Northeastern

# *How do you know your GPU or manycore program is correct?*


Prof. Miriam Leeser
Department of Electrical and Computer Engineering
Northeastern University
Boston, MA
mel@coe.neu.edu

Northeastern

# Typical Radar Processing

Raw Radar Data

Range

Cross-range

2-D Fourier Matched Filtering

**Range Processing:**

FFT → Matrix Multiply (Convolution)

**Cross-Range Processing:**

Matrix Multiply → FFT
Bandwidth Expansion → $FFT^{-1}$
Matrix Multiply → FFT

Spatial Frequency Interpolation

**Range Loop:**

Polar to Rectangular Conversion

**Transform back to Spatial Domain:**

2-D $FFT^{-1}$

Final Image

Range

Cross-range

SSAR Block Diagram

From http://www.codesourcery.com/vsiplplusplus/ssar_whitepaper.pdf

Northeastern

# Typical Radar Processing

Raw Radar Data

Range

Cross-range

fix2float

2-D Fourier Matched Filtering

**Range Processing:**

FFT → Matrix Multiply (Convolution)

**Cross-Range Processing:**

Matrix Multiply → FFT
Bandwidth Expansion → $FFT^{-1}$
Matrix Multiply → FFT

Spatial Frequency Interpolation

**Range Loop:**

Polar to Rectangular Conversion

**Transform back to Spatial Domain:**

2-D $FFT^{-1}$

float2fix

Final Image

Range

Cross-range

SSAR Block Diagram

From http://www.codesourcery.com/vsiplplusplus/ssar_whitepaper.pdf

# What everyone knows about floating point

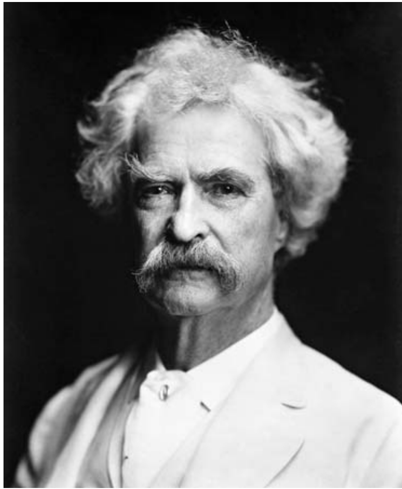
- Floating point is not associative
  - I will get different results on different manycore processors
  - If the few least significant bits are different it doesn’t really matter

- As long as I check the floating point parts of my problem and it is correct within a few least significant bits, then I am fine

Northeastern

# Knowledge

"The trouble with the world is not that
people know too little,
but that they know so many things
that ain't so."



Mark Twain

http://userweb.port.ac.uk/~joyce1/abinitio/images/twain3.jpg

# Analysis of Image Reconstruction

- A biomedical imaging application
  - Image reconstruction for tomosynthesis
- Similar to radar processing
  - Sensor data is fixed point
  - Processing done using floating point data types
  - Convert to integer to display on a screen
- Results:
  - We were able to make CPU and GPU versions match exactly
    - Bit by bit
  - Differences in original code due to casting from floating point to integer as well as with floating point arithmetic

Northeastern

# Outline

- A brief introduction to floating point on NVIDIA GPUs
- DBT: Digital Breast Tomosynthesis
  - Comparing CPU and GPU implementations
- Floating Point and associativity
  - Calculating $\pi$
- Lessons learned

Northeastern

...?

**Scientific Notation**

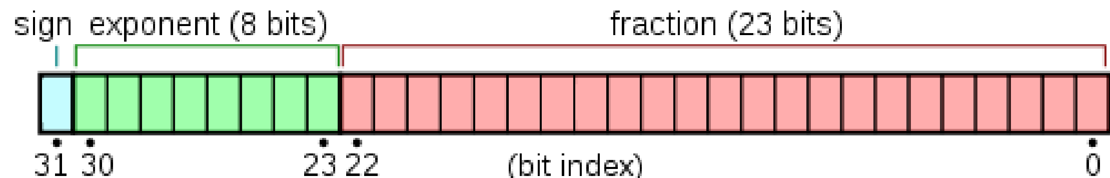
$$1.00195 * 10^{-10}$$

**Floating-point Notation**

sign bit

$$\overbrace{0}^{\text{sign bit}} \underbrace{\cdot 101010101}_{\text{mantissa}} \times \underbrace{010101}_{\text{exponent}}$$

**32 bit Floating-point Storage (float)**






Images from http://upload.wikimedia.org/, www.gleezorp.com and Bores Signal Processing

Northeastern

# Floating Point Correctness

- Most published work on floating point correctness predates the release of double precision GPUs
- Now GPUs support double precision
  - Do we need to worry about correctness anymore?
- Correctness of individual operations
  - add, sub, div, sqrt
  - These are all (?) IEEE Floating Point Compliant
    - Does that mean that code produces the same answer on every CPU or GPU that is IEEE Floating Point Compliant?

Northeastern

# What you need to know about NVIDIA GPUs and Floating Point

- Each NVIDIA board has an associated "compute capability"
  - This is different from the version of the software tools
- NVIDIA introduced double precision support with compute capability 1.3
  - They assumed that if you used single precision you didn't care about the precision of your results
  - In compute capability 1.3, single precision is *NOT* IEEE compliant. Double precision *is* IEEE compliant.
- In compute capability 2.0 and later, single precision is also IEEE compliant
- For more information, see:
  - http://developer.download.nvidia.com/assets/cuda/files/NVIDIA-CUDA-Floating-Point.pdf

# Why do you care about compute capability?

- You need to know what compute capability your hardware has to know what your floating point results will be
- NVIDIA (incorrectly) assumed that if you were using single precision you didn't care about IEEE compliance
  - If you analyze the dynamic range of your calculations based on your sensor data, then choose single precision because that is what you need for your application, you may get imprecise results
- NVIDIA fixed this in compute capability 2.0 and higher

Northeastern

# Digital Breast Tomosynthesis (DBT)

Real world – Real problem

CPU

GPU

X-ray tube

Compression Paddle

Axis of Rotation

Digital Detector

Compressed Breast

30 Minutes

18 Seconds

**~100x Speedup!**

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*

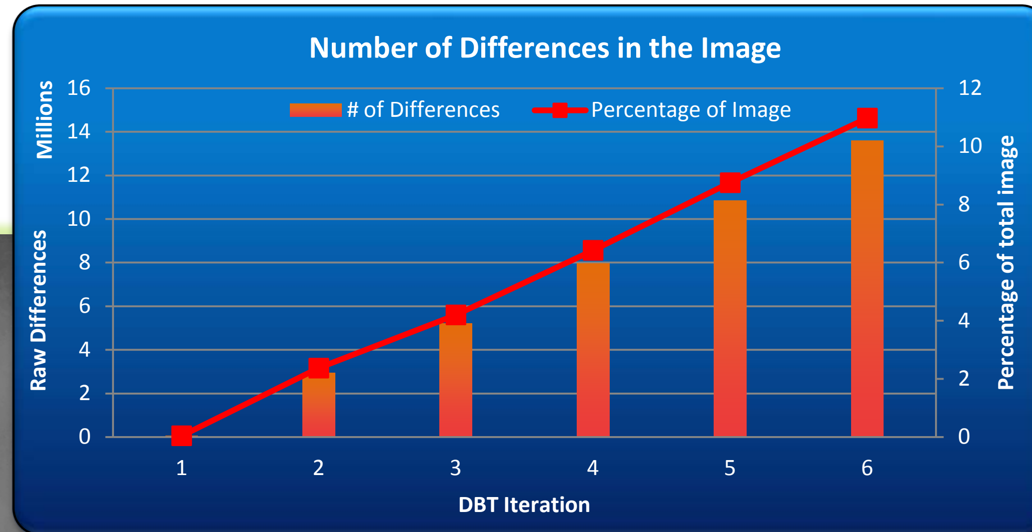
Northeastern

# Digital Breast Tomosynthesis

Not so fast…

CPU

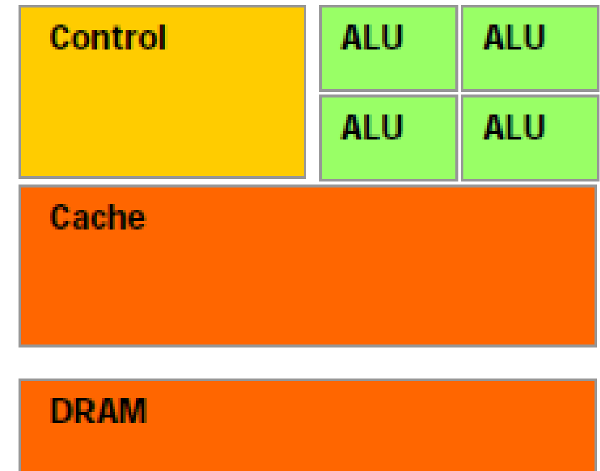
GPU

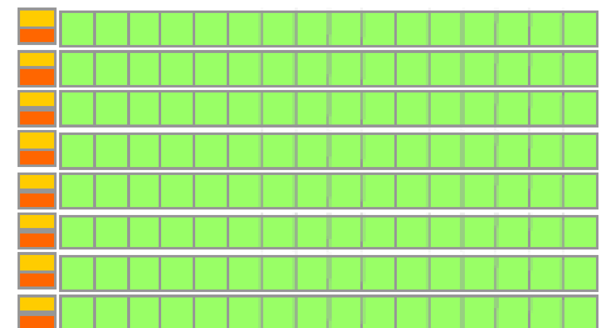



D. Schaa, B. Jang, P. Mistry, R. Dominguez, D. Kaeli, R. Moore, D. B. Kopans, "GPU Acceleration of Iterative Digital Breast Tomosynthesis," *GPU Gems 4*

# Experimental Setup

- CPU
  - Intel Xeon W3580 (4 core, 8KB L1 cache)
  - 6GB system RAM
- GPU
  - NVIDIA Tesla C1060 (GT 200)
    - Compute 1.3 (double precision capable)
    - 240 Cores, 933GFLOP/s
    - 4GB RAM, 102MBit/s



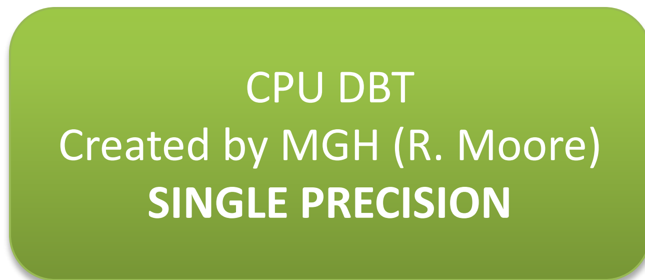



# Digital Breast Tomosynthesis

```
initialize 3D volume {make a guess}
while likelihood is too low do
 Forward Projection
  for each projection do
    for each ray do
      trace ray using volume guess
      compute new projection value
      compute attenuation value
    end for
  end for
 Backward Projection
  for each voxel in the 3D volume do
    for each X-ray source do
      compute error of ray (projection) with the
          measured value (sinogram)
      use attenuation values to refine volume guess
    end for
  end for
end while
```

# DBT Test Setup

Originally, 2 versions of SP DBT were provided…



Experimental setup combined CPU and GPU codes to debug…

Combined CPU and GPU code to allow testing.

SINGLE PRECISION

Combined CPU and GPU code to allow testing.

DOUBLE PRECISION

x86-64 Assembly

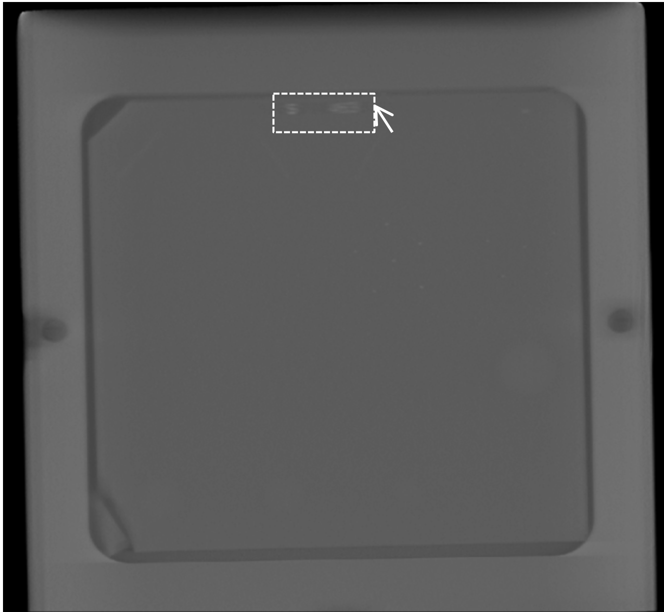
GCC

NVCC

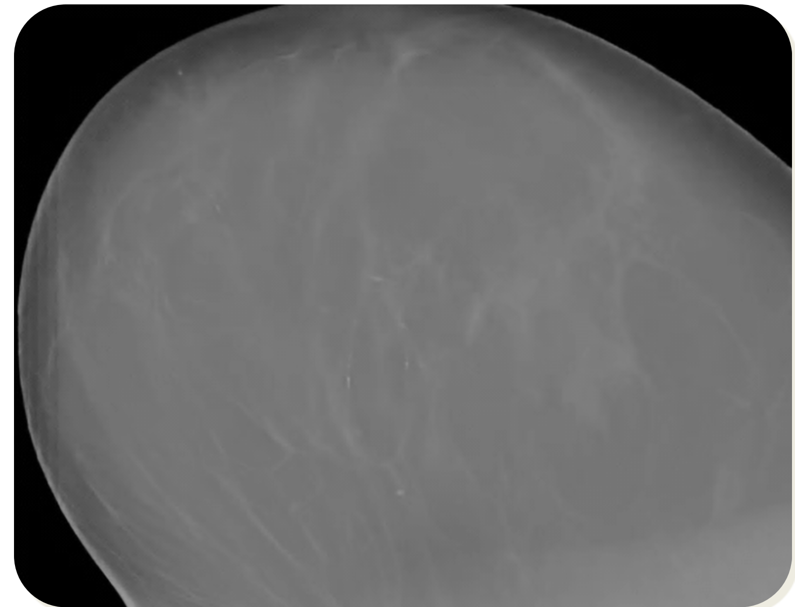
NVIDIA PTX

# DBT Inputs

- American College of Radiologists Phantom

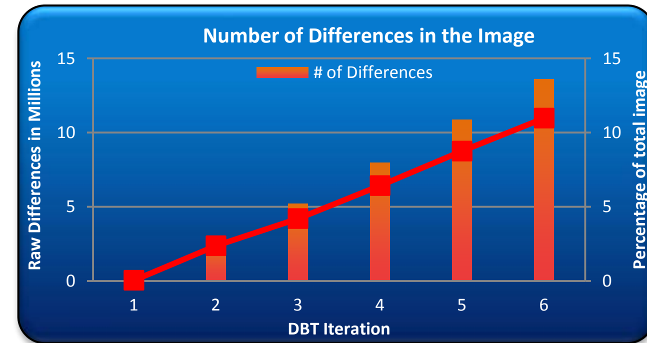


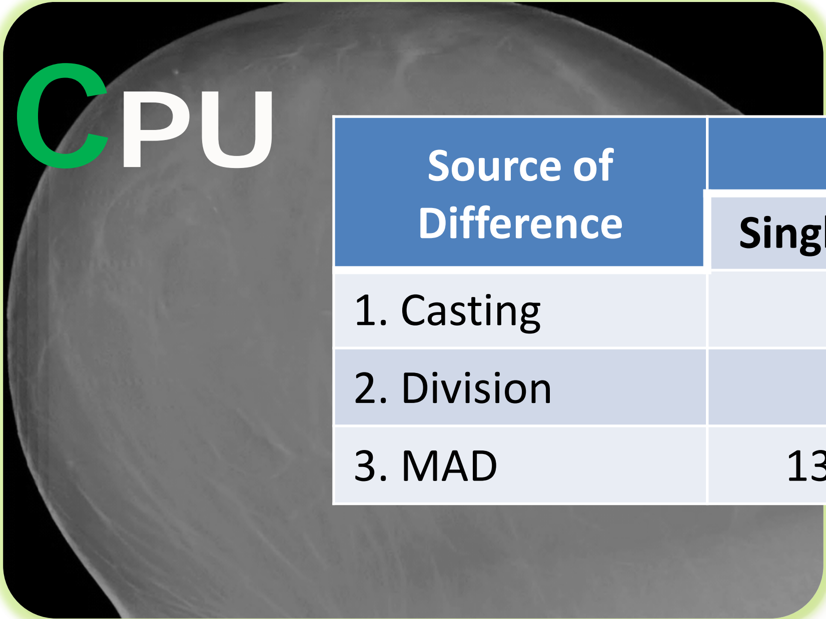
- Breast Tissue Phantom

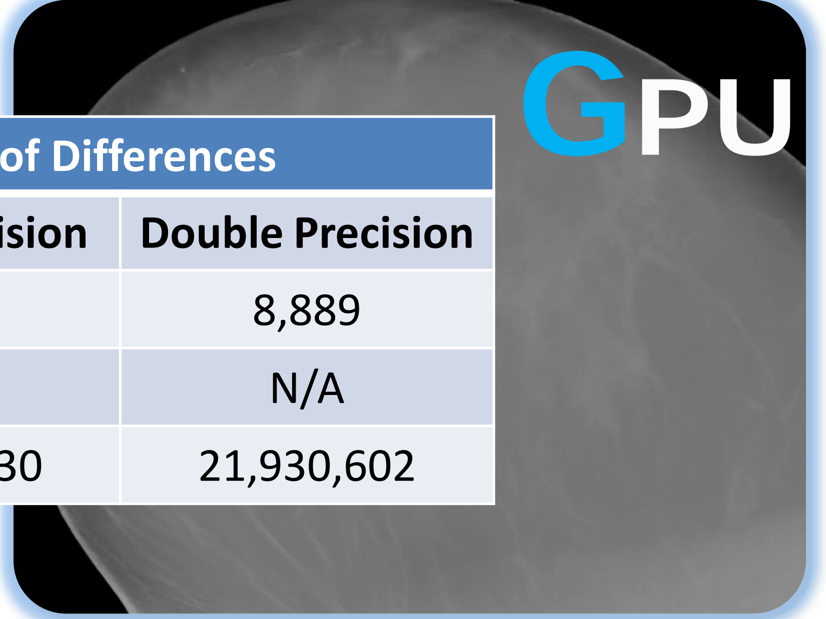


# Why the Tomosynthesis Results Don't Match

- Floating-point differences
  1. Converting floating-point incorrectly
  2. Non-IEEE compliant division by default in CUDA
  3. CUDA combines multiply and add into MAD




| Source of Difference | # of Differences | |
|---|---|---|
| | Single Precision | Double Precision |
| 1. Casting | 2,978 | 8,889 |
| 2. Division | 50,958 | N/A |
| 3. MAD | 13,608,730 | 21,930,602 |

Northeastern

# DBT: Casting Error

- CUDA GPU code is compiled using NVIDIA compiler
- CPU (C++) code is compiled in g++ for x86-64 target
- Naïve code can be handled differently in CUDA vs. gcc
- Casting is one example:
  - NVIDIA provides special functions, C++ has functions and/or rules

**Code**

```
float NegFloat = -69.235
unsigned short ShortResult = (unsigned short) NegFloat
```

**CPU**
ShortResult = 65467

**GPU**
ShortResult = 0

**BEST**

CPU: unsigned short ShortResult = (unsigned short) (long) NegFloat

GPU: unsigned short ShortResult = (unsigned short)__float2uint_rn(NegFloat)

ShortResult = 0



***Caused approximately 3,000 differences in DBT!***

# DBT: Casting Error

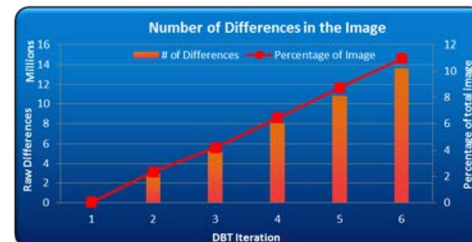
Is this important?

**Ariane 5 Rocket**

- 10+ years of research
- 7 billion dollars for development

First Launch...

- 15 years later, DBT.



# Lessons from Casting

- This problem has been around for a long time, yet programmers still get it wrong

  **unsigned short ShortResult = (unsigned short) NegFloat**

  - This is the naïve but natural way to program the cast
  - The compiler gives a warning about datatypes that the programmer ignores
    - He knows a cast is a dangerous operation which is what he thinks the warning is telling him
  - You cannot ignore the boundary between integer and floating point … yet all correctness publications do!
- The most errors, and the worst errors were due to this problem in DBT

# DBT: Division

A / B = **C**?

- **GT200** (compute 1.x) series uses reciprocal division in single precision (2 ULP error, but fast)

| Instruction (PTX) | Operation (CUDA) | IEEE |
|---|---|---|
| div.approx.f32 | __fdividef(x,y) | ✗ |
| div.full.f32 | x/y | ✗ |
| div.r[n,z,u,d].f32/f64 | __fdiv_r[n,z,u,d](x,y) | ✓ |

SPEED

ACCURACY

= Default

***Caused approximately 50,000 differences in DBT!***

# DBT: Division

CPU and GPU results from a simple floating-point division

| Example Equation | Device | Instruction | Result |
|---|---|---|---|
| $\frac{5.0}{0.01904403604}$ | CPU | `divss -40(%rbp), %xmm0` | 262.549377 |
| | GPU | `div.full.f32 %f3, %f1, %f2;` | 262.549407 |
| | Matlab | `vpa(5.00/0.01904403604, 30)` | 262.5493875... |

# GPU: Division

| | | Default SP | | Default DP |
|---|---|---|---|---|
| **GPU:** | *div.approx.f32* | *div.full.f32* | *div.f32* | *div.rn.f64* |
| **CPU:** | | | *divss* | *divsd* |

| Instruction | Execution time in milliseconds |
|---|---|
| Approx(fast) SP | 54.16 |
| Approx SP | 84.90 |
| IEEE SP | 962.34 |
| IEEE DP | 1,854.86 |
| CPU IEEE SP | 15,983.86 |
| CPU IEEE DP | 24,955.41 |

# GT200 Multiply-ADD (FMAD)

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

**Pseudo-Code**

```
float test1 = 1151.0
float test2 = 221.99993
float test3 = 2.12221
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)
```

***Caused approximately 13,000,000 differences in DBT!***

# DBT: MAD

**X= Y*W+Z**

- GT200 (NVIDIA compute capability 1.3)
  - Compiler combines add and multiply for speed
    - FMAD – Floating-point Multiply-Add
    - SP: Truncates between multiply and add (less precise)
    - DP: IEEE
- Fermi
  - Compiler combines add and multiply for speed
    - FFMA –Floating-point Fused Multiply-Add
      - IEEE 754-2008 compliant
    - More precise - Single rounding for two instructions

Northeastern

# GPU: MAD

| | | *Default SP* | | *Default DP* |
|---|---|---|---|---|
| **GPU:** | *mul.rn.f32 & add.f32* | *mad.f32* | *mul.rn.f64 & add.rn.f64* | *mad.rn.f64* |
| **CPU:** | *mullss & addss* | | *mulsd & addsd* | |

# DBT Modifications vs. Execution Time

**Single Precision Performance**

Seconds

18.6
18.55
18.5
18.45
18.4
18.35
18.3
18.25
18.2
18.15
18.1

Naïve
Cast
DIV
FMAD

# Fermi (Compute 2.x)

- Division IEEE 754-2008 compliant by default
- FMA available in both single and double precision (IEEE)
  - Still need to manually remove these to match CPU
  - CPUs are currently behind NVIDIA GPUs in IEEE compliance
- Compiler options for division modes

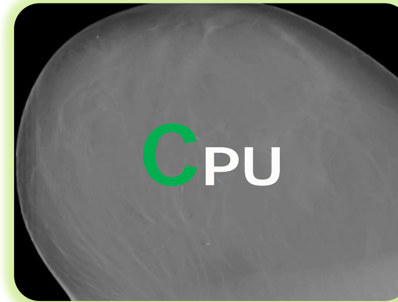
# Different Versions of DBT

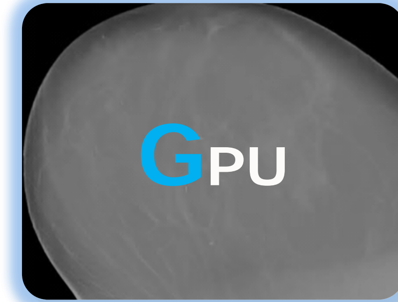
| | Naïve | CPU Matching | Correct | Fastest | Fermi |
|---|---|---|---|---|---|
| **Casting** | ERROR | FIX | FIX | FIX | FIX |
| **SP Division** | Approx | Use IEEE (__fdiv_rn) | Use IEEE (__fdiv_rn) | IEEE? | IEEE |
| **DP Division** | IEEE | IEEE | IEEE | -- | IEEE |
| **SP MAD** | Approx | Restrict (__fadd_rn) | Restrict (__fadd_rn) | Use Default (Approx) | Use Default (FMA) |
| **DP MAD** | IEEE | Restrict (__fadd_rn) | Use Default (IEEE) | Use Default (IEEE) | Use Default (FMA) |

Northeastern

# DBT Case Study Conclusions

- Some CUDA codes can match CPU
  - Safe to use
  - True speedup
  - Are results correct?






- Programmers need to have a renewed awareness of floating-point
  - Division
  - MAD
  - Type conversions

- Analysis like this on image reconstruction code is important
- There are no associativity issues with the DBT case study

# Associativity Example: Sequential Computation of π

```
Static  long  num_steps = 100000;
  float step;
  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;
       }
```

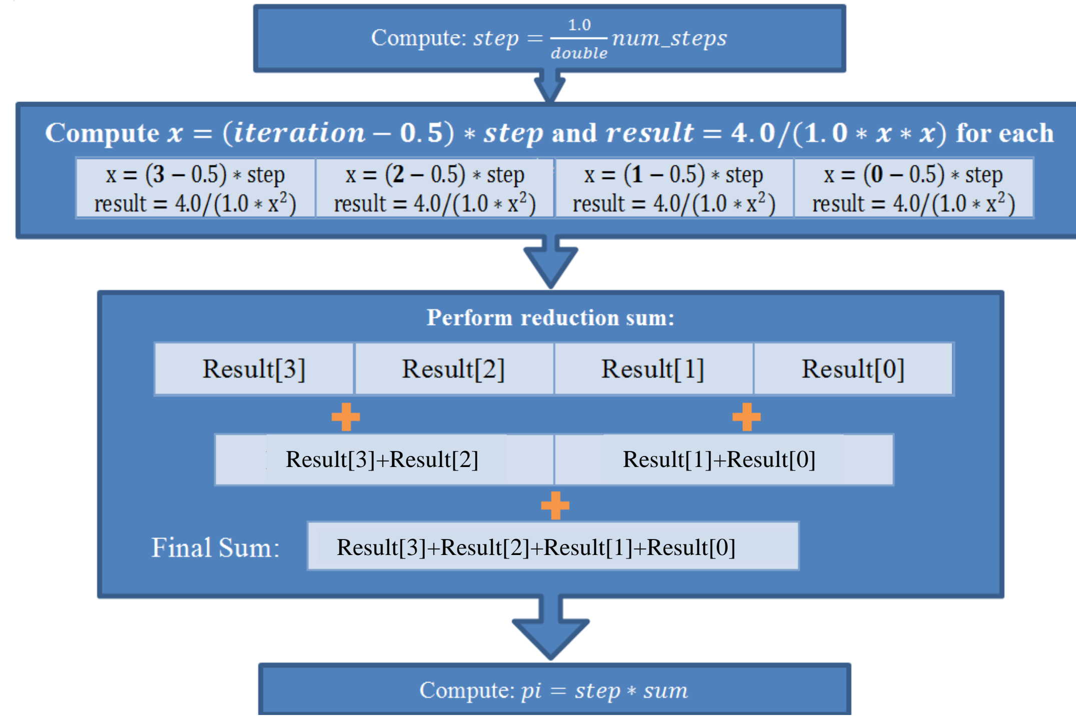
- From OpenMP tutorial given at SC'99 by Tim Mattson and Rudolf Eigenmann: http://www.openmp.org/presentations/sc99/sc99_tutorial.pdf

# Parallel Computation of $\pi$

## CPU

```
1:   static long num_steps = 100000;
2:   float step;
3:
4:   void main()
5:   {    int i;  float x, pi, sum = 0.0;
6:
7:        step = 1.0/(float) num_steps;
8:
9:        for (i=1; i<= num_steps; i++) {
10:            x=(i-0.5)*step;
11:            sum = sum + 4.0/(1.0+x*x);
12:       }
13:       pi = step * sum;
14:  }
```

## GPU




GPU and CPU are using different implementations: rewriting code

# What is this code doing?

```
1:    static long num_steps = 100000;
2:    float step;
3:
4:    void main()
5:    {    int i;  float x, pi, sum = 0.0;
6:
7:         step = 1.0/(float) num_steps;
8:
9:         for (i=1; i<= num_steps; i++) {
10:             x=(i-0.5)*step;
11:             sum = sum + 4.0/(1.0+x*x);
12:        }
13:        pi = step * sum;
14:   }
```

- It is performing an integral of the of the derivative of the arctangent
- The derivative of the arctangent is $1/(1+x^2)$
- A definite integral from 0 to 1 gives

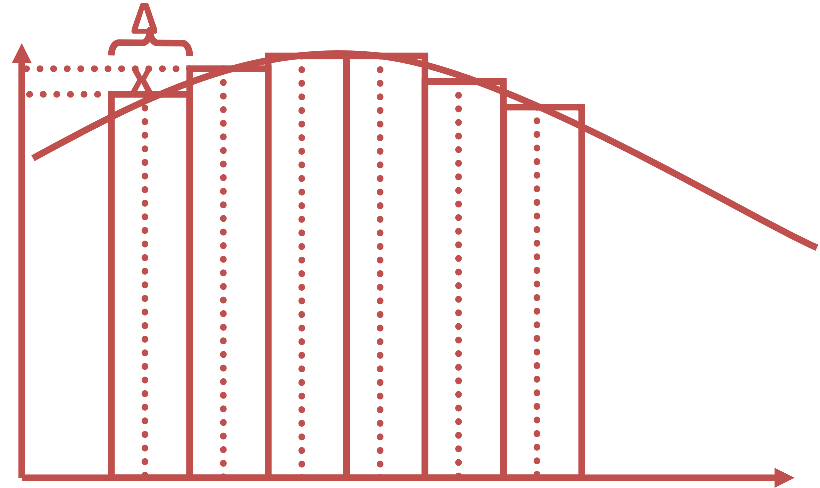
  arctan(1) - arctan(0)

  – which is pi/4 - 0 or just pi/4
- The code folds the factor of 4 into the integral

# Approximating an Integral

- The code is doing a Riemann sum approximation of the integral using segment midpoints with num_steps segments.

```
Static  long  num_steps = 100000;
  float step;
  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;
        }
```




- What I learned in high school math:
  - The more rectangles, the closer to the exact solution

Northeastern

# Calculating π in single precision

**Accuracy of Pi Result**

Value of Result

3.150
3.145
3.140
3.135
3.130

0.0E+00 1.0E+06 2.0E+06 3.0E+06 4.0E+06

**Number of Steps**

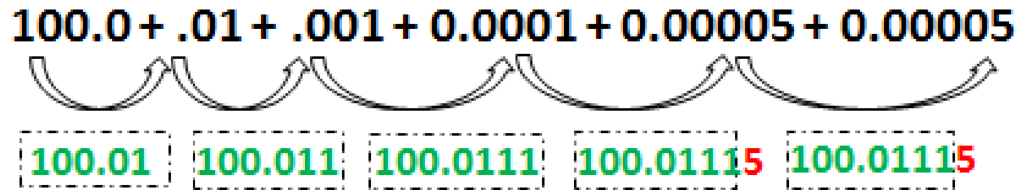
Intel CPU

NVIDIA GTX 280

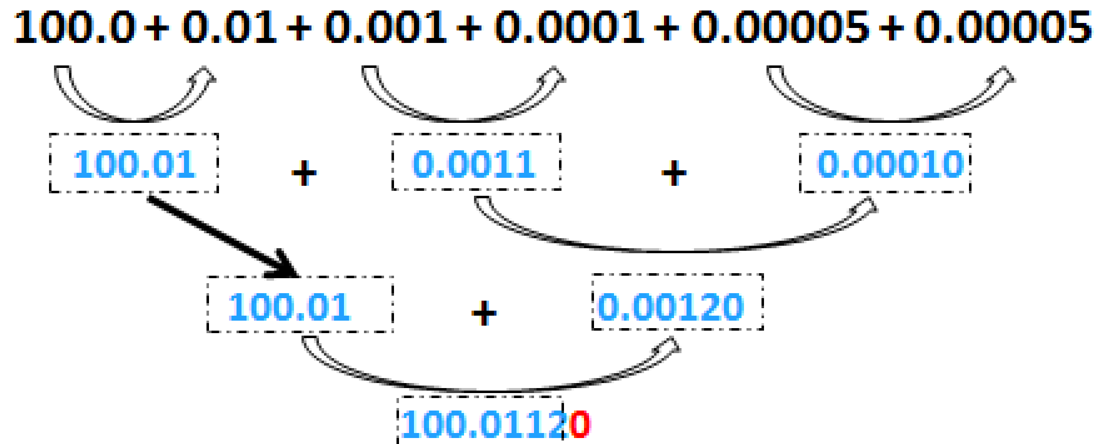
Northeastern

# Floating-point is NOT Associative

**CPU Result**




**GPU Result**




**Correct answer = 100.0112**

# Lessons from π example

- Floating point is not associative
  - Order of operations matter
- CPU code is “hopelessly naïve”:
  - Should accumulate CPU code using binning, etc.
  - Why is this code part of a parallel programming tutorial?
- Assumption: all FP operations are implemented correctly
  - This has nothing to do with IEEE compliance!
  - Verifying operations is important but does not address this problem
- This is not due to differences in rounding modes
  - Both versions use same rounding modes
- Double precision makes these issues harder to see
  - They don’t go away
- Massive parallelism:
  - Naïve way of writing code reorders operations
- *GPU code may be more accurate than CPU !*

Northeastern

# Conclusions

- Need to know whether or not your hardware is IEEE compliant
- Need to check end-to-end correctness of applications
  - Errors arise due to conversions as well as due to calculations
- Need to decide how important correctness is to your application
  - Can identify sources of differences between different implementations
  - Can make different implementations on different machines identical in some cases
- Sequential code is not necessarily the ``gold standard''
  - Parallel code may be more correct

# Thank You!

Miriam Leeser
mel@coe.neu.edu
http://www.coe.neu.edu/Research/rcl/index.php

Devon Yablonski did the DBT case study for his MS Thesis, available from:

http://www.coe.neu.edu/Research/rcl/publications.php#theses

This project is supported by The MathWorks and by the National Science Foundation Award: *A Biomedical Imaging Acceleration Testbed (BIAT)*