Debugging Floating Point Implementations on GPUs

Devon Yablonski
Mercury Computer Systems, Chelmsford, MA. dyablons@mc.com

Miriam Leeser
Northeastern University, Boston, MA. mel@coe.neu.edu

Goal

Discover where differences arise in scientific codes between CPU and GPU implementations and evaluate the effects on performance and accuracy.

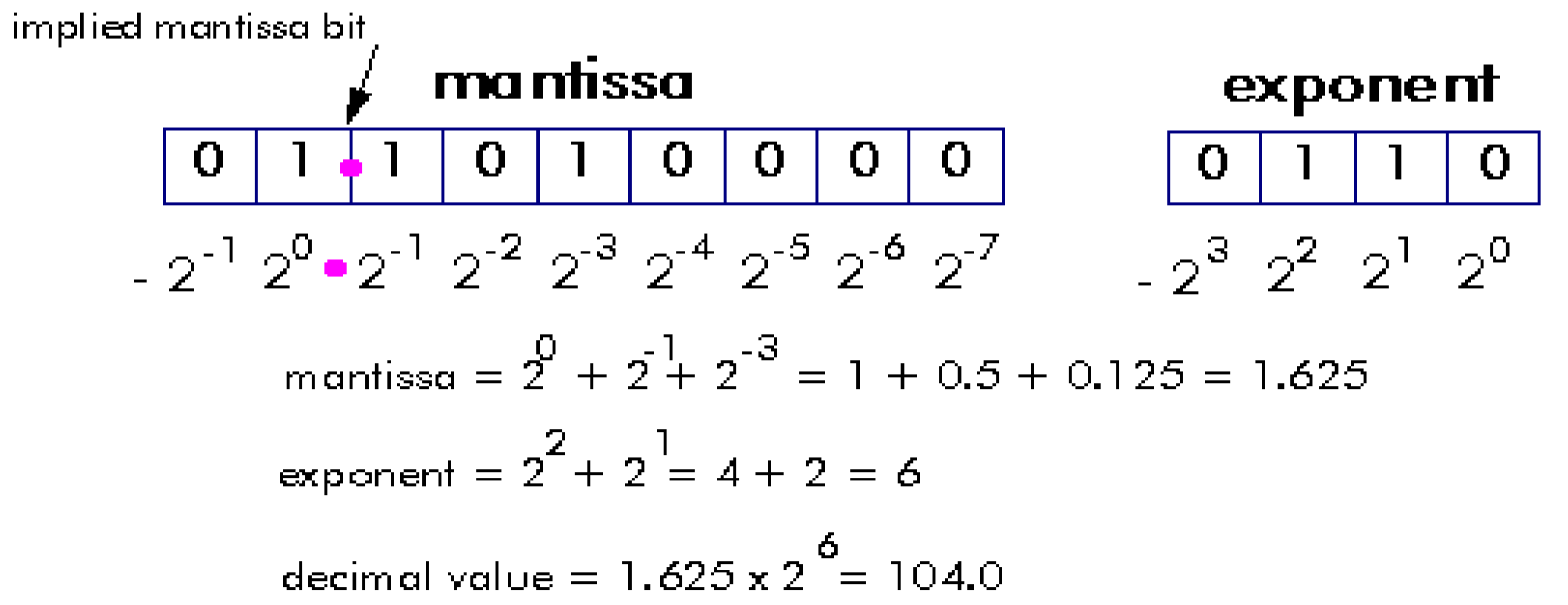
Myths about Debugging Scientific Code

Many scientists do not thoroughly debug their code or compare results between GPU and CPU due to the assumption that results will differ on different platforms. Our research shows that comparing the results helps to debug both CPU code and GPU code. By understanding the sources of differences, errors can be corrected. In the medical reconstruction code we studied, all differences were removed and a bug was fixed in the CPU version of the code.

[1] Devon Yablonski, “**Numerical Accuracy Differences in CPU and GPGPU Codes**” MS Thesis Northeastern Uni. Sept. 2011.

What is Floating-point?

Floating-point is the scientific notation format for binary values. It allows for very large and very small numbers to be represented with one data type. A number is represented by an exponent and a mantissa, usually normalized.



$$\text{mantissa} = 2^0 + 2^{-1} + 2^{-3} = 1 + 0.5 + 0.125 = 1.625$$

$$\text{exponent} = 2^2 + 2^1 = 4 + 2 = 6$$

$$\text{decimal value} = 1.625 \times 2^6 = 104.0$$

Fig. 1: Example of 12 bit floating-point. Our work uses 32 and 64 bit floatiing-point

Differences Between GPU and CPU code

Common sources of differences between GPU and CPU code

- Reordering of instructions: Floating-point is not associative
- Implementation of instructions
  - Fused Multiply Add is implemented in NVIDIA compute capability 2.0 and above, not (currently) in x86 CPUs
- Choices made by the compiler:
  - Fusing mult-add and reordering instructions are examples

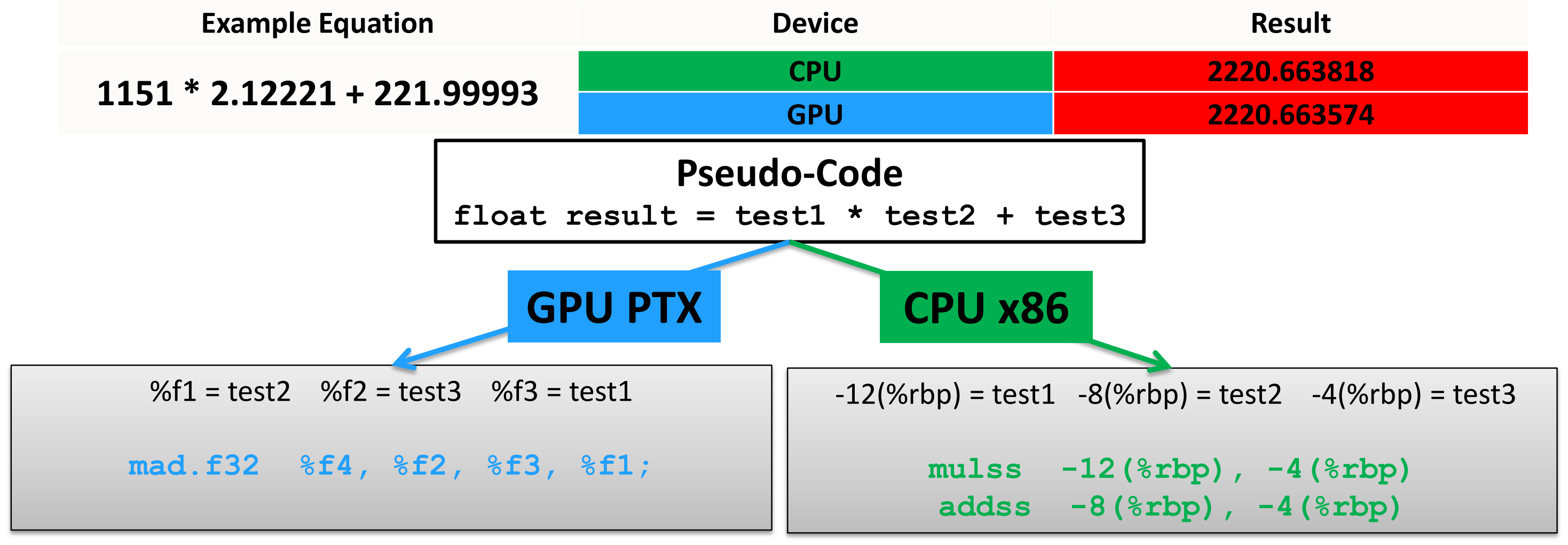
How to identify differences and debug your code:

- Compare intermediate results and identify where the divergence occurs.

Double precision does not fix the problems! It only makes them less apparent, which makes them harder to recognize.
Single precision is used for scientifically important code, even though NVIDIA assumes it is not -- scientists analyze range of values.

Compiler Induced Differences

- NVIDIA’s CUDA compiler aggressively combines double precision addition and multiplication into **floating point-multiply-add (FMAD)** instructions [3] on GT200 GPU hardware. The addition is truncated to be faster on the GPU which makes **the CPU more accurate.**



Example Equation	Device	Result
1151 * 2.12221 + 221.99993	CPU	2220.663818
	GPU	2220.663574

Pseudo-Code
```
float result = test1 * test2 + test3
```

GPU PTX
```
%f1 = test2   %f2 = test3   %f3 = test1

mad.f32  %f4, %f2, %f3, %f1;
```

CPU x86
```
-12(%rbp) = test1   -8(%rbp) = test2   -4(%rbp) = test3

mulss  -12(%rbp), -4(%rbp)
addss  -8(%rbp), -4(%rbp)
```

Fig. 2: Instruction level (IL) MAD code. PTX is GPU IL code.

- On Fermi, a **fused-multiply-add (FMA)** is used. The addition and multiplication only encounter a single rounding. **The GPU is more accurate.**

```
fma.f32  %f4, %f2, %f3, %f1;   Rounded 1x
```

```
mulss  -12(%rbp), -4(%rbp)
addss  -8(%rbp), -4(%rbp)      Rounded 2x!
```

Fig. 3: Instruction level (IL) FMA code. PTX is GPU IL code.

Test Example - Tomosynthesis

- Digital Breast Tomosynthesis (DBT) is a mammography algorithm that creates a 3D image from the data of 15 X-ray scans to aid in the search for cancerous tissues.
- Original code was developed by MGH and ported to the GPU at Northeastern by Schaa et al [4].

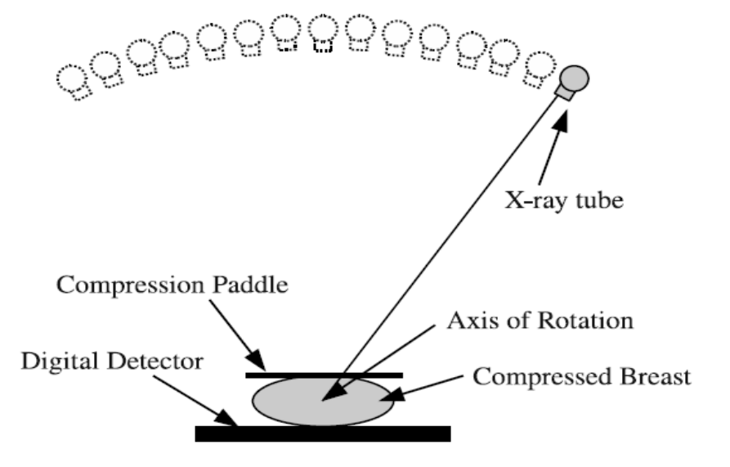
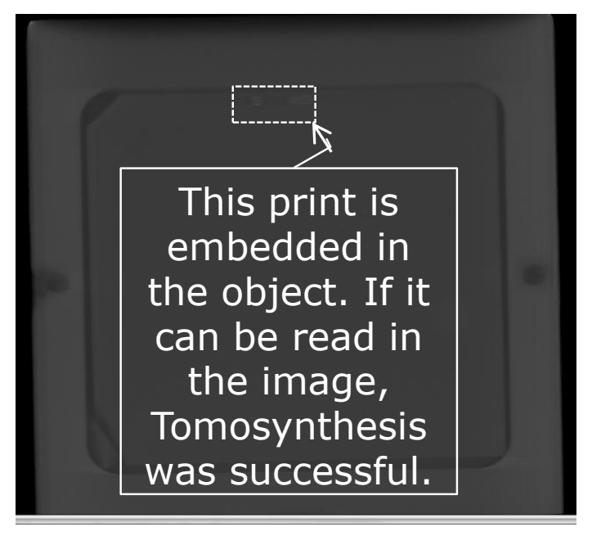


Fig. 8: 15 X-Rays scan the subject



This print is embedded in the object. If it can be read in the image, Tomosynthesis was successful.

Fig. 9: Result from a test dataset used for this work

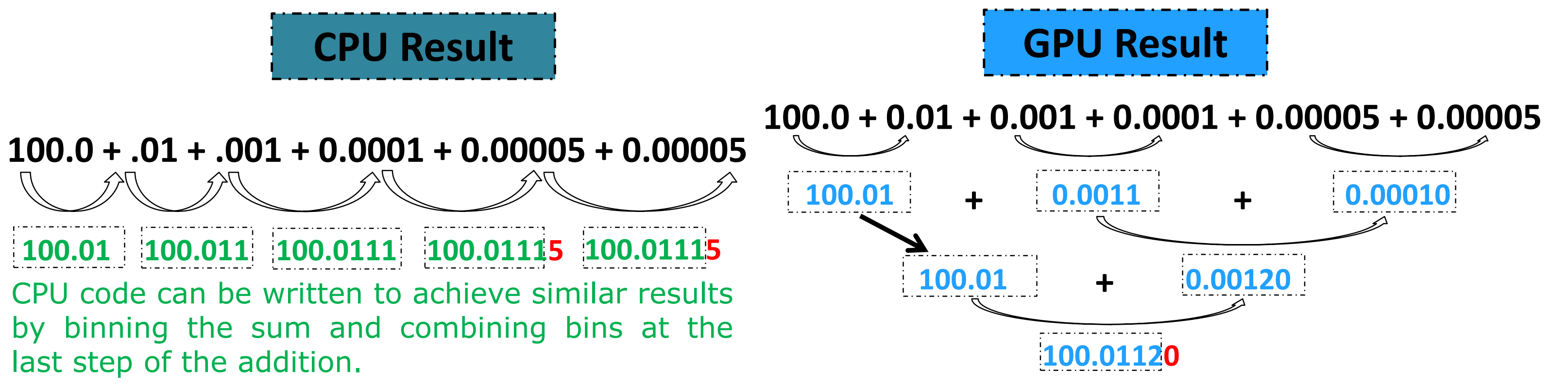
Table 2: Code Performance* for 8 iterations of Tomosynthesis

Implementation	Time	Speedup
CPU Intel XEON W3580 @ 3.33GHz	29min 47s	-
GPU NVIDIA C1060	19s	**97x**

* These results are based on our test setup and may differ from published results of Schaa et al.

Associativity – Accumulation vs. Reduction

Manycore introduces **new concepts of programming**. For example, accumulating a large number of values on a serial CPU may involve a serial loop. Once the CPU sum gets large, it does not have enough precision to reflect the addition of a small value. The GPU bins its results by doing a reduction - consistently adding similar sized values. **The GPU is faster AND more accurate!**



CPU code can be written to achieve similar results by binning the sum and combining bins at the last step of the addition.

Fig. 4: CPU serial addition compared with tree reduce GPU version

Case Study: Pi

```
static long num_steps = 100000;
void main()
{
    int i;  float x, pi, sum = 0.0;

    step = 1.0/(float) 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;
}
```

Fig. 5: Pi Calculation Pseudo-code

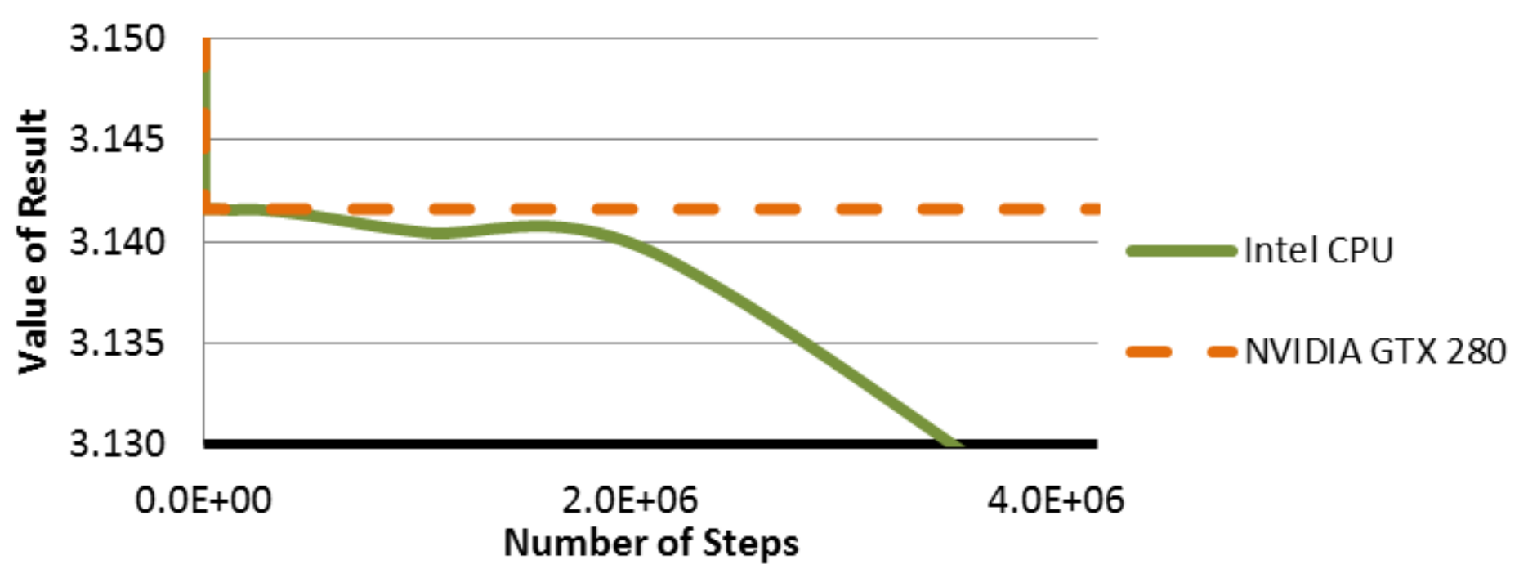


Fig. 6: Pi Calculated Value (per number of steps)

Results of CPU and GPU Tomosynthesis

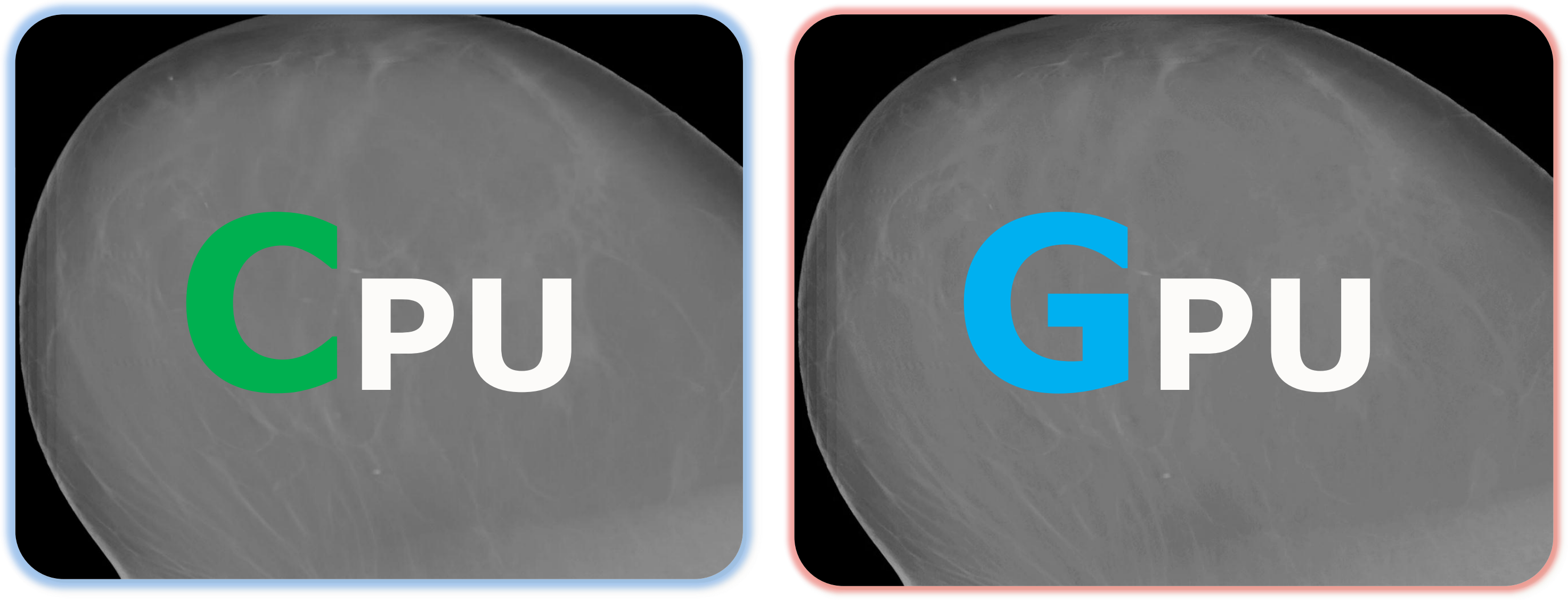


Fig. 10: Image results from CPU and GPU tomosynthesis implementations look nearly identical

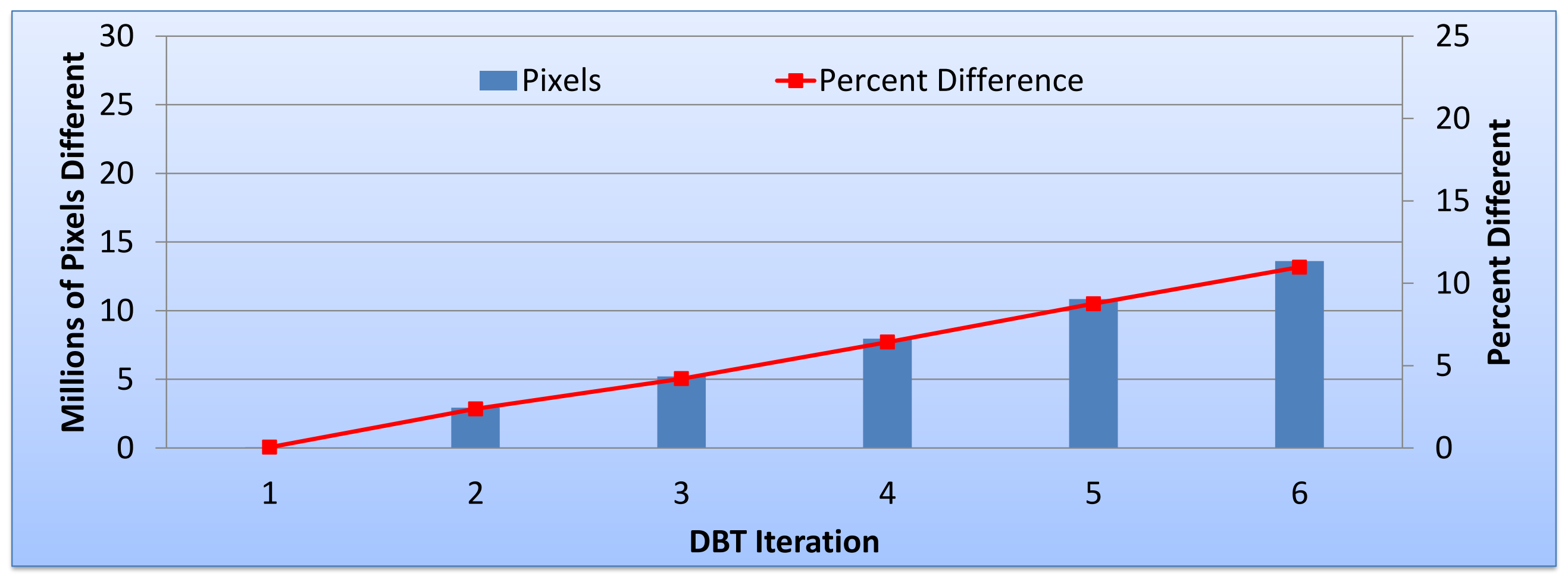


Fig. 11: Numerically, the images differ in over 10% of the pixels after 6 iterations of DBT

Fixed-point to Floating-point Boundaries

- A common source of errors is the boundary between floating and fixed point code. Floating point values are often cast to integers for image display. Errors are easily introduced. Correct casting should be used.

CPU
float NegFloat = -69.235
Unsigned short ShortResult = (unsigned short) NegFloat
ShortResult = 65467

GPU
float NegFloat = -69.235
unsigned short ShortResult = (unsigned short) NegFloat
ShortResult = 0

BEST
Float NegFloat = -69.235
CPU: unsigned short ShortResult = (unsigned short) (long) NegFloat
GPU: unsigned short ShortResult = (unsigned short)__float2int_rn(NegFloat)
ShortResult = 0

Fig. 7: Casting, GPU vs. CPU

- **Different modes of division** - The default on compute capability 1.3 GPUs is approximate division in single precision. Using default conditions, **the CPU is more accurate.**

Table 1: CPU and GPU results from a simple floating-point division

Example Equation	Device	Instruction	Result
$\frac{5.0}{0.0190440360456705}$	CPU	`divss-40(%rbp), %xmm0`	262.549377
	GPU	`div.full.f32 %f3, %f1, %f2;`	262.549407
	Matlab Extended Precision	`vpa(5.00/0.0190440360456705, 30)`	262.5493875…

Performance

- We match the GPU code to the CPU code in single and double precision. Each change lowers total number of differences shown in Fig. 11.

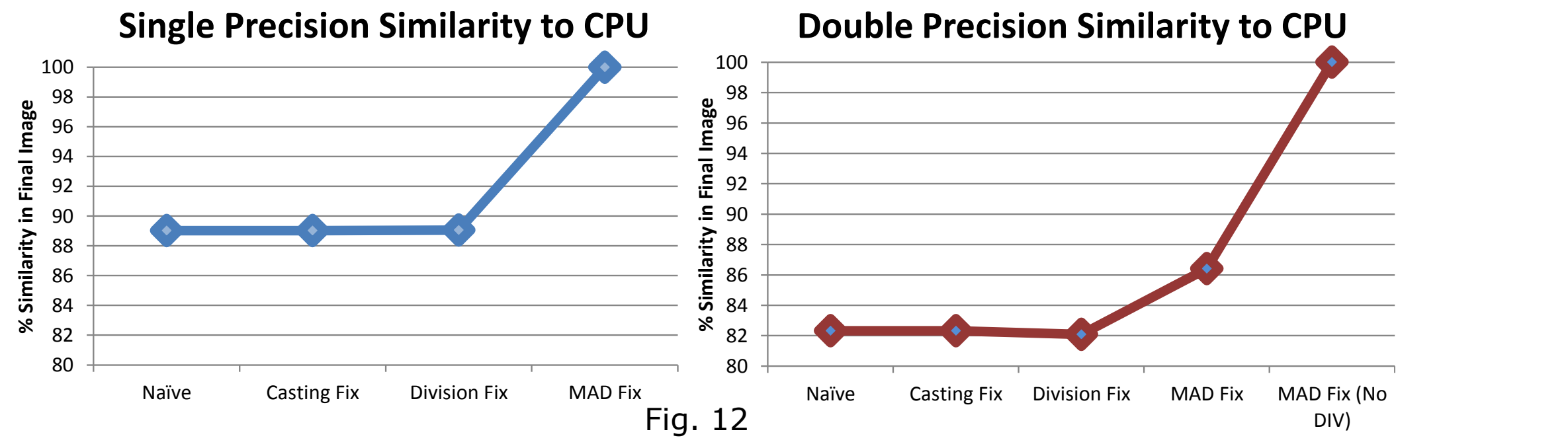


Fig. 12

- We created GPU code that produces **identical results** to the CPU code.

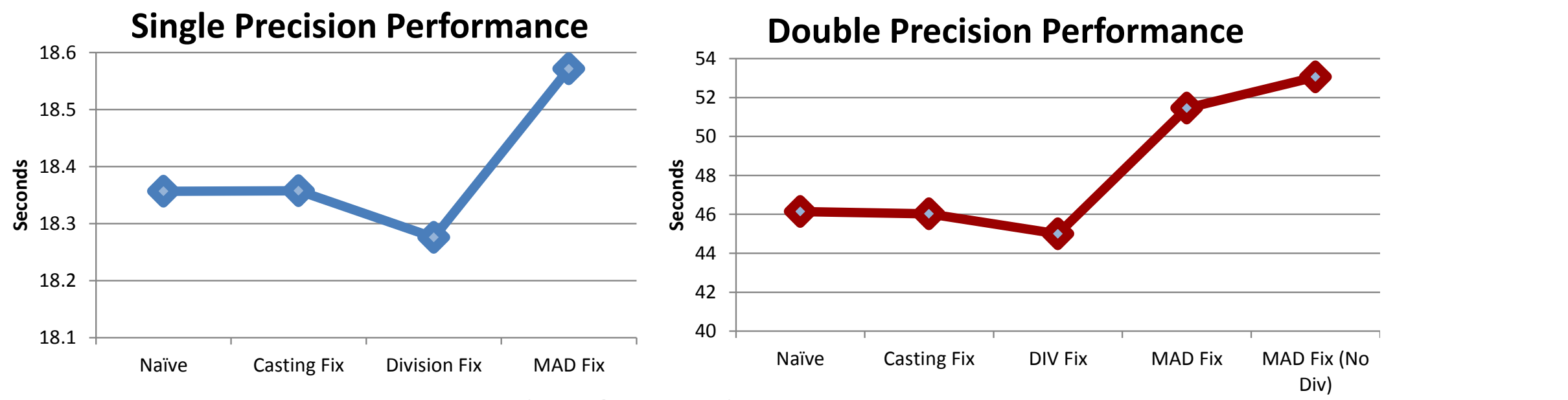


Fig. 13: Some fixes hurt performance, some improve it!

Future Work

- Provide better debugging tools for floating point implementations on GPUs.

Acknowledgement: This work was supported in part by the National Science Foundation Engineering Research Centers Innovations Program (Award Number EEC-0946463)

[1] "IEEE Std 754-2008," *IEEE Std 754-2008,* pp. 1-58, 29 August 2008.
[2] N. Whitehead, A. Fit-Florea. “Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs”. NVIDIA, 2011.
[3] NVIDIA, “NVIDIA CUDA Programming Guide,” http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Programming_Guide.pdf
[4] D. Schaa, B. Jang, P. Mistry, R. Dominguez, D. Kaeli, R. Moore, D. B. Kopans, "GPU Acceleration of Iterative Digital Breast Tomosynthesis," *GPU Gems 4*