cdots, n$. The tradeoff here is precision against runtime: the Polyhedra domain can store more complex constraints than the Octagons domain but it also has a much worse runtime complexity which is a decisive factor when used with programs that contain many numerical variables. In the evaluation in Chapter 5, we argue that for the abstract interpretation of GPU kernel hyperproperties, the Octagons domain is the preferable abstract domain.

2.3 Hyperproperties

In this section, we first explain the concept of hyperproperties. In the first subsection, we discuss product programs, which allow hyperproperties to be expressed as properties of single execution traces. The second subsection is about trace partitioning, which is a technique that we use for the more precise abstract interpretation of product programs.

Hyperproperties are program properties that relate multiple execution traces. More precisely, *k-safety hyperproperties* relate finite prefixes of k execution traces. In contrast to *liveness* properties, *safety* properties ("Something bad never happens") are violated in finite time.

Example. *Determinism* is a 2-safety hyperproperty because it relates two finite execution traces: A procedure f with inputs $\overline{\mathtt{x}}$ and outputs $\overline{\mathtt{y}}$ is deterministic if and only if two executions with the same inputs $\overline{\mathtt{x}}$ produce the same outputs $\overline{\mathtt{y}}$. Further examples of 2-safety hyperproperties are non-interference, injectivity, monotonicity etc. △

The three aforementioned hyperproperties on branch divergence, shared memory bank conflicts, and global memory coalescing are 2-safety hyperproperties as well. The following subsection shows how k-safety hyperproperties can be analyzed using product programs.

2.3.1 Product Programs

The idea behind k-modular product programs is to use self-composition in order to represent k simultaneous executions of the program. To this end, each statement in the original program is replicated k times and every replicated statement is executed under the condition that the corresponding execution is active. Whether or not a particular execution is active, is stored in boolean *activation variables*, in the following denoted by $\mathtt{p}^0$, …, $\mathtt{p}^{\mathtt{k}-1}$.

k-safety hyperproperties relate multiple execution traces. One way to reason about such properties is to transform the given program into a so-called *k-modular product program* such that the hyperproperties of the original program become properties of a single execution trace of the product program. A detailed discussion of k-modular product programs can be found in [7].

Example. We want to show that the following simple procedure *increment* is deterministic:

$$\mathtt{procedure}\ \textit{increment}(\mathtt{x})\ \mathtt{returns}\ (\mathtt{y})\ \{\ \mathtt{y} := \mathtt{x} + 1\ \}.$$

As mentioned before, determinism is a 2-safety hyperproperty. Therefore, we transform *increment* into a 2-product program that looks as follows:

```
procedure $\textit{increment}(\mathtt{p}^0, \mathtt{p}^1, \mathtt{x}^0, \mathtt{x}^1)$ returns $(\mathtt{y}^0, \mathtt{y}^1)$ {
  if ($\mathtt{p}^0$) then { $\mathtt{y}^0 := \mathtt{x}^0 + 1$ };
  if ($\mathtt{p}^1$) then { $\mathtt{y}^1 := \mathtt{x}^1 + 1$ }
}.
```

In order to verify that *increment* is deterministic, we verify the implication

$$\mathtt{p}^0 \ \wedge\ \mathtt{p}^1 \ \wedge\ \mathtt{x}^0 = \mathtt{x}^1 \quad \Rightarrow \quad \mathtt{y}^0 = \mathtt{y}^1.$$ △

Remark. In the rest of this thesis, we continue to use the superscript notation $\mathtt{x}^0$, $\mathtt{x}^1$ to denote some variable $\mathtt{x}$ in executions 0 and 1, respectively.

2.3.2 Trace Partitioning

Product programs are constructed in such a way that the control flow of the original program is altered and, in some sense, erased. *Trace partitioning* helps to restore the original control flow by partitioning all execution traces into sets of traces that agree on all activation variables in the product program (which are the traces that have the same control flow in the original program). Each set is analyzed separately which leads to more precise results, as explained further below.

To understand how the program traces are partitioned, recall from the previous subsection that boolean activation variables are added in product programs. Activation variables are needed to condition the execution of a statement in the product program on whether the corresponding execution is active at the statement in the original program. During the abstract interpretation of product programs with trace partitioning, the set of possible program traces is partitioned according to each activation variable being *true* or *false*. Intuitively, for each conditional statement in the original program, trace partitioning makes that all program trace partitions are, individually, split into 2^k partitions where each of the k executions is either active in this conditional branch or not. Abstract interpretation is then performed separately on every trace partition.

If abstract interpretation is performed on a product program without trace partitioning, it may yield very imprecise results. To illustrate this, consider the statement

$$\texttt{if } (\texttt{p}^0) \texttt{ then } \{ \texttt{ y}^0 := \texttt{x}^0 + 1 \ \}$$

from our product program example in the previous subsection. In the abstract state before the interpretation of this statement, $\texttt{x}^0$ and $\texttt{y}^0$ are unrelated. If $\texttt{p}^0$ is *false* (i.e., execution 0 is inactive at this point in the original program), the abstract state does not change. If, however, $\texttt{p}^0$ is *true* (i.e., execution 0 is active at this point), then the new abstract state in a sufficiently powerful abstract domain contains the constraint $\texttt{y}^0 - \texttt{x}^0 = 1$. Since, in general, activation variables can be both *true* or *false*, abstract interpretation chooses an approximation that is sound in both cases, and hence, the constraint $\texttt{y}^0 - \texttt{x}^0 = 1$ is dropped. If trace partitioning is used instead, we get two different abstract states, for the traces in which $\texttt{p}^0$ is *true* or *false*, respectively.

Trace partitioning can be implemented through a special abstract domain called a *Binary Decision Tree (BDT) domain* which is parametric in the abstract domain that is used for the abstract interpretation of every individual trace partition, e.g., Octagons or Polyhedra. As mentioned in the introduction, Knabenhans [9] has implemented an abstract interpretation framework for hyperproperties which uses a parametric BDT domain internally. The main drawback of this BDT domain is that the number of abstract states and interpretation steps to perform is exponential in the number of activation variables. However, heuristic optimizations can be applied to reduce this number of abstract states.

2.4 GPU Architecture Model

In this thesis, we statically analyze GPU kernels with respect to a concrete GPU architecture. The following section discusses the chosen architectural GPU model. There are two perspectives on the GPU architecture, the programmer's perspective and the hardware's perspective, each with a dedicated subsection. The hardware model also contains information about factors that have a positive or negative impact on the performance of kernels running on such GPU architectures.

2.4.1 Programming Model

The programming model defines an abstraction of the GPU architecture that is visible to the programmer. Since there are many different GPU designs and implementations, such an abstraction makes life easier for the programmer who thereby does not have to worry about compatibility issues. However, in order to make code run faster on a GPU, the programmer still needs some understanding of how the code is executed in hardware.

CUDA vs OpenCL

The two major GPU programming models are *CUDA* (Compute Unified Device Architecture) and *OpenCL* (Open Computing Language). CUDA is being developed by Nvidia for the proprietary use with their GPUs; OpenCL is an open standard maintained by the Khronos Group and implemented by GPU architectures of different manufacturers.

In our thesis, we refer to the CUDA programming model and use terms that are specific to CUDA. One reason for this decision is that Nvidia provides a detailed CUDA programming guide [11] which also contains information about the hardware model in Nvidia's GPU design. All explanations given in the rest of this section are paraphrased from the CUDA programming guide.

One important thing to note is that we analyze hyperproperties that are not properties of the high-level GPU kernel as much as properties of the concrete execution traces on a GPU multiprocessor. This implies that, although the hyperproperties are defined in terms of the given kernel (since we are doing a static analysis), these definitions need to take into account the concrete underlying GPU architecture.

Choosing OpenCL as a programming model is fine if, for the chosen GPU architecture, the hardware implementation of the OpenCL standard is known. However, in contrast to OpenCL, CUDA kernels can be related to their execution in hardware more easily because CUDA is only used with Nvidia GPUs which all implement a similar hardware model. That is why, in this thesis, we focus on CUDA kernels executed on Nvidia GPUs.

Multithreaded Kernels in CUDA

CUDA is not a programming language on its own but can be used in different programming languages (C, C++, Fortran etc.) and provides a language extension for these languages. In such an environment, a *kernel* is a function written in the extended language, which can be run with the same arguments N times in parallel by N different *threads* on a CUDA-enabled GPU.

The parallel execution of kernels in separate threads makes little sense if all threads perform the same sequence of instructions on the exact same values. Therefore, kernels are parametrized by a non-negative integer t, called the *thread ID*, which starts at zero and is unique for each and every thread. In other words, if the kernel runs N times in total, then it runs once for every $t = 0, 1, 2, \dots, N-1$.

In CUDA, threads are organized in so-called *blocks* of at most 1024 threads. Several thread blocks are executed concurrently if a GPU has several multiprocessors. Inside a kernel, the thread ID is not directly available to the programmer but can in many cases be computed as follows:

$$t := \langle \textit{block index} \rangle \cdot \langle \textit{block size} \rangle + \langle \textit{thread index} \rangle.$$

Blocks can, however, be set up in multiple dimensions which makes the above computation more complicated. Therefore, we make the simplifying assumption that blocks are one-dimensional and model the thread ID t as an additional input variable `t`. Given the block size, which is usually constant, the block index can be computed from t by

$$\langle \textit{block index} \rangle := t \ / \ \langle \textit{block size} \rangle$$

if needed.

Example. The *copy* kernel efficiently copies a large array `a` to `b`, where both `a` and `b` have size $N \geq 0$. Both the thread ID and the array size are given to the kernel as input variables `t`, `N`:

```
procedure copy(t, a, b, N) {
  if (t < N) { a[t] := b[t] }
}
```

△

Remark. In OpenCL, threads are called *work items* and blocks are called *work groups*.

2.4.2 Hardware Model

The hardware model specifies how a specific GPU architecture implements the execution of instructions in hardware. Because this is a vast subject, we only introduce three topics that are relevant to this thesis: SIMT architecture, shared memory, and global memory. Each of these topics are, respectively, associated to an important factor in kernel performance: branch divergence, shared memory bank conflicts, and global memory coalescing.

Since we use the CUDA programming model, we refer to a hardware model that is similar to the one implemented by Nvidia's Maxwell Architecture. All subsequent architectures (Pascal, Volta, Turing) are compatible enhancements of the Maxwell Architecture and are therefore also valid hardware models for our considerations, although some of the performance issues mentioned in this subsection might be nonexistent in later architectures.

SIMT Architecture and Branch Divergence

Like all Nvidia GPU architectures, the Maxwell Architecture follows a SIMT (Single Instruction, Multiple Threads) approach, meaning that the same instruction is issued to multiple threads at the same time. More precisely, a GPU consists of one or more multiprocessors, and every multiprocessor executes a given instruction in groups of 32 threads, so-called *warps* (or *wavefronts* in OpenCL).

Remark. Unlike thread blocks, warps are not visible to the programmer. However, threads belonging to the same warp have successive thread IDs, with the first thread in the first warp having ID zero. Hence, for $i \geq 0$, the i-th warp contains all threads with thread IDs

$$32i,\ 32i+1,\ \ldots,\ 32i+31.$$

In the Maxwell Architecture, warps execute in lockstep, which means that all threads in a warp have to execute the same instruction. This leads to a problem called *branch divergence* when two threads in the same warp take a different branch after a control flow instruction. Since the two threads cannot execute different instructions at the same time, the two branches are executed serially and, in most cases, lead to a lower instruction throughput. The threads that are not meant to execute the respective branch are called *inactive* and are masked by the responsible control flow instruction. Inactive threads do not commit any instruction until they are unmasked and become active again.

Example. Consider the following kernel:

```
procedure divergence(t) {
   // (1) branch divergence
   if (t % 2 = 0) then { ... } else { ... }
   // (2) no branch divergence
   if ((t / 32) % 2 = 0) then { ... } else { ... }
}
```

In (1), all threads with even IDs take the `then` branch, all threads with odd IDs take the `else` branch. Thus, the threads diverge inside their warp. In (2), every second warp takes the `then` branch, all other warps take the `else` branch. As a general rule, as soon as the branch condition only depends on `t` / 32, there can be no branch divergence. △

It is worth noting that performance is not necessarily affected by every diverging branch, as suggested above. An idiomatic example is when the kernel operates on arrays (e.g., vector addition) and does some bound-checking:

```
if (t < N) { c[t] := a[t] + b[t] }
```

If N is not a multiple of 32, the warp will contain some inactive threads, namely the ones with thread ID $t \geq N$. In spite of the diverging branch, the instruction throughput is optimal for this scenario since there is no `else` branch and, thus, no other instructions that have to be serialized.

Shared Memory and Bank Conflicts

Shared memory (called *local memory* in OpenCL) is fast memory located on all of the GPU's multiprocessors. In CUDA, data in shared memory is declared as `__shared__`. The Maxwell Architecture specifies that shared memory is organized in 32 memory banks where successive banks store successive 32-bit words (4 bytes) of data. All 32 memory banks can be accessed in parallel, meaning that all threads can be served at once if they access memory addresses located at pairwise different memory banks.

So-called *bank conflicts* can happen when memory banks are accessed simultaneously by two or more threads of the same warp. In case of a bank conflict, the conflicting memory accesses have to be serialized and lower the memory throughput.

Example. The following kernel shows two different access patterns to shared memory:

```
procedure $conflicts$(t) {
    __shared__ s[32];
    // (1) bank conflicts
    s[t * 32] := ...
    // (2) no bank conflicts
    ... := s[t]
}
```

(1) shows the worst case for shared memory accesses: all threads in the same warp write to the same memory bank, bank zero. All 32 accesses have to be serialized and we expect a 32-fold slow-down. In (2), all threads read from shared memory through a different memory bank. Therefore, no bank conflicts happen. △

There is a special case where no bank conflicts happen although two memory banks are accessed simultaneously, namely if the accessed address location is exactly the same. When multiple threads read from the same address, the memory bank broadcasts the value to all those threads at once. Similarly, when multiple threads concurrently write to the same address, the memory bank does not serialize all writes but arbitrarily executes a single write.

Global Memory and Coalescing

Unlike shared memory, *global memory* (marked as `__device__` in CUDA) is not located on the different multiprocessors but in the GPU's DRAM and is therefore much slower compared to shared memory. A warp can access global memory via 32-, 64-, or 128-byte memory transactions, each aligned to their size.

To speed up global memory accesses, a warp *coalesces* accesses to memory locations that lie in the same address range, aligned to 128 bytes at most, into one memory transaction. The fewer memory transactions are needed, the bigger the speed-up is in executing the memory access. Hence, coalescing increases the memory throughput.

Example. Consider this kernel, containing different accesses to global memory:

```
procedure $coalescing$(t) {
    __device__ g[4048];
    // (1) coalescing
    ... := g[t]
    // (2) no coalescing
    g[t * 128] := ...
}
```

In (1), all threads read from the global array `g` in a linear fashion. The indices all lie in a range aligned to 32. If, say, `g` is an array of 32-bit integers, all accessed memory locations lie in a range of 128 bytes and can be coalesced into one or two memory transactions at most. In (2), all global memory accesses go to memory locations that are at least 128 bytes apart from each other. This is the worst case because for every access, a new memory transaction has to be issued. △

Chapter 3

Approach

3.1 Problem Statement

In this thesis, we develop a static analysis for the following three hyperproperties related to GPU kernel performance: no branch divergence at control flow statements, no bank conflicts at shared memory accesses, and coalescing at global memory accesses. In contrast with hyperproperties like determinism or injectivity, these properties are not about the end result of a program or method, but about branch conditions and accessed memory addresses. We first give an intuition for all three hyperproperties and then discuss each of them in more detail below.

1. *No branch divergence.*
   A control flow statement (i.e., `if` or `while` statement) has diverging branches if two threads in the same warp disagree on the value of the branch condition and take a different execution path. Diverging control flow is bad for instruction throughput, hence, the goal of our analysis is, for every control flow statement, to prove that its branches are non-diverging or to report a possible branch divergence otherwise. The analysis is sound if and only if it never misses any diverging branches.
2. *No shared memory bank conflicts.*
   A shared memory bank conflict happens if two threads in the same warp access different memory addresses that map to the same shared memory bank. Bank conflicts reduce memory throughput, therefore, for every shared memory access, our analysis aims to prove that it is conflict-free or, otherwise, reports a possible bank conflict. If and only if it never misses a bank conflict, the analysis is sound.
3. *Global memory coalescing.*
   A global memory access is well coalesced if the total number of memory transactions produced by the warp's access pattern is not greater than the number of memory transactions produced by a linear access pattern. This is the case if, for all pairs of threads in the same warp, their access indices fall into the same range of size 32, starting at a multiple of 32. The converse is not true in general (i.e., we check a stronger statement), but this stronger statement has a form that is more convenient to prove in our analysis. For every global memory access, we either want to prove that it is well coalesced or report that it is possibly not. Our analysis is sound if it overapproximates global memory transactions.

These are 2-safety hyperproperties because they relate finite prefixes of two execution traces. The execution traces have to stem from two threads that belong to the same warp.

Before we formalize each hyperproperty, we therefore first formalize the requirement that two threads belong to the same warp.

We use the notation introduced in Section 2.3 and write superscript ^{0} and ^{1} to distinguish variables from the respective executions. Thus, let t^0 and t^1 be the thread IDs in both executions. All three hyperproperties can then be written as implications of the form

$$\mathsf{SameWarp}(t^0,\, t^1) \quad \Rightarrow \quad \mathsf{P} \tag{3.1}$$

where the predicate $\mathsf{SameWarp}(t^0,\, t^1)$ is true if and only if t^0's and t^1's respective threads belong to the same warp and P is some appropriate predicate that is defined differently for each hyperproperty. As described in Section 2.4.2, for any $q \geq 0$, the q-th warp contains all threads with IDs

$$32q,\ 32q+1,\ \ldots,\ 32q+31.$$

Therefore, we can define $\mathsf{SameWarp}$ as follows:

$$\mathsf{SameWarp}(t^0,\, t^1) \ := \ \exists\, q \geq 0 : \ t^0 = 32q + r^0 \ \wedge \ t^1 = 32q + r^1, \tag{3.2}$$

for some $0 \leq r^0 < r^1 < 32$. We impose $r^0 < r^1$ (and thus, $t^0 < t^1$) without loss of generality because we only consider executions from different threads and we choose execution 0 to be the execution with lower thread ID.

Now, we formalize the aforementioned hyperproperties and define the right-hand side P of the implication in (3.1) for each of them. Note that all hyperproperties are defined with respect to the Nvidia Maxwell Architecture. Different hardware models might require a different definition for each hyperproperty but this would exceed the scope of this thesis.

3.1.1 No Branch Divergence

For each control flow statement s with condition c, we try to prove that s does not cause a branch divergence. Since c can depend on the thread ID t, we write c as a function of the thread ID t. Then the control flow in s is non-diverging if and only if $c(t)$ evaluates to the same truth value for all t in the same warp.

Instead of quantifying over all t in the same warp, it is equivalent to say that in any two executions in the same warp, with respective thread IDs t^0 and t^1, the condition c evaluates to the same truth value. Hence, an `if` or `while` statement with condition $c(t)$ has no branch divergence if

$$\mathsf{SameWarp}(t^0,\, t^1) \quad \Rightarrow \quad c(t^0) \Leftrightarrow c(t^1). \tag{3.3}$$

As mentioned in Section 2.4.2, the occurrence of branch divergence does not always have a negative effect on instruction throughput. Conversely, the instruction throughput is optimal if no branch divergence happens. Therefore, the static analysis remains sound if we ignore special cases, like checks on array bounds, where performance is not affected.

Remark. When we say that an expression e is a function of t or can depend on t, this includes hidden dependencies, i.e., dependencies via other variables.

3.1.2 No Shared Memory Bank Conflicts

For each shared memory access, we try to prove the absence of bank conflicts. For convenience, we assume that shared memory is accessed through an index i on bank-sized memory chunks (4 bytes) such that two successive indices i, $i+1$ map to two successive memory banks. Since i can depend on the thread ID t, we write i as a function of t.

A shared memory access with index $i(t)$ is bank conflict-free if for any two thread executions in the same warp, the indices $i(t^0)$, $i(t^1)$ map to different memory banks, or if the indices are equal. The latter case allows for simultaneous reads and writes to the same address. Since there are 32 banks, $i(t^0)$ and $i(t^1)$ map to different banks if and only if they are not congruent modulo 32. Hence, a shared memory access with index $i(t)$ has no bank conflicts if

$$\mathsf{SameWarp}(t^0, t^1) \quad \Rightarrow \quad i(t^0) \not\equiv_{32} i(t^1) \vee i(t^0) = i(t^1). \tag{3.4}$$

3.1.3 Global Memory Coalescing

As with shared memory, we assume that we can access global memory through an index i which is a function of t. We waive the restriction that the stride S between two indices is 4 bytes. However, we still require that i and $i+1$ map to consecutive address chunks of size S.

For every global memory access, we try to show that it is well coalesced, which is true if the warp's accessed indices i lie in a range aligned to 32 and of size 32 at most. Intuitively, if all accessed indices are close to each other, in a range of size ≤ 32, then the accessed addresses in global memory must also be close to each other, in a range of size $\leq 32S$, where S is the stride between two consecutive indices.

Remark. The requirement that the accessed indices need to be aligned to 32 is actually stronger than necessary but lets the hyperproperty be expressed as a property of two execution traces and has a form that is convenient in our approach. A case where this leads to imprecision is when we have sparse accesses to memory chunks of 1 or 2 bytes but where these accesses are still in a range aligned to 128 bytes. However, this is not a soundness issue for our analysis and we estimate that in most reasonable scenarios, the checked property gives a good measure for how well global memory accesses are coalesced.

Formally, a global memory access with index $i(t)$ is well coalesced if

$$\mathsf{SameWarp}(t^0, t^1) \quad \Rightarrow \quad \left\lfloor \frac{i(t^0)}{32} \right\rfloor = \left\lfloor \frac{i(t^1)}{32} \right\rfloor. \tag{3.5}$$

Example. Suppose that, in some kernel, the j-th warp is accessing successive elements in an array of 32-bit integers, i.e., $S = 4$ bytes and $i(t) = t$ with $t \in \{32j, 32j+1, \dots, 32j+31\}$. Since $\lfloor \frac{i(t)}{32} \rfloor = j$ for all threads in the j-th warp, the implication in (3.5) holds and we expect a low number of memory transactions. The range of locations accessed by the warp is $32S = 128$ bytes. If additionally the address at $i(32j)$ is a multiple of 128 bytes, all memory accesses can be coalesced into a single 128-byte memory transaction. Otherwise we get two memory transactions which is still acceptable. $\triangle$

3.2 Method

In this section, we give an overview of our method to statically analyze the aforementioned GPU kernel performance hyperproperties. From a high-level perspective, we perform the static analysis in three steps:

1. *Adding specifications and assertions.*
   In the first step, we annotate the kernel with the specifications and assertions that are needed in order to check each hyperproperty. Recall from (3.1) that all hyperproperties in question can be formalized as implications with common left-hand side $\mathsf{SameWarp}(t^0, t^1)$. Hence, we add $\mathsf{SameWarp}(t^0, t^1)$ as a precondition to the beginning of the kernel and insert the right-hand sides of the respective implications (3.3), (3.4), and (3.5) as assertions before all corresponding statements. We state these preconditions and assertions as relational specifications about variables from two different executions in a way such that the product transformation in the second step can translate them correctly.

2. *Program transformation.*
   Secondly, the kernel is put in a form that allows us to check the hyperproperties more easily. In particular, we apply the k-product transformation introduced in Section 2.3.1 for $k = 2$, reducing the 2-safety hyperproperties to properties of single execution traces.

3. *Abstract interpretation.*
   In the final step, abstract interpretation is used to analyze the modified kernel and to verify the assertions added in the previous step. As abstract domains, we use a Binary Decision Tree (BDT) domain, which enables trace partitioning (see Section 2.3.2). We parametrize the BDT domain with a relational numerical abstract domain like Octagon or Polyhedra to relate variables from both execution traces.

3.3 Alignment Transformation

3.3.1 Motivation

We first discuss a direct approach to our task that and show why it fails. From there we show how to develop a more sophisticated approach and why.

Direct Approach

Recall from Section 3.1 that the thread ID t of the r-th thread in the q-th warp can be written as

$$t = 32q + r,$$

for $q \geq 0$ and $0 \leq r < 32$. Necessarily, two different threads in the same warp have equal q's and different r's. Hence, the idea of the direct approach is as follows.

Assume that not only the thread ID t is given as an input variable `t` to the kernel but also two input variables `q` and `r` which, respectively, represent q and r such that $t = 32q + r$. We proceed in four steps.

Step one. Make the above relationship explicit by adding the precondition

$$\mathtt{t} = 32 * \mathtt{q} + \mathtt{r} \ \wedge \ \mathtt{q} \geq 0 \ \wedge \ 0 \leq \mathtt{r} \ \wedge \ \mathtt{r} < 32$$

to the kernel. To state the assumption that both threads are different but belong to the same warp, add a second precondition, now using the transformed variables:

$$\mathtt{q}^0 = \mathtt{q}^1 \ \wedge \ \mathtt{r}^0 < \mathtt{r}^1.$$

Remember that $\mathtt{r}^0 < \mathtt{r}^1$ is equivalent to $\mathtt{r}^0 \neq \mathtt{r}^1$ without loss of generality because we can choose execution 0 to be the execution with the lower thread ID.

Step two. Next, apply a 2-product transformation to the given kernel such that every variable `x` in the original kernel is replaced by two variables $\mathtt{x}^0$, $\mathtt{x}^1$, each $\mathtt{x}^k$ for $k \in \{0, 1\}$ representing `x` in execution k. In particular, the kernel is given six input variables $\mathtt{t}^0$, $\mathtt{t}^1$, $\mathtt{q}^0$, $\mathtt{q}^1$, $\mathtt{r}^0$, $\mathtt{r}^1$. The precondition above is automatically duplicated by the 2-product transformation, and the new input variables are substituted accordingly.

Step three. For each hyperproperty, insert assertions before the corresponding statements to check whether this property holds:

- To check for branch divergence, add the following assertion before every `if` or `while` statement with condition $c(t)$, as well as at the end of the `while` loop:

$$(c(\mathtt{t}^0) \ \wedge \ c(\mathtt{t}^1)) \ \vee \ (\neg c(\mathtt{t}^0) \ \wedge \ \neg c(\mathtt{t}^1)).$$

- Insert the following assertion before every shared memory access with index $i(t)$ to check the absence of bank conflicts:

$$i(\mathtt{t}^0) \ \% \ 32 \neq i(\mathtt{t}^1) \ \% \ 32 \ \vee \ i(\mathtt{t}^0) = i(\mathtt{t}^1).$$

- Before every global memory access with index $i(t)$, assert

$$i(\mathtt{t}^0) \ / \ 32 = i(\mathtt{t}^1) \ / \ 32$$

  to check whether it is well coalesced.

Step four. After that, analyze the modified kernel by means of abstract interpretation, using a relational numerical abstract domain embedded in a BDT domain.

Shortcomings of the Direct Approach

Although the kernel contains all information that is needed to infer the desired hyperproperties, standard abstract domains like Octagon and Polyhedra are not able to prove them in any interesting cases and imprecisely report possible assertion violations that are false positives.

Some very simple examples where the analysis is precise when used with Polyhedra:

- non-diverging branch conditions $\texttt{t} < N$, where N is constant and a multiple of 32,
- conflict-free shared memory accesses with constant index $N \geq 0$,
- well coalesced global memory accesses with constant index $N \geq 0$.

In the following basic cases, the abstract domains Octagon, Polyhedra, and a reduced product of Polyhedra with support for linear congruences are imprecise:

- any non-diverging branch condition that is a function of $\texttt{t}$ / 32, e.g., $\texttt{t}$ / $32 = 0$,
- conflict-free shared memory accesses with indices $\texttt{t} + N$, for $N \geq 0$,
- well coalesced global memory accesses with index $\texttt{t}$.

We draw the conclusion that Octagon and Polyhedra are both unable to derive

$$(32 * \texttt{q} + \texttt{r}) \;/\; 32 = \texttt{q}, \qquad \text{given that} \quad \texttt{q} \geq 0 \quad \text{and} \quad 0 \leq \texttt{r} < 32,$$

and, similarly,

$$(32 * \texttt{q} + \texttt{r}) \;\%\; 32 = \texttt{r}, \qquad \text{given that} \quad \texttt{q} \geq 0 \quad \text{and} \quad 0 \leq \texttt{r} < 32.$$

These derivations are essential for the inference of GPU kernel performance hyperproperties. Since Octagon and Polyhedra are uncapable of doing these derivations, we make the derivations explicit in the kernel itself via the m-alignment transformation. While this might not be the only solution (see Section 3.6 for a discussion), it is clear that the direct approach using the aforesaid abstract domains is not sufficient to carry out our task.

General Idea of the m-Alignment Transformation

Given any integer-typed expression e where it is known that $e = 32e_q + e_r$ for expressions $e_q \geq 0$ and $0 \leq e_r < 32$, we want to derive that $e \;/\; 32 = e_q$ and $e \;\%\; 32 = e_r$.

Our solution is similar to the direct approach in that we introduce new variables $\texttt{q}$, $\texttt{r}$ for the thread ID variable $\texttt{t}$, however, in this case not only for $\texttt{t}$ but for all integer-typed program variables. Also, we do not keep any of the "original" variables but use the new variables instead. We can then redefine all statements, in particular also division by 32 and modulo 32 operations, in terms of these new variables.

Although we are only interested in division and modulo involving 32, due to the warp size in Nvidia GPUs being 32, we define the m-alignment transformation for all $m > 1$ instead of only $m = 32$, so that we can adapt the transformation to different warp sizes or use it for other applications (see discussion in Section 3.6). The next subsection provides some mathematical results that show why this is possible, and introduces important terminology.

3.3.2 m-Aligned Integer Representation

The following theorem is known as the (Euclidean) Division Algorithm and can be found together with a complete proof in [3].

Theorem (Division Algorithm)**.** *Given integers $n \in \mathbb{Z}$ and $m \in \mathbb{Z}$, with $m \neq 0$, there exist unique integers $q \in \mathbb{Z}$ and $r \in \mathbb{Z}$ satisfying*

$$n = mq + r, \qquad \text{and} \qquad 0 \leq r < |m|.$$

q and r are called, respectively, the quotient and remainder in the division of n by m.

Remark. In a closed form, q and r are given by

$$q = \operatorname{sgn}(m) \cdot \left\lfloor \frac{n}{|m|} \right\rfloor \qquad \text{and} \qquad r = n - m \left\lfloor \frac{n}{|m|} \right\rfloor .$$

For $m > 1$, a short proof is given in Section 3.5.

A consequence of this theorem is that for every non-zero integer m there is a bijection

$$\phi_m : \mathbb{Z} \mapsto \mathbb{Z} \times \{0, \ldots, |m| - 1\}, \qquad \text{given by} \qquad \phi_m(n) = (q, r),$$

where q and r are the quotient and remainder in the division of n by m. In other words, for a fixed m, the tuple (q, r) is a unique representation of n. This allows us to state the general idea from above more precisely:

The idea in our approach is to choose $m = 32$ and to rewrite a given kernel as an equivalent kernel where every integer-typed variable `x` *and constant k in the original kernel is replaced and represented, respectively, by two variables $\mathtt{x_q}$, $\mathtt{x_r}$ and constants k_q, k_r according to the bijection ϕ_m.*

Definitions. We call above an *m-alignment transformation* and the transformed kernel an *m-aligned kernel.* In a broader context, we speak of *m-aligned programs.* Let `x` be a variable in the original program with value n, and $\mathtt{x_q}$ and $\mathtt{x_r}$ the corresponding two variables in the m-aligned program with values q and r, respectively. We say that $\mathtt{x_q}$ *and* $\mathtt{x_r}$ *are an m-aligned representation of* `x` if $(q, r) = \phi_m(n)$, or, equivalently, if

$$n = mq + r \qquad \text{and} \qquad 0 \leq r < |m|.$$

The term *m-alignment* owes to the fact that for a fixed $m \neq 0$ and a given q, the set

$$S_m(q) = \{mq,\ mq + 1,\ \ldots,\ mq + |m| - 1\}$$

defines a range of size m that is aligned to a multiple of m.

All procedures, statements and expressions in m-aligned programs have to be rewritten appropriately such that they operate on quotient and remainder variables, and such that the semantics of the program does not change. The latter is achieved by satisfying the invariant that for any integer variable `x` in the original program, its two corresponding variables $\mathtt{x_q}$, $\mathtt{x_r}$ in the transformed program are an m-aligned representation of `x`, at any time during the program execution. We further discuss this invariant in Section 3.5.

Example. Let $m = 2$. In a 2-aligned program, the assignment $\mathtt{x} := 15$ is replaced by $\mathtt{x_q} := 7; \mathtt{x_r} := 1$. We can verify that $\mathtt{x_q}$ and $\mathtt{x_r}$ are indeed a 2-aligned representation of `x` by checking that $\mathtt{x} = 2\mathtt{x_q} + \mathtt{x_r}$, and $0 \leq \mathtt{x_r} < 2$. In a 2-aligned program, the remainder variable $\mathtt{x_r}$ tells us whether `x` in the original program is even or odd. $\triangle$

3.3.3 Problem Reformulation

After applying a 32-aligned program transformation, the three hyperproperties concerning branch divergence, shared memory bank conflicts, and global memory coalescing can be rephrased more simply. Recall from Section 3.1 that every hyperproperty can be analyzed using an implication of the form

$$\mathsf{SameWarp}(t^0,\, t^1) \quad \Rightarrow \quad \mathsf{P},$$

where t^0 and t^1 denote the respective thread IDs from the original kernel in both executions, the predicate $\mathsf{SameWarp}(t^0,\, t^1)$ is true if and only if t^0 and t^1 belong to the same warp, and P is a predicate defined separately for each hyperproperty. We can rewrite the definition of $\mathsf{SameWarp}$ in (3.2) using new variables t_q, t_r which are a 32-aligned representation of t (i.e., $t = 32t_q + t_r$):

$$\mathsf{SameWarp}(t_q^0,\, t_r^0,\, t_q^1,\, t_r^1) \;:=\; t_q^0 = t_q^1 \wedge t_r^0 < t_r^1. \tag{3.6}$$

No Branch Divergence

The implication that checks for branch divergence does not change much. Since we now write t in terms of t_q and t_r, we need to make every branch condition c a function of both t_q, t_r. Then an `if` or `while` statement with branch condition $c(t_q, t_r)$ has no branch divergence if

$$\mathsf{SameWarp}(t_q^0,\, t_r^0,\, t_q^1,\, t_r^1) \quad \Rightarrow \quad c(t_q^0,\, t_r^0) \Leftrightarrow c(t_q^1,\, t_r^1). \tag{3.7}$$

No Shared Memory Bank Conflicts

Similarly, we rewrite the shared memory access index i as a function of both t_q and t_r. In addition, since i is an integer, we split i into i_q and i_r such that they too are a 32-aligned representation of i. This allows us to rephrase the hyperproperty as follows: a shared memory access at index $i(t_q, t_r) = 32i_q(t_q, t_r) + i_r(t_q, t_r)$ is bank conflict-free if

$$\mathsf{SameWarp}(t_q^0,\, t_r^0,\, t_q^1,\, t_r^1) \quad \Rightarrow \quad i_r(t_q^0,\, t_r^0) \neq i_r(t_q^1,\, t_r^1) \vee i_q(t_q^0,\, t_r^0) = i_q(t_q^1,\, t_r^1). \tag{3.8}$$

Global Memory Coalescing

Using the same notation as before, we say that a global memory access at index $i(t_q, t_r) = 32i_q(t_q, t_r) + i_r(t_q, t_r)$ is well coalesced if

$$\mathsf{SameWarp}(t_q^0,\, t_r^0,\, t_q^1,\, t_r^1) \quad \Rightarrow \quad i_q(t_q^0,\, t_r^0) = i_q(t_q^1,\, t_r^1). \tag{3.9}$$

3.4 Construction of m-Aligned Programs

This section defines how to construct an m-aligned program P_m from a program P such that P_m and P are equivalent. Our definition of program equivalence and an informal correctness proof for our construction are given in Section 3.5.

We denote the m-alignment transformation of any language construct by $[\![\,\cdot\,]\!]_m$. The transformation parameter $m \in \mathbb{N} \setminus \{0\}$ remains constant during the entire program construction. As a simplification, we define the transformation for $m > 1$ only. Note that

$$\phi_1(n) = (n, 0), \qquad \text{and} \qquad \phi_{-m}(n) = (-q, r) \Leftrightarrow (q, r) = \phi_m(n).$$

In other words, for $m = 1$, the quotient q equals the original value n and the remainder r is always 0, and for negative m, q is simply the negation of the quotient corresponding to m's positive counterpart. Therefore, we think that the m-alignment transformation is of little interest if $m = 1$ or $m < 0$, and we assume from now on that $m > 1$.

Furthermore, we write $\equiv$ and $:\equiv$ for *syntactic* equality and definition, in order to make a clear distinction from *semantic* equality and definition which are denoted by $=$ and $:=$.

3.4.1 Programs and Procedures

A program is a set of one or more procedures. The alignment transformation is modular in the sense that every procedure can be transformed independently. Therefore, the transformation of a program consists in transforming all its procedures.

A procedure f with body s takes a list $\overline{\mathtt{x}}$ of zero or more parameters and returns a list $\overline{\mathtt{y}}$ of zero or more return variables. The transformation of such a procedure f is given by

$$[\![\, \texttt{procedure}\ f(\overline{\mathtt{x}})\ \texttt{returns}\ (\overline{\mathtt{y}})\ \{s\} \,]\!]_m \ :\equiv\ \texttt{procedure}\ f(\overline{\mathtt{x}}_{\mathtt{qr}})\ \texttt{returns}\ (\overline{\mathtt{y}}_{\mathtt{qr}})\ \{[\![\, s \,]\!]_m\},$$

where each parameter $\mathtt{x}$ in $\overline{\mathtt{x}}$ is replaced by two new variables $\mathtt{x_q}$ and $\mathtt{x_r}$ in $\overline{\mathtt{x}}_{\mathtt{qr}}$, and each return variable $\mathtt{y}$ in $\overline{\mathtt{y}}$ is replaced by $\mathtt{y_q}$ and $\mathtt{y_r}$ in $\overline{\mathtt{y}}_{\mathtt{qr}}$. The transformation of s ensures that if all parameters $\mathtt{x_q}$, $\mathtt{x_r}$ are m-aligned representations of $\mathtt{x}$, all return variables $\mathtt{y_q}$, $\mathtt{y_r}$ too are m-aligned representations of $\mathtt{y}$ when the procedure returns.

3.4.2 Statements and Expressions

The transformation of statements and expressions is intertwined. Hence, we describe the transformations for each statement type and add explanations on the transformation of expressions where it is appropriate.

Skip

The `skip` statement remains unchanged under transformation: $[\![\, \texttt{skip} \,]\!]_m \ :\equiv\ \texttt{skip}$.

Sequence

Sequences are transformed sequentially: $[\![\, s_1;\ s_2 \,]\!]_m \ :\equiv\ [\![\, s_1 \,]\!]_m\, ;\ [\![\, s_2 \,]\!]_m$

Assignment

The assignment statement $\mathtt{x} := e$ assigns an integer-typed expression e to a variable $\mathtt{x}$. Since $\mathtt{x}$ in the original program is replaced by two variables $\mathtt{x_q}, \mathtt{x_r}$ in the aligned program, both $\mathtt{x_q}$ and $\mathtt{x_r}$ will be assigned in the transformed assignment statement. Yet, the way in which an assignment is transformed depends on the structure of the right-hand side expression. We distinguish between *atomic* and *compound* expressions.

Atomic expressions (or *atoms*) are expressions that contain no proper subexpressions. Hence, atoms are either variables (denoted by $\mathtt{x}, \mathtt{y}$) or constant literals (denoted by k). To designate arbitrary atoms, we use the letters $\mathtt{a}, \mathtt{b}$ throughout this section.

Compound expressions, on the other hand, are of the form $e_1 \star e_2$, where $\star \in \{+, -, *, /, \%\}$ and e_1, e_2 are arbitrary integer-typed subexpressions. If all proper subexpressions of a compound expression are atomic, we call it *simple*, otherwise we call it *complex*.

First of all, we consider assignments with atomic right-hand sides. Then, for assignments with compound right-hand sides, we start by defining the transformation in the simple case, and thereafter reduce the complex to the simple case through an additional extraction step.

Atomic right-hand sides. The transformation of an assignment $\mathtt{x} := \mathsf{a}$ is given by

$$[\![\, \mathtt{x} := \mathsf{a} \,]\!]_m \;:\equiv\; \mathtt{x_q} := \mathsf{a_q};\; \mathtt{x_r} := \mathsf{a_r}. \tag{3.10}$$

We define the syntactic meaning of $\mathsf{a_q}$ and $\mathsf{a_r}$ for an arbitrary atom a as

$$\mathsf{a_q} :\equiv \begin{cases} \mathtt{y_q} & \text{if } \mathsf{a} \equiv \mathtt{y} \\ k_q & \text{if } \mathsf{a} \equiv k, \text{where } k_q = \left\lfloor \frac{k}{m} \right\rfloor, \end{cases} \tag{3.11}$$

$$\mathsf{a_r} :\equiv \begin{cases} \mathtt{y_r} & \text{if } \mathsf{a} \equiv \mathtt{y} \\ k_r & \text{if } \mathsf{a} \equiv k, \text{where } k_r = k - m \left\lfloor \frac{k}{m} \right\rfloor. \end{cases} \tag{3.12}$$

Simple compound right-hand sides. We consider assignments of the form $\mathtt{x} := \mathsf{a} \star \mathsf{b}$, for $\star \in \{+, -, *, /, \%\}$ and atoms a, b. Given m-aligned representations $\mathsf{a_q}, \mathsf{a_r}$ of a and $\mathsf{b_q}, \mathsf{b_r}$ of b, the goal is to define the m-aligned representation $\mathtt{x_q}, \mathtt{x_r}$ of $\mathtt{x}$ such that

$$m\mathtt{x_q} + \mathtt{x_r} \overset{!}{=} (m\mathsf{a_q} + \mathsf{a_r}) \star (m\mathsf{b_q} + \mathsf{b_r}). \tag{3.13}$$

In theory, we could define a trivial transformation that assigns $(m\mathsf{a_q} + \mathsf{a_r}) \star (m\mathsf{b_q} + \mathsf{b_r})$ to a new variable $\mathtt{x_{new}}$ (which is equivalent to the original program variable $\mathtt{x}$) and finds $\mathtt{x_q}, \mathtt{x_r}$ by applying the division algorithm to $\mathtt{x_{new}}$ (and thus to $\mathtt{x}$):

$$\begin{aligned} [\![\, \mathtt{x} := \mathsf{a} \star \mathsf{b} \,]\!]_m \;:\equiv\;& \mathtt{x_{new}} := (m * \mathsf{a_q} + \mathsf{a_r}) \star (m * \mathsf{b_q} + \mathsf{b_r}); \\ & \mathtt{if}\ (\mathtt{x_{new}} \,\%\, m \geq 0)\ \mathtt{then}\ \{ \\ & \quad \mathtt{x_q} := \mathtt{x_{new}} \,/\, m \\ & \}\ \mathtt{else}\ \{ \\ & \quad \mathtt{x_q} := \mathtt{x_{new}} \,/\, m - 1 \\ & \}; \\ & \mathtt{x_r} := \mathtt{x_{new}} - (m * \mathtt{x_q}) \end{aligned} \tag{3.14}$$

However, such a transformation is not of much interest. If a static analyzer could infer reasonable constraints for assignments transformed in that way, it would most likely also perform well on the original assignments. Aside from that, the transformation is long and cumbersome. The point of constructing an m-aligned program is to take advantage of mathematical properties of the m-aligned representation of integers, in order to transform assignments and arithmetic expressions in a way that helps the static analysis to produce more precise results. Therefore, there is a distinct transformation for every arithmetic operation. We derive the complete transformation of addition assignments as a showcase to illustrate this.

Recall from (3.13) above that we want to find $\mathtt{x_q}$, $\mathtt{x_r}$ such that

$$m\mathtt{x_q} + \mathtt{x_r} \overset{!}{=} (m\mathsf{a_q} + \mathsf{a_r}) + (m\mathsf{b_q} + \mathsf{b_r}) = m\underbrace{(\mathsf{a_q} + \mathsf{b_q})}_{\mathtt{x_q}} + \underbrace{(\mathsf{a_r} + \mathsf{b_r})}_{\mathtt{x_r}}.$$

We see immediately that this equation holds with $\mathtt{x_q} = \mathsf{a_q} + \mathsf{b_q}$ and $\mathtt{x_r} = \mathsf{a_r} + \mathsf{b_r}$. However, for m-alignment we also require that $0 \leq \mathtt{x_r} < m$. Since we assume that $0 \leq \mathsf{a_r}, \mathsf{b_r} < m$, we know that $0 \leq \mathsf{a_r} + \mathsf{b_r} < 2m$. So in the case where $\mathsf{a_r} + \mathsf{b_r} \geq m$, it suffices to subtract m from $\mathtt{x_r}$ and to add 1 to $\mathtt{x_q}$. The resulting transformation is

```
⟦ x := a + b ⟧_m  :≡  if (a_r + b_r < m) then {
                         x_q := a_q + b_q;
                         x_r := a_r + b_r
                      } else { // “overflow”
                         x_q := a_q + b_q + 1;
                         x_r := a_r + b_r − m
                      }.
```

The subtraction (and negation) transformation works analogously. For multiplication, we can use the distributive law:

$$\begin{aligned} m\mathtt{x_q} + \mathtt{x_r} &\overset{!}{=} (m\mathsf{a_q} + \mathsf{a_r})(m\mathsf{b_q} + \mathsf{b_r}) \\ &= m^2\mathsf{a_q}\mathsf{b_q} + m\mathsf{a_q}\mathsf{b_r} + m\mathsf{a_r}\mathsf{b_q} + \mathsf{a_r}\mathsf{b_r} \\ &= m\underbrace{\left(m\mathsf{a_q}\mathsf{b_q} + \mathsf{a_q}\mathsf{b_r} + \mathsf{a_r}\mathsf{b_q} + \left\lfloor \frac{\mathsf{a_r}\mathsf{b_r}}{m} \right\rfloor\right)}_{\mathtt{x_q}} + \underbrace{(\mathsf{a_r}\mathsf{b_r} \bmod m)}_{\mathtt{x_r}}. \end{aligned}$$

Thus, the multiplication transformation can be stated as

$$\begin{aligned} [\![\, \mathsf{x} := \mathsf{a} * \mathsf{b} \,]\!]_m \; :\equiv \; & \mathtt{x_q} = m * \mathsf{a_q} * \mathsf{b_q} + \mathsf{a_q} * \mathsf{b_r} + \mathsf{a_r} * \mathsf{b_q} + (\mathsf{a_r} * \mathsf{b_r}) \;/\; m; \\ & \mathtt{x_r} = (\mathsf{a_r} * \mathsf{b_r}) \;\%\; m. \end{aligned}$$

Remark. In the above transformation, we have translated the mathematical $\lfloor \frac{n}{m} \rfloor$ (integer division rounded towards negative infinity) as $n \;/\; m$ whose semantics is integer division rounded towards zero, and the mathematical $n \bmod m$ which is always positive, as $n \;\%\; m$ whose sign depends on n. For non-negative n, the two division and modulo operations are semantically equivalent. Since we know that $\mathsf{a_r} \geq 0$ and $\mathsf{b_r} \geq 0$, we have $\mathsf{a_r}\mathsf{b_r} \geq 0$ and our translation is therefore correct. For negative expressions, the respective operations have different semantics and must be used with care.

The transformations of division and modulo expressions have much in common. In general, they are both difficult to handle, except for the cases where the divisor and the modulus are, respectively, non-zero multiples of m:

$$m\mathtt{x_q} + \mathtt{x_r} \overset{!}{=} (m\mathsf{a_q} + \mathsf{a_r}) \;/\; (m\mathsf{b_q}) = \mathsf{a_q} \;/\; \mathsf{b_q} = m\underbrace{\left\lfloor \frac{\mathsf{a_q} \;/\; \mathsf{b_q}}{m} \right\rfloor}_{\mathtt{x_q}} + \underbrace{(\mathsf{a_q} \;/\; \mathsf{b_q} \bmod m)}_{\mathtt{x_r}},$$

$$m\mathtt{x_q} + \mathtt{x_r} \overset{!}{=} (m\mathsf{a_q} + \mathsf{a_r}) \;\%\; (m\mathsf{b_q}) = m\underbrace{(\mathsf{a_q} \;\%\; \mathsf{b_q})}_{\mathtt{x_q}} + \underbrace{\mathsf{a_r}}_{\mathtt{x_r}}.$$

There is another case that has simple transformations, namely when the divisor and the modulus divide m, respectively. However, for the remaining cases there is no simple transformation, and we have to handle these cases by computing quotient and remainder explicitly, as shown in (3.14). The complete transformations for division and modulo can be found in the appendix, along with all other transformations.

Complex compound right-hand sides. The more general case is when the right-hand side of the assignment is a complex compound expression $e_1 \star e_2$, where $\star \in \{+, -, *, /, \%\}$ and where e_1 or e_2 (or both) are non-atomic. The transformation of such an assignment is defined recursively and has the form

$$[\![\, \mathtt{x} := e_1 \star e_2 \,]\!]_m \;:\equiv\; [\![\, \mathtt{y_1} := e_1 \,]\!]_m \,;\; [\![\, \mathtt{y_2} := e_2 \,]\!]_m \,;\; [\![\, \mathtt{x} := \mathtt{y_1} \star \mathtt{y_2} \,]\!]_m \,,$$

where $\mathtt{y_1}, \mathtt{y_2}$ are fresh variables. Extracting the expressions e_1, e_2 to new variables $\mathtt{y_1}$, $\mathtt{y_2}$ recursively reduces the transformation of assignments with complex compound right-hand sides to the case with simple compound right-hand sides. Note that if one of e_1 or e_2 is already atomic, there is no need to extract the respective expression to a new variable, but it is not wrong to do so, either. In the implementation, we use this optimization and do not extract atomic expressions. In general, it may be possible to define the transformation with no extraction at all, however, this is likely to yield very complex and redundant expressions.

Procedure Call

In procedure calls $\overline{\mathtt{y}} := \mathtt{call}\ f(\overline{e})$, we need to provide two target variables $\mathtt{y_q}$, $\mathtt{y_r}$ for each original target variable $\mathtt{y}$ in $\overline{\mathtt{y}}$. As in the section on program and procedure transformation, we will denote this transformed set of target variables by $\overline{\mathtt{y}}_{\mathtt{qr}}$. Every argument expression e_i in $\overline{e}$ is extracted to a new variable $\mathtt{x_i}$ and transformed before calling the procedure. Calls to procedures with empty argument lists are thus transformed as

$$[\![\, \overline{\mathtt{y}} := \mathtt{call}\ f() \,]\!]_m \;:\equiv\; \overline{\mathtt{y}}_{\mathtt{qr}} := \mathtt{call}\ f(),$$

and calls to procedures with $k \geq 1$ arguments as

$$[\![\, \overline{\mathtt{y}} := \mathtt{call}\ f(\overline{e}) \,]\!]_m \;:\equiv\; [\![\, \mathtt{x_1} := e_1 \,]\!]_m \,;\; \ldots\; [\![\, \mathtt{x_k} := e_k \,]\!]_m \,;\; \overline{\mathtt{y}}_{\mathtt{qr}} := \mathtt{call}\ f(\overline{\mathtt{x}}_{\mathtt{qr}}),$$

where $\overline{\mathtt{x}}_{\mathtt{qr}} \equiv \mathtt{x_{1q}}, \mathtt{x_{1r}}, \ldots, \mathtt{x_{kq}}, \mathtt{x_{kr}}$. As before, if $e_i \equiv \mathsf{a}$ for some atom a, the extraction to a new variable $\mathtt{x_i}$ is optional and $\mathsf{a_q}$, $\mathsf{a_r}$ can be passed to the procedure directly.

If Statement

The `if` statement's `then` and `else` branches are statements and can be transformed accordingly. If the boolean branch condition does not contain any comparison of integer expressions, it remains unchanged. In most cases, however, the condition contains one or more expressions of the form $e_1 \circ e_2$, where e_1 and e_2 are integer-typed expressions and $\circ \in \{=, \neq, <, >, \leq, \geq\}$.

Similarly to the definition of the m-aligned assignment transformation, we start by stating the transformation for `if` statements with simple branch conditions and use this to recursively define the transformation for more complex branch conditions. More specifically, we begin with `if` statements whose branch condition is a comparison of atoms a, b. We then look at conditions that are comparisons of compound expressions, and lastly at conditions of any form, namely conditions containing boolean literals and multiple comparisons contained in logical expressions.

Comparisons of atoms. Consider `if` statements with branch condition $\mathsf{a} \circ \mathsf{b}$, where $\circ$ is a comparison operator as defined above. Their transformation is given by

$$[\![\, \texttt{if}\ (\mathsf{a} \circ \mathsf{b})\ \texttt{then}\ \{s_1\}\ \texttt{else}\ \{s_2\} \,]\!]_m \;:\equiv\; \texttt{if}\ ([\![\, \mathsf{a} \circ \mathsf{b} \,]\!]_m)\ \texttt{then}\ \{\, [\![\, s_1 \,]\!]_m \,\}\ \texttt{else}\ \{\, [\![\, s_2 \,]\!]_m \,\},$$

where comparisons are transformed as follows:

$$[\![\, \mathsf{a} \circ \mathsf{b} \,]\!]_m \;:\equiv\; \begin{cases} \mathsf{a_q} = \mathsf{b_q} \wedge \mathsf{a_r} = \mathsf{b_r} & \text{if } \circ \equiv = \\ \mathsf{a_q} \neq \mathsf{b_q} \vee \mathsf{a_r} \neq \mathsf{b_r} & \text{if } \circ \equiv \neq \\ \mathsf{a_q} < \mathsf{b_q} \vee (\mathsf{a_q} = \mathsf{b_q} \wedge \mathsf{a_r} \circ \mathsf{b_r}) & \text{if } \circ \in \{<, \leq\} \\ \mathsf{a_q} > \mathsf{b_q} \vee (\mathsf{a_q} = \mathsf{b_q} \wedge \mathsf{a_r} \circ \mathsf{b_r}) & \text{if } \circ \in \{>, \geq\}. \end{cases} \tag{3.15}$$

Conditions of any form. In the case where the branch condition c is a boolean expression that contains any number of comparisons on any kind of integer-typed expressions, we want to extract all non-atomic integer expressions from these comparisons and assign them to fresh variables. This step is similar to the extraction applied to assignments with complex compound right-hand sides, however, we do not extract any boolean expressions and must therefore proceed differently.

The extraction can be done by recursively traversing c, by "collecting" all extraction assignments and by replacing all comparisons with their transformed version. To this end, we define a new extraction operator $(\!|\, . \,|\!)_m$ for boolean expressions that takes a condition c and returns a tuple $(\tilde{s}, \tilde{c})$, where $\tilde{s}$ is the sequence of all collected assignment statements and $\tilde{c}$ is the transformed condition.

More formally,

$$(\!|\, c \,|\!)_m \;:\equiv\; \begin{cases} (\mathsf{y}_1 := e_1;\ \mathsf{y}_2 := e_2,\ [\![\, \mathsf{y}_1 \circ \mathsf{y}_2 \,]\!]_m) & \text{if } c \equiv e_1 \circ e_2 \text{ and } \circ \in \{=, \neq, <, >, \leq, \geq\}, \\ (\texttt{skip},\ c) & \text{if } c \in \{\texttt{true},\ \texttt{false}\}, \\ (\tilde{s}_0,\ \neg\tilde{c}_0) & \text{if } c \equiv \neg c_0, \\ (\tilde{s}_1;\ \tilde{s}_2,\ \tilde{c}_1 \wedge \tilde{c}_2) & \text{if } c \equiv c_1 \wedge c_2, \\ (\tilde{s}_1;\ \tilde{s}_2,\ \tilde{c}_1 \vee \tilde{c}_2) & \text{if } c \equiv c_1 \vee c_2, \end{cases}$$

where $(\tilde{s}_0, \tilde{c}_0) = (\!|\, c_0 \,|\!)_m$, $(\tilde{s}_1, \tilde{c}_1) = (\!|\, c_1 \,|\!)_m$, and $(\tilde{s}_2, \tilde{c}_2) = (\!|\, c_2 \,|\!)_m$.

Using this extraction operator, we can state the m-alignment transformation for `if` statements with an arbitrary branch condition c. Let $(\tilde{s}, \tilde{c}) = (\!|\, c \,|\!)_m$. The transformation is defined as

$$[\![\, \texttt{if}\ (c)\ \texttt{then}\ \{s_1\}\ \texttt{else}\ \{s_2\} \,]\!]_m \;:\equiv\; [\![\, \tilde{s} \,]\!]_m\,;\ \texttt{if}\ (\tilde{c})\ \texttt{then}\ \{\, [\![\, s_1 \,]\!]_m \,\}\ \texttt{else}\ \{\, [\![\, s_2 \,]\!]_m \,\}.$$

While Loop

The transformation of `while` loops is very similar to the transformation of `if` statements. The main difference is that all statements that stem from extracting the condition have to be added to the end of the loop body as well. Therefore, the transformation of `while` loops is defined as

$$[\![\, \texttt{while}\ (c)\ \texttt{do}\ \{s\} \,]\!]_m \;:\equiv\; [\![\, \tilde{s} \,]\!]_m\,;\ \texttt{while}\ (\tilde{c})\ \texttt{do}\ \{[\![\, s \,]\!]_m\,;\ [\![\, \tilde{s} \,]\!]_m\},$$

where $(\tilde{s},\ \tilde{c}) \equiv (\!|\, c \,|\!)_m$.

3.5 Correctness

This section reasons about the correctness of the m-aligned program construction, as defined in the previous section. We do not give a complete formal proof but rather prove some parts of the construction and provide informal reasoning for the rest. The m-aligned program construction is correct if the m-aligned program is equivalent to the original program in a way that is defined below.

3.5.1 Definition of Program Equivalence

Since the original and the transformed program do not operate on the same set of variables, it is important to define exactly what we mean when we say that a program P and its m-aligned version P_m are equivalent.

Intuitively, this is the case if for every variable `x` in P, P_m contains two variables $\mathtt{x_q}$, $\mathtt{x_r}$ such that the following holds: if P_m's inputs are m-aligned representations of P's inputs and if P and P_m are executed simultaneously step-by-step, then $\mathtt{x_q}$ and $\mathtt{x_r}$ are a valid m-aligned representation of `x` at every step, i.e., before and afterwards. One such step consists of executing the next statement s in P and its transformed statement(s) $[\![\, s \,]\!]_m$ in P_m.

In terms of small-step semantics, we can define program equivalence more precisely as follows. A configuration of the program P with set of integer variables $\mathbb{V}$ is a tuple $\langle s, \sigma \rangle$ where s is the sequence of remaining statements in P and the state function $\sigma : \mathbb{V} \mapsto \mathbb{Z}$ maps integer variables in P to their values. Similarly, let $\sigma_m : \mathbb{V}_m \mapsto \mathbb{Z}$ denote the state function of the m-aligned program P_m with set of integer variables $\mathbb{V}_m$. Let $\langle s, \sigma \rangle \rightarrow \langle s', \sigma' \rangle$ denote one step of the small-step transition between two program configurations. Arbitrary numbers of steps are denoted by $\rightarrow^*$. Also, let ϕ_m be as defined in Section 3.3.2.

Then P and P_m are equivalent if and only if for every possible configuration $\langle s, \sigma \rangle$ of P and every variable $\mathtt{x} \in \mathbb{V}$ with corresponding quotient and remainder variables $\mathtt{x_q}, \mathtt{x_r} \in \mathbb{V}_m$ in P_m, it holds that

$$\begin{gathered} \langle s, \sigma \rangle \rightarrow \langle s', \sigma' \rangle \quad \wedge \quad \phi_m(\sigma(\mathtt{x})) = (\tilde{\sigma}(\mathtt{x_q}), \tilde{\sigma}(\mathtt{x_r})) \\ \Rightarrow \\ \langle [\![\, s \,]\!]_m, \tilde{\sigma} \rangle \rightarrow^* \langle [\![\, s' \,]\!]_m, \tilde{\sigma}' \rangle \quad \wedge \quad \phi_m(\sigma'(\mathtt{x})) = (\tilde{\sigma}'(\mathtt{x_q}), \tilde{\sigma}'(\mathtt{x_r})). \end{gathered} \tag{3.16}$$

The only statements that alter the state function are procedure calls and assignment statements. We only need to consider the latter because procedures and their return variables are transformed correctly by assumption. We also have to make sure that the transformed program has the same control flow as the original program, such that there always exists a transition $\langle [\![\, s \,]\!]_m, \tilde{\sigma} \rangle \rightarrow^* \langle [\![\, s' \,]\!]_m, \tilde{\sigma}' \rangle$.

We claim without proof that the m-aligned program is equivalent to the original program if the transformation of assignments with atomic or simple compound right-hand sides, and of branch conditions with comparisons are correct. The correctness of all other transformations follows from the correctness of these basic transformations and from the fact that extracting an expression to a fresh variable does not change the semantics of a program.

In the following proofs, we do not explicitly refer to (3.16) or use any small-step notation in order to keep them short and comprehensible.

3.5.2 Correctness of the Comparison Transformation

The transformation for comparisons of atoms is given in (3.15). We briefly show that the transformed comparisons applied to integers a_q, a_r, b_q, b_r are equivalent to the original comparisons applied to integers a, b. To this end, we assume that $m > 1$ is fixed and that

$$\phi_m(a) = (a_q, a_r), \qquad \text{and} \qquad \phi_m(b) = (b_q, b_r).$$

More formally, we prove for every comparison operator $\circ \in \{=, \neq, <, >, \leq, \geq\}$ that $a \circ b \Leftrightarrow \llbracket\, a \circ b \,\rrbracket_m$:

- $a = b \quad \Leftrightarrow \quad a_q = b_q \wedge a_r = b_r$ *(equality):*
  This equivalence follows directly from the fact that ϕ_m is a bijection.
- $a \neq b \quad \Leftrightarrow \quad a_q \neq b_q \vee a_r \neq b_r$ *(disequality):*
  This equivalence can be obtained from the one above by applying De Morgan's laws.
- $a < b \quad \Leftrightarrow \quad a_q < b_q \vee (a_q = b_q \wedge a_r < b_r)$ *(strict inequality):*
  We show both directions separately:
  - $\Rightarrow$) We know that $ma_q + a_r = a < b = mb_q + b_r$. Suppose $a_q \neq b_q$. If $a_q > b_q$, $a - b = m(a_q - b_q) + (a_r - b_r) > m + (a_r - b_r) > m - b_r > 0$. This is a contradiction to $a < b$. Hence, $a_q < b_q$. Suppose $a_q = b_q$. By subtraction we get $a_r = a - ma_q < b - mb_q = b_r$.
  - $\Leftarrow$) Suppose $a_q < b_q$. Then $a = ma_q + a_r < m(a_q + 1) \leq mb_q \leq mb_q + b_r = b$. Suppose $a_q = b_q \wedge a_r < b_r$. Then $a = ma_q + a_r = mb_q + a_r < mb_q + b_r = b$.
- $a \leq b \quad \Leftrightarrow \quad a_q < b_q \vee (a_q = b_q \wedge a_r \leq b_r)$ *(non-strict inequality):*
  The proof is analogous to the previous proof for strict inequalities.
- The equivalences for the reverse inequalities follow directly from flipping a and b.

Since the transformed comparisons are equivalent when comparing integers, they are also equivalent when comparing integer-typed program variables and literals.

3.5.3 Correctness of the Assignment Transformation

We consider assignments where the left-hand side in the original program is a variable $\mathtt{x}$. The hypothesis is that, before the assignment, all transformed program variables are m-aligned representations of their respective corresponding variables in the original program. We show that, after the assignment, the variables $\mathtt{x_q}, \mathtt{x_r}$ in the transformed program are an m-aligned representation of $\mathtt{x}$.

Atomic right-hand sides. First, consider an assignment where the right-hand side is an atom. The transformation of such an assignment is defined in (3.10), (3.11) and (3.12). If the original right-hand side is a variable $\mathtt{y}$, then the transformed assignment is

$$\llbracket\, \mathtt{x} := \mathtt{y} \,\rrbracket_m \;:\equiv\; \mathtt{x_q} := \mathtt{y_q};\; \mathtt{x_r} := \mathtt{y_r}.$$

By assumption, $\mathtt{y_q}$ and $\mathtt{y_r}$ are an m-aligned representation of $\mathtt{y}$, hence, $\mathtt{x_q}$ and $\mathtt{x_r}$ are an m-aligned representation of $\mathtt{x}$. Otherwise, if the original right-hand side is a constant literal k, then the assignment is transformed as

$$\llbracket\, \mathtt{x} := k \,\rrbracket_m \;:\equiv\; \mathtt{x_q} := k_q;\; \mathtt{x_r} := k_r,$$

where $k_q = \left\lfloor \frac{k}{m} \right\rfloor$ and $k_r = k - m \left\lfloor \frac{k}{m} \right\rfloor$. In this case, $\mathtt{x_q}$ and $\mathtt{x_r}$ are an m-aligned representation of `x` because k_q and k_r are defined such that $\phi_m(k) = (k_q, k_r)$. This can be shown by proving (a) $mk_q + k_r = k$ and (b) $0 \leq k_r < m$.

$$\text{(a)} \quad mk_q + k_r = m \left\lfloor \frac{k}{m} \right\rfloor + k - m \left\lfloor \frac{k}{m} \right\rfloor = k,$$

$$\text{(b)} \quad 0 = k - m\frac{k}{m} \leq k_r = k - m \left\lfloor \frac{k}{m} \right\rfloor < k - m\frac{k-m}{m} = m.32$$

In (b) we used the fact that $\frac{k}{m} \geq \left\lfloor \frac{k}{m} \right\rfloor > \frac{k-m}{m}$ and that $m > 1$.

Simple compound right-hand sides. Recall that assignments with simple compound right-hand sides are of the form $\mathtt{x} := \mathtt{a} \star \mathtt{b}$ where `a` and `b` are atoms and $\star \in \{+, -, *, /, \%\}$. We have derived a specific transformation for each operation, and these can be found in the appendix.

Using the online frontend[1] of the Viper verification infrastructure [10], we are able to automatically verify the correctness of these transformations for assignments involving addition, subtraction, multiplication, negation, and the general cases for division and modulo. For the special cases of division and modulo (divisor/modulus is a multiple of m, divisor/modulus divides m), the verification times out.

As an example, Listing 3.1 shows the Viper program used for verifying the transformation of assignments with simple addition expressions.

Listing 3.1: Viper program proving the transformation of addition assignments correct.

```
method add(a: Int, a_q: Int, a_r: Int, b: Int, b_q: Int, b_r: Int, m: Int)
returns (x: Int, x_q: Int, x_r: Int)
    requires 0 <= a_r && a_r < m
    requires a == m * a_q + a_r
    requires 0 <= b_r && b_r < m
    requires b == m * b_q + b_r
    ensures 0 <= x_r && x_r < m
    ensures x == m * x_q + x_r
{
    x := a + b

    if (a_r + b_r < m) {
        x_q := a_q + b_q
        x_r := a_r + b_r
    } else {
        x_q := a_q + b_q + 1
        x_r := a_r + b_r - m
    }
}
```

[1]http://viper.ethz.ch/examples/blank-example.html

3.6 Discussion

In this section, we discuss the benefits and limitations of our approach. We also give examples of alternative approaches for our task or for the m-alignment transformation in general. Lastly, we discuss further applications for the m-alignment transformation.

3.6.1 Benefits of the m-Alignment Transformation

There are two essential benefits of our m-alignment transformation regarding the static analysis of GPU kernel performance hyperproperties:

1. Even if we use abstract domains that do not handle integer division and modulo operations well, we have a good chance of inferring the desired GPU kernel hyperproperties if we apply an alignment transformation with $m = 32$, i.e., if we choose m to be equal to the warp size. The m-alignment transformation is not more precise for all division and modulo operations except for division and modulo by 32 which are the operations by means of which our hyperproperties are defined and, thus, the operations we are most interested in.
2. We can statically analyze the transformed GPU kernel using *any* existing analysis framework that is capable of dealing with hyperproperties. In particular, we can apply a product program transformation and perform abstract interpretation on the twice transformed kernel.

The m-alignment transformation may have much more useful applications than only the static analysis of GPU kernel performance hyperproperties. We suggest some possible applications later in this section.

3.6.2 Limitations

The main drawback of our method is that m-aligned programs tend to become very long. Nested expressions get factored out and introduce two new variables per subexpression. Even a simple addition or subtraction of two variables becomes an `if` statement with two branches containing two assignments each, not to mention division or modulo expressions. In Chapter 4 on the implementation of our approach, we explain how the length of m-aligned programs can be reduced heuristically. In addition, after the m-alignment transformation, we are constructing a product program from the m-aligned program, additionally doubling the number of variables and statements.

Since the program to analyze becomes longer, the static analysis has to perform more steps. Furthermore, the number of program variables is increased and for standard relational abstract domains, the runtime complexity of each analysis step is superlinear in the number of program variables. Therefore, the m-alignment transformation leads to a substantial increase in runtime. In other words, we trade longer runtime for higher precision.

3.6.3 Alternative Approaches

One alternative approach to the m-aligned transformation of a kernel is presented in Section 3.3.1. This direct approach is too weak to prove the desired hyperproperties but can serve as a basis for more sophisticated approaches.

m-Alignment Transformation as Abstract Domain

It is also possible to create a specialized abstract domain that implements our m-aligned transformation through its abstract transformers. Such an abstract domain runs on the original program but internally uses the m-aligned representation of integers as an abstraction of the program variables. Additionally, the domain takes another abstract domain as parameter and applies it to its internal set of variables. Since the m-alignment transformation is performed by the abstract domain internally, this method probably makes the static analysis faster.

Note, however, that for the static analysis of GPU kernel hyperproperties, such a domain cannot be used in the same way as the explicit m-alignment transformation. Either it has to be used in conjunction with a specialized domain for hyperproperties, or it runs on the product program in which case the order of transformations is switched.

Direct Approach with Specialized Abstract Domain

As mentioned before, Octagons and Polyhedra are unable to derive that, given an integer $m > 1$, and non-constant integer expressions $e = me_q + e_r$, $e_q \geq 0$, and $0 \leq e_r < m$, the following equalities hold:

$$e \; / \; m = (me_q + e_r) \; / \; m = e_q \qquad \text{and} \qquad e \,\%\, m = (me_q + e_r) \,\%\, m = e_r.$$

Therefore, one alternative approach is to proceed as described in the direct approach but to use a new abstract domain, call it D, that is both relational and numerical, and that is able to make these derivations automatically. If, for example, the expressions e, e_q and e_r are some variables `t`, `q`, `r` as in the direct approach, the constraints $\texttt{t} = m * \texttt{q} + \texttt{r}$, $\texttt{q} \geq 0$, and $0 \leq \texttt{r} < m$ can be captured by the Polyhedra domain. Thus, such a new abstract domain D can perhaps be implemented as a wrapper around Polyhedra.

The main benefit of using a specialized abstract domain is that the m-alignment transformation is not needed and that the analysis can therefore run faster. The difficulty of this method lies in making it precise for a bigger variety of expressions, e.g., when e_q and e_r are not atomic but compound expressions.

3.6.4 Further Applications

Cache Performance Properties

Cache accesses (on CPUs or GPUs) are similar to both shared and global memory accesses on GPUs. On the one hand, like shared memory, caches are organized in cache banks which can serve multiple memory accesses simultaneously if the accessed memory locations map to different banks. On the other hand, dense memory accesses lead to a much bigger cache hit rate than sparse memory accesses, just as dense global memory accesses lead to better coalescing.

It is probable that the m-alignment transformation can be used to analyze the cache usage of a program and that our static analysis of GPU kernel performance hyperproperties can be applied to the analysis of cache performance properties with relatively few changes to the implementation.

Mathematical Proofs by Case Distinction

In some mathematical proofs, it is necessary to perform a case distinction on some integer n. A common case distinction is the distinction between n being a) even, or b) odd. In mathematical terms, we write

a) $n = 2q$ for some integer q, and b) $n = 2q + 1$ for some integer q.

Further case distinctions following the same principle are:

- Distinguish $n = 3q, 3q + 1, 3q + 2$ for some integer q.
- Distinguish $n = 4q, 4q + 1, 4q + 2, 4q + 3$ for some integer q.

  $\vdots$
- Distinguish $n = mq, mq + 1, mq + 2, \ldots, mq + m - 1$ for some integers $m > 1, q$.

The mathematical way of making such case distinctions is by rewriting n in terms of the constant $m > 1$, multiplied with some integer q, plus a constant rest r such that $0 \leq r < m$. It is easy to see that this rewriting step is analogous to our m-alignment transformation.

Next, we provide an example of such a proof by case distinction. Mathematically, it is straightforward to prove. Then we automate the proof using Viper and analyze two different programs, with and without m-alignment transformation.

Example. *Let n be an integer. Prove that if 3 does not divide n, then 3 divides $n^2 - 1$.*

Proof. We make a case distinction on n: a) $n = 3q$, b) $n = 3q + 1$, c) $n = 3q + 2$. In every case we need to prove that either $n = 0 \bmod 3$ or $n^2 - 1 = 0 \bmod 3$.

a) Let $n = 3q$ for some integer q. Then $n = 0 \bmod 3$.

b) Let $n = 3q + 1$ for some integer q. Then $n = 1 \bmod 3$, and thus also $n^2 = 1 \bmod 3$. We get $n^2 - 1 = 0 \bmod 3$ as desired.

c) Let $n = 3q + 2$ for some integer q. Then $n = 2 \bmod 3$, and thus also $n^2 = 1 \bmod 3$. Again, we get $n^2 - 1 = 0 \bmod 3$ which concludes this proof. △

We can write the statement above as an assertion that

$$n \,\%\, 3 = 0 \;\vee\; (n * n - 1) \,\%\, 3 = 0.$$

A simple Viper program containing this assertion can be found in Listing 3.2 below. Using this program, Viper is not able to prove the desired statement.

After applying an m-alignment transformation for $m = 3$, the resulting program looks like in Listing 3.3. We have manually simplified the modulo assignments to make the program more concise. For this program, Viper is capable of proving it without any further hints. A possible hint would be to assume $n_r = 0, 1, 2$ in three separate analyses, thus covering all cases. This can be done efficiently since the number of cases to consider is always constant in such proofs.

As an improvement, it would make sense to extend the analyzed language and the m-alignment transformation by a mathematical mod function or operator because this is more appropriate for the use in such proofs. The % modulo operator is only semantically equivalent to mod if the left-hand operand is non-negative or divisible by the modulus. The latter is true for our example, therefore we can use the % operator as a replacement for mod in the programs listed below.

Listing 3.2: Viper encoding of proof without m-alignment transformation.

```
method proof(n: Int)
{
    assert n % 3 == 0 || (n * n - 1) % 3 == 0
}
```

Listing 3.3: Viper encoding of proof with m-alignment transformation for $m = 3$.

```
method proof(n_q: Int, n_r: Int)
    requires 0 <= n_r && n_r < 3
{
    var w_q: Int
    var w_r: Int
    var x_q: Int
    var x_r: Int
    var y_q: Int
    var y_r: Int
    var z_q: Int
    var z_r: Int

    // w := n mod 3
    w_q := 0
    w_r := n_r

    // x := n * n
    x_q := 3 * n_q * n_q + n_q * n_r + n_r * n_q + n_r * n_r / 3
    x_r := n_r * n_r % 3

    // y := x - 1
    if (x_r - 1 >= 0) {
        y_q := x_q - 0
        y_r := x_r - 1
    } else {
        y_q := x_q - 0 - 1
        y_r := x_r - 1 + 3
    }

    // z := y mod 3
    z_q := 0
    z_r := y_r

    assert (w_q == 0 && w_r == 0) || (z_q == 0 && z_r == 0)
}
```

Chapter 4

Implementation

As part of this thesis, a static analysis for GPU kernel performance hyperproperties according to the approach described in the previous chapter was implemented. Our implementation is written in Scala and extends an existing framework [9] which is part of the Viper [10] verification infrastructure. This framework provides code for the static analysis of hyperproperties, using k-product programs and abstract interpretation with trace partitioning. An introduction of these topics can be found in the preliminaries in Chapter 2.

On a high level, we implement our analysis by adding two classes to the existing analysis framework. On the one hand, the AlignmentTransformer class takes a parameter $m > 1$ and a program in form of its abstract syntax tree (AST) on which it operates, and returns the AST of the m-aligned program. On the other hand, the GPUAnalysis class performs the whole static analysis of GPU kernel performance hyperproperties, using the AlignmentTransformer class for the 32-alignment transformation as well as other classes for the product transformation and the abstract interpretation with trace partitioning. In this chapter, we present these two main parts of our implementation. We focus on practical issues and various optimizations that are not explained in the approach.

4.1 m-Alignment Transformation

The m-alignment transformation, as described in Section 3.4, can be implemented for any C-like programming language. Since the m-alignment transformation is implemented within the Viper infrastructure, our implementation is designed to work with Viper programs. We support the m-alignment transformation of programs using any integer parameter $m > 1$.

This section briefly discusses two problems that can arise when constructing an m-aligned program and how to solve them. The first problem addresses the transformation of method pre- and postconditions which is special compared to the transformations described in the previous chapter. The second problem lies in potentially useless statements introduced by the m-alignment transformation.

4.1.1 Translating Pre- and Postconditions

Method pre- and postconditions are an important part of Viper programs because Viper is a language used for program analysis and verification purposes. A method's pre- and postconditions are expressions that specify constraints on the arguments passed to the method and guarantees for its return variables, respectively. They are different from assertions or loop invariants in that they do not contain local variables and that no statements can be put in front of them.

Therefore, compound expressions occuring in pre- and postconditions cannot be extracted to fresh variables as usual but have to be transformed inside the respective condition. The only place in which pre- and postconditions, and boolean expressions in general, contain integer-typed expressions are arithmetic comparisons. In the paragraph on the transformation of `if` statements in Section 3.4.2 we describe how such comparisons are translated. Recall that comparisons of atomic expressions like program variables or constant literals do not require an additional extraction step but can be transformed directly as a combination of multiple comparisons. Hence, every comparison c of atomic expressions that is contained in a pre- or postcondition can be transformed in this way.

If, however, for some comparison operator $\circ$, c is a comparison $c \equiv e_1 \circ e_2$ where either e_1 or e_2 or both are compound expressions, the comparison cannot be transformed as before without extracting the compound expression to a fresh variable. The straightforward solution to this is to replace all variables $\mathtt{x}$ in c with their explicit m-aligned representation $m * \mathtt{x_q} + \mathtt{x_r}$ in the aligned program. A comparison transformed in this way is less expressive and loses one great benefit of the m-alignment transformation, that being the possibility to operate on the quotient and remainder variables separately. However, it is a working solution and keeps the transformation simple. It is the solution we use in our implemention.

Example. We assume that a method is transformed using an m-alignment transformation with $m = 2$. If the original method with parameters `x` and `y` contains a precondition that requires

$$\mathtt{x} = 5 \ \wedge \ \mathtt{x} + \mathtt{y} = 6,$$

then we translate this precondition in the aligned program as follows:

$$\mathtt{x_q} = 2 \ \wedge \ \mathtt{x_r} = 1 \ \wedge \ (2 * \mathtt{x_q} + \mathtt{x_r}) + (2 * \mathtt{y_q} + \mathtt{y_r}) = 6. \qquad \triangle$$

An alternative, yet more complex solution would be to rewrite the precondition as several implications such that the implications imitate the transformation of the corresponding operation.

Example. In the same alignment transformation for $m = 2$, the precondition from the previous example could be rewritten as follows:

$$\begin{aligned} &\mathtt{x_q} = 2 \ \wedge \ \mathtt{x_r} = 1 \\ &\wedge \ (\mathtt{x_r} + \mathtt{y_r} < 2 \ \Rightarrow \ \mathtt{x_q} + \mathtt{y_q} = 3 \ \wedge \ \mathtt{x_r} + \mathtt{y_r} = 0) \\ &\wedge \ (\mathtt{x_r} + \mathtt{y_r} \geq 2 \ \Rightarrow \ \mathtt{x_q} + \mathtt{y_q} + 1 = 3 \ \wedge \ \mathtt{x_r} + \mathtt{y_r} - 2 = 0). \end{aligned}$$

Compare this transformation to the transformation of an assignment $\mathtt{x} := \mathsf{a} + \mathsf{b}$ for $m = 2$ as described in Section 3.4.2. $\triangle$

For more complex preconditions, this solution will result in long and complicated implications chains. For simplicity and performance reasons, we use the previous solution instead.

4.1.2 Simplification of Redundancies

The alignment transformation can create redundant statements and expressions that introduce an avoidable overhead to the program execution or analysis. We simplify these redundancies by applying a set of simplification rules after the transformation. This is primarily a performance optimization, meant to speed up the static analysis. In the evaluation of our static analysis which can be found in the next chapter, we compare the runtimes of the analysis with and without this simplification.

Note that in most cases, these simplifications do not only apply to m-aligned GPU kernels but to any m-aligned program that contains redundant statements and expressions due to the m-alignment transformation.

Artificial examples of redundant expressions are operations on two integer literals, for instance in an assignment $\mathtt{x} := 1 + 1$. Without any simplifications, this assignment is transformed as

$$\begin{aligned} [\![\, \mathtt{x} := 1 + 1 \,]\!]_m \;\equiv\; & \mathtt{if}\ (1 + 1 < m)\ \mathtt{then}\ \{ \\ & \quad \mathtt{x_q} := 0 + 0; \\ & \quad \mathtt{x_r} := 1 + 1 \\ & \}\ \mathtt{else}\ \{\ //\ \text{``}\mathtt{overflow}\text{''} \\ & \quad \mathtt{x_q} := 0 + 0 + 1; \\ & \quad \mathtt{x_r} := 1 + 1 - m \\ & \}. \end{aligned}$$

For a concrete m, say $m = 2$, the transformation can be simplified to

$$\mathtt{x_q} := 1;\ \mathtt{x_r} := 0,$$

either by evaluating all constant expressions and removing unreachable code, or by simplifying the original assignment to $\mathtt{x} := 2$ in the first place. The redundancy in this example is caused by a redundancy in the original program. However, this is not always the case. Let $m = 5$ and consider the statement $\mathtt{x} := \mathtt{y} + 10$ whose alignment transformation is given by

$$\begin{aligned} [\![\, \mathtt{x} := \mathtt{y} + 10 \,]\!]_5 \;\equiv\; & \mathtt{if}\ (\mathtt{y_r} + 0 < 5)\ \mathtt{then}\ \{ \\ & \quad \mathtt{x_q} := \mathtt{y_q} + 2; \\ & \quad \mathtt{x_r} := \mathtt{y_r} + 0 \\ & \}\ \mathtt{else}\ \{\ //\ \text{``}\mathtt{overflow}\text{''} \\ & \quad \mathtt{x_q} := \mathtt{y_q} + 2 + 1; \\ & \quad \mathtt{x_r} := \mathtt{y_r} + 0 - 5 \\ & \}. \end{aligned}$$

Since we know that $\mathtt{y_r} < m = 5$, the code in the `else` branch is unreachable and can be removed, just like the branch condition which always evaluates to true. Furthermore, any addition with zero is redundant and can be removed as well. The simplified statement is thus

$$\mathtt{x_q} := \mathtt{y_q} + 2;\ \mathtt{x_r} := \mathtt{y_r}.$$

We now state a set of rules which are not meant to be complete but to cover the basic redundant statements and expressions. We use the symbol $\models$ to denote a simplification rule, e.g., we write $s \models s'$ if the statement s can be simplified to the statement s'. We can simplify statements and both integer and boolean expressions. The simplification makes most sense if it applies the simplification rules recursively, i.e., before an expression is simplified, all its subexpressions are simplified first. Similarly, before a statement is simplified, all the statements and expressions it consists of are simplified first.

Arithmetic Expressions

Any arithmetic operation between two constant literals is redundant and can be simplified by directly replacing the expression with the evaluated constant. We do not state this simplification explicitly.

Let e be an arbitrary integer-typed expression. We have the following rules:

- *Addition:*
  $e + 0 \models e, \quad 0 + e \models e.$
- *Subtraction/Negation:*
  $e - 0 \models e, \quad 0 - e \models -e, \quad -(-e) \models e.$
- *Multiplication:*
  $e * 0 \models 0, \quad 0 * e \models 0, \quad e * 1 \models e, \quad 1 * e \models e, \quad e * -1 \models -e, \quad -1 * e \models -e.$
- *Division:*
  $0 \,/\, e \models 0, \quad e \,/\, 1 \models e, \quad e \,/\, -1 \models -e.$
- *Modulo:*
  $e \,\%\, 1 \models 0, \quad e \,\%\, -1 \models 0.$

Comparisons

As before, any comparison between two constant literals can be replaced by the direct evaluation *true* or *false* for this comparison.

Let $\mathtt{x_r}$ be an arbitrary remainder variable in an m-aligned program. We write $k_{<c}$, $k_{\leq c}$, $k_{>c}$ and $k_{\geq c}$ to denote all integers k such that the subscript condition holds for k and the specified constant c. Note that m denotes the parameter of the m-alignment transformation. The follwing rules apply:

- *Less or equal than:*
  $k_{\leq 0} \leq \mathtt{x_r} \models true, \quad \mathtt{x_r} \leq k_{\geq m-1} \models true, \quad k_{\geq m} \leq \mathtt{x_r} \models false, \quad \mathtt{x_r} \leq k_{<0} \models false.$
- *Less than:*
  $k_{<0} < \mathtt{x_r} \models true, \quad \mathtt{x_r} < k_{\geq m} \models true, \quad k_{\geq m-1} < \mathtt{x_r} \models false, \quad \mathtt{x_r} < k_{\leq 0} \models false.$

For *greater or equal than* and *greater than*, the same simplifications apply with flipped left- and right-hand sides.

A more interesting simplification applicable to GPU kernels takes advantage of the fact that we model the thread ID t as an additional input variable `t` which is non-negative. In the m-aligned kernel, `t` is replaced by $\mathtt{t_q}$ and $\mathtt{t_r}$. Since the division and modulo transformations are `if` statements that contain branch conditions with non-negativity constraints, we can simplify these transformed statements a lot if we simplify comparisons of the form $\mathtt{t_q} \geq 0 \models true$.

Boolean Logic Expressions

Let b be an arbitrary boolean expression. For logical boolean expressions, we can use the following simplifications:

- *Negation:*
  $\neg true \models false, \quad \neg false \models true.$

- *Conjunction:*
  $b \wedge false \models false, \quad false \wedge b \models false, \quad b \wedge true \models b, \quad true \wedge b \models b.$
- *Disjunction:*
  $b \vee false \models b, \quad false \vee b \models b, \quad b \vee true \models true, \quad true \vee b \models true.$

Statements

Given arbitrary statements s, s_1, s_2, and an arbitrary variable `x`, we can state the following simplification rules for statements:

- *Assignment:*
  $\texttt{x} := \texttt{x} \models \texttt{skip}$.
- *Sequence:*
  $s; \texttt{skip} \models s, \quad \texttt{skip}; s \models s.$
- *If statement:*
  $\texttt{if } (true) \texttt{ then } \{s_1\} \texttt{ else } \{s_2\} \models s_1, \quad \texttt{if } (false) \texttt{ then } \{s_1\} \texttt{ else } \{s_2\} \models s_2.$
- *While loop:*
  $\texttt{while } (false) \texttt{ do } \{s\} \models \texttt{skip}.$

4.2 GPU Kernel Analysis

In order to statically analyze performance hyperproperties for GPU kernels, we have implemented a static analysis that takes as input one or more files with GPU kernels encoded in Viper, and that outputs a detailed report for every kernel, indicating all possible performance issues.

The approach to this static analysis is described in detail in Chapter 3. In this section, we cover different implementation-specific topics, including precision and performance optimizations.

4.2.1 Check Keyword

The hyperproperties that we are analyzing have to be checked right before the concerned statements, i.e., for non-diverging control flow before the `if` or `while` statement, for conflict-free shared memory accesses and well-coalesced global memory accesses before the respective memory access. We check a hyperproperty by verifying the right-hand sides of the implications (3.7), (3.8), and (3.9) in Section 3.3.3, after having assumed the corresponding left-hand side in the kernel preconditions.

The obvious method to verify an assumption about the program state at some point in the program is to insert an assertion and to let the static analysis check whether the assertion always holds or whether it might fail, in which case we report a possible assertion violation. In Viper, the keyword introducing an assertion is `assert`.

The problem with using assertions to check hyperproperties is that they are not intended to be used to test expressions that might be false, but rather for sanity checks that should never fail. In fact, if the program analyzed through abstract interpretation contains an assertion, the abstract domain verifies it, reports a possible violation, and subsequently assumes that it holds. We, however, do not want the abstract domain to assume our hyperproperty checks because then, property violations from two identical branch conditions or memory accesses might be reported only once by the static analysis.

Therefore, we extend Viper with the keyword `check` that introduces an expression that should be checked by the static analysis but is not required to be true. As with assertions, the static analysis should report a possible check violation if the checked expressions might be false. However, the difference is that the checked expression must not be assumed in the subsequent program states. Additionally, the `check` statement is given the line number of the statement from which the hyperproperty check originates. Thus, when reporting a possible check violation, the analysis can specify in which line in the kernel the hyperproperty might not hold.

4.2.2 Equality Assumptions

There is a frequent setting which is a consequence of the product construction and that makes the Octagons domain and sometimes the Polyhedra domain lose knowledge about the information flow during the analysis of product programs. The scenario in which precision is lost can be described in three steps:

1. Let $\mathtt{x}$ be some variable in the original program and $\mathtt{x}^1, \mathtt{x}^2, \ldots, \mathtt{x}^\mathtt{k}$ be k copies of $\mathtt{x}$ in the product program. Assume that at some point in the program, the abstract domain knows that
$$\mathtt{x}^1 = \mathtt{x}^2 = \ldots = \mathtt{x}^\mathtt{k}.$$
2. Assume that, in terms of the original program, the next statement to analyze is an assignment to a different variable, $\mathtt{y}$, and that the right-hand side expression e of this assignment only depends on variables with equal values and on constants, e.g., $\mathtt{y} := 2 * \mathtt{x} + 1$. Since such an expression e is equal in all executions, it follows that after this statement, the variable $\mathtt{y}$ is equal in all executions, given that all executions executed the assignment.
3. Abstract interpretation with Octagons is in most cases unable to derive that $\mathtt{y}$ is equal in all executions. To illustrate what happens, we consider the example above, an assignment $\mathtt{y} := 2 * \mathtt{x} + 1$. In the k-product program, this assignment is translated as a sequence of assignments
$$\begin{gathered}\mathtt{if}\ (\mathtt{p}^1)\ \mathtt{then}\ \{\ \mathtt{y}^1 := 2 * \mathtt{x}^1 + 1\ \},\\ \mathtt{if}\ (\mathtt{p}^2)\ \mathtt{then}\ \{\ \mathtt{y}^2 := 2 * \mathtt{x}^2 + 1\ \},\\ \vdots\\ \mathtt{if}\ (\mathtt{p}^\mathtt{k})\ \mathtt{then}\ \{\ \mathtt{y}^\mathtt{k} := 2 * \mathtt{x}^\mathtt{k} + 1\ \},\end{gathered}$$
where $\mathtt{p}^\mathtt{i}$ is the activation variable in the i-th execution. We assume that all activation variables are set to *true*. There are two reasons why the Octagon domain cannot derive that $\mathtt{y}$ is equal in all executions:
   (a) because Octagons can only store constraints of the form $\pm\mathtt{x} \pm \mathtt{y} \leq c$, and
   (b) because the assignments in the product program are interpreted sequentially and not all at once.

   It is the combination of both (a) and (b) that prevents Octagons from inferring the equality constraints for $\mathtt{y}$ between the different assignments. Note that Polyhedra is better in keeping track of such constraints but also fails to do so for more complex assignments like $\mathtt{y} := \mathtt{x}\ \%\ 2$.

For the static analysis of GPU kernels, we need to know whether epressions or variables are equal in several executions in order to prove the absence of branch divergences. To decide whether, for an `if` or `while` statement, all threads in a warp have the same control flow, we have to check whether a given branch condition evaluates to the same value in all threads belonging to the same warp. To this end, let $\mathtt{t_q}$ and $\mathtt{t_r}$ be the m-aligned representation of the thread ID variable $\mathtt{t}$ in an m-aligned kernel with $m = 32$ and consider two executions from threads in the same warp. Recall that in such a setting, all input variables, and in particular $\mathtt{t_q}$, are equal in both executions, except $\mathtt{t_r}$. To prove non-divergence, it is sufficient to show that the branch condition only depends on variables with equal values in both executions like $\mathtt{t_q}$.

Since Octagons runs much faster than Polyhedra with a big number of program variables, we want to benefit from the perfomance of the Octagons domain while still being able to infer equality constraints. In our implementation of the alignment transformation, we therefore optionally insert relational assumptions which tell Octagons and Polyhedra that assignments with equivalent right-hand sides result in equivalent left-hand sides, given that the assignments are executed in both threads. More formally, let $\mathsf{Equal}(\mathtt{x})$ be a predicate defined as

$$\mathsf{Equal}(\mathtt{x}) \;:=\; \mathtt{x}^1 = \mathtt{x}^2 = \ldots = \mathtt{x}^k,$$

and consider an assignment $\mathtt{y} := e$ after which we then insert the following assumption:

$$(\forall\ \mathtt{x} \text{ occurring in } e:\ \mathsf{Equal}(\mathtt{x})) \quad \Rightarrow \quad \mathsf{Equal}(\mathtt{y}).$$

In Viper, such an assumption can be stated using the keyword `inhale`.

We also add a similar hint for injective operations like addition and subtraction that if all variables in the assignment's right-hand side are equal except one, then the left-hand side is different in all executions. For the analysis of GPU kernels, this is useful to infer the absence of bank conflicts in shared memory accesses.

More equality assumptions could be given to the abstract domain for procedure calls, however, we have not implemented this because GPU kernels do not usually contain many procedure calls.

Remark. As mentioned before, this precision issue is common to all k-product programs. Therefore, we suggest that in the future, the precision optimization described above is moved to the code logic that implements the product transformation, as an optional optimization, and that it is generalized to support any $k \geq 2$.

4.2.3 Program Slicing

Since GPU kernels tend to contain many complex computations that do not affect branch conditions or memory access indices, the performance of the static analysis can be improved if all irrelevant statements in the kernel are removed. It is important to clarify the difference with the optimization proposed in Section 4.1.2. With program slicing, we consider statements that are not redundant but irrelevant for the analysis of GPU kernel performance hyperproperties, i.e., all statements on which branch conditions and memory access indices are not dependent, in particular all assignments to variables that are not contained in the dependency graph of any branch condition or memory access index.

We have not implemented automatic program slicing in our framework but we have extensively applied it manually while preparing sample GPU kernels for the evaluation of our analysis. Automatic program slicing can be done efficiently and is a possible improvement of our analysis.

Chapter 5

Evaluation

The evaluation of our static GPU kernel analysis is based on the analysis of 13 different kernels. 12 out of the 13 kernels were originally written in CUDA C and CUDA C++, one of them was written in OpenCL. Three kernels are taken from the CUDA C Programming Guide [11], an Nvidia tutorial [8], and the Nvidia developer forum [6]. The remaining ten kernels are real kernels used in practice and originate from the benchmark set of the GPUVerify project [1]. We have translated all kernels to Viper and manually applied the program slicing mentioned in Section 4.2.3.

The transformed kernels are between 8 and 93 lines of code (LOC) in size, three of them contain `while` loops, and all kernels implement functions of different computational complexity. In total, the kernels contain

- 39 branch conditions,
- 46 shared memory accesses, and
- 54 global memory accesses.

In this chapter, we evaluate our static analysis of GPU kernel performance hyperproperties on these kernels using various measures: precision, accuracy, transformed program size, and runtime. We analyze each kernel with and without the following optimizations:

E : Insertion of equality assumptions that help the abstract domain to maintain equality constraints during the abstract interpretation of arithmetic expressions in product programs, thus making the analysis more precise. This optimization is presented in Section 4.2.2.

S : Simplification of redundant statements and expressions introduced by the m-alignment transformation (see Section 4.1.2). This is a performance optimization.

We have analyzed the kernels using different relational numerical abstract domains and implementations: Sample Octagons, Apron Octagons, and Apron Polyhedra. However, as we will explain in Section 5.3, the latter two are not appropriate for our purposes. In the following, we therefore focus on results coming from the evaluation with Sample Octagons.

5.1 Precision and Accuracy

Intuitively, *precision* and *accuracy* are measures that indicate how well the analysis can decide a specific hyperproperty. Our static analysis can be seen as a binary classifier that divides branch conditions and memory accesses into two classes: *positive* (the corresponding hyperproperty is violated), and *negative* (the hyperproperty holds). In binary classification, precision and accuracy are defined as follows:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}, \quad \text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{P} + \text{N}},$$

where TP (True Positives) and TN (True Negatives) are the number of correctly classified negatives and positives, respectively, and P and N are the number of actual positives and negatives.

Since the analysis is sound, the number of false negatives (FN), i.e., the number of missed hyperproperty violations is zero. The analysis is precise if it yields a relatively small number of false alarms (FP), i.e., if it can prove a hyperproperty whenever it holds, and the analysis is accurate if it correctly classifies a relatively large number of negatives *and* positives.

Table 5.1 gives an overview of the number of branch conditions, shared memory accesses, and global memory accesses, and serves as a reference for the following evaluation tables. The true number of positives and negatives was determined manually.

Kernel	**BD (P+N)**	**SMBC (P+N)**	**GMC (P+N)**
InlinePTX	1 (1+0)	0 (0+0)	1 (0+1)
Divergence	2 (1+1)	0 (0+0)	2 (0+2)
DxInterOp	0 (0+0)	0 (0+0)	8 (2+6)
UniformUpdate	1 (1+0)	1 (0+1)	3 (0+3)
ReduceKernel	1 (1+0)	0 (0+0)	2 (0+2)
VectorAdd	1 (1+0)	0 (0+0)	3 (0+3)
Reverse	1 (1+0)	2 (0+2)	2 (0+2)
ScanExclusive	2 (2+0)	0 (0+0)	3 (2+1)
Particles	5 (5+0)	3 (0+3)	8 (2+6)
FastWalsh	0 (0+0)	0 (0+0)	8 (0+8)
Reduction	16 (3+13)	20 (0+20)	3 (2+1)
ShflScan	8 (4+4)	4 (0+4)	3 (0+3)
MergeSort	1 (0+1)	16 (4+12)	8 (0+8)

Table 5.1: Total number of branch conditions and shared/global memory accesses for each analyzed kernel, corresponding to the hyperproperties branch divergence (BD), shared memory bank conflicts (SMBC), and global memory coalescing (GMC), together with the number of positives and negatives (P+N).

The precision of our analysis evaluated on each kernel is presented in Table 5.2 and the accuracy in Table 5.3. Interesting results are highlighted using bold font. The analysis of the MergeSort kernel runs out of memory, which is why we have no results for this kernel. If the MergeSort kernel is not considered, our evaluation yields an average precision of 81.6% and an average accuracy of 87.1% for the analysis with the insertion of equality assumptions (**E**). Without this optimization, the analysis is precise in 73.8% and accurate in 82.7% of the cases, on average.

Interpretation

In the Divergence, Reduction, and ShflScan kernels, the analysis results are more precise for the branch divergence hyperproperty when enabling the optimization **E**. The reason for that is that these kernels contain branch conditions which, in the 32-aligned kernel, only depend on the quotient variable of the thread ID which is equal in both

Kernel	BD		SMBC		GMC		Total Precision	
	E off	**E on**	**E off**	**E on**	**E off**	**E on**	**E off**	**E on**
InlinePTX	1/1	1/1	0/0	0/0	0/0	0/0	100.0%	100.0%
Divergence	**1/2**	**1/1**	0/0	0/0	0/0	0/0	50.0%	100.0%
DxInterOp	0/0	0/0	0/0	0/0	2/2	2/2	100.0%	100.0%
UniformUpdate	1/1	1/1	0/0	0/0	0/0	0/0	100.0%	100.0%
ReduceKernel	1/1	1/1	0/0	0/0	**0/1**	**0/1**	50.0%	50.0%
VectorAdd	1/1	1/1	0/0	0/0	0/0	0/0	100.0%	100.0%
Reverse	1/1	1/1	0/0	0/0	0/0	0/0	100.0%	100.0%
ScanExclusive	2/2	2/2	0/0	0/0	2/2	2/2	100.0%	100.0%
Particles	5/5	5/5	0/0	0/0	2/2	2/2	100.0%	100.0%
FastWalsh	0/0	0/0	0/0	0/0	**0/8**	**0/8**	0.0%	0.0%
Reduction	**3/12**	**3/6**	0/0	0/0	2/2	2/2	35.7%	62.5%
ShflScan	**4/8**	**4/6**	0/0	0/0	0/0	0/0	50.0%	66.7%
MergeSort	Analysis runs out of memory before termination.							

Table 5.2: Precision as a fraction TP/(TP+FP) with and without the optimization **E** for each hyperproperty: branch divergence (BD), shared memory bank conflicts (SMBC), and global memory coalescing (GMC). The total precision is computed per kernel over all three hyperproperties.

executions (i.e., there is no branch divergence) but are too complex for the abstract domain to know that thus the whole branch condition has the same truth value in both executions. However, with the insertion of equality assumptions we explicitly state that this equality holds such that the analysis is precise enough to prove the absence of branch divergence.

In the kernel called ReduceKernel, the analysis is imprecise due to widening.

The FastWalsh kernel contains a complex computation at the beginning such that we get a very imprecise abstract state at the beginning of the kernel and the analysis remains imprecise subsequently.

As mentioned before, our analysis is precise in most cases. To some extent, the precision traded against runtime which seems to be exponential in the program size. In the next section, we show how kernel size and analysis runtime behave in our evaluated kernels.

5.2 Kernel Size and Runtime

While the previous section was about evaluating precision and accuracy of our GPU kernel analysis, we now focus on two efficiency measures and how they relate. First, we look at the size of the transformed kernels that result from the 32-alignment and 2-product transformations. We give the size in lines of code (LOC). Second, we show the runtime of our analysis run on the transformed kernels, with and without the simplification **S** introduced at the beginning of the chapter.

Table 5.4 contains the sizes of the original and transformed kernels as well as the runtimes of the analysis for each kernel. The analysis configuration used for the creation of these results is the optimal configuration regarding precision and runtime. It includes the precision optimization **E** which leads to slightly longer runtime but to more precise results in some cases. Further optimizations were applied to reduce the number and complexity of abstract states, as described later in this section. Again, interesting results are highlighted in the table with a bold font.

	BD		SMBC		GMC		Total Accuracy	
Kernel	**E off**	**E on**	**E off**	**E on**	**E off**	**E on**	**E off**	**E on**
InlinePTX	1/1	1/1	0/0	0/0	1/1	1/1	100.0%	100.0%
Divergence	**1/2**	**2/2**	0/0	0/0	2/2	2/2	75.0%	100.0%
DxInterOp	0/0	0/0	0/0	0/0	8/8	8/8	100.0%	100.0%
UniformUpdate	1/1	1/1	1/1	1/1	3/3	3/3	100.0%	100.0%
ReduceKernel	1/1	1/1	0/0	0/0	**1/2**	**1/2**	66.7%	66.7%
VectorAdd	1/1	1/1	0/0	0/0	3/3	3/3	100.0%	100.0%
Reverse	1/1	1/1	2/2	2/2	2/2	2/2	100.0%	100.0%
ScanExclusive	2/2	2/2	0/0	0/0	3/3	3/3	100.0%	100.0%
Particles	5/5	5/5	3/3	3/3	8/8	8/8	100.0%	100.0%
FastWalsh	0/0	0/0	0/0	0/0	**0/8**	**0/8**	0.0%	0.0%
Reduction	**7/16**	**13/16**	20/20	20/20	3/3	3/3	76.9%	92.3%
ShflScan	**4/8**	**6/8**	4/4	4/4	3/3	3/3	73.3%	86.7%
MergeSort	Analysis runs out of memory before termination.							

Table 5.3: Accuracy as a fraction (TP+TN)/(P+N) with and without the optimization **E** for each hyperproperty: branch divergence (BD), shared memory bank conflicts (SMBC), and global memory coalescing (GMC). The total accuracy is computed per kernel over all three hyperproperties.

Interpretation

For many kernels, the simplification **S** shows little to no effect on the alignment which follows from the fact that the respective aligned kernels with and without applying **S** have the same number of lines. Still, there usually is a difference in the runtimes which does not come from the analysis itself but from a slightly different execution speed of the CPU. These differences could be reduced by averaging the runtimes over several executions.

The optimization **S** makes a significant difference in the Divergence, Reduction, ShflScan, and MergeSort kernels. The principal reason for that is that these kernels contain expressions of the form `t` / e or `t` % e, where `t` is the thread ID variable and e is some integer-valued expression. The alignment transformation of division and modulo expressions can often be simplified a lot if it is known in advance that the left-hand side (in our case, the variable `t`) is non-negative. Since thread IDs are always non-negative, we implemented this prior knowledge into the simplification **S** which makes that the above kernels can be simplified significantly. This idea could be spinned further in the following way: before the alignment transformation, it would be useful to perform abstract interpretation on the original kernel using a simple sign domain, in order to know the signs of the left-hand sides in division and modulo expressions, so as to simplify the corresponding transformed statements.

Table 5.4 also shows the main drawback of our method, which is the potentially huge blow-up of the kernel size through the alignment and product transformations. In our evaluation, the kernels 32-aligned with the simplification **S** are between 2.4x and 12.8x longer than the original kernels and 5.6x on average. The product program transformation again scales up the aligned kernels by a factor between 2.4x and 4.2x, and 3.5x on average. The total blow-up of the kernel sizes through both transformations lies between 5.8x in our best case InlinePTX and 49x in our worst case MergeSort. The size increases of both transformations are explicable: during the alignment transformation, compound expressions are assigned to fresh variables and these assignments are transformed as statements of several lines; for instance, the MergeSort kernel is not the longest kernel in size in our evaluation set, however, it contains a large number of complex compound expressions which leads to the huge size increase in the alignment transformation. The 2-product

Kernel	Kernel Sizes (LOC) S off	Kernel Sizes (LOC) S on	Runtime S off	Runtime S on
InlinePTX	8/19/46	8/19/46	0.041s	0.053s
Divergence	**26/114/478**	**26/77/272**	**0.91s**	**0.35s**
DxInterOp	20/88/365	20/88/365	0.96s	0.70s
UniformUpdate	19/89/373	19/89/373	0.95s	0.98s
ReduceKernel	19/75/217	19/75/217	2.2s	2.3s
VectorAdd	15/53/181	15/53/181	1.5s	1.5s
Reverse	18/82/258	18/82/258	2.6s	2.5s
ScanExclusive	19/161/481	19/161/481	8.5s	8.1s
Particles	44/168/665	44/168/665	31s	31s
FastWalsh	32/404/1306	32/404/1306	40s	40s
Reduction	**93/453/1722**	**93/417/1594**	**6min 20s**	**4min 30s**
ShflScan	**52/252/1020**	**52/227/873**	**11min 20s**	**10min 10s**
MergeSort	**60/802/3165**	**60/770/2933**	Out of memory.	

Table 5.4: Kernel sizes OS/AS/APS with and without the simplification **S** to the analysis' runtime. OS stands for the size of the original kernel, AS for the size of the 32-aligned kernel, and APS for the size of the 32-aligned 2-product kernel (which is the kernel that our analysis performs abstract interpretation on). The runtimes were measured on a Lenovo ThinkPad T480 with an 8-core Intel i7 processor and 16 GB of memory, running Linux Ubuntu 18.04.3. The runtimes were not averaged over multiple runs, instead we experimented with many different configurations that are not shown in the table above because they did not achieve better results and runtimes.

transformation also leads to an increase in kernel size because a single assignment is translated into two assignments, each conditioned on the respective activation variable, resulting in six lines of code.

The next observation concerns the relation between transformed kernel size and runtime. If the sizes of the transformed kernels are plotted in a linear scale against the corresponding analysis runtimes in a logarithmic scale, we get a graph almost looking like a straight line which indicates a runtime complexity which is nearly exponential in the kernel size. We give two possible explanations for this observation:

1. Generally, the analysis is exponential in the number of activation variables which usually grows with larger kernels. This exponential complexity comes from the Binary Decision Tree abstract domain used for trace partitioning, which performs abstract interpretation on all trace partitions separately, and the number of trace partitions is, in general, exponential in the number of activation variables. Knabenhans proposes different heuristics to reduce the number of trace partitions by merging partitions at some point during the abstract interpretation [9]. For our analysis, we chose the most rigorous merge heuristic, which merges on activation variables after their last occurrence in the program. This merge strategy can lead to precision loss, however, in our evaluation, we did not get more precise results when using less rigorous merge strategies or no merge strategies at all. When merging trace partitions in this way, there are usually no more than eight trace partitions at a time, meaning that the BDT domain alone cannot be held responsible for the large increases in runtime.
2. At least as important as trace partitioning is the number of variables that has to be tracked by the abstract domain. The larger the kernel, the more variables it usually contains and the more complex every abstract interpretation step becomes. Since we use relational domains, the runtime *and* memory complexity of the abstract interpretation is at least quadratic in the number of variables captured by the

abstract domain. In our analysis, we reduce the number of variables with the following optimization: after each partition merge on some activation variable `p`, we delete `p` from the set of variables contained in the current abstract state. With this optimization, the number of activation variables considered by the abstract domain is usually not larger than 8. In the configuration used in Table 5.4, it makes the analysis run up to two times faster, and in other configurations, it prevents execution abortion due to memory excess. More work can be done on the removal of variables from the abstract domain once they have no importance for the abstract interpretation of the remaining statements and expressions. A sophisticated approach would be to create a dependency graph for the kernel before abstract interpretation and to remove a variable after its last occurrence in the kernel once no variables in the remaining statements and expressions depend on it.

5.3 Apron Octagons and Polyhedra

The evaluation results listed in Tables 5.2, 5.3, and 5.4 come from our analysis using the Sample implementation of the Octagons abstract domain. As mentioned before, we have also evaluated the analysis using the Apron implementation of the Octagons and the Polyhedra abstract domains which turn out to be inappropriate for our purposes. Here, we give a short overview of why this is the case.

In contrast to the Sample Octagons implementation which is written in Scala, the Apron Octagons implementation is written in C++ and highly optimized, thus being faster than Sample Octagons. However, we have found that the use of the Apron Octagons domain can lead to unsound analysis results. Since we use Apron over Java bindings and, thus, have no direct access to the Apron Octagons implementation, it is difficult to make a reliable diagnosis why we use Apron Octagons in an unsound way. One possibility is that there are differences in the way Apron and Sample handle integer variables and certain operations like integer division, for instance. We have compiled Apron Octagons with different integer representations but without success. Assuming that the implementations of Apron Octagons and the Sample BDT domain are sound, we suppose that this unsoundness stems from an erroneous usage of the Apron interface. Since we require soundness of the static analysis, it is clear that as long as the analysis using Apron Octagons is unsound, we have to use the Sample Octagons implementation instead.

Using the Apron Polyhedra domain is not practicable either. For the kernels that we analyzed, the analysis runs out of memory very quickly, even for the smaller kernels. The source of the problem is described in the previous section, being the large number of variables contained in the kernel which result in large abstract interpretation runtime and memory usage. The computation of an abstract state in the Polyhedra domain is of some higher-polynomial complexity (both runtime and memory usage) in the number of variables and therefore worse than the Octagons domain whose memory usage is quadratic and runtime is cubic in the number of variables. Since our approach introduces a large number of variables, the Polyhedra domain is currently too inefficient for the analysis to terminate without exception. Even if the number of variables held in the Polyhedra domain can be minimized and acceptable runtimes and memory usage can be achieved, we suppose that the Polyhedra domain will rarely achieve more precise results than the Octagons domain. The reason for that is that after the m-alignment transformation, the specifications of the analyzed GPU kernel performance hyperproperties consist of equalities or disequalities, which are specifications that the Octagons domain is good proving at.

Chapter 6

Conclusion

6.1 Summary

The goal of this thesis was to develop a static analysis for three GPU kernel performance hyperproperties concerning branch divergence, shared memory bank conflicts, and global memory coalescing. We have implemented such a static analysis using a newly developed m-alignment transformation, an existing k-product transformation, and abstract interpretation with trace partitioning and the relational Octagons domain.

The m-alignment transformation is a transformation that, for a fixed $m > 1$, represents every variable `x` in the original kernel by two new variables $\mathtt{x_q}$, $\mathtt{x_r}$ such that $\mathtt{x} = m\mathtt{x_q} + \mathtt{x_r}$ and $0 \leq \mathtt{x_r} < m$. For our static analysis, we choose $m = 32$ because the relational hyperproperty specifications originally containing division by 32 and modulo 32 expressions can then be replaced by simpler equalities and disequalities on the new set of variables. This m-alignment transformation can also be used in other contexts where a precise analysis of division by m and modulo m expressions is needed.

We transform the 32-aligned kernel using a k-product transformation for $k = 2$ to reduce the kernel analysis of 2-safety hyperproperties to the analysis of properties of single execution traces. This 32-aligned product kernel is statically analyzed using abstract interpretation with the BDT domain allowing for trace partitioning, parametrized by the Sample Octagons domain. The Apron Octagons or Polyhedra domains are inappropriate for our purposes because their usage is unsound or too inefficient, respectively.

Several optimizations are applied to make the analysis faster and more precise. To gain precision, we extend the m-alignment transformation to insert assumptions that help the analysis with keeping track of equality constraints. In three out of 13 kernels, this optimization leads to a more precise analysis, in particular, nine false branch divergence alarms can be eliminated. With the m-alignment transformation, we include a simplification that removes newly introduced redundant expressions and statements and that, in our evaluation, reduces the transformed kernel size by a factor of up to 1.8x and runtime by a factor of up to 2.7x. Further optimizations address the high runtime and memory complexity of the abstract interpretation with trace partitioning and relational abstract domains: trace partitions split on some activation variable `p` are merged after the last occurrence of `p` in the kernel and `p` is removed from the set of variables tracked by the abstract domain. The merge strategy keeps the number of trace partitions at a more or less constant number of usually no more than eight partitions and is essential for the practicality of our analysis when applied to real kernels. Removing the activation variables from the abstract domain makes the analysis run up to twice as fast and, in some cases, prevents early termination due to memory excess. A last performance optimization was applied manually during the translation of the evaluated kernels from CUDA/OpenCL to Viper: we applied program slicing to keep only the statements in the kernel that are relevant for the hyperproperties to be analyzed. Program slicing can be automated and is important for the performance of our analysis since most of the actual computations done by the kernel are not relevant for the analysis itself.

We have evaluated the resulting static analysis on 13 kernels among which ten kernels are real kernels used in practice. First of all, the analysis is sound. On our evaluation set, it is precise in more than 80% of the cases (i.e., less than 20% of false alarms), and has an accuracy rate of over 85%. The drawback of our analysis is that is does not scale very well with large kernels. The evaluated runtimes reach from around $\frac{1}{20}$ seconds for the shortest kernel to several minutes for the larger kernels. For one kernel, we have no evaluation results because the analysis runs out of memory despite all optimizations. In the next section, we summarize some ideas on how our analysis can be made even faster and other ways in which this thesis could be extended.

6.2 Future Work

We first propose optimizations and extensions specifically for our static analysis of GPU kernel performance hyperproperties, then, possible improvements of our m-alignment transformation in general, and lastly, we recapitulate some ideas for further applications of m-aligned programs. Where possible, we refer to a more detailed explanation in an earlier section.

Our static analysis of GPU kernel performance hyperproperties can be improved in many ways:

- The program slicing described in Section 4.2.3 can be automated.
- We suppose that the major reason for the analysis' large runtimes and memory usage on some of our evaluated kernels is the great number of program variables tracked by the relational abstract domain. As explained in Section 5.2, we apply a simple heuristic for removing unused variables from the abstract domain but more work can be done to further reduce this number of tracked variables.
- In Section 5.3, we remark that the analysis using the Apron Octagons domain occasionally leads to unsound results. If the source of this unsoundness problem can be detected and eliminated, then the Apron Octagons domain can replace Sample Octagons which will reduce runtime and memory usage due to the more efficient implementation.
- It is interesting to note that much of our analysis is based on tracking equalities between quotient variables and disequalities between remainder variables of two executions. It is possible that this observation can be used to construct a simpler and more efficient static analysis that is almost as precise as ours and that only tracks whether specific statements (e.g., assignments) maintain initial equalities or disequalities between the quotient or remainder parts of variables without explicitly using the 32-alignment transformation. In support of our approach, it is also important to note that in some cases, the numerical analysis of quotient and remainder variables is necessary to get precise results (as an example, consider the branch condition `t` < 32 which has no branch divergence, with `t` being the thread ID). Nevertheless, the usefulness of such an analysis can be examined.

The following ideas can be pursued to enhance the m-alignment transformation:

- As described in Section 4.1.2, the alignment transformation can sometimes introduce redundant statements, especially for the transformation of division and modulo expressions. In the same section, we propose a simplification heuristic that uses syntactic rules to eliminate redundant statements or expressions. A more sophisticated approach would perform a simple static analysis on the initial program, e.g., abstract interpretation using a Sign, Interval, or Octagons domain, and use information from the abstract states at each program point to eliminate unused branches or to simplify expressions. For example, the m-alignment transformation of an assignment $\mathtt{y} := \mathtt{x}/m$ can be simplified a lot if abstract interpretation using a Sign domain yields $\mathtt{x} \geq 0$ at the program point before this assignment.

- In this thesis, the m-alignment transformations of division and modulo assignments are defined in a way that is as simple as possible but still sufficiently precise for analyzing GPU kernels. They can be further refined so that more edge cases with simple transformations can be covered (e.g., if the left-hand side is negative and the right-hand side is a multiple of m). If used in combination with the previous simplification approach, this can allow for more precise results while the transformation remains concise.

- In Section 3.6.3, we suggest that the idea of the m-alignment transformation can be used to create a new m-alignment abstract domain, which internally represents every variable $\mathtt{x}$ by two variables $\mathtt{x_q}$ and $\mathtt{x_r}$, and which implements the m-alignment transformation in its abstract transformers.

As mentioned in Section 3.6.4, the m-alignment transformation is promising to have further applications: the analysis of cache access performance (e.g., cache bank conflicts, temporal and spatial locality of cache accesses etc.), which is similar to the analysis carried out in this thesis, the simplification of mathematical proofs by case distinction, and, in general, every static analysis problem that requires reasoning about division by m or modulo m expressions for a fixed m.

Appendix A

m-Alignment Transformations

The following list is intended to give a quick and uncommented overview of all m-alignment transformations. For more details on the individual transformations, see Section 3.4.

Procedure

$$[\![\, \texttt{procedure}\ f(\overline{\texttt{x}})\ \texttt{returns}\ (\overline{\texttt{y}})\ \{s\}\,]\!]_m \ :\equiv\ \texttt{procedure}\ f(\overline{\texttt{x}}_{\texttt{qr}})\ \texttt{returns}\ (\overline{\texttt{y}}_{\texttt{qr}})\ \{[\![\, s\,]\!]_m\}$$

Skip

$$[\![\, \texttt{skip}\,]\!]_m \ :\equiv\ \texttt{skip}$$

Sequence

$$[\![\, s_1;\ s_2\,]\!]_m \ :\equiv\ [\![\, s_1\,]\!]_m;\ [\![\, s_2\,]\!]_m$$

Quotient and Remainder Variables

$$\mathsf{a_q} :\equiv \begin{cases} \texttt{y}_\texttt{q} & \text{if } \mathsf{a} \equiv \texttt{y} \\ k_q & \text{if } \mathsf{a} \equiv k, \text{where } k_q = \left\lfloor \frac{k}{m} \right\rfloor, \end{cases}$$

$$\mathsf{a_r} :\equiv \begin{cases} \texttt{y}_\texttt{r} & \text{if } \mathsf{a} \equiv \texttt{y} \\ k_r & \text{if } \mathsf{a} \equiv k, \text{where } k_r = k - m\left\lfloor \frac{k}{m} \right\rfloor \end{cases}$$

Atom Assignment

$$[\![\, \texttt{x} := \mathsf{a}\,]\!]_m \ :\equiv\ \texttt{x}_\texttt{q} := \mathsf{a_q};\ \texttt{x}_\texttt{r} := \mathsf{a_r}$$

Negation Assignment

```
$[\![ \mathsf{x} := -\mathsf{a} ]\!]_m :\equiv$ if $(\mathsf{a_r} = 0)$ then {
        $\mathtt{x_q} := -\mathsf{a_q}$;
        $\mathtt{x_r} := 0$
    } else {
        $\mathtt{x_q} := -\mathsf{a_q} - 1$;
        $\mathtt{x_r} := -\mathsf{a_r} + m$
    }
```

Addition Assignment

```
$[\![ \mathsf{x} := \mathsf{a} + \mathsf{b} ]\!]_m :\equiv$ if $(\mathsf{a_r} + \mathsf{b_r} < m)$ then {
        $\mathtt{x_q} := \mathsf{a_q} + \mathsf{b_q}$;
        $\mathtt{x_r} := \mathsf{a_r} + \mathsf{b_r}$
    } else {
        $\mathtt{x_q} := \mathsf{a_q} + \mathsf{b_q} + 1$;
        $\mathtt{x_r} := \mathsf{a_r} + \mathsf{b_r} - m$
    }
```

Subtraction Assignment

```
$[\![ \mathsf{x} := \mathsf{a} - \mathsf{b} ]\!]_m :\equiv$ if $(\mathsf{a_r} - \mathsf{b_r} \geq 0)$ then {
        $\mathtt{x_q} := \mathsf{a_q} - \mathsf{b_q}$;
        $\mathtt{x_r} := \mathsf{a_r} - \mathsf{b_r}$
    } else {
        $\mathtt{x_q} := \mathsf{a_q} - \mathsf{b_q} - 1$;
        $\mathtt{x_r} := \mathsf{a_r} - \mathsf{b_r} + m$
    }
```

Multiplication Assignment

```
$[\![ \mathsf{x} := \mathsf{a} * \mathsf{b} ]\!]_m :\equiv \mathsf{x_q} = m * \mathsf{a_q} * \mathsf{b_q} + \mathsf{a_q} * +\mathsf{a_r} * \mathsf{b_q} + (\mathsf{a_r} * \mathsf{b_r}) \;/\; m$;
    $\mathtt{x_r} = (\mathsf{a_r} * \mathsf{b_r}) \;\%\; m$
```

Division Assignment

```
$[\![\, \mathsf{x} := \mathsf{a} \ / \ \mathsf{b} \,]\!]_m :\equiv$ if $(\mathsf{a_q} \geq 0 \wedge \mathsf{b_q} \geq 0 \wedge \mathsf{b_r} = 0)$ then {
    $\mathtt{x_q} := (\mathsf{a_q} \ / \ \mathsf{b_q}) \ / \ m$;
    $\mathtt{x_r} := (\mathsf{a_q} \ / \ \mathsf{b_q}) \ \% \ m$
} else if $(\mathsf{a_q} \geq 0 \wedge \mathsf{b_q} = 0 \wedge m \ \% \ \mathsf{b_r} = 0)$ then {
    $\mathtt{x_q} := \mathsf{a_q} \ / \ \mathsf{b_r}$;
    $\mathtt{x_r} := \mathsf{a_r} \ / \ \mathsf{b_r} + m \ / \ \mathsf{b_r} * (\mathsf{a_q} \ \% \ \mathsf{b_r})$
} else {
    $\mathtt{x_{new}} := (m * \mathsf{a_q} + \mathsf{a_r}) \ / \ (m * \mathsf{b_q} + \mathsf{b_r})$;
    if $(\mathtt{x_{new}} \ \% \ m \geq 0)$ then {
        $\mathtt{x_q} := \mathtt{x_{new}} \ / \ m$
    } else {
        $\mathtt{x_q} := \mathtt{x_{new}} \ / \ m - 1$
    };
    $\mathtt{x_r} := \mathtt{x_{new}} - (m * \mathtt{x_q})$
}
```

Modulo Assignment

```
$[\![\, \mathsf{x} := \mathsf{a} \ \% \ \mathsf{b} \,]\!]_m :\equiv$ if $(\mathsf{a_q} \geq 0 \wedge \mathsf{b_q} \geq 0 \wedge \mathsf{b_r} = 0)$ then {
    $\mathtt{x_q} := \mathsf{a_q} \ \% \ \mathsf{b_q}$;
    $\mathtt{x_r} := \mathsf{a_r}$
} else if $(\mathsf{a_q} \geq 0 \wedge \mathsf{b_q} = 0 \wedge m \ \% \ \mathsf{b_r} = 0)$ then {
    $\mathtt{x_q} := 0$;
    $\mathtt{x_r} := \mathsf{a_r} \ \% \ \mathsf{b_r}$
} else {
    $\mathtt{x_{new}} := (m * \mathsf{a_q} + \mathsf{a_r}) \ \% \ (m * \mathsf{b_q} + \mathsf{b_r})$;
    if $(\mathtt{x_{new}} \ \% \ m \geq 0)$ then {
        $\mathtt{x_q} := \mathtt{x_{new}} \ / \ m$
    } else {
        $\mathtt{x_q} := \mathtt{x_{new}} \ / \ m - 1$
    };
    $\mathtt{x_r} := \mathtt{x_{new}} - (m * \mathtt{x_q})$
}
```

Extraction in Complex Assignment

$$[\![\, \mathsf{x} := e_1 \star e_2 \,]\!]_m \ :\equiv \ [\![\, \mathtt{y_1} := e_1 \,]\!]_m \, ; \ [\![\, \mathtt{y_2} := e_2 \,]\!]_m \, ; \ [\![\, \mathsf{x} := \mathtt{y_1} \star \mathtt{y_2} \,]\!]_m$$

Procedure Call

$$[\![\, \overline{\mathtt{y}} := \mathtt{call}\ f(\overline{e}) \,]\!]_m \ :\equiv \ [\![\, \mathtt{x_1} := e_1 \,]\!]_m \, ; \ \ldots \ [\![\, \mathtt{x_k} := e_k \,]\!]_m \, ; \ \overline{\mathtt{y}}_{\mathtt{qr}} := \mathtt{call}\ f(\overline{\mathtt{x}}_{\mathtt{qr}})$$

Comparison

$$[\![\, \mathsf{a} \circ \mathsf{b} \,]\!]_m \;:\equiv\; \begin{cases} \mathsf{a_q} = \mathsf{b_q} \wedge \mathsf{a_r} = \mathsf{b_r} & \text{if } \circ \equiv = \\ \mathsf{a_q} \neq \mathsf{b_q} \vee \mathsf{a_r} \neq \mathsf{b_r} & \text{if } \circ \equiv \neq \\ \mathsf{a_q} < \mathsf{b_q} \vee (\mathsf{a_q} = \mathsf{b_q} \wedge \mathsf{a_r} \circ \mathsf{b_r}) & \text{if } \circ \in \{<, \leq\} \\ \mathsf{a_q} > \mathsf{b_q} \vee (\mathsf{a_q} = \mathsf{b_q} \wedge \mathsf{a_r} \circ \mathsf{b_r}) & \text{if } \circ \in \{>, \geq\} \end{cases}$$

Extraction in Boolean Expression

$$(\!|\, c \,|\!)_m \;:\equiv\; \begin{cases} (\mathtt{y_1} := e_1;\ \mathtt{y_2} := e_2,\ [\![\, \mathtt{y_1} \circ \mathtt{y_2} \,]\!]_m) & \text{if } c \equiv e_1 \circ e_2 \text{ and } \circ \in \{=, \neq, <, >, \leq, \geq\}, \\ (\mathtt{skip},\ c) & \text{if } c \in \{\mathtt{true}, \mathtt{false}\}, \\ (\tilde{s}_0,\ \neg\tilde{c}_0) & \text{if } c \equiv \neg c_0, \\ (\tilde{s}_1;\ \tilde{s}_2,\ \tilde{c}_1 \wedge \tilde{c}_2) & \text{if } c \equiv c_1 \wedge c_2, \\ (\tilde{s}_1;\ \tilde{s}_2,\ \tilde{c}_1 \vee \tilde{c}_2) & \text{if } c \equiv c_1 \vee c_2, \end{cases}$$

where $(\tilde{s}_0, \tilde{c}_0) = (\!|\, c_0 \,|\!)_m$, $(\tilde{s}_1, \tilde{c}_1) = (\!|\, c_1 \,|\!)_m$, and $(\tilde{s}_2, \tilde{c}_2) = (\!|\, c_2 \,|\!)_m$

If Statement

$$[\![\, \mathtt{if}\ (c)\ \mathtt{then}\ \{s_1\}\ \mathtt{else}\ \{s_2\} \,]\!]_m \;:\equiv\; [\![\, \tilde{s} \,]\!]_m\,;\ \mathtt{if}\ (\tilde{c})\ \mathtt{then}\ \{\, [\![\, s_1 \,]\!]_m \,\}\ \mathtt{else}\ \{\, [\![\, s_2 \,]\!]_m \,\},$$

where $(\tilde{s}, \tilde{c}) = (\!|\, c \,|\!)_m$

While Loop

$$[\![\, \mathtt{while}\ (c)\ \mathtt{do}\ \{s\} \,]\!]_m \;:\equiv\; [\![\, \tilde{s} \,]\!]_m\,;\ \mathtt{while}\ (\tilde{c})\ \mathtt{do}\ \{[\![\, s \,]\!]_m\,;\ [\![\, \tilde{s} \,]\!]_m\},$$

where $(\tilde{s},\ \tilde{c}) \equiv (\!|\, c \,|\!)_m$

Appendix B

Evaluation Set

InlinePTX [1]

```
method global_write(i: Int)

method sequence_gpu(tid: Int, length: Int)
{
    if (tid < length) {
        global_write(tid)
    }
}
```

Divergence [6]

```
method global_write(index: Int)

method badDiv(tid: Int)
{
    var ia: Int
    var ib: Int
    if (tid % 2 == 0) {
        ia := 0
    } else {
        ib := 1
    }
    global_write(tid)
}


method goodDiv(tid: Int)
{
    var ia: Int
    var ib: Int
    if ((tid / 32) % 2 == 0) {
        ia := 0
    } else {
        ib := 1
    }
    global_write(tid)
}
```

DxInterOp [1]

```
method global_write(i: Int)
method global_read(i: Int) returns (res: Int)
method shared_write(i: Int)
method shared_read(i: Int) returns (res: Int)

method run(tid: Int)
{
    var tmp: Int

    tmp := global_read(tid)
    tmp := global_read(tid)
    global_write(tid)

    tmp := global_read(tid)
    tmp := global_read(tid)
    global_write(tid)

    tmp := global_read(tid+1)
    global_write(tid+1)
}
```

UniformUpdate [1]

```
method global_write(i: Int)
method global_read(i: Int) returns (res: Int)
method shared_write(i: Int)
method shared_read(i: Int) returns (res: Int)

method uniformUpdate(tid: Int)
{
    var blockDim: Int := 256
    var tmp: Int

    if (tid % blockDim == 0)
    {
        tmp := global_read(tid / blockDim)
        shared_write(0)
    }

    tmp := global_read(tid)
    global_write(tid)
}
```

ReduceKernel [1]

```
method global_write(i: Int)
method global_read(i: Int) returns (res: Int)
method shared_write(i: Int)
method shared_read(i: Int) returns (res: Int)

method reduceKernel(tid: Int, N: Int)
{
    var blockDim: Int := 256
    var gridDim: Int := 32
    var threadN: Int := gridDim * blockDim
    var tmp: Int

    var pos: Int := tid
    while (pos < N) {
        tmp := global_read(pos)
        pos := pos + threadN
    }
    global_write(tid)
}
```

VectorAdd [11]

```
method global_read(i: Int) returns (res: Int)
method global_write(i: Int)

method vectorAdd(iNumElements: Int, tid: Int)
{
    if (tid < iNumElements) {
        var x: Int
        var y: Int
        var z: Int
        x := global_read(tid)
        y := global_read(tid)
        z := x + y
        global_write(tid)
    }
}
```

Reverse [8]

```
method global_read(i: Int) returns (res: Int)
method global_write(i: Int)
method shared_read(i: Int) returns (res: Int)
method shared_write(i: Int)

method reverse(tid: Int, n: Int)
{
    var tr: Int
    var tmp: Int

    if (tid < n) {
        tr := n - tid - 1
        tmp := global_read(tid)
        shared_write(tid)
        tmp := shared_read(tr)
        global_write(tid)
    }
}
```

ScanExclusive [1]

```
method global_write(i: Int)
method global_read(i: Int) returns (res: Int)
method shared_write(i: Int)
method shared_read(i: Int) returns (res: Int)

method scanExclusiveShared2(tid: Int, N: Int)
{
    var blockDim: Int := 256
    var tmp: Int

    if (tid < N) {
        tmp := global_read(4 * blockDim - 1 + 4 * blockDim * tid)
        tmp := global_read(4 * blockDim - 1 + 4 * blockDim * tid)
    }

    if (tid < N) {
        global_write(tid)
    }
}
```

Particles [1]

```
method global_write(i: Int)
method global_read(i: Int) returns (res: Int)
method shared_write(i: Int)
method shared_read(i: Int) returns (res: Int)

method reorderDataAndFindCellStartD(tid: Int, numParticles: Int)
{
    var blockDim: Int := 256
    var threadIdx: Int := tid % blockDim
    var hash: Int
    var tmp: Int

    if (tid < numParticles)
    {
        hash := global_read(tid)
        shared_write(threadIdx + 1)

        if (tid > 0 && threadIdx == 0)
        {
            tmp := global_read(tid - 1)
            shared_write(0)
        }

        tmp := shared_read(threadIdx)
        if (tid == 0 || hash != tmp)
        {
            global_write(hash)

            if (tid > 0) {
                tmp := shared_read(threadIdx)
                global_write(tmp)
            }
        }

        if (tid == numParticles - 1)
        {
            global_write(hash)
        }

        tmp := global_read(tid)
        global_write(tid)
        global_write(tid)
    }
}
```

FastWalsh [1]

```
method global_write(i: Int)
method global_read(i: Int) returns (res: Int)
method shared_write(i: Int)
method shared_read(i: Int) returns (res: Int)

method fwtBatch2Kernel(tid: Int, blockY: Int, addr_in: Int, addr_out: Int)
{
    var blockDim: Int := 256
    var gridDim: Int := 8192
    var N: Int := blockDim * gridDim * 4
    var stride: Int := 2048
    var tmp: Int

    var offsetSrc: Int := addr_in + blockY * N
    var offsetDst: Int := addr_out + blockY * N

    var lo: Int := tid % stride
    var i0: Int := 4 * (tid - lo) + lo
    var i1: Int := i0 + stride
    var i2: Int := i1 + stride
    var i3: Int := i2 + stride

    tmp := global_read(offsetSrc + i0)
    tmp := global_read(offsetSrc + i1)
    tmp := global_read(offsetSrc + i2)
    tmp := global_read(offsetSrc + i3)

    global_write(offsetDst + i0)
    global_write(offsetDst + i1)
    global_write(offsetDst + i2)
    global_write(offsetDst + i3)
}
```

Reduction [1]

```
method global_write(i: Int)
method global_read(i: Int) returns (res: Int)
method shared_write(i: Int)
method shared_read(i: Int) returns (res: Int)

method reduce5(tid: Int, n: Int, blockSize: Int)
{
    var blockDim: Int := 256
    var threadIdx: Int := tid % blockDim
    var blockIdx: Int := tid / blockDim
    var i: Int := blockIdx * (blockSize*2) + threadIdx;
    var tmp: Int

    if (i < n) {
        tmp := global_read(i)
    } else {
        tmp := 0
    }

    if (i + blockSize < n) {
        tmp := global_read(i+blockSize)
    }

    shared_write(threadIdx)

    if (blockSize >= 512)
    {
        if (threadIdx < 256)
        {
            tmp := shared_read(threadIdx + 256)
            shared_write(threadIdx)
        }
    }
    if (blockSize >= 256)
    {
        if (threadIdx < 128)
        {
            tmp := shared_read(threadIdx + 128)
            shared_write(threadIdx)
        }
    }
    if (blockSize >= 128)
    {
        if (threadIdx < 64)
        {
            tmp := shared_read(threadIdx + 64)
            shared_write(threadIdx)
        }
    }
    if (threadIdx < 32)
    {
        if (blockSize >= 64)
        {
            tmp := shared_read(threadIdx + 32)
            shared_write(threadIdx)
        }

        if (blockSize >= 32)
        {
            tmp := shared_read(threadIdx + 16)
            shared_write(threadIdx)
        }

        if (blockSize >= 16)
        {
            tmp := shared_read(threadIdx + 8)
            shared_write(threadIdx)
        }

        if (blockSize >= 8)
        {
            tmp := shared_read(threadIdx + 4)
            shared_write(threadIdx)
        }

        if (blockSize >= 4)
        {
            tmp := shared_read(threadIdx + 2)
            shared_write(threadIdx)
        }

        if (blockSize >= 2)
        {
            tmp := shared_read(threadIdx + 1)
            shared_write(threadIdx)
        }
    }

    if (threadIdx == 0) {
        tmp := shared_read(0)
        global_write(blockIdx)
    }
}
```

ShflScan [1]

```
method global_write(i: Int)
method global_read(i: Int) returns (res: Int)
method shared_write(i: Int)
method shared_read(i: Int) returns (res: Int)

method shfl_scan_test(tid: Int, width: Int)
{
    var blockDim: Int := 256
    var threadIdx: Int := tid % blockDim
    var lane_id: Int := tid % 32
    var warp_id: Int := threadIdx / 32
    var tmp: Int

    tmp := global_read(tid)

    var i: Int := 1
    while (i <= width) {
        if (lane_id >= i) {
            tmp := tmp
        }
        i := i * 2
    }

    if (threadIdx % 32 == 31)
    {
        shared_write(warp_id)
    }

    if (warp_id == 0)
    {
        tmp := shared_read(lane_id)
        i := 1
        while (i <= width) {
            if (lane_id >= i) {
                tmp := tmp
            }
            i := i * 2
        }
        shared_write(lane_id)
    }

    if (warp_id > 0)
    {
        tmp := shared_read(warp_id-1)
    }

    global_write(tid)

    if (threadIdx == blockDim-1) {
        global_write(tid/blockDim)
    }
}
```

MergeSort [1]

```
method global_write(i: Int)
method global_read(i: Int) returns (res: Int)
method shared_write(i: Int)
method shared_read(i: Int) returns (res: Int)

method binarySearchExclusive() returns (res: Int)
method binarySearchInclusive() returns (res: Int)

method mergeSortSharedKernel(tid: Int, arrayLength: Int)
{
    var blockDim: Int := 512
    var blockIdx: Int := tid / blockDim
    var threadIdx: Int := tid % blockDim
    var sharedSizeLimit: Int := 1024
    var offset: Int := blockIdx * sharedSizeLimit + threadIdx
    var tmp: Int

    tmp := global_read(offset)
    shared_write(threadIdx)
    tmp := global_read(offset)
    shared_write(threadIdx)
    tmp := global_read(offset + sharedSizeLimit / 2)
    shared_write(threadIdx + sharedSizeLimit / 2)
    tmp := global_read(offset + sharedSizeLimit / 2)
    shared_write(threadIdx + sharedSizeLimit / 2)

    var stride: Int := 1
    while (stride < arrayLength) {
        var lPos: Int
        lPos := threadIdx % stride
        var offset2: Int := 2 * (threadIdx - lPos)

        tmp := shared_read(offset2 + lPos)
        tmp := shared_read(offset2 + lPos)
        tmp := shared_read(offset2 + lPos + stride)
        tmp := shared_read(offset2 + lPos + stride)
        var posA: Int
        var posB: Int
        tmp := binarySearchExclusive()
        posA := tmp + lPos
        tmp := binarySearchInclusive()
        posB := tmp + lPos

        shared_write(offset2 + posA)
        shared_write(offset2 + posA)
        shared_write(offset2 + posB)
        shared_write(offset2 + posB)

        stride := stride * 2
    }

    tmp := shared_read(threadIdx)
    global_write(offset)
    tmp := shared_read(threadIdx)
    global_write(offset)
    tmp := shared_read(threadIdx + sharedSizeLimit / 2)
    global_write(offset + sharedSizeLimit / 2)
    tmp := shared_read(threadIdx + sharedSizeLimit / 2)
    global_write(offset + sharedSizeLimit / 2)
}
```

Appendix C

Bibliography

[1] A. Betts, N. Chong, P. Deligiannis, A. F. Donaldson, and J. Ketema. Implementing and Evaluating Candidate-Based Invariant Generation. `http://multicore.doc.ic.ac.uk/tools/GPUVerify/IEEE_TSE/`, January 2015. Accessed 2019-07-25.

[2] A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson. GPUVerify: A Verifier for GPU Kernels. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA)*, pages 113–132, Tucson, Arizona, USA, October 2012.

[3] D. M. Burton. *Elementary Number Theory.* Allyn and Bacon, Inc., Boston, Massachusetts, 1980.

[4] P. Cousot. Abstract Interpretation in a Nutshell. `https://www.di.ens.fr/~cousot/AI/IntroAbsInt.html`, 2005. Accessed 2019-09-07.

[5] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL)*, pages 238–252, Los Angeles, California, January 1977.

[6] Dante003 and Robert_Crovella. Branch Divergence. `https://devtalk.nvidia.com/default/topic/883094/branch-divergence/`, July 2015. Accessed 2019-06-20.

[7] M. Eilers, P. Müller, and S. Hitz. Modular Product Programs. In A. Ahmed, editor, *Programming Languages and Systems*, volume 10801 of *Lecture Notes in Computer Science*, pages 502–529, Cham, 2018. European Symposion on Programming (ESOP), Springer.

[8] M. Harris. Using Shared Memory in CUDA C/C++. `https://devblogs.nvidia.com/using-shared-memory-cuda-cc/`, January 2013. Accessed 2019-03-15.

[9] C. Knabenhans. Automatic Inference of Hyperproperties. Bachelor's Thesis, ETH Zurich, August 2018.

[10] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A Verification Infrastructure for Permission-Based Reasoning. In B. Jobstmann and K. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *Lecture Notes in Computer Science*, pages 41–62, Berlin, Heidelberg, 2016. Springer.

[11] NVIDIA Corporation. CUDA C Programming Guide. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`, February 2019. Accessed 2019-03-12.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Static Analysis of GPU Kernel Performance Hyperproperties

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Blarer	Mathias

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Altendorf, 14.09.2019	[illegible]

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.