Imperial College London
Department of Computing

Scalable Verification Techniques for Data-Parallel Programs

Nathan Yong Seng Chong

September 2014

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London
and the Diploma of Imperial College London

Declaration

This thesis and the work it presents are my own except where otherwise acknowledged.

Nathan Yong Seng Chong

Abstract

This thesis is about *scalable* formal verification techniques for software. A verification technique is scalable if it is able to scale to reasoning about real (rather than synthetic or toy) programs. Scalable verification techniques are essential for practical program verifiers. In this work, we consider three key characteristics of scalability: precision, performance and automation. We explore trade-offs between these factors by developing verification techniques in the context of *data-parallel* programs, as exemplified by *graphics processing unit* (GPU) programs (called *kernels*). This thesis makes three original contributions to the field of program verification:

- An empirical study of *candidate-based invariant generation* that explores the trade-offs between precision and performance. An *invariant* is a property that captures program behaviours by expressing a fact that always holds at a particular program point. The generation of invariants is critical for automatic and precise verification. Over a benchmark suite comprising 356 GPU kernels, we find that candidate-based invariant generation allows precise reasoning for 256 (72%) kernels.

- *Barrier invariants*: a new abstraction for precise and scalable reasoning about *data-dependent* GPU kernels, an important class of kernel beyond the scope of existing techniques. Our evaluation shows that barrier invariants enable us to capture a functional specification for three distinct prefix sum implementations for problem sizes using hundreds of threads and race-freedom for a real-world stream compaction example.

- The *interval of summations*: a new abstraction for precise and scalable reasoning for parallel prefix sums, an important data-parallel primitive. We give theoretical results showing that the interval of summations is, surprisingly, both sound and complete. That is, all correct prefix sums can be precisely captured by this abstraction. Our evaluation shows that the interval of summations allow us to automatically prove full functional correctness of four distinct prefix sum implementations for all power-of-two problem sizes up to 2^{20}.

Acknowledgements

Alastair F. Donaldson, my supervisor, has been instrumental to my work and I thank him unreservedly. Ally's insights and encouragement were the driving force behind all of my best research. I remember weeks of bringing obtuse little diagrams to Ally's office where we would drink coffee and try our best to decipher just a few lines of code; it was a wonderful time! Just as importantly, I have always felt that Ally was 'in my corner' and he has spent many hours reading this thesis, helping me to improve it. Thank you Ally, I could not imagine completing this work without your support.

Paul H. J. Kelly, my second supervisor, has been a guide and friend to me since my MSc and he has advised me soundly and sagely. It was Paul's off-the-cuff observation about the sequence $0, 1, 1, 2, 1, 2, 2, 3, \ldots$ — "that's the *Hamming weight*" — scribbled in my notebook that provided the key to unravelling the Blelloch prefix sum given in Chapter 4. I thank Jeroen Ketema, my co-author, colleague and friend, who showed me how rigor and insight go hand-in-hand, and with whom I proudly share (with Ally) the discovery of the interval of summations. I thank Samin Ishtiaq, my long-time mentor, who taught me how to be a researcher and whose time, expertise and friendship I appreciate very much.

I thank Adam Betts and Paul Thomson, the other 'founding' members of the Multicore Programming Group, for being good friends; and, of course, the later members too: Ethel Bardsley, Pantazis Deligiannis, Daniel Liew and John Wickerson. I also thank Emre Özer, Alastair Reid and Per Strid, my former colleagues at ARM, for their time and advice; Emma Armitage for her tireless efforts to teach me good design; Derek Graham, Kim Jarvis, Nick Johnson, Tom Riley and Rob Yates for all the fun times; Julie Angel, Ido Portal, Christopher Sommer, Vic Verdier and Stephane Vigroux for inspiring me; and, not least, Julian Donovan and my friends in the G— Gym who I am proud to move with.

I gratefully acknowledge financial support from the EU FP7 STREP project CARP (project number 287767) and the EPSRC PSL project (EP/I006761/1).

I thank my parents who gave me everything including my sister and brother, Rachael and Daniel, and all my family around the world who I love (and also thank!). This is for you. I also thank Sukong for his steadfast belief in me. Finally, I thank Phương-Dzung for her constant love and support and for being my Springtime.

Contents

List of Tables

List of Figures

1. Introduction

This thesis is about techniques for building better software. In particular, *verification* techniques, which can give guarantees of program correctness beyond the guarantees that can be provided by testing. To enable practical verification tools we require *scalable* verification techniques that are capable of reasoning about real programs. This thesis examines the problem of scalable verification techniques in the context of *data-parallel* programs.

1.1. Motivation

Software is the intangible fabric that enables modern life. Or, as Marc Andreessen wrote, "software is eating the world" [And11]. The central argument of his now famous essay is that software is disrupting, revolutionising and sometimes replacing (or, eating) entire industries: communications, retail, entertainment, finance, transportation, infrastructure and much more; software governs it all.

Yet all non-trivial programs inevitably contains bugs or unintended defects since programming is part science and art [Knu74], involving both logic and intuition. As programmers will attest: it is surprisingly difficult to tell a computer what you mean. The density of bugs in industry delivered code has been estimated to be between one and twenty-five errors per thousand lines of code [McC04, p. 521]. Rephrasing a syllogism of DeMillo, Lipton and Perlis [DLP79]:

> All human artifacts of sufficient size and complexity are imperfect.
>
> All real-life programs are sizeable and complex human artifacts.
>
> Hence all real-life programs are imperfect.

The combination of these two observations (that software is both pervasive and inevitably prone to errors) means that the consequences of bugs can be catastrophic. Prominent cases include the Therac-25 radiation therapy machine [Lev93], the European Space Agency's Ariane-5 rocket [Dow97], and the high-frequency trading firm, Knight Capital Group [U.S13], where software bugs can be held accountable, respectively, for fatalities,

the destruction of a 370 million dollar rocket (in 37 seconds), and near-bankruptcy caused by the loss of 440 million dollars (in 46 minutes). More prosaically, a 2002 report by U.S. Department of Commerce estimates the cost of faulty software to "range in the tens of billions of dollars per year" [Res02].

Good software engineering practice uses many techniques, tools and processes to defend against bugs in software. These include coding standards, code reviews, defensive programming and testing (McConnell [McC04] covers each of these techniques). Testing has many different guises (such as unit, integration and regression testing) but fundamentally takes the form of running test cases over the program and checking the output or behaviour against an expected outcome. Testing is very widely used and very effective [McC04]. However, it is an unending task because testing can never give guarantees of correctness since exploring the space of test cases, even for modest programs, is infeasible.

In this thesis we focus on *program verification*, which allows us to prove programs correct with respect to some specification. It is important to emphasise that program verification is not "a silver bullet" [Bro87]: verified programs are not perfect. Nevertheless, program verification can give guarantees of correctness that cannot be given by testing.

Program verification is not new technology: it is as old as the discipline of software engineering itself [Mac01, pp. 34–55]. However, verification is rarely used in industry except in safety-critical applications [BH05]. The vision of program verification as given by Hoare [Hoa03] — "a software industry that embraces verification and verification technology throughout the software life-cycle"— is not a reality for everyday programmers. How can we address this?

1.2. Contributions

Practical program verifiers require underlying scalable verification techniques. In this work, we regard a verification technique to be scalable if it is able to scale to reasoning about real (rather than synthetic or toy) programs. We regard a program to be real if it is found 'in the wild' and therefore reflects the idioms and language features employed by real programmers. In contrast, a synthetic program is found 'in captivity'. It was designed for an explicit purpose such as performance or verification (usually to show that a given technique is better than another) and consequently may or may not reflect the features found in real programs. This does not mean that synthetic programs are useless (we will use them many times in this thesis to illustrate our techniques), but rather that synthetic programs are not necessarily accurate reflections of typical programs. Real programs using realistic problem sizes are the ultimate test for program verifiers.

In this thesis we regard the performance of the verifier when analysing real programs against increasingly large problem sizes to be the critical measure of scalability. Other key characteristics that we consider are precision (how accurate the verifier is at discerning bugs) and automation (the degree of programmer assistance required to use the verifier). There are important trade-offs to be made between each of these characteristics.

We will investigate scalable verification techniques in the context of data-parallel programs. This class of parallel program has seen renewed interest with the rise of general-purpose *graphic processing unit* (GPU) computing. This has seen GPUs become commonplace as accelerators for many diverse applications across many different platforms from high-performance supercomputers to embedded devices.

After reviewing the background of our work in Chapter 2, we turn to the three main contributions of this thesis:

- Chapter 3 explores the trade-offs between automation and performance. We conduct an empirical study of invariant generation, which is critical for automatic verification, and performance, the responsiveness of the verifier.
- Chapter 4 develops *barrier invariants*, a new abstraction and verification technique that enables precise and scalable reasoning for a class of program previously beyond the scope of existing techniques.
- Chapter 5 develops an automatic, precise and highly-scalable verification technique using a new abstraction, the *interval of summations*, for reasoning about prefix sums, an important primitive for data-parallel programs.

1.3. Publications

Some of the original material in this thesis has been previously published by the author in two co-authored papers. The barrier invariant technique presented in Chapter 4 appears in [CDK+13]. The interval of summations technique presented in Chapter 5 appears in [CDK14]. We refer to two further co-authored papers [BCD+12, BBC+14] as prior work because they represent the context in which the original work in this thesis was conducted.

1.4. Formal Acknowledgements

My declaration of originality states that "this thesis and the work it presents are my own except where otherwise acknowledged." I list the most important exceptions here.

I gratefully acknowledge and thank my main co-authors, Alastair F. Donaldson and Jeroen Ketema, for their significant contributions to [CDK+13] and [CDK14] (which form the basis of Chapters 4 and 5 respectively). In particular, they were both instrumental in the formalisation of barrier invariants including its soundness result; recognising the utility of the interval of summations as a single dynamic test case; and, the formalisation of the interval of summations including its theoretical results. I also gratefully acknowledge and thank the following individuals.

- Adam Betts for (i) implementing the dynamic analysis refutation engine discussed in Chapter 3 and (ii) helping to conduct the performance experiments of Chapter 3.
- Pantazis Deligiannis for implementing the bounded loop unrolling refutation engine, splitting loop checking refutation engines and parallel refutation sharing discussed in Chapter 3.
- Alastair F. Donaldson for conducting the data race-freedom experiments of Chapter 5.
- Jeroen Ketema for helping to conduct the evolution and precision experiments of Chapter 3.
- Ethel Bardsley and John Wickerson for implementing KernelInterceptor [BDW14] used in the methodology of Chapter 3.

Finally, I acknowledge and thank the contributors to GPUVerify who I have not listed above: Peter Collingbourne, Egor Kyshtymov, Daniel Liew, Paul Thomson and Shaz Qadeer.

2. Race Checking for GPU Kernels

In this chapter we review techniques for checking or verifying that data-parallel programs are race-free. Our main aim is to give a detailed introduction to the verification method of the GPUVerify tool [BCD+12], discussed in Sections 2.2.4 and 2.3, since it is the basis of this thesis.

2.1. A Brief Review

To begin we briefly review program verification, the challenge of concurrency and data-parallel programs.

2.1.1. Program Verification

The aim of program verification is to prove that programs are correct with respect to some logical specification. Just as a mathematical conjecture cannot be proven by testing (modulo exhaustive testing) neither can a program. We need logic for rigorous reasoning. The foundations of program verification were laid by Floyd, Hoare and Dijkstra. *Floyd-Hoare logic* is a logical framework for reasoning about the correctness of programs [Flo67, Hoa69] in which programs are specified by *Hoare triples*, $\{\phi\}\ P\ \{\psi\}$. We interpret this judgement to mean that all executions of the program P that start in a state satisfying the precondition ϕ either (i) terminate in a state satisfying the postcondition ψ or (ii) do not terminate. Because we do not constrain the behaviour of non-terminating executions we call this *partial correctness* (in contrast to *total correctness* which requires termination). In this thesis we use correctness in the sense of partial correctness and therefore avoid having to prove termination. A program proof is a program annotated with Hoare triple annotations and can be checked for correctness. We distinguish between *safety* properties (something bad never happens) and *liveness* (something good always eventually happens).

Floyd-Hoare logic gives a method for *checking* a program proof but it does not in itself give a method for generating such a proof. Dijkstra showed how to mechanise the generation of such a proof using *predicate transformers* [Dij76], which can translate a program into a *verification condition*: a logical formula whose *validity* implies the correctness

of the program. Subsequently, a verification condition can be discharged to a theorem prover. Using an automated theorem prover such as a *Satisfiability Modulo Theories* (SMT) solver [dMB11] means we can automate this step and generate *counterexamples* if we cannot prove the program correct (i.e., if we cannot prove the validity of the verification condition).[1] SMT solvers are the workhorses of automated reasoning. They are constraint solvers for logic combining *decision procedures* for theories [KS08] such as arithmetic, bit-vectors and uninterpreted functions. Combining these theories gives a rich language for describing many problems including program verification. We refer the reader to Bradley and Manna for a technical overview of these topics [BM07] and to Mackenzie for a historical and sociological account of the development of verification [Mac01].

A *program verifier* is a tool using automated Floyd-Hoare logic, verification condition generation and theorem proving. Examples of program verifiers are the Extended Static Checker for Java (ESC/Java) [FLL$^+$02], Boogie [BCD$^+$05] and Why3 [FP13]. Boogie and Why3 are also examples of intermediate verification languages designed to be the target of program verifiers for other languages. For example, the Spec# [BLS05] and Dafny [Lei10] program verifiers are both based on top of Boogie.

Key characteristics for practical program verifiers are precision, performance and automation.

Precision For precision, verifiers are characterised as being *unsound* or *incomplete*. Verifiers that are unsound may report *false negatives*: claiming that a program is correct when it actually contains errors. Verifiers that are incomplete may report *false positives*: claiming that a program is incorrect when it is actually error-free (we also refer to this as a *spurious error*). Neither property is desirable: unsoundness means that we may miss bugs in critical programs whereas incompleteness impacts usability since the user of the verifier must sort through bug reports to distinguish true errors from false errors. It is impossible to design a perfect verifier that is both sound and complete for all programs due to Rice's theorem [Ric53], which states that all non-trivial properties of programs are undecidable. In general, most verifiers aim for soundness and tolerate incompleteness (which is conservative). Related to precision are *assumptions* made by or hardcoded into the verifier. For example, a verifier may assume that integers are mathematical integers rather than bit-vectors (machine integers).[2] Assumptions can be seen as simplifications and must be

[1] We can check a verification condition for validity by negating the formula and checking for *satisfiability* (an assignment of values to variables such that the formula is true). Then a satisfying assignment is a counterexample to the correctness of the program.

[2] The assertion $x < y \Rightarrow x < y + 1$ is always true for mathematical integers (which are unbounded) but does not hold for bit-vectors (which have wrap around behaviour).

carefully checked to ensure that they do not invalidate the result of verification. In particular, contradictory assumptions can lead to *vacuous* verification (since $\mathsf{false} \Rightarrow P$ is true for all P).

Performance Concerns for the performance of program verifiers are speed and predictability. That is, the time to compute a response and the run-to-run variation. Performance is easier to characterise than precision since we always desire faster and more predictable techniques. Because the techniques that we consider are based on SMT solvers we are guided by their limitations. For example, a well-known brittleness is the handling of quantifiers [DNS05][3]; hence, in this thesis we seek *quantifier-free* methods.

Automation Automation refers to the level of programmer assistance required to use the program verifier, which ranges from fully manual to fully automatic. When discussing automation it is important to state what is being verified since program verification covers a spectrum from lightweight properties (such as the absence of data races) to full functional specification. There is a general trade-off between the degree of automation and the strength of properties being proven.

In this thesis, we say that a program verifier is *scalable* if it is capable of scaling to practical real-world programs. This is a combination of precision, performance and automation. We require precision since the tool must be able to reason precisely about the programs of interest. We require performance because the users of the tool should expect a response within a reasonable amount of time and with a high-degree of predictability. For automation we favour techniques requiring minimal programmer assistance rather than manual methods. We will focus our efforts on sound verification of lightweight safety properties.

2.1.2. The Challenge of Concurrency

A concurrent program structures its computation over multiple communicating processes. The two dominant forms of inter-process communication are message-passing or shared-memory. In this thesis we focus on shared-memory concurrency where processes implicitly message through shared memory. New challenges posed by this setting are schedule explosion, orchestrating synchronisation and dealing with interference between processes. Errors that can occur include data races, deadlocks and atomicity violations [LPSZ08].

[3] Detlefs et al. [DNS05] write "quantifiers are near the heart of all the essential difficulties [of designing an SMT solver]."

Techniques for verifying concurrent programs include systematic concurrency testing which aims to systematically explore all schedules [God97, MQB+08, TDB14], model checking a finite-state system abstracted from the program [DKKW11, DKK+12], concurrent program verifiers [CDH+09, LMS09] based on concurrent program logics [Vaf08, chap. 2] and sequential transformation which turns the concurrent program into a semantically equivalent sequential program and enables sequential techniques to be applied. In this thesis we focus on the latter technique.

2.1.3. Data-Parallel Programs

Parallelism enables multiple concurrent processes to be executed at the same time to improve performance. We distinguish between parallelism that can be exploited at the bit, instruction, data and task level. In this thesis we focus on data parallelism which is characterised by "simultaneous operations across large sets of data, rather than from multiple threads of control" [HS86]. Examples of data-parallel processors include vector, SIMD (single instruction multiple data) and GPU (graphics processing unit) machines [HP11, chap. 4]. In this thesis we focus on the verification of data-parallel GPU programs.

GPUs Modern GPUs are parallel many-core processors designed for throughput processing [GK10]. Originally designed as accelerators for graphics, where the manipulation of pixels is inherently parallel, the trend towards greater programmability and flexibility — known as general purpose GPU (GPGPU) computing — means that GPUs are now attractive targets for applications beyond graphics.

GPUs are commonly used as parallel coprocessors under the control of a host CPU in a *heterogeneous* system. The host CPU is responsible for copying input data across to the GPU, launching parallel programs on the GPU and fetching resulting data back from the GPU. A typical GPU consists of multiple multiprocessors each made up of many simple cores. Unlike CPU cores, these are simple in-order processors that eschew aggressive instruction-level parallelism techniques such as branch prediction or speculation. Each multiprocessor executes instructions in a SIMD fashion where multiple cores execute the same instruction on different data. Distinct multiprocessors can execute independent SIMD instructions at the same time. The memory hierarchy consists of a private memory per core, a local memory per multiprocessor accessible by all cores on the same multiprocessor and a main memory accessible by all cores (across the whole GPU). It is usual for the shared memory to be organised as a *software-managed cache* for scratchpad data. For example, the NVIDIA Fermi architecture [NVI09] uses 16 multiprocessors each consist-

ing of 32 cores. Each multiprocessor executes 32-wide SIMD instructions (called *warps* in NVIDIA terminology). The local memory per multiprocessor is an on-chip *shared* memory and is organised as a split level 1 data cache and software-managed cache. Fermi has a unified level 2 cache and main memory is an off-chip *global* memory.

GPUs are programmed using a single program multiple data (SPMD) execution model where the same program is executed in parallel over multiple cores. In this thesis we consider the two main GPGPU programming models: CUDA and OpenCL.

CUDA CUDA [NVI12a] is an extension of the C language that allows the programmer to write parallel programs (*kernels*) as well as an application programming interface (API) for managing and orchestrating the execution of kernels on the GPU. A kernel is a template that specifies the behaviour of an arbitrary thread. Kernels are executed by many threads in parallel, organised hierarchically as a *grid* of *thread-blocks*. A thread-block (or, more simply, a *block*) is a one-, two- or three-dimensional group of threads. Each thread within a block is identified by a built-in 3-component variable `threadIdx` (indexed by `x`, `y` and `z`). Blocks are themselves organised into a one-, two- or three-dimensional grid. Each block within the grid is identified by a built-in 3-component variable `blockIdx`. This means that the total number of threads that execute a kernel is the number of blocks (the *grid dimension*) times the number of threads per block (the *block dimension*). The grid and block dimension can be queried in a kernel by the built-in 3-component variables `blockDim` and `gridDim`. Figure 2.1 gives an example of a two-dimensional 3×2 grid with 4 threads per block. Using thread and block identifiers allows distinct threads to operate on separate data and execute different control flow through the kernel.

When a kernel is executed each block of the grid is dynamically scheduled by hardware to a multiprocessor. A multiprocessor can be assigned multiple blocks and execute them concurrently. On each multiprocessor, a block is executed by partitioning it into a set of warps. On NVIDIA hardware a warp is a group of 32 threads. For example a block of 512 threads is partitioned into a set of 16 warps. Warps are dynamically scheduled by hardware on the multiprocessor. A warp instruction is a data-parallel SIMD instruction: each thread is mapped to a core and executes the same operation over multiple data in lock-step. This means that threads in the same warp follow the same control flow and require *predication* to deal with divergent control flow. For example, if threads in the same warp diverge due to a conditional branch then the hardware uses masking to disable threads that do not need to take the branch (disabled threads execute the instruction as a no-op) and then converging threads (by re-enabling masked threads) back to the same execution path afterwards.

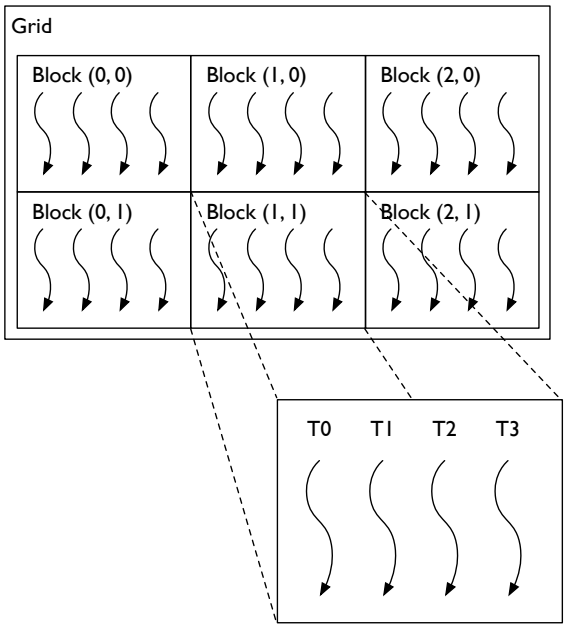


Figure 2.1.: A two-dimensional CUDA grid of 3×2 blocks each consisting of 4 threads

The CUDA memory hierarchy specifies *private* memory per-thread, *shared* memory per-block and *global* memory per-grid. The shared and global memory spaces map to the per-multiprocessor shared and off-chip global memory of the GPU, respectively. Hence, shared memory is visible to all threads in the same block and global memory is visible to all threads.

Threads in the same block can synchronise using a *barrier* operation, `__syncthreads()`. A barrier causes a thread to stall until all threads in the block have reached the same barrier. This allows communication between threads in the same block via shared or global memory since a barrier can be used to coordinate memory accesses. There are no mechanisms for inter-block synchronisation whilst executing a kernel.[4]

Figure 2.2 gives a simple CUDA application that computes a one-dimensional stencil. The stencil kernel to be run on the GPU (also called the device in CUDA) is identified by the `__global__` function qualifier. Given an input array `A` of length n the kernel computes the values of a new output array `B` where each element is the sum of its neighbouring elements. For example, given the input $[0, 1, 2, 3, 4, 5, 6, 7]$ with $n = 8$ and radius 1 the kernel computes $[(0+1), (0+1+2), \ldots, (5+6+7), (6+7)]$ to yield $[1, 3, 6, 9, 12, 15, 18, 13]$. In the

[4] A workaround is to synchronise at the kernel level by splitting the kernel into two and executing the kernels serially. Then all accesses of the first kernel are guaranteed to have completed (and be visible) to any thread of the second kernel.

```
__global__ void stencil(int *A, int *B, int radius, unsigned n) {
  unsigned tid = blockDim.x * blockIdx.x + threadIdx.x;

  int sum = 0;
  for (int i=-radius; i<=radius; i++) {
    int idx = tid + i;
    if (0 <= idx && idx < n) {
      sum += A[idx];
    }
  }

  if (tid < n) B[tid] = sum;
}

void hostStencil(int *A, int *B, int radius, unsigned n) {
  // device copies of A and B
  int *d_A; int *d_B;

  // allocate arrays on device
  size_t sz = sizeof(int)*n;
  cudaMalloc(&d_A, sz); cudaMalloc(&d_B, sz);

  // copy input to device
  cudaMemcpy(d_A, A, sz, cudaMemcpyHostToDevice);

  // launch kernel
  stencil<<<1,n>>>(d_A, d_B, radius, n);

  // copy output from device
  cudaMemcpy(B, d_B, sz, cudaMemcpyDeviceToHost);

  // free allocated arrays
  cudaFree(d_A); cudaFree(d_B);
}
```

Figure 2.2.: A CUDA stencil application

kernel, each thread has a private variable `tid` which is assigned the id of the thread using the built-in variables `threadIdx`, `blockIdx` and `blockDim`. Data parallelism is achieved because each thread computes a different output element: each thread accumulates the sum of its neighbouring elements into a private variable, `sum`, and writes the result to `B`. The kernel has divergent control flow due to the condition $0 \leq$ `idx` $< n$ within the loop. This ensures that threads assigned elements at the start or end of the array will not access out-of-bound locations.

The kernel is launched by the CPU (also called the host in CUDA) in the `hostStencil` function by using the triple bracket syntax `<<<gridDim, blockDim>>>` which specifies the number of threads that will execute the kernel: `gridDim` gives the number of blocks in the grid and `blockDim` gives the number of threads per block. In this instance, the host launches a single group of n threads. Before executing the kernel the host is responsible for allocating the arrays and initialising the input. This is possible because both the host

Table 2.1.: Equivalent terminology for CUDA and OpenCL

Term	**CUDA**	**OpenCL**
Thread	Thread	Work-item
Subgroup	Warp	Subgroup[a]
Group	Block	Work-group
Grid	Grid	NDRange
Local Memory	Shared	Local
Global Memory	Global	Global
Group Synchronisation	`__syncthreads()`	`barrier()`[b]

[a] Introduced by OpenCL 2.0 extension and not exposed in the core feature set of OpenCL [Khr13b]
[b] An OpenCL barrier can specify a set of flags to determine the scope of the barrier with respect to local and global memory [Khr13b]

and device can access the global memory space of the GPU. After allocating two arrays in global memory (accessible through the device pointers `d_A` and `d_B`) the host initialises the input `A` using a memory copy. Then the kernel can be launched and executed in parallel. Scalar formal parameters (i.e., `radius` and `n`) to the kernel are passed directly at invocation. After the kernel has completed the host copies the output back. In this application the malloc, memcpy and kernel execution are synchronous and therefore each call blocks until it completes.[5]

OpenCL OpenCL is an open industry standard for programming heterogeneous systems developed by the Khronos group [Khr13b]. OpenCL bears many similarities with CUDA but is designed to support a wider range of target architectures including multicore CPUs and embedded devices. The terminology also varies from CUDA and we summarise these in Table 2.1. In general, we will use the terminology in the first column except when discussing CUDA or OpenCL directly. Because OpenCL is designed for a range of target architectures the division of a group into subgroup (a warp in CUDA terminology) is not programmer visible in the core OpenCL feature set.[6]

Figures 2.3 and 2.4 give an OpenCL translation of the CUDA stencil example of Figure 2.2. The translation of the kernel is straightforward where we use the built-in function `get_global_id(0)` instead of computing the id of the thread and we are required to annotate the array pointers with the memory space in which they reside. The translation

[5] Concurrent execution of CUDA calls is possible using *streams*. This allows concurrent kernel execution and overlapped kernel computation and data transfer between the host and device. In the simple example of Figure 2.2 we rely on the *default stream* which is implicitly in-order and synchronised.
[6] Subgroups were introduced in the OpenCL 2.0 extension and are an implementation-defined feature.

```
__kernel void stencil(__global int *A, __global int *B, int radius, unsigned n) {
  unsigned tid = get_global_id(0);

  int sum = 0;
  for (int i=-radius; i<=radius; i++) {
    int idx = tid + i;
    if (0 <= idx && idx < n) {
      sum += A[idx];
    }
  }

  if (tid < n) B[tid] = sum;
}
```

Figure 2.3.: OpenCL version of the stencil kernel in Figure 2.2

of the host code requires additional setup to choose a target device. The code also shows that kernels can be compiled at runtime in OpenCL.

2.1.4. Data Races and Barrier Divergence in GPU Kernels

Data Races In this thesis we distinguish between two types of data race in GPU kernels. An *inter-group data race* occurs if there exist two distinct threads s and t in *different* groups such that: (i) s writes to a location in global memory and (ii) t writes to or reads from the same location. An *intra-group data race* occurs if there exist two distinct threads s and t in the *same* group such that: (i) s writes to a location in global or local memory, (ii) t writes to or reads from the same location and (iii) there is no intervening barrier synchronisation.

GPU programming models admit further kinds of data race that we do not consider further. An *inter-kernel data race* occurs if there exist two distinct threads s and t in *different kernels* such that (i) s writes to a location in global memory and (ii) t writes to or reads from the same location. This is possible in CUDA and OpenCL due to the ability to launch and execute multiple kernels on the same GPU. Ignoring this scenario is akin to assuming that multiple kernels must be launched and executed serially. A *host-kernel race* occurs if there exists a thread s in the kernel such that: (i) s writes to a location in global memory and (ii) the host writes to or reads from the same location; or, the reverse situation where (i) the host writes to a location in global memory and (ii) s writes to or reads from the same location. This is possible in CUDA and OpenCL due to the ability for the host to continue executing whilst a kernel executes on the GPU. Ignoring this scenario is akin to assuming that the host blocks until a kernel terminates.

```
void hostStencil(int *A, int *B, int radius, unsigned n) {
  // setup OpenCL platform and device
  cl_platform_id platform;
  cl_device_id device;
  clGetPlatformIDs(1, &platform, NULL);
  clGetDeviceIDs(platform, CL_DEVICE_TYPE_DEFAULT, 1, &device, NULL);
  cl_context context = clCreateContext(NULL, 1, &device, NULL, NULL, NULL);
  cl_command_queue queue = clCreateCommandQueue(context, device, 0, NULL);

  // fetch and compile kernel
  std::ifstream f("stencil.cl", std::ios::in);
  std::stringstream ss;
  ss << f.rdbuf();
  const std::string& str = ss.str();
  const char *s = str.c_str();
  cl_program program = clCreateProgramWithSource(context, 1, (const char **)&s, NULL, NULL);
  clBuildProgram(program, 1, &device, "", NULL, NULL);
  cl_kernel kernel = clCreateKernel(program, "stencil", NULL);

  // device copies of A and B
  cl_mem d_A; cl_mem d_B;

  // allocate arrays on device
  size_t sz = sizeof(int)*n;
  d_A = clCreateBuffer(context, CL_MEM_READ_ONLY, sz, NULL, NULL);
  d_B = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sz, NULL, NULL);

  // copy input to device
  clEnqueueWriteBuffer(queue, d_A, CL_TRUE, 0, sz, A, 0, NULL, NULL);

  // launch kernel
  clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&d_A);
  clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&d_B);
  clSetKernelArg(kernel, 2, sizeof(int), (void *)&radius);
  clSetKernelArg(kernel, 3, sizeof(unsigned), (void *)&n);
  size_t global_work_size = n; size_t local_work_size = n;
  clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &global_work_size, &local_work_size, 0, NULL, NULL);
  clFinish(queue);

  // copy output from device
  clEnqueueReadBuffer(queue, d_B, CL_TRUE, 0, sz, B, 0, NULL, NULL);

  // free allocated arrays
  clReleaseMemObject(d_A);
  clReleaseMemObject(d_B);
}
```

Figure 2.4.: OpenCL version of the stencil host code in Figure 2.2

(a)

```
if ((tid % 2) == 0)
  barrier();
else
  barrier();
```

(b)

```
x = (tid == 0) ? 1 : 4;
y = (tid == 0) ? 4 : 1;
for (i=0; i<x; i++) {
  for (j=0; j<y; j++) {
    barrier();
  }
}
```

Figure 2.5.: Examples of barrier divergence. The examples are taken and adapted from prior work [BCD+12].

Barrier Divergence We also consider a further erroneous scenario for GPU kernels. The semantics of barriers means that all threads in the same group must synchronise at the same syntactic barrier and furthermore, if the barrier is in a loop then every thread must reach the barrier having executed the same number of loop iterations [BCD+12]. If this is not the case then the kernel is *barrier divergent* and the behaviour of the kernel is undefined [NVI12a, Khr13b]. The examples in Figure 2.5 exhibit barrier divergence. In the code we use `tid` to denote the id of a thread. Example (a) is barrier divergent because it allows different syntactic barriers to be reached. Due to the if-statement, threads with even ids reach the first barrier while threads with odd ids reach the second barrier. In example (b), the same syntactic barrier is reached by all threads but is barrier divergent because not all threads execute the same number of loop iterations. Thread 0 will execute the outer loop once and the inner loop four times; whereas, all other threads will execute the outer loop four times and the inner loop once per iteration of the outer loop. We discuss barrier divergence in more detail in Section 3.3.3.

2.2. Race Checking Techniques for GPU Kernels

In the following we review techniques for detecting races in GPU kernels. We can broadly classify the techniques as *dynamic* or *static*. Dynamic techniques involve running the program on concrete test cases to derive information about the executions of the program; whereas, static techniques use program verification to reason about all executions of the program without having to run it. Generally, we see that dynamic techniques are better suited to bug-finding whereas static techniques are better suited to verification; this is true beyond the domain of GPU kernels.

2.2.1. Dynamic Instrumentation

Given a parallel program, dynamic instrumentation works by modifying (instrumenting) and executing the program to dynamically record all memory accesses. Subsequently, we can determine whether the execution has a data race by examining the logs for conflicting accesses. This technique is simple to apply and races that are uncovered are true bugs (no false positives) because the program is executed concretely. The main weakness of dynamic instrumentation is that it cannot give guarantees of correctness since it only checks a single test case (out of many). Hence, dynamic instrumentation is a bug-finding technique rather than a verification technique.

Tools that use dynamic instrumentation include work by Boyer et al. [BSW08], cuda-memcheck [NVI12b] and GRace [ZRQA11]. The GRace tool additionally uses static analysis to avoid logging of accesses that can be conservatively assumed to be safe. This helps reduce the runtime overheads of the dynamic instrumentation, which can otherwise lead to large (order of magnitude) slowdowns over the uninstrumented program.

2.2.2. Dynamic Symbolic Execution

Dynamic symbolic execution uses *concolic* (concrete and symbolic) execution to explore the executions of a program. Under concolic execution the program is executed in a symbolic virtual machine (VM) where program variables can be marked as concrete or symbolic. Symbolic variables are initially unconstrained. During execution the VM can *fork* execution paths whenever it encounters a non-deterministic situation (e.g., a conditional where both choices can be true or dereferencing a symbolic pointer that may point to multiple objects). Execution can be forced along a particular path by generating constraints over symbolic variables. For example, if execution reaches an if-statement with condition `(x == 0)` where x is an unconstrained symbolic variable then we will explore execution paths where the condition holds ($x = 0$) and does not ($x \neq 0$). In this way, possible execution paths are explored.

The main advantage of this technique is that it executes the program and so does not suffer from false positives. Additionally, if a bug is uncovered (e.g., an assertion failure or a null pointer dereference) then the constraints over symbolic variables (called the *path condition*) can be used to automatically generate a concrete test case. The main disadvantage of dynamic symbolic execution is that it cannot give guarantees of correctness for non-trivial programs. With enough time, exhaustive exploration of all execution paths can give a brute-force verification guarantee. However, the *path explosion* problem means that this is infeasible for non-trivial programs. Hence, this technique is better suited to

bug-finding than verification.

Tools that have applied dynamic symbolic execution to GPU kernels include GKLEE [LLS+12] and KLEE-CL [CCK11b]. In these tools the symbolic VM is adapted to handle multiple threads and other features of GPU programming models: GKLEE handles CUDA and KLEE-CL handles OpenCL. Both GKLEE and KLEE-CL have uncovered data race bugs in real-world examples. A key result used by GKLEE is that, for the purpose of checking for data races, it is sufficient to consider a single schedule: the *canonical schedule*. Informally, this tells us that if a kernel is racy then all schedules are capable of uncovering a race (although, not necessarily exactly the same race). We refer to this result as the *canonical schedule result*, which was given in earlier work by Li and Gopalakrishnan [LG10].[7] GKLEE uses this result to avoid symbolically executing all schedules; it need only consider a single canonical schedule. A second improvement to the performance of GKLEE is given by the notion of a *barrier interval* (introduced in the same work by Li and Gopalakrishnan [LG10]). A barrier interval is the set of instructions occurring between the execution of two barriers. Because threads within the same group synchronise at barriers it is impossible for intra-group races to occur across barriers. GKLEE uses this result to consider one barrier interval at a time: dynamically executing the canonical schedule from one barrier to the next.

GKLEE and KLEE-CL explicitly represent the number of threads executing the GPU kernel. GKLEE$_p$ is an extension to GKLEE that can avoid having to represent all threads using *parametric flows*. The parametric flows of a kernel are equivalence classes that partition the threads of the kernel. Threads in the same parametric flow share the same thread-dependent control flow. The idea is that the behaviours of all threads in the same parametric flow can be reduced to a single representative thread. Hence only pairs of representative threads in different parametric flows need to be considered for race checking. This is a form of thread reduction and gives scalability improvements over GKLEE.

2.2.3. Hybrid Techniques

Hybrid techniques uses both dynamic and static techniques in conjunction. Li et al. have extended the ideas of GKLEE and GKLEE$_p$ into a hybrid tool, SESA [LLG14], combining dynamic symbolic execution and static analysis. The key idea is to use a taint analysis to identify kernel parameters that can be made concrete without affecting path coverage. This information is used to merge parametric flows and hence give scalability improvements over GKLEE$_p$.

[7] This work introduced the PUG verifier, which we discuss in Section 2.2.4.

Work by Leung et al. [LGA+12] has applied *test amplification* to GPU kernels. The idea of test amplification is to generalise the results of a single dynamic run into a stronger property about all program behaviours. Initially, a CUDA kernel is instrumented and executed with dynamic race checking enabled. If no race is uncovered then the race-freedom result (of the single dynamic run) can be amplified using a static flow analysis. The static flow analysis determines whether the kernel is *access invariant*: if the kernel always issues the same set of memory accesses *regardless* of the kernel inputs. The analysis is a taint analysis that conservatively tracks the inputs of the kernel and ensures that (i) no tainted variable is used in the address computation of any memory access and (ii) no memory access is control-dependent on a tainted variable. If the kernel is access invariant then race-freedom can be amplified to all executions of the program. At a high-level, test amplification can be viewed as a form of dynamic symbolic execution that reduces the number of paths that need to be explored to the path associated with a single test case. For the purposes of race checking all executions of an access invariant kernel are equivalent. The main advantage of this technique is that it combines the strengths of dynamic and static approaches. However, test amplification is inapplicable for *access variant* kernels and is difficult to generalise to richer functional properties.

2.2.4. Static Verification

The following techniques are program verifiers using static verification. Unlike dynamic techniques, static verification can offer guarantees of correctness. However, a recurring weakness of these techniques is the need for *invariants*. An *invariant* is a property that captures program behaviours by expressing a fact that always holds at a particular program point (e.g., that a variable is always greater than 0). Invariants are vital in static verification for precise reasoning and avoiding false positives. We discuss invariants in detail in Chapter 3.

Permission-Based Separation Logic Work by Blom et al. [BHM14] has applied permission-based separation logic for proving data race-freedom and functional specifications of GPU kernels. Separation logic [Rey02] is an extension to Floyd-Hoare logic that enables reasoning about shared resources such as dynamically-allocated memory (i.e., the heap). A key property is the ability to write assertions that split the resource into mutually exclusive disjoint parts. This is a useful property for concurrent programs since two concurrent procedures operating on disjoint resources cannot interfere and thus can be verified separately. Permission-based separation logic allows resources (e.g., memory

locations) to be tagged with a *permission* which allows us to express sharing of resources. The full permission 1 provides exclusive access and means that the resource can be inspected (i.e., the memory location can be read) and updated (i.e., the memory location can be written to). A fractional permission in the interval $(0, 1)$ provides non-exclusive access and means that the resource can only be inspected (but not updated). A permission can be split and combined. For example, a full (write) permission can be split into multiple fractional (read) permissions, and vice versa by combining. Importantly, we can only re-acquire a full permission by combining *all* the fractions into which it was split.

Applying this to GPU kernels means that if we can specify the permissions of a kernel with respect to shared state then the kernel is race-free. This is because the soundness of the logic means that only a single thread may hold a full write permission at any point of the kernel. The authors have implemented this method as a program verifier on top of the Chalice tool [LMS09]. This approach allows full functional correctness of GPU kernels to be proven. The main disadvantage of this technique is that non-trivial kernel annotations and invariants must be manually provided and hence this technique cannot in its current form be used for automatic race checking.

Non-Interference Checking The work of Tripakis et al. [TSL10] is concerned with proving the equivalence of SPMD programs (in particular, CUDA kernels). For example, showing that an optimised kernel is functionally equivalent to a naive implementation. The work focuses on race-freedom (called non-interference by the paper) because this is critical for showing that a GPU kernel is deterministic.[8] Race-freedom is checked by generating a verification condition that asserts that all pairwise memory accesses made by different threads are to distinct locations. This entails generating a quadratic number of inequalities. The verification condition is then passed to an SMT solver. The tool implementing this work is limited by loops and the need for invariants, which must be manually provided.

PUG The PUG tool [LG10] verifies GPU kernels for race-freedom by encoding a similar verification condition as the technique by Tripakis et al. Given a CUDA program, PUG generates a verification condition that reflects the accesses of the program from the point of view of two symbolic (arbitrary) threads. PUG uses the canonical schedule result to reduce the number of thread schedules that must be considered. PUG also uses the idea of barrier intervals to incrementally consider one barrier interval at a time.

[8] This result is also used in the Test Amplification work of Leung et al. [LGA+12] and is similar to the canonical schedule result [LG10]. We require and prove an analogous result in Chapter 5.

PUG addresses the problem of generating loop invariants by incorporating loop abstraction techniques. The tool recognises certain loop patterns and encodes invariants into the verification condition. For example, in the following CUDA loop where `A` is an array in shared memory:

```
for (int i=threadIdx.x; i<=N; i+=blockDim.x) {
  A[i] = threadIdx.x;
}
```

each thread writes to elements of `A` at intervals of `blockDim.x` and hence the loop is race-free. For example, if $N = 15$ and `blockDim.x` $= 4$ then thread 0 will write to elements 0, 4, 8 and 12. PUG recognises this striding behaviour and *normalises* the loop into the form:

```
for (int j=0; j<=(N-threadIdx.x)/blockDim.x; j++) {
  A[j*blockDim.x+threadIdx.x] = threadIdx.x;
}
```

where the new loop index j is incremented by one at each iteration. PUG inserts the invariants $0 \leq j_t < (N-t)/B$ and $i_t = j_t \cdot B + t$ where t and B denote an arbitrary thread id and the block dimension, respectively, and we use i_t and j_t to denote the thread-private variables of t. The verification condition for the loop is then:

$$\begin{array}{lll} \textit{Symbolic thread ids} & 0 \leq s < B \wedge 0 \leq t < B \wedge s \neq t & \wedge \\ \textit{Loop invariants} & 0 \leq j_s < (N-s)/B \wedge 0 \leq j_t < (N-t)/B & \wedge \\ & i_s = (j_s \cdot B) + s \wedge i_t = (j_t \cdot B) + t & \wedge \\ \textit{Race-freedom} & i_s = i_t & \end{array}$$

where we use the symbolic variables s and t to represent two distinct threads. Because the verification condition is unsatisfiable PUG infers that the loop is race-free. Without the invariants the variables i_t and j_t would be unconstrained and lead to a false positive.

The need for invariants is the largest source of imprecision in PUG. The tool allows the programmer to annotate loops with invariants. However, these user-supplied invariants (as well as the ones inserted automatically by PUG) are assumed to be true rather than being checked and hence are a potential source of unsoundness (leading to false negatives).

GPUVerify GPUVerify is a program verifier for GPU kernels [BCD+12, BBC+14]. At a high-level, GPUVerify is similar to PUG and the work of Tripakis et al. The principal difference is the technique used to generate verification conditions. GPUVerify also exploits the property that race-freedom is a pairwise property but, rather than directly encoding the kernel as a logical formula, GPUVerify translates the kernel into a sequential program

that models the behaviour of an arbitrary pair of threads; the correctness of this program implies the race-freedom of the given kernel. This has the advantage that the soundness of GPUVerify depends only on (i) showing that the translation is sound and (ii) applying a sound program verifier to the resulting sequential program. A second advantage is that GPUVerify can use standard sequential verification technology for verifying the sequential program. GPUVerify translates the kernel into a sequential Boogie [BCD+05] program. Section 2.3 gives a detailed description of the transformation of a kernel into a sequential program.

After transforming the program GPUVerify also generates loop invariants to avoid spurious errors. Unlike PUG, these are checked and so an inconsistent invariant cannot lead to false negatives.

2.3. Kernel Transformation

We now review the verification method of GPUVerify, which has also been explained in prior work [BCD+12, BBC+14]. As discussed, the essential idea of GPUVerify is to translate a parallel GPU kernel K into a sequential program P such that the correctness of the program P implies the race-freedom of the kernel K. We refer to this as the *kernel transformation*. The kernel transformation involves three steps: predication, race instrumentation and a two-thread reduction with shared state abstraction.

2.3.1. Predication

A *predicated statement* $p \Rightarrow \mathsf{stmt}$, where p is a predicate and stmt is a statement, has the same effect as stmt if p is true and behaves as a no-op if p is false. Predication allows us to convert control flow into data flow [AKPW83] and is essential for handling loops and conditions using a lock-step semantics. Predication introduces new thread-private variables which determine whether a thread is enabled or disabled at each statement. An if-statement can be predicated by introducing a fresh thread-private variable p into which all threads evaluate the condition, subsequently all threads execute both the **then** and **else** branches of the condition, predicated by p and $\neg p$, respectively. Similarly, a loop can be predicated by introducing a fresh thread-private variable p into which all threads evaluate the guard, subsequently all threads repeatedly execute the loop body (including the re-evaluation of the guard) under the predicate p until the guard is false for all threads. Predication ensures that threads that do not need to continue the execution of the loop (i.e., if they evaluate the guard to false) will execute the statements in the body of the

loop as no-ops.

Predication is the key transformation for yielding a sequential program. The parallel kernel is transformed into a lock-step sequential program where each thread follows the same control flow. Predication chooses a fixed schedule that removes reasoning about thread interleavings. The soundness of this approach relies on the canonical schedule result [LG10]. Work by Collingbourne et al. [CDKQ13] has shown the equivalence between lock-step semantics and interleaving semantics for GPU kernels.

2.3.2. Two-Thread Reduction with Shared State Abstraction

An important observation used by PUG and GPUVerify is that data race-freedom and barrier divergence are *pairwise properties*: a race occurs when accesses by two threads conflict and barrier divergence occurs when two threads reach a barrier and one thread is enabled and the other is disabled. Based on this observation we consider a translation of the kernel where we only model the execution of a pair of threads. If we can prove this translated program is race-free and divergence-free for a pair of distinct but otherwise *arbitrary* pair of threads then we have shown that the original kernel is race-free and divergence-free for all pairs of distinct threads (i.e., the kernel is race-free and divergence-free). We call this translation the *two-thread reduction*. The two-thread reduction is the key transformation for yielding a tractable sequential program for scalable verification.

As part of the two-thread reduction we *duplicate* thread and group id variables so that each thread has a private copy. A precondition over these variables ensures that we choose a distinct and arbitrary pair of threads. As a simplified example, for a one-dimensional group of N threads, we introduce two thread ids `tid$1` and `tid$2` corresponding to the id of the first and second thread, respectively, with the following precondition:

$$0 \leq \texttt{tid\$1} < N \wedge 0 \leq \texttt{tid\$2} < N \wedge \texttt{tid\$1} \neq \texttt{tid\$2}$$

This straightforwardly generalises to multi-dimensional groups and grids with multiple groups.

Variable duplication also applies to thread-private variables. Each thread-private variable v is duplicated into a pair of `v$1` and `v$2` variables. This includes scalar formal parameters to the kernel. More generally, given an expression e we can *dualise* it into e\$1 and e\$2 expressions that correspond to the expression for the first and second thread, respectively. For example, the expression `(v * w) + tid` where v and w are thread-private variables is dualised into `(v$1 * w$1) + tid$1` and `(v$2 * w$2) + tid$2`.

For soundness the two thread-reduction must abstract shared state. Intuitively, the

```
assume (∀ x : A[x] == x);
A[tid] = 0;
barrier();
if (tid != N-1) {
  B[A[tid+1]] = tid;
}
```

Figure 2.6.: A counterexample showing the necessity of shared state abstraction

purpose of the *shared state abstraction* is to over-approximate the behaviour of threads not modeled by the two-thread reduction. In particular, the shared state abstraction determines the *contents* of shared state, as seen by each thread.

Necessity of Shared State Abstraction Figure 2.6 gives a counterexample showing that the two-thread reduction is unsound if we do not abstract shared state. The code uses `tid` to denote the id of a thread and `A` and `B` are arrays in shared state. We assume that the kernel is executed by a single group of N threads and that initially $\texttt{A}[x] = x$ for all elements x. The example has a race because all threads set their corresponding element of `A` to 0 and so, after the barrier, all threads (except for thread $N-1$) will race to write to `B`[0]. Suppose that we do not abstract shared state. Then, because every thread s only writes 0 to its corresponding element $\texttt{A}[s]$ and has *not* modified `A`[s+1] we erroneously preserve the initial condition that $\texttt{A}[s+1] = s+1$. This holds similarly for another distinct thread t so that the writes to `B` will be disjoint: thread s and t will write to $\texttt{B}[s+1]$ and $\texttt{B}[t+1]$, respectively (if the threads s and t are adjacent when $s+1 = t$ then s will write to `B`[0] and t will write to $\texttt{B}[t+1]$). That is, a race will not be reported.

A proof of the soundness of the two-thread reduction with shared state abstraction is given in prior work [BCD+12]. The simplest and coarsest abstraction is the *adversarial abstraction*, employed by both GPUVerify and PUG, where shared state is removed.[9] Under this abstraction all reads to shared state receive arbitrary values and all writes to shared state are no-ops. A second abstraction called the *equality abstraction*, employed only by GPUVerify, models a *consistent* but still arbitrary shared state. Under this abstraction reads from the same location receive consistent, but still arbitrary, values and all writes to shared state remain as no-ops. The essential difference is that reads by different threads will see the same value under the equality abstraction (as long as neither thread has written to the location, which is a race condition); this is not guaranteed under the adversarial abstraction. We discuss this further and develop richer abstractions for shared

[9] PUG implicitly uses the adversarial abstraction by leaving the values read from shared state unconstrained.

state in Chapter 4.

2.3.3. Race Instrumentation

The purpose of race instrumentation is to record accesses made by the kernel for race checking. Intuitively, race instrumentation allows us to track the read and write sets of an arbitrary pair of distinct threads (introduced by the two-thread reduction). Then a race has occurred if the write set of the first thread intersected with the read or write set of the second thread is non-empty. A direct encoding of sets requires quantifiers and is undesirable since quantifiers are difficult for SMT solvers to reason about.[10] Instead we give two quantifier-free encodings for race instrumentation given in prior work: a *non-deterministic set* encoding [BCD+12] and a *watchdog* encoding [BBC+14].

For simplicity we initially describe both encodings omitting predication and limited to checking intra-group data races (ignoring inter-group data races). Ignoring inter-group races simplifies the presentation because we can treat arrays in local and global memory spaces equivalently. We discuss how to lift these restrictions when we summarise the encodings at the end of this section. In the following we refer to an array in *shared state* to mean an array in either memory space.

Non-Deterministic Set Encoding Under this encoding we represent the read and write sets implicitly by exploiting non-determinism, as introduced by work from Donaldson et al. on DMA (direct memory access) race analysis for heterogeneous multicore programs [DKR11]. For each array `A` in shared state we introduce

- Boolean variables `rdExistsA` and `wrExistsA`, and
- integer variables `rdOffsetA` and `wrOffsetA`.

The pair (`rdExistsA`, `rdOffsetA`) forms an *option type* that can log *at most one* read access from the array. If `rdExistsA` is `true`, then `rdOffsetA` is the (byte) offset of an access to `A` that has been logged; otherwise, `rdExistsA` is `false` and no read access has been captured

[10] We can encode a set as a map of type $\mathsf{Int} \rightarrow \mathsf{Bool}$ giving the *characteristic function* of the set. That is, if an element i is a member of the set then i is mapped to `true` and mapped to `false` otherwise. Using this encoding requires quantifiers to (i) initialise and empty the set (i.e., to assume for all elements i that the map of i is `false`) and (ii) express assertions (in particular, loop invariants) about the elements of the set (e.g., the property $P(i)$ holds for all elements i where the map of i is `true`). Therefore, due to the *loop-cutting transformation* (Section 3.1.1), this entails both asserting and assuming a quantified property for each abstracted loop. An early prototype of GPUVerify used this direct encoding but moved to quantifier-free encodings due to performance issues [BCD+12].

(the value of `rdOffsetA` is irrelevant). Initially the `rdExistsA` and `wrExistsA` variables are false and they are reset at each barrier.

Each read from `A` in the original kernel is translated into a call to a logging procedure and a second call to a checking procedure. The logging procedure (respectively, checking procedure) is called by the first thread (respectively, second thread) of the two-thread reduction. This asymmetric treatment of threads is sound because the two-thread reduction will consider all pairs of threads: i.e., if the pair (s, t) is considered then so will (t, s). Each call is passed the offset of the read (in bytes), `offset`. The logging procedure non-deterministically decides to *do nothing* or capture the read access by setting `rdExistsA` to `true` and setting `rdOffsetA` to `offset`. The non-determinism allows us to capture an *arbitrary* element of the read set. The machinery is analogous for the logging of write accesses to `A` using the pair (`wrExistsA`, `wrOffsetA`). The checking procedures assert that the intersection of the read and write sets is empty. That is, the read checking procedure asserts

$$\texttt{wrExistsA} \Rightarrow \texttt{wrOffsetA} \neq \texttt{offset}$$

and the write checking procedure asserts

$$\begin{array}{lcl} (\texttt{wrExistsA} & \Rightarrow & \texttt{wrOffsetA} \neq \texttt{offset}) \quad \wedge \\ (\texttt{rdExistsA} & \Rightarrow & \texttt{rdOffsetA} \neq \texttt{offset}). \end{array}$$

The non-deterministic logging of accesses that ensures that all possible pairs of accesses will be checked.

At a barrier we must reset the tracking of accesses. An implementation choice for achieving this is to *assign* false to the Boolean variables `rdExistsA` and `wrExistsA`, which is analogous to clearing the read and write sets. However, assignment is problematic for loops containing barriers. In this case, both `rdExistsA` and `wrExistsA` will be added to the *modset* (modifies set) of the loop, even if the loop never accesses the array being instrumented. Consequently, due to the loop-cutting transformation for verifying loops (discussed in Section 3.1.1), the loop will require invariants to constrain these variables. We can avoid this problem by *assuming* that `rdExistsA` and `wrExistsA` are false at a barrier, which has the same effect of clearing the read and write sets but without using assignment. We are free to do this because there always exists a path through the program where each non-deterministic choice for capturing an access chose false.

Watchdog Under the non-deterministic set encoding, each array access leads to a non-deterministic choice (in the logging procedure) so the number of paths through the in-

strumented program grows exponentially with the number of accesses. This motivated the development of a different encoding to avoid this blow-up. The idea of watchdog race checking is to use a single non-deterministic offset that all accesses are checked against. We refer to this offset as the *watched offset*. Verification involves proving for every array that a data race at the watched offset is impossible. Hence, because the watched offset is arbitrary, this implies that every offset of every array is race-free.

The translation introduces the watched offset as an integer variable `trOffset`. The watched offset is used to check the whole program and we use the same offset for checking all arrays. For each array `A` in shared state we introduce Boolean variables `rdExistsA` and `wrExistsA`. Initially these variables are false and they are reset at each barrier.

As in the non-deterministic set encoding we transform every read into a call to a logging procedure and a call to a checking procedure; where the logging procedure (respectively, checking procedure) is called by the first thread (respectively, second thread) of the two-thread reduction. The logging procedure sets the `rdExistsA` variable if the offset being read matches the watched offset. The machinery is analogous for the logging of write accesses, which sets the `wrExistsA` variable if the offset being written to matches the watched offset. The checking procedures assert that a data race at the watched offset has not occurred. The read checking procedure asserts

$$\texttt{wrExistsA} \Rightarrow \texttt{trOffset} \neq \texttt{offset}$$

and the write checking procedure asserts

$$\begin{array}{lcl} (\texttt{wrExistsA} & \Rightarrow & \texttt{trOffset} \neq \texttt{offset}) \quad \wedge \\ (\texttt{rdExistsA} & \Rightarrow & \texttt{trOffset} \neq \texttt{offset}). \end{array}$$

As for the non-deterministic set encoding, we need to avoid assigning to the Boolean variables `rdExistsA` and `wrExistsA` at each barrier. However, we cannot simply assume that these variables are false at a barrier as we did for the non-deterministic set encoding. This is because the watchdog encoding, as described above, *always* captures an access in the logging procedure and hence, there does *not* always exist a path where the variables `rdExistsA` and `wrExistsA` are false. This is equivalent to assuming false, which is unsound (all assertions in the execution after this point will pass vacuously). To avoid this problem we introduce a new Boolean variable `tracking` that we *havoc* (assign an arbitrary value) at each barrier and use to non-deterministically choose whether logging occurs or not. Using this variable means that there is always a path where the logging variables are false (when the non-deterministic choice at each barrier for the value of `tracking` chose false)

and hence we can soundly use assume statements to reset the logging variables at a barrier.

Under the watchdog encoding the number of paths through the instrumented program grows exponentially with the number of barriers compared to exponentially with the number of array accesses under the non-deterministic set encoding. Because the number of barriers in a kernel is typically much smaller than the number of array accesses we expect the watchdog encoding to lead to faster verification. We confirmed this experimentally in prior work [BBC+14].

Summarising the Encodings Table 2.2 summarises the two encodings in the presence of predication. With predication we account for the predicate p, which we pass to the logging, checking and barrier procedures. If p is true then the logging, checking and barrier procedures function as before. If p is false then the logging procedure will do nothing (no-op) and the checking and barrier procedures will vacuously pass. The addition of predicates allows us to check for barrier divergence within the barrier procedure. We assert that both threads are either enabled or disabled (i.e., $p\$1 = p\$2$). This precisely captures the notion of barrier divergence formalised in prior work (see the G-DIVERGENCE rule of [BCD+12]).

We can lift the encodings to handle inter-group data races by case splitting when (i) the arbitrary pair of threads under consideration by the two-thread reduction are in the same group or not and (ii) if the instrumented array resides in local or global memory. The instrumentation variables introduced by the encodings remain the same. If the pair of threads are in the same group then the procedures are as before. If the pair of threads are in different groups and:

- if the array resides in local memory then race checking (for this array) can be omitted altogether since this scenario guarantees that the threads can never race since they will access separate group-only copies of the array; otherwise,
- if the array resides in global memory then the logging and checking procedures are as before, but barriers cannot reset the instrumentation variables since there are no mechanisms for inter-group synchronisation in CUDA and OpenCL [NVI12a, Khr13b].

2.3.4. Kernel Transformation Summary and Example

Table 2.3 summarises the kernel transformation for a simple structured kernel language showing predication, race instrumentation and the two-thread reduction with the adversarial shared state abstraction. The transformation takes a statement stmt and a predicate

Table 2.2.: Non-deterministic set and watchdog encodings for race instrumentation. The predicate `p` is introduced by predication and determines whether accesses are tracked and also whether barrier divergence has occurred. The star `*` denotes an arbitrary Boolean value.

	Encoding	
	Non-deterministic set	**Watchdog**

Per-program instrumentation variables		`bool tracking; // unconstrained`
		`const int trOffset; // unconstrained`

For each array A in shared state introduce	`bool rdExistsA; // initially false`
	`bool wrExistsA; // initially false`
	`int rdOffsetA;`
	`int wrOffsetA;`
		`bool rdExistsA; // initially false`
		`bool wrExistsA; // initially false`

Procedure `LogRdA(int offset, bool p)` defined	`if (p && *) {`
	`  rdExistsA = true;`
	`  rdOffsetA = offset;`
	`}`
		`if (p &&`
		`  tracking && trOffset == offset) {`
		`  rdExistsA = true;`
		`}`

Procedure `LogWrA(int offset, bool p)` defined	`if (p && *) {`
	`  wrExistsA = true;`
	`  wrOffsetA = offset;`
	`}`
		`if (p &&`
		`  tracking && trOffset == offset) {`
		`  wrExistsA = true;`
		`}`

Procedure `ChkRdA(int offset, bool p)` *asserts*	`(p && wrExistsA ⇒` `  wrOffsetA != offset)`	`(p && wrExistsA ⇒` `  trOffset != offset)`

Procedure `ChkWrA(int offset, bool p)` *asserts*	`(p && wrExistsA ⇒`
	`  wrOffsetA != offset) &&`
	`(p && rdExistsA ⇒`
	`  rdOffsetA != offset)`
		`(p && wrExistsA ⇒`
		`  trOffset != offset) &&`
		`(p && rdExistsA ⇒`
		`  trOffset != offset)`

Procedure `Barrier(bool p$1, bool p$2)` defined	`assert (p$1 == p$2);`
	`assume (p$1 ⇒`
	`  !rdExistsA && !wrExistsA);`
		`assert (p$1 == p$2);`
		`assume (p$1 ⇒`
		`  !rdExistsA && !wrExistsA);`
		`havoc(tracking);`

```
__kernel void nbor(__local int *A, unsigned i, unsigned n) {
  int v, w;
  unsigned tid = get_local_id(0);
  v = 0;
  if (tid + i < n) {
    v = A[tid + i];
  }
  w = A[tid];
  // barrier required here
  A[tid] = v + w;
}
```

Figure 2.7.: A simple OpenCL kernel containing a data race. The intention of the kernel is to add the `i`th neighbouring element to each element of the array.

p (initially true) and models the execution of `stmt` under p for two arbitrary threads. We use the notation $e\$1$ and $e\$2$ to refer to an expression e that has been dualised for the first and second thread considered by the two-thread reduction, respectively. For example, the assignment `v = e` is translated into two statements, one assignment for each thread.

As an example we consider the transformation of the OpenCL kernel in Figure 2.7. The kernel is a simpler version of the stencil kernel given in Figure 2.3. The main difference is that the array `A` is now both the input and output of the kernel. The intention of the kernel is to take each element of the array `A` and sum in the ith neighbour. We assume the array is of length n and test to make sure that we do not cause an out-of-bounds array access. This kernel contains a data race if $i \neq 0$ because there is no synchronisation between the read of the neighbour `A[tid+i]` into `v` and the write of the result into `A[tid]`. For example, if $i = 1$ then thread 0 will read from and thread 1 will write to `A[1]`. Figure 2.8 gives the transformed kernel after predication, the two-thread reduction using the adversarial shared state abstraction and race instrumentation.

2.3.5. Further Considerations

The kernel transformation that we have presented covers the core ideas of GPUVerify. However there are many further aspects, addressed by other GPUVerify papers [BCD+12, CDKQ13, BD14, BBC+14], that are important for building a practical program verifier. We discuss the most important issues here.

Pointers Reasoning about pointers is a long-standing problem for verification due to aliasing. However, in prior work [BCD+12], we observe that GPU kernels do not typically use complicated pointer manipulation. This enables a straightforward approach used in GPUVerify [BCD+12]. Pointers are modeled as a pair of an array enumeration `base` and

Table 2.3.: Kernel transformation for structured programs showing predication, race instrumentation and the two-thread reduction with the adversarial shared state abstraction. The predicated statement $p \Rightarrow$ stmt has the same effect as stmt if p is true and behaves as a no-op if p is false. We simplify the predicated statement true $\Rightarrow$ stmt by writing stmt. The race instrumentation and barrier procedures take the predicates of each thread as arguments.

stmt	*translate*(stmt, p)
`v = e;`	`p$1 ⇒ v$1 = e$1;` `p$2 ⇒ v$2 = e$2;`

`x = A[e];`	`LogRdA(e$1, p$1);`
	`ChkRdA(e$2, p$2);`
	`p$1 ⇒ v$1 = *; // adversarial abstraction`
	`p$2 ⇒ v$2 = *; //`

`A[e] = f;`	`LogWrA(e$1, p$1);`
	`ChkWrA(e$2, p$2);`
	`// writes become no-ops`

`barrier();`	`Barrier(p$1, p$2);`
`S; T;`	*translate*`(S, p);` *translate*`(T, p);`

`if (e) { S } else { T }`	`// q and r are fresh`
	`q$1 = p$1 &&  e$1;`
	`q$2 = p$2 &&  e$2;`
	`r$1 = p$1 && !e$1;`
	`r$2 = p$2 && !e$2;`
	translate`(S, q);`
	translate`(T, r);`

`while (e) { S }`	`// q is fresh`
	`q$1 = p$1 && e$1;`
	`q$2 = p$2 && e$2;`
	`while (q$1 || q$2) {`
	`  // translate body`
	`  translate(S, q);`
	`  // re-evaluate guard`
	`  q$1 = q$1 && e$1;`
	`  q$2 = q$2 && e$2;`
	`}`

```
// Two-thread reduction introduces 2 arbitrary threads
int tid$1;
int tid$2;

// Assume there are N threads in total
const int N;
axiom N == 8;

// Shared state abstraction removes declaration of array A
// Two-thread reduction dualises scalar formals i and n
procedure nbor(int i$1, int i$2, int n$1, int n$2, bool P$1, bool P$2)
  // Ensure threads are in range and distinct
  requires 0 <= tid$1 && tid$1 < N;
  requires 0 <= tid$2 && tid$2 < N;
  requires tid$1 != tid$2;
  // Scalar formals initially equal
  requires i$1 == i$2;
  requires n$1 == n$2;
  // Clear read/write sets
  requires !rdExistsA && !wrExistsA;
  // Predication assumes enabledness
  requires P$1 && P$2;
{
  // Dualised local variables
  int v$1, v$2;
  int w$1, w$2;

  // Fresh Predicates
  bool Q$1, Q$2;

  // Translation of "v = 0"
  P$1 ⇒ v$1 = 0;
  P$2 ⇒ v$2 = 0;

  // Translation of if-statement
  Q$1 = P$1 && (tid$1 + i$1 < n$1);
  Q$2 = P$2 && (tid$2 + i$2 < n$2);
  // Translation of "v = A[tid+i]"
  LogRdA(tid$1 + i$1, Q$1);
  ChkRdA(tid$2 + i$2, Q$2);
  Q$1 ⇒ v$1 = *;
  Q$2 ⇒ v$2 = *;

  // Translation of "w = A[tid]"
  LogRdA(tid$1, P$1);
  ChkRdA(tid$2, P$2);
  P$1 ⇒ w$1 = *;
  P$2 ⇒ w$2 = *;

  // Uncommenting this barrier fixes the race
  // Barrier(P$1, P$2);

  // Translation of "A[tid] = v + w"
  LogWrA(tid$1, P$1);
  ChkWrA(tid$2, P$2);
  // write to A[tid] is no-op
}
```

Figure 2.8.: Kernel transformation of the kernel in Figure 2.7

an integer `offset`. The array enumeration is the list of arrays in shared state (including special `NULL` and uninitialised values) and the offset captures the element of the array pointed-to by the pointer. We use Steensgaard's analysis [Ste96] to over-approximate aliasing and use this information as a case-split within the logging and checking procedures.

Unstructured Control Flow The use of *gotos* (as well as short-circuit evaluations, switch statements, break, continue and arbitrary returns) means that GPU kernels can exhibit unstructured control flow. Work by Collingbourne et al. [CDKQ13] addresses the kernel transformation for programs with *reducible* control flow graphs, which, in practice covers all unstructured GPU programs.[11] This work also enables GPUVerify to use the Clang [Cla] compiler as a frontend, which greatly aids the robustness of the tool for handling tricky syntactic features.

Benign Data Races A benign data race is a data race that does not cause observable non-determinism. Benign data races are usually intended and do not affect the correctness of the program. For example, consider a kernel that tests if there exists an element of an array that satisfies some predicate and sets a flag (in shared state and initially false) if this is the case. The test of each element can be computed in parallel and all threads where the predicate evaluates to true can update the output flag. This is an example of a benign write-write data race where all threads (that update the output) race but are guaranteed to write the same value. A benign read-write data race occurs if the writing thread is guaranteed to write the same value that the reading thread will observe.

It is not possible to precisely capture this behaviour using the adversarial abstraction since reads always receive arbitrary values. However with the equality abstraction we can tolerate benign races by additionally logging the values read or written and adding a conjunct to the check procedures to ensure that the values of the accesses are different [BCD+12].

Subgroups and Atomics Work by Bardsley and Donaldson has addressed both subgroups and atomics [BD14]. The kernel transformation must take these features into account since they both affect race checking. As discussed in Section 2.1.3, a subgroup is an implementation detail that specifies the division of a group into smaller subgroups for execution. Because threads in the same subgroup execute in lock-step they are implicitly

[11] All cycles in a reducible control flow graph (CFG) are *natural loops* with a unique header, single entry point and one or more *back edges* from nodes of the loop body to the header [ALSU06]. Collingbourne et al. note that all kernels that they have examined have reducible control flow and whether irreducible control flow is legal in OpenCL is implementation-defined [Khr13a].

synchronised and this means explicit barriers can be omitted when it is known that only a single subgroup will access memory. Using this as an optimisation depends on knowing the size of a subgroup on the target architecture for correctness. Previous versions of the CUDA programming guide encouraged this optimisation, however, we note this advice was removed in later versions (≥ 5.0) [NVI12a]. CUDA and OpenCL also support read-modify-write atomic operations such as compare-and-swap. Concurrent atomic operations to the same location are not considered to cause a data race.

Termination As discussed in Section 2.1.1 in this thesis we use correctness in the sense of partial correctness, ignoring termination. Work by Ketema and Donaldson has addressed the problem of termination analysis for GPU kernels by adapting an existing termination tool [KD14].

Performance and Error Reporting Performance and error reporting are key characteristics for usability. In prior work we gave optimisations for improving verification performance and discussed the need for meaningful error reports [BBC+14]. These aspects, which greatly affect the user experience, are important considerations for practical program verifiers since programmers will not tolerate unresponsive or opaque tools. When discussing industry uptake of the Coverity analyser [BBC+10], the authors write, "most [customers] require that checking runs complete in 12 hours, though those with larger code bases grudgingly accept 24 hours" about performance. They also observe that it is better to completely abandon analyses than to present difficult-to-understand reports.

2.4. Summary

We have reviewed dynamic, hybrid and static techniques for analysing data races in data-parallel GPU kernels and, in particular, focused on GPUVerify, which can verify that a kernel is data race-free.

The kernel transformation, which is the heart of GPUVerify, is sound, but incomplete: i.e., errors reported by the tool may be spurious (false positives). These false positives can arise from a number of sources: (i) abstract handling of floating-point operations, (ii) the abstraction of shared state, or (iii) insufficiently strong loop invariants. For example, the following example causes a spurious error due to (i) abstract handling of floating-point operations. In the code we use `tid` to denote the id of a thread and assume `A` is an array in shared state.

```
float ftid = tid + 0.0f;
```

```
A[(unsigned)ftid] = tid;
```

The spurious error is reported because the floating-point addition is handled by GPUVerify as an uninterpreted function and hence the value of `ftid` is arbitrary. In practice, based on an analysis of a benchmark suite consisting of 564 kernels collected from nine sources (further detailed in Chapter 3), we find that (iii) is by far the most common limitation, (ii) sometimes occurs and (i) was not encountered. Motivated by this observation we will turn to the problem of insufficiently strong loop invariants in Chapter 3 and the problem of the abstraction of shared state in Chapter 4.

3. Scaling Up Candidate-Based Invariant Generation

In this chapter we investigate methods to achieve scalable and automatic verification for GPU kernels. A key problem in automating GPUVerify is the generation of *invariants*. An *invariant* is a property that captures program behaviours by expressing a fact that always holds at a particular program point. Invariants are vital in static verification for modular reasoning about loops and procedure calls [Flo67]. Inside GPUVerify, invariant generation for loops is critical for precise reasoning and avoiding spurious errors. The problem of discovering invariants automatically is known as *invariant generation*. GPUVerify uses a *candidate-based* invariant generation approach where invariants are generated from a set of possible candidates, themselves generated by rules or simple analyses. An important trade-off is that of precision and performance: generating more candidates is potentially useful for precise analysis but also potentially detrimental to performance. The main results of this chapter are:

- A set of candidate generation rules that enable precise reasoning of GPU kernels.
- A new parallelisable method for accelerating candidate-based invariant generation via under-approximating analyses.
- A principled exploration of the trade-offs between automatic and scalable verification for GPU kernels with respect to lightweight safety properties. Over a benchmark suite comprising 356 GPU kernels, we find that candidate-based invariant generation allows precise reasoning for 256 (72%) kernels and our acceleration strategies yield an overall speedup of 1.25× across all kernels.

3.1. Preliminaries

To begin, we present details on the use of loop invariants in program verification (3.1.1), how candidate-based invariant generation operates (3.1.2) and the role of invariant genera-

tion in GPUVerify (3.1.3). We discuss the relationship between candidate-based invariant generation and other invariant generation techniques in Section 3.7.

3.1.1. Loop Invariants

A loop invariant captures the behaviour of an arbitrary loop iteration by expressing a property that holds each time the loop head is reached during execution. Loop invariants are established *inductively*: they must hold on loop entry, the *base case*, and must be maintained by each execution of the loop, the *step case*. The step case assumes that the invariant (the inductive hypothesis) and loop guard hold and then checks that executing the body results in a state which satisfies the invariant. Figure 3.1 gives two examples.

(a)

```
// i, j are integers
i = 0; j = 0;
while (i < 100)
  invariant j == 2*i;
  invariant j <= 200;
  invariant 0 <= j;
{
  i = i + 1;
  j = j + 2;


}
assert j == 200;
```

(b)

```
// i is an integer; v, n, m are 32-bit bit-vectors
n = v; m = 0; i = 0;
while (n != 0)
  invariant n == v >> i;
  invariant m == v & ((1 << i) - 1);
{
  if ((n & 1) == 1) {
    m |= 1 << i;
  }
  i = i + 1;
  n = n >> 1;
}
assert m == v;
```

Figure 3.1.: Two loops requiring loop invariants

In example (a) the loop repeatedly increments two variables, i and j. The invariants $j = 2i$ and $j \leq 200$ suffice to prove the assertion at the end of the program because assuming only these invariants and the negation of the loop guard we can prove that the assertion condition holds, i.e., $j = 2i \ \wedge \ j \leq 200 \ \wedge \ 100 \leq i \Rightarrow j = 200$. The first invariant $j = 2i$ is *inductive* in isolation since it holds on entry to the loop when $i = j = 0$ (base case) and is maintained by the loop (step case): assuming the invariant and loop guard then executing the body preserves the invariant, i.e., $j = 2i \ \wedge \ i < 100 \Rightarrow (j + 2) = 2(i + 1)$. The second invariant is not inductive in isolation since $j \leq 200 \ \wedge \ i < 100 \not\Rightarrow (j + 2) \leq 200$ (e.g., if $i = 0$ and $j = 200$); it is only inductive in conjunction with the first invariant. The third invariant $0 \leq j$ is inductive by itself[1] but is not necessary for proving the assertion, nor strong enough to prove the assertion in isolation: $0 \leq j \ \wedge \ 100 \leq i \not\Rightarrow j = 200$ (e.g., if $i = 100$ and $j = 0$).

[1]The invariant $0 \leq j$ is only inductive by itself if we assume mathematical integers; this would not be the case with machine integers because overflow allows $0 \leq j \ \wedge \ i < 100 \not\Rightarrow 0 \leq (j + 2)$ (e.g., if j is the maximum positive integer value).

In example (b) the loop is an obfuscated assignment of v into m using bitwise operations; the value of v is copied bit-by-bit. Assume that v, n and m are 32-bit bit-vectors that can be interpreted as unsigned integers. Initially, we assign v into n and then proceed to a loop that will destruct n. The loop continues while the bitwise value of n has some bits that are set. At each iteration, (i) if the least-significant bit of n is set then we set the corresponding bit i of m; and (ii) we shift n right and increment i. The two invariants capture the essential relationship between v, m and i and suffice to prove the assertion. The first invariant is inductive in isolation; the second invariant is only inductive in conjunction with the first invariant.

More formally, the *loop-cutting transformation*, originating from the foundational work of Floyd [Flo67] and described, for example, by Barnett and Leino [BL05], replaces a loop with a loop-free sequence of statements that over-approximates the behaviour of the loop using its invariant, making the checks of the base and step cases for the invariant explicit (Figure 3.2). The correctness of the transformed loop implies the correctness of the original loop. The transformation yields an *over-approximation* of the loop because of the statement `havoc modset(B)`. The *modset* (modifies set) of `B` is every variable that *may* be assigned in the loop body and `havoc`-ing sets each of these variables to a non-deterministic value. This conservatively over-approximates the loop because it captures *at least* those states that can be reached through execution of the loop. Informally, we can think of the havoc of the modset followed by the assumption of the invariant as 'teleporting' to an arbitrary loop iteration satisfying the invariant. After applying this transformation we have a new program with one fewer loop that over-approximates the original program. We can proceed by repeatedly applying the loop-cutting transformation until we have a loop-free program whose correctness implies the correctness of the original program.

```
while (c) invariant φ {
  B;
}
```

```
assert φ;   // (base case)
havoc modset(B);
assume φ;   // inductive hypothesis
if (c) {
  B;
  assert φ; // (step case)
  assume false;
}
```

Figure 3.2.: The loop-cutting transformation summarises a loop using its invariant [BL05]

```
i = 0;
x = 1; y = 2; z = 3;
while (i < 10000)
  candidate C0: i = 0;
  candidate C1: i != 0;
  candidate C2: 0 <= i;
  candidate C3: 0 < i;
  candidate C4: i < 10000;
  candidate C5: i <= 10000;
  candidate C6: x != y;
{
    temp = x; x = y; y = z; z = temp;
    i = i + 1;
}
```

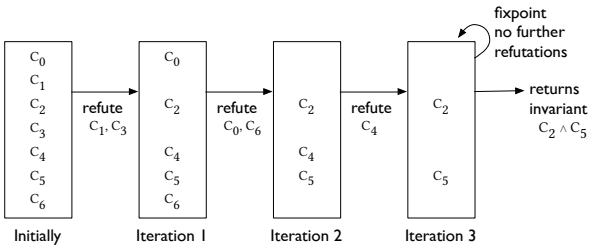


Figure 3.3.: An example program and Houdini run showing the candidates refuted at each iteration until a fixpoint is reached. The example program is adapted from a paper by Donaldson et al. [DHKR11].

3.1.2. Candidate-Based Invariant Generation using Houdini

Candidate-based invariant generation has two phases: *guess* and *check*. The *guess* phase generates candidates for a given program. This yields a finite set of candidates, which can be considered to be the raw material for the *check* phase. The *check* phase uses the Houdini algorithm [FL01] to compute the unique, maximal conjunction of candidates that is an inductive invariant (for a proof of this property see [FJL01]). In the best case this is the entire set of candidates; in the worst case it is the empty set, corresponding to the invariant `true`.

Guess Phase This phase generates a finite set of candidates for a given input program. These can be generated from any source including static or dynamic methods. Importantly, this phase can be aggressive, generating a large set of *potential* invariants: candidates that turn out to be false cannot introduce unsoundness; they will simply be refuted by Houdini. In Section 3.3, we give the methodology and design of rules we have implemented for GPUVerify.

Check Phase This phase computes the maximal subset of candidates that collectively form an inductive invariant, by successively removing candidates that cannot be proven until a fixpoint is reached. A candidate may fail to be proven either because it is false or because it is not inductive (corresponding to base or step case assertion failures in Figure 3.2). For brevity, in both cases we refer to the candidate as *false*.

In this section we use the program in Figure 3.3, adapted from a paper by Donaldson

et al. [DHKR11], as a simple running example. The program repeatedly cycles the values $1, 2, 3$ around the variables x, y, z. We assume the guess phase has given us the candidates C_0 through C_6. Houdini must now compute the maximal inductive invariant from this set of candidates.

Initially, Houdini tries to verify the program using the conjunction of all candidates as an invariant. This is checked by generating the verification condition for the program [BL05] and checking for its satisfiability using an underlying SMT solver (e.g. Z3 [dMB08] or CVC4 [BCD$^+$11]). If verification succeeds then the entire candidate set forms an inductive invariant. Otherwise, the failed verification attempt identifies a subset of the candidates that cannot be proven. These candidates are removed and verification using the remaining candidates is attempted; again, if verification succeeds then the candidate set is inductive. This process continues until a fixpoint is reached (in the worst case, the fixpoint is the empty set).

In Figure 3.3 we show the set of candidates for each iteration of Houdini. Houdini is unable to verify the program using the conjunction of all candidates because C_1 and C_3 are false: they are not true at loop entry (base case). No further candidates can be removed at this iteration since (i) all other candidates are true at loop entry and (ii) assuming the candidates C_0 and C_1 is inconsistent so all candidates are vacuously true when checking the step case. At the second iteration, the candidates C_0 and C_6 are refuted because they are not preserved by the loop. To see why C_6 is not preserved consider the state where $x = 1$ and $y = z = 2$, which satisfies the assumption on loop entry but results in a state that does not satisfy C_6 after the loop executes. At the third iteration, the candidate C_4 is refuted. This candidate could not be removed until C_0 was removed since assuming C_0 allows C_4 to be preserved by the loop. This illustrates dependencies between candidates, where refutation of a candidate is only possible after refutation of certain other candidates. In the final iteration a fixpoint is reached: all remaining candidates collectively form an inductive invariant and Houdini returns $C_2 \wedge C_5$.

The Houdini algorithm is sound and terminating. This is because each iteration of Houdini only removes false candidates (soundness) and so the number of remaining candidates is strictly monotonically decreasing with each iteration, until the fixpoint is reached (termination). We also argue that Houdini is in some sense predictable. Because Houdini only considers conjunctions of candidates the maximum number of iterations is linear in the number of candidates when considering a single procedure (catering for multiple procedures requires a quadratic number of iterations in the worse case). This gives us an upper bound on the number of SMT queries because the search space (the set of candidates) is known upfront. Although each iteration results in an SMT query and (depending on

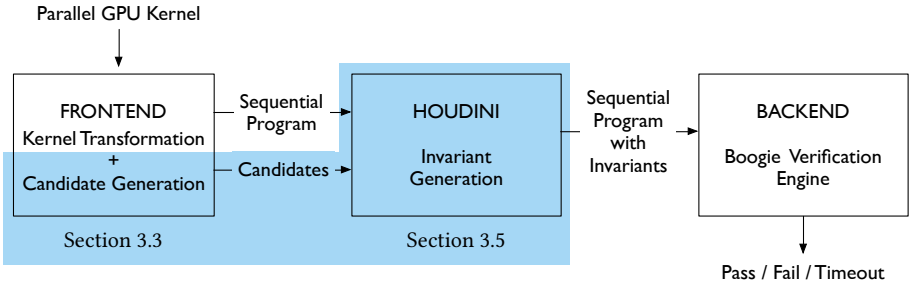


Figure 3.4.: The architecture of GPUVerify. The components responsible for invariant generation are shaded.

the theories involved and the details of the query in question) the speed of an SMT query can be unpredictable, this bound on the number of queries offers a significant degree of predictability in practice.

3.1.3. Invariant Generation in GPUVerify

Figure 3.4 shows the architecture of GPUVerify. As discussed in Chapter 2, the key idea behind GPUVerify is the translation of a parallel GPU kernel into a sequential program. The kernel transformation (Section 2.3) applied by GPUVerify is sound, but incomplete: i.e., errors reported by the tool may be spurious (false positives). These false positives can arise from three sources: (i) abstract handling of floating-point operations, (ii) the abstraction of shared state, or (iii) insufficiently strong loop invariants. In practice we find that (iii) is by far the most common limitation. To combat this, GPUVerify uses candidate-based invariant generation using Houdini. The frontend is responsible for kernel translation and candidate generation using pre-defined rules designed for the domain of GPU kernels. We discuss these rules in detail in Section 3.3. Houdini then takes the sequential program with candidates and computes the strongest invariant, which is subsequently passed onto the backend solver. We discuss methods for accelerating Houdini in Section 3.5.

3.2. Benchmarks

An assumption of candidate-based invariant generation is that the programs to be analysed share similar characteristics. That is, the candidate generation rules should be generally applicable. To develop our rules we conducted our study with respect to a set of 356 benchmarks, collected from nine sources:

- *AMD Accelerated Parallel Processing SDK* v2.6 [AMD] (53 OpenCL kernels)

- *NVIDIA GPU Computing SDK* v5.0 [NVI] (97 CUDA kernels); we also include 7 CUDA kernels from a previous version of the SDK (v2.0)
- Microsoft *C++ AMP Sample Projects* [Mic] (16 kernels, hand translated to CUDA)
- The *gpgpu-sim* benchmarks [BYF+09] (20 CUDA kernels)
- The *Parboil* v2.5 [SRS+12] (18 OpenCL kernels)
- The *Rodinia* suite v2.4 [CSLS09] (24 OpenCL kernels)
- The *SHOC* suite [DMM+10] (48 OpenCL kernels)
- The *PolyBench/GPU* suite [GGXS+12] (49 compiler-generated OpenCL kernels)
- Rightware *Basemark CL* v1.1 [Rig] (24 OpenCL kernels)

Each suite is publicly available except for *Basemark CL* which was provided to us under an academic license. This collection covers all publicly available GPU benchmark suites that we are aware of. The kernel counts above do not include 208 kernels that we manually removed from our study: (i) 3 kernels cause GPUVerify to fault internally, (ii) 16 kernels are trivially race-free as they are run by a single thread, (iii) 8 kernels use features that are currently unsupported by GPUVerify, such as CUDA *surfaces*, (iv) 24 kernels are *data-dependent* and require refinements of the GPUVerify verification method that cannot be applied automatically [CDK+13] (we address this problem in Chapter 4), and finally (v) 157 kernels are loop-free and hence do not require invariant generation. We refer to the whole benchmark suite as the LOOP set.

In Section 1.2, we distinguished between real and synthetic programs. An assumption of the work in this chapter is that the kernels considered in our benchmark suite are real. We have guarded against this by gathering benchmarks from a variety of sources. However, a minority of the kernels, particularly in the SDKs, are somewhat synthetic. For example, they demonstrate a language feature (e.g., kernels using function pointers in the CUDA SDK) or repeatedly execute the same loop body for the purpose of performance timing (e.g., the matrix transpose kernels in the CUDA SDK).

Tables 3.1 and 3.2 indicate the extent to which GPUVerify is confronted with loops when analysing the LOOP set. GPUVerify performs full inlining before analysis (this is possible in practice because recursion and function pointers are prohibited in OpenCL [Khr13b] and rare in CUDA).[2] In Table 3.2 we give the number of loops and depth of the deepest

[2]No kernels in our benchmarks use recursion although it is supported in CUDA 5.0 [NVI12a]. Two kernels, `FunctionPointers/Sobel`, in the CUDA SDK use function pointers.

Table 3.1.: Number of loops of the kernels in our study

Number of loops	**1**	**2**	**3**	**4**	**5**	**6**	**7+**
Kernels	154	100	38	32	14	3	15

Table 3.2.: Loop nest depth of the kernels in our study

Max loop nest depth	**1**	**2**	**3**	**4**
Kernels	234	75	38	9

loop nest across the benchmark set after inlining has been performed. This shows that the majority (71%) of kernels feature a single loop or a pair of loops, but that 15 kernels exhibit a large number of loops (7 or more). The 1143-line `heartwall` kernel from the Rodinia suite features 48 loops which are syntactically distinct (i.e., they do not arise as a result of inlining). The majority of kernels do not exhibit nested loops (indicated by a nesting depth of 1), however, more than a third of the kernels (34%) exhibit larger nesting depths.

3.3. Candidate Generation Rules for GPU Kernels

We now turn to the candidate generation rules that we have designed for reasoning about GPU kernels, based on the benchmarks of Section 3.2. We begin by reviewing the translation applied by GPUVerify, which introduces instrumentation variables that must be accounted for by loop invariants: race instrumentation (Section 3.3.1), and uniformity (Section 3.3.3). To illustrate this, we consider three patterns found in GPU kernels and the form of loop invariants required for precise reasoning (Section 3.3.2). Finally we discuss our methodology (Section 3.3.4) for designing new rules, and the rules themselves (Section 3.3.5).

3.3.1. Invariants for Handling Race Instrumentation

The translation applied by GPUVerify introduces instrumentation variables to cater for race checking. Intuitively, the instrumentation variables track the read and write sets of an arbitrary pair of distinct threads (introduced by the two-thread reduction). A race error is reported if the intersection of the read and write sets is non-empty (for *some* distinct pair of threads). Crucially, if an access occurs in a loop then the corresponding instrumentation variables will be a member of the modset of the loop (Section 2.3.3).

Hence, under the loop-cutting transformation (Figure 3.2), the instrumentation variables tracking accesses will be set arbitrarily when considering the step case and may result in spurious race errors.

In the previous chapter (Section 2.3.3) we described two quantifier-free encodings for race instrumentation: a *non-deterministic set* encoding [BCD+12] and a *watchdog* encoding [BBC+14]. We review their behaviour by considering a simple racy kernel given in Figure 3.5(a). In the code, `tid` denotes the id of a thread, `v` and `w` are thread-private variables and `A` is an array in shared state.

(a) Racy kernel

```
v = A[tid];

w = A[tid+1];

A[tid] = v + w;
```

(b) Translation

```
LogRdA(tid$1);
ChkRdA(tid$2);
havoc(v$1, v$2);

LogRdA(tid$1+1);
//
//
//
ChkRdA(tid$2+1);
havoc(w$1, w$2);

LogWrA(tid$1);
ChkWrA(tid$2);
```

(c) Non-deterministic set

```
LogRdA(tid$1);
ChkRdA(tid$2);
havoc(v$1, v$2);

if (*) {
  rdExistsA = true;
  rdOffsetA = tid$1+1;
}
ChkRdA(tid$2+1);
havoc(w$1, w$2);

LogWrA(tid$1);
assert (rdExistsA
  ⇒ rdOffsetA != tid$2);
assert (wrExistsA
  ⇒ wrOffsetA != tid$2);
```

(d) Watchdog

```
LogRdA(tid$1);
ChkRdA(tid$2);
havoc(v$1, v$2);

if (tid$1+1 == trOffset) {
  rdExistsA = true;

}
ChkRdA(tid$2+1);
havoc(w$1, w$2);

LogWrA(tid$1);
assert (rdExistsA
  ⇒ trOffset != tid$2);
assert (wrExistsA
  ⇒ trOffset != tid$2);
```

Figure 3.5.: A racy kernel and its translation using race instrumentation

The kernel in (a) is racy because the read from `A[tid+1]` of one thread can conflict with the write to `A[tid]` of another (e.g., thread 0 and thread 1). We highlight the racing accesses.

The code in (b) shows the kernel after translation. We omit predication since every statement is enabled. The translation has applied:

- Two-thread reduction: introduces a pair of arbitrary threads (s, t). Thread-private variables (including `tid`) are duplicated into `$1` and `$2` versions corresponding to the version of the variable for s and t, respectively.
- Adversarial shared state abstraction: shared-state is not modeled — i.e., reads from `A` result in a non-deterministic assignment using `havoc` and writes to `A` become no-ops.
- Access logging: each read or write access to `A` is translated into a pair of log and check procedure calls. The threads are treated asymmetrically: the log procedure

(respectively check procedure) is called by the first thread s (respectively the second thread t) of the two-thread reduction. This is sound because the two-thread reduction will consider all pairs of threads: i.e., if the pair (s, t) is considered then so will (t, s).

The code in (b) highlights the conflicting log and check procedures. Suppose we used a *quantified* encoding of the read and write sets. Then we would capture the read and write sets of s as $\{s, s+1\}$ and $\{s\}$, respectively; and, the read and write sets of t as $\{t, t+1\}$ and $\{t\}$, respectively. Since $s + 1 = t$ is satisfiable (if $s = t - 1$) the read and write sets conflict and a race error is reported.

In the code of (c) and (d) we inline these procedures using the quantifier-free non-deterministic set and watchdog encodings, respectively.[3] For clarity we have only inlined the conflicting log and check procedures and left other calls unmodified. Under the non-deterministic set encoding, the log procedure non-deterministically chooses whether to track the read of s (by setting `rdExistsA` and updating `rdOffsetA`) or not (by leaving the instrumentation variables unmodified). Under the watchdog encoding, the log procedure tracks the read if it matches an arbitrary offset (the tracked offset, `trOffset`) by setting `rdExistsA`. The check procedure for both encodings asserts that the write access of t does not conflict with the access tracked in the instrumentation variables. We highlight the assertion failure in both examples.

Now consider a loop that contains an access to `A`. In this case, the corresponding instrumentation variables of `A` will be in the modset of the instrumented loop and may cause spurious race errors. For example, suppose the statements in Figure 3.5(a) were in a loop. The instrumentation variables modified in Figure 3.5(c) or (d) would then be in the modset of the instrumented loop. Thus, for precise reasoning, loop invariants must constrain these instrumentation variables by capturing the *access pattern* of the loop, which summarises the assignment of threads to elements of the array and constrains the instrumentation variables.

3.3.2. Example Invariants for Access Patterns

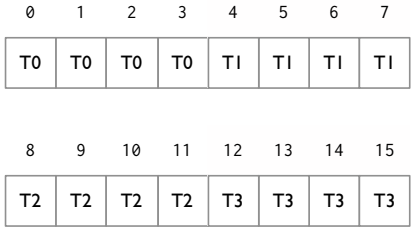
To illustrate this, we consider three access patterns found in GPU kernels and the form of loop invariants that suffice for precise reasoning. In this section, we will use the variables

[3] Our translation for the watchdog encoding (d) omits the use of the `tracking` variable, which is an optimisation. We use this variable to non-deterministically enable race checking at a barrier (Section 2.3.3). Doing this allows the instrumentation variables to be set to false at a barrier by *assuming* they are false (rather than assigning to them) and hence avoids unnecessarily adding these variables to the modset of loops containing barriers.

```cuda
//--gridDim=[1] --blockDim=[4]
#define N 4
__global__ void saxpy(float a, float *x, float *y) {
  unsigned tid = threadIdx.x;

  for (int i=0; i<N; i++) {
    unsigned idx = (tid * N) + i;
    y[idx] = a * x[idx] + y[idx];
  }
}
```

(a) CUDA source code



(b) Access pattern: each thread handles a 'slice' of N elements

Figure 3.6.: Saxpy kernel using slicing

r_A and w_A to refer to an *arbitrary* read and write offset of the array `A`, which correspond to:

- `rdOffsetA` and `wrOffsetA` (when `rdExistsA` = `true` and `wrExistsA` = `true`, respectively) under the non-deterministic set encoding; and
- `trOffset` (when `rdExistsA` = `true` and `wrExistsA` = `true`) under the watchdog encoding.

Slicing Figure 3.6a gives a CUDA kernel that performs a parallel vector operation $\vec{y} = \alpha\vec{x} + \vec{y}$. For exposition purposes, we give a simplified kernel that is invoked by a single block. Each thread in the block is assigned N elements of the vector computation. We assume that the vectors $\vec{x}$ and $\vec{y}$ are of length $N \cdot \texttt{blockDim.x}$. Figure 3.6b gives the access pattern when $\texttt{blockDim.x} = 4$. The kernel is data race-free since the array `x` is read-only and each thread reads and writes to a contiguous 'slice' of the array `y`. We can capture this pattern using the following invariants:

$$\begin{aligned} r_y \;/\; N &= \texttt{tid} \\ w_y \;/\; N &= \texttt{tid} \end{aligned}$$

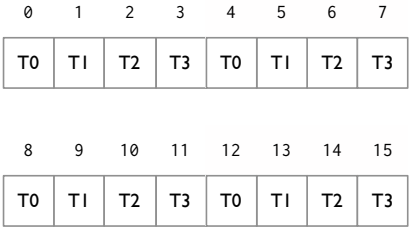
which use integer division to express the assignment of elements to threads. An alternative is to give lower and upper bounds to the accesses:

$$\begin{aligned} \texttt{tid} \cdot N \leq\; & r_y < (\texttt{tid} + 1) \cdot N \\ \texttt{tid} \cdot N \leq\; & w_y < (\texttt{tid} + 1) \cdot N \end{aligned}$$

```
//--gridDim=[1] --blockDim=[4]
#define N 4
__global__ void saxpy(float a, float *x, float *y) {
  unsigned tid = threadIdx.x;

  for (int i=0; i<N; i++) {
    unsigned idx = (i * N) + tid;
    y[idx] = a * x[idx] + y[idx];
  }
}
```

(a) CUDA source code



(b) Access pattern: each thread handles strided elements at intervals of N

Figure 3.7.: Saxpy kernel using striding

Striding Figure 3.7a gives a CUDA kernel that performs the same parallel vector operation $\vec{y} = \alpha\vec{x} + \vec{y}$ as in the previous section, but using a different access pattern. Figure 3.7b (for `blockDim.x` $= 4$) shows that the assignment of threads to elements is strided. Each thread t handles elements at intervals of N. For example, thread 0 handles elements 0, 4, 8 and 12. We can capture this pattern using the following invariants:

$$\begin{aligned} r_y \,\%\, N &= \texttt{tid} \\ w_y \,\%\, N &= \texttt{tid} \end{aligned}$$

which use modulo arithmetic to express the striding assignment of elements to threads.

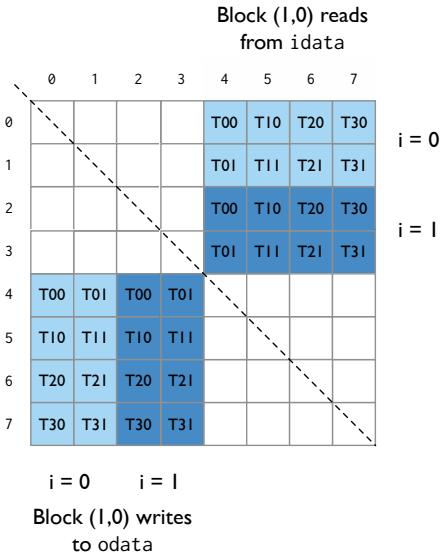
Tiling We now turn to a more realistic example that exhibits both slicing and striding patterns. Figure 3.8a gives a CUDA kernel taken from the CUDA SDK [NVI] that performs a parallel matrix transpose. The kernel reads from an input matrix `idata` of dimension `width` by `height` and writes to an output matrix `odata`. The matrices are stored in row-major order so an element A_{ij} of a matrix A is stored in a linear array at offset $(\texttt{i} + \texttt{width} \cdot \texttt{j})$. The kernel is invoked with a two-dimensional grid of two-dimensional blocks. Each block of threads is assigned a square tile of dimension `TILE_DIM` of the input and output matrices. Individual threads within a block stride along their assigned tile in the loop by `BLOCK_ROWS` steps. At each iteration of the loop, each block of threads copies `TILE_DIM` by `BLOCK_ROWS` elements from `idata` to `odata`. For example, if the matrix dimension is 8×8, with `TILE_DIM` 4 and `BLOCK_ROWS` 2 then the kernel is invoked with a 2×2 grid of blocks of 4×2 threads. Each of the blocks is assigned a tile of dimension 4×4 elements. The read and write assignment of block $(1, 0)$ is shown in Figure 3.8b. For example, thread $(1, 1)$ of block $(1, 0)$ assigns $\texttt{odata}_{5,1}$ from $\texttt{idata}_{1,5}$ and $\texttt{odata}_{3,1}$ from $\texttt{idata}_{1,3}$ when $i = 0$ and 1, respectively.

```
//--gridDim=[2,2] --blockDim=[4,2]
#define TILE_DIM 4
#define BLOCK_ROWS 2
__global__ void transpose(
  float *odata, float *idata, int width, int height) {
  int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;

  int index_in = xIndex + width * yIndex;
  int index_out = yIndex + height * xIndex;

  for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
    odata[index_out+i] = idata[index_in+i*width];
  }
}
```

(a) CUDA source code



(b) Reads of `idata` and writes of `odata` of block $(1, 0)$ for iterations `i` $= 0$ and 1. N.B., the reads and writes of the kernel are not in-place over the same matrix.

Figure 3.8.: Matrix transpose kernel (from the CUDA SDK)

The kernel is data race-free since the input `idata` is read-only and distinct threads write to distinct offsets of the output `odata`. We can capture this pattern using an invariant that expresses an arbitrary write $w_{\texttt{odata}}$ in terms of thread and block identifiers. Intuitively, this is useful because for any two distinct threads it must be the case that (at least) one of `threadIdx.x`, `threadIdx.y`, `blockIdx.x` and `blockIdx.y` are distinct. In the following, we write H and D to denote `height` and `TILE_DIM`, respectively; and, write `tid`.$\{x, y\}$ and `bid`.$\{x, y\}$ to refer to the two-dimensional components of the thread and block identifier, respectively.

$$\begin{aligned}
((w_{\texttt{odata}} \ / \ H) \ \% \ D) &= \texttt{tid}.x \ \wedge \\
((w_{\texttt{odata}} \ \% \ H) \ \% \ D) &= \texttt{tid}.y \ \wedge \\
((w_{\texttt{odata}} \ / \ H) \ / \ D) &= \texttt{bid}.x \ \wedge \\
((w_{\texttt{odata}} \ \% \ H) \ / \ D) &= \texttt{bid}.y
\end{aligned}$$

This invariant is trivially satisfied on loop entry since no writes have occurred at this point. To see that it is maintained by the loop consider the form of the write $w_{\texttt{odata}}$. The access will be of the form `index_out` $+$ `i`, where `i` is some multiple of `BLOCK_ROWS`.

Rewriting, by expanding variable definitions, gives us:

$$w_{\texttt{odata}} = (\texttt{tid}.x \cdot H) + \texttt{tid}.y + (\texttt{bid}.x \cdot H \cdot D) + (\texttt{bid}.y \cdot D) + \texttt{i}$$

which shows that the use of division and modulo (as used in slicing and striding, respectively) allows the invariant to relate $w_{\texttt{odata}}$ to the components of the thread and block identifiers. For example, in the first conjunct, $w_{\texttt{odata}}$ is divided through by H to leave $\texttt{tid}.x + (\texttt{bid}.x \cdot D)$ so that subsequent modding by D yields $\texttt{tid}.x$.

3.3.3. Invariants for Handling Uniformity

As discussed in Section 2.3.1, the kernel translation performed by GPUVerify uses *predication* to deal with thread divergent control-flow [BCD+12]. After predication, each statement has a thread-private variable (a predicate) that determines whether a thread is enabled or disabled. If all threads are enabled or disabled at a given statement then we say the control-flow is *uniform* (at this execution point). Barrier divergence occurs if a barrier is reached under non-uniform control flow. Therefore, for precise reasoning about barrier divergence we must reason about uniformity in loop invariants. Figure 3.9 (deliberately contrived for the purpose of illustration) gives an example that is barrier divergence-free.

(a) Original kernel

```
// n is a formal parameter
// i, j are thread-private variables
i = 0;
if (tid < n) {
  j = 1;
} else {
  j = 2;
}
while (i < n) {
  barrier();
  j = j + 3;
  i = i + 4;
}
```

(b) After predication

```
true ⇒ i$1 = 0;
true ⇒ i$2 = 0;
true ⇒ p$1 = tid$1 < n;
true ⇒ p$2 = tid$2 < n;
p$1  ⇒ j$1 = 1;
p$2  ⇒ j$2 = 1;
!p$1 ⇒ j$1 = 2;
!p$2 ⇒ j$2 = 2;
true ⇒ q$1 = i$1 < n;
true ⇒ q$2 = i$2 < n;
while (q$1 || q$2) {
  barrier(q$1, q$2);
  q$1 ⇒ j$1 = j$1 + 3;
  q$2 ⇒ j$2 = j$2 + 3;
  q$1 ⇒ i$1 = i$1 + 4;
  q$2 ⇒ i$2 = i$2 + 4;
  q$1 ⇒ q$1 = i$1 < n;
  q$2 ⇒ q$2 = i$2 < n;
}
```

(c) With uniformity analysis

```
i = 0;

p$1 = tid$1 < n;
p$2 = tid$2 < n;
p$1  ⇒ j$1 = 1;
p$2  ⇒ j$2 = 1;
!p$1 ⇒ j$1 = 2;
!p$2 ⇒ j$2 = 2;


while (i < n) {
  barrier(true, true);
  j$1 = j$1 + 3;
  j$2 = j$2 + 3;
  i = i + 4;



}
```

Figure 3.9.: Predication and uniformity analysis example

The code in (a) shows a simple kernel with a thread-sensitive conditional and loop.

Applying predication with the two-thread reduction yields the code in (b) where `p` and `q` are predicates giving the enabledness condition for the condition and loop, respectively. Let us use the variable `en` to refer to the predicate at a given execution point. Then execution is uniform at a given statement if `en$1` = `en$2`. Given an expression e, we write $\mathsf{uniform}(e)$ to denote the condition $e\$1 = e\$2$. For example, in (b), the execution of the loop is uniform. This is important because the barrier requires uniform control-flow (i.e., the barrier will assert `q$1` = `q$2`). Thread-private variables can also be uniform if they always take the same values, for all threads. For example, the loop index variable i is uniform, but the variable j is not. Hence, two loop invariants are required for precise reasoning for this example: $\mathsf{uniform}(\texttt{en})$ and $\mathsf{uniform}(i)$.

In prior work [BBC+14], we introduced *uniformity analysis* to recover uniformity information statically. This is a taint analysis at the level of the control-flow graph using the program dependence graph [FOW87]. The analysis computes the transitive closure of basic blocks or variables that are control- or data-dependent on thread-sensitive identifiers. Initially, every basic block and thread-private variable is marked as uniform, except for thread identifiers which are non-uniform. Then, (i) a basic block is marked as non-uniform if it is control-dependent on a condition containing a non-uniform variable and (ii) a variable is marked as non-uniform if it is assigned an expression that contains a non-uniform variable. The analysis iterates until it reaches a fixpoint. Following this analysis, predication is only required for non-uniform blocks and only non-uniform thread-private variables need be duplicated under the two-thread reduction. For example, uniformity will recover the uniformity of `q` and i in (b) yielding the code in (c). Uniformity analysis reduces the burden on invariant generation and leads to smaller SMT formulas due to the reduction in duplicated variables.

However, because uniformity analysis is conservative there are still kernels that require invariants to re-capture uniformity. Consider the following kernel, which is barrier divergence-free.

```
// C is an integer constant
for (i=threadIdx.x; i<C*blockDim.x; i+=blockDim.x) {
  __syncthreads();
}
```

The loop index variable i is non-uniform, but the loop predicate is uniform because every thread will execute the loop C times. Hence, we require the invariant $\mathsf{uniform}(\texttt{en})$.

3.3.4. Methodology

Obtaining kernel parameters A GPU kernel is invoked with respect to a thread configuration that specifies how threads are arranged in a multi-dimensional grid of blocks. The invocation also gives the set of input values: the initial values of the kernel's formal parameters. Most kernels are not designed to be correct for arbitrary thread configurations and input values; they make implicit assumptions about these preconditions. For example, the matrix transpose kernel of Figure 3.8a requires, amongst several other preconditions, that `width` = `gridDim.x` · `TILE_DIM`. Unfortunately these preconditions are rarely fully documented, and any documentation that does exist is provided as informal source code comments.[4]

We used the following process to pragmatically choose suitable thread configurations and input value preconditions. For each kernel:

- We ran the application in which the kernel was embedded using dynamic library interception of CUDA and OpenCL calls. For this purpose we developed a shim library for CUDA applications[5] and used KernelInterceptor [BDW14] for OpenCL applications. Intercepting these calls gave us observations of invocation parameters for the kernel.

- We then ran GPUVerify on the kernel in bug-finding mode assuming the observed thread configuration but with unconstrained input parameters. The bug-finding mode of GPUVerify involves loop unrolling for fixed depth 2 to avoid the need to generate loop invariants and eliminate the possibility of spurious errors due to insufficiently strong loop invariants (spurious errors due to the abstract handling of floating-point operations and the abstraction of shared state are still possible). If GPUVerify reported a possible data race then we determined whether the cause was due to an unconstrained integer input parameter. In this case, we added the constraint observed at runtime and repeated until either GPUVerify was satisfied or we had constrained all integer parameters. We only add integer preconditions because GPUVerify handles other data types, such as floating-point, and shared state abstractly. It is rare for the data race-freedom of a kernel to depend on the values of non-integer preconditions.

This provided a pragmatic mechanism for obtaining a suitable, but not overly con-

[4] CUDA supports kernel-level assertions, but we have rarely seen their use in the kernels we have examined (One kernel, `simpleAssert`, in the CUDA SDK uses a kernel-level assertion, but only for the purposes of demonstrating an assertion failure).

[5] `https://github.com/nchong/cudahook`

strained, precondition with respect to which the kernel could be verified. For example, in the matrix transpose kernel, this led to preconditions for `width` and `height` that in conjunction with the thread configuration satisfy the implicit constraint above. This method does not discover most general implicit assumptions, but pragmatically works around them via specific values that are used in practice.[6]

Developing rules After determining preconditions for the LOOP set we applied GPUVerify and partitioned the kernels into two sets according to whether they could or could not be automatically verified with the current loop invariant generation capabilities of the tool. Beginning with a set of five rules, defined in prior work [BCD+12], we iteratively developed further rules using the following process:

- We picked a number of kernels from the *not verified* set.
- We manually examined the kernels and manually determined a minimal set of invariants that enabled them to be verified. Each kernel was updated to include these annotations as *user-supplied* invariants.
- We then identified patterns in these invariants and reasoned whether they could be extrapolated. In particular, we were guided by whether we believed the pattern would be generally applicable (we evaluate the precision of our rules in Section 3.6.2). If so, then we implemented a rule to detect the pattern and removed any user-supplied invariants that were subsumed by the rule.
- Finally, we re-ran GPUVerify on the entire *not verified* set, moving new kernels that verified into the *verified* set.

3.3.5. Rules

We now describe the rules that we have developed. Our rules fall into the following categories: (i) patterns over accesses, which summarise read and write accesses issued by the execution of a loop, (ii) judgements about whether reads or writes can be issued by the execution of the loop, (iii) patterns over loop guard variables that summarise ranges and values that the loop guard variable can take, (iv) guesses over the uniformity of

[6] After the completion of our study we reviewed the number of preconditions inserted using this methodology. We found that 51% of kernels required preconditions and on average each kernel required 2 (standard deviation of 1.2) preconditions. The largest number of preconditions required for a single kernel was 8 (the `FunctionPointers/SobelShared` kernel in the CUDA SDK).

predicates and variables, and (v) patterns over variables, in particular, recognising power-of-two values. For each rule we give the essence of the rule and a motivating CUDA kernel fragment. In the following, let `A` be an array in shared memory, i be a loop index variable, C be an integer constant value and e be an expression. We write w to refer to an arbitrary write access to `A`.

r0. accessBreak Given an access involving thread or block components (i.e., `threadIdx` or `blockIdx` in CUDA, and `get_local_id()` or `get_group_id()` in OpenCL), such as the tiling example in Section 3.3.2, this rule attempts to extract different components of the thread and block identifiers using rewriting. Intuitively, this is useful because for any two distinct threads it must be the case that (at least) one of the thread or block components is distinct. The rule begins by fully expanding the access so that every thread or block component is a separate expression. For example, the matrix transpose access from the tiling example, after expanding variable definitions and distributing multiplication is:

$$w = (\texttt{tid}.x \cdot H) + \texttt{tid}.y + (\texttt{bid}.x \cdot H \cdot D) + (\texttt{bid}.y \cdot D) + \texttt{i}$$

where we use H and D to denote `height` and `TILE_DIM`, respectively; and `i` is the variant of the loop and will be ignored by the rule since it does not contain a thread or block component. Then, each component is extracted using subtraction and division. This results in four candidates:

$$\begin{aligned}
\texttt{tid}.x &= (w/H) - (\texttt{tid}.y/H) - (\texttt{bid}.x \cdot D) - (\texttt{bid}.y \cdot D/H) \\
\texttt{tid}.y &= w - \texttt{tid}.x - (\texttt{bid}.x \cdot H \cdot D) - (\texttt{bid}.y \cdot D) \\
\texttt{bid}.x &= (w/H/D) - (\texttt{tid}.x/D) - (\texttt{tid}.y/H/D) - (\texttt{bid}.y/H) \\
\texttt{bid}.y &= (w/D) - (\texttt{tid}.x \cdot H/D) - (\texttt{tid}.y/D) - (\texttt{bid}.x \cdot H)
\end{aligned}$$

r1. accessSlice This rule identifies when a thread is assigned a contiguous range of an array, such as the slicing example in Section 3.3.2, due to an access of the form $(C \cdot \texttt{tid}) + \texttt{i}$, where `i` is the variant of the loop. The rule generates two candidates that bound the range of the access. In the following loop, the rule will generate the candidates $C \cdot \texttt{tid} \leq w$ and $w < C \cdot (\texttt{tid} + 1)$.

```
for (i=0; i<C; i++) {
  A[C*threadIdx.x+i] = e;
}
```

r2. accessStride This rule identifies stride patterns and strength reduction loops, such as the stride example in Section 3.3.2. In the following loop, the rule will generate the candidate $w \,\%\, C = \texttt{tid}$.

```
for (i=0; i<4; i++) {
  A[i*C+threadIdx.x] = e;
}
```

r3. accessSliceBlockLower and r4. accessSliceBlockUpper These rules are analogous to the accessSlice rule except they generate candidates when the access expression uses block indexes. In the following loop, the rules will generate the candidates $C \cdot \texttt{bid} \leq w$ and $w < C \cdot (\texttt{bid} + 1)$, respectively.

```
for (i=threadIdx.x; i<C; i+=blockDim.x) {
  A[(blockIdx.x*C)+i] = e;
}
```

r5. noAccessConditionalLoop This rule captures loops that are within a thread-dependent conditional, such as $\texttt{threadIdx.x} < C$. In this case, threads for which the conditional is false will not execute the loop and will not issue accesses. In the following loop, where the expression e contains a thread identifier, the rule will generate the candidates $\neg e \Rightarrow \neg\texttt{rdExistsA}$ and $\neg e \Rightarrow \neg\texttt{wrExistsA}$.

```
// e contains a thread identifier
if (e) {
  for (...) {
    ...
  }
}
```

r6. guardMinusInitialIsUniform This rule guesses that if a loop contains a barrier and the guard variable is non-uniform then the guard minus its initial value is uniform. This is based on the intuition that barrier divergence will occur otherwise. In the following loop, the rule will generate the candidate $\mathsf{uniform}(i - e)$.

```
for (i=e; ...) {
  ...
  __syncthreads();
}
```

r7. guardNonNeg This rule guesses that every guard variable is non-negative. In the following loop, the rule will generate the candidate $0 \leq i$.

```
for (i=...) {
  ...
}
```

r8. guardBound This rule guesses that the initial value of a guard variable bounds the range of the guard. In the following loop, the rule will generate candidates $e \leq i$ and $i \leq e$.

```
for (i=e; ...) {
  ...
}
```

r9. guardStride This rule is analogous to the accessStride rule except for guard variables. In the following loop, the rule will generate the candidate $i \,\%\, C = \texttt{tid}$.

```
for (i=0; i<4*C; i+=C) {
  ...
}
```

r10. loopPredicateUniform This rule guesses that if a loop contains a barrier and the guard variable is non-uniform then the loop predicate is uniformly enabled. This is based on the intuition that barrier divergence will occur otherwise. In the following loop, the rule will generate the candidate uniform(en).

```
for (i=threadIdx.x; i<C*blockDim.x; i+=blockDim.x) {
  ...
  __syncthreads();
}
```

r11. noRead and r12. noWrite These rules guess that if a barrier appears within a loop and the array `A` is read from or written to by the loop body then no accesses to `A` can be in-flight at the loop head. This is based on the intuition that the barrier will clear the read and write sets. In the following loop, the rules will generate the candidates ¬`rdExistsA` and ¬`wrExistsA`, respectively.

```
for (...) {
  ... = A[e];
  ...
  A[e] = ...;
  __syncthreads();
}
```

r13. varPow2 and r14. varPow2NotZero These rules identify variables that are assigned power-of-two values. This is based on the intuition that power-of-two values are often used as bitmasks and in tree reductions and prefix sums. In the following loop, the rules will generate the candidates $(i \text{ bitwise\_and } (i-1)) = 0$ and $i \neq 0$, respectively, where we use the common bitwise test for an integer power-of-two value [War12].

```
for (i = C; i>0; i>>=1) {
  ...
}
```

r15. varUniform This rule is concerned with recovering uniformity for variables. In the following loop, the rule will generate the candidate $\texttt{en\$1} \wedge \texttt{en\$2} \Rightarrow \mathsf{uniform}(i)$.

```
for (i=threadIdx.x/blockDim.x; i<C; i++) {
  ...
  __syncthreads();
}
```

r16. varRelationalPow2 This rule identifies possible pairs of power-of-two variables such than one is incrementing and the other is decrementing and guesses a lock-step relation between them. We frequently see this pattern in prefix sum implementations [CDK14]. For the following loop, the rule generates candidates for a range of power-of-two values: $ij = 1$, $ij = 2$, $ij = 4$, $\cdots$, $ij = 2^{15}$.

```
j = 1;
for (i=C; i>0; i>>=1) {
  ...
  j <<= 1;
}
```

Correspondence to Prior Work In prior work [BCD+12, sec. 4.3] we gave five rules for invariant generation. These correspond to the following rules in this chapter:

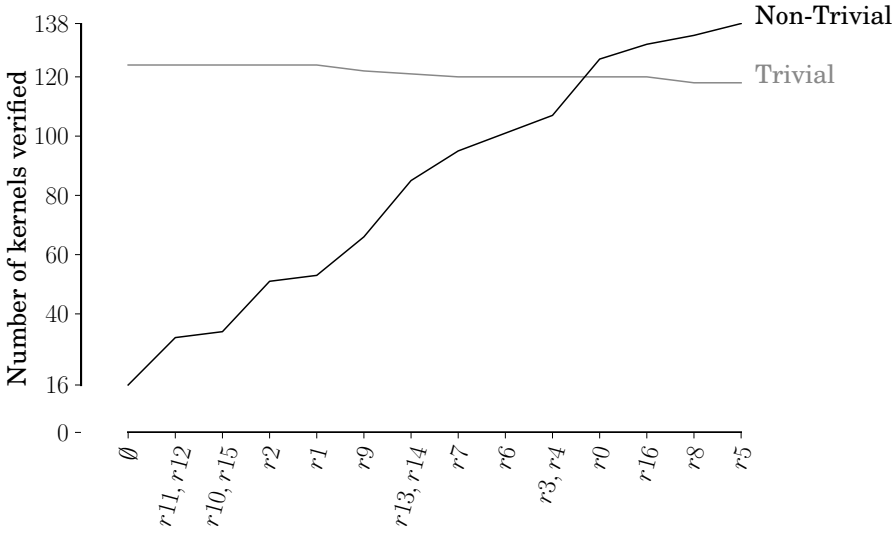
- Rules "access at thread id plus offset", "access at thread id plus strided offset" and "access at thread id plus strided offset, with strength reduction" are subsumed by rule r2 (accessStride).
- The rule "access contiguous range" is subsumed by rule r1 (accessSlice).
- The rule "variable is zero or power of two" is subsumed by rules r13 and r14 (varPow2 and varPow2NotZero).

3.4. Evolution of Precision and Performance

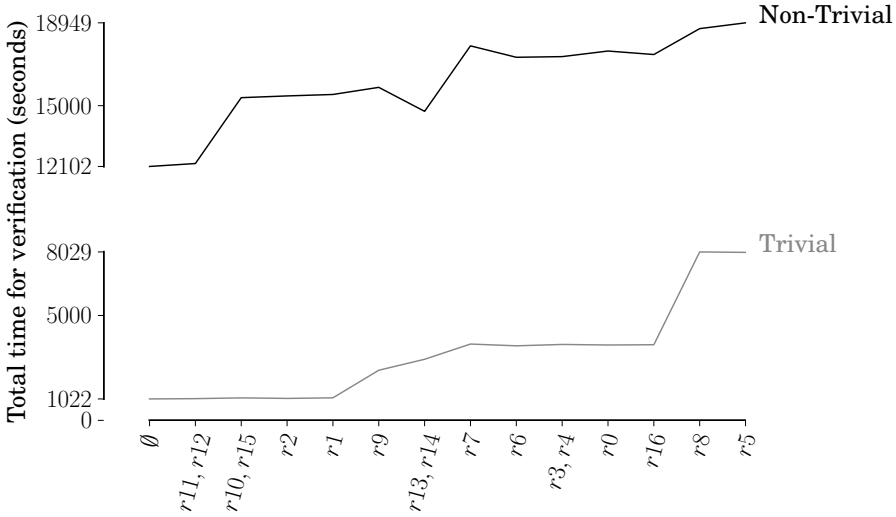
Starting with a configuration of GPUVerify that *enabled* user-supplied invariants but *disabled* all candidate generation rules, we evaluated the effect on precision and performance by running further configurations that allowed successively more rules to be used for candidate generation. By enabling rules in the same order that we developed them, this experiment mimics the evolution of GPUVerify's precision and performance. For each configuration, we measured precision as the number of kernels that verify using only the rules enabled by the configuration and performance as the time to process the LOOP set, including any timeouts. We ran this experiment with a timeout of 10 minutes. Our hypothesis was that our rules would enable a large number of kernels to verify, but at the cost of performance.

Figure 3.10 plots the development of our rules on the x-axis. Each successive point along the x-axis allows successively more rules to be used for candidate generation. We order the rules along the x-axis in the approximate order that the rules were designed and implemented in GPUVerify.[7] We give two plots for (a) precision and (b) performance. As we will show in our experimental evaluation (Experiment F0, Section 3.6), the LOOP set contains *trivial* and *non-trivial* kernels, where a trivial kernel is defined as a kernel that does not require invariant generation to verify (i.e., the invariant `true` suffices for each loop). In practice, GPUVerify does not know upfront whether a kernel is trivial or not. However, since we have found that invariant generation only negatively impacts trivial kernels, we have separated these sets in the graphs. For trivial kernels we see that the rules have a detrimental effect for precision, since some kernels now timeout due to unnecessary candidates, and also in performance: a 7.8× slowdown. These timeouts account for over half of this slowdown since we include these times in the plots of performance. For non-trivial kernels, the precision results begin (when all rules are disabled) at 16 kernels due to user-supplied invariants. We see the rules have a positive effect for precision allowing 122 further kernels to verify but at a cost of a 1.5× slowdown; although, not all rules cause slowdown (e.g., the introduction of rules r13 and r14). Considering both sets in conjunction the precision increases by 116 kernels (accounting for 6 trivial kernels that time out when enabling all rules) but at a cost of a 2× slowdown (from 218 minutes to 449 minutes for processing the whole LOOP set). Motivated by these slowdowns, we now turn to the problem of accelerating Houdini.

[7] We are only able to give an approximate ordering due to the development of GPUVerify and the subsuming of rules (see *Correspondence to Earlier Work* above).



(a) Impact on precision as successively more rules are enabled



(b) Impact on performance as successively more rules are enabled

Figure 3.10.: The evolution of (a) precision and (b) performance of GPUVerify as new rules were developed. Each successive point along the x-axis allows successively more rules to be used for candidate generation.

3.5. Accelerating Houdini using Under-Approximation

The performance of Houdini is the time it takes to return the unique, maximal conjunction from the set of candidates. Since this invariant is computed by iteratively removing false candidates until a fixpoint is reached this performance is critically dependent on (i) the number of false candidates and (ii) the underlying SMT solver's ability to refute false candidates. As Figure 3.10 shows, aggressively guessing a larger set of candidates was useful for precision but detrimental to performance. In this section we give methods to accelerate the performance of Houdini using under-approximating techniques and parallelisation.

First, we review some definitions. We say that a program S is *correct* if every execution of S is free from assertion failures. Note that this defines *partial* correctness because we do not require termination. Informally, we say that the program S *over-approximates* T (or, equivalently that T *under-approximates* S) if S has more execution behaviours than T. More formally, S over-approximates T if the correctness of S implies the correctness of T. The converse does not hold: the correctness of T does not imply the correctness of S. Intuitively, we can think of this as saying that the under-approximation T can fail in the same or *fewer* ways than S. The loop-cutting transformation (Figure 3.2) yields an over-approximation S of the original program T since the correctness of S implies the correctness of T.

3.5.1. Exploiting Under-Approximation

Our key observation is that if a candidate is refuted by an under-approximation T of a program S then the candidate is also false with respect to the original program S. That is, a refutation from an under-approximating analysis can be trusted. The converse does not hold; a candidate that is not refuted by the under-approximating analysis may or may not be a true invariant. This suggests the following strategy: run an under-approximating analysis either upfront or in parallel with the standard Houdini algorithm. Any refutations from the under-approximating analysis can be safely shared with the standard Houdini algorithm. This is potentially helpful because the under-approximating analysis may be able to cheaply refute false candidates that the standard Houdini algorithm struggles with. In other words, adding an under-approximating analysis may enable complementary refutation of false candidates. We emphasise that the standard Houdini algorithm remains ultimately responsible for computing the maximal subset of the candidates (Section 3.1.2), but that an under-approximating analysis may *accelerate* the process by refuting a false candidate more quickly. Additionally, different under-approximating analyses may be arbitrarily combined since their refutation performance may also be complementary. We

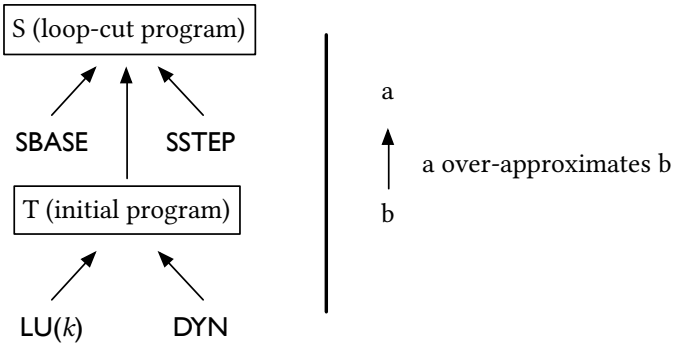


Figure 3.11.: Given an initial program T, Houdini operates on the loop-cut over-approximation S. We consider four under-approximating refutation engines: bounded loop unrolling of depth k (**LU(k)**), only checking base cases (**SBASE**), only checking step cases (**SSTEP**), and dynamic analysis (**DYN**).

now discuss four under-approximating analyses that we will evaluate in our empirical study (Section 3.6). We refer to each under-approximating analysis used to accelerate Houdini as a *refutation engine*. In Section 3.5.2 we consider how to combine refutation engines for parallel acceleration of Houdini.

In the following, let T be the initial program and S be over-approximation of T following loop-cutting (see Figure 3.2). Since Houdini operates over the loop-cut program S, we are free to use under-approximations of S or under-approximations of the initial program T for accelerating refutations. Figure 3.11 summarises four under-approximations that we now consider.

Bounded Loop Unrolling, LU(k) Loop unrolling a program for a given depth k yields a loop-free program where only the first k iterations of each loop are considered. Figure 3.12 shows the transformation of a loop after loop unrolling with $k = 2$. The loop-free fragment models k iterations of the loop. The resulting program is an under-approximation because it does not consider behaviours that require further loop iterations. The `assume false` statement means that any execution that would continue past k iterations is infeasible; it will not be considered [BL05]. Hence bounded loop unrolling is an under-approximation of the initial program T.

Splitting Loop Checking, SBASE and SSTEP As discussed in the introduction, an invariant for a loop must be established inductively. Figure 3.2 shows the loop-cutting

```
while (c) invariant φ {
  B;
}
```

```
if (c) {
  assert φ;
  B;
  if (c) {
    assert φ;
    B;
    if (c) {
      assume false;
    }
  }
}
```

Figure 3.12.: Loop unrolling of the loop on the left for $k = 2$ depth yields the loop-free program on the right

SBASE

```
assert φ;   // (base case)
havoc modset(B);
assume φ;
if (c) {
  B;
  // step case omitted
  assume false;
}
```

SSTEP

```
// base case omitted
havoc modset(B);
assume φ;
if (c) {
  B;
  assert φ; // (step case)
  assume false;
}
```

Figure 3.13.: Splitting the loop checking for a loop-cut program S yields two under-approximations, **SBASE** and **SSTEP**

transformation of the program T to the loop-free program S. It shows that the invariant must hold on loop entry (base case) and be maintained by the loop (step case). Omitting either of these assertions yields an under-approximation because the resulting program can fail in fewer ways than the original. This gives us two under-approximations of S: keeping only the base case (**SBASE**) or keeping only the step case (**SSTEP**). We also think of this as splitting the program S into two subprograms, Figure 3.13, where the correctness of both subprograms implies the correctness of S. Our intuition is that refuting candidates in each subprogram separately may be faster than dealing with the program as a whole.

Dynamic Analysis, DYN Dynamic analysis through running the program is a classic under-approximating analysis since each execution explores a real behaviour of the program. Unlike the other under-approximations we consider, dynamic analysis does not operate via a program transformation; instead, statements of the original program are simply interpreted. Similar to loop unrolling, dynamic analysis *cannot* refute candidates that are falsified due to being non-inductive. This is because they are both under-approximations of T.

For this purpose we developed our own Boogie interpreter. The Boogaloo tool [PFW13] is an existing tool that can interpret Boogie, however it does not support bit-vectors and building our own interpreter allowed us to exploit domain-specific knowledge related to the kernel transformation of GPUVerify. Particular challenges that we faced were (i) instantiating thread and group identifiers, as well as under-constrained kernel preconditions with interesting values likely to lead to refutation of candidates, (ii) evaluating candidate with respect to very large cross-products of values, and (iii) avoiding getting lost executing loops with very large static bounds, without refuting any candidates in the process. To resolve the latter challenge we designed various heuristics to enable us to stop dynamic analysis after exploring a certain loop depth, instruction count or number of loop iterations with no further refutations.

3.5.2. Parallel Refutation Sharing

Our final observation for accelerating Houdini is that multiple refutation engines can be run in parallel in a synchronisation-free manner. Parallel refutation sharing works by exchanging refutations between multiple refutation engines and the standard Houdini algorithm. Each refutation engine updates a shared pool of refutations that is read by Houdini at each iteration. We note this exchange process can be asynchronous because (i) Houdini guarantees that the number of remaining candidates strictly monotonically decreases with iteration and (ii) every refutation from an under-approximating refutation engine may be trusted. The exchange may race against refutation updates and miss picking up refutations, but the race cannot affect the soundness of the standard Houdini algorithm. In particular, invariant generation must only terminate when Houdini terminates; a refutation engine can terminate earlier, but Houdini is critical for enforcing soundness.

Parallel refutation sharing allows us to also consider running multiple Houdini instances in parallel. For example, multiple instances each using a different underlying SMT solver or configured with different option parameters. This may be advantageous due to the large configuration space offered by modern SMT solvers. In this case, invariant generation can terminate when *any* Houdini instance terminates.

3.6. Experimental Evaluation

We now evaluate the effectiveness of the techniques introduced in this chapter. The key questions that we answer:

- Precision: to what extent do the rules we have designed increase the set of kernels that GPUVerify can automatically verify? This has already been coarsely addressed by the evolution experiments summarised in Figure 3.10. So, in this section, we will quantify the dependencies and precision provided by different rules.
- Performance: how much speedup can be attained using refutation engines?

Experimental Setup Experiments were conducted on a compute cluster using 12-core nodes with Intel Xeon X5640 cores at 2.6 GHz with 16 GB RAM running RedHat Linux 6.4 using CVC4 v1.4-prerelease (29-01-2014). Times reported are averages over three runs.

Experiments are labelled Fn, Pn or An to distinguish between experiments designed to answer **f**undamental, **p**recision or **a**cceleration questions. We refer to our benchmark suite of 356 kernels, detailed in Section 3.2, as the LOOP set.

3.6.1. Potential for Precision and Performance Improvement

We ran the following experiments because they are fundamental for justifying the work in this chapter.

- **Experiment F0** establishes whether invariant generation is critical for precise reasoning in our domain of interest. We measured how many kernels trivially verify when invariant generation is disabled. Our hypothesis was that the majority of kernels *do* require invariant generation for precise reasoning, justifying the design of custom rules.
- **Experiment F1** establishes whether invariant generation is a significant bottleneck in the verification process cost of verification. It quantifies how much processing time is consumed by invariant generation in GPUVerify. Our hypothesis was that invariant generation is a significant bottleneck, justifying the design of acceleration techniques.

Experiment F0 We ran GPUVerify over the LOOP set with both user-supplied invariants and invariant generation *disabled* and noted whether a kernel verified or not. Note this experiment differs from the one in Section 3.4 because user-supplied invariants are disabled. If a kernel verified without the aid of manually or automatically generated invariants then we classify the kernel as *trivial* from an invariant generation perspective; otherwise, invariant generation is critical for precise reasoning about the kernel. This identified that invariant generation is critical for 232 out of 356 (65%) kernels; the majority of

the LOOP set. The remaining 124 kernels do not require invariant generation. Kernels in this set have loops that are irrelevant for data race-freedom checking, such as computing a thread-private value or only performing reads from shared state. Invariant generation can only slow down verification time for these kernels. In practice, GPUVerify does not know upfront whether a kernel is trivial or not: every kernel incurs the cost of invariant generation, and it is important that advanced invariant generation methods for non-trivial kernels do not adversely affect verification for kernels for which invariant generation is not actually necessary.

Experiment F1 To evaluate the value of accelerating Houdini we ran GPUVerify over the LOOP set and measured the time spent in each of the (i) frontend (including candidate generation), (ii) Houdini, and (iii) verification stages (Figure 3.4). After removing results for 26 kernels that timed-out after 10 minutes (leaving 330 kernels), we found that the total time was 12043 seconds and each stage of GPUVerify took (i) 373 seconds for the frontend, (ii) 10075 seconds for Houdini and (iii) 1595 seconds for verification, respectively. That is, Houdini consumed 84% of the total processing time of GPUVerify, a significant proportion of the total verification cost. The theoretical maximum speedup we can achieve from acceleration of Houdini across our benchmark set is thus 6×, if it were possible to completely eliminate the overhead associated with the Houdini stage.

3.6.2. Evaluating Precision

We ran the following experiments to determine the precision of the rules we have designed. All experiments in this section (i) included our user-supplied invariants and (ii) used a large timeout of 30 minutes. We chose these parameters because we are interested in the precision of the invariants afforded by our rules, regardless of performance. Using these parameters allowed our precision experiments to explore more kernels and present a larger evaluation: 10 kernels timed-out; results in this section thus report over 346 kernels.

- **Experiment P0** measures the applicability of our rules. It quantifies how many candidates, invariants and refutations were generated by each rule. Our hypothesis was that the rules we had designed would be general-purpose, with each rule yielding invariants for a significant number of benchmarks.

- **Experiment P1** measures how often each rule was *essential* for verifying a kernel. It quantifies how many kernels cannot be verified if a given rule is disabled. Each rule was devised in response to at least one kernel and with generality in mind, so

our hypothesis was that each rule would prove essential for a significant number of kernels.

- **Experiment P2** measures how often pairs of rules *influence* each other. It quantifies how often two rules generate invariants that rely on each other to be inductive. As we largely devised rules in isolation from one another, our hypothesis was that there would be few dependencies between rules.

Experiment P0 We ran GPUVerify over the LOOP set and (i) counted how many candidates were generated for each rule, and also, (ii) determined the number of candidates that were true invariants or refutations. The results are summarised in Table 3.3. The sum of the number of invariants and the number of refutations is always equal to the number of candidates generated by the rule. The *hit-rate* of a rule is the percentage of candidates that were true invariants. This shows that only one rule r1 (accessSlice) generated true invariants all of the time (100% hit-rate). We did not expect to find a large number of rules with 100% hit-rate since we designed our rules to aggressively guess candidates. Conversely, the hit-rate reveals that two rules r15 and r16 (varUniform and varRelationalPow2) rarely speculate true invariants. In the case of r16 this is because the rule generates 15 mutually exclusive candidates, where at most one candidate can be true. This holds similarly for r8 (guardBound) which generates 2 mutually exclusive candidates. This indicates that these rules could be refined to fire only under more specific conditions. Table 3.4 gives measures of central tendency for the distribution of the candidates, invariants and refutations, respectively.

Experiment P1 We say that a rule is *essential* for a kernel if the kernel cannot be verified without the rule. That is, (i) the kernel verifies when all rules are enabled, and (ii) disabling all candidates generated by the rule means the kernel is no longer verifiable: either a spurious error is reported or timeout is reached. For each rule, we ran GPUVerify over the LOOP set with that rule disabled and counted the number of kernels for which the result of verification changed.

The figures on the right of Table 3.3 summarise our results. This shows for 196 kernels (not all different since multiple rules may be essential for the same kernel), some single rule was essential for verification to succeed. Rules that are not essential for any kernels (r1, r7 and r14) are redundant and should perhaps be removed. They have been superseded by other more general rules.

In addition to noting when a rule was essential we counted two other scenarios. Firstly, we noted when the result of analysing the kernel changed from verification fail, when the

Table 3.3.: Per-rule statistics of number of candidates, refutations and invariants. The Candidates column gives two numbers for each rule: (i) the number of kernels where the rule guessed at least one candidate and (ii) the total number of candidates generated by the rule over all kernels. †10 kernels are excluded due to timeout.

	Across 346 kernels in LOOP †				**Change if rule is disabled**		
Rule	Candidates	Invariants	Refutations	Hit-rate%	Essential	Fail → TO	TO → Pass
r0. accessBreak	(81) 277	133	144	48.0	21	1	0
r1. accessSlice	(3) 4	4	0	100.0	0	0	0
r2. accessStride	(168) 518	443	75	85.5	58	3	1
r3. accessSliceBlockLower	(33) 56	40	16	71.4	1	3	1
r4. accessSliceBlockUpper	(33) 56	30	26	53.6	1	2	0
r5. noAccessConditionalLoop	(93) 630	546	84	86.7	4	0	0
r6. guardMinusInitialIsUniform	(56) 182	138	44	75.8	7	0	0
r7. guardNonNeg	(254) 1008	838	170	83.1	0	0	0
r8. guardBound	(346) 5640	2532	3108	44.9	5	0	0
r9. guardStride	(124) 392	334	58	85.2	33	0	0
r10. loopPredicateUniform	(56) 198	153	45	77.3	9	0	0
r11. noRead	(104) 261	78	183	29.9	17	0	0
r12. noWrite	(131) 306	91	215	29.7	22	0	0
r13. varPow2	(65) 286	97	189	33.9	11	0	0
r14. varPow2NotZero	(65) 286	58	228	20.3	0	1	0
r15. varUniform	(56) 2509	26	2483	1.0	2	0	0
r16. varRelationalPow2	(15) 768	30	738	3.9	5	0	0
Sum	13377	5571	7806	41.6	196	10	2

Table 3.4.: Per-kernel statistics of number of candidates, refutations and invariants

	Per-kernel measures				
	Min	Max	Median	Mean	Std
Candidates	8	256	22.0	38.7	39.9
Invariants	0	107	11.5	16.1	14.4
Refutations	0	209	10.0	22.6	30.9

rule is enabled, to timeout, when the rule is disabled. This can be seen as a positive characteristic of the rule: although the rule was not sufficient to prove the kernel correct; it did enable GPUVerify to return a response in a timely fashion. Secondly, we noted when the result of analysing the kernel changed from timeout, when the rule is enabled, to pass, when the rule is disabled. This is a negative characteristic of the rule. In this case, the same kernel[8] was affected by the disabling of either rule r2 or r3. We conclude that a candidate generated from either rule is difficult for CVC4 (the SMT solver to which verification conditions are discharged) to reason about and causes timeout.

Experiment P2 We say that a rule r *influences* another rule s if disabling the candidates generated by r affects the invariants that are generated for s. This can be the case if a candidate guessed by s is only inductive in the presence of a candidate generated by r, in which case disabling the candidates of r means that some candidate of s will be refuted. We ran GPUVerify with each rule in turn disabled and counted the number of kernels where the computed invariant differed from the invariant computed when all rules were enabled. Figure 3.14 summarises the results of this experiment. For each pair of rules (r, s) we give the number of kernels where r influenced s. We see that the matrix is sparse (with only 24 non-zero entries), which suggests that the majority of our rules are non-interfering. An exception is rule r9, which was introduced early during the development of GPUVerify.

3.6.3. Evaluating Refutation Engine Performance

We ran the following experiments to determine the performance of refutation engines for accelerating Houdini. All experiments in this section (i) included our user-supplied invariants and (ii) used a timeout of 10 minutes. We chose these parameters because we are interested in measuring the performance improvement of Houdini using a timeout we believe to be reasonable for day-to-day use of GPUVerify. In this section we denote

[8]The `sgemmNN` kernel in the SHOC suite.

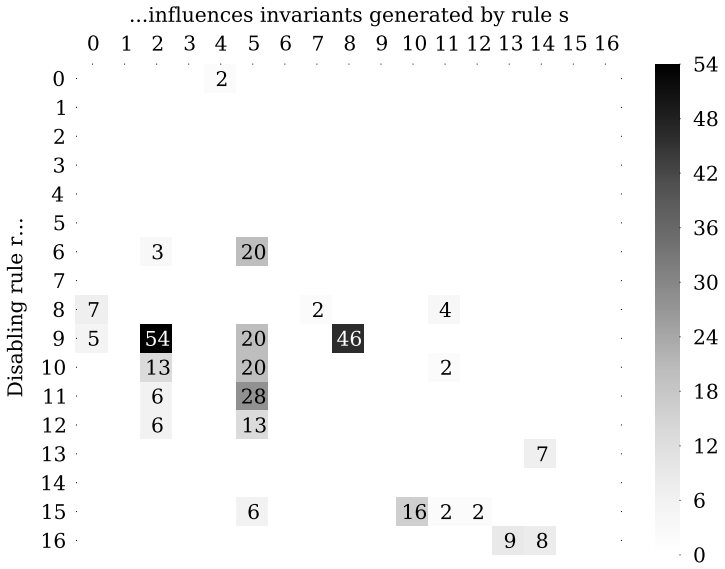


Figure 3.14.: Heatmap for each pair of rules (r, s) showing the number of kernels where r influenced s

the standard Houdini algorithm as **H** and evaluate refutation engines that perform loop unrolling for depth 1 and 2, **LU**(1) and **LU**(2), splitting loop checking, **SBASE** and **SSTEP**, and dynamic analysis, **DYN**.

- **Experiment A0** measures the performance of each of our proposed refutation engines in isolation. For each refutation engine we quantify the time taken to reach a fixpoint and number of refuted candidates. Our hypothesis was that different refutation engines would have different performance characteristics and have complementary refutation ability.
- **Experiment A1** evaluates the speedup of using a refutation engine as an upfront accelerator for Houdini. It quantifies the speedup of using a series of refutation engines sequentially followed by the standard Houdini algorithm.
- **Experiment A2** evaluates the speedup of using our refutation engines in parallel with Houdini. It quantifies the speedup of using parallel refutation sharing.

Experiment A0 We ran each refutation engine in isolation and measured the time to compute a fixpoint and the number of refuted candidates. To limit dynamic analysis runtime we chose to stop execution after 1000 loop iterations. The results are summarised

Table 3.5.: Refutation engine performance and throughput

Engine	Refutations	Total time (sec)	Throughput (refutations/sec)
H	5310	16425	0.32
LU(1)	3115	45430	0.07
LU(2)	2528	61216	0.04
SBASE	2451	21725	0.11
SSTEP	2183	55880	0.04
DYN	2477	1813	1.35

in Table 3.5 where we use throughput, the number of refutations per second, as an indicator of performance. We see that dynamic analysis is extremely effective. We also see that **LU(2)** is *less* effective than **LU(1)**, due to the large verification conditions that arise from the deeper unwinding. We thus do not consider **LU(2)** further.

The same experimental results can be used to examine the extent to which different refutation engines are complementary or redundant with respect to the candidates that they refute. The Venn diagram of Figure 3.15 shows the number of refutations common to the refutation engines we consider. We define the redundancy (or similarity) between two refutation engines e_1 and e_2 as the Jaccard index [Jac12]: the size of the intersection (the number of shared refutations) divided by the union (the total number of refutations), i.e., $|e_1 \cap e_2| \;/\; |e_1 \cup e_2|$. High redundancy indicates that the engines refute a similar subset of candidates; low redundancy indicates complementarity. Using this metric, dynamic analysis is the most complementary refutation engine since its redundancy with any of the other three engines is low (≤ 0.27) compared with all other combinations, such as the most redundant pairing of **LU(1)** and **SBASE** (0.59).

Experiment A1 Our second performance experiment ran each refutation engine as an upfront accelerator for Houdini. We use the notation e; **H** to denote the sequential pipeline of using the refutation engine e before Houdini. For each configuration we measured the total time for the accelerator and Houdini to return an inductive invariant. The top half of Table 3.6 summarises the results compared to the standard Houdini algorithm. The table shows that **DYN** and **LU(1)** gave the best overall speedups (1.25× and 1.13×, respectively), and that **SBASE** led to a modest slow down overall. The maximum speedup results show a best case speedup of 778× for **DYN** for the Rodinia `lud_internal` kernel: by quickly eliminating some hard-to-refute candidates, **DYN** enables invariant generation to complete in less than a second where otherwise the timeout limit of 10 minutes is

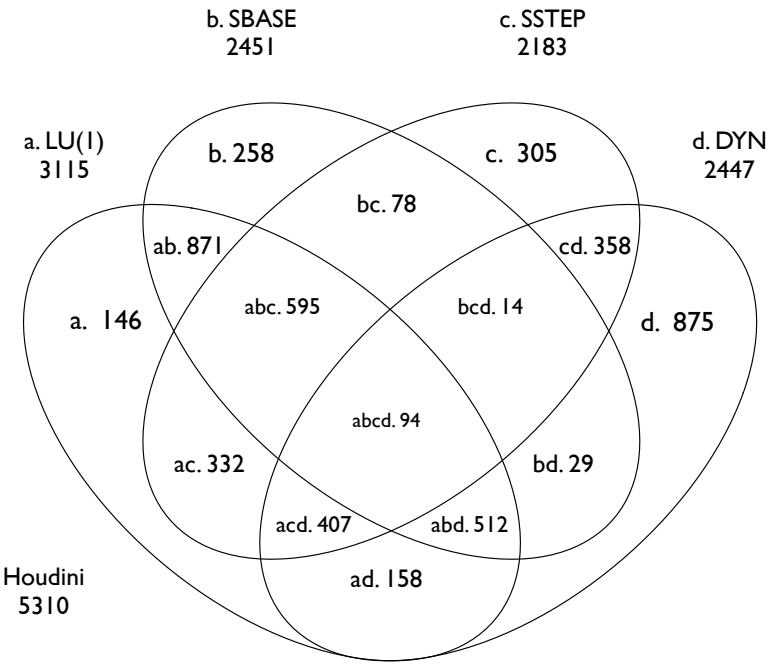


Figure 3.15.: Venn diagram showing the number of complementary and redundant refutations generated by each engine

Table 3.6.: Speedups from using refutation engines sequentially and in parallel

			Per-kernel speedup			
Configuration	**Time (sec)**	**Speedup**	Min	Max	Median	Geomean
H	20661	1.00				
LU(1); **H**	18269	1.13	0.14	269.54	0.92	0.99
SBASE; **H**	21078	0.95	0.07	2.51	0.89	0.87
SSTEP; **H**	19668	1.05	0.07	3.40	0.93	0.97
DYN; **H**	16496	1.25	0.01	778.21	1.17	1.42
Parallel	12570	1.64	0.20	612.87	1.02	1.49

reached. **LU**(1) gives a speedup of two orders of magnitude for the *gpgpu-sim* `md5` kernel, for similar reasons.

Experiment A2 In our final experiment we ran all refutation engines in parallel with Houdini using parallel refutation sharing. The bottom of Table 3.6 summarises the result (labelled 'Parallel'). In this configuration dynamic analysis was not stopped after exploring a certain number of iterations; instead, it was allowed to run until Houdini completed. We see that a parallel configuration enables a 1.64× speedup above the standard Houdini algorithm and 1.31× speedup above the best sequential configuration **DYN**; **H**.

3.7. Related Work

Invariant generation is a long-standing challenge in program verification that has received significant attention. Techniques proposed to address this problem include abstract interpretation [JM09], predicate abstraction [BPR01], Craig interpolation [McM06], template-based constraint solving [GR09], abduction [DDLM13] and dynamic analysis [ECGN01]. In this section we discuss work that is most closely related to the study in this chapter.

Candidate-Based Invariant Generation Houdini was introduced as an annotation assistant for the Extended Static Checker for Java (ESC/Java) [FL01], a lightweight static analysis tool that checks for runtime errors such as null pointer dereferences. The original aim of Houdini was to reduce the cost of user-supplied invariants. A formal presentation of Houdini, including a proof of its termination, is given in [FJL01]. Houdini can be viewed as a restriction of *predicate abstraction* [GS97] restricted to conjunctions of predicates. Under predicate abstraction, program state is abstractly represented as a valuation of

predicates and program statements are predicate transformers. The predicate themselves are usually generated from assertions in the program or refinement techniques (in the context of counterexample-guided abstract refinement [CGJ$^+$03]). Each Houdini iteration is a form of predicate abstraction where each candidate is a predicate and each loop is a predicate transformer. The restriction to only considering conjunctions of predicates reduces the maximum number of SMT calls from exponential [BPR01] to linear [FL01] in the number of predicates and is what makes the performance of Houdini predictable. This restriction also makes it impossible to generate invariants with disjunctions or implications over predicates using Houdini.

Abstract Interpretation Abstract interpretation is a general framework for building program analyses [CC77]. A central idea is that of an *abstract domain* that approximates, in a well-defined mathematical sense, the concrete properties of the program that we wish to track. The abstract domain is a partially-ordered structure (e.g., a *lattice*) where the ordering corresponds to the precision of the property being tracked. For example, the interval abstract domain approximates the range (or interval) of values that a variable can take and ordering is given by interval inclusion: e.g., the property $0 \leq x \leq 0$ is ordered below (and hence is more precise) than the property $0 \leq x \leq 100$. Other well-known numerical abstract domains include octagons and convex polyhedra. Predicate abstraction is abstract interpretation over the abstract domain of Boolean combinations of predicates and hence Houdini is also a form of abstract interpretation where we further restrict the abstract domain to conjunctions of predicates (the powerset of candidates). Invariant generation using abstract interpretation involves computing the least fixpoint of the loop over a given abstract domain. Different abstract domains give different trade-offs between precision and performance. The least fixpoint corresponds to the most precise property that we can state about the loop. For example, the Interproc analyser can generate invariants over the abstract domains of intervals, octagons and convex polyhedra using the Apron library [JM09].

The main disadvantage of invariant generation using abstract interpretation is that it is inflexible: the chosen abstract domain determines the form of invariants that can be generated. For example, the interval domain generates invariants of the form $c \leq x$ and $x \leq c$ (ranges of values where x is a program variable and c is a constant); the octagon domain generates invariants of the form $\pm x \pm y \leq c$ (polyhedra with at most 8 faces where x, y are program variables and c is a constant); and the convex polyhedra domain generates invariants of the form $\mathbf{A}\vec{x} \leq \vec{b}$ (a system of linear inequalities where A is an $m \times n$ matrix, $\vec{x}$ is a vector of n program variables and $\vec{b}$ is a vector of m constants) [JM09].

Hence, generating invariants beyond a given abstract domain requires new abstract domains to be created. This is heavyweight compared to candidate-based methods that can add new rules on an example-drive basis, as we have done with GPUVerify. A second disadvantage is predictability. If the chosen abstract domain does not satisfy the ascending chain condition (all increasing sequences of elements eventually converge) then the fixpoint computation may not converge and termination is not guaranteed. In this case, a heuristic (*widening*) can be employed to force convergence to a post fixpoint and guarantee termination; subsequently, further heuristics (*narrowing*) can be employed to regain some of the lost precision [CC77]. In contrast, the convergence to a fixpoint is predictable when using Houdini (because the number of candidates, and hence the abstract domain, is finite).

Invariant Generation for Affine Programs An affine program is a program that is restricted to operate on unbounded integers and affine (linear) expressions over program variables. Invariant generation techniques for affine programs includes Craig interpolation [McM06], abduction [DDLM13] and abstract acceleration [JSS14]. The main disadvantage of these techniques is the restriction on input programs. In the domain of GPU kernels, programs operate on fixed-width bit-vectors and floating-point numbers and use non-affine arithmetic. We require invariants over bit-vectors to reason precisely about arithmetic using power-of-two values (frequently encoded using shifting, masking and other bitwise-operators) and require support for uninterpreted functions to abstract floating-point operators. Examples of non-affine arithmetic in GPU kernels are reduction operators that involve loops in which a counter exponentially varies power-of-two values between an upper and lower bound.

Array Invariants The work of Kovács and Voronkov has examined the problem of generating loop invariants for programs containing arrays [KV09]. The key idea is to automatically derive *update predicates* that express the updates applied to arrays in a loop. The update predicates capture the element written to, the value written and the iteration in which the write occurred. These update predicates are then generalised by an automatic first-order theorem prover, which enables non-trivial invariants (containing quantifier alternations) to be generated. Since many of our rules are concerned with the access pattern of the loop (analogous to the update predicates of this work), there is scope for applying this work to GPU programs. A challenge will be eliminating quantifiers that are implicitly given by the two-thread reduction.

Dynamic Invariant Generation The techniques discussed above use static analysis to generate invariants with certainty. In contrast, dynamic invariant generation techniques speculate *likely* invariants from program executions. A likely invariant is an assertion that has met some statistical confidence level over the observed program executions; it may or may not be a true invariant. Dynamic invariant generation was first introduced by the Daikon system [ECGN01]. Likely invariants are discovered by instrumenting the program to trace variables of interest, running the instrumented program over a test suite and guessing and checking invariants with respect to the observed program executions. The accuracy of the generated likely invariants depends in part on the quality and completeness of dynamic executions. The refutation engine using dynamic analysis introduced in this work (Section 3.5.1) is similar in the sense that we use dynamic analysis to eliminate false candidates. The main difference is that we automatically generate test cases using kernel preconditions rather than using a test suite.

Nimmer and Ernst compared the effectiveness of Daikon against Houdini for helping programmers to write invariants using ESC/Java [NE02]. This found that both tools were beneficial to users in different ways: Daikon helped users to express more invariants than required with no loss of time and Houdini helped users to express more invariants in fewer annotations.

3.8. Summary

A key problem in automating program verification is the generation of *invariants*. In the context of GPUVerify, we have introduced new rules for reasoning precisely about GPU kernels using candidate-based invariant generation. Refutations are a new parallelisable method for accelerating the invariant generation computation of Houdini. Finally we have studied the effectiveness of these techniques and evaluated it along two important axes, precision and performance, against a large suite of 356 kernels.

4. Barrier Invariants: Precise Reasoning for Data-Dependent Kernels

In this chapter we address the problem of precise and scalable reasoning for *data-dependent* GPU kernels. This important class of kernel, characterised by control or data flow that is dependent on shared state, is beyond the scope of existing techniques. We present a new abstraction, *barrier invariants*, that enables precise reasoning for data-dependent kernels whilst retaining the two-thread reduction necessary for scalable verification. The main results of this chapter are:

- Identifying the problem of data-dependent GPU kernels and describing why existing techniques fail, thus motivating the development of barrier invariants as a method for recapturing this lost precision.
- A semantics based on barrier invariants, and soundness result that yields a verification method. We then show how it can be made quantifier-free.
- A case study using barrier invariants to precisely reason about a real-world data-dependent kernel, stream compaction. A critical subprocedure of stream compaction is the *prefix sum operation* and we use barrier invariants to prove a functional specification of three distinct and widely-used prefix sum implementations.
- An experimental evaluation showing that barrier invariants enable precise and scalable reasoning for data-dependent GPU kernels.

Relation to Published Work The core material of this chapter was published in [CDK+13]. We give additional material in the form of: (i) an extended syntax, semantics and soundness proof for a kernel language supporting conditionals and loops; (ii) a result showing the correspondence between barrier invariants and a refinement of the equality abstraction [BCD+12]; and (iii) barrier invariants for Brent-Kung and Kogge-Stone prefix sum kernels, which are only covered briefly by the paper.

4.1. The Need for Barrier Invariants

As discussed in Chapter 2, the techniques of GPUVerify and PUG achieve scalable verification using a two-thread reduction and ensure soundness using shared state abstraction. The purpose of the shared state abstraction is to over-approximate the behaviour of threads not modeled by the two-thread reduction. In particular, the shared state abstraction determines the *contents* of shared state, as seen by each thread. A barrier guarantees synchronisation and memory consistency; a kernel uses barriers to allow communication between threads via shared state.[1] On reaching a barrier a thread stalls until all threads have reached the barrier and all shared memory accesses issued by all threads have completed. If no races have occurred then the contents of shared state is deterministic when the barrier itself completes. The contents of shared state is determined by the state of the kernel at the previous barrier (or at kernel entry when considering the first barrier) and the set of shared memory writes issued by all threads since the previous barrier (see the G-SYNC rule of [BCD+12]). However, the two-thread reduction means the contents of shared state is unknown (except for the locations directly accessed by the pair of threads under consideration). Thus, the role of the shared state abstraction is to over-approximate the possible behaviours of other threads.

The existing techniques of GPUVerify and PUG consider two simple abstractions. The *adversarial abstraction*, considered by both tools, makes no assumptions about the contents of shared state. Shared state is simply not modeled: every read from shared state receives a *non-deterministic* value and every write to shared state is ignored (a no-op). Consequently, a thread may read any arbitrary value from shared state and, moreover, multiple reads of the same location by the same thread can yield different values. The *equality abstraction*, considered only by GPUVerify, models shared state but sets it *non-deterministically* at each barrier so that each thread sees a consistent view of memory. In [BCD+12], the semantics equip each thread with a *shadow copy* of the shared state. At a barrier, when using the equality abstraction, each shadow copy is set non-deterministically, but equally. That is, each thread sees an arbitrary but consistent value (both threads agree on the value) from each location in shared state.

The examples of Figure 4.1 (deliberately contrived for the purpose of illustration) show differences between the choice of shared state abstraction. All the examples are race-free and correct, however, not all abstractions can precisely capture this fact. For each example and abstraction, a tick ✓ or cross × denotes whether the example can be precisely reasoned

[1] In CUDA and OpenCL, barriers only guarantee synchronisation and memory consistency for threads *within* the *same* group. There are no reliable mechanisms for inter-group synchronisation whilst executing a kernel. Consequently, in this chapter we restrict our discussion to the single-group case.

	(a)	(b)	(c)	(d)

	`v = 2 * tid;`
	`w = 2 * tid + 1;`
	`A[v] = A[w];`
		`v = B[0];`
		`w = v ? tid : tid+1;`
		`A[w] = tid;`
		`v' = B[0];`
		`assert(v == v');`
			`A[tid] = tid;`
			`barrier();`
			`v = tid;`
			`assert(A[v] == v);`
				`A[tid] = tid;`
				`barrier();`
				`v = (tid+1)%n;`
				`assert(A[v] == v);`

Adversarial	✓	×	×	×
Equality	✓	✓	×	×
Known-Equality	✓	✓	✓	×
Barrier Invariants	✓	✓	✓	✓

Figure 4.1.: Examples with differing results using the adversarial and equality abstractions

about using this abstraction. The figure lists two further abstractions in anticipation of the developments of this chapter — a refinement of the equality abstraction, *known-equality*, and barrier invariants. These new abstractions are more refined than either the adversarial or equality abstractions.

In these examples, `tid` denotes the id of a thread, `v` and `w` are thread-private variables and `A` and `B` are arrays in shared state. In example (a) each thread is responsible for two neighbouring indices of `A`. Consider threads with ids $0, 1, 2, 3$ then the respective values of the indices `v` and `w` are $0, 2, 4, 6$ and $1, 3, 5, 7$. This example is race-free using any abstraction because the contents of shared state is irrelevant for verifying the correctness of this code.

In example (b) we test two behaviours that distinguish the adversarial abstraction. Initially, each thread reads the first element of `B` into a thread-private variable `v`. Then, depending on the value of `v`, we assign either `tid` or `tid+1` into `w`, which is then used as an index for writing to `A`. Due to consistency the value read into `v` will be the same for all threads and hence either *every* thread will write to `A[tid]` or every thread will write to `A[tid+1]`; because these cases are mutually exclusive, the writes to `A` are race-free. For example, consider threads with ids $0, 1, 2, 3$ then the respective values of the `w` index will be either $0, 1, 2, 3$ or $1, 2, 3, 4$. Using the adversarial abstraction the reads of `B[0]` may yield different values, violating the mutually exclusive choice of assignment for `w`, resulting in a (false positive) data race. All other abstractions model consistency and so correctly reason about the example. The second behaviour, tested by the assert, shows that the adversarial abstraction is even weaker. The same location `B[0]` is read again into a second thread-private variable `v'`. Using the adversarial abstraction these reads (even from the same location) can yield different values so the assert fails.

In example (c) each thread writes its identifier `tid` into its corresponding index of the array `A`. After the barrier, the example asserts that the value of `A` at index `tid` is still

`tid`. This example fails under the adversarial abstraction and the equality abstraction. Using the adversarial abstraction the read of the assert can yield any non-deterministic value. The same result occurs under the equality abstraction because shared state is set non-deterministically at the barrier. If we moved the assertion above the barrier then the equality abstraction would suffice. We defer discussion of example (d) to Section 4.3.5, where we introduce the known-equality abstraction.

We now introduce data-dependent kernels that are beyond the scope of the existing adversarial or equality abstractions. A data-dependent GPU kernel has control or data flow that is dependent on shared state. Consequently, the behaviour of a single thread may depend on the control or data flow of other threads via communication through shared memory. In particular, the access pattern of a given thread may be determined by the control or data flow of other threads. The abstractions considered by existing work are effective for *data-independent* kernels; however, they cannot reason precisely about data-dependent kernels.

A Simple Data-Dependent Kernel Consider the following kernel where `A` and `B` are arrays in shared memory, `tid` denotes the id of a thread, and `f` is a side-effect free procedure which may read from shared state and ensures that for distinct threads s and t, $\texttt{f}(s) \neq \texttt{f}(t)$.

```
A[tid] = f(tid);
barrier();
B[A[tid]] = tid;
```

This kernel is data-dependent because array `B` is written to at an index which is computed by reading from shared state. The kernel is race-free because of the guarantees of the procedure `f`. To see why this cannot be reasoned about precisely using the shared state abstractions above consider the execution with respect to an arbitrary pair of distinct threads, s and t. Because the threads are distinct, $s \neq t$, execution of `A[tid] = f(tid)` is race-free and will lead to a state in which $\texttt{A}[s] \neq \texttt{A}[t]$. Using the adversarial abstraction any value can be read from shared state. However, using the equality abstraction the arrays `A` and `B` are set non-deterministically at the barrier. In particular, a possible assignment that satisfies either abstraction is $\texttt{A}[s] = \texttt{A}[t] = 0$, which causes the statement `B[A[tid]] = tid` to result in a (false positive) data race on `B` at index 0. That is, the adversarial and equality abstractions are too coarse to reason precisely about this simple data-dependent kernel.

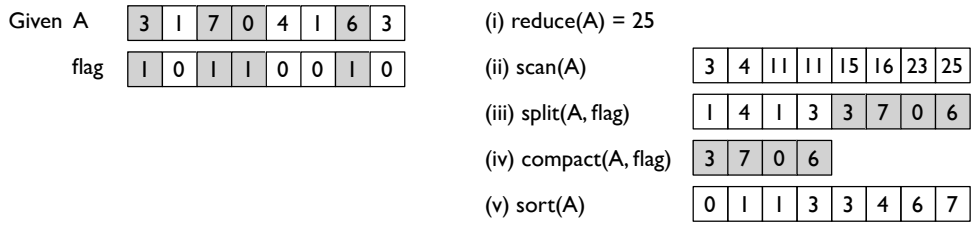


Figure 4.2.: Data-parallel primitive examples. Given the inputs `A` and `flag`: (i) *reduce* computes the sum of the elements, $a_1 + a_2 + \cdots + a_n$; (ii) *scan* computes the sum of all inclusive prefixes, $[a_1, (a_1 + a_2), \ldots, (a_1 + a_2 + \cdots + a_n)]$; (iii) *split* packs the elements of `A` so that elements with `flag` $= 0$ precede elements with `flag` $= 1$; (iv) *compact* outputs the elements of `A` where `flag` $= 1$; (v) *sort* computes a sorted array. The split, compact and sort primitives can themselves be built using the scan primitive [Ble93, Hor05] and are data-dependent.

The Importance of Data-Dependent GPU Kernels The class of data-dependent GPU kernels is small but important. The experimental results of GPUVerify and PUG show that the majority of kernels are not data-dependent and require only coarse shared state abstractions. In a review of 605 kernels from nine sources we found that 20 kernels (3%) were data-dependent [BBC+14]. Despite this, data-dependent GPU kernels are critical when considering data-parallel primitives. A data-parallel primitive is a common building block or pattern for designing parallel algorithms and applications [Ble93]. Examples of data-parallel primitives include reduce, scan, split, compact and sorting operations.[2] Figure 4.2 gives examples of these primitives. The scan or prefix sum operation [Hor05, HSO07, SHZO07] is a key data-parallel primitive: it can be used to build other primitives (split, compact and sorting) that are data-dependent. In turn, any algorithm or application that uses these primitives exhibits data-dependent behaviour. In Section 4.2 we discuss a stream compaction application, a generalisation of the compact primitive, which is data-dependent.

Barrier Invariants This chapter develops *barrier invariants* as a technique to address the problem of precise reasoning for data-dependent kernels. The key idea is to annotate each barrier with an invariant: a property of shared state that must hold each time the barrier is reached. This allows a richer shared state abstraction: instead of setting the shared state to an arbitrary value, the shared state is set to an arbitrary value *satisfying*

[2] GPU libraries implementing data-parallel primitives include the Thrust Parallel Algorithms Library (`http://thrust.github.io`), the CUDA Data-Parallel Primitives Library (`http://cudpp.github.io`) and the OpenCL Data-Parallel Primitives Library (`https://code.google.com/p/clpp/`).

the barrier invariant. Properties captured before the barrier can be maintained across the barrier. This coincides with the programmer's intuition that a barrier allows sharing of values, and therefore properties, between threads. Barrier invariants enable more precise reasoning whilst retaining the scalable approach of the two-thread reduction.

For example, a barrier invariant allows us to re-capture the precision lost by the adversarial and equality abstractions in the problematic data-dependent kernel. The invariant states that the elements of `A` are distinct (where x and y range over thread ids):

```
A[tid] = f(tid);
barrier(); barrier_invariant(∀x ≠ y : A[x] ≠ A[y]);
B[A[tid]] = tid;
```

Using the two-thread reduction, the invariant is established by checking that $\texttt{A}[s] \neq \texttt{A}[t]$ for an arbitrary pair of distinct threads s and t. Because s and t are arbitrary this proves that the invariant holds for *all* pairs of distinct threads (the *forall introduction* rule [HR04, sec. 2.3]). Therefore, after the barrier, it is sound to assume that the invariant for all pairs of distinct threads and, in particular, the threads s and t under consideration. This property is then sufficient to show race-freedom of the write to `B`.

Barrier Invariant Instantiation In fact, in the example above, it suffices to assume the barrier invariant *only* for the pair (s, t). That is, only the assumption $\texttt{A}[s] \neq \texttt{A}[t]$ after the barrier is necessary for proving race-freedom; other pairs of distinct threads are irrelevant. This observation is useful for avoiding a quadratic number of assumptions (for all pairs of distinct threads) after the barrier. We will use this intuition for formalising the instantiation of a barrier invariant in a quantifier-free manner (Section 4.3.4).

4.2. Stream Compaction Example

Figure 4.3 presents pseudo-code for a *stream compaction* algorithm: a parallel program that filters an input array with respect to a predicate p. Stream compaction is commonly used for removing redundant or dead elements from a data set and has many applications in GPU programming, including parallel breadth first tree traversal, ray tracing and collision detection [BOA09]. For an input array `data`, each thread t tests its respective element `data`$[t]$ against p and writes 1 to the temporary array `flag` if the element satisfies p and 0 otherwise. In the *compact* stage, each thread t for which p`(data[`t`])` holds must write to an index of output array `out` such that all elements to be kept are written contiguously. This index is the sum of the values in `flag` at indices $0 \leq i < t$.

```
// data, out, idx: arrays in shared memory
procedure compact(data, out)
  // (i) test each element with predicate p
  in parallel for each thread tid
    flag[tid] = p(data[tid])
  // (ii) compute indices for compaction
  idx = prescan(flag)
  // (iii) compaction
  in parallel for each thread tid
    if (flag[tid])
      out[idx[tid]] = data[tid]
```

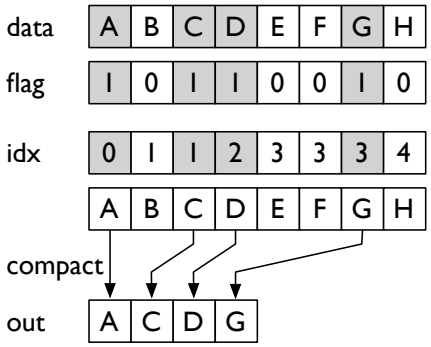


Figure 4.3.: Stream compaction program and example (image courtesy M. Harris [HSO07])

The index can be computed using an *exclusive prefix sum* operator, also known as a *prescan* [Ble93, HSO07]. This is a variant of the scan operation (Figure 4.2). Given an array $[a_1, a_2, \ldots, a_n]$ and an associative operator $\oplus$ with identity e, the prescan operator computes the sums of all exclusive prefixes: $[e, a_1, (a_1 \oplus a_2), \ldots, (a_1 \oplus a_2 \oplus \cdots \oplus a_{n-1})]$. For example, taking $\oplus$ and e to be $+$ and 0, respectively, the prescan of the array $[3, 1, 7, 0, 4, 1, 6, 3]$ is $[0, 3, 4, 11, 11, 15, 16, 22]$. The prescan can be formed by computing the scan and then shifting the elements right by one and inserting the identity.

A kernel implementing stream compaction is *data-dependent* because the access pattern of a thread is determined by the values of the input array: not just the element that 'belongs' to a thread, but also the `flag` result of all preceding threads. Determining data race-freedom depends on the correctness of the prescan and, in particular, the constraints the operation imposes on the shared array `idx`: if $\texttt{flag}[s] = \texttt{flag}[t] = 1$ for distinct threads s and t (so that both s and t will write to the output array), then race-freedom requires that $\texttt{idx}[s] \neq \texttt{idx}[t]$.

Figure 4.4 presents an OpenCL kernel for stream compaction, to be executed by a group of `n` threads. The φ annotations that follow `barrier()` statements are barrier invariants that we will use to prove the prescan specification in Section 4.4. The id of a thread is denoted `tid`, and `data`, `out`, `flag` and `idx` are arrays declared in GPU shared memory. The prescan operation is performed inline, implemented from an algorithm due to Blelloch [Ble93]. This prescan implementation is used frequently in the GPU code repositories we have examined.

Although the algorithm works in-place over the `idx` array, it is helpful to see that the correctness comes from viewing the array as a tree. If the length n of `idx` is a power-of-two, the array can be viewed as a balanced binary tree of depth $\lg n$. Figure 4.5, due to Blelloch [Ble93], shows the state of `idx` as the prescan proceeds: colouring the

```
__kernel void compact(
  __global TYPE *out,
  __global TYPE *data,
  __local unsigned *flag,
  __local unsigned *idx,
  unsigned n) {

  unsigned left, right;
  unsigned tid = get_local_id(0);

  // (i) test each element with predicate p
  flag[tid] = p(data[tid]);
  // (ii) compute indices for compaction
  barrier(CLK_LOCAL_MEM_FENCE); // φ_load
  if (tid < n/2) {
    idx[2*tid]     = flag[2*tid];
    idx[2*tid + 1] = flag[2*tid + 1];
  }

  // (ii)(a) upsweep
  unsigned offset = 1;
  for (unsigned d = n/2; d > 0; d /= 2) {
    barrier(CLK_LOCAL_MEM_FENCE); // φ_us
    if (tid < d) {
      left  = offset * (2 * tid + 1) - 1;
      right = offset * (2 * tid + 2) - 1;
      idx[right] += idx[left];
    }
    offset *= 2;
  }

  // (ii)(b) downsweep
  if (tid == 0) idx[n-1] = 0;

  for (unsigned d = 1; d < n; d *= 2) {
    offset /= 2;
    barrier(CLK_LOCAL_MEM_FENCE); // φ_ds
    if (tid < d) {
      left  = offset * (2 * tid + 1) - 1;
      right = offset * (2 * tid + 2) - 1;
      unsigned temp = idx[left];
      idx[left] = idx[right];
      idx[right] += temp;
    }
  }
  barrier(CLK_LOCAL_MEM_FENCE); // φ_spec

  // (iii) compaction
  if (flag[tid]) out[idx[tid]] = data[tid];
}
```

Figure 4.4.: OpenCL stream compaction kernel using n threads, prescan inlined. The kernel assumes that `data` and `out` are arrays of `TYPE` (defined by some macro) and that `p()` is a predicate defined over this type.

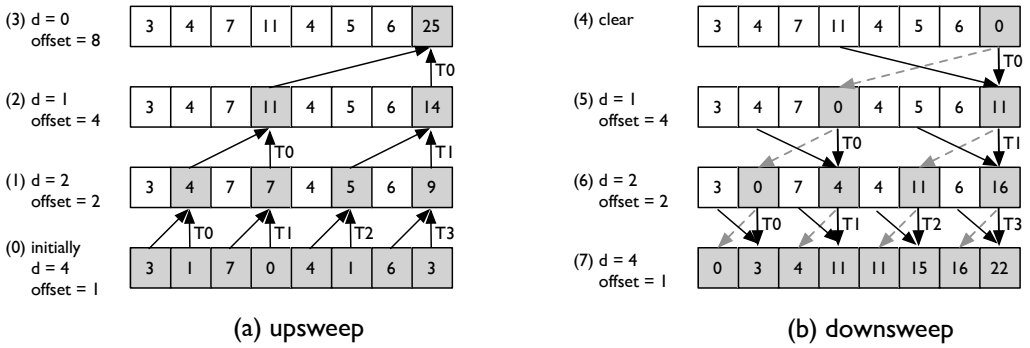


Figure 4.5.: Blelloch prescan example using $n = 8$ elements

elements that are updated at each iteration shows the tree traversed by the algorithm. Each iteration of the upsweep and downsweep can be identified by the value of `offset`, which gives the 'distance' between vertices at that depth of the tree, and `d`, which denotes the number of active threads. The parallelism of the algorithm is due to the vertices at the same depth being updatable simultaneously. The figure shows the assigned thread (T0, T1, T2, T3) per sub-operation of the upsweep and downsweep. Note that the thread-to-element assignment changes at different depths of the tree.

a. The **upsweep** is a reduction working from the leaves of the tree up to the root. Pairs of vertices (`idx[left]`, `idx[right]`) are summed until the root contains the full reduction and other elements form partial sums.

b. The **downsweep** combines the upsweep partial sums to give the prescan result. Initially, the identity 0 is inserted into the root, marked 'clear' in step (4) of Figure 4.4. The downsweep then traverses down the tree. At each level of the tree, an active vertex: (i) copies its value to its left child `idx[left]` (the dotted arrow), and (ii) sums its value with the old value of its left child `temp` (this value is a partial sum generated by the upsweep), storing the value in its right child `idx[right]` (the black arrows).

The prescan specification ensures that $\texttt{idx}[x] = \sum_{0 \le i < x} \texttt{flag}[i]$ for all x. The stream compaction kernel additionally guarantees that the input is non-negative: $0 \le \texttt{flag}[x]$ for all x. Together these imply that the output satisfies a monotonic property: for all $x < y$ we have $\texttt{idx}[x] + \texttt{flag}[x] \le \texttt{idx}[y]$.

Proof. By the prescan specification, we have $\texttt{idx}[x] = \sum_{0 \le i < x} \texttt{flag}[i]$ and $\texttt{idx}[y] = \sum_{0 \le i < y} \texttt{flag}[i]$. Then, because $x < y$ we have $\texttt{idx}[y] = \sum_{0 \le i < x} \texttt{flag}[i] + \sum_{x \le i < y} \texttt{flag}[i]$,

which is $\texttt{idx}[x] + \sum_{x \leq i < y} \texttt{flag}[i]$. Additionally since all elements of flag are non-negative we know $\texttt{flag}[x] \leq \sum_{x \leq i < y} \texttt{flag}[i]$. Hence, we have $\texttt{idx}[x] + \texttt{flag}[x] \leq \texttt{idx}[y]$, as required. □

Hence, for all $x \neq y$, if $0 < \texttt{flag}[x] \wedge 0 < \texttt{flag}[y]$ then $\texttt{idx}[x] \neq \texttt{idx}[y]$, which suffices to prove race-freedom.

Proof. Assume that $x < y$. Then the monotonic property means that $\texttt{idx}[x] + \texttt{flag}[x] \leq \texttt{idx}[y]$. Additionally, since $0 < \texttt{flag}[x]$ we must have a strict inequality $\texttt{idx}[x] < \texttt{idx}[y]$. This holds symmetrically if $y < x$ and therefore $\texttt{idx}[x] \neq \texttt{idx}[y]$, as required. □

Blelloch proved that his algorithm satisfies the prescan specification using induction on the pre-order traversal of the tree [Ble93]. Unfortunately, we cannot use this proof directly as we need to prove the specification of the kernel implementing the algorithm (direct source code analysis), rather than the algorithm itself. Furthermore, the proof by Blelloch requires a global view of shared state, whereas the use of the two-thread reduction means that we require a method that can reason from the point of view of an arbitrary thread. The development of barrier invariants will give us another route of attack.

4.3. The Two-Thread Reduction with Barrier Invariants

We now give a formal semantics to barrier invariants. Starting with a simple kernel programming language with barrier invariants (Section 4.3.1) we give (i) a concrete lock-step semantics and (ii) an abstract semantics using the two-thread reduction (Section 4.3.2). The main result of this section is a soundness theorem: if a kernel is correct with respect to the abstract semantics then the kernel is correct with respect to the concrete semantics (Section 4.3.3). Hence, it suffices to check a kernel with respect to the abstract semantics. We combine this result with the notion of barrier invariant instantiation to yield a *quantifier-free* verification method (Section 4.3.4).

4.3.1. Syntax

Figure 4.6 gives the syntax for a simple kernel programming language with barrier invariants. The language is an extension of prior work [CDK+13] that formalised barrier invariants for straight-line GPU kernels without consideration for conditionals or loops.

A kernel consists of a declaration of the number of threads that will execute (**threads**: number) and a (possibly compound) statement, which is the body of the kernel. The set name denotes the local variables of a kernel and includes a reserved read-only variable *tid*,

kernel	::=	**threads**: number;
		main: stmt;
stmt	::=	basic_stmt
	|	stmt; stmt
	|	$\textbf{barrier}_{\text{iexpr}}$
	|	**if** (expr) **then** {stmt} **else** {stmt}
	|	**while** (expr) **do** {stmt}
basic_stmt	::=	name := expr | name := sh[expr] | sh[expr] := expr
expr	::=	constant literal | name | expr *op* expr
name	::=	thread identifier *tid* | any valid C name
iexpr	::=	constant literal | sh[iexpr] | name | $\overline{\text{name}}$ | iexpr *op* iexpr

Figure 4.6.: Syntax for a kernel programming language with barrier invariants

the unique identifier of each thread. Kernel statements allow assignment to local variables, access to shared memory (sh) and barrier synchronisation. Statements can be combined sequentially and also in conditionals and loops. The syntax of expressions does not allow referencing of shared memory. This ensures that there is at most one read or write per statement and no reads of shared memory can occur in the guards of conditionals or loops. A kernel without these restrictions can easily be preprocessed into this form.

Each **barrier** statement is annotated with a barrier invariant $\varphi \in \mathsf{iexpr}$. The syntax of barrier invariants extends the expr syntax to cater for references to shared memory. A barrier invariant is implicitly quantified over all pairs of distinct threads.[3] Given a local variable $v \in \mathsf{name}$ we use the notation v or $\overline{v}$ to refer, respectively, to the v of the first or second thread of the pair. For example, if x is a local variable then the invariant expression $sh[x + tid] = sh[\overline{x} + \overline{tid}]$ can be read as $\forall s \neq t : sh[x_s + s] = sh[x_t + t]$ where x_s and x_t refer to the local variable x of threads s and t, respectively, and where tid is replaced by s and $\overline{tid}$ by t. Together, this means that the syntax of barrier invariants allows us to state properties about shared memory from the point of view of an arbitrary pair of threads.

4.3.2. Semantics

Let P be a kernel executed by n threads and Word be the type of memory words. We assume that a value of type Word can be interpreted as a bit-vector or Boolean. A *thread state* σ for P is a tuple:

$$(sh, l, R, W) \in \mathsf{ThreadStates}$$

[3] As in prior work [CDK+13], we present barrier invariants with respect to a two-thread reduction. There is no conceptual difficulty with generalising barrier invariants to larger thread reductions. However, there would be a notation burden (to refer to each thread version of a local variable) and we have not found a practical need for barrier invariants involving more than two threads simultaneously to prove data race-freedom of a kernel.

where $sh : \mathbb{N} \to \mathsf{Word}$ is the shared memory of the kernel; $l : \mathsf{name} \to \mathsf{Word}$ is the thread local store; and $R, W \subseteq \mathbb{N}$ are *read* and *write* sets recording the locations in shared memory that the thread has accessed since the last barrier. The read and write sets are used for race checking.

A *predicated statement* is a pair (s, p) where $s \in \mathsf{stmt}$ and $p \in \mathsf{expr}$. Intuitively, (s, p) denotes a statement s that should be executed if the predicate p holds and has no effect otherwise. The set of all predicated statements is denoted $\mathsf{PredStmts}$.

The rules of Figure 4.7 define a binary relation

$$\to \subseteq (\mathsf{ThreadStates} \times \mathsf{PredStmts}) \times \mathsf{ThreadStates}$$

describing the evolution of a single thread state whilst executing a predicated basic statement. Similar to earlier work [BCD+12] we combine each thread state with a predicated statement in anticipation of our lock-step semantics. Under lock-step semantics, multiple threads execute the same statement of a kernel at the same time. Predication caters for the fact that distinct threads may have divergent control-flow. For readability, given a thread state (sh, l, R, W) and a predicated statement (s, p) we write $((sh, l, R, W), s, p)$ rather than $((sh, l, R, W), (s, p))$.

The evaluation of an expression e with respect to a local store l is denoted e^l and defined inductively.

$$\begin{aligned} \text{constant literal}^l &= \text{constant literal} \\ \mathsf{name}^l &= l(\mathsf{name}) \\ (\mathsf{expr}_1 \mathit{op}\ \mathsf{expr}_2)^l &= \mathsf{expr}_1^l\ \mathit{op}\ \mathsf{expr}_2^l \end{aligned}$$

The evaluation of a predicate p (a Boolean expression) is written as p^l.

The rule T-Disabled ensures that a predicated statement has no effect if the predicate p does not hold (i.e., $\neg p^l$). The remaining rules for local store and shared state updates are defined in a standard manner: T-Assign updates the local store l according to the assignment (the evaluation of the expression e); T-Read updates the local store with a value from shared state and records the location read-from in R; and T-Write updates shared memory according to the assignment and records the location written to in W.

A *group state* Σ for P is a tuple:

$$(sh, (l_0, R_0, W_0), \ldots (l_{n-1}, R_{n-1}, W_{n-1}))$$

T-Disabled
$$\frac{\neg p^l}{((sh, l, R, W), \mathsf{basic\_stmt}, p) \to (sh, l, R, W)}$$

T-Assign
$$\frac{p^l \qquad l' = l[v \mapsto e^l]}{((sh, l, R, W), v := e, p) \to (sh, l', R, W)}$$

T-Read
$$\frac{p^l \qquad l' = l[v \mapsto sh(e^l)] \qquad R' = R \cup \{e^l\}}{((sh, l, R, W), v := sh[e], p) \to (sh, l', R', W)}$$

T-Write
$$\frac{p^l \qquad sh' = sh[e_1^l \mapsto e_2^l] \qquad W' = W \cup \{e_1^l\}}{((sh, l, R, W), sh[e_1] := e_2, p) \to (sh', l, R, W')}$$

Figure 4.7.: Rules for predicated thread execution of basic statements

where $l_t(tid) = t$ for all threads $0 \leq t < n$; and sh is the shared memory of the kernel; and (sh, l_t, R_t, W_t) is the thread state of thread t.

We refer to the shared memory of a group state Σ as $\Sigma.sh$. We write $\Sigma(t)$ to refer to the thread state (sh, l_t, R_t, W_t) of thread t. We also write $\Sigma(t).l$, $\Sigma(t).R$ and $\Sigma(t).W$ to refer to the thread-specific components of this thread state.

A *kernel state* is a tuple (Σ, ss) where Σ is a group state and ss is an ordered sequence of predicated statements to be executed by the kernel. The set of all kernel states is denoted $\mathsf{KernelStates}$. A kernel state (Σ, ss) is a valid *initial state* if $\Sigma(t).R = \Sigma(t).W = \emptyset$ for all threads $0 \leq t < n$ and $ss = \langle (s, \mathsf{true}) \rangle$ where s is the body of the kernel (declared using **main** : s).

The rules of Figure 4.8 define a binary relation

$$\to_k \subseteq \mathsf{KernelStates} \times (\mathsf{KernelStates} \cup \{error\})$$

where *error* is a designated error state. We overload the operator @ to denote prepending a single predicated statement to the front of a sequence of predicated statements, $s@ss$, and concatenation of two predicated statement sequences, $ss@ss'$. The semantics define a *lock-step* execution of the kernel: all threads execute a statement of the kernel before proceeding to the next statement. In rules K-Race and K-Step, $\sigma_0, \ldots, \sigma_{n-1}$ are the thread states reached by each thread after executing the given predicated statement using the rules of Figure 4.7. The predicate race checks whether the execution of this basic

statement would cause a data race using the read and write sets of the threads:

$$\mathsf{race}(\sigma_0, \ldots, \sigma_{n-1}) \triangleq \exists\, 0 \leq s \neq t < n : (\sigma_s.R \cup \sigma_s.W) \cap \sigma_t.W \neq \emptyset\,.$$

That is, if there exist two threads where the reads or writes of the first thread s conflict with the writes of the second thread t. If a race occurs then K-Race causes execution to go to the designated error state *error*. Otherwise, the rule K-Step allows the kernel to transition to a new kernel state where the local store and read/write sets for thread t are taken from thread state σ_t, and where the shared state is derived by merging the shared state associated with each thread state σ_t according to the write sets:

$$\mathsf{merge}(\sigma_0, \ldots, \sigma_{n-1})(z) \triangleq \begin{cases} \sigma_t.sh(z) & \text{if } z \in \sigma_t.W \\ \sigma_0.sh(z) & \text{otherwise.} \end{cases}$$

Since K-Step requires that no race has occurred there is at most one t with $z \in \sigma_t.W$. That is, merge always yields a unique successor shared state.

On reaching a barrier, K-Bar-Diverge and K-Bar-Nop check whether the barrier has been reached under divergent (there exist two threads with the first disabled and the other enabled) or uniformly vacuous (all threads are disabled) control-flow, respectively. Divergent control-flow leads to *error*, whereas uniformly vacuous control flow results in a no-op. In the case where control flow is not uniformly vacuous (i.e., all threads are enabled) the rules K-Bar-Err and K-Bar-Inv check whether the associated barrier invariant φ holds for all pairs of distinct threads. The valuation of a barrier invariant φ with respect to the group state Σ and distinct threads s and t is denoted $[\![\varphi]\!]^{s,t}_{\sigma}$ and defined inductively.

$$\begin{aligned}
[\![\text{constant literal}]\!]^{s,t}_{\Sigma} &= \text{constant literal} \\
[\![sh[\mathsf{iexpr}]]\!]^{s,t}_{\Sigma} &= \Sigma.sh([\![\mathsf{iexpr}]\!]^{s,t}_{\Sigma}) \\
[\![\mathsf{name}]\!]^{s,t}_{\Sigma} &= \Sigma(s).l(\mathsf{name}) \\
[\![\overline{\mathsf{name}}]\!]^{s,t}_{\Sigma} &= \Sigma(t).l(\mathsf{name}) \\
[\![\mathsf{iexpr}_1\, op\, \mathsf{iexpr}_2]\!]^{s,t}_{\Sigma} &= [\![\mathsf{iexpr}_1]\!]^{s,t}_{\Sigma}\ op\ [\![\mathsf{iexpr}_2]\!]^{s,t}_{\Sigma}
\end{aligned}$$

Note that a variable v occurring as v, respectively $\overline{v}$, is evaluated in the context of thread s, respectively t. If the invariant does not hold for all pairs of distinct threads then the execution goes to *error* using K-Bar-Err. Otherwise, rule K-Bar-Inv resets the read and write sets for all threads and allows execution to proceed beyond the barrier.

The remaining rules are for execution of compound statements. Rule K-Seq is straight-

K-Race

$$\frac{\forall\, 0 \le t < n : (\Sigma(t), \mathsf{basic\_stmt}, p) \to \sigma_t \qquad \mathsf{race}(\sigma_0, \ldots, \sigma_{n-1})}{(\Sigma, (\mathsf{basic\_stmt}, p)@ss) \to_k \mathit{error}}$$

K-Step

$$\frac{\forall\, 0 \le t < n : (\Sigma(t), \mathsf{basic\_stmt}, p) \to \sigma_t \qquad \neg\mathsf{race}(\sigma_0, \ldots, \sigma_{n-1}) \quad sh' = \mathsf{merge}(\sigma_0, \ldots, \sigma_{n-1}) \qquad \forall\, 0 \le t < n : \Sigma'(t) = (sh', \sigma_t.l, \sigma_t.R, \sigma_t.W)}{(\Sigma, (\mathsf{basic\_stmt}, p)@ss) \to_k (\Sigma', ss)}$$

K-Bar-Diverge

$$\frac{\exists\, 0 \le s \ne t < n : p^{\Sigma(s).l} \wedge \neg p^{\Sigma(t).l}}{(\Sigma, (\mathbf{barrier}_\varphi, p)@ss) \to_k \mathit{error}}$$

K-Bar-Err

$$\frac{\forall\, 0 \le t < n : p^{\Sigma(t).l} \qquad \exists\, 0 \le s \ne t < n : \neg[\![\varphi]\!]_\Sigma^{s,t}}{(\Sigma, (\mathbf{barrier}_\varphi, p)@ss) \to_k \mathit{error}}$$

K-Bar-Nop

$$\frac{\forall\, 0 \le t < n : \neg p^{\Sigma(t).l}}{(\Sigma, (\mathbf{barrier}_\varphi, p)@ss) \to_k (\Sigma, ss)}$$

K-Bar-Inv

$$\frac{\forall\, 0 \le t < n : p^{\Sigma(t).l} \qquad \forall\, 0 \le s \ne t < n : [\![\varphi]\!]_\Sigma^{s,t} \qquad \forall\, 0 \le t < n : \Sigma'(t) = (\Sigma.sh, \Sigma(t).l, \emptyset, \emptyset)}{(\Sigma, (\mathbf{barrier}_\varphi, p)@ss) \to_k (\Sigma', ss)}$$

K-Seq

$$(\Sigma, (S_1; S_2, p)@ss) \to_k (\Sigma, (S_1, p)@(S_2, p)@ss)$$

K-Cond

$$\frac{v \text{ is fresh}}{(\Sigma, (\mathbf{if}\ (e)\ \mathbf{then}\ \{S_1\}\ \mathbf{else}\ \{S_2\}, p)@ss) \to_k (\Sigma, (v := e, p)@(S_1, p \wedge v)@(S_2, p \wedge \neg v)@ss)}$$

K-Loop

$$\frac{\exists\, 0 \le t < n : (p \wedge e)^{\Sigma(t).l} \qquad v \text{ is fresh}}{(\Sigma, (\mathbf{while}\ (e)\ \mathbf{do}\ \{S\}, p)@ss) \to_k (\Sigma, (v := e, p)@(S, p \wedge v)@(\mathbf{while}\ (e)\ \mathbf{do}\ \{S\}, p)@ss)}$$

K-Done

$$\frac{\forall\, 0 \le t < n : \neg(p \wedge e)^{\Sigma(t).l}}{(\Sigma, (\mathbf{while}\ (e)\ \mathbf{do}\ \{S\}, p)@ss) \to_k (\Sigma, ss)}$$

Figure 4.8.: Rules for concrete lock-step semantics

forward. The rule K-Cond decomposes a conditional statement into a sequence of predicated statements. The condition guard e is evaluated by each thread in a fresh local variable v, the **then** branch S_1 is executed by all threads under the predicate $p \wedge v$ (where p is the initial predicate on entry to the conditional), and the **else** branch S_2 is executed by all threads under the predicate $p \wedge \neg v$. The rule K-Loop and K-Done handle loops. A loop is executed if the loop guard e holds for some thread t. Threads that are not enabled for the loop execute no-ops. The rule K-Loop unpeels an iteration of the loop: the loop guard is evaluated in a fresh local variable v and the loop body is executed under the predicate $p \wedge v$. The rule ensures that we the kernel keeps looping as long as there exists a thread that needs to loop. The loop exits using K-Done when all threads evaluate the loop guard as false.

Abstract Semantics We now proceed to the abstract semantics which describes the execution of a kernel P with respect to a pair of distinct threads (s, t). We refer to (s, t) as the pair of threads under consideration. We begin by giving the informal intuition of our abstract semantics before describing them formally. The threads behave as if they are the *only* threads executing the kernel: they perform local updates, access shared state and record all shared locations to their respective read and write sets. If a data race or barrier divergence occurs then the execution aborts.

On reaching a barrier, A-Bar-Diverge and A-Bar-Nop check whether the barrier has been reached under divergent (one thread is disabled with the other enabled) or uniformly vacuously (both threads disabled) control-flow. When both threads are enabled — i.e., A-Bar-Inv or A-Bar-Err — a check determines whether the associated barrier invariant φ holds. This check must take into account updates to shared memory by additional threads not considered by the two-thread reduction. We handle this by considering whether a shared memory location v was accessed by s or t since the last barrier. There are two key cases:

- v **was accessed by at least one of** s **or** t: We say v is a *known* location. Moreover it is sound to assume that v has not been modified by any other thread $r \notin \{s, t\}$ because the two-thread reduction considers *all possible pairs of distinct threads*. In particular, if r did write to v then because some $x \in \{s, t\}$ accesses v a data race will be detected by the pair (x, r) and the kernel will not be deemed correct.

- v **was not accessed by** s **or** t: We say v is an *unknown* location. In this case, v may have been modified by some thread $r \notin \{s, t\}$. A safe approximation is to assume that v contains an *arbitrary* value when evaluating the barrier invariant.

If there exists an assignment to the unknown locations yielding a state in which φ does not hold for (s,t) then execution aborts. This is because there may be a concrete kernel state that matches this state.

On the other hand, if φ holds for the pair (s,t) under all assignments to unknown locations then, because (s,t) is arbitrary, it is sound to assume φ holds for *all* pairs of distinct threads. Execution continues by transitioning to an abstract state in which:

- the local stores for s and t and the values of *known* locations in shared memory are preserved;
- the read and write sets for s and t are cleared; and
- the barrier invariant φ is assumed to hold for all pairs of distinct threads.

More formally, an *abstract group state* T for P with respect to threads s and t is a tuple:

$$(sh, (l_s, R_s, W_s), (l_t, R_t, W_t)) .$$

It is similar to a (concrete) group state except that only the states of threads s and t are represented. We use $T(s)$ and $T(t)$ to denote the thread states of s and t. We write $\mathsf{known}^{s,t}(T)$ for the set of shared locations collectively accessed by s and t since the last barrier:

$$\mathsf{known}^{s,t}(T) \triangleq T(s).R \cup T(s).W \cup T(t).R \cup T(t).W .$$

The rules of Figure 4.9 define the evolution of abstract group states whilst executing a sequence of predicated statements. The majority of rules are a straightforward restriction of their concrete counterpart to an arbitrary pair of threads. That is, we remove quantifiers over all threads ($0 \leq t < n$) and replace the premises with explicit instances for threads s and t. The main differences are highlighted and correspond to the rules concerned with barrier invariants. The merge function restricted to two threads is:

$$\mathsf{merge}(\sigma_s, \sigma_t)(z) \triangleq \begin{cases} \sigma_t.sh(z) & \text{if } z \in \sigma_t.W \\ \sigma_s.sh(z) & \text{otherwise.} \end{cases}$$

On reaching a barrier where the predicate is enabled uniformly, the rules A-Bar-Err and A-Bar-Inv check whether the associated barrier invariant φ holds in all concrete states that agree with the current abstract state on the local stores of s and t and on the values of *known* locations. Formally, we define a concretisation operator $\gamma^{s,t}$ yielding the

set of such concrete group states given an abstract group state.

$$\gamma^{s,t}(T) \triangleq \{\Sigma \text{ a concrete group state} \mid \begin{array}{l}\Sigma(s).l = T(s).l \wedge \\ \Sigma(t).l = T(t).l \wedge \\ \bigwedge_{v \in \mathsf{known}^{s,t}(T)} (\Sigma.sh(v) = T.sh(v))\}\end{array}$$

Rule A-Bar-Err aborts if there exists a concrete state $\Sigma \in \gamma^{s,t}(T)$ such that the barrier invariant does not hold with respect to Σ for the pair (s,t). If the barrier invariant holds for (s,t) in *every* state of $\gamma^{s,t}(T)$ then execution proceeds past the barrier by rule A-Bar-Inv. A concrete state $\Sigma' \in \gamma^{s,t}(T)$ is chosen arbitrarily such that the barrier invariant φ holds in Σ' for *every* pair of distinct threads (x,y). Any successor state satisfying this condition is considered by the rule. The successor abstract state T' is obtained by projecting Σ' with respect to threads s and t while emptying the read/write sets of s and t and discarding the local stores and read/write sets of all other threads. We define an abstraction operator $\alpha^{s,t}$.

$$\alpha^{s,t}(\Sigma) \triangleq (\Sigma.sh, (\Sigma(s).l, \emptyset, \emptyset), (\Sigma(t).l, \emptyset, \emptyset)) \,.$$

Necessity of Known Locations We now demonstrate why it is necessary to consider *known* locations when checking barrier invariants. Consider a version of the rule A-Bar-Inv, which uses a concretisation operator that does not restrict shared state to known locations. Instead, concretised states use the abstract shared memory directly.

$$\overline{\gamma}^{s,t}(T) \triangleq \{\Sigma \text{ a concrete group state} \mid \begin{array}{l}\Sigma(s).l = T(s).l \wedge \\ \Sigma(t).l = T(t).l \wedge \\ \Sigma.sh = T.sh\}\end{array}$$

This version of the abstract rules is unsound. Consider the following incorrect kernel where `A` is an array in shared memory.[4]

```
A[tid] = 0;
barrier(); // φ1
A[tid] = tid + 1;
barrier(); // φ2
```

The first barrier invariant $\varphi_1 \triangleq \mathtt{A}[tid] = 0$ establishes that all elements of `A` are zero. Then each thread updates its corresponding element by $tid + 1$. For example, if the kernel is executed with 4 threads then $\mathtt{A} = [1,2,3,4]$ and, in particular, $\mathtt{A}[0] \neq 0$ when the second barrier is reached. The second barrier invariant $\varphi_2 \triangleq tid \neq 0 \wedge \overline{tid} \neq 0 \Rightarrow \mathtt{A}[0] = 0$

[4] Thanks to Jeroen Ketema for this counterexample.

A-Race
$$\frac{(T(s), \mathsf{basic\_stmt}) \to \sigma_s \qquad (T(t), \mathsf{basic\_stmt}) \to \sigma_t \qquad \mathsf{race}(\sigma_s, \sigma_t)}{(T, (\mathsf{basic\_stmt}, p)@ss) \to_k \mathit{error}}$$

A-Step
$$\frac{(T(s), \mathsf{basic\_stmt}) \to \sigma_s \qquad (T(t), \mathsf{basic\_stmt}) \to \sigma_t \qquad \neg\mathsf{race}(\sigma_s, \sigma_t) \qquad sh' = \mathsf{merge}(\sigma_s, \sigma_t) \qquad T' = (sh', (\sigma_s.l, \sigma_s.R, \sigma_s.W), (\sigma_t.l, \sigma_t.R, \sigma_t.W))}{(T, (\mathsf{basic\_stmt}, p)@ss) \to_k (T', ss)}$$

A-Bar-Diverge
$$\frac{p^{T(s).l} \wedge \neg p^{T(t).l}}{(T, (\mathbf{barrier}_\varphi, p)@ss) \to_k \mathit{error}}$$

A-Bar-Err
$$\frac{p^{T(s).l} \wedge p^{T(t).l} \qquad \exists\, \Sigma \in \gamma^{s,t}(T) : \neg[\![\varphi]\!]^{s,t}_{\Sigma}}{(T, (\mathbf{barrier}_\varphi, p)@ss) \to_k \mathit{error}}$$

A-Bar-Nop
$$\frac{\neg p^{T(s).l} \wedge \neg p^{T(t).l}}{(T, (\mathbf{barrier}_\varphi, p)@ss) \to_k (T, ss)}$$

A-Bar-Inv
$$\frac{p^{T(s).l} \wedge p^{T(t).l} \qquad \forall\, \Sigma \in \gamma^{s,t}(T) : [\![\varphi]\!]^{s,t}_{\Sigma} \qquad \Sigma' \in \gamma^{s,t}(T) \qquad \forall\, 0 \le x \neq y < n : [\![\varphi]\!]^{x,y}_{\Sigma'} \qquad T' = \alpha^{s,t}(\Sigma')}{(T, (\mathbf{barrier}_\varphi, p)@ss) \to_k (T', ss)}$$

A-Seq
$$(T, (S_1; S_2, p)@ss) \to_k (T, (S_1, p)@(S_2, p)@ss)$$

A-Cond
$$\frac{v \text{ is fresh}}{(T, (\mathbf{if}\ (e)\ \mathbf{then}\ \{S_1\}\ \mathbf{else}\ \{S_2\}, p)@ss) \to_k (T, (v := e, p)@(S_1, p \wedge v)@(S_2, p \wedge \neg v)@ss)}$$

A-Loop
$$\frac{(p \wedge e)^{T(s).l} \vee (p \wedge e)^{T(t).l} \qquad v \text{ is fresh}}{(T, (\mathbf{while}\ (e)\ \mathbf{do}\ \{S\}, p)@ss) \to_k (T, (v := e, p)@(S, p \wedge v)@(\mathbf{while}\ (e)\ \mathbf{do}\ \{S\}, p)@ss)}$$

A-Done
$$\frac{\neg(p \wedge e)^{T(s).l} \wedge \neg(p \wedge e)^{T(t).l}}{(T, (\mathbf{while}\ (e)\ \mathbf{do}\ \{S\}, p)@ss) \to_k (T, ss)}$$

Figure 4.9.: Rules for abstract two-threaded semantics

incorrectly asserts that if the pair of threads under consideration does not include thread 0 then $\mathtt{A}[0] = 0$. In a concrete execution φ_2 is false.

However, consider an abstract execution where the threads under consideration do not include thread 0 and we use the erroneous concretisation operator $\overline{\gamma}^{s,t}$ in A-Bar-Inv. After the first barrier the invariant φ_1 is assumed for all pairs of distinct threads. In particular, the shared state satisfies $\mathtt{A}[0] = 0$. Then the two threads under consideration update their respective elements (which will not include element 0). At the second barrier, φ_2 is checked under all concrete states that satisfy the erroneous concretisation operator. Since this operator preserves all locations in shared state without distinguishing between known and unknown locations we have $\mathtt{A}[0] = 0$, which does not reflect the true state of shared memory, and thus φ_2 holds. Intuitively, the problem is that the barrier invariant reasons about locations of shared state that the thread has not accessed. In particular, the location $\mathtt{A}[0]$ is unknown if the threads under consideration do not include thread 0. Thus checking a barrier invariant with respect to known locations is critical for soundness.

4.3.3. Soundness

Theorem 4.1 (Soundness)**.** *Let P be a kernel executed by n threads. If for every pair of distinct threads $0 \leq s \neq t < n$, no execution of the kernel P with respect to the abstract semantics (denoted $\mathcal{A}^{s,t}(P)$) from a valid initial state leads to error, then no execution of P from a valid initial state leads to error.*

Before presenting the proof we define a projection operator that takes a concrete kernel state and a pair of threads (s, t) and returns a set of abstract kernel states.

$$\pi^{s,t}(\Sigma, ss) \triangleq \{((sh', \pi^s(\Sigma), \pi^t(\Sigma)), tt) \mid \forall\, v \in \mathsf{known}^{s,t}(\Sigma) : sh'(v) = \Sigma.sh(v) \wedge \exists\, tt' \in \mathsf{PredStmts} : ss = tt' @ tt\}$$

where $\pi^x(\Sigma) \triangleq (\Sigma(x).l, \Sigma(x).R, \Sigma(x).W)$ and $\mathsf{known}^{s,t}(\Sigma) \triangleq \Sigma(s).R \cup \Sigma(s).W \cup \Sigma(t).R \cup \Sigma(t).W$. Two properties hold for every abstract kernel state (T, tt) in $\pi^{s,t}(\Sigma, ss)$: (i) all *known* locations are preserved between $\Sigma.sh$ and $T.sh$; and (ii) the abstract sequence of predicated statements tt is some suffix of the concrete sequence ss.

Proof. The proof is by contradiction. Therefore, assume that no execution of $\mathcal{A}^{s,t}(P)$ for any pair of threads s and t leads to *error*, but that P either has (i) a data race, (ii) barrier divergence or (iii) violates a barrier invariant. Hence, there is an execution of P from a valid initial state to a state where either (i) K-Race or (ii) K-Bar-Diverge or (iii)

K-Bar-Err applies. In all cases, observe that the rule applies due to a particular pair of threads, say s and t.

We will construct an abstract execution with respect to these threads that must lead to *error*: thus yielding a contradiction. Suppose $\rho = (\Sigma_1, ss_1), (\Sigma_2, ss_2), \ldots, (\Sigma_n, ss_n)$ is the sequence of kernel states successively assumed by P during the erroneous execution excluding the final K-Race or K-Bar-Diverge or K-Bar-Err step. We will show by induction that there exists a sequence $\pi(\rho) = A_1, A_2, \ldots, A_n$ with $A_i \in \pi^{s,t}(\Sigma_i, ss_i)$ for all $0 \leq i \leq n$ such that this sequence is stuttering equivalent to an execution of $\mathcal{A}^{s,t}(P)$ that leads to *error*. That is, the concrete rules K-Step, K-Bar-Nop, K-Bar-Inv, K-Seq, K-Cond, K-Loop, K-Done are replaced by A-Step, A-Bar-Nop, A-Bar-Inv, A-Seq, A-Cond, A-Loop, A-Done, respectively. We allow stuttering due to the handling of loops in our proof; ignoring them, temporarily, a proof sketch is:

$$\begin{array}{rccccc} \text{Concrete trace } \rho & (\Sigma_1, ss_1) & \xrightarrow{\text{K-Rule}}_k & (\Sigma_2, ss_2) & \rightarrow_k \cdots & \rightarrow_k (\Sigma_n, ss_n) \\ & \Downarrow & & \Downarrow & & \Downarrow \\ \text{Abstract trace } \pi(\rho) & (T_1, tt_1) & \xrightarrow{\text{A-Rule}}_k & (T_2, tt_2) & \rightarrow_k \cdots & \rightarrow_k (T_n, tt_n) \end{array}$$

where we use $\Downarrow$ to denote the projection and K-Rule and A-Rule to denote a matching pair of rules in the concrete and abstract trace. Because we have chosen s and t to be the pair of threads that cause the error in (Σ_n, ss_n) it follows that either (i) A-Race or (ii) A-Bar-Diverge or (iii) A-Bar-Err applies to some $(T_n, tt_n) \in \pi^{s,t}(\Sigma_n, ss_n)$ and so contradicts our initial assumption that no execution of $\mathcal{A}^{s,t}(P)$ leads to *error*.

The base case of the induction is straightforward. If (Σ_1, ss_1) is a valid (concrete) initial state then any $(T_1, ss_1) \in \pi^{s,t}(\Sigma_1, ss_1)$ where we pick the same sequence of predicated statements ss_1 is also a valid abstract initial state.

For the step case we consider each non-error yielding rule in turn. We show that if $(\Sigma_i, ss_i) \rightarrow_k (\Sigma_{i+1}, ss_{i+1})$ then either (i) the abstract trace follows exactly: there exists A_i and A_{i+1} such that $A_i \in \pi^{s,t}(\Sigma_i, ss_i)$ and $A_{i+1} \in \pi^{s,t}(\Sigma_{i+1}, ss_{i+1})$ and $A_i \rightarrow_k A_{i+1}$; or (ii) stuttering holds: there exists A_i such that $A_i \in \pi^{s,t}(\Sigma_i, ss_i)$ and $A_i \in \pi^{s,t}(\Sigma_{i+1}, ss_{i+1})$. In the following, let $(T, ss_i) \in \pi^{s,t}(\Sigma_i, ss_i)$ and $(T', ss_{i+1}) \in \pi^{s,t}(\Sigma_{i+1}, ss_{i+1})$ for the case where the abstract trace follows exactly (the sequence of predicated statements is preserved exactly by the projection).

- Case K-Step. The concrete transition can be matched by A-Step since there is no race between any pair of distinct threads, including s and t employed by the two-thread reduction. Furthermore, the state required by s and t in the abstract transition is completely captured by T. We have $T(s) = \Sigma_i(s)$ and $T(t) = \Sigma_i(t)$ by

the definition of $\pi^{s,t}$ and hence the next thread states σ_s and σ_t of T' match those of Σ_{i+1}.

- Case K-Bar-Nop. The concrete transition can be matched by A-Bar-Nop because the premise $\neg p^{\Sigma_i(t).l}$ for all threads $0 \leq t < n$ (from K-Bar-Nop) implies that the predicate will not hold for s and t under the abstract execution. That is, $\neg p^{T(s).l} \wedge \neg p^{T(t).l}$ holds.

- Case K-Bar-Inv. Due to the top-level assumption that the execution $\mathcal{A}^{s,t}(P)$ is error-free it must be the case that for all $\Sigma \in \gamma^{s,t}(T)$ that $[\![\varphi]\!]^{s,t}_{\Sigma}$ holds. In particular, $\Sigma_i \in \gamma^{s,t}(T)$ by the definition of $\gamma^{s,t}$ and $\pi^{s,t}$. It follows that T' is a valid choice of abstract state because Σ_{i+1} is simply Σ_i with the read and write sets of all threads cleared.

- Case K-Seq. Straightforward as there are no rule premises. That is, $T' = T'$.

- Case K-Cond. Similar to K-Seq.

- Case K-Loop. There are two subcases to consider:

  1. **At least one of s or t are enabled**. That is, either $(p \wedge e)^{T(s).l}$ or $(p \wedge e)^{T(t).l}$ holds. Then the concrete transition can be matched by A-Loop.

  2. **Both s or t are disabled**. Then there must exist some thread r distinct from s or t that is enabled (i.e., $(p \wedge e)^{\Sigma(r).l}$). In this case the concrete transition cannot be matched with an abstract transition. The threads s and t, employed by the two-thread reduction, are both disabled so only A-Done applies. The enabledness of thread r is not captured. In other words,

$$\begin{array}{ccc} (\Sigma_i, ss_i) & \xrightarrow{\text{K-Loop}}_k & (\Sigma_{i+1}, ss_{i+1}) \\ \Downarrow & & \\ (T_i, ss_i) & \xrightarrow{\text{A-Loop}}\!\!\!\!\!\!/\;\;\;_k & \end{array}$$

     We will show that there exists a natural number m such that the concrete successor state (Σ_{i+m}, ss_{i+m}) has $ss_{i+m} = ss_i$ and K-Done applies. Call this subsequence of concrete states $(\Sigma_i, ss_i), (\Sigma_{i+1}, ss_{i+1}), \ldots, (\Sigma_{i+m}, ss_{i+m})$ the *intermediate* states. We further show that (T_i, ss_i) is a member of the projection

of all intermediate states. In other words,

$$\begin{array}{ccccccc}
(\Sigma_i, ss_i) & \xrightarrow{\text{K-Done}}_k & (\Sigma_{i+1}, ss_{i+1}) & \rightarrow_k \cdots & \rightarrow_k (\Sigma_{i+m}, ss_i) & \xrightarrow{\text{K-Done}}_k \\
\Downarrow & & \Downarrow & & \Downarrow & \\
(T_i, ss_i) & \xrightarrow{\tau} & (T_i, ss_i) & \xrightarrow{\tau} \cdots \xrightarrow{\tau} & (T_i, ss_i) & \xrightarrow{\text{A-Done}}_k
\end{array}$$

where we use $\xrightarrow{\tau}$ to denote the stuttering of the abstract trace. The stuttering allows the concrete execution to continue until the correspondence between traces can resume.

There is always such a state (Σ_{i+m}, ss_i). First we note that an error cannot occur during this stuttering. The error (at Σ_n) is caused by threads s and t which we have explicitly chosen for the two-thread reduction. In the subsequent concrete execution of the loop (which cannot be followed by the abstract execution) these threads are both disabled. By inspection of the concrete rules, we see that it is not possible to error when the two threads causing the violation are both disabled. If the error did in fact occur within the loop execution then the choice of s and t would mean that at least one of s or t must be enabled (giving subcase 1). That is, the error occurs beyond the completion of the loop. Since the concrete execution is finite and ends in error it must be the case that the concrete execution continues until the loop completes.

Now we show that (T_i, ss_i) is a member of the projection of all intermediate state. Because the execution is error-free (and in particular, race-free) the remaining enabled threads are only at liberty to update *unknown* locations of shared memory. So the state required by s and t in the abstract transition is completely captured by T_i. Finally, every intermediate state has a sequence of predicated statements whose suffix is ss_i.

- Case K-Done. The concrete transition can be matched by A-Done because the premise $\neg(p \wedge e)^{\Sigma(t).l}$ for all threads $0 \leq t < n$ (from K-Done) implies that the guard will not hold for s and t under the abstract execution. □

4.3.4. Verification Method

We now outline a quantifier-free verification method based on the semantics and soundness result of the previous section. The verification method is an extension of the GPUVerify verification method [BCD+12] using race instrumentation and the two-thread reduction using shared state abstraction. We extend the shared state abstraction to use barrier in-

variants by encoding the rules A-Bar-Err and A-Bar-Inv. The main technical difficulty is eliminating the quantifiers used in these rules. The rules use quantifiers to (i) consider *all* concretisations of the current abstract state when checking the barrier invariant and (ii) selecting a successor state in which the barrier invariant holds for *all* pairs of distinct threads. We consider each of these quantifiers in turn.

Concretisation The role of the forall quantifier in $\forall\ \Sigma \in \gamma^{s,t}(T)$ is to ensure that the barrier invariant is checked with respect to all concretisations of the current abstract state T. The essential idea for eliminating the quantifier is to ensure that the barrier invariant is checked only with respect to known locations (preserved by the concretisation operator) and *independent* of unknown locations. We consider two methods for achieving this. The first method ensures that the truth of a barrier invariant is independent of unknown locations, whereas our second method ensures that all locations used by a barrier invariant are known.

Method 1, given in prior work [CDK+13], tracks unknown locations and translates the barrier invariant so that any reference to an unknown location yields an arbitrary value. A barrier invariant can only refer to a finite number of shared memory locations because iexpr is quantifier-free. Let d be the largest number of syntactically distinct shared memory expressions for any barrier invariant in a given kernel. We introduce d auxiliary variables, $u_1, \ldots, u_d$, which can track at most d *unknown* locations. At the start of kernel execution and after every barrier we *havoc* these variables (set each variable to a non-deterministic value) to reflect the fact that the known set is empty. After each access to a shared memory location computed from an expression e under predicate p we insert a statement $\texttt{assume}(p \Rightarrow \bigwedge_{1 \leq i \leq d} u_i \neq e)$. Implicitly, we have added the location computed from e to the known set. For a barrier invariant φ, let $e_1, \ldots, e_f$ be the syntactically distinct sub-expressions of φ that refer to the shared state, where $f \leq d$. For ease of explanation, assume for the moment that there is no nesting between these sub-expressions. We now substitute every occurrence of $sh[e_i]$ in φ with an *if-then-else* expression $\texttt{ite}(e_i = u_i, \star, sh[e_i])$. This evaluates to $sh[e_i]$ *unless* e_i is equal to the ith *unknown* location, in which case the evaluation is non-deterministic. The barrier invariant φ' is then checked for the pair of threads under consideration. The transformation ensures that the truth of a barrier invariant is independent of unknown shared locations. With nested sub-expressions the substitution is applied recursively.

Method 2 ensures that all shared memory locations in a barrier invariant are known and aborts if this is not the case. We introduce a new logging variable, `notKnown`, which implicitly models the known set. At the start of kernel execution and after every barrier we

havoc `notKnown` to reflect the fact that the known set is empty. After each shared memory access to a location computed from an expression e under predicate p we insert a statement $\texttt{assume}(p \Rightarrow \texttt{notKnown} \neq e)$. Implicitly, we have added the location to the known set. For a barrier invariant φ, let $e_1, \ldots, e_f$ be the syntactically distinct sub-expressions of φ that refer to the shared state and $q_1, \ldots, q_f$ be the path condition paired with each sub-expression. The path condition gives the condition under which its associated sub-expression is evaluated: given the syntax tree of the barrier invariant the path condition is the conjunction of conditions to reach the sub-expression. At the barrier, in addition to checking φ, we also check that $\bigwedge_{1 \leq i \leq f} q_i \Rightarrow \texttt{notKnown} \neq e_i$ holds. That is, we ensure that every sub-expression e (whose path condition is satisfied) is a member of the known set. The path condition allows us to deal with conditions within the barrier invariant.

We demonstrate the difference between these methods for three simple kernels where `A` is an array in shared memory and we assume the kernels are executed by n threads. Example (a) is taken from our discussion of the necessity of known locations and is incorrect; examples (b) and (c) are correct.

(a)

```
A[tid] = 0;
barrier(); // φ1
A[tid] = tid + 1;
barrier(); // φ2
```

(b)

```
A[tid] = tid;
barrier(); // φ3
A[n + 2*A[tid]] = tid;
barrier(); // φ4
```

(c)

```
if (tid % 2 == 0)
  A[2*tid] = 0;
else
  A[2*tid+1] = 1;
barrier(); // φ5
```

- $\varphi_1 \triangleq \texttt{A}[tid] = 0$

  This invariant has sub-expression $e_1 = tid$ with the path condition $q_1 = \texttt{true}$. Under method 1, this becomes $\texttt{ite}(tid = u_1, \star, \texttt{A}[tid]) = 0$. Method 2 keeps φ_1 but additionally checks that $\texttt{true} \Rightarrow \texttt{notKnown} \neq tid$ holds.

- $\varphi_2 \triangleq tid \neq 0 \wedge \overline{tid} \neq 0 \Rightarrow \texttt{A}[0] = 0$.

  This invariant incorrectly asserts that if the pair of threads under consideration does not include thread 0 then $\texttt{A}[0] = 0$. Under method 1, this becomes $tid \neq 0 \wedge \overline{tid} \neq 0 \Rightarrow \texttt{ite}(0 = u_1, \star, \texttt{A}[0]) = 0$. Method 2 keeps φ_2 but additionally checks that $\texttt{true} \Rightarrow \texttt{notKnown} \neq 0$ holds. Because location 0 is unknown (if the pair of threads under consideration does not include thread 0) both methods cause an assertion failure, as expected.

- $\varphi_3 \triangleq \texttt{A}[tid] = tid$

  Similar to the treatment of φ_1.

- $\varphi_4 \triangleq \texttt{A}[n + 2 \cdot \texttt{A}[tid]] = tid$

This invariant has nested sub-expressions $e_1 = n + 2 \cdot \mathtt{A}[tid]$ and $e_2 = tid$ with path conditions $q_1 = q_2 = \mathsf{true}$. Under method 1, this becomes $\mathtt{ite}((n + 2 \cdot \mathtt{ite}(tid = u_2, \star, \mathtt{A}[tid])) = u_1, \star, \mathtt{A}[n + 2 \cdot \mathtt{ite}(tid = u_2, \star, \mathtt{A}[tid])]) = tid$ by applying the substitution recursively. Method 2 keeps φ_4 but additionally checks that $\mathsf{true} \Rightarrow \mathtt{notKnown} \neq e_1 \wedge \mathsf{true} \Rightarrow \mathtt{notKnown} \neq e_2$. Because this barrier invariant only considers known locations both methods pass, as expected.

- $\varphi_5 \triangleq \mathtt{ite}(tid \bmod 2 = 0, \mathtt{A}[2 \cdot tid] = 0, \mathtt{A}[2 \cdot tid + 1] = 1)$

  This invariant has sub-expressions $e_1 = 2 \cdot tid$ and $e_2 = 2 \cdot tid + 1$ with path conditions $q_1 = (tid \bmod 2 = 0)$ and $q_2 = \neg q_1$. Under method 1, this becomes $\mathtt{ite}(tid \bmod 2 = 0, \mathtt{ite}(2 \cdot tid = u_1, \star, \mathtt{A}[2 \cdot tid]) = 0, \mathtt{ite}(2 \cdot tid + 1 = u_2, \star, \mathtt{A}[2 \cdot tid + 1]) = 1)$. Method 2 keeps φ_5 but additionally checks that $q_1 \Rightarrow \mathtt{notKnown} \neq e_1 \wedge q_2 \Rightarrow \mathtt{notKnown} \neq e_2$.

Successor State The role of the forall quantifier in $\forall\, 0 \leq x \neq y < n : [\![\varphi]\!]^{x,y}_{\Sigma'}$ is to choose a successor state where the barrier invariant holds for *all* pairs of distinct threads. This entails assuming the barrier invariant for a quadratic number of pairs. We observe that it is sound to weaken the assumption to only consider a *subset* of the pairs of distinct threads. Therefore, the quantifier can be eliminated by accompanying each barrier invariant with a set of *instantiation expressions* giving this subset explicitly. An instantiation expression can be a function of the pair of threads under consideration. For example, to assume a barrier invariant for the pair of threads under consideration and their immediate right neighbours under wrap around we specify the set $\{(tid, \overline{tid}), ((tid + 1) \bmod n, (\overline{tid} + 1) \bmod n)\}$.

In practice, we find that a small subset of distinct pairs is sufficient for verification to succeed. For example, the simple data-dependent kernel of Section 4.1, can be verified by assuming the barrier invariant for a single pair. In Section 4.4 we give instantiation expressions for each barrier invariant of the Blelloch prescan.

Barrier Invariants and Loop Invariants We extend the syntax of loop invariants to also refer to shared state in a similar fashion to barrier invariants (i.e., using expressions of type iexpr). In this case, the loop invariant is established with respect to the pair of threads under consideration. The essential difference between barrier invariants and loop invariants is the state of shared memory after the barrier or loop head. Following a barrier invariant, the successor state is free to assume the barrier invariant for all pairs of distinct threads. This is not the case for a loop invariant because threads are not guaranteed to see a consistent view of shared memory after reaching a loop head; only a barrier guarantees

this property. Hence, a loop invariant may only assume the invariant for the current pair of threads under consideration and not *all* pairs.

4.3.5. Relation to Equality Abstraction

We now give a correspondence result that relates the results of this chapter with prior work described in Chapter 2. We show that the abstract semantics where all barrier invariants are replaced with the vacuous assertion `true` yields a minor refinement of the equality abstraction. The rules for the abstract semantics under equality abstraction require a change to A-Bar-Inv, which becomes A-Bar-Eq. We give our refinement of the equality abstraction as the rule A-Bar-Known-Eq.

$$\text{A-Bar-Eq}\quad \frac{p^{T(s).l} \wedge p^{T(t).l} \qquad T' = (sh', (T(s).l, \emptyset, \emptyset), (T(t).l, \emptyset, \emptyset))}{(T, (\mathbf{barrier}_\varphi, p)@ss) \rightarrow_k (T', ss)}$$

$$\text{A-Bar-Known-Eq}\quad \frac{p^{T(s).l} \wedge p^{T(t).l} \qquad \forall\, v \in \mathsf{known}^{s,t}(T) : sh'(v) = T.sh(v) \qquad T' = (sh', (T(s).l, \emptyset, \emptyset), (T(t).l, \emptyset, \emptyset))}{(T, (\mathbf{barrier}_\varphi, p)@ss) \rightarrow_k (T', ss)}$$

The barrier invariant φ is ignored in both rules. Under A-Bar-Eq the sh' of the successor state T' is unconstrained [BCD+12]. Thus, each thread sees an arbitrary, but consistent, view of shared state after each barrier. The refinement given by A-Bar-Known-Eq additionally preserves the known locations of T across the barrier. We refer to this refinement as the *known-equality* abstraction. All other rules are imported directly except for the rule A-Bar-Err (since barrier invariants are ignored under the equality abstraction).

The examples (c) and (d) of Figure 4.1, reproduced below, show the difference between the equality and known-equality abstractions. Example (c) fails using the equality abstraction because shared state is set non-deterministically at the barrier. This example is correct using the known-equality abstraction because the known location `tid` is preserved across the barrier. Example (d) is similar except that the assertion checks the neighbouring element under wrap around (assume there are n threads). In this case, the example fails under the equality and known-equality abstractions but can be captured using a barrier invariant.

Theorem 4.2 (Known-Equality Equivalence)**.** *Let P be a kernel executed by n threads. Let P' be the kernel where every barrier invariant φ is replaced with* `true`*. The abstract semantics of P' is equivalent to the abstract semantics under the known-equality abstraction.*

(c)

```
A[tid] = tid;
barrier();
v = tid;
assert(A[v] == v);
```

(d)

```
A[tid] = tid;
barrier();
v = (tid+1)%n;
assert(A[v] == v);
```

Figure 4.10.: Examples with differing results using the equality and known-equality abstractions

Proof. The proof follows by showing that the rules A-Bar-Inv with $\varphi = \mathsf{true}$ and A-Bar-Known-Eq are substitutable. That is, any chosen subsequent state T' from either rule satisfies the other.

The subsequent state under A-Bar-Inv satisfies $T' = \alpha^{s,t}(\Sigma')$ where $\Sigma' \in \gamma^{s,t}(T)$. In particular, the known locations of T are preserved but all other locations are arbitrary. Additionally, because the barrier invariant is vacuous, the check $[\![\varphi]\!]^{x,y}_{\Sigma'}$ for all distinct threads x and y also holds vacuously. It follows that the abstraction using A-Bar-Inv yields exactly the same next state as A-Bar-Known-Eq. □

This gives us the following hierarchy of shared state abstractions, from least refined to most refined: adversarial, equality, known-equality and barrier invariants. Soundness is preserved from less refined abstractions to more refined abstractions. That is, if a kernel is correct with respect to a less refined abstraction (e.g., the adversarial abstraction) then the kernel is correct with respect to a more refined abstraction (e.g., the known-equality abstraction).

4.4. Barrier Invariants for Stream Compaction

We are now equipped to tackle the data-dependent stream compaction kernel of Figure 4.4. We use a staged and modular verification strategy. The heart of the kernel is the Blelloch prescan, which we outline into a separate procedure. We then prove that the Blelloch prescan is (i) data race-free and (ii) satisfies a monotonic specification using staged verification. Finally, we use this specification in the outer stream compaction kernel to prove race-freedom using modular verification. In this section, we focus our attention on proving the Blelloch prescan specification using barrier invariants. After introducing some notation we give a set of barrier invariants and instantiation expressions that allow us to capture the specification we require. We also discuss the formalisation of two further prefix sum implementations using barrier invariants.

Preliminaries For presentation purposes we separate the `idx` array used by both the upsweep and downsweep phases into two arrays: a `sum` array used in the tree reduction of the upsweep and a `prescan` array used by the downsweep that will contain the expected output of the prescan. This simplification can be avoided by introducing `sum` as a *ghost variable* (an auxiliary variable used only for verification): the upsweep and downsweep work over the same array and we take a snapshot of the array in-between the two loops and store it in `sum`.

The specification we will prove is that the output is monotonic: for all threads $s < t$, $\texttt{prescan}[s] + \texttt{flag}[s] \leq \texttt{prescan}[t]$.[5] For unsigned (i.e., non-negative) inputs we can use barrier invariants to prove this specification under the assumption that addition of unsigned integers does not overflow. Without this assumption the specification does not hold: for sufficiently large inputs, the additions performed during the prescan will overflow, leading to unexpected results in the `idx` array, and thus erroneous accesses to the `out` array. In applications, stream compaction is used under the implicit assumption that the inputs will not lead to overflow, thus our assumption is pragmatic.

As discussed in Section 4.2, although the prescan works in-place over the same `idx` array, we can view the upsweep and downsweep as working over a logical tree where we can identify each iteration of the upsweep and downsweep by the value of the `offset` variable, which gives the 'distance' between vertices at that depth of the tree. In Figure 4.11 we show that a vertex of the tree can be specified as an (element, offset)-coordinate (x, θ) using the predicate $\mathsf{isvertex}(x, \theta) \triangleq (x + 1) \bmod \theta = 0$. For example, $(5, 2)$ is a vertex as element $x = 5$ is updated when offset $\theta = 2$, while $(5, 4)$ is not a vertex. For each element x we also give the binary encoding, which can be interpreted as the path from the root to the element leaf vertex: use 0 to mean left and 1 to mean right and read from most-to-least significant bit. For example, the path from the root to the leaf vertex 6, with binary encoding `0b110`, is right (1), right (1) and left (0). We use the indexing functions $\mathsf{ai}(\mathit{tid}, \theta) \triangleq \theta(2 \cdot \mathit{tid} + 1) - 1$ and $\mathsf{bi}(\mathit{tid}, \theta) \triangleq \theta(2 \cdot \mathit{tid} + 2) - 1$, which give the indices of the left and right child vertices for a thread tid at offset θ. For example, $\mathsf{ai}(1, 2) = 5$ and $\mathsf{bi}(1, 2) = 7$ (see Figures 4.4(a) and (b)).

We note that the thread-private variables `d` and `offset` are used in both the upsweep and downsweep loops and additionally are uniform between threads: all threads agree on the value of these variables at each barrier. This property is encoded using the barrier invariant $\texttt{d} = \overline{\texttt{d}} \wedge \texttt{offset} = \overline{\texttt{offset}}$ and is important because the barrier invariants that follow use the `offset` variable as a measure of progress of the upsweep and downsweep.

[5] As discussed in Section 4.2, this is a weaker specification implied by the full functional specification (assuming given non-negative inputs) but is sufficient for showing race-freedom.

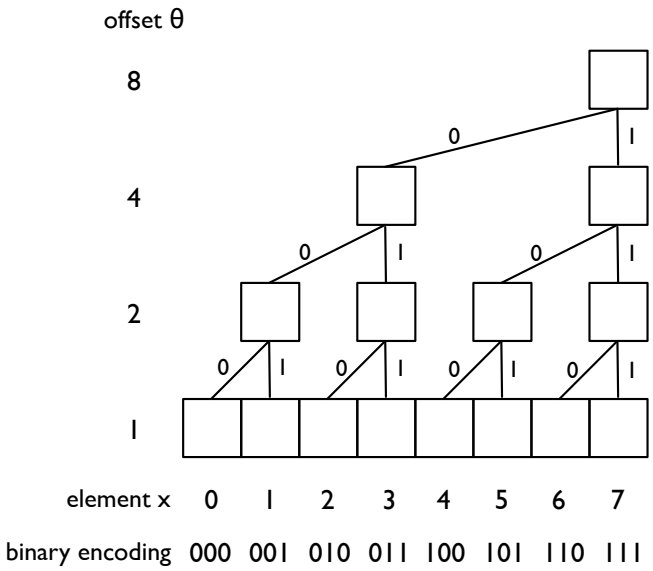


Figure 4.11.: Tree structure of the Blelloch algorithm for $n = 8$

GPUVerify infers such *uniform* invariants automatically using static analysis [BBC+14] (see Section 3.3.3). The variable `offset` is always a power-of-two and ranges from 1 to n in the upsweep loop and back from n to 1 in the downsweep loop.

4.4.1. Blelloch Prescan

We use barrier invariants to establish equalities between the shared arrays `flag`, `sum` and `prescan`: the φ_{us} invariant, in the upsweep loop, gives equalities between elements of `flag` and `sum`, while the φ_{ds} invariant, in the downsweep loop, gives equalities between elements of `sum` and `prescan`. Figure 4.12 gives the set of equalities given by the barrier invariants for $n = 8$ at the end of their respective loops. The monotonic specification, expressed as the invariant φ_{spec} for the final barrier, is then a combination of the relevant equalities. For example, consider the case $\texttt{prescan}[1] + \texttt{flag}[1] \leq \texttt{prescan}[5]$. By the downsweep equalities for $\texttt{prescan}[1]$ and $\texttt{prescan}[5]$ we rewrite this as $\texttt{sum}[0] + \texttt{flag}[1] \leq \texttt{sum}[3] + \texttt{sum}[4]$. Then, by the upsweep equalities and cancellations, this becomes $0 \leq \texttt{sum}[2] + \texttt{flag}[3] + \texttt{sum}[4]$, which holds given unsigned (and thus non-negative) inputs.

Load Invariant The stream compaction kernel is defined over n threads, however, only $n/2$ threads are required for the prescan. After a parallel test of each element using n threads, we synchronise with a barrier so $n/2$ threads can continue with the prescan.

	offset				
	1	2	4	8	
$\texttt{sum}[0] =$	$\texttt{flag}[0]$				$\texttt{prescan}[0] = e$
$\texttt{sum}[1] =$	$\texttt{flag}[1]$	$+\ \texttt{sum}[0]$			$\texttt{prescan}[1] = \texttt{sum}[0]$
$\texttt{sum}[2] =$	$\texttt{flag}[2]$				$\texttt{prescan}[2] = \texttt{sum}[1]$
$\texttt{sum}[3] =$	$\texttt{flag}[3]$	$+\ \texttt{sum}[2]$	$+\ \texttt{sum}[1]$		$\texttt{prescan}[3] = \texttt{sum}[1] + \texttt{sum}[2]$
$\texttt{sum}[4] =$	$\texttt{flag}[4]$				$\texttt{prescan}[4] = \texttt{sum}[3]$
$\texttt{sum}[5] =$	$\texttt{flag}[5]$	$+\ \texttt{sum}[4]$			$\texttt{prescan}[5] = \texttt{sum}[3] + \texttt{sum}[4]$
$\texttt{sum}[6] =$	$\texttt{flag}[6]$				$\texttt{prescan}[6] = \texttt{sum}[3] + \texttt{sum}[5]$
$\texttt{sum}[7] =$	$\texttt{flag}[7]$	$+\ \texttt{sum}[6]$	$+\ \texttt{sum}[5]$	$+\ \texttt{sum}[3]$	$\texttt{prescan}[7] = \texttt{sum}[3] + \texttt{sum}[5] + \texttt{sum}[6]$

Figure 4.12.: Upsweep and downsweep equalities for $n = 8$

There is no need to carry any property about shared state through this barrier, therefore, $\varphi_{\text{load}} \triangleq \mathsf{true}$.

Upsweep Invariant The upsweep is a reduction working from the leaves of a tree up to its root. At each iteration of the upsweep, each parent vertex is given the sum of its left and right child vertices. The upsweep proceeds until the root of the tree contains the full reduction and other elements form partial sums: each vertex of the tree will contain the sum of the leaf vertices below it in the tree.

In Figure 4.11 we show the tree structure for the concrete case where $n = 8$ and we shade right child vertices (for the moment, ignore the labels of each vertex, which will be used in the downsweep). The upsweep loop has the loop invariant:

$$(\texttt{d} = 4 \wedge \texttt{offset} = 1) \vee (\texttt{d} = 2 \wedge \texttt{offset} = 2) \vee (\texttt{d} = 1 \wedge \texttt{offset} = 4) \vee (\texttt{d} = 0 \wedge \texttt{offset} = 8)$$

where each conjunct characterises a loop iteration (and the last conjunct corresponds to the final time the loop head is evaluated and the loop exits). By considering the state of the array `sum` at the start of each iteration we can construct a barrier invariant φ_{us} for the barrier inside the loop body. The columns of the upsweep table in Figure 4.12 summarise the per-element equalities formed by the upsweep as the `offset` changes.

Informally, the summations of an element x at offset θ can be found by traversing the tree from the vertex (x, θ) to the leaf vertex $(x, 1)$ of the element: if a right child vertex (x, θ') is encountered the term $\texttt{sum}[x - \theta']$ is added to the summation. Furthermore, an index $x - \theta'$ is a summation term if and only if $\mathsf{isvertex}(x, 2\theta')$ holds. This gives us the following per-element invariant `upsweep`, which captures the state of any vertex (x, θ) of

the tree.

$$\mathsf{upsweep}(x,\theta) \triangleq \mathtt{sum}[x] = \mathtt{flag}[x] + \sum_{\substack{\theta' \in \{2^j \mid 0 \le j < \lg\theta\} \\ \mathsf{isvertex}(x,2\theta')}} \mathtt{sum}[x-\theta'] \,.$$

Using this per-element invariant, we define the barrier invariant φ_{us} by considering the elements of `sum` known to a thread *tid* in each iteration of the upsweep loop with offset θ. At loop entry, when `offset` $= 1$, each thread $tid < n/2$ knows about exactly two elements: $\mathsf{ai}(tid, 1)$ and $\mathsf{bi}(tid, 1)$. In subsequent loop iterations we have an uneven distribution of elements over threads as more threads become disabled while the upsweep proceeds. For each previous `offset` value $\theta'(< \theta)$, each thread $tid < n/\theta'$ will continue to know about its left child at that depth of the tree, i.e., $\mathsf{ai}(tid, \theta'/2)$, because this vertex will have no further summation terms. For the current `offset` value θ, each thread $tid < n/\theta$, i.e., each thread active in the iteration of the upsweep just completed, will know about its left and right vertices: $\mathsf{ai}(tid, \theta/2)$ and $\mathsf{bi}(tid, \theta/2)$. Thus φ_{us} is defined as

$$tid < n/2 \Rightarrow (\mathsf{upsweep}(\mathsf{ai}(tid,1),1) \wedge \mathsf{upsweep}(\mathsf{bi}(tid,1),1))$$

in the case $\theta = 1$, and as

$$\bigwedge_{\theta' \in \{2^i \mid 1 \le i \le \lg\theta\}} (tid < n/\theta' \Rightarrow \mathsf{upsweep}(\mathsf{ai}(tid,\theta'/2),\theta)) \\ \wedge \quad (tid < n/\theta \Rightarrow \mathsf{upsweep}(\mathsf{bi}(tid,\theta/2),\theta))$$

otherwise.

Downsweep Invariant The downsweep combines the partial sums formed in the upsweep to give the expected prescan result. For presentation purposes we show the downsweep operating over an array `prescan` different from the array `sum` used in the upsweep, where initially `prescan`$[x] =$ `sum`$[x]$ for all elements x.

The downsweep traverses the tree from the root to the leaves after clearing the root with the identity element e. At each iteration of the downsweep, each vertex (i) copies its value to its left child and (ii) sums its value with the old value of the left child into the right child. Consequently, a vertex is only summed into when it is a right child; otherwise, it receives the value of its parent vertex.

In Figure 4.13 we consider the concrete case where $n = 8$. We shade right child vertices and label each vertex with the summation terms of each vertex after the downsweep has processed that level of the tree. For example, the root $(7, 8)$ is labelled with e to denote

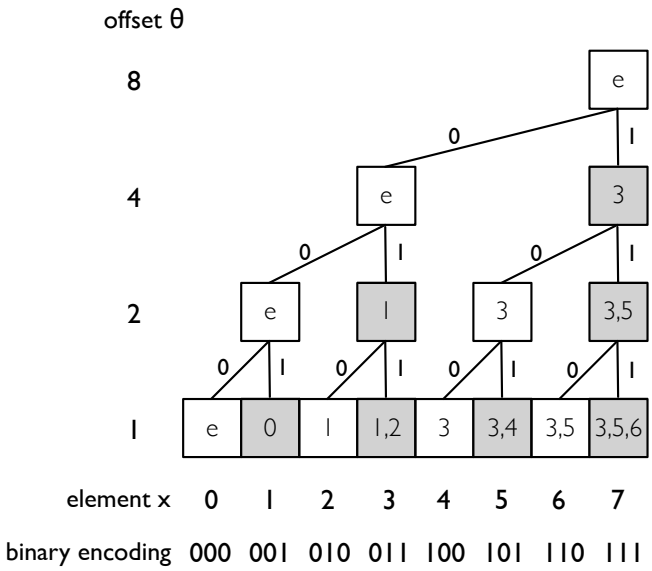


Figure 4.13.: Tree structure of the Blelloch downsweep for $n = 8$. We shade right child vertices. Each vertex (x, θ) is labelled with the summations formed as the downsweep proceeds, where $a, b, \ldots$ denotes $\texttt{prescan}[x] = \texttt{sum}[a] + \texttt{sum}[b] + \cdots$ and e denotes the identity.

that it is equal to the identity (i.e., $\texttt{prescan}[7] = e$), when $\texttt{offset} = 8$.

At the next downsweep iteration, when $\texttt{offset} = 4$, only $(7, 8)$ is a vertex. The vertex copies its value e into its left child $(3, 4)$ and sums this value with the previous value of its left child, i.e., $\texttt{sum}[7] + \texttt{sum}[3] = e + \texttt{sum}[3] = \texttt{sum}[3]$, storing the result in its right child $(7, 4)$. In the figure, we write this summation by labelling the vertex $(7, 4)$ with '3'.

In the next iteration, when $\texttt{offset} = 2$, vertex $(7, 4)$ copies its value $\texttt{sum}[3]$ into its left child $(5, 2)$ and sums this value with the previous value of its left child, i.e., $\texttt{prescan}[7] + \texttt{sum}[5] = \texttt{sum}[3] + \texttt{sum}[5]$ into its right child $(7, 2)$. Hence, in the figure, the left $(5, 2)$ and right $(7, 2)$ child vertices are respectively labelled '3' and '3, 5'. The remaining vertices are similarly computed.

Now consider the leaf vertices of the tree, which are the final output of the prescan. Note that the number of terms in the summation for an element x is the number of right child vertices in the path from the root vertex to the leaf vertex. For example, element 5 has two right child vertices on the path from the root vertex $(7, 8)$ to its leaf vertex.

Informally, the summations of an element x at offset θ can be found by traversing the tree from the vertex (x, θ) to the root and gathering terms: if a right child vertex (x', θ') is encountered then add the term $\texttt{sum}[x' - \theta']$ to the summation. This observation leads

to the following invariant, which defines the elements of `prescan` in terms of the offset θ:

$$\mathsf{downsweep}(x,\theta) \triangleq \texttt{prescan}[x] = \begin{cases} \sum_{\theta' \in B(x,\theta)} \texttt{sum}[y(x,\theta')] & \text{if } \mathsf{isvertex}(x, 2\theta) \\ \texttt{sum}[x] & \text{otherwise} \end{cases}$$

where $B(x,\theta) \triangleq \{2^i \mid \lg\theta \leq i < \lg n \wedge \mathsf{bit}(x,i) = 1\}$ and $y(x,\theta') \triangleq x - \theta' + \sum_{\substack{0 \leq j < \lg\theta' \\ \mathsf{bit}(x,j)=0}} 2^j$.

We exploit the fact that the binary encoding of x can be interpreted as the path from the root to the element leaf vertex (Figure 4.11). The function $B(x,\theta)$ considers the path from the root to the current offset ($\lg\theta \leq i < \lg n$) and returns the set of offsets where the path contains a right child vertex ($\mathsf{bit}(x,i) = 1$).

Finally, we define the invariant φ_{ds} by considering the elements of `prescan` accessed by the thread *tid* in each iteration of the downsweep loop with offset θ:

$$\begin{aligned} \varphi_{\text{ds}} \triangleq & \bigwedge_{0 \leq i < \lg n} \left(tid < n/2^{i+1} \wedge \theta \geq \lfloor 2^i/2 \rfloor \Rightarrow \mathsf{downsweep}(\mathsf{ai}(tid, 2^i), \theta)\right) \\ \wedge & \bigwedge_{0 \leq i < \lg n} \left(tid < n/2^{i+1} \wedge \theta \bowtie \lfloor 2^i/2 \rfloor \Rightarrow \mathsf{downsweep}(\mathsf{bi}(tid, 2^i), \theta)\right) \end{aligned}$$

where $\bowtie$ is defined as $\geq$ if $2^i = n/2$, and as $=$ otherwise.

Specification Invariant The result of the prescan is expressed in the final barrier invariant:

$$\varphi_{\text{spec}} \triangleq tid < \overline{tid} \Rightarrow (\texttt{prescan}[tid] + \texttt{flag}[tid] \leq \texttt{prescan}[\overline{tid}])$$

This follows as a consequence of the equalities formed by φ_{us} and φ_{ds}, i.e. at $\theta = n$ for φ_{us} and $\theta = 0$ for φ_{ds} (i.e., Figure 4.12 for the concrete case where $n = 8$).

Barrier Invariant Instantiation As discussed in Section 4.3.4, our verification method requires each barrier invariant to specify a set of instantiation expressions to eliminate the quantifier for successor state selection. We give minimal instantiations for the Blelloch barrier invariants where assuming the invariant for further threads does not add useful information for verification.

- In the upsweep of Figure 4.5(a), we see at each iteration that a thread *tid* uses the results produced by threads $2 \cdot tid$ and $2 \cdot tid + 1$ from the previous iteration. For example, thread 1 at offset 2 uses the sums formed by threads 2 and 3 when the

upsweep was at offset 1. Therefore, it suffices to instantiate φ_{us} for threads $2 \cdot \mathit{tid}$ and $2 \cdot \mathit{tid} + 1$ after the upsweep barrier.

- In the downsweep of Figure 4.5(b), we see at each iteration that a thread *tid* uses the results produced by itself *tid* and thread $\mathit{tid}/2$ from the previous iteration. For example, thread 1 at offset 2 uses the results formed by itself and thread 0. Therefore, it suffices to instantiate φ_{ds} for threads *tid* and $\mathit{tid}/2$ after the downsweep barrier.
- At the final barrier we prove the monotonic specification that we require for proving race-freedom of the outer stream compaction kernel. This requires a *re-instantiation* of the upsweep and downsweep barrier invariants, which we have carried through to this point. We recall that the upsweep results in equalities between the elements of `flag` and `sum`, while the downsweep results in equalities between the elements of `sum` and `prescan`. Combining relevant equalities allows us to prove the monotonic specification. We require a linear number of instantiations for the upsweep barrier invariant $\{\mathit{tid}/2^i \mid 0 < i < \lg n\}$ and a constant number of instantiations for the downsweep barrier *tid*. The specification barrier itself requires *tid* and $\overline{\mathit{tid}}$.

4.4.2. Brent-Kung Scan

We now consider a different prefix sum algorithm due to Brent and Kung [BK82]. Figure 4.14 presents an OpenCL kernel that computes a scan. Like the Blelloch prescan, the Brent-Kung scan consists of an upsweep and downsweep phase, and satisfies the same monotonic specification. In the same figure, we show the circuit representation of the algorithm for $n = 8$ elements. There is a wire for each input, corresponding to the array elements of `flag`, and data flows top-down through the diagram. Each node $\bullet$ adds its two inputs and produces an output that passes downward and also optionally across the circuit through a diagonal wire.

Upsweep Invariant The Brent-Kung upsweep matches the Blelloch upsweep so we reuse the same upsweep barrier invariant.

Downsweep Invariant The downsweep combines the partial sums formed in the upsweep to give the expected scan result. Unlike the Blelloch downsweep, which operates over a logical tree, the Brent-Kung downsweep operates over a logical *forest* of trees. We show this in Figure 4.14 by highlighting the two trees of the downsweep: the first of height 1 and another of height 2. An element x resides in a tree of height $h = \lfloor \lg(x+1) \rfloor$ and the binary encoding of the h least significant bits of $x + 1$ gives the path from the root to

```
__kernel void brentkung(
  __local unsigned *idx, unsigned n) {

  unsigned left, right;
  unsigned tid = get_local_id(0);

  // (a) upsweep
  unsigned offset = 1;
  for (unsigned d = n/2; d > 0; d /= 2) {
    barrier(CLK_LOCAL_MEM_FENCE); // φ_us
    if (tid < d) {
      left  = offset * (2 * tid + 1) - 1;
      right = offset * (2 * tid + 2) - 1;
      idx[right] += idx[left];
    }
    offset *= 2;
  }

  // (b) downsweep
  for (unsigned d = 2; d < n; d *= 2) {
    offset /= 2;
    barrier(CLK_LOCAL_MEM_FENCE); // φ_bk-ds
    if (tid < (d - 1)) {
      left  = (offset * (tid + 1)) - 1;
      right = left + (offset / 2);
      idx[right] += left[idx];
    }
  }
}
```

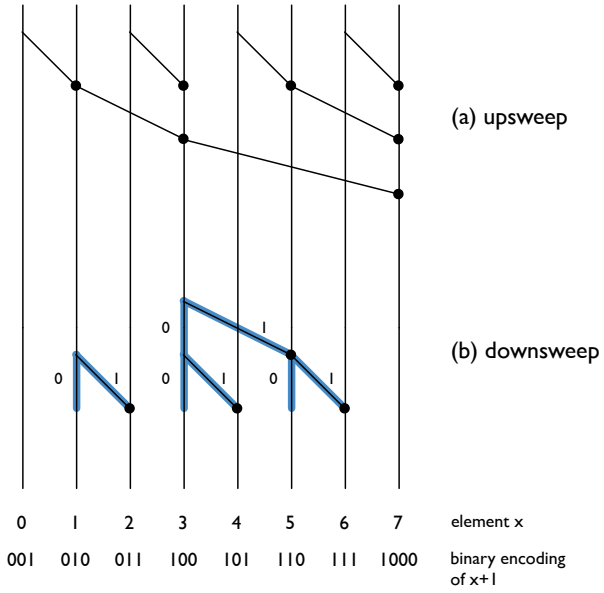


Figure 4.14.: Brent-Kung prefix sum kernel and circuit diagram for $n = 8$

the element leaf vertex using 0 to mean left and 1 to mean right. For example, the path from the root to the leaf vertex 5, with binary encoding $x + 1 = 6 = \texttt{0b110}$, is right (1) and left (0), ignoring bits ≥ 2.

The downsweep updates an output exactly once or not at all. Similar to the Blelloch downsweep, the summations of an element x can be found by traversing the tree from the leaf vertex to the root and gathering a term for each right child vertex. This gives the following per-element invariant, which defines the elements of `scan` in terms of the offset θ:

$$\mathsf{bk\text{-}downsweep}(x, \theta) \triangleq \texttt{scan}[x] = \begin{cases} \texttt{sum}[x] + \sum_{\theta' \in B(x)} \texttt{sum}[y(x, \theta')] & \text{if } \mathsf{updated}(x, \theta) \\ \texttt{sum}[x] & \text{otherwise} \end{cases}$$

where $\mathsf{updated}(x, \theta) \triangleq \bigvee_{\theta' \in \{2^i \mid \lg \theta \leq i < \lg n\}} \theta' < x \wedge \mathsf{isvertex}(x - \theta', 2\theta')$ and $B(x) \triangleq \{2^i \mid 0 \leq i < \lfloor \lg(x+1) \rfloor \wedge \mathsf{bit}(x+1, i)\}$ and $y(x, \theta') \triangleq x - \sum_{\substack{0 \leq j \leq \lg \theta' \\ \mathsf{bit}(x+1, j)}} 2^j$.

The function $\mathsf{updated}(x, \theta)$ returns a Boolean depending on whether the element x has been updated at this stage of the downsweep or not. An element x is updated at an offset θ' if (x, θ') is a right child vertex; this is the case if the element $x - \theta'$ is a vertex at the previous offset $2\theta'$. The function $B(x)$ gives the set of offsets where the path from the leaf vertex to the root contains a right child vertex. We exploit the binary encoding of $x + 1$ to traverse the path and locate right child vertices. The function $y(x, \theta')$ computes the index of the left child vertex summed into the vertex. For example, for element 6 the function is evaluated with $y(6, 2)$ and $y(6, 4)$ to yield the summation terms $\texttt{sum}[5]$ and $\texttt{sum}[3]$, respectively.

Using this per-element invariant, we define the barrier invariant $\varphi_{\text{bk-ds}}$ by considering the elements of `scan` accessed by the thread *tid* in each iteration of the downsweep loop with offset θ. We use the indexing functions $\mathsf{bk\text{-}ai}(\mathit{tid}, \theta) \triangleq (\theta(\mathit{tid} + 1)) - 1$ and $\mathsf{bk\text{-}bi}(\mathit{tid}, \theta) \triangleq \mathsf{bk\text{-}ai}(\mathit{tid}, \theta) + \theta/2$. Table 4.1 gives the ownership of each element for the $n = 8$.

Table 4.1.: Brent-Kung downsweep thread assignment of elements for $n = 8$

	Element x							
offset θ	0	1	2	3	4	5	6	7
8	T0	T0	T1	T0	T2	T1	T3	T0
4	T0	T0	T1	T0	T2	T1	T3	T0
2	T0	T0	T1	T0	T2	T1	T3	T0

$$
\begin{aligned}
\varphi_{\text{bk-ds}} \triangleq & \bigwedge_{1 \le i < \lg n} \left(tid < n/2^i \wedge \theta > 2^i \Rightarrow \mathsf{bk\text{-}downsweep}(\mathsf{ai}(tid, 2^{i-1}), \theta) \right) \\
\wedge & \quad tid = 0 \Rightarrow \mathsf{bk\text{-}downsweep}(\mathsf{ai}(tid, \theta/2), \theta) \\
\wedge & \quad tid = 0 \Rightarrow \mathsf{bk\text{-}downsweep}(\mathsf{bi}(tid, n/2), \theta) \\
\wedge & \bigwedge_{1 \le i < \lg n} \left(tid < (n/2^i - 1) \wedge \theta = 2^i \Rightarrow \mathsf{bk\text{-}downsweep}(\mathsf{bk\text{-}ai}(tid, 2^i), \theta) \right) \\
\wedge & \bigwedge_{1 \le i < \lg n} \left(tid < (n/2^i - 1) \wedge \theta = 2^i \Rightarrow \mathsf{bk\text{-}downsweep}(\mathsf{bk\text{-}bi}(tid, 2^i), \theta) \right)
\end{aligned}
$$

Because the Brent-Kung downsweep does not update all elements of the output the clauses of the barrier invariant involve elements carried over from the upsweep. This accounts for the clauses defined using ai and bi. In particular, thread 0, which is involved in all iterations of the downsweep retains ownership of the last element. The final clauses using bk-ai and bk-bi model the thread assignment of the downsweep.

4.4.3. Kogge-Stone Scan

We consider one last prefix sum algorithm due to Kogge and Stone [KS73] (and also attributed to Hillis and Steele [HS86]). Figure 4.15 presents an OpenCL kernel and circuit representation of the algorithm for $n = 8$ elements. The algorithm takes $\lg n$ iterations and adds elements from successively larger power-of-two offsets.

Initially, at offset 1, we have for each element x that $\texttt{idx}[x] = \texttt{flag}[x] = \sum_{x \le i \le x} \texttt{flag}[i]$. After the first iteration, at offset 2, each element $x \ge 1$ has $\texttt{idx}[x] = \sum_{x-1 \le i \le x} \texttt{flag}[i]$. After the second iteration, at offset 4, each element $x \ge 2$ has $\texttt{idx}[x] = \sum_{x-3 \le i \le x} \texttt{flag}[i]$. We observe that the algorithm always adds adjacent summation intervals: each addition has the form $\sum_{a \le i \le b} \texttt{flag}[i] + \sum_{b+1 \le c} \texttt{flag}[i] = \sum_{a \le c} \texttt{flag}[i]$. In general, we note the

```
__kernel void koggestone(
  __local unsigned *idx, unsigned n) {

  unsigned offset, temp;
  unsigned tid = get_local_id(0);

  for (offset = 1; offset < n; offset *= 2) {
    if (tid >= offset) {
      temp = idx[tid-offset];
    }

    barrier(CLK_LOCAL_MEM_FENCE); // φks1

    if (tid >= offset) {
      idx[tid] += temp;
    }

    barrier(CLK_LOCAL_MEM_FENCE); // φks2
  }
}
```

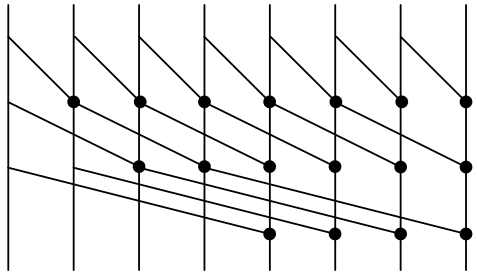


Figure 4.15.: Kogge-Stone prefix sum kernel and circuit diagram for $n = 8$

following relationship between an element x and the offset θ:

$$\texttt{idx}[x] = \begin{cases} \sum_{0 \le i \le x} \texttt{flag}[i] & \text{if } x < \theta \\ \sum_{x-\theta+1 \le i \le x} \texttt{flag}[i] & \text{otherwise} \end{cases}$$

Encoding this observation directly requires recursion to express the summation. The barrier invariants of Blelloch and Brent-Kung avoided this by expressing summations over the elements of `idx` (rather than the input `flag`). In these kernels an element of `idx` is stable (not updated) after it has been used as in a summation. This is not the case for Kogge-Stone. For example, `idx`[3] is summed with `idx`[5] at offset 2 but is subsequently updated itself at offset 4.

We addressed this problem by applying an abstraction to the source code that allows our observation to be encoded without recursion. We change the type of `idx` to representing an abstract interval (a, b) denoting the summation $\sum_{a \le i \le b} \texttt{flag}[i]$. Addition of adjacent intervals is defined as $(a, b) \oplus (b+1, c) = (a, c)$. This allows us to rephrase our observation using abstract intervals.

$$\begin{aligned} \mathsf{ks}(x, \theta) \quad \triangleq \quad & x < \theta \Rightarrow \texttt{idx}[x] = (0, x) \wedge \\ & x \ge \theta \Rightarrow \texttt{idx}[x] = (x - \theta + 1, x) \end{aligned}$$

We now use this invariant to define barrier invariants for the two barriers of the Kogge-

Stone implementation. The structure of these barrier invariants is straightforward compared to the Blelloch or Brent-Kung barrier invariants. This is due to the use of our abstraction and also the implementation choice to assigns a single thread to each element for the duration of the kernel. In both the Blelloch and Brent-Kung kernels, the thread assignment of elements changes as the kernel proceeds.

$$\begin{aligned} \varphi_{\text{ks1}} &\triangleq \mathtt{ks}(tid, \theta) \wedge \mathtt{temp} = \mathtt{idx}[tid - \theta] \\ \varphi_{\text{ks2}} &\triangleq \mathtt{ks}(tid, 2\theta) \end{aligned}$$

The specification proved using barrier invariants and the interval abstraction differs from the monotonic specification proved for the Blelloch and Brent-Kung implementations. Using the abstract interval, the scan (inclusive prefix sum) specification can be written as $\mathtt{idx}[x] = (0, x)$ for all elements x. We call this the abstract specification to distinguish it from the monotonic specification. We will revisit and further develop this abstraction in Chapter 5.

4.5. Experimental Evaluation

We now evaluate the effectiveness of barrier invariants for precise reasoning for data-dependent GPU kernels. The main findings of our experiments are:

- Modular verification of the data-dependent stream compaction kernel allows GPUVerify to scale to problem sizes up to 2^{31} threads.
- Staged verification of the Blelloch prescan and Brent-Kung scan using barrier invariants allows GPUVerify to prove a monotonic specification for problem sizes involving hundreds of threads.
- Staged verification of the Kogge-Stone scan using barrier invariants and the interval abstraction allows GPUVerify to prove an abstract specification for problem sizes up to 2^{31} threads.

We compare GPUVerify using barrier invariants against the GKLEE [LLS+12] and GKLEE$_p$ [LLG12] tools. As discussed in Chapter 2, both tools are based on dynamic symbolic execution [CDE08]. Although designed for bug-finding rather than verification these tools can give brute-force verification guarantees by exhaustive path exploration. We do not compare against KLEE-CL due to bugs in the tool; however, due to many similarities between KLEE-CL and GKLEE, we expect that a comparison would will yield similar results. We do not compare to GPUVerify without barrier invariants [BCD+12],

Table 4.2.: Modular verification of the stream compaction kernel

	Number of Threads													
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768
GPUVerify	1.47 ±0.04	1.43 ±0.01	1.43 ±0.05	1.47 ±0.04	1.39 ±0.02	1.40 ±0.02	1.38 ±0.02	1.38 ±0.02	1.41 ±0.02	1.38 ±0.02	1.39 ±0.03	1.40 ±0.02	1.41 ±0.02	1.41 ±0.02
GKLEE	0.80 ±0.08	76.95 ±0.27	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
GKLEE$_p$	0.57 ±0.01	1.03 ±0.04	3.38 ±0.06	9.40 ±0.19	39.44 ±0.97	220.89 ±5.01	1838.71 ±110.81	TO	TO	TO	TO	TO	TO	TO

PUG [LG10] or Test Amplification [LGA+12] as they do not support reasoning about data-dependent GPU kernels.

Experimental Setup All experiments were performed on a compute cluster using nodes with Intel Xeon EP-2620 cores at 2 GHz with 16 GB RAM running RedHat Linux 6.3, revision 2784 of Boogie and Z3 v4.3.1. We used a timeout of 3 hours for each experiment.

4.5.1. Modular Verification of Stream Compaction

Our first experiment compares the time taken by GPUVerify, GKLEE and GKLEE$_p$ for establishing race-freedom of the stream compaction kernel where the prefix sum is replaced with its monotonic specification. For GKLEE and GKLEE$_p$, which do not support modular verification directly, we encoded the monotonic specification using a set of *assume* commands. We varied the problem size for all power-of-two thread counts from 2 to 2^{31}.

Table 4.2 shows for each thread count up to 2^{15} the time taken in seconds for analysis with GPUVerify, GKLEE and GKLEE$_p$. Times, in seconds, are averages over 10 runs, and the variation (95% confidence interval using a two-tailed t-distribution) between runs is also shown. Timeouts are indicated by 'TO'. In all cases, the analysis with GPUVerify succeeds in *less than two seconds* and this trend continues for all thread counts up to 2^{32}. GKLEE is capable of analysis up to 8 threads before timeout occurs and GKLEE$_p$ scales further to 256 threads within the experiment resource limits. That is, modular verification allows GPUVerify to scale to problem sizes beyond the effective range of dynamic symbolic execution.

4.5.2. Staged Verification of Blelloch and Brent-Kung

The correctness of the modular results rely on proving that the prefix sum implementation satisfies the monotonic specification. Our second experiment compares the time taken by GPUVerify and GKLEE for establishing the monotonic specification for the Blelloch and Brent-Kung kernels. We do not compare against GKLEE$_p$ because we found that analysis

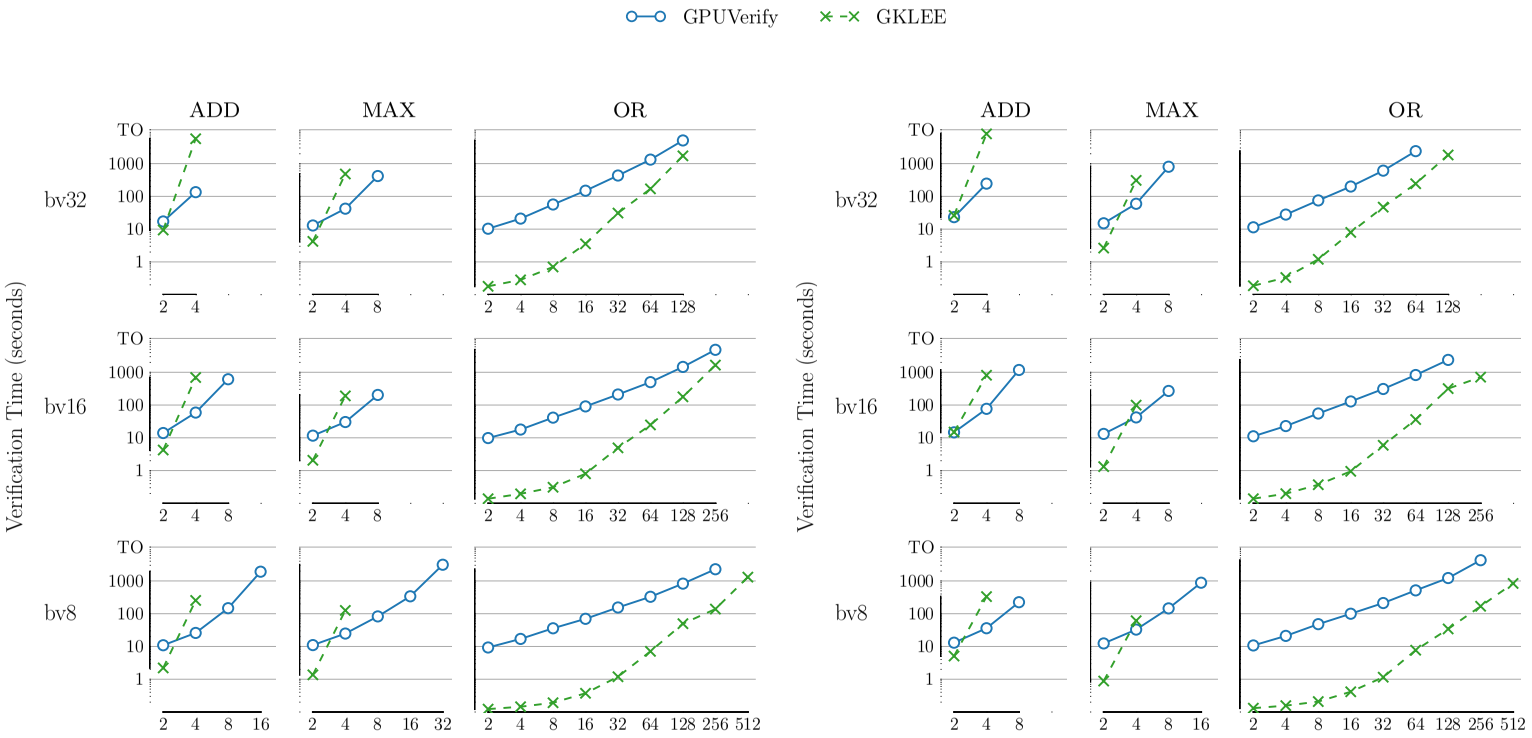
of the prefix sum kernels led to false positive reports of the monotonic postcondition failing. This is because the parametric flow technique used by GKLEE$_p$ is a type of thread reduction and therefore requires a form of shared state abstraction. GKLEE, which models all threads without reductions, does not share this problem.

For stream compaction we require the prefix sum with the binary add operator with identity 0. As discussed in Section 4.4, the monotonic specification does not hold in the case of overflow. For our experiments with GPUVerify we defined a non-overflowing add operator. To add two n-bit integers we zero-extend both operands and perform the addition using $n + 1$ bits. We then assume that the top bit of the result is zero, which restricts analysis to the case where no-overflow has occurred. Finally, we contract back to n-bits and return the result. It was not possible to add similar support directly to GKLEE, thus for our experiments with GKLEE we model bit-vector addition as saturating.

We varied the problem size for all power-of-two thread sizes from 2, until we hit a resource limit. We verified the monotonic specification for three binary operators (addition, max and bitwise-or) using three different bit-vector widths (8-, 16- and 32-bit integers). We varied the binary operator because the prefix sum operation is defined over an associative binary operator with an identity and the monotonic specification, with equivalent barrier invariant, also holds for the max and bit-wise or operators, both with identity 0.

The graphs of Figure 4.16 show times for verifying (a) the Blelloch prescan and (b) the Brent-Kung scan prefix sums for GPUVerify (data points with circles) and GKLEE (data points with crosses). We vary the problem size, in terms of number of threads, across the x-axis. We consider the binary operators for addition, max and bitwise-or (ADD, MAX and OR) using 8-, 16- and 32- bit-vector widths (bv8, bv16 and bv32). For GPUVerify, each data point is the total time to verify the kernel using a staged verification strategy: i.e., the time to verify the implementation race-free plus the time to verify the monotonic specification (with race checking disabled). For GKLEE, each data point is the time taken for exhaustive path analysis. For each tool, absence of a data point indicates that a timeout or memory limit was reached.

The results show that for both Blelloch and Brent-Kung with the addition and max operators that GPUVerify scaled to larger problem sizes (tens of threads) than GKLEE, although at smaller problem sizes the overhead of using barrier invariants was greater than exhaustive path exploration. For the bitwise-or operator, both tools are capable of reasoning to significantly larger problem sizes involving hundreds of threads. In four cases, GKLEE is able to verify one larger thread configuration than GPUVerify.



(a) Blelloch prescan

(b) Brent-Kung scan

Figure 4.16.: Verification results for (a) Blelloch and (b) Brent-Kung prefix sums

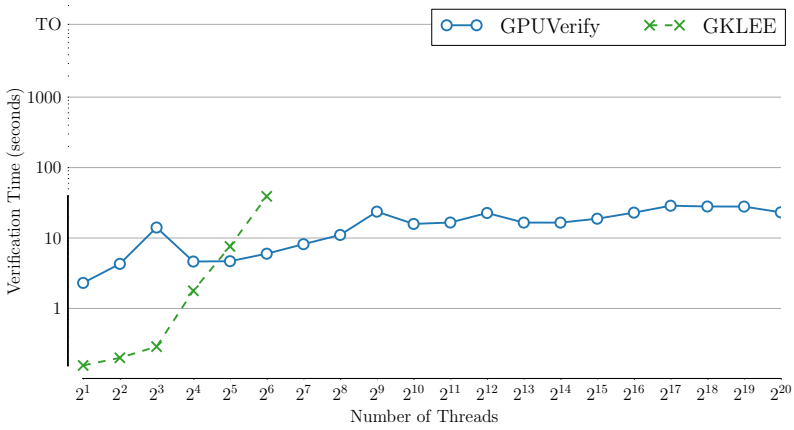


Figure 4.17.: Verification results for Kogge-Stone prefix sum

4.5.3. Staged Verification of Kogge-Stone

This experiment uses barrier invariants and the interval abstraction to verify the abstract specification of the Kogge-Stone scan kernel. Figure 4.17 shows times for verifying the kernel for GPUVerify (data points with circles) and GKLEE (data points with crosses).

The graph shows that GPUVerify scales to problem sizes well beyond the range of GKLEE. GPUVerify was able to verify the kernel up to 2^{31} threads (the graph shows results up to 2^{20} threads) showing the scalability that can be achieved by combining the two-thread reduction, barrier invariants and additional abstractions.

4.6. Related Work

Thread Contracts The work of *thread contracts* [KMM11] also addresses the problem of race-freedom for data-parallel programs, including data-dependent programs, which the paper identifies as parallelism using *indirection arrays*. In this approach the programmer annotates the program with a *coordination strategy*: a logical description of the parts of shared memory that an arbitrary thread will access. The key idea is that a coordination strategy can be automatically checked, by translation into an SMT query, to see whether the annotations imply race-freedom of the program. However, the paper does not address the problem of ensuring that the coordination strategy itself is correct; that is, verifying that the parallel program obeys its coordination strategy. Instead, this technique uses runtime assertions, generated automatically from the coordination strategy, to *test* whether the coordination strategy is correct. Barrier invariants can be seen as a type of coordination strategy tailored for the analysis of GPU kernels. The main difference

between barrier invariants and thread contracts is that the verification method for barrier invariants ensures the validity of each barrier invariant as well as using them to prove race-freedom.

Collective Loop Invariants The work of *collective loop invariants*, introduced by Siegel and Zirkel [SZ12], is concerned with generalising loop invariants to parallel programs. In particular, for verifying MPI programs using symbolic execution. A collective loop invariant is an assertion defined over a set of processes $\mathcal{P}$ and a set of loop heads $\mathcal{L}$. The assertion must hold in any state where every process in $\mathcal{P}$ is at a loop head in $\mathcal{L}$. Similar to barrier invariants, collective loop invariants can establish properties over sets of processes or threads. However, they are otherwise orthogonal: a barrier invariant captures shared state properties when all threads in a GPU kernel synchronise at the *same* barrier; a collective loop invariant captures state properties when processes synchronise (infrequently) at (possibly distinct) loop heads.

Protocol Verification The CMP method [CMP04] (named after the authors) addresses the problem of verifying cache coherence protocols. Under this method, a protocol for an arbitrary number of processes is verified by model checking a system where a small number of processes are modeled explicitly with an additional non-deterministic 'other' process. The purpose of the 'other' process is to over-approximate the possible behaviours of the processes not modeled explicitly. This is a form of reduction similar to the two-thread reduction employed by GPUVerify where the 'other' process is analogous to the shared state abstraction. In a similar fashion to coarser shared state abstractions the unconstrained behaviour of the 'other' process can lead to spurious errors. To avoid this, the CMP method uses *non-interference lemmas* to refine the behaviour of the 'other' process. Hence, at a high-level, non-interference lemmas can be seen as an analogous idea to barrier invariants: a refinement of a coarse abstraction to avoid spurious errors. However, in practice, the techniques differ due to their application domains: non-interference lemmas describe interaction sequences between processes that communicate through message passing; whereas, barrier invariants capture properties of shared state in data-parallel programs communicating through shared memory.

Permission-Based Separation Logic As discussed in Chapter 2, the application of *permission-based separation logic* to GPU kernels [BHM14] allows a kernel to be annotated with a specification that captures the assignment of read or write permissions to memory locations on a per-thread basis. The soundness of the logic ensures that if a proof can be

written for the kernel (i.e., the kernel verifies) then the write permissions of all threads are exclusive and hence the kernel is data race-free. In this verification technique, barriers can be used to exchange permissions between threads using a *barrier specification*. For example, a barrier specification can capture the wrap around exchange of permissions from each thread to its neighbouring thread in example (d) of Figure 4.10. This is analogous to the instantiation expressions required for barrier invariants.

4.7. Summary

Barrier invariants are a new shared state abstraction for addressing the problem of precise and scalable reasoning for data-dependent GPU kernels. The key feature of this abstraction is the ability to precisely capture properties about shared state whilst retaining scalability through the use of the two-thread reduction. We have demonstrated the applicability of barrier invariants through a detailed study of stream compaction, where barrier invariants enabled us to reason about three different prefix sum implementations and prove their specifications.

5. The Interval of Summations: Functional Correctness for Prefix Sums

In this chapter we further develop the interval abstraction introduced in Chapter 4 for the verification of the Kogge-Stone prefix sum. Motivated by the importance of prefix sums as a parallel primitive, we extend out observation that the Kogge-Stone algorithm uses adjacent intervals into a general result about all prefix sums. We introduce a new abstraction, the *interval of summations*, that enables precise reasoning about prefix sums and yields an automatic and highly-scalable verification method. The surprising result is that the abstraction is an "exact fit" for this class of algorithm. All correct prefix sums can be precisely captured by the abstraction. The main results of this chapter are:

- A soundness and completeness result showing that a *sequential* prefix sum implementation is correct for an array of length n if and only if the implementation computes the correct result for a specific test case using the interval of summations.
- A verification method that uses this result to establish the correctness of a sequential prefix sum implementation by running a single interval of summations test case requiring $O(n \lg n)$ space for the input and output of the implementation.
- An extension of the abstraction and results for *data-parallel* prefix sum implementations.
- An experimental evaluation that applies our verification method to four distinct data-parallel prefix sums showing that our verification method is automatic and scales to all power-of-two problem sizes up to 2^{20}.

Relation to Published Work The core material in this chapter was published in [CDK14].

Table 5.1.: Some applications of parallel prefix sums. Stream compaction requires an exclusive prefix sum; the other applications employ inclusive prefix sums. The carry operator and functional composition are examples of non-commutative operators.

Prefix sum application	**Data-type**	**Operator**
Stream compaction [HSO07, BOA09]	Int	$+$
Sorting algorithms [HSO07, SHG09]	Int	$+$
Polynomial interpolation [EGK90]	Float	$*^a$
Line-of-sight calculation [Ble93]	Int	max
Binary addition [LF80]	pairs-of-bits	carry operator ¢
Finite state machine simulation [LF80]	transition functions	function composition

[a]Floating point multiplication is not actually associative, but is often treated as such in applications where some error can be tolerated.

5.1. Prefix Sums and Their Applications

The prefix sum operation, which computes the sums of all prefixes of an array, is a fundamental parallel primitive. In Chapter 4, we introduced prefix sums in the context of stream compaction (Figure 4.3). In this application a group of threads coordinate to filter an array with respect to some predicate and the prefix sum gives the index where each thread must write to ensure a compacted output. This application of the prefix sum is one of many. The utility of prefix sums extends to both hardware and software; we give some examples in Table 5.1. In hardware, prefix sums were originally studied as circuits for carry-propagation logic in adders [Skl60]. Well known circuits include prefix sums due to Kogge and Stone [KS73], Ladner and Fischer [LF80], and Brent and Kung [BK82]. The design space has been further explored by Sheeran [She11]. In software, Blelloch identifies the prefix sum as "one of the simplest and most useful building blocks for parallel algorithms" and lists 13 applications including string comparison, polynomial evaluation and recurrence relation solving [Ble93]. The prefix sum is found in all GPU data-parallel primitive libraries, including the Thrust Parallel Algorithms Library, the CUDA Data-Parallel Primitives Library and the OpenCL Data-Parallel Primitives Library.[1]

Example: Binary Addition In digital logic, a full-adder sums two 1-bit operands with a carry-in to produce a pair of sum and carry-out bits. If the two operands are bits a and b with carry-in bit c then the sum s and carry-out c' can be defined as $s \triangleq a \oplus b \oplus c$ and

[1]See `http://thrust.github.io`, `http://cudpp.github.io` and `https://code.google.com/p/clpp/`

$c' \triangleq (a \cdot b) \oplus (c \cdot (a \oplus b))$ (in this sub-section we use $\oplus$ to denote exclusive-or and write $x \cdot y$ to denote the bitwise-and of x and y). By combining multiple full-adders we can add n-bit operands using a full-adder for each bit of the addition. If the carry is propagated through the circuit by passing the carry-out of each full-adder into the carry-in of the next full-adder then we have a ripple-carry adder (Figure 5.1(a) where 'FA' denotes a full-adder). In a ripple-carry adder the critical path is the length of the carry chain.

A carry-lookahead adder avoids a carry chain by computing the carry-out for each full-adder in parallel. One method for achieving this is an inclusive prefix sum. We modify each full-adder to produce a pair of *propagate* and *generate* bits $(p_i, g_i) \triangleq (a_i \oplus b_i, a_i \cdot b_i)$. The propagate bit means that the carry-out of the full-adder is conditional on the carry-in (the carry-out should be set if the carry-in is set) whereas the generate bit means that the carry-out is unconditional (the carry-out is always set). The prefix sum of the propagate-generate bits with respect to the carry operator $(p_{i-1}, g_{i-1}) \mathbin{¢} (p_i, g_i) \triangleq (p_i \cdot p_{i-1}, g_i \oplus (p_i \cdot g_{i-1}))$ results in a combined pair-of-bits (p'_i, g'_i) for each full-adder. The carry operator is associative and non-commutative with identity $(1, 0)$. For example, the 4-bit carry-lookahead logic in Figure 5.1(b) will produce:

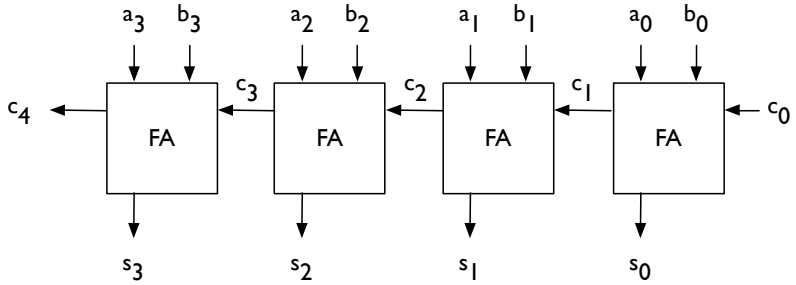
$$\begin{array}{lcl} p'_0, g'_0 & = & p_0, g_0 \\ p'_1, g'_1 & = & p_1 \cdot p_0, g_1 \oplus p_1 \cdot g_0 \\ p'_2, g'_2 & = & p_2 \cdot p_1 \cdot p_0, g_2 \oplus p_2 \cdot g_1 \oplus p_2 \cdot p_1 \cdot g_0 \\ p'_3, g'_3 & = & p_3 \cdot p_2 \cdot p_1 \cdot p_0, g_3 \oplus p_3 \cdot g_2 \oplus p_3 \cdot p_2 \cdot g_1 \oplus p_3 \cdot p_2 \cdot p_1 \cdot g_0 \end{array}$$

Intuitively, the combined propagation bit p'_i means that the initial carry-in c_0 propagates through the full-adder intact whereas the combined generate bit g'_i means that the carry-out should be set unconditionally. Thus the result of each full-adder can be computed in two further parallel steps. The carry out for each full-adder $i > 0$ is $c_i \triangleq g'_{i-1} \oplus (p'_{i-1} \cdot c_0)$ and the sum of each full-adder is $s_i \triangleq p_i \oplus c_i$. Note that the sum uses the original propagate bit p rather than the combined bit p' resulting from the prefix sum.

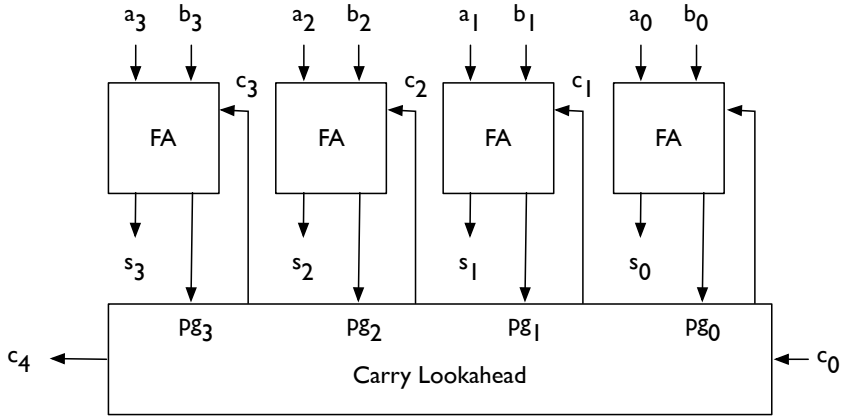
Definition We now formally define the prefix sum operation, which we also discussed in the previous chapter. For convenience we repeat the definition here. Let $\mathbb{S}$ be a set with an associative binary operator $\oplus$. That is, $(\mathbb{S}, \oplus)$ is a semigroup. Then the *inclusive prefix sum* of an array $[s_1, s_2, \ldots, s_n]$ of elements of $\mathbb{S}$ is the array:

$$[s_1, s_1 \oplus s_2, \ldots, s_1 \oplus s_2 \oplus \cdots \oplus s_n]$$

of all sums of inclusive prefixes, in increasing order of length.



(a) Ripple-carry adder



(b) Carry-Lookahead adder

Figure 5.1.: 4-bit ripple-carry and carry-lookahead adders

If $(\mathbb{S}, \oplus)$ has an identity element $\mathbf{1}$ (so that $(\mathbb{S}, \oplus)$ is a monoid) then the *exclusive prefix sum* of $[s_1, s_2, \ldots, s_n]$ is the array:

$$[\mathbf{1}, s_1, s_1 \oplus s_2, \ldots, s_1 \oplus s_2 \oplus \cdots \oplus s_{n-1}]$$

of all sums of exclusive prefixes.

For example, if $\mathbb{S}$ is the set of integers under addition with identity 0 then the inclusive and exclusive prefix sum of the array $[3, 1, 7, 0, 4, 1, 6, 3]$ is $[3, 4, 11, 11, 15, 16, 22, 25]$ and $[0, 3, 4, 11, 11, 15, 16, 22]$, respectively. The inclusive and exclusive prefix sum can be computed from one another. The inclusive prefix sum can be formed by computing the exclusive prefix sum and then shifting the elements left by one and inserting the reduction of the input. Similarly, the exclusive prefix sum can be formed by computing the inclusive prefix sum and then shifting the elements right by one and inserting the identity.

The Interval of Summations The *interval of summations* abstraction is a tailored abstraction for reasoning about prefix sums. The key insight that we exploit is the observation that a prefix sum is defined only for an associative binary operator. For two indices $i < j$, the interval of summations uses the abstract interval (i, j) to represent a contiguous summation interval of input elements $s_i \oplus s_{i+1} \oplus \cdots \oplus s_j$ (with respect to any data type and associative operator $\oplus$). Two abstract intervals can only be added together if they are adjacent or 'kiss': the abstract sum of (i, j) and (k, l) is (i, l) if $j + 1 = k$; otherwise, the result is a special value $\top$ that represents all non-contiguous sums. Any abstract sum involving $\top$ yields $\top$, modeling the fact that using only associativity it is impossible to create a contiguous summation interval from a non-contiguous interval using addition.

We will prove that the interval of summations is a sound and complete abstraction for prefix sums (Section 5.2.5). Furthermore, this result enables a sound *hybrid* verification method for data-parallel prefix sums (Section 5.4). Similar to the test amplification work [LGA+12] discussed in Section 2.2.3, we show that a *single dynamic run* (using the interval of summations) is sufficient for establishing the correctness of a prefix sum implementation of a given length n. We can test a generic prefix sum implementation (defined over all monoids) using an interval of summations input $[(0, 0), (1, 1), (2, 2) \ldots (n-1, n-1)]$ of length n and checking that the output matches $[(0, 0), (0, 1), (0, 2), \ldots, (0, n-1)]$ (for an inclusive prefix sum). In this case, the input can be regarded as a concrete input. Informally, each (i, i) of the input is an abstract representation of *any* element s_i of *any* monoid and each $(0, i)$ of the output is an abstract representation of the expected summation for *any* monoid. Hence, because of the soundness and completenesss result proved in this

```
program ::= vars: decl;
            main: stmts;
decl    ::= name : type
          | arrayname : type[]
          | decl ; decl
stmts   ::= ε
          | stmt ; stmts
stmt    ::= name := expr
          | arrayname[expr] := expr
          | if (expr) then {stmts} else {stmts}
          | while (expr) do {stmts}
expr    ::= constant literal
          | name
          | arrayname[expr]
          | expr op expr
```

Figure 5.2.: Syntax for a simple sequential imperative language

chapter, this is sufficient to prove the correctness of the prefix sum implementation for all other monoids. Another interpretation of this result[2] is that given a prefix sum instantiated for some specific operator (e.g., integer addition) the interval of summations gives an efficient abstraction for symbolic execution. In this case, the input can be regarded as a symbolic input. Both views are compatible.

5.2. Sequential Setting

We begin by developing the interval of summations in the context of a sequential imperative language, which leads to our main soundness and completeness result. We then extend this result to data-parallel prefix sums yielding an automatic and highly-scalable verification method, which we evaluate in Section 5.5.

5.2.1. Syntax

Figure 5.2 gives the syntax for a simple typed imperative language. A program consists of a list of type declarations and a sequence of statements, which is the body of the program. The sets name and arrayname denote the scalar and array variables of the program, respectively. This syntactically eliminates nested array declarations and expressions (e.g., $A[B[e]]$). Statements can be combined in conditionals and loops.

[2] Thanks to Mike Dodds for this view.

$$\frac{c \text{ of type } T}{\Gamma \vdash c : T}\ (\text{T-Literal}) \qquad \frac{v : T \in \Gamma}{\Gamma \vdash v : T}\ (\text{T-Variable})$$

$$\frac{A : \mathsf{Array}(T) \in \Gamma \qquad \Gamma \vdash e : \mathsf{Int}}{\Gamma \vdash A[e] : T}\ (\text{T-Array})$$

$$\frac{op \text{ of type } T_1 \times T_2 \to T_3 \qquad \Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1 \ \ op \ \ e_2 : T_3}\ (\text{T-Op})$$

$$\frac{v : T \in \Gamma \qquad \Gamma \vdash e : T}{\Gamma \vdash v := e : \mathsf{Unit}}\ (\text{T-Assign})$$

$$\frac{A : \mathsf{Array}(T) \in \Gamma \qquad \Gamma \vdash e_1 : \mathsf{Int} \qquad \Gamma \vdash e_2 : T}{A[e_1] := e_2 : \mathsf{Unit}}\ (\text{T-Array-Assign})$$

$$\frac{\Gamma \vdash e : \mathsf{Bool} \qquad \Gamma \vdash ss_1 : \mathsf{Unit} \qquad \Gamma \vdash ss_2 : \mathsf{Unit}}{\Gamma \vdash \textbf{if } (e) \textbf{ then } \{ss_1\} \textbf{ else } \{ss_2\} : \mathsf{Unit}}\ (\text{T-Ite})$$

$$\frac{\Gamma \vdash e : \mathsf{Bool} \qquad \Gamma \vdash ss : \mathsf{Unit}}{\Gamma \vdash \textbf{while } (e) \textbf{ do } \{ss\} : \mathsf{Unit}}\ (\text{T-Loop}) \qquad \frac{}{\Gamma \vdash \varepsilon : \mathsf{Unit}}\ (\text{T-Empty})$$

$$\frac{\Gamma \vdash s : \mathsf{Unit} \qquad \Gamma \vdash ss : \mathsf{Unit}}{\Gamma \vdash s; ss : \mathsf{Unit}}\ (\text{T-Seq})$$

Figure 5.3.: Typing rules for expressions and statements

5.2.2. Typing

Figure 5.3 gives typing rules for the language. The rules are straightforward. We use c to range over constant literal values, v to range over scalar variables, A to range over array variables and op to range over an unspecified set of binary operators. We assume a set of base types $\mathcal{T}$, ranged over by T, which includes at least the integers and Booleans (denoted Int and Bool) and the single-element type Unit. Constant literal and scalar variable types are drawn from $\mathcal{T}$. Array variables have type $\mathsf{Array}(T)$ which denotes all maps of type $\mathsf{Int} \to T$. For ease of presentation we assume no out-of-bounds errors occur and we do not allow arrays of arrays. The typing rules ensure that only Boolean valued expressions are used in conditionals and only integer valued expressions are used as array indices.

5.2.3. Semantics

Let P be a program. A program state $\mathcal{S}$ for P is a tuple:

$$(\sigma_{\mathrm{v}}, \sigma_{\mathrm{A}}, ss)$$

where $\sigma_{\mathrm{v}} : \mathsf{name} \to \biguplus_{T \in \mathcal{T}} T$ is the mapping of scalar variables to values such that if $v \in \mathsf{name}$ is of type T then $\sigma_{\mathrm{v}}(v)$ is of type T; and $\sigma_{\mathrm{A}} : \mathsf{arrayname} \to \biguplus_{T \in \mathcal{T}} \mathsf{Array}(T)$ is the mapping of array variables such that if $A \in \mathsf{arrayname}$ is of type $\mathsf{Array}(T)$ then $\sigma_{\mathrm{A}}(A)$ is of type $\mathsf{Array}(T)$; and ss is an ordered sequence of statements. The disjoint union of all base types is given by $\biguplus_{T \in \mathcal{T}} T$. The set of all program states is denoted State. An *initial state of* P is a program state $(\sigma_{\mathrm{v}}, \sigma_{\mathrm{A}}, ss)$ where ss is the body of the program (declared using **main** : ss).

The valuation of an expression with respect to a variable store σ_{v} and array store σ_{A} is defined inductively.

$$\begin{aligned}
[\![\text{constant literal}]\!]^{\sigma_{\mathrm{v}}}_{\sigma_{\mathrm{A}}} &= \text{constant literal} \\
[\![\mathsf{name}]\!]^{\sigma_{\mathrm{v}}}_{\sigma_{\mathrm{A}}} &= \sigma_{\mathrm{v}}(\mathsf{name}) \\
[\![\mathsf{arrayname}[\mathsf{expr}]]\!]^{\sigma_{\mathrm{v}}}_{\sigma_{\mathrm{A}}} &= \sigma_{\mathrm{A}}(\mathsf{arrayname})([\![\mathsf{expr}]\!]^{\sigma_{\mathrm{v}}}_{\sigma_{\mathrm{A}}}) \\
[\![\mathsf{expr}_1 \mathit{\ op\ } \mathsf{expr}_2]\!]^{\sigma_{\mathrm{v}}}_{\sigma_{\mathrm{A}}} &= [\![\mathsf{expr}_1]\!]^{\sigma_{\mathrm{v}}}_{\sigma_{\mathrm{A}}} \mathit{\ op\ } [\![\mathsf{expr}_2]\!]^{\sigma_{\mathrm{v}}}_{\sigma_{\mathrm{A}}}
\end{aligned}$$

The rules of Figure 5.4 define the evolution of a program state. We write $ss_1 \cdot ss_2$ to denote the concatenation of sequences of statements ss_1 and ss_2. The rules are straightforward except for the split of the program store into a (scalar) variable and array store. This split is not strictly necessary here but anticipates the extension to a data-parallel semantics where we will regard variables as thread-private and arrays as shared among all threads.

Given an initial state $\mathcal{S}_0$ of P, an *execution of* P is a finite or infinite sequence of program states:

$$\mathcal{S}_0 \to_{\mathrm{s}} \mathcal{S}_1 \to_{\mathrm{s}} \cdots \to_{\mathrm{s}} \mathcal{S}_i \to_{\mathrm{s}} \mathcal{S}_{i+1} \to_{\mathrm{s}} \cdots$$

We will use $\mathcal{S} \to_{\mathrm{s}}^{+} \mathcal{S}'$ to denote the transitive closure of the relation $\to_{\mathrm{s}}$. An execution is *maximal* if it (i) cannot be extended by applying one of the rules of the operational semantics or (ii) is infinite. We say P *terminates for an initial state* $\mathcal{S}$ if all maximal executions starting from $\mathcal{S}$ are finite. For our sequential semantics a maximal execution is unique because execution is deterministic; this will not be the case when we extend out results to a data-parallel setting.

The proofs for progress and preservation [Pie02, sec. 8.3] are standard. Type preser-

vation for expressions tells us that if $e : T$ then $[\![e]\!]^{\sigma_\mathrm{v}}_{\sigma_\mathrm{A}} : T$; this is straightforward by induction on the structure of e. Then type safety for statements follows by induction on the structure of stmt.

Theorem 5.1 (seq) (Progress and Preservation)**.** *Let σ_v be a variable store and σ_A be an array store and ss : Unit be a well-typed sequence of statements. Then ss is either a value (a sequence of statements that cannot be further reduced: i.e., the empty sequence) or $(\sigma_\mathrm{v}, \sigma_\mathrm{A}, ss) \to_\mathrm{s} (\sigma'_\mathrm{v}, \sigma'_\mathrm{A}, ss')$ with ss' a well-typed sequence of statements (of type Unit).*

Proof. By induction on the structure of ss. If ss is the empty sequence ε then the result is immediate since T-Empty ensures that ε : Unit. Otherwise, $ss = s; ss''$ for some statement s and statement sequence ss''. By T-Seq, we know that s : Unit and ss'' : Unit are well-typed. We consider each case of s and show that in all cases the resulting ss' of the next program state is well-typed.

- Case $s = v := e$. Then S-Assign applies and $ss' = ss''$.
- Case $s = A[e_1] := e_2$. Then S-Array-Assign applies and $ss' = ss''$.
- Case $s =$ **if** (e) **then** $\{ss_1\}$ **else** $\{ss_2\}$. By T-Ite, we have e : Bool, ss_1 : Unit and ss_2 : Unit. Using type preservation for expressions we have $[\![e]\!]^{\sigma_\mathrm{v}}_{\sigma_\mathrm{A}}$: Bool so either S-Ite-T or S-Ite-F applies. Therefore, either $ss' = ss_1 \cdot ss''$ or $ss' = ss_2 \cdot ss''$. In each case, by T-Seq, we know that ss' has type Unit.
- Case $s =$ **while** (e) **do** $\{ss_1\}$. Then by T-Loop we have e : Bool and ss_1 : Unit. Using type preservation for expressions we have $[\![e]\!]^{\sigma_\mathrm{v}}_{\sigma_\mathrm{A}}$: Bool so either S-Loop-T or S-Loop-F applies. Therefore, either $ss' = ss_1 \cdot ss''$ (in which case, by T-Seq, we know that ss' has type Unit) or $ss' = ss''$ (in which case it is immediate that ss' has type Unit). □

Prefix Sum Algorithms We now define what it means for a program in our language to implement a prefix sum algorithm. As discussed earlier, an inclusive and exclusive prefix sum is defined with respect to a semigroup and monoid, respectively. Because most prefix sums of interest in practice are over monoids (all applications in Table 5.1 are over monoids) we focus on this case. It is straightforward to restrict the results we present over monoids to semigroups. We write M to refer to a monoid with elements $\mathbb{S}_M \in \mathcal{T}$, binary operator $\oplus_M$ (assumed to be a programming language operator) and identity $\mathbf{1}_M \in \mathbb{S}_M$. All monoids satisfy the following properties:

S-Assign

$$\frac{\sigma'_{\mathrm{v}} = \sigma_{\mathrm{v}}[v \mapsto [\![e]\!]^{\sigma_{\mathrm{v}}}_{\sigma_{\mathrm{A}}}]}{(\sigma_{\mathrm{v}}, \sigma_{\mathrm{A}}, v := e; ss') \rightarrow_{\mathrm{s}} (\sigma'_{\mathrm{v}}, \sigma_{\mathrm{A}}, ss')}$$

S-Array

$$\frac{[\![e_1]\!]^{\sigma_{\mathrm{v}}}_{\sigma_{\mathrm{A}}} = m \qquad A' = \sigma_{\mathrm{A}}(A)[m \mapsto [\![e_2]\!]^{\sigma_{\mathrm{v}}}_{\sigma_{\mathrm{A}}}] \qquad \sigma'_{\mathrm{A}} = \sigma_{\mathrm{A}}[A \mapsto A']}{(\sigma_{\mathrm{v}}, \sigma_{\mathrm{A}}, A[e_1] := e_2; ss') \rightarrow_{\mathrm{s}} (\sigma_{\mathrm{v}}, \sigma'_{\mathrm{A}}, ss')}$$

S-Ite-T

$$\frac{[\![e]\!]^{\sigma_{\mathrm{v}}}_{\sigma_{\mathrm{A}}}}{(\sigma_{\mathrm{v}}, \sigma_{\mathrm{A}}, \textbf{if } (e) \textbf{ then } \{ss_1\} \textbf{ else } \{ss_2\}; ss') \rightarrow_{\mathrm{s}} (\sigma_{\mathrm{v}}, \sigma_{\mathrm{A}}, ss_1 \cdot ss')}$$

S-Ite-F

$$\frac{\neg[\![e]\!]^{\sigma_{\mathrm{v}}}_{\sigma_{\mathrm{A}}}}{(\sigma_{\mathrm{v}}, \sigma_{\mathrm{A}}, \textbf{if } (e) \textbf{ then } \{ss_1\} \textbf{ else } \{ss_2\}; ss') \rightarrow_{\mathrm{s}} (\sigma_{\mathrm{v}}, \sigma_{\mathrm{A}}, ss_2 \cdot ss')}$$

S-Loop-T

$$\frac{[\![e]\!]^{\sigma_{\mathrm{v}}}_{\sigma_{\mathrm{A}}}}{(\sigma_{\mathrm{v}}, \sigma_{\mathrm{A}}, \textbf{while } (e) \textbf{ do } \{ss\}; ss') \rightarrow_{\mathrm{s}} (\sigma_{\mathrm{v}}, \sigma_{\mathrm{A}}, ss \cdot \textbf{while } (e) \textbf{ do } \{ss\}; ss')}$$

S-Loop-F

$$\frac{\neg[\![e]\!]^{\sigma_{\mathrm{v}}}_{\sigma_{\mathrm{A}}}}{(\sigma_{\mathrm{v}}, \sigma_{\mathrm{A}}, \textbf{while } (e) \textbf{ do } \{ss\}; ss') \rightarrow_{\mathrm{s}} (\sigma_{\mathrm{v}}, \sigma_{\mathrm{A}}, ss')}$$

Figure 5.4.: Operational semantics of our sequential programming language

- For all $x, y \in \mathbb{S}_M$ we have $x \oplus_M y \in \mathbb{S}_M$ (Closure)
- For all $x, y, z \in \mathbb{S}_M$ we have $x \oplus_M (y \oplus_M z) = (x \oplus_M y) \oplus_M z$ (Associativity)
- For all $m \in \mathbb{S}_M$ we have $m \oplus_M \mathbf{1}_M = \mathbf{1}_M \oplus_M m = m$ (Identity)

Additionally, we say a variable v, respectively an array A, is *read-only in* P if no assignment of the form $v := \cdots$, respectively $A[e] := \cdots$, occurs in P.

Definition 1. *Let M be a monoid, P a program, n a natural number and $\texttt{in}, \texttt{out}$ arrays of type $\mathsf{Array}(\mathbb{S}_M)$ such that $\texttt{in}$ is read-only in P. The program P* computes an M-prefix sum of length n from `in` to `out` for an initial state $\mathcal{S}$ *if P terminates for $\mathcal{S}$ and for each final array store σ_A we have $\sigma_\mathrm{A}(\texttt{out})(k) = \bigoplus_{M,\, 0 \le i \le k} \sigma_\mathrm{A}(\texttt{in})(i)$ for all $0 \le k < n$.*

The program P implements an M-prefix sum of length n from `in` to `out` *if P computes an M-prefix sum of length n from $\texttt{in}$ to $\texttt{out}$ for every initial state.*

The definitions for the computation and implementation of an exclusive prefix sum are defined analogously. We can establish whether a program *computes* a prefix sum for a given input by running the program. However, determining whether a program *implements* a prefix sum is equivalent to functional verification. A program implements a prefix sum if it computes the expected result for all monoids.

Generic Prefix Sums We now extend our programming language with a fresh *generic type* $\mathbb{S}_X$, a new operator $\oplus_X : \mathbb{S}_X \times \mathbb{S}_X \to \mathbb{S}_X$, and a distinguished literal value $\mathbf{1}_X$ of type $\mathbb{S}_X$. The intention of this generic type is to model an arbitrary monoid $X = (\mathbb{S}_X, \oplus_X)$ with identity $\mathbf{1}_X$. Because our generic type is fresh there are no operators that can convert elements of type $\mathbb{S}_X$ into other program types; the only valid operator on values of this type is $\oplus_X$. In particular, because of the typing rules, it is impossible to use constants or variables of type $\mathbb{S}_X$ as conditions (Boolean expressions) or array indices (integer expressions). Even equality testing for values of $\mathbb{S}_X$ is impossible.

A program that uses $\mathbb{S}_X$ is a *generic program*. A generic program cannot be executed directly. Instead, it must first be instantiated with respect to a specific type in a similar fashion to a generic method in Java or a templated method in C++.

Definition 2. *Let P be a generic program and M a monoid. We write $P[M]$ to denote the program that is identical to P except that every occurrence of $\mathbb{S}_X$, $\oplus_X$ and $\mathbf{1}_X$ is replaced with $\mathbb{S}_M$, $\oplus_M$ and $\mathbf{1}_M$, respectively. We refer to the process of obtaining $P[M]$ from P as a* monoid substitution.

Figure 5.5 gives the formal rules for monoid substitution. The substitution is defined recursively over the structure of the program. Since it is impossible for elements of $\mathbb{S}_X$ to appear in Boolean expressions or array indexing expressions we omit these cases from the recursion. Let P be a generic program and M a monoid. If P is well-typed then $P[M]$ is also a well-typed program. This follows by induction on the structure of P using the typing rules and the definition of monoid substitution.

There is a capturing problem if M already occurs in P prior to the monoid substitution. In this case, we cannot distinguish between uses of M that were already present in P or introduced through the substitution. We handle this problem by choosing a monoid M' isomorphic to M such that M' does not occur in P. We then define $P[M]$ to be $P[M']$. For ease of presentation, we still refer to the monoid M' as M. For example, consider the generic program P below and its monoid substitution using $M = (\mathsf{Int}, +)$ with identity 0. In this case (labelled $P[\mathsf{Int}]$), we can no longer distinguish between the original integer literal, variables and addition and those introduced by the monoid substitution. We avoid this by substituting with respect to $M' = (\mathsf{Int}', +')$ with identity $0'$ (labelled $P[\mathsf{Int}']$). Avoiding this problem means that monoid substitution has a well-defined inverse. That is, given an instantiated program, $P[M]$ we can recover the original generic program P. We will require this property in our proof of soundness and completeness in order to relate a generic program instantiated with an arbitrary monoid M with the instantiation of the program using the interval of summations.

P

```
vars:
  A : 𝕊_X[];
  v : 𝕊_X; w : 𝕊_X;
  n : Int; m : Int;
main:
  n := 0;
  w := 1_X;
  m := n + 1;
  v := A[m] ⊕_X w;
```

$P[\mathsf{Int}]$

```
vars:
  A : Int[];
  v : Int; w : Int;
  n : Int; m : Int;
main:
  n := 0;
  w := 0;
  m := n + 1;
  v := A[m] + w;
```

$P[\mathsf{Int}']$

```
vars:
  A : Int'[];
  v : Int'; w : Int';
  n : Int; m : Int;
main:
  n := 0;
  w := 0';
  m := n + 1;
  v := A[m] +' w;
```

Definition 3. *Let P be a generic program and M a monoid. Then P* implements a generic prefix sum of length n *if* `in` *and* `out` *are of type* $\mathsf{Array}(\mathbb{S}_X)$ *and for every monoid M, the monoid substitution $P[M]$ implements an M-prefix sum of length n.*

The definition for the implementation of a generic exclusive prefix sum is defined analogously.

$$(\textbf{vars} : \mathsf{decl}; \textbf{main} : \mathsf{stmts};)[M] = \textbf{vars} : \mathsf{decl}[M]; \textbf{main} : \mathsf{stmts}[M];$$

$$(\mathsf{name} : \mathsf{type})[M] = \begin{cases} \mathsf{name} : \mathbb{S}_M & \text{if type is } \mathbb{S}_X \\ \mathsf{name} : \mathsf{type} & \text{otherwise} \end{cases}$$
$$(\mathsf{name} : \mathsf{type}[])[M] = \begin{cases} \mathsf{name} : \mathbb{S}_M[] & \text{if type is } \mathbb{S}_X \\ \mathsf{name} : \mathsf{type}[] & \text{otherwise} \end{cases}$$
$$(\mathsf{decl}; \mathsf{decl})[M] = \mathsf{decl}[M]; \mathsf{decl}[M]$$

$$(\varepsilon)[M] = \varepsilon$$
$$(s; ss)[M] = s[M]; ss[M]$$

$$(\mathsf{name} := \mathsf{expr})[M] = \mathsf{name} := \mathsf{expr}[M]$$
$$(\mathsf{arrayname}[\mathsf{expr}] := \mathsf{expr})[M] = \mathsf{arrayname}[\mathsf{expr}] := \mathsf{expr}[M]$$
$$(\textbf{if } (\mathsf{expr}) \textbf{ then } \{\mathsf{stmts}\} \textbf{ else } \{\mathsf{stmts}\})[M] = \textbf{if } (\mathsf{expr}) \textbf{ then } \{\mathsf{stmts}[M]\} \textbf{ else } \{\mathsf{stmts}[M]\}$$
$$(\textbf{while } (\mathsf{expr}) \textbf{ do } \{\mathsf{stmts}\})[M] = \textbf{while } (\mathsf{expr}) \textbf{ do } \{\mathsf{stmts}[M]\}$$

$$(\text{constant literal})[M] = \begin{cases} \mathbf{1}_M & \text{if the literal is } \mathbf{1}_X \\ \text{constant literal} & \text{otherwise} \end{cases}$$
$$(\mathsf{name})[M] = \mathsf{name}$$
$$(\mathsf{arrayname}[\mathsf{expr}])[M] = \mathsf{arrayname}[\mathsf{expr}]$$
$$(\mathsf{expr}\ op\ \mathsf{expr})[M] = \begin{cases} \mathsf{expr}[M] \oplus_M \mathsf{expr}[M] & \text{if } op \text{ is } \oplus_X \\ \mathsf{expr}\ op\ \mathsf{expr} & \text{otherwise} \end{cases}$$

Figure 5.5.: Monoid substitution: these rules instantiate a generic program using $\mathbb{S}_X$ with respect to a monoid M to yield an executable program

5.2.4. The Interval of Summations

We now formalise the observation that a correct generic prefix sum can only work by combining contiguous summation intervals. Our key insight is that a generic prefix sum can only rely on the properties of a monoid: closure, associativity and the existence of an identity element. In particular, additional properties such as commutativity, idempotence or distributivity with respect to another operator cannot be exploited.

We begin with an informal argument that this is the case. Consider a correct prefix sum that does not combine contiguous summation intervals. Then at some point of the execution a summation is formed that is not a contiguous summation interval. For example, there is a 'gap' in the summation (e.g., $\texttt{in}[0] \oplus \texttt{in}[2]$, missing $\texttt{in}[1]$) or an element is added multiple times (e.g., $\texttt{in}[0] \oplus \texttt{in}[1] \oplus \texttt{in}[1]$). Assume that this non-contiguous summation will be part of the final output of the prefix sum. Then since the prefix sum is correct the non-contiguous summation must be made contiguous to meet the functional specification. That is, the gap must be filled with the correct term or the element added multiple times must be cancelled. For example, $\texttt{in}[1]$ must be added in the middle of $\texttt{in}[0] \oplus \texttt{in}[2]$, or $\texttt{in}[1]$ must be cancelled from the right-hand side of $\texttt{in}[0] \oplus \texttt{in}[1] \oplus \texttt{in}[1]$. However, this contradicts the assumption that the prefix sum is defined with respect to a monoid. For example, filling the gap relies on the operator being commutative to reorder the terms into the expected result (e.g., reordering $\texttt{in}[0] \oplus \texttt{in}[2] \oplus \texttt{in}[1]$ into $\texttt{in}[0] \oplus \texttt{in}[1] \oplus \texttt{in}[2]$) and canceling elements relies on every element of the monoid having an inverse. Therefore any correct generic prefix sum relying only on the properties of a monoid *must* only combine contiguous summation intervals.

This observation does not preclude prefix sums that *exploit* additional properties. For example, a prefix sum defined with respect to a commutative monoid or a group. We are aware of work by Sergeev that examines prefix sums that exploit the identity $x \oplus y \oplus y = x$ (such as the exclusive-or operator) and hence rely on a self-inverse property [Ser13]. However, we do not know of any further results that exploit richer properties. All prefix sums that we have examined in GPU data-parallel primitive libraries are defined with respect to a monoid.

Definition 4. *The* interval of summations monoid I *has the elements*

$$\mathbb{S}_I \triangleq \{(i_1, i_2) \in \mathsf{Int} \times \mathsf{Int} \mid i_1 \leq i_2\} \cup \{\mathbf{1}_I, \top\}$$

and a binary operator $\oplus_I$ *defined by:*

$$\begin{aligned}
\mathbf{1}_I \oplus_I x = x \oplus_I \mathbf{1}_I &= x \quad \textit{for all } x \in \mathbb{S}_I \\
\top \oplus_I x = x \oplus_I \top &= \top \quad \textit{for all } x \in \mathbb{S}_I \\
(i_1, i_2) \oplus_I (i_3, i_4) &= \begin{cases} (i_1, i_4) & \textit{if } i_2 + 1 = i_3 \\ \top & \textit{otherwise.} \end{cases}
\end{aligned}$$

This operator allows us to combine summation intervals. Adding an empty summation (to either side) of a summation interval has no effect. Contiguous summation intervals can be joined into a larger interval. Finally, the treatment of $\top$ captures the informal argument above: using only the properties of a monoid it is not possible to transform a non-contiguous summation into a contiguous interval. Notice that $\top$ is an *absorbing element*, or *annihilating element* of I: once a summation has become non-contiguous there can be no return.

It is straightforward to check that I defines a monoid. For all $x, y \in \mathbb{S}_I$ the binary operation $x \oplus_I y$ yields either $\mathbf{1}_I$, $\top$ or an abstract interval (a, b) for some pair of integers a, b. In all cases the result is a member of $\mathbb{S}_I$ so closure is satisfied. For all $x, y, z \in \mathbb{S}_I$ we observe that associativity is satisfied by considering cases. If at least one of x, y, z is $\top$ then $x \oplus_I (y \oplus_I z) = (x \oplus_I y) \oplus_I z = \top$ because $\top$ is an absorbing element. Otherwise, if $x = \mathbf{1}_I$ then $x \oplus_I (y \oplus_I z) = \mathbf{1}_I \oplus_I (y \oplus_I z) = y \oplus_I z = (\mathbf{1}_I \oplus_I y) \oplus_I z = (x \oplus_I y) \oplus_I z$; and this holds similarly if y or z is the identity. Finally, suppose $x = (a, b)$, $y = (c, d)$ and $z = (e, f)$. If $b+1 \neq c$ or $d+1 \neq e$ then at least one of the additions will yield $\top$ and hence $x \oplus_I (y \oplus_I z) = (x \oplus_I y) \oplus_I z = \top$. Otherwise we have $x \oplus_I (y \oplus_I z) = (a, b) \oplus_I ((c, d) \oplus_I (e, f)) = (a, b) \oplus_I (c, f) = (a, f) = (a, d) \oplus_I (e, f) = ((a, b) \oplus_I (c, d)) \oplus_I (e, f) = (x \oplus_I y) \oplus_I z$. Finally, the identity element $\mathbf{1}_I$ satisfies the identity property by definition.

We now define a condition on initial program states. For a given natural number n, the *singleton condition* ensures that the first n elements of `in` are abstracted in the interval monoid by appropriate singleton intervals. That is, the kth element of `in` should be set to (k, k) which abstractly represents the value of `in`$[k]$ under any monoid. All other array elements and variables of type $\mathbb{S}_I$ have values that are not known to be summation intervals.

Definition 5 (Singleton condition for sequential programs)**.** *Let P be a generic program with* `in` *of type* $\mathsf{Array}(\mathbb{S}_X)$*, σ_{v} be a variable store and σ_{A} be an array store. An initial state of $P[I]$ with variable store σ_{v} and array store σ_{A} satisfies the* singleton condition for n *if:*

1. *for all v of type $\mathbb{S}_X$ in P we have $\sigma_{\mathrm{v}}(v) = \top$; and*

2. for all A of type $\mathsf{Array}(\mathbb{S}_X)$ *in P and $k \in \mathsf{Int}$,*

$$\sigma_{\mathrm{A}}(A)(k) = \begin{cases} (k,k) & \text{if } A = \textit{in} \text{ and } k \in 0,\ldots,n-1 \\ \top & \text{otherwise.} \end{cases}$$

5.2.5. Soundness and Completeness

We are now equipped to state our main theorem formally. This theorem shows that the interval monoid is a sound and complete abstraction for generic prefix sums: a program implements a generic prefix sum (i.e., is functionally correct) if and only if it implements a prefix sum when instantiated with the interval monoid. In fact, this theorem states a stronger result because it only considers initial program states (of the interval monoid instantiation) that satisfy the singleton condition rather than all initial program states.

Theorem 5.1 (seq). *Let P be a generic program and n a natural number. Then,*

$P[I]$ computes an I-prefix sum of length n for every initial state satisfying the singleton condition for n

$$\Longleftrightarrow$$

P implements a generic prefix sum of length n.

Our proof uses a simulation argument: given a generic program P we can simulate any (concrete) execution of $P[M]$ for any monoid M by an (abstract) execution of $P[I]$, and vice versa, whilst relating the states encountered in each of the executions. We begin by formalising the relation between concrete and abstract array stores and then lifting this to program states.

Definition 6 (Reification for array stores)**.** *Let M be a monoid. Let* $\mathsf{ArrayStore}$ *denote the type of array stores. The reification function* $\mathrm{Reify}_M : \mathbb{S}_I \times \mathsf{ArrayStore} \to \mathcal{P}(\mathbb{S}_M)$ *is defined as follows*

$$\begin{aligned} \mathrm{Reify}_M((i_1,i_2),\sigma_{\mathrm{A}}) &= \{\textstyle\bigoplus_{M,\, i_1 \le i \le i_2} \sigma_{\mathrm{A}}(\textit{in})(i)\} \\ \mathrm{Reify}_M(\mathbf{1}_I,\sigma_{\mathrm{A}}) &= \{\mathbf{1}_M\} \\ \mathrm{Reify}_M(\top,\sigma_{\mathrm{A}}) &= \mathbb{S}_M \end{aligned}$$

That is, Reify_M maps an abstract summation of the interval monoid to the set containing the corresponding concrete summation in the monoid M and maps an unknown summation, represented by $\top$ in the interval monoid, to the full set of elements of M.

Definition 7 (Reification for program states)**.** *Let P be a generic program and M be a monoid. Let $\mathsf{State}_{P[M]}$ and $\mathsf{State}_{P[I]}$ denote the set of all program states of $P[M]$ and $P[I]$, respectively. Then*

$$\mathrm{Reify}_M : \mathsf{State}_{P[I]} \to \mathcal{P}(\mathsf{State}_{P[M]})$$

is defined by $(\sigma'_{\mathrm{v}}, \sigma'_{\mathrm{A}}, ss') \in \mathrm{Reify}_M(\sigma_{\mathrm{v}}, \sigma_{\mathrm{A}}, ss)$ if and only if

- *for all v of type T in P*

$$\begin{aligned} &\sigma'_{\mathrm{v}}(v) = \sigma_{\mathrm{v}}(v) && \text{if } T \neq \mathbb{S}_X \\ &\sigma'_{\mathrm{v}}(v) \in \mathrm{Reify}_M(\sigma_{\mathrm{v}}(v), \sigma_{\mathrm{A}}) && \text{if } T = \mathbb{S}_X \end{aligned}$$

- *for all A of type $\mathsf{Array}(T)$ in P and $k \in \mathsf{Int}$*

$$\begin{aligned} &\sigma'_{\mathrm{A}}(A)(k) = \sigma_{\mathrm{A}}(A)(k) && \text{if } T \neq \mathbb{S}_X \\ &\sigma'_{\mathrm{A}}(A)(k) \in \mathrm{Reify}_M(\sigma_{\mathrm{A}}(A)(k), \sigma_{\mathrm{A}}) && \text{if } T = \mathbb{S}_X \end{aligned}$$

- *there exists a generic program Q such that $ss = Q[I]$ and $ss' = Q[M]$.*

We refer to the conditions imposed by these definitions on program states including variable stores and array stores as the *reify condition*. As a final preparation for the simulation proof we show that all non-generic expressions evaluate identically in concrete and abstract program states related by reification.

Lemma 5.2 (seq). *Let P be a generic program, M a monoid, $\mathcal{S}_I = (\sigma_{\mathrm{v}}, \sigma_{\mathrm{A}}, ss)$ a program state of $P[I]$, $\mathcal{S}_M = (\sigma'_{\mathrm{v}}, \sigma'_{\mathrm{A}}, ss') \in \mathrm{Reify}_M(\mathcal{S}_I)$ a program state of $P[M]$, and $e : T$ a well-typed expression. If $T \neq \mathbb{S}_X$ then $[\![e]\!]^{\sigma_{\mathrm{v}}}_{\sigma_{\mathrm{A}}} = [\![e]\!]^{\sigma'_{\mathrm{v}}}_{\sigma'_{\mathrm{A}}}$.*

Proof. First, we observe that if the expression e is not of type $\mathbb{S}_X$ then e is the same under any monoid substitution. This is by induction on the structure of e and immediate by the definition of monoid substitution where we note that: (i) in the case of a constant literal that $\mathbf{1}_X$ is the only literal of type $\mathbb{S}_X$; and (ii) in the case of a binary operation that *op* cannot be $\oplus_X$ (since, by design, this is the only binary operator that yields a result of type $\mathbb{S}_X$).

The proof now follows by induction on the structure of e. In the case of a constant literal this is immediate by the definition of expression evaluation. In the case of a variable v we have $\sigma'_{\mathrm{v}}(v) = \sigma_{\mathrm{v}}(v)$ by the definition of Reify_M when $T \neq \mathbb{S}_X$. In the case of an

array expression $A[e']$ we have $A : \mathsf{Array}(T)$ and $e' : \mathsf{Int}$ by T-Array. By the induction hypothesis, $[\![e']\!]^{\sigma_\mathrm{v}}_{\sigma_\mathrm{A}} = [\![e']\!]^{\sigma'_\mathrm{v}}_{\sigma'_\mathrm{A}}$; let us call this index m. Then, by the definition of Reify_M when $T \neq \mathbb{S}_X$ we have $\sigma'_\mathrm{A}(A)(m) = \sigma_\mathrm{A}(A)(m)$. Finally, in the case of a binary operation $e_1 \mathbin{op} e_2$, since op cannot be $\oplus_X$ (by the argument used above) it must be the case that e_1 and e_2 are not of type $\mathbb{S}_X$ and we apply the induction hypothesis to each side. □

Simulation We now prove our simulation result.

Lemma 5.3 (seq) (one-step simulation)**.** *Let P be a generic program, M a monoid, $\mathcal{S}_I = (\sigma_\mathrm{v}, \sigma_\mathrm{A}, ss)$ a program state of $P[I]$, and $\mathcal{S}_M = (\sigma'_\mathrm{v}, \sigma'_\mathrm{A}, ss') \in \mathrm{Reify}_M(\mathcal{S}_I)$ a program state of $P[M]$. Then,*

- *if $\mathcal{S}_I \to_\mathrm{s} \mathcal{S}'_I$ then there exists $\mathcal{S}'_M \in \mathrm{Reify}_M(\mathcal{S}'_I)$ of $P[M]$ such that $\mathcal{S}_M \to_\mathrm{s} \mathcal{S}'_M$, and*
- *if $\mathcal{S}_M \to_\mathrm{s} \mathcal{S}'_M$ then there exists $\mathcal{S}'_I$ of $P[I]$ such that $\mathcal{S}_I \to_\mathrm{s} \mathcal{S}'_I$ and $\mathcal{S}'_M \in \mathrm{Reify}_M(\mathcal{S}'_I)$.*

Proof. Let Q be the generic program such that $Q[I] = ss$ and $Q[M] = ss'$, the program components of $\mathcal{S}_I$ and $\mathcal{S}_M$, respectively. This is always possible by the renaming condition on monoid substitution discussed above.

We begin by showing that whatever the form of Q that the same rule from the operational semantics applies in both $\mathcal{S}_I$ and $\mathcal{S}_M$. If the first statement of Q is an assignment to a variable or array then this is immediate. In the case of a conditional or loop with guard e then by Lemma 5.2 (seq), since $e : \mathsf{Bool}$, we have that e will evaluate identically in $\mathcal{S}_I$ and $\mathcal{S}_M$. Hence the same rule for conditional or loop execution applies.

It remains to show that the conditions on the stores of $\mathcal{S}'_I$ and $\mathcal{S}'_M$ are satisfied. If the applied rule was one of S-Ite-T, S-Ite-F, S-Loop-T, S-Loop-F, this is immediate, as the stores are identical before and after the steps.

For S-Assign, there are two cases to consider. Let $v := e$ be the first statement of Q. By T-Assign, both v and e are the same type T.

- If T is not of type $\mathbb{S}_X$ in Q then, by Lemma 5.2 (seq), the expression e will evaluate identically in $\mathcal{S}_I$ and $\mathcal{S}_M$. Hence the updated variable stores in $\mathcal{S}'_I$ and $\mathcal{S}'_M$ satisfy the reify condition.
- Otherwise, if T is of type $\mathbb{S}_X$ in Q, then by the typing rules and associativity of $\oplus_X$ the expression e must be of the form $x_0 \oplus_X \cdots \oplus_X x_k$ where each x_i (for $0 \leq i \leq k$) is either (a) the identity element $\mathbf{1}_X$, (b) a variable or (c) an array element.

  By the assumption that the reify condition holds for $\mathcal{S}_I$ and $\mathcal{S}_M$, (a) for each $x_i = \mathbf{1}_X$ we have $\mathbf{1}_M \in \mathrm{Reify}_M(\mathbf{1}_I, \sigma_\mathrm{A})$, (b) for each variable v among x_i we have

$\sigma'_{\mathrm{v}}(v) \in \mathrm{Reify}_M(\sigma_{\mathrm{v}}(v), \sigma_{\mathrm{A}})$, and (c) for each $A[e']$ among x_i we have $\sigma'_{\mathrm{A}}(A)(m) \in \mathrm{Reify}_M(\sigma_{\mathrm{A}}(A)(m), \sigma_{\mathrm{A}})$ where $m = [\![e']\!]^{\sigma_{\mathrm{v}}}_{\sigma_{\mathrm{A}}} = [\![e']\!]^{\sigma'_{\mathrm{v}}}_{\sigma'_{\mathrm{A}}}$ (since the index expression e' evaluates identically in $\mathcal{S}_I$ and $\mathcal{S}_M$ by Lemma 5.2 (seq)). Consider the resulting summation $[\![e]\!]^{\sigma_{\mathrm{v}}}_{\sigma_{\mathrm{A}}}$ under $\mathcal{S}_I$ and $[\![e]\!]^{\sigma'_{\mathrm{v}}}_{\sigma'_{\mathrm{A}}}$ under $\mathcal{S}_M$. We must show that $[\![e]\!]^{\sigma'_{\mathrm{v}}}_{\sigma'_{\mathrm{A}}} \in \mathrm{Reify}_M([\![e]\!]^{\sigma_{\mathrm{v}}}_{\sigma_{\mathrm{A}}}, \sigma_{\mathrm{A}})$.[3] By the definition of $\oplus_I$ there are three cases:

1. Case $[\![e]\!]^{\sigma_{\mathrm{v}}}_{\sigma_{\mathrm{A}}} = \mathbf{1}_I$ if and only if all $x_i = \mathbf{1}_X$ (for $0 \leq i \leq k$). Hence $[\![\sigma'_{\mathrm{v}}]\!]^{\sigma'_{\mathrm{A}}}_{=} \mathbf{1}_M$ and the reify condition holds.
2. Case $[\![e]\!]^{\sigma_{\mathrm{v}}}_{\sigma_{\mathrm{A}}} = \top$. In this case there must be some x_i whose value under $\mathcal{S}_I$ is $\top$ or some pair x_i, x_{i+1} whose abstract intervals are not adjacent. Then the reify condition holds trivially since $\mathrm{Reify}_M(\top, \sigma_{\mathrm{A}}) = \mathbb{S}_M$ and, by the property of monoid closure, $[\![e]\!]^{\sigma'_{\mathrm{v}}}_{\sigma'_{\mathrm{A}}} \in \mathbb{S}_M$.
3. Case $[\![e]\!]^{\sigma_{\mathrm{v}}}_{\sigma_{\mathrm{A}}}$ is an abstract interval. In this case, let us ignore occurrences of identity elements since, by definition, they cannot affect the resulting summation. Then, every successive pair of terms x_i, x_{i+1} (for $0 \leq i \leq k-1$) is a pair of adjacent abstract intervals under $\mathcal{S}_I$ that, by assumption, map to a concrete (adjacent) summation interval under $\mathcal{S}_M$. Therefore, by the definition of $\oplus_I$ and $\oplus_M$, $[\![e]\!]^{\sigma'_{\mathrm{v}}}_{\sigma'_{\mathrm{A}}} \in \mathrm{Reify}_M([\![e]\!]^{\sigma_{\mathrm{v}}}_{\sigma_{\mathrm{A}}}, \sigma_{\mathrm{A}})$, as required.

For S-Array, let $A[e_1] := e_2$ be the first statement of Q. By Lemma 5.2 (seq), the indexing expression e_1 evaluates identically in $\mathcal{S}_I$ and $\mathcal{S}_M$. That is, the same array element is updated by each application of the rule. There are two cases to consider, i.e., A is or is not of type $\mathsf{Array}(\mathbb{S}_X)$ in Q. These cases are identical to those of S-Assign. □

Lemma 5.4 (seq) (multiple-step simulation)**.** *Let P be a generic program, M a monoid, and n a natural number. If $\mathcal{S}_M$ is a program state of $P[M]$ then*

- *there exists an initial state $\mathcal{S}_I$ of $P[I]$ satisfying the singleton condition for n such that $\mathcal{S}_M \in \mathrm{Reify}_M(\mathcal{S}_I)$, and*
- *if $\mathcal{S}_I \rightarrow_{\mathrm{s}}^{+} \mathcal{S}'_I$ then there exists $\mathcal{S}'_M \in \mathrm{Reify}_M(\mathcal{S}'_I)$ of $P[M]$ such that $\mathcal{S}_M \rightarrow_{\mathrm{s}}^{+} \mathcal{S}'_M$, and*
- *if $\mathcal{S}_M \rightarrow_{\mathrm{s}}^{+} \mathcal{S}'_M$ then there exists $\mathcal{S}'_I$ of $P[I]$ such that $\mathcal{S}_I \rightarrow_{\mathrm{s}}^{+} \mathcal{S}'_I$ and $\mathcal{S}'_M \in \mathrm{Reify}_M(\mathcal{S}'_I)$.*

Proof. Let $\mathcal{S}_M = (\sigma'_{\mathrm{v}}, \sigma'_{\mathrm{A}}, P[M])$ be an initial state of $P[M]$. The proof is by induction on the number of steps where we apply Lemma 5.3 (seq) after first establishing that an initial state $\mathcal{S}_I$ of $P[I]$ exists that satisfies the singleton condition for n such that $\mathcal{S}_M \in \mathrm{Reify}_M(\mathcal{S}_I)$. Let $\mathcal{S}_I = (\sigma_{\mathrm{v}}, \sigma_{\mathrm{A}}, P[I])$ with

[3] Strictly we require the condition for σ_{A} of $\mathcal{S}'_I$, but since we require `in` to be read-only there is no difference.

- for all v of type T in P,

$$\sigma_{\mathrm{v}}(v) = \begin{cases} \sigma'_{\mathrm{v}}(v) & \text{if } T \neq \mathbb{S}_X \\ \top & \text{if } T = \mathbb{S}_X \end{cases}$$

- for all A of type $\mathsf{Array}(T)$ in P and $k \in \mathsf{Int}$,

$$\sigma_{\mathrm{A}}(A)(k) = \begin{cases} (k, k) & \text{if } A = \mathtt{in} \text{ and } k \in \{0, \ldots, n-1\} \\ \top & \text{if } A = \mathtt{in} \text{ and } k \notin \{0, \ldots, n-1\} \\ \top & \text{if } A \neq \mathtt{in} \text{ and } T = \mathbb{S}_X \\ \sigma'_{\mathrm{A}}(A)(k) & \text{otherwise} \end{cases}$$

That $\mathcal{S}_I$ satisfies the singleton condition for n and that we have $\mathcal{S}_M \in \mathrm{Reify}_M(\mathcal{S}_I)$ is now immediate. □

Proof (Theorem 5.1 (seq)). The $\Leftarrow$-direction is trivial because I is a monoid.

For the $\Rightarrow$-direction, let M be a monoid and $\mathcal{S}_M$ an initial state of $P[M]$. By Lemma 5.4 (seq) there exists an initial state $\mathcal{S}_I$ of $P[I]$ such that $\mathcal{S}_I$ satisfies the singleton condition and $\mathcal{S}_M \in \mathrm{Reify}_M(\mathcal{S}_I)$. All executions from $\mathcal{S}_I$ are terminating because $P[I]$ implements an I-prefix sum. By Lemma 5.4 (seq), every terminating execution $\mathcal{S}_I \to_{\mathrm{s}}^+ \mathcal{S}'_I$ has a corresponding terminating execution $\mathcal{S}_M \to_{\mathrm{s}}^+ \mathcal{S}'_M$ such that $\mathcal{S}'_M \in \mathrm{Reify}_M(\mathcal{S}'_I)$. Since $P[I]$ computes an I-prefix sum we have for the array store σ_{A} of $\mathcal{S}'_I$ that $\sigma_{\mathrm{A}}(\mathtt{out})(k) = \bigoplus_{I,\, 0 \leq i \leq k} \sigma_{\mathrm{A}}(\mathtt{in})(i) = (0, k)$ for all $k \in \{0, \ldots, n-1\}$. Hence, by the definition of reification for array stores yielding σ'_{A} of $\mathcal{S}'_M$ we have $\sigma'_{\mathrm{A}}(\mathtt{out})(k) = \bigoplus_{M,\, 0 \leq i \leq k} \sigma_{\mathrm{A}}(\mathtt{in})(i)$ for all $k \in \{0, \ldots, n-1\}$. □

5.3. Data-Parallel Setting

We now extend our results to data-parallel programs, including GPU kernels, in which multiple threads execute the same program and synchronise using barriers. We formalise an interleaving semantics that follows the MPI programming model [GLS99] in which threads can synchronise at syntactically distinct barriers.

5.3.1. Syntax, Typing and Semantics

Figure 5.6 extends the syntax of Figure 5.2 with a *barrier synchronisation* statement. We highlight the differences from the sequential semantics. A data-parallel program addition-

program	::=	**vars**: decl;
		threads: Int;
		main: stmts;
decl	::=	name : type
	|	arrayname : type[]
	|	decl ; decl
stmts	::=	ε
	|	stmt ; stmts
stmt	::=	name := expr
	|	arrayname[expr] := expr
	|	**if** (expr) **then** {stmts} **else** {stmts}
	|	**while** (expr) **do** {stmts}
	|	**barrier**
expr	::=	constant literal
	|	name
	|	arrayname[expr]
	|	expr *op* expr

Figure 5.6.: Syntax for a simple data-parallel language

ally declares the number of threads that will execute (**threads**: Int) the statement body. The typing rule for barrier statements extends the rules of Figure 5.3.

$$\frac{}{\Gamma \vdash \textbf{barrier} : \mathsf{Unit}} \text{ (T-Barrier)}$$

Let P be a program executed by N threads. Then the finite set of thread identifiers is defined by $D = \{0, \ldots, N-1\}$. A data-parallel program state $\mathcal{S}$ for P is a tuple:

$$(\sigma_{\mathrm{A}}, K)$$

where σ_{A} is an array store and K is a mapping of thread identifiers to (variable store, sequence of statement)-pairs such that for each $t \in D$ and variable store σ_{v} of $K(t)$ we have $\sigma_{\mathrm{v}}(\mathit{tid}) = t$. The variable *tid* gives the identity of a thread and must occur read-only in every program P. An *initial state of* P is a program state where $K(t) = ss$ for every $t \in D$ and ss is the body of the program (declared using **main** : ss).

The rules of Figure 5.7 define the evolution of a data-parallel program state where $\rightarrow_{\mathrm{s}}$ is the relation as defined in Figure 5.4. The semantics is an interleaving semantics (K-Step) with barrier synchronisation (K-Barrier): threads can be scheduled in any order but stall at barriers. The rule K-Step allows any thread to make individual forward progress by executing a single statement, except in the case when the thread has reached a barrier statement. In this case the thread stalls until K-Barrier holds, which ensures

K-Step

$$\frac{K(t) = (\sigma_{\mathrm{v}}, ss) \qquad (\sigma_{\mathrm{v}}, \sigma_{\mathrm{A}}, ss) \rightarrow_{\mathrm{s}} (\sigma'_{\mathrm{v}}, \sigma'_{\mathrm{A}}, ss') \qquad K' = K[t \mapsto (\sigma'_{\mathrm{v}}, ss')]}{(\sigma_{\mathrm{A}}, K) \rightarrow_{\mathrm{k}} (\sigma'_{\mathrm{A}}, K')}$$

K-Barrier

$$\frac{\left(\forall t : \exists \sigma_{\mathrm{v}} : \bigvee \begin{array}{c} (\exists ss : K(t) = (\sigma_{\mathrm{v}}, \mathbf{barrier}; ss) \wedge K'(t) = (\sigma_{\mathrm{v}}, ss)) \\ (K(t) = (\sigma_{\mathrm{v}}, \varepsilon) \qquad\qquad \wedge K'(t) = (\sigma_{\mathrm{v}}, \varepsilon)) \end{array}\right) \quad \exists t, \sigma_{\mathrm{v}}, ss : K(t) = (\sigma_{\mathrm{v}}, \mathbf{barrier}; ss)}{(\sigma_{\mathrm{A}}, K) \rightarrow_{\mathrm{k}} (\sigma_{\mathrm{A}}, K')}$$

Figure 5.7.: Operational semantics of our kernel programming language, extending the sequential rules of Figure 5.4

that every thread has either reached a barrier (not necessarily the *same* syntactic barrier) or has terminated. Permitting synchronisation at syntactically distinct barriers is *less restrictive* than the requirements for barrier synchronisation under CUDA and OpenCL, which require that all threads must synchronise at the same syntactic barrier and furthermore, if the barrier is in a loop then every thread must reach the barrier having executed the same number of loop iterations [BCD+12]. Proving our results in this more general setting is simpler and means that our results are more widely applicable. It is straightforward to restrict our results to the stricter requirements of CUDA and OpenCL. The terminating condition of K-Step models an implicit barrier at the end of each thread execution. The additional condition that at least one thread has reached a barrier ensures that K-Barrier and termination are mutually exclusive.

Given an initial state $\mathcal{S}_0$ of P, an *execution of* P is a finite or infinite sequence of program states:

$$\mathcal{S}_0 \rightarrow_{\mathrm{k}} \mathcal{S}_1 \rightarrow_{\mathrm{k}} \cdots \rightarrow_{\mathrm{k}} \mathcal{S}_i \rightarrow_{\mathrm{k}} \mathcal{S}_{i+1} \rightarrow_{\mathrm{k}} \cdots$$

We will use $\mathcal{S} \rightarrow_{\mathrm{k}}^{+} \mathcal{S}'$ to denote the transitive closure of the relation $\rightarrow_{\mathrm{k}}$. The definitions for maximal and terminating executions are defined as in the sequential setting. Due to the non-deterministic choice of thread t in K-Step there may be multiple maximal executions, contrary to the sequential setting.

Prefix Sums and Generic Prefix Sums The definition for computing and implementing a prefix sum is defined as in the sequential case (Definition 1) with P interpreted as a data-parallel program. To cater for generic data-parallel programs we extend monoid substitution in Figure 5.8; the generic type does not affect barrier statements. A generic

$$
\begin{array}{rcl}
(\textbf{vars} : \mathsf{decl}; \textbf{threads} : \mathsf{number}; \textbf{main} : \mathsf{stmts};)[M] & = & \textbf{vars} : \mathsf{decl}[M]; \\
 & & \textbf{threads} : \mathsf{number}; \\
 & & \textbf{main} : \mathsf{stmts}[M]; \\
(\textbf{barrier})[M] & = & \textbf{barrier}
\end{array}
$$

Figure 5.8.: Extended monoid substitution rules

prefix sum is defined as in the sequential case (Definition 3) with P interpreted as a data-parallel program.

Interval Monoid The definition for the interval of summations monoid (Definition 4) is unchanged. We extend the singleton condition to data-parallel programs by taking into account that there is now a variable store per thread; the condition for the array store is the same as in the sequential case (Definition 5).

Definition 8 (Singleton condition for data-parallel programs)**.** *Let P be a generic data-parallel program with* ***in*** *of type* $\mathsf{Array}(\mathbb{S}_X)$*, σ_{v} be a variable store and σ_{A} be an array store. An initial state of (σ_{A}, K) of $P[I]$ satisfies the* singleton condition for n *if:*

1. *for all $t \in D$ and v of type $\mathbb{S}_X$ in P we have $\sigma_{\mathrm{v}}(v) = \top$ with σ_{v} the variable store of $K(t)$; and*
2. *for all A of type* $\mathsf{Array}(\mathbb{S}_X)$ *in P and $k \in \mathsf{Int}$,*

$$
\sigma_{\mathrm{A}}(A)(k) = \begin{cases} (k, k) & \text{if } A = \textbf{\textit{in}} \text{ and } k \in 0, \ldots, n-1 \\ \top & \text{otherwise.} \end{cases}
$$

5.3.2. Soundness and Completeness

The structure of our data-parallel results closely follows our sequential results. We restate the main soundness and completeness theorem for data-parallel programs and reuse the same proof strategy, a simulation argument. The principal difference is catering for multiple threads with separate variable store and statement components; this affects our definition of reification for program states. Following this, adapting the lemmas used for proving our main theorem is straightforward.

Theorem 5.1 (par). *Let P be a generic data-parallel program and n a natural number. Then,*

$P[I]$ computes an I-prefix sum of length n for every initial state satisfying the singleton condition for n

$\iff$

P implements a generic prefix sum of length n.

Definition 9 (Reification for data-parallel program states)**.** *Let P be a generic data-parallel program and M be a monoid. Let us overload $\mathsf{State}_{P[M]}$ and $\mathsf{State}_{P[I]}$ to denote the set of all data-parallel program states of $P[M]$ and $P[I]$, respectively. Then*

$$\mathrm{Reify}_M : \mathsf{State}_{P[I]} \to \mathcal{P}(\mathsf{State}_{P[M]})$$

is defined by $(\sigma'_\mathrm{A}, K') \in \mathrm{Reify}_M(\sigma_\mathrm{A}, K)$ if and only if

- *for all $t \in D$ and v of type T in P, where σ_v and σ'_v are the variable stores of $K(t)$ and $K'(t)$, respectively:*

$$\begin{aligned} &\sigma'_\mathrm{v}(v) = \sigma_\mathrm{v}(v) && \text{if } T \neq \mathbb{S}_X \\ &\sigma'_\mathrm{v}(v) \in \mathrm{Reify}_M(\sigma_\mathrm{v}(v), \sigma_\mathrm{A}) && \text{if } T = \mathbb{S}_X \end{aligned}$$

- *for all A of type $\mathsf{Array}(T)$ in P and $k \in \mathsf{Int}$*

$$\begin{aligned} &\sigma'_\mathrm{A}(A)(k) = \sigma_\mathrm{A}(A)(k) && \text{if } T \neq \mathbb{S}_X \\ &\sigma'_\mathrm{A}(A)(k) \in \mathrm{Reify}_M(\sigma_\mathrm{A}(A)(k), \sigma_\mathrm{A}) && \text{if } T = \mathbb{S}_X \end{aligned}$$

- *for all $t \in D$, where ss and ss' are the statements of $K(t)$ and $K'(t)$, respectively, there exists a generic program Q such that $ss = Q[I]$ and $ss' = Q[M]$.*

Lemma 5.2 (par). *Let P be a generic data-parallel program, M a monoid, $\mathcal{S}_I = (\sigma_\mathrm{A}, K)$ a program state of $P[I]$, $\mathcal{S}_M = (\sigma'_\mathrm{A}, K') \in \mathrm{Reify}_M(\mathcal{S}_I)$ a program state of $P[M]$, and $e : T$ a well-typed expression. For all $t \in D$, with σ_v and σ'_v the variable stores of $K(t)$ and $K'(t)$, respectively, if $T \neq \mathbb{S}_X$ then $[\![e]\!]^{\sigma_\mathrm{v}}_{\sigma_\mathrm{A}} = [\![e]\!]^{\sigma'_\mathrm{v}}_{\sigma'_\mathrm{A}}$.*

Proof. The proof is by induction on the structure of e using the same observations as in the sequential version of this lemma (Lemma 5.2 (seq)) and the data-parallel definition of reification. □

Lemma 5.3 (par) (one-step simulation)**.** *Let P be a generic data-parallel program, M a monoid, $\mathcal{S}_I = (\sigma_\mathrm{A}, K)$ a program state of $P[I]$, and $\mathcal{S}_M = (\sigma'_\mathrm{A}, K') \in \mathrm{Reify}_M(\mathcal{S}_I)$ a program state of $P[M]$. Then,*

- *if* $\mathcal{S}_I \to_{\mathrm{k}} \mathcal{S}'_I$ *then there exists* $\mathcal{S}'_M \in \mathrm{Reify}_M(\mathcal{S}'_I)$ *of* $P[M]$ *such that* $\mathcal{S}_M \to_{\mathrm{k}} \mathcal{S}'_M$*, and*
- *if* $\mathcal{S}_M \to_{\mathrm{k}} \mathcal{S}'_M$ *then there exists* $\mathcal{S}'_I$ *of* $P[I]$ *such that* $\mathcal{S}_I \to_{\mathrm{k}} \mathcal{S}'_I$ *and* $\mathcal{S}'_M \in \mathrm{Reify}_M(\mathcal{S}'_I)$*.*

Proof. By the definition of reification for data-parallel program states, let Q_t be the generic program such that $Q_t[I]$ and $Q_t[M]$ are the program components of $K(t)$ and $K'(t)$, respectively, for each $t \in D$.

If K-Barrier holds then every Q_t for $t \in D$ either (i) begins with a barrier statement or (ii) is empty. By the definition of monoid substitution, which does not change barrier statements, this ensures that K-Barrier applies to both $\mathcal{S}_I$ or $\mathcal{S}_M$. Furthermore, the condition on the stores of the successor states $\mathcal{S}'_I$ and $\mathcal{S}'_M$ are immediately satisfied since K-Barrier preserves stores.

Otherwise, the rule applied must be K-Step for some thread t in which the first statement of Q_t is not a barrier statement. Using Lemma 5.3 (seq), we know that the same rule from the sequential operational semantics applies and that the successor array store and variable store of t satisfy the sequential reification condition. Hence, the data-parallel reification condition on stores is also satisfied since all variable stores not belonging to t are preserved by K-Step. □

Lemma 5.4 (par) (multiple-step simulation)**.** *Let* P *be a generic data-parallel program,* M *a monoid, and* n *a natural number. If* $\mathcal{S}_M$ *is a program state of* $P[M]$ *then*

- *there exists an initial state* $\mathcal{S}_I$ *of* $P[I]$ *satisfying the data-parallel singleton condition for* n *such that* $\mathcal{S}_M \in \mathrm{Reify}_M(\mathcal{S}_I)$*, and*
- *if* $\mathcal{S}_I \to_{\mathrm{k}}^+ \mathcal{S}'_I$ *then there exists* $\mathcal{S}'_M \in \mathrm{Reify}_M(\mathcal{S}'_I)$ *of* $P[M]$ *such that* $\mathcal{S}_M \to_{\mathrm{k}}^+ \mathcal{S}'_M$*, and*
- *if* $\mathcal{S}_M \to_{\mathrm{k}}^+ \mathcal{S}'_M$ *then there exists* $\mathcal{S}'_I$ *of* $P[I]$ *such that* $\mathcal{S}_I \to_{\mathrm{k}}^+ \mathcal{S}'_I$ *and* $\mathcal{S}'_M \in \mathrm{Reify}_M(\mathcal{S}'_I)$*.*

Proof. Let $\mathcal{S}_M = (\sigma'_{\mathrm{A}}, K')$ be an initial state of $P[M]$. Similar to the sequential version of this lemma, the proof is by induction on the number of steps where we apply Lemma 5.3 (par) after first establishing that an initial state $\mathcal{S}_I$ of $P[I]$ exists that satisfies the data-parallel singleton condition for n such that $\mathcal{S}_M \in \mathrm{Reify}_M(\mathcal{S}_I)$. The definition given in Lemma 5.3 (seq) is easily adapted for this purpose. Let $\mathcal{S}_I = (\sigma_{\mathrm{A}}, K)$ with

- for all $t \in D$ and v of type T in P, where σ_{v} and σ'_{v} are the variable stores of $K(t)$ and $K'(t)$, respectively:

$$\sigma_{\mathrm{v}}(v) = \begin{cases} \sigma'_{\mathrm{v}}(v) & \text{if } T \neq \mathbb{S}_X \\ \top & \text{if } T = \mathbb{S}_X \end{cases}$$

- for all A of type $\mathsf{Array}(T)$ in P and $k \in \mathsf{Int}$,

$$\sigma_{\mathrm{A}}(A)(k) = \begin{cases} (k,k) & \text{if } A = \texttt{in} \text{ and } k \in \{0,\ldots,n-1\} \\ \top & \text{if } A = \texttt{in} \text{ and } k \notin \{0,\ldots,n-1\} \\ \top & \text{if } A \neq \texttt{in} \text{ and } T = \mathbb{S}_X \\ \sigma'_{\mathrm{A}}(A)(k) & \text{otherwise} \end{cases}$$

That $\mathcal{S}_I$ satisfies the data-parallel singleton condition for n and that we have $\mathcal{S}_M \in \mathrm{Reify}_M(\mathcal{S}_I)$ is now immediate. □

5.3.3. Data Race-Freedom

The semantics of Figure 5.7 assume that statements are executed atomically by threads. In particular, a thread may execute a statement involving multiple shared state references in a single step, e.g., $A[i] := A[j]$. This is not valid in practice since such a statement would involve separate, possibly multiple, load and store instructions (and hence memory accesses) between which other threads could interleave. Moreover, even if we refined our semantics to reflect this, we would still need to account for weak memory semantics [HP11, sec. 5.6]. Under weak memory semantics, there is *no global consistent ordering* of memory accesses and each thread may observe writes to shared memory in different orders. Both the CUDA and OpenCL programming models specify weak memory models [NVI12a, Khr13b].

Fortunately, we can avoid reasoning at this level of detail by relying on the *data race-free* (DRF) guarantee [AH90] provided by "all sane memory models" [Vaf14]. The DRF guarantee states that if a program is free of data races then the program can only exhibit *sequentially consistent* behaviours. Under sequential consistency, there is a global consistent ordering of memory accesses and all threads observe the same ordering of memory accesses [Lam79]; that is, an interleaved semantics, as we have given above. In other words, assuming an interleaving semantics is valid if we use it to reason about data race-free programs. The CUDA memory model is not well-specified, but our understanding (from personal communication) is that the intention is for it to provide the DRF guarantee. The OpenCL memory model specifies the DRF guarantee [Khr13b, sec. 3.3.4].

We now formalise what it means for a data-parallel program to exhibit a data race. Furthermore, we show that if a data-parallel program is data race-free then, due to the properties of barrier synchronisation, the result is deterministic. Sequential consistency, by itself, does not guarantee determinism.

Say that we *read from an array element* $A[i]$ in an execution step if $\sigma_{\mathrm{A}}(A)(i)$ is referenced during the evaluation of any expression occurring in the step. Likewise, say that we *write to an array element* $A[i]$ in an execution step if $\sigma_{\mathrm{A}}(A)(i)$ is updated in the step. An *array element* $A[i]$ *is accessed* if it is either read from or written to in the execution step.

Definition 10. *Let* $\mathcal{S}_0 \rightarrow_{\mathrm{k}}^{+} \mathcal{S}_n$ *be an execution of a data-parallel program* P*. The execution has a* data race *if there are steps* $\mathcal{S}_i \rightarrow_{\mathrm{k}} \mathcal{S}_{i+1}$ *and* $\mathcal{S}_j \rightarrow_{\mathrm{k}} \mathcal{S}_{j+1}$ *such that*

- *distinct threads are responsible for these steps,*
- *a common array element is accessed in both execution steps,*
- *at least one of the accesses writes to the array element, and*
- *no application of* K-Barrier *occurs in between the steps.*

A data-parallel program P is data race-free if for every initial state $\mathcal{S}_0$ of P and execution starting from $\mathcal{S}_0$ it holds that the execution does not have a data race.

Theorem 5.5. *Let* P *be a generic data-parallel program and let* M *be a monoid. Then,* $P[M]$ *is data race-free* $\iff$ $P[M']$ *is data race-free for all monoids* M'*.*

Proof. The $\Leftarrow$-direction is trivial because M is a monoid.

For the $\Rightarrow$-direction, observe that neither the control-flow nor the array accesses can be influenced by the choice of M' by the typing rules and genericity of P. □

We now prove our determinism result which informally says that if a data-parallel program is data race-free and terminating then the program is deterministic and all executions will result in the same final program state. This result has a similar flavour to the PUG canonical schedule result [LG10] which proves that if a GPU kernel is racy then all schedules are capable of uncovering a race (although, not necessarily exactly the same race). Similar results also appear in the Test Amplification work of Leung et al. [LGA+12] and the non-interference work of Tripakis et al. [TSL10].

Theorem 5.6. *Let* P *be a data-parallel program that is data race-free and* $\mathcal{S}$ *be an initial program state of* P*. If there exists a finite, maximal execution starting from* $\mathcal{S}$ *with final array store* σ_{A} *then all executions starting from* $\mathcal{S}$ *are finite and for all maximal executions the final array store is* σ_{A}*.*

Proof. Let $\rho = \mathcal{S} \rightarrow_{\mathrm{k}} \mathcal{S}_2 \rightarrow_{\mathrm{k}} \cdots \rightarrow_{\mathrm{k}} \mathcal{S}_n$ be a finite, maximal execution of P. By the operational semantics, every step of the execution is an application of K-Step for some

thread t or K-Barrier. We will say that a *barrier synchronisation occurs* whenever K-Barrier is applied.

Consider the sequence of program states of ρ from the initial program state $\mathcal{S}$ until the first barrier synchronisation occurs; call this subsequence a *barrier interval* (this terminology was introduced by Li and Gopalakrishnan [LG10]). By definition, every step of the barrier interval is an application of K-Step and chooses some interleaving of threads. Because P is data race-free, the execution of a single thread cannot depend on the actions of another until after the barrier synchronisation. In particular, within this barrier interval,

- if a thread t reads from an array element $A[i]$ then no other thread can write to the same array element; and
- if a thread t writes to an array element $A[i]$ then no other thread can access the same array element.

Hence, the values that are read or written to the shared array store σ_{A} are deterministic for every thread. That is, although threads may interleave non-deterministically their individual execution within a barrier interval is deterministic. Therefore the state of the array store is deterministic at the first barrier synchronisation for all executions of P. We further note that since the barrier interval is finite all other interleavings considered by other executions must also be finite.

The proof now follows by applying this argument inductively on the barrier intervals of the execution ρ. □

5.4. Verification Method

We now outline a verification method based on the theoretical results of the previous section. Our results show that the functional correctness of a generic data-parallel prefix sum of length n, implemented as a barrier-synchronising program, can be established by (i) proving that the program is data race-free and (ii) testing that the program behaves correctly with respect to the interval monoid for every initial state with $\mathtt{in} = [(0,0),(1,1),\ldots,(n-1,n-1)]$.[4] In practice, because prefix sum implementations take no inputs except for `in`, this means there is just a single input to test. Another advantage of this hybrid verification method, is that if this test fails then we have a concrete counterexample to the implementation being a correct prefix sum. This is because the interval of summations is itself a monoid.

[4] That is, the result is $[(0,0),(0,1),\ldots,(0,i),\ldots(0,n-1)]$ for an inclusive prefix sum.

Since GPU kernels are barrier-synchronising programs that are *required* to be data race-free — the semantics of an OpenCL or CUDA kernels is undefined if the kernel has a data race [Khr13b, NVI12a] — we can use this observation to establish the functional correctness of a GPU kernel that claims to implement a generic prefix sum for a single input array:

1. Prove that the GPU kernel is data race-free; and
2. Run a single test case using the interval monoid to check functional correctness.

Each step can be established individually but the functional correctness guarantee provided by step 2 is conditional on the result of step 1 (due to Theorem 5.6). We now discuss the main issues in using this verification method in practice.

Generic Prefix Sums The CUDA programming language supports generic functions using function templates. The OpenCL programming language does not support generic functions directly. However, we can still describe a generic prefix sum in OpenCL using the preprocessor. Each prefix sum implementation is equipped with a symbol `TYPE` and macro `OPERATOR` as placeholders for the concrete type and operator. To execute the prefix sum we include an appropriate header with definitions for `TYPE` and `OPERATOR`.

Verification of Data Race-Freedom Step 1 can be discharged to any sound verifier for GPU kernels capable of proving race-freedom (Chapter 2). By Theorem 5.5, we are free to prove race-freedom of an implementation for any choice of binary operator. This step establishes the race-freedom of the kernel for all input lengths n. In our experimental evaluation in Section 5.5 we will use GPUVerify for this purpose.

Encoding the Interval of Summations The dynamic analysis of step 2 requires an encoding of the interval monoid. To check a prefix sum of length n we must encode elements (i, j) for $0 \leq i \leq j < n$ as well as the identity $\mathbf{1}_I$ and unknown $\top$ elements. An element can therefore be encoded using $O(\lg n)$ bits meaning that $O(n \lg n)$ bits are required for the test input and output. In practice, for OpenCL, we can use the `uint2` vector data type.

Soundness, Completeness and Automation Our verification method is sound due to the guarantees given by Theorem 5.1 (par) (soundness and completeness), together with Theorem 5.5 (race-freedom) and Theorem 5.6 (determinism). Additionally, step 1 is sound due to the requirement that a sound technique for verifying race-freedom is used. However,

the soundness of step 2 depends on the integrity of the compiler, driver and hardware implementation on which the test is executed. We can guard against this potential source of unsoundness by testing with respect to multiple platforms using different compilers, drivers and hardware.

Our verification method is only *complete* if the technique used to verify race-freedom is complete. As discussed in Chapter 2, GPUVerify is incomplete and may report false positives.

Our verification method requires some manual effort to provide loop invariants for GPUVerify during step 1. Although, GPUVerify has automatic inference of loop invariants (which we studied in Chapter 3), we disabled this feature when performing our experimental evaluation and provided invariants manually. The dynamic analysis for step 2 is fully automatic.

Handling Extended Programming Language Features Our results are based on simple imperative and data-parallel languages that omit real-world language features such as procedures, unstructured control flow and pointers. We believe our results extend to real GPU programming languages under the condition that data of generic type $\mathbb{S}_X$ is never accessed via pointers with different element types. Without this condition, a program could cast the `out` array into a `char` pointer and write the expected output (for the interval monoid) byte-by-byte; allowing a false negative result for the dynamic analysis (step 2). Figure 5.9 gives an OpenCL kernel that can pass the interval of summations test case, but is not a correct prefix sum. In practice, we do not see prefix sum implementations using these features.

5.5. Experimental Evaluation

We now evaluate the effectiveness of the interval of summations for functional correctness testing for four distinct prefix sum implementations. The main findings of our experiments are:

- GPUVerify was able to verify all prefix sum implementations for all power-of-two problem sizes up to 2^{31} threads.
- The interval of summations allowed dynamic checking of all prefix sum implementations for all power-of-two problem sizes up to 2^{20} threads. This result is beyond the range of an existing method by Voigtländer.

```
__kernel void badprefixsum(
  __local TYPE *out, __local TYPE *in, unsigned n) {
  unsigned tid = get_local_id(0);

  // Set out[tid] = (0,tid) byte-by-byte
  // assuming TYPE is uint2
  //      and a little-endian machine
  char *A = (char *)out;
  A[(2*tid)  *4+0] = (char) 0;
  A[(2*tid)  *4+1] = (char) 0;
  A[(2*tid)  *4+2] = (char) 0;
  A[(2*tid)  *4+3] = (char) 0;
  A[(2*tid+1)*4+0] = (char) ((tid >> 0)  & 0xff);
  A[(2*tid+1)*4+1] = (char) ((tid >> 8)  & 0xff);
  A[(2*tid+1)*4+2] = (char) ((tid >> 16) & 0xff);
  A[(2*tid+1)*4+3] = (char) ((tid >> 24) & 0xff);
}
```

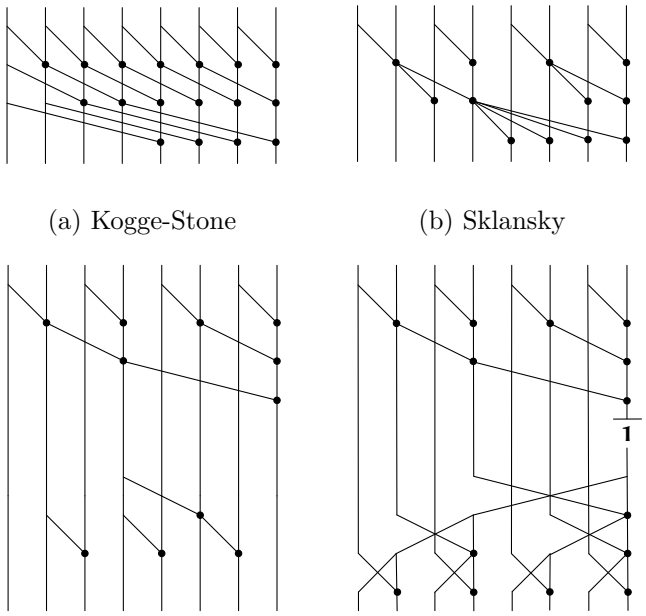
Figure 5.9.: An OpenCL kernel that can pass the interval of summations test case using casting but is not a correct prefix sum

We compare the interval of summations against an existing result, discussed further in Section 5.6, due to Voigtländer [Voi08]. This shows that a generic prefix sum is functionally correct *for all* input lengths if the correct result is computed for all inputs over a set of three elements with respect to two binary operators. Voigtländer further shows that it is sufficient to consider $O(n)$ test cases: $n(n+1)/2$ tests using the first operator and $n-1$ tests using the second operator. We can use this to result to yield a verification method by noting that (i) although Voigtländer's result considers *all* input lengths n it also restricts to the case of consider a particular n; and (ii) Theorem 5.6 allows us to lift this method to race-free barrier-synchronising programs. Similar to the interval of summations dynamic analysis, we can run Voigtländer's test cases dynamically. The functional guarantee provided by Voigtländer's method is conditional on proving race-freedom so the overhead of race checking applies to both methods.

Choice of Prefix Sums Our evaluation considers four distinct prefix sum algorithms, implemented in OpenCL: Kogge-Stone [KS73], Sklansky [Skl60], Brent-Kung [BK82] and Blelloch [Ble93]. The Blelloch algorithm is an exclusive prefix sum; all others are inclusive prefix sums. We surveyed several GPU code repositories — the AMD APP SDK[5], the NVIDIA CUDA SDK[6], and the SHOC [DMM+10], Rodinia [CSLS09] and Parboil [SRS+12] benchmark suites — and found that Kogge-Stone is the most widely-used GPU implementation in practice, Blelloch is used when an exclusive prefix sum is

[5] `http://developer.amd.com/tools/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk/`

[6] `https://developer.nvidia.com/gpu-computing-sdk`



(a) Kogge-Stone (b) Sklansky

(c) Brent-Kung (d) Blelloch

Figure 5.10.: Circuit representations of the prefix sum algorithms for $n = 8$ elements

required, and Brent-Kung is used several times in one large CUDA kernel that computes eigenvalues.

Figure 5.10 gives a circuit diagram for each algorithm for a fixed $n = 8$ input. There is a wire for each input and data flows top-down through the circuit. Each node $\bullet$ performs the binary associative operator on its two inputs and produces an output that passes downwards and also optionally across the circuit (through a diagonal wire).

Experimental Setup As discussed in Section 5.4, we use GPUVerify to prove that each kernel is free for data races. These experiments were performed on a Linux machine with a 1.15 GHz AMD Phenom 9600B 4-core processor using a version of GPUVerify downloaded from the tool web page on 26 June 2013.[7]

We ran our dynamic analysis on five different platforms: two NVIDIA GPUs – a 1.16 GHz GeForce GTX 570 and a 1.15 GHz Tesla M2050; two Intel CPUs – a 2.13 GHz 4 core Xeon E5606 and a 2.67 GHz 6 core Xeon X5650; and one ARM GPU – a 533 MHz 4 core Mali T604. These devices exhibit a range of power and performance characteristics. The NVIDIA GPUs offer a large number of parallel processing elements running at a relatively

[7] `http://multicore.doc.ic.ac.uk/tools/GPUVerify/`

Table 5.2.: Time taken to check data race-freedom using GPUVerify. Times shown are in seconds with 95% confidence intervals.

	Length of Input											
	2^1	2^2	…	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	…	2^{29}	2^{30}	2^{31}
Kogge-Stone	5.68	5.81		5.81	5.97	5.59	5.77	5.74	5.62		6.40	6.42	6.60
	±0.05	±0.03		±0.03	±0.03	±0.04	±0.03	±0.20	±0.07		±0.06	±0.13	±0.10
Sklansky	5.62	6.11		6.58	6.69	6.77	6.67	6.55	6.44		7.35	8.06	7.94
	±0.03	±0.19		±0.17	±0.22	±0.21	±0.11	±0.15	±0.19		±0.14	±0.03	±0.01
Brent-Kung	5.90	6.65		11.39	12.85	10.22	10.30	10.70	12.58		14.87	11.11	16.42
	±0.20	±0.12		±0.22	±0.17	±0.11	±0.06	±0.05	±0.14		±0.29	±0.15	±0.12
Blelloch	5.99	7.39		12.56	13.77	11.97	14.10	11.56	12.25		15.51	12.97	14.84
	±0.08	±0.02		±0.12	±0.07	±0.11	±0.05	±0.37	±0.15		±0.77	±0.35	±0.25

low clock-rate; the Intel CPUs run at a higher clock-rate but exhibit less parallelism; and the ARM GPU is designed for high power-efficiency. We chose to run on multiple platforms to guard against possible unsound results as a result of a particular compiler, driver or hardware configuration.

Each experiment was run with a timeout of 1 hour. All the timing results we present are averages over three runs.

5.5.1. Verification of Data Race-Freedom

Table 5.2 shows the time taken in seconds (with 95% confidence intervals using a two-tailed t-distribution) for GPUVerify to prove race-freedom for each of our prefix sum implementations. By Theorem 5.5, we are free to prove race-freedom of an implementation for any choice of binary operator. In this experiment we use the interval monoid. We varied the problem size for all power-of-two input sizes up to 2^{32}. For brevity, we omit some intermediate results which are in the same order of magnitude for each implementation. In all cases, the analysis with GPUVerify succeeds in *less than twenty seconds*.

5.5.2. Dynamic Analysis

Table 5.4 shows the time taken in seconds for the dynamic analysis using the Interval of Summations (rows labelled **I**) and Voigtländer's method (rows labelled **V**). Timeouts are indicated by 'TO'. We varied the problem size for all power-of-two input sizes up to 2^{20}. Each time is the end-to-end time to run the test case(s) required for each method: i.e., device initialisation, memory allocation of buffers, kernel compilation, copy input data to the device, running the test, copying result data to the host and validating the result. For the Voigtländer method, which requires a quadratic number of test cases to be checked for each problem size, we designed the test program to perform device initialisation, memory

Table 5.3.: Number of test cases required for Voigtländer method

n	**Number of Test Cases** $n(n+1)/2+(n-1)$
2^9	131839
2^{10}	525823
2^{11}	2100223
2^{12}	8394751
2^{13}	33566719
2^{14}	134242303
2^{15}	536920063
2^{16}	2147581951
2^{17}	8590131199
2^{18}	34360131583
2^{19}	137439739903
2^{20}	549757386751

allocation and kernel compilation once and amortised this cost across all test cases.

Our results show that the interval of summations method significantly outperforms the Voigtländer method, across all platforms. This is due to the quadratic growth in the number of test cases for the Voigtländer method; whereas, the interval of summations always requires a single test case. Table 5.3 shows the number of test cases required for increasingly large problem sizes. On all platforms, we found that problem sizes of $n \geq 2^{14}$ exhausted our time limit of 1 hour when using Voigtländer's method.

Our results indicate that we could tackle even larger problem sizes using the interval of summations. In this experiment we limited our problem sizes because we found that problem sizes greater than 2^{20} exceeded resource limits for the ARM platform.

5.6. Related Work

Sheeran and Voigtländer The closest work to the results in this chapter is a paper by Voigtländer [Voi08], which gives two results for sequential generic prefix sums. The first result shows, using relational parametricity [Wad89], that a generic prefix sum is correct if and only if, for all input lengths n, it computes the correct result for the list $[[0], [1], \ldots, [n-1]]$ using list concatenation as the binary operator; that is, it yields the output $[[0], [0,1], \ldots, [0, \ldots, n-1]]$. Voigtländer states that this result is due to earlier unpublished work by Sheeran. This is confirmed in a later paper by Sheeran [She11], which explores the design space of prefix sums. We refer to this result as the Sheeran result. Because the Sheeran result also holds for fixed lengths n it is similar to Theorem 5.1 (seq) for sequential generic prefix sums. However, we cannot use the Sheeran result as the basis for a verification method because $O(n^2 \lg n)$ space is required to represent the

Table 5.4.: Time (in seconds) taken to establish correctness of prefix sum implementations using the interval of summations (**I**) and Voigtländer (**V**) method

		Length of Input									
		2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	...	2^{18}	2^{19}	2^{20}
		NVIDIA GeForce GTX 570									
Kogge-Stone	**V**	19.2	76.2	337.6	1463.9	TO	TO		TO	TO	TO
	I	0.4	0.5	0.3	0.4	0.3	0.4		0.4	0.4	0.4
Sklansky	**V**	18.5	71.8	320.2	1438.3	TO	TO		TO	TO	TO
	I	0.4	0.4	0.4	0.4	0.4	0.4		0.4	0.4	0.4
Brent-Kung	**V**	19.0	69.3	317.3	1454.3	TO	TO		TO	TO	TO
	I	0.5	0.4	0.4	0.4	0.4	0.4		0.4	0.4	0.4
Blelloch	**V**	19.2	75.4	324.2	1595.3	TO	TO		TO	TO	TO
	I	0.5	0.4	0.4	0.4	0.4	0.4		0.4	0.4	0.4
		NVIDIA Tesla M2050									
Kogge-Stone	**V**	14.6	36.1	160.9	653.0	2796.8	TO		TO	TO	TO
	I	0.3	0.3	0.6	0.5	0.5	0.5		0.6	0.5	0.6
Sklansky	**V**	12.2	32.5	149.6	609.3	2601.5	TO		TO	TO	TO
	I	0.3	0.3	0.5	0.5	0.5	0.5		0.6	0.6	0.6
Brent-Kung	**V**	10.9	35.9	165.7	678.1	2889.2	TO		TO	TO	TO
	I	0.3	0.3	0.6	0.5	0.6	0.5		0.6	0.6	0.6
Blelloch	**V**	10.9	36.7	166.0	679.9	2931.8	TO		TO	TO	TO
	I	0.3	0.3	0.5	0.6	0.6	0.6		0.6	0.6	0.6
		Intel Xeon X5650									
Kogge-Stone	**V**	21.1	109.8	660.8	3467.1	TO	TO		TO	TO	TO
	I	1.3	1.3	1.3	1.3	1.3	1.3		1.3	1.3	1.3
Sklansky	**V**	12.2	78.4	404.8	2003.9	TO	TO		TO	TO	TO
	I	1.3	1.3	1.3	1.3	1.3	1.3		1.3	1.3	1.3
Brent-Kung	**V**	12.6	80.4	469.9	2272.5	TO	TO		TO	TO	TO
	I	1.3	1.3	1.3	1.3	1.3	1.3		1.3	1.3	1.3
Blelloch	**V**	12.9	80.7	472.1	2332.5	TO	TO		TO	TO	TO
	I	1.3	1.3	1.3	1.3	1.3	1.3		1.3	1.4	1.3
		Intel Xeon E5606									
Kogge-Stone	**V**	107.0	909.2	TO	TO	TO	TO		TO	TO	TO
	I	1.9	1.9	2.1	2.1	2.0	2.1		2.4	2.7	3.2
Sklansky	**V**	38.7	266.0	1221.5	TO	TO	TO		TO	TO	TO
	I	1.9	1.9	2.0	2.1	2.0	2.0		2.2	2.3	2.4
Brent-Kung	**V**	51.5	409.1	1793.8	TO	TO	TO		TO	TO	TO
	I	2.0	2.0	2.1	2.1	2.1	2.1		2.3	2.3	2.5
Blelloch	**V**	54.3	429.9	1900.3	TO	TO	TO		TO	TO	TO
	I	1.9	2.0	2.1	2.1	2.1	2.1		2.4	2.4	2.6
		ARM Mali T604									
Kogge-Stone	**V**	287.0	1166.1	TO	TO	TO	TO		TO	TO	TO
	I	0.3	0.3	0.3	0.3	0.3	0.3		0.4	0.4	0.5
Sklansky	**V**	287.2	1147.6	TO	TO	TO	TO		TO	TO	TO
	I	0.3	0.3	0.3	0.3	0.3	0.3		0.4	0.4	0.5
Brent-Kung	**V**	287.4	1108.1	TO	TO	TO	TO		TO	TO	TO
	I	0.4	0.4	0.4	0.4	0.4	0.4		0.4	0.5	0.6
Blelloch	**V**	277.0	1105.3	TO	TO	TO	TO		TO	TO	TO
	I	0.4	0.4	0.4	0.4	0.4	0.4		0.4	0.5	0.6

list-of-lists output. In fact, this space complexity only applies to correct algorithms since an incorrect algorithm could apply the concatenation operator arbitrarily many times and require unbounded space. The interval of summations monoid avoids this space complexity by using a compact representation for intervals that exploits our key observation that a correct generic prefix sum should *never* compute a non-contiguous summation. Let L denote the monoid of integer lists under list-concatenation. Then a straightforward homomorphism $\theta : L \to I$ is defined as:

$$\theta([x_1, \ldots, x_m]) = \begin{cases} \mathbf{1}_I & \text{if } m = 0 \\ (x_1, x_m) & \text{if } m > 0 \text{ and } \quad x_{i+1} = x_i + 1 \ (1 \leq i < m) \\ \top & \text{otherwise} \end{cases}$$

The second result, also using relational parametricity, is an elegant proof of the '0-1-2 principle' for prefix sums. In the same spirit as the Knuth 0-1 principle for sorting networks [Knu98, sec. 5.3.4][8], Voigtländer's 0-1-2 principle states that if a generic prefix sum computes the correct result for all ternary sequences (each element of the input is 0, 1, or 2) with every associative operator (defined over the ternary set) then it is correct for all input sequences with all associative operators. In fact, after some reflection, Voigtländer improves this result and shows that it suffices to consider just *two* operators and a quadratic number of ternary sequences: $O(n)$ sequences with the first operator and $n-1$ sequences with the second operator. We refer to this result as the Voigtländer result and we evaluate a method derived from this result in Section 5.5. Table 5.5 compares the Sheeran and Voigtländer with the interval of summations.

Table 5.5.: Asymptotic behaviour of the interval of summations against existing results

	Number of Tests	**Input Space**	**Output Space**
Interval of Summations	$O(1)$	$O(n \lg n)$	$O(n \lg n)$
Sheeran	$O(1)$	$O(n \lg n)$	$O(n^2 \lg n)$
Voigtländer	$O(n^2)$	$O(n \lg n)$	$O(n \lg n)$

Both Sheeran and Voigtländer focus on sequential Haskell implementations of prefix sums; in particular, Sheeran uses Haskell to describe circuit representations of prefix sums. In contrast, our results extend smoothly to handling parallel barrier-synchronising

[8] A sorting network is a circuit representation of an *oblivious* sorting algorithm. A sorting algorithm is *oblivious* if it performs the same sequence of comparisons independent of the values of the input, hence a circuit or network of comparators is an ideal representation. The 0-1 principle states that if a sorting network for n inputs sorts all 2^n Boolean sequences (each element of the input is 0 or 1) into non-decreasing order then it is correct for any sequence over any total ordered value set.

programs, leading to a practical verification method for verifying GPU prefix sum implementations. Due to our imperative, data-parallel setting, we present our theoretical results using a direct simulation argument; we are not aware of any work using relational parametricity to reason about shared-memory data-parallel programs.

Correct-by-Derivation Prefix Sums An alternative approach by Hinze [Hin04] is to construct prefix sums that are correct by derivation. In Hinze's work, rather than verifying candidate implementations for functional correctness, new prefix sum circuits are built from a set of basic primitives and combinators. The correctness of these circuits is ensured by a set of algebraic laws derived for these combinators. Similar to the work of Sheeran, prefix sums are given as HASKELL programs describing circuit layouts. We are not aware of any work that translates this approach to a data-parallel setting.

Barrier Invariants In Chapter 4, we introduced barrier invariants as a method for reasoning about data-dependent GPU kernels [CDK+13]. We used barrier invariants to statically prove functional correctness for three distinct prefix sums: Blelloch, Brent-Kung and Kogge-Stone. Each distinct prefix sum implementation required a different and complicated set of barrier invariants. The interval of summations method is automatic and significantly outperforms the results using barrier invariants. However, the concept of barrier invariants has wider applicability beyond prefix sums.

5.7. Summary

The interval of summations is a new abstraction for reasoning about prefix sums. The key insight of our abstraction is that it precisely captures the behaviour of prefix sums due to the restriction that prefix sums are only defined with respect to monoids. In particular, the interval of summations is a 'tailored abstraction': it is only precise for prefix sums and would not yield interesting results for other programs. We have given theoretical guarantees provided by this abstraction and shown that it can be used as the basis for an automatic and highly-scalable verification method.

6. Conclusions and Open Problems

This thesis set out to build scalable verification techniques for data-parallel programs. To conclude, we examine our contributions against this aim, discuss the limitations of our work and point to future work.

6.1. Contributions

This thesis has made the following original contributions to the field of program verification:

- Chapter 3 gave an empirical study of candidate-based invariant generation in the domain of GPU kernels. We developed new *candidate-generation rules* that enable automatic and precise reasoning for 256 (72%) GPU kernels from a set of 356 benchmarks. To our knowledge this is the largest invariant generation study of its kind (in terms of number of benchmarks). We introduced *refutation engines*, based on the idea of under-approximating analyses, as a mechanism for accelerating the Houdini [FL01] computation of invariants. This yielded a speedup of 1.25× across all benchmarks.

- Chapter 4 addressed the problem of *data-dependent* GPU kernels. We developed *barrier invariants* as a new abstraction and verification technique for catering for this small, but important, class of kernel. Barrier invariants allowed us to capture a functional specification for three distinct prefix sum implementations and race-freedom for a real-world stream compaction example. These examples were previously beyond the scope of existing techniques.

- Chapter 5 developed the *interval of summations*: a novel and bespoke abstraction for reasoning about prefix sums, a key data-parallel primitive. We proved strong properties about the interval of summations showing that the abstraction is both sound and complete (when analysing prefix sum implementations). The power of this result is that it enables a hybrid verification method that allows us to verify the full functional correctness of a given prefix sum implementation using a single test

case. We demonstrated the applicability of this technique by tackling four real-world parallel prefix sum implementations.

Let us evaluate these contributions against the criteria of precision, performance and automation, which we regard as important characteristics of scalability.

Candidate-Based Invariant Generation Study Invariant generation is critical for precision (avoiding false positives) since we cannot in general reason precisely about loops without invariants. However, with a candidate-based invariant generation approach, precision and performance are at odds since generating more candidates is potentially detrimental to performance. We introduced refutation engines as a method for mitigating this problem. Invariant generation is also essential for automation and avoiding the need for programmer annotations.

Barrier Invariants Barrier invariants enable precise reasoning for data-dependent GPU kernels, which cannot be reasoned about using coarser abstractions. Our experimental evaluation shows that the verification method using this abstraction can handle hundreds of threads when using a timeout of 3 hours; this is heavyweight compared to the performance experiments we ran in our candidate-based invariant generation study where we used a timeout of 10 minutes. We did not automate the generation of barrier invariants since the invariants we required were complex and problem-specific. However, we emphasise that the verification technique that we developed does not assume the truth of user-supplied barrier invariants, but checks that they do indeed hold. Therefore, compared to the techniques made by our first contribution, barrier invariants enable (i) precise reasoning for more challenging kernels and (ii) richer properties to be proven, but at a cost of automation. Barrier invariants require expert users to guide their application.

Interval of Summations The interval of summations is a sound and complete abstraction for prefix sums. The abstraction is a precise fit for prefix sums (but, of course, is not a precise abstraction for other kinds of GPU kernel). Our experimental evaluation shows that the hybrid verification method using this abstraction can handle all feasible sizes of four distinct prefix sum implementations (for all power-of-two problem sizes up to 2^{20}) in seconds and with minimal programmer assistance. The interval of summation yields a highly-scalable, precise and automatic verification technique, but is restricted in its application to one class of kernel.

6.2. Limitations

We now discuss limitations of our work including underlying assumptions that may limit the applicability of our contributions.

Unrealistic Benchmarks In this thesis we have been careful to distinguish between real and synthetic programs (Section 1.2). A real program is found 'in the wild' and therefore reflects the idioms and language features employed by real programmers. A key assumption of our work is that the programs we have studied are real and not synthetic. If the programs that we have studied are not a true reflection of the state of GPU programming then we cannot say whether the techniques developed in this thesis have been truly scalable.

To guard against this problem, the benchmarks used in our invariant generation study were gathered from different sources. We collected a large number of kernels from nine distinct sources including software developer kits [AMD, NVI], hand-translated examples [Mic], performance benchmarks [BYF+09, SRS+12, CSLS09, DMM+10, Rig], and compiler-generated benchmarks [GGXS+12]. We noted in Section 3.2, that a minority of these kernels are somewhat synthetic but we believe the majority are a true reflection of GPU programs. The utility of the prefix sum in data-parallel programming is well-established [Ble93] and prefix sums appear in all GPU data-parallel primitive libraries that we have examined so this increases our confidence in the applicability of barrier invariants and the interval of summations.

Other than the kernels in the Rightware benchmarks [Rig], we have not applied our techniques to programs written in industry. We are now pursuing this through collaboration with ARM and Imagination Technologies, where we hope to integrate GPUVerify with their toolchains. As a data point, the GPUVerify development team recently received a bug report (personal communication in March 2014) from a programmer in industry that has attempted to verify a kernel consisting of ten thousand lines of OpenCL code. GPUVerify was unable to reason about the kernel, although, the bug report additionally noted that the kernel had proved problematic even for the AMD and NVIDIA industry compilers.

GPU Kernels Are Tractable Targets GPU kernels exhibit a number of characteristics that enable scalable verification techniques. More specifically, because all threads execute the same templated program (parameterised by thread and group id variables) the two-thread reduction used in the kernel transformation can be employed. Also, GPU kernels do not typically exhibit complex pointer manipulation, dynamic memory allocation

or recursion, which are known difficulties for verification. Hence our results are limited to reasoning about data-parallel GPU kernels and it is not obvious how to extend them beyond this domain.

We respond to this limitation by noting that these restrictions are part of the reason why GPGPU programming has been successful (since they enable a simple mapping to hardware for performance). Therefore, it is unsurprising that we have been able to exploit these characteristics for more general verification purposes. Two of the contributions of this thesis are applicable beyond the domain of GPU kernels. Firstly, the idea of refutation engines is generally applicable to invariant generation techniques based on Houdini. Secondly, the interval of summation results are generally applicable to barrier synchronising programs (such as MPI). More generally, we are not against domain-specific verification. Work in domain-specific languages shows that there is much to be gained by incorporating domain knowledge for optimisation and performance portability. We believe the same is true for verification and point to the interval of summations as a good example.

Verified Programs Are Not Perfect This limitation is an argument against verification, of which there have been many [DLP79, Fet88, Mac01]. The argument boils down to saying that it is meaningless to say that a program is verified. Firstly, there will always be further bugs. Saying that a program is verified is simply a paucity of imagination. Secondly, verifying a program only shows adherence to some specification. Importantly, it does not say whether the specification itself is valid or not. In other words, we can never establish the absolute correctness of a program.

Precisely. It is a mistake to assume that verified programs cannot go wrong. As Fetzer pointed out, verification cannot account for the physical world in which programs operate, for example, a random soft error [Fet88]. We agree with Nelson, who wrote "the message at the successful exit of program verifiers should be changed from *Verified* to *Sorry, can't find any more errors*" [Nel81, p. 4]. We take a pragmatic view: a program verifier is worth using if the benefits outweight the costs. We believe scalable verification techniques are vital for tipping the scale and for making verification a practical reality.

6.3. Future Work

We now discuss possible future directions of research based on the work in this thesis. Some of our suggestions are concerned with making verification more practical for everyday programmers, which we view as a central challenge. We believe programmers are pragmatic and will take up verification tools when there is a tangible benefit, over and beyond the

cost, to using them.

Correct Data-Parallel Primitive Libraries An area where the cost-benefit for verification is favourable is extensively-used library code since the effort of verifying a frequently used library function can be amortised over many possible users. In data-parallel programming, we find that certain primitives are used frequently and many data-parallel primitive libraries exist for GPU programming.[1] Both barrier invariants and the interval of summations can be used to functionally verify parallel prefix sums. Other important primitives include split, compact and sorting operations (discussed briefly in Figure 4.2 of Chapter 4). Barrier invariants should be capable of capturing the functional specification of these primitives. More speculatively, it may be possible to find new abstractions, similar to the interval of summations, that completely capture a class of data-parallel primitive.

Better User-Interfaces for Program Verifiers A critical factor for pragmatic program verification that we have not examined is usability. In prior work, we addressed the problem of reporting meaningful error messages when using GPUVerify [BBC+14]. We believe this could be enhanced by automatically generating concrete test cases that cause failures in a similar fashion to dynamic symbolic execution tools such as GKLEE [LLS+12]. An advantage of this approach is that programmers are well versed in understanding test cases and naturally understand their utility.

Related to the problem of usability, there is a need for rigorous user studies to quantify the effectiveness of verification. The work on the PUG tool by Li and Gopalakrishnan [LG10] reports an interesting case study where PUG was applied to 57 kernels written by a graduate class on GPU programming. This found that PUG was able to find some "serious (but non-obvious) bugs in beginner examples." However, the study did not follow this up by returning the error messages to the students as feedback. A study comparing the effectiveness of dynamic, hybrid and static race analysis techniques would be interesting and informative to conduct.

Verification of Kernels Using Atomics and Fences Kernels using atomics and fences are beyond the scope of the verification techniques in this thesis. The principal problem of atomics is that they relax the definition of data races. Using atomics it is possible for multiple threads to concurrently update a memory location without racing.

[1] Data-parallel primitive libraries for GPU programming include the Thrust Parallel Algorithms Library (`http://thrust.github.io`), the CUDA Data-Parallel Primitives Library (`http://cudpp.github.io`) and the OpenCL Data-Parallel Primitives Library (`https://code.google.com/p/clpp/`).

Hence, atomics are a valid source of non-determinism and so violate the assumption that race-free GPU kernels are deterministic. This means that the *canonical schedule* result of PUG [LG10] (also used by GKLEE and GPUVerify) does not hold. Existing work for analysing atomics in GPU kernels is given by Chiang et al. [CGLR13] and Bardsley and Donaldson [BD14], which extend GKLEE and GPUVerify, respectively. The work of Chiang et al. uses conflict detection to note when alternate schedules must be explored. As with other techniques based on dynamic symbolic execution, this work is better suited to bug-finding than verification. For verification, the work of Bardsley and Donaldson extends the kernel transformation and shared state abstraction to cater for simple uses of atomics. Interestingly, the paper notes two kernels (the `Mandelbrot` kernels in the CUDA SDK) that break the two-thread reduction and so cannot be reasoned about precisely.

Related to the use of atomics are *fences*, which affect the ordering of memory operations, and can be combined with atomics to form lightweight locks for mutual exclusion. Although these idioms are not yet common[2] we expect that kernels using these features to become more prevalent as more complicated applications are accelerated on GPUs. No work to our knowledge has addressed the problem of verifying kernels with these features.

Functional Verification of Kernels Using Floating-Point Our results from Chapter 3 show that race-freedom does not generally require precise reasoning about floating-point computation; in both GPUVerify and GKLEE, floating-point operations are abstracted. This does not hold when we consider wanting to prove functional specifications of kernels. For example, a kernel that computes an image filter may have a specification that each pixel of the output is some floating-point function of its neighbours. The work of KLEE-FP [CCK11a] presents a technique for crosschecking an OpenCL kernel against a SIMD implementation. The SIMD implementation can be seen as the specification against which the OpenCL kernel is checked for equivalence. Because KLEE-FP is based on dynamic symbolic execution the tool is limited to bounded equivalence checking (i.e., equivalence up to a certain size of input). An ideal verifier would allow precise reasoning for floating-point operations and enable verification to scale up to large or even unconstrained problem sizes.

Verification for Performance Tuning Verification can be used to inform the programmer about potential bottlenecks or performance problems. For example, both the PUG [LG10] and GKLEE [LLS+12] tools report possible performance problems: non-

[2] In the benchmarks discussed in Section 3.2 only two kernels were eliminated from the study due to atomics or fences.

coalesced memory accesses, bank conflicts and divergent warp behaviour. In PUG, this is achieved by encoding the performance problem as part of the verification condition; GKLEE checks for these conditions during dynamic symbolic execution. Inferring unnecessary synchronisation barriers is another performance problem that could be detected automatically. Extensions to this work would consider encoding GPU microarchitectural features, such as the size of shared memory, as part of the verification condition.

Kernels from the PolyBench/GPU suite [GGXS+12] were amongst the most difficult to automatically infer invariants due to deep loop nesting and non-trivial loop conditions. The kernels were automatically parallelised and generated from sequential code using polyhedral methods [ALSU06, chap. 11]. Hence it should be possible to exploit the knowledge used by the auto-parallelising tool to automatically generate invariants required for race-freedom. More speculatively, a verifier using the performance tuning techniques above may be able to guide an auto-parallelising tool or autotuning framework to find high-performance implementations without having to execute many candidate implementations.

6.4. Summary

We have developed scalable verification techniques by studying the characteristics of precision, performance and automation in the context of data-parallel programs. Verification cannot build perfect software, but it can be a powerful and complementary technique for building better software. We offer the results in this thesis as a step towards practical verification.

Bibliography

[AH90] Sarita V. Adve and Mark D. Hill. Weak Ordering – A New Definition. In *Proceedings of the 17th International Symposium on Computer Architecture*, ISCA '90, 1990. (Cited on page 155.)

[AKPW83] John R. Allen, Ken Kennedy, Carrie Porterfield, and Joe D. Warren. Conversion of Control Dependence to Data Dependence. In *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, POPL '83, 1983. (Cited on page 31.)

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson Education, Inc, 2nd edition, 2006. (Cited on pages 42 and 173.)

[AMD] AMD. Accelerated Parallel Processing SDK. `http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/`, accessed September 1, 2014. (Cited on pages 50 and 169.)

[And11] Marc Andreessen. Why Software is Eating the World. *The Wall Street Journal*, 2011. (Cited on page 11.)

[BBC+10] Al Bessey, Ken Block, Benjamin Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles-Henri Gros, Asya Kamsky, Scott McPeak, and Dawson R. Engler. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Communications of the ACM*, 53(2):66–75, 2010. (Cited on page 43.)

[BBC+14] Ethel Bardsley, Adam Betts, Nathan Chong, Peter Collingbourne, Pantazis Deligiannis, Alastair F. Donaldson, Jeroen Ketema, Daniel Liew, and Shaz Qadeer. Engineering a Static Verification Tool for GPU Kernels. In *Proceedings of the 26th International Conference on Computer Aided Verification*, CAV '14, 2014. (Cited on pages 13, 30, 31, 34, 37, 39, 43, 53, 59, 88, 113, and 171.)

[BCD+05] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Proceedings of the 4th International Conference on Formal Methods for Components and Objects*, FMCO '05, 2005. (Cited on pages 16 and 31.)

[BCD+11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV '11, 2011. (Cited on page 49.)

[BCD+12] Adam Betts, Nathan Chong, Alastair F. Donaldson, Shaz Qadeer, and Paul Thomson. GPUVerify: a Verifier for GPU Kernels. In *Proceedings of the ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '12, 2012. (Cited on pages 13, 15, 25, 30, 31, 33, 34, 37, 39, 42, 53, 58, 61, 65, 84, 85, 95, 106, 110, 123, and 151.)

[BD14] Ethel Bardsley and Alastair F. Donaldson. Warps and Atomics: Beyond Barrier Synchronization in the Verification of GPU Kernels. In *NASA Formal Methods Workshop*, 2014. (Cited on pages 39, 42, and 172.)

[BDW14] Ethel Bardsley, Alastair F. Donaldson, and John Wickerson. KernelInterceptor: automating GPU kernel verification by intercepting and generalising kernel parameters. In *Proceedings of the International Workshop on OpenCL*, IWOCL '14, 2014. (Cited on pages 14 and 60.)

[BH05] Jonathan P. Bowen and Michael G. Hinchey. Ten Commandments Revisited: A Ten-Year Perspective on the Industrial Application of Formal Methods. In *Proceedings of the 10th International Workshop on Formal Methods for Industrial Critical Systems*, FMICS '05, 2005. (Cited on page 12.)

[BHM14] Stefan Blom, Marieke Huisman, and Matej Mihelčić. Specification and verification of GPGPU programs. *Science of Computer Programming*, 2014. (Cited on pages 28 and 128.)

[BK82] Richard P. Brent and H. T. Kung. A Regular Layout for Parallel Adders. *IEEE Transactions on Computers*, 31(3):260–264, 1982. (Cited on pages 118, 131, and 160.)

[BL05] Michael Barnett and K. Rustan M. Leino. Weakest-Precondition of Unstructured Programs. In *Proceedings of the 6th ACM Workshop on Program analysis for Software Tools and Engineering*, PASTE '05, 2005. (Cited on pages 47, 49, and 69.)

[Ble93] Guy E. Blelloch. Prefix Sums and Their Applications. In John H. Reif, editor, *Synthesis of Parallel Algorithms*. Morgan Kaufmann, 1993. (Cited on pages 88, 90, 93, 131, 160, and 169.)

[BLS05] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# Programming System: An Overview. In *Construction and Analysis of Safe, Secure and Interoperable Smart devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2005. (Cited on page 16.)

[BM07] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation*. Springer, 1st edition, 2007. (Cited on page 16.)

[BOA09] Markus Billeter, Ola Olsson, and Ulf Assarsson. Efficient Stream Compaction on Wide SIMD Many-Core Architectures. In *Proceedings of the Conference on High Performance Graphics*, HPG '09, 2009. (Cited on pages 89 and 131.)

[BPR01] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and Cartesian Abstraction for Model Checking C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 268–283. Springer, 2001. (Cited on pages 80 and 81.)

[Bro87] Frederick P. Brooks, Jr. No Silver Bullet – Essence and Accidents of Software Engineering. *IEEE Computer*, 20(4):10–19, 1987. (Cited on page 12.)

[BSW08] Michael Boyer, Kevin Skadron, and Westley Weimer. Automated Dynamic Analysis of CUDA Programs. In *Proceedings of the Third Workshop on Software Tools for MultiCore Systems*, 2008. (Cited on page 26.)

[BYF+09] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems*, ISPASS '09, 2009. (Cited on pages 51 and 169.)

[CC77] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation

of Fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, POPL '14, 1977. (Cited on pages 81 and 82.)

[CCK11a] Peter Collingbourne, Cristian Cadar, and Paul H. J. Kelly. Symbolic Crosschecking of Floating-Point and SIMD Code. In *Proceedigns of the European Conference on Computer Systems*, EuroSys '11, 2011. (Cited on page 172.)

[CCK11b] Peter Collingbourne, Cristian Cadar, and Paul H. J. Kelly. Symbolic Testing of OpenCL Code. In *Proceedings of the 7th International Haifa Verification Conference on Hardware and Software: Verification and Testing*, HVC '11, 2011. (Cited on page 27.)

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '08, 2008. (Cited on page 123.)

[CDH+09] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A Practical System for Verifying Concurrent C. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, 2009. (Cited on page 18.)

[CDK+13] Nathan Chong, Alastair F. Donaldson, Paul H. J. Kelly, Jeroen Ketema, and Shaz Qadeer. Barrier Invariants: a Shared State Abstraction for the Analysis of Data-Dependent GPU Kernels. In *Proceedings of the ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '13, 2013. (Cited on pages 13, 14, 51, 84, 93, 94, 107, and 166.)

[CDK14] Nathan Chong, Alastair F. Donaldson, and Jeroen Ketema. A Sound and Complete Abstraction for Reasoning About Parallel Prefix Sums. In *Proceedings of the 41st ACM Symposium on Principles of Programming Languages*, POPL '14, 2014. (Cited on pages 13, 14, 65, and 130.)

[CDKQ13] Peter Collingbourne, Alastair F. Donaldson, Jeroen Ketema, and Shaz Qadeer. Interleaving and Lock-Step Semantics for Analysis and Verification of GPU kernels. In *Proceedings of the 22nd European Symposium on Programming*, ESOP '13, 2013. (Cited on pages 32, 39, and 42.)

[CGJ+03] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided Abstraction Refinement for Symbolic Model Checking. *Journal of the ACM*, 50(5):752–794, 2003. (Cited on page 81.)

[CGLR13] Wei-Fan Chiang, Ganesh Gopalakrishnan, Guodong Li, and Zvonimir Rakamarić. Formal Analysis of GPU Programs with Atomics via Conflict-Directed Delay-Bounding. In *NASA Formal Methods Workshop*, 2013. (Cited on page 172.)

[Cla] `clang`: a C language family frontend for LLVM. `http://clang.llvm.org/`, accessed September 1, 2014. (Cited on page 42.)

[CMP04] Ching-Tsun Chou, Phanindra K. Mannava, and Seungjoon Park. A Simple Method for Parameterized Verification of Cache Coherence Protocols. In *Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design*, FMCAD'04, 2004. (Cited on page 128.)

[CSLS09] Shuai Che, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the IEEE International Symposium on Workload Characterization*, 2009. (Cited on pages 51, 160, and 169.)

[DDLM13] Isil Dillig, Thomas Dillig, Boyang Li, and Ken L. McMillan. Inductive Invariant Generation via Abductive Inference. In *Proceedings of the ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '13, 2013. (Cited on pages 80 and 82.)

[DHKR11] Alastair F. Donaldson, Leopold Haller, Daniel Kroening, and Philipp Rümmer. Software Verification Using *k*-Induction. In *Proceedings of the 18th International Conference on Static Analysis*, SAS '11, 2011. (Cited on pages 48 and 49.)

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976. (Cited on page 15.)

[DKK+12] Alastair F. Donaldson, Alexander Kaiser, Daniel Kroening, Michael Tautschnig, and Thomas Wahl. Counterexample-guided abstraction refinement for symmetric concurrent programs. *Formal Methods in System Design*, 41(1), 2012. (Cited on page 18.)

[DKKW11] Alastair F. Donaldson, Alexander Kaiser, Daniel Kroening, and Thomas Wahl. Symmetry-aware Predicate Abstraction for Shared-Variable Concurrent Programs. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV '11, 2011. (Cited on page 18.)

[DKR11] Alastair F. Donaldson, Daniel Kroening, and Philipp Rümmer. Automatic Analysis of DMA Races Using Model Checking and *k*-induction. *Formal Methods in System Design*, 39(1):83–113, 2011. (Cited on page 34.)

[DLP79] Richard A. DeMillo, Richard J. Lipton, and Alan J. Perlis. Social Processes and Proofs of Theorems and Programs. *Communications of the ACM*, 22(5):271–280, 1979. (Cited on pages 11 and 170.)

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008. (Cited on page 49.)

[dMB11] Leonardo de Moura and Nikolaj Bjørner. Satisfiability Modulo Theories: Introduction and Applications. *Communications of the ACM*, 54(9):69–77, 2011. (Cited on page 16.)

[DMM+10] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. The Scalable HeterOgeneous Computing (SHOC) Benchmark Suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, 2010. (Cited on pages 51, 160, and 169.)

[DNS05] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A Theorem Prover for Program Checking. *Journal of the ACM*, 52(3):365–473, 2005. (Cited on page 17.)

[Dow97] Mark Dowson. The Ariane 5 Software Failure. *ACM SIGSOFT Software Engineering Notes*, 22(2):84–, 1997. (Cited on page 11.)

[ECGN01] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001. (Cited on pages 80 and 83.)

[EGK90] O. Eğecioğlu, E. Gallopoulos, and C. Koç. A Parallel Method for Fast and Practical High-order Newton Interpolation. *BIT Numerical Mathematics*, 30:268–288, 1990. (Cited on page 131.)

[Fet88] James H. Fetzer. Program Verification: The Very Idea. *Communications of the ACM*, 31(9):1048–1063, 1988. (Cited on page 170.)

[FJL01] Cormac Flanagan, Rajeev Joshi, and K. Rustan M. Leino. Annotation inference for modular checkers. *Information Processing Letters*, 77(2-4):97–108, 2001. (Cited on pages 48 and 80.)

[FL01] Cormac Flanagan and K. Rustan M. Leino. Houdini, an Annotation Assistant for ESC/Java. In *Proceedings of the International Symposium of Formal Methods Europe*, FME '01, 2001. (Cited on pages 48, 80, 81, and 167.)

[FLL+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, PLDI '02, 2002. (Cited on page 16.)

[Flo67] Robert W. Floyd. Assigning Meanings to Programs. *Mathematical Aspects of Computer Science*, 19:19–32, 1967. (Cited on pages 15, 45, and 47.)

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987. (Cited on page 59.)

[FP13] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — Where Programs Meet Provers. In *Proceedings of the 22nd European Symposium on Programming*, ESOP '13, 2013. (Cited on page 16.)

[GGXS+12] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. Auto-tuning a High-Level Language Targeted to GPU Codes. In *Proceedings of the IEEE Conference on Innovative Parallel Computing*, InPar '12, 2012. (Cited on pages 51, 169, and 173.)

[GK10] Michael Garland and David B. Kirk. Understanding Throughput-Oriented Architectures. *Communications of the ACM*, 53:58–66, 2010. (Cited on page 18.)

[GLS99] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 2nd edition, 1999. (Cited on page 149.)

[God97] Patrice Godefroid. Model Checking for Programming Languages Using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, POPL '97, 1997. (Cited on page 18.)

[GR09] Ashutosh Gupta and Andrey Rybalchenko. InvGen: An Efficient Invariant Generator. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, 2009. (Cited on page 80.)

[GS97] Susanne Graf and Hassen Saïdi. Construction of Abstract State Graphs with PVS. In *Proceedings of the 9th International Conference on Computer Aided Verification*, CAV '97, 1997. (Cited on page 80.)

[Hin04] Ralf Hinze. An Algebra of Scans. In *Mathematics of Program Construction*, Lecture Notes in Computer Science, pages 186–210. Springer, 2004. (Cited on page 166.)

[Hoa69] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969. (Cited on page 15.)

[Hoa03] C. A. R. Hoare. The Verifying Compiler: A Grand Challenge for Computing Research. *Journal of the ACM*, 50(1):63–69, 2003. (Cited on page 12.)

[Hor05] Daniel Horn. Stream Reduction Operations for GPGPU Applications. In Matt Pharr, editor, *GPU Gems 2*. Addison-Wesley, 2005. (Cited on page 88.)

[HP11] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 5th edition, 2011. (Cited on pages 18 and 155.)

[HR04] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning About Systems*. Cambridge University Press, 2nd edition, 2004. (Cited on page 89.)

[HS86] Daniel W. Hillis and Guy L. Steele, Jr. Data Parallel Algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986. (Cited on pages 18 and 121.)

[HSO07] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel Prefix Sum (Scan) with CUDA. In Hubert Nguyen, editor, *GPU Gems 3*. Addison-Wesley, 2007. (Cited on pages 88, 90, 131, and 188.)

[Jac12] Paul Jaccard. The Distribution of the Flora in the Alpine Zone. *New Phytologist*, 11(2):37–50, 1912. (Cited on page 78.)

[JM09] Bertrand Jeannet and Antoine Miné. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, 2009. (Cited on pages 80 and 81.)

[JSS14] Bertrand Jeannet, Peter Schrammel, and Sriram Sankaranarayanan. Abstract Acceleration of General Linear Loops. In *Proceedings of the 41st ACM Symposium on Principles of Programming Languages*, POPL '14, 2014. (Cited on page 82.)

[KD14] Jeroen Ketema and Alastair F. Donaldson. Automatic Termination Analysis for GPU Kernels. In *Proceedings of the 14th International Workshop on Termination*, 2014. (Cited on page 43.)

[Khr13a] Khronos OpenCL Working Group. The OpenCL C Specification (Version 2.0), 2013. (Cited on page 42.)

[Khr13b] Khronos OpenCL Working Group. The OpenCL Specification (Version 2.0), 2013. (Cited on pages 22, 25, 37, 51, 155, and 158.)

[KMM11] Rajesh Karmani, P. Madhusudan, and Brandon Moore. Thread contracts for safe parallelism. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, 2011. (Cited on page 127.)

[Knu74] Donald E. Knuth. Computer Programming as an Art. *Communications of the ACM*, 17(12):667–673, 1974. (Cited on page 11.)

[Knu98] Donald E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, 2nd edition, 1998. (Cited on page 165.)

[KS73] Peter M. Kogge and Harold S. Stone. A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations. *IEEE Transactions on Computers*, C-22(8):786–793, 1973. (Cited on pages 121, 131, and 160.)

[KS08] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer, 1st edition, 2008. (Cited on page 16.)

[KV09] Laura Kovács and Andrei Voronkov. Finding Loop Invariants for Programs over Arrays Using a Theorem Prover. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering*, FASE '09, 2009. (Cited on page 82.)

[Lam79] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, 28(9):690–691, 1979. (Cited on page 155.)

[Lei10] K. Rustan M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR '10, 2010. (Cited on page 16.)

[Lev93] Nancy G. Leveson. An Investigation of the Therac-25 Accidents. *IEEE Computer*, 26:18–41, 1993. (Cited on page 11.)

[LF80] Richard E. Ladner and Michael J. Fischer. Parallel Prefix Computation. *Journal of the ACM*, 27(4):831–838, 1980. (Cited on page 131.)

[LG10] Guodong Li and Ganesh Gopalakrishnan. Scalable SMT-based Verification of GPU Kernel Functions. In *Proceedings of the 18th ACM International Symposium on the Foundations of Software Engineering*, FSE '18, 2010. (Cited on pages 27, 29, 32, 124, 156, 157, 171, and 172.)

[LGA+12] Alan Leung, Manish Gupta, Yuvraj Agarwal, Rajest Gupta, Ranjit Jhala, and Sorin Lerner. Verifying GPU Kernels by Test Amplification. In *Proceedings of the 33rd ACM Conference on Programming Language Design and Implementation*, PLDI '12, 2012. (Cited on pages 28, 29, 124, 134, and 156.)

[LLG12] Peng Li, Guodong Li, and Ganesh Gopalakrishnan. Parametric flows: Automated behavior equivalencing for symbolic analysis of races in CUDA programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '12, 2012. (Cited on page 123.)

[LLG14] Peng Li, Guodong Li, and Ganesh Gopalakrishnan. Practical Symbolic Race Checking of GPU Programs. In *Proceedings of the International Conference*

for High Performance Computing, Networking, Storage and Analysis, SC '14, 2014. (Cited on page 27.)

[LLS+12] Guodong Li, Peng Li, Geoffrey Sawaya, Ganesh Gopalakrishnan, Indradeep Ghosh, and Sreeranga P. Rajan. GKLEE: Concolic Verification and Test Generation for GPUs. In *Proceedings of the 17th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, 2012. (Cited on pages 27, 123, 171, and 172.)

[LMS09] K. Rustan M. Leino, Peter Müller, and Jans Smans. Verification of Concurrent Programs with Chalice. In *Foundations of Security Analysis and Design V*, volume 5705 of *Lecture Notes in Computer Science*, pages 195–222. Springer, 2009. (Cited on pages 18 and 29.)

[LPSZ08] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from Mistakes — A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '08, 2008. (Cited on page 17.)

[Mac01] Donald Mackenzie. *Mechanizing Proof: Computing, Risk, and Trust*. MIT Press, 1st edition, 2001. (Cited on pages 12, 16, and 170.)

[McC04] Steve McConnell. *Code Complete*. Microsoft Press, 2nd edition, 2004. (Cited on pages 11 and 12.)

[McM06] Ken L. McMillan. Lazy Abstraction with Interpolants. In *Proceedings of the 18th International Conference on Computer Aided Verification*, CAV '06, 2006. (Cited on pages 80 and 82.)

[Mic] Microsoft Corporation. C++ AMP Sample Projects for Download (MSDN blog). `http://blogs.msdn.com/b/nativeconcurrency/archive/2012/01/30/c-amp-sample-projects-for-download.aspx`, accessed September 1, 2014. (Cited on pages 51 and 169.)

[MQB+08] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Nainar Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI '08, 2008. (Cited on page 18.)

[NE02] Jeremy W. Nimmer and Michael D. Ernst. Invariant Inference for Static Checking: An Empirical Evaluation. In *Proceedings of the 10th ACM International Symposium on the Foundations of Software Engineering*, FSE '02, 2002. (Cited on page 83.)

[Nel81] Greg Nelson. Techniques for Program Verification. Technical Report CSL-81-10, Xerox Palo Alto Research Center, June 1981. (Cited on page 170.)

[NVI] NVIDIA. CUDA Code Samples. `https://developer.nvidia.com/gpu-computing-sdk`, accessed September 1, 2014. (Cited on pages 51, 56, and 169.)

[NVI09] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. Whitepaper, 2009. (Cited on page 18.)

[NVI12a] NVIDIA. CUDA C Programming Guide (Version 5.0), 2012. (Cited on pages 19, 25, 37, 43, 51, 155, and 158.)

[NVI12b] NVIDIA. CUDA-MEMCHECK: User Manual (Version 5.0), October 2012. (Cited on page 26.)

[PFW13] Nadia Polikarpova, Carlo A. Furia, and Scott West. To Run What No One Has Run Before: Executing an Intermediate Verification Language. In *Proceedings of the 4th International Conference on Runtime Verification*, RV '13, 2013. (Cited on page 71.)

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002. (Cited on page 137.)

[Res02] Research Triangle Institute. The Economic Impacts of Inadequate Infrastructure for Software Testing. Technical Report Planning Report 02-3, National Institute of Standards and Technology, May 2002. (Cited on page 12.)

[Rey02] John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, LICS '02, 2002. (Cited on page 28.)

[Ric53] Henry G. Rice. Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical Society*, 74:358–366, 1953. (Cited on page 16.)

[Rig] Rightware. Basemark CL. `http://www.rightware.com/benchmarking-software/basemark-cl/`, accessed September 1, 2014. (Cited on pages 51 and 169.)

[Ser13] Igor Sergeev. On the complexity of parallel prefix circuits. Technical Report TR13-041, Electronic Colloquium on Computational Complexity, 2013. (Cited on page 143.)

[She11] Mary Sheeran. Functional and dynamic programming in the design of parallel prefix networks. *Journal of Functional Programming*, 21(1):59–114, 2011. (Cited on pages 131 and 163.)

[SHG09] Nadathur Satish, Mark Harris, and Michael Garland. Designing Efficient Sorting Algorithms for Manycore GPUs. In *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing*, IPDPS '09, 2009. (Cited on page 131.)

[SHZO07] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for GPU computing. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH '07, 2007. (Cited on page 88.)

[Skl60] Jack Sklansky. Conditional-Sum Addition Logic. *IRE Transactions on Electronic Computers*, EC-9:226–231, 1960. (Cited on pages 131 and 160.)

[SRS+12] John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen mei W. Hwu. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. Technical Report IMPACT-12-01, University of Illinois at Urbana-Champaign, March 2012. (Cited on pages 51, 160, and 169.)

[Ste96] Bjarne Steensgaard. Points-to Analysis in Almost Linear Time. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, POPL '96, 1996. (Cited on page 42.)

[SZ12] Stephen F. Siegel and Timothy K. Zirkel. Loop Invariant Symbolic Execution for Parallel Programs. In *Proceedings of the 13th international conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI '12, 2012. (Cited on page 128.)

[TDB14] Paul Thomson, Alastair F. Donaldson, and Adam Betts. Concurrency Testing Using Schedule Bounding: An Empirical Study. In *Proceedings of the 19th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, 2014. (Cited on page 18.)

[TSL10] Stavros Tripakis, Christos Stergiou, and Roberto Lublinerman. Checking Non-Interference in SPMD Programs. In *Proceedings of the 2nd USENIX Workshop on Hot Topics in Parallelism*, HotPar '10, 2010. (Cited on pages 29 and 156.)

[U.S13] U.S. Securities and Exchange Commission. In the Matter of Knight Capital Americas LLC, October 2013. `http://www.sec.gov/litigation/admin/2013/34-70694.pdf`, accessed September 1, 2014. (Cited on page 11.)

[Vaf08] Viktor Vafeiadis. Modular fine-grained concurrency verification. Technical Report UCAM-CL-TR-726, University of Cambridge, Computer Laboratory, July 2008. (Cited on page 18.)

[Vaf14] Viktor Vafeiadis. Relaxed Separation Logic (Tutorial at POPL '14), January 2014. `http://www.mpi-sws.org/~viktor/slides/rsl-tutorial.pdf`, accessed September 1, 2014. (Cited on page 155.)

[Voi08] Janis Voigtländer. Much Ado about Two (Pearl): A Pearl on Parallel Prefix Computation. In *Proceedings of the 35th ACM Symposium on Principles of Programming Languages*, POPL '08, 2008. (Cited on pages 160 and 163.)

[Wad89] Philip Wadler. Theorems for Free! In *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, 1989. (Cited on page 163.)

[War12] Henry S. Warren. *Hacker's Delight*. Addison-Wesley, 2nd edition, 2012. (Cited on page 65.)

[ZRQA11] Mai Zheng, Vignesh T. Ravi, Feng Qin, and Gagan Agrawal. GRace: A Low-Overhead Mechanism for Detecting Data Races in GPU Programs. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, 2011. (Cited on page 26.)

A. Permissions

Figure 4.3 reproduces a figure from GPU Gems 3 [HSO07]. Permission granted by M. Harris (author) and Pearson (publisher).

PEARSON

Legal/Permissions
One Lake Street
Upper Saddle River, NJ 07458
Fax: 201-236-3290
Phone: 201-236-3564
Cheryl.Freeman@Pearson.com

Oct 17, 2014

PE Ref # 187258

NATHAN CHONG
Imperial College Longdon
180 Queens Gates
London, UNITED KINGDOM

Dear Mr. Chong:

You have our permission to include content from our text, ***GPU GEMS 3, 1st Ed. by NGUYEN, HUBERT,*** in your PhD dissertation thesis for your Department of Computing studies at IMPERIAL COLLEGE LONDON.

Content to be published electronically on Spiral, Imperial College's online repository:

- Figure 39-9 Stream Compaction Example

Please credit our material as follows:
NGUYEN, HUBERT, GPU GEMS 3, 1st Edition, © 2008. Reprinted by permission of Pearson Education, Inc., Upper Saddle River, NJ

Sincerely,

Cheryl Freeman, Permissions Administrator