Behavioral Non-portability in Scientific Numeric Computing

Yijia Gu[1](✉), Thomas Wahl[1], Mahsa Bayati[2](✉), and Miriam Leeser[2]

[1] College of Computer and Information Science,
Northeastern University, Boston, USA
{guyijia,wahl}@ccs.neu.edu

[2] Department of Electrical and Computer Engineering,
Northeastern University, Boston, USA
{mbayati,mel}@coe.neu.edu

Abstract. The precise semantics of floating-point arithmetic programs depends on the execution platform, including the compiler and the target hardware. Platform dependencies are particularly pronounced for arithmetic-intensive parallel numeric programs and infringe on the highly desirable goal of software portability (which is nonetheless promised by heterogeneous computing frameworks like OpenCL): the same program run on the same inputs on different platforms often produces different results. Serious doubts on the portability of numeric applications arise when these differences are *behavioral*, i.e. when they lead to changes in the control flow of a program. In this paper we present an algorithm that takes a numeric procedure and determines an input that may lead to different *branching decisions* depending on how the arithmetic in the procedure is compiled. We illustrate the algorithm on a diverse set of examples, characteristic of scientific numeric computing, where control flow divergence actually occurs across different execution platforms.

1 Introduction

Many high performance computing applications make use of floating-point arithmetic. It is well known that floating-point expressions produce different results on different machines, due to lack of associativity, etc. Most practitioners assume that this affects only the last few bits of a computation, and can safely be ignored. In this paper, we present several examples where code run on different platforms on the *same* inputs can produce not just different results but different *control flow*. We use OpenCL as a programming language [9], which promises cross-platform portability. Yet, as our experiments show, this promise does in fact not guarantee portability of control flow.

All the computer hardware that we target complies with the IEEE 754–2008 standard [5]. The code generated to run on compliant hardware has several degrees of freedom. A compiler may reorder expressions, which affects the numerical values of the results. An example of such reordering is the use of *reductions*

Work supported by the US National Science Foundation grant CCF-1218075.

J.L. Träff et al. (Eds.): Euro-Par 2015, LNCS 9233, pp. 558–569, 2015.
DOI: 10.1007/978-3-662-48096-0_43

to compute long sums by parallel threads. Further, the IEEE standard permits the use of a fused multiply-add (FMA) instruction (which contracts a multiplication followed by an addition into a singly-rounded operation) but gives no guidance on how a compiler should employ such an instruction (if it exists at all). In most applications, there are several *different*, *IEEE-compliant* ways for the compiler to implement an expression using FMA.

Using examples characteristic of floating-point computations in parallel computing, we demonstrate in this paper that the above vagaries of IEEE floating point can impact control flow in a platform-dependent way. We present an algorithm that, given a numeric procedure, determines an input that may lead to different branching decisions due to (fully IEEE compliant) expression reordering or the use/non-use of FMA instructions. Our two-stage algorithm first uses *symbolic execution* to determine inputs that make a given branching decision unreliable. It then examines such inputs for different ways of evaluating expressions.

Motivating Example: Ray Tracing. Consider the following `C` program, taken from http://www.cc.gatech.edu/~phlosoft/photon/. We have elided the code following the branching decision `D > 0`, since here we are merely interested in whether that code is executed at all, depending on the execution platform.

```
float dot3(float *a, float *b) {
return a[0] * b[0] + a[1] * b[1] + a[2] * b[2]; }

int raySphere(float *r, float *s, float radiusSq) {
float A = dot3(r,r);
float B = -2.0 * dot3(s,r);
float C = dot3(s,s) - radiusSq;
float D = B*B - 4*A*C;
if (D > 0)
  ...; }
```

This code employs well-known high school arithmetic. When "arithmetic" means floating point, however, the results are no longer so obvious: the computation of vector dot products in `dot3`, common in high performance libraries, depends on the compiler's choice to evaluate the sum left to right or vice versa, and whether to use FMA instructions and in which of several possible ways. For certain inputs, these choices translate into platform-dependent control flows across the `if` statement involving D. Such divergence is likely not accounted for and undesirable. In Sect. 2 we describe an algorithm to find such inputs. Our algorithm determines that, given the following inputs (trailing 0's omitted) to procedure `raySphere`, the NVIDIA Quadro 600 GPU computes a value $D_N \approx -3.56$, while an Intel 64-bit CPU computes $D_I \approx 4.55$; we observe $D_N \ll 0 \ll D_I$:

$$r = (-33.999900817871094, -54.0, -53.0); \quad \text{radiusSq} = 0.000000029802322;$$
$$s = (-33.370471954345703, -53.0, -52.01855468750).$$

Related Work. The scientific computing community has long been aware that floating point vagaries can affect a computation's output. Shewchuk shows how a near-zero determinant may cause flat-out incorrect results due to floating-point rounding errors [10]. Several works have demonstrated that rounding errors can cause the control flow of a floating-point program to differ from that of the corresponding idealistic real-arithmetic program [2,6]. Our technique is distinct in both motivational and technical aspects: (i) we compare the control flows on different floating-point platforms; (ii) instead of a purely dynamic (testing) approach, we use SMT technology to locate potentially problematic inputs.

2 Finding Inputs Witnessing Behavioral Non-portability

Motivated by the observations made in Sect. 1, the goal now is an algorithm that determines inputs to the given program that are likely to expose behavioral non-portability when the program is run on certain diverse execution platforms.

2.1 Problem Formulation

Behavioral non-portability is frequently caused by expressions whose floating-point semantics is dependent on the evaluation order, and on the use of hardware features such as fused multiply-add. We call such expressions *volatile* in this paper. These are expressions $<ve>$ defined by the following grammar:

$$\begin{array}{ll}
<ve> & :: <ve_{\oplus}> \mid <ve_{\otimes}> \mid <ve_{dot}> \mid <ve_{fma}> \\
<ve_{\oplus}> & :: <e> \oplus \ \ldots \ \oplus <e> \\
<ve_{\otimes}> & :: <e> \otimes \ \ldots \ \otimes <e> \\
<ve_{dot}> & :: <e> \otimes <e> \oplus \ \ldots \ \oplus <e> \otimes <e> \\
<ve_{fma}> & :: fma(<e>, <e>, <e>) \\
<e> & :: c \mid var \mid <ve> \mid <e> < op > <e> \\
<op> & :: \oplus \mid \ominus \mid \otimes \mid \oslash
\end{array}$$

where c is a floating-point constant and var is a floating-point program variable. The semantics of $fma(x, y, z)$ is the term $(x \cdot y) \oplus z$, which represents the real value $x \cdot y + z$ followed by a single rounding step. Volatile expressions are *unparenthesized* sums, products, dot products, or FMA expressions over floating-point constants, variables, and other expressions.

Branch Point. We are interested in the effect of behavioral non-portability on the program control flow and, therefore, on conditional statements such as `if` statements and loops. We call such statements *branch points*. Each branch point refers to a conditional q, which is a Boolean-valued formula over atomic floating-point subformulas ψ of the (normalized) form

$$\psi :: X \rhd c, \quad \rhd \in \{\leq, \geq, >, <, ==\} \tag{1}$$

where c is a floating-point constant and X a floating-point valued expression.

We can now define the concept we are investigating in this paper. The value of the conditional q depends not only on the program input I, but also on platform parameters such as the availability of FMA and decisions made by the compiler about evaluating volatile expressions. We refer to an instantiation of such platform parameters as an *expression evaluation model* M. For example, a particular expression evaluation model might state that there is no FMA support, and that sums and products are evaluated left-to-right. An expression evaluation model therefore disambiguates among many of the common and IEEE-754 compliant ways expressions can be rendered by the compiler.

Let $q(I, M)$ denote the value of expression q for program inputs I (that cause q to be reached) and expression evaluation model M, and consider a program P.

Definition 1 (Control-Flow Instability). *Let q denote a Boolean-valued expression used as a conditional in program P. Input I is said to cause* ***control-flow instability*** *if there exist two evaluation models M_1 and M_2 such that*

$$q(I, M_1) \neq q(I, M_2). \tag{2}$$

Intuitively, input I is a candidate for causing program P to exhibit different control flows on different platforms, caused by different Boolean values of the conditional q, for the same input I, computed on those platforms. Control flow instabilities are likely undesirable. In the rest of this paper we describe an algorithm that, given program P and a branch point with conditional q, determines whether there exists an input that renders the control flow unstable. The algorithm *efficiently* searches through all possible inputs and evaluation models with the goal of finding I, M_1, M_2 such that $q(I, M_1)$ and $q(I, M_2)$ differ.

2.2 Detecting Behavioral Non-portability: Overview

Our algorithm for finding behavioral non-portability proceeds in two phases. Given is a program P with volatile expressions and a branch point, identified by the user to be of interest, with conditional q over atomic subformulas of the form $\psi_i{::}X_i \rhd c_i$ (1). The first phase determines a *candidate input*, i.e. an input I_0 such that minor variations of I_0 cause q to flip. Numerically, this requires that there exists ψ_k in q such that X_k is close to c_k for input I_0. This phase is implemented using *symbolic execution*: we build a formula for the path leading from the program entry point to the conditional q. We change the comparison operator in q to an equality $=$ (or an approximate equality, see below) and solve the obtained path formula using a constraint solver.

In the second phase, the algorithm computes the *minimum and maximum* value of X_k for input I_0 and under all possible expression evaluation models. For two models that give rise to the minimum and maximum value, the chances are that the value of X_k is on either side of c_k, causing the conditional q to flip. The algorithmic challenge is to search among all these models efficiently.

Figure 1 shows our overall approach. We explain the details of each step in the following subsections.

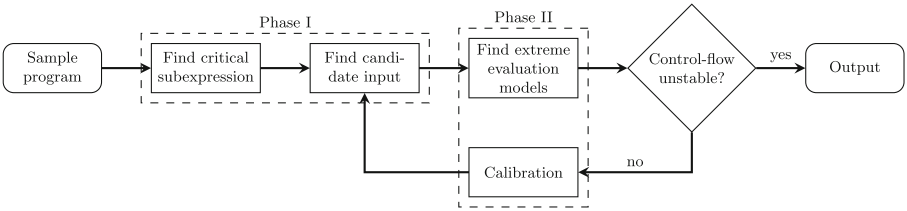

Fig. 1. Finding behavioral non-portability: overall approach

2.3 Phase I: Finding Candidate Input

Recall the form $\psi_i{::}X_i \rhd c_i$ (1) of the n atomic subformulas of q. Let $b_1, \ldots, b_n$ be fresh Boolean variables and $\overline{q}$ be the *Boolean skeleton* of q, i.e. the formula obtained from q by replacing each ψ_i by b_i. For a Boolean assignment $A : \{b_1, \ldots, b_n\} \rightarrow \{0, 1\}$, let $A|_{b_i \rightarrow v}$ denote A except that b_i is assigned value v.

Finding Critical Subexpression. A prerequisite for finding a candidate input is to identify a *critical subexpression* ψ_k in the conditional q: an index k and a Boolean assignment A_0 such that, under that assignment, flipping the value of ψ_k flips the value of q. We formalize this condition via the skeleton:

$$\overline{q}(A_0) \neq \overline{q}(A_0|_{b_k \rightarrow \neg b_k(A_0)}) \tag{3}$$

Algorithm 2.1 finds such an index k and a satisfying assignment A_0. In line 2, we use a SAT solver to check whether $\overline{q}(\ldots, b_k, \ldots) \veebar \overline{q}(\ldots, \neg b_k, \ldots)$ is satisfiable; $\veebar$ denotes exclusive-or.

Algorithm 2.1. Finding Critical Subexpression

```
   Input: Boolean formula $\overline{q}(b_1, \ldots, b_n)$
1  for $k$ = 1 to $n$ do
2      if $\overline{q}(\ldots, b_k, \ldots) \veebar \overline{q}(\ldots, \neg b_k, \ldots)$ satisfiable then
3          return index $k$ and sat. assignment
```

Finding Candidate Input. In this step we generate, from the original C code, constraints whose solutions serve as possible candidate inputs I_0. We split the generated constraints into the following parts:

1. Path Constraint ϕ_{path}: this part symbolically encodes the execution path from the program entry point to the conditional q, following the appropriate branches at all intermediate program branch points.
2. Boolean Assignment Constraint ϕ_{assgn}: input I_0 must assign to all subexpressions ψ_i, for $i \neq k$, the same Boolean value as A_0 (determined in (3)) to b_i, and this should hold independently of the expression evaluation model M:

$$\forall M \; \forall i : \; i \neq k \Rightarrow \psi_i(I_0, M) = b_i(A_0) \tag{4}$$

Solving this constraint is very costly, not least due to the limited support for floating-point arithmetic in automated solvers. We therefore interpret (4) and other arithmetic constraints in this section over the *reals*. As a result, ϕ_{assgn} simplifies to

$$\forall i: \ i \neq k \Rightarrow \psi_i(I_0) = b_i(A_0) \tag{5}$$

since real arithmetic results do not depend on the evaluation model. A problem is of course that we lose precision: real results may not hold in floating-point arithmetic. However, the goal in Phase I is merely to determine an input I_0 that brings the conditional q *close* to the tipping point. The numerical differences caused by the interpretation of the code over $\mathbb{R}$ instead of over floating-point arithmetic will not affect this goal, as long as they are small. The solution will be made precise in Phase II of the algorithm.

3. Approximation Constraint ϕ_{appr}: the critical subexpression $\psi_k : X_k \rhd c_k$ must be unreliable, i.e.:

$$X_k(I_0) = c_k \pm \epsilon, \tag{6}$$

where ϵ is the smallest non-negative number that permits a solution to (6).

The total constraint for candidate inputs is then $\phi_{path} \wedge \phi_{assgn} \wedge \phi_{appr}$. As an example, given the `raySphere` program from Sect. 1 with the volatile expression $D = B \otimes B \ominus 4 \otimes A \otimes C$ and the conditional $D > 0$, we generate the *real-arithmetic* constraint $B \cdot B - 4 \cdot A \cdot C = 0$ and pass it to a suitable decision procedure for candidate input generation.

2.4 Phase II (a): Efficiently Searching Evaluation Models

Finding Extreme Evaluation Models. Given the candidate input I_0 that results in the conditional q to be unreliable, we now determine the minimum and maximum value of X, under any possible expression evaluation model that may reorder expressions and use FMA instructions. We first introduce algorithms $getMin_T$ and $getMax_T$ ($T \in \{\oplus, \otimes, dot, fma\}$) that compute these extreme values *in polynomial time* when X is a basic volatile expression. Later we extend these algorithms to handle general expressions.

Minimizing Volatile Sum: Given volatile expression $ve_\oplus = v_1 \oplus \ldots \oplus v_n$, where the v_i ($1 \leq i \leq n$) are floating-point constants, the goal is to efficiently determine $\min_M ve_\oplus(\boldsymbol{v}, M)$, i.e. $ve_\oplus$ minimized over all evaluation models M. To this end, consider the following array, for $1 \leq i \leq j \leq n$:

$$N[i,j] = \begin{cases} v_i & \text{if } i = j \\ \min\limits_{i \leq k < j} \{N[i,k] \oplus N[k+1,j]\} & \text{if } i < j \end{cases} \tag{7}$$

We now claim that $N[1,n]$ is the quantity we are looking for:

Theorem 1. *$N[1,n] = \min_M ve_\oplus(\boldsymbol{v}, M)$.*

In order to prove this theorem, we strengthen it as follows:

Lemma 2. *$N[i,j]$ equals the minimum, over all possible orderings, of the floating-point sum of the numbers in the range $v_i, \ldots, v_j$.*

Proof. We induct over the quantity $j - i$. If $j - i = 0$, then $N[i,j] = v_i = v_j$, and the claim follows since there is only one element.

For the inductive step, assume that for all i', j' such that $j' - i' < j - i$, $N[i',j']$ is the minimum value of the sum of the elements $v_i', \ldots, v_j'$ (IH). Let k be one of the values that, in the definition of $N[i,j]$, gives rise to the minimum, i.e. $N[i,j] = N[i,k] \oplus N[k+1,j]$. Let further $N_o[i,j]$ be the sum for any fixed order o. We show that $N[i,j] \leq N_o[i,j]$.

The top-level $\oplus$ in the fixed-order sum $N_o[i,j]$ splits the sum from i to j into two sub-ranges i to l and $l+1$ to j, such that $i \leq l$ and $l+1 \leq j$. Thus:

$$\begin{aligned}
N[i,j] &= N[i,k] \oplus N[k+1,j] && \{ \text{ def. } k \text{ and def. } N[i,j] \ \} \\
&\leq N[i,l] \oplus N[l+1,j] && \{ \text{ def. } k\text{: min sum in } N[i,j] \ \} \\
&= rd(N[i,l] + N[l+1,j]) && \{ \text{ def. } \oplus \ (+ \text{ denotes addition in } \mathbb{R} \ \} \\
&\leq rd(N_o[i,l] + N_o[l+1,j]) && \{ \text{ IH: } l - i < j - i, j - (l+1) < j - i \ \} \\
&= N_o[i,l] \oplus N_o[l+1,j] && \{ \text{ def. } \oplus \ \} \\
&= N_o[i,j] && \{ \text{ def. } l \ \}.
\end{aligned}$$

The second $\leq$ step exploits the monotonicity of the rounding function rd. □

Value $\max_M ve_\oplus(\boldsymbol{v}, M)$ can be computed analogously. Both algorithms, denoted $getMin_\oplus$ and $getMax_\oplus$ in the sequel, can be implemented in $\mathcal{O}(n^3)$ time, by filling $N[i,j]$ "bottom-up". We use similar procedures $getMin_\otimes$ and $getMax_\otimes$ to minimize and maximize volatile products. In the following we describe how our method applies to the computation of extreme values for FMA expressions and volatile dot products, resulting in the four procedures $getMin_{\{fma|dot\}}$ and $getMax_{\{fma|dot\}}$. The correctness argument and the algorithmic complexity are the same as in the summation case and omitted.

Minimizing Volatile FMA: For an expression $ve_{fma} = fma(v_1, v_2, v_3)$, there are only two possible evaluation models, namely using or not using FMA. These two models can simply be compared against each other.

Minimizing Volatile Dot Product: Consider an expression $ve_{dot} = v_{11} \otimes v_{21} \oplus \ldots \oplus v_{1n} \otimes v_{2n}$ under an evaluation model that supports FMA. Here we need not only consider different ways of parenthesizing the expression, but also different ways of applying FMA. For example, for $n = 3$ the above expression can be evaluated in many different ways, among others the following:

$$\begin{array}{ll}
v_{11} \otimes v_{21} \oplus (v_{12} \otimes v_{22} \oplus v_{13} \otimes v_{23}) & fma(v_{11}, v_{21}, v_{12} \otimes v_{22} \oplus v_{13} \otimes v_{23}) \\
fma(v_{11}, v_{21}, fma(v_{13}, v_{23}, v_{12} \otimes v_{22})) & fma(v_{13}, v_{23}, fma(v_{12}, v_{22}, v_{11} \otimes v_{21}))
\end{array}$$

Our method can be used to compute the minimum over all different evaluation models. The following equations determine $\min_M ve_{dot}(\boldsymbol{v}, M)$ to be $N[1, n]$:

$$N[i,j] = \begin{cases} v_{1i} \otimes v_{2i} & \text{if } i = j \\ \min\{fma(v_{1i}, v_{2i}, N[i+1, j]), & \\ \quad \min\limits_{i<k<j-1} \{N[i,k] \oplus N[k+1, j]\}, & \\ \quad fma(v_{1j}, v_{2j}, N[i, j-1])\} & \text{if } i < j \end{cases}$$

Minimizing General Volatile Expression: For an arbitrary floating-point expression e, we approximate the maximum and minimum values using *interval analysis* [4]. Interval bounds on floating-point expressions soundly propagate to sums, products, etc. as shown in Eq. (8) [7], where $[e]$ denotes an interval bound on expression e with lower and upper bounds $\downarrow e$ and $\uparrow e$:

$$\begin{aligned} [e_1 \oplus e_2] &= [\downarrow e_1 \oplus \downarrow e_2, \uparrow e_1 \oplus \uparrow e_2] \\ [e_1 \ominus e_2] &= [\downarrow e_1 \ominus \uparrow e_2, \uparrow e_1 \ominus \downarrow e_2] \\ [e_1 \otimes e_2] &= [\min(\downarrow e_1 \otimes \downarrow e_2, \downarrow e_1 \otimes \uparrow e_2, \uparrow e_1 \otimes \downarrow e_2, \uparrow e_1 \otimes \uparrow e_2), \\ &\quad\ \max(\downarrow e_1 \otimes \downarrow e_2, \downarrow e_1 \otimes \uparrow e_2, \uparrow e_1 \otimes \downarrow e_2, \uparrow e_1 \otimes \uparrow e_2)] \\ [e_1 \oslash e_2] &= [e_1 \otimes [\tfrac{1}{\uparrow e_2}, \tfrac{1}{\downarrow e_2}]] \quad \text{if } 0 \notin [e_2] \\ [ve_T(e_1, \ldots, e_n)] &= getBound_T([e_1], \ldots, [e_n]) \quad (T \in \{\oplus, \otimes, dot, fma\}) \end{aligned} \tag{8}$$

Equation (8) suggests to find an interval for e under evaluation model variations by composing intervals for its subexpressions, computed recursively. In particular, function $getBound_T$ defined in Algorithm 2.2 computes an interval for volatile expression ve, given intervals for subexpressions e_i. The algorithm considers all 2^n combinations of lower/upper bound for expression e_i $(1 \leq i \leq n)$, and call functions $getMin_T$ and $getMax_T$, which implement the min/max computations described earlier in this section. Note that at the "leaves" of the recursive descent the e_i's are constants, so $[e_i]$ degenerates to the single point. The loop in line 3 goes through **one** iteration in this case. For non-leaves, n is bounded by the code size of expressions in the program text.

Using (8) we can prove that $[\downarrow e, \uparrow e]$ contains the exact interval $[e^{\min}, e^{\max}]$.

Algorithm 2.2. $getBound_T$

```
  Input: $\{[e_1], [e_2], \ldots, [e_n]\}$
1 $\downarrow e = +\infty$ $\uparrow e = -\infty$
2 for $(v_1, \ldots, v_n) \in \{\downarrow e_1, \uparrow e_1\} \times \ldots \times \{\downarrow e_n, \uparrow e_n\}$ do
3     $v_{min} = getMin_T(v_1, \ldots, v_n)$
      $v_{max} = getMax_T(v_1, \ldots, v_n)$
      $\uparrow e = max(\uparrow e, v_{max})$
      $\downarrow e = min(\downarrow e, v_{min})$
4 return $[\downarrow e, \uparrow e]$
```

Algorithm 2.3. Calibration

```
Input: constraint $\psi_k$: : $X_k \rhd c_k$,
       interval $[\downarrow X_k, \uparrow X_k]$ that
       does not contain $c_k$
if $c_k < \downarrow X_k$ then
    $\epsilon = \downarrow X_k - c_k$ // $\epsilon > 0$
else
    $\epsilon = \uparrow X_k - c_k$ // $\epsilon < 0$ $(\uparrow X_k < c_k)$
$\phi_{appr}$ := $X_k(I_0) \approx c_k - \epsilon$
```

2.5 Phase II (b): Calibration

Given the conditional q over atomic subformulas $\psi_i{::}X_i \rhd c_i$, we use the above methods to compute an input I_0 and the interval $[\downarrow X_i, \uparrow X_i]$ $(1 \le i \le n)$. We now check whether these results satisfy our requirements from Sect. 2.3: ϕ_{path}: given input I_0 the program follows the execution path that leads to the conditional q; ϕ_{assgn}: for $i \neq k$, ψ_i has the same Boolean value as A_0 assigns to b_i, for **both** bounds $\downarrow X_i$ and $\uparrow X_i$; ϕ_{appr}: the values of ψ_k for $X_k = \downarrow X_k$ and $X_k = \uparrow X_k$ differ: $\downarrow X_k \rhd c_k \;\neq\; \uparrow X_k \rhd c_k$.

If I_0 does not satisfy ϕ_{path} or ϕ_{assgn}, we ask the solver to generate a different input I_0, using a suitable blocking constraint. If I_0 and the interval fail ϕ_{appr}, we further distinguish the following cases: (a) the lower and upper bounds of X_k are the same: $\downarrow X_k = \uparrow X_k$. We deal with this as before by asking the solver for a new input; or (b) $\downarrow X_k \neq \uparrow X_k$ but the two bounds are on the same side of c_k: $\downarrow X_k \rhd c_k \;=\; \uparrow X_k \rhd c_k$. Here we employ a step-by-step calibration strategy shown in Algorithm 2.3. If, for example, $c_k < \downarrow X_k$, then the values returned by I_0 are slightly too large. We define ϵ to be the "error" $\downarrow X_k - c_k$ and adjust formula ϕ_{appr} to account for this error, by reducing the point of comparison c_k. We now repeat the process using the new set of constraints. In our experiments, we typically needed to go through 10–15 iterations of this calibration loop (see Fig. 1) if we were able to find an unstable input. The process is halted after some user-specified number of unsuccessful calibrations.

3 Empirical Results

We have vetted the algorithm described in Sect. 2 on several examples to generate inputs that may trigger control flow divergence. The examples are then executed on several different hardware platforms on these inputs, and the results are compared. The examples include the ray tracing code from Sect. 1, long summations, and molecular dynamics. More details are available at our website.[1]

Hardware Used in Comparisons. We target a range of different computer hardware for our experiments, including: (1) two different CPUs (Intel and AMD 64 bit processors), (2) an AMD Radeon 6550D GPU, and (3) three different NVIDIA GPUs (Quadro 600, Tesla C2075 and Tesla K20). All targets have OpenCL compilers, provided by each manufacturer, which were used to generate the results. All the target hardware is IEEE 754–2008 compliant. The Intel and AMD CPUs are 64 bits. Intel has FMA, but only in its multimedia instructions (AVX) which are not used in these experiments. The AMD processor does not have FMA. The three NVIDIA GPUs all have FMA instructions and the NVIDIA OpenCL compiler applies FMA aggressively. The AMD GPU does not have hardware FMA instructions.

[1] http://www.ccs.neu.edu/home/wahl/Research/FPA-Heterogeneous/Non-Portability.

Tool Specifics. We have implemented the algorithm on top of the KLEE symbolic analysis engine [3] with LLVM-2.9. We use the Z3 theorem prover [8], version 4.3.2, as KLEE's internal solver. The experiments are run on a Ubuntu 14.04.1 LTS machine with Intel Core-i7 3.10 GHz processor and 8 GB RAM. With the exception of the calls to Z3 for finding the candidate inputs, **the running time of our algorithm is negligible and in the tens of milliseconds**, which is why we omit performance details.

Code Instrumentation. We test our algorithm on the C code version of the examples. In the first step, we apply transformations to each program: we attach a **main** function that calls the tested program, and we annotate the symbolic variables and volatile expressions, so that our algorithm can detect these later.

For a conditional q in the execution path, there may exist multiple inputs that exhibit control flow instability. In our experiments we split the domain of each input variable into subintervals of length 0.01, and run our algorithm on each of these subintervals. For ray tracing, for example, this produced 408 sets of inputs over the range [−50, 50] that cause control flow instability according to Definition 1.

3.1 Examples and Control Flow Divergence

Ray Tracing. The OpenCL ray tracing code was run with the 408 different sets of inputs generated by our algorithm to produce results where D is close to zero. Inputs provided were the three dimensions of the sphere: $s[0], s[1]$ and $s[2]$, the three dimensions of the ray, $r[0], r[1]$ and $r[2]$ and the *radiusSq*. All 408 input sets generated differences on different architectures. 45 of the 408 sets of inputs produce results that are on either side of zero for the comparison when the same code was run on different platforms.

Summation. Summations of floating-point values are common in scientific computing. We compare serial C code which accumulates a value to a register, and a reduction kernel, written in OpenCL which is the common way to implement long summation on a parallel architecture. The result of the sum is compared to a threshold, set to zero for these experiments.

We ran our OpenCL kernel for a sum of 32 values generated by the algorithm. We received 100 different sets of inputs from the algorithm, of which 58 gave us different results. Three sets of results are shown in Table 1. The reduction values differed from the serial summation, as expected. Most platforms produced the same results of the reduction sum except for the NVIDIA Tesla 2075, which produced different results. These differences are due to reordering. We were able to illustrate differences even with a short list of input values (32) where the range of these values is small ($[-1.05, 1]$). For practical applications where both the number of values and their range will be larger, we expect these differences to be more dramatic.

Table 1. Results for reduction sum (left) and MD (right)

Red Sum 1	Red Sum 2	Serial Sum
-4.6566E-8	-8.7544E-8	4.4936E-8
-5.9605E-8	-8.7544E-8	1.0547E-7
-5.9605E-8	-8.7544E-8	9.1502E-8

R1	R2
0.0	-2.3841858E-7
-2.3841858E-7	0.0

Molecular Dynamics. MD is a popular high performance computing application with many versions available that run on parallel processors and on GPUs [1]. MD is sensitive to drift in floating-point calculations due to the large number of time steps in simulation. Taufer et al. [11] show this and use a tuple of values to represent both the floating-point number and its error. Our experiments use open source code[2] to calculate the Lennard Jones potential energy of molecular systems. We specifically focus on the comparison with *rrCut*, a constant which specifies the cutoff distance (see Listing 1.1). Atoms further away than *rrCut* are assumed to not affect the result.

```
/* Doubly-nested loop over atomic pairs */
 ...
   /* Computes the squared atomic distance */
       for (rr=0.0, k=0; k<3; k++) {
         dr[k] = r[j1][k] - r[j2][k];
         dr[k] = dr[k] - SignR(RegionH[k],dr[k]-RegionH[k])
                       - SignR(RegionH[k],dr[k]+RegionH[k]);
         rr = rr + dr[k]*dr[k]; }
/* Computes acceleration & potential within the cut-off distance */
if (rr < rrCut) { ... }
```

Listing 1.1. Molecular Dynamics

We implemented this code in OpenCL and ran it on our six target platforms. We set *rrCut* to 2.25. The algorithm was used to generate inputs $r[j1][0]$, $r[j1][1]$, $r[j1][2]$, $r[j2][0]$, $r[j2][1]$, and $r[j2][2]$ that bring rr close to 2.25. It found twelve sets of values where this decision is on either side of *rrCut*. Table 1 shows the difference between 2.25 and the value rr. 6 sets of inputs produced one of these results and 6 sets produced the other. R1 results are produced by the two CPUs and the AMD GPU, none of which use FMA. The R2 results are produced on the NVIDIA GPUs, all of which use FMA. Note that MD simulations run for a long time, and the value of *rrCut* affects the run time by determining how many calculations are done. Our algorithm can be used to help set *rrCut* and thus reduce the overall run time.

4 Conclusions and Future Work

We have shown that floating-point instabilities can lead to different control flows in code, and have introduced an algorithm to find values that potentially exhibit

[2] http://cacs.usc.edu/education/cs596/src/md/.

such instability when code is run on the same inputs on different machines. Our algorithm can inform programmers whether their code has instabilities for certain ranges of input and parameter choices. For molecular dynamics, it can help improve run times by allowing an intelligent choice of cut off distance. In the future we plan to improve the usability of the algorithm and to apply it to more complicated examples.

References

1. Anderson, J.A., Lorenz, C.D., Travesset, A.: General purpose molecular dynamics simulations fully implemented on graphics processing units. J. Comput. Phys. **227**(10), 5342–5359 (2008)
2. Bao, T., Zhang, X.: On-the-fly detection of instability problems in floating-point program execution. SIGPLAN Not. **48**(10), 817–832 (2013). http://doi.acm.org/10.1145/2544173.2509526
3. Cadar, C., Dunbar, D., Engler, D.: Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI 2008, pp. 209–224. USENIX Association, Berkeley (2008). http://dl.acm.org/citation.cfm?id=1855741.1855756
4. Hickey, T., Ju, Q., Van Emden, M.H.: Interval arithmetic: From principles to implementation. J. ACM **48**(5), 1038–1068 (2001). http://doi.acm.org/10.1145/502102.502106
5. Institute of Electrical and Electronics Engineers (IEEE): 754–2008 – IEEE standard for floating-point arithmetic, pp. 1–58. IEEE (2008)
6. Jzquel, F., Chesneaux, J.M.: Cadna: a library for estimating round-off error propagation. Comput. Phys. Commun. **178**(12), 933–955 (2008). www.sciencedirect.com/science/article/pii/S0010465508000775
7. Miné, A.: Relational abstract domains for the detection of floating-point run-time errors. In: Schmidt, D. (ed.) ESOP 2004. LNCS, vol. 2986, pp. 3–17. Springer, Heidelberg (2004)
8. de Moura, L., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
9. Munshi, A.: The OpenCL Specification, version 1.2 (2012). https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf
10. Shewchuk, J.R.: Robust adaptive floating-point geometric predicates. In: Proceedings of the Twelfth Annual Symposium on Computational Geometry, pp. 141–150. ACM (1996)
11. Taufer, M., Padron, O., Saponaro, P., Patel, S.: Improving numerical reproducibility and stability in large-scale numerical simulations on gpus. In: IEEE International Symposium on Parallel and Distributed Processing (IPDPS), 2010, pp. 1–9. IEEE (2010)