Accelerating GPU computation through mixed-precision methods

Michael Clark
Harvard-Smithsonian Center for Astrophysics
Harvard University

Outline

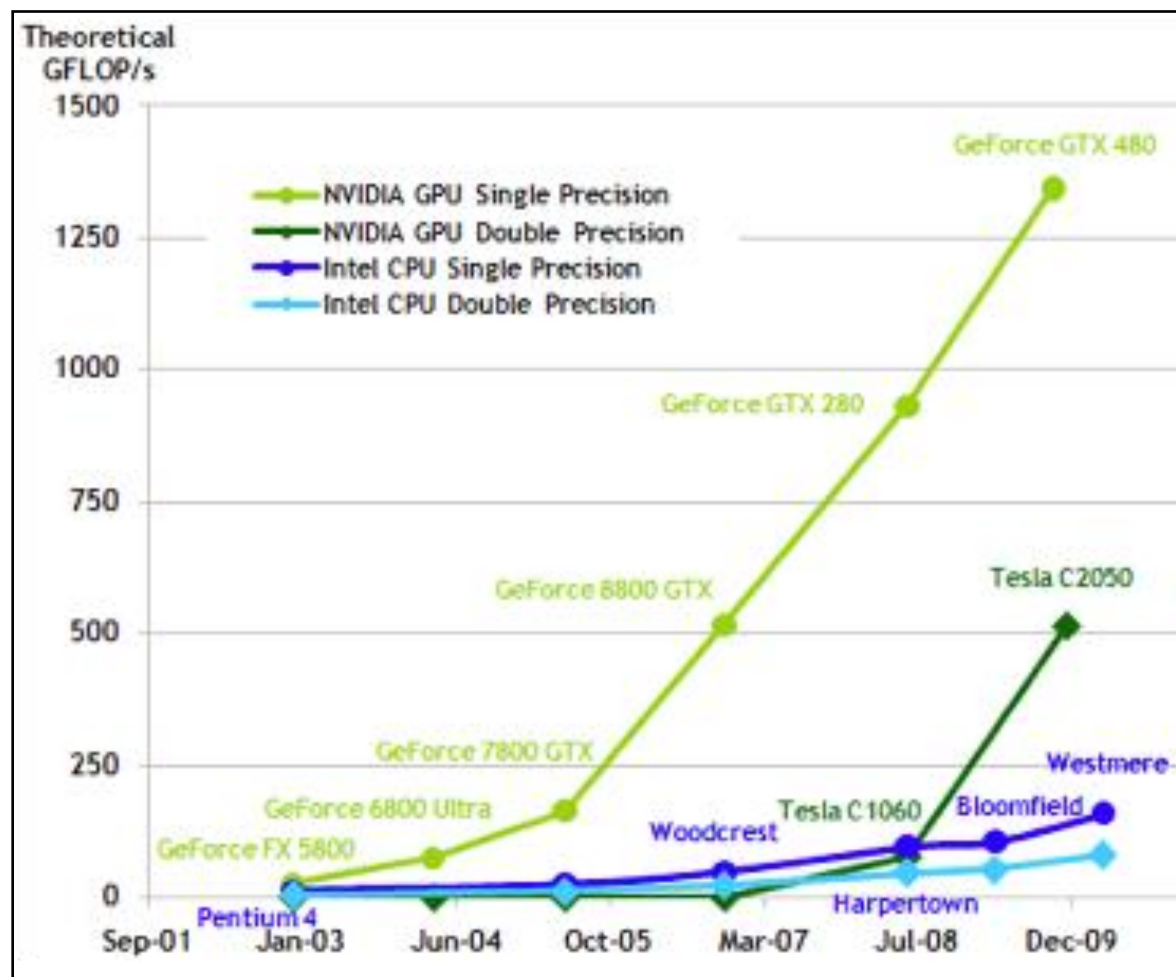
- Motivation
- Truncated Precision using CUDA
- Solving Linear Equations
- Conclusion

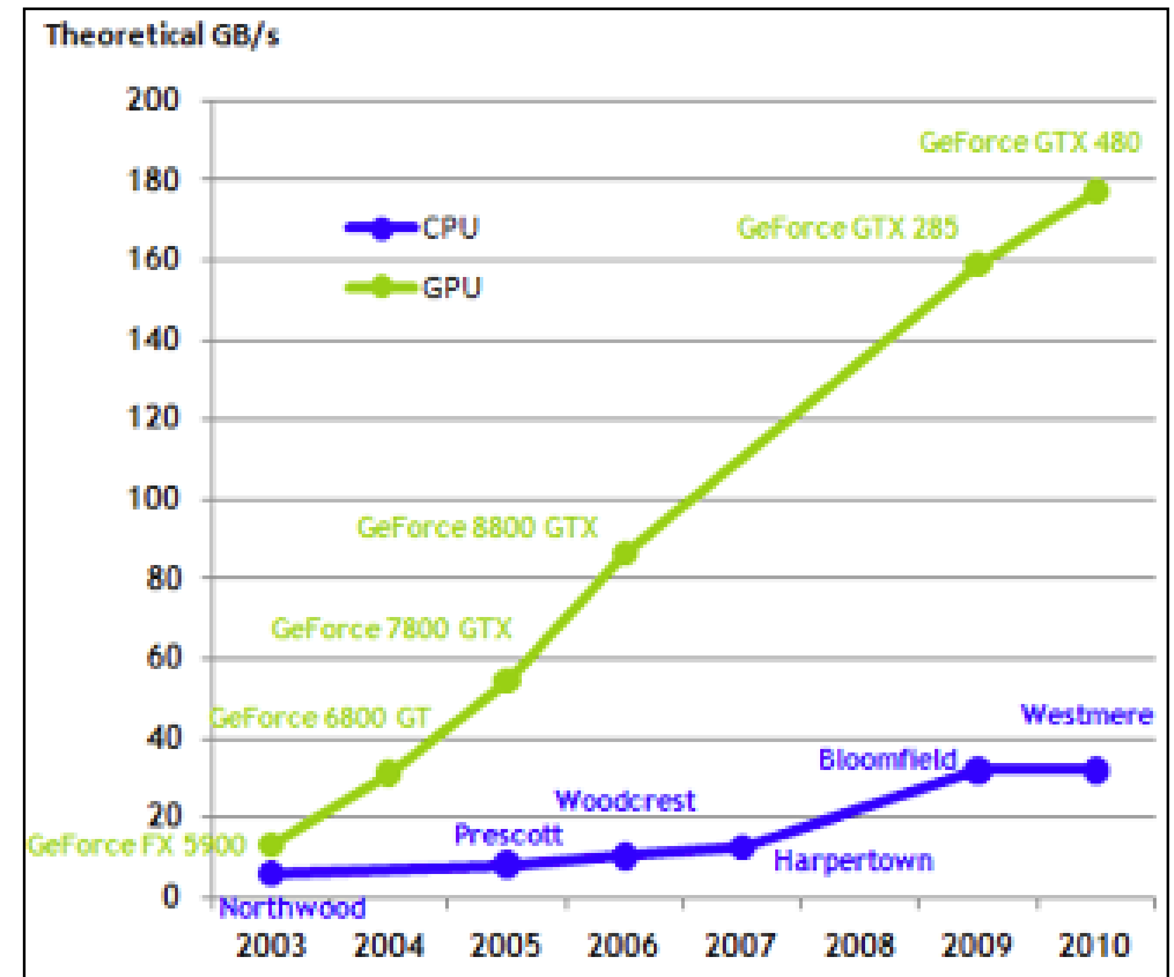
Motivation

- Scientific computation demands accuracy
  - Double precision is the norm
- GPUs historically have have been poor at double precision
  - No support until GT200 (x8 slower than single precision)
  - Commonplace in GPGPU to use mixed precision methods
- Fermi brings the disparity down to x2 slower
  - Do we still need mixed-precision methods?

Motivation

- Both raw flops and memory bandwidth outpaced CPUs





(CUDA Programming Guide)

Super Computer Comparison

Tesla C2050

(the future)

BlueGene/P

(traditional)

	Tesla C2050*	BlueGene/P*
32-bit Gflops	1030	13.6
64-bit Gflops	515	13.6
Watts	225	24
64-bit Gflop/Watt	2.3	0.57
memory b/w GBs^{-1}	144 / 115	13.6
64-bit flop/byte	3.6 / 4.5	1

(*per chip)

Motivation

- GPUs are more memory bound than traditional supercomputers
  - As we go to the Exascale, it’s only going to get worse (for all architectures)
- Even with single and double flops parity, x2 in memory traffic
  - For memory bound algorithms, single precision always x2 faster
- Many problems can be reformulated using mixed-precision methods
  - No loss in precision for final result
  - Large speedup because of reduced memory traffic
- Memory storage can be a limiting factor with GPU computing
  - Truncated precision is a lossy compression allowing larger problems

Precision Truncation using CUDA

Native floating point and integer types in CUDA

- CUDA natively supports
  - single and double precision floating point types
    - e.g., float, double, double3, float4, etc.
  - a variety of integer types
    - char, short, int, long long int (8-bit thru 64-bit)
- CUDA does not support
  - half type (fp16)
  - 8-bit and 16-bit integer operations (char and shorts cost same as int)

Precision Truncation in CUDA

- Don't require native operation support for truncated precision types
  - Just need to be able load and save these types to reduce memory traffic
- Once in registers, we can convert to native types
- CUDA supports a variety of fast type conversions
  - Single instruction intrinsics
  - Texture units

Precision Truncation in CUDA - Half Precision

- Intrinsics for conversion fp16 <-> fp32

```
float __half2float(ushort x);           half -> float

ushort __float2half_rn(float x);        float -> half
```

  - half types are encoded as ushorts
  - hardware accelerated conversion (single instruction)

- Need to get data into fp16 format
  - Copy to 32-bit data to device, do setup kernel before actual computation
  - Create fp16 on host (e.g., OpenEXR includes half precision class)

http://www.openexr.com

Precision Truncation in CUDA - Texture Unit

- Load 8-bit / 16-bit integers through the texture unit

```
texture<short2, 1, cudaReadElementType> texRef;
```
device declaration

```
short2 x = tex1Dfetch(texRef, index);
```
kernel code

- “Free” conversion to fp32 (uints -> [0,1], ints -> [-1,1])

```
texture<short2, 1, cudaReadModeNormalizedFloat> texRef;
```
device declaration

```
float2 x = tex1Dfetch(texRef, index);
```
kernel code

- Useful for fixed-point storage, but floating point compute

One gotcha though...

- Need to be careful to maintain full memory coalescing
- Need a minimum of 32-bit word load per thread to saturate memory bandwidth
  - e.g. Tesla C1060

(Micikevicius)

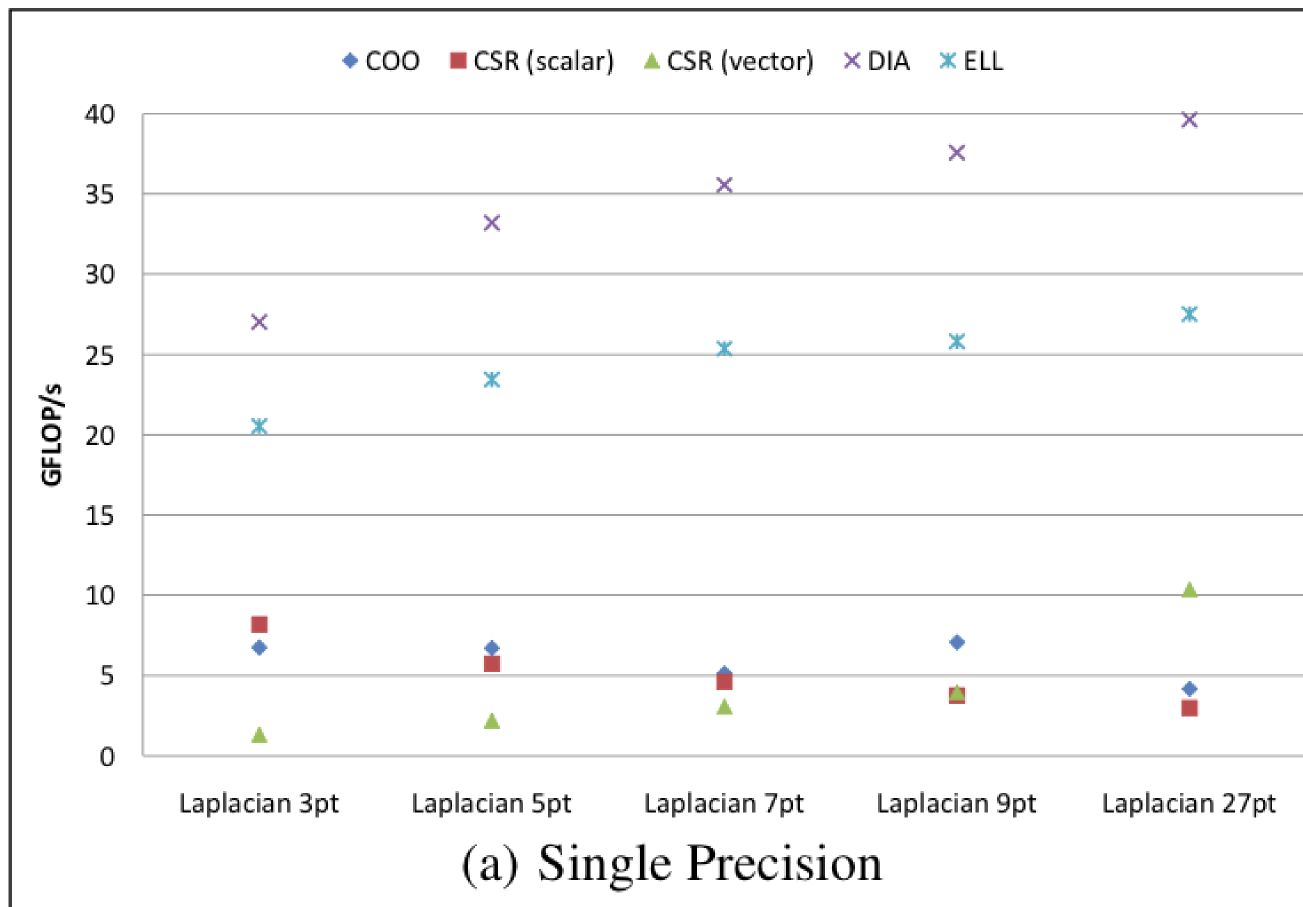
float	77 GB/s
ushort (fp16)	68 GB/s
ushort2 (fp16x2)	77 GB/s

- Need to use vectorized types (32-bit, 64-bit or 128-bit words)

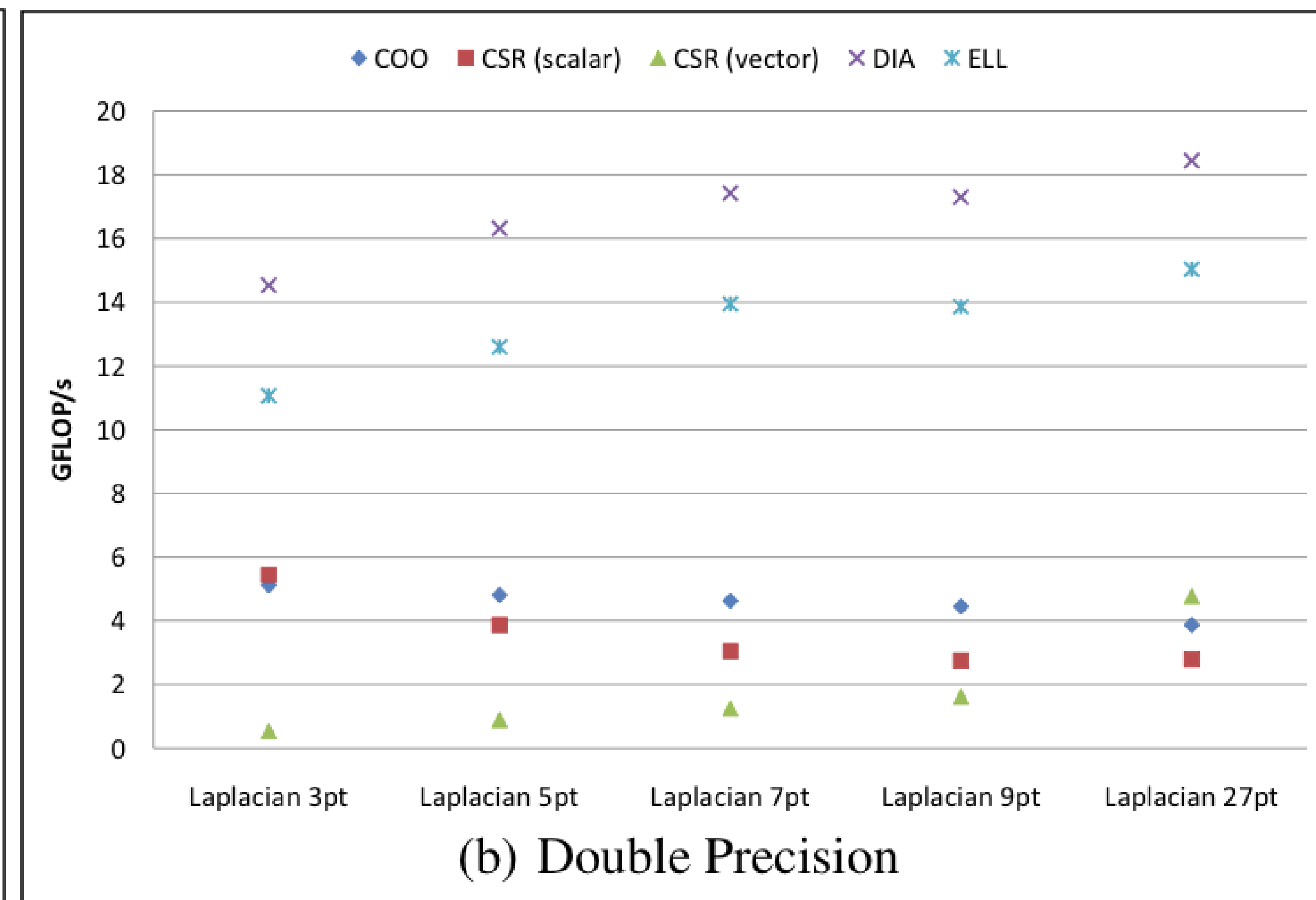
Solving Systems of Linear Equations

Sparse-Matrix Vector Product (SpMV)

- Known to be a memory bound operation
- Single precision x2 faster than double precision
- e.g., Bell and Garland (SC09) - CUSP library



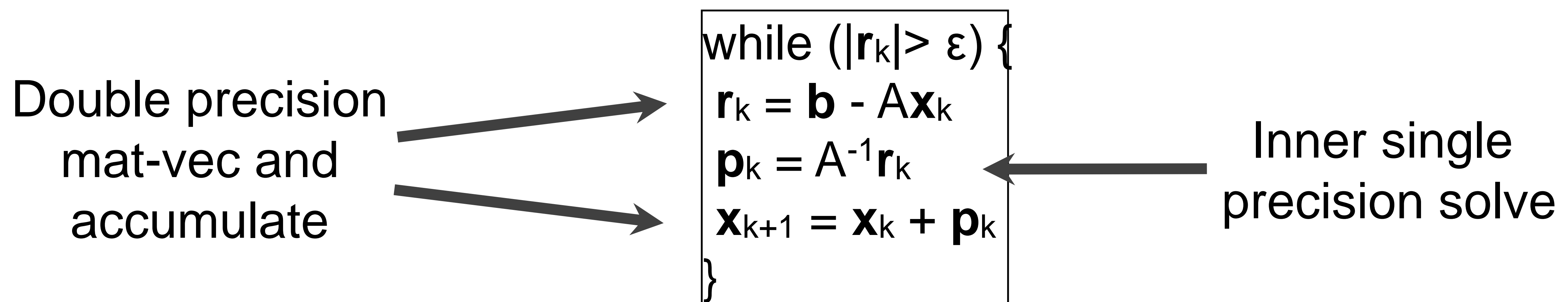
(a) Single Precision



(b) Double Precision

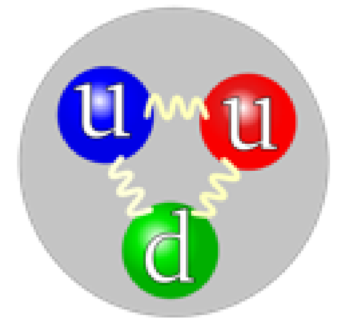
Mixed-Precision Solvers

- Main application for SpMV is solving linear systems $A\mathbf{x} = \mathbf{b}$
- Require solver tolerance beyond limit of single precision
- e.g., Use defect-correction (aka Richardson iteration, iterative refinement)

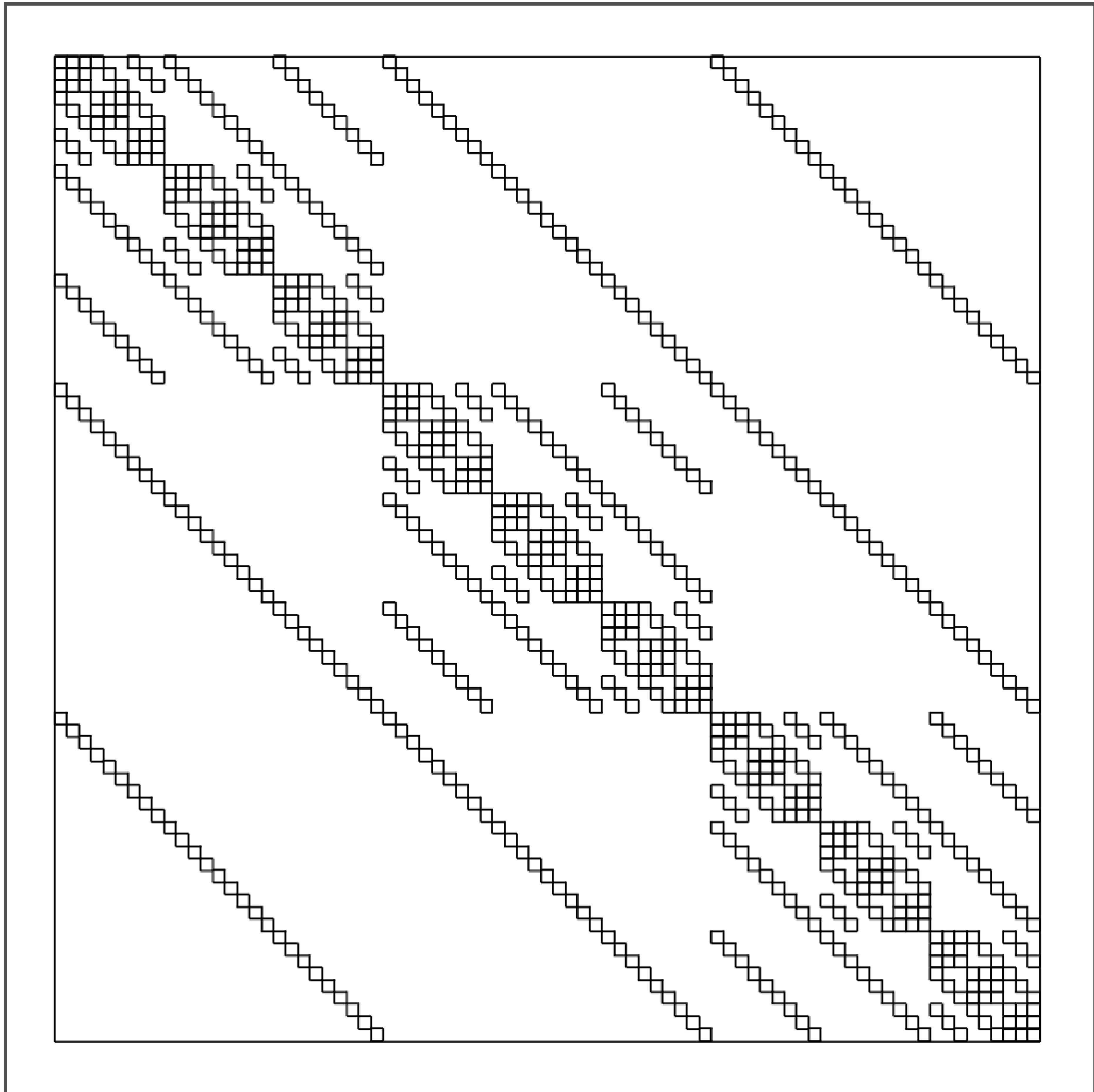


- Double precision can be done on CPU or GPU
  - Can always check GPU gets correct answer
- Can achieve double precision accuracy twice as fast
- We know SpMV is bandwidth limited so how about half precision?

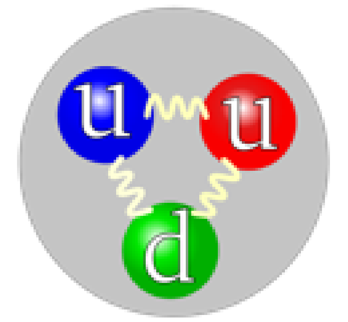
Case Study: Quantum ChromoDynamics



- QCD is the theory of the strong force that binds nucleons
- Grand challenge problem -> requires Exaflops for full solution
- Bulk of computation lies in solution to system of linear equations $A\mathbf{x} = \mathbf{b}$
- A is the Dirac Operator - describes propagation of quarks
  - A is a very large sparse matrix (10^8-10^9 degrees of freedom)
  - Essentially a 1st order PDE acting on a 4 dimensional grid (spacetime)
- Need to be able to perform SpMV (apply A to a vector) as fast as possible

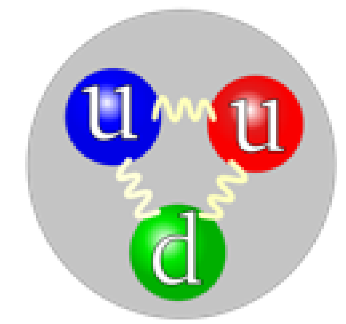


Case Study: Quantum ChromoDynamics

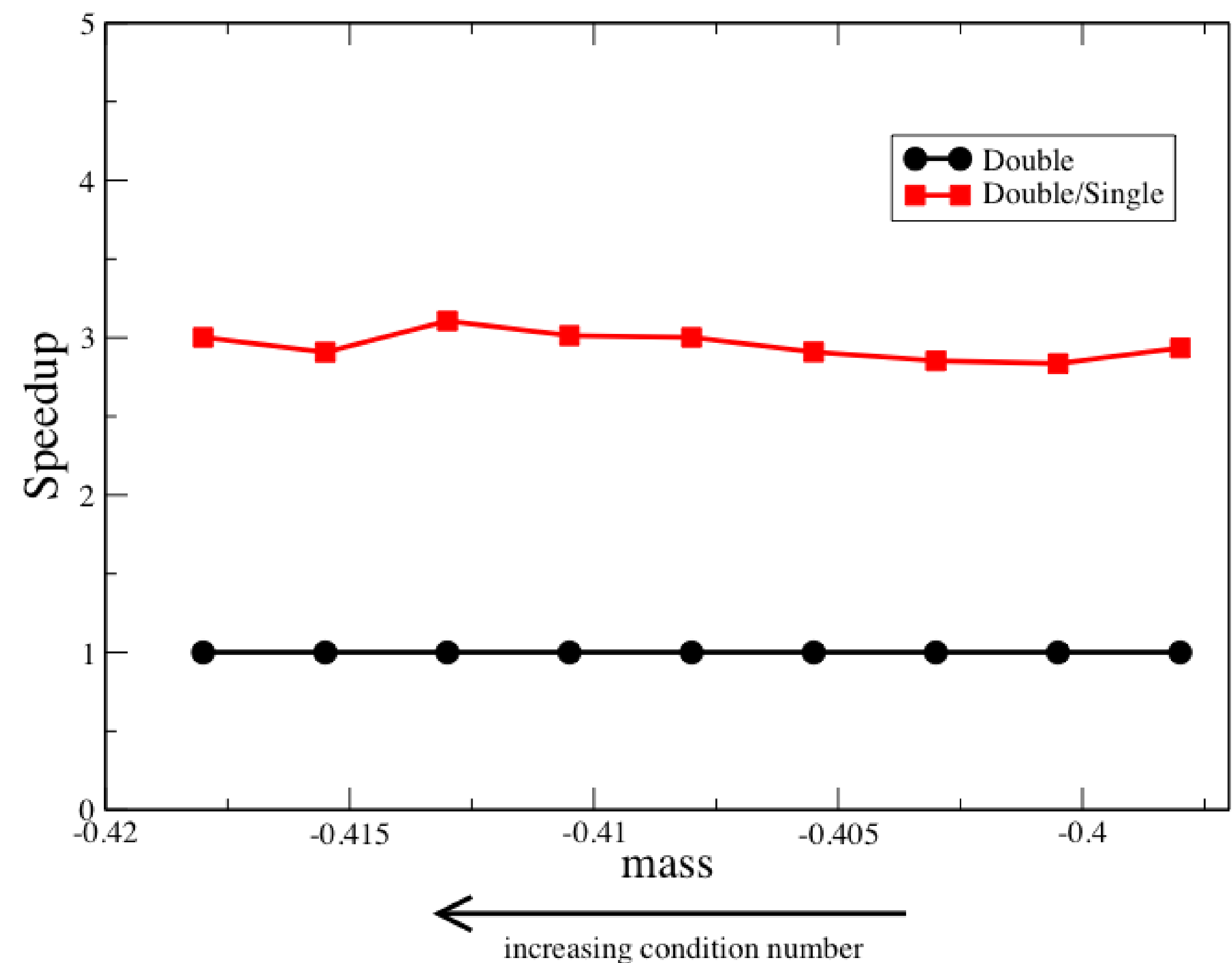


- Sparse matrix -> can use CUSP / CUSPARSE
  - 2544 flops : 4680 bytes (single precision)
  - Recall C2050 7.2 : 1 ratio - extremely bandwidth bound
- Matrix is highly structured
  - Use knowledge of problem to write fast custom kernel
  - From symmetry -> 1368 flops : 1440 bytes
  - Flops are free - do extra computation to reduce memory traffic
    - 1368 : 960 bytes still bandwidth bound

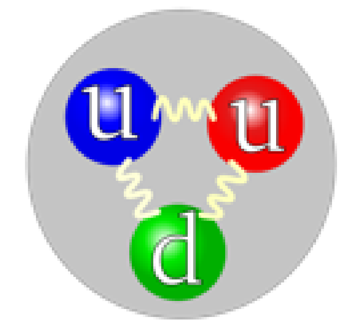
Case Study: Quantum ChromoDynamics



- SpMV performance results (GTX 480)
  - Single Precision 208 Gflops
  - Double Precision 65 Gflops
- Mixed-precision solver 3x faster
  - Full double precision accuracy
  - Single precision only 15% peak



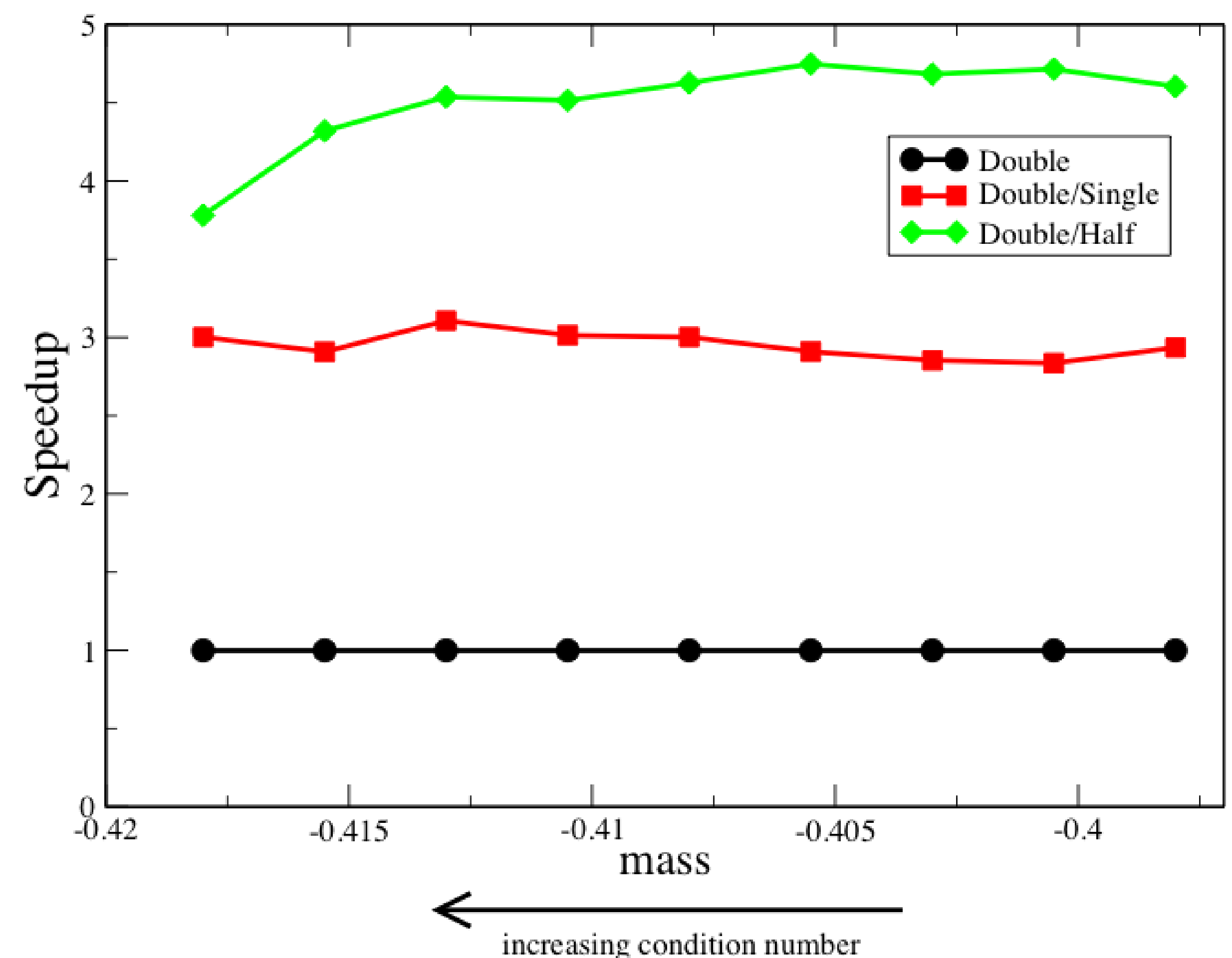
Case Study: Quantum ChromoDynamics



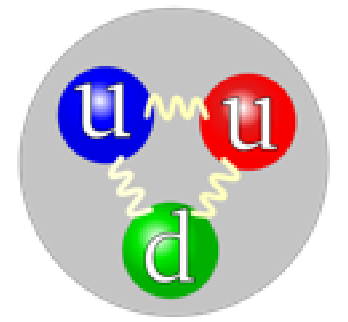
http://arxiv.org/abs/0911.3191

- Use 16-bit precision
  - Need 32-bit words for full bandwidth utilization
  - Pack 1 complex number into a single 32-bit word
- Performance
  - Single Precision 208 Gflops
  - Double Precision 65 Gflops
  - Half Precision 435 Gflops
- Mixed-precision solver 4.5x faster

(speedup not 6x because solver iterations increase)



Case Study: Quantum ChromoDynamics



- 2004: First 1 Tflops sustained for QCD (P. Vranas)
  - 1 rack Blue Gene/L
  - ~ $1M in 2005/2006
- 2010: 1 Tflops sustained, under your desk
  - Dual-socket node with 4 GPUs
  - ~ $13K (80x improvement in price/ performance)

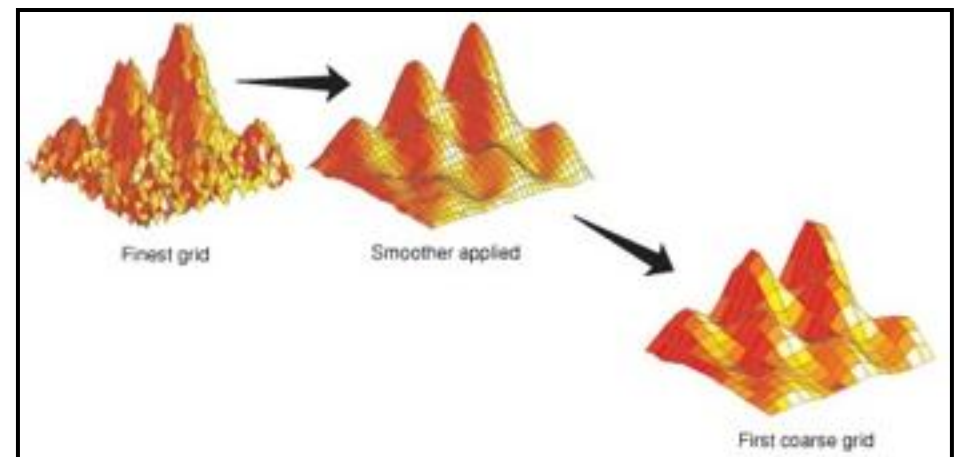
… for problems that fit

(1 rack BG/L has 512 GB RAM vs. 12 GB for 4 C2050s)

Each use should be accompanied by the credit line and notice, "Courtesy of International Business Machines Corporation. Unauthorized use not permitted." Copying of images for further distribution or commercial use is prohibited without the express written consent of IBM.

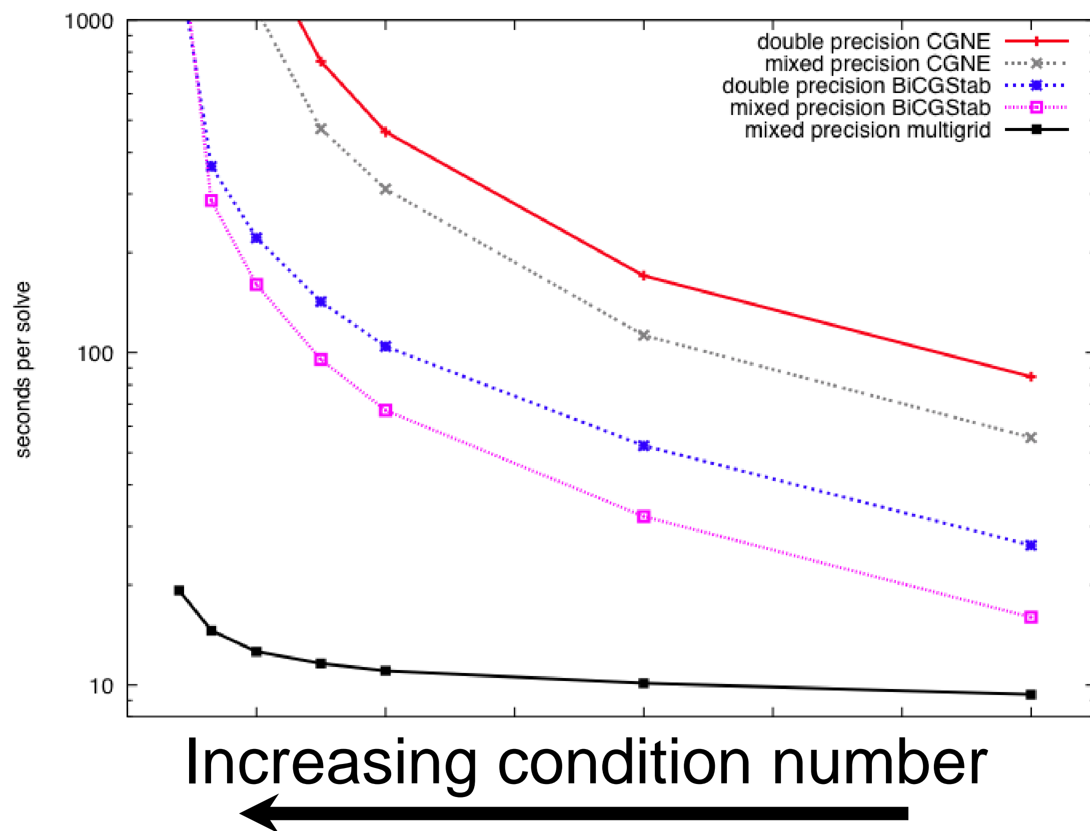
Case Study: Multigrid

- Multigrid is known as an *optimal* method for solving elliptic PDEs (A**x** = **b**)
  - Constant time to solution regardless of condition number
  - Iteration count scales linearly with volume



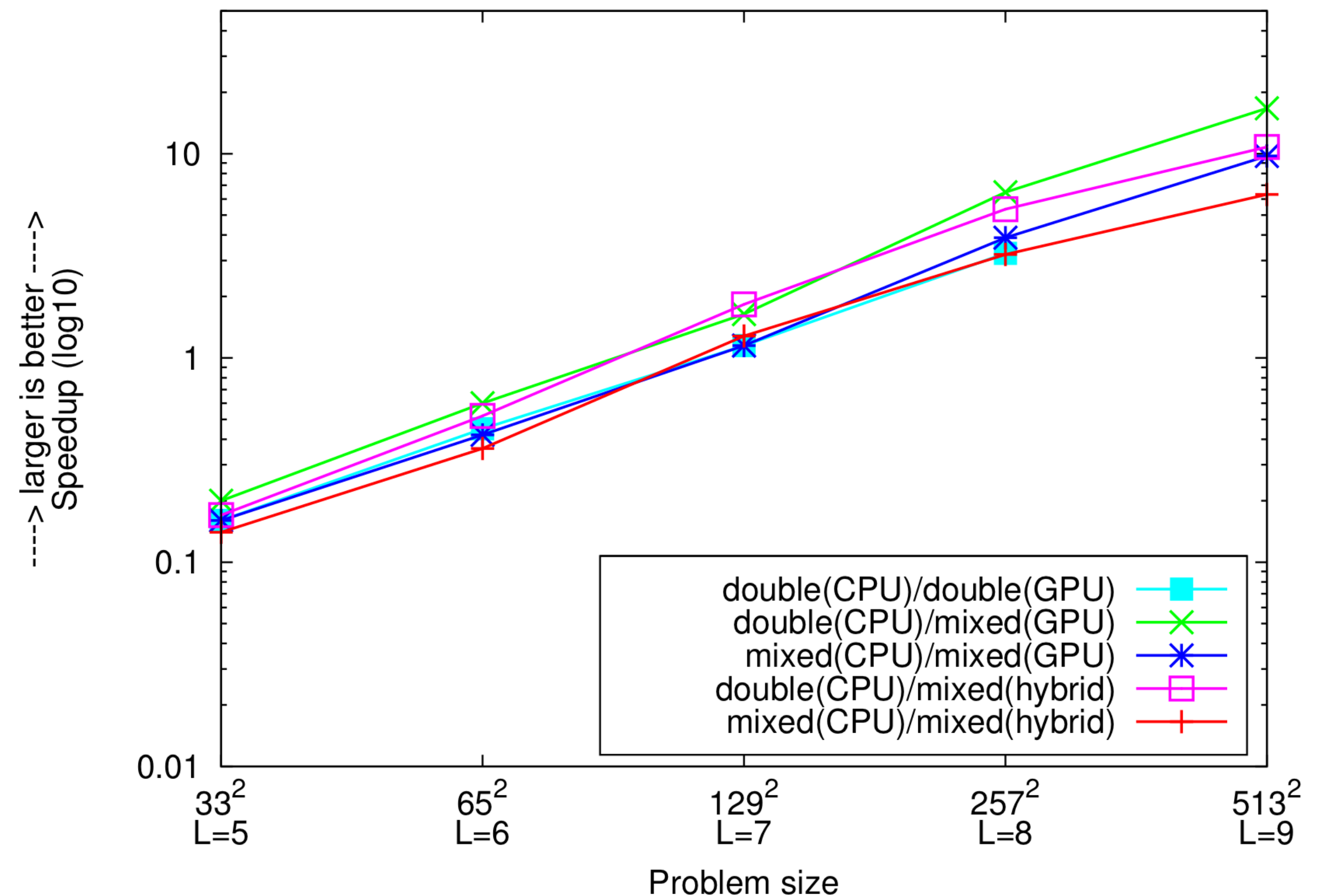
- How to use in mixed-precision?
  - Wrap multigrid in a Krylov solver and use as a *preconditioner*
  - Only require high precision for outer solver
  - Preconditioner has low accuracy requirements
  - Double-single, Double-half, etc. much faster than plain double

Case study: Multigrid



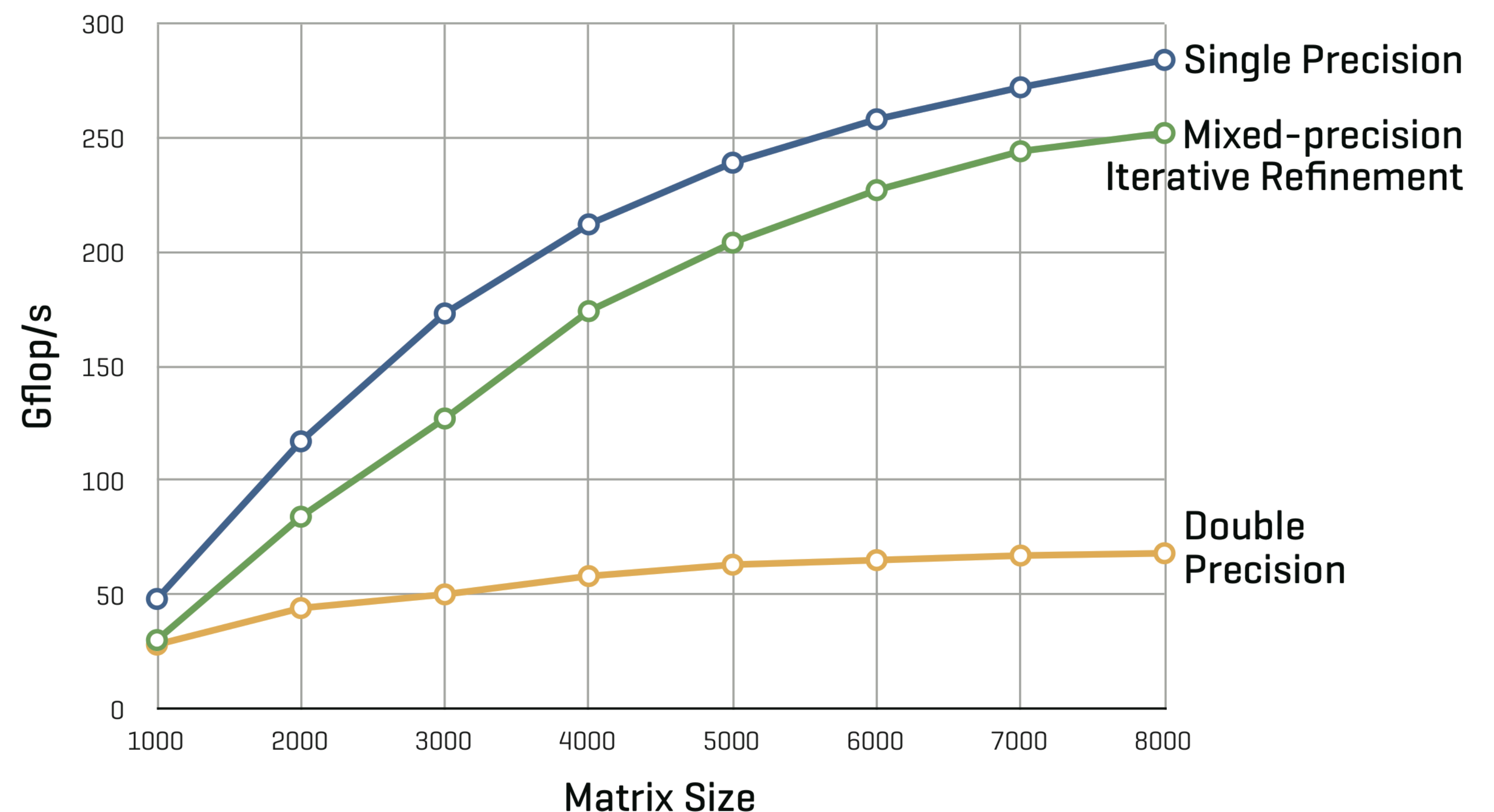
Tesla C1060
10^7 dof

Domenic Göddeke's thesis
http://hdl.handle.net/2003/27243



Mixed-precision isn’t just for sparse systems

- Also useful when solving dense linear systems too (not bandwidth bound)
- E.g., Solving Ax=b through LU factorization
  - Analogous to multigrid - LU is a preconditioner



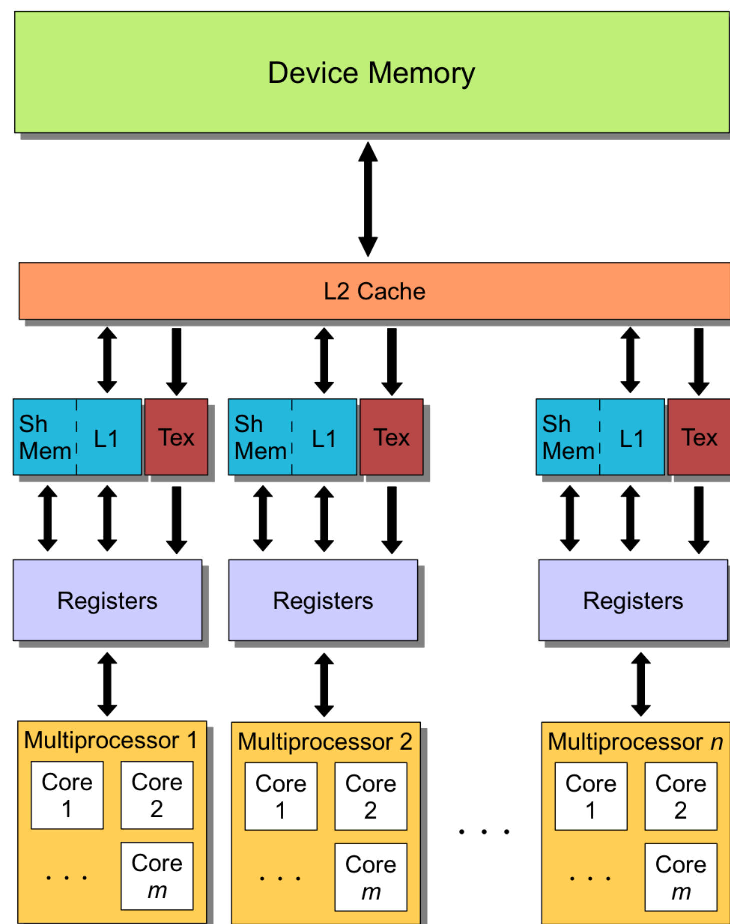
MAGMA (http://icl.cs.utk.edu/magma/)

Advanced precision optimization

- Recall QCD performance
  - Half 435 Gflops
  - Single 202 Gflops
  - Double 65 Gflops

Why is half > 2x faster than single?

Why is single > 2x faster than double?



L1 and L2 cache means super-linear speedup possible (can fit more in cache with smaller datatypes)

Kernels often limited by available registers / shared memory

- Registers are always 32-bit
- Shared memory only requires sizeof(type)

Increase amount of useful information held in fast memory

```
__shared__ ushort A[BLOCK_SIZE];
float B = __half2float(A[threadIdx.x]);
```

Other mixed-precision applications / algorithms

- MGEMM - mixed precision matrix-matrix multiplication (Olivares-Amaya *et al*)
  - Partition matrix into large and small components
  - large multiplications use double, small use single

$$C = \left(A^{\text{large}} + A^{\text{small}}\right) \cdot \left(B^{\text{large}} + B^{\text{small}}\right) = \underbrace{AB^{\text{large}} + A^{\text{large}} B^{\text{small}}}_{\text{CPU}} + \underbrace{A^{\text{small}} B^{\text{small}}}_{\text{GPU}}$$

- Low precision data summed into high precision accumulator
  - e.g., reductions, force summations, signal processing
- Extended precision possible in CUDA (Lu, He and Luo)
  - GPUs > order of magnitude faster at double-double, quad-double than CPUs
  - Mixed-precision methods can make extended precision reasonable

Summary

- GPUs (and future HPC in general) are increasingly bandwidth bound
  - Precision truncation can help alleviate memory traffic
- CUDA supports a variety of limited precision IO types
  - half float (fp16), char, short
- Large speedups possible using mixed-precision
  - Solving linear systems
- Not just for accelerating double-precision computation with single-precision
  - 16-bit precision can speed up bandwidth bound problems
  - Beyond double precision also sees large speedups