Computational Challenges in the Use of Emerging Many-Core Architectures for DoD Applications

David Richie

Brown Deer Technology

August 17th, 2009

Outline

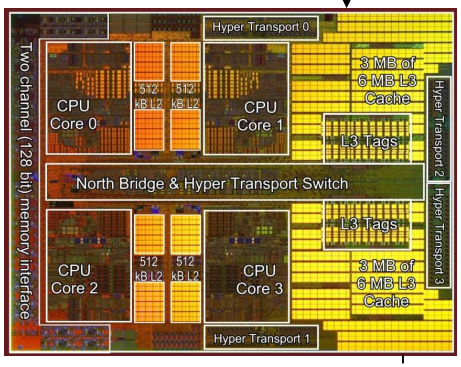
- **Many-core processors**
- Challenges
- Motivation
  - Obvious benefit: performance
  - Not so obvious benefit: mobile HPC
- OpenCL: problem solved, more problems
- Future Developments

~~Future~~ Parallelism of HPC Architectures

AMD64 Linux Cluster

- Multi-node ~ 1,000 nodes
- Distributed - MPI

- Challenges in productivity
- Very high level parallel languages



AMD Opteron (Shanghai)

- Multi-core ~ 10 cores
- SMP - OpenMP

- Moderate challenges
- Not so different from SMP nodes

AMD Radeon HD 4870X2
(2.4 TFLOPS single-precision)

- Many-core ~ 1,000 cores
- Stream, SIMD, SIMT - OpenCL

- New level of parallelism
- Significant Challenges

Many-core: Massive Chip-Level Parallelism

- DoD DSRC major-center-scale Linux cluster
  - Scheduled operation through 2011
  - 4,400 cores, 26.4 TFLOPS (double precision)
  - Cost: multi-million dollar acquisition

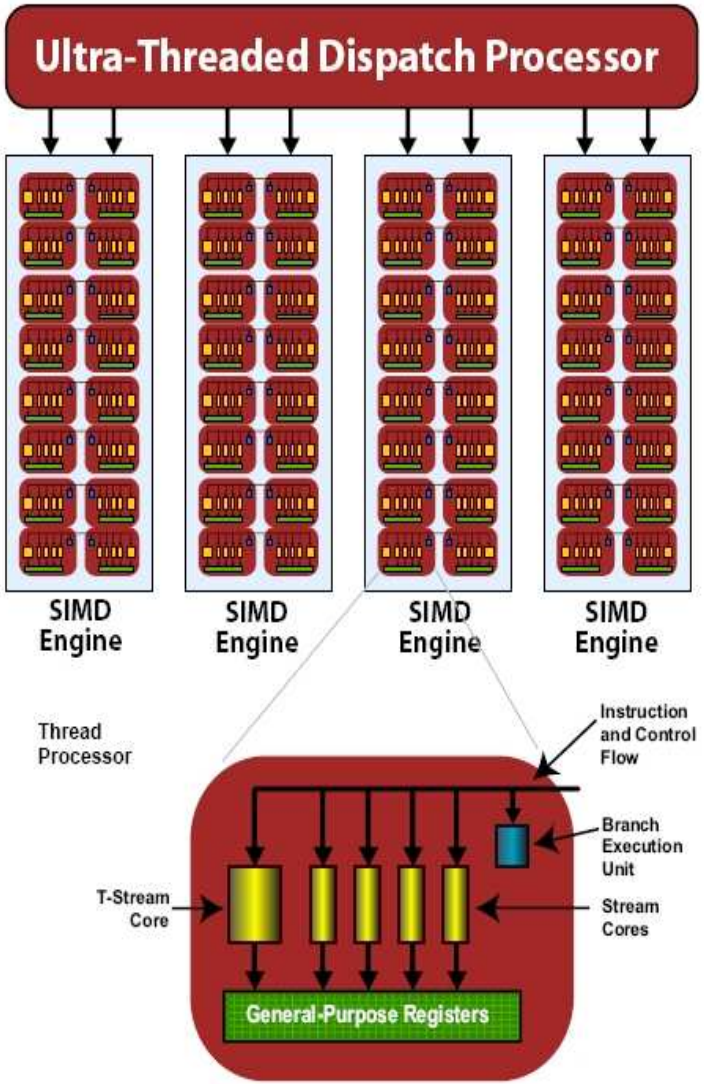
Linux Cluster

- GP-GPU Workstation "Supercomputer" (paper spec)
  - Can be built today w/existing COTS parts (for gamers)
  - 11,200 cores, 16-20 TFLOPS (single-precision)
  - cost: < $10,000 (plus effort and ingenuity)

GP-GPU Workstation

- Many-core processors can provide as many cores per compute node as there are computer nodes
- Represents a complete inversion of the HPC paradigm familiar to HPC software developers

Many-core (GPU) Architectures



- For years HPC asked for a chip with "lots of FP units"
  - Who needs register renaming, out-of-order execution? ...
- Here they are – 800 FP units (example shown)
- Read the fine print:
  - Most of the complexity of a "core" has been removed
  - Highly constrained execution model
  - Limited number of registers
  - Constrained memory architecture
  - Thread aggregation (SIMT model)
- Question: how dependent have HPC software developers become on the capabilities of a modern core architecture, e.g., Nehalem or Istanbul?
- ... port your code to a GPU and find out

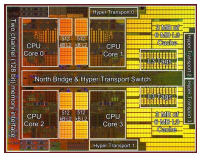
Many-core vs. Multi-core

- Many-core is not the logical evolution of multi-core
  - Issue is not number of cores, rather the cores themselves
- Distinction is between ‘heavy-weight” cores and “light-weight” cores
- Will likely generate debate similar to RISC vs. CISC
  - Better to have 32 capable cores, or 1600 weak cores?
- Distinction is invariant, silicon has finite dimensions
- What about “medium-weight” cores (not-so-many-core)?
  - These will be thrown into the debate also

Many-core Evolution

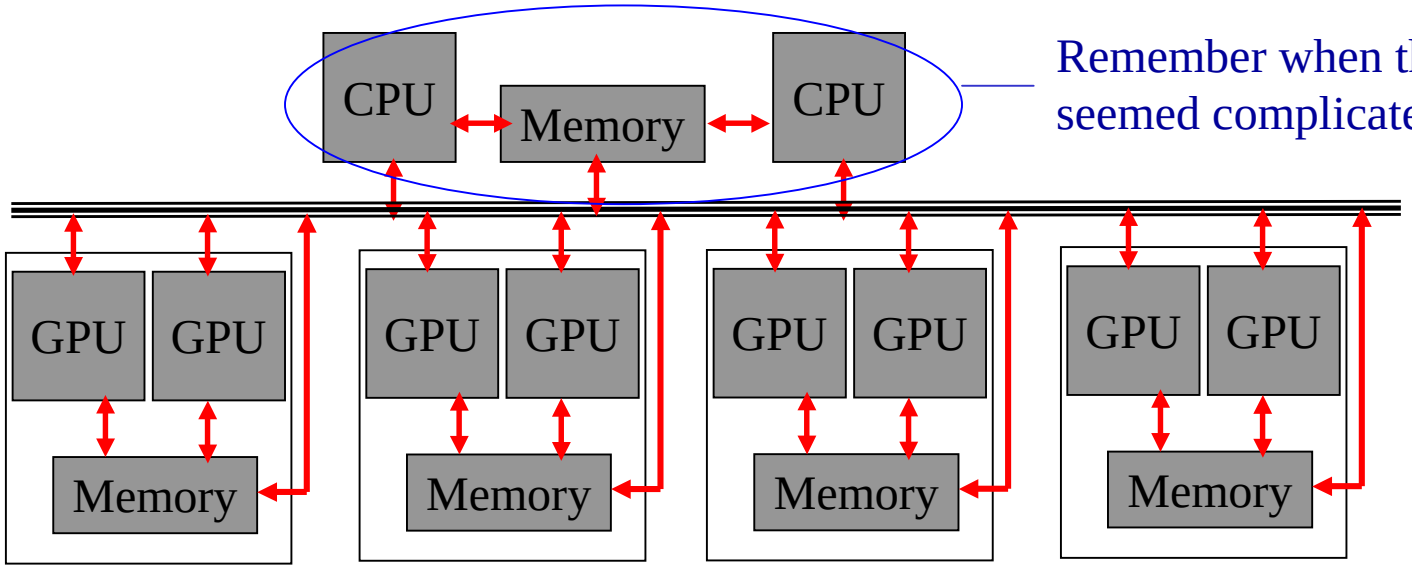
- GPU (ancient times)
  - Non-IEEE compliant FP units
  - OpenGL, DirectX, Shader languages, ...
- GP-GPU (now)
  - IEEE compliant FP units (sort of)
  - RV790 (FireStream), GT200 (Tesla)
  - CUDA, Brook
- Many-core (drop “graphics”, improve legitimacy) (near future)
  - RV870 (Evergreen), GT300, Larrabee(manycore or multi-core+vector?)
  - OpenCL(?)
- Is HPC driving the evolution? Of course not, HPC is a post-roadmap add-on
  - Consumer market is driving the technology
    - “Data parallel” closest driver related to HPC
  - Understanding this provides a guide for what to (not) expect
  - HPC community successfully exploited x86_64, same deal

The Many-core + Multi-core Problem



8 cores

6,400 cores



- Co-processors are back, along with the unsolved problems, and entirely new problems
- Data and control must be orchestrated between distributed resources – cores + memory
- Problem differs significantly from recent distributed HPC challenges
  - Very serious latency and bandwidth constraints
  - Problems: locking, memory consistency, asynchronous operations, concurrency
    - Doesn't the operating system take care of this? ... No, not anymore – see the OpenCL spec

Challenges

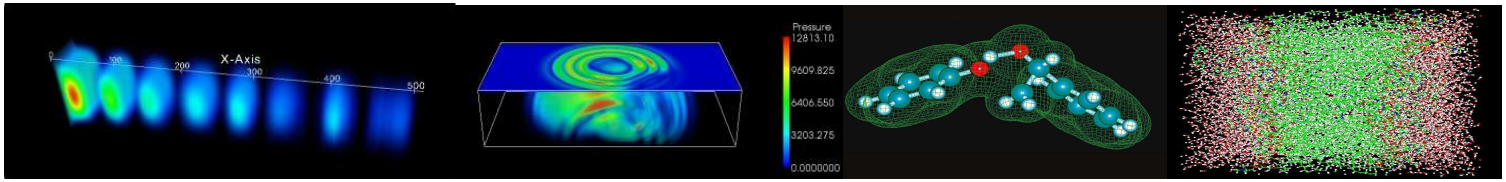
- Software is the greatest challenge – always is
- Theoretical challenge – the programming model
  - Many-core offers worse case of many long-standing problems
    - Co-processors, distributed shared memory, thread synchronization, ...
  - Many-core adds third tier to parallelism requiring new API
  - What should/will the SDK look like? Automation or expression?
  - A programming model is a contract with the programmer
    - What are the likely terms for many-core? agreeable?
- Practical challenges
  - Quality of compilers and vendor-provided run-time
  - Code portability, compliance, new compilation models
  - Software developers will find the “many cores” primitive

(Outline)

- Many-core processors
- Challenges: software (anyone surprised)
- **Motivation**
  - **Obvious benefit: performance**
  - Not so obvious benefit: mobile HPC
- OpenCL: problem solved, more problems
- Future Developments

Investigation of 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



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**

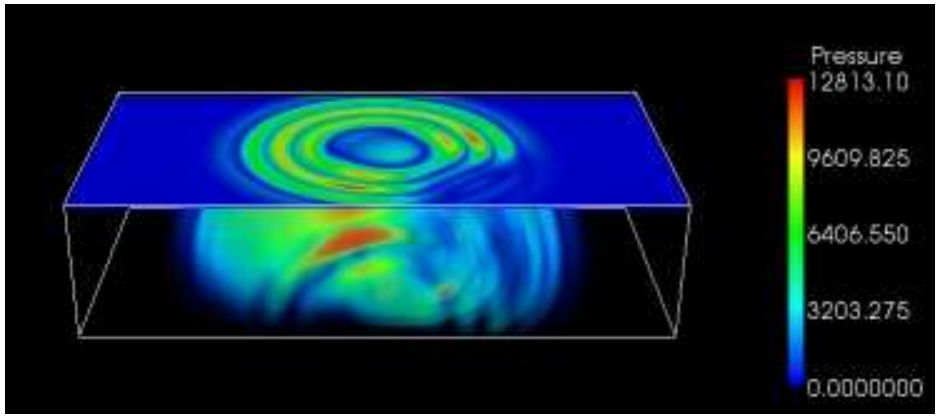
- **Commercial applications for oil and gas exploration**

- **Military applications for detecting buried structures**

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

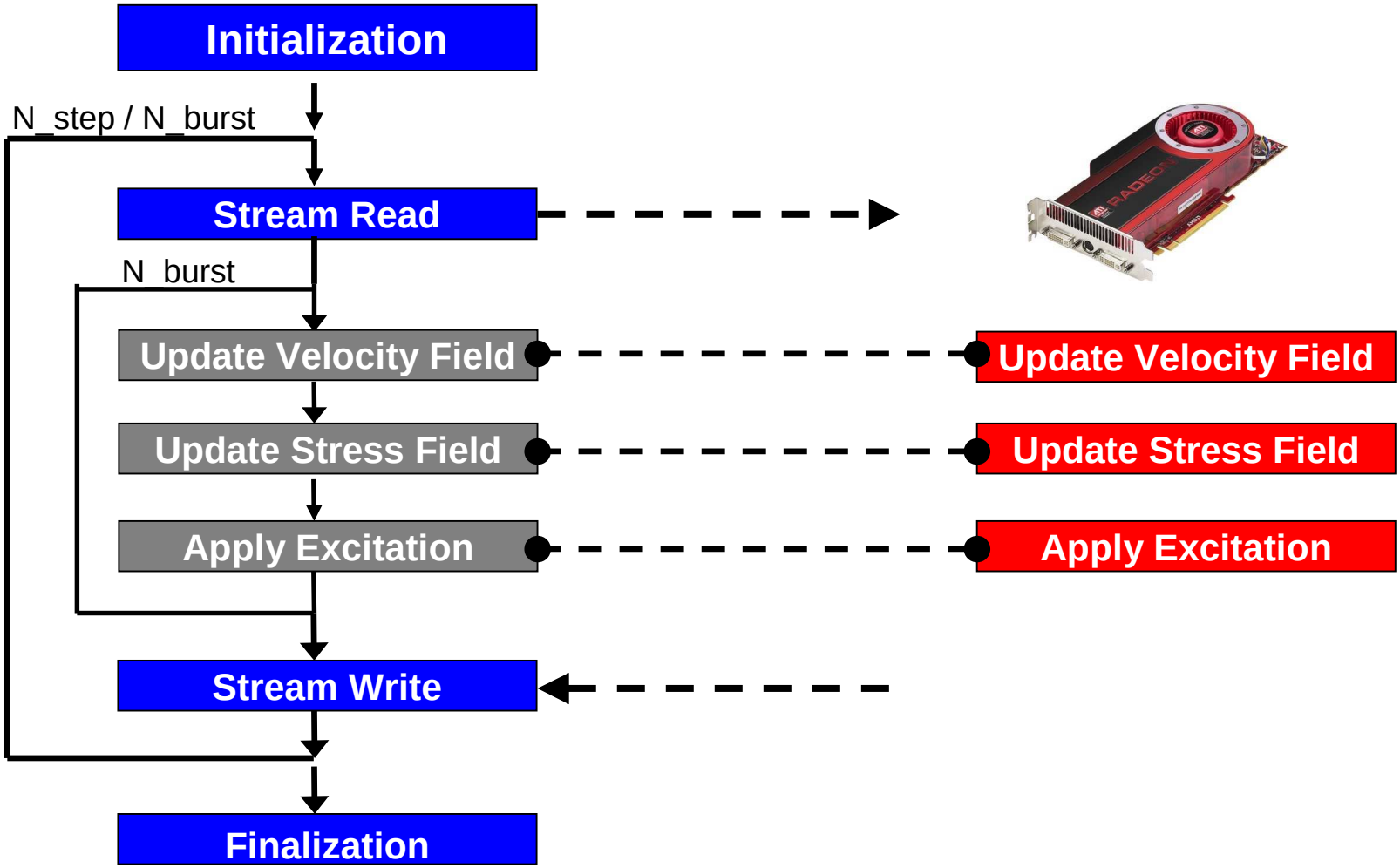
$$\frac{\partial \sigma_{ii}}{\partial t} = (\lambda + 2\mu)\frac{\partial v_i}{\partial x_i} + \lambda\left(\frac{\partial v_j}{\partial x_j} + \frac{\partial v_k}{\partial x_k}\right)$$

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



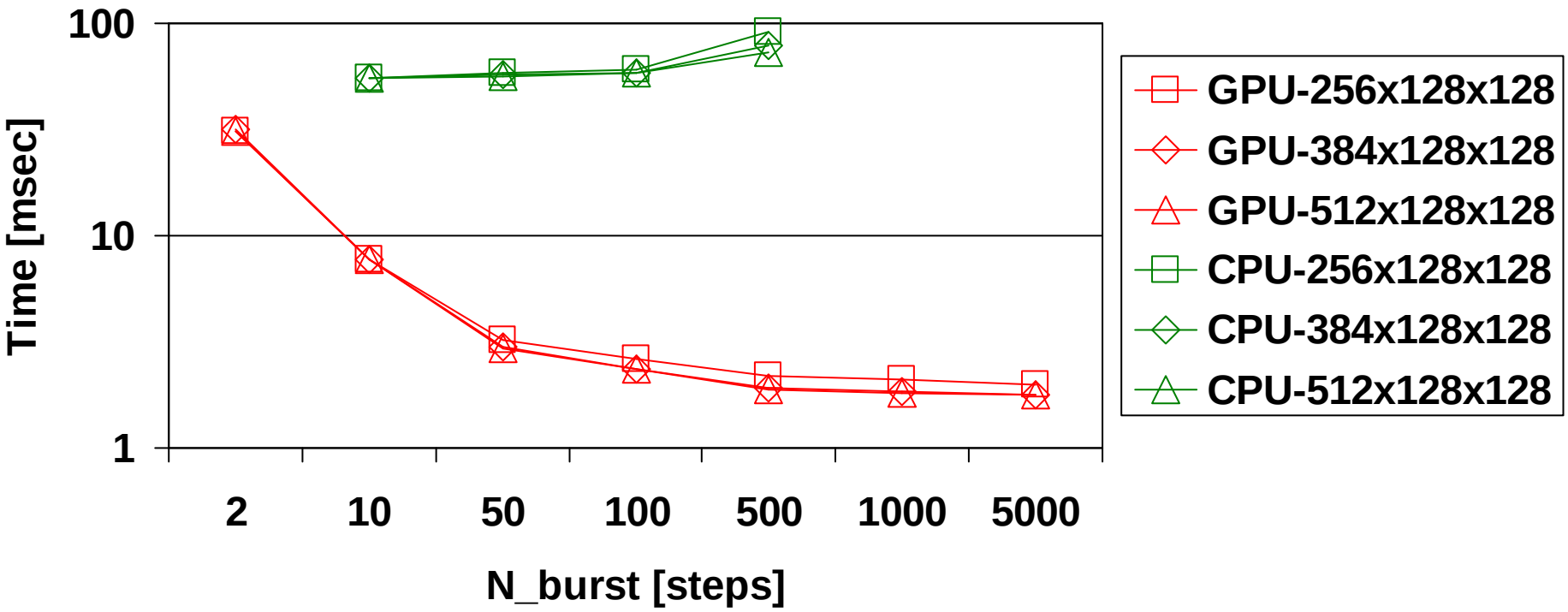
Seismic: 3D VS-FDTD

GPU Acceleration



Seismic: 3D VS-FDTD: Benchmarks

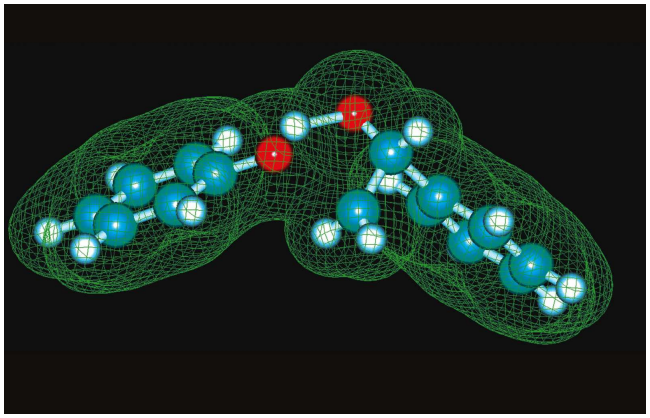
GPU vs. CPU: Time per Million Points



- Performing many iterations in between data transfer mitigates PCIe bottleneck
- 31x speedup for largest grid

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



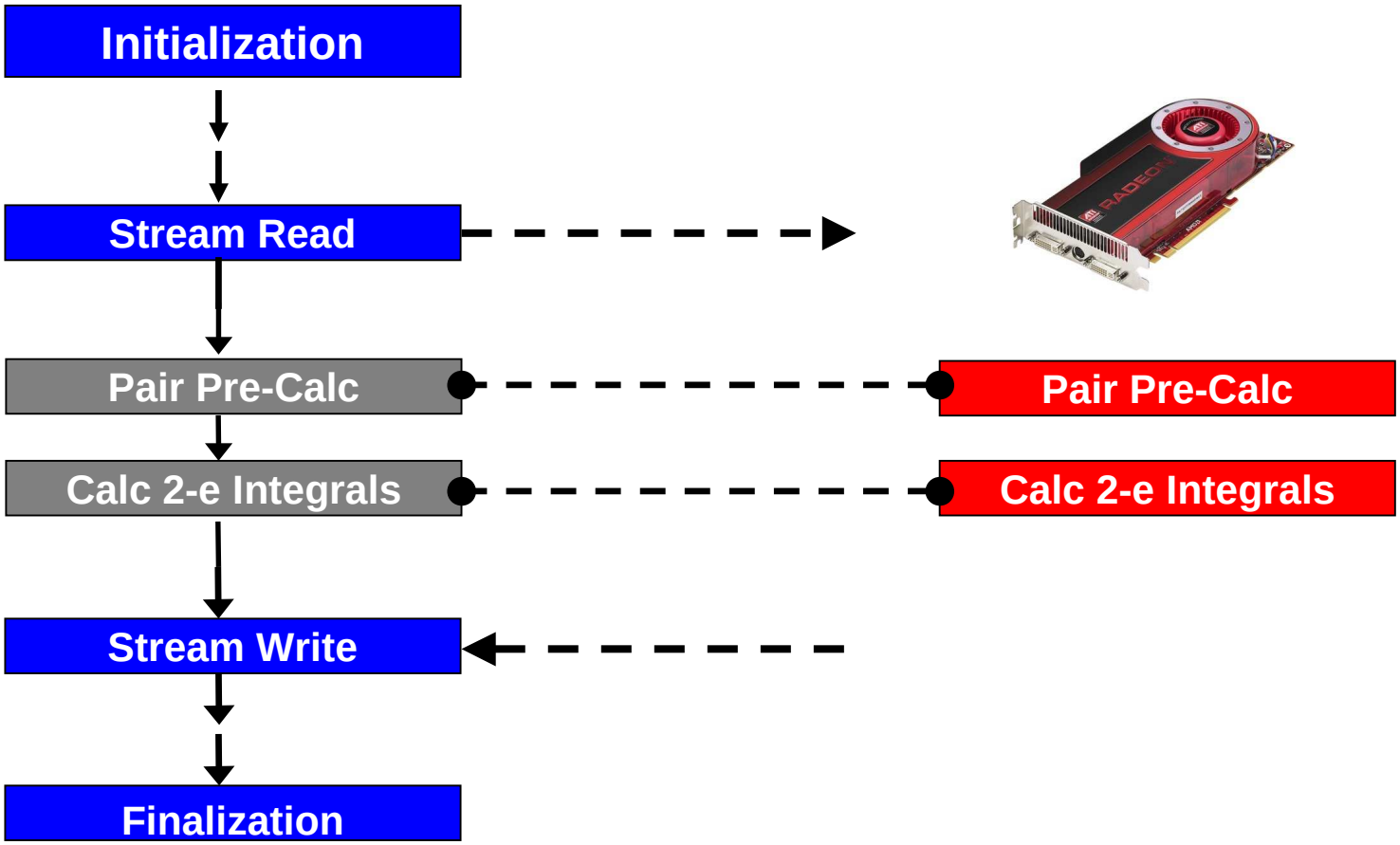
$$(\mu\nu|\lambda\tau) = \iint dr_1\, dr_2\, \phi_\mu(r_1)\, \phi_\nu(r_2) \frac{1}{r_{12}} \phi_\lambda(r_1)\, \phi_\tau(r_2)$$

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

- 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()

Quantum Chemistry: Two-Electron Integrals

GPU Acceleration



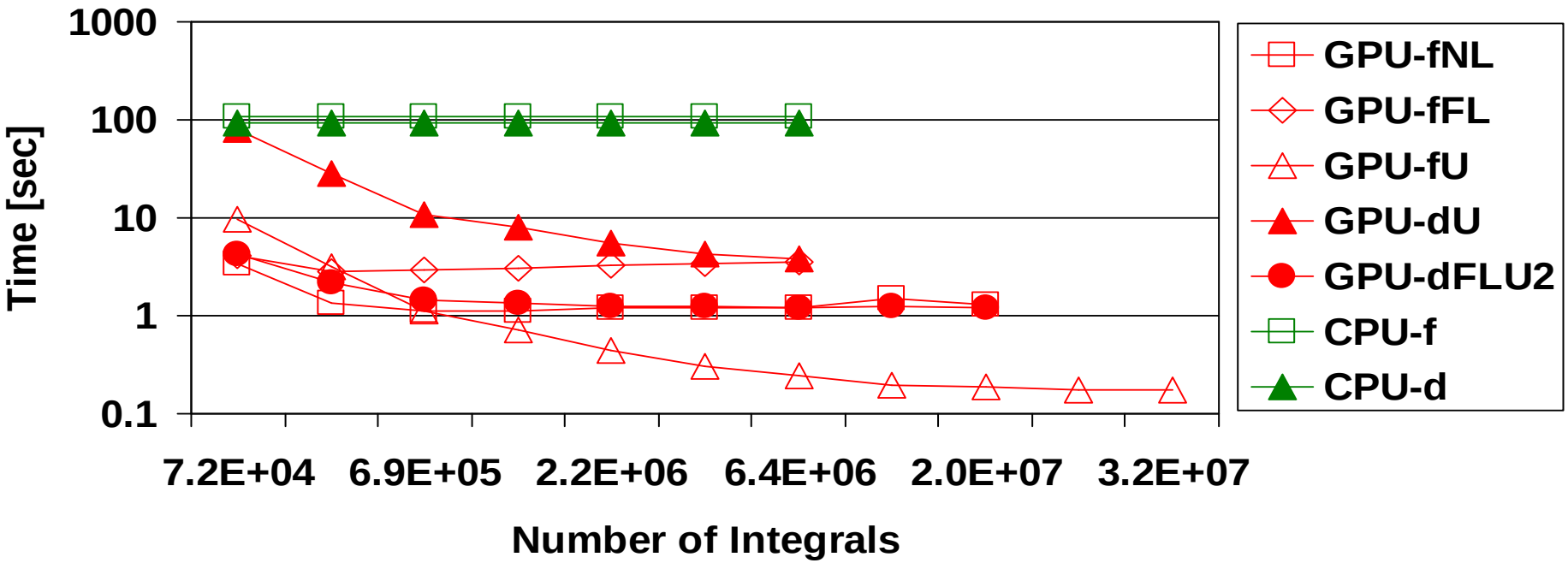
Quantum Chemistry: Two-Electron Integrals

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

Quantum Chemistry: Two-Electron Integrals

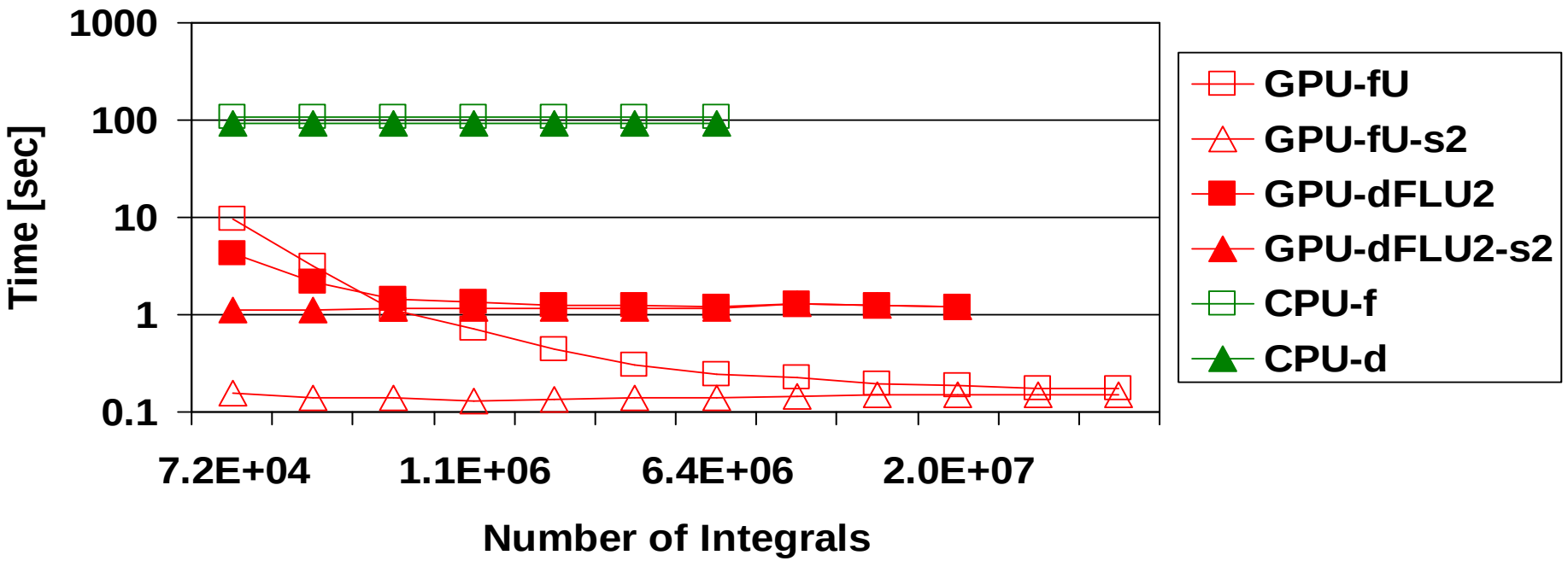
GPU vs. CPU: Time per Million 2-e Integrals



- 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

Quantum Chemistry: Two-Electron Integrals

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

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		

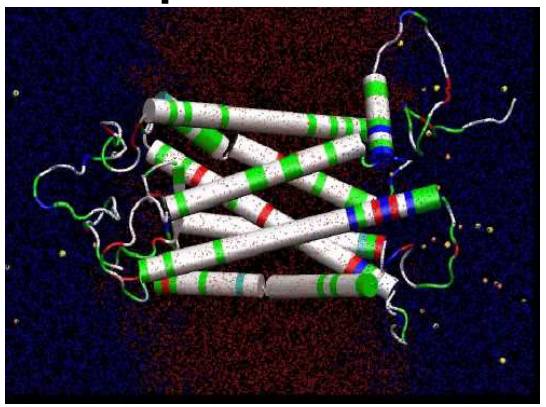
*Ufimtsev and Martinez

- Large number of integral limit (~10 million)
  - SP: 814x speedup
  - DP: 80x speedup
- CPU implementation definitely not optimized
- GPU performance/speedup will nevertheless be substantial

Molecular Dynamics: LAMMPS

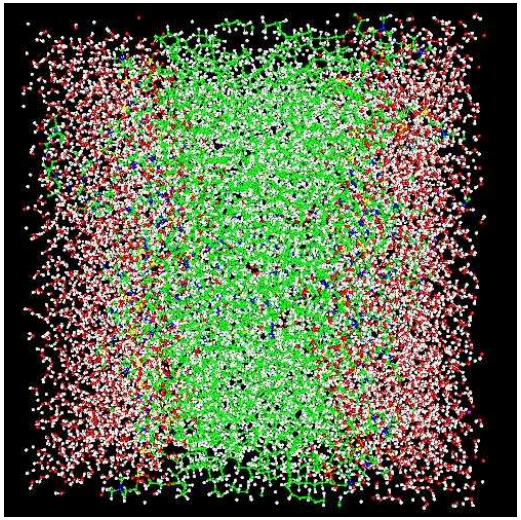
- Fundamental technique for molecular modeling
- Simulate motion of particles subject to inter-particle forces
- LAMMPS is open-source MD code from DOE/Sandia
  - Dr. Steve Plimpton, http://lammps.sandia.gov
- Goal: accelerate inter-particle force calculation

Rhodopsin Protein



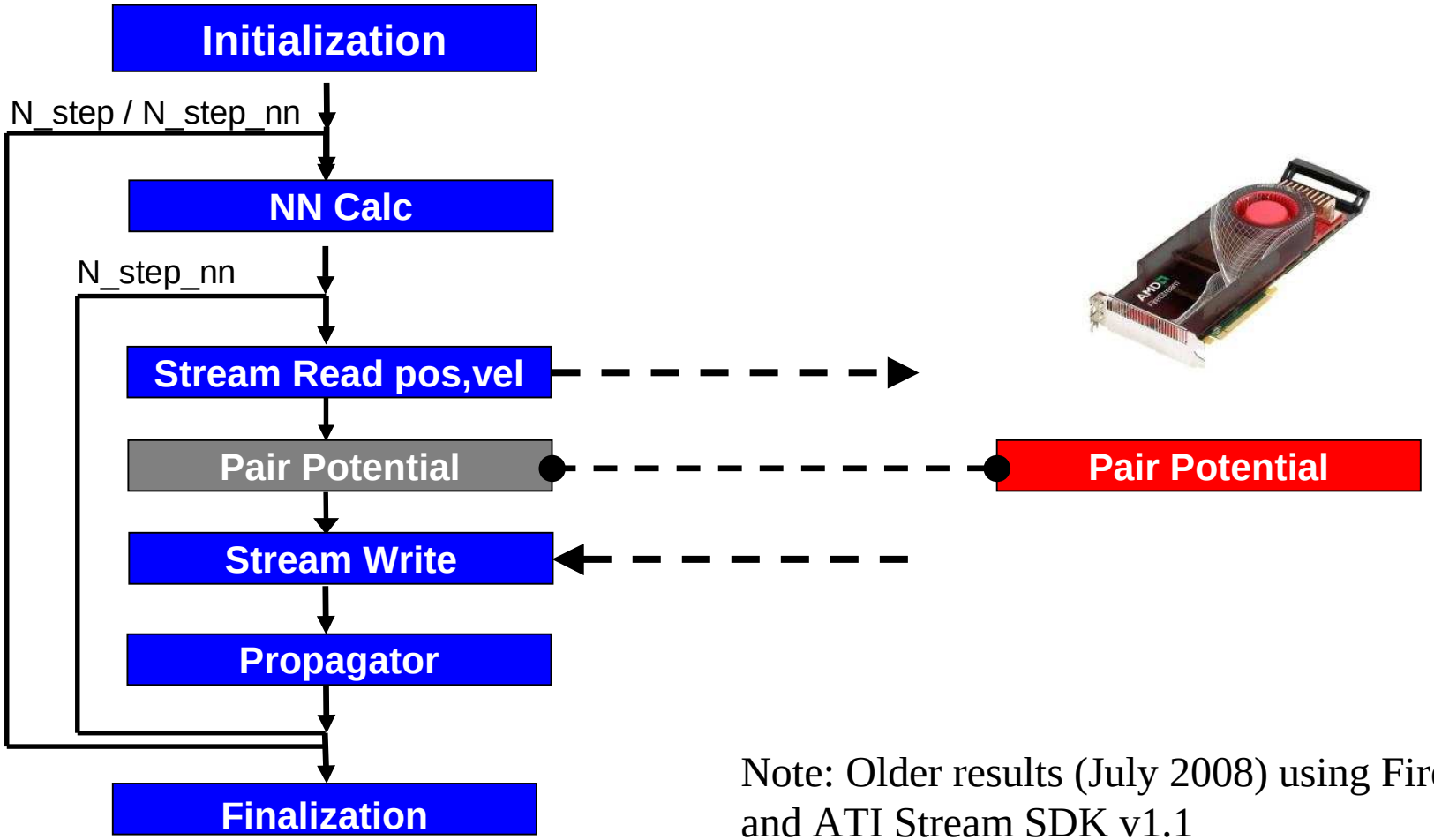
*Original work due to Paul Crozier and Mark Stevens at Sandia National Labs

- 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



Molecular Dynamics: LAMMPS

GPU Acceleration



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

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”

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

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

Direct comparison (THEORY)

- 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

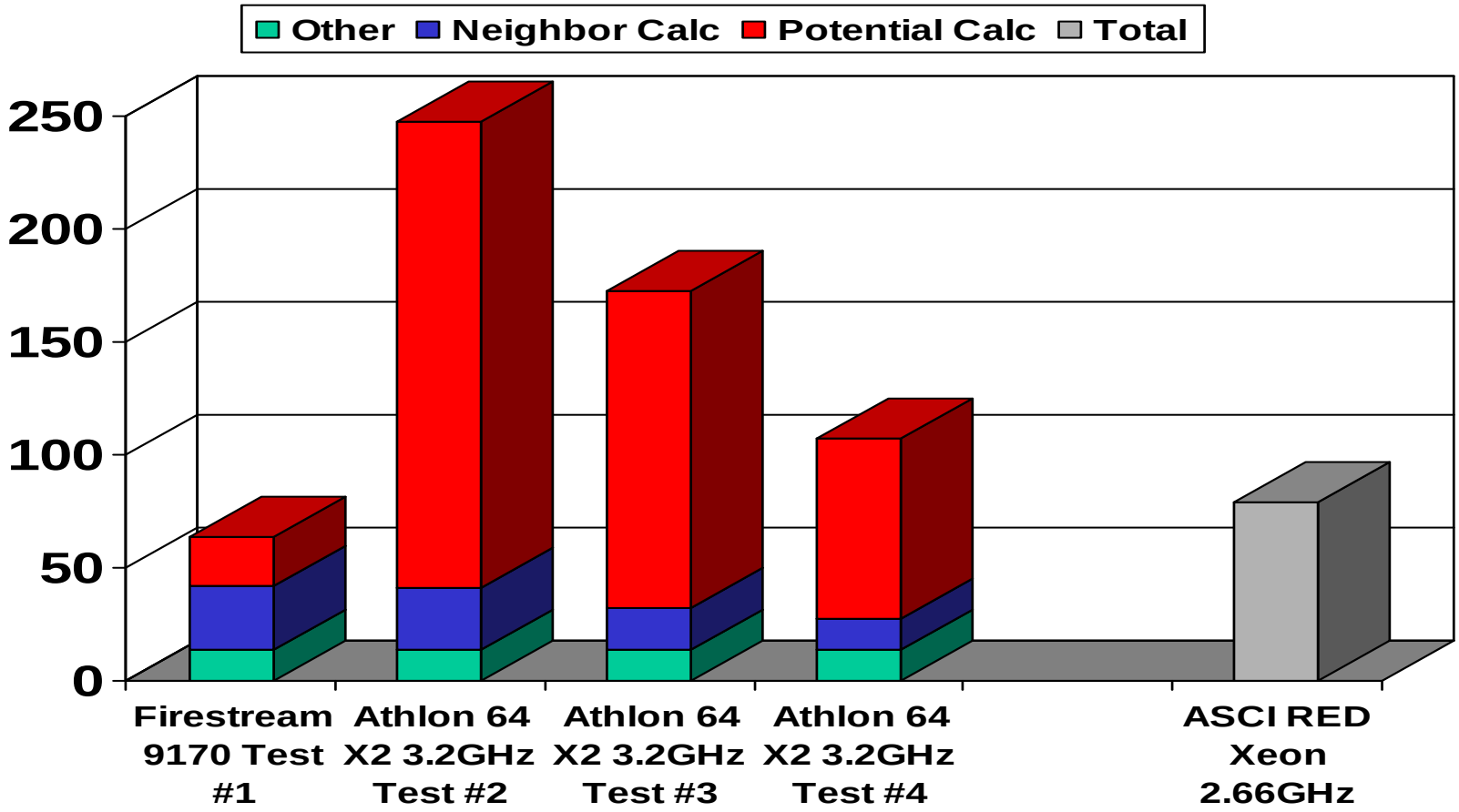
Architecture Optimized (REALITY)

- 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

Molecular Dynamics: LAMMPS

Rhodopsin Benchmark



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

(Outline)

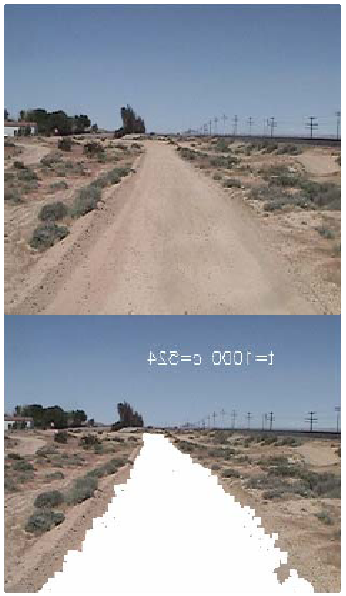
- Many-core processors
- Challenges: software (anyone surprised)
- **Motivation**
  - Obvious benefit: performance
  - **Not so obvious benefit: mobile HPC**
- OpenCL: problem solved, more problems
- Future Developments

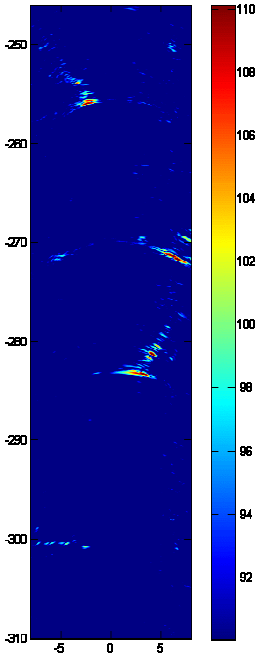
100 TFLOPS, battlefield deployable, by 2012?

- **What can be built today?**
  - COTS solution: 2U+4U - 16 RV770 GPUs - 16 TFLOPS - 2.5 KW
- **Future assumptions**
  - Architecture: assume 3x performance increase
    - RV770 (55nm) - 800 cores - Today
    - RV870 (40nm) – 2000(?) cores – 2009
    - RV970 (32nm?) – 2400(?) cores - 2010
  - Design: assume 2x performance increase
    - Dual-GPU boards available now, dual-slot form factor
    - Dual-GPU boards, single-slot via lower power + liquid cooling
  - Power: assume power constrained 200W/per board(?)
- **Result:**
- **96 TFLOPS - 3.2 KW ~2 cu. ft. (2U+4U) by 2011**
- **What will the software look like?**
  - Programming model? Compilers? Runtime? Portability?
- **Impact of deployable HPC for battlefield applications?**

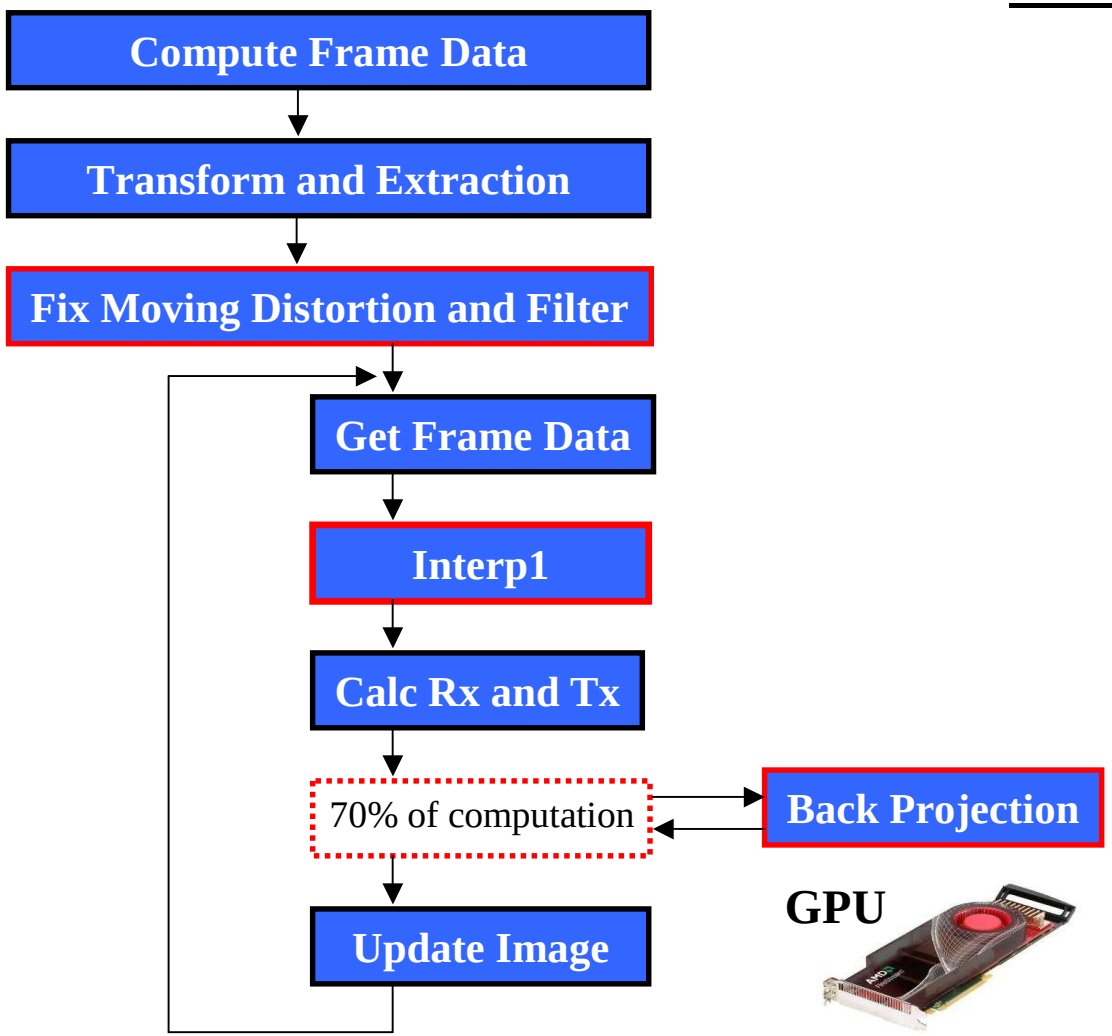
Battlefield Application: UWB SIRE RADAR

- Ultra Wide-Band Synchronous Impulse Reconstruction RADAR
  - Obstacle avoidance and concealed target detection
  - Under development by researchers at ARL/SEDD
  - Algorithms developed in MATLAB, being ported to C and GPUs





GPU Acceleration of SIRE Back Projection



Host code using ATI Stream Brook+ compiler

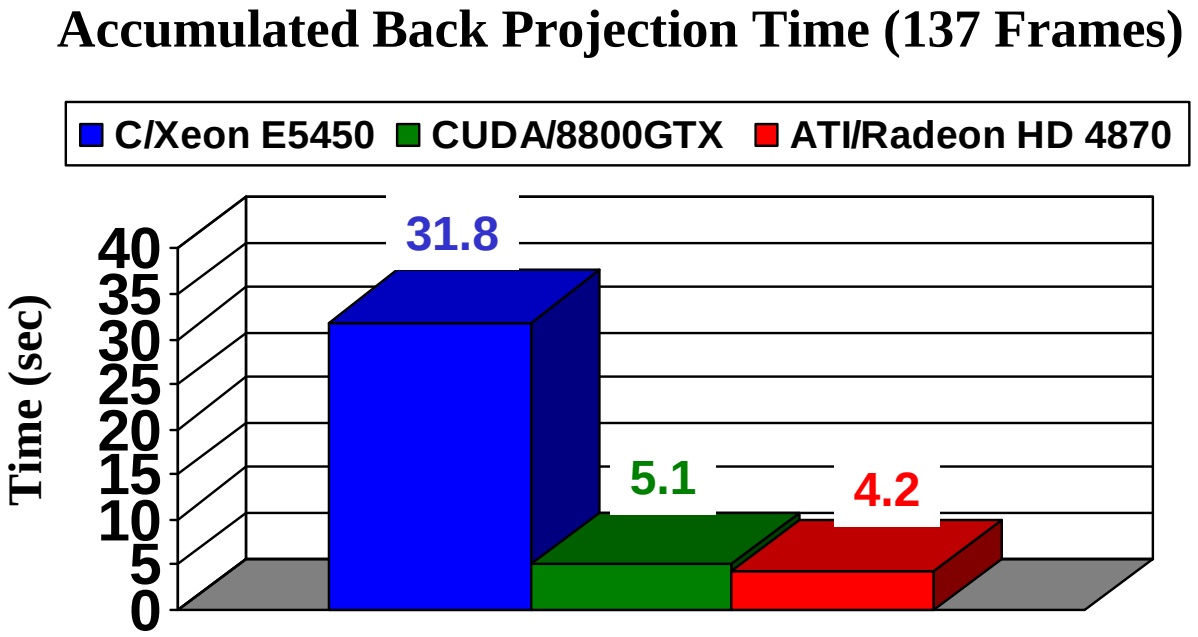
```
...
float s_data<nas>;
float4 s_rx<na>;
float4 s_tx<na>;
float4 s_img<100,64>;

streamRead(s_data,data_all);
streamRead(s_rx,rx4);
streamRead(s_tx,tx4);

backprojection_gpu_kern(
        (float)na, (float)ns
        (float)nrange2,
        (float)nxrange2,
        yref, xr_inc, r_inc,
        r_start, rdr,
        coef1,coef2,coef3,
        s_rx, s_tx,
        s_data,s_img
);

streamWrite(s_img,img);
...
```

UWB SIRE RADAR Initial Benchmarks



- **CPU baseline uses a single-core – opportunity for SSE and OpenMP optimization**
- **GPU implementations have opportunity for optimization as well**
- **Impact on real-time capability**
  - C/Xeon E5450: total time 45.5sec ⇒ **13 mph**
  - ATI/Radeon HD 4870: total time ⇒ **34 mph**
- **Amdahl's Law appears: relative cost of Back Projection 70% → 23%**
  - Need to examine other parts of the overall algorithm

(Outline)

- Many-core processors
- Challenges: software (anyone surprised)
- Motivation
  - Obvious benefit: performance
  - Not so obvious benefit: mobile HPC
- **OpenCL: problem solved, more problems**
- Future Developments

OpenCL – What It Is, What It Is Not

- Industry standard for parallel programming of heterogeneous computing platforms
- Substance: OpenCL = CAL + CUDA + Brook + OpenGL buffer sharing
- Two parts:

Platform and runtime API

- Operating system moved into user-space
- Good news, programmer has control over
  - Device discovery, registration, setup
  - Creating work queues
  - Memory consistency
- Bad news, programmer has responsibility for ...

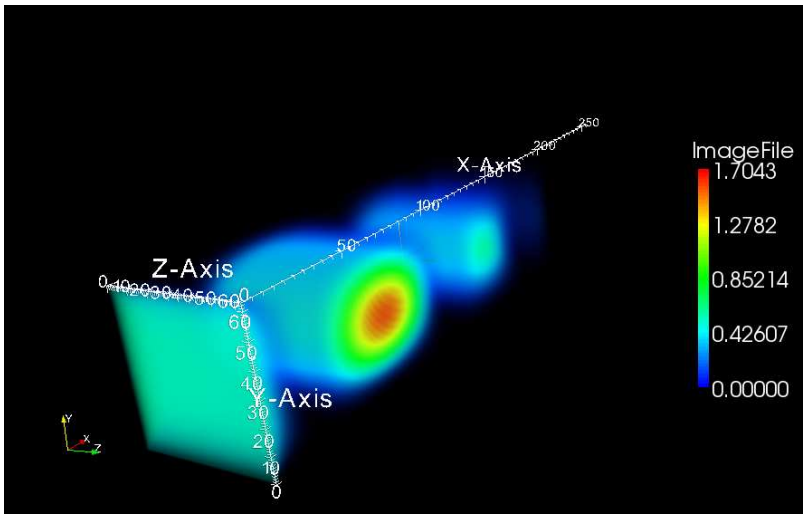
Programming language

- C extensions for device programming
- Execution context is a kernel
- Familiar with Brook/CUDA, no surprises

- OpenCL is NOT designed to make programming GPUs easier
- OpenCL is a very low-level standard designed to support platform independent software stack

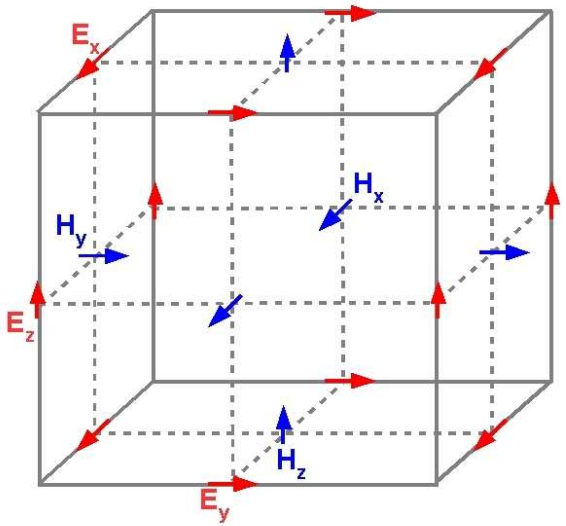
Electromagnetics: 3D FDTD

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



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

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



- Implemented using AMD OpenCL CPU Beta
  - OpenCL implementation submitted for certification

OpenCL By Example (1)

```
#include <CL/cl.h>

ctx = clCreateContextFromType(...);
clGetDeviceInfo(...);
cmdq = clCreateCommandQueue(...);

ee_buf = clCreateBuffer(...);
hh_buf = clCreateBuffer(...);

prg = clCreateProgramWithSource(...);
clBuildProgram(...);
cl_kernel hcomp_krn = clCreateKernel(...);
cl_kernel ecomp_krn = clCreateKernel(...);
```

- Device discovery, registration
- Create work queues
- Create buffers for data transfer
- Run-time/just-in-time (JIT) compilation

OpenCL By Example (2)

```
for(step = 0; step < nstep; step += nburst ) {

  clSetKernelArg(hcomp_krn,1,...);
  clSetKernelArg(hcomp_krn,2,...);
  ...

  for(burst = 0; burst < nburst; burst++ ) {
    clEnqueueNDRangeKernel(cmdq,hcomp_krn, ..., &kev[2*burst]);
    clEnqueueNDRangeKernel(cmdq,ecomp_krn, ...,&kev[2*burst+1]);
  }

  for(i=0;i<2*nburst;i++) clWaitForEvents(1,&kev[i]);

  clEnqueueReadBuffer(cmdq,ee_buf, ...);
  clEnqueueReadBuffer(cmdq,hh_buf, ...);

  clWaitForEvents(1,&ev[2]);
  clWaitForEvents(1,&ev[3]);
}
```

"Push" arguments

Scheduler

Wait

Setup DMA

Wait

OpenCL By Example (3)

```
__kernel void ecomp_kern(
   float t, float omega,
   uint nx, uint ny, uint nz, uint nb1, uint nt,
   float ax, float ay, float az,
   __global float* ee, __global float* hh,
   __local float* eblock, __local float* hblock
)
{
   uint gi000 = 4*nb*get_global_id(0);
   uint gj000 = 4*nb*get_global_id(1);
   uint gk000 = 4*nb*get_global_id(2);

   for(ijk=0;ijk<nb3;ijk++) {
      eblock[ci+EX] = ee[gci+EX];
      ...
   }

   barrier(CLK_LOCAL_MEM_FENCE);

   for(ijk=0;ijk<nb3;ijk++) {
      float ex000 = eblock[ci000+EX];
      float hy000 = hblock[ci000+HY];
      float hy001 = hblock[ci001+HY];
      ...
      ex000 += az * (hy001 - hy000) + ...
      float q = sin(t*omega);
      ez000 = (gi == 4)? q : ez000;
      ee[gci+EX] = ex000;
      ...
   }
}
```

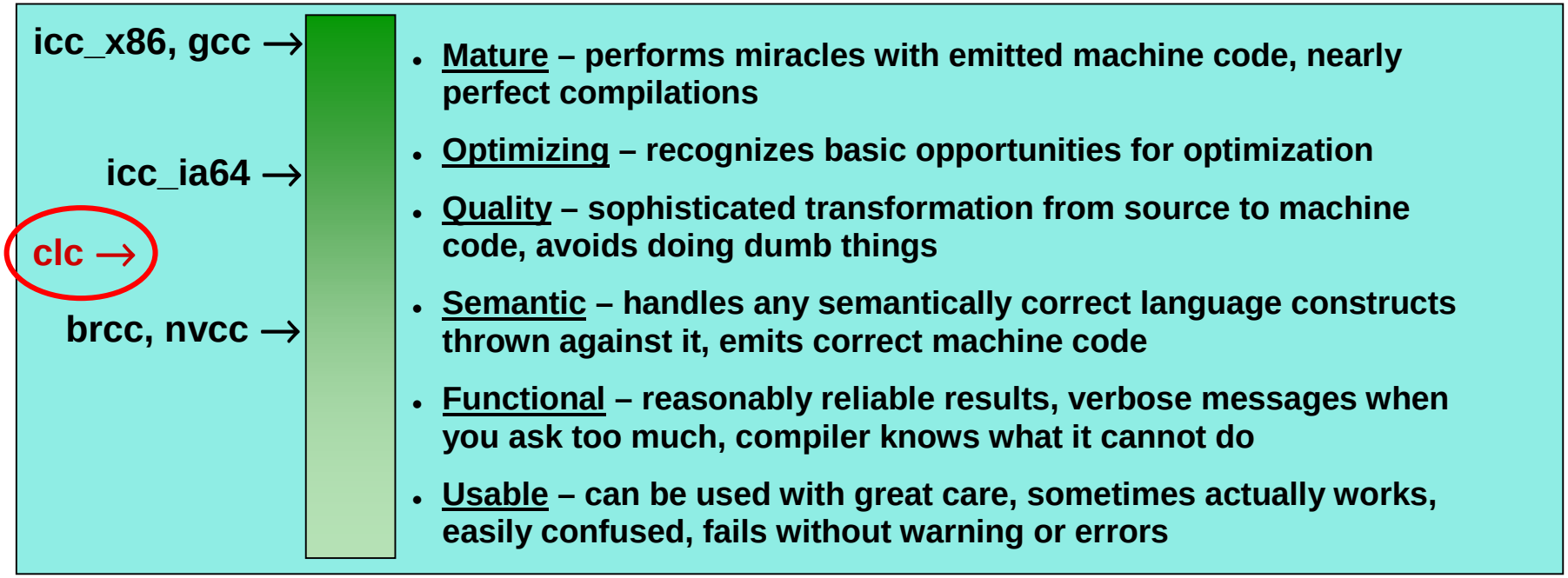
Get thread index

Memory consistency

(Outline)

- Many-core processors
- Challenges: software (anyone surprised)
- Motivation
  - Obvious benefit: performance
  - Not so obvious benefit: mobile HPC
- OpenCL: problem solved, more problems
- **Future Developments**

State of Compilers



- Many-core compiler technology well behind industry-standard CPUs (x86_64)
  - Trails lesser CPU compilers, e.g., Itanium
- Reality: requires (at least) decade to build up complexity found in x86_64 compilers
- Both vendors (Nvidia and ATI) offer relatively immature compilers
  - (I have personally broken both of them, source-level tests indicate they're not optimizing)
- Hardware makes effort worthwhile, no more difficult than SSE/OpenMP code opts
- Compilers are improving rapidly, market forces ($$$) will drive advances

LLVM (UIUC)

- What is it: compiler technology
- Importance for many-core: supports many features critical to OpenCL
  - AMD OpenCL implementation based on LLVM
- Impact on compilation model (compilation is VERY cheap)
  - Current model is entrenched, pragmatic, but also archaic
  - Run-time/just-in-time (JIT) compilation
- Portable code changes from compile time to runtime issue
- This project has significant importance for future HPC

Building Upon OpenCL

- stdcl: POSIX-like extensions supporting OpenCL
  - Embed (static link) CL code into ELF objects
  - Initialize the most common use case by default
  - Pre-load and compile CL kernels for identified devices
    - `stdcpu[0], stdcpu[1], ... stdgpu[0], stdgpu[2], ...`
  - Add additional convenience functions in spirit of `stdio, stdlib`
  - Support dynamic/shared CL similar to dynamic libraries
    - `clopen(), clsym(), clclose()`
  - No interference with direct OpenCL support
- coprthr: pthreads extensions for co-processing
- Integrate OpenCL with existing, proven APIs, e.g., OpenMP
  - OpenMP fork-join model represents typical many-core use case
- Other ideas?
- OpenCL is foundation, expect open-source community to build software stack

Conclusions

- Many-core creates an inversion of HPC parallelism
- Many challenges, mostly confronted by software developer
- OpenCL may provide a foundation for programming model
- Much will depend on vendor delivery of good compilers and runtime implementations
- Introduces new concepts of compilation and portability
- Reconciling OpenCL software stack within HPC?