Stream Computing for GPU-Accelerated HPC Applications

David Richie
Brown Deer Technology

April 6th, 2009

Air Force Research Laboratory
Wright-Patterson Air Force Base

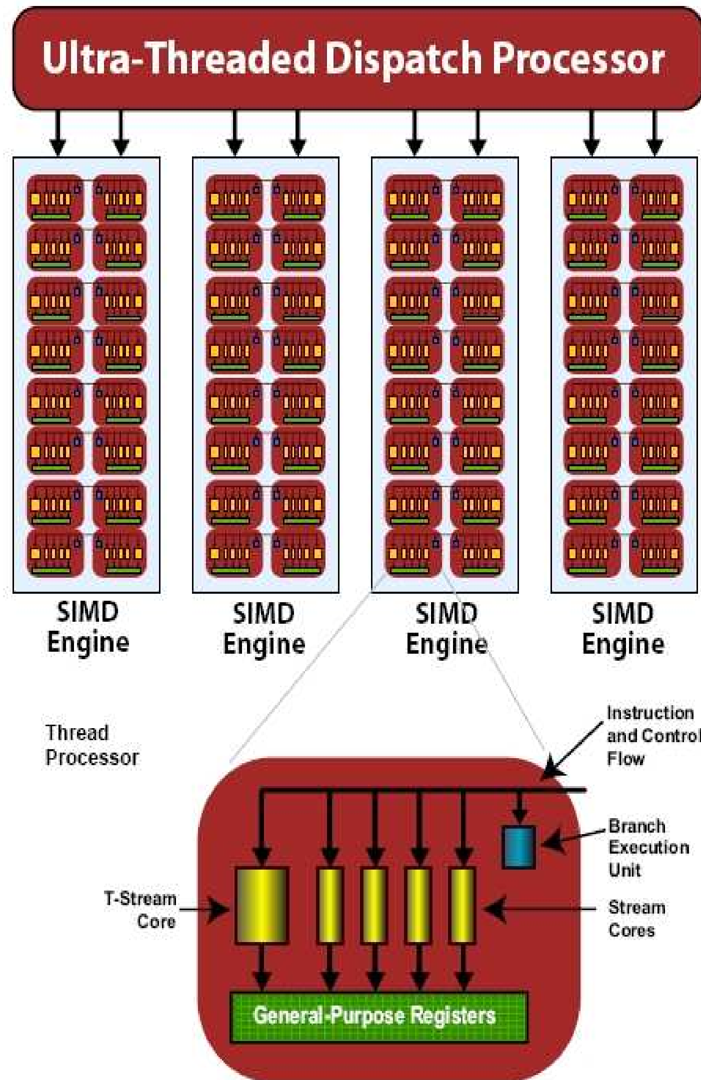
Copyright © 2009 Brown Deer Technology, LLC. All Rights Reserved.

Outline

- Modern GPUs
- Stream Computing Model
- Hardware/Software Test Setup
- Applications
  - Electromagnetics: 3D FDTD (Maxwell's Equations)
  - Seismic: 3D VS-FDTD (Elastic Wave Equation)
  - Quantum Chemistry: Two-Electron Integrals (STO-6G 1s)
  - Molecular Dynamics: LAMMPS (PairLJCharmmCoulLong)
- State of the Technology
- Conclusions

Copyright © 2009 Brown Deer Technology, LLC. All Rights Reserved.

Modern GPU Architectures



FireStream 9250

- AMD RV770 Architecture
- 800 SIMD superscalar processors
  - Supports SSE-like vec4 operations
  - IEEE single/double precision
- 1 TFLOP peak single precision
- 200 GFLOPS peak double-precision
- 1 GB GDDR3 on-board memory
- < 120 W max - 80 W typical
- MSRP $999

Copyright © 2009 Brown Deer Technology, LLC. All Rights Reserved.

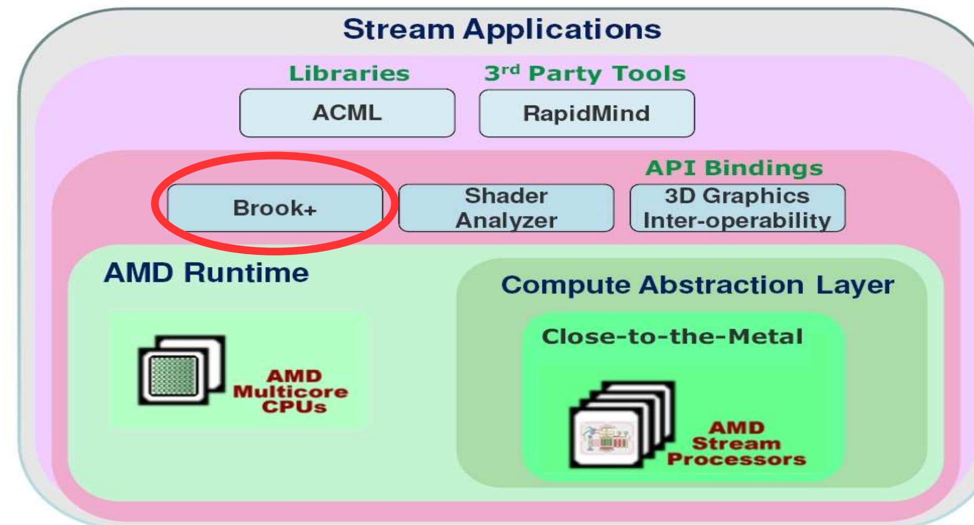
AMD/ATI GPU Form Factors

	clock[MHz]	memory	single	double	Power
Radeon HD 4850($170)	625	1GB GDDR3	1.0 TFLOPS	200 GFLOPS	
Radeon HD 4870($250)	**750**	**512MB GDDR5**	**1.2 TFLOPS**	**240 GFLOPS**	**160W**
Radeon HD 4870X2($430)	750	2GB GDDR5	2.4 TFLOPS	480 GFLOPS	
Radeon HD 4890($250)	850	1GB GDDR5	1.36 TFLOPS	272 GFLOPS	190W
Radeon HD 4890OC($265)	900	1GB GDDR5	1.44 TFLOPS	288 GFLOPS	
HPC:					
FireStream 9250($999)	625	1GB GDDR3	1.0 TFLOPS	200 GFLOPS	< 120W
FireStream 9270($1499)	750	2GB GDDR5	1.2 TFLOPS	240 GFLOPS	< 220W
1U Quad 9250 Server	625	4GB GDDR3	4.0 TFLOPS	800 GFLOPS	< 480W

*Data gathered April 4, 2009

Copyright © 2009 Brown Deer Technology, LLC. All Rights Reserved.

ATI Stream SDK



- **SDK (v1.3)**
- **Open-systems approach**
  - **Brook+ Compiler (C/C++ variant)** **BASIC**
  - **CAL low-level IL (generic ASM)** **EXPERT**
  - **CAL Run-Time API (C++)**
- **Stream paradigm:**
  - **Formulate algorithm as a SIMD kernel**
  - **Read/write streams between host/board**

Copyright © 2009 Brown Deer Technology, LLC. All Rights Reserved.

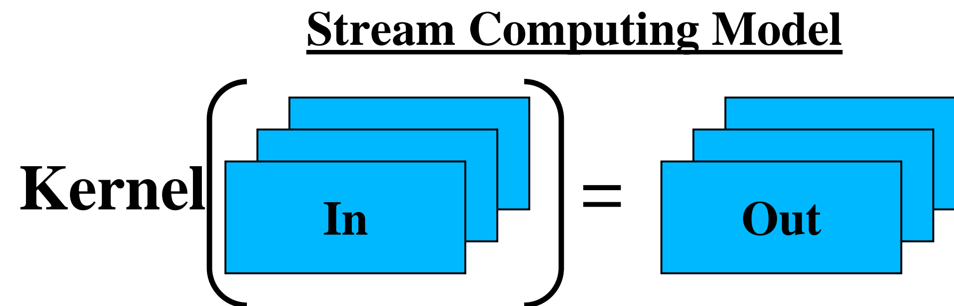
Stream Computing

- **Pure Stream Computing: Elegant, not useful.**
  - **Formulate algorithms based on the element-wise processing of multiple input streams into multiple output streams**

- **Pragmatic Stream Computing: Allows treatment of algorithms that do not fit a pure stream computing model – most algorithms fall in this category**
  - **Allows scatter/gather memory access which is needed in most algorithms**
  - **ATI Stream release of Brook+ compiler fits this model**

- **One or more computational kernels are applied to a 1D, 2D or 3D stream**
  - **SIMT domain driven implicitly by the dimensions of the out put stream**
  - **“Natural” streams are 2D, others use address translation**

Stream Computing Model



Copyright © 2009 Brown Deer Technology, LLC. All Rights Reserved.

Brook+ Programming Model

prog.cpp:

```
...
float a[256];
float b[256];
float c[256];

for(i=0;i<N;i++)
{ c[i] = a[i]*b[i]; }
...
```

prog.cpp~:

```
...
float a[256];
float b[256];
float c[256];

Stream<float> s_a(1,N);
Stream<float> s_b(1,N);
Stream<float> s_c(1,N);

// for(i=0;i<N;i++)
// { c[i] = a[i]*b[i]; }
s_a.read(a);
s_b.read(b);
foo_kern(s_a, s_b, s_c);
s_c.write(c);
...
```

foo.br:

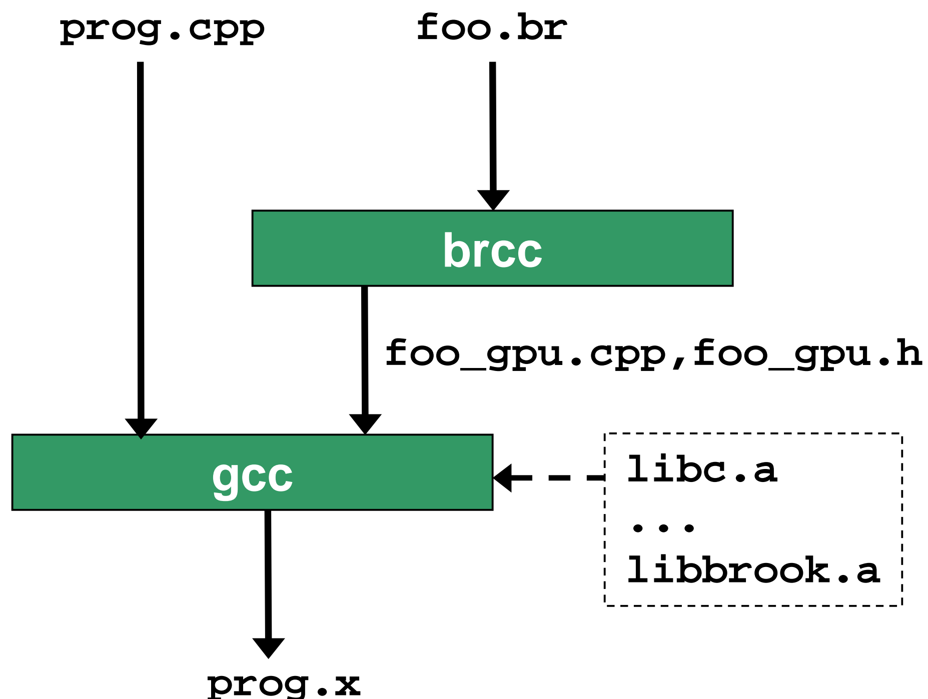
```
kernel void
foo_kern(
        float A<>,
        float B<>,
        out float
C<>
)
{ C = A * B; }
```

Recipe for Brook+ acceleration:

1) Identify critical loop or section
2) Create streams for datasets
3) Replace loop or section with
   a) Data transfer Host-to-GPU
   b) Execution of GPU kernel(s)
   c) Data transfer GPU-to-Host
4) Transform loop body or section into SIMD kernel,
5) And move it to a brook+ file (.br)

Copyright © 2009 Brown Deer Technology, LLC. All Rights Reserved.

Brook+ Workflow



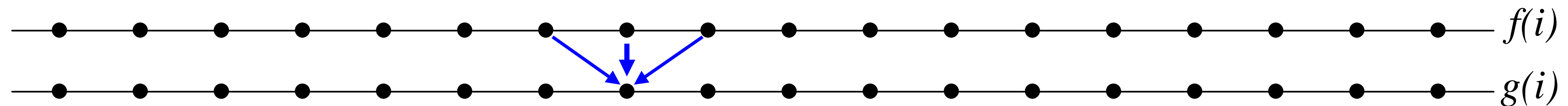
Makefile

```
NAME = prog
BRSRCS = foo.br
OBJS += $(BRSRCS:.br=.o)
INCS = -I/usr/local/atibrook/sdk/include
LIBS = -L/usr/local/atibrook/sdk/lib -lbrook
BRCC = brcc
all: $(NAME).x
$(NAME).x: $(NAME).o $(OBJS)
        $(CXX) $(CXXFLAGS) $(INCS) -o
$(NAME).x \
        $(NAME).o $(OBJS) $(LIBS)
.SUFFIXES:
.SUFFIXES: .br .cpp .o
.br.cpp:
        $(BRCC) $(BRCCFLAGS) -o $* $<
.cpp.o:
        $(CXX) $(CXXFLAGS) $(INCS) -c $<
```

Copyright © 2009 Brown Deer Technology, LLC. All Rights Reserved.

GPU Kernel: Memory Model

Simple example: local weighted average of a 1D array




```
 1:     kernel void
 2:     waverage3_kern(
 3:        float w0, float w1, float w2,
 4:        float s_f[],
 5:        out float s_g<>
 6:     )
 7:     {
 8:        float i1 = indexof(g);
 9:        float i0 = i1 - 1.0f;
10:        float i2 = i1 + 1.0f;
11:
12:        float f0 = f[i0];
13:        float f1 = f[i1];
14:        float f2 = f[i2];
15:
16:        float g = w0*f0 + w1*f1 + w2*f2;
17:
18:        s_g = g;
19:     }
```

(3) Simple scalar values, e.g., coefficients

(4) Input (gather) stream, any element is accessible like a normal array

(5) Output stream, kernel applied per element, only that element can be written

(8) `indexof()` returns index of stream element for this kernel invocation

(12-14) Gather (random) access needed for non-local stencil

(16) Local calculation of weighted sum

(18) Assign result to output stream, this must/can be done only once

Copyright © 2009 Brown Deer Technology, LLC. All Rights Reserved.

Performance Optimizations: float4 data

Simple Particle Pair Interactions

(pseudo-code)

```
for(i=0;i<natoms;i++) for(j=0;j<natoms;j++) {
   float4 pos0 = s_pos[i];
   float4 pos1 = s_pos[j];
   float4 d = pos1 - pos0;
   float rsq = d.x*d.x + d.y*d.y+ d.z*d.z;
   float f += (complicated function of rsq)
```

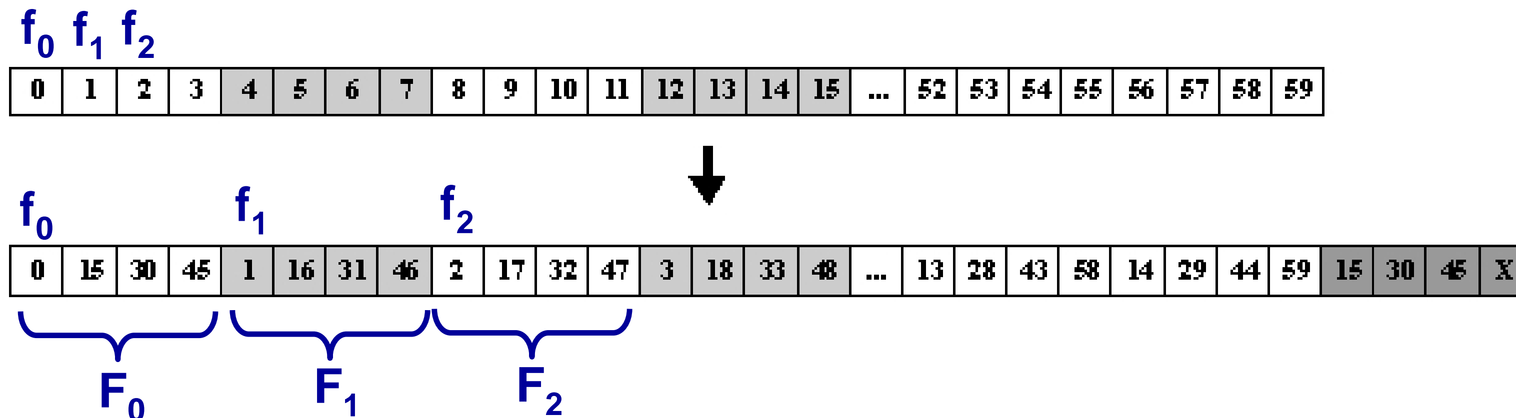
- "Obvious" choice - store particle positions in float4, (x,y,z) ~ pos.xyz
- Creates scalar bottleneck
- Calculates force on only one particle per kernel invocation

```
for(i=0;i<natoms/4;i++) for(j=0;j<natoms/4;j++) {
   float4 x0 = s_x[i];
   float4 y0 = s_y[i];
   float4 z0 = s_z[i];
   float4 x1 = s_x[j];
   float4 y1 = s_y[j];
   float4 z1 = s_z[j];
   float4 dx = x1 - x0;
   float4 dy = y1 - y0;
   float4 dz = z1 - z0;
   float4 rsq = dx*dx + dy*dy+ dz*dz;
   float4 f += (complicated function of rsq)
   /* now repeat 3 times, swizzeling x1,y1,z1 */
   float4 x1 = x1.yzwx; /* swizzel */
   ...
```

- Instead, pack quantited into separate streams to better exploit SIMD ops
- x0.xyzw ~ x-position of four particles, etc.
- Entire calculation exploits SIMD ops
- Calculates force on four particles per kernel invocation (~4x speedup)
- Requires some tricks, e.g., swizzeling

Copyright © 2009 Brown Deer Technology, LLC. All Rights Reserved.

Performance Optimizations: Shuffled Grids



- Consider simple 1D stencil: $g_1 = a*f_0 + b*f_1 + c*f_2$
- Would like to exploit float4 SIMD operations and update 4 points at once
- Problem: Stencils manipulate adjacent points $\Rightarrow$ mixing components of the 4-vectors
- Solution: “shuffle” the grid as shown to align adjacent points in same 4-vector positions
- Consider float4 SIMD operation using the shuffled grid: $G_1 = c_0*F_0 + c_1*F_1 + c_2*F_2$
- Equivalent to performing:

$$\left\{ \begin{array}{l} g_1 = a*f_0 + b*f_1 + c*f_2 \\ g_{16} = a*f_{15} + b*f_{16} + c*f_{17} \\ g_{31} = a*f_{30} + b*f_{31} + c*f_{32} \\ g_{46} = a*f_{45} + b*f_{34} + c*f_{47} \end{array} \right\}$$

- Shuffling/unshuffling the grids can be done as a pre- and post-processing step

Copyright © 2009 Brown Deer Technology, LLC. All Rights Reserved.

Test Setup

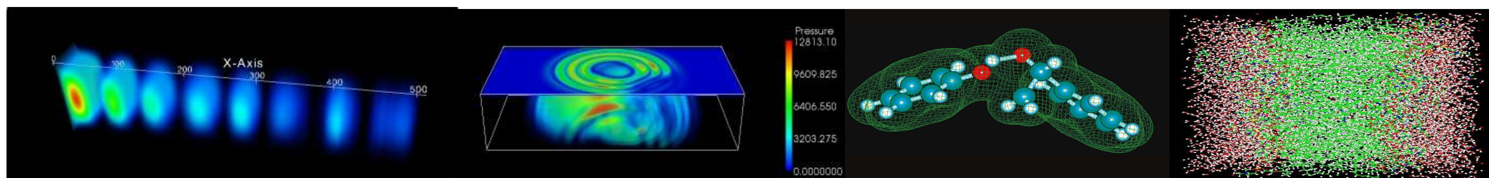
- Hardware
  - Host system was a simple desktop (CPU overclocked)
  - Phenom X2 9950 Black Edition 3.0 GHz overclock. (15x Multiplier)
  - ASUS M3A78-T motherboard
  - 4 GB OCZ ReaperDDR2 1066 MHz (5-5-5-18) Memory
  - AMD Radeon HD 4870 (512 MB GDDR5)
- Software
  - Scientific Linux 5.1
  - AMD SDK (Brook+ Compiler) v1.3
  - GCC-4.1

(Results for older hardware/software will be noted)

Copyright © 2009 Brown Deer Technology, LLC. All Rights Reserved.

Application Kernels

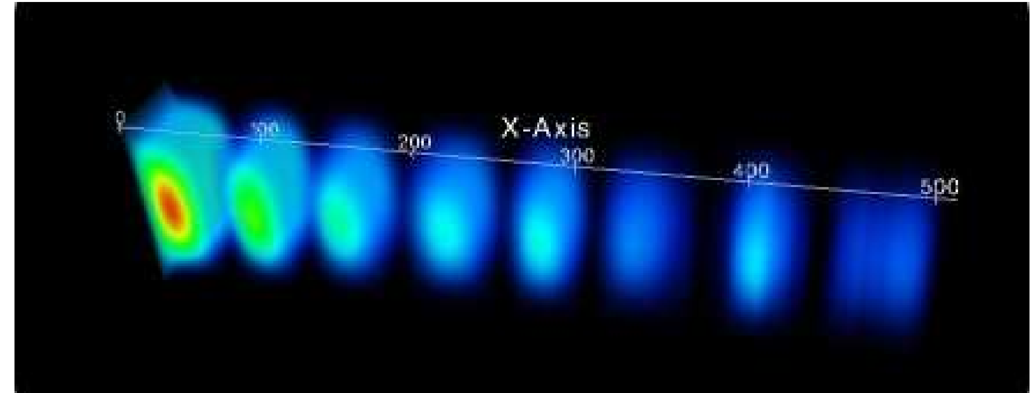
- Objectives
  - Evaluate representative computational kernels important in HPC
    - grids, finite-differencing, overlap integrals, particles
  - Understand GPU architecture, performance and optimisations
  - Understand how to design GPU-optimised stream applications
- Approach
  - Develop “clean” test codes, not full applications
    - Easy to instrument and modify
  - Exception is LAMMPS, a real production code from DOE/Sandia
    - Exercise was to investigate treatment of a “real code”
    - Brings complexity, e.g., data structures not GPU-friendly



Copyright © 2009 Brown Deer Technology, LLC. All Rights Reserved.

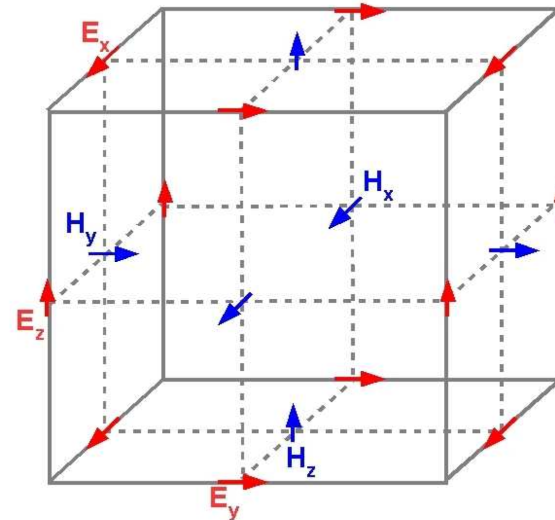
Electromagnetics: 3D FDTD

- **Direct iterative solution of Maxwell's Equations**
- **Important for modeling electromagnetic radiation from small devices to large-scale radar applications**
- **Grid-based finite-differencing**



$$\frac{\partial H_i}{\partial t} = a\left(\frac{\partial E_j}{\partial x_k} - \frac{\partial E_k}{\partial x_j}\right)$$

$$\frac{\partial E_i}{\partial t} = b\left(\frac{\partial H_k}{\partial x_j} - \frac{\partial H_j}{\partial x_k} - \sigma E_i\right)$$



Copyright © 2009 Brown Deer Technology, LLC. All Rights Reserved.

Electromagnetics: 3D FDTD: Implementation

GPU Acceleration

Initialization

N_step / N_burst

Stream Read

N_burst

Update E Field

Update H Field

Apply Excitation

Stream Write

Finalization

Update E Field

Update H Field

Apply Excitation

Copyright © 2009 Brown Deer Technology, LLC. All Rights Reserved.

Electromagnetics: 3D FDTD

```
kernel void hcomp_gpu(
         float t,
         float nx, float ny, float nz4,
         float bx, float by, float bz,
         float4 EX[][], float4 EY[][], float4 EZ[][],
         float4 HX0<>, float4 HY0<>, float4 HZ0<>,
         out float4 HX<>, out float4 HY<>, out float4 HZ<>
) {
         const float4 zero4 = float4(0.0f,0.0f,0.0f,0.0f);
         const float4 one4 = float4(1.0f,1.0f,1.0f,1.0f);
         const float4 bx4 = float4(bx,bx,bx,bx);
         const float4 by4 = float4(by,by,by,by);
         const float4 bz4 = float4(bz,bz,bz,bz);

         float2 i000 = indexof(HX).xy;

         float4 ix = float4(i000.y,i000.y,i000.y,i000.y);
         float iy1 = floor(i000.x/nz4);
         float4 iy = float4(iy1,iy1,iy1,iy1);
         float iz1 = i000.x - iy1*nz4;
         float4 iz = float4(iz1,iz1,iz1,iz1);

         float2 i100 = float2(i000.x,i000.y+1.0f);
         float2 i010 = float2(i000.x+nz4,i000.y);
         float2 i001 = float2(i000.x+1.0f,i000.y);
         float2 i00o = float2(iy.x*nz4,i000.y);
...
```

Copyright © 2009 Brown Deer Technology, LLC. All Rights Reserved.

Electromagnetics: 3D FDTD

```
...
        float mx1 = nx-1.0f;
        float4 mx = float4(mx1,mx1,mx1,mx1);
        float my1 = ny-1.0f;
        float4 my = float4(my1,my1,my1,my1);
        float mz1 = nz4-1.0f;
        float4 mz4 = float4(mz1,mz1,mz1,mz1);

        float4 tmphx = HX0;
        float4 tmphy = HY0;
        float4 tmphz = HZ0;

        float4 ex00o = EX[i00o];
        float4 ex000 = EX[i000];
        float4 ex001 = EX[i001];
        float4 ex010 = EX[i010];
        float4 ey00o = EY[i00o];
        float4 ey000 = EY[i000];
        float4 ey001 = EY[i001];
        float4 ey100 = EY[i100];
        float4 ez000 = EZ[i000];
        float4 ez010 = EZ[i010];
        float4 ez100 = EZ[i100];

        float4 maska,maskb,maskc;
...
```

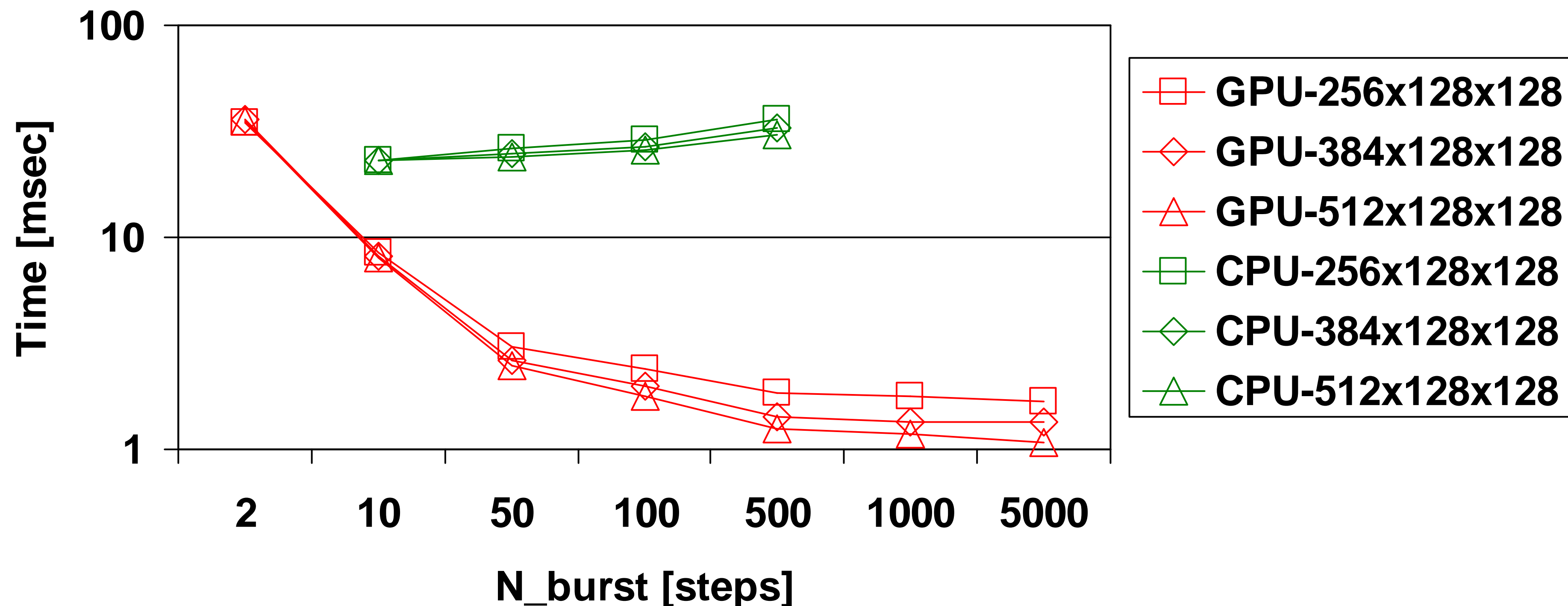
Copyright © 2009 Brown Deer Technology, LLC. All Rights Reserved.

Electromagnetics: 3D FDTD

```
...
        float4 maskx = one4;
        float4 masky = one4;
        float4 maskz = one4;
        float4 maskg = float4(1.0f,1.0f,1.0f,0.0f);

        ex001 = (iz == mz4)? ex00o.yzwx : ex001;
        ey001 = (iz == mz4)? ey00o.yzwx : ey001;

        maska = (ix == mx)? zero4 : one4;
        maskb = (iy == my)? zero4 : one4;
        maskc = (iz == mz4)? maskg : one4;

        tmphx += bz4 * (ey001 - ey000) + by4 * (ez000 - ez010);
        tmphy += bx4 * (ez100 - ez000) + bz4 * (ex000 - ex001);
        tmphz += by4 * (ex010 - ex000) + bx4 * (ey000 - ey100);

        tmphx *= maskb*maskc;
        tmphy *= maska*maskc;
        tmphz *= maska*maskb;

        HX = tmphx;
        HY = tmphy;
        HZ = tmphz;

}
```

Copyright © 2009 Brown Deer Technology, LLC. All Rights Reserved.

Electromagnetics: 3D FDTD



- Performing many iterations in between data transfer mitigates PCIe bottleneck
- 28x speedup for largest grid

Copyright © 2009 Brown Deer Technology, LLC. All Rights Reserved.

Seismic: 3D VS-FDTD

- Seismic Simulation of Velocity-Stress Wave Propagation
  - Important algorithm for seismic forward modeling techniques
  - Used for iterative refinement and validation of sub-surface geological models

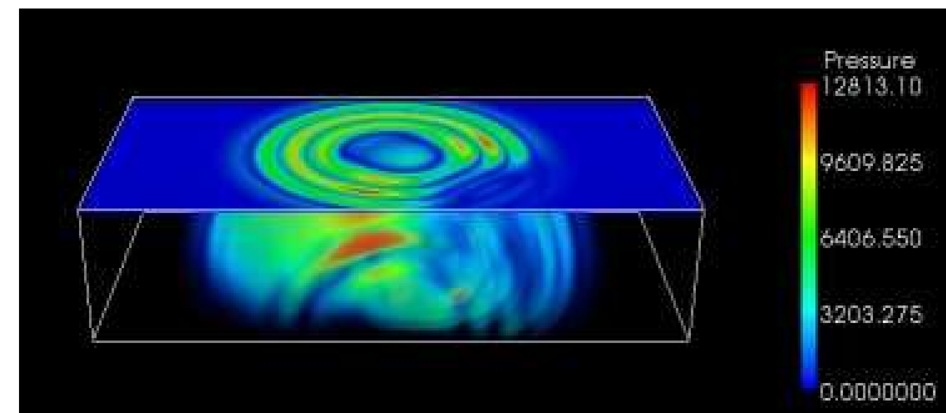
- Commercial applications for oil and gas exploration

- Military applications for detecting buried structures

$$\frac{\partial v_i}{\partial t} = b\left(\frac{\partial \sigma_{ii}}{\partial x_i} + \frac{\partial \sigma_{ij}}{\partial x_j} + \frac{\partial \sigma_{ik}}{\partial x_k} + f_i\right)$$

$$\frac{\partial \sigma_{ii}}{\partial t} = (\lambda + 2\mu)\frac{\partial v_i}{\partial x_i} + \lambda\left(\frac{\partial v_j}{\partial x_j} + \frac{\partial v_k}{\partial x_k}\right)$$

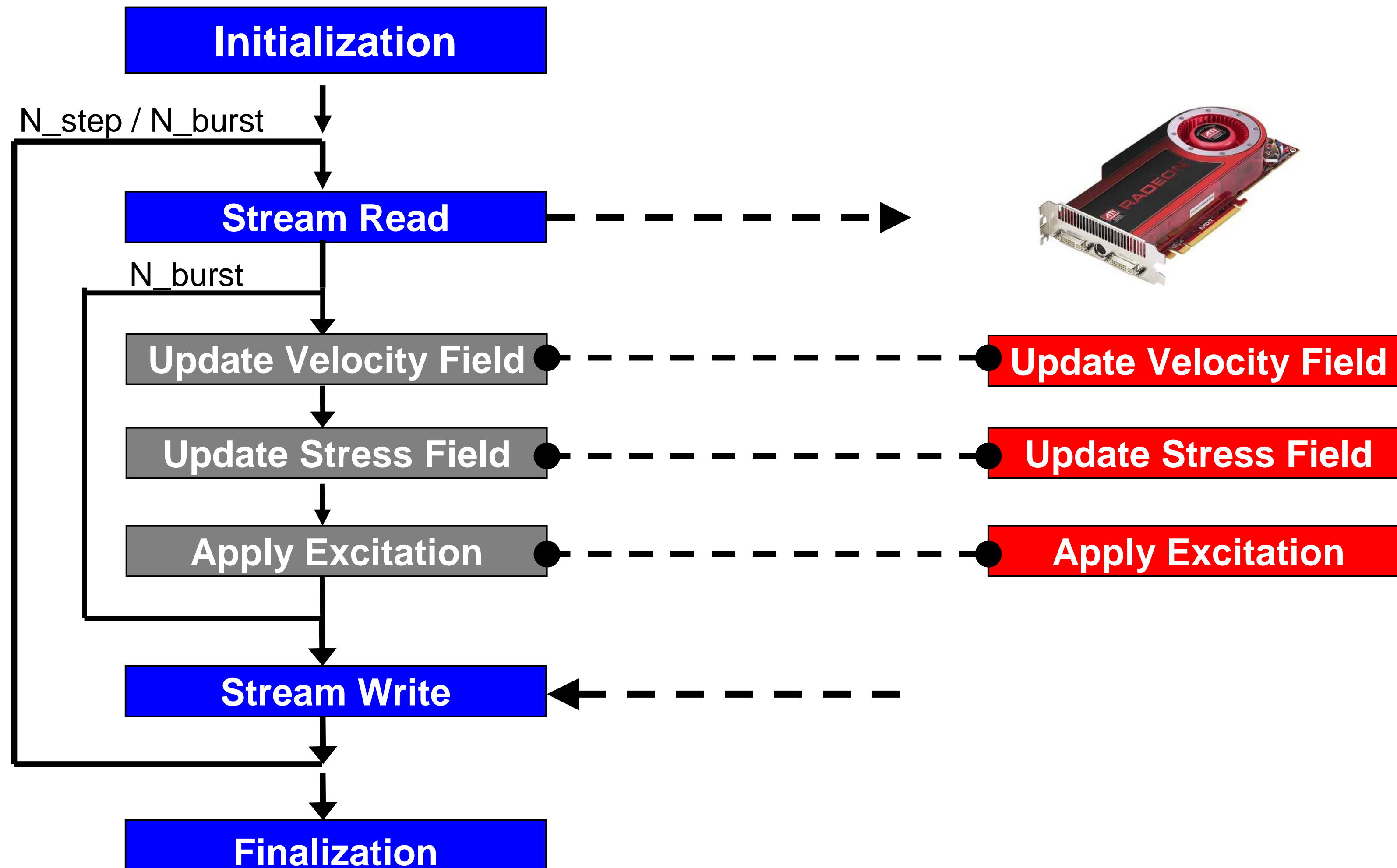
$$\frac{\partial \sigma_{ij}}{\partial t} = \lambda\left(\frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i}\right)$$



Copyright © 2009 Brown Deer Technology, LLC. All Rights Reserved.

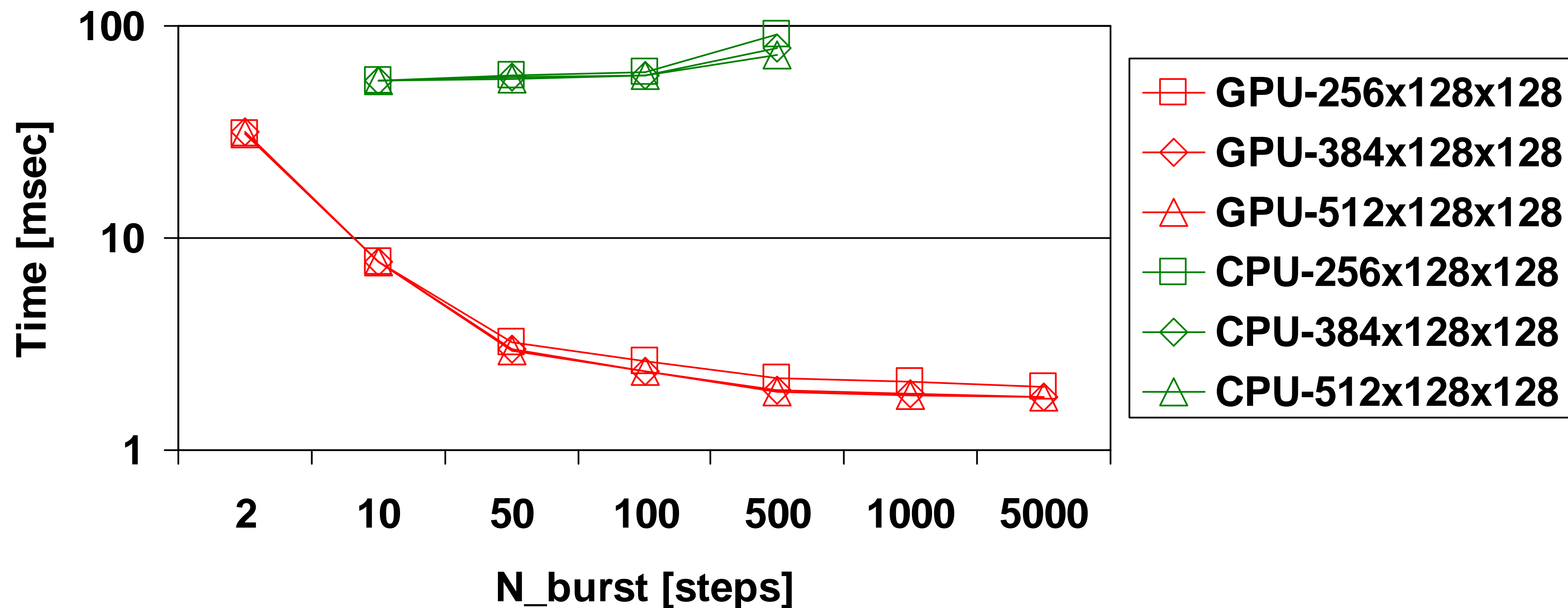
Seismic: 3D VS-FDTD

GPU Acceleration



Copyright © 2009 Brown Deer Technology, LLC. All Rights Reserved.

Seismic: 3D VS-FDTD: Benchmarks

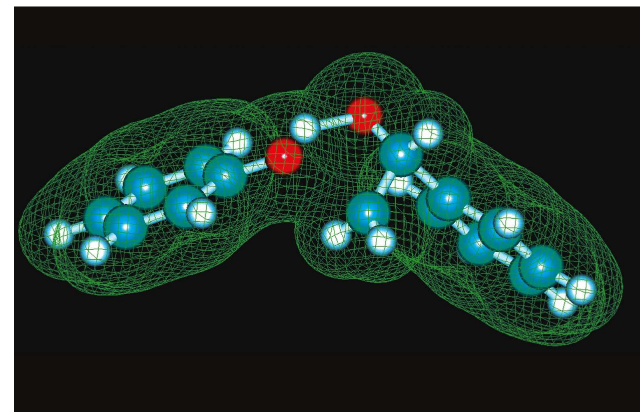


- Results similar to electromagnetic FDTD
- 31x speedup for largest grid

Copyright © 2009 Brown Deer Technology, LLC. All Rights Reserved.

Quantum Chemistry: Two-Electron Integrals

- One of the most common approaches in quantum chemical modeling employs gaussian basis sets to represent the electronic orbitals of the system
- A computationally costly component of these calculations involves the evaluation of two-electron integrals



$$(\mu\nu|\lambda\tau)=\iint dr_1\,dr_2\,\phi_\mu(r_1)\,\phi_\nu(r_2)\frac{1}{r_{12}}\phi_\lambda(r_1)\,\phi_\tau(r_2)$$

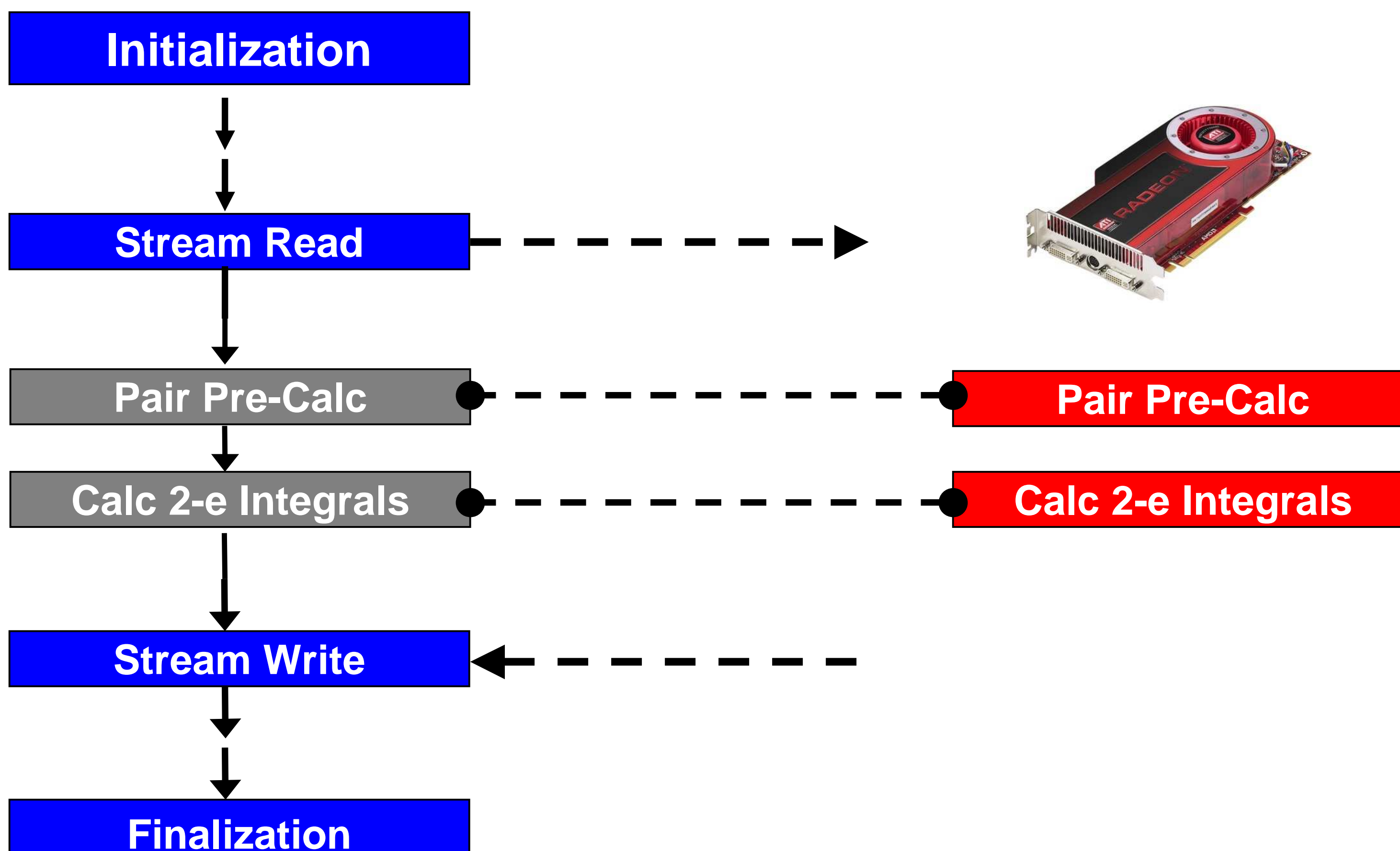
$$\phi_\mu(r)\sim\sum_k d_k\,g(\alpha_k,r) \qquad g(\alpha_k,r)\sim\exp(-\alpha_k r^2)$$

- For a gaussian basis, evaluation of two-electron integrals reduces to summation over closed-form expression (Boys, 1949)
- Features of expression required to be evaluated:
  - Certain pair quantities can be factored and pre-calculated
  - Expression contains +, -, *, /, sqrt(), exp(), erf()

Copyright © 2009 Brown Deer Technology, LLC. All Rights Reserved.

Quantum Chemistry: Two-Electron Integrals

GPU Acceleration



Copyright © 2009 Brown Deer Technology, LLC. All Rights Reserved.

Quantum Chemistry: Two-Electron Integrals

Implementation Details

- Consider simple test case: 3D lattice of Hydrogen atoms using a STO-6G basis (1s only)
- Evaluation of two-electron integrals reduces to many summations over 36•36= 1296 terms
- Use of float4 SIMD ops requires inner loop of only 36•9 iterations
- Use of double2 SIMD ops requires inner loop of only 36•18 iterations
- Most difficult part of implementation involved the erf() for which no hardware instr exists
- Most CPU-based codes us a piecewise approximation due to Cody (1968?)
  - Good for CPUs, reduces FLOPS at expense of branching
  - Terrible for GPUs, branching is a performance killer
- Used approximation by Hastings (1949?) valid for entire domain (with a few tricks)
  - Quality of the erf() approximation warrants further investigation
- Benchmarks performed for various lattice dimensions (Nx,Ny,Nz) leading to wide span in terms of number of integrals evaluated

Copyright © 2009 Brown Deer Technology, LLC. All Rights Reserved.

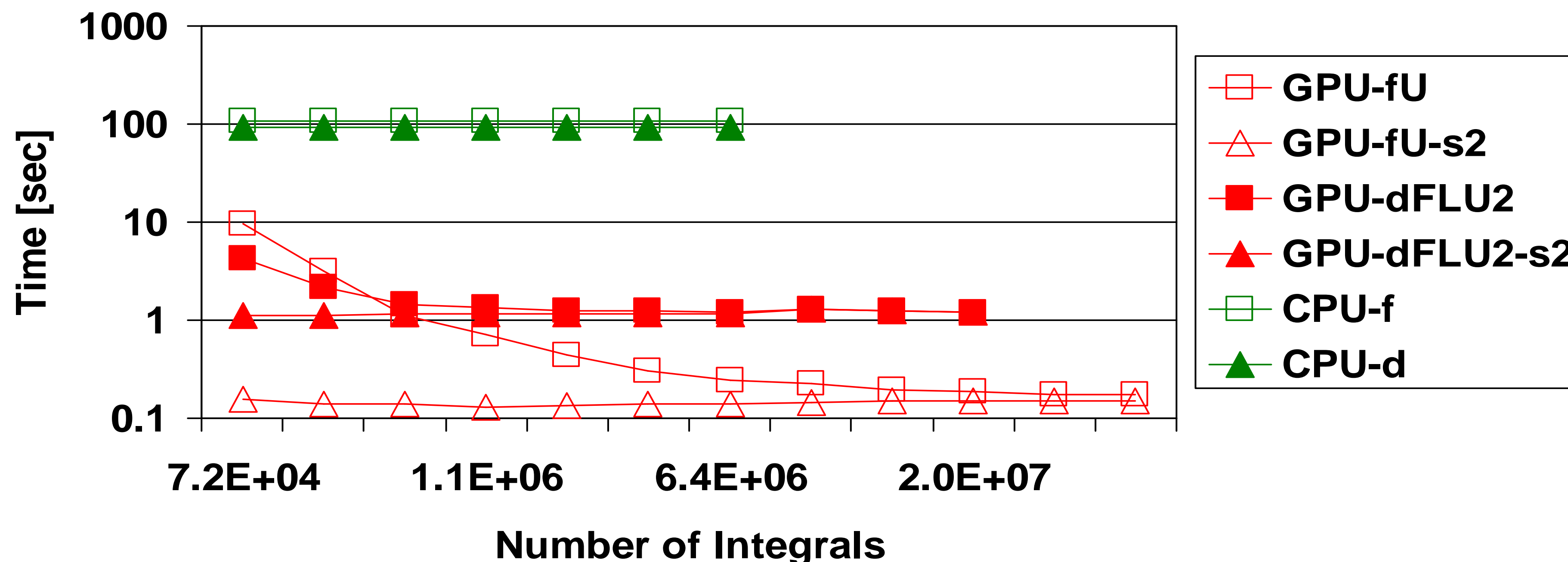
Quantum Chemistry: Two-Electron Integrals

GPU vs. CPU: Time per Million 2-e Integrals

- **Various implementations:**
  - **float(f)/double(d), Nested-Loop (NL), Fused-Loop (FL), Unrolled (U)**
- **Results are complex, reveal a lot about the architecture and run-time API**
- **Best float implementation: fully unrolled loop (9 iterations)**
- **Best double implementation: fused-loop w/partial (2 iteration) unroll**

Copyright © 2009 Brown Deer Technology, LLC. All Rights Reserved.

Quantum Chemistry: Two-Electron Integrals



- **Large numbers of integrals: latency and GPU setup time is completely amortized**
- **Small numbers of integrals: repeating calculation (s2) reveals GPU setup/compute time**
  - **Entire calculation is repeated including complete data transfer**
  - **s2 time more reflective of real codes (integrals re-evaluate repeatedly)**

Copyright © 2009 Brown Deer Technology, LLC. All Rights Reserved.

Quantum Chemistry: Two-Electron Integrals

STO-6G(1s) 4x4x4

	Total	GPU Setup	GPU Compute
ATI/4870/single	0.968 sec	0.678 sec	0.290 sec
AMD/9950(3GHz)/single	236.242 sec		
	244x		814x
ATI/4870/double	2.728 sec	0.241 sec	2.487 sec
AMD/9950(3GHz)/double	198.749 sec		
	72x		80x
Nvidia/8800GTX/single*	1.123 sec		
AMD/175/GAMESS*	90.6 sec		

*Ufimtsev and Martinez

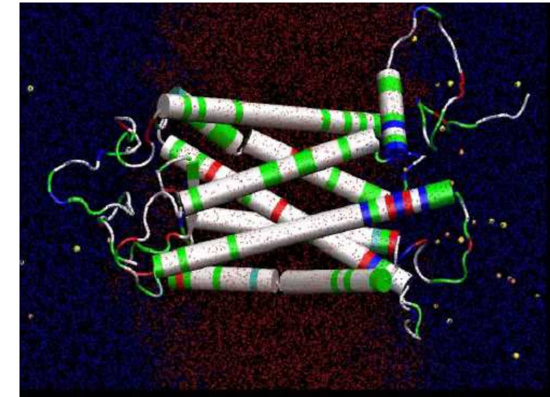
- **Large number of integral limit (~10 million)**
  - **SP: 774x speedup**
  - **DP: 77x speedup**
- **CPU implementation definitely not optimized**
- **GPU performance/speedup will nevertheless be substantial**

Copyright © 2009 Brown Deer Technology, LLC. All Rights Reserved.

Molecular Dynamics: LAMMPS

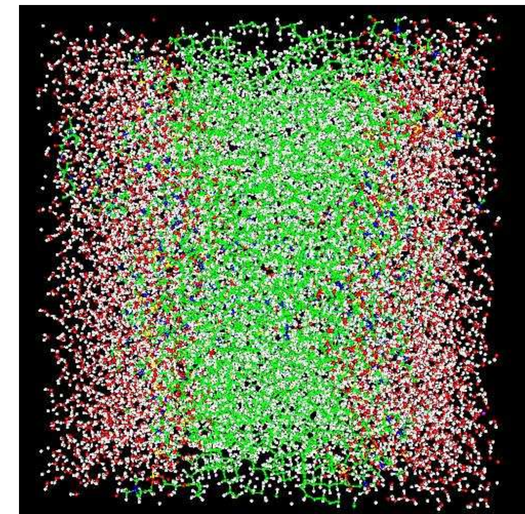
- Fundamental technique for molecular modeling
- Simulate motion of particles subject to inter-particle forces
- LAMMPS is open-source MD code from DOE/Sandia
  - Dr. Steve Plimpton, http://lammps.sandia.gov
- Goal: accelerate inter-particle force calculation

Rhodopsin Protein



*Original work due to Paul Crozier and Mark Stevens at Sandia National Labs

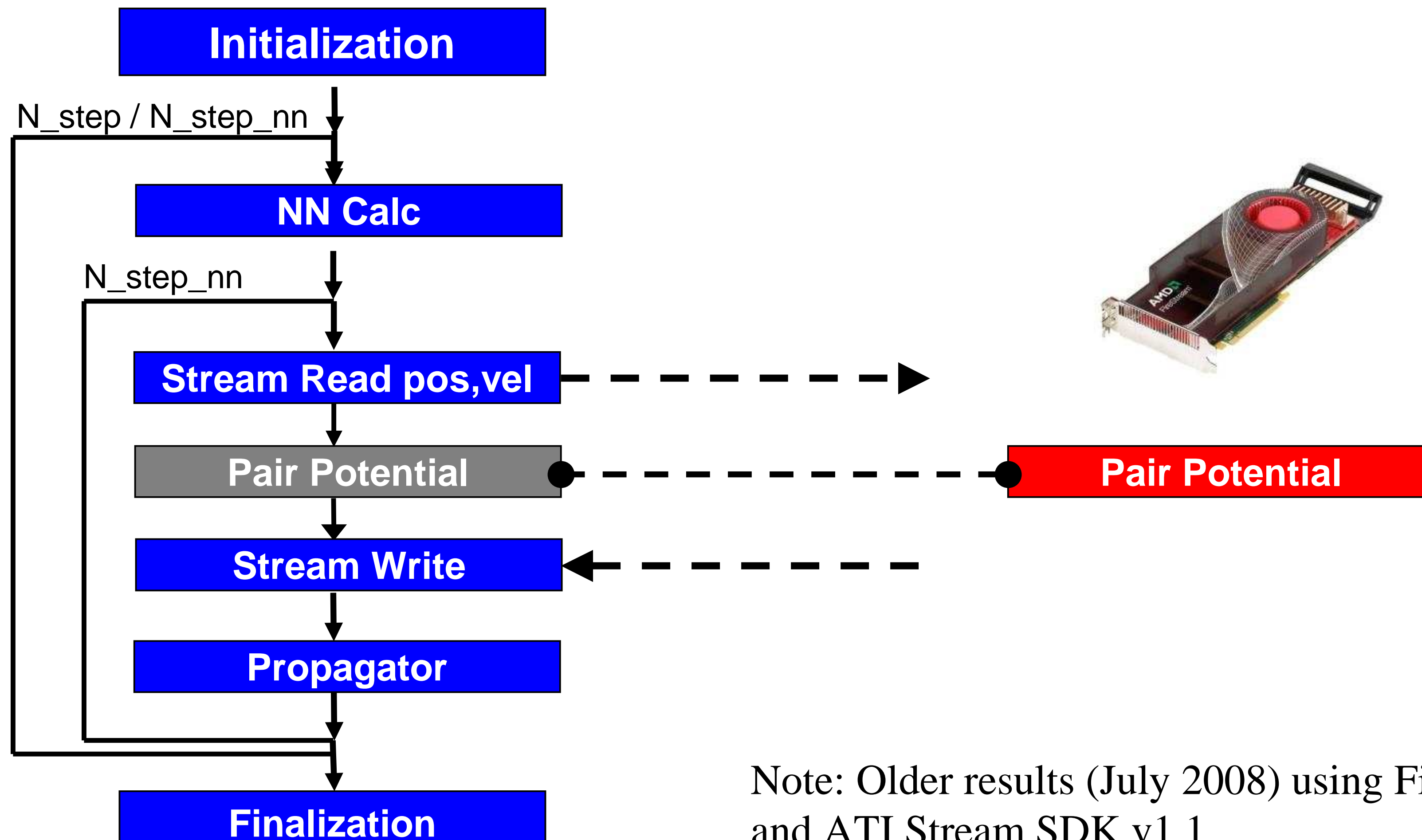
- Rhodopsin Protein Benchmark (most difficult)
- Details: All-atom rhodopsin protein in solvated lipid bilayer with CHARMM force field, long-range Coulomb via PPPM, SHAKE constraints, system contains counter-ions and a reduced amount of water
- Benchmark: 32,000 atoms for 100 timesteps



Copyright © 2009 Brown Deer Technology, LLC. All Rights Reserved.

Molecular Dynamics: LAMMPS

GPU Acceleration



Note: Older results (July 2008) using FireStream 9170 and ATI Stream SDK v1.1

Copyright © 2009 Brown Deer Technology, LLC. All Rights Reserved.

Molecular Dynamics: LAMMPS

Implementation Details

- Only pair potential calculation moved to GPGPU (~> 80% run time on CPU)
- Specifically: PairLJCharmmCoulLong::compute()
- Basic algorithm: "foreach atom-i calculate force from atom-j"
- Atom-i accessed in-order, atom-j accessed out-of-order
- Pairs defined by pre-calculated nearest-neighbor list (updated periodically)
- CPU efficiency achieved by using "half list" such that j > i
  - Eliminates redundant force calculations
- Cannot be done with GPU/Brook+ due to out-of-order writeback
- Must use "full list" on GPU (~ 2x penalty)
- LAMMPS neighbor list calculation modified to generate "full list"

Copyright © 2009 Brown Deer Technology, LLC. All Rights Reserved.

Molecular Dynamics: LAMMPS

Implementation (More) Details

- **Host-side details:**
  - Pair potential compute function intercepted with call to special GPGPU function
  - Nearest-neighbor list re-packed and sent to board (only if new)
  - Position/charge/type arrays repacked into GPGPU format and sent to board
  - Per-particle kernel called
  - Force array read back and unpacked into LAMMPS format
  - Energies and virial accumulated on CPU (reduce kernel slower than CPU)
- **GPU per-atom kernel details:**
  - Used 2D arrays accept for neighbor list
  - Neighbor list used large 1D buffer(s) (no gain from use of 2D array)
  - Neighbor list padded modulo 8 (per-atom) to allow concurrent force updates
  - Calculated 4 force contributions per loop (no gain from 8)
  - Neighbor list larger than max stream (float4 <4194304>), broken up into 8 lists
  - Force update performed using 8 successive kernel invocations

Copyright © 2009 Brown Deer Technology, LLC. All Rights Reserved.

Molecular Dynamics: LAMMPS

Benchmark Tests

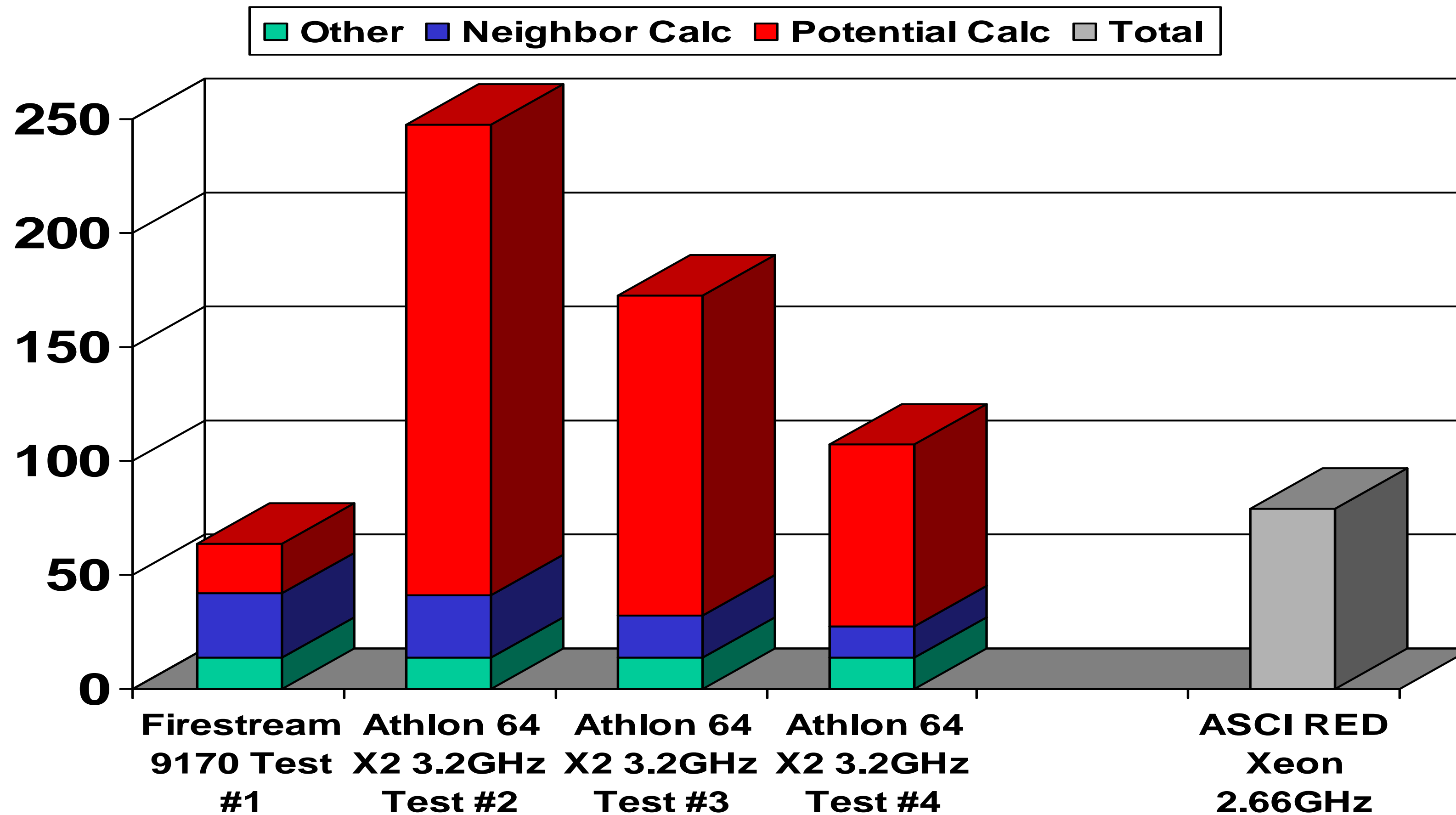
- General:
  - Single-core performance benchmarks
  - GPGPU implementation single-precision
  - 32,000 atoms, 100 timesteps (standard LAMMPS benchmark)
- Test #1: GPGPU
  - Pair Potential calc on GPGPU, full neighbor list, newton=off, no Coulomb table
- Test #2: CPU ("identical" algorithm, identical model)
  - Pair Potential calc on CPU, full neighbor list, newton=off, no Coulomb table
- Test #3: CPU (optimized algorithm, identical model)
  - Pair Potential calc on CPU, half neighbor list, newton=off, no Coulomb table
- Test #4: CPU (optimized algorithm, optimized model)
  - Pair Potential calc on CPU, half neighbor list, newton=on, Coulomb table
- ASCI RED single-core performance (from LAMMPS website)
  - Most likely a Test #4, included here for reference

Direct comparison (THEORY)
Architecture Optimized (REALITY)

Copyright © 2009 Brown Deer Technology, LLC. All Rights Reserved.

Molecular Dynamics: LAMMPS

Rhodopsin Benchmark

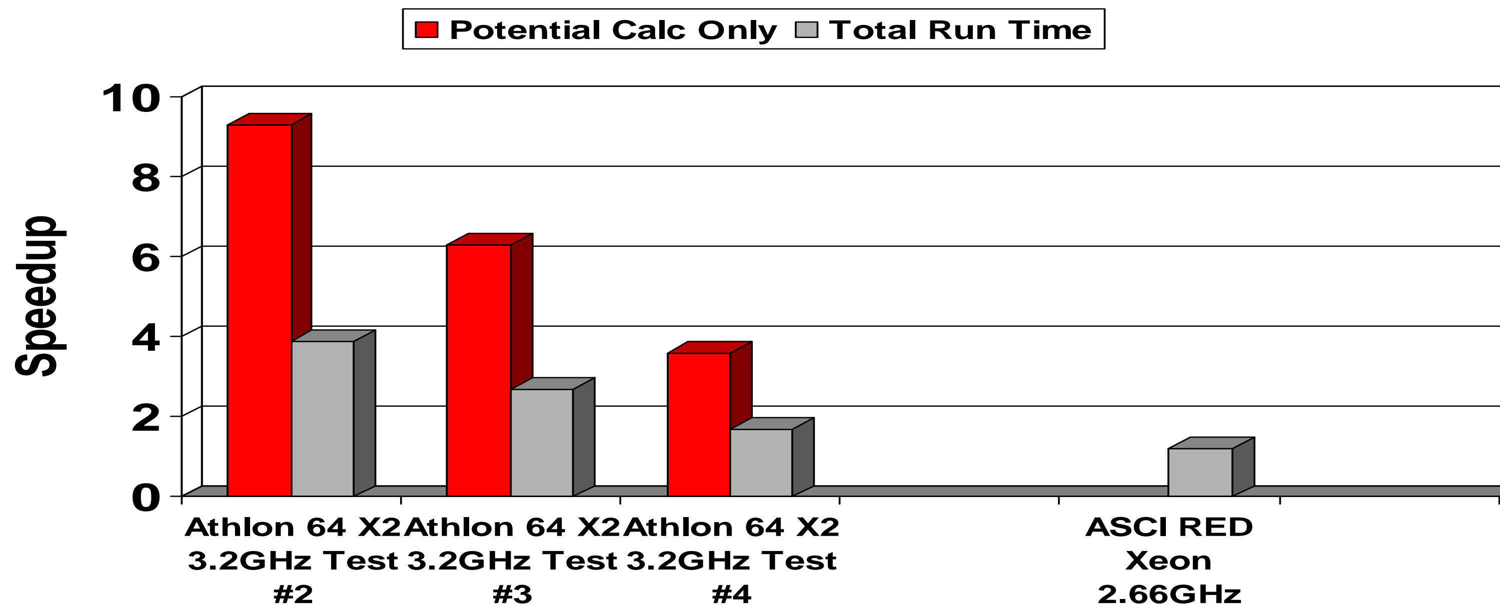


Amadahl's Law: Pair Potential compared with total time: 35%(Test#1), 75%(Test#2), 83%(Test#4)

Copyright © 2009 Brown Deer Technology, LLC. All Rights Reserved.

Molecular Dynamics: LAMMPS

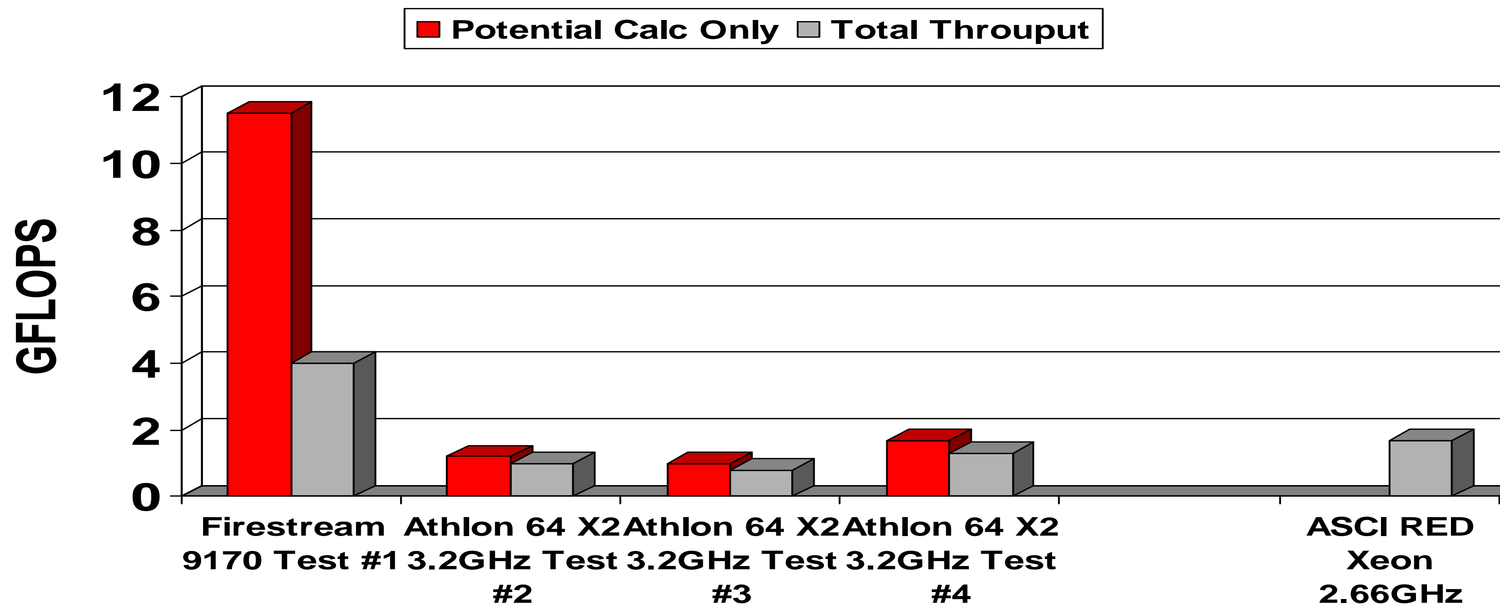
Rhodopsin Benchmark



Copyright © 2009 Brown Deer Technology, LLC. All Rights Reserved.

Molecular Dynamics: LAMMPS

Rhodopsin Benchmark



Copyright © 2009 Brown Deer Technology, LLC. All Rights Reserved.

Conclusions

- GPUs provide tremendous raw floating-point performance
  - No longer limited to single-precision
- Compiler technology remains immature, however, it is relatively easy to accelerate real algorithms
  - The days of trying to re-factor physics equations into OpenGL are over
- The GPU technology is advancing (in terms of price/performance, performance/power) very rapidly
- The commercial market that really drives this technology (video games) will not slow down
- This presents a very useful, fortunate situation for scientists and engineers who wish to exploit this technology

Contact: drichie@browndeertechnology.com

Copyright © 2009 Brown Deer Technology, LLC. All Rights Reserved.