Identifying Volatile Numeric Expressions in OpenCL Applications

Mahsa Bayati, Brian Crafton
and Miriam Leeser
Electrical and Computer Engineering
Northeastern University
Boston, MA 02115

Yijia Gu and Thomas Wahl
College of Computer and
Information Science
Northeastern University
Boston, MA 02115

***Abstract*—The results of numerical computations with floating-point numbers depend on the execution platform. One reason is that, even for similar floating point hardware, compilers have significant freedom in deciding how to evaluate a floating-point expression, as such evaluation is not standardized. We call an expression *volatile* if its value, for a given input, differs across platforms. Differences can become particularly large across (heterogeneous) parallel architectures. This may be surprising to a programmer who conflates the portability promised by programming standards such as OpenCL with reproducibility.**

In this paper we present experiments, conducted on a variety of platforms including CPUs and GPUs, that showcase the differences that can occur even for *randomly selected* inputs. We present a theoretical technique that determines tight bounds for volatile expressions, and present experiments that show that the bounds are indeed observed experimentally. The information provided by these bounds can be used to direct the programmer or compiler to focus on those portions of the program where reproducibility is important.

I. Introduction

OpenCL promises portability across a wide range of platforms, including CPUs and GPUs. While the same code can indeed be run, it is common to observe different numerical results for floating point computations over the same inputs. We demonstrate this by running SHOC benchmarks as well as an application (Jacobi Successive Over-relaxation) from the SciMark Benchmark on different platforms, where we define each platform as hardware plus compiler. All the hardware we target is IEEE 754-2008 compliant; however, differences are observed due to reordering and different usage of fused-multiply add by different hardware and different compilers.

The main contribution of this research is a method for identifying *volatile* expressions, where volatile is defined as an expression likely to produce different results on different platforms. Our approach differs from others in that our tools address differences between platforms rather than differences between floating point and real numbers. The method, based on dynamic programming, determines a maximum and a minimum value for such an expression. If these minimum and maximum values are not equal, then we have identified a volatile expression and there is a good possibility that a programmer will observe different results when evaluating that volatile expression on different platforms. Our experimental results show that the bounds discovered by our method are tight, but do indeed bound the volatility found when running the program with the same inputs on different platforms. Armed with the information about the bounds of different volatile expressions, the programmer can use this to apply constraints on compiler optimizations such as expression ordering only to portions of their programs where the outcome sensitivity has the greatest impact.

II. Different Results on Different Machines

For our experiments we target a range of different computer hardware all compliant with IEEE 754-2008. These target machines include AMD and Intel CPUs as well as NVIDIA Tesla GPUs and AMD Radeon APUs. (Details available at [1]). We run the same OpenCL code on the same inputs on all these platforms and observe differences.

We selected three applications from the Scalable HeterOgeneous Computing (SHOC) Benchmark Suite and ran them on the hardware platforms:
1. MD: Molecular Dynamics performs an n-body pairwise computation (the Lennard-Jones potential from molecular dynamics).
2. SPMV: Sparse Matrix-Vector Multiplication, multiplies sparse matrix with a dense vector.
3. Stencil2D: performs a 2D, 9-point single and double precision stencil computation.

We present the absolute differences for MD in Table I on two sets of inputs; for more results visit [1].

Table I. Largest Absolute Difference, MD

	MD-InputSet1	MD-InputSet2
AMDCPU, AMDGPU	9.33E+17	1.53E+14
AMDCPU, Intel	0	0
AMDCPU, NVIDIA	0	2560
AMDGPU, Intel	9.33E+17	1.53E+14
AMDGPU, NVIDIA	9.33E+17	1.53E+14
Intel, NVIDIA	0	2560

III. SOR Methodology and Results

Our results show that, while OpenCL is indeed portable in the sense of being able to run the same code on different platforms, the results are not *reproducible*. Our research addresses the question: "What can we do to give feedback to the programmer about when these differences matter?"

In earlier work [2], we developed an analysis method that takes a program P, a fixed input i and an expression

Table II. LARGEST ABSOLUTE DIFFERENCE SOR

	Input set 1
AMDCPU, AMDGPU	0.0
AMDCPU, Intel	0.0
AMDCPU, NVIDIA	2.38419E-07
AMDGPU, Intel	0.0
AMDGPU, NVIDIA	2.38419E-07
Intel, NVIDIA	2.38419E-07

x representing some intermediate result of the program. The method determines an *upper bound* I on the range of values $x(i)$ program P can produce for x, on input i, across different computational platforms.

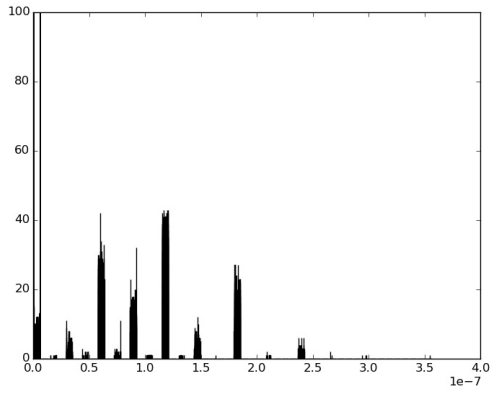
Since I overapproximates the range, not all values contained in I correspond to values for x that P will actually produce on input i, on some platform. The analysis method used to determine I roughly works as follows (for details, see [2]). Given input i and expression x, we first determine statically whether x is volatile: this is the case if x contains chains of operations that are subject to reordering, especially addition, or if x contains sub-expressions of the form $a*b+c$, which on many platforms can be compiled into a single *fused-multiply add* (FMA) instruction, where the intermediate rounding after the multiplication is avoided. For each volatile expression, the analysis now computes the *minimal* and *maximal* value x can have, under all possible evaluations. This problem can be solved using dynamic programming, polynomial in time in the size of expression x. These minimal and maximal values form the left and right boundary of interval I.

Part of the goal of this work is to demonstrate that the (theoretical) range interval I computed by our analysis method does not *vastly* overapproximate this range, but is in fact fairly tight. This is important information: smaller ranges permit more reliable estimates for the value of $x(i)$ independently of the computational platform. This in turn can help determine whether uses of x, such as in tests like `if (x < 0)`, are *fragile*: the test outcome depends on the platform, rendering the code highly non-portable.

To illustrate our technique we use Jacobi Successive Over-Relaxation (SOR). The C code is taken from SciMark benchmark [3] which we rewrote in OpenCL (listing 1). SOR is run on a 100x100 grid and is a stencil computation typical of finite difference applications. We selected *random* inputs (e.g. matrices with random cell contents), and determined, using our min/max technique, the range interval I for each input. We then ran these programs on a diverse set of platforms, observed the resulting differences, and compared them to the predicted range I. Figure 1 shows a histogram of output differences, with the maximum 3.5E-07; Table II shows the results of running this code on multiple platforms. The experimental results are indeed within the tight theoretical bound. In this particular example, the bounds are very tight, as there are constraints between different loops which limit the reordering of expressions. We plan to apply our technique to the SHOC benchmarks which exhibit larger differences and demonstrate that our tool does indeed produce tight bounds.

```
kernel void SOR(global float* A, int M, int N, float
    w){
int tx = get_global_id(0);
int ty = get_global_id(1);
```

Figure 1. Theoretical analysis of Max difference




```
int i, j;
 for(i=1; i<M-1; i++){
  for(j=1; j<N-1; j++){
    if(i == tx && j == ty){
          A[i*N + j] = (w/4) * (A[i*N + (j + 1)] + A
              [i*N + (j - 1)] +
          A[(i+1)*N + j] + A[(i-1)*N + j]) + (1.0-w)
               * A[i*N + j];
        } } } }
```

Listing 1. SOR OpenCL Kernel Code

IV. CONCLUSIONS AND FUTURE WORK

In this paper we have presented initial experiments that quantify differences that numeric programs produce, for the same input, across computational platforms. These differences arise due to lax expression evaluation rules in the IEEE 754 floating-point standard, and even in language-specific standards. The goal of the experiments was to show that these differences are "real" and occur not only for specific critical inputs, but in fact even for randomly chosen ones. We also demonstrated that the range of values predicted by our earlier (theoretical) method are fairly tight.

Ongoing and future work will lift our technique from a mere analysis to one that actually aides the programmer in making their programs more robust against platform changes. Such robustness can be achieved by inhibiting precision-enhancing hardware shortcuts like FMA, and by forcing certain expression evaluation orders. To minimize the impact on overall program performance, such measures should be applied only to small regions of the program that contribute most to the computational differences, leaving the compiler free to rearrange other parts.

REFERENCES

[1] "Floating point comparison for different platforms," http://www.coe.neu.edu/Research/rcl/projects/FloatingpointComparison/index.html.

[2] Y. Gu, T. Wahl, M. Bayati, and M. Leeser, "Behavioral non-portability in scientific numeric computing," in *Parallel and Distributed Computing (EURO-PAR)*, 2015, pp. 558–569.

[3] "Scimark 2.0," http://math.nist.gov/scimark2.