Accelerating Biomolecular Modeling with CUDA and GPU Clusters

James Phillips
John Stone
Klaus Schulten

http://www.ks.uiuc.edu/Research/gpu/
Beckman Institute
University of Illinois at Urbana-Champaign

Theoretical and Computational Biophysics Group
National Center for Supercomputing Applications

NIH Resource for Macromolecular Modeling and Bioinformatics
http://www.ks.uiuc.edu/

Beckman Institute, UIUC
Computational Microscopy

Ribosome: synthesizes proteins from genetic information, target for antibiotics

Silicon nanopore: bionanodevice for sequencing DNA efficiently
NAMD: Practical Supercomputing

• 35,000 users can’t all be computer experts.
  – 18% are NIH-funded; many in other countries.
  – 8200 have downloaded more than one version.

• User experience is the same on all platforms.
  – No change in input, output, or configuration files.
  – Run any simulation on any number of processors.
  – Precompiled binaries available when possible.

• Desktops and laptops – setup and testing
  – x86 and x86-64 Windows, and Macintosh
  – Allow both shared-memory and network-based parallelism.

• Linux clusters – affordable workhorses
  – x86, x86-64, and Itanium processors
  – Gigabit ethernet, Myrinet, InfiniBand, Quadrics, Altix, etc

Our Goal: Practical Acceleration

- Broadly applicable to scientific computing
  - Programmable by domain scientists
  - Scalable from small to large machines
- Broadly available to researchers
  - Price driven by commodity market
  - Low burden on system administration
- Sustainable performance advantage
  - Performance driven by Moore’s law
  - Stable market and supply chain
Acceleration Options for NAMD

• Outlook in 2005-2006:
  – FPGA reconfigurable computing (with NCSA)
    • Difficult to program, slow floating point, expensive
  – Cell processor (NCSA hardware)
    • Relatively easy to program, expensive
  – ClearSpeed (direct contact with company)
    • Limited memory and memory bandwidth, expensive
  – MDGRAPE
    • Inflexible and expensive
  – Graphics processor (GPU)
    • Program must be expressed as graphics operations
CUDA: Practical Performance

November 2006: NVIDIA announces CUDA for G80 GPU.

- CUDA makes GPU acceleration usable:
  - Developed and supported by NVIDIA.
  - No masquerading as graphics rendering.
  - New shared memory and synchronization.
  - No OpenGL or display device hassles.
  - Multiple processes per card (or vice versa).

- Resource and collaborators make it useful:
  - Experience from VMD development
  - David Kirk (Chief Scientist, NVIDIA)
  - Wen-mei Hwu (ECE Professor, UIUC)

VMD – “Visual Molecular Dynamics”

• Visualization and analysis of molecular dynamics simulations, sequence data, volumetric data, quantum chemistry simulations, particle systems, …
• User extensible with scripting and plugins
• http://www.ks.uiuc.edu/Research/vmd/
CUDA Acceleration in VMD

- Electrostatic field calculation, ion placement: 20x to 44x faster
- Molecular orbital calculation and display: 100x to 120x faster
- Imaging of gas migration pathways in proteins with implicit ligand sampling: 20x to 30x faster

NIH Resource for Macromolecular Modeling and Bioinformatics
http://www.ks.uiuc.edu/

Beckman Institute, UIUC
# Apples to Oranges Performance Results: Molecular Orbital Kernels

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Cores</th>
<th>Runtime (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel QX6700 CPU ICC-SSE (SSE intrinsics)</td>
<td>1</td>
<td>46.580</td>
<td>1.00</td>
</tr>
<tr>
<td>Intel Core2 Duo CPU OpenCL scalar</td>
<td>2</td>
<td>43.342</td>
<td>1.07</td>
</tr>
<tr>
<td>Intel QX6700 CPU ICC-SSE (SSE intrinsics)</td>
<td>4</td>
<td>11.740</td>
<td>3.97</td>
</tr>
<tr>
<td>Intel Core2 Duo CPU OpenCL vec4</td>
<td>2</td>
<td>8.499</td>
<td>5.36</td>
</tr>
<tr>
<td>Cell OpenCL vec4*** no __constant</td>
<td>16</td>
<td>6.075</td>
<td>7.67</td>
</tr>
<tr>
<td>Radeon 4870 OpenCL scalar</td>
<td>10</td>
<td>2.108</td>
<td>22.1</td>
</tr>
<tr>
<td>Radeon 4870 OpenCL vec4</td>
<td>10</td>
<td>1.016</td>
<td>45.8</td>
</tr>
<tr>
<td>GeForce GTX 285 OpenCL vec4</td>
<td>30</td>
<td>0.364</td>
<td>127.9</td>
</tr>
<tr>
<td>GeForce GTX 285 CUDA 2.1 scalar</td>
<td>30</td>
<td>0.361</td>
<td>129.0</td>
</tr>
<tr>
<td>GeForce GTX 285 OpenCL scalar</td>
<td>30</td>
<td>0.335</td>
<td>139.0</td>
</tr>
<tr>
<td>GeForce GTX 285 CUDA 2.0 scalar</td>
<td>30</td>
<td>0.327</td>
<td>142.4</td>
</tr>
</tbody>
</table>

CUDA results demonstrate performance variability with compiler revisions, and that with vendor effort, OpenCL has the potential to match the performance of other APIs.
NAMD Hybrid Decomposition


- Spatially decompose data and communication.
- Separate but related work decomposition.
- “Compute objects” facilitate iterative, measurement-based load balancing system.
NAMD Code is Message-Driven

- No receive calls as in “message passing”
- Messages sent to object “entry points”
- Incoming messages placed in queue
  - Priorities are necessary for performance
- Execution generates new messages
- Implemented in Charm++ on top of MPI
  - Can be emulated in MPI alone
  - Charm++ provides tools and idioms
  - Parallel Programming Lab: http://charm.cs.uiuc.edu/
System Noise Example
Timeline from Charm++ tool “Projections” http://charm.cs.uiuc.edu/
NAMD Overlapping Execution


Objects are assigned to processors and queued as data arrives.
Message-Driven CUDA?

• No, CUDA is too coarse-grained.
  – CPU needs fine-grained work to interleave and pipeline.
  – GPU needs large numbers of tasks submitted all at once.

• No, CUDA lacks priorities.
  – FIFO isn’t enough.

• Perhaps in a future interface:
  – Stream data to GPU.
  – Append blocks to a running kernel invocation.
  – Stream data out as blocks complete.

• Fermi looks very promising!
Nonbonded Forces on CUDA GPU

- Start with most expensive calculation: direct nonbonded interactions.
- Decompose work into pairs of patches, identical to NAMD structure.
- GPU hardware assigns patch-pairs to multiprocessors dynamically.

Force computation on single multiprocessor (GeForce 8800 GTX has 16)

texture<float4> force_table;
__constant__ unsigned int exclusions[];
__shared__ atom jatom[];
atom iatom; // per-thread atom, stored in registers
float4 iforce; // per-thread force, stored in registers

for ( int j = 0; j < jatom_count; ++j ) {
    float dx = jatom[j].x - iatom.x;
    float dy = jatom[j].y - iatom.y;
    float dz = jatom[j].z - iatom.z;
    float r2 = dx*dx + dy*dy + dz*dz;
    if ( r2 < cutoff2 ) {
        float4 ft = texfetch(force_table, 1.f/sqrt(r2));
        bool excluded = false;
        int indexdiff = iatom.index - jatom[j].index;
        if ( abs(indexdiff) <= (int)jatom[j].excl_maxdiff ) {
            indexdiff += jatom[j].excl_index;
            excluded = ((exclusions[indexdiff>>5] & (1<<(indexdiff&31))) != 0);
        }
        float f = iatom.half_sigma + jatom[j].half_sigma; // sigma
        f *= f*f; // sigma^3
        f *= f; // sigma^6
        f *= ( f * ft.x + ft.y ); // sigma^12 * fi.x - sigma^6 * fi.y
        f *= iatom.sqrt_epsilon * jatom[j].sqrt_epsilon;
        float qq = iatom.charge * jatom[j].charge;
        if ( excluded ) { f = qq * ft.w; } // PME correction
        else { f += qq * ft.z; } // Coulomb
        iforce.x += dx * f;
        iforce.y += dy * f;
        iforce.z += dz * f;
        iforce.w += 1.f; // interaction count or energy
    }
}

Overlapping GPU and CPU with Communication

One Timestep
“Remote Forces”

- Forces on atoms in a local patch are “local”
- Forces on atoms in a remote patch are “remote”
- Calculate remote forces first to overlap force communication with local force calculation
- Not enough work to overlap with position communication

Work done by one processor
Actual Timelines from NAMD

Generated using Charm++ tool “Projections” http://charm.cs.uiuc.edu/
NCSA “4+4” QuadroPlex Cluster

seconds per step

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>60</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU only</td>
<td>6.76</td>
<td>3.33</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>with GPU</td>
<td></td>
<td></td>
<td>0.40</td>
<td>0.20</td>
<td>0.04</td>
</tr>
<tr>
<td>GPU</td>
<td></td>
<td></td>
<td>0.30</td>
<td>0.10</td>
<td>0.01</td>
</tr>
</tbody>
</table>

2.4 GHz Opteron + Quadro FX 5600

faster
Current GPU Clusters at NCSA

• Lincoln
  – Production system available via the standard NCSA/TeraGrid HPC allocation

• AC
  – Experimental system available for exploring GPU computing
CUDA/OpenCL Wrapper Library

• Basic operation principle:
  – Use /etc/ld.so.preload to overload (intercept) a subset of CUDA/OpenCL functions, e.g. {cu|cuda} {Get|Set}Device, clGetDeviceIDs, etc

• Purpose:
  – Enables controlled GPU device visibility and access, extending resource allocation to the workload manager
  – Prove or disprove feature usefulness, with the hope of eventual uptake or reimplementation of proven features by the vendor
  – Provides a platform for rapid implementation and testing of HPC relevant features not available in NVIDIA APIs

• Features:
  – NUMA Affinity mapping
    • Sets thread affinity to CPU core nearest the gpu device
  – Shared host, multi-gpu device fencing
    • Only GPUs allocated by scheduler are visible or accessible to user
    • GPU device numbers are virtualized, with a fixed mapping to a physical device per user environment
    • User always sees allocated GPU devices indexed from 0
CUDA/OpenCL Wrapper Library

• Features (cont’d):
  – Device Rotation (deprecated)
    • Virtual to Physical device mapping rotated for each process accessing a GPU device
    • Allowed for common execution parameters (e.g. Target gpu0 with 4 processes, each one gets separate gpu, assuming 4 gpus available)
    • CUDA 2.2 introduced compute-exclusive device mode, which includes fallback to next device. Device rotation feature may no longer needed
  – Memory Scrubber
    • Independent utility from wrapper, but packaged with it
    • Linux kernel does no management of GPU device memory
    • Must run between user jobs to ensure security between users

• Availability
  – NCSA/UofI Open Source License
  – https://sourceforge.net/projects/cudawrapper/
CUDA Memtest

- 4GB of Tesla GPU memory is not ECC protected
- Hunt for “soft error”
- Features
  - Full re-implementation of every test included in memtest86
  - Random and fixed test patterns, error reports, error addresses, test specification
  - Email notification
  - Includes additional stress test for software and hardware errors
- Usage scenarios
  - Hardware test for defective GPU memory chips
  - CUDA API/driver software bugs detection
  - Hardware test for detecting soft errors due to non-ECC memory
- No soft error detected in 2 years x 4 gig of cumulative runtime
- Availability
  - NCSA/UofI Open Source License
  - https://sourceforge.net/projects/cudagpumemtest/
GPU Node Pre/Post Allocation Sequence

• Pre-Job (minimized for rapid device acquisition)
  – Assemble detected device file unless it exists
  – Sanity check results
  – Checkout requested GPU devices from that file
  – Initialize CUDA wrapper shared memory segment with unique key for user (allows user to ssh to node outside of job environment and have same gpu devices visible)

• Post-Job
  – Use quick memtest run to verify healthy GPU state
  – If bad state detected, mark node offline if other jobs present on node
  – If no other jobs, reload kernel module to “heal” node (for CUDA 2.2 driver bug)
  – Run memscrubber utility to clear gpu device memory
  – Notify of any failure events with job details
  – Terminate wrapper shared memory segment
  – Check-in GPUs back to global file of detected devices
AMD Opteron Tesla Linux Cluster AC

- **HP xw9400 workstation**
  - 2216 AMD Opteron 2.4 GHz dual socket dual core
  - 8 GB DDR2
  - Infiniband QDR
- **Tesla S1070 1U 4-GPU Server**
  - 1.3 GHz Tesla T10 processors
  - 4x4 GB GDDR3 SDRAM
- **Cluster**
  - Servers: 32
  - Accelerator Units: 32 (128 GPUs, 128 TF SP, 10 TF DP)
Intel 64 Tesla Linux Cluster *Lincoln*

- **Dell PowerEdge 1955 server**
  - Intel 64 (Harpertown) 2.33 GHz dual socket quad core
  - 16 GB DDR2
  - Infiniband SDR
- **Tesla S1070 1U 4-GPU Server**
  - 1.3 GHz Tesla T10 processors
  - 4x4 GB GDDR3 SDRAM
- **Cluster**
  - Servers: 192
  - Accelerator Units: 96 (384 GPUs, 384 TF SP, 32 TF DP)
NCSA “8+2” Lincoln Cluster

• How to share a GPU among 4 CPU cores?
  – Send all GPU work to one process?
  – Coordinate via messages to avoid conflict?
  – Or just hope for the best?
NCSA Lincoln Cluster Performance
(8 Intel cores and 2 NVIDIA Telsa GPUs per node)

STMV (1M atoms) s/step

- 2 GPUs = 24 cores
- 4 GPUs
- 8 GPUs
- 16 GPUs

~2.8

CPU cores: 8, 16, 32, 64
NCSA Lincoln Cluster Performance
(8 cores and 2 GPUs per node)

STMV s/step

2 GPUs = 24 cores
4 GPUs
8 GPUs
16 GPUs

8 GPUs = 96 CPU cores

NIH Resource for Macromolecular Modeling and Bioinformatics
http://www.ks.uiuc.edu/

Beckman Institute, UIUC
No GPU Sharing (Ideal World)
GPU Sharing (Desired)

Client 1

Remote Force

Local Force

Client 2

Remote Force

Local Force
GPU Sharing (Feared)

Client 1

Client 2
GPU Sharing (Observed)
GPU Sharing (Explained)

- CUDA is behaving reasonably, but
- Force calculation is actually two kernels
  - Longer kernel writes to multiple arrays
  - Shorter kernel combines output
- Possible solutions:
  - Modify CUDA to be less “fair” (please!)
  - Use locks (atomics) to merge kernels (not G80)
  - Explicit inter-client coordination
Inter-client Communication

- First identify which processes share a GPU
  - Need to know physical node for each process
  - GPU-assignment must reveal real device ID
  - Threads don’t eliminate the problem
  - Production code can’t make assumptions

- Token-passing is simple and predictable
  - Rotate clients in fixed order
  - High-priority, yield, low-priority, yield, …
Token-Passing GPU-Sharing

GPU1

GPU2

Local  Remote  Local  Remote
GPU-Sharing with PME
Weakness of Token-Passing

• GPU is idle while token is being passed
  – Busy client delays itself and others

• Next strategy requires threads:
  – One process per GPU, one thread per core
  – Funnel CUDA calls through a single stream
  – No local work until all remote work is queued
  – Typically funnels MPI as well
Recent NAMD GPU Developments

• Production features in 2.7b2 release:
  – Full electrostatics with PME
  – 1-4 exclusions
  – Constant-pressure simulation
  – Improved force accuracy:
    • Patch-centered atom coordinates
    • Increased precision of force interpolation

• Performance enhancements (in progress):
  – Recursive bisection within patch on 32-atom boundaries
  – Block-based pairlists based on sorted atoms
  – Sort blocks in order of decreasing work
GPU-Accelerated NAMD Plans

• Serial performance
  – Target NVIDIA Fermi architecture
  – Revisit GPU kernel design decisions made in 2007
  – Improve performance of remaining CPU code

• Parallel scaling
  – Target NSF Track 2D Keeneland cluster at ORNL
  – Finer-grained work units on GPU (feature of Fermi)
  – One process per GPU, one thread per CPU core
  – Dynamic load balancing of GPU work

• Wider range of simulation options and features
Conclusions and Outlook

• CUDA today is sufficient for
  – Single-GPU acceleration (the mass market)
  – Coarse-grained multi-GPU parallelism
    • Enough work per call to spin up all multiprocessors

• Improvements in CUDA are needed for
  – Assigning GPUs to processes
  – Sharing GPUs between processes
  – Fine-grained multi-GPU parallelism
    • Fewer blocks per call than chip has multiprocessors
  – Moving data between GPUs (same or different node)

• Eager to test Fermi architecture and features!
Acknowledgements

- Theoretical and Computational Biophysics Group, University of Illinois at Urbana-Champaign
- Prof. Wen-mei Hwu, Chris Rodrigues, IMPACT Group, University of Illinois at Urbana-Champaign
- Mike Showerman, Jeremy Enos, NCSA
- David Kirk, Massimiliano Fatica, others at NVIDIA
- UIUC NVIDIA CUDA Center of Excellence
- NIH support: P41-RR05969

http://www.ks.uiuc.edu/Research/gpu/