GPU Floating-Point Paranoia

Karl Hillesland and Anselmo Lastra
University of North Carolina, at Chapel Hill.
email

What do you mean by "Paranoia"?

Paranoia [1] is the name for a program written by William Kahan in the early 80�s. It was designed to characterize floating-point behavior of computer systems. The goal of this project is to characterize the floating-point behavior of graphics hardware, and in fact adopts some of the tests from the original "Paranoia" program.

What do you mean by "Floating-Point"?

A typical binary floating-point representation would be:

Sign Significand x 2^{exponent}

The sign is a single bit. The significand (or mantissa) and exponent are each represented by a fixed number of bits. The important thing to note is that this represents a finite set, not a continuous number system.

What can you do with floating-point numbers?

You can add, subtract, multiply, and divide, but not in the abstract mathematical sense. This is because the set of floating-point numbers is not closed over these operations. For example: Assume a 4 bit significand. We will ignore the issue of the implied leading one for now, and use one of the bits to store it explicitly. We will use a subscript to indicate the base of the representation. Let�s try the following addition:

$$(16 + 1.5)_{10} = (1.000 \text{ x } 2^4 + 1.100 \text{ x } 2^0)_2 = (1.0011 \text{ x } 2^4)_2$$

The problem is that the exact result cannot be represented with a 4 bit significand. The closest we can come is either $(17)_{10} = (1.001 \text{ x } 2^4)_2$ or $(18)_{10} = (1.010 \text{ x } 2^4)_2$.

Which one should we choose?

In 1987 the IEEE developed a standard for this choice [2] The rule is to round to the nearest value, and in the case of a tie (as in the above case), round to whichever value is even. Therefore, the result would be 18. The error in the answer, referred to as "rounding error" is 1/2 ulp (unit in last place). In this case the ulp is one, so the error is 0.5, which is the maximum error you can get for this particular exponent. CPUs are designed to follow the IEEE standard. GPUs do not.

Then what does a GPU do?

This is the very question we are trying to answer. There is currently no standard in place for GPUs. Additionally, GPU manufacturers won�t even tell us what they do. So our only recourse is to figure it out for ourselves. Below are the results for two abstract systems and two real systems.

Table 1. Floating-Point Error in ulps*

Operation	Exact Rounding (IEEE-754)	Chopped	R300/arbfp**	NV35/fp30**
Addition	[-0.5,0.5]	(-1,0]	[-1.000, 0.000]	[-1.000, 0.000]
Subtraction	[-0.5,0.5]	(-1,1)	[-1.000, 1.000]	[-0.750, 0.750]
Multiplication	[-0.5,0.5]	(-1,0]	[-0.989, 0.125]	[-0.782, 0.625]
Division	[-0.5,0.5]	(-1,0]	[-2.869, 0.094]	[-1.199, 1.375]

*** Note that the R300 has a 16 bit significand, where as the NV35 has 23 bits. Therefore 1 ulp on an R300 is equivalent to 2^7 ulps on an NV35.**

**** Results obtained with Cg version 1.2.1, ATI driver 6.14.10.6444, NVIDIA dreiver 56.72 on a Windows/OpenGL platform**

Why is the error in division higher?

It turns out that neither of these GPUs have an instruction for division. Instead, it is performed by a reciprocal and a multiply, so error is actually incurred twice. Also, division is more expensive to perform, and is therefore more likely to involve aggressive approximation relative to other operations.

How did you do this?

We measure the error empirically. To exhaustively try all combinations of all possible floating-point numbers for all four operations would be impractical. Instead, we chose a subset of floating-point numbers that we believe does a reasonable job of characterizing the entire set. This is an approach used by others for testing correct operation of floating-point hardware. Schryer, for example, selected an important set of significands for testing floating-point [3]. Since we are interested in measurement rather than yes/no answers, we expanded the set and included additional exponents in order to push the limites of round-off error and cases where the most work must be performed, such as extensive carry propagation. The significands and exponents are given below.

significand patterns:

> 1.00...010... (all zeros except a single one, or Schryer's "Type 1")
>
> 1.10...010... (same as previous, but with additional leading one)
>
> 1.11...101... (all ones except a single zero)
>
> 1.00...01... (zeros then ones)
>
> 1.10...01... (same as previous, but with additional leading one)
>
> 1.11...10... (ones then zeros, or Schryer's "Type 2")

relative exponents:

> 0, -1, -(significand bits / 2), -significand bits

So that's all there is to it?

No, there's plenty more to consider. First of all, there are a number of factors that affect floating-point behaviour here: hardware, driver, compiler, host sytem, etc. This is why we developed a tool that can be easly run whenever needed.

Semantics for programming GPUs currently allow for considerable leeway in how a program is implemented. Instructions can be re-ordered. Subexpressions involving constants or "uniform" parameters may be evaluated on the CPU. Associative and distributive properties, which do not hold for floating-point operations, may be applied in optimization. Our tool does not take into consideration the kinds of optimizations possible in larger program contexts.

What does this have to do with the original Paranoia?

The original Paranoia uses six tests for each operation. The tests are chosen to be tough cases for exact rounding, or failing that, chopping. Therefore, the results are "seems to be rounded correctly", "seems to be chopped" or "neither". There are further tests for a sticky bit, but those tests are only used if rounding looks good up to that point.

Note that in the case of exact rounding or chopping, you know the error bars. They are given in Table 1. If the result is "neither", you can�t say anything about the accuracy of the floating-point operations (although there are earlier checks to make sure the accuracy is at least reasonable).

Where can I learn more?

A good place to start is *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, which is listed in the references [4]. I�ve also included a link to the extended abstract that will go into the GP^2 Proceedings [5].

Can I try it?

Of course! I�ll give you source code and windows executables, but there are a few warnings here.

In general this is the raw code and executables I used to get numbers. There�s nothing nice about it! Repercussions follow:

- I have only done this for Windows

- Output will have many cryptic items that may not make any sense to you. The final error bars, which are probably all you care about, are at the end of the output. I did try to cut out a lot of extra prints, but its still kinda ugly looking.
- (source code) There is a bunch of cryptic stuff in the code because I was originally going to support both GLSL and Cg, but stopped supporting GLSL halfway.
- If you're wondering why I used Cg, it's because I wanted to be able to see the so called "assembly" that is produces. I'll leave issues of what is meant by "assembly" for another day.
- There are still places in the code where *runtime* subexpression collapses might invalidate certain tests, but I didn't observe that behavior. That of course doesn't preclude errors in these tests due to future optimizations. Error bar measurements are among those that are safe from this kind of optimization.

I may do some cleaning and updating in the future, but that will depend partly on whether this turns out to be useful.

Now that you've been properly warned, here's the info you need to use it:

- I used glut. I'll give you the dll.
- I used glew. I'll give you the dll.
- I used RenderTexture 1.0. No extra download necessary.
- I used Cg version 1.2.1. I will leave it to you to choose and install the version you want to use. I should warn you that you can get weird behavior if you try to use the executables with a different version of Cg.
- The system does not currently measure bits of significand. It only verifies what you tell it. I pre-compiled two versions: one with the default for ATI (16 bit significand, called Paranoia16.exe) and one with the default for NVIDIA (23 bit significand, called Paranoia23.exe). You can change the default using the arguements "mbits 23" for example.
- Both versions are "debug" versions. Annoyingly, they are single-threaded (default for VC6 project), even though GLUT is a multi-threaded target.
- I compiled these using VC6. I'll give you the .dsp file.
- **I am currently cleaning up the source code. Email me if you really want it soon.**

References

1. R. Karpinski. 1985. **Paranoia: A floating-point benchmark** [source code URL, the original is the version in BASIC]. *Byte Magazine 10*, 2 (Feb.), 223-235.
2. IEEE. 1987. **IEEE standard for binary floating-point arithmetic** [url]. *ACM SIGPLAN Notices 22*, 2 (Feb.), 9-25.
3. N. L. Schryer. 1981. **a test of a computer�s floating-point arithmetic unit** [url]. Tech. Rep. Computer Science Technical Report 89, AT&T Bell Laboratories, Feb.
4. D. Goldberg. 1991. what every **computer scientist should know about floating-point arithmetic** [ps]. ACM Computing Surveys, 23, 1 (March).
5. K. Hillesland, A. Lastra. **GPU floating-point paranoia** [pdf]. to appear in the proceedings of GP^2. 2004. [Note: This version has some minor corrections relative to proceedings.]