Imperial College London
Department of Computing

Symbolic Crosschecking of Data-Parallel Floating Point Code

Peter Cyrus Collingbourne

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, 2012

Declaration of Originality

I hereby declare that this work is my own and that all other work is appropriately acknowledged.

Abstract

In this thesis we present a symbolic execution-based technique for cross-checking programs accelerated using SIMD or OpenCL against an unaccelerated version, as well as a technique for detecting data races in OpenCL programs. Our techniques are implemented in KLEE-CL, a symbolic execution engine based on KLEE that supports symbolic reasoning on the equivalence between expressions involving both integer and floating-point operations.

While the current generation of constraint solvers provide good support for integer arithmetic, there is little support available for floating-point arithmetic, due to the complexity inherent in such computations. The key insight behind our approach is that floating-point values are only reliably equal if they are essentially built by the same operations. This allows us to use an algorithm based on symbolic expression matching augmented with canonicalisation rules to determine path equivalence.

Under symbolic execution, we have to verify equivalence along every feasible control-flow path. We reduce the branching factor of this process by aggressively merging conditionals, if-converting branches into `select` operations via an aggressive phi-node folding transformation.

To support the Intel Streaming SIMD Extension (SSE) instruction set, we lower SSE instructions to equivalent generic vector operations, which in turn are interpreted in terms of primitive integer and floating-point operations.

To support OpenCL programs, we symbolically model the OpenCL environment using an OpenCL runtime library targeted to symbolic execution. We detect data races by keeping track of all memory accesses using a memory log, and reporting a race whenever we detect that two accesses conflict. By representing the memory log symbolically, we are also able to detect races associated with symbolically indexed accesses of memory objects.

We used KLEE-CL to find a number of issues in a variety of open source projects that use SSE and OpenCL, including mismatches between implementations, memory errors, race conditions and compiler bugs.

Acknowledgements

This thesis has been an exercise in serendipity. I would like to thank the following people for appearing at the right time to set me straight on a genuinely interesting research topic after 2 years of meanderings [CK10, CK09]:

- David J. Pearce, for suggesting that I might want to look into SMT solvers;
- Dan Quinlan et al., for introducing me to symbolic execution and allowing me to get experience writing my own symbolic execution tool during my time at LLNL;
- Cristian Cadar, for getting me to switch to KLEE. It was certainly the right choice as KLEE was much more mature than my tool was at that point, and was based on LLVM, which I had already enjoyed spending some spare time hacking on. KLEE provided me with the perfect excuse to hack on LLVM during my studies, culminating in a Silicon Valley job offer.

I would like to thank Cristian Cadar and Daniel Dunbar for writing KLEE and releasing it as open source, and Stefan Bucur and the Cloud9 team for providing their threading extensions to KLEE. Without something to base my work on, I would have had to spend a lot more time reinventing the wheel rather than focusing on research problems.

I would like to thank Alastair Donaldson for our discussions regarding our different approaches to GPU testing and verification.

I would like to thank my supervisors, Cristian Cadar and Paul Kelly, for their advice and guidance.

Finally, I would like to thank my family for their support.

Contents

List of Tables

List of Figures

List of Listings

Chapter 1

Introduction

There exist a number of programming models which allow for improved program performance by exploiting data level parallelism. Such models include SIMD (Single Instruction Multiple Data) and GPGPU (General Purpose Graphics Processing Unit) computing.

Today, most commercial CPU designs include SIMD capabilities, such as the Streaming SIMD Extensions (SSE), 3DNow! and Advanced Vector Extensions (AVX) for x86, NEON for ARM, and AltiVec for PowerPC. GPGPU computing is another popularly supported model, with AMD, NVIDIA, ARM, Intel and Imagination Technologies all providing OpenCL compliant interfaces to the compute capabilities of their GPUs.

The challenge posed to the developer wishing to take advantage of one of these programming models is to develop a correct translation from existing serial code to SIMD or OpenCL enabled data parallel code. While automatic vectorisation is an active area of research [EWO04, LA00, NBBDZ03], the difficulty of reasoning about data dependencies and arithmetic precision means that this translation is still a mostly manual process.

Furthermore, these programming models can be difficult to understand and use correctly. The Intel Instruction Set Reference [Int10a, Int10b] is a 1674 page document describing over 400 machine instructions, over 100 of which are SIMD instructions. The OpenCL 1.1 specification [Khr10] comprises 385 pages of technical documentation describing more than 600 individ-

ual functions. Any programming error in the translation may cause the translated code to act differently from the purportedly equivalent serial version.

Furthermore, because OpenCL is an open standard, each vendor has its own implementation. A developer cannot easily determine that their code is compliant with the OpenCL specification, because he or she may unknowingly be using undocumented quirks of their particular implementation.

In this thesis, we present a crosschecking and data race detection technique based on symbolic execution [Kin75], which provides a systematic way of exploring all feasible paths in a program for inputs up to a certain size. On each explored path, our technique works by building the symbolic expressions associated with the serial and translated data parallel versions of the code, and proving their equivalence. During symbolic execution of OpenCL kernels, we also maintain a log of all memory accesses for use in race detection.

1.1 Floating Point Arithmetic

Floating point arithmetic is a commonly available facility for performing imprecise computation over subsets of the real numbers. The standard for floating point arithmetic is IEEE 754-2008 [IEE08], which defines five floating point formats, of which the two most frequently used are binary32 (commonly known as *single precision*) and binary64 (commonly known as *double precision*). The binary32 format is a 32-bit format which allows for the representation of values between -2^{128} (exclusive) and -2^{-126} (inclusive), and between 2^{-126} (inclusive) and 2^{128} (exclusive) with 23 bits of precision, while binary64 is a 64-bit format which allows for representation of values between -2^{1024} (exclusive) and -2^{-1022} (inclusive), and between 2^{-1022} (inclusive) and 2^{1024} (exclusive) with 52 bits of precision [IEE08, §3.2].

IEEE 754 also contains support for the representation of zeros (both positive and negative – while negative zero is not a real number, it may be obtained, for example, by multiplying a negative number by zero), infinities (both positive and negative), `NaN`s (i.e. uncomputable results) and *denormalised* numbers, which are used to represent small numbers: for single pre-

cision, multiples of 2^{-149} between -2^{-126} (exclusive) and -2^{-149} (inclusive), and between 2^{-149} (inclusive) and 2^{-126} (exclusive) and for double precision, multiples of 2^{-1074} between -2^{-1022} (exclusive) and -2^{-1074} (inclusive), and between 2^{-1074} (inclusive) and 2^{-1022} (exclusive).

IEEE 754 sets out a deterministic algorithm for performing arithmetic operations, such as addition, multiplication and division, on floating point numbers: the operation:

> shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then rounded that intermediate result, if necessary, to fit in the destination's format [IEE08, §5.1]

Thus, a user of a language implementation conforming to the IEEE 754 specification should expect reproducible results.

1.2 SIMD

The SIMD (Single Instruction Multiple Data) capabilities of a processor may be used to perform the same operation on multiple data items using a single machine instruction. Because the processor typically carries out these operations in parallel, data level parallelism is achieved. Common applications of SIMD include image processing, signal processing, computer vision and multimedia. In one experiment [TJLE00], a speedup of up to 5.5x has been observed for SIMD versions of various signal processing and multimedia algorithms on a 3-way superscalar out of order execution machine resembling the Intel Pentium II.

SIMD processors operate on one-dimensional arrays of data known as vectors, and provide several vector registers for this purpose (for example, the first version of Intel SSE provided eight 128-bit vector registers each holding four 32-bit single precision floating point numbers). A typical SIMD instruction will take one or more input vector register operands, and perform an operation elementwise on each operand element, storing the result in an output vector register. For example, Figure 1.1 shows the operation of the Intel SSE instruction `MULPS`.

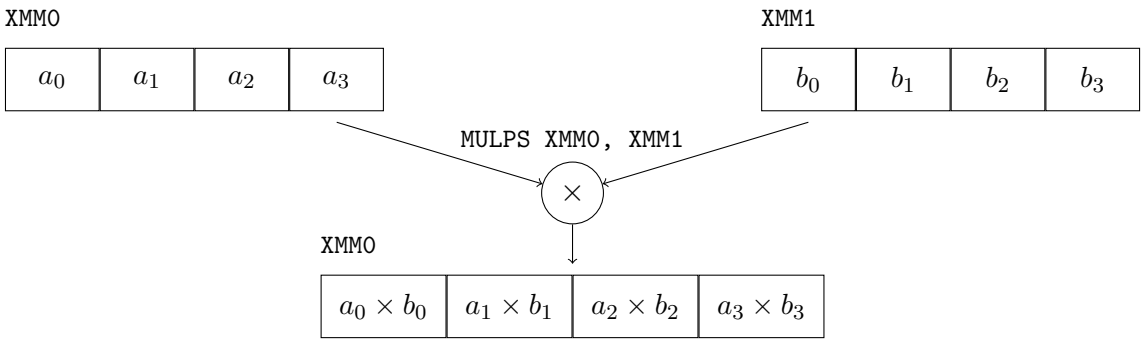

Figure 1.1: The SSE `MULPS` instruction.

1.3 The GPGPU Architecture and OpenCL

General purpose Graphics Processing Units (GPGPU) offer a new commonly available facility for highly parallel computing. GPGPU architectures are most commonly SPMD (Single Program Multiple Data) in nature, an evolution of the GPU's ability to perform multiple 3D rendering calculations in parallel.

OpenCL (Open Computing Language) is an open standard for general purpose parallel programming. OpenCL is designed for heterogeneous architectures, with existing implementations targeting CPUs, GPGPUs, dedicated accelerators and other processors. In order to utilise the computing power of GPGPUs most effectively, OpenCL is based on the SPMD model.

The fundamental unit of execution in OpenCL is the *work-item*, which represents a single invocation of a specified kernel function. A kernel invocation constitutes the parallel execution of a set of work-items, optionally organised into *work-groups*, which can share common resources such as local memory. Each work-item conceptually resides at a point in the kernel invocation's iteration space, referred to as the n-dimensional range, or *NDRange*. Data-level parallelism is achieved by having the kernel function vary the data items accessed depending on the position of the work-item in the iteration space.

Figure 1.2 illustrates a two-dimensional iteration space, and shows how multi-dimensional NDRanges are partitioned into work-groups. Work-items are assigned to work-groups according to a work-group size supplied by the OpenCL program. Work-group sizes are typically

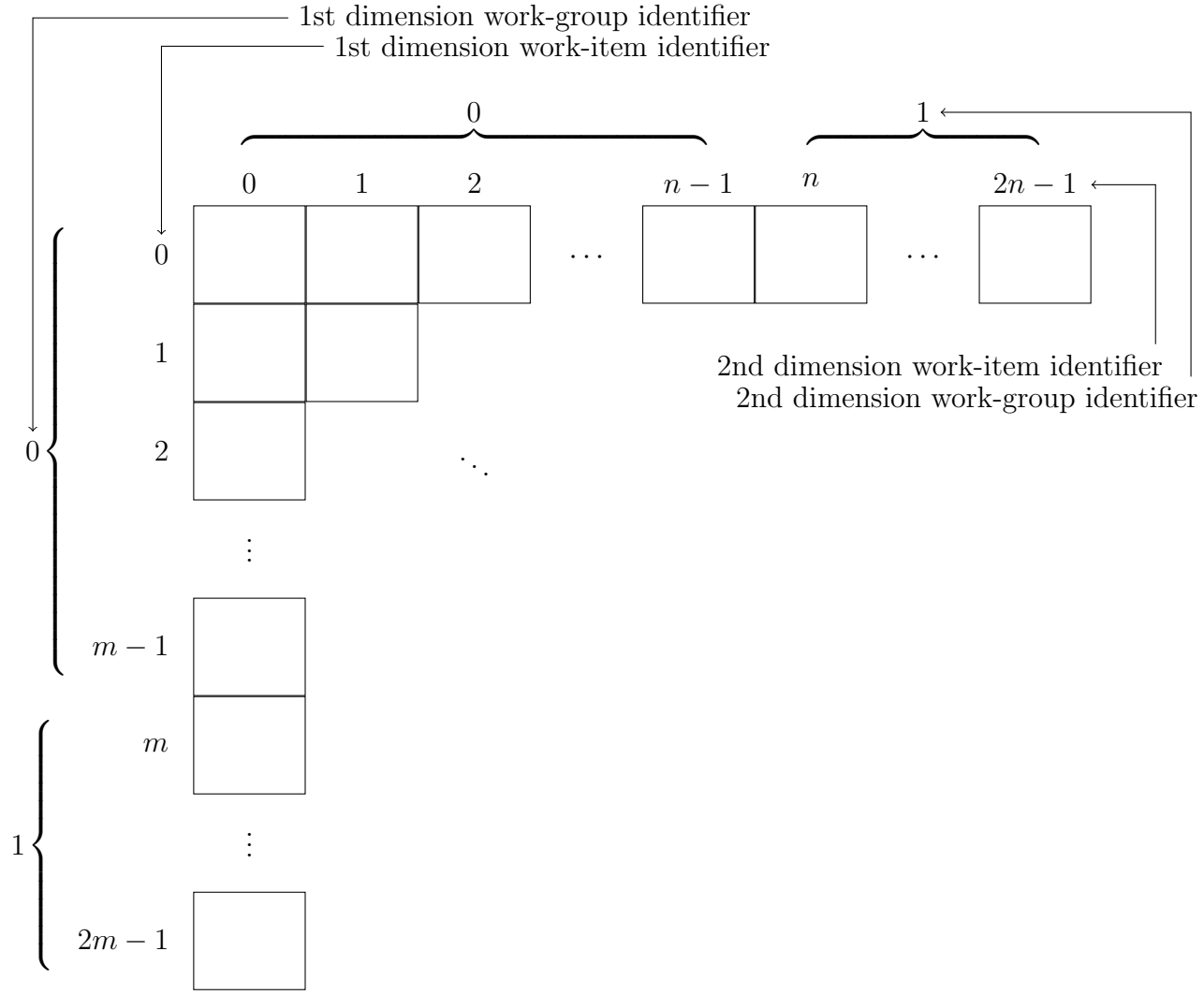

Figure 1.2: A two-dimensional NDRange of work-group size (m, n) and work-item count $(2m, 2n)$. Each grid square represents a single work-item.

chosen based on the size of the problem or the nature of the algorithm but are usually subject to hardware constraints.

1.4 Symbolic Execution and KLEE

At a high level, symbolic execution is a technique that allows the automatic exploration of paths in a program. It works by executing the program on *symbolic* input, which is initially unconstrained. As the program runs, any operations that depend on the symbolic input add constraints on the input. For example, if the program input is represented by variable `x`, than the statement `y = x+3` would add the constraint that $y = x + 3$. Furthermore, whenever a branch that depends on the symbolic input is reached, the technique first checks if both sides are feasible, and if so, it forks execution and follows each side separately, adding the constraint that the branch condition is true on the true side and false on the other side. For example, given the symbolic input `x`, where `x` is unconstrained, the symbolic execution of the branch `if (x == 3)` would result in two paths being explored, one on which $x = 3$ and one on which $x \neq 3$. The conjunction of all constraints encountered on a particular path is referred to as the *path condition*.

Figure 1.3 illustrates the symbolic execution of a simple program. On line 2 we assign an unconstrained symbolic value to the previously declared variable x. On line 4 we branch based on the `if` statement's controlling expression $x > 0$. Since x is unconstrained, both $x > 0$ and $\neg(x > 0)$ are feasible, so execution forks into two paths, one with the constraint $x > 0$ and the other with the constraint $\neg(x > 0)$. Each path executes its own branch of the `if` statement, and both paths reach line 10 of the program, where another `if` statement is encountered. The first path forks again into two paths, because both $x > 10$ and $\neg(x > 10)$ are feasible given $x > 0$. But the second path does not fork, because $x > 10$ is infeasible given $\neg(x > 0)$.

In our work, we use symbolic execution to explore the different paths in the serial and data parallel implementations being tested, and for each pair of paths, we check whether (1) there are no memory errors (these checks are by default performed by KLEE); (2) the outputs computed

```
1  int x;
2  mksymbolic(x);
3
4  if (x > 0) {
5    ...
6  } else {
7    ...
8  }
9
10 if (x > 10) {
11   ...
12 } else {
13   ...
14 }
```

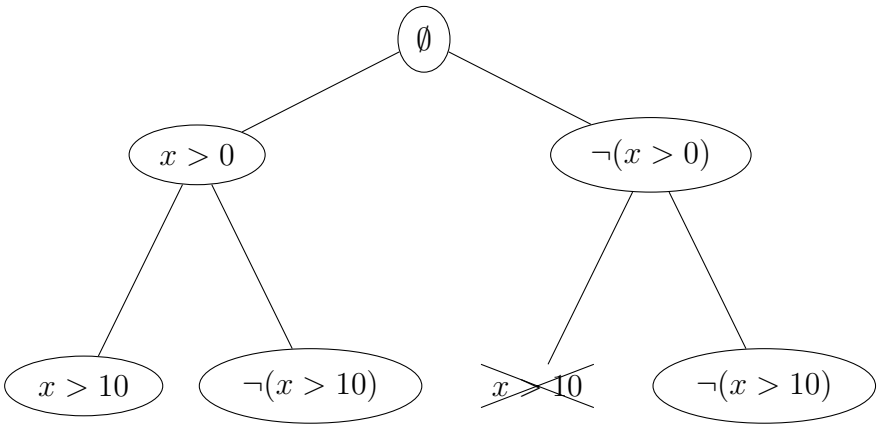

Figure 1.3: Example of symbolic execution.

by the two implementations are equivalent (Section 5.2), and (3) the implementations are race free (Section 5.3).

One fundamental limitation of symbolic execution is that it only handles objects of fixed size (i.e., each data structure in a program usually has to be assigned a concrete size, as in the normal execution of the program). For OpenCL, the number of work-items also must be concrete; in a typical OpenCL program, the number of work-items (i.e., the size of the NDRange) depends on the size of the input being processed. For our work, this means that we can verify the *bounded* equivalence of serial and data parallel programs, i.e. we can verify they are equivalent up to a certain input size.

The symbolic execution tool developed as part of this work is based on KLEE [KLE], an open source symbolic execution engine that operates on programs in the LLVM [LA04] intermediate representation (IR) format. LLVM is a static single assignment (SSA) based IR used by a number of compilers, including the Clang [cla] compiler which features robust support for a number of C family languages including C, C++, Objective C and OpenCL C, making it a highly practical base for a symbolic execution tool.

To symbolically execute a program, the user begins by compiling it using a compiler such as `llvm-gcc` or Clang that produces a compact serialised representation of the LLVM IR, known

as *bitcode*. An example command invocation is:

```
$ clang -c -emit-llvm -o simple.bc simple.c
```

This command uses Clang to compile the C program `simple.c` into LLVM bitcode format, storing it in the file `simple.bc`. Next, the user invokes KLEE, supplying the name of the bitcode file as an argument:

```
$ klee simple.bc
```

This command causes KLEE to find the `main` function in the given bitcode file and to begin symbolically executing it. If on any branch KLEE encounters an error, such as an assertion failure or an out of bounds memory access, KLEE will produce a concrete test case which demonstrates that error. KLEE provides tools that allow the user to easily reproduce the error during normal (concrete) execution by providing the generated test case as program input.

1.5 Publications

The work presented herein has previously been disseminated in the following publications:

- Peter Collingbourne, Cristian Cadar, and Paul H. J. Kelly. Symbolic testing of OpenCL code. In *Haifa Verification Conference (HVC 2011)*, Haifa, Israel, 2011.
- Peter Collingbourne, Cristian Cadar, and Paul H. J. Kelly. Symbolic crosschecking of floating-point and SIMD code. In *Proc. of the 6th European Conference on Computer Systems (EuroSys'11)*, April 2011.

1.6 Contributions

This thesis makes the following contributions:

1. We present a symbolic execution based technique for crosschecking data parallel programs in SIMD and OpenCL against their serial equivalents.

2. We reason about floating-point values (which KLEE's constraint solver cannot handle), using expression matching augmented with canonicalisation rules that express strict equivalences in floating-point and mixed FP-integer expressions. As far as we know, this is the first practical symbolic execution based technique that can precisely handle IEEE 754 floating point arithmetic.

3. We address the path explosion problem associated with symbolic execution by statically merging paths using phi node folding, a form of if-conversion.

4. We present a technique for symbolically testing for the presence of data races in OpenCL programs using a memory access log.

5. We implement our techniques in a tool called KLEE-CL, an extension to the open source symbolic execution tool KLEE [KLE].

6. We evaluate KLEE-CL by applying it to the OpenCV computer vision library, three Parboil benchmarks, the Bullet physics library and the OP2 library, and show that it can find real bugs, including memory errors, race conditions, and implementation mismatches.

Chapter 2

Background and Related Work

This thesis primarily concerns itself with a crosschecking and data race detection tool for programs written using such languages as C, C++ and OpenCL C, as opposed to a constructed language. While symbolic execution has proven itself to be a viable technique for such a task, significant remaining challenges include the modelling of various (simulated) environments, the computational overhead associated with path explosion and the adaptation of constraint solvers to specific problems.

In this chapter, we will survey other works in the field of symbolic execution that attempt to address these challenges, together with a number of other dynamic and static analysis techniques. We will also examine background topics related to this research, namely constraint solvers, GPGPU programming models, data race detection and floating point arithmetic.

2.1 Constraint Solvers

Symbolic execution engines such as KLEE depend on constraint solvers to decide the feasibility of the constraint sets built during the symbolic execution of a program, and to produce concrete test cases. A SAT (Satisfiability) solver is a specific kind of constraint solver, a decision procedure which, given a set of propositional formulas, determines whether the conjunction of

```
1 (benchmark test
2   :logic QF_AUFBV
3   :status unsat
4   :extrafuns ((x BitVec[32]))
5   :formula (not (= x (bvneg (bvneg x))))
6 )
```

Listing 2.1: A simple SMT-LIB problem.

those formulas is satisfiable, and produces a satisfying *assignment* for each variable used in the problem if it is satisfiable, which acts as proof that the problem is satisfiable.

SMT (Satisfiability Modulo Theories) solvers provide additional capabilities over SAT solvers, specifically support for a set of high level *theories*, and accept problems written in a language specified by the SMT-LIB initiative [BST10]. SMT-LIB theories include `Reals` (the theory of real numbers), `Fixed_Size_BitVectors` (the theory of fixed-size bitvectors, which includes a comprehensive set of integer arithmetic operations) and `ArraysEx` (the theory of functional arrays with extensionality). STP [GD07], the constraint solver used by KLEE, is a member of the SMT solver family. Most SMT solvers, including STP, can produce satisfying assignments, which KLEE uses to construct test cases.

Each SMT problem must belong to a single *logic* which provides one or more theories. Examples of logics are `QF_AUFBV` which provides the `Fixed_Size_BitVectors` and `ArraysEx` theories, and `QF_NRA` which provides the `Reals` theory. Listing 2.1 shows an example of an SMT-LIB problem in the `QF_AUFBV` logic, which attempts to test whether $x \neq -(-x)$ is satisfiable, where x is a 32-bit two's complement integer. Because this problem is unsatisfiable, it is marked as such using the `:status unsat` flag. `:status` is optional, but is useful when constructing test cases for SMT solvers.

The `Fixed_Size_BitVectors` and `ArraysEx` theories together embody most of the functionality required by the C and C++ languages, with the notable exception of floating-point arithmetic. Most verification tools for C and C++ with an SMT backend, including KLEE, therefore use `QF_AUFBV` or some variation thereof to achieve bit-level accuracy.

Two categories of techniques exist for SMT solving: eager and lazy [NOT06]. Under an eager

technique, a theory predicate is efficiently encoded as an equivalent SAT predicate, while under a lazy technique, such as DPLL(T) [NOT06], theory predicates are represented as unconstrained SAT predicates, and satisfiable assignments produced for these predicates by a SAT solver are used to form a set of theory constraints and solved by a theory solver. If the theory solver reports that the problem is unsatisfiable, additional clauses are added to the SAT problem to constrain the search. This process repeats until either the SAT solver reports that its problem is unsatisfiable (in which case the entire problem is deemed unsatisfiable) or the theory solver reports that its problem is satisfiable (in which case the entire problem is deemed satisfiable).

Lazy techniques can be optimised by integrating the theory solver more tightly into the SAT procedure. For example, DPLL(T) extends DPLL with, among other operations, a theory propagation operation which propagates consequences of the current set of defined theory predicates in a similar fashion to unit propagation in standard DPLL.

Bitvector theories can be solved using either an eager or a lazy technique, due to the one-to-one correspondence between individual bits and SAT predicates. STP is an example of an eager solver, as it works by converting SMT bitvector expressions to SAT predicates [GD07] (this technique is referred to as *bit-blasting*). While alternatives to bit-blasting such as integer linear arithmetic constraint encodings [BD02] and abstraction [BKO$^+$07, BCF$^+$07] have been proposed, many SMT solvers with `QF_AUFBV` support, such as STP, Z3 [dMB08], Boolector [BB09] and Yices [DdM], use bit-blasting for bitvectors.

Our benchmarks (Chapter 6) incorporate mixed integer and floating point constraints, including conversions between integers and floating point numbers. The existing `Fixed_Size_BitVectors` and `Reals` theories are insufficient to model mixed integer and floating point arithmetic using reals for floats, as they do not specify conversions between bitvectors and reals (nor can they, being separate theories). Floating point arithmetic may only be approximated by reals using a new theory that would incorporate the real and bitvector theories together with conversions between bitvectors and reals. To our knowledge, no existing solver supports such a theory.

2.2 Symbolic Execution for the Real World

Symbolic execution was first described in the 1970s by King et al [Kin76], in which the basic common elements of symbolic execution were set out: symbolic values, paths, path conditions and automated decision procedures (although at that time no automated decision procedures had been developed). Since then, an explosion of available computing power has led to the development of a wide variety of strategies and tools for symbolic execution.

Automated decision procedures in general, and SMT solvers in particular, have been described as disruptive technologies in the context of formal verification [Rus06]. The existence of these decision procedures has led to the development of a variety of automated symbolic execution tools starting in the latter half of the last decade. These symbolic execution tools are capable of testing real-world code written in industrial programming languages such as C, C++ and Java [CGP+06, CDE08, GKS05, APV08]. Frequently, these systems have been used to find real bugs which have been reported to the original developers and fixed.

One of the earliest systems to be developed as part of the modern era of symbolic execution tools was EXE [CGP+06], a symbolic execution engine for C and C++. EXE has been used to find bugs in such esoteric code as the BSD and Linux operating system kernels. KLEE [CDE08] is an evolution of EXE, and the tool upon this work is based. KLEE was originally used to find bugs in `coreutils`, an open source implementation of 89 core POSIX commands. KLEE's use of LLVM [LA04] allowed it to use production quality compiler frontends such as `llvm-gcc` and Clang.

A number of works have been developed in an attempt to address the challenges associated with symbolic execution: the modelling of various (simulated) environments, the computational overhead associated with path explosion and the adaptation of constraint solvers to specific problems. A significant proportion of this work is based on KLEE, which was released as an open source project [KLE].

Path explosion. The symbolic execution of multiple paths involves no interaction between paths, and is thus an embarrassingly parallel problem. Therefore, the execution of individual paths

can be distributed across multiple cores and even across machines. Cloud9 [CZB+10, CBZ10, BUZC11] addresses path explosion in this way using parallel distributed symbolic execution. The authors report an average increase of 13% in coverage over KLEE in a trial run of each `coreutils` utility for 10 minutes in both KLEE and a 12-worker instance of Cloud9. While this may not sound impressive, it is certainly a worthwhile improvement, but it may indicate that "throwing hardware at" an exponential problem at a linear increase in cost is a viable strategy only after one has also considered strategies for reducing the number of paths explored.

Our static path merging technique, an application of if-conversion [LA04, CCF03], is an example of such a strategy. As an alternative to static path merging, a dynamic path merging technique for symbolic execution is proposed by Hansen et al. [HSS09]. This technique works by identifying join points in the control flow graph and attempting to merge the state of paths reaching the join point. The main advantage of this approach is greater applicability – for example, it can handle conditional paths containing side effects. However, there are runtime costs associated with dynamic path merging, such as the increased complexity of the scheduler, which must ensure that states reach the join point at the same time, as well as the cost of performing the merge, which in the worst case would involve inspecting every memory location in the two states, and which must be incurred at every join point. Furthermore, the path condition would need to be formed from a disjunction, which can be taxing on SMT solvers [GD07], although well-known Boolean simplifications can be applied in many cases [HSS09]. By contrast, the cost of static path merging is only incurred once up front and imposes no restrictions on further execution, while solving the majority of path explosion issues we encountered during symbolic execution of data parallel programs.

If-conversion has already been applied successfully in the static analysis context, particularly by bounded model checkers [KCY03, SMF12] in order to generate a single monolithic verification condition. The authors of [HSS09] also acknowledge that the performance of their tool may benefit from analysing programs optimised by if-conversion. As far as we know, this is the first symbolic execution based technique which relies on the deliberate application of a modified aggressive if-conversion transformation.

Recent work in the area provides alternative approaches that we could apply to reduce the number of paths explored: using compositional dynamic test generation to create function summaries [God07], using read-write sets to track the values accessed by the program [BCE08], or using information partitions to track information flow between inputs [MX09].

KleeNet [SLA+10, SDK+11] is an extension of KLEE tailored for wireless sensor networks. It uses a variety of strategies based on analysing communication patterns between simulated wireless sensor network nodes to minimise the number of nodes that need to be forked. On average, their most effective strategy, *Super DStates*, resulted in approximately a 100× improvement in execution time over the naïve *copy-on-branch* strategy of forking the state of the entire set of nodes.

S^2E [CKC11] is a system for selective symbolic execution of x86 binaries. Using S^2E, a user may symbolically execute x86 binaries in combination with concrete execution of an operating system or environment, such as Windows. S^2E addresses the path explosion problem by only symbolically executing code that is of interest to the user, as opposed to the entire operating system. While this can be imprecise in the event that symbolic data leaks into the concretely executed environment, the authors claim that in many cases symbolic data can be passed through the environment without inspection.

Tools such as DART [GKS05] and CUTE [SMA05] use the technique of dynamic directed test generation to attempt to minimise the number of paths explored while maximising coverage. Under dynamic directed test generation, a program is first executed using random inputs. The technique then attempts to drive the resultant execution trace towards uncovered execution branches (which may contain error conditions) by constructing a symbolic path condition with some uncovered branch's condition inverted, solving those constraints and (if feasible) re-executing the program using the counterexample.

Environments. The construction of an appropriate environmental model for the symbolic execution of a program has deserved significant study. Not only must the model correctly simulate a substantial underlying system but in some cases must be tailored for a symbolic environment. For example, in a networked environment, the ability of a program to recover from errors caused

by unreliable networks is crucial, and to this end, the networking models of both Cloud9 and KleeNet include support for injection of failures such as node outages and duplicate packets. Using failure injection, KleeNet was able to detect a number of bugs in the TCP/IP stack of the Contiki [DGV04] sensor network operating system.

As an alternative to constructing a symbolic environment, S^2E's support for concrete environments allows for programs to be tested without first needing to construct a model for the environment, which is beneficial for large or poorly specified environments, or in cases where there is only one significant implementation of the environment (such as is the case for Windows).

In our case, our focus is on detecting specification violations in OpenCL programs, namely those that result in data races. While we could have modelled real hardware, such as an NVIDIA GPU, this would not only complicate our model significantly but make it more difficult to detect data races that do not manifest on NVIDIA hardware. Instead, our model consists of a generic OpenCL device. We did not see a need to add support for failure injection because OpenCL devices typically have a resilient connection to the host.

A similar approach was taken by STLlint [GS06], a program analysis for the C++ Standard Template Library. It uses symbolic execution and a detailed, standards based model of the STL to check for errors in STL usage. This approach avoids the "hidden specification" problem common among symbolic execution engines. However, this tool relies on a detailed model of the library and is only capable of detecting errors that it has specifically been programmed to find.

Constraint solvers. An enhanced constraint solver can improve symbolic execution performance and allow the symbolic execution engine to effectively handle new types of problems. In our case, we required some level of support for floating point arithmetic, and this topic is covered in more detail in Section 2.6.

String constraints are a useful way of improving the search space of symbolic execution in the context of programs which use textual input (such as parsers and Web applications). A string

constraint normally takes the form of a context-free grammar which the constraint solver then attempts to match against a constrained symbolic string.

KLEE has been extended with support for string constraints using the HAMPI constraint solver [KGG+09]. HAMPI was able to improve KLEE's test coverage of three open source programs – `cueconvert`, `logictree` and `bc` – by constraining the program input with a string constraint automatically derived from the programs' input grammars.

The technique of string constraints in combination with symbolic execution has also been used for Web security analysis. The Kudzu [SAH+10] symbolic execution engine is capable of symbolically executing client-side JavaScript code and testing it against vulnerabilities, such as cross-site scripting (XSS) attacks. The decision procedure used by Kudzu is Kaluza, a constraint solver for strings. Kudzu uses Kaluza to test symbolic strings provided to *critical sinks* susceptible to exploitation (such as the `eval` function, which evaluates a given string as JavaScript code) against *attack patterns* (for example, for the `eval` function, a sequence of valid statements followed by a malicious payload).

Equivalence testing. Our approach of using symbolic execution combined with expression matching and canonicalisation rules has been successfully used in the past to verify code equivalence in other contexts, such as hardware verification [CK03], embedded software [CFF+06], compiler optimisations [Nec00] and block cipher implementations [SD08].

2.3 Other Approaches to Testing and Verification

Symbolic execution is an example of a dynamic program analysis technique. In this section, we will examine instrumentation, another dynamic technique, together with static analysis techniques such as model checking, predicate abstraction, abstract interpretation and meta-level compilation.

Instrumentation is a widely used dynamic analysis technique. Under instrumentation, a program is augmented with checks that detect various errors, such as memory errors and data

races (discussed in further detail in Section 2.5). Instrumentation is generally intended to be used during development, as it can slow down a program considerably due to the overhead associated with maintaining additional state.

Valgrind [NS07] is an example of a dynamic instrumentation tool based on dynamic binary translation. Using *shadow values* to track supplementary information about each bit of data in registers or memory, it has been used as a basis for a wide range of dynamic analyses, including memory error detection [SN05], taint tracking [NS05] and data race detection [SI09]. While two main advantages of binary translation are ease of use and language independence, dynamic recompilation tends to be less efficient than static instrumentation at compile time (e.g. [VBPS12], although this tool detects a different set of bugs). Accuracy can also suffer as a result of a lack of access to high level information, such as types.

Model checking is a verification technique which systematically explores the state space of a system for the purpose of verifying a given property. If a state is reached in which the property is falsified, the model checker will produce the inputs as a counterexample.

Model checking has been used as a technique to verify hardware circuits and computer software. However, model checking on its own is most effective as a technique for verifying hardware circuits, as the size of the state space is generally fixed. On the other hand, the cost of model checking a sequential program can become unpredictably large in the presence of loops and recursion. *Bounded model checking* provides a solution to this problem by imposing a specified bound on the iteration count of any loops and the depth of recursion, thus making the problem more tractable (although necessarily incomplete).

Bounded model checking for sequential programs is at its core a very similar technique to symbolic execution. The main distinction lies in how program state is represented at different points in the program. Under symbolic execution, each program point has a one-to-many relationship with program state – the specific program state at a program point depends on the current path, together with the current stack frame and loop iteration, as during normal execution. Under bounded model checking, program points have a one-to-one relationship with program state, and all paths through the program form part of the same problem. Thus,

neither loops nor recursion are supported directly. To support loops and recursion, bounded model checkers first apply a preprocessing step which inlines all function calls, and unrolls loops and recursive function calls up to the given bound.

Reducing all program paths to a single problem has the advantage of reducing the burden on the constraint solver. However, this can also complicate interaction with the environment, such as file I/O. While symbolic execution engines can provide access to the real environment on a per path basis, under bounded model checking, all functions must be visible to the model checker, which generally precludes access to a real environment.

Bounded model checking was the basis for one of the earliest systems to be developed for real world C code: CBMC [CKL04], the C Bounded Model Checker. CBMC contains support for both C and the Verilog hardware description language, and has been used to cross check C implementations of the DES algorithm [CK03] and of a simple RISC hardware architecture, DLX [KCY03] against Verilog implementations.

Regression verification [GS09] is a technique designed to improve the tractability of proving the equivalence of similar programs by only considering the equivalence between syntactically different subroutines. While this technique has the potential of improving tractability for the types of problems encountered during crosschecking of serial and data parallel programs, no benefit will be gained in the case where the serial and data parallel implementations are completely independent, as was the case for several of the benchmarks we evaluated (Chapter 6). Furthermore, the technique is not directly applicable to libraries, such as OpenCV (Section 6.1) and Bullet Physics (Section 6.3), in which the serial and data parallel codes are part of the same library and reside in different functions, complicating the problem of computing a pairing between them as required by the algorithm.

Predicate abstraction [GS97] has been proposed as a technique for improving the tractability of model checking, in conjunction with *abstraction refinement*. Under predicate abstraction, a program is first modified to keep track of only certain predicates required to check the property under test, and model checking is performed on the abstracted program. If a property under test is falsified, abstraction refinement is used to adjust the set of predicates tracked.

SATABS [CKSY05] is a system for predicate abstraction refinement of ANSI C and C++ programs. It operates by transforming a C or C++ program into an abstract Boolean program, which is then checked by a model checker such as Boom [BHK+10].

Predicate abstraction has also been shown to offer a significant speedup for cross checking DLX as compared to bounded model checking with CBMC [CK04].

Predicate abstraction is important for the effective verification of programs which use floating point arithmetic, as we shall see in Section 2.6.

Abstract interpretation [CC92, CC77] is a common technique used in static analysis, and is another form of abstraction. Abstract interpretation is normally used to track abstractions of variable values within a program – an example of a simple abstraction for an integer would be its sign. Abstract semantics are then assigned to operations over those value. For example, assuming no integer overflow, the addition of two positive integers will produce another positive integer. Abstract values are then propagated through the program, combining results at control flow convergence points using a join function which produces the least upper bound of its arguments. A simple example of a least upper bound is the set union operator. For example, the least upper bound of two positive numbers is a positive number, and the least upper bound of a positive and a negative number can be written $\{+, -\}$, representing either a positive or negative number.

Abstract interpretation has been applied to the analysis of floating point programs in order to track the maximum rounding error that may be applied to particular computations [CCF+05, DGP+09]. Further applications of abstract interpretation include pointer alias analysis [WL95], security analysis [WFBA00, BBF02] and execution time estimation [FMWA99].

Meta-level compilation [CCEH00] is the technique of checking, transforming and optimising system-specific operations at the compiler level. Such checks may be derived from system-level restrictions which cannot easily be enforced at compile time using the language alone, such as type checks. For example, Linux kernel code is generally forbidden from performing floating point operations, due to the cost associated with saving and restoring the state of the floating

point unit [Lov10]. Another example of such a restriction is that imposed by a communications protocol, whereby operations have to be carried out in a specific order.

Meta-level compilation is perhaps the most widely used static analysis technique for C and C++ code. GCC and Clang (among other compilers) will by default check for a range of issues relating to the C and POSIX standard libraries, emitting warnings if any of those issues are detected, and contain special optimisation support for those libraries. For example, Clang will warn about known incorrect sizes supplied to calls to `memset` and other standard C memory functions, and can optimise a call to `strlen` with a constant string literal argument to the actual length of the string. While useful to users of standard system libraries, neither GCC nor Clang contain support for easy development of new checks or optimisations without significant compiler expertise.

The xg++ [ECCH00] project attempted to address this issue using a domain specific state machine language, Metal, which non-experts may use to develop new restrictions to be imposed by the compiler. xg++ has been used to enforce restrictions associated with the FLASH operating system [CCEH00], in which a total of 34 bugs were found relating to the violation of restrictions on buffer management, communication patterns, use of floating point arithmetic and other issues.

Another project in the same field is CHS [CK09], a Haskell-based plugin infrastructure for GCC which provides a general data-flow analysis. The data-flow analysis has been used to check that a program conforms to a communication pattern specified as a session type.

While the technique of meta-level compilation has been shown to provide rapid user feedback for a wide variety of errors, it is most effective at detecting errors which are easily detectable at the AST level and which it has specifically been programmed to find. Meta-level compilation is therefore most effectively used in combination with other static or dynamic techniques.

```
1  void square(float *squares, const float *nums, size_t size) {
2      while (size >= 4) {
3          __m128 numv = _mm_loadu_ps(nums);
4          __m128 squarev = _mm_mul_ps(numv, numv);
5          _mm_storeu_ps(squares, squarev);
6          squares += 4; nums += 4; size -= 4;
7      }
8      while (size) {
9          *squares = *nums * *nums;
10         squares++; nums++; size--;
11     }
12 }
```

Listing 2.2: The `square` function, a simple example of Intel SSE code.

2.4 SIMD and GPGPU Programming Models

A wide range of programming models are available today for data parallel computing, and in this section we will examine programming models specifically used for SIMD and GPGPU computing.

SIMD. SIMD code is most commonly written explicitly, using special compiler *intrinsic* functions which each correspond to a single SIMD instruction, and which manipulate values of vector data types. Alternatives to explicit SIMD coding include translation from alternative programming models such as SPMD [Rot11, PM12, PBBR07], and automatic vectorisation of ordinary serial code [EWO04, LA00, NBBDZ03, BGGT02, GZA+11]. However, such alternatives have not been as widely adopted in practice as intrinsics have been – not only are alternative programming models less convenient to use, and support for automatic vectorisation less widespread (only one major compiler–the Intel C Compiler [BGGT02]–supports automatic vectorisation out of the box as of this writing), but hand written SIMD code is free to make assumptions about pointer aliasing and data dependencies which the compiler may not be able to make.

Intrinsics are generally standardised across compilers targeting a given processor family (such as Intel x86, which is *de facto* standardised via its architecture and compiler documentation). For example, the `_mm_mul_ps` intrinsic provides the `MULPS` instruction.

Listing 2.2 gives an example of a C function which uses SIMD (specifically, Intel SSE) to compute the square of each of an array of numbers `nums`, storing the results in an array `squares`. Each iteration of the loop at lines 2–7 processes four elements of `nums` at a time. The variables `numv` and `squarev` are of type `__m128`, i.e., 128-bit vectors consisting of four `float`s each. The code first loads four values from `nums` into `numv` by using the SIMD instruction `_mm_loadu_ps()` (line 3). It then multiplies each element of `numv` with itself using `_mm_mul_ps`, storing the result in `squarev` (line 4). Finally it stores `squarev` to the four array elements pointed to by `squares` (line 5), and prepares for the next iteration (line 6).

Because the SIMD loop can only process four elements of `nums` at a time, an additional loop is required to process the 1–3 last elements of `squares` in the event that the array size is not an exact multiple of 4. This loop is shown at lines 8–11.

GPGPU. The two most popular GPGPU programming languages are CUDA [NVI10] and OpenCL [Khr10], both based on the SPMD programming model common to GPGPUs – that is, the user provides a kernel function together with the number of parallel invocations required.

As with SIMD, GPGPU programming languages exist which are based on translation from another programming model to SPMD. Examples of such languages are OpenHMPP [hmp], an extension of C and Fortran which provides a set of `#pragma` directives which control the generation of data parallel code from serial code (the intent being that the program does not change its semantics when interpreted without directives), and C++ AMP [cpp], an extension of C++11 [Int11] which provides *restriction specifiers* which request that certain functions be compiled for the GPU, and a set of generic data parallel algorithms which may be used to invoke those functions. However, such languages have had a lower rate of adoption than CUDA and OpenCL, perhaps due to a lack of available tools or cross-platform compatibility.

While CUDA is probably the more popular GPGPU programming language, it is specifically targeted towards NVIDIA's GPUs. In this thesis we aim to develop techniques which are generally applicable to a wide range of underlying hardware devices, and as such we concentrate on the OpenCL model and use OpenCL terminology.

The OpenCL programming model is made up of two distinct parts: the runtime library, a C application programming interface (API) used to manage the execution of OpenCL kernels, and the OpenCL C programming language, a C99 [Int99] dialect in which OpenCL kernels are written.

The OpenCL runtime library is specified by two sections of the OpenCL specification: the OpenCL Platform Layer [Khr10, § 4] and the OpenCL Runtime [Khr10, § 5]. The Platform Layer is used to query the set of available OpenCL devices, while the Runtime is used to query and manipulate objects on a specific device or set of devices such as device-side memory buffers and compiled OpenCL programs.

OpenCL C is specified as part of the OpenCL specification [Khr10, § 6]. Among the language extensions provided by OpenCL C are vector data types, specialised memory address spaces and a set of built-in functions. These built-in functions include *work-item functions*, which may be used by the kernel to query various properties of the current execution's index space, including the current work-item and work-group identifiers for each dimension; *math functions*, which perform various mathematical operations (including vectorised variants); and *synchronisation functions*, which co-ordinate communication between work-items.

To give an overview of the main features of OpenCL, we show in Listing 2.3 a C function `gpu_arr_sqrt`, an OpenCL implementation of a function that computes the square root of every element of an array `in`, storing it into an array `out`. This function makes use of the OpenCL kernel `arr_sqrt_kern` shown in Listing 2.4.

On lines 5–7 we use the OpenCL Platform Layer API to obtain a *context* reference for the default OpenCL device, to be used by OpenCL Runtime API functions.

On lines 9–20 we obtain a reference to the `arr_sqrt_kern` kernel shown in Listing 2.4, first by loading the OpenCL C program source code from the disk using the C standard library, and then by using the `clCreateProgramWithSource` and `clBuildProgram` functions to compile the program and finally the `clCreateKernel` function to obtain a reference to `arr_sqrt_kern`.

On lines 24–29 we create *memory buffer* references for the `in` and `out` arrays. By using the

```
1  void gpu_arr_sqrt(float *out, const float *in, size_t size) {
2    cl_platform_id platform;
3    cl_device_id device;
4
5    clGetPlatformIDs(1, &platform, NULL);
6    clGetDeviceIDs(platform, CL_DEVICE_TYPE_DEFAULT, 1, &device, NULL);
7    cl_context ctx = clCreateContext(NULL, 1, &device, NULL, NULL, NULL);
8
9    FILE *bin = fopen("opencl-sqrt.cl", "r");
10   fseek(bin, 0, SEEK_END);
11   size_t code_len = (size_t) ftell(bin);
12   fseek(bin, 0, SEEK_SET);
13   char *code_ptr = malloc(code_len);
14   fread(code_ptr, code_len, 1, bin);
15   fclose(bin);
16
17   cl_program prog =
18     clCreateProgramWithSource(ctx, 1, &code_ptr, &code_len, NULL);
19   clBuildProgram(prog, 1, &device, "", NULL, NULL);
20   cl_kernel kernel = clCreateKernel(prog, "arr_sqrt_kern", NULL);
21
22   cl_command_queue cmd_queue = clCreateCommandQueue(ctx, device, 0, NULL);
23
24   cl_mem in_buf = clCreateBuffer(ctx,
25         CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
26         size * sizeof(float), in, NULL);
27   cl_mem out_buf = clCreateBuffer(ctx,
28         CL_MEM_WRITE_ONLY | CL_MEM_USE_HOST_PTR,
29         size * sizeof(float), out, NULL);
30
31   clSetKernelArg(kernel, 0, sizeof(cl_mem), &out_buf);
32   clSetKernelArg(kernel, 1, sizeof(cl_mem), &in_buf);
33
34   clEnqueueNDRangeKernel(cmd_queue, kernel,
35     /* work_dim */ 1,
36     /* global_work_offset */ NULL,
37     /* global_work_size */ &size,
38     NULL, 0, NULL, NULL);
39
40   clFinish(cmd_queue);
41 }
```

Listing 2.3: The `gpu_arr_sqrt` function.

```
1  __kernel void arr_sqrt_kern(__global float *out,
2                              __global const float *in) {
3    size_t i = get_global_id(0);
4    out[i] = sqrt(in[i]);
5  }
```

Listing 2.4: The OpenCL C `arr_sqrt_kern` kernel used by `gpu_arr_sqrt`.

`CL_MEM_USE_HOST_PTR` flag, we cause the OpenCL implementation to use the memory regions referred to by `in` and `out` directly (i.e. without creating a copy). For a GPU based implementation of OpenCL, this means that in practice `in` will be automatically copied to the GPU by the OpenCL implementation immediately before kernel invocation, and `out` will be copied from the GPU afterwards.

On line 31 we set the first kernel function argument to `out_buf`, and on line 32 the second to `in_buf`. Then on line 34 we call `clEnqueueNDRangeKernel`, which schedules the execution of `arr_sqrt_kern` on the device. Finally on line 40 we call `clFinish`, which blocks until kernel execution terminates. After the call to `clFinish` returns, the data generated by the kernel is guaranteed to be present in the `out` array on the host.

The call to `clEnqueueNDRangeKernel` specifies the bounds of the NDRange used to invoke `arr_sqrt_kern`. In this case, the `work_dim` argument is set to 1, so the NDRange has one dimension; `global_work_size` is a pointer to `size`, so the NDRange will have `size` elements; and `global_work_offset` is `NULL`, so the NDRange starts at work-item identifier 0.

The `arr_sqrt_kern` kernel function is called `size` times in parallel, once for each element of this NDRange. The `get_global_id(0)` function call on line 3 is used to retrieve the work-item identifier for the first (and only) dimension, which can range between 0 and `size`-1, and which is used to index the `in` and `out` arrays.

As this discussion shows, OpenCL contains many dynamic features, such as runtime compilation, dynamic lookup of kernel functions and dynamic construction of kernel arguments. We therefore consider a dynamic analysis technique, such as symbolic execution, to be a good fit for the analysis of unmodified OpenCL programs in tandem with OpenCL C kernels.

2.5 Detecting Errors in GPGPU Programs: Data Race and Barrier Divergence Detection

GPGPU programs may contain additional classes of error over serial programs. Some of these, such as data races, are common to many parallel programming models, while others, such as misuse of execution barriers, are unique to SPMD programming models such as those targeting GPGPUs. This section will discuss static and dynamic analysis techniques for detecting such errors.

Data races. Data race detection techniques for GPGPU code are based on existing techniques for parallel code, which run the gamut from static analysis to dynamic instrumentation.

The accuracy of the various techniques is in general a function of the number of execution schedules considered, although techniques such as partial order reduction [FG05] and language-specific heuristics [KYKS09] exist for pruning the number of schedules while preserving soundness. Some programming models can be checked soundly using a single arbitrary schedule, such as atomic-free OpenCL or CUDA [LG10], allowing for a simpler algorithm to be used than in the case of traditional synchronisation primitives such as locks and semaphores.

A significant body of work focuses on dynamic race detection approaches for CPU code [FF09, OC03, SBN+97, SI09]. Dynamic approaches normally consider only one schedule, and thus are unsound for programs using functionality such as atomics, locks and semaphores, due to data dependencies between threads. Nevertheless, dynamic single schedule race detectors have been shown to find real bugs in large programs.

A number of race detectors are based on Lamport's happens-before ($\prec$) relation [Lam78], which is used to declare a pair of accesses a_1 and a_2 to be raceful unless $a_1 \prec a_2$, or both of a_1 and a_2 are reads. Happens-before is conventionally applied to programs that use message passing and/or wait/notify synchronisation primitives, for which $\prec$ is defined in terms of send/receive or wait/notify operations. *Vector clocks* [Fid88, Mat88] are a conventional implementation technique in this case.

Locksets [SBN+97] are one common technique for detecting data races in programs that use locks [SBN+97, SI09, OC03]. Under the lockset algorithm, a lockset $C(v)$, initially the set of all locks within a program, is maintained for every shared variable v. Upon each memory access, $C(v)$ is set to the intersection of $C(v)$ and the current set of locks held by the thread. If this causes $C(v)$ to become the empty set, this indicates that no lock protects v at every program point and that the program is therefore likely to contain a data race.

A hybrid approach that utilises both happens-before and locksets has been proposed [OC03]. In this approach, both happens-before and locksets are tracked simultaneously, and an access is considered a data race only if both techniques consider it as such. This approach can be used to more accurately detect races involving both message passing and locks, and indeed has been used to find data races in large scale real world programs such as the Tomcat and Resin web servers [OC03] and the Chromium web browser [SI09].

We shall now consider how the fundamental dynamic race detection techniques contained within the literature may be applied in the context of OpenCL C kernels. The two synchronisation methods provided by OpenCL C are atomic functions and execution barriers.

Atomic functions allow for a mathematical or logical operation to be performed *atomically* at a given memory location, i.e. without risk of data races. All atomic operations provided by OpenCL C are *read-modify-write* operations, meaning that they read a value at a given memory location, modify the value by performing some operation on it, write the new value to the memory location and return the old value. For example, the `atomic_add` function atomically adds a given value to the value stored at a given memory location and returns the old value. A set of OpenCL C work-items can use `atomic_add` to co-operatively compute the integer sum of a set of values computed by each of the work-items.

Execution barriers, provided by the `barrier` function, allow the kernel to control both the scheduling and memory synchronisation behaviour of a work-item in relation to the other work-items within a work-group. When a work-item reaches a call to `barrier`, execution is blocked until all work-items in the work-group have reached the call to `barrier`, at which point a memory fence is queued to ensure the correct ordering of memory operations between the

work-items, and all work-items in the work-group resume execution.

Of these synchronisation methods, an OpenCL C kernel containing execution barriers alone can be correctly modelled using a single, canonical execution schedule. However, because the result of running an OpenCL C kernel which uses atomic functions can depend on the execution schedule, correctly modelling atomic functions would require us to consider all possible execution schedules (although in Section 7.5 we present a technique that may allow us to only need to consider one execution schedule in certain common cases).

We have found that in OpenCL C kernels, execution barriers are much more commonly used for synchronisation than atomic functions. In fact, none of the OpenCL C kernels we evaluated (Chapter 6) used atomic functions. While techniques for pruning the number of execution schedules exist [EQT, KYKS09], adding full atomic function support would still require us to support multiple execution schedules, the implementation cost of which would outweigh the potential benefits due to their infrequent use. Therefore, in this work, we only consider a single execution schedule, and do not model atomic functions, only execution barriers.

The approach proposed in this thesis for detecting races involving execution barrier synchronisation is fundamentally most similar to happens-before. We can define $\prec$ for OpenCL C as follows, where $\text{wi}(x)$ is a unique identifier for the work-item that performed x, $\prec_{wi}$ is the intra-work-item happens-before relation, $\text{wg}(x)$ is a unique identifier for the work-group that performed x and $\text{bar}(x)$ is the number of execution barriers encountered by the work-item before performing x:

$$a_1 \prec a_2 \leftrightarrow (\text{wi}(a_1) = \text{wi}(a_2) \wedge a_1 \prec_{wi} a_2) \vee (\text{wg}(a_1) = \text{wg}(a_2) \wedge \text{bar}(a_1) < \text{bar}(a_2))$$

This definition of $\prec$ gives way to simpler internal representations and/or verification condition (VC) encodings than the conventional vector clock approach. Our encoding (Section 5.3) uses a per-byte memory access record, similar to the shadow value technique used by Valgrind [NS07], but designed to take advantage of the facilities provided by SMT solvers, specifically the theory of arrays.

Execution barrier divergence. While execution barriers are a useful synchronisation mechanism, they come with a few caveats. Firstly, it must be remembered that execution barriers are always per-work-group, and unlike atomic functions, execution barriers cannot be used to communicate between work-groups, regardless of whether a barrier is also encountered in another work-group. Secondly, an execution barrier must either be encountered by all work-items in a work-group, or by none of them. If work-items in a work-group reach different calls to the `barrier` function, or reach the same call during different iterations of a loop (we refer to either situation as *execution barrier divergence*), the behaviour of the OpenCL implementation is undefined [Khr10, §6.11.8].

In this work, we do not attempt to detect errors resulting from execution barrier divergence. Our reason for doing so is that the presence of divergence errors is normally evident to the user during normal testing (most implementations will deadlock in this case) as opposed to "silent" errors resulting from data races, and that there would therefore be less value in adding detection support. On the other hand, implementations are free to silently tolerate divergence errors which may cause issues with other OpenCL implementations, and there is certainly value in the ability to provide a detailed report to the user. Execution barrier divergence testing is therefore explored as future work (Section 7.4).

Related work. Most previous work in the area of race detection for GPU code has used a similar underlying approach of conflict checking between previously logged memory accesses. A number of authors [LG10, TSL10, LLS+12, BCD+12] have proposed race detection techniques based on automated constraint solving using various different VC encodings.

Aiken and Gay [AG98] describe a technique for detecting execution barrier divergence errors in SPMD programs by defining a *single-valuedness* property for each expression within a program (intuitively, the notion that the expression will evaluate to the same value in each SPMD thread). The single-valuedness of an `if` statement's controlling expression is used to determine whether the same path is guaranteed to be followed in each thread. Because the single-valuedness analysis is conservative, the technique is prone to false positives.

PUG [LG10] is a static verifier for CUDA code that can test for functional correctness via assertion tests, and can also check for errors such as data races and execution barrier divergence,

as well as CUDA hardware-specific efficiency issues such as bank conflicts.

Tripakis et al. [TSL10] propose an SMT-based technique for verifying SPMD programs, focusing on CUDA. Their technique can check for non-interference (essentially data race freedom) and verify the functional equivalence of two CUDA programs via cross-checking.

GKLEE [LLS+12] is an extension of KLEE which includes a CUDA model and support for detecting a range of errors in CUDA programs, including data races and execution barrier divergence, and efficiency issues such as control flow divergence within thread warps, bank conflicts and non-coalesced memory accesses. Like KLEE-CL, GKLEE can check functional correctness (via cross-checking, or by other means). However, it lacks support for symbolic floating point arithmetic, and as such it cannot be used to analyse the functional correctness of CUDA floating point programs.

GPUVerify [BCD+12] is a static GPU verification tool which can detect data races and barrier divergence in GPU kernels. It works by transforming a GPU program into a Boogie [BCD+05] program representing the parallel execution of an arbitrary distinct pair of work-items.

PUG, Tripakis et al. and GPUVerify are examples of static analyses which analyse a single work-item (or pair of distinct work-items) in the NDRange. By contrast, KLEE-CL and GKLEE are dynamic analyses based on symbolic execution which analyse each work-item in the NDRange individually. The main advantage of a static analysis approach is coverage: our dynamic approach depends on the number of paths explored by symbolic execution in a given time budget and can only reason about objects with concrete bounds. On the other hand, static analysis suffers from false positives, due to various over-approximations resulting from, e.g., analysing kernels in isolation and loop unrolling.

The VC encoding used by PUG, GKLEE and Tripakis translates pairs of potentially conflicting memory accesses into SMT constraints based on equality checks. This is probably the most direct encoding, however the complexity of the generated constraints is $O(n^2)$. This encoding is therefore probably better suited to static analyses which analyse a small number of work-items than to dynamic analyses where hundreds or thousands of work-items may perform separate

memory accesses to a given array.

GPUVerify uses a different VC encoding for race detection. Under its *element* encoding, for each potentially conflicting array and for each of the two analysed work-items, it builds a symbolic expression representing an arbitrary member of a set of array indices accessed by a particular work-item using unconstrained Boolean variables (represented at the Boogie level by a non-deterministic conditional). The Boogie program generated by GPUVerify then tests the satisfiability of the equality of the two symbolic expressions built for the two analysed work-items.

By contrast, our encoding uses a combination of a concrete representation with an $O(1)$ cost per memory access for data race detection involving concrete array indices (which we found to be the most common case encountered during dynamic analysis) and a symbolic representation which uses the SMT theory of arrays (with $O(n)$ array updates). Our encoding only switches to the symbolic representation when required by a symbolically indexed memory access. Whether our encoding is less or more efficient to solve than the encodings described in previous work depends fundamentally on the constraint solver used.

An instrumentation based dynamic race detection approach similar to ours is introduced by Boyer et al. in the context of CUDA programs [BSW08]. A more recent technique from Zheng et al. [ZRQA11] combines dynamic race detection with a static analysis pass that removes accesses that can be statically proven to be safe or unsafe, resulting in a system with a relatively small runtime overhead. The main weakness of these techniques is that they depend on the concrete inputs with which the program is run. Instead, our approach can check for symbolic race conditions on all the different paths explored via symbolic execution.

The restrictions specified for execution barrier divergence in CUDA are less strict than those in OpenCL C: specifically, CUDA does not forbid reaching the same execution barrier during different iterations of a loop [NVI10, §B.3]. GKLEE detects execution barrier divergence errors according to this restriction, by partitioning a CUDA program into barrier intervals, and at each barrier, checking the textual alignment [KY05] of all barriers encountered.

The GPUVerify authors found [BCD+12] that CUDA's execution barrier divergence restrictions are incomplete (in that programs that comply with the restrictions can exhibit raceful behaviour on NVIDIA hardware), and give a more precise definition of barrier divergence freedom:

> [...] if a barrier is encountered by a group of threads executing in lock-step under a predicate, the predicate must hold uniformly across the group, i.e., the predicate must be true for all threads, or false for all threads.

GPUVerify verifies barrier divergence freedom according to this definition by transforming the Boogie program to use predicated execution, and checking the consistency of the two predicates for the two analysed work-items at each execution barrier.

The technique we propose as future work (Section 7.4) is designed to detect execution barrier divergence errors in OpenCL C. It is fundamentally similar to GKLEE's technique for CUDA, except that it maintains a trip count for each loop in order to enforce the additional restriction imposed by OpenCL C.

2.6 Floating Point Arithmetic

In this thesis, we will concentrate on floating point arithmetic in the context of C, C++ and OpenCL. Despite common belief, the requirement for a C implementation to provide IEEE 754 compliant floating point arithmetic is optional [Int99, Annex F], and the C++ standard leaves the representation of floating point numbers entirely up to the implementation [Int03, §3.9.1(8)]. Nevertheless, most C compilers implement the optional Annex F to the C standard, and most C++ compilers provide equivalent support.

The OpenCL specification requires that floating point numbers use an IEEE 754 representation [Khr10, §6.1.1], however certain arithmetic operations, such as single precision division, are not required to be accurate [Khr10, §7.4, §9.3.9]. A *maximum relative error* is specified for these operations. Maximum relative error is defined in terms of ulp, or units in the last place. The specification defines ulp as follows [Khr10, §7.4]:

	Maximum relative error for	
Operation	single precision	double precision
$+$	accurate	accurate
$-$	accurate	accurate
$\times$	accurate	accurate
$\div$	≤ 2.5 ulp	accurate
`sqrt`	≤ 3 ulp	accurate

Table 2.1: Maximum relative error of common floating point operations in OpenCL.

> If x is a real number that lies between two finite consecutive floating-point numbers a and b, without being equal to one of them, then $\text{ulp}(x) = |b - a|$, otherwise $\text{ulp}(x)$ is the distance between the two non-equal finite floating-point numbers nearest x. Moreover, ulp(`NaN`) is `NaN`.

The maximum relative error of various common floating point operations in OpenCL is shown in Table 2.1. For example, the `sqrt` function has a maximum relative error of 3 ulp, meaning that the result may differ by up to 3 ulp from the infinitely precise result.

What this means for tools such as KLEE-CL is that we are required to be pessimistic about the floating point capabilities provided by the OpenCL implementation; thus, those floating point arithmetic operations which are permitted by the specification to be inaccurate are assumed to be so. The result of a computation in OpenCL C with a defined maximum relative error cannot definitively be declared equivalent to the result of the same computation in C or C++ (although it can be found equivalent to itself).

In this thesis, we assume an Annex F compliant C and C++ implementation, and that the OpenCL implementation fulfills the floating point storage and computation requirements as set out in the OpenCL specification.

While there has been much work on various verification and testing techniques for floating point programs, much of this work has focused on techniques which are insufficiently rigorous, applicable either only after significant manual effort or have only been shown to work on very small hand-crafted programs. By contrast, the technique described in this thesis requires little manual effort in the most common cases we found in real world code.

Theorem provers such as Coq [BF07] and HOL Light [Har07] have been extended with support for floating point arithmetic, including a set of associated theorems, such as Sterbenz [Ste74]. Using a theorem prover in conjunction with a proof assistant, a user can build a proof that a program satisfies its postconditions given its preconditions. While a theorem prover is one of the most rigorous verification techniques, the user is required to construct a valid set of preconditions and postconditions, and in some cases construct a proof manually with the help of a proof assistant. Not only is this time consuming but the skills required to do so are uncommon, and it would therefore be impractical to expect developers to verify all but the most critical code to this level of detail.

Another proposed technique is that of constraint solving based on approximation with rationals or reals using interval arithmetic [Hol95]. While constraint solving techniques for arbitrary Boolean combinations of polynomial constraints have been proposed, in order to support real arithmetic theories in SMT solvers [AL10], as mentioned in Section 2.1, to our knowledge no constraint solver supports a theory containing both real and bitvector operations and conversions between them. As an alternative approximation, the CBMC [CKL04] constraint solver provides support for fixed point arithmetic as an approximation for floating point arithmetic. This allows for a straightforward bitvector mapping to be used.

Even if we had used rational, real or fixed point arithmetic to approximate floating point arithmetic, such approximations can disguise real problems in floating point code, such as differences in results caused by associativity, precision and rounding. In an attempt to address this issue, floating point constraint solving techniques have been proposed based on interval propagation [BGM06] and projection functions [Mic02]. While promising, these techniques have only been shown to work on small hand-crafted programs, and in particular not programs which use both integer and floating point arithmetic, including conversions between them.

The ASTRÉE [CCF+05] and FLUCTUAT [DGP+09] analysers model floating point arithmetic using an overapproximation based on real arithmetic operations perturbed by an arbitrary absolute or relative error constrained to the maximum error for that operation. While this model is useful for tracking rounding errors, it is insufficient for crosschecking purposes, as it

does not take into account the fact the IEEE 754 floating point operations are deterministic.

More recently, an SMT-LIB theory has been proposed for floating point arithmetic (FPA) [RW10], which specifies a variety of IEEE-754 compliant floating point operations to be used in SMT problems. The proposed theory makes no provision for conversions between integer and floating point values, although this presumably cannot be done within the theory itself, but rather within a combined theory incorporating the FPA and bitvector theories. While there has been work done on the Z3 [dMB08] constraint solver to add support for this theory [RW], the extended constraint solver has yet to be released to the public. More recently, the MathSAT [HGBK12] and SONOLAR [Kro12] SMT solvers have implemented FPA.

In 2009, the CBMC model checker received support for floating point arithmetic [BKW09]. Its `floatbv` module [K$^+$] implements an eager theory solver for floating point arithmetic constraints, i.e., it can be used to bit-blast (see Section 2.1) floating point constraints to equivalent SAT constraints. This module supports most of the operations specified by the FPA theory plus conversions to/from integer bitvectors. In Section 5.2.4 we describe a preliminary integration of `floatbv` into KLEE-CL.

Abstraction can sometimes be a more efficient approach to solving satisfiability problems than bit-blasting alone. In experiments conducted using bitvector [BKO$^+$07, BCF$^+$07] and floating point [BKW09] theory solvers, abstraction gave a speed improvement for the majority of benchmarks. Under the floating point abstraction described in [BKW09], floating point operations are computed using a reduced precision. The result is then extended to the correct precision using either underapproximation (by setting the lower order bits of the mantissa equal to 0) or overapproximation (by leaving them unconstrained). The assignment or proof generated by the decision procedure is then checked against the unapproximated problem, possibly resulting in a refined abstraction being generated using increased precision.

The strategy described in this thesis of building implied constraints (Section 5.2.3) can be viewed as a form of abstraction using overapproximation. We have not implemented automated abstraction refinement, although one potential refinement is an exact bit-blast as discussed in Section 7.3.

Siegel et al. [SMAC06] present a technique based on model checking and symbolic execution for verifying correctness of a parallel floating point program based on its equivalence to a sequential version. Their technique has fundamental similarities to that described in this thesis: floating point operations are treated as uninterpreted functions, and the issue of different expression evaluation orders in parallel programs, which we address using floating point assumptions (Section 5.2.1), is addressed by Siegel et al. using multiple notions of equivalence based on IEEE and real arithmetic. However, this technique has a number of drawbacks. The authors acknowledge that input programs must be manually translated into a model, whereas the technique described in this thesis can operate on unmodified programs. Furthermore, and more fundamentally, the authors do not attempt to address the problem of path explosion caused by symbolic branching, except in the context of one specific benchmark for which a manual adjustment to the benchmark was required [SMAC06, §3.2.4]. By contrast, the static path merging technique described herein (Section 5.1) is capable of automatically eliminating path explosion in certain cases.

An alternative to formal verification or constraint solving is testing using concrete inputs. Random testing can easily be applied to large applications, but random testing alone is known to usually provide low code coverage [GKS05]. The problem is exacerbated under floating point arithmetic, due to the complexity of each individual operation and their associated corner cases [AAF+03]. To address this, random testing has been coupled with constraint solving using constraints derived from previous program runs [GKS05, SMA05] or hand written constraints that are known to cover corner cases [AAF+03].

Chapter 3

Overview of KLEE-CL

In this chapter we give an overview of our techniques for symbolically executing SIMD and OpenCL code, as well as our implementation of those techniques in KLEE-CL.

To apply symbolic execution to the verification of SIMD and OpenCL code, we need to address a series of challenges. First, we need to model the semantics of both a real SIMD instruction set and the OpenCL API, which the current generation of symbolic execution tools do not handle. Second, and more importantly, both SIMD and OpenCL code make intensive use of floating point operations. Due to the complexity of floating point semantics [IEE08], it is extremely difficult — if not infeasible — to build a constraint solver for floating point that is capable of solving a wide variety of real world problems, and as a result such constraint solvers have only recently begun to be developed. Thus, in this work we take a different approach, in which we prove the equivalence of two symbolic floating point expressions by first applying a series of expression canonicalisation rules, and then syntactically matching the two expressions. The key insight into why our approach works is that constructing two equivalent values from the same inputs in floating point can usually only be done reliably by performing the same operations.

To achieve this, the semantics for a substantial portion of the Intel SSE instruction set are implemented via translation to an intermediate representation, and the semantics for a substantial portion of OpenCL is implemented using a symbolic OpenCL model implementing both host-side and device-side functionality. We improve the tractability of our technique by

implementing an aggressive variant of if-conversion using phi-node folding [LA04, CCF03], to replace control-flow forking with predicated `select` instructions (which has similar semantics to the C `?:` operator, except that all operands are evaluated), in order to reduce the number of paths explored by symbolic execution.

3.1 Architecture

Crosschecking a data parallel routine against its serial equivalent using our technique involves three main stages, as illustrated graphically in Figure 3.1.

First, we write a test harness that invokes the serial and data parallel versions of the code on the same symbolic input, and asserts that their results are equal.

Second, we increase the applicability of our technique by transforming the program's LLVM intermediate representation. We first apply an aggressive version of *phi node folding* to statically merge paths (Section 5.1), which reduces the number of paths we have to track by an exponential factor on some benchmarks, and then transform SSE instructions to generic LLVM instructions, which allows us to analyse programs which use SSE (Section 4.3).

Third, we use *symbolic execution* to explore all the feasible paths in the code under test (Section 1.4), while checking whether there are any race conditions (Section 5.3). In order to be able to reason about OpenCL code, our technique implements a symbolic OpenCL model (Section 4.5). Then, on each explored path, we try to prove that the symbolic expressions corresponding to the serial and SIMD variants are equivalent. To do so, we first canonicalise the expressions through a series of expression rewrite rules and analyses, and then use expression matching and constraint solving to prove that the resulting expressions are equivalent (Section 5.2).

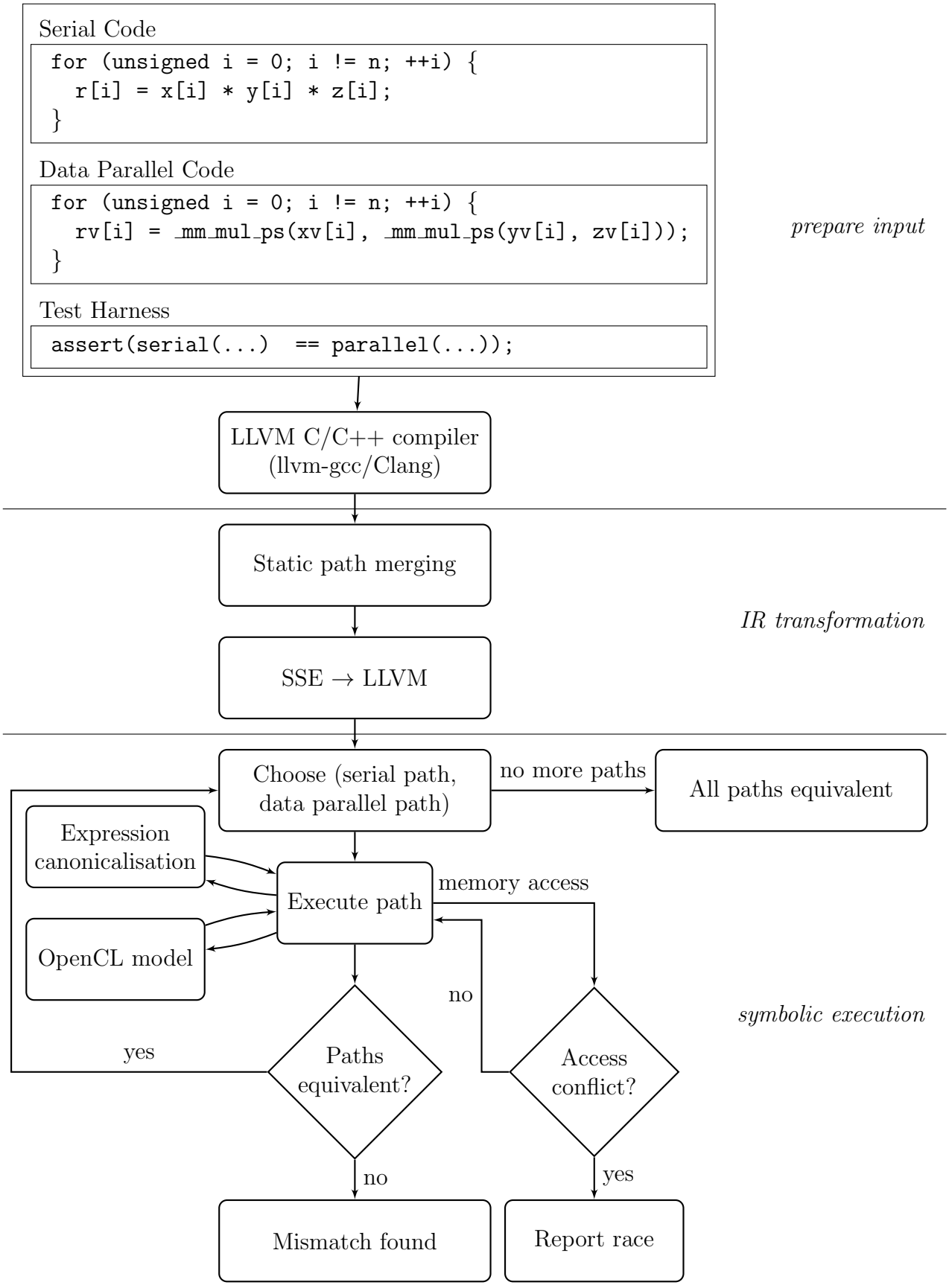

Figure 3.1: Architecture diagram for KLEE-CL.

```
1  void zlimit(int simd, float *src, float *dst,
2              size_t size) {
3    if (simd) {
4      __m128 zero4 = _mm_set1_ps(0.f);
5      while (size >= 4) {
6        __m128 srcv = _mm_loadu_ps(src);
7        __m128 cmpv = _mm_cmpgt_ps(srcv, zero4);
8        __m128 dstv = _mm_and_ps(cmpv, srcv);
9        _mm_storeu_ps(dst, dstv);
10       src += 4; dst += 4; size -= 4;
11     }
12   }
13   while (size) {
14     *dst = *src > 0.f ? *src : 0.f;
15     src++; dst++; size--;
16   }
17 }
18
19 int main(void) {
20   float src[64], dstv[64], dsts[64];
21   uint32_t *dstvi = (uint32_t *)dstv;
22   uint32_t *dstsi = (uint32_t *)dsts;
23   unsigned i;
24   klee_make_symbolic(src, sizeof(src), "src");
25   zlimit(0, src, dsts, 64);
26   zlimit(1, src, dstv, 64);
27   for (i = 0; i < 64; ++i)
28     assert(dstvi[i] == dstsi[i]);
29 }
```

Listing 3.1: Simple test benchmark.

3.2 Walkthrough

This section illustrates the main features of our technique by showing how it can be used to verify the equivalence between a scalar and an SIMD implementation of a simple routine. Our code example, shown in Listing 3.1, is based on one of the OpenCV benchmarks we evaluated[1]. The code defines a routine called `zlimit`, which takes as input a floating point array `src` of size `size`, and returns as output the array `dst` of the same size. Each element of `dst` is the greater of the corresponding elements of `src` and 0. The routine consists of both a scalar and an SIMD implementation; users choose between the two versions via the `simd` argument. The

[1]Specifically `thresh(BINARY_INV, f32)`; see Section 6.1.

SIMD implementation makes use of Intel's SSE instruction set.

The first loop of the routine, at lines 5–11, contains the core of the SIMD implementation, and is a good illustration of how SIMD code is structured. Each iteration of the loop processes four elements of array `src` at a time. The variables `srcv`, `cmpv` and `dstv` are of type `__m128`, i.e., 128-bit vectors consisting of four `float`s each. The code first loads four values from `src` into `srcv` by using the SIMD instruction `_mm_loadu_ps()` (line 6). It then compares each element of `srcv` to the corresponding element of `zero4`, which was initialised on line 4 to a vector of four 0 values (line 7). The output vector `cmpv` contains the result of each comparison as a vector of four 32-bit bitmasks each consisting of all-ones (if the `srcv` element was > 0) or all-zeros (otherwise). Next it applies the `cmpv` bitmask to `srcv` by performing a bitwise `AND` of `cmpv` and `srcv` to produce `dstv`, a copy of `srcv` with values ≤ 0 replaced by 0 (line 8). Finally, it stores `dstv` into `dst` (line 9).

The second loop of the `zlimit` routine, at lines 13–16, is the scalar implementation, which is also used by the SIMD version to process the last few elements of `src` when the size is not an exact multiple of 4.

The `main` function constitutes the test harness. In order to use KLEE-CL, developers have to identify the scalar and the SIMD versions of the code being checked, and the inputs and outputs to these routines. In our example, we have one input, namely the array `src`. Thus, the first step is to mark this array as *symbolic*, meaning that its elements could initially have any value (see Section 1.4 for more details). This is accomplished on line 24 by calling the function `klee_make_symbolic()` provided by KLEE, which takes three arguments: the address of the memory region to be made symbolic, its size in bytes, and a name used for debugging purposes only. Then, on line 25 we call the scalar version of the code and store the result in `dsts`, and on line 26 we call the SIMD version and store the result in `dstv`. Finally, on lines 27–28 each element of `dstv` is compared against the corresponding element of `dsts`. Note that we use bitcasting to integers via the pointers `dstvi` and `dstsi` for a bitwise comparison. As we will further discuss in Chapter 4, this is necessary because in the presence of `NaN` (*Not a Number*) values, the C floating point comparison operator `==` does not always return `true` if

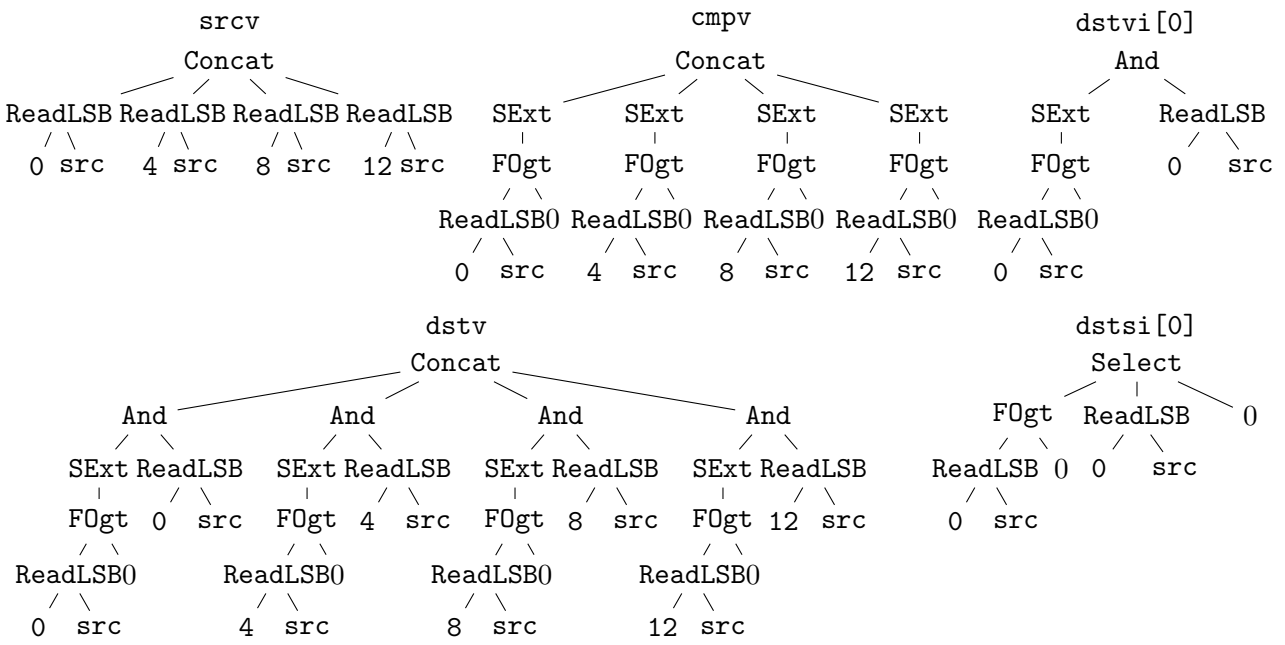

Figure 3.2: Symbolic expressions assigned to variables `srcv`, `cmpv`, `dstv` and to the array elements `dstvi[0]` and `dstsi[0]` of Listing 3.1. `src` represents the symbolic array `src`. The `ReadLSB` (Read Least Significant Byte first) node represents a 4-byte little-endian array read, `FOgt` floating point greater-than comparison, `SExt` sign extension, `Select` the equivalent of the C ternary operator and `Concat` bitwise concatenation.

its floating-point operands are the same, as distinguished from a bitwise comparison.

Before KLEE-CL begins executing the program, it first carries out a number of transformations. One of these is a lowering pass that replaces instruction-set specific SIMD operations with standard, instruction-set neutral instructions. Section 4.3 discusses this pass in more detail.

KLEE-CL interprets a program by evaluating the transformed IR instructions sequentially. During symbolic execution, values representing variables and intermediate expressions are manipulated. Both vector and scalar values are represented as bitvectors: concrete values by bitvector constants and symbolic ones by bitvector expressions. Vectors have bitwidth $s \times n$, where s is the bitwidth of the underlying scalar and n is the number of elements in the vector. Chapter 4 gives more details on our modelling approach.

For example, during the first iteration of the `zlimit` SIMD loop, the variables `srcv`, `cmpv` and `dstv` defined at lines 6–8 in Listing 3.1 are represented by the three expressions shown on the

left hand side of Figure 3.2. Similarly, the results `dstvi[0]` and `dstsi[0]` are represented by the two expressions shown on the right side of Figure 3.2.

When KLEE-CL reaches an `assert` statement, it tries to prove that the associated expression is always `true`. For example, during the first iteration of the loop at lines 27–28 the expressions `dstvi[0]` and `dstsi[0]` are compared. To this end, KLEE-CL applies a series of expression rewrite rules, whose goal is to bring the expressions to a canonical normal form. As discussed in Section 5.2.2, one of our canonicalisation rules transforms an expression tree of the form $\texttt{And}(\texttt{SExt}(P), X)$ into $\texttt{Select}(P, X, 0)$, where P is an arbitrary boolean predicate and X an arbitrary expression. For our example, this rule transforms the expression corresponding to `dstvi[0]` shown in Figure 3.2 to be identical to expression `dstsi[0]`, shown in the same figure. Once both expressions are canonicalised, we attempt to prove their equivalence by (1) using a simple syntactical matching for the floating-point subtrees, and (2) using a constraint solver for the integer subtrees. As highlighted in the introduction, the reason we are able to prove the equivalence of floating-point expressions by bringing them to canonical form and then syntactically matching them is that constructing two equivalent values from the same inputs in floating point can usually only be done reliably in a limited number of ways. As a consequence, we found that in practice we only need a relatively small number of expression canonicalisation rules in order to apply our technique to real code (see Section 5.2.2).

One concern not covered by this simple example, which has a single execution path, is the number of proofs that are needed: under symbolic execution, every feasible program path is explored, and we have to conduct the proof on every path. Thus, an important optimisation is to reduce the number of paths explored by merging multiple ones together. This optimisation is discussed in detail in Section 5.1.

Chapter 4

Modelling Data Parallel Operations

This chapter discusses our approach to modelling floating point arithmetic, SIMD and OpenCL in KLEE-CL. In Section 4.1 we start by presenting our floating point extension to KLEE. Then, in Section 4.2 we describe our modelling of SIMD vector operations, and in Section 4.3 we present our lowering pass that translates SSE intrinsics into standard LLVM operations. In Section 4.4 we discuss the way in which we handle LLVM atomic intrinsics. Finally, in Section 4.5 we discuss our approach to modelling the OpenCL runtime library.

4.1 Floating Point Operations

In order to add support for floating point arithmetic, we extended KLEE's constraint language to include floating point types and operations. Floating point operation semantics are derived from those presented by LLVM, which are themselves derived from the semantics defined in the IEEE 754-2008 standard [IEE08]. The set of operations includes $+$, $-$, $\times$, $\div$, remainder, conversion to and from signed or unsigned integer values (`FPToSI`, `FPToUI`, `UIToFP`, `SIToFP`), conversion between floating point precisions (`FPExt`, `FPTrunc`) and the relational operators $<$, $=$, $>$, $\leq$, $\geq$ and $\neq$. We support the two most common floating point types specified by IEEE 754-2008, namely single precision (binary32) and double precision (binary64), together with half precision, 80-bit double extended precision and quadruple precision. Because all symbolic

FCmp operation	Shorthand	Meaning
$\texttt{FCmp}(X, Y, \emptyset)$	$\bot$	False (always simplified)
$\texttt{FCmp}(X, Y, \{=\})$	$\texttt{FOeq}(X, Y)$	Ordered $=$
$\texttt{FCmp}(X, Y, \{<\})$	$\texttt{FOlt}(X, Y)$	Ordered $<$
$\texttt{FCmp}(X, Y, \{<, =\})$	$\texttt{FOle}(X, Y)$	Ordered $\leq$
$\texttt{FCmp}(X, Y, \{>\})$	$\texttt{FOgt}(X, Y)$	Ordered $>$
$\texttt{FCmp}(X, Y, \{=, >\})$	$\texttt{FOge}(X, Y)$	Ordered $\geq$
$\texttt{FCmp}(X, Y, \{<, >\})$	$\texttt{FOne}(X, Y)$	Ordered $\neq$
$\texttt{FCmp}(X, Y, \{<, =, >\})$	$\texttt{FOrd}(X, Y)$	Ordered test
$\texttt{FCmp}(X, Y, \{\texttt{UNO}\})$	$\texttt{FUno}(X, Y)$	Unordered test
$\texttt{FCmp}(X, Y, \{\texttt{UNO}, =\})$	$\texttt{FUeq}(X, Y)$	Unordered $=$
$\texttt{FCmp}(X, Y, \{\texttt{UNO}, <\})$	$\texttt{FUlt}(X, Y)$	Unordered $<$
$\texttt{FCmp}(X, Y, \{\texttt{UNO}, <, =\})$	$\texttt{FUle}(X, Y)$	Unordered $\leq$
$\texttt{FCmp}(X, Y, \{\texttt{UNO}, >\})$	$\texttt{FUgt}(X, Y)$	Unordered $>$
$\texttt{FCmp}(X, Y, \{\texttt{UNO}, =, >\})$	$\texttt{FUge}(X, Y)$	Unordered $\geq$
$\texttt{FCmp}(X, Y, \{\texttt{UNO}, <, >\})$	$\texttt{FUne}(X, Y)$	Unordered $\neq$
$\texttt{FCmp}(X, Y, \{\texttt{UNO}, <, =, >\})$	$\top$	True (always simplified)

Table 4.1: Floating point predicate shorthand semantics.

expressions are untyped bitvectors, type information is associated with operations, rather than operands.

Of particular importance for our crosschecking algorithm (Section 5.2) is the fact that relational operators can occur in both *ordered* and *unordered* form. Ordered and unordered operators differ in the way they treat `NaN` values: if any operand is a `NaN`, ordered comparisons always evaluate to `false` while unordered ones to `true`. C implementations that comply with Annex F of the ISO C standard [Int99] are required to provide ordered relational operators, except for `!=`, which is unordered $\neq$ [IEE08, §5.7], however unordered variants of all operators are accessible using the `!` and `||` operators (for example, `!(x < y || x > y)` is equivalent to unordered $=$).

A comparison of two floating point values x and y must have one of four mutually exclusive outcomes: $x < y$, $x = y$, $x > y$ or x `UNO` y (*unordered*, i.e. either or both of x and y are `NaN`). We establish a set $\mathbf{O} = \{<, =, >, \texttt{UNO}\}$ of these outcomes. Then, any floating point relational operator may be represented by a subset of $\mathbf{O}$: for example, ordered $\leq$ (`FOle`) is represented by $\{<, =\}$.

In KLEE-CL, all floating point relational operators are represented using a generic `FCmp` expression. The first two operands to `FCmp` are the comparison operands, whereas the third

operand is a subset of **O**, known as the *outcome set* (represented internally using a vector of four bits, based on the floating point predicate representation used by LLVM [LA04]). In this thesis we normally refer to predicate operations using shorthand names rather than using `FCmp`. Table 4.1 gives a list of mappings between shorthand names and associated `FCmp` operations. In Section 5.2.2 we show how outcome sets can be used to simplify expressions involving floating-point comparisons.

In floating point arithmetic, each non-relational floating point operation uses a *rounding mode* to round the infinitely precise result to one that can be represented as a floating point value, or in the case of a floating point to integer conversion, an integer value. The default rounding mode is *round to nearest, ties to even* which rounds results to the nearest representable value but in the case that the result is equidistant from two representable values, chooses the representation in which the least significant bit of the mantissa is 0. Another rounding mode, which is used for floating point to integer conversions in C and C++, is *round to zero*, which always discards the fractional component, rounding values towards 0. Because none of the code we worked with changes the rounding mode, we did not find it necessary to model the current rounding mode. However, SSE provides a floating point to integer conversion which uses the current rounding mode. Therefore, all operations use round to nearest, ties to even, except for float to int conversions, which have an associated rounding mode of either round to nearest or round to zero.

In the LLVM intermediate representation, floating point operations which are permitted to be inaccurate may be marked with special metadata which indicates the maximum relative error of the result of that operation, in ulps (units in the last place; see Section 2.6). The Clang compiler that we use to compile OpenCL C kernels will add this metadata to single precision floating point division operations.[1] When KLEE-CL encounters this metadata on an LLVM floating point instruction it will build an unconstrained symbolic value, rather than the expression that would normally be created.

[1]This is the only instance of a floating point operator which is permitted by the OpenCL C standard to be inaccurate. All other potentially inaccurate operations, such as `sqrt`, are provided as builtin functions. The necessary support does not currently exist in Clang for builtin functions to receive this metadata, and as such we do not model this aspect of `sqrt` and other OpenCL C builtin functions correctly.

Note that the value is unconstrained because any constraints we may impose on it (for example, to bring it within the required range of the correct result) would not be recognised by our expression matching technique (Section 5.2), which is based on subexpression matching rather than constraints, and is only designed to work with exact equality tests rather than inequalities formed from ranged equivalence tests. Thus, it would be futile to add such constraints. Given a more precise floating point constraint solver, such as the bit blaster discussed in Section 5.2.4, such constraints could be used to eliminate false positives in certain cases, particularly those involving ranged equivalence tests, although this has not been implemented.

Note also that we return a fresh unconstrained value wherever we encounter an inaccurate floating point operation, regardless of whether the same operation may have been encountered before with the same operands (which may, for example, occur when cross-checking two independent OpenCL based implementations of an algorithm). This is weaker than our normal uninterpreted-function treatment of floating point operations, and is necessary because the OpenCL C compiler is free to use any implementation of the floating point operation at any given program point (provided that it fulfills the accuracy requirements), and is not required to be consistent between program points. For example, the compiler may choose to use an exact floating point division at one program point in a first implementation of an algorithm, and an inexact floating point division at another program point in a second implementation.

4.2 SIMD Operations

KLEE-CL's implementations of SSE and of OpenCL C's SIMD capabilities are based on generic support for SIMD vector operations.

Intel's Streaming SIMD Extension operates on a set of eight 128-bit vector registers, called *XMM* registers. Each of these registers can be used to pack together either four 32-bit single-precision floats, two 64-bit double-precision floats, or various combinations of integer values (e.g., four 32-bit ints, or eight 16-bit shorts).

The OpenCL C language provides a wider range of vector types. The base type of a vector may

be an integer type of width between 8 and 64 bits or a floating point type of width between 16 and 64 bits. The number of vector elements may range between 2 and 16. Thus, OpenCL C vectors may be between 16 and 1024 bits wide.

Since a vector may be bit-cast to another vector of the same size but of a different data type (for example, by using the `as_type<N>` operator in OpenCL C), it is possible to perform an operation of a certain type on the result of an operation of a different type: e.g., one could perform a single-precision computation on the result of a double-precision, or even integer, computation. As a consequence, in order to capture the precise semantics of SIMD vector operations, it is important to model SIMD vectors at the bit level. Fortunately, KLEE already models its constraints with bit-level accuracy [CDE08] by using the *bitvector* data type provided by its underlying constraint solver, STP [GD07]. Thus, we model each vector as an STP bitvector that can be treated as storing different data types, depending on the instruction that uses the vector.

At the LLVM intermediate language level, SIMD vectors are represented as typed arrays. There are three generic operations that operate on these vectors: `insertelement`, `extractelement` and `shufflevector`. Many other LLVM instructions, such as `add`, perform element-wise operations on vectors. Some SSE instructions, together with all of the OpenCL C builtin functions that we implemented, are implemented in terms of these instructions. All other SSE instructions are implemented as LLVM intrinsics, as discussed in the next section.

The `extractelement` operation takes as arguments a vector (e.g., an eight element vector of 16-bit integers) and an offset into this vector, and returns the element at that offset. For example,

```
%res = extractelement <8 x i16> %a, i32 3
```

extracts the fourth element of the vector `a` (which contains eight 16-bit shorts) and stores it in `%res`.

The `insertelement` instruction takes a vector, a value and an offset, and returns a vector identical to the supplied vector except with the value at the given offset replaced with the

given value. For example,

```
%res = insertelement <8 x i16> %a, i16 10, i32 2
```

returns in `%res` a vector with all values equal to those of the vector `%a` except for the third element which receives the value 10.

The `shufflevector` instruction takes two vectors of the same type and returns a permutation of elements from those two vectors. The permutation is specified using an immediate vector argument whose elements represent offsets into the vectors. For example,

```
%res = shufflevector <4 x float> %a, <4 x float> %b,
    <4 x i32> <i32 0, i32 1, i32 4, i32 5>
```

returns in `%res` a vector with its 2 lower order elements taken from the 2 lower order elements of `%a` and its 2 higher order elements from the 2 lower order elements of `%b`.

In our implementation, we model these three operations, together with the element-wise LLVM instructions, using the bitvector extraction and concatenation primitives provided by STP. The modelling is straightforward. For example, if A is the 128-bit bitvector representing the vector `%a`, $\texttt{Extract}^{16}(A, 48)$ is the bitvector expression encoding the `extractelement` operation above, where $\texttt{Extract}^{W}(BV, k)$ extracts a bitvector of size W starting at offset k of bitvector BV.

4.3 SSE Intrinsic Lowering

Not all SSE instructions are implemented in terms of vector operations; most of them are represented using LLVM intrinsics. To enable comparison with scalar code, we implemented a pass that translates them into standard LLVM instructions by making use of the `extractelement` and `insertelement` operations presented in Section 4.2.

We added support for the 37 SSE intrinsics shown in Table 4.2. These 37 intrinsics were sufficient to handle the benchmarks with which we evaluated our technique (Chapter 6). An example of a call to an SSE-specific intrinsic is shown below:

LLVM intrinsic (`llvm.x86.`)	# Occurrences in OpenCV	Instruction	Function
sse2.storel.dq	67	MOVQ	Move Quadword
sse2.loadu.dq	207	MOVDQU	Move Unaligned Double Quadword
sse2.storeu.dq	139		
sse.loadu.ps	221	MOVUPS	Move Unaligned Packed Single-Precision Floating-Point Values
sse.storeu.ps	109		
sse.cmp.ps	19	CMPPS	Compare Packed Single-Precision Floating-Point Values
sse2.cvtdq2ps	57	CVTDQ2PS	Convert Packed Dword Integers to Packed Single-Precision FP Values
sse2.cvtps2dq	64	CVTPS2DQ	Convert Packed Single-Precision FP Values to Packed Dword Integers
sse2.cvtsd2si	1	CVTSD2SI	Convert Scalar Double-Precision FP Value to Integer
sse.max.ps	4	MAXPS	Return Maximum Packed Single-Precision Floating-Point Values
sse.min.ps	6	MINPS	Return Minimum Packed Single-Precision Floating-Point Values
sse2.packssdw.128	91	PACKSSDW	Pack with Signed Saturation
mmx.packssdw	0		
sse2.packsswb.128	1	PACKSSWB	
mmx.packsswb	0		
sse2.packuswb.128	23	PACKUSWB	Pack with Unsigned Saturation
mmx.packuswb	0		
sse2.padds.w	36	PADDSW	Add Packed Signed Integers with Signed Saturation
sse2.paddus.b	3	PADDUSB	Add Packed Unsigned Integers with Unsigned Saturation
sse2.paddus.w	4	PADDUSW	
sse2.pcmpgt.b	13	PCMPGTB	Compare Packed Signed Integers for Greater Than
sse2.pcmpgt.w	13	PCMPGTW	
sse2.pmadd.wd	39	PMADDWD	Multiply and Add Packed Integers
sse2.pmaxs.w	6	PMAXSW	Maximum of Packed Signed Word Integers
sse2.pmaxu.b	5	PMAXUB	Maximum of Packed Unsigned Byte Integers
sse2.pmins.w	37	PMINSW	Minimum of Packed Signed Word Integers
sse2.pminu.b	5	PMINUB	Minimum of Packed Unsigned Byte Integers
sse2.pmulh.w	31	PMULHW	Multiply Packed Signed Integers and Store High Result
sse2.psad.bw	1	PSADBW	Compute Sum of Absolute Differences
sse2.psll.dq.bs	16	PSLLDQ	Shift Double Quadword Left Logical
sse2.pslli.w	5	PSLLW	Shift Packed Data Left Logical
sse2.psrai.d	60	PSRAD	Shift Packed Data Right Arithmetic
sse2.psrai.w	5	PSRAW	
sse2.psrl.dq.bs	20	PSRLDQ	Shift Double Quadword Right Logical
sse2.psrli.w	3	PSRLW	Shift Packed Data Right Logical
sse2.psubus.b	17	PSUBUSB	Subtract Packed Unsigned Integers with Unsigned Saturation
sse2.psubus.w	11	PSUBUSW	

Table 4.2: SSE intrinsics supported by KLEE-CL.

```
%res = call <8 x i16> @llvm.x86.sse2.pslli.w(
           <8 x i16> %arg, i32 1)
```

This instruction shifts every element of `%arg` left by 1 yielding `%res`. The lowering pass transforms this call into the following sequence of instructions:

```
%1 = extractelement <8 x i16> %arg, i32 0
%2 = shl i16 %1, 1
%3 = insertelement <8 x i16> undef, i16 %2, i32 0
%4 = extractelement <8 x i16> %arg, i32 1
%5 = shl i16 %4, 1
%6 = insertelement <8 x i16> %3, i16 %5, i32 1
...
%22 = extractelement <8 x i16> %arg, i32 7
%23 = shl i16 %22, 1
%res = insertelement <8 x i16> %21, i16 %23, i32 7
```

These instructions carry out the same task as the intrinsic but are expressed in terms of the standard LLVM instructions `insertelement`, `extractelement` and `shl`.

4.4 Atomic Intrinsics

LLVM provides a number of intrinsics which are used to represent read-modify-write atomic operations. Since our OpenCV benchmarks (Section 6.1) use atomic operations, we needed to add support for them to KLEE-CL.

An example of such an LLVM atomic intrinsic is the following:

```
%res = call i32 @llvm.atomic.load.add.i32.p0i32(
           i32* %ptr, i32 1)
```

This operation atomically loads a 32-bit integer from the given memory pointer `%ptr`, increments it, stores the result to `%ptr` and returns the value originally loaded from `%ptr` in `%res`.

To handle atomic operations, KLEE-CL applies a transformation pass to all input programs which simply lowers them to equivalent sequences of non-atomic instructions. For example, the atomic operation shown above is translated to:

```
%res = load i32* %ptr
%1 = add i32 %res, 1
store i32 %1, i32* %ptr
```

Our atomic lowering pass handles all 13 atomic intrinsics supported by LLVM 2.7, and was subsequently contributed to the main LLVM branch to be used by similar tools.

Of course, this approach to handling atomic operations is only sound under the assumption that the program is free of preemptible operations, such as threading or signals. As mentioned in Section 2.5, KLEE-CL does not attempt to model atomic operations in OpenCL, and the builtin atomic functions provided by OpenCL (which may be implemented in terms of these intrinsics) are not available to OpenCL C kernels in KLEE-CL.

4.5 OpenCL

Our OpenCL model is a partial implementation of the Khronos OpenCL 1.1 Specification [Khr10], which has been developed to meet the needs of a wide range of OpenCL client programs, including those we evaluated (Chapter 6). It focuses on the general-purpose computation features of OpenCL, rather than its graphical functionality such as textures, samplers and OpenGL interoperability.

The OpenCL model is made up of two distinct parts: the runtime library, which is used by the host to manage the execution of OpenCL kernels, and the OpenCL C environment, which models the execution of a kernel on the device.

4.5.1 The OpenCL C Work-Item Environment

In this section, we describe our modelling of the execution of an entire NDRange, including the facilities presented to the OpenCL C program.

In our model, each work-item in the NDRange is modelled using a single POSIX thread. We use the POSIX threading model added to KLEE by Cloud9 [BUZC11]. This threading model sequentialises thread execution using a run-until-yield scheduling strategy, meaning that a thread will run until it explicitly yields its execution to other threads. An OpenCL work-item can only yield using an execution barrier (i.e. a call to the `barrier` function) or when the kernel function returns, so if the kernel does not contain any execution barriers each work-item will run to completion sequentially.

An example of a valid NDRange schedule containing execution barriers is illustrated in Figure 4.1. As can be seen, an execution barrier causes the preemption of all work-items in the work-group until the point at which all work-items in the work-group have reached the call to `barrier`. Correct modelling of `barrier` is essential for our race detection technique to function correctly, as will be discussed in Section 5.3.

To implement `barrier`, we use the *wait list* synchronisation primitive provided by Cloud9. Waiting on a wait list causes a thread to block until another thread *notifies* the wait list. To support `barrier`, we implemented an additional synchronisation function, `klee_thread_barrier`, which causes the thread to wait on the wait list unless the number of threads waiting on the wait list (plus the thread that called `klee_thread_barrier`) has reached a specified size. If that is the case, `klee_thread_barrier` will notify the wait list, unblocking the other threads, and then reset the memory access records associated with the data race detector (explained in further detail in Section 5.3).

There is one wait list per work-group (the *local wait list*), plus a wait list for the entire NDRange (the *global wait list*). When a kernel function calls `barrier`, we call `klee_thread_barrier` on the work-group's local wait list with a size equal to that of the work-group, and once a kernel function returns, we call `klee_thread_barrier` on the global wait list with a size equal to that

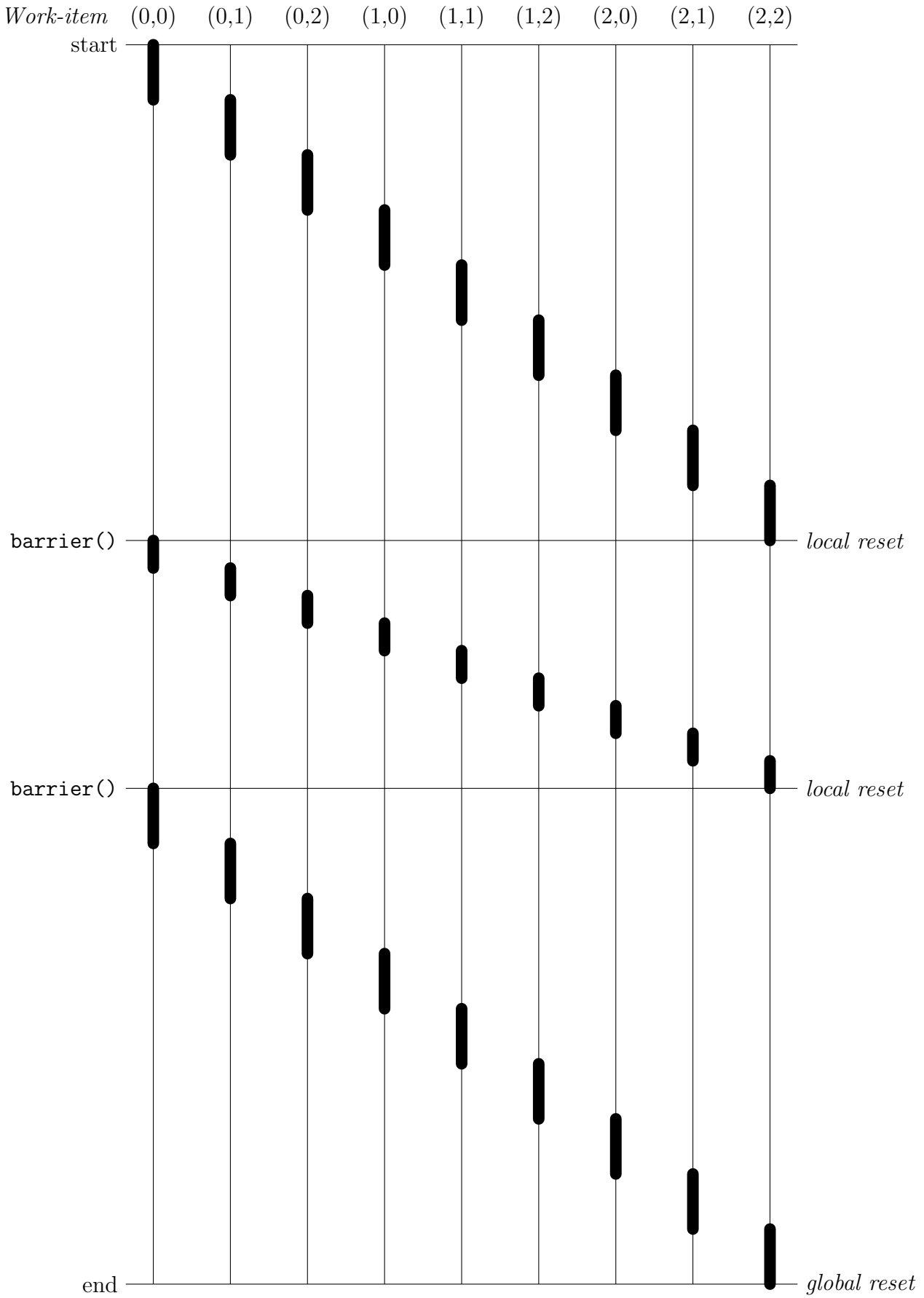

Figure 4.1: Work-item scheduling for a work-group of size (3,3) whose kernel contains two calls to the `barrier` function.

of the NDRange.

The vector data types provided by OpenCL are used to exploit the SIMD capabilities common among GPUs. For example, `float4` is the name of a data type referring to a vector of four `float` values. Vector types are implemented via the SIMD support discussed in Section 4.2.

The four disjoint address spaces provided by OpenCL are named `__global`, `__local`, `__constant` and `__private`. Globally available data resides in `__global`, data local to a work-group in `__local`, read-only data in `__constant` and function arguments and local variables in `__private`.

Three of these address spaces (`__global`, `__constant` and `__private`) can be modelled using the generic address space used by regular C code, which is shared across all work-items. The `__constant` address space is protected from modification by the language [Khr10, §6.5.3], so there is no need to use a separate address space in KLEE-CL. It may seem unintuitive to model `__private` using a shared memory space, however it is not normally possible for two work-items to legally share pointers to each other's `__private` variables, so it is generally safe to do this.[2] The `__local` address space, however, needs special attention because `__local` data must be shared between work-items in the same work-group, and each work-group must have its own `__local` data. To model `__local`, we added a *group-local* address space, which is an address space shared between user-created thread groups. Each thread belongs to a single thread group. Before beginning kernel execution, we create one thread group for each work-group, and set each thread's group to match its work-group.

We found that most OpenCL C programs use very few of the available built-in functions. Thus, our model implements 18 of the over 500 built-in functions specified by the OpenCL 1.1 specification, which are enough to run our benchmarks. These include various work-item functions, math functions and the `barrier` synchronisation function. The available built-in functions are listed in Table 4.3.

[2] A trick with barrier synchronisation can be used to share pointers to `__private` memory, but such pointers are unusable in other work-items. Because this is quite an esoteric trick, we did not find it necessary to model it.

Section	Function
Work-Item Functions	get_work_dim get_global_size get_global_id get_local_size get_local_id get_num_groups get_group_id get_global_offset
Math Functions	native_divide native_cos native_sin native_sqrt
Geometric Functions	cross dot length normalize
Relational Functions	select
Synchronization Functions	barrier

Table 4.3: OpenCL C builtin functions supported by KLEE-CL.

4.5.2 The OpenCL Runtime Library

The OpenCL runtime library is specified by two sections of the OpenCL specification: the OpenCL Platform Layer [Khr10, § 4] and the OpenCL Runtime [Khr10, § 5]. The Platform Layer is used to query the set of available OpenCL devices, while the Runtime is used to query and manipulate objects on a specific device or set of devices such as device-side memory buffers and compiled OpenCL programs. In total, our model implements 30 of the 98 runtime library functions specified as part of the Platform Layer and Runtime. Table 4.4 lists the OpenCL runtime library functions implemented as part of our model. Below, we discuss our modelling of each of the modelled subsections of Sections 4 and 5 of the OpenCL specification.

Platform Layer

The Platform Layer implementation presents a single OpenCL device to the client program. This device presents itself as a CPU-based device with support for the `cl_khr_fp64` extension, which allows the kernel to use double-precision floating point arithmetic.

Section	Function
Querying Platform Info	`clGetPlatformIDs` `clGetPlatformInfo`
Querying Devices	`clGetDeviceIDs` `clGetDeviceInfo`

Contexts	`clCreateContext`
	`clRetainContext`
	`clReleaseContext`
	`clGetContextInfo`

Command Queues	`clCreateCommandQueue`
	`clRetainCommandQueue`
	`clReleaseCommandQueue`

Buffer Objects	`clCreateBuffer`
	`clEnqueueReadBuffer`
	`clEnqueueWriteBuffer`

Memory Objects	`clRetainMemObject` `clReleaseMemObject`

Program Objects	`clCreateProgramWithSource`
	`clBuildProgram`
	`clRetainProgram`
	`clReleaseProgram`

Kernel Objects	`clCreateKernel`
	`clRetainKernel`
	`clReleaseKernel`
	`clSetKernelArg`

Executing Kernels	`clEnqueueNDRangeKernel` `clEnqueueTask`

Event Objects	`clWaitForEvents`
	`clRetainEvent`
	`clReleaseEvent`

Flush and Finish	`clFinish`

Table 4.4: OpenCL runtime library functions supported by KLEE-CL.

Command Queues

An OpenCL *command queue* represents a queue of operations to be performed on the device. Command queues are created using the `clCreateCommandQueue` function.

A client program may create an unlimited number of command queues per OpenCL device, and schedule work on them independently of one another. Client programs may also create *out-of-order* command queues, which permit the implementation to schedule commands out of order, by supplying the `CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE` flag at command queue creation time.

By scheduling multiple kernel invocations on an out-of-order command queue, or by scheduling kernel invocations across multiple command queues, a client program may cause kernel NDRanges to run in parallel such that races may occur between NDRanges. Because this would complicate race detection (Section 5.3), in this work, we concern ourselves only with the more common in-order case where only one NDRange is executing at a time. Therefore, we do not correctly model programs which create multiple command queues or which use the `CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE` flag. In Section 7.2 we discuss one possible extension of this work which would allow us to correctly model and detect races between kernel invocations in the case where multiple or out-of-order command queues are used.

Event Objects

An OpenCL *event* object refers to a specific pending command on a command queue, such as a kernel invocation or a memory access, however in our model, we only model kernel invocation events (all other commands are performed synchronously). An event is modelled as a reference to a set of POSIX threads, and in the case of a kernel invocation the set encompasses all work-items in the kernel invocation's NDRange.

The `clWaitForEvents` function is used to wait for a given set of events to complete. If `clWaitForEvents` is called we wait for each thread in the set to terminate using `pthread_join`.

Flush and Finish

The `clFinish` function is used to wait for all commands in a given command queue to terminate. Our implementation uses `clWaitForEvents` to wait for each pending event in the command queue.

Buffer Objects

An OpenCL *buffer* represents a block of device accessible memory. Buffers may be created using the `clCreateBuffer` function and destroyed using the `clReleaseMemObject` function. Buffers may reside either on the host or on the device, and the implementation is responsible for copying data between the host and the device as necessary. KLEE-CL models all memory buffers as dynamically allocated blocks of memory in the generic address space, except for memory buffers allocated using the `CL_MEM_USE_HOST_PTR` flag, which causes the implementation to use a user supplied block of memory directly.

By dynamically allocating and deallocating memory at the request of the user of the API, we are able to utilise KLEE's existing support for detecting dynamic memory errors, such as use-after-free, to detect memory errors resulting from the interaction between OpenCL programs and OpenCL C kernels, as we will discuss in Section 4.5.3.

`clEnqueueReadBuffer` and `clEnqueueWriteBuffer` are the two functions most commonly used to read from and write to memory buffers. These functions support two modes of operation: blocking and non-blocking. Blocking operations are carried out synchronously, and are therefore guaranteed to have completed (together with any previous commands in the command queue) before the function returns. On the other hand, non-blocking operations merely add the memory access command to the command queue and return immediately.

In our model, we currently only model blocking memory accesses accurately, because we currently do not support memory access event objects. Both blocking and non-blocking memory accesses will wait for the supplied command queue to be emptied using `clFinish` and then perform the memory access by copying data to/from the buffer's memory block. We must also

wait in the non-blocking case to preserve the ordering relationship between any kernel invocations in the command queue and the present memory access, and as we will see in Section 4.5.3, this reduces the scope of detectable memory errors.

Program Objects

An OpenCL *program* object refers to a compiled OpenCL C translation unit. Program objects may be created using source code (using `clCreateProgramWithSource`) or precompiled binaries (using `clCreateProgramWithBinary`). Our model only supports creation from source code.

OpenCL program objects created using `clCreateProgramWithSource` must be compiled using the `clBuildProgram` function before they can be used. Our implementation of `clBuildProgram` invokes a compiler based on the OpenCL C front-end provided by the Clang [cla] compiler. Clang is designed to be used as a library, which made it easy to integrate into KLEE-CL. Clang produces an LLVM [LA04] module representing the compiled program.

LLVM modules built using Clang are then dynamically loaded into the running instance of KLEE-CL. To implement this, we added dynamic multiple LLVM module support to KLEE-CL. This multiple module support allows for new LLVM modules to be dynamically introduced into a running instance of KLEE-CL, and for globals to be dynamically looked up by name using a new special function, `klee_lookup_module_global`.

While dynamic multiple LLVM module support is only used for OpenCL C modules in KLEE-CL, an example of another use case for this feature would be to implement a model for the POSIX interface to the dynamic linking loader (i.e. the `dlopen`, `dlsym`, etc. functions).

Kernel Objects

An OpenCL *kernel* object refers to an individual kernel function (a function marked with the `__kernel` attribute) within an OpenCL C translation unit. The `clCreateKernel` function is used to look up a kernel with a specified name given a reference to an OpenCL C program object. `klee_lookup_module_global` was used to implement this function.

The `clEnqueueNDRangeKernel` function discussed in Section 1.3 is implemented by creating a local thread group for each work-group and the local and global wait lists mentioned in Section 4.5.1, and then starting one modelled POSIX thread for each work-item in the NDRange. Each thread sets its thread group to the thread group assigned for its work-group, initialises thread-local variables indicating the local work-item identifier, and then calls the kernel function.

Because kernel invocations must occur in the correct order (assuming an ordered command queue), our implementation of `clEnqueueNDRangeKernel` waits for all previous kernel invocations to terminate using `clFinish` before starting threads for the current kernel invocation. In practice, this implies that at most one event may reside on a command queue at a time, and that that event must be a kernel invocation. In Section 4.5.3 we discuss the consequences of this with regard to memory error detection.

The `clSetKernelArg` function is used to set individual arguments to the kernel function. This function takes a zero-based offset together with the value of the argument. `clSetKernelArg` is required to detect and diagnose (through an error code) any invalid uses of that function, such as incorrectly typed arguments and out-of-range offsets.

To support detection of invalid uses of `clSetKernelArg`, to extract the correct information from arguments supplied to `clSetKernelArg` and to call a kernel function correctly required us to implement a mechanism for (1) introspecting functions to discover the types of their arguments and for (2) calling functions using a pre-built list of arguments, similar in some ways to the *reflection* capability provided by the Java standard library.

To support (1), KLEE-CL provides two introspection functions which, given a function pointer, will return specific information about that function. `klee_ocl_get_arg_count` will return the number of arguments accepted by the given function, and `klee_ocl_get_arg_type` takes a zero-based offset and returns an integer value indicating the type of the argument at that offset.

To support (2), KLEE-CL contains support for indirect function calls. The `klee_icall` special function is used to call a given function with a specified argument list. To manage argument

```
1 cl_mem buf1 = clCreateBuffer(context, ...);
2 cl_mem buf2 = clCreateBuffer(context, ...);
3 clEnqueueWriteBuffer(cmd_queue, buf1, /*blocking_write*/ CL_TRUE, ...);
4 clSetKernelArg(kernel, 0, sizeof(cl_mem), &buf1);
5 clSetKernelArg(kernel, 1, sizeof(cl_mem), &buf2);
6 clEnqueueNDRangeKernel(cmd_queue, kernel, ...);
7 clReleaseMemObject(buf1);
8 clEnqueueReadBuffer(cmd_queue, buf2, /*blocking_read*/ CL_TRUE, ...);
```

Listing 4.1: Fragment of an OpenCL program containing a use-after-free error.

lists, three special functions are provided: `klee_icall_create_arg_list`, which creates an argument list; `klee_icall_add_arg`, which appends an argument to the end of an argument list; and `klee_icall_destroy_arg_list`, which destroys an argument list.

4.5.3 Detecting Memory Errors in OpenCL Programs

As mentioned in Section 4.5.2, OpenCL memory buffers are modelled by dynamically allocating and deallocating device-accessible memory on demand. Because an OpenCL C program cannot dynamically allocate or deallocate memory, use-after-free errors are normally caused by an OpenCL C program accessing memory which has been deallocated by the host. Our model has been constructed so as to detect many errors of this sort.

While use-after-free errors are traditionally detected by examining memory usage in a serial fashion, OpenCL C introduces a more insidious class of use-after-free errors, due to the asynchronous nature of kernel invocation. For example, consider the host program fragment shown in Listing 4.1 (simplified from our Parboil `mri-q` benchmark, see Section 6.2). On lines 1 and 2 we create two memory buffer objects `buf1` and `buf2`, and on line 3 we perform a blocking write of input data to `buf1`. On lines 4 and 5 we set the first and second arguments to kernel `k` (shown in Listing 4.2) to (respectively) `buf1` and `buf2`, and on line 6 we enqueue an invocation of `k`, a kernel which accesses memory from both of its arguments. Then on line 7, we free the memory buffer `buf1`, and this is where the error lies. Because `k` was invoked asynchronously, it may not have completed execution before the host reaches line 7, and a read of `buf1` in `k`, such as that on line 2, would result in a memory error.

```
1 __kernel void k(__global int *buf1, __global int *buf2) {
2   buf2[get_global_id(0)] = buf1[get_global_id(0)]*2;
3 }
```

Listing 4.2: An OpenCL C kernel demonstrating the use-after-free error.

While most OpenCL objects use reference counting to retain references where appropriate, the `clSetKernelArg` function is an exception to this rule (the specification [Khr10, §5.7.2] gives a justification for this), and so the kernel invocation mechanism is uniquely susceptible to this problem.

To fix such errors, one needs to ensure that, before releasing a memory buffer object, no command queue contains a kernel invocation which may access that memory buffer. This can be done in several ways:

1. By calling the `clFinish` function as many times as necessary supplying as an argument any command queues containing such kernel invocations.
2. By calling the `clWaitForEvents` function supplying as arguments the event objects for any such kernel invocations.
3. By calling a function such as `clEnqueueReadBuffer` or `clEnqueueWriteBuffer` in blocking mode (i.e. with the `blocking_read` or `blocking_write` argument set to `CL_TRUE`), supplying as an argument the command queue containing such kernel invocations (the OpenCL runtime contains other blocking functions which may be used for this purpose, but which are not part of our model).

In the case of Listing 4.1, we can fix the error by simply swapping lines 7 and 8, as shown in Listing 4.3.

Compounding the difficulty of avoiding use-after-free errors in OpenCL, it is easy to imagine how such errors may be obscured. In the original `mri-q` benchmark, the calls to `clReleaseMemObject` and `clEnqueueReadBuffer` were in different functions in the source code. Furthermore, it is not always clear to the untrained eye that functions such as `clEnqueueReadBuffer` or `clEnqueueWriteBuffer` will have the desired effect if invoked in blocking mode.

```
1 cl_mem buf1 = clCreateBuffer(context, ...);
2 cl_mem buf2 = clCreateBuffer(context, ...);
3 clEnqueueWriteBuffer(cmd_queue, buf1, /*blocking_write*/ CL_TRUE, ...);
4 clSetKernelArg(kernel, 0, sizeof(cl_mem), &buf1);
5 clSetKernelArg(kernel, 1, sizeof(cl_mem), &buf2);
6 clEnqueueNDRangeKernel(cmd_queue, kernel, ...);
7 clEnqueueReadBuffer(cmd_queue, buf2, /*blocking_read*/ CL_TRUE, ...);
8 clReleaseMemObject(buf1);
```

Listing 4.3: The program fragment shown in Listing 4.1 after fixing the use-after-free error.

Our strategy for detecting use-after-free errors is to cause the main thread (i.e. the thread hosting the OpenCL host program) to continue running after an event is scheduled on the command queue until it explicitly yields. Because the `clFinish`, `clWaitForEvents`, `clEnqueueReadBuffer` and `clEnqueueWriteBuffer` functions in our model will all yield until their respective events have completed, the host program is given the opportunity to cause an error by freeing memory until it yields using one of these methods, and in this way we are able to detect the majority of use-after-free errors. However, due to deficiencies in our model, as outlined below, we are unable to detect all use-after-free errors.

As mentioned in Section 4.5.2, we only allow a command queue to hold one event at a time, and that event must be a kernel invocation event. If an asynchronous memory operation or a second kernel invocation is enqueued while a kernel invocation event is already in the queue, a false negative may result, as our model will empty the command queue in this case, invoking any kernels which may have otherwise caused a use-after-free error to be detected later. For example, consider the behaviour of the code in Listing 4.3 if line 7 were modified to make the read non-blocking. The program would then contain an error, as the implementation is free to execute the kernel thereby accessing the deallocated buffer `buf1` at any point after line 8, but this would not be detected by our model, because it would wait for the kernel invocation to finish on line 7.

Furthermore, this technique would not work as is for an OpenCL program which uses multiple command queues, as a yield resulting from a wait for a first command queue may result in work-items from a second command queue being run.

One way to solve both problems may be to defer all asynchronous command queue operations until they are waited on by the host program. A call to `clWaitForEvents`, `clFinish` or a blocking runtime function would then result in a minimal subset of commands being executed. In Section 7.2 we discuss one strategy for computing a minimal set of commands to execute.

4.6 Summary

This chapter has given a description of our models for floating point arithmetic, SIMD, atomic operations and OpenCL. Our models are designed to deal with these systems in the manner presented to the symbolic execution engine by the compiler, through the LLVM intermediate representation. This ensures that no modifications to the code under test are required to use the models. In all cases, extensions to the symbolic execution engine were required in order to model the systems correctly. In Chapter 5 we will explain how KLEE-CL uses these models to detect errors in programs which use them.

Chapter 5

Symbolic Error Detection

This chapter discusses how KLEE-CL builds on the modelling approach discussed in Chapter 4 to detect errors in a program that uses floating point arithmetic, SIMD and/or OpenCL. KLEE-CL first prepares the program under test to be efficiently executed symbolically by statically reducing the number of paths that need to be executed (Section 5.1), and then symbolically executes the program. The ability to crosscheck two implementations of a floating point algorithm was implemented by adding support for testing for the equivalence of floating point expressions (Section 5.2). This was accomplished in two ways: by transforming floating point expressions into a canonical form (Section 5.2.2) and by enhancing the constraint solver to transform floating point expressions into implied integer expressions (Section 5.2.3). While symbolically executing the program, KLEE-CL detects data races in OpenCL kernels (Section 5.3).

5.1 Static Path Merging

This section describes the static path merging pass that is applied before symbolic execution takes place. Static path merging can be used to dramatically lower the number of program paths that need to be executed by KLEE-CL, especially for programs containing conditionals within loops.

5.1.1 Background

One limitation of symbolic execution is that the number of paths in a program is exponential in the number of unresolvable branches encountered during execution. The worst case for data parallel programs is that an unresolvable branch is encountered within a tight loop iterating over the input elements, causing the number of paths to become an exponential factor of the input size. Since KLEE attempts to execute every path to completion, this behaviour can have a detrimental effect on efficiency, and may often prevent verification of the correctness of data parallel optimisations in a practical amount of time even for small input sizes.

To reduce the number of paths that need to be explored, we apply an aggressive variant of *phi-node folding* (also known as *if-conversion*) [LA04, CCF03], which attempts to statically merge program paths. Phi-node folding usually operates on the static single-assignment (SSA) form of a program [AWZ88] and targets branches with a control flow structure matching the diamond pattern shown in Figure 5.1, commonly associated with `if` statements and the C ternary operator. The beginning of block D contains one or more *phi* nodes, which select the correct register values (in our example, that of `%r`) depending on what block was previously executed.

Phi-node folding reduces the amount of forking in a program by merging all four basic blocks in a diamond pattern into a single block. This is accomplished by unconditionally executing blocks B and C and using the branch predicate p to select the result via `select` instructions. The `select` instruction has similar behaviour to the C ternary operator, but can be represented directly at the constraint solver level without need for forking.

The traditional application of phi-node folding in compilers has both *safety* and *performance* restrictions. Because blocks B and C are executed unconditionally, it is only safe to perform the transformation if neither block contains an instruction that may throw an exception or cause any other side effects. Most arithmetic instructions satisfy these constraints. However, floating point instructions do not, because they may throw an exception if either operand is a `NaN`. Furthermore, the transformation is only performed when folding is cheap enough, in order

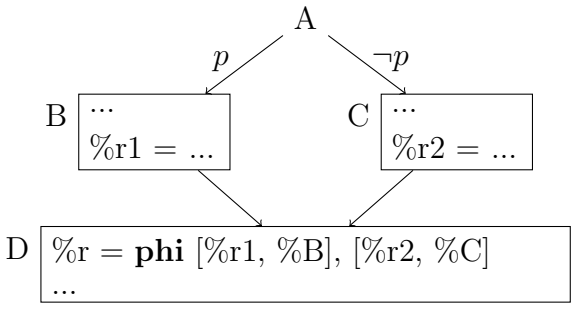

Figure 5.1: Diamond control flow pattern.

Instruction	Cost	Unsafe?
`Load`	1	
`GetElementPtr`	1	
`Add`	1	
`Sub`	1	
`And`	1	
`Or`	1	
`Xor`	1	
`Shl`	1	
`LShr`	1	
`AShr`	1	
`ICmp`	1	

Instruction	Cost	Unsafe?
`Trunc`	1	
`ZExt`	1	
`SExt`	1	
`Select`	2	
`FAdd`	1	✓
`FSub`	1	✓
`FMul`	1	✓
`FDiv`	1	✓
`FAdd`	1	✓
`FCmp`	1	✓

Table 5.1: Phi node folding instruction costs.

to minimise the amount of unnecessary work done by the CPU.

Due to forking, the cost of not applying the optimisation in a symbolic execution context is usually greater than that of applying it. Furthermore, since KLEE-CL does not model floating point exceptions, it is always safe to fold floating point instructions in KLEE-CL.

Thus, we have adapted phi-node folding to aggressively merge paths when we encounter the diamond pattern shown in Figure 5.1.

5.1.2 Implementation

Our implementation is built on top of LLVM's `SimplifyCFG` pass, which already contained an implementation of phi node folding. The existing pass was highly conservative in that it only applied if each of the basic blocks B and C contained at most one computation instruction.

We enhanced the pass in two ways. Firstly, we introduced the concept of a *phi node folding threshold*, a value such that the sum of the costs of LLVM instructions that are evaluated to compute the unused operand of the `select` instruction is never more than the given threshold (the costs we assigned to various LLVM instructions are shown in Table 5.1). Practically, this means that neither of the cost sums for each operand may exceed the threshold. The user may tune the phi node folding threshold using a hidden command line parameter. Secondly, we added an option to enable *unsafe phi node folding*, which allowed the optimisation to be applied to floating point instructions in spite of the side effects described above.

Note that in all cases, the phi node folding pass will only be applied to a set of basic blocks if all LLVM instructions contained within the conditional blocks have no side effects in the context of KLEE-CL. All instructions listed in Table 5.1 not marked as unsafe, with the exception of the `load` instruction, are defined by the LLVM language reference [LLV] as having no side effects, and to the best of our knowledge KLEE-CL faithfully models these instructions. The `load` instruction may have the effect of aborting the program (in the case of an invalid pointer) and as such the phi node folding pass will only consider a load to have no side effects if the pointer is known to be valid (for example, if it points directly to a global variable). KLEE-CL's exception free modelling of floating point instructions ensures that no side effects are observed for those instructions marked as unsafe.

In our experiments (Chapter 6) we set a very high cost threshold (1000) and enable unsafe phi node folding. Practically, this allows the optimisation to be applied in all possible circumstances.

Our modifications to `SimplifyCFG` were subsequently contributed back to LLVM. These changes have turned out to have potential applications outside of symbolic execution; indeed, LLVM developers have considered adjusting the default threshold to allow additional optimisations to be applied [pr1].

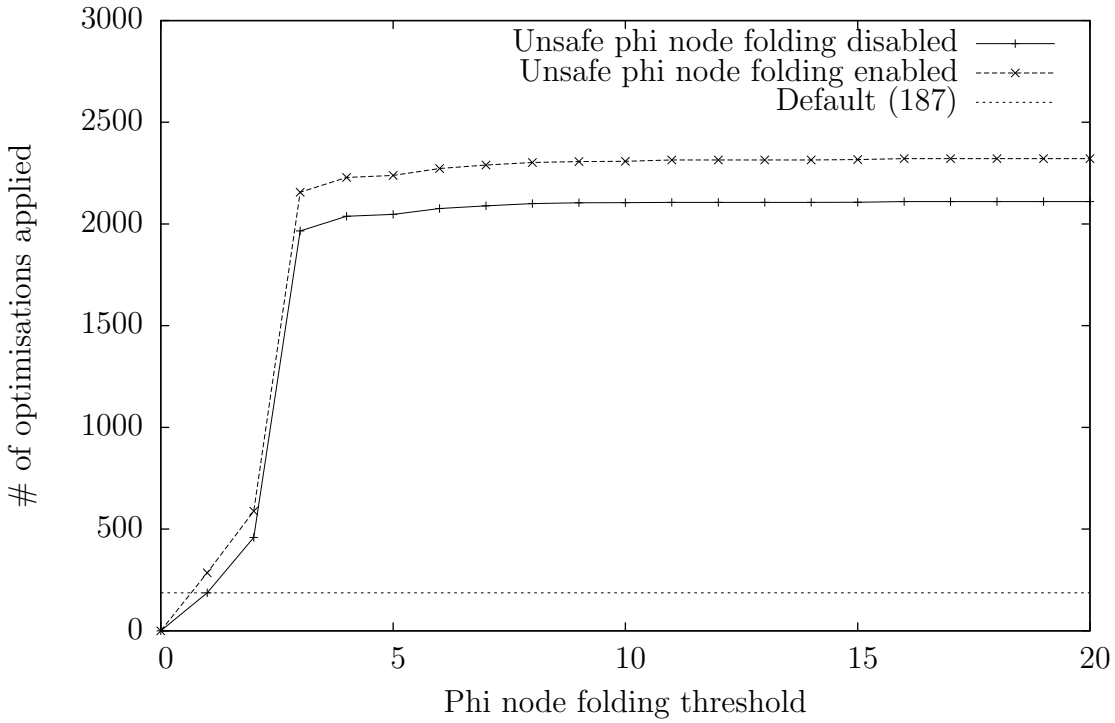

Figure 5.2: Number of phi node folding optimisations applied for thresholds between 0 and 20.

5.1.3 Evaluation

In order to measure the benefits of static path merging both statically and dynamically, we evaluated our enhanced phi node folding pass in two ways. Firstly, we measured the number of times it was successfully applied to an LLVM bitcode file containing the entire OpenCV library for a range of thresholds between 0 and 1000. Secondly, we measured the number of execution paths for our OpenCV benchmarks (Section 6.1) both with and without phi node folding enabled.

Figure 5.2 shows the number of locations within the LLVM intermediate representation (IR) that phi node folding was successfully applied both with unsafe phi node folding disabled and enabled for thresholds between 0 and 20 (above 20, no change was observed), together with the number of times that the optimisation was applied at the LLVM default settings. As can be seen, a large improvement was observed for thresholds above 2, with a maximum improvement of over 10× over the LLVM defaults with unsafe phi node folding enabled. However, it must be remembered that the OpenCV library contains a substantial amount of code that is not covered

by our benchmarks, and that the diamond control flow pattern is very common in C and C++ code, being formed from `if` statements and `?:` operators, and that in the most common case the controlling expression would not be symbolic. This data is best interpreted together with dynamic measurements which reveal how much the optimisation helps in practice.

To measure the practical effect of phi node folding, we measured the number of paths explored for each of our OpenCV benchmarks for an input size of 4×4 (except for `transcf.43`, `transff.43` and `transsf.43`, which use fixed sizes of 3×4 and 4×4) both with and without phi node folding enabled. Three of our benchmarks benefited from phi node folding – `silhouette`, `transcf.43` and `transsf.43` – by an exponential factor of the number of elements in the input image. In all cases, we were able to merge all program branches into a single large `select` expression. For example, for the largest image we tested in the `silhouette` benchmarks, sized 16×16, the number of paths decreased from approximately 2^{256} paths (according to our theoretical calculations) to 1.

5.2 Equivalence Testing

On every path explored via symbolic execution, KLEE-CL tries to prove that the symbolic floating-point expressions associated with the scalar and the SIMD implementations are equivalent.

Proving that two floating point expressions are equivalent involves two main steps. First, KLEE-CL applies a series of expression rewrite rules that aim at bringing each expression to a simple canonical form. These transformations include, among others, category analysis, identity reduction, folding of bitwise operations, and concat merging, and are discussed in detail in Section 5.2.2.

After these canonicalisation rules are applied, KLEE-CL determines if the two normalised expressions are equivalent by using a simple expression matching algorithm. Starting at the root of each expression, KLEE-CL recursively compares pairs of subtrees from the two expressions. For integer subtrees, the STP constraint solver is used to determine the equivalence

of the two subtrees. On the other hand, for floating point subtrees, the algorithm does not use the semantics of the floating point expressions themselves, which are instead treated as uninterpreted functions. While this may not work very well for integers, it is a good fit for floating point — unlike integer arithmetic, it is likely the case that constructing two equivalent values from the same inputs in floating point can usually only be done reliably by performing the same operations. Even if this were not true, our observation is that developers tend to imitate the data flow of existing serial code when writing data parallel code, and a benefit is therefore derived from developing tools, such as KLEE-CL, which operate on the assumption that it is true, and therefore match the expectations of those developers.

If the matching algorithm fails to prove expression equivalence, we try to substitute rewritten constraints that are implied by the original constraints (i.e., they impose fewer constraints on the input). This has the important property that no false negatives are produced, i.e., that there are no undetected errors. Any input that invalidates the original equivalence will also invalidate the less constrained rewritten one. Our technique for building implied constraints is discussed further in Section 5.2.3.

5.2.1 Assumptions

In floating point arithmetic, it is unsound to perform certain expression simplifications that are sound under ordinary real number arithmetic. For example, it is invalid to simplify $x + 0$ to x in floating point because if x is negative zero, the result is positive zero. However, developers are often not interested in such edge cases, and therefore we added the option to allow the expression simplifier to make certain normally-unsound *assumptions* about the floating point model. Many of these assumptions were primarily motivated by the different computational structure inherent to parallel programming.

For example, a reduction operation in a serial program is typically computed using a $O(n)$ `for` loop which maintains an accumulated result based on the data elements processed thus far, an example of which is shown in Figure 5.1. This implementation technique is inefficient for a data parallel program, as it does not permit exploitation of parallel resources. By exploiting

```
result = 0;
for (unsigned i = 0; i < size; ++i)
  result += temp[i];
```

Listing 5.1: Serial reduction.

```
__kernel void reduce(__global float *out,
                     __global float *in,
                     __local float *temp) {
  size_t tid  = get_local_id(0);
  size_t size = get_local_size(0);
  size_t d    = 1;

  temp[tid] = in[get_global_id(0)];

  for (; d<size; d<<=1) {
    barrier(CLK_LOCAL_MEM_FENCE);
    if (tid % (d*2) == 0)
      temp[tid] += temp[tid+d];
  }

  out[get_group_id(0)] = temp[0];
}
```

Listing 5.2: OpenCL parallel reduction.

parallelism we can implement a reduction algorithm that operates in $O(\log n)$ time. An example of a parallel algorithm in OpenCL is shown in Figure 5.2, and its data flow is shown in Figure 5.3.

When executing the codes symbolically using KLEE-CL, we may obtain expressions of the form:

$$\textbf{Serial}\quad (((((((0 + t_0) + t_1) + t_2) + t_3) + t_4) + t_5) + t_6) + t_7$$

$$\textbf{Parallel}\quad ((t_0 + t_1) + (t_2 + t_3)) + ((t_4 + t_5) + (t_6 + t_7))$$

Suppose that we now wish to show equivalence between these two expressions. While it is unsound in general to treat them equivalently ($0 + t_0$ cannot be simplified to t_0, and the $+$ operator is not associative), the developer may decide that these differences are acceptable and enable assumptions that allow KLEE-CL to simplify one expression into the form of the other.

We implemented a total of four assumptions, each of which may be enabled via individual command line arguments:

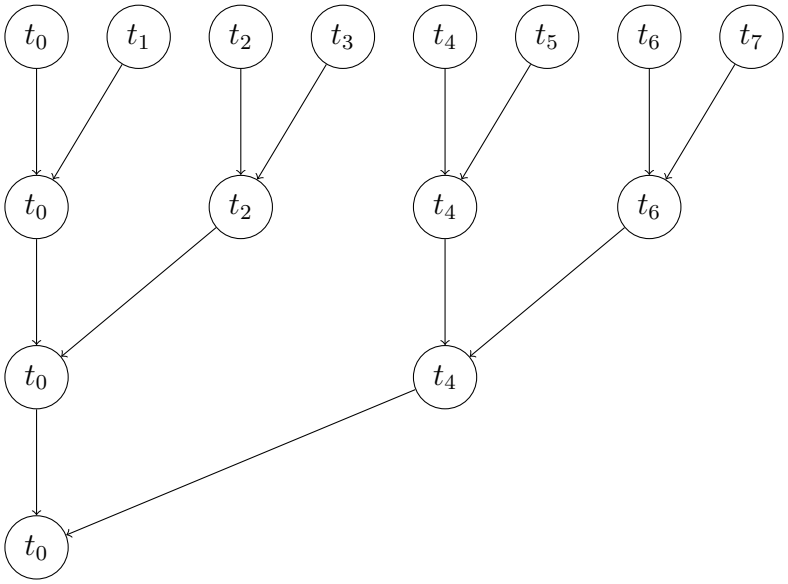

Figure 5.3: OpenCL parallel reduction data flow.

- The *positive zero* assumption allows the simplifier to assume that no operand of a floating point operation is negative zero.

- The *finite* assumption allows the simplifier to assume all operands of floating point operations are finite (i.e. not $\pm\infty$ or `NaN`).

- The *ordered* assumption allows the simplifier to assume all operands of floating point operations are ordered (i.e. not `NaN`).

- The *associativity* assumption allows the simplifier to assume that all operands of floating point operations are such as to render those operations associative.

Each of these assumptions is implemented through additional expression simplification rules which are enabled together with the assumption. Further details of each simplification is found in Section 5.2.2.

While we have not checked these assumptions for contradictory rewrites, we consider that it is up to the developer to use assumptions only when appropriate, as each assumption can result in arbitrary deltas.

5.2.2 Expression Transformations

The expression canonicalisation rules presented in this section are essential to the success of our expression matching approach. Their main goal is to bring expressions to a simplified normal form, in which they are easier to compare.

Table 5.2 lists the main rewrite rules we implemented. The first ten are specifically targeted to floating point expressions, while the other eight are applicable to both floating point and integer ones. The remainder of this section discusses these rules in more detail.

1. *Floating point relational operators*

   As explained in Section 4.1, each floating point relational operator has an associated outcome set. Rules 1–3 apply simplifications to boolean `And`, `Or` and `Not` operators by manipulating the outcome set. For example, $\texttt{Or}(\texttt{FOlt}(X, Y), \texttt{FOeq}(X, Y))$ simplifies to $\texttt{FOle}(X, Y)$.

   Rules 4–6 implement similar simplifications, making use of the `swap` function defined below:

$$\begin{aligned} \text{If } o \cap \{<, >\} = \{>\}, &\quad \texttt{swap}(o) = (o \setminus \{>\}) \cup \{<\} \\ \text{If } o \cap \{<, >\} = \{<\}, &\quad \texttt{swap}(o) = (o \setminus \{<\}) \cup \{>\} \\ \text{Otherwise} &\quad \texttt{swap}(o) = o \end{aligned}$$

2. *Category analysis*

   Category analysis, a simplified form of interval analysis [MY59], affords us a crude means of expression optimisation using a simple abstract interpretation of the semantics of certain floating point expressions. We establish a category set $\mathbf{C} = \{\texttt{NaN}, -\infty, -, 0, +, +\infty\}$ which covers all categories of floating point values (`NaN` values, negative infinity, negative values except negative zero/infinity, positive or negative zero, positive values except positive zero/infinity, and positive infinity). The category set $\text{cat}(x) \subseteq \mathbf{C}$ of an expression

#	Condition/Assumption	Expression	Result	Section
1	-	$\texttt{And}(\texttt{FCmp}(X,Y,O_1),\texttt{FCmp}(X,Y,O_2))$	$\texttt{FCmp}(X,Y,O_1 \cap O_2)$	§5.2.2(1)
2	-	$\texttt{Or}(\texttt{FCmp}(X,Y,O_1),\texttt{FCmp}(X,Y,O_2))$	$\texttt{FCmp}(X,Y,O_1 \cup O_2)$	
3	-	$\texttt{Eq}(\texttt{FCmp}(X,Y,O),\texttt{false})$	$\texttt{FCmp}(X,Y,\mathbf{O} \setminus O)$	
4	$O \cap \{<,>\} = \{>\}$	$\texttt{FCmp}(X,Y,O)$	$\texttt{FCmp}(Y,X,\texttt{swap}(O))$	
5	-	$\texttt{And}(\texttt{FCmp}(X,Y,O_1),\texttt{FCmp}(Y,X,O_2))$	$\texttt{FCmp}(X,Y,O_1 \cap \texttt{swap}(O_2))$	
6	-	$\texttt{Or}(\texttt{FCmp}(X,Y,O_1),\texttt{FCmp}(Y,X,O_2))$	$\texttt{FCmp}(X,Y,O_1 \cup \texttt{swap}(O_2))$	
7	-	Category analysis		§5.2.2(2)
8	C constant, see §5.2.2(3)	$\texttt{FOeq}(\texttt{SIToFP}(X),C)$	$\texttt{Eq}(X,\texttt{FPToSI}(C))$	§5.2.2(3)
9	C constant, see §5.2.2(3)	$\texttt{FOeq}(\texttt{UIToFP}(X),C)$	$\texttt{Eq}(X,\texttt{FPToUI}(C))$	
10	$f \in \{\texttt{FPToSI},\texttt{FPToUI}\}$	$f(\texttt{FPExt}(X))$	$f(X)$	§5.2.2(4)
11	C_1, C_2 constants	$\texttt{Concat}(C_1,\texttt{Concat}(C_2,X))$	$\texttt{Concat}(\texttt{Concat}(C_1,C_2),X)$	§5.2.2(5)
12	-	Partial constant folding with equality		§5.2.2(6)
13	-	$\texttt{ZExt}(X)$	$\texttt{Concat}(0,X)$	§5.2.2(7)
14	-	$\texttt{And}(\texttt{SExt}(P^1),X)$	$\texttt{Select}(P^1,X,0)$	
15	C constant	$\texttt{Shl}^W(X,C)$	$\texttt{Concat}(\texttt{Extract}^{W-C}(X,C),0^C)$	
16	C constant	$\texttt{LShr}^W(X,C)$	$\texttt{Concat}(0^C,\texttt{Extract}^{W-C}(X,0))$	
17	$f \in \{\texttt{Or},\texttt{And},\texttt{Xor}\}$, $\text{width}(X_0) = \text{width}(X_1)$	$f(\texttt{Concat}(X_0,Y_0),\texttt{Concat}(X_1,Y_1))$	$\texttt{Concat}(f(X_0,X_1),f(Y_0,Y_1))$	§5.2.2(8)
18	$f \in \{\texttt{Or},\texttt{And},\texttt{Xor}\}$	$\texttt{Extract}^W(f(X,Y),N)$	$f(\texttt{Extract}^W(X,N),\texttt{Extract}^W(Y,N))$	
19	-	$\texttt{FMul}(X,1)$	X	§5.2.2(9)
20	-	$\texttt{FMul}(1,X)$	X	
21	*positive zero*	$\texttt{FAdd}(X,0)$	X	
22	*positive zero*	$\texttt{FAdd}(0,X)$	X	
23	*finite*, *positive zero*	$\texttt{FMul}(X,0)$	0	
24	*finite*, *positive zero*	$\texttt{FMul}(0,X)$	0	
25	*associativity*	$\texttt{FAdd}(X,\texttt{FAdd}(Y,Z))$	$\texttt{FAdd}(\texttt{FAdd}(X,Y),Z)$	§5.2.2(10)
26	*associativity*	$\texttt{FMul}(X,\texttt{FMul}(Y,Z))$	$\texttt{FMul}(\texttt{FMul}(X,Y),Z)$	

Table 5.2: Symbolic expression canonicalisation rules. Where necessary, bitwidths of expressions are denoted by superscripts.

x is defined as the set of categories the expression x may be in. We define $\text{cat}(x)$ recursively based on the category sets of subexpressions of x. For example, if $+ \in \text{cat}(x)$ and $+ \in \text{cat}(y)$ then $\{+, +\infty\} \subseteq \text{cat}(x + y)$. Our system is capable of computing an accurate category set for most floating point expressions.

Category sets are used to simplify and normalise floating point relational operations. For example, if $\text{cat}(x) = \{0, -\}$ and $\text{cat}(y) = \{0, +\}$ then both $x > y$ and x `UNO` y are infeasible. Therefore $x > y$ is simplified to `false`, $x \leq y$ to `true` and $\neg(x < y)$ (unordered $\geq$) is normalised to $x = y$.

3. *Floating point equality comparison*

SSE code sometimes performs integer comparisons by first converting to floating point format. This may be due to combining floating point and integer comparisons in a single expression. An example of this is found in the OpenCV routine `cvUpdateMotionHistory` in the `silhouette` benchmark, which converts an integer vector to a floating point vector s0, compares the elements to 0 and performs a logical `AND` with another operation:

```
__m128 s0 = _mm_cvtepi32_ps(...);
__m128 fz = _mm_setzero_ps();
__m128 m0 = _mm_and_ps(_mm_xor_ps(v0, ts4),
                       _mm_cmpneq_ps(s0, fz));
```

The corresponding scalar code performs a straightforward integer comparison of the values here loaded to s0.

Rewrite rules 8 and 9 support such cases by providing a normalisation of floating point comparisons to integer comparisons. It is not sound to perform this normalisation unless two conditions are met. First, C must be representable in X's type. This means that C must not have a fractional component and must satisfy $-2^{W-1} \leq C < 2^{W-1}$ (for signed conversion) or $0 \leq C < 2^W$ (for unsigned conversion) where $W = \text{width}(X)$. If C does not meet these requirements, the comparison will always yield `false`.

Second, X must not be subject to rounding if it is to match C. If X could be rounded, then the comparison would match multiple values of X. For example, using the IEEE

single precision format, with a 23-bit mantissa, the values 2^{24} and $2^{24}+2$ have adjacent representations. If X were $2^{24}+1$ it would be rounded to $2^{24}+2$ during integer to floating-point conversion and would match a C of that value. We must therefore require that $|C| < 2^{M+1}$ where M is the mantissa bitwidth of C's type.

4. *Removing unnecessary FPExt operations*

   Transformation rule 10 eliminates redundant floating-point extensions (e.g., from `float` to `double`) where the result is coerced to integer.

5. *Folding* `Concat` *sequences*

   Rule 11 performs constant folding on sequences of `Concat` operations. For example, $\texttt{Concat}(11, \texttt{Concat}(00, X))$ gets simplified to $\texttt{Concat}(1100, X)$.

6. *Partial constant folding with equality*

   Given an expression of the form $\texttt{Eq}(C, \texttt{Concat}(X, Y))$ where C is a constant, if either X or Y is constant then we compare the higher order bits of C to X (or the lower order bits to Y). If the bits are not equal, we can safely replace the entire expression with `false`. If the bits are equal, we replace the expression with an equality comparison of either the lower order bits of C with Y (if X constant) or the higher order bits of C with X (if Y constant).

7. *Simple normalisation rules*

   Rules 13–16 implement simple expression transformations via which certain bit-level operations are rewritten using `Concat`, `Extract` and `Select`. For example, a shift left on W bits by a constant amount C can be rewritten as an extract of length $W - C$ from offset C concatenated with C zero bits.

8. *Folding and unfolding of bitwise operations*

   Rewrite rule 17 implements folding of bitwise operations through `Concat` to take advantage of partial constant folding. For example, if $f = \texttt{And}$ and $X_0 = 0$ then X_1 can be completely eliminated since $\texttt{And}(0, X_1)$ reduces to 0.

Note that this rewrite rule can also be applied if any of the operands to the bitwise operation is a constant expression, by treating the constant as a `Concat` of two smaller constants.

Rewrite rule 18 implements a similar transformation that unfolds the `Extract` of a bitwise operation to take advantage of partial constant folding. For example, if $W = 2$, $N = 0$, $f =$ `Or` and $Y = 1100$, then the rule will simplify the entire expression to bitvector 00.

9. *Arithmetic equivalences for floating point*

   These rules implement a set of straightforward arithmetic equivalences. Rules 19 and 20 always hold under floating point arithmetic, regardless of the value of x. It is therefore safe to always apply this rule.

   Rules 21 and 22 do not hold universally under floating point arithmetic. In the case where x is a negative zero, the expression $x + 0$ evaluates to positive zero. These two values are distinct at the bit level, which prevents us from applying the simplification in the general case. We therefore only enable this rule if the positive zero assumption is enabled.

   Rules 23 and 24 do not hold universally either. If x is negative, $x \times 0$ evaluates to negative zero. If x is infinite or `NaN`, $x \times 0$ evaluates to `NaN`. Therefore, this rule is only applied if the positive zero and finite assumptions are enabled.

10. *Floating point associativity*

    These rules implement associativity for the floating point $+$ and $\times$ operators by re-arranging right associative operations into left associative ones. This is almost always an invalid transformation; therefore, the associativity assumption is required.

5.2.3 Building Implied Constraints

Implied constraints act as input to STP. They must therefore be free of floating point subexpressions. The goal of the implied constraint builder is therefore to build a new predicate which is not only implied by the given predicate but also built solely from integer operations.

Normally, a floating point relation (aside from ordered $\neq$ or bitwise $\neq$) will be simplified to `true`, which is implied by any predicate. For ordered $\neq$ and bitwise $\neq$, however, we can produce an implied constraint by pattern matching the two operands.

In our case, an ordered or bitwise $\neq$ constraint arises on the false branch of an `assert` statement that checks the equality of two results. If the false branch is deemed infeasible then we have successfully proven the equality of the two operands. If the pattern match succeeds, the operands are known to be equal at the bit level, and the operation will always yield `false`, so we can safely substitute a `false` constraint (it would be unsound to pattern match unordered $\neq$, such as the C `!=` operator, because `NaN != NaN` is `true`).

Otherwise, the operation *may* return `true`, so we normally substitute `true`, which as mentioned is implied by any predicate, but in the case where the two floating point expression trees contain purely integer subexpressions, and are identical except for these subexpressions, we substitute an implied constraint of the form $a_1 \neq b_1 \vee a_2 \neq b_2 \vee \ldots$, where a_i are integer subexpressions that appear in the left-hand operand of the $\neq$ constraint, and b_i are subexpressions that appear in the same place in the right-hand operand of the $\neq$ constraint.

Note that, since we do not attempt to pattern match unordered $\neq$, `assert` statements in benchmarks must use bit-casting to integers (and a bitwise integer comparison) or ordered $\neq$ (provided by `islessgreater` in C99) to compare floating-point values.

The implied constraint builder's pattern matcher can also recognise the expression idiom representing the floating-point `min` and `max` operations:

$$\texttt{min}(X, Y) = \texttt{Select}(\texttt{FOlt}(X, Y), X, Y)$$
$$\texttt{max}(X, Y) = \texttt{Select}(\texttt{FOlt}(Y, X), X, Y)$$

and attempt to match the operands of a chain of `min` and `max` operations where it is safe to do so. Because floating-point `min` and `max` are not commutative, and are in general not associative, it is usually unsafe to do this. The root cause for `min` and `max` not being commutative or associative is that `FOlt`, the ordered floating point $<$ operator, is not a total order in the presence of `NaN`s.

To see why the operations are not commutative, consider the evaluation of $\texttt{min}(X, Y)$ where one of the operands is `NaN` and the other is not `NaN`. In this case, the condition would always evaluate to `false` and Y is always returned regardless of which operand is `NaN`. A similar result can be drawn for `max`.

To see why the operations are not associative, consider the expressions $\texttt{min}(\texttt{min}(X, \texttt{NaN}), Y)$ and $\texttt{min}(X, \texttt{min}(\texttt{NaN}, Y))$. As we have seen $\texttt{min}(X, \texttt{NaN})$ evaluates to `NaN` and $\texttt{min}(\texttt{NaN}, Y)$ to Y so the expressions reduce to Y and $\texttt{min}(X, Y)$ respectively.

There are two cases in which it is safe to match operands. One possibility is that if the ordered assumption is enabled, we are allowed to assume that the operands to the `FOlt` operation are ordered, and that therefore `FOlt` is a total order. The other possibility is if the `min`/`max` chain is of the form:

$$\texttt{min}(X, \texttt{min}(Y, \texttt{min}(Z, +\infty)))$$
$$\text{or } \texttt{max}(X, \texttt{max}(Y, \texttt{max}(Z, -\infty)))$$

If any of the operands X, Y or Z are `NaN`, that operand will effectively be excluded from the computation, because the second operand (the remainder of the `min`/`max` chain) will be selected.

Each constraint C in the constraint set is replaced with $rw(C)$, where rw is defined in Figure 5.4 together with two helper functions, rw' and ce. rw' takes an argument representing the sense (positive or negative) of the current expression, such that for any KLEE-CL expression E, $E \to rw'(E, \bot)$ and $rw'(E, \top) \to E$. ce builds an expression such that for any pair of KLEE-CL expressions X, Y, $ce(X, Y) \to X = Y$. rw' and ce are evaluated in a top-down pattern matching fashion, whereby the first rule whose pattern matches and whose conditions are satisfied is used, regardless of whether any other rule matches.

rw' and ce use the following functions:

- $hasFP(x)$, which is true iff x contains any floating point subexpressions other than expressions of the form $\texttt{FPToSI}(X)$ and $\texttt{FPToUI}(X)$ (conversion from floating point to integer)

and subexpressions thereof, which are handled separately;

- $minOps(x)$ and $maxOps(x)$ which, if the given expression is an idiomatic `min` (resp. `max`) operation whose operands are safe to match according to the rules given above and the current set of enabled assumptions, returns the operand set, else returns $\{x\}$.

After applying these rewrite rules, each expression of the form $\texttt{FPToSI}(X)$ and $\texttt{FPToUI}(X)$ is substituted by an unconstrained symbolic integer variable. While a new variable is created for each unique expression of this type, identical expressions are substituted with references to the same variable.

We can now use our constraint solver STP to determine if the rewritten integer constraints are satisfiable. If not, then we know that the original constraints are also unsatisfiable. If we were on the false branch of an equivalence checking `assert` statement, we have shown the two expressions to be equivalent. However, if the constraint solver finds our rewritten constraints to be satisfiable, the mismatch could be a false positive.

We show below an example of a constraint that may be encountered during the evaluation of an `assert` statement of the form:

```
assert(bitwise_eq(x + y, y + x));
```

where x and y are unconstrained symbolic expressions, and `bitwise_eq` is a function that tests for bitwise equality over floats. The false branch constraint will be of the form:

$$\texttt{Eq}(\texttt{Eq}(\texttt{FAdd}(X, Y), \texttt{FAdd}(Y, X)), \texttt{false})$$

The application of the rw function to the expression is shown below (each function application or expression simplification step is shown on a separate line):

1. $rw(\texttt{Eq}(\texttt{Eq}(\texttt{FAdd}(X, Y), \texttt{FAdd}(Y, X)), \texttt{false}))$
2. $rw'(\texttt{Eq}(\texttt{Eq}(\texttt{FAdd}(X, Y), \texttt{FAdd}(Y, X)), \texttt{false}), \bot)$ (rw application)
3. $\texttt{Eq}(rw'(\texttt{Eq}(\texttt{FAdd}(X, Y), \texttt{FAdd}(Y, X)), \top), \texttt{false})$ (rw application)
4. $\texttt{Eq}(ce(\texttt{FAdd}(X, Y), \texttt{FAdd}(Y, X)), \texttt{false})$ (rw' application)

5. $\texttt{Eq}(\texttt{Or}(\texttt{And}(ce(X,Y), ce(Y,X)), \texttt{And}(ce(X,X), ce(Y,Y))), \texttt{false})$ (ce application)
6. $\texttt{Eq}(\texttt{Or}(\texttt{And}(\texttt{false}, \texttt{false}), \texttt{And}(\texttt{true}, \texttt{true})), \texttt{false})$ (ce application)
7. $\texttt{Eq}(\texttt{Or}(\texttt{false}, \texttt{true}), \texttt{false})$ (And constant folding)
8. $\texttt{Eq}(\texttt{true}, \texttt{false})$ (Or constant folding)
9. $\texttt{false}$ (Eq constant folding)

A more complex constraint involving both floating point and integer subexpressions is shown below:

$$\texttt{Eq}(\texttt{Eq}(\texttt{FSqrt}(\texttt{SIToFP}(\texttt{Mul}(X, 2))), \texttt{FSqrt}(\texttt{SIToFP}(\texttt{Shl}(X, 1)))), \texttt{false})$$

The rw function application:

1. $rw(\texttt{Eq}(\texttt{Eq}(\texttt{FSqrt}(\texttt{SIToFP}(\texttt{Mul}(X, 2))), \texttt{FSqrt}(\texttt{SIToFP}(\texttt{Shl}(X, 1)))), \texttt{false}))$
2. $rw'(\texttt{Eq}(\texttt{Eq}(\texttt{FSqrt}(\texttt{SIToFP}(\texttt{Mul}(X, 2))), \texttt{FSqrt}(\texttt{SIToFP}(\texttt{Shl}(X, 1)))), \texttt{false}), \bot)$ (rw)
3. $\texttt{Eq}(rw'(\texttt{Eq}(\texttt{FSqrt}(\texttt{SIToFP}(\texttt{Mul}(X, 2))), \texttt{FSqrt}(\texttt{SIToFP}(\texttt{Shl}(X, 1)))), \top), \texttt{false})$ (rw')
4. $\texttt{Eq}(ce(\texttt{FSqrt}(\texttt{SIToFP}(\texttt{Mul}(X, 2))), \texttt{FSqrt}(\texttt{SIToFP}(\texttt{Shl}(X, 1)))), \texttt{false})$ (rw')
5. $\texttt{Eq}(ce(\texttt{SIToFP}(\texttt{Mul}(X, 2)), \texttt{SIToFP}(\texttt{Shl}(X, 1))), \texttt{false})$ (ce)
6. $\texttt{Eq}(ce(\texttt{Mul}(X, 2), \texttt{Shl}(X, 1)), \texttt{false})$ (ce)
7. $\texttt{Eq}(\texttt{Eq}(\texttt{Mul}(X, 2), \texttt{Shl}(X, 1)), \texttt{false})$ (ce)

5.2.4 Bit Blasting Floating Point Operations

As we shall see in Chapter 6, we have found the floating point implied constraint building technique set out in this thesis to be an effective technique for detecting mismatches between two implementations of floating point algorithms. However, it has two disadvantages: it can produce false positives (although we found the number of false positives to be relatively low), and it fails to produce counterexamples for mismatches. Both of these disadvantages can be overcome using a constraint solver with support for exact reasoning over floating point arithmetic, such as a floating point bit blaster, although this can be inefficient, especially for larger problems.

$$
\begin{array}{rcl}
rw(E) & = & rw'(E, \bot) \\
\\
rw'(\texttt{And}(X^1, Y^1), n) & = & \texttt{And}(rw'(X, n), rw'(Y, n))) \\
rw'(\texttt{Or}(X^1, Y^1), n) & = & \texttt{Or}(rw'(X, n), rw'(Y, n))) \\
rw'(\texttt{Eq}(X, \texttt{false}), n) & = & \texttt{Eq}(rw'(X, \neg n), \texttt{false}) \\
rw'(\texttt{Eq}(X, Y), n) & = & ce(X, Y) \text{ if } n \\
rw'(\texttt{FUeq}(X, Y), n) & = & ce(X, Y) \text{ if } n \\
rw'(\texttt{FOne}(X, Y), n) & = & \texttt{Eq}(ce(X, Y), \texttt{false}) \text{ if } \neg n \\
rw'(E, n) & = & \begin{cases} \texttt{true} & \text{if } hasFP(E) \wedge \neg n \\ \texttt{false} & \text{if } hasFP(E) \wedge n \\ E & \text{otherwise} \end{cases} \\
\\
ce(E, E) & = & \texttt{true} \\
ce(E_0, E_1) & = & \texttt{Eq}(E_0, E_1) \text{ if } \neg hasFP(E_0) \wedge \neg hasFP(E_1) \\
ce(\texttt{FAdd}(X_0, Y_0), \texttt{FAdd}(X_1, Y_1)) & = & \texttt{Or}(\texttt{And}(ce(X_0, X_1), ce(Y_0, Y_1)), \texttt{And}(ce(X_0, Y_1), ce(Y_0, X_1))) \\
ce(\texttt{FMul}(X_0, Y_0), \texttt{FMul}(X_1, Y_1)) & = & \texttt{Or}(\texttt{And}(ce(X_0, X_1), ce(Y_0, Y_1)), \texttt{And}(ce(X_0, Y_1), ce(Y_0, X_1))) \\
ce(f(X_0, Y_0), f(X_1, Y_1)) & = & \texttt{And}(ce(X_0, X_1), ce(Y_0, Y_1)) \text{ if } f \in \{\texttt{FSub}, \texttt{FDiv}, \texttt{FRem}\} \\
ce(\texttt{FCmp}(X_0, Y_0, O), \texttt{FCmp}(X_1, Y_1, O)) & = & \begin{cases} \texttt{Or}(\texttt{And}(ce(X_0, X_1), ce(Y_0, Y_1)), & \text{if } O \cap \{<, >\} = \emptyset \vee O \cap \{<, >\} = \{<, >\} \\ \quad \texttt{And}(ce(X_0, Y_1), ce(Y_0, X_1))) & \\ \texttt{And}(ce(X_0, X_1), ce(Y_0, Y_1)) & \text{otherwise} \end{cases} \\
ce(f(X_0), f(X_1)) & = & ce(X_0, X_1) \text{ if } f \in \{\texttt{FSqrt}, \texttt{FCos}, \texttt{FSin}\} \\
ce(f(X_0^W), f(X_1^W)) & = & ce(X_0, X_1) \text{ if } f \in \{\texttt{FPExt}, \texttt{FPTrunc}\} \\
ce(\texttt{UIToFP}(X_0^{W_0}), \texttt{UIToFP}(X_1^{W_1})) & = & \begin{cases} ce(\texttt{ZExt}^{W_1}(X_0), X_1) & \text{if } W_0 < W_1 \\ ce(X_0, \texttt{ZExt}^{W_0}(X_1)) & \text{if } W_0 > W_1 \\ ce(X_0, X_1) & \text{otherwise} \end{cases} \\
ce(\texttt{SIToFP}(X_0^{W_0}), \texttt{SIToFP}(X_1^{W_1})) & = & \begin{cases} ce(\texttt{SExt}^{W_1}(X_0), X_1) & \text{if } W_0 < W_1 \\ ce(X_0, \texttt{SExt}^{W_0}(X_1)) & \text{if } W_0 > W_1 \\ ce(X_0, X_1) & \text{otherwise} \end{cases} \\
ce(S_0@\texttt{Select}(P_0, X_0, Y_0), S_1@\texttt{Select}(P_1, X_1, Y_1)) & = & \begin{cases} \texttt{true} & \text{if } P_0 = \texttt{Eq}(P_1, \texttt{false}) \wedge X_0 = Y_1 \wedge X_1 = Y_0 \\ \texttt{true} & \text{if } minOps(S_0) = minOps(S_1) \\ \texttt{true} & \text{if } maxOps(S_0) = maxOps(S_1) \end{cases} \\
ce(E_0, E_1) & = & \texttt{false}
\end{array}
$$

Figure 5.4: The rw rewriting function, and its helper functions rw' and ce.

We implemented floating point bit blasting support in KLEE-CL as an alternative to the implied constraint builder for small problems, by porting `floatbv`, CBMC's floating point bit blaster ([K+]; see Section 2.6) to KLEE-CL. This was done by replacing CBMC's bitvector expression builder with one that builds KLEE-CL bitvector expressions, and integrating the module into KLEE-CL by causing it to be invoked in the STP expression builder wherever a floating point expression was encountered. The user may switch between the bit blaster and the implied constraint builder using a command line option.

Bit blasting is not necessarily compatible with floating point assumptions (Section 5.2.1), and as such we did not attempt to handle assumptions when bit blasting. Even if we were able to represent the assumptions at the constraint solver level, any counterexamples produced by the constraint solver under floating point assumptions will not necessarily hold under normal floating point arithmetic. Furthermore, a constraint introduced by an assumption may cause inconsistencies, and therefore false negatives, in a path or verification condition. For example, the ordered assumption may cause the evaluation of the expression $x + y$ (where x and y are floating point numbers) to result in a constraint of the form $\texttt{FOrd}(\texttt{FAdd}(x, y), \texttt{FAdd}(x, y))$ being added to the path condition. However, if the result must be `NaN`, this constraint would cause the entire path condition to become unsatisfiable.

Note that the symbolic expression canonicalisation rules we have defined for floating point arithmetic (Table 5.2; specifically rules 1–10 and 19–20) serve a dual role as simplification rules when a bit blaster is being used, as they reduce the complexity of the problem that is provided to the SAT or SMT solver after bit blasting.

5.3 Data Race Detection for OpenCL

Data race detection is used when executing OpenCL C kernels to detect conflicts between memory accesses carried out by different work-items. Our analysis is able to detect races involving both concrete and symbolic memory addresses. In this section we give a description of our analysis and illustrate it using a number of case studies.

5.3.1 Description

Our model implements race detection capable of detecting, on each path explored, read-write and write-write races across work-items. Note that as mentioned in Section 4.5.2, our analysis is targeted towards detecting races between work-items in the same NDRange, and not between multiple NDRanges running concurrently, as may occur when using multiple or out-of-order command queues. Neither is our analysis intended to detect data races between a work-item and the host program – for the purposes of our analysis, all memory accesses performed by the host program are ignored.

Data races within an NDRange are easy to create accidentally, as the result of a single statement which accesses shared memory. On the other hand, the conditions which can cause data races between NDRanges are rare (all kernels we benchmarked (Chapter 6) used a single in-order command queue), and data races between an NDRange and the host program are normally the result of the host reading data from an OpenCL memory buffer without waiting for a kernel to terminate; this behaviour is also associated with errors such as use-after-free which KLEE-CL is capable of detecting in certain cases (Section 4.5.3).

To detect data races, we keep for each byte in the generic and group-local address spaces a *memory access record (MAR)* of accesses to that byte by a work-item thread. Each item in the MAR consists of:

1. the thread identifier of the most recent work-item to access the byte without an intervening execution barrier (*thread-id*);
2. the work-group identifier of the most recent work-group to access the byte (*wg-id*);
3. four flags indicating whether the byte was:
   (a) written by one or more work-items (*write*),
   (b) read by one or more work-items (*read*),
   (c) read by multiple work-items without an intervening execution barrier (*many-read*), and

(d) read by multiple work-groups (*wg-many-read*).

The purpose of storing many-read and wg-many-read separately is to correctly model the behaviour of execution barriers – the analysis needs to be able to preserve the fact that a byte has been read by multiple work-groups across execution barriers, because execution barriers do not prevent inter-work-group accesses from racing.

The MAR for each byte is initialised such that each identifier is set to zero, and each flag is cleared. The work-item identifier zero is treated specially by our analysis, and is used to indicate that no work-item has accessed that byte since the previous execution barrier, or since the start of the program, if no execution barrier has been encountered yet. It is for this reason that no work-item may use zero as its identifier if it is to participate in the analysis (in KLEE-CL, the host program uses identifier zero, and as mentioned is ignored by our analysis).

The MAR may be stored concretely or symbolically. The concrete representation of the MAR is an array of structs, each holding the MAR for one byte in the array. The symbolic representation of the MAR is a set of 6 symbolic arrays, each as large as the underlying array, and each representing one of the MAR attributes. For efficiency we store the MARs concretely by default, but if a symbolically indexed memory access is performed, the array's MARs are converted to the symbolic representation.

Whenever a memory access occurs, the MAR is inspected for any race conditions, and then updated. A race condition can be a read-after-write, a write-after-write or a write-after-read performed by a work-item or work-group other than that identified by the corresponding entry in the MAR, or any write-after-read if either of the *many-read* or *wg-many-read* flags are set.

For our race detection technique to be sound, we must correctly handle both execution barriers and the end of kernel function execution. Specifically, we must ensure that intra-work-group memory accesses on either side of an execution barrier are not considered to race, but that inter-work-group accesses are considered to race. We must also ensure that all memory accesses performed by the present kernel invocation are not considered to race with memory accesses performed by future kernel invocations.

This is implemented by causing the `klee_thread_barrier` function, which we use to implement `barrier` and which is also called once the kernel function returns (see Section 4.5.1), to reset certain fields of the MAR before it returns.

When `klee_thread_barrier` is called from `barrier`, we *locally reset* the MAR by setting the work-item identifier to zero and clearing the many-read flag of each MAR whose work-group identifier matches the work-group performing the `barrier`.

The `barrier` function takes an argument in the form of a combination of flags, indicating which memory address spaces are to be fenced. Our model uses this argument to control which MARs are locally reset. If the `CLK_LOCAL_MEM_FENCE` flag is set, which requests a memory fence over local memory, the MARs for the group-local address space are reset. Similarly, if the `CLK_GLOBAL_MEM_FENCE` flag is set, which requests a memory fence over global memory, the MARs for the generic address space are reset. Note that as well as resetting the MARs for `__global`, as intended, this also resets the MARs for `__constant` and `__private`. Because `__constant` is read-only, and `__private` is local to a work-item, neither of these address spaces can be used to cause a data race, so there is no harm in also resetting them.

When `klee_thread_barrier` is called after the kernel function returns, we *globally reset* MARs for both the generic and group-local address spaces by setting all identifiers to zero and clearing all flags.

5.3.2 Race Condition Test and MAR Updates

The race condition test, together with the required MAR updates, are shown in Figure 5.5. If the MAR is being stored concretely, we perform the test and the MAR updates directly. If the MAR is being stored symbolically, the test is performed by querying the constraint solver as to whether the symbolic expression representing the race condition test is satisfiable, and the MAR updates are performed by appending an update to the symbolic arrays.

The (*thread-id*[*index*], *wg-id*[*index*]) pair for a given array index *index* will be in one of three states:

Read

$write[index] \wedge (wg\text{-}id[index] \neq wg\text{-}id \vee (thread\text{-}id[index] \neq 0 \wedge thread\text{-}id[index] \neq thread\text{-}id))$		
$many\text{-}read[index]$	$\leftarrow$	$many\text{-}read[index] \vee (read[index] \wedge thread\text{-}id[index] \neq 0 \wedge thread\text{-}id[index] \neq thread\text{-}id)$
$wg\text{-}many\text{-}read[index]$	$\leftarrow$	$wg\text{-}many\text{-}read[index] \vee (read[index] \wedge wg\text{-}id[index] \neq wg\text{-}id)$
$thread\text{-}id[index]$	$\leftarrow$	$thread\text{-}id$
$wg\text{-}id[index]$	$\leftarrow$	$wg\text{-}id$
$read[index]$	$\leftarrow$	$\top$

Write

$many\text{-}read[index] \vee wg\text{-}many\text{-}read[index] \vee ((read[index] \vee write[index]) \wedge (wg\text{-}id[index] \neq wg\text{-}id \vee (thread\text{-}id[index] \neq 0 \wedge thread\text{-}id[index] \neq thread\text{-}id))$		
$thread\text{-}id[index]$	$\leftarrow$	$thread\text{-}id$
$wg\text{-}id[index]$	$\leftarrow$	$wg\text{-}id$
$write[index]$	$\leftarrow$	$\top$

Figure 5.5: Race condition test and MAR updates.

1. $(0, 0)$, indicating that the memory location has yet to be accessed by any work-item or has been globally reset,

2. $(0, n)$, $n \neq 0$, indicating that the location has been accessed by a work-item in work-group n but has been subsequently locally reset by an execution barrier (i.e. we are only concerned with memory accesses in work-groups other than n) or

3. (m, n), $m \neq 0$, $n \neq 0$, indicating that the location has been accessed by work-item m in work-group n without an intervening reset (i.e. we are concerned with memory accesses in work-items other than m, including work-items in work-groups other than n).

In cases (2) and (3), $read[index]$ and/or $write[index]$ may be set, but in case (1), neither $read[index]$ nor $write[index]$ will be set.

The first conjunct of the race condition test for reads is $write[index]$. This excludes case (1), as required. The second conjunct is $wg\text{-}id[index] \neq wg\text{-}id \vee (thread\text{-}id[index] \neq 0 \wedge thread\text{-}id[index] \neq thread\text{-}id)$. For case (2), $wg\text{-}id[index] \neq wg\text{-}id$ will hold in the case where the work-group identifier differs from the stored work-group identifier, and $thread\text{-}id[index] \neq 0 \wedge thread\text{-}id[index] \neq thread\text{-}id$ does not hold because $thread\text{-}id[index] \neq 0$ does not hold by definition. So the entire race condition test holds for (2) only if a previous write occurred and

the work-group identifiers differ. For case (3), *thread-id*[*index*] $\neq$ *thread-id* will hold in the case where the work-item identifier differs from the stored work-item, and *thread-id*[*index*] $\neq 0$ always holds by definition. If the work-group identifiers differ then the work-item identifiers will also differ, so *wg-id*[*index*] $\neq$ *wg-id* does not affect the satisfiability of its disjunction. So the entire race condition test holds for (3) only if a previous write occurred and the work-item identifiers differ.

Upon a memory read, in the case where all memory reads for a particular memory location are performed by the same work-group during the execution of a kernel, the *many-read*[*index*] flag is set iff the memory location has been read by multiple work-items without an intervening execution barrier. This is true in case (3) when the work-item identifier differs from the stored work-item identifier, hence the conjunct *thread-id*[*index*] $\neq$ *thread-id*. However, it is not true in case (2) because of the intervening execution barrier, nor is it true in case (1), hence the conjunct *thread-id*[*index*] $\neq 0$. *many-read*[*index*] remains set until the execution barrier for that work-group, hence the disjunct *many-read*[*index*]. The value of *many-read*[*index*] is indeterminate if multiple work-groups have accessed the memory location – in such a case, the value of *many-read*[*index*] at any program point depends on the scheduling of the work-group relative to other work-groups because it uses *thread-id*[*index*], which may be set and cleared by other work-groups independently of the current work-group. However, this does not affect the results of our analysis, as we shall see later.

Upon a memory read, the *wg-many-read*[*index*] flag is set iff the memory location has been read by multiple work-groups, and remains set until the kernel terminates execution. The analysis is similar to *many-read*, except that *wg-many-read* is not affected by execution barriers, and thus cases (2) and (3) are treated identically, hence the conjunct *wg-id*[*index*] $\neq$ *wg-id*.

The race condition test for writes uses three disjuncts. The first two, *many-read*[*index*] and *wg-many-read*[*index*], are used to test whether a data race has been caused by multiple preceding memory reads, either (in the case where all reads are performed by the same work-group) multiple reads within the same work-group (*many-read*[*index*]) or (in the case where reads are performed by multiple work-groups) multiple reads by multiple work-groups

($wg\text{-}many\text{-}read[index]$). Recall that $many\text{-}read[index]$ is indeterminate in the latter case – because $wg\text{-}many\text{-}read[index]$ will also be set in this case, the satisfiability of its disjunction is not affected. The final disjunct is used to detect conflicts in the case where only one work-item has accessed the memory location, and the analysis is similar to that for the race condition test for reads, except that $read[index] \vee write[index]$ is used, because writes conflict with both earlier writes and earlier reads.

5.3.3 Examples

To illustrate the race detection technique described above, we use the code in Figure 5.6. This code contains two simple kernels, `avg` and `avg2`, the purpose of which is to store in each element of array `a` the mean of that element and the two adjacent elements.

The `avg` kernel contains a race condition, while `avg2` uses an execution barrier to avoid the race. For each statement in the kernels, we show alongside it the state of the MAR for the first element of array `a` after execution of that statement. Note that in KLEE-CL we execute each work-item in its entirety until it reaches an execution barrier or terminates; however, our race detection algorithm would work with any other execution schedule. Thus, for `avg` the entirety of work-item 1 is executed before work-item 2, and the MAR persists from the end of execution of work-item 1 to the beginning of execution of work-item 2. For `avg2` the first five lines of work-item 1 are executed (up to the barrier), then the first five lines of work-item 2, the memory access records are locally reset, the last two lines of work-item 1 are executed and finally the last two lines of work-item 2.

On line 4 of `avg` in work-item 2, we report a read-after-write race. This is due to the earlier write of work-item 1 on line 7 causing the write flag to be set. This race does not exist in `avg2` because on line 4 of `avg2` in work-item 2, line 8 in work-item 1 had not yet been reached, as it had been preempted by the barrier on line 7.

Our second example, shown in Figure 5.7, illustrates data races across memory barriers, as well as the purpose of the many-read and wg-many-read flags. A data race exists and is reported

due to the write on line 4 in work-item 2 conflicting with the read on line 2 in work-item 1. Because the work-items are in different work-groups, the execution barrier on line 3 does not protect against the race (recall that execution barriers are local to work-groups). Neither does the execution barrier affect the execution order (assuming a single work-item per work-group) so the entirety of work-item 1 is executed followed by work-item 2 (though, as before, scheduling does not affect race detection). The read on line 2 in work-item 1 sets the work-item identifier, work-group identifier and the read flag. The same read in work-item 2 also sets the many-read and wg-many-read flags due to the work-group identifier stored in the MAR differing from work-item 2's work-group identifier.

When execution reaches line 3 in work-item 2, the many-read flag is cleared, but the wg-many-read flag remains set. Therefore, a race is reported at line 4. This demonstrates the purpose of the many-read and wg-many-read flags – because the work-item and work-group identifiers in the MAR are equal to the work-item's identifiers, there is no other way to determine that another work-item has read the byte. Note that if work-items 1 and 2 were in the same work-group, only the many-read flag would have been set at line 2, which would be cleared at line 3, so no race would be reported at line 4. In this scenario, if a write were to occur between lines 2 and 3, this would result in a data race being reported due to the many-read flag being set.

5.4 Summary

This chapter has described our technique for detecting errors in data parallel programs. We use a combination of program transformation, expression pattern matching, constraint solver enhancements and a memory access log to symbolically execute data parallel programs in an efficient fashion, while detecting implementation mismatches and data races.

Using these techniques, together with existing memory error detection support in KLEE, we were able to perform bounded verification of a number of open source algorithm implementations, and also find a number of issues including mismatches between implementations, memory errors, race conditions and compiler bugs, as we shall see in Chapter 6.

```
1  __kernel void avg(__global float *a) {
2    size_t lid  = get_local_id (0),
3           lsize = get_local_size (0);
4    float r0 = lid > 0 ? a[lid-1] : 0;
5    float r1 = a[lid];
6    float r2 = lid+1 < lsize ? a[lid+1] : 0;
7    a[lid] = (r0 + r1 + r2) / 3;
8  }
```

work-item 1, work-group 1

T_{id}	W_{id}	R	W	MR	WMR	Con
0	0					
0	0					
0	0					
1	1	✓				
1	1	✓				
1	1	✓	✓			

work-item 2, work-group 1

T_{id}	W_{id}	R	W	MR	WMR	Con
1	1	✓	✓			
1	1	✓	✓			
2	1	✓	✓	✓		w/r
2	1	✓	✓	✓		
2	1	✓	✓	✓		
2	1	✓	✓	✓		

```
1  __kernel void avg2(__global float *a) {
2    size_t lid  = get_local_id (0),
3           lsize = get_local_size (0);
4    float r0 = lid > 0 ? a[lid-1] : 0;
5    float r1 = a[lid];
6    float r2 = lid+1 < lsize ? a[lid+1] : 0;
7    barrier(CLK_GLOBAL_MEM_FENCE);
8    a[lid] = (r0 + r1 + r2) / 3;
9  }
```

work-item 1, work-group 1

T_{id}	W_{id}	R	W	MR	WMR	Con
0	0					
0	0					
0	0					
1	1	✓				
1	1	✓				
0	1	✓				
1	1	✓	✓			

work-item 2, work-group 1

T_{id}	W_{id}	R	W	MR	WMR	Con
1	1	✓				
1	1	✓				
2	1	✓		✓		
2	1	✓		✓		
2	1	✓		✓		
1	1	✓	✓			
1	1	✓	✓			

Figure 5.6: Intermediate MARs for the memory location at a[0] during execution of work-items 1 and 2. Column T_{id} shows the byte's work-item identifier, W_{id} its work-group identifier, R the read flag, W the write flag, MR the many-read flag, WMR the wg-many-read flag and Con (if present) the nature of the conflict detected at that line.

```
1  __kernel void copy(__global int *a) {
2    int x = a[2];
3    barrier(CLK_GLOBAL_MEM_FENCE);
4    a[get_group_id (0)] = x*2;
5  }
```

work-item 1, work-group 1

T_{id}	W_{id}	R	W	MR	WMR	Con
1	1	✓				
0	1	✓				
0	1	✓				

work-item 2, work-group 2

T_{id}	W_{id}	R	W	MR	WMR	Con
2	2	✓		✓	✓	
0	2	✓			✓	
0	2	✓	✓		✓	r/w

Figure 5.7: Intermediate MARs for a[2] during execution of work-items 1 and 2.

Chapter 6

Evaluation

We evaluated our techniques on a set of benchmarks that compare serial and data parallel variants of code developed independently by third parties. The codebases that we selected were the OpenCV computer vision library [BK08, IW], the Parboil benchmark suite [IMP], the Bullet physics library [C+] and the OP2 [GMS+11] library.

The implied constraint builder (rather than the floating point bit blaster) was used in all tests, unless otherwise stated.

6.1 SSE Acceleration in OpenCV

We evaluated a selection of computer vision algorithms from OpenCV 2.1.0, a popular C++ open source computer vision library initially developed by Intel and now by Willow Garage. It is an open-source project available under a BSD license [BK08, IW].

Although we had to make some changes to OpenCV for compatibility with KLEE-CL, these were minimal—they either replaced inline assembly code, which KLEE does not support, or disabled some functionality unrelated to the SSE code under test, but which KLEE had trouble executing.

Our benchmarks test a substantial amount of SSE code in OpenCV. Due to time constraints, out

Source File (`src/`)	Benchmarks	# SIMD	Coverage
`cv/cvcorner.cpp`	`eigenval` `harris`	44	100%
`cv/cvfilter.cpp`	`filter`	1332	0%

`cv/cvimgwarp.cpp`	`remap`
	`resize`
	`warpaff`
		1070	74.6%

`cv/cvmoments.cpp`	`moments`	35	100%
`cv/cvmorph.cpp`	`morph`	1220	43.6%
`cv/cvmotempl.cpp`	`silhouette`	43	100%
`cv/cvpyramids.cpp`	`pyramid`	125	44.0%
`cv/cvstereobm.cpp`	`stereobm`	270	53.3%
`cv/cvthresh.cpp`	`thresh`	238	100%

`cxcore/cxmatmul.cpp`	`transcf.43`
	`transsf.43`
	`transff.43`
	`transff.44`
		352	100%

Table 6.1: OpenCV code we tested with KLEE-CL. Coverage data refers to coverage of SIMD instructions, where an SIMD instruction is any instruction of vector type, any `extractelement` instruction, stores of vector operand type, casts from vector type and SSE intrinsics (name begins `llvm.x86.mmx`, `llvm.x86.sse` or `llvm.x86.ssse`).

of the twenty OpenCV source code files containing SSE code, we arbitrarily selected ten files for testing with KLEE-CL. To build benchmarks, we had to acquire a (brief) understanding of how to invoke each OpenCV algorithm in order to build a test harness similar to that in Listing 3.1. Section 6.5 provides more details regarding the manual effort involved in constructing a test harness.

Table 6.1 presents the ten files we tested, together with a list of benchmarks for that code and coverage data. Each of our benchmarks tests one of the algorithms provided by OpenCV. For example, `harris` tests the Harris corner detection algorithm, which finds a *corner* in a given image, intuitively a window that produces large variations when moved in any direction [BK08]. Each benchmark takes a number of parameters, including the size and format of the input and output images (represented by matrices) and the specific algorithm to test (for example, the `morph` benchmark can test an `erode` algorithm, which returns in each cell of the output matrix the minimum value of the corresponding cell in the input matrix and its neighbours, and a `dilate` algorithm which instead takes the maximum).

Since we are unable to use symbolically sized images (see Section 1.4), our methodology was

instead to test each benchmark on all possible image sizes up to 16×16 pixels. More precisely, we start with the minimum size for which an SSE variant of the algorithm under test exists (usually 4×1 pixels), and test all possible sizes until we reach images of 16×16 pixels or are unable to test any further due to the high complexity of the generated queries.

The SIMD instruction count for each source file gives a rough approximation of the overall complexity of the SSE code tested by our benchmarks. While it does not necessarily follow that the equivalent scalar code or the surrounding control flow is of similar complexity, we found the SIMD instruction count to be a good metric for the complexity of the computational routines of interest to us.

Some coverage numbers do not reach 100%. We found that this was generally caused by the presence of unrolled SSE code that was unreachable due to query complexity. The `filter` benchmark has 0% coverage because we weren't able to run it at all. We discuss the reasons in Section 6.5.

We constructed a total of 58 benchmarks to cover the functions in these ten files. KLEE-CL was able to successfully verify 41 benchmarks up to a certain image size (Section 6.1.1) and find mismatches in 10 benchmarks (Section 6.1.2). In addition, three benchmarks triggered false positives when using the implied constraint builder (Section 6.5(3)) and four benchmarks couldn't be run at all by KLEE-CL (Section 6.5(4)).

6.1.1 Benchmarks verified up to a certain image size

Table 6.2 presents the list of benchmarks and associated parameters that we were able to verify using KLEE-CL up to a certain image size. The Format column shows the format of the input and output images in terms of the data type (`f` = floating point, `s` = signed integer, `u` = unsigned integer) and the bitwidth of the format. The Maximum Size column shows the maximum image size we tested using our methodology. Sizes of the form $X \rightarrow Y$ indicate that the benchmark's input and output images are of different sizes: X is the maximum input image size, and Y the maximum output image size that we tested.

#	Benchmark	Algorithm	Kernel	Format	Maximum Size
1	morph	dilate	rectangular	u8	5×5
2				s16	16×16
3				u16	16×16
4			non-rectangular	u8	8×3
5				s16	16×16
6				u16	16×16
7				f32	15×15
8		erode	rectangular	u8	4×4
9				s16	16×16
10				u16	16×16
11			non-rectangular	s16	16×16
12				u16	16×16
13	pyramid			u8	$8 \times 2 \to 4 \times 1$
14	remap	nearest neighbour		u8	16×16
15				s16	16×16
16				u16	16×16
17				f32	16×16
18		linear		u8	16×16
19				s16	16×16
20				u16	16×16
21				f32	16×16
22		cubic		u8	16×16
23				s16	16×16
24				u16	16×16
25				f32	16×16
26	resize	linear		s16	$8 \times 8 \to 8 \times 8$
27				u16	$4 \times 1 \to 8 \times 2$
28				f32	$8 \times 8 \to 8 \times 8$
29		cubic		s16	$8 \times 8 \to 8 \times 8$
30				u16	$4 \times 1 \to 8 \times 2$
31				f32	$8 \times 8 \to 8 \times 8$
32	silhouette			u8 f32	16×16
33	thresh	BINARY		u8	16×16
34				f32	16×16
35		BINARY_INV		u8	16×16
36				f32	16×16
37		TRUNC		u8	16×16
38		TOZERO		u8	16×16
39				f32	16×16
40		TOZERO_INV		u8	16×16
41				f32	16×16
42	transff.43			f32	See §6.1.1
43	transff.44			f32	See §6.1.1

Table 6.2: OpenCV benchmarks verified up to a certain size.

The `transff`, `transsf` and `transcf` benchmarks use fixed size matrices. The `.43` variants take a 3-channel source array of size 4×4 and a 1-channel transformation matrix of size 3×4 and produce a 3-channel array of size 4×4, while the `.44` variants take a 4-channel source array of size 4×4 and a 1-channel transformation matrix of size 4×4 and produce a 4-channel array of size 4×4.

The `remap` benchmark tests the `cvRemap` routine, which performs symbolic conditional branching over the data contained in two of its three input matrices. Because the phi node folding pass is unable to simplify this branching structure, exponential forking results. Our compromise for this benchmark is to supply two concrete matrices and one symbolic matrix to `cvRemap`.

Two benchmarks—namely `resize` (linear, `u16`) and `resize` (cubic, `u16`—used query expressions of the form `FPToSI`(X) or `FPToUI`(X), which were converted to unconstrained variables when using the implied constraint builder (see Section 5.2.3). While the variable was unconstrained, the underlying floating point expression X was limited in its range, and STP produced counterexamples for the unconstrained variables outside of their feasible range. To test these benchmarks, we used the floating point bit blaster and the smallest image size that would trigger the execution of SIMD code, in order to produce constraint solver queries of a reasonable complexity.

As mentioned before, we ran each benchmark on matrices of up to 16×16 pixels or until we were unable to test any further due to the high complexity of the generated queries. While these are relatively small matrices, our results should be viewed in combination with the SIMD coverage data which shows that the image sizes we tested cover most SIMD code.

We measured the execution time taken by KLEE-CL for all of our experiments. However, because we ran our benchmarks on a heterogeneous cluster of machines, these times are mainly intended to give a rough idea of the computational cost involved in using our tool. The runtime of individual experiments (i.e., one benchmark run with a single matrix size) varied between less than one second to more than 40 hours. The total cumulative execution time per benchmark (i.e., for all matrix sizes) ranged from only a few seconds (for the `transff` benchmarks, which only work with a fixed matrix size) up to 27 days for `morph` (dilate, R, `u16`). Approximately

#	Benchmark	Algorithm	K	Format	Size	Description
1	`eigenval`			`f32`	4×4	Precision
2	`harris`			`f32`	4×4	Precision, associativity
3	`morph`	dilate	R	`f32`	4×1	Order of min/max operations
4			NR	`f32`	4×1	
5		erode	R	`f32`	4×1	
6	`thresh`	`TRUNC`		`f32`	4×4	
7	`pyramid`			`f32`	$16 \times 2 \rightarrow 8 \times 1$	Associativity, distributivity
8	`resize`	linear		`u8`	$4 \times 4 \rightarrow 8 \times 8$	Precision
9		cubic		`u8`	$4 \times 1 \rightarrow 8 \times 2$	Integer/FP differences
10	`transsf.43`			`s16 f32`	See §6.1.1	Rounding issue
11	`transcf.43`			`u8 f32`	See §6.1.1	Integer/FP differences

Table 6.3: OpenCV benchmarks in which we found mismatches between the scalar and the SSE versions.

21.1% of benchmarks had cumulative execution times of under ten minutes, 34.2% between ten minutes and one hour, 18.4% between one and twelve hours, and 26.3% over twelve hours.

6.1.2 Invalidated Benchmarks

Table 6.3 presents the list of benchmarks in which we found mismatches between the scalar and SSE implementations. Each mismatch was detected by KLEE-CL in less than 30 seconds.

We discuss each of the mismatches found below:

1. **eigenval and harris:**

   Both the `eigenval` and `harris` benchmarks compute certain values in double precision in the scalar implementation, which are computed in single precision in the SSE implementation. To determine whether this was the only difference between the implementations, we modified the scalar implementation to use single precision by replacing `double` with `float` and casting to single precision where appropriate (in C, a binary operation taking two floating point values promotes the lower precision operand to the type of the higher precision operand [Int99, §6.3.1.8]).

   This modification caused `eigenval` to pass our tests, but there was a further issue with `harris` regarding associativity. The scalar implementation of `eigenval` computes the

expression:

```
((float)k)*(a + c)*(a + c)
```

which the SSE code computes as:

```
_mm_mul_ps(_mm_mul_ps(t, t), k4)
```

where the variable `t` initially holds the four $a + c$ values, and `k4` holds four copies of k.

The IEEE floating point operations + and × are not associative, so these two expressions are not equivalent. The associativity issue may not be immediately obvious, but because `*` in C is left associative [Int99, §6.5.5], the scalar multiplication is implicitly bracketed as `(((float)k)*(a + c))*(a + c)`, which is clearly not equivalent to the SSE version. The discrepancy is also revealed by KLEE-CL, which is capable of printing the symbolic expressions involved. In this case, KLEE-CL outputs the following expressions, where N_0 and N_{65} are complex subexpressions shared between the two expressions:

$$\begin{aligned} \textbf{SIMD}: &\quad N_0 - ((N_{65} \times N_{65}) \times 0.04) \\ \textbf{Scalar}: &\quad N_0 - ((0.04 \times N_{65}) \times N_{65}) \end{aligned}$$

As it can be seen, the KLEE-CL encoding of the operation, which provides explicit bracketing, makes associativity errors such as this much easier to spot.

2. **morph (f32) and thresh (`TRUNC`, f32)**

Both benchmarks involve floating point `min` and/or `max` operations. The SSE and scalar variants of the implementations apply `min` and `max` to the same operands but in a different order. We cannot consider the two expressions to be equivalent because the `min` and `max` operations used are idiomatic and therefore, as we saw in Section 5.2.3, neither associative nor commutative.

The SSE instructions `MINPS` and `MAXPS` implement the `min` and `max` operations using the

idiom directly:

$$\texttt{sse\_min}(X,Y) = \texttt{min}(X,Y) = \texttt{Select}(\texttt{FOlt}(X,Y),X,Y)$$
$$\texttt{sse\_max}(X,Y) = \texttt{max}(X,Y) = \texttt{Select}(\texttt{FOlt}(Y,X),X,Y)$$

The STL functions `std::min` and `std::max` used by the scalar variants of the benchmarks are not required by the C++ 2003 standard [Int03] to be implemented in any specific way (the result is undefined if either of the operands is `NaN`, because the $<$ operator for floating point numbers is not a strict weak ordering [Int03, §25.3] in the presence of `NaN`s). The GNU STL implements them idiomatically, but with the operand order reversed:

$$\texttt{stl\_min}(X,Y) = \texttt{min}(Y,X)$$
$$\texttt{stl\_max}(X,Y) = \texttt{max}(Y,X)$$

3. **pyramid (f32)**

   The SIMD variant of this code produces radically different symbolic expressions than the scalar variant. To give an example, we show below an expression extracted from the scalar variant of the algorithm:

$$((N_0 + N_0) + (N_0 + N_0)) + ((N_3 + N_0) \times 4.0)$$

   The corresponding SSE expression at the same position is:

$$(((N_0 \times 6.0) + (N_3 \times 4.0)) + N_0) + N_0$$

   N_0 and N_3 are complex subexpressions shared between the two expressions. To rearrange the first form into the second would require not only associativity but distributivity properties. Because the IEEE floating point $+$ and $\times$ are neither associative nor distributive, the equality does not hold.

4. **resize (linear, u8)**

The scalar variant of this code produces expressions of the form (simplified to remove irrelevant saturation checks):

$$(((1536 \times N_0) + (512 \times N_0)) + 2097152) >> 22$$

whereas the SIMD variant produces expressions of the form:

$$(2 + (((1536 \times (N_0 >> 4)) >> 16) +$$
$$((512 \times (N_0 >> 4)) >> 16))) >> 2$$

All intermediate values are 32 bits. The SIMD variant loses 11 bits of precision through right shifts before the addition operation, while the scalar variant retains all precision until the final right shift. This leads to differences where the lower 11 bits of N_0 affect the upper 10 bits of the addition result.

5. **resize (cubic, u8)**

   The SSE variant of resize (cubic, u8) performed floating point calculations whereas the scalar variant performed integer calculations. Analysis of such expressions would require reasoning about floating point semantics, so we used the floating point bit blaster for this benchmark. KLEE-CL reported a mismatch; we ran the benchmark concretely with the generated counterexample, and found this to be a true mismatch.

6. **transsf.43**

   The scalar variant of this code performs a rounds-to-nearest floating-point to unsigned 16-bit integer conversion. Because of the CPU's lack of support for floating-point to unsigned integer conversion, the conversion is performed by converting to a signed 32-bit integer and downcasting. On the other hand, the SIMD variant performs the conversion by first subtracting 32768 from the floating point number, performing a conversion directly to a 16-bit signed integer and adding 32768 to the result. While this may appear correct, it will produce different results in certain edge cases.

For example, consider the value $0.5 + \epsilon$, where ϵ is a value sufficient to shift 0.5 to the next higher floating point representation. If this value is converted directly to an integer, as in the scalar version of the code, the value is rounded up to the nearest integer value, this being 1. On the other hand if we subtract 32768 from the floating point value, as in the SIMD variant of this code, ϵ will be lost during rounding and the result is -32767.5. When this value is converted to an integer, it is rounded *down* to -32768 (under this rounding mode, ties are rounded to the nearest even value), and the result is 0 after adding 32768 back.

7. **transcf.43**

   The scalar variant of this code performs floating point calculations whereas the SIMD variant operates over 32-bit fixed point values with 10 bits of precision below the radix point. When the SIMD variant converts the floating point input values into this format, precision can be lost if the floating point exponent is less than 13. This leads to different results where the lower order bits of the floating point input values affect the final result.

We reported the mismatches we found to the OpenCV developers. At the time of this writing, we have received an answer for five out of the ten mismatches listed in Table 6.3. The developers confirmed the precision and associativity mismatches in the `eigenval` and `harris` benchmarks as real issues and informed us of their intention to fix them. In response to the mismatches in `morph` caused by the different order of min/max operations, we received the following answer:

"I wonder, if your tool can be told to ignore the NaN's in the certain function? Because we never assumed that NaN's are possible in the morphological functions' input data and do not see any reason for such assumption." (Vadim Pisarevsky, personal communication)

In response, we added the ordered assumption mentioned in Section 5.2.1, and made it apply to min/max operations as discussed in Section 5.2.3. With this assumption enabled, KLEE-CL was able to prove the equivalence of the respective benchmarks on images up to 15×15. The tool reported another mismatch on an image of 16×16, which we are currently investigating.

6.2 OpenCL Acceleration in Parboil

Parboil [IMP] is a popular GPU benchmark suite, which contains C and CUDA [NVI10] implementations of various algorithms. In order to be able to run Parboil benchmarks using KLEE-CL, we used Grewe et al's [GO11] translation of certain Parboil 1 benchmarks from CUDA to OpenCL. The translation comprised four benchmarks in total, and we tested three of these: `cp` (Coulombic Potential), `mri-q` (Magnetic Resonance Imaging – Q) and `mri-fhd` (Magnetic Resonance Imaging – FHD). We were unable to test the fourth benchmark, `rpes` (Rys Polynomial Equation Solver) for reasons discussed in Section 6.5.

We modified the code for each benchmark to incorporate the C and OpenCL versions of the benchmarks into the same executable. This allowed us to construct simple test harnesses similar to the one in Listing 3.1 which invoke both versions of the benchmarks with the same symbolic arguments.

By running these benchmark programs using KLEE-CL, we detected three mismatches between the C and OpenCL implementations of `cp`. We also found three memory errors in `mri-q` and `mri-fhd` as a result of the memory bounds checking performed during symbolic execution.

Mismatches: The `cp` benchmark computes the Coulombic potential for a set of points on a grid. The computation of a Coulombic potential at a grid point involves the calculation of the Euclidean distance of the form $\sqrt{\delta x^2 + \delta y^2 + \delta z^2}$ between an electrically charged particle and that point.

The first mismatch for `cp` is due to an associativity issue. The OpenCL implementation uses an unrolled loop in which a set of adjacent grid points are computed during each iteration. Because only the x coordinate varies during an iteration, the values of δy and δz remain constant, allowing $\delta y^2 + \delta z^2$ to be precomputed at the start of each iteration. So the expression is evaluated as $\sqrt{\delta x^2 + (\delta y^2 + \delta z^2)}$. In the C implementation, the inner expression is left unbracketed and normal C associativity rules apply. Because + is left-associative in C [Int99], the expression is evaluated as $\sqrt{(\delta x^2 + \delta y^2) + \delta z^2}$. Since + in floating point is not associative, the two expressions do not match.

The second mismatch arises in the context of computing δx in the two implementations. In the C implementation, this is done by subtracting the atom's x coordinate from the grid's x coordinate. In the OpenCL implementation, δx for the iteration's first grid point is computed in the same way. However, for subsequent points in the iteration, δx is computed by adding the grid's spacing to the value of δx for the previous point. Since floating point $+$ and $\times$ are neither associative nor distributive, the expressions do not match.

Whether these mismatches are important or not depends on the specific application. KLEE-CL's job is to flag such mismatches, but it is up to the developer to assess whether strict equivalence should be enforced. Furthermore, developers can use the assumptions discussed in Section 5.2 to ignore the cause of different mismatches. For the current example, developers could add the assumption that floating point operations are associative and rerun KLEE-CL to find other problems. With this assumption enabled, KLEE-CL verifies a variant of this benchmark in which the second mismatch, but not the first, has been fixed.

Memory errors: A non-obvious use-after-free error was found in `mri-q`. After the OpenCL kernel is invoked, `mri-q` deallocates some OpenCL memory buffers and then copies some data from the GPU to the host. Because OpenCL kernel invocation is asynchronous, the memory buffers may be deallocated by the time that the kernel accesses them. Using the technique described in Section 4.5.3, KLEE-CL was able to detect this error, which we fixed by moving the data copies before the memory deallocations. Since the data copies were synchronous, they caused execution of the main thread to be preempted until after kernel execution.

A memory error found in both `mri-q` and `mri-fhd` was caused by a read beyond the end of a memory buffer used to store (x, y, z) coordinates. This memory buffer was indexed using the work-item identifier, which ranged between 0 and a multiple of the work-group size. This error was never caught, perhaps due to the fact that all benchmark data provided with Parboil had a size that was a multiple of the work-group size. We fixed these errors by enclosing the relevant part of the kernel inside an `if` statement.

A memory error found in `mri-fhd` is related to the use of uninitialised memory. This benchmark allocates a buffer of output data using `memalign`, which was assumed to be zero initialised. Since

`memalign` buffers are uninitialised, and KLEE-CL models this, incorrect results were produced. The fix was simply to initialise the buffer using `memset`.

6.3 OpenCL Acceleration in the Bullet Physics Library

Bullet [C+] is a physics library primarily used in gaming and 3D applications. It incorporates a number of physics simulation algorithms, including a soft body simulation. This can be used to simulate objects such as cloths which are freely deformable within the environment. Bullet provides a C++ and an OpenCL implementation of the soft body simulation.

We implemented two benchmark programs which create a simulation with two soft body objects, each containing three vertices connected by three edges. The coordinates of the vertices are concrete values, but all other simulation parameters are symbolic. The program runs a single simulation step using both the C++ and the OpenCL implementations, and compares the results.

The first of our benchmarks (`softbody`) tests the soft body simulation in isolation, while the second benchmark (`dynworld`) tests the simulation using a soft rigid dynamics world, which exercises more of the soft body code.

We used our benchmark programs to test SVN revision 2357 of Bullet. For the `softbody` benchmark, KLEE-CL verified that the C++ and OpenCL code produce the same results. For `dynworld`, KLEE-CL was able to verify equivalence under the finite and positive zero assumptions, i.e. the assumption that $x \times 0 = 0$ in floating point.

At the time that we initially performed this test, the LLVM IR generated by the Clang compiler did not provide the accuracy of each individual operation, and therefore we did not model the single precision floating point division operation correctly. Using the technique described in Appendix A, we ran a test using both a CPU and real GPU hardware (an NVIDIA Tesla C1060), and found that discrepancies between the CPU and GPU results were introduced by such an operation. After adding floating point accuracy support we re-ran the benchmark in KLEE-CL,

which correctly reported a mismatch. We attempted to rectify the issue in the OpenCL code by casting the operands of the division operator to double precision ([Khr10, § 9.3.9] requires double precision division to be correctly rounded). With this change, KLEE-CL was able to verify the program's correctness.

OpenCL compiler bug: Of course, these equivalence results hold under the additional assumption that all the components involved in running the code—from compilers to hardware—are correct. The bug discussed below illustrates this point.

After fixing the single precision issue mentioned above, we were surprised to see that the test run on real GPU hardware still showed discrepancies between the OpenCL and C++ implementations, despite the fact that we were able to verify their equivalence. After further investigation, we found that the PTX assembly code produced by NVIDIA's OpenCL compiler continued to use a single precision division instruction (`div.full.f32`), despite the cast to double precision. If we disabled compiler optimisations, using the `-cl-opt-disable` flag to the OpenCL compiler, the double precision division instruction (`div.rn.f64`) was used. This suggested that the problem may lie in the optimiser.

We worked around this issue by postprocessing the PTX code to replace `div.full.f32` with `div.rn.f64` together with appropriate conversions, similar to the unoptimised code. After doing this, the results obtained were identical.

We reported the issue to NVIDIA who confirmed our bug report, and as of this writing had fixed the bug, but had not yet released a version of their OpenCL implementation with the fix.

6.4 OpenCL Acceleration in OP2

OP2 [GMS+11] is a library for generating parallel executables of applications using unstructured grids. OP2 enables users to write a single program targeting multiple platforms. OP2 has four implementations: a serial reference (library) implementation and source-to-source trans-

```
1  int tid = get_local_id(0), d = get_local_size(0)>>1;
2  __local volatile float *vtemp = temp;
3  ...
4  for (; d>0; d>>=1) {   /* d is at most 16 here */
5    if (tid<d) {
6      ...
7      vtemp[tid] = vtemp[tid] + vtemp[tid+d];
8      ...
9    }
10 }
```

Listing 6.1: OP2's unsynchronised loop (slightly modified for formatting purposes).

formations to CUDA, OpenCL and OpenMP.

Among the operations offered by OP2 is the *global reduction* operation, which is used to reduce a set of results computed across a set of grid nodes to a single result. We used KLEE-CL to test the correctness of the OpenCL implementation of the global reduction operation by extracting the relevant kernel from the OP2 source code and constructing a benchmark program which uses this kernel to perform a global reduction on an array of symbolic data.

KLEE-CL detected a race condition in this kernel, and the problematic code is shown in Listing 6.1. Each iteration of the `for` loop on lines 4–10 uses a result computed in an earlier iteration by another work-item (specifically, work-item `tid` uses a result computed by work-item `tid+d`) without using an execution barrier beforehand. Because of the lack of synchronisation, the behaviour of the kernel is undefined by the OpenCL specification.

To understand why this loop was written in this way, one must consider the history of the code. The OpenCL implementation was heavily based on the CUDA implementation and was in many places developed by replacing CUDA constructs with the relevant OpenCL constructs. In CUDA (and the NVIDIA GPU architecture), each group of 32 work-items within a work-group (referred to as a *warp*) is executed in lockstep with implicit synchronisation between work-items [NVI10]. However, no such feature is present in OpenCL, and OpenCL code relying on warps has implementation-defined behaviour. In the case of the NVIDIA implementation of OpenCL this happens to function correctly, however there is no requirement that it do so on other architectures.

We modified the kernel to introduce a local execution barrier using the `barrier` function before each iteration of the loop (between lines 4 and 5). With this modification in place, KLEE-CL does not report a race condition.

6.5 Applicability and Limitations

Our experimental evaluation has helped us better understand the applicability of our tool, and its main limitations. We have identified four main aspects that developers should be aware of when using KLEE-CL:

1. **KLEE-CL as a development tool:** Manually translating serial code into an equivalent data parallel version is a difficult process. Due to the restrictions of floating point arithmetic, constructing two equivalent floating point expressions usually requires the same sequence of operations, and as a result, we found that in writing parallel code, developers tend to closely imitate the operations performed by the scalar code. Unfortunately, the process is error-prone, and developers often make invalid assumptions about the properties of floating point arithmetic, such as those related to associativity, distributivity, precision, and rounding. We believe that KLEE-CL could be effectively applied as a development-time tool that would assist programmers with the parallelisation process, or with any other optimisation task that requires the equivalence of two different code fragments.

   We believe the initial feedback we received from the OpenCV developers is consistent with our envisioned use of KLEE-CL as a development tool. Developers would incrementally apply our technique on increasingly bigger inputs until no more mismatches are found and/or they gain enough confidence in their translation. Once a mismatch is found, they would either fix the code and look for more problems, or they would improve the precision of the tool by adding additional expression rewrite rules. To improve the usability of KLEE-CL for the latter scenario, the tool would benefit from the ability to specify

additional rules in a higher-level language like the one we use to describe the rules in Table 5.2.

2. **Manual effort:** To use our tool, developers have to write a test harness, similar to the one implemented by the `main()` function in Listing 3.1. This requires the ability to construct the input data structures required to invoke the function under testing, and to identify the output structures that should be compared for equivalence. In the case of code operating on complex, application-specific data structures, this can be a difficult task, especially for people not familiar with the codebase under testing. This is a problem shared with testing in general, and unit testing in particular, and represents the main reason for which we did not have time to test all the SIMD code in OpenCV. However, KLEE-CL is designed as a developer tool, and the software developers familiar with the API of the code under testing would be in a better position to rapidly develop this kind of test harnesses.

3. **False positives and counterexamples:** Because KLEE-CL's implied constraint builder is based on expression matching augmented by canonicalisation rules, it is prone to false positives, i.e., it can say that two expressions are not equivalent when in fact they are, although this can be avoided using the floating point bit blaster, at the expense of execution time. However, remember that KLEE-CL has no false negatives, i.e., when it says that two expressions are equivalent, this is guaranteed to be true.

   We discovered three suspected false positives in the OpenCV experiments when using the implied constraint builder, two of which could be verified for the specified image size using the bit blaster (`resize` (linear, `u16`) and `resize` (cubic, `u16`)) and one of which we were able to produce a counterexample for using the bit blaster (`resize` (cubic, `u8`)).

   Our experience integrating an external floating point bit-blaster into KLEE-CL has shown that while this is a promising means of improving accuracy and obtaining counterexamples for floating point programs, there exist at least two drawbacks:

   (a) As we predicted, the constraint solver can take a significant amount of time to terminate, even for small problems. With a dual core Intel Core 2 Duo E6850 at 3.0

GHz with 8 GB of RAM, KLEE-CL was able to successfully verify `resize` (linear, `u16`) in 42 seconds and `resize` (cubic, `u16`) in 46 minutes 36 seconds for only the small image sizes given.

(b) Existing floating point bit blasters are still fragile, and are also prone to exposing bugs in KLEE-CL or STP due to the complex nature of the bit vector level constraints produced. While porting `floatbv` to KLEE-CL we found and fixed several bugs in KLEE-CL's STP expression builder. Furthermore, while we were able to obtain a true counterexample after implementing round to nearest float to int conversion, the same version of KLEE-CL also produced several false counterexamples which we were unable to track down – the bug may reside in `floatbv`, KLEE-CL or STP.

The first drawback can be mitigated to some extent using automated abstraction refinement, as discussed in Section 7.3.

4. **Symbolic execution and constraint solving limitations:** There were also five benchmarks that we were unable to run at all using KLEE-CL.

   For OpenCV, the `filter` benchmark invoked `malloc` with a symbolic argument. While KLEE is normally able to recover from a symbolic memory allocation using STP to determine the maximum value of the argument, in this case the argument was built from a floating point expression and KLEE-CL was unable to find a maximum, resulting in an error. The other three benchmarks (`stereobm`, `moments` and `warpaff`) presented queries to STP that were too complex to handle, meaning that they caused STP to run for an unbounded amount of time or consume all available memory.

   For Parboil, the `rpes` benchmark could not be executed because it created a very large number of work-items (> 30000) even for small problems, which KLEE-CL could not execute in a reasonable amount of time.

Chapter 7

Conclusion and Future Work

This thesis has made the following contributions:

1. We have presented a symbolic execution based technique for crosschecking data parallel programs in SIMD and OpenCL against their serial equivalents.

   We have found this technique to be effective for two main reasons:

   - It requires little more than existing serial and data parallel implementations of a given algorithm, which as we have observed tend to both already exist, allowing developers to utilise this technique without significant implementation work.
   - It builds on existing infrastructure for symbolic execution of serial code in languages such as C and C++, languages which are commonly used to write highly performant code.

2. We reason about floating-point values (which KLEE's constraint solver cannot handle), using expression matching augmented with canonicalisation rules that express strict equivalences in floating-point and mixed FP-integer expressions. As far as we know, this is the first practical symbolic execution based technique that can precisely handle IEEE 754 floating point arithmetic.

   Our evaluation (Chapter 6) has shown that this technique can be applied in the vast majority of cases, while exact bitblasting can be used in the remainder of cases.

3. We address the path explosion problem associated with symbolic execution by statically merging paths using phi node folding, a form of if-conversion (Section 5.1).

   Our evaluation (Section 5.1.3) has shown that this technique can be applied to three of our ten OpenCV benchmarks, allowing us to symbolically execute them in a reasonable amount of time.

4. We present a technique for symbolically testing for the presence of data races in OpenCL programs using a memory access log (Section 5.3).

   The memory access log technique has an $O(1)$ runtime complexity in the common case of concrete memory accesses, and has successfully found a data race in one of our benchmarks (Section 6.4).

5. We implement our techniques in a tool called KLEE-CL, an extension to the open source symbolic execution tool KLEE [KLE] (Chapter 3).

6. We evaluate KLEE-CL by applying it to the OpenCV computer vision library, three Parboil benchmarks, the Bullet physics library and the OP2 library, and show that it can find real bugs, including memory errors, race conditions, and implementation mismatches (Chapter 6).

KLEE-CL and the benchmarks used in this thesis are freely available from our website:

`http://www.pcc.me.uk/~peter/klee-cl/`

The remainder of this chapter examines a number of possible extensions to this work.

7.1 Symbolic Testing of Automatic Vectorisations

Automatic vectorisation techniques provide an alternative to verifying the correctness of manually written SIMD code [EWO04, LA00, NBBDZ03]. However, even as these techniques will

start to be more widely adopted, the approach presented in this thesis can be applied to verify these automatically generated SIMD vectorisations.

The *polytope model* is one technique for automatic parallelisation, including vectorisation. It is a powerful means of re-arranging a nested loop to eliminate data dependencies between iterations. This elimination process allows for loops to be parallelised. Polly [GZA+11] is a polyhedral optimiser that works with the same LLVM intermediate representation used by KLEE and builds parallelised programs, either using SIMD or OpenMP shared memory parallelism. OpenCL support is also planned [GA11]. KLEE-CL could in principle be used to verify SIMD and OpenCL parallelisations built by Polly using the crosschecking technique described in this thesis, by symbolically executing both the unoptimised and optimised LLVM intermediate representations.

7.2 Detecting Inter-Event Data Races

One topic that has not been explored is inter-event data race testing in OpenCL. An inter-event data race can occur if a pair of events (normally a pair of kernel invocation events, but this can also include any type of event, such as data transfers) contain a data dependency, but no explicit ordering dependency. While none of our benchmarks use parallel event invocation, such behaviour is likely to become more common as applications increase their reliance on parallelism.

We can support inter-event data race testing by extending the happens-before relation between memory accesses to take into account the relation between OpenCL events. One can imagine the happens-before relation between events as a directed acyclic graph, with events represented as nodes, and ordering dependencies represented as edges. An order edge will be created from each newly enqueued event node to the following nodes:

- If the event's command queue is not out-of-order, the (previously) most recent event enqueued within its command queue, if any.

- Each event on the event's wait list (i.e. the `event_wait_list` parameter supplied to most command enqueueing functions).
- The command queue's synchronisation point event, if any.
- The context's synchronisation point event, if any.

The happens-before relation for memory accesses defined in Section 2.5 is then extended as follows, where event(x) is a unique identifier for the event that performed x and $x \Rightarrow y$ is true iff there exists a path in the DAG from x to y:

$$a_1 \prec a_2 \leftrightarrow (\text{wi}(a_1) = \text{wi}(a_2) \wedge a_1 \prec_{wi} a_2) \vee (\text{wg}(a_1) = \text{wg}(a_2) \wedge \text{bar}(a_1) < \text{bar}(a_2)) \vee \text{event}(a_1) \Rightarrow \text{event}(a_2)$$

Each OpenCL context and command queue has an associated *synchronisation point* event. The context's synchronisation point event would be set by `clFinish`, `clWaitForEvents` and the blocking runtime functions, which impose an ordering on all future events enqueued to the device, while the command queue's synchronisation point event would be set by `clEnqueueBarrier` and `clEnqueueWaitForEvents` (for a description of these functions, see [Khr10, §5.10]), which only impose an ordering on all future events enqueued to that command queue. In each case, the synchronisation point event acts only as a phony no-op event intended only to enforce an ordering between other events. In the case of `clFinish` and `clEnqueueBarrier`, the event would have an ordering edge to each event node associated with the command queue without an in-edge from another event associated with the same command queue. In the case of `clWaitForEvents`, `clEnqueueWaitForEvents` and the blocking runtime functions, the event would have an edge to each event node supplied as a parameter to the function. In all cases, the synchronisation point event would also have an edge to the command queue's and context's previous synchronisation point events, if any.

The happens-before DAG would be used not only to form the happens-before relation but also to decide the minimal set of commands to execute when `clWaitForEvents`, `clFinish` or

a blocking runtime function are called, as well as the order in which to execute them. As mentioned in Section 4.5.3, we can detect memory errors most accurately if a minimal subset of commands is executed when the host program is blocked waiting on command execution. At such a point, we would execute only those (uncompleted) events which are reachable from the event nodes supplied as an argument to the `clWaitForEvents` or blocking runtime function, or the event nodes belonging to the command queue supplied as an argument to `clFinish`, in any valid order according to the happens-before DAG.

A number of issues will need to be resolved for this to be implemented. The memory access record will need to be extended to include enough information to decide the reachability between two nodes in the happens-before DAG, together with a suitable SMT query to support symbolic race detection.

7.3 Floating Point Bit Blasting as an Abstraction Refinement

It can be observed that implied constraint building (Section 5.2.3) can act not only as an alternative to full floating point bit blasting (Section 5.2.4), but as an overapproximation of it. Using an abstraction refinement approach similar to that set out in [BKW09], it is possible to use the implied constraint builder in the first instance to produce an overapproximated constraint set. Should the constraint solver then find the constraint set produced by the implied constraint builder to be satisfiable, a floating point bit blaster would then be used to verify that the refined constraint is satisfiable, and if so to produce a satisfying counterexample. In this way, we benefit from both the advantages of implied constraint building (an efficient algorithm in the majority of unsatisfiable cross-checking cases) and of bit blasting (zero false positives, ability to produce counterexamples).

```
1  __kernel void foo() {
2
3    int x = (get_local_id() == 0 ? 4 : 1);
4    int y = (get_local_id() == 0 ? 1 : 4);
5
6    for(int i = 0; i < x; i++) {
7      for(int j = 0; j < y; j++) {
8        barrier(CLK_LOCAL_MEM_FENCE);
9        // Compute something
10     }
11   }
12
13 }
```

Listing 7.1: Barrier divergence example with loops.

7.4 Execution Barrier Divergence Testing

Currently, KLEE-CL does not attempt to detect barrier divergence errors. Programs with divergent barriers will have unpredictable behaviour in KLEE-CL, and are likely to deadlock. As mentioned in Section 2.5, there are several benefits to adding execution barrier divergence testing, including more comprehensive checks and improved diagnostics.

A lack of barrier divergence implies that all work-items within the work-group will always reach the same textual `barrier` call within the source code together. Thus, a naïve method of adding barrier divergence checking to KLEE-CL may be to cause the compiler to add a unique identifier to each individual call to the `barrier` function, and at each `barrier` call, check that each work-item's identifiers are equivalent. However, this is not sufficient, due to the rule which requires that each work-item reach the barrier during the same iteration of a loop.

For example, consider the kernel shown in Listing 7.1 (courtesy of Alastair Donaldson). This kernel will always reach the same textual execution barrier on line 8 four times regardless of which work-item is being run, satisfying the identifier-based check. However, barrier n will be reached during the nth iteration of the outer loop in work-item 0, and the nth iteration of the inner loop in all other work-items. Thus, the loop rule for `barrier` is not satisfied.

Our solution to this issue is to maintain a trip count for each loop currently being executed.

Operation	Value Stored
`atomic_add(p, val)`	$*p + val$
`atomic_sub(p, val)`	$*p - val$
`atomic_xchg(p, val)`	val
`atomic_inc(p)`	$*p + 1$
`atomic_dec(p)`	$*p - 1$
`atomic_cmpxchg(p, cmp, val)`	if $*p = cmp$ then val else $*p$
`atomic_min(p, val)`	$\min(*p, val)$
`atomic_max(p, val)`	$\max(*p, val)$
`atomic_and(p, val)`	$*p$ `AND` val
`atomic_or(p, val)`	$*p$ `OR` val
`atomic_xor(p, val)`	$*p$ `XOR` val

Table 7.1: List of atomic operations supported by OpenCL C.

These trip counts would be supplied to the `barrier` function, and the relevant test would then be whether both the `barrier` identifier and trip counts are equal.

7.5 Verification of OpenCL Kernels which use Atomics

As mentioned in Section 2.5, OpenCL C provides a set of read-modify-write atomic operations, a full list of which is shown in Table 7.1. One limitation of our current technique is that it does not handle these operations. In this section, we will briefly discuss one technique which we believe may be feasible for handling atomics.

The key difficulty for handling atomics is that in general it is necessary to consider every possible execution schedule. While techniques such as partial order reduction [FG05] may be a viable technique for pruning the number of schedules, such techniques may not scale to data parallel programs containing on the order of hundreds of threads. In this section, we will briefly discuss another technique based on SMT constraints which may eliminate the need to consider more than one schedule in some cases.

Note that our technique presupposes that the same operation is atomically applied to the memory location in every work-item, and that the operation is commutative and associative. Because in data parallel languages such as OpenCL it is highly likely that the same operation

will be applied in every work-item, and because almost every atomic operation offered by OpenCL C – `add`, `sub`, `inc`, `dec`, `min`, `max`, `and`, `or`, `xor` – is both associative and commutative (all atomic operations operate on integers), we believe that it is worth considering only this common case.

7.5.1 Modelling Atomic Operation Semantics

We can correctly model the memory-access portion of the atomic operation by loading, operating and storing directly, while taking care to adjust the MAR such that any non-atomic memory access will conflict with the atomic memory access, but that two atomic memory accesses will not conflict (this may be accomplished by adding another field to the MAR).

Returning the old value is more complicated, because this value may be used straight away by the kernel, while it may be affected by other atomic memory accesses which may not yet have been observed by the symbolic execution engine. Our solution is to return an unconstrained symbolic value, which we will later constrain as described in the next section. Every time the symbolic execution engine encounters an atomic write for a given variable it will return an unconstrained variable $seen_n$ (where *seen* is a series of unconstrained variables associated with the atomic variable, and n is a value, initially 0, associated with the atomic variable) and increment n.

7.5.2 Constraining Atomic Results

All unconstrained values associated with an atomic variable can be constrained at a point (henceforth known as a *synchronisation point*) at which it is known that for all atomic operations e_1, e_2 that affect that variable, where e_1 has been observed by the symbolic execution engine before the synchronisation point and e_2 may be observed after the synchronisation point, $e_1 \prec e_2$. For atomic variables in local memory, there is a synchronisation point at the start of kernel execution, at each execution barrier and at the end of kernel execution. However, for atomic variables in global memory, there are synchronisation points only at the start and end of kernel

execution, as execution barriers are local to the work-group, and it is therefore not known whether a work-item belonging to another work-group will access the atomic variable until all work-items in the NDRange have completed.[1]

For each variable that has been modified atomically between this synchronisation point and the previous one, we build a set of n unconstrained variables $ord_{0\ldots n-1}$ representing the order in which the modifications occurred, and constrain those variables such that they contain a unique integer between 0 and $n-1$, where n is the number of atomic writes to that variable. These variables are referred to as *order variables*. For example, for a set of 3 order variables, we could use the constraint:

$$ord_0 < 3 \wedge ord_1 < 3 \wedge ord_2 < 3 \wedge ord_0 \neq ord_1 \wedge ord_0 \neq ord_2 \wedge ord_1 \neq ord_2$$

An example assignment for $ord_0 \ldots ord_2$ is $[1, 2, 0]$, meaning that the second atomic operation encountered by the symbolic execution engine actually happened first, followed by the third operation encountered, followed by the first encountered.

We then use the numeric operands provided to each of the atomic operations to build an array of operands in the order encountered by the symbolic execution engine, which we refer to as *op*. For example, presuming that each of the atomic operations encountered respectively added 15, 20 and 25 to the atomic variable, *op* would be $[15, 20, 25]$.

We then use the order variables and *op* to build a series of expressions $ordVal$ representing the return values of the atomic operation in the order that the atomic variable was modified according to the order variables, such that $ordVal_0$ is the original value of the atomic variable and $\forall i$ between 1 and $n-1$, $ordVal_i$ is the result of applying the atomic operation with $ordVal_{i-1}$ and $op[ord_{i-1}]$ as operands. For example, if the operation was atomic addition, $ordVal_i = ordVal_{i-1} + op[ord_{i-1}]$.

The $ordVal$ series is then used to build an array $seenVal$ containing the resultant expressions,

[1] Note however that this is not true in the case where multiple kernels have access to the memory region in which the atomic variable resides, and have no happens-before relationship between them.

such that $\forall i$ between 0 and $n-1$, $seenVal[ord_i] = ordVal_i$.

We then constrain the variables $seen_0 \ldots seen_{n-1}$ such that $\forall i$ between 0 and $n-1$, $seen_i = seenVal[i]$. Finally, we set n to 0 and de-associate the *seen* series of variables with the atomic variable such that any further accesses to the atomic variable use a fresh series of variables. The MAR for the atomic variable is also locally or globally reset, as usual.

While the formulation described above is fully general, it may impose a significant burden on the constraint solver due to heavy use of the theory of arrays. An optimisation may be applied in the case where each operation adds an identical delta δ (which may be negative) to the atomic variable. Under this optimisation, the variables $seen_0 \ldots seen_{n-1}$ are constrained such that $\forall i$ between 0 and $n-1$, $seen_i = ordVal_0 + (\delta \times ord_i)$, and so we avoid constructing the *op* and $seenVal$ arrays. Note that under the optimisation, the meaning of the order variables is effectively inverted. So the assignment $[1, 2, 0]$ for $ord_0 \ldots ord_2$ would mean that the third operation encountered happened first, followed by the first, followed by the second.

7.5.3 Preliminary Evaluation

The technique described in this section has not been implemented, but we can make some estimates regarding its burden on the symbolic execution engine. First of all, we can observe that the symbolic execution time for programs which do not use the result of atomic operations should not increase significantly as a result of this technique, provided that the constraint solver can eliminate unused variables. An informal search has revealed a significant proportion of programs which do not use the return value of atomic operations.

In the case where return values are used, we can estimate the burden on the constraint solver by constructing benchmark SMT problems similar to those which may be constructed during symbolic execution, and measuring the constraint solver execution time for each problem. In our case, we chose to examine bounds checking problems which may be derived from a kernel such as that shown in Listing 7.2, specifically the array access on line 8. Listing 7.3 shows an example of such a problem with 3 updates (i.e. 3 work-items). Because each offset is constant,

```
1 __kernel void f(__global int *a) {
2   __local int index;
3   if (get_global_id(0) == 0)
4     index = 1;
5   barrier(CLK_LOCAL_MEM_FENCE);
6
7   int i = atomic_add(&index, 3);
8   a[i] = 1;
9 }
```

Listing 7.2: Example OpenCL kernel that uses atomics.

this particular problem may be optimised to eliminate use of the array theory. The optimised problem is shown in Listing 7.4.

Figure 7.1 graphs the execution time of STP revision 1603 on a dual core Intel Core 2 Duo E6850 at 3.0 GHz with 8 GB of RAM for both unoptimised and optimised problems with an update count between 2 and 16. While execution time appears to increase at a linear rate for the optimised problem, it appears to increase at a super-exponential rate for the unoptimised problem (possibly $O(n!)$, as there would be on the order of $n!$ permutations for an update count of n). Further research will be required to determine whether this can be improved further at the constraint solver level, whether there are further domain level optimisations to be applied or indeed whether there is a need to address this scenario at all.

Note that the bounds checking query is an example of a problem which is conventionally solved during symbolic execution at the point of memory access, i.e. before we have definitively observed every atomic memory access and are able to constrain *seen* variables. For queries which do not result in forking, such as bounds checking and data race tests, the query must be postponed until the execution barrier is reached. An open question is how to handle queries which may result in forking, such as a branch which depends on the result of an atomic variable.

```
(benchmark atomic
  :logic QF_AUFBV
  :extrafuns (
    (ord0 BitVec[8])
    (ord1 BitVec[8])
    (ord2 BitVec[8])
    (initialArray Array[8:32])
  )
  :assumption (bvult ord0 bv3[8])
  :assumption (bvult ord1 bv3[8])
  :assumption (bvult ord2 bv3[8])
  :assumption (not (= ord0 ord1))
  :assumption (not (= ord0 ord2))
  :assumption (not (= ord1 ord0))
  :assumption (not (= ord1 ord2))
  :assumption (not (= ord2 ord0))
  :assumption (not (= ord2 ord1))
  :formula
  (let (?op (store (store (store initialArray
    bv0[8] bv3[32])
    bv1[8] bv3[32])
    bv2[8] bv3[32])
  )
  (let (?ordVal0 bv1[32])
  (let (?ordVal1 (bvadd ?ordVal0 (select ?op ord0)))
  (let (?ordVal2 (bvadd ?ordVal1 (select ?op ord1)))
  (let (?seenVal (store (store (store initialArray
    ord0 ?ordVal0)
    ord1 ?ordVal1)
    ord2 ?ordVal2)
  )
    (not (bvult (select ?seenVal bv0[8]) bv8[32]))
  )))))
)
```

Listing 7.3: Unoptimised bounds checking problem involving atomic accesses in SMT-LIB format.

```
(benchmark atomic
  :logic QF_AUFBV
  :extrafuns (
    (ord0 BitVec[8])
    (ord1 BitVec[8])
    (ord2 BitVec[8])
  )
  :assumption (bvult ord0 bv3[8])
  :assumption (bvult ord1 bv3[8])
  :assumption (bvult ord2 bv3[8])
  :assumption (not (= ord0 ord1))
  :assumption (not (= ord0 ord2))
  :assumption (not (= ord1 ord0))
  :assumption (not (= ord1 ord2))
  :assumption (not (= ord2 ord0))
  :assumption (not (= ord2 ord1))
  :formula
  (let (?ordVal0 bv1[32])
    (not (bvult (bvadd (bvmul (concat bv0[24] ord0) bv3[32])
                       ?ordVal0)
                bv8[32]))
  )
)
```

Listing 7.4: Optimised bounds checking problem involving atomic accesses in SMT-LIB format.

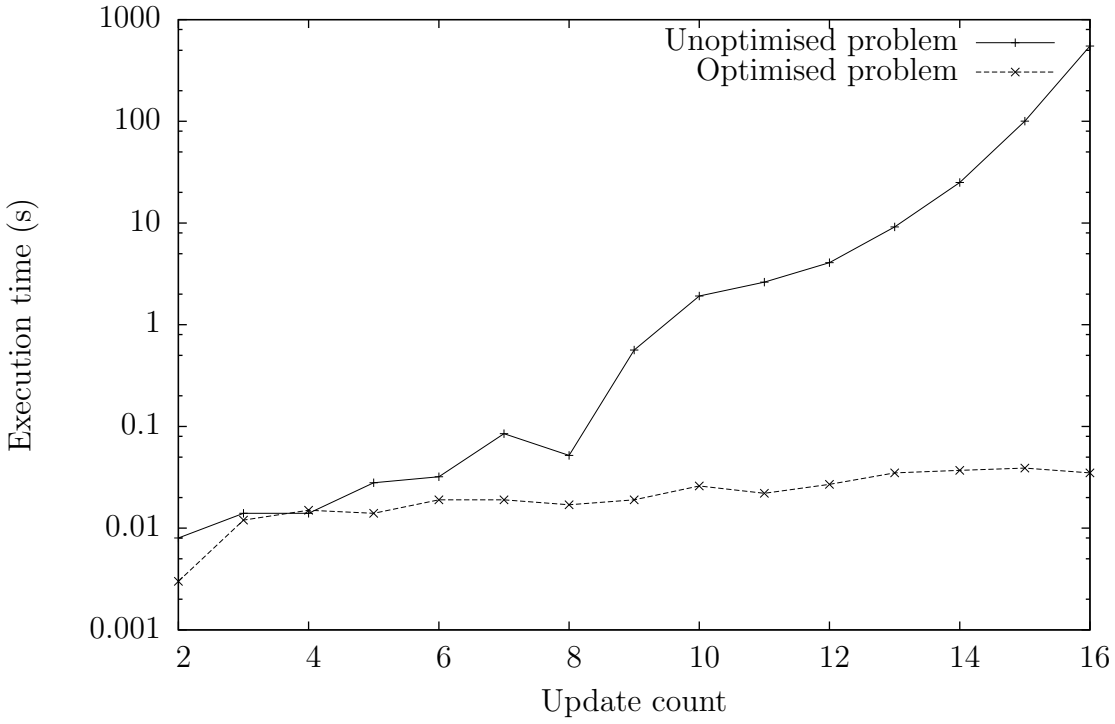

Figure 7.1: Constraint solver (STP) execution time for atomic problems.

7.6 Summary

This chapter has restated the contributions of this thesis and provided a series of possible extensions to this work, which were not considered in the course of this research due to limited applicability to the kernels we evaluated. For most of these extensions (Sections 7.1, 7.3 and 7.4), a preliminary design has been laid out, while others may require additional research (Sections 7.2 and 7.5).

Bibliography

[AAF+03] M. Aharoni, S. Asaf, L. Fournier, A. Koifman, and R. Nagel. FPgen – a test generation framework for datapath floating-point verification. In *HLDVT '03*, 2003.

[AG98] Alexander Aiken and David Gay. Barrier inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '98, pages 342–354, New York, NY, USA, 1998. ACM.

[AL10] Erika Ábrahám and Ulrich Loup. SMT-solving for the real algebra. Technical report, RWTH Aachen University, Germany, 2010.

[APV08] Saswat Anand, Corina S. Pasareanu, and Willem Visser. JPF-SE: A symbolic execution extension to Java PathFinder. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)*, pages 134–138, March-April 2008.

[AWZ88] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM Symposium on the Principles of Programming Languages (POPL'88)*, January 1988.

[BB09] Robert Brummayer and Armin Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09)*, March 2009.

[BBF02] Roberto Barbuti, Cinzia Bernardeschi, and Nicoletta De Francesco. Checking security of Java bytecode by abstract interpretation. In *SAC '02: Proceedings of the 2002 ACM symposium on Applied computing*, pages 229–236, New York, NY, USA, 2002. ACM.

[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 Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005.

[BCD+12] Adam Betts, Nathan Chong, Alastair F. Donaldson, Shaz Qadeer, and Paul Thomson. GPUVerify: a verifier for GPU kernels. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'12)*. ACM, 2012. To appear.

[BCE08] Peter Boonstoppel, Cristian Cadar, and Dawson Engler. RWset: Attacking path explosion in constraint-based test generation. In *TACAS'08* [tac08].

[BCF+07] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, Ziyad Hanna, Alexander Nadel, Amit Palti, and Roberto Sebastiani. A lazy and layered SMT(BV) solver for hard industrial verification problems. In *Proceedings of the 19th international conference on Computer aided verification*, CAV'07, pages 547–560, Berlin, Heidelberg, 2007. Springer-Verlag.

[BD02] Raik Brinkmann and Rolf Drechsler. Rtl-datapath verification using integer linear programming. In *Proceedings of the 2002 Asia and South Pacific Design Automation Conference*, ASP-DAC '02, pages 741–, Washington, DC, USA, 2002. IEEE Computer Society.

[BF07] Sylvie Boldo and Jean-Christophe Filliatre. Formal verification of floating-point programs. In *ARITH '07*, 2007.

[BGGT02] Aart J. C. Bik, Milind Girkar, Paul M. Grey, and Xinmin Tian. Automatic intra-register vectorization for the Intel® architecture. *International Journal of Parallel Programming*, 30:65–98, 2002. 10.1023/A:1014230429447.

[BGM06] Bernard Botella, Arnaud Gotlieb, and Claude Michel. Symbolic execution of floating-point computations. *Software Testing, Verification and Reliability*, 16(2):97–121, 2006.

[BHK+10] Gerard Basler, Matthew Hague, Daniel Kroening, Luke Ong, Thomas Wahl, and Haoxian Zhao. Boom: Taking Boolean program model checking one step further. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'10)*, pages 145–149, March 2010.

[BK08] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly Media, 2008.

[BKO+07] Randal E. Bryant, Daniel Kroening, Joel Ouaknine, Sanjit A. Seshia, Ofer Strichman, and Bryan Brady. Deciding bit-vector arithmetic with abstraction. In *Proceedings of TACAS 2007*, volume 4424 of *Lecture Notes in Computer Science*, pages 358–372. Springer, 2007.

[BKW09] Angelo Brillout, Daniel Kroening, and Thomas Wahl. Mixed abstractions for floating-point arithmetic. In *Proceedings of FMCAD 2009*, pages 69–76. IEEE, 2009.

[BST10] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.

[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 (STMCS'08)*, April 2008.

[BUZC11] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. Parallel symbolic execution for automated real-world software testing. In *EuroSys'11* [eur11].

[C+] Erwin Coumans et al. Bullet continuous collision detection and physics library. `http://bulletphysics.org/`. Retrieved 25 April 2012.

[CBZ10] George Candea, Stefan Bucur, and Cristian Zamfir. Automated software testing as a service. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 155–160, New York, NY, USA, 2010. ACM.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, New York, NY, USA, 1977. ACM.

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992.

[CCEH00] Andy Chou, Benjamin Chelf, Dawson Engler, and Mark Heinrich. Using meta-level compilation to check flash protocol code. *SIGPLAN Not.*, 35:59–70, November 2000.

[CCF03] Weihaw Chuang, Brad Calder, and Jeanne Ferrante. Phi-predication for light-weight if-conversion. In *CGO 2003*, March 2003.

[CCF+05] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The astreé analyzer. In Shmuel Sagiv, editor, *ESOP*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer, 2005.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson 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)*, December 2008.

[CFF+06] David Currie, Xiushan Feng, Masahiro Fujita, Alan J. Hu, Mark Kwan, and Sreeranga Rajan. Embedded software verification using symbolic execution and

uninterpreted functions. *International Journal of Parallel Programming*, 34(1):61–91, 2006.

[CGP+06] Cristian Cadar, Vijay Ganesh, Peter Pawlowski, David Dill, and Dawson Engler. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS'06)*, October-November 2006.

[CK03] Edmund Clarke and Daniel Kroening. Hardware verification using ANSI-C programs as a reference. In *Proceedings of the 8th Asia and South Pacific Design Automation Conference (ASP-DAC'03)*, January 2003.

[CK04] Edmund Clarke and Daniel Kroening. Checking consistency of C and Verilog using predicate abstraction and induction. In *In Proceedings of ICCAD*, pages 66–72. IEEE, 2004.

[CK09] Peter Collingbourne and Paul H. J. Kelly. A compile-time infrastructure for GCC using Haskell. In *GROW 2009*, 2009.

[CK10] Peter Collingbourne and Paul H. J. Kelly. Inference of session types from control flow. *Electr. Notes Theor. Comput. Sci.*, 238(6):15–40, 2010.

[CKC11] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: a platform for in-vivo multi-path analysis of software systems. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, pages 265–278, New York, NY, USA, 2011. ACM.

[CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, March-April 2004.

[CKSY05] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In Nicolas Halbwachs and Lenore

Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *Lecture Notes in Computer Science*, pages 570–574. Springer Berlin / Heidelberg, 2005.

[cla] clang: a C language family frontend for LLVM. `http://clang.llvm.org/`. Retrieved 25 April 2012.

[cpp] C++ AMP reference documentation. `http://msdn.microsoft.com/en-us/library/hh265137(v=vs.110).aspx`. Retrieved 25 April 2012.

[CZB+10] Liviu Ciortea, Cristian Zamfir, Stefan Bucur, Vitaly Chipounov, and George Candea. Cloud9: a software testing service. *SIGOPS Oper. Syst. Rev.*, 43:5–10, January 2010.

[DdM] Bruno Dutertre and Leonardo de Moura. The YICES SMT solver. `http://yices.csl.sri.com/tool-paper.pdf`. Retrieved 25 April 2012.

[DGP+09] David Delmas, Eric Goubault, Sylvie Putot, Jean Souyris, Karim Tekkal, and Franck Vdrine. Towards an industrial use of FLUCTUAT on safety-critical avionics software. In Mara Alpuente, Byron Cook, and Christophe Joubert, editors, *FMICS*, volume 5825 of *Lecture Notes in Computer Science*, pages 53–69. Springer, 2009.

[DGV04] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki – a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, LCN '04, pages 455–462, Washington, DC, USA, 2004. IEEE Computer Society.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS'08* [tac08].

[ECCH00] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th conference on Symposium on Operating System Design & Imple-*

mentation – Volume 4, OSDI'00, pages 1–1, Berkeley, CA, USA, 2000. USENIX Association.

[EQT] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Precise race detection and efficient model checking using locksets. Technical Report MSR-TR-2005-118, Microsoft Research.

[eur11] *Proceedings of the 6th European Conference on Computer Systems (EuroSys'11)*, April 2011.

[EWO04] Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. Vectorization for SIMD architectures with alignment constraints. In *Proceedings of the Conference on Programing Language Design and Implementation (PLDI'04)*, June 2004.

[FF09] Cormac Flanagan and Stephen N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *Proceedings of the Conference on Programing Language Design and Implementation (PLDI'09)*, June 2009.

[FG05] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. *ACM SIGPLAN Notices*, 40:110–121, January 2005.

[Fid88] Colin J. Fidge. Timestamps in Message-Passing Systems that Preserve the Partial Ordering. In *Proceedings of the 11th Australian Computer Science Conference*, pages 55–66, University of Queensland, Australia, 1988.

[FMWA99] Christian Ferdinand, Florian Martin, Reinhard Wilhelm, and Martin Alt. Cache behavior prediction by abstract interpretation. *Sci. Comput. Program.*, 35(2-3):163–189, 1999.

[GA11] Tobias Grosser and Raghesh Aloor. Polly – first successful optimizations – how to proceed? LLVM developers meeting 2011. `http://llvm.org/devmtg/2011-11/`, November 2011.

[GD07] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Proceedings of the 19th International Conference on Computer-Aided Verification (CAV'07)*, July 2007.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *Proceedings of the Conference on Programing Language Design and Implementation (PLDI'05)*, June 2005.

[GMS+11] Mike B. Giles, Gihan R. Mudalige, Z. Sharif, Graham R. Markall, and Paul H. J. Kelly. Performance analysis of the OP2 framework on many-core architectures. *SIGMETRICS Performance Evaluation Review*, 38(4):9–15, 2011.

[GO11] Dominik Grewe and Michael F.P. O'Boyle. A static task partitioning approach for heterogeneous systems using OpenCL. In *Proceedings of the 20th International Conference on Compiler Construction (CC'11)*, March-April 2011.

[God07] Patrice Godefroid. Compositional dynamic test generation. In *Proceedings of the 34th ACM Symposium on the Principles of Programming Languages (POPL'07)*, January 2007.

[GS97] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Proceedings of the 9th International Conference on Computer-Aided Verification (CAV'97)*, June 1997.

[GS06] Douglas Gregor and Sibylle Schupp. STLlint: lifting static checking from languages to libraries. *Software–Practice & Experience*, 36:225–254, March 2006.

[GS09] Benny Godlin and Ofer Strichman. Regression verification. In *DAC*, pages 466–471. ACM, 2009.

[GZA+11] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Grösslinger, and Louis-Noël Pouchet. Polly – polyhedral optimization in LLVM. In *First International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*, Chamonix, France, April 2011.

[Har07] John Harrison. Floating-point verification. *Journal of Universal Computer Science*, 13(5):629–638, 2007.

[HGBK12] Leopold Haller, Alberto Griggio, Martin Brain, and Daniel Kroening. Deciding floating-point logic with systematic abstraction. In *Proceedings of the 12th Formal Methods in Computer-Aided Design (FMCAD'12)*, October 2012.

[hmp] OpenHMPP website. `http://www.openhmpp.org/`. Retrieved 25 April 2012.

[Hol95] Christian Holzbaur. clp(q,r) manual rev. 1.3.2. Technical report, Austrian Research Institute for Artificial Intelligence, Vienna, 1995.

[HSS09] Trevor Hansen, Peter Schachte, and Harald Søndergaard. Runtime verification. chapter State Joining and Splitting for the Symbolic Execution of Binaries, pages 76–92. Springer-Verlag, Berlin, Heidelberg, 2009.

[IEE08] IEEE Task P754. *IEEE 754-2008, Standard for Floating-Point Arithmetic*. August 2008.

[IMP] IMPACT Research Group, UIUC. Parboil benchmark suite. `http://impact.crhc.illinois.edu/parboil.php`. Retrieved 25 April 2012.

[Int99] International Organization for Standardization. *ISO/IEC 9899-1999: Programming Languages—C*, December 1999.

[Int03] International Organization for Standardization. *ISO/IEC 14882:2003(E): Programming Languages—C++*, October 2003.

[Int10a] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A: Instruction Set Reference A-M.* Number 253666-034US. March 2010.

[Int10b] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B: Instruction Set Reference N-Z.* Number 253667-034US. March 2010.

[Int11] International Organization for Standardization. *ISO/IEC 14882:2011: Programming Languages—C++*, September 2011.

[IW] Intel and Willow Garage. OpenCV 2.1.0: Open source computer vision library. `http://opencv.willowgarage.com/`. Retrieved 25 April 2012.

[K+] Daniel Kroening et al. CBMC Subversion repository. `http://www.cprover.org/svn/cbmc/trunk/src/floatbv/`. Retrieved 25 April 2012.

[KCY03] Daniel Kroening, Edmund Clarke, and Karen Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of the 40th Design Automation Conference (DAC'03)*, June 2003.

[KGG+09] Adam Kiezun, Vijay Ganesh, Philip J. Guo, Michael D. Ernst, and Pieter Hooimeijer. HAMPI: A solver for string constraints. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'09)*, July 2009.

[Khr10] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.1, revision 36*, September 2010.

[Kin75] James C. King. A new approach to program testing. In *Proceedings of the International Conference on Reliable Software (ICRS'75)*, April 1975.

[Kin76] James C. King. Symbolic execution and program testing. *Communications of the Association for Computing Machinery (CACM)*, 19(7):385–394, July 1976.

[KLE] KLEE website. `http://klee.llvm.org`. Retrieved 25 April 2012.

[Kro12] Daniel Kroening. Personal communication, 2012.

[KY05] Amir Kamil and Katherine A. Yelick. Concurrency analysis for parallel programs with textually aligned barriers. In Eduard Ayguadé, Gerald Baumgartner, J. Ramanujam, and P. Sadayappan, editors, *LCPC*, volume 4339 of *Lecture Notes in Computer Science*, pages 185–199. Springer, 2005.

[KYKS09] KyungHee Kim, Tuba Yavuz-Kahveci, and Beverly A. Sanders. Precise data race detection in a relaxed memory model using heuristic-based model checking. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 495–499, Washington, DC, USA, 2009. IEEE Computer Society.

[LA00] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *PLDI'00* [pld00].

[LA04] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'04)*, March 2004.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21:558–565, July 1978.

[LG10] Guodong Li and Ganesh Gopalakrishnan. Scalable SMT-based verification of GPU kernel functions. In *Proceedings of the ACM Symposium on the Foundations of Software Engineering (FSE'10)*, November 2010.

[LLS+12] Guodong Li, Peng Li, Geof Sawaya, Ganesh Gopalakrishnan, Sreeranga Rajan, and Indradeep Ghosh. GKLEE: Concolic verification and test generation for GPUs. In *Proceedings of the Seventeenth ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'12)*, February 2012.

[LLV] LLVM language reference. `http://llvm.org/docs/LangRef.html`. Retrieved 4 September 2012.

[Lov10] Robert Love. *Linux Kernel Development (3rd Edition)*. July 2010.

[Mat88] Friedemann Mattern. Virtual time and global states of distributed systems. In *Proceedings of the Parallel and Distributed Algorithms Conference*, pages 215–226. North-Holland, 1988.

[Mic02] Claude Michel. Exact projection functions for floating point number constraints. In *AMAI '02*, 2002.

[MX09] Rupak Majumdar and Ru-Gang Xu. Reducing test inputs using information partitions. In *Proceedings of the 21st International Conference on Computer-Aided Verification (CAV'09)*, July 2009.

[MY59] Ramon E. Moore and C. T. Yang. Interval analysis I. Technical Document LMSD-285875, Lockheed Missiles and Space Division, Sunnyvale, CA, USA, 1959.

[NBBDZ03] Dorit Naishlos, Marina Biberstein, Shay Ben-David, and Ayal Zaks. Vectorizing for a SIMdD DSP architecture. In *CASES '03*, Oct.–Nov. 2003.

[Nec00] George C. Necula. Translation validation for an optimizing compiler. In *PLDI'00* [pld00].

[NOT06] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM*, 53(6):937–977, November 2006.

[NS05] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*. The Internet Society, 2005.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[NVI10] NVIDIA. *NVIDIA CUDA Programming Guide, Version 3.0*, February 2010.

[OC03] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *Proceedings of the Ninth ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'03)*, June 2003.

[PBBR07] Steven G. Parker, Solomon Boulos, James Bigler, and Austin Robison. RTSL: a ray tracing shading language. In *IEEE Symposium on Interactive Ray Tracing, 2007 (RT '07)*, pages 149–160, Sept 2007.

[pld00] *Proceedings of the Conference on Programing Language Design and Implementation (PLDI'00)*, May 2000.

[PM12] Matt Pharr and William R. Mark. ispc: A SPMD compiler for high-performance CPU programming. In *InPar 2012 – Innovative Parallel Computing*, 2012.

[pr1] LLVM bug database. `http://llvm.org/PR10054`. Retrieved 25 April 2012.

[Rot11] Nadav Rotem. Intel OpenCL SDK vectorizer. `http://llvm.org/devmtg/2011-11/`, November 2011.

[Rus06] John Rushby. Harnessing disruptive innovation in formal verification. In *Software Engineering and Formal Methods, 2006. SEFM 2006. Fourth IEEE International Conference on*, pages 21 –30, sept. 2006.

[RW] Philipp Rümmer and Thomas Wahl. Satisfiability modulo floating-point arithmetic. `http://www.cprover.org/SMT-LIB-Float/`. Retrieved 25 April 2012.

[RW10] Philipp Rümmer and Thomas Wahl. An SMT-LIB theory of binary floating-point arithmetic. In *Informal proceedings of 8th International Workshop on Satisfiability Modulo Theories (SMT) at FLoC, Edinburgh, Scotland*, 2010.

[SAH+10] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for JavaScript. In *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P'10)*, May 2010.

[SBN+97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. Eraser: A dynamic data race detector for multithreaded programs. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP'97)*, October 1997.

[SD08] Eric Whitman Smith and David L. Dill. Automatic formal verification of block cipher implementations. In *Proceedings of the 2nd Formal Methods in Computer-Aided Design (FMCAD'08)*, November 2008.

[SDK+11] Raimondas Sasnauskas, Oscar Soria Dustmann, Benjamin Lucien Kaminski, Carsten Weise, Stefan Kowalewski, and Klaus Wehrle. Scalable symbolic execution of distributed systems. In *Proceedings of the 31st IEEE Int'l Conference on Distributed Computing Systems, (ICDCS 2011)*, June 2011.

[SI09] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer – data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications (WBIA'09)*, December 2009.

[SLA+10] Raimondas Sasnauskas, Olaf Landsiedel, Muhammad Hamad Alizai, Carsten Weise, Stefan Kowalewski, and Klaus Wehrle. KleeNet: Discovering insidious interaction bugs in wireless sensor networks before deployment. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN 2010)*, April 2010.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'05)*, September 2005.

[SMAC06] Stephen F. Siegel, Anastasia Mironova, George S. Avrunin, and Lori A. Clarke. Using model checking with symbolic execution to verify parallel numerical programs. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'06)*, pages 157–168, July 2006.

[SMF12] Carsten Sinz, Florian Merz, and Stephan Falke. LLBMC: A bounded model checker for LLVM's intermediate representation (competition contribution). In Cormac Flanagan and Barbara König, editors, *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of*

Systems (TACAS '12), Tallinn, Estonia, volume 7214 of *Lecture Notes in Computer Science*, pages 542–544. Springer-Verlag, 2012.

[SN05] Julian Seward and Nicholas Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 2–2, Berkeley, CA, USA, 2005. USENIX Association.

[Ste74] Pat H. Sterbenz. *Floating-point computation.* Prentice-Hall series in automatic computation. Prentice-Hall, 1974.

[tac08] *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, March-April 2008.

[TJLE00] Deependra Talla, Lizy K. John, Viktor Lapinskii, and Brian L. Evans. Evaluating signal processing and multimedia applications on SIMD, VLIW and superscalar architectures. In *In Proceedings of the IEEE International Conference on Computer Design ICCD-2000*, pages 163–172, 2000.

[TSL10] Stavros Tripakis, Christos Stergiou, and Roberto Lublinerman. Checking noninterference in SPMD programs. In *Proceedings of the Second USENIX Workshop on Hot Topics in Parallelism (HotPar'10)*, June 2010.

[VBPS12] Dmitry Vyukov, Derek Bruening, Alexander Potapenko, and Konstantin Serebryany. AddressSanitizer: A fast address sanity checker. In *USENIX ATC 2012*, 2012. (to appear).

[WFBA00] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, San Diego, CA, February 2000.

[WL95] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. *SIGPLAN Not.*, 30(6):1–12, 1995.

[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 (PPoPP'11)*, February 2011.

Appendix A

Testing an OpenCL Implementation

As a cross-checking tool, KLEE-CL can be used to find the root cause of a mismatch in computation results. Such a mismatch can usually be found in user-written code, but it is also possible to use KLEE-CL as a tool to help narrow down OpenCL implementation bugs. Where KLEE-CL finds that the expressions produced by the CPU and GPU are equivalent, but the results obtained on real hardware differ, this may indicate that either KLEE-CL, the vendor's OpenCL compiler or another component of the implementation contains a bug.

KLEE-CL incorporates a *C printer*, a tool used to print a symbolic expression in the form of a C function which evaluates that expression (known as the *expression evaluator*), together with the definition of a `struct` data type used by the expression evaluator to store value bindings for subexpressions (known as the *binding record*). The C printer is capable of printing evaluators targeting both C and OpenCL.

The first parameter to the evaluator is a pointer to the binding record, and any additional parameters are pointers to symbolic variables used by the expression. For example, Listing A.1 shows the generated evaluator for the expression $2 \times x + 1$ (where x is a `char`).

To cross-check an algorithm producing erroneous results on real hardware, one can use the C printer to produce C and OpenCL C evaluators for the incorrectly evaluated expression. Then, one runs the evaluator on both the CPU and the GPU using identical values for symbolic

```
 1 struct CPbinding {
 2   uint8_t n3;
 3   int32_t n2;
 4   uint32_t n1;
 5   uint32_t n0;
 6 };
 7
 8 __kernel uint32_t CPeval(__global struct CPbinding *bindings,
 9                           __global char *x) {
10   bindings->n3 = x[((uint32_t)0UL)];
11   bindings->n2 = ((int8_t) bindings->n3);
12   bindings->n1 = ((uint32_t)2UL) *
13                  ((uint32_t) bindings->n2);
14   bindings->n0 = ((uint32_t)1UL) + bindings->n1;
15   return bindings->n0;
16 }
```

Listing A.1: C printer example output for OpenCL C.

variables but separate binding records. One can then perform a pairwise comparison of the fields of the CPU and GPU binding records. The source of the computational error can generally be identified as the computation producing the first field pair to mismatch.