Tools and Techniques for Floating-Point Analysis
(the slides are available under “Presentation Materials” in the above URL)
Date: October 16, 2019
Presented by: Ignacio Laguna (Lawrence Livermore National Laboratory)

Q. Anything interesting in IEEE 754 update from 2018?

A. In my opinion, the most interesting update was the addition of the fused multiply-add operation (FMA). As I discussed in the webinar, it can have unexpected effects. This Wikipedia page documents the 2008 revision: https://en.wikipedia.org/wiki/IEEE_754-2008_revision.

Q. Is floating-point ever reproducible? Are there reproducible libraries that perform floating point?

A. The answer depends on how we define reproducibility. If we consider compiler reproducibility only, I would say that’s almost impossible (as you can see from the webinar examples). If we are more strict and we consider run-to-run reproducibility for a fixed system and configuration (i.e., same compiler, same flags, same hardware, …) that’s a more feasible scenario for repro. Most of the large codes I know use “partial” reproducibility (not bit-by-bit). There is a significant amount of research work on floating-point reproducibility. I’m not aware of any library or framework that guarantees 100% reproducibility in the real world.

Q. What is a “random test” ? How are these generated? The examples shown all seem like they are examples that use FMA

A. A random test is simply a C program that is randomly generated, i.e., another program randomly generates the test program, instead of a human. We use a grammar that is a subset of the C language to specify the kind of programs that can be generated. The grammar supports features of typical HPC programs, e.g., floating-point variables, arrays, loops, if conditions, etc. Yes, these examples may suggest to the compilers to use FMA, but we do have other examples with variations on which this is not the case. Note that the example themselves *do not* use FMA -- it is the compiler that decides whether FMA code should be generated or not.

Q. Is the xlc -O0 result using FMA the “correct” result?

A. The presented tool cannot determine what result is the correct one -- all the tool can do is to identify variations when the same code is compiled with different compilers and when, for a given input, the results are different. In my experience, correctness usually depends on the application; since these tests are not an actual application, we omit specifying which result is correct and which is not.

Q. Maybe a useful metric to be able to provide from these tools is to give users an expected absolute or relative “tolerance” to expect in floating point values across the range of compilers, flags, architectures, etc. that are available on a given system (for common cases, not exceptional ones that result in unexpectedly very large diffs)?

A. This is a very insightful suggestion. Yes,I believe that this would be very useful to the users of a system. Thank you for the suggestion!

Q. Is it possible that FPchecker changes answers? Sometimes a bug only shows up in a release build and not a debug build. Could something similar happen with FPChecker?

A. Unless there are race conditions in the GPU code, where results may vary run-to-run, I wouldn’t expect FPChecker to change answers. We designed it in a way that it doesn’t affect thread scheduling (every GPU thread performs the same check without synchronizing). I could see, however, a situation where the bug manifests for an input that we could not run with FPChecker, e.g., because the overhead is too high for that input to be run with FPChecker. If that happens, we have techniques we can apply to reduce the overhead.

Q. Are formalized tools feasible on larger scale codes (e.g. Coq+floqc)

A. This is a big question. In fact, I co-organize a workshop on correctness at the SC conference where we are trying to make formal methods more accessible to large-scale real applications, like those that DOE labs run. I personally believe that formal methods such as Coq, Z3, or others are becoming more feasible to large codes; having said that I haven’t seen any of these methods applied to a real large code (at least not within DOE). A feasible approach is to apply them to some of the most important modules of an application. I think that more research is needed to make these methods more scalable and practical to programmers of HPC codes. This DOE report summarizes some of the challenges: https://www.osti.gov/biblio/1470989.

Q. What is the overhead of the checker?

A. I believe by checker we mean FPChecker. If so, the slowdown that we have seen for the 3-4 applications we have tested it on is about 1.2x to 1.5x.

Q: Are subnormal numbers same as underflow?

A. They are not strictly the same. However, for the tools I develop I consider subnormal numbers the results of underflows; this is also what many compilers assume when checking for underflows. More formally, if a value is smaller than the smallest value representable as a normal floating point number (for a given precision), I consider it an underflow.

Here’s an example with the IBM XL compiler. Consider this code:

```
$ cat test_subnormal.c

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
  double x = atof(argv[1]);
  x = x * 1.1;
  printf("x: %.17g\n", x);

  return 0;
}
```

When we compile it with -O0 and give a subnormal number as input we get a subnormal number:

```
$ xlc -O0 -o test_subnormal test_subnormal.c
$ ./test_subnormal 1e-319
x: 1.0999877539009513e-319
```

Now, we tell the XL compiler to detect underflow exceptions. This is done by using this flag: “-qflttrap=enable:und”

```
$ xlc -O0 -g9 -qflttrap=enable:und -o test_subnormal test_subnormal.c
[laguna@ray23 tmp]$ ./test_subnormal 1e-319
Floating point exception
```

We get a floating-point underflow exception error.

Q. Kahan (father of IEEE-754) has several examples of where subnormal numbers help improve the results of expressions. Comments? (See wikipedia article on denormal number for links)

A. The important question here is whether a subnormal number is better with respect to flushing to zero, or better with respect to a normal number. If we compare them to flushing to zero, yes, I could imagine how in several cases it is better to use a subnormal than zero. If we are saying that a subnormal number is better than a representation of the number as a normal number (e.g., using a higher precision format), that I’m not sure if it is the case. I would like to see examples of the later, but I couldn’t find them. In general, I still suggest avoiding the real of subnormal numbers if possible because it could affect results in unexpected manners when high levels of optimizations are used.

Q. In the “fixed” raja example, the value still seemed to differ in the 4th significant digit. Isn’t that still really bad?

A. Yes, that’s still bad. Later in the investigation within the root-cause function, we identified some lines of code that we had to modify to completely fix the issue. I just didn’t have time to present that part of the work in the webinar. Got point!

Q. (for end of presentation) Do you have thoughts/comments on how lossy compression of floating point data and/or using half-precision (either IEEE or home-grown) impact these issues?

A. Unfortunately, I don’t have enough experience in lossy compression to make a meaningful comment. Regarding half precision, this is something we are exploring in some applications. Half precision of course limits the range of floating-point values that your application can use (overflows/underflows may be more probable), but if your code can leverage them, the potential performance improvements can be large.

Q. Do the tools you presented up to now support parallel programs (either OpenMP and/or MPI)?

A. Most of the tools can use MPI with some extra work, e.g., traces can be gathered on different MPI processes and can be analyzed offline. For tools, such as FPChecker, MPI runs are supported naturally (each MPI process performs its own check when using GPUs). FLiT will support MPI very soon (the work is ongoing). The SC tutorial presents some tools that are specific to MPI and OpenMP, e.g., ReMPI to control MPI nondeterminism and ARCHER to check for OpenMP data races.

Q. What is needed for these tools to work with posits / unums?

A. Posits/unums are a great idea. I, however, do not have much experience with them, primarily because I do not see many applications using them. Most of the real-world applications I deal with use floating-point. Having said that, I do believe that posits/unums solve some of the issues of floating-point, so perhaps if more applications use them, less tools would be needed to isolate floating-point problems.

Q. Do you believe the return on investment in these mixed precision strategies (e.g. much higher code complexity, more difficulty in reproducing, etc.) is likely to produce higher performance benefits than other/typical performance improvement strategies?

A. It depends on what we mean by “other/typical performance improvement strategies”. The typical expected improvement out of mixed-precision is 2X. This is because most codes use double precision (64 bits) in HPC, and programmers think that the best they can so is compute all in single precision (32 bits). These back-of-the-envelope estimates do not consider however using three-level mixed-precision, i.e., double, single, and half precision. I believe that

three-level mixed-precision could deliver improvements higher than 2X, perhaps 5X or even higher. Note that performance improvements not only come from higher compute throughput but also from effects on memory usage, memory transfer, communication, etc.

C: Subnormals are numbers that cannot be represented using the implied 1.xxxx and thus they need to increase the exponent value. Over/Underflow means that the exponent is too big/small to be represented by IEEE754 bin FP format. (This comment is related to one of the previous questions.)

A. Thanks for the comment. Strictly speaking this is correct. Some compilers, however, have a different definition of underflows (see my example above with the XL compiler). Also even sites define underflows as anything small than cannot be represented as a normal number; see this definition here:
https://en.wikipedia.org/wiki/Arithmetic_underflow#targetText=The%20term%20arithmetic%20underflow%20(or,in%20memory%20on%20its%20CPU.

"Arithmetic underflow can occur when the true result of a floating point operation is smaller in magnitude (that is, closer to zero) than the smallest value representable as a normal floating point number in the target datatype."

For practical tools, I use a less strict approach and I do not make the distinction between subnormal numbers and underflows just because we want to inform to the programmer that the program is getting in the domain of these very small numbers and care should be taken.

Q. In trying to optimize mixed-precision code, does each variable contribute to the error and speed-up independently? If changing the precision of variable 1 yields some error and speed-up and changing the precision of variable 2 yields some error and speed-up, does the total error and speed-up add together?

A. First, I would like to say that the method of modifying variables to improve speedup is wrong; we should focus on modifying hardware instructions. A variable may influence 10 instructions, but an optimal case may be to convert 3 of these instructions into lower precision; I don't have a way to do that with variable type changes (either I modify all the 10 instructions or not).
Coming back to your question: I don't believe one can add the effects of variables 1 and 2 and expect a monotonic decrease or increase. In general, the error amount is not monotonic as a function of variables changes. This is why the search for mixed-precision configurations that satisfy both accuracy and speedup is complex and long.
Thanks for the great questions! Ignacio.

Thanks so much to Ignacio! Great presentation