OpenCL Floating Point Software on Heterogeneous Architectures — Portable or Not?

Miriam Leeser[a], Jaideep Ramachandran[b], Thomas Wahl[b], Devon Yablonski[c]

Northeastern University, Boston, USA

[a]mel@coe.neu.edu
[b]{jaideep,wahl}@ccs.neu.edu
[c]dyablons@mc.com

Abstract

OpenCL is an emerging platform for parallel computing that promises portability of applications across different architectures. This promise is seriously undermined, however, by the frequent use of floating-point arithmetic in scientific applications. Floating-point computations can yield vastly different results on different architectures — even IEEE 754-compliant ones —, potentially causing changes in control flow and ultimately incorrect (not just imprecise) output for the entire program. In this paper, we illustrate a few instances of non-trivial diverging floating-point computations and thus present a case for rigorous static analysis and verification methods for parallel floating point software running on IEEE-754 2008 compliant hardware. We discuss plans for such methods, with the goal to facilitate the automated prediction of portability issues in floating-point software.

1. Introduction

Scientific applications, like high-throughput medical imaging, high-performance simulations of vehicles, climate development, molecular dynamics behavior, etc., demand great computational resources. These scientific applications are typically run on multi-core and multi-processor environments. To best take advantage of different types of parallelism, such environments are increasingly heterogeneous and may include Graphics Processing Units (GPUs) for fine-grained parallelism, as well as Field Programmable Gate Arrays (FPGAs) for energy efficiency, alongside traditional Central Processing Units (CPUs). OpenCL (Open Computing Language) [5] has emerged as a possible standard for programming systems with a variety of computational devices. Adopters of OpenCL include AMD, IBM, Intel, NVIDIA and Altera.

OpenCL specifically aims to deliver *portability* of parallel, heterogeneous programs, promising *write once, run anywhere* functionality. In many cases, however, the same program can produce different results when used with different architectures and compilers, a sensitivity that is especially critical for floating-point computations. The recently (2008) revised IEEE 754 floating-point standard leaves many implementation decisions to the execution platform. As a result, floating-point results generally do not agree across different such platforms, questioning the portability of OpenCL code.

OpenCL provides an excellent vehicle for examining correctness and stability issues with diverse floating-point implementations since the same code can be run on different architectures; the results can be compared. Debugging tools for OpenCL programs are currently very rudimentary. A generic debugger that debugs parallel code running on different hardware seems a distant goal. Finally, the shortcomings of testing parallel programs, with nondeterminism, synchronization issues and potential data races, are well known.

We are planning to address these issues by static analysis and verification methods based on model checking that analyze OpenCL code for the occurrence of floating-point constructs whose results are highly dependent on the available hardware — such code is a prime suspect for violating portability. We are conducting a thorough analysis of the behavior of OpenCL floating-point benchmark code on a variety of heterogeneous architectures to understand the sources of portability issues. The results of this study will be used to inform the development of analysis tools.

In this note we share some well-known and some less well-known dependencies of floating-point behavior on execution order, available hardware, and compiler decisions. We then sketch the work we plan to address these issues, and report on existing work in this area.

2. Floating-Point Issues in Sequential and Parallel Code

Floating-point code may produce different results on different processors. Some of the sources of these differences are highlighted below.

Non-associativity of floating-point. Floating-point arithmetic is not associative. If a compiler reorders computations, different results may arise. This is well-known and prevents certain compiler optimizations in sequential programs from being sound. When comparing sequential and parallel programs, however, the effects can be much more interesting.

Consider the program in Figure 1, which computes a Riemann sum approximation of π [8], using segment midpoints with *num_steps* segments. One expects that, the greater the number of segments (*num_steps*), the closer the computed value will be to the exact answer.

However, this is not observed for this sequential program running on a CPU, as seen in Figure 3 (green line) [10]: the addition of very large and very small floating-point values can result in a greater loss of precision than when summing values of similar magnitude. Floating-point can represent very large or very small values because it acts as a sliding window. If the window of representation slides towards a large value, a very small value cannot be represented in the same window and can be partially or completely lost in an operation like addition. More specifically, accumulating values sequentially will sometimes result in a

```c
static long num_steps = 100000;
double step;

void main(){
  int i; double x, pi, sum = 0.0;
  step = 1.0 / (double)num_steps;
  for (i=1; i<=num_steps; i++) {
    x = (i-0.5) * step;
    sum = sum + 4.0 / (1.0 + x * x);
  }
  pi = step * sum;
}
```

Figure 1: Sequential C code for π computation

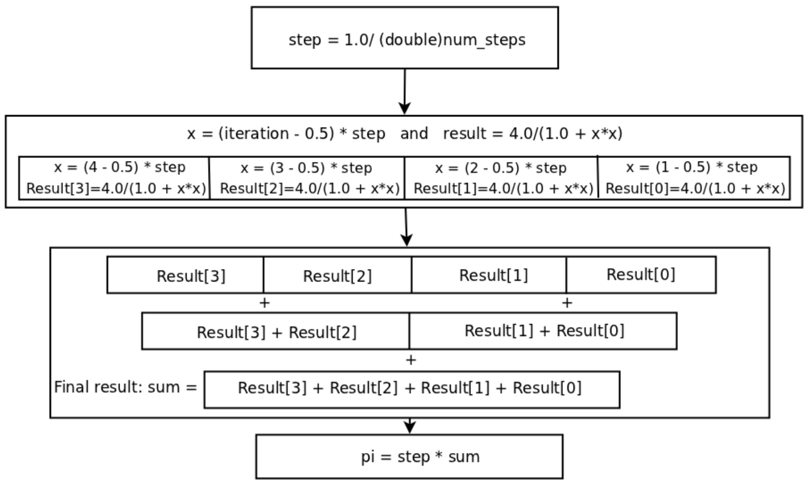
step = 1.0/ (double)num_steps
x = (iteration - 0.5) * step and result = 4.0/(1.0 + x*x)
x = (4 - 0.5) * step
Result[3]=4.0/(1.0 + x*x)
x = (3 - 0.5) * step
Result[2]=4.0/(1.0 + x*x)
x = (2 - 0.5) * step
Result[1]=4.0/(1.0 + x*x)
x = (1 - 0.5) * step
Result[0]=4.0/(1.0 + x*x)
Result[3]
Result[2]
Result[1]
Result[0]
+
+
Result[3] + Result[2]
Result[1] + Result[0]
+
Final result: sum =
Result[3] + Result[2] + Result[1] + Result[0]
pi = step * sum

Figure 2: Parallel π computation

large value that each successive small value is added to, resulting in diminished accuracy of the results.

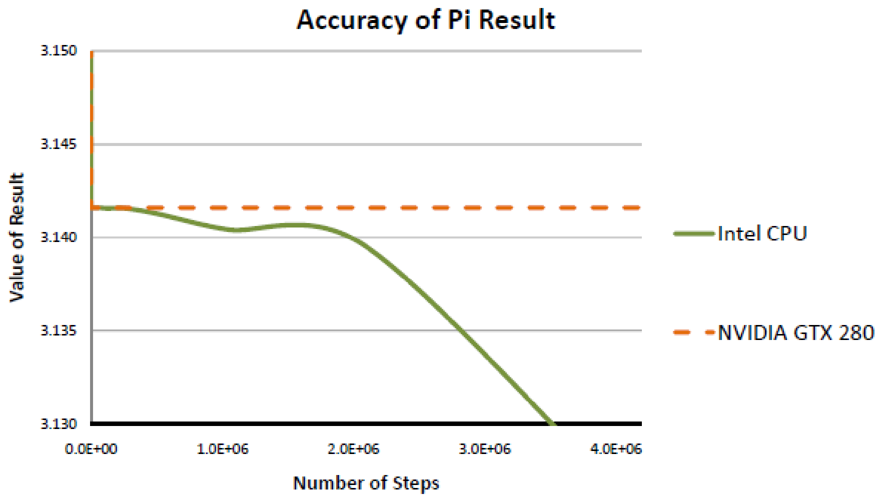

Figure 3: Sequential vs parallel computation of π

Consider now the parallel program in Figure 2. Here, the contributions to the final `sum` are accumulated in a way such that similar-in-magnitude values are being added first. This typical reduction-style of parallel computation thus increases the accuracy of the π computation. Figure 4 [10] illustrates this phenomenon.

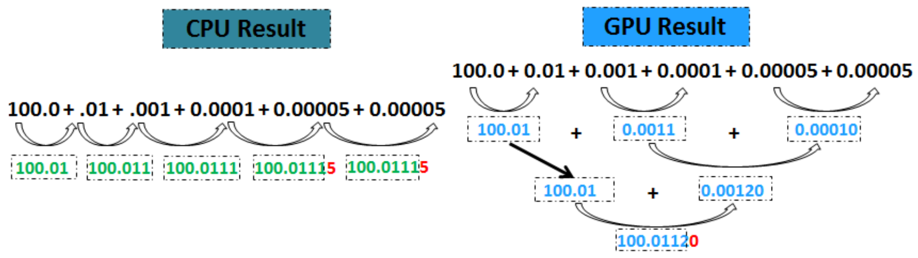

Figure 4: CPU vs GPU

Fused Multiply Add. The 2008 version of the IEEE floating-point standard acknowledged the *fused multiply-add* (FMA) operation, which had been around in hardware implementations of floating-point arithmetic since the 1990s. Many scientific codes implement computations, such as matrix multiplication, that consist of operations of the form $a \times b + c$ where a, b and c are represented in floating-point. Hardware support for such operations allows the result to be computed in an atomic step, without intermediate rounding and normalization between the multiplication and the addition. In most cases, this *fused* multiply-add operation gives more accurate results. Before 2008, however, the Standard assumed this expression was computed with separate multiplication and addition operations; the result $a \times b$ would be rounded and normalized before being added to c. Differences that arise between the use of FMA and the use of separate instructions for multiply and add is a common source of errors. There is – of course – no requirement that FMA hardware be used to implement $a \times b + c$. Its use will depend on the existence of supporting hardware as well as compilers that assign the computation to the FMA unit.

Different number of bits in intermediate representations. The FMA issue is a special case of a broader class of differences: floating-point implementations may use different numbers of bits in their internal representations when implementing operations. For example, floating-point computation running on a processor that uses the Intel x87 architecture may produce different results from the same code running on the same processor but using the SIMD (AVX) unit. Results are converted to IEEE floating-point double precision standard format, but the number of bits internal to the operation may change the value of the results.

Catastrophic cancellation and other issues. Many of the aforementioned issues may appear to only affect the last few bits. This is not the case when there is a *catastrophic cancellation* [4]. For example, consider the expression: $x \times y + x \times z$. For integer arithmetic, one may rewrite this to $x \times (y + z)$, which is not equivalent, however, for floating-point. Suppose $y = -z$. The result of $(y + z)$ will likely be a very small value but not precisely zero. As a result, the answers to the two different forms of this expression can differ in a large number of significant bits. A similar case arises when calculating $((x \times x) - (y \times y))$ given $x = y$ [6]. This computation may produce very different results depending on whether or not a fused multiply add is used.

3. Planned Work

Our planned work will be based on the assumption that the architectures and compilers under consideration comply with the IEEE 754 floating-point standard. The following are our high-level objectives:

- conduct a thorough (partly manual?) analysis of the behavior of OpenCL floating-point benchmark code on a variety of heterogeneous architectures, to establish a systematic study of the portability issues, along the lines sketched in the previous section. The findings of this study will be used to make informed decisions during tool building.

- develop tools (based on model checking) that analyze the code for the occurrence of floating-point constructs whose results highly depend on the available hardware and the compiler used and are thus prime suspects for violating portability.

We also plan to support a form of *cross-architecture equivalence checking*. When our tool is passed an OpenCL program and two architecture/compiler specifications, it would determine whether it is possible for the control flow to differ when this program is run on those two architectures. If so, the output will be the prefix of a path up to the point where the flows diverge. Note that exact equality of computational results across architectures is neither achievable, nor expected by programmers. However, control flow divergence is a red flag when it comes to architectural portability of the code.

4. Existing Work

The only published work we are aware of to analyze floating-point-intensive data-parallel programs is based on the KLEE tool, specifically KLEE-FP and KLEE-CL [3, 2]. These tools are not based on an IEEE 754-compliant logical modeling of floating-point programs, but instead on a *syntactic matching* of floating-point expression trees, after a series of canonizing rewritings. Such syntactic comparisons make sense only for equivalence-checking similarly generated floating-point formulas: KLEE-FP performs equivalence checks between a SIMD vectorized implementation of a floating-point program against an original scalar (single-threaded) implementation. The work in [9] also requires a sequential reference model to compare against. In contrast, we do not assume the existence of a scalar version of the code: we have in fact shown in Section 2 that pre-existing sequential code may not be the gold standard when it comes to correctness of code involving floating point operations. Our tools are planned to be applicable to scientific software directly written for multi-core architectures.

There are a number of analysis tools targeting programs written in NVIDIA's CUDA architecture, which permits writing programs specifically for GPUs. Our work, in contrast, addresses programs written in OpenCL, which targets much broader types of architectures. GKlee [7] analyzes programs with respect to properties like race detection and deadlocks. In addition to being CUDA specific, GKlee (i) does not support floating-point programs (as neither does the underlying symbolic execution engine Klee); and (ii) analyzes bytecode produced by the LLVM, a low-level virtual machine compiler. Since we are addressing floating-point issues in OpenCL software, we are targeting the source code level for our analysis. CBMC, a Bounded Model Checker for ANSI-C and C++ programs, has basic floating-point support for sequential programs [1]. We plan to build on its framework and extend it to analyze OpenCL programs.

Acknowledgments. We thank James Brock and Nicholas Moore for their assistance with running OpenCL programs on the heterogeneous infrastructure at the Reconfigurable and GPU Computing Laboratory of Northeastern University.

References

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

[2] Peter Collingbourne, Cristian Cadar, and Paul Kelly. Symbolic testing of OpenCL code. In *Haifa Verification Conference*, 2011.

[3] Peter Collingbourne, Cristian Cadar, and Paul H. J. Kelly. Symbolic cross-checking of floating-point and SIMD code. In *EuroSys*, pages 315–328, 2011.

[4] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, March 1991.

[5] Khronos Group. OpenCL – the open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/, 2011.

[6] W. Kahan. The improbability of probabalistic error analyses for numerical computations. http://www.eecs.berkeley.edu/ wkahan/improber.pdf, 1996.

[7] Guodong Li, Peng Li, Geoffrey Sawaya, Ganesh Gopalakrishnan, Indradeep Ghosh, and Sreeranga P. Rajan. GKLEE: Concolic verification and test generation for GPUs. In *PPOPP*, pages 215–224, 2012.

[8] Tim Mattson and Rudolph Eigenmann. OpenMP: An API for writing portable SMP application software. `http://www.openmp.org/presentations/sc99/sc99_tutorial.pdf`. Tutorial.

[9] Stephen F. Siegel, Anastasia Mironova, George S. Avrunin, and Lori A. Clarke. Combining symbolic execution with model checking to verify parallel numerical programs. *ACM Trans. Softw. Eng. Methodol.*, 17(2):10:1–10:34, May 2008.

[10] Devon Yablonski. Numerical accuracy differences in CPU and GPGPU codes. Master's thesis, Northeastern University, 2011. `http://www.coe.neu.edu/Research/rcl/publications.php#theses`.