I CAN'T BELIEVE IT'S NOT MOLECULAR DYNAMICS (IT'S MACHINE LEARNING TOO)

Scott Le Grand

Outline

- Some Things Never Change (GPUs vs the World)
- How Best to Exploit GPUs
- Molecular Dynamics or Matrix Factorization?
- Determinism and Numerical Stability
- Dynamic Range for both MD and NNs
- Latest AMBER PME Numbers
- Conclusions

Brawny versus Wimpy

Brawny cores still beat wimpy cores, most of the time

Urs Hölzle
Google

"Slower but energy efficient "wimpy" cores only win for general workloads if their single-core speed is reasonably close to that of mid-range "brawny" cores."

Insinuation: GPU cores are wimpy

NVIDIA fell for it…

- GeForce GTX Titan X: 3,072 "CORES!"
- GeForce GTX 980: 2,048 "CORES!"

One SIMD Lane == One Core

By this definition, GPUs are really wimpy…

(And a Haswell CPU has up to 144 "cores" making it really, really wimpy, but I digress)

My Definition of "Core"

Core: a set of processing elements that share an L1 cache (or equivalent) and register file

Processor: One or more cores on a single die

(I personally prefer cores with more cache and registers per thread over "brawny" vs "wimpy")

CPUs are indeed brawny cores

Fast CPU: Intel Xeon E5-2699 v3 Haswell 2.3 GHZ (3.6 GHz Turbo Boost) 45 MB L3 Cache LGA 2011-v3 145W 18-Core Server Processor ($4,632.00 on Amazon)

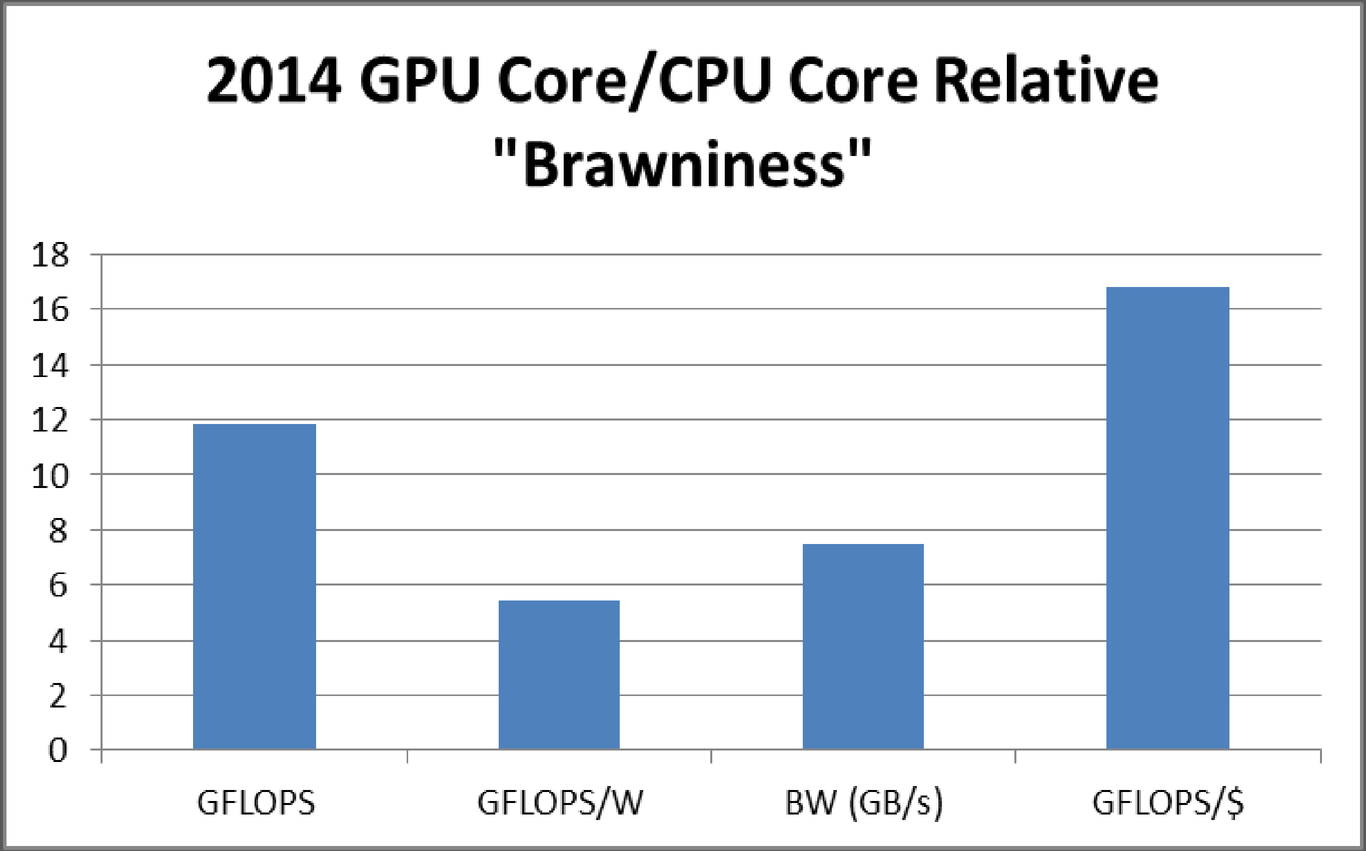
Peak GFLOPS:	~662
GFLOPS/W:	~4.6
GFLOPS/Core:	~37
GFLOPS/$:	~0.14

But GPUs are brawnier™ cores

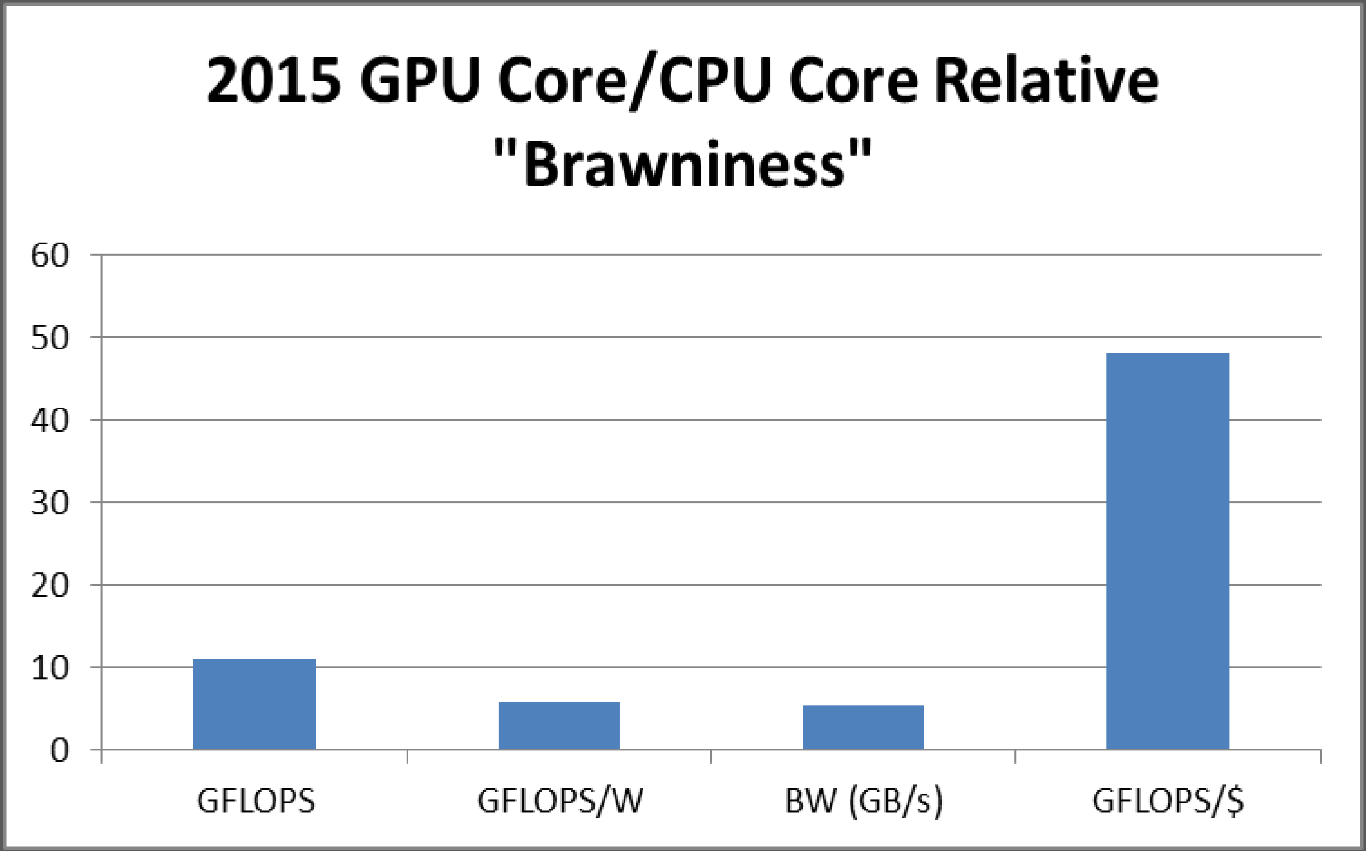
Fast GPU: NVIDIA Ge Force GTX Titan X, 24-core, 1088 GHz TDP 250W ($999 announced)

Peak GFLOPS:	~6,695
GFLOPS/W:	~27
GFLOPS/Core:	~280
GFLOPS/$:	~6.7

GPU Cores were roughly 5-15x brawnier (2014)…



GPUs have 2x Better GFLOPS/$ (2015)



So GPUs themselves should be 5-50x brawnier than CPUs or you're not doing it right…

Or the algorithm you’re running is inherently serial*…

*But then why exactly are you running it on 1,000+ machines at once**.

**Because you’re I/O bound? Well then you’re just wasting power using “Brawny” cores, spend your money on better hard drives and networking.

How about FPGAs?

“FPGAs are (up to) 10x faster and up to 50x more power-efficient than CPUs!!!!”

FPGAs Are Interesting

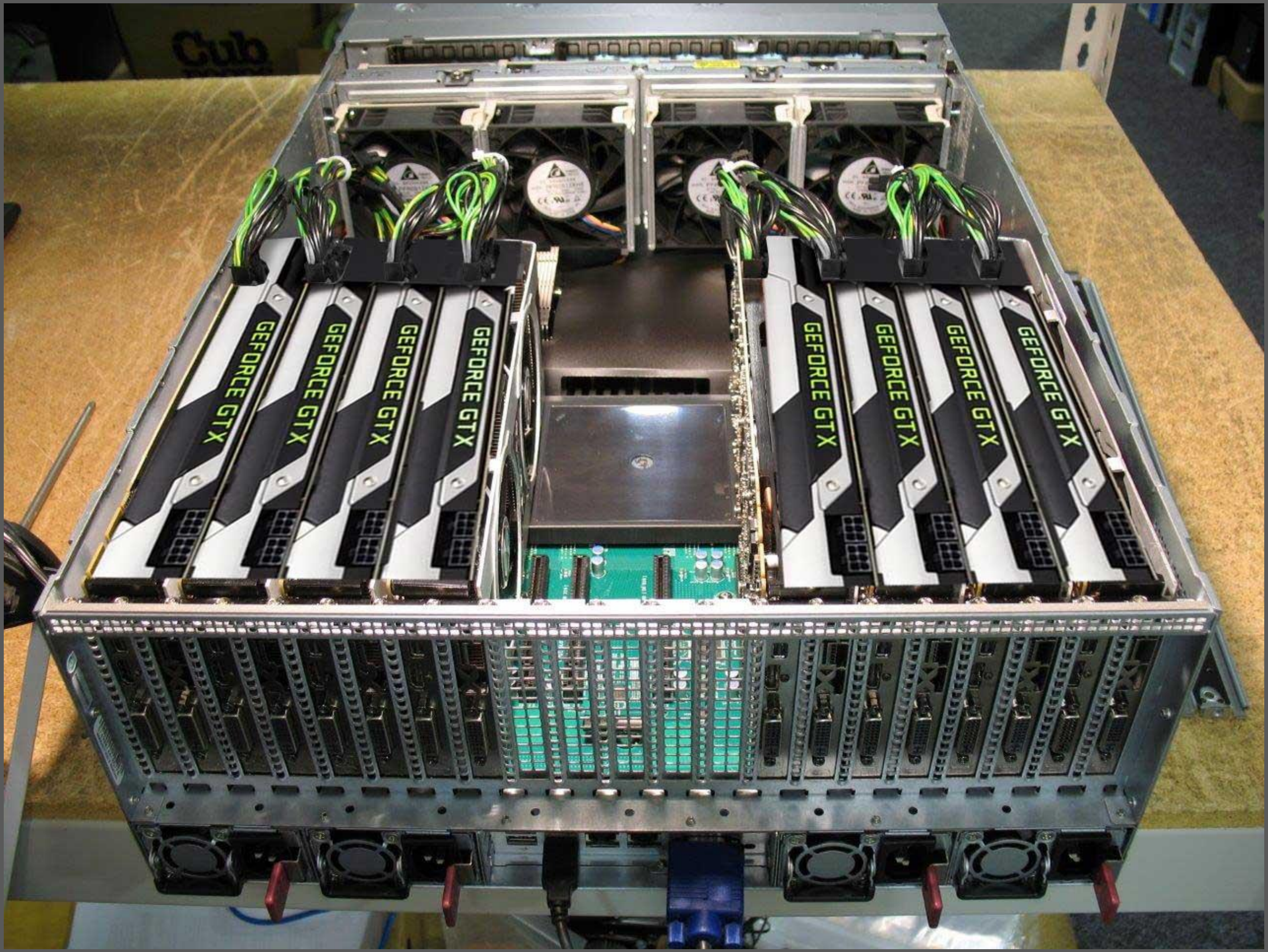
FPGA:	Altera Arria 10 (1150GX)
Peak GFLOPS:	1,366*
GFLOPS/W:	40**

*https://www.altera.com/en_US/pdfs/literature/hb/arria-10/a10_overview.pdf
**http://www.enterprisetech.com/2015/02/23/microsoft-accelerates-datacenter-with-fpgas/

FPGA Reality

- Maybe 1.5-2x better Perf/W
- 1.37 TFLOPS is something between a GF110 and a GK104
- You can only stuff so many of these things in a server (8 or so), is power your real constraint?
- Nervana is getting ~3.7 TFLOPs (out of ~4.6) running CNNs on GM204

Or is it physical space?



Amended FPGA Statement

"2x CPU performance* with ~1.5x the power-efficiency of a GPU"

*~11x better GFLOPS/W than CPUs, which is nice

FPGAs versus GPUs

Good News for FPGAs

- Altera is adding OpenCL support to FPGAs

Bad News for FPGAs (FUD)

- Compilation time is hours versus seconds
- No FPGA cuFFT, cuBLAS, cuRand, etc libraries
- You can buy GPUs on Amazon
- Linux/Windows GPU drivers freely available

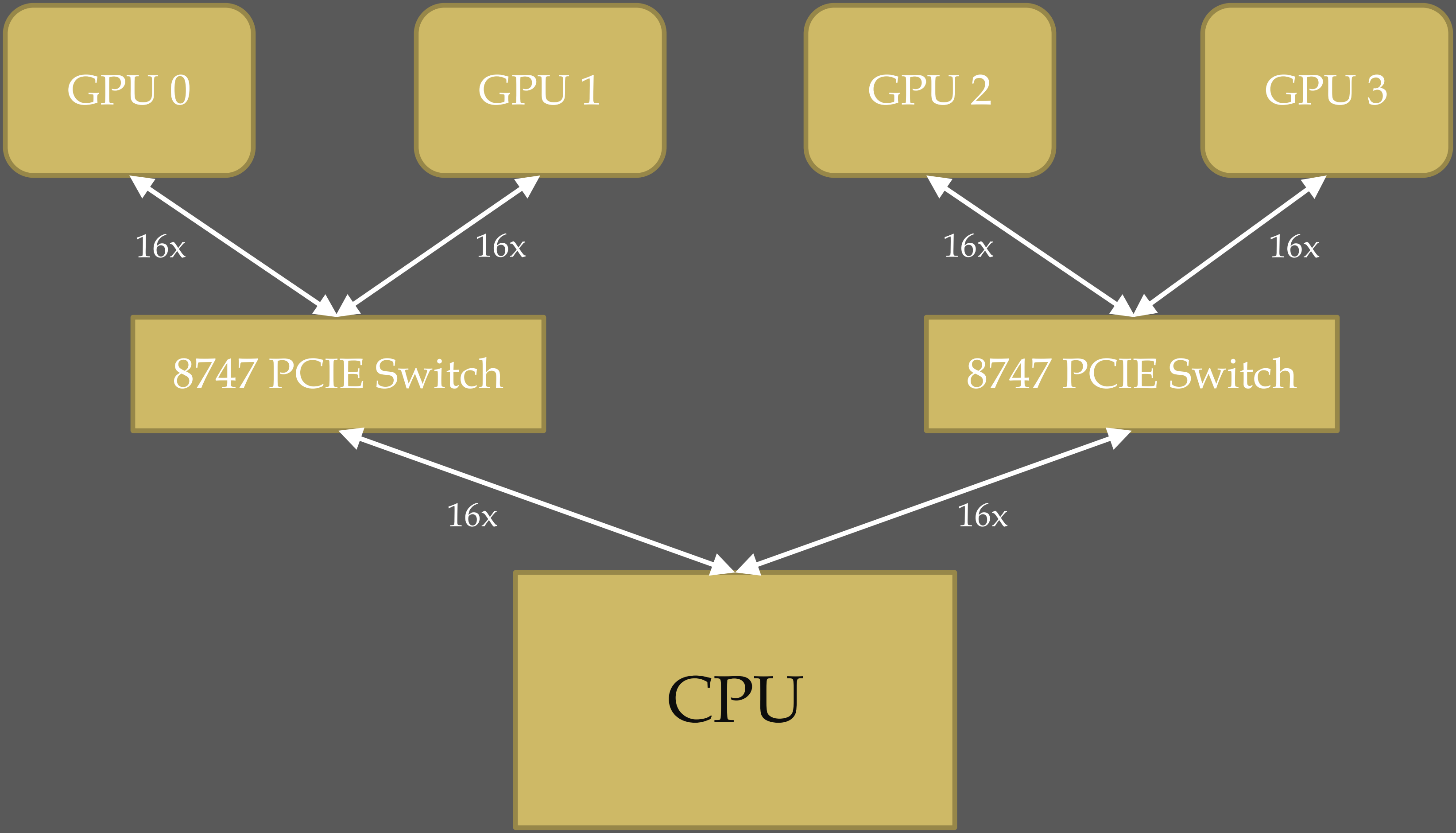
General GPU Hints

- Avoid SandyBridge CPUs!
- They only support PCIE Gen 2 (1/2 PCIE Gen 3)
- They don’t work reliably with GM2xx

- Avoid GTX 970 (~$200 < GTX 980)
- Last 512MB has BW issues
- Keep your life simple, time is money

- Avoid crazily overclocked GPUs

DIY Digits Dev Box



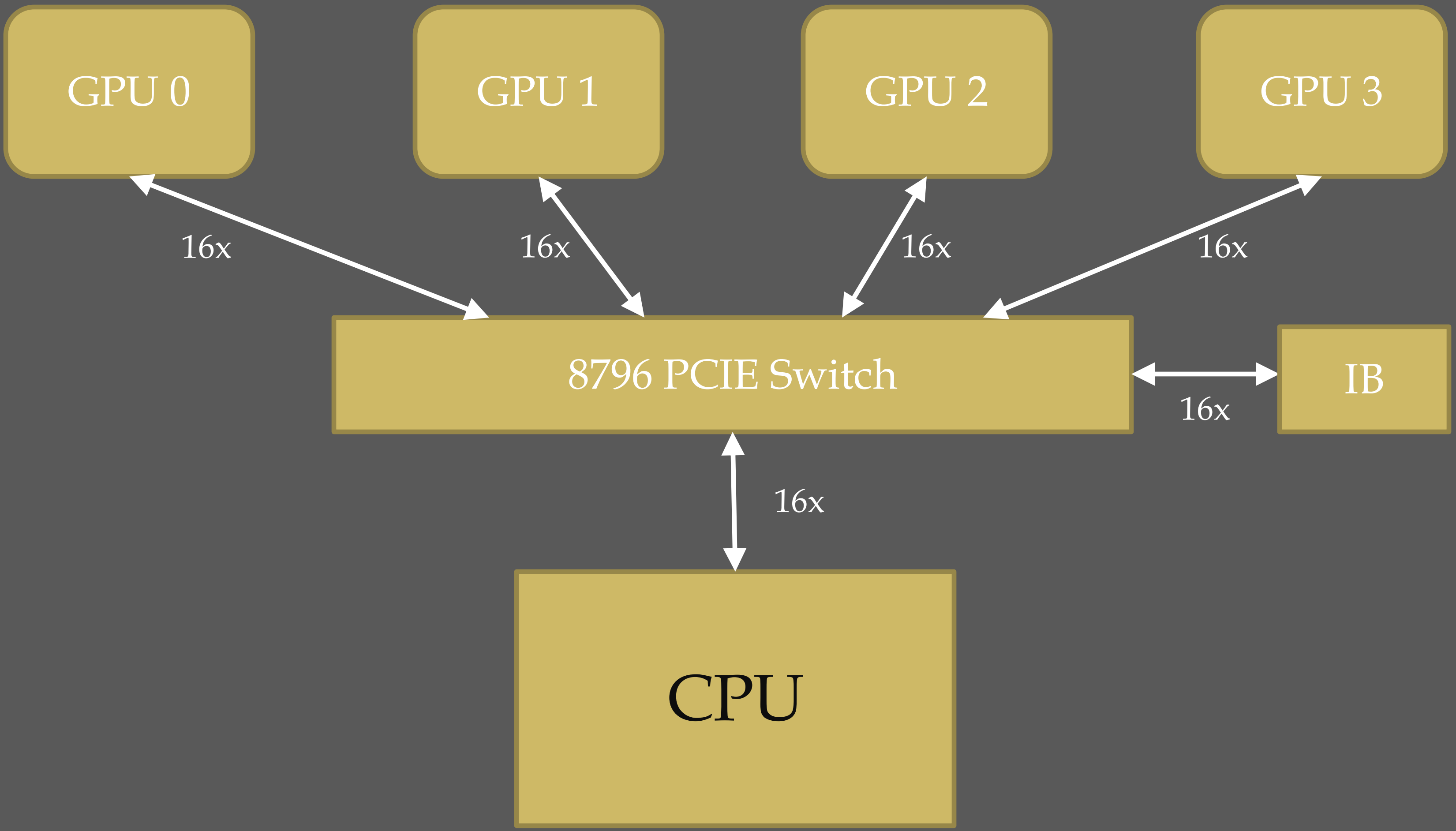
Good Choices

- Asus P9X79-E WS MB ($500) plus Intel Core-i7 4820 (Ivybridge) CPU ($320)
- Asus X99-E WS MB ($520) plus Intel Core-i7 5930K (Haswell) CPU ($560)
- 1st alternative saves about $260
- 25 TFLOPs for $7,000! (<50% of Digits DevBox)

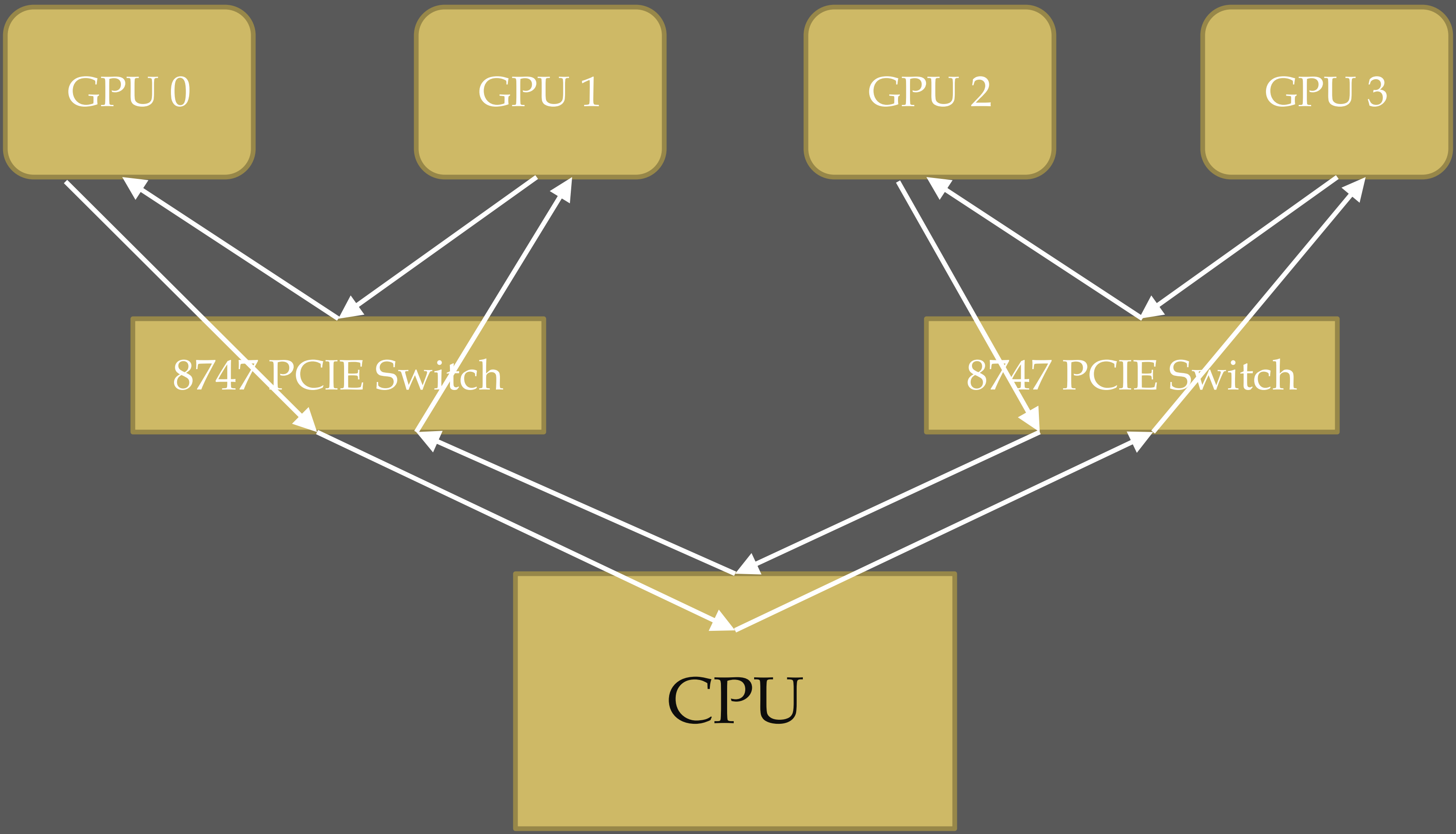
For The Data Center

Dell C4130 1U Quad-GPU Server

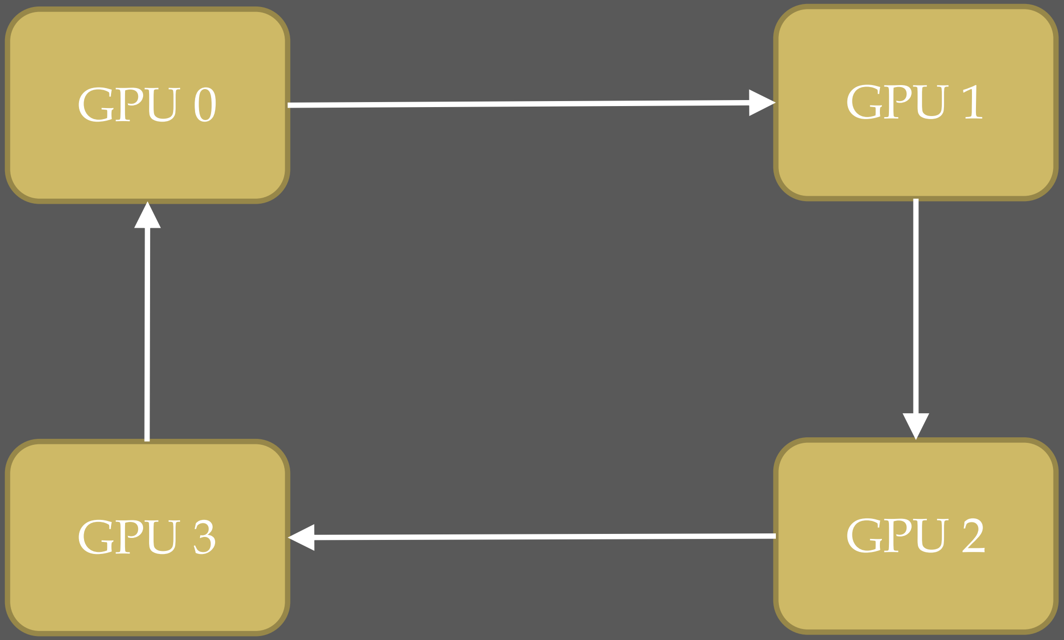
Full P2P Bisection BW



Workaround



Simplified



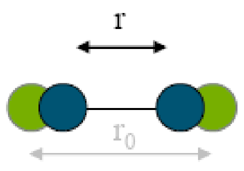
Building a Multi-GPU App

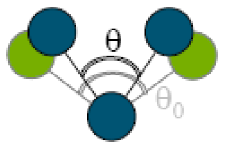
- Install a recent build of OpenMPI or MPICH2 (do not install what comes with linux distros)
- Do not enable GPUDirect
- Do not use MPI 2.x primitives
- Use MPI for process control and synchronization
- Use Interprocess P2P within CUDA to send messages between the GPUs. I repeat, do not rely on GPUDirect

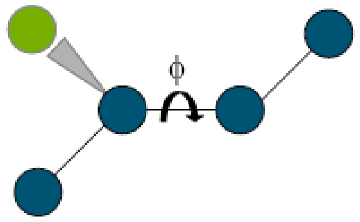
Algorithmic Rules of Thumb

$O(N^2)$	Embarrassingly Parallel (Learn CUDA)
$O(N \log N)$	Annoyingly Parallel (Hire an Expert)
$O(N)$	Likely I/O-Bound (don't bother)

Molecular Dynamics







$$E_{stretching} = \sum_{1,2\,pairs} K_r (r - r_0)^2$$

$$E_{bending} = \sum_{1,2\,pairs} K_\theta (\theta - \theta_0)^2$$

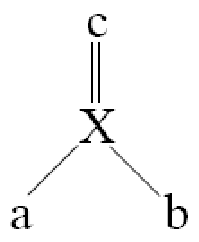
$$E_{torsion} = \sum_{1,4\,pairs} K_\phi (1 - \cos(n\phi - \delta))$$

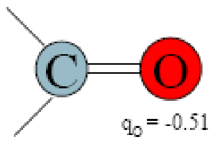
$$\boxed{E = E_{stretch} + E_{bend} + E_{torsion} + E_{impropers} + E_{electrostatic} + E_{vanderWaals}}$$

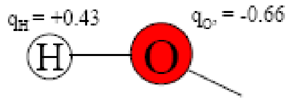
$$E_{van-der-Waals} = \sum_{\substack{nonbonded\\pairs}} \left(\frac{\boxed{A_{ik}}}{r_{ik}^{12}} - \frac{\boxed{C_{ik}}}{r_{ik}^{6}} \right)$$

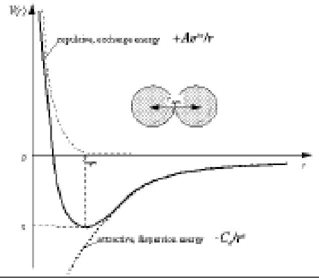
$$E_{impropers} = \sum_{impr.} K_\varphi (\varphi - \varphi_0)^2$$

$$E_{electrostatic} = \sum_{\substack{\text{nonbonded}\\\text{i-k pairs}}} q_i . q_k / D . r_{ik}$$









Molecular Dynamics on GPUs
(or how to keep 21,504++ threads occupied)

- On a CPU, the dominant performance spike is:

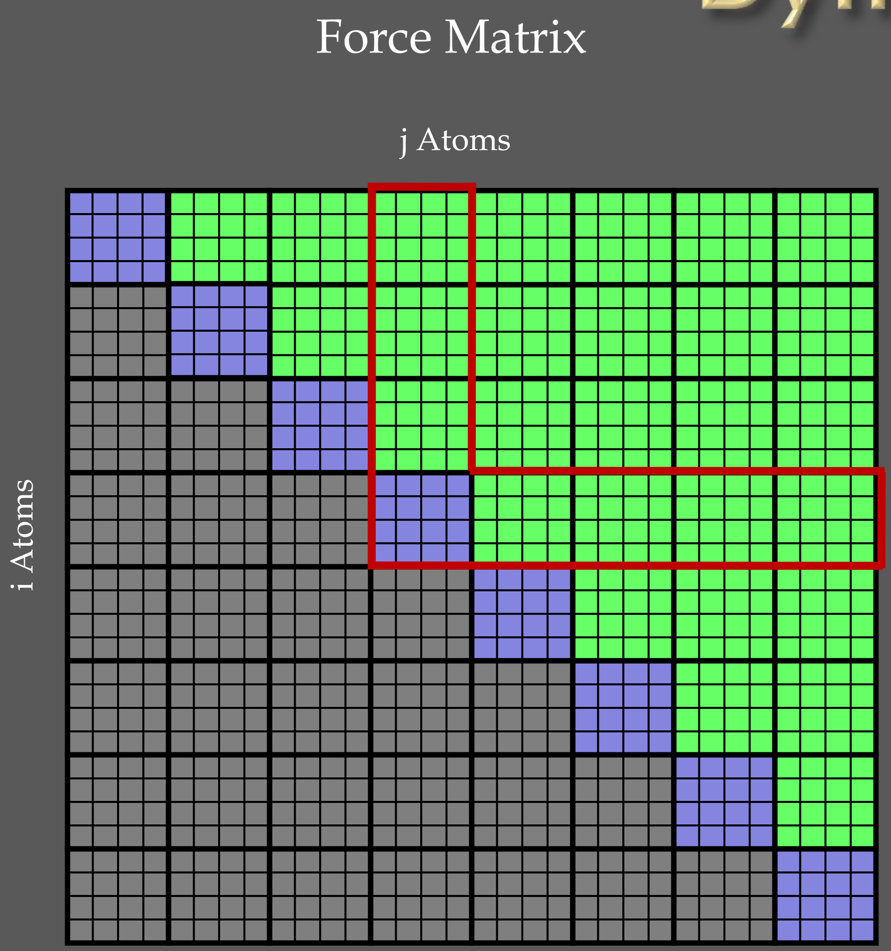
```
for ($i$ =0; $i$ < N; $i$++)
    for ($j$ = $i$ + 1; $j$ < N; $j$++)
        Calculate f$ij$, f$ji$;
```

O(N^2) Calculation

If we naively ported this to a GPU, it would die the death of a thousand race conditions and memory overwrites

Solution: Map the problem into many subtasks and reduce the results

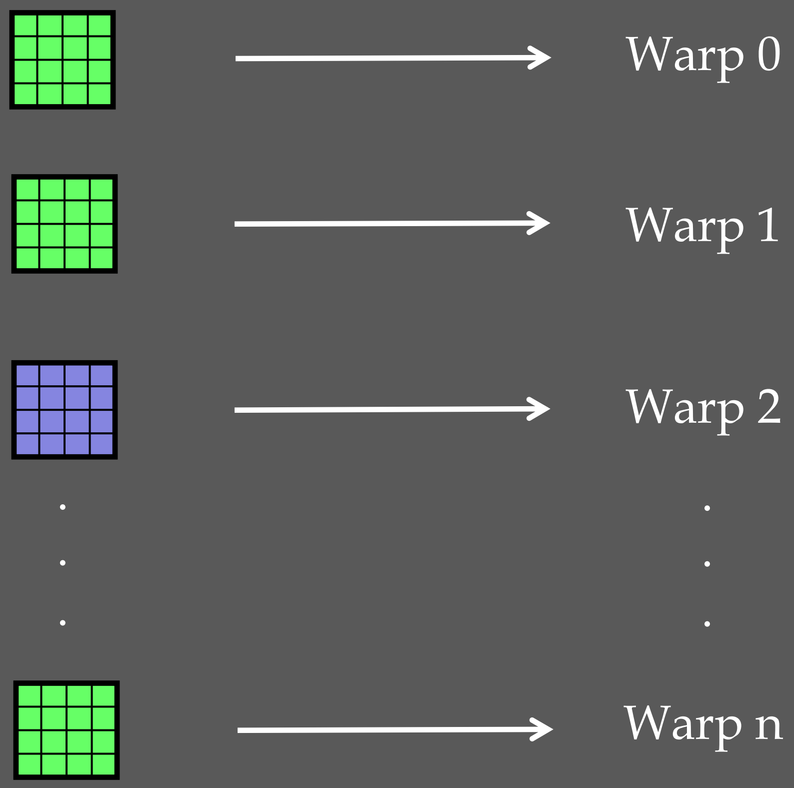
MapReduced Molecular Dynamics



Subdivide force matrix into 3 classes of independent tiles

🟩	Off-diagonal
🟦	On-diagonal
▨	Redundant

Map each nonredundant tile to a warp



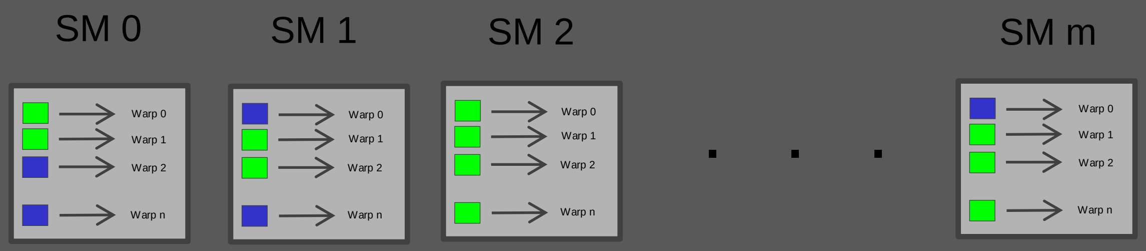
Wait, what’s a warp?

- The smallest unit of execution in a GPU
- Up through GM2xx, it’s groups of 32 consecutive threads within the same core that execute in lockstep
- GPU cores each run 8-64 warps at once
- May change in the future
- “lock-free computing”

What’s So Special About Warps?

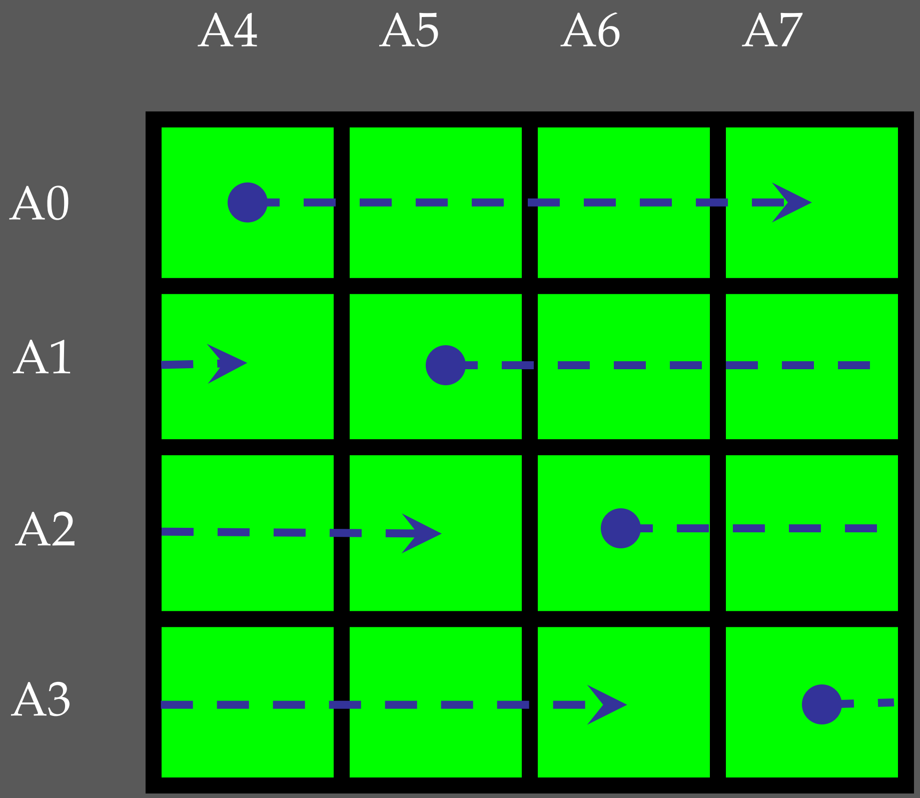
__shfl:	Exchanges data between warp threads
__ballot:	Each bit gives state of a predicate for each warp thread
__all:	True if predicate is true across all warp threads
_any:	True if predicate is true on any warp thread

Each iteration produces force tiles…

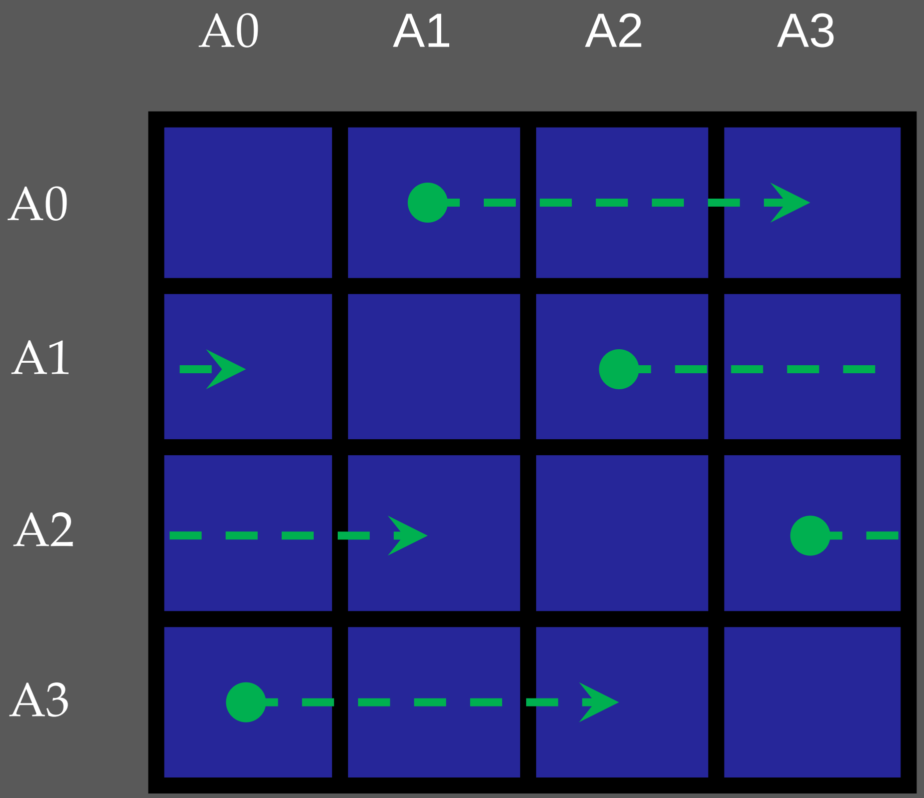


Each warp in the GPU cores consumes them…

Off-Diagonal Tile Calculation in Detail



On-Diagonal Tile Calculation in Detail



Inner Loop

```
float xi      = pAtomX[i];
float yi      = pAtomY[i];
float zi      = pAtomZ[i];
float xj      = pAtomX[j];
float yj      = pAtomY[j];
float zj      = pAtomZ[j];
int pos       = theadIdx.x & 0x1f;
int shIdx     = (pos + 1) & 0x1f;
do
{
   float xij  = xi - xj;
   float yij  = yi - yj;
   float zij  = zi - zj;
   float r2   = xij * xij + yij * yij + zij * zij;
   float r    = sqrt(r2);
   .
   Calculate Forces (lots of Muls and Adds)
   .
   xj         = __shfl(xj, shIdx);
   yj         = __shfl(yj, shIdx);
   zj         = __shfl(zj, shIdx);
   pos        = (pos + 1) & 0x1;
}
while (pos != ((threadIdx.x + 1) & 0x1f));
```

How Many Warps?

- GK110: 1,280 threads/SMX, 15 SMXs, 600 warps
- GM204: 1,024 threads/SM, 16 SMs, 512 warps
- GM200: 1,024 threads/SM, 24 SMs, 768 Warps

24,576 threads!!!!!

- Implies you need about 1,280 (40 * 32) atoms to fill the GPU: (40 * 41) / 2 tiles == 820 warps
- And it's only going to get worse
- Not a problem past 10,000 atoms or so

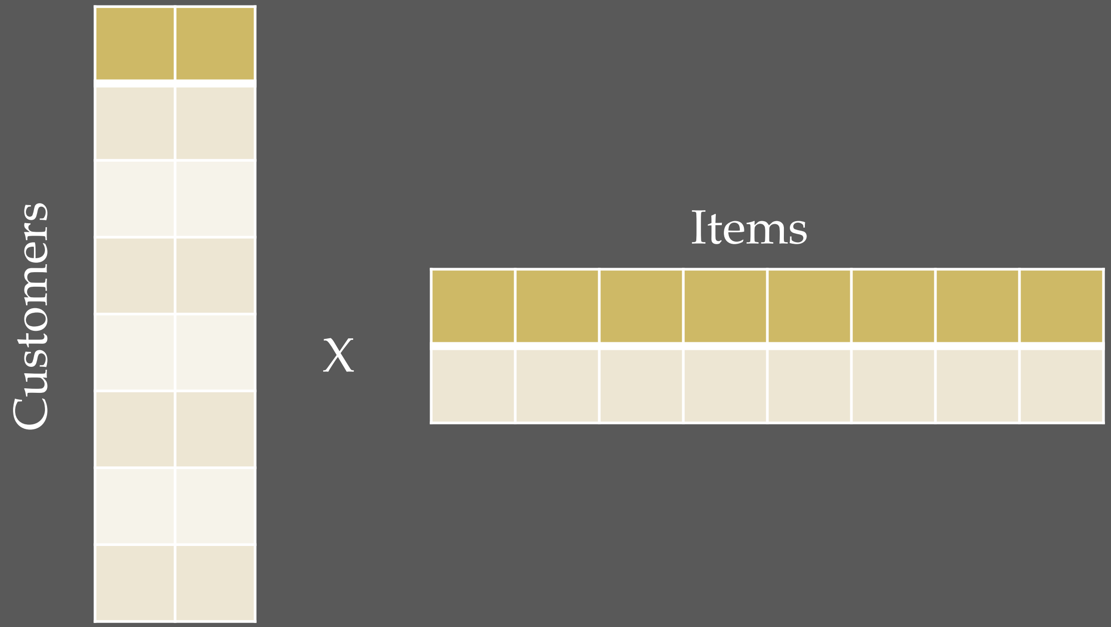
Matrix Factorization

Items

Customers

?	?	1	?	?	?	1	?
?	?	?	1	?	?	?	?
?	1	1	?	?	?	1	1
?	1	?	1	?	?	?	?
?	?	1	?	?	1	?	?
1	?	?	?	?	1	?	?
?	?	?	1	?	?	?	1
1	?	1	?	?	?	1	?

Latent Factor Matrices



Innermost Loop

$$A_{ij} = Customer_i \circ Item_j$$

Naïve Inner Loop

```
// Calculate dot product
int wid                        = threadIdx.x & 0x1f;
int pos                        = wid;
float dp                       = 0;
while (pos < length)
{
        dp                    += pCustomer[pos] * pItem[pos];
        pos                   += 32;
}

// Reduce results
dp                            += __shfl(dp, wid ^ 1);
dp                            += __shfl(dp, wid ^ 2);
dp                            += __shfl(dp, wid ^ 4);
dp                            += __shfl(dp, wid ^ 8);
dp                            += __shfl(dp, wid ^ 16);
```

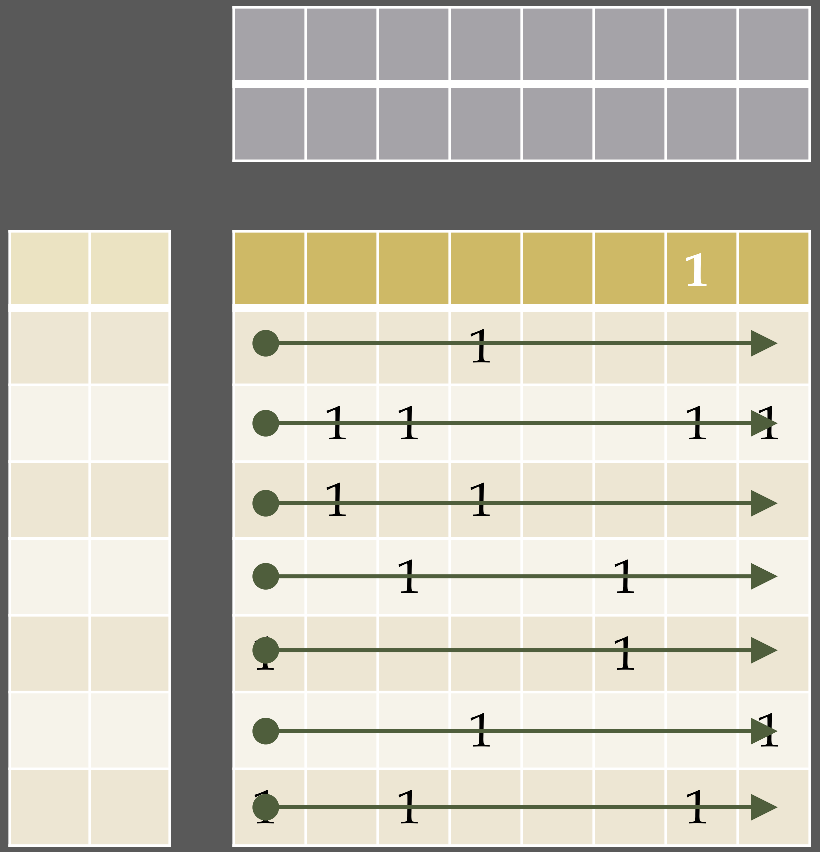
Faster Inner Loop

```
// Calculate dot product
int wid                          = threadIdx.x & 0x31;
int pos                          = wid;
float dp                         = 0;

// Unrolled register vs memory sum
dp                               += rCustomer0 * pItem[pos];
pos                              += 32;
dp                               += rCustomer1 * pItem[pos];
pos                              += 32;
.
.

// Reduce results
dp                               += __shfl(dp, wid ^ 1);
dp                               += __shfl(dp, wid ^ 2);
dp                               += __shfl(dp, wid ^ 4);
dp                               += __shfl(dp, wid ^ 8);
dp                               += __shfl(dp, wid ^ 16);
```

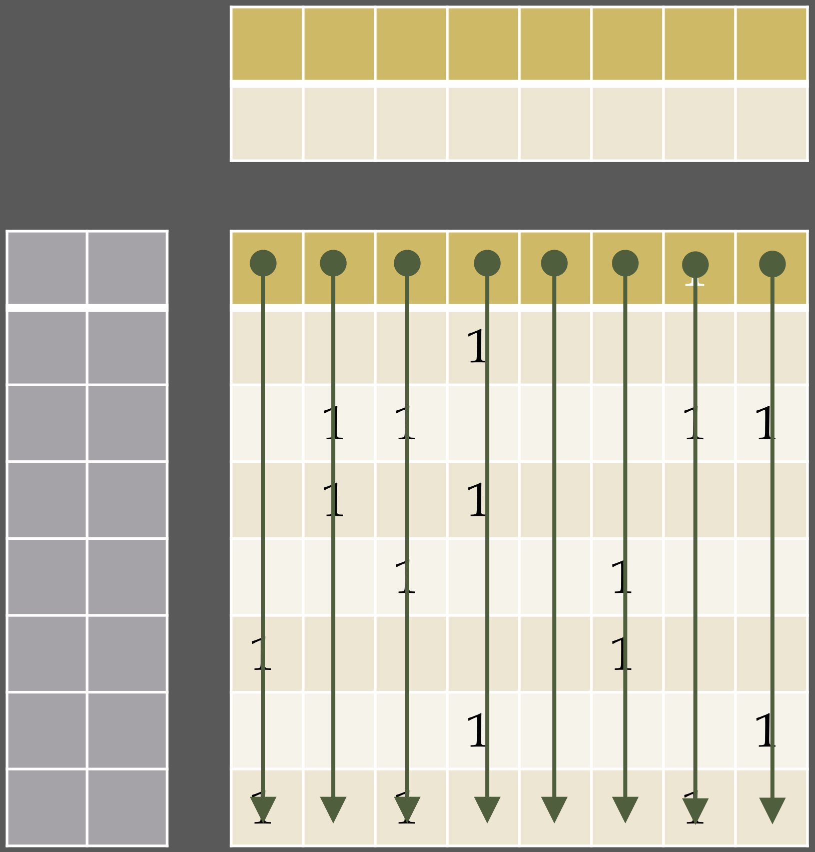
Expectation Maximization

						1	
			1				
	1	1				1	1
	1		1				
		1			1		
1					1		
			1				1
1		1				1	

Expectation Maximization



Expectation Maximization



Dynamic Range and Molecular Dynamics

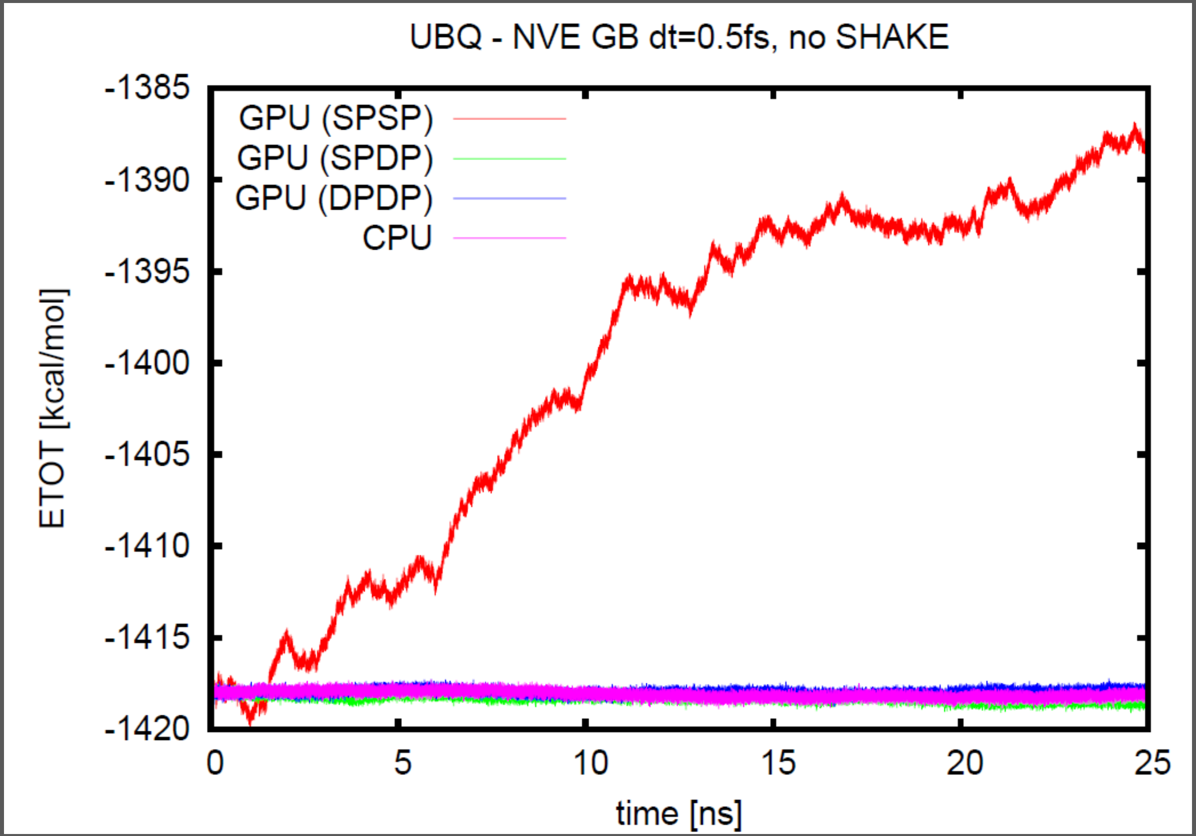
32-bit floating point has approximately 7 significant figures

```
  1.4567020
 +0.3046714
 ---------------
  1.7613730
 -1.4567020
 --------------
  0.3046710
Lost a sig fig
```

```
  1456702.0000000
+       0.3046714
-------------------------
  1456702.0000000
 -1456702.0000000
-------------------------
        0.0000000
Lost everything.
```

When it happens: PBC, SHAKE, and Force Accumulation in MD, backpropagation and recurrence in Neural Networks

Dynamic Range Matters… A Lot…



Reproducible Results Matter… A Lot…

Can you spot the defective GPU?

GPU #1	GPU #2
ETot = -288,718.2326	ETot = -288,718.2326
ETot = -288,718,2325	Etot = -288,718,2326

Let’s make it easier…

GPU #1	GPU #2
ETot = -288,718.2326	ETot = -288,718.2326
ETot = -288,718,2325	Etot = -288,718,2326

Non-Deterministic Single-Precision

GPU #1	GPU #2
ETot = -288,456.6774	ETot = -288,458.5931
ETot = -288,453.8133	Etot = -288,454.1539

GeForce GPUs are not QAed for HPC/ML

Hear Me Now, Believe Me Later

“If your massively parallel code isn’t deterministic, it’s crap.”

Deterministic Stable MD (using single-precision)

- Acceptable force error is ~10^{-5}
- Single-precision error is ~10^{-7}
- So calculate forces in single precision, but accumulate in extended precision
- Before Kepler, we used double-precision
- GK104 made it necessary to switch to 64-bit fixed point
- But this then allowed us to exploit its fast Atomic Adds for accumulation

Use 64-bit fixed point for accumulation

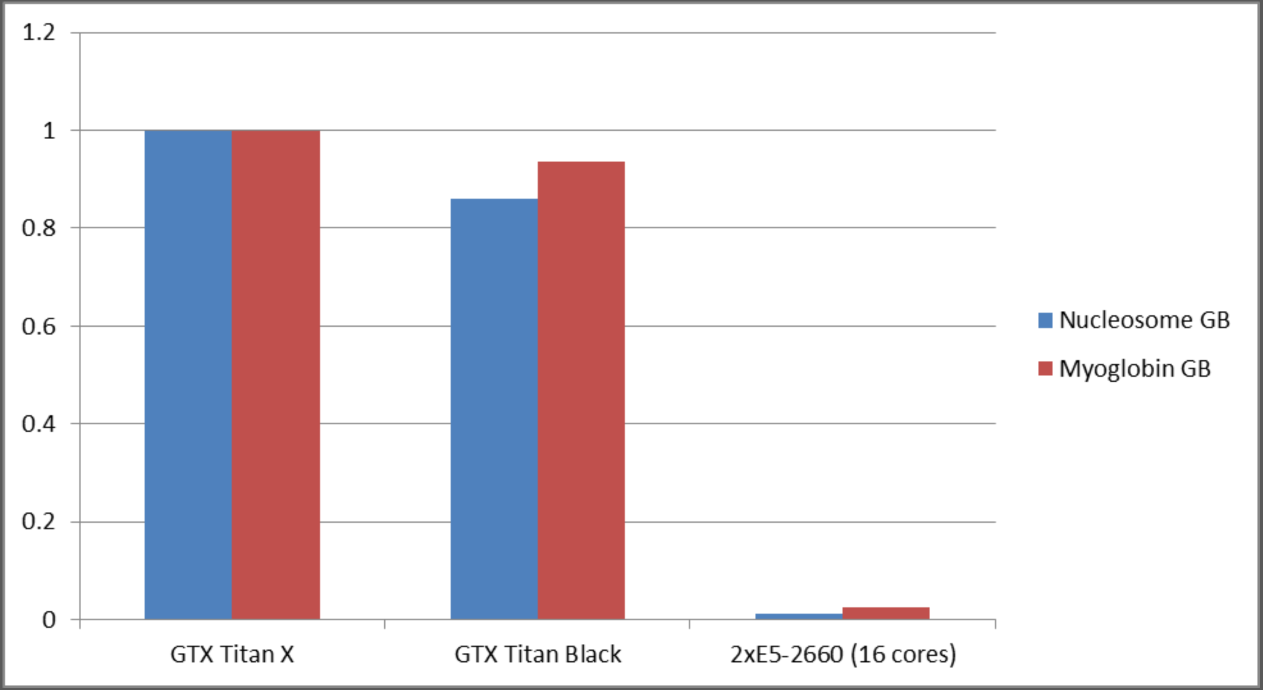
- Each iteration of the main kernel in PMEMD uses 9 double-precision operations
- Fermi double-precision was ¼ to 1/10th of single-precision
- GTX6xx double-precision is 1/24th single precision!
- So accumulate forces in 64-bit fixed point
- Fixed point forces are *perfectly* conserved
- 3 double-precision operations per iteration
- Integer extended math (add with carry) is 32-bit!

Associativity

Floating Point: A + B + C + D != C + D +A + B

Fixed Point: A + B + C + D == C + D + A + B

Generalized Born Performance



Along Came GM2xx

- On GM2xx, double-precision was further reduced to 1/32 that of single-precision whilst nearly doubling attainable single-precision performance (GM200 versus GK110, GM204 versus GK104)
- Initially GM204 is slightly better than GTX 780, GM200 ~20% better than GK110
- Fortunately, we had a solution waiting in the wings that we developed for GK1xx

Use 2 x 32 bits (~48-bit FP)

Extended-Precision Floating-Point Numbers for GPU Computation - Andrew Thall, Alma College

http://andrewthall.org/papers/df64_qf128.pdf

High-Performance Quasi Double-Precison Method Using Single-Precision Hardware for Molecular Dynamics on GPUs – Tetsuo Narumi *et al.*

HPC Asia and APAN 2009

Narumi Summation

Represented as a float and an int

```
const int   NARUMI_LARGE_SHIFT          = 21;
const float NARUMI_LARGE                = (float)(1 <<
(NARUMI_LARGE_SHIFT - 1));

struct Accumulator {
    float hs;
    int   li;
    Accumulator() : hs(NARUMI_LARGE), li(0) {}
};
```

Addition

```
void add_narumi(Accumulator& a, float ys)
{
    float hs, ls, ws;

    // Knuth and Dekker addition
    hs              = a.hs + ys;
    ws              = hs - a.hs;
    ls              = ys - ws;

    // Inner Narumi correction
    a.hs            = hs;
    a.li           += (int)(ls * NARUMI_LOWER_FACTOR);
}
```

Conversion to double

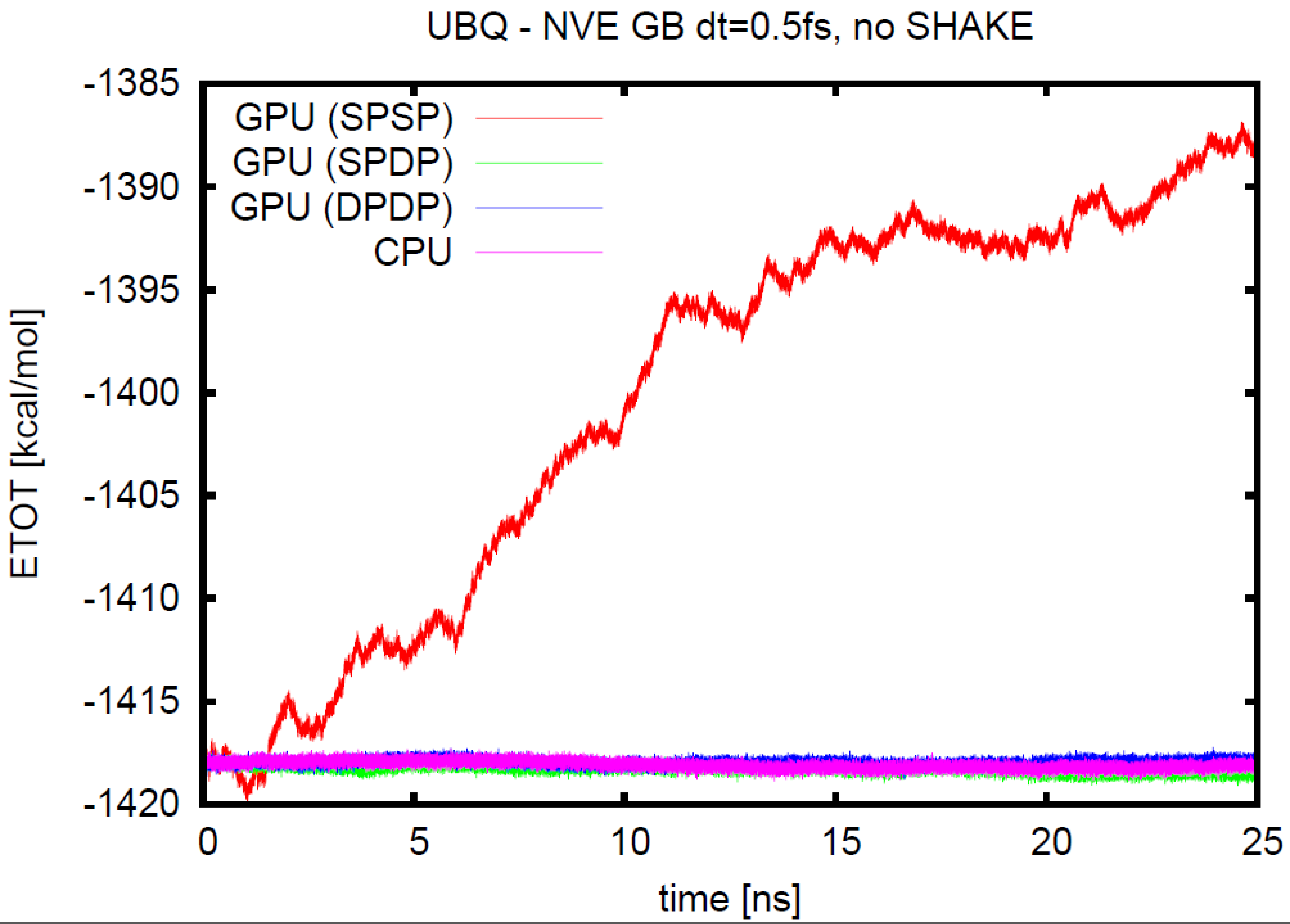
```
double upcast_narumi(Accumulator& a)
{
    double d        = (double)(a.hs - NARUMI_LARGE);
    d              += a.li * NARUMI_LOWER_FACTOR_1_D;
    return d;
}
```

Something for Everyone

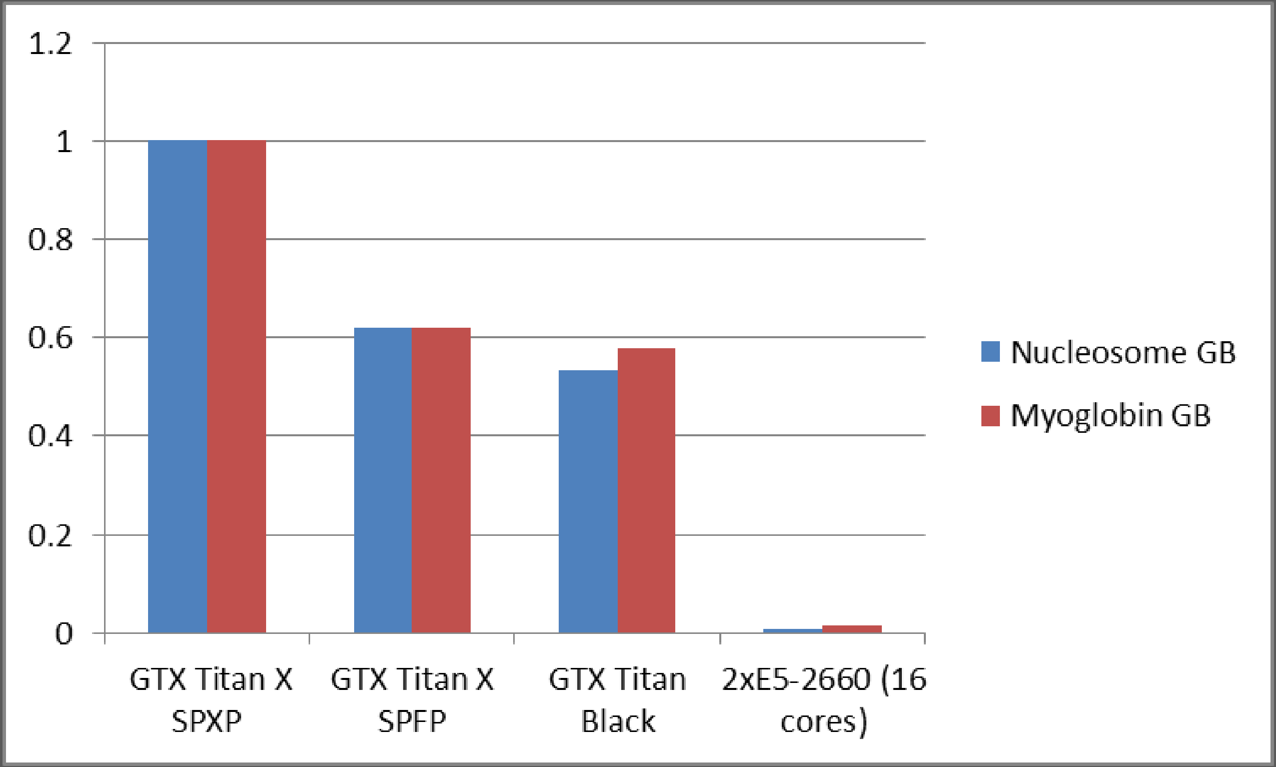
DPFP	64-bit everything
SPFP	32-bit forces, U64 force summation, 64-bit state
SPXP	32-bit forces, Narumi force summation for inner loops, U64 summation, 64-bit state

Side by Side

DP:	22.855216396810960
DPFP:	22.855216396810960
SPFP:	22.855216396810xxx
SPXP:	22.8552163xxxxxxxx
SP:	22.855xxxxxxxxxxxx


UBQ - NVE GB dt=0.5fs, no SHAKE
GPU (SPSP)
GPU (SPDP)
GPU (DPDP)
CPU
ETOT [kcal/mol]
-1385
-1390
-1395
-1400
-1405
-1410
-1415
-1420
0
5
10
15
20
25
time [ns]

If All Goes Well…



Neural Networks

- World’s most lucrative application of the chain rule from calculus
- x is the input data
- A1 and A2 are linear transformations
- f1 and f2 are some sort of nonlinear function

$$\begin{tikzcd}
x \arrow[r] & A1 \arrow[r] & f1 \arrow[r] & A2 \arrow[r] & f2
\end{tikzcd}$$

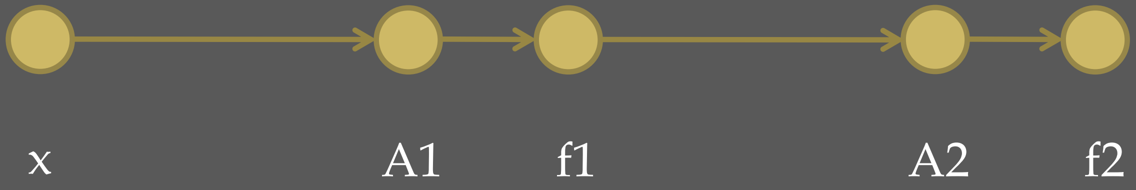
$$y = f2\left(A2\left(f1\left(A1(x)\right)\right)\right)$$

Nonlinear functions

- Linear: =x
- Sigmoid: $=\frac{1}{1+e^{-x}}$
- Tanh: $=\frac{e^{x}+e^{-x}}{e^{x}-e^{-x}}$
- Relu: =max(x, 0)
- SoftPlus: =log(1+e^{x})
- SoftSign: $=\frac{1}{1+|x|}$
- SoftMax: $=\frac{e^{x_i}}{\sum_j e^{x_{ij}}}$

Neural Network Training

Training: Minimize an Error Function E(y, t)



L1: $E(y, t) = |y - t|$

L2: $E(y, t) = (y - t)^2$

Cross Entropy: $E(y, t) = -t*\log(y) - (1-t)*\log(1-y)$

Neural Network Derivatives (BackPropagation)

$$\begin{tikzcd}
x \arrow[r] & A1 \arrow[r] & f1 \arrow[r] & A2 \arrow[r] & f2
\end{tikzcd}$$

$$\frac{@E}{@x} = \frac{@E}{@f2}\frac{@f2}{@A2}\frac{@A2}{@f1}\frac{@f1}{@A1}\frac{@A1}{@x}$$

$$\frac{@E}{@A2_{ij}} = \frac{@E}{@f2}\frac{@f2}{@A2}\frac{@A2}{@A2_{ij}}$$

$$\frac{@E}{@A1_{ij}} = \frac{@E}{@f2}\frac{@f2}{@A2}\frac{@A2}{@f1}\frac{@f1}{@A1}\frac{@A1}{@A1_{ij}}$$

A Bunch of Muls and Adds

Neural Network backpropagation faces the twin dilemmas of vanishing and exploding gradients

Molecular Dynamics force accumulation mostly faces exploding gradients

But both are dealing with dynamic range issues

16-bit Floating Point only has 3 significant figures…

Image Data Only Has 2 Significant Figures

(But I’m wary of its general applicability, it was a disaster for Molecular Dynamics Forces)

The Answer Isn’t Always AlexNet

Classifying Plankton With Deep Neural Networks

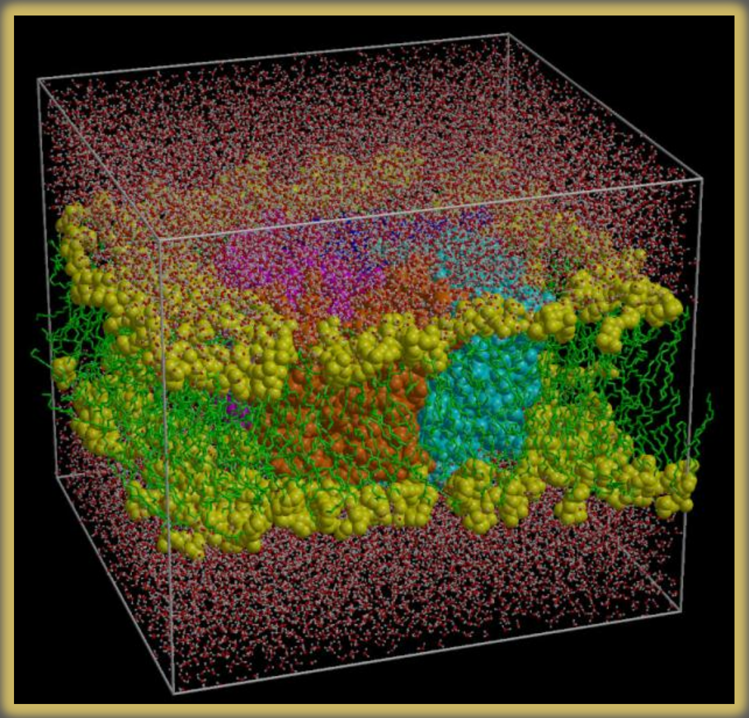
Sander Dieleman

http://benanne.github.io/2015/03/17/plankton.html

Middle Ground

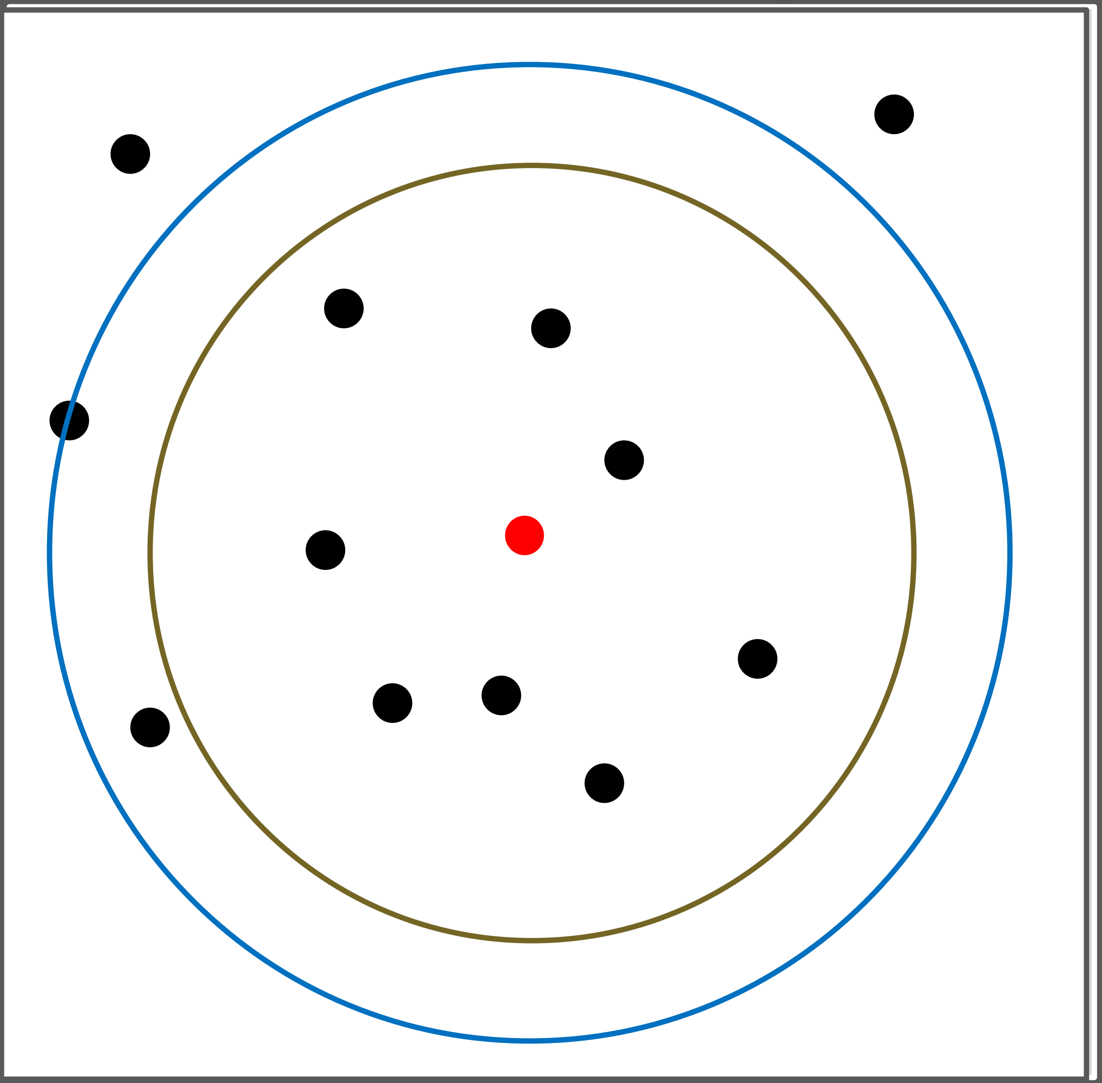
- Store weights, hidden units, deltas, etc. as FP16 and get all the bandwidth acceleration
- For training, do all math in FP32
- All CUDA-capable GPUS support this already
- If it works, do prediction in FP16 on Pascal

Particle Mesh Ewald (PME)

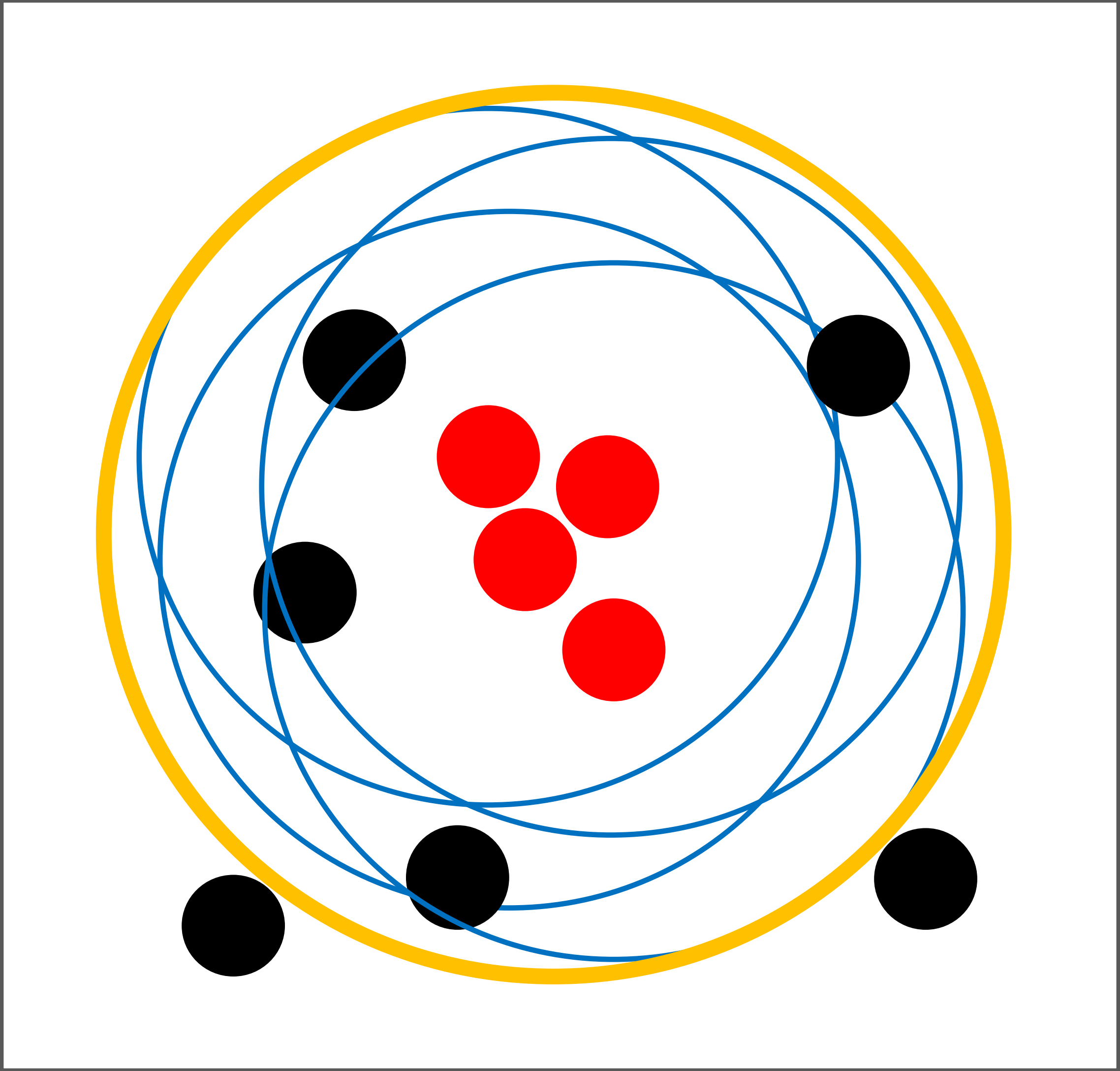


- O(N log N) Annoyingly Parallel
- More relevant than Generalized Born
- Rate-limited by a 3D FFT
- Approximates long-range interactions

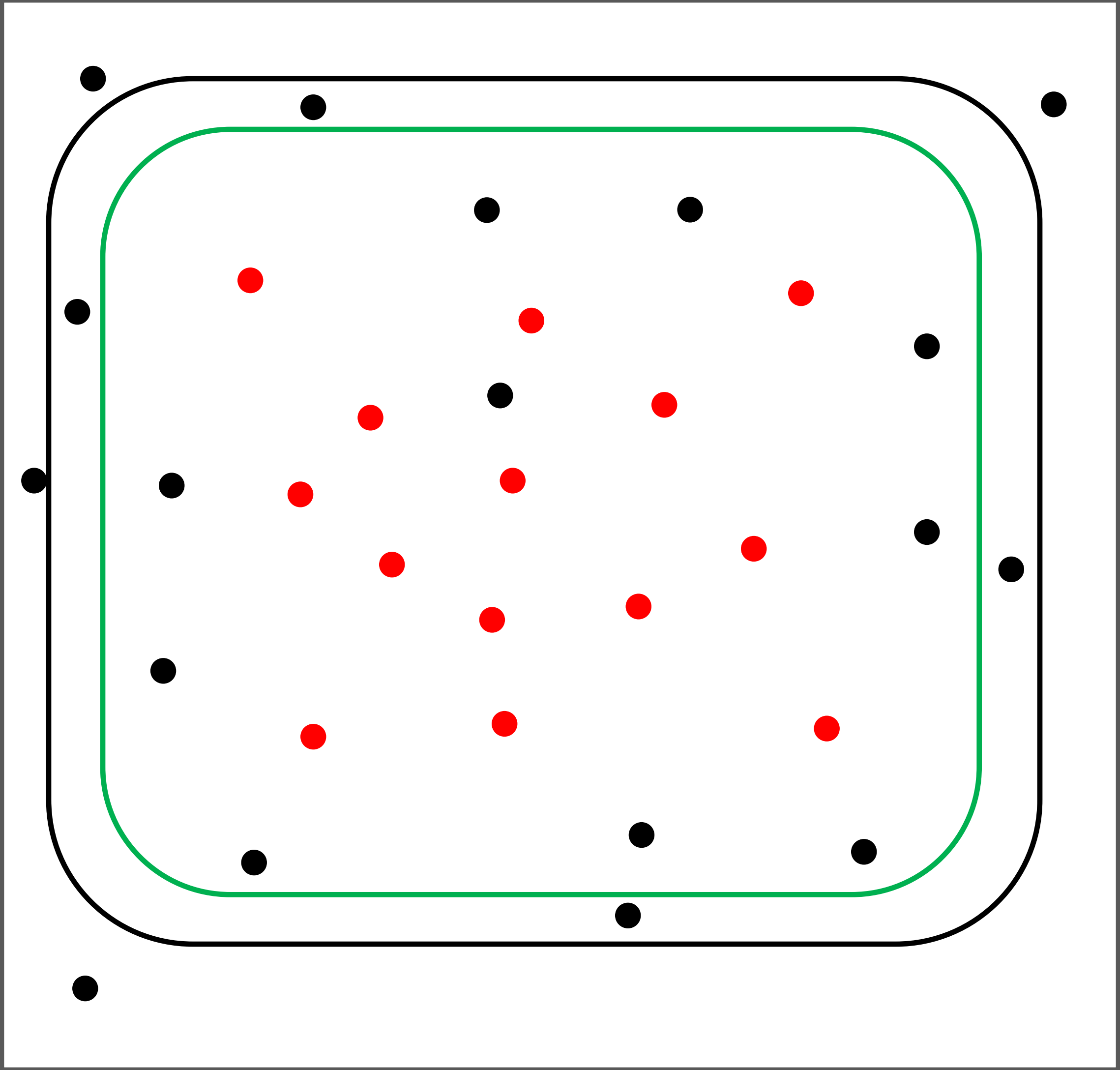
Nonbond Cutoff plus Skin



In Parallel



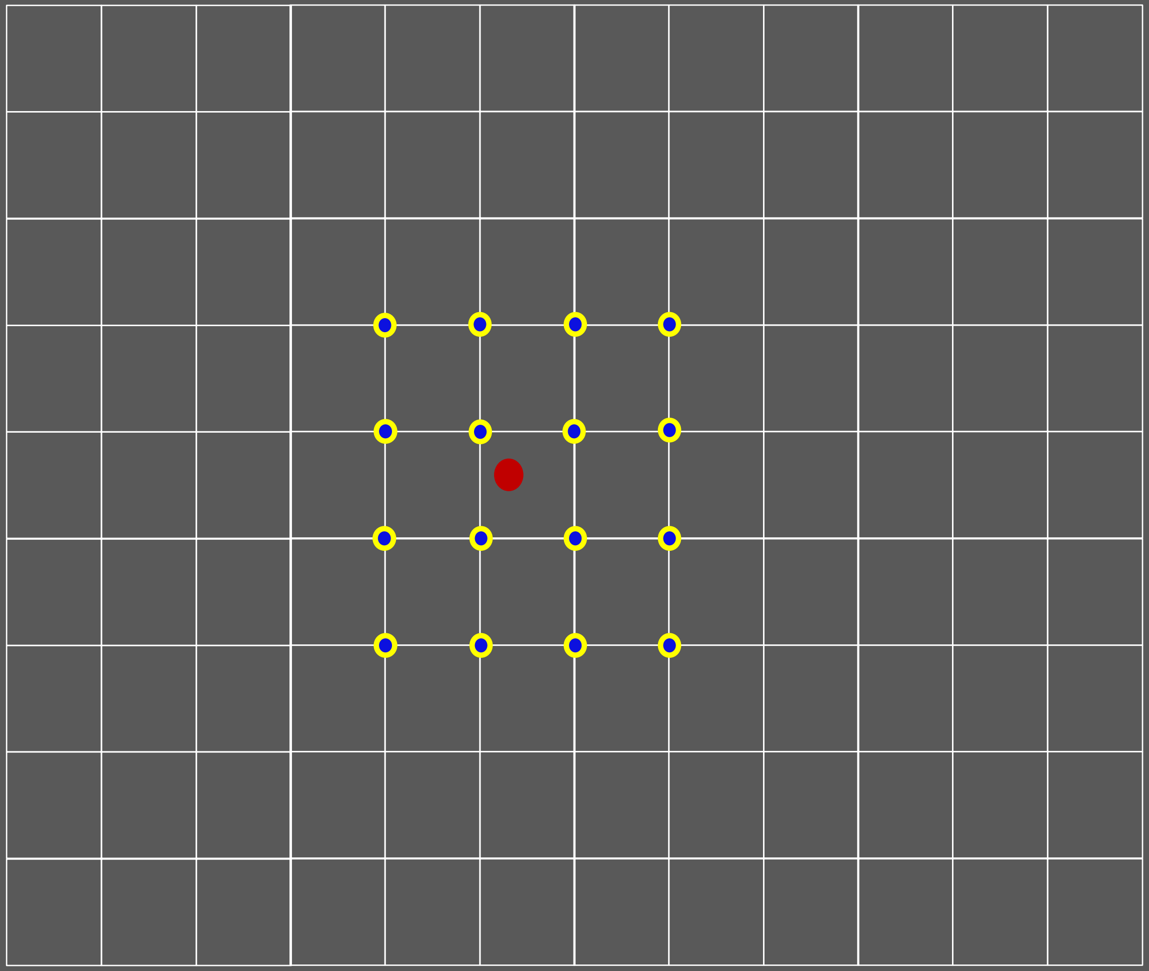
Bounding Boxes



Inner Loop

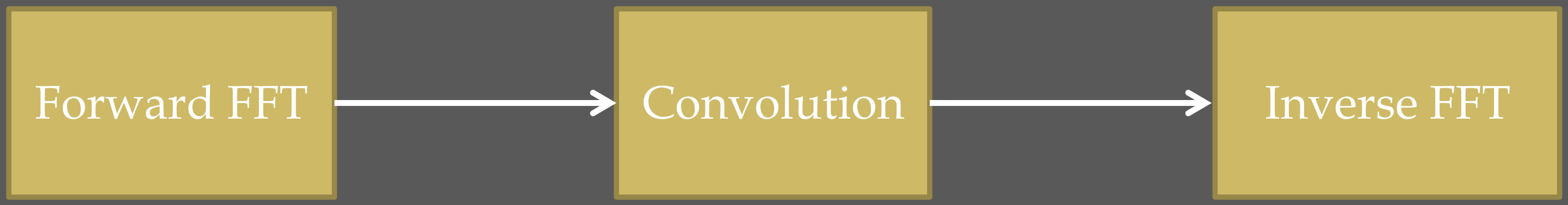
```
float xi        = pAtomX[i];
float yi        = pAtomY[i];
float zi        = pAtomZ[i];
float xj        = pAtomX[j];
float yj        = pAtomY[j];
float zj        = pAtomZ[j];
int pos         = theadIdx.x & 0x1f;
int shIdx     = (pos + 1) & 0x1f;
do
{
   float xij      = xi - xj;
   float yij      = yi - yj;
   float zij      = zi - zj;
   float r2    = xij * xij + yij * yij + zij * zij;
   if (r2 < cutoff_squared)
   {
      float r      = sqrt(r2);
      .
      Calculate Forces (lots of Muls and Adds)
      .
   }
   xj             = __shfl(xj, shIdx);
   yj             = __shfl(yj, shIdx);
   zj             = __shfl(zj, shIdx);
   pos            = (pos + 1) & 0x1;
}
while (pos != ((threadIdx.x + 1) & 0x1f));
```

Reciprocal Forces

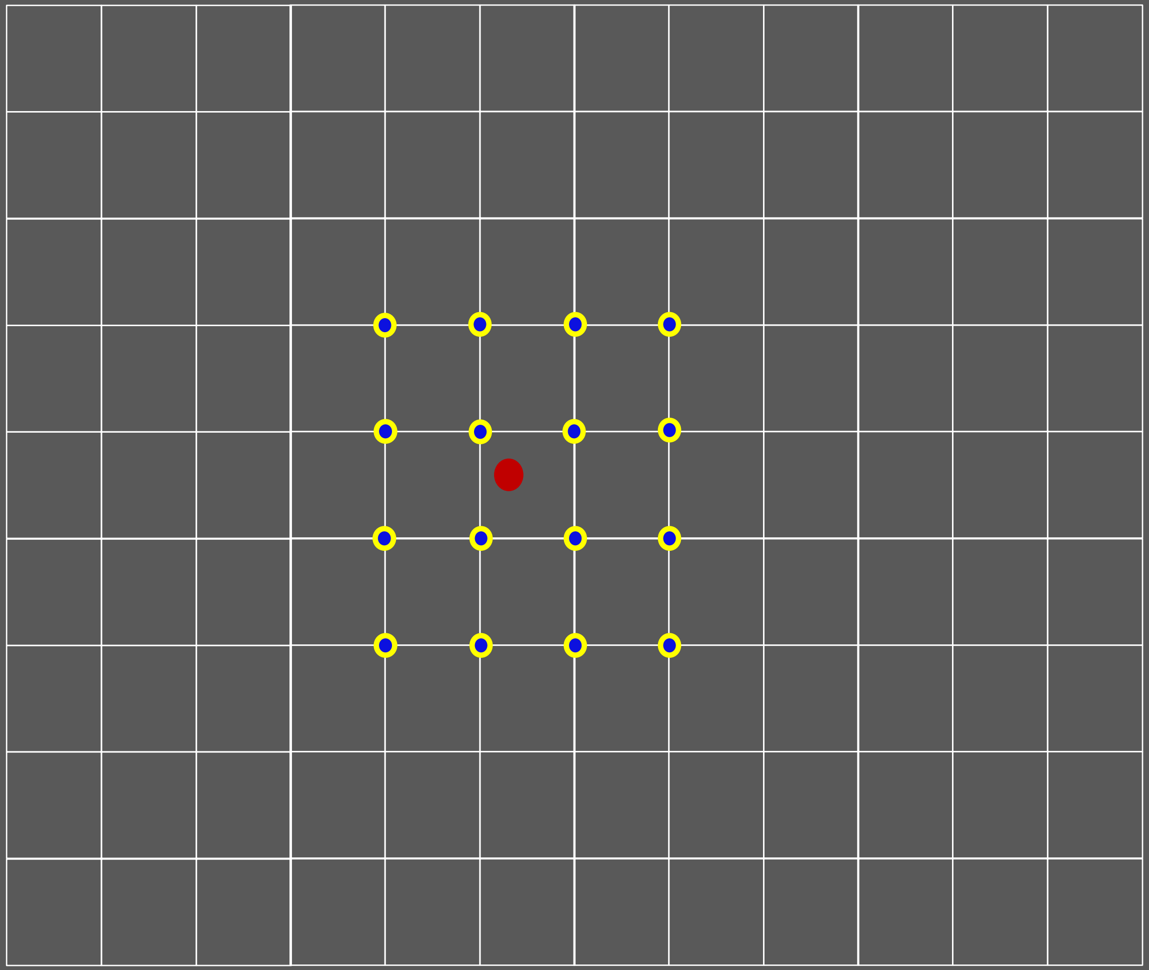


Spline Interpolate charges onto local 4x4x4 grid

Convolution



Reciprocal Forces



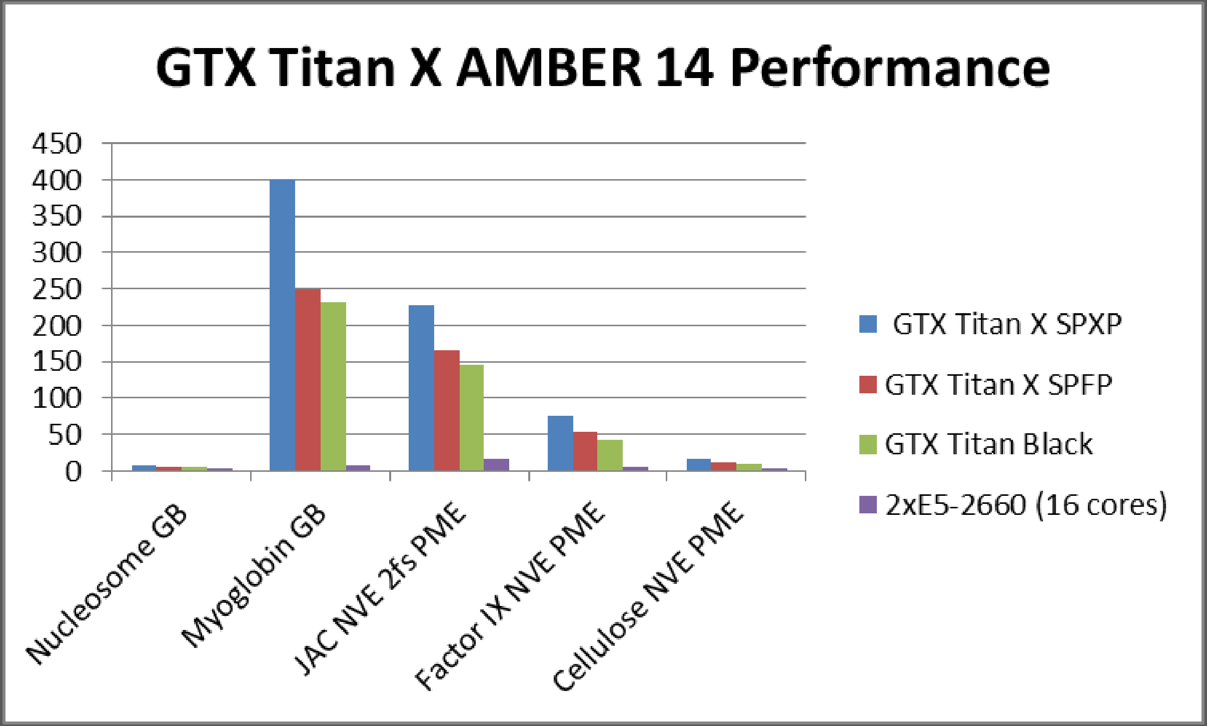
Spline Interpolate forces from local 4x4x4 grid

Not entirely unlike CNNs

Fast Training of Convolutional Networks through FFTs - Michael Mathieu, Mikael Henaff, Yann LeCun

Fast Convolutional Nets With fbfft: A GPU Performance Evaluation - Nicolas Vasilache, Jeff Johnson, Michael Mathieu, Soumith Chintala, Serkan Piantino, Yann LeCun

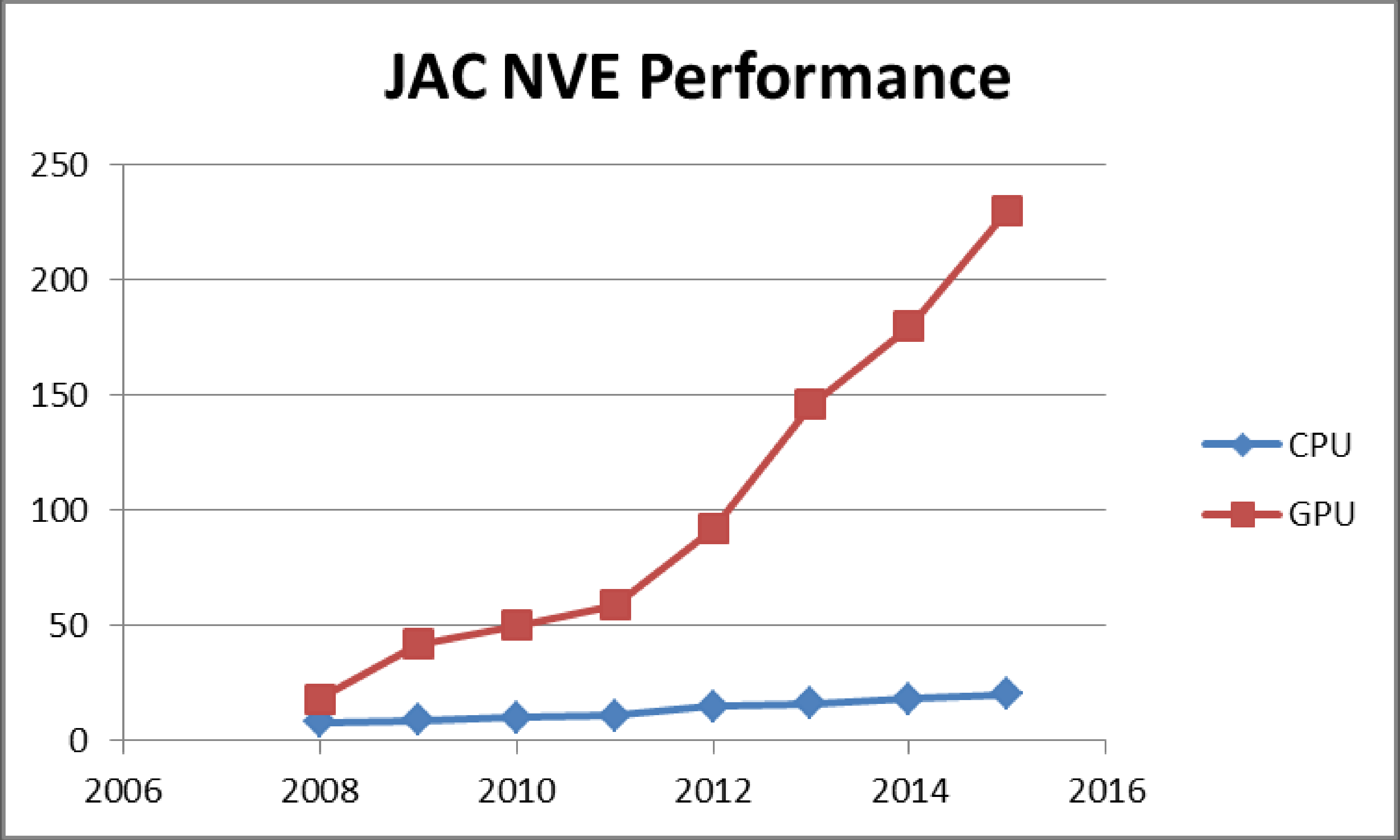
Performance (AMBER 14)



Should You Learn CUDA?

- Why didn't I just use SSE/AVX/FPGAs/Xeon Phi etc?
- Without a ground up redesign tailored to each platform, it just doesn't work
- Don't believe me? Go ahead, make my day…
- Caffe, Theano, Cuda-Convnet are GPU-resident
- Why not OpenCL? Not free on x86, no cuFFT, or cuBLAS, and AMD GPU drivers still suck

AMBER Performance



Two Worst Cases

1. Use Deep Neural Networks
2. ???
3. PROFIT!!!!

Two Worst Cases

1. Use Deep Neural Networks
2. ???
3. SKYNET!!!!*

*https://plus.google.com/101855192190887761500/posts/ETa2wt5J29k

Summary

- Everything we learned building Molecular Dynamics code applies to Machine Learning
- NVIDIA: I love GTX Titan X, but are you done crippling FP64 yet?
- But it's great for O(N^2) Neural Networks and Generalized Born MD
- SPXP validation coming soon

Acknowledgments

AMBER: Ross Walker, Perri Needham, Romelia Salomon-Ferrer, Levi Pierce, David Case, Adrian Roitberg, Jason Swails, Ben Madej, Grace Liu

Amazon: Rejith George Joseph, Vijai Mohan, Srikanth Thirumulai, Avishkar Misra, Leo Dirac, Matias Benitez, Nick Wilt

NVIDIA: Mark Berger, Duncan Poole, Simon Layton, Jerry Chen, and Sarah Tariq