Symbolic Crosschecking of Data-Parallel Floating-Point Code

Peter Collingbourne, Cristian Cadar, *Member, IEEE*, and Paul H.J. Kelly

Abstract—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 tool based on the symbolic execution engine 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 effective support for integer arithmetic, the situation is different 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 prove the bounded equivalence between scalar and data-parallel versions of floating-point programs and 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 a compiler bug.

Index Terms—Data-parallel code, floating point, symbolic execution, SIMD, OpenCL, KLEE-CL

1 Introduction

Recent years have seen the emergence of a number of programming models that improve program performance by exploiting data-level parallelism. Such models include single instruction multiple data (SIMD) and general purpose graphics processing unit (GPGPU) computing, sometimes called single instruction multiple threads (SIMT).

Today, most commercial CPU designs include SIMD capabilities, such as the Streaming SIMD Extensions (SSE), 3DNow! and Advanced Vector Extensions (AVX) for ×86; 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 [19], [38], [46], the difficulty of reasoning about data dependencies and arithmetic precision means that this translation is still a mostly manual process.

These programming models can be difficult to understand and use correctly. The Intel Instruction Set [30], [31] is a 1,674 page document describing over 400 machine instructions, over 100 of which are SIMD instructions. The OpenCL 1.1 specification [34] comprises 385 pages of technical documentation describing more than 600 individual 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. Developers cannot easily determine that their code is compliant with the OpenCL specification, because they may unknowingly be using undocumented quirks of their particular implementation.

In this paper, we present a technique for reasoning about the correctness of data-parallel optimisations involving floating point. Our technique is based on symbolic execution, 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 trying to prove their equivalence. During symbolic execution of OpenCL kernels, we also maintain a log of all memory accesses for

- *P. Collingbourne is with Google Inc. E-mail: peter@pcc.me.uk.*
- *C. Cadar and P.H.J. Kelley are with the Department of Computing, Imperial College London, London, SW7 2AZ, United Kingdom. E-mail: {c.cadar, p.kelly}@imperial.ac.uk.*

Manuscript received 5 Dec. 2012; revised 7 Oct. 2013; accepted 17 Dec. 2013. Date of publication 1 Jan. 2014; date of current version 18 July 2014. Recommended for acceptance by W. Schulte. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TSE.2013.2297120

use in race detection. In particular, this paper makes the following contributions:

1. We reason about the equivalence of floating-point code using expression matching augmented with canonicalisation rules; as far as we know, this is the first practical symbolic execution based technique that can precisely reason about the equivalence of IEEE 754 floating-point arithmetic.
2. We address the path explosion problem associated with symbolic execution by statically merging paths using phi-node folding, a form of if-conversion.
3. We present a technique for symbolically testing for the presence of concurrency errors in OpenCL kernels which (a) explores a single canonical interleaving that is likely to detect common use-after-free errors, and (b) is able to detect races associated with symbolically-indexed accesses of memory objects.

We implemented our techniques in a tool called KLEE-CL, which we make freely available as open-source. We evaluate KLEE-CL by applying it to code in the OpenCV computer vision library, the Parboil benchmark suite, the Bullet physics library and the OP2 library, and show that it can prove the bounded equivalence between scalar and data-parallel versions of floating-point code, as well as find real bugs, including memory errors, race conditions, and implementation mismatches.

We emphasise from the start that our technique is targeted toward proving the bounded equivalence of data-parallel optimisations involving floating point, and is not effective for general testing of floating-point programs. In particular, our technique is prone to false positives and is typically unable to generate concrete test cases triggering the errors found.

2 Background

This section discusses the relevant background material: we start by presenting the main concepts related to floating-point arithmetic (Section 2.1), SIMD (Section 2.2) and GPGPU (Section 2.3), and then give a brief overview of symbolic execution and the KLEE system (Section 2.4).

2.1 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 [27], 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 $-2^{1,024}$ (exclusive) and $-2^{-1,022}$ (inclusive), and between $2^{-1,022}$ (inclusive) and $2^{1,024}$ (exclusive) with 52 bits of precision [27], Section 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), NaNs (i.e., uncomputable results) and *denormalised* numbers, which are used to represent small numbers: for single precision, 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 $2^{-1,074}$ between $-2^{-1,022}$ (exclusive) and $-2^{-1,074}$ (inclusive), and between $2^{-1,074}$ (inclusive) and $2^{-1,022}$ (exclusive).

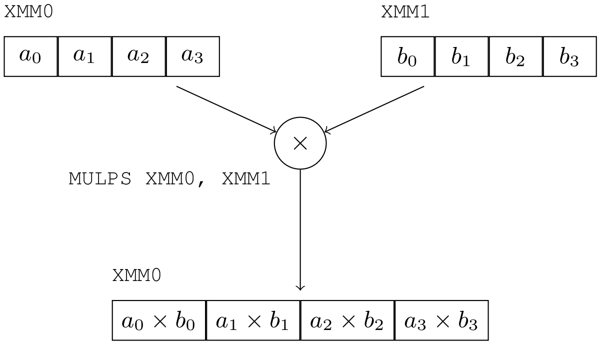


Fig. 1. The SSE MULPS instruction.

2.2 SIMD

The 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 [53], a speedup of up to 5.5× has been observed for SIMD versions of various signal processing and multimedia algorithms on a three-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. A typical SIMD instruction will take one or more input vector register operands, and perform an operation element-wise on each operand element, storing the result in an output vector register. For example, Fig. 1 shows the operation of the Intel SSE multiplication instruction MULPS.

2.3 The GPGPU Architecture and OpenCL

General-purpose graphics processing units offer a new commonly available facility for highly-parallel computing. GPGPU architectures are most commonly single program multiple data (SPMD) in nature, an evolution of the GPU's ability to perform multiple 3D rendering calculations in parallel.

Open Computing Language (OpenCL) 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.

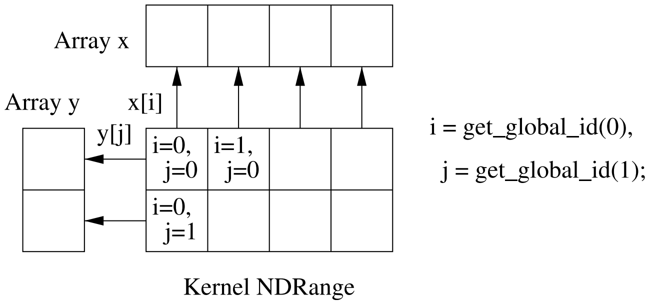


Fig. 2. Using a two-dimensional NDRange iteration space to vary the data items accessed. The `get_global_id` function returns the unique global work-item ID value for the specified dimension.

```
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 }
```

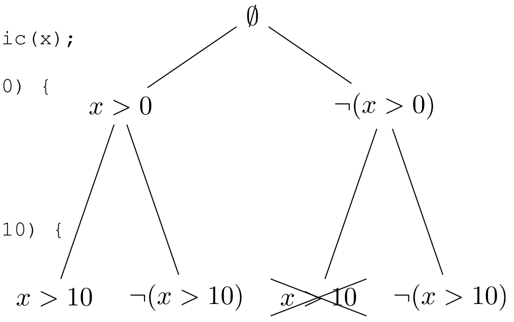


Fig. 3. Example of symbolic execution.

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 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. Fig. 2, where each square represents a work-item, shows an example of how work-item functions can be used for this purpose.

2.4 Symbolic Execution and KLEE

At a high level, symbolic execution [35] 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*.

Fig. 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 condition $x > 0$. 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 by the two implementations are equivalent (Section 6), and (3) the implementations are race free (Section 7).

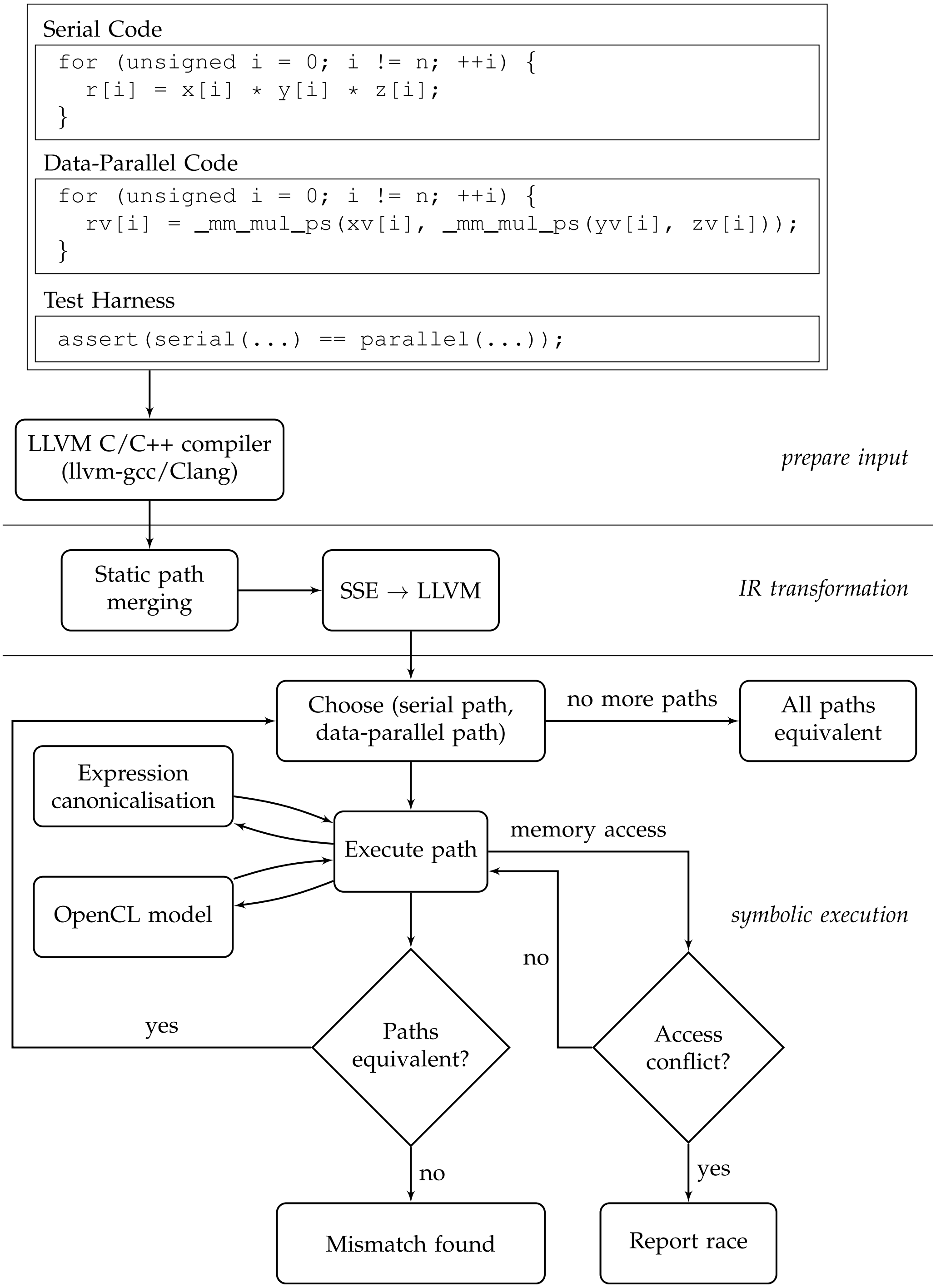
One fundamental limitation of symbolic execution is that it only handles objects of fixed size (i.e., each data structure in a program usually has to be assigned a concrete size, as in the normal execution of the program). For OpenCL, this means that the number of work-items must be concrete; in a typical OpenCL program, the number of work-items (i.e., the size of the NDRange) depends on the size of the input being processed. As a result, we can only 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 [11], an open-source symbolic execution engine that operates on programs in the LLVM [39] intermediate representation (IR) format. LLVM is a static single assignment based IR used by a number of compilers, including the Clang compiler [13], which features robust support for a number of C family languages including C, C++ and OpenCL C, making it a practical base for a symbolic execution tool.

3 Overview of KLEE-CL

In this section 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 does 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 [27], 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



Serial Code

```
for (unsigned i = 0; i != n; ++i) {
  r[i] = x[i] * y[i] * z[i];
}
```

Data-Parallel Code

```
for (unsigned i = 0; i != n; ++i) {
  rv[i] = _mm_mul_ps(xv[i], _mm_mul_ps(yv[i], zv[i]));
}
```

Test Harness

```
assert(serial(...) == parallel(...));
```

Fig. 4. Architecture diagram for KLEE-CL.

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 (Sections 4.2 and 4.3), and the semantics for a substantial portion of OpenCL is implemented using a symbolic OpenCL model implementing both host-side and device-side functionality (Section 4.4). We improve the tractability of our technique by implementing an aggressive variant of if-conversion using phi-node folding [12], [39], to replace control-flow forking with predicated `select` instructions (which has similar semantics to the C `?:` operator), in order to reduce the number of paths explored by symbolic execution (Section 5).

3.1 Design and Architecture

Crosschecking a data-parallel routine against its serial equivalent using our technique involves three main stages, as illustrated graphically in Fig. 4.

```
 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 }
```

Fig. 5. Simple test benchmark.

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), 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 feasible paths (potentially including some infeasible ones, see below) in the code under test, while also checking for race conditions (Section 7). In order to be able to reason about OpenCL code, our technique implements a symbolic OpenCL model (Section 4.4). Then, on each explored path, we try to prove that the symbolic expressions corresponding to the serial and data-parallel 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 wresulting expressions are equivalent (Section 6).

The fact that KLEE-CL can only reason about the *equivalence* of floating-point expressions means that we cannot always determine the feasibility of each side of a symbolic branch. When this happens, our approach is to *always* fork, potentially exploring infeasible paths. This could result in false positives (as mismatches on infeasible paths are irrelevant) but no false negatives (we only claim bounded equivalence when we can prove equivalence on all explored paths, which include all feasible paths up to the given input size).

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 Fig. 5, 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 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 data-parallel 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 2.4 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 data-parallel 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 Section 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 its floating-point operands are the same, as distinguished from a bitwise comparison.

[1] Specifically `thresh(BINARY_INV`, f32); see Section 8.1.

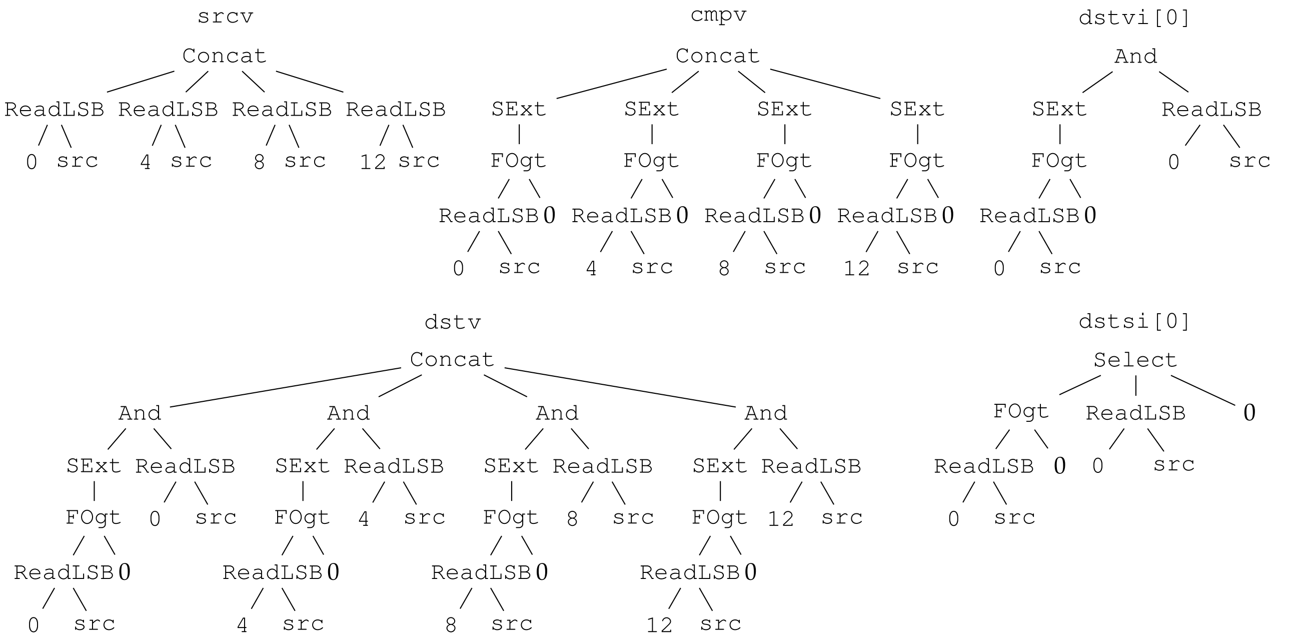


Fig. 6. Symbolic expressions assigned to variables `srcv`, `cmpv`, `dstv` and to the array elements `dstvi[0]` and `dstsi[0]` of Fig. 5. `src` represents the symbolic array `src`. The `ReadLSB` (read least significant byte first) node represents a 4-byte little-endian array read, `FOgt` floating-point greater-than comparison, `SExt` sign extension, `Select` the equivalent of the C ternary operator and `Concat` bitwise concatenation.

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. Section 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 Fig. 5 are represented by the three expressions shown on the left-hand side of Fig. 6. Similarly, the results `dstvi[0]` and `dstsi[0]` are represented by the two expressions shown on the right side of Fig. 6.

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 6.2, one of our canonicalisation rules (#14 in Table 3) 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 Fig. 6 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.

One concern not covered by this simple example, which has a single execution path, is the number of proofs that are needed: during symbolic execution, every feasible program path (and some infeasible ones) 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.

4 Modelling Data-Parallel Operations

This section 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. Finally, in Sections 4.4 and 4.5 we discuss our approach to modelling the OpenCL environment and 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

TABLE 1
Floating-Point Predicate Shorthand Semantics False and True Are Always Simplified

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

themselves derived from the semantics defined in the IEEE 754-2008 standard [27]. 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 expressions are untyped bitvectors, type information is associated with operations, rather than operands.

Of particular importance for our crosschecking algorithm 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 [32] are required to provide ordered relational operators, except for `!=`, which is unordered $\neq$ [27, Section 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 $\mathbf{O}$: e.g., 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 $\mathbf{O}$, known as the *outcome set* (represented internally using a vector of four bits, based on the floating-point predicate representation used by LLVM [39]). In this paper, we normally refer to predicate operations using shorthand names rather than using `FCmp`. Table 1 gives a list of mappings between shorthand names and `FCmp` operations. In Section 6.2 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). The Clang compiler that we use to compile OpenCL C kernels will add this metadata to single precision floating-point division operations.[2] 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. 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 6), 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.

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

[2] 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.

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 1,024 bits.

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 [11] by using the *bitvector* data type provided by its underlying constraint solver, STP [21]. 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 two lower order elements taken from the two lower order elements of `%a` and its two higher order elements from the two 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.

We added support for the 37 SSE intrinsics, which were sufficient to handle the benchmarks with which we evaluated our technique (Section 8). An example of a call to an SSE-specific intrinsic is shown below:

```
%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 OpenCL C Environment

Our OpenCL model is a partial implementation of the Khronos OpenCL 1.1 Specification [34], which has been developed to meet the needs of a wide range of OpenCL client programs, including those we evaluated (Section 8). 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 OpenCL C environment, which models the execution of a kernel on the device, and which we discuss in this section, and the runtime library, which is used by the host to manage the execution of OpenCL kernels, and which will be discussed in the next section.

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 [10]. 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.

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 (MARs) associated with the data race detector (explained in further detail in Section 7).

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 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 [[34], Section 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.[3] 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.

4.5 The OpenCL Runtime Library

The OpenCL runtime library is specified by two sections of the OpenCL specification: the OpenCL Platform Layer [34, Section 4], and the OpenCL Runtime [[34], Section 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; we discuss some of the more interesting ones below.

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.

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 7), 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.

[3] 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.

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.

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 [13] compiler. Clang is designed to be used as a library, which made it easy to integrate into KLEE-CL.

LLVM modules built using Clang are then dynamically loaded by KLEE-CL. To implement this, we added dynamic multiple LLVM module support to KLEE-CL, which 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`.

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. The `klee_lookup_module_global` discussed above was used to implement this function.

The `clEnqueueNDRangeKernel` function discussed in Section 2.3 is implemented by creating a local thread group for each work-group and the local and global wait lists mentioned in Section 4.4, 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.

```
 1  cl_mem buf1 = clCreateBuffer(context, ...);
 2  cl_mem buf2 = clCreateBuffer(context, ...);
 3  clEnqueueWriteBuffer(cmd_queue, buf1,
 4                   /*blocking_write*/ CL_TRUE, ...);
 5  clSetKernelArg(kernel, 0, sizeof(cl_mem), &buf1);
 6  clSetKernelArg(kernel, 1, sizeof(cl_mem), &buf2);
 7  clEnqueueNDRangeKernel(cmd_queue, kernel, ...);
 8  clReleaseMemObject(buf1);
 9  clEnqueueReadBuffer(cmd_queue, buf2,
10                   /*blocking_read*/ CL_TRUE, ...);
11  ...
12  __kernel void k(__global int *buf1,
13                  __global int *buf2) {
14    buf2[get_global_id(0)] =
15      buf1[get_global_id(0)]*2;
16  }
```

Fig. 7. OpenCL code containing a use-after-free error.

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 the next section, we discuss the consequences of this with regard to memory error detection.

4.6 Detecting Use-After-Free Errors in OpenCL Code

While use-after-free errors are traditionally detected by examining memory usage in a serial fashion, OpenCL C introduces a more insidious class of such errors, due to the asynchronous nature of kernel invocation. For example, consider the host program fragment shown in Fig. 7 (simplified from our Parboil `mri-q` benchmark, see Section 8.4). 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 5 and 6 we set the first and second arguments to kernel `k` (shown on lines 12-16) to `buf1` and `buf2`, and on line 7 we enqueue an invocation of `k`, a kernel which accesses memory from both of its arguments. Then on line 8, 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 8, and a read of `buf1` in `k`, such as that on line 15, would result in a memory error. (We can fix the error by simply swapping lines 8 and 9.)

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.

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`

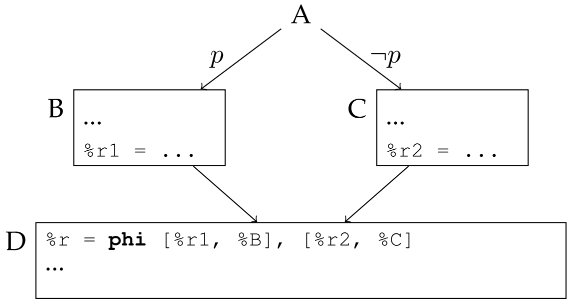


Fig. 8. Diamond control flow pattern.

TABLE 2
Phi Node Folding Instruction Costs

Instruction	Cost	Unsafe?
`Load, GetElementPtr, Add, Sub, And, Or, Xor, Shl, LShr, AShr, ICmp, Trunc, ZExt, SExt`	1	
`Select`	2	
`FAdd, FSub, FMul, FDiv, FAdd, FCmp`	1	✓

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, 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. 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.

5 Static Path Merging

With the data-parallel modeling of Section 4, KLEE-CL is ready to start executing the program symbolically. One limitation of symbolic execution is that the number of paths in a program is exponential in the number of symbolic branches encountered during execution. The worst case for data-parallel programs is that a symbolic 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 may often prevent the (bounded) verification of the correctness of data-parallel optimisations in a practical amount of time even for small input sizes.

To alleviate this problem, we have designed a static path merging pass that we apply before symbolic execution takes place. We have found this pass to be highly effective in our equivalence checking context (leading to a dramatic reduction in the number of explored paths on several benchmarks), although in general the pass can lead to more complex constraints that can put excessive strain on the constraint solver.

Our merging pass is an aggressive variant of *phi-node folding*, also known as *if-conversion* [12], [39]. Phi-node folding usually operates on the static single-assignment (SSA) form of a program [3] and targets branches with a control-flow structure matching the diamond pattern shown in Fig. 8, 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 to minimise the amount of unnecessary work done by the CPU.

Since KLEE-CL does not model floating-point exceptions, the behaviour of code which speculates evaluation of the sides of an if statement involving floating-point expressions is modeled identically to code which branches. Therefore it is always safe to fold floating-point instructions in KLEE-CL. Furthermore, due to forking, the cost of not applying the optimisation in the context of symbolically executing data-parallel code is usually greater than that of applying it.

As a result, we have adapted phi-node folding to aggressively merge paths when we encounter the diamond pattern shown in Fig. 8. Our implementation is built on top of LLVM's `SimplifyCFG` pass, which contained a default implementation of phi node folding. The existing pass was highly conservative in that it was only triggered if each of the basic blocks B and C contained at most one computation instruction.

We enhanced the pass in two ways. First, 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 2). Second, we added an option to enable *unsafe phi*

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

Fig. 9. Serial reduction.

node folding, which allowed the optimisation to be applied to floating-point instructions despite the side effects described above.[4] In our experiments (Section 8) we set a very high cost threshold (1,000) and enable unsafe phi-node folding. Practically, this allows the optimisation to be applied in all possible circumstances.

To evaluate our static path merging technique, we performed two types of experiments. First, we measured the number of times it was applied to an LLVM bitcode file containing the entire OpenCV library (a popular computer vision library discussed in Section 8). Second, we measured the number of execution paths explored for our OpenCV benchmarks both with and without phi-node folding enabled.

Using 1,000 as the threshold value and enabling unsafe folding, the optimisation was applied 2,321 times. Of course, not all these merged locations are executed by our benchmarks, or are executed only with concrete inputs. Therefore, this data is best interpreted together with dynamic measurements which reveal the reduction in number of executed paths. For each of our OpenCV benchmarks, we measured the number of paths explored 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 on the number of elements in the input image. For these benchmarks, our technique was 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.

6 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 data-parallel 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 to bring 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 6.2.

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, constructing two equivalent values from the same inputs in floating point can usually only be done reliably in a limited number of ways.

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

  size_t d    = 1;
  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];
}
```

Fig. 10. OpenCL parallel reduction.

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 6.3.

6.1 Assumptions

In floating-point arithmetic, it is unsound to perform certain expression simplifications that are valid under ordinary real number arithmetic. For example, it is unsound 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 in parallel programming.

For example, a reduction operation in a serial program is typically computed using an $O(n)$ `for` loop which maintains an accumulated result based on the data elements processed thus far, an example of which is shown in Fig. 9. This implementation technique is inefficient for a data-parallel program, as it does not permit the exploitation of parallel resources. By exploiting 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 Fig. 10. When executing this code symbolically using KLEE-CL, we may obtain expressions of the form:

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

[4] Our modifications to `SimplifyCFG` were subsequently contributed back to LLVM, and have turned out to have potential applications outside of symbolic execution; for example, LLVM developers have considered adjusting the default threshold to allow additional optimisations to be applied [42].

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, which can be enabled via individual command line arguments:

1. The *positive zero* assumption allows the simplifier to disregard the difference between positive and negative zero, which is usually inconsequential.
2. The *finite* assumption allows the simplifier to assume all results are finite (i.e., not $\pm\infty$ or `NaN`).
3. The *ordered* assumption allows the simplifier to assume all results are ordered (i.e., not `NaN`).
4. The *associativity* assumption allows the simplifier to assume that floating-point operations are associative.

Each of these assumptions is implemented through additional expression simplification rules which are enabled by the assumption. Further details on each simplification rules are found in the next section.

6.2 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 3 lists the main rewrite rules we implemented. The first 10 are specifically targeted toward floating-point expressions, while the next eight are applicable to both floating-point and integer ones. We note that not all rules preserve floating-point exception behaviour. For example, rule 19 may postpone or even eliminate an exception if `X` is `NaN`; such rules are typically applied by compilers, but if a stricter form of equivalence is required, they should be disabled:

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, `Or`(`FOlt`(X, Y),`FOeq`(X, Y)) simplifies to `FOle`(X, Y). Rules 4-6 implement similar simplifications, making use of the swap function defined below:

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

2. *Category analysis.* Category analysis, a simplified form of interval analysis [45], 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 $\mathrm{cat}(x) \subseteq \mathbf{C}$ of an expression x is defined as the set of categories the expression x may be in. We define $\mathrm{cat}(x)$ recursively based on the category sets of subexpressions of x. For example, if $+ \in \mathrm{cat}(x)$ and $+ \in \mathrm{cat}(y)$ then $\{+, +\infty\} \subseteq \mathrm{cat}(x + y)$. 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 $\mathrm{cat}(x) = \{0, -\}$ and $\mathrm{cat}(y) = \{0, +\}$ 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 vector:

```
__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 loaded here 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 $W = \mathrm{width}(X)$. 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, 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, `Concat`$(11,$ `Concat`$(00, X))$ gets simplified to `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 $W = 2$, $N = 0$, $f = \texttt{Or}$ and $Y = 1,100$, 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 × operators by re-arranging right associative operations into left associative ones. This rule is only applied if the associativity assumption is enabled.

6.3 Building Implied Constraints

If the matching algorithm fails to prove expression equivalence, we try to use rewritten expressions that are implied by the original constraints. In this way, 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. For example, one important way in which we use this idea is in handling expressions of the form FPToSI(X) and FPToUI(X) (conversion from floating point to integer). Each expression of this form 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. After the substitution, we can use our constraint solver STP to determine if the rewritten integer expressions are equivalent. If this is the case, then we know the original expressions are also equivalent. However, if the constraint solver cannot prove the equivalence, the mismatch could be a false positive.

Another example are chains of `min` and `max` operations. Our technique recognises the expression idiom representing the floating-point `min` and `max` operations:

$$\begin{aligned} \texttt{min}(X, Y) &= \texttt{Select}(\texttt{FOlt}(X, Y), X, Y) \\ \texttt{max}(X, Y) &= \texttt{Select}(\texttt{FOlt}(Y, X), X, Y) \end{aligned}$$

and attempts 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:

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

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.

While the FPToSI/FPToUI and the min/max examples below provide a feel for the type of implied constraints that we generate, we formalize the process below, for the more technical reader. Each constraint C in the constraint set is replaced with $rw(C)$, where rw is defined in Fig. 11 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,

$$
\begin{aligned}
rw(E) &= rw'(E, \bot) \\[1em]
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} \\[1em]
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 \\ \quad \texttt{And}(ce(X_0, Y_1), ce(Y_0, X_1))) & \quad O \cap \{<, >\} = \{<, >\} \\ \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} \\
\begin{array}{r} ce(S_0@\texttt{Select}(P_0, X_0, Y_0), \\ S_1@\texttt{Select}(P_1, X_1, Y_1)) \end{array} &= \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{aligned}
$$

Fig. 11. The rw rewriting function, and its helper functions rw' and ce.

$E \rightarrow rw'(E, \bot)$ and $rw'(E, \top) \rightarrow E$. ce builds an expression such that for any pair of KLEE-CL expressions X, Y, $ce(X, Y) \rightarrow X = Y$. 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, as discussed at the beginning of the section. 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 in Fig. 12 (each function application or expression simplification step is shown on a separate line).

6.4 Integrating a Precise Floating-Point Solver

Because our equivalence checking algorithm 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. (However,

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)

Fig. 12. Application of rw to expression $\texttt{Eq}(\texttt{Eq}(\texttt{FAdd}(X,Y),\texttt{FAdd}(Y,X)),\texttt{false})$.

remember that KLEE-CL has no false negatives, i.e., when it says that two expressions are equivalent, this is guaranteed to be true.)

To better understand the nature of these false positives, we integrated a bit-precise floating-point constraint solver recently made available in CBMC [36]. We report in Section 8.7 our experience using this solver to resolve some of the mismatches reported by KLEE-CL, as well as discuss its performance limitations.

7 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.

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, 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.

To detect data races, we keep for each byte in the generic and group-local address spaces a *memory access record* 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 barrier (*many-read*),
   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 six 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 4.4), 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`.

Read

$$\mathit{write}[\mathit{index}] \wedge (\mathit{wg\text{-}id}[\mathit{index}] \neq \mathit{wg\text{-}id} \vee (\mathit{thread\text{-}id}[\mathit{index}] \neq 0 \wedge \mathit{thread\text{-}id}[\mathit{index}] \neq \mathit{thread\text{-}id}))$$

$$\begin{aligned}
\mathit{many\text{-}read}[\mathit{index}] &\leftarrow \mathit{many\text{-}read}[\mathit{index}] \vee (\mathit{read}[\mathit{index}] \wedge \mathit{thread\text{-}id}[\mathit{index}] \neq 0\ \wedge \\
&\qquad \mathit{thread\text{-}id}[\mathit{index}] \neq \mathit{thread\text{-}id}) \\
\mathit{wg\text{-}many\text{-}read}[\mathit{index}] &\leftarrow \mathit{wg\text{-}many\text{-}read}[\mathit{index}] \vee (\mathit{read}[\mathit{index}] \wedge \mathit{wg\text{-}id}[\mathit{index}] \neq \mathit{wg\text{-}id}) \\
\mathit{thread\text{-}id}[\mathit{index}] &\leftarrow \mathit{thread\text{-}id} \\
\mathit{wg\text{-}id}[\mathit{index}] &\leftarrow \mathit{wg\text{-}id} \\
\mathit{read}[\mathit{index}] &\leftarrow \top
\end{aligned}$$

Write

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

$$\begin{aligned}
\mathit{thread\text{-}id}[\mathit{index}] &\leftarrow \mathit{thread\text{-}id} \\
\mathit{wg\text{-}id}[\mathit{index}] &\leftarrow \mathit{wg\text{-}id} \\
\mathit{write}[\mathit{index}] &\leftarrow \top
\end{aligned}$$

Fig. 13. Race condition test and MAR updates.

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.

7.1 Race Condition Test and MAR Updates

The race condition test, together with the required MAR updates, are shown in Fig. 13. 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 ($\mathit{thread\text{-}id}[\mathit{index}]$, $\mathit{wg\text{-}id}[\mathit{index}]$) pair for a given array index *index* will be in one of three states:

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), $\mathit{read}[\mathit{index}]$ and/or $\mathit{write}[\mathit{index}]$ may be set, but in case (1), neither $\mathit{read}[\mathit{index}]$ nor $\mathit{write}[\mathit{index}]$ will be set.

The first conjunct of the race condition test for reads is $\mathit{write}[\mathit{index}]$. This excludes case (1), as required. The second conjunct is $\mathit{wg\text{-}id}[\mathit{index}] \neq \mathit{wg\text{-}id} \vee (\mathit{thread\text{-}id}[\mathit{index}] \neq 0 \wedge \mathit{thread\text{-}id}[\mathit{index}] \neq \mathit{thread\text{-}id})$. For case (2), $\mathit{wg\text{-}id}[\mathit{index}] \neq \mathit{wg\text{-}id}$ will hold in the case where the work-group identifier differs from the stored work-group identifier, and $\mathit{thread\text{-}id}[\mathit{index}] \neq 0 \wedge \mathit{thread\text{-}id}[\mathit{index}] \neq \mathit{thread\text{-}id}$ does not hold because $\mathit{thread\text{-}id}[\mathit{index}] \neq 0$ 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), $\mathit{thread\text{-}id}[\mathit{index}] \neq \mathit{thread\text{-}id}$ will hold in the case where the work-item identifier differs from the stored work-item, and $\mathit{thread\text{-}id}[\mathit{index}] \neq 0$ always holds by definition. If the work-group identifiers differ then the work-item identifiers will also differ, so $\mathit{wg\text{-}id}[\mathit{index}] \neq \mathit{wg\text{-}id}$ 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 $\mathit{many\text{-}read}[\mathit{index}]$ 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 $\mathit{thread\text{-}id}[\mathit{index}] \neq \mathit{thread\text{-}id}$. However, it is not true in case (2) because of the intervening execution barrier, nor is it true in case (1), hence the conjunct $\mathit{thread\text{-}id}[\mathit{index}] \neq 0$. $\mathit{many\text{-}read}[\mathit{index}]$ remains set until the execution barrier for that work-group, hence the disjunct $\mathit{many\text{-}read}[\mathit{index}]$. The value of $\mathit{many\text{-}read}[\mathit{index}]$ is indeterminate if multiple work-groups have accessed the memory location—in such a case, the value of $\mathit{many\text{-}read}[\mathit{index}]$ at any program point depends on the scheduling of the work-group relative to other work-groups because it uses $\mathit{thread\text{-}id}[\mathit{index}]$, which may be set and cleared by other

```
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	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	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	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	Con
1	1	✓			
1	1	✓			
2	1	✓		✓	
2	1	✓		✓	
2	1	✓		✓	
1	1	✓	✓		
1	1	✓	✓		

Fig. 14. 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, and Con (if present) the nature of the conflict detected at that line. The WMR (wg-many-read) flag is not shown as it is never set.

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 $\mathit{wg\text{-}many\text{-}read}[\mathit{index}]$ 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 $\mathit{wg\text{-}id}[\mathit{index}] \neq \mathit{wg\text{-}id}$.

The race condition test for writes uses three disjuncts. The first two, $\mathit{many\text{-}read}[\mathit{index}]$ and $\mathit{wg\text{-}many\text{-}read}[\mathit{index}]$, 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 ($\mathit{many\text{-}read}[\mathit{index}]$) or (in the case where reads are performed by multiple work-groups) multiple reads by multiple work-groups ($\mathit{wg\text{-}many\text{-}read}[\mathit{index}]$). Recall that $\mathit{many\text{-}read}[\mathit{index}]$ is indeterminate in the latter case—because $\mathit{wg\text{-}many\text{-}read}[\mathit{index}]$ 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 $\mathit{read}[\mathit{index}] \vee \mathit{write}[\mathit{index}]$ is used, because writes conflict with both earlier writes and earlier reads.

7.2 Examples

To illustrate the race detection technique described above, we use the code in Fig. 14. 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 (i.e., for a[0]) 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.

Thus, in the case of avg, we start by running work-item 1 on the entire kernel. The first four lines do not access a[0], so no flags are set. Line 5 reads a[0], and so the read flag is set. Line 6 does not access a[0] so the flags remain unchanged. Then, line 7 writes to a[0] so the write flag is set.

Then, work-item 2 starts executing with the read and write flags set. Lines 1–3 do not access a[0], so the flags remain unchanged. Line 4 reads a[0] so 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 when line 4 is executed by work-item 2, work-item 1 had not executed line 8, because it was preempted by the barrier on line 7.

Our second example, shown in Fig. 15, 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

```
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
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

Fig. 15. Intermediate MARs for a[2] during execution of work-items 1 and 2.

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.

8 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 [9], [29], the Parboil benchmark suite [28], the Bullet physics library [17] and the OP2 library [22]. KLEE-CL was configured to use KLEE's default strategy, which interleaves, in a round-robin fashion, random path search with a heuristic that tries to minimise the distance to uncovered code [11].

8.1 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.

TABLE 3
Symbolic Expression Canonicalisation Rules Where Necessary, Bitwidths of Expressions are Denoted by Superscripts

#	Condition/Assumption	Expression	Result	Section
1	-	$\texttt{And}(\texttt{FCmp}(X,Y,O_1),\texttt{FCmp}(X,Y,O_2))$	$\texttt{FCmp}(X,Y,O_1 \cap O_2)$	§6.2(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		§6.2(2)
8	C constant, see §6.2(3)	$\texttt{FOeq}(\texttt{SIToFP}(X),C)$	$\texttt{Eq}(X,\texttt{FPToSI}(C))$	§6.2(3)
9	C constant, see §6.2(3)	$\texttt{FOeq}(\texttt{UIToFP}(X),C)$	$\texttt{Eq}(X,\texttt{FPToUI}(C))$	
10	$f \in \{\texttt{FPToSI},\texttt{FPToUI}\}$	$f(\texttt{FPExt}(X))$	$f(X)$	§6.2(4)
11	C_1, C_2 constants	$\texttt{Concat}(C_1,\texttt{Concat}(C_2,X))$	$\texttt{Concat}(\texttt{Concat}(C_1,C_2),X)$	§6.2(5)
12	-	Partial constant folding with equality		§6.2(6)
13	-	$\texttt{ZExt}(X)$	$\texttt{Concat}(0,X)$	§6.2(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	$f \in \{\texttt{Or},\texttt{And},\texttt{Xor}\}$, $\text{width}(X_0) = \text{width}(X_1)$	$f(\texttt{Concat}(X_0,Y_0),\texttt{Concat}(X_1,Y_1))$	$\texttt{Concat}(f(X_0,X_1),f(Y_0,Y_1))$	§6.2(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	§6.2(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*	$\texttt{FAdd}(X,\texttt{FAdd}(Y,Z))$	$\texttt{FAdd}(\texttt{FAdd}(X,Y),Z)$	§6.2(10)
26	*associativity*	$\texttt{FMul}(X,\texttt{FMul}(Y,Z))$	$\texttt{FMul}(\texttt{FMul}(X,Y),Z)$	

TABLE 4
OpenCV Code We Tested with KLEE-CL

Source File (src/)	Benchmarks	#	Cov
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%

The third column shows the number 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). Coverage data refers to coverage of SIMD instructions.

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. Out of the 20 OpenCV source code files containing SSE code, we arbitrarily selected 10 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 Fig. 5.

Table 4 presents the 10 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 [9]. 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 erodealgorithm, 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 2.4), 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 percent. 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 percent coverage because we weren't able to run it at all. We discuss the reasons in Section 8.7.

We constructed a total of 58 benchmarks to cover the functions in these 10 files. KLEE-CL was able to successfully verify 41 benchmarks up to a certain image size (Section 8.2) and find mismatches in 10 benchmarks (Section 8.3). In addition, three benchmarks triggered false positives (Section 8.7(3)) and four benchmarks couldn't be run at all by KLEE-CL (Section 8.7(4)).

8.2 OpenCV Benchmarks Verified up to a Certain Image Size

Table 5 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.

The transff, transsf and transcf benchmarks use fixed size matrices. The .43 variants take a three-channel source array of size 4×4 and a one-channel transformation matrix of size 3×4 and produce a three-channel array of size 4×4, while the .44 variants take a four-channel source array of size 4×4 and a one-channel transformation matrix of size 4×4 and produce a four-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 6.3). 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 CBMC solver 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

TABLE 5
OpenCV Benchmarks Verified Up to a Certain Size

#	Bench	Algo	K	Format	Maximum Size
1	morph	dilate	R	u8	5 × 5
2				s16	16 × 16
3				u16	16 × 16
4			NR	u8	8 × 3
5				s16	16 × 16
6				u16	16 × 16
7				f32	15 × 15
8		erode	R	u8	4 × 4
9				s16	16 × 16
10				u16	16 × 16
11			NR	s16	16 × 16
12				u16	16 × 16
13	pyramid			u8	8 × 2 → 4 × 1
14	remap	nearest neighbor		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 × 8 → 8 × 8
27				u16	4 × 1 → 8 × 2
28				f32	8 × 8 → 8 × 8
29		cubic		s16	8 × 8 → 8 × 8
30				u16	4 × 1 → 8 × 2
31				f32	8 × 8 → 8 × 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 §8.2
43	transff.44			f32	See §8.2

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 (the cx1 cluster described at http://www3.imperial.ac.uk/ict/services/hpc/highperformancecomputing/services/hardware), 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 1 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 21.1 percent of benchmarks had cumulative execution times of under 10 minutes, 34.2 percent between 10 minutes and one hour, 18.4 percent between 1 and 12 hours, and 26.3 percent over 12 hours.

8.3 Invalidated OpenCV Benchmarks

Table 6 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 [32], Section 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.

TABLE 6
OpenCV Benchmarks in Which We Found Mismatches between the Scalar and the SSE Versions

#	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 × 2 → 8 × 1	Associativity, distributivity
8	resize	linear		u8	4 × 4 → 8 × 8	Precision
9		cubic		u8	4 × 1 → 8 × 2	Integer/FP differences
10	transsf.43			s16 f32	See §8.2	Rounding issue
11	transcf.43			u8 f32	See §8.2	Integer/FP differences

The IEEE floating-point operations $+$ and $\times$ 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 [32], Section 6.5.5], the scalar multiplication is implicitly bracketed as $(((\texttt{float})\texttt{k})* (\texttt{a}+\texttt{c}))*(\texttt{a}+\texttt{c})$, 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 6.3, neither associative nor commutative.

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

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

   The STL functions `std::min` and `std::max` used by the scalar variants of the benchmarks are not required by the C++ 2003 standard [33] 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 [33], Section 25.3] in the presence of `NaN`s). The GNU STL implements them idiomatically, but with the operand order reversed:

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

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 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):

$$(((1536 \times N_0) + (512 \times N_0)) + 2097152) \gg 22,$$

   whereas the SIMD variant produces expressions of the form:

$$\begin{aligned}(2 + (((1536 \times (N_0 \gg 4)) \gg 16)\\ + ((512 \times (N_0 \gg 4)) \gg 16))) \gg 2.\end{aligned}$$

   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 CBMC floating-point solver 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 32,768 from the floating-point number, performing a conversion directly to a 16-bit signed integer and adding 32,768 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 32,768 from the floating-point value, as in the SIMD variant of this code, ϵ will be lost during rounding and the result is $-32{,}767.5$. When this value is converted to an integer, it is rounded *down* to $-32{,}768$ (under this rounding mode, ties are rounded to the nearest even value), and the result is 0 after adding 32,768 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 10 mismatches listed in Table 6. 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 discussed in Section 6.1, and made it apply to min/max rewrite rules presented in Section 6.3. With this in place, 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 haven't investigated yet.

8.4 OpenCL Acceleration in Parboil

Parboil [28] is a popular GPU benchmark suite, which contains C and CUDA [48] implementations of various algorithms. In order to be able to run Parboil benchmarks using KLEE-CL, we used Grewe and O'Boyle [24] translation of certain Parboil 1 benchmarks from CUDA to OpenCL. The translation comprised four benchmarks in total, and we tested three of these: Coulombic Potential (`cp`), Magnetic Resonance Imaging - Q (`mri-q`) and Magnetic Resonance Imaging - FHD (`mri-fhd`). We were unable to test the fourth benchmark, Rys Polynomial Equation Solver (`rpes`) for reasons discussed in Section 8.7. The three benchmarks have 66 (`cp`), 118 (`mri-fhd`) and respectively 90 (`mri-q`) lines of code, based on a line count of each of the .cl files that are compiled.

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 Fig. 5 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 [32], 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 6 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 4.6, 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`.

8.5 OpenCL Acceleration in Bullet

Bullet [17] 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. The two benchmarks share the same OpenCL code, which consists of 1,187 lines.

We used our benchmark programs to test SVN revision 2,357 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. To diagnose this issue, 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 ([34], Section 9.3.9] requires double precision division to be correctly rounded). With this change, KLEE-CL claimed that the two pieces of code were equivalent.

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.

```
 1 int tid = get_local_id(0),
 2     d = get_local_size(0)>>1;
 3 __local volatile float *vtemp = temp;
 4 ...
 5 for (; d>0; d>>=1) { /* d is at most 16 here */
 6   if (tid<d) {
 7     ...
 8     vtemp[tid] = vtemp[tid] + vtemp[tid+d];
 9     ...
10   }
11 }
```

Fig. 16. OP2's unsynchronised loop (slightly modified for formatting purposes).

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.

8.6 OpenCL Acceleration in OP2

OP2 [22] 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 transformations 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. The benchmark (the `.cl` file) has 75 lines of code.

KLEE-CL detected a race condition in this kernel, and the problematic code is shown in Fig. 16. Each iteration of the `for` loop on lines 5-11 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 [48]. 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 5 and 6). With this modification in place, KLEE-CL does not report a race condition.

8.7 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 3.
2. *Manual effort*: To use our tool, developers have to write a test harness, similar to the one implemented by the `main()` function in Fig. 5. 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. *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 ($>30{,}000$) even for small problems, which KLEE-CL could not execute in a reasonable amount of time.
4. *Experience using a bit-precise FP solver.* As discussed in Section 6.4, we integrated into KLEE-CL a bit-precise floating-point solver recently made available in CBMC. Using this solver, we were able to verify two out of the three suspected false positives in the OpenCV benchmarks and produce a counterexample for the other one.

   There are two important drawbacks of using such a solver by itself. The first one is that without support for quantifiers, it is not possible to add assumptions (Section 6.1), so the benchmarks which rely on floating-point assumptions cannot be verified. The second drawback is performance. While we were able to verify some of the benchmarks not relying on assumptions using the CBMC solver by itself, some of them (e.g., `harris` and `eigenval`) exceeded our one-hour timeout (on an Intel Core i7-3667U at 2 GHz with 8 GB of RAM). As another example, it took over 46 minutes to verify one of the false positives mentioned above for only the small image sizes given (on a dual core Intel Core 2 Duo E6850 at 3 GHz with 8 GB of RAM).

   Instead, we believe that in the context of verifying the correctness of optimisations that involve floating point, bit-precise floating-point solvers should be used in conjunction with our technique: our symbolic expression transformation and rewriting algorithm should be applied first, with a floating-point solver being used to validate any (non-obvious) mismatches.

9 Related Work

Previous work on formally verifying floating-point programs has used theorem proving [5], [25], constraint solving based on approximation with rationals or reals [26] and symbolic execution using projection functions over floating-point intervals [7], [44]. While promising, these techniques have only been shown to work on very small hand-crafted programs.

An alternative to formal verification is testing. For example, [1] uses randomized testing coupled with coverage requirements to test floating-point programs. Random testing can easily be applied to large applications, but misses corner-case bugs that are common in floating-point programs.

Our approach of using symbolic execution combined with Expression matching and canonicalisation rules have been successfully used in the past to verify code equivalence, e.g., in the context of hardware verification [14], embedded software [18], compiler optimizations [47], block cipher implementations [52], and parallel numerical programs [51]. We extend this line of research in the context of SIMD vectorizations and GPGPU optimisations, which requires techniques for handling floating-point arithmetic, the modelling of SIMD and OpenCL operations (which includes support for the OpenCL runtime, OpenCL address spaces, and runtime compilation of kernels), as well as support for concurrency and race detection.

Our application of phi-node folding [12], [39] aims at reducing the state space explored by symbolic execution by statically merging paths, in the context of data-parallel code. Recent work in the area [37] has investigated more sophisticated strategies for merging paths in symbolic execution in a more general context. Alternative approaches that we could apply to reduce the number of paths explored include using compositional dynamic test generation to create function summaries [23], using read-write sets to track the values accessed by the program [6], or using information partitions to track information flow between inputs [43].

Automatic vectorization techniques provide an alternative to verifying the correctness of manually-written SIMD code [19], [38], [46]. However, even as these techniques will start to be more widely adopted, the approach presented in this paper can be applied to verify these automatically generated SIMD-vectorizations.

Despite the growing popularity of GPU languages, there has been relatively little work on testing and verification techniques for code written in these languages. Most previous work in this space has focused on race detection. Li et al. [40], Tripakis et al. [54] and Betts et al. [4] propose static race-detection techniques based on translating CUDA or OpenCL code into SMT constraints. 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 a fixed number of threads and objects with concrete bounds. On the other hand, static analysis suffers from false positives, due to various approximations resulting from, e.g., analysing kernels in isolation and loop unrolling.

Aiken and Gay [2] 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.

GKLEE [41], developed concurrently and independently with our work, 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.

A dynamic race detection approach similar to our technique is introduced by Boyer et al. in the context of CUDA programs [8]. A more recent technique [55] 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.

Our approach is also similar in spirit to previous dynamic race detection approaches for CPU code [20], [49], [50], although the barrier-based synchronisation model used in OpenCL allows for a simpler algorithm than in the case of traditional synchronisation primitives such as locks and semaphores.

10 Conclusion and Future Work

Manually translating scalar code into an equivalent data-parallel version is a difficult task, because any programming error may cause the hand-optimised code to act differently from the original scalar version. In this paper, we introduced an effective technique for symbolic crosschecking of data-parallel floating-point code, and for detecting race conditions in OpenCL code. We implemented our approach in the KLEE-CL tool, and applied it to several real code bases, in which it was able to check the bounded equivalence between scalar and data-parallel code, as well as detect memory errors, semantic mismatches and data races in the optimised code.

KLEE-CL is freely available from our website at http://www.pcc.me.uk/~peter/klee-cl/.

Acknowledgments

The authors would like to thank Alastair Donaldson, Dan Liew, Hristina Palikareva, the TSE editor Wolfram Schulte and the anonymous reviewers for their valuable feedback on their research and the paper. This work has been supported in part by the UK EPSRC through grants EP/I012036/1 and EP/I006761/1. This paper is partly based on a paper published in the *Proceedings of the European Conference on Computer Systems (EuroSys 2011)* [16] and on a paper published in the *Proceedings of the Haifa Verification Conference (HVC 2011)* [15].

References

[1] M. Aharoni, S. Asaf, L. Fournier, A. Koifman, and R. Nagel, "FPgen—A Test Generation Framework for Datapath Floating-Point Verification," *Proc. Eighth IEEE Int'l High-Level Design Validation and Test Workshop (HLDVT '03)*, 2003.

[2] A. Aiken and D. Gay, "Barrier Inference," *Proc. 25th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL '98)*, pp. 342-354, http://doi.acm.org/10.1145/268946.268974, 1998.

[3] B. Alpern, M.N. Wegman, and F.K. Zadeck, "Detecting Equality of Variables in Programs," *Proc. 15th ACM Symp. Principles of Programming Languages (POPL '88)*, Jan. 1988.

[4] A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson, "GPUVerify: A Verifier for GPU Kernels," *Proc. 27th Ann. Conf. Object-Oriented Programming Systems, Languages and Applications (OOPSLA '12)*, Oct. 2012.

[5] S. Boldo and J.-C. Filliatre, "Formal Verification of Floating-Point Programs," *Proc. 18th IEEE Symp. Computer Arithmetic (ARITH '07)*, 2007.

[6] P. Boonstoppel, C. Cadar, and D. Engler, "RWset: Attacking Path Explosion in Constraint-Based Test Generation," *Proc. Int'l Conf. Tools and Algorithms for the Constructions and Analysis of Systems (TACAS'08)*, Mar./Apr. 2008.

[7] B. Botella, A. Gotlieb, and C. Michel, "Symbolic Execution of Floating-Point Computations," *Software Testing, Verification and Reliability*, vol. 16, no. 2, pp. 97-121, 2006.

[8] M. Boyer, K. Skadron, and W. Weimer, "Automated Dynamic Analysis of CUDA Programs," *Proc. Third Workshop Software Tools for MultiCore Systems (STMCS '08)*, Apr. 2008.

[9] G. Bradski and A. Kaehler, *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly Media, 2008.

[10] S. Bucur, V. Ureche, C. Zamfir, and G. Candea, "Parallel Symbolic Execution for Automated Real-World Software Testing," *Proc. Sixth Conf. Computer Systems (EuroSys '11)*, Apr. 2011.

[11] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," *Proc. Eighth USENIX Symp. Operating Systems Design and Implementation (OSDI '08)*, Dec. 2008.

[12] W. Chuang, B. Calder, and J. Ferrante, "Phi-Predication for Light-Weight If-Conversion," *Proc. Int'l Symp. Code Generation and Optimization: Feedback-Directed and Runtime Optimization (CGO '03)*, Mar. 2003.

[13] "Clang: A C Language Family Frontend for LLVM," http://clang.llvm.org/, Apr. 2012.

[14] E. Clarke and D. Kroening, "Hardware Verification Using ANSI-C Programs as a Reference," *Proc. Eighth Asia and South Pacific Design Automation Conf. (ASP-DAC '03)*, Jan. 2003.

[15] P. Collingbourne, C. Cadar, and P.H.J. Kelly, "Symbolic Testing of OpenCL Code," *Proc. Haifa Verification Conf. (HVC '11)*, 2011.

[16] P. Collingbourne, C. Cadar, and P.H. Kelly, "Symbolic Crosschecking of Floating-Point and SIMD Code," *Proc. Sixth Conf. Computer Systems (EuroSys '11)*, Apr. 2011.

[17] E. Coumans et al., "Bullet Continuous Collision Detection and Physics Library," http://bulletphysics.org/, Apr. 2012..

[18] D. Currie, X. Feng, M. Fujita, A.J. Hu, M. Kwan, and S. Rajan, "Embedded Software Verification Using Symbolic Execution and Uninterpreted Functions," *Int'l J. Parallel Programming*, vol. 34, no. 1, pp. 61-91, 2006.

[19] A.E. Eichenberger, P. Wu, and K. O'Brien, "Vectorization for SIMD Architectures with Alignment Constraints," *Proc. Conf. Programing Language Design and Implementation (PLDI '04)*, June 2004.

[20] C. Flanagan and S.N. Freund, "Fasttrack: Efficient and Precise Dynamic Race Detection," *Proc. Conf. Programing Language Design and Implementation (PLDI '09)*, June 2009.

[21] V. Ganesh and D.L. Dill, "A Decision Procedure for Bit-Vectors and Arrays," *Proc. 19th Int'l Conf. Computer-Aided Verification (CAV '07)*, July 2007.

[22] M.B. Giles, G.R. Mudalige, Z. Sharif, G.R. Markall, and P.H.J. Kelly, "Performance Analysis of the OP2 Framework on Many-Core Architectures," *ACM SIGMETRICS Performance Evaluation Rev.*, vol. 38, no. 4, pp. 9-15, 2011.

[23] P. Godefroid, "Compositional Dynamic Test Generation," *Proc. 34th ACM Symp. Principles of Programming Languages (POPL '07)*, Jan. 2007.

[24] D. Grewe and M.F. O'Boyle, "A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL," *Proc. 20th Int'l Conf. Compiler Construction (CC '11)*, Mar./Apr. 2011.

[25] J. Harrison, "Floating-Point Verification," *J. Universal Computer Science*, vol. 13, no. 5, pp. 629-638, 2007.

[26] C. Holzbaur, "clp(q,r) Manual Rev. 1.3.2," technical report, Austrian Research Inst. for Artificial Intelligence, Vienna, 1995.

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

[28] IMPACT Research Group, UIUC, "Parboil Benchmark Suite," http://impact.crhc.illinois.edu/parboil.php, Apr. 2012.

[29] Intel and Willow Garage, "OpenCV 2.1.0: Open Source Computer Vision Library," http://opencv.willowgarage.com/. Apr. 2012.

[30] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2A: Instruction Set Reference A-M*, no. 253666-034US, Mar. 2010.

[31] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2B: Instruction Set Reference N-Z*, no. 253667-034US, Mar. 2010.

[32] *ISO/IEC 9899-1999: Programming Languages—C*, Int'l Organization for Standardization, Dec. 1999.

[33] *ISO/IEC 14882:2003(E): Programming Languages—C++*, Int'l Organization for Standardization, Oct. 2003.

[34] Khronos OpenCL Working Group, *The OpenCL Specification, Version 1.1, Revision 36*, http://khronos.org/registry/cl/specs/opencl-1.1.pdf, Sept. 2010.

[35] J.C. King, "A New Approach to Program Testing," *Proc. Int'l Conf. Reliable Software (ICRS '75)*, Apr. 1975.

[36] D. Kroening et al., "CBMC Subversion Repository," http://www.cprover.org/svn/cbmc/trunk/src/floatbv/, Apr. 2012.

[37] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient State Merging in Symbolic Execution," *Proc. Conf. Programming Language Design and Implementation (PLDI '12)*, June 2012.

[38] S. Larsen and S. Amarasinghe, "Exploiting Superword Level Parallelism with Multimedia Instruction Sets," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI '00)*, May 2000.

[39] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," *Proc. Int'l Symp. Code Generation and Optimization (CGO '04)*, Mar. 2004.

[40] G. Li and G. Gopalakrishnan, "Scalable SMT-Based Verification of GPU Kernel Functions," *Proc. ACM Symp. Foundations of Software Eng. (FSE '10)*, Nov. 2010.

[41] G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, S. Rajan, and I. Ghosh, "GKLEE: Concolic Verification and Test Generation for GPUs," *Proc. 17th ACM Symp. Principles and Practice of Parallel Programming (PPoPP '12)*, Feb. 2012.

[42] http://llvm.org/PR10054, Apr. 2012.

[43] R. Majumdar and R.-G. Xu, "Reducing Test Inputs Using Information Partitions," *Proc. 21st Int'l Conf. Computer-Aided Verification (CAV '09)*, July 2009.

[44] C. Michel, "Exact Projection Functions for Floating Point Number Constraints," *Proc. Seventh Int'l Symp. Artificial Intelligence and Mathematics (AMAI '02)*, Jan. 2002.

[45] R.E. Moore and C.T. Yang, "Interval Analysis I," Technical Document LMSD-285875, Lockheed Missiles and Space Division, Sunnyvale, CA, 1959.

[46] D. Naishlos, M. Biberstein, S. Ben David, and A. Zaks, "Vectorizing for a SIMdD DSP Architecture," *Proc. Int'l Conf. Compilers, Architecture and Synthesis for Embedded Systems (CASES '03)*, Oct./Nov. 2003.

[47] G.C. Necula, "Translation Validation for an Optimizing Compiler," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI '00)*, May 2000.

[48] NVIDIA, *NVIDIA CUDA Programming Guide, Version 3.0*, http://www.nvidia.com/, Feb. 2010..

[49] R. O'Callahan and J.-D. Choi, "Hybrid Dynamic Data Race Detection," *Proc. Ninth ACM Symp. Principles and Practice of Parallel Programming (PPoPP '03)*, June 2003.

[50] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T.E. Anderson, "Eraser: A Dynamic Data Race Detector for Multithreaded Programs," *Proc. 16th ACM Symp. Operating Systems Principles (SOSP '97)*, Oct. 1997.

[51] S.F. Siegel, A. Mironova, G.S. Avrunin, and L.A. Clarke, "Using Model Checking with Symbolic Execution to Verify Parallel Numerical Programs," *Proc. Int'l Symp. Software Testing and Analysis (ISSTA '06)*, July 2006.

[52] E.W. Smith and D.L. Dill, "Automatic Formal Verification of Block Cipher Implementations," *Proc. Second Formal Methods in Computer-Aided Design (FMCAD '08)*, Nov. 2008.

[53] D. Talla, L.K. John, V. Lapinskii, and B.L. Evans, "Evaluating Signal Processing and Multimedia Applications on SIMD, VLIW and Superscalar Architectures," *Proc. IEEE Int'l Conf. Computer Design (ICCD '00)*, pp. 163-172, 2000.

[54] S. Tripakis, C. Stergiou, and R. Lublinerman, "Checking Non-Interference in SPMD Programs," *Proc. Second USENIX Workshop Hot Topics in Parallelism (HotPar '10)*, June 2010.

[55] M. Zheng, V.T. Ravi, F. Qin, and G. Agrawal, "GRace: A Low-Overhead Mechanism for Detecting Data Races in GPU Programs," *Proc. 16th ACM Symp. Principles and Practice of Parallel Programming (PPoPP '11)*, Feb. 2011.

Peter Collingbourne received the MEng and PhD degrees in computing from Imperial College London in 2007 and 2013, respectively. He is currently a software engineer at Google.

Cristian Cadar received the undergraduate and master's degrees from the Massachusetts Institute of Technology, and the PhD degree in computer science from Stanford University. He leads the Software Reliability Group at Imperial College London. His research interests span the areas of software engineering, computer systems and security, with an emphasis on building practical tools for improving the reliability and security of software systems. He is a member of the IEEE.

Paul H.J. Kelly received the graduate degree in computer science from University College London, and the PhD degree from Westfield College, University of London. He leads the Software Performance Optimisation group at Imperial College London. His research contributions span single-address-space operating systems, scalable large shared-memory architectures, compilers (bounds checking and pointer analysis), graph algorithms, domain-specific languages, particularly for computational science, and compilation for performance-portability across diverse parallel architectures.