OpenCL for Molecular Modeling Applications: Early Experiences

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

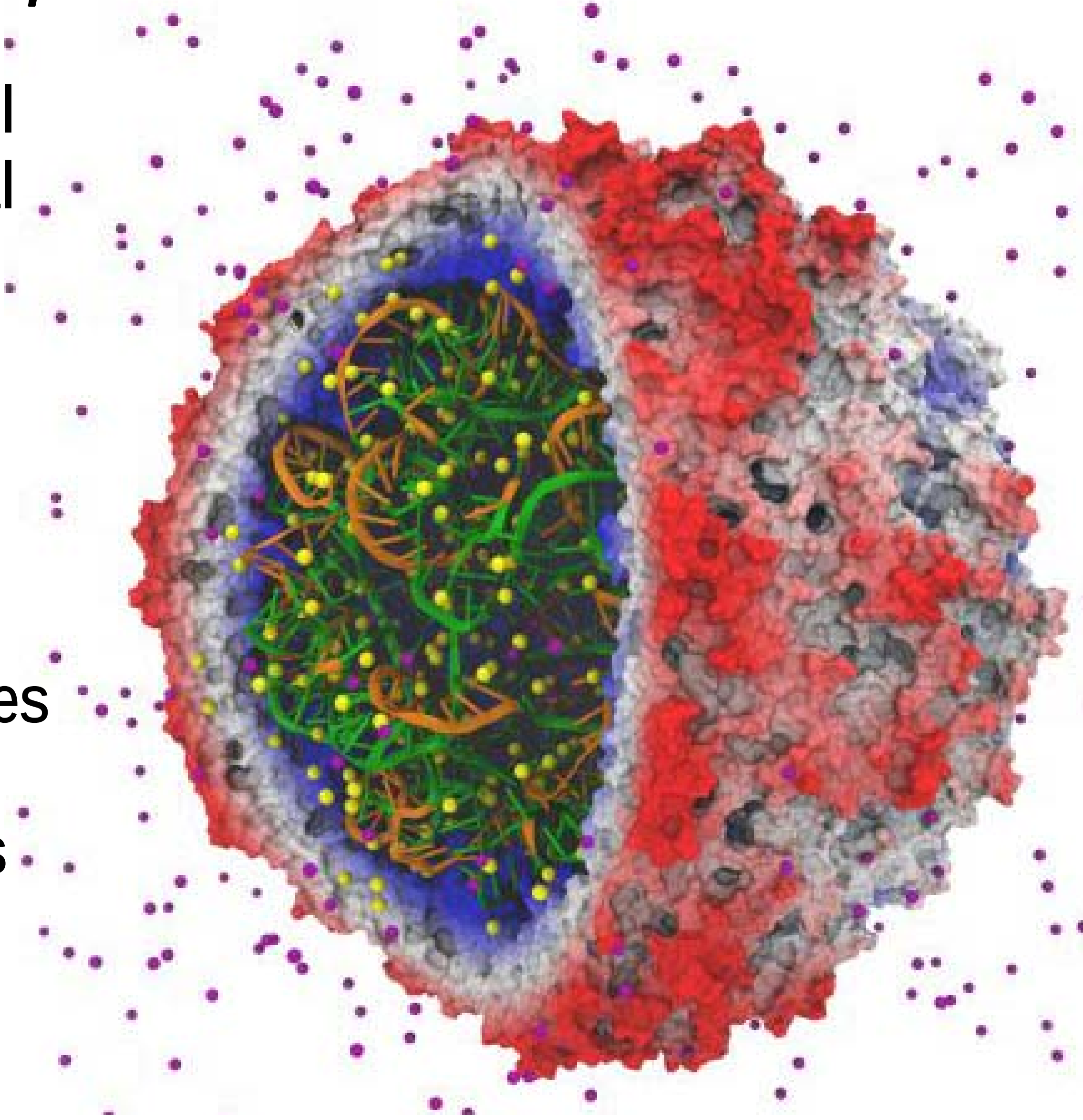
SC2009 OpenCL BOF, November 18, 2009

Overview

- Molecular modeling applications, need for higher performance, increased energy efficiency
- Potential benefits of OpenCL for molecular modeling applications
- Early experiences with OpenCL 1.0 beta implementations
- Some crude performance results using existing OpenCL toolkits, both production and alpha/beta
- Detailed comparison of some OpenCL kernels targeting CPUs, GPUs, Cell, etc.
- OpenCL middleware opportunities, existing efforts

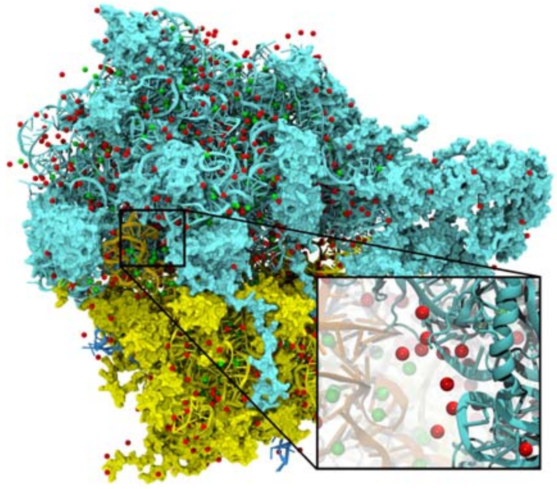
Computational Biology's Insatiable Demand for Computational Power

- Many simulations still fall short of biological timescales
- Large simulations extremely difficult to prepare, analyze
- Performance increases allow use of more sophisticated models



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

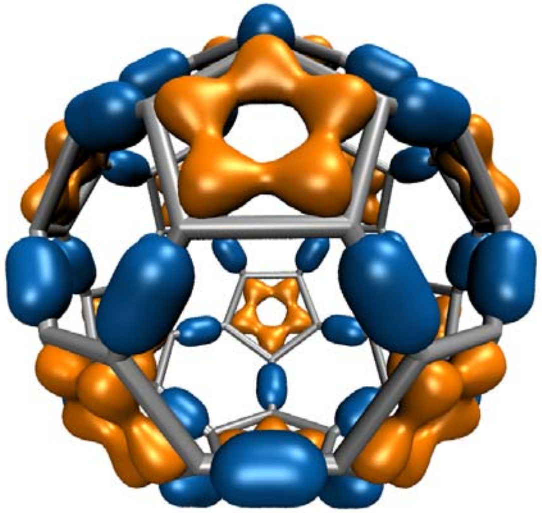
CUDA+OpenCL Acceleration in VMD



Electrostatic field

calculation, ion placement

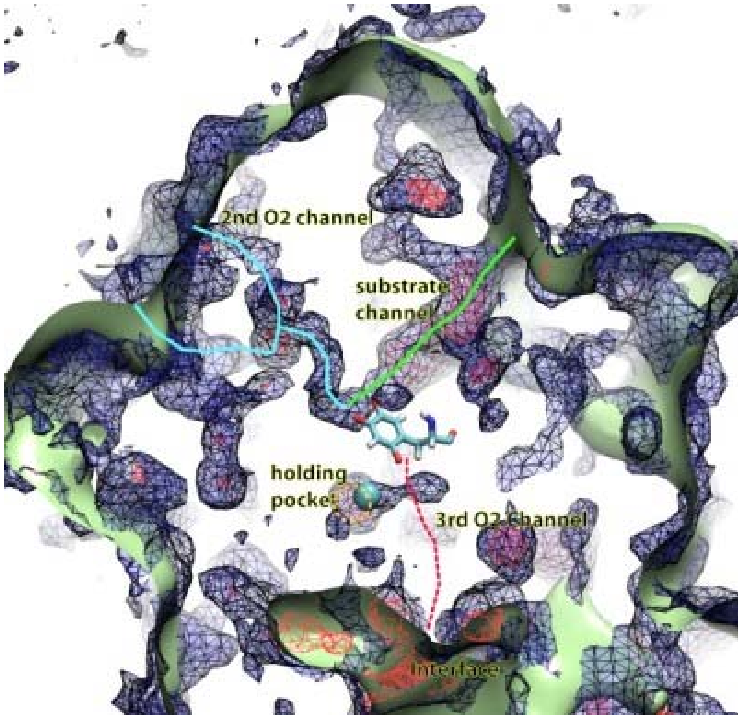
20x to 44x faster



Molecular orbital

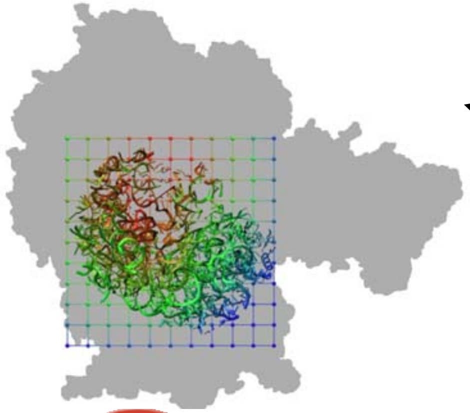
calculation and display

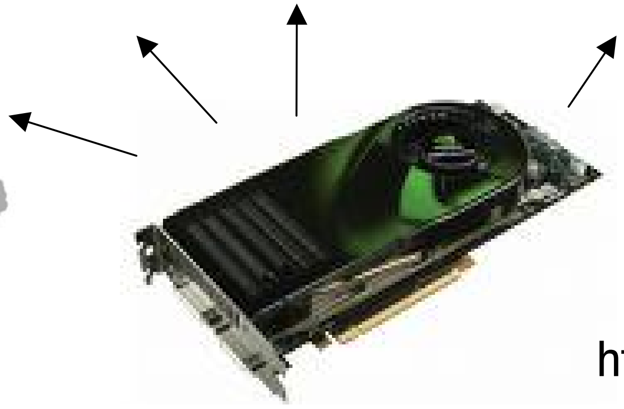
100x to 120x faster



Imaging of gas migration pathways in proteins with implicit ligand sampling

20x to 30x faster





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

Supporting Diverse Accelerator Hardware in Production Codes….

- Development of HPC-oriented scientific software is already challenging
- Maintaining unique code paths for each accelerator type is costly and impractical beyond a certain point
- Diversity and rapid evolution of accelerators exacerbates these issues
- OpenCL ameliorates several key problems:
  - Targets CPUs, GPUs, and other accelerator devices
  - Common language for writing computational "kernels"
  - Common API for managing execution on target device

Strengths and Weaknesses of Current OpenCL Implementations

- Code is portable to multiple OpenCL device types
  - Correctly written OpenCL code will run and yield correct results on multiple devices
- Performance is not necessarily portable
  - Some OpenCL API/language constructs naturally map better to some target devices than others
  - OpenCL can't hide significant differences in HW architecture
- Room for improvement in existing OpenCL compilers:
  - Sophisticated batch-mode compilers have long provided autovectorization and high-end optimization techniques
  - Current alpha/beta OpenCL implementations aren't quite there yet
  - Some compiler optimization techniques might be too slow to be made available in the typical runtime-compilation usage of OpenCL

Strengths and Weaknesses of Current OpenCL Implementations (2)

- OpenCL is a low-level API
  - Freedom to wire up your code in a variety of ways
  - Developers are responsible for a lot of plumbing, lots of objects/handles to keep track of
  - A basic OpenCL "hello world" is **_much_** more code to write than doing the same thing in the CUDA runtime API
- Developers are responsible for enforcing thread-safety in many cases
  - Some multi-accelerator codes are much more difficult to write than in the CUDA runtime API
- Great need for OpenCL middleware and/or libraries
  - Simplified device management, integration of large numbers of kernels into legacy apps, auto-selection of best kernels for device...
  - Tools to better support OpenCL apps in large HPC environments, e.g. clusters (more on this at the end of my talk)

Electrostatic Potential Maps

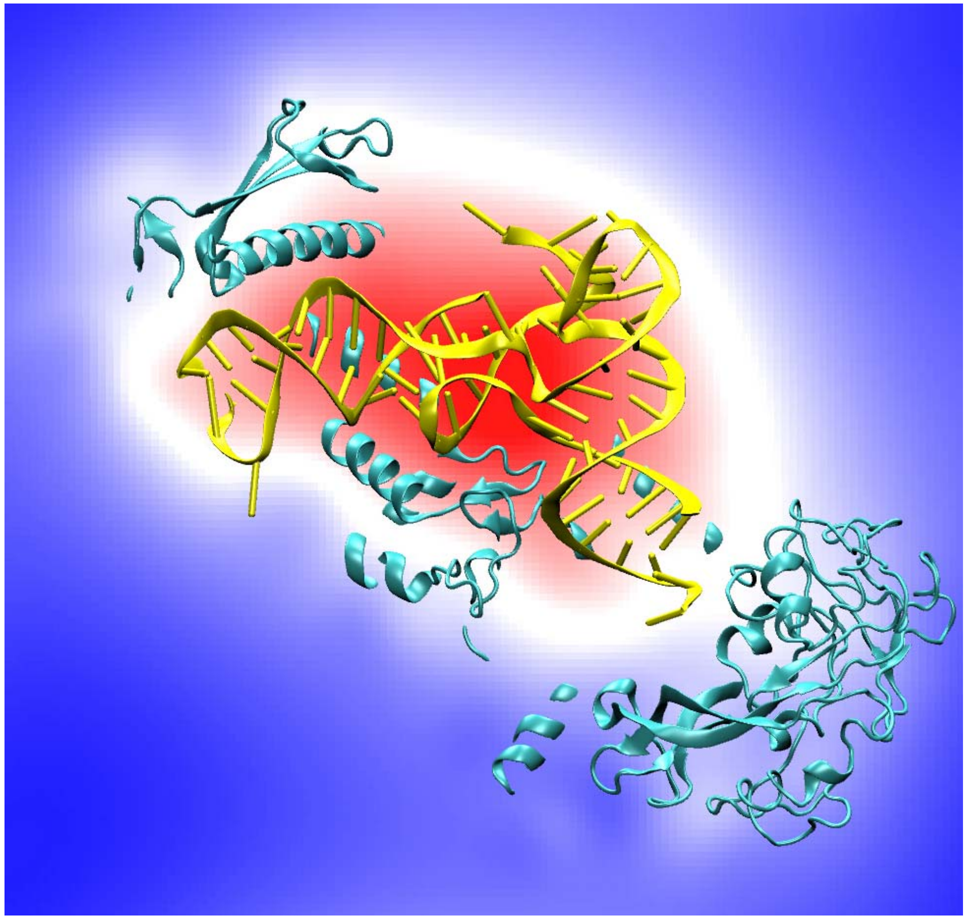
- Electrostatic potentials evaluated on 3-D lattice:

$$V_i = \sum_j \frac{q_j}{4\pi\epsilon_0 |\mathbf{r}_j - \mathbf{r}_i|}$$

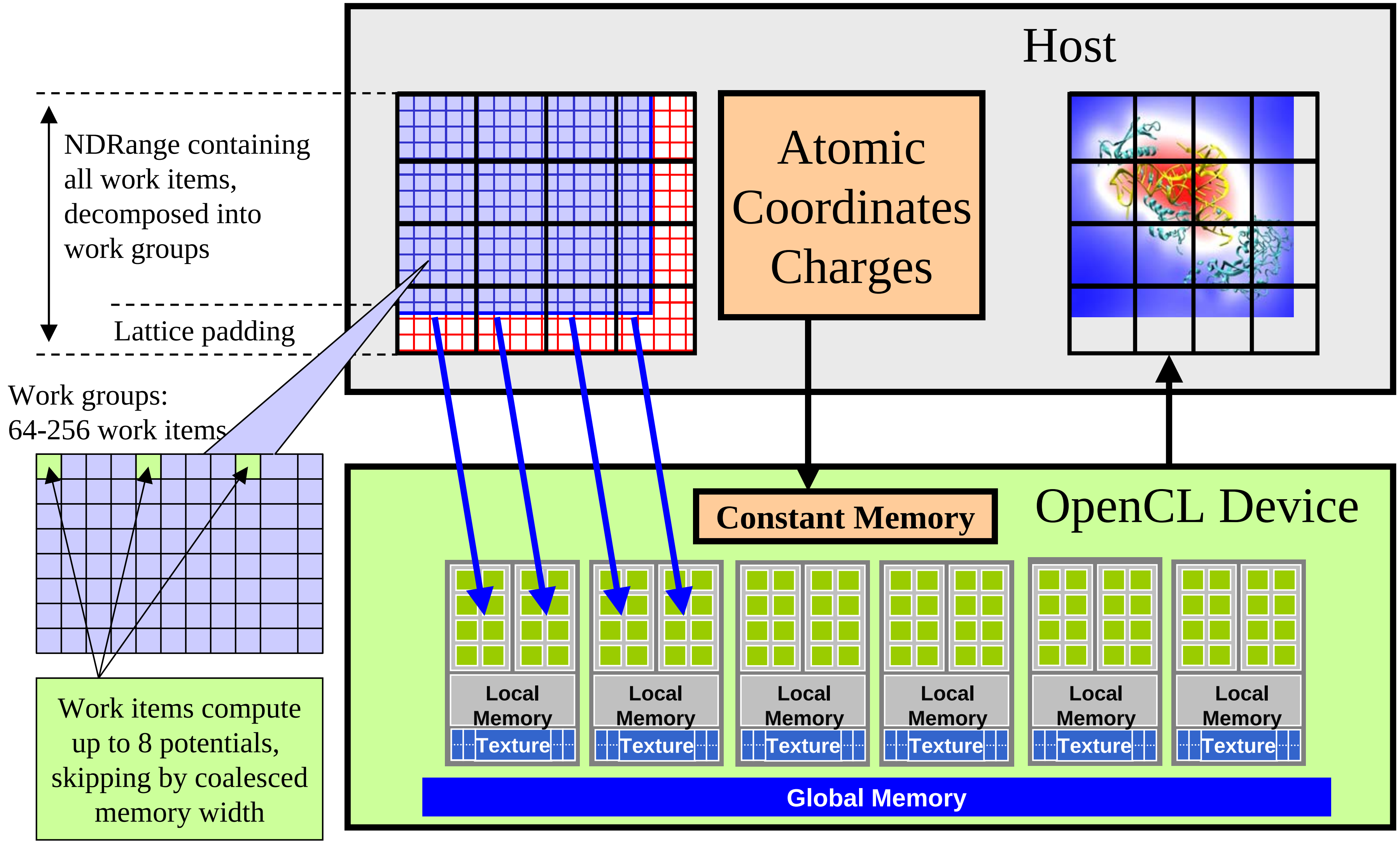
- Applications include:
  - Ion placement for structure building
  - Time-averaged potentials for simulation
  - Visualization and analysis

Accelerating molecular modeling applications with graphics processors.

Stone et al., . *J. Comp. Chem.*, 28:2618-2640, 2007.

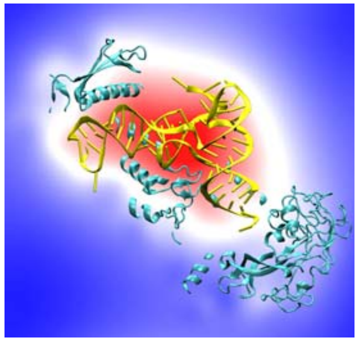

Isoleucine tRNA synthetase

Direct Coulomb Summation in OpenCL, Building Block for Better Algorithms


Host
NDRange containing all work items, decomposed into work groups
Lattice padding
Work groups: 64-256 work items
Atomic Coordinates Charges
Work items compute up to 8 potentials, skipping by coalesced memory width
Constant Memory
OpenCL Device
Local Memory
Texture
Global Memory

Stone et al., J. Comp. Chem., 28:2618-2640, 2007.

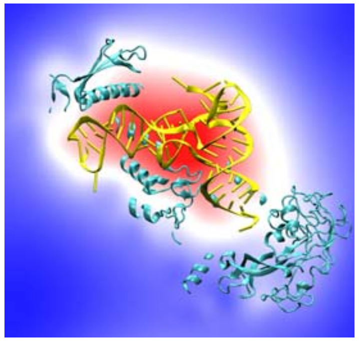
DCS Inner Loop, Scalar OpenCL



```
…for (atomid=0; atomid<numatoms; atomid++) {
    float dy = coory - atominfo[atomid].y;
    float dyz2 = (dy * dy) + atominfo[atomid].z;
    float dx1 = coorx – atominfo[atomid].x;
    float dx2 = dx1 + gridspacing_coalesce;
    float dx3 = dx2 + gridspacing_coalesce;
    float dx4 = dx3 + gridspacing_coalesce;
    float charge = atominfo[atomid].w;
    energyvalx1 += charge * native_rsqrt(dx1*dx1 + dyz2);
    energyvalx2 += charge * native_rsqrt(dx2*dx2 + dyz2);
    energyvalx3 += charge * native_rsqrt(dx3*dx3 + dyz2);
    energyvalx4 += charge * native_rsqrt(dx4*dx4 + dyz2);
  }
```

Well-written CUDA code can often be easily ported to OpenCL if C++ features and pointer arithmetic aren't used in kernels.

DCS Inner Loop, Vectorized OpenCL



```
  float4 gridspacing_u4 = { 0.f, 1.f, 2.f, 3.f };
  gridspacing_u4 *= gridspacing_coalesce;
  float4 energyvalx=0.0f;
…
  for (atomid=0; atomid<numatoms          +) {
     float dy = coory - atominfo[at      ].y;
     float dyz2 = (dy * dy) + ato    nfo[atomid].z;
     float4 dx = gridspacing_u4 + (coorx – atominfo[atomid].x);
     float charge = atominfo[atomid].w;
     energyvalx1 += charge * native_rsqrt(dx1*dx1 + dyz2);
  }
```

CPUs, AMD GPUs, and Cell often perform better with vectorized kernels.
Use of vector types may increase register pressure; sometimes a delicate balance…

Apples to Oranges Performance Results:
OpenCL Direct Coulomb Summation Kernels

OpenCL Target Device	OpenCL "cores"	Scalar Kernel: Ported from original CUDA kernel	4-Vector Kernel: Replaced manually unrolled loop iterations with float4 vector ops
AMD 2.2GHz Opteron 148 CPU (a very old Linux test box)	1	0.30 Bevals/sec, 2.19 GFLOPS	0.49 Bevals/sec, 3.59 GFLOPS
Intel 2.2Ghz Core2 Duo, (Apple MacBook Pro)	2	0.88 Bevals/sec, 6.55 GFLOPS	2.38 Bevals/sec, 17.56 GFLOPS
IBM QS22 CellBE *** __constant not implemented yet	16	2.33 Bevals/sec, 17.16 GFLOPS ****	6.21 Bevals/sec, 45.81 GFLOPS ****
AMD Radeon 4870 GPU	10	41.20 Bevals/sec, 303.93 GFLOPS	31.49 Bevals/sec, 232.24 GFLOPS
NVIDIA GeForce GTX 285 GPU	30	75.26 Bevals/sec, 555.10 GFLOPS	73.37 Bevals/sec, 541.12 GFLOPS

MADD, RSQRT = 2 FLOPS All other FP instructions = 1 FLOP

Getting More Performance: Adapting DCS Kernel to OpenCL on Cell

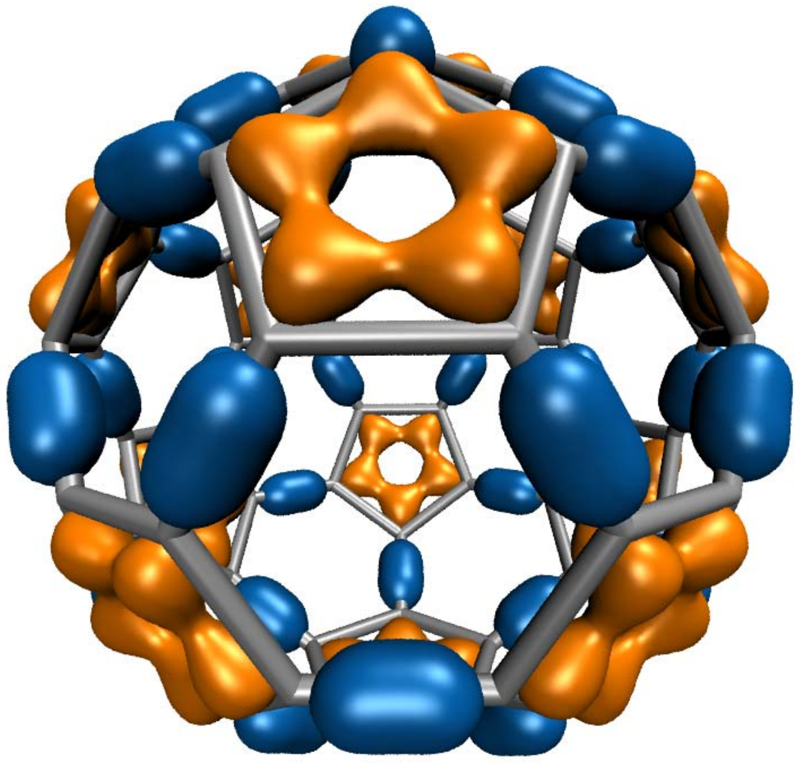
OpenCL Target Device	Scalar Kernel: Ported directly from original CUDA kernel	4-Vector Kernel: Replaced manually unrolled loop iterations with float4 vector ops	Async Copy Kernel: Replaced __constant accesses with use of async_work_group_copy(), use float16 vector ops
IBM QS22 CellBE *** __constant not implemented	2.33 Bevals/sec, 17.16 GFLOPS ****	6.21 Bevals/sec, 45.81 GFLOPS ****	16.22 Bevals/sec, 119.65 GFLOPS

Replacing the use of constant memory with loads of atom data to __local memory via async_work_group_copy() increases performance significantly since Cell doesn't implement __constant memory yet.

Tests show that the speed of native_rsqrt() is currently a performance limiter for Cell. Replacing native_rsqrt() with a multiply results in a ~3x increase in execution rate.

Computing Molecular Orbitals

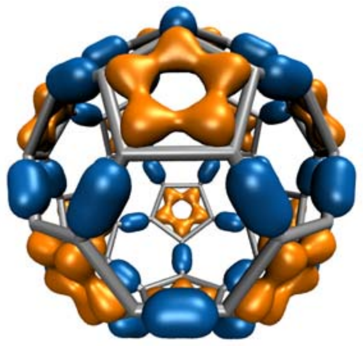
- Visualization of MOs aids in understanding the chemistry of molecular system
- MO spatial distribution is correlated with electron probability density
- Calculation of high resolution MO grids can require tens to hundreds of seconds on CPUs
- >100x speedup allows interactive animation of MOs @ 10 FPS



C_{60}

High Performance Computation and Interactive Display of Molecular Orbitals on GPUs and Multi-core CPUs. Stone et al., *GPGPU-2, ACM International Conference Proceeding Series*, volume 383, pp. 9-18, 2009

Molecular Orbital Inner Loop, Hand-Coded SSE Hard to Read, Isn't It? *(And this is the "pretty" version!)*

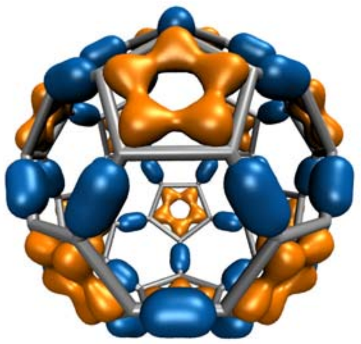


```
for (shell=0; shell < maxshell; shell++) {
   __m128 Cgto = _mm_setzero_ps();
   for (prim=0; prim<num_prim_per_shell[shell_counter]; prim++) {
      float exponent       = -basis_array[prim_counter    ];
      float contract_coeff =  basis_array[prim_counter + 1];
      __m128 expval = _mm_mul_ps(_mm_load_ps1(&exponent), dist2);
      __m128 ctmp = _mm_mul_ps(_mm_load_ps1(&contract_coeff), exp_ps(expval));
      Cgto = _mm_add_ps(contracted_gto, ctmp);
      prim_counter += 2;
   }
   __m128 tshell = _mm_setzero_ps();
   switch (shell_types[shell_counter]) {
      case S_SHELL:
         value = _mm_add_ps(value, _mm_mul_ps(_mm_lo      s1(&wave_f[ifunc++]), Cgto));   break;
      case P_SHELL:
         tshell = _mm_add_ps(tshell, _mm_mul_ps(_mm_load_ps1(&wave_f[ifunc++]), xdist));
         tshell = _mm_add_ps(tshell, _mm_mul_ps(_mm_load_ps1(&wave_f[ifunc++]), ydist));
         tshell = _mm_add_ps(tshell, _mm_mul_ps(_mm_load_ps1(&wave_f[ifunc++]), zdist));
         value = _mm_add_ps(value, _mm_mul_ps(tshell, Cgto));
         break;
```

Until now, writing SSE kernels for CPUs required assembly language, compiler intrinsics, various libraries, or a really smart autovectorizing compiler and lots of luck...

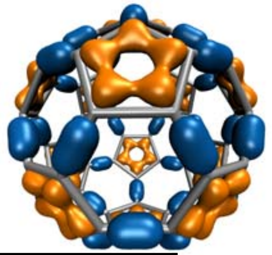
Molecular Orbital Inner Loop, OpenCL Vec4
Ahhh, much easier to read!!!

```
for (shell=0; shell < maxshell; shell++) {
    float4 contracted_gto = 0.0f;
    for (prim=0; prim < const_num_prim_per_shell[shell_counter];  prim++) {
        float exponent          = const_basis_array[prim_counter     ];
        float contract_coeff = const_basis_array[prim_counter + 1];
        contracted_gto += contract_coeff * native_exp2(-exponent*dist2);
        prim_counter += 2;
    }
    float4 tmpshell=0.0f;
    switch (const_shell_symmetry[shell_counter]) {
        case S_SHELL:
            value += const_wave_f[ifunc++] * contracted_gto;       break;
        case P_SHELL:
            tmpshell += const_wave_f[ifunc++] * xdist;
            tmpshell += const_wave_f[ifunc++] * ydist;
            tmpshell += const_wave_f[ifunc++] * zdist;
            value += tmpshell * contracted_gto;
            break;
```



OpenCL's C-like kernel language is easy to read, even 4-way vectorized kernels can look similar to scalar CPU code.
All 4-way vectors shown in green.

Apples to Oranges Performance Results: OpenCL Molecular Orbital Kernels



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

Minor varations in compiler quality can have a strong effect on “tight” kernels. The two results shown for CUDA demonstrate performance variability with compiler revisions, and that with vendor effort, OpenCL has the potential to match the performance of other APIs.

MCUDA for OpenCL

- MCUDA → “Multicore CUDA”: Compilation framework originally developed to allow retargeting CUDA kernels to multicore CPUs
  - "MCUDA: An Efficient Implementation of CUDA Kernels on Multi-cores“, J. Stratton, S. Stone, W. Hwu. Technical report, University of Illinois at Urbana-Champaign, IMPACT-08-01, March, 2008.
- Potential extensions to MCUDA for OpenCL:
  - Make OpenCL performance portable: generate vectorized OpenCL kernels from scalar CUDA or OpenCL kernels, as needed by specific target devices
  - Translate CUDA kernels to OpenCL
- Availability:

  http://impact.crhc.illinois.edu/mcuda.php

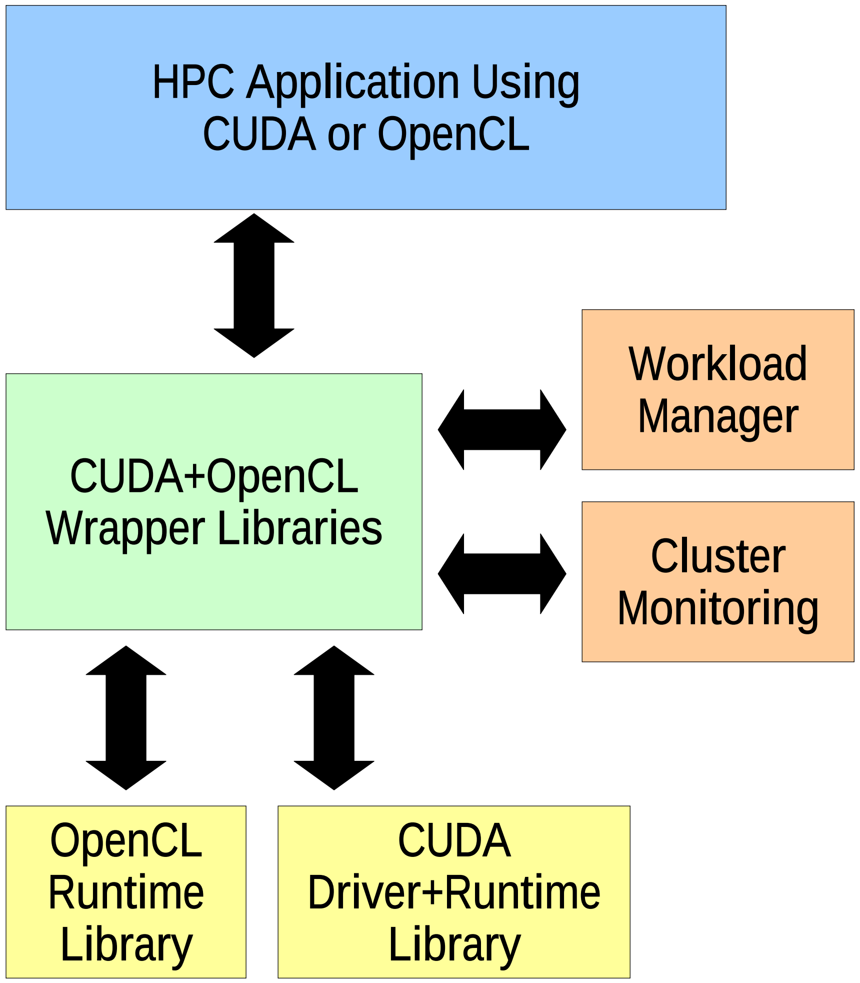
OpenCL on GPU Clusters at NCSA

- Lincoln
  - 1536 CPUs, 384 Tesla GPUs
  - Production system
- AC
  - 128 CPUs, 128 Tesla GPUs
  - Available for GPU devel & experimentation

LINCOLN
LINCOLN
Peak 47.5 Teraflops
Architecture Dell PowerEdge Xeon cluster with NVIDIA Tesla accelerators
Total CPUs 1536 CPUs, 384 GPUs
Total Memory 3.072 Terabytes
Launch Date 2009
NCSA
DELL

NCSA CUDA/OpenCL Wrapper Library

- Virtualize accelerator devices
- Workload manager control of device visibility, access, and resource allocation
- Transparent monitoring of application, accelerator activity:
  - Measure accelerator utilization and performance by individual HPC codes
  - Track accelerator and API usage by user community
- Rapid implementation and evaluation of HPC-relevant features not available in standard CUDA / OpenCL APIs
- Hope for eventual uptake of proven features by vendors and standards organizations


HPC Application Using CUDA or OpenCL
CUDA+OpenCL Wrapper Libraries
Workload Manager
Cluster Monitoring
OpenCL Runtime Library
CUDA Driver+Runtime Library

Kindratenko et al. Workshop on Parallel Programming on Accelerator Clusters (PPAC), IEEE Cluster 2009.

NCSA CUDA/OpenCL Wrapper Library

- Principle of operation:
  - Use /etc/ld.so.preload to overload (intercept) a subset of CUDA / OpenCL device management APIs, e.g. cudaSetDevice(), clGetDeviceIDs(), etc…
- Features:
  - NUMA affinity mapping:
    - Sets application thread affinity to the CPU core nearest to the target device
    - Maximizes host-device transfer bandwidth, particularly on multi-GPU hosts
  - Shared host, multi-GPU device fencing
    - Only GPUs allocated by batch scheduler are visible / accessible to application
    - GPU device IDs are virtualized, with a fixed mapping to a physical device per user environment
    - User always sees allocated GPU devices indexed from 0
  - Device ID Rotation
    - 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)

NCSA CUDA/OpenCL Wrapper Library

- Memory Scrubber Utility
  - Linux kernel does no management of GPU device memory
  - Must run between user jobs to ensure security between users
  - Independent utility from wrapper, but packaged with it
- CUDA/OpenCL Wrapper Authors:
  - Jeremy Enos <jenos at ncsa.uiuc.edu>
  - Guochun Shi <gshi at ncsa.uiuc.edu>
  - Volodymyr Kindratenko <kindrtnk at illinois.edu>
- Wrapper Software Availability
  - NCSA / UIUC Open Source License
  - https://sourceforge.net/projects/cudawrapper/

Acknowledgements

- University of Illinois at Urbana-Champaign
  - Theoretical and Computational Biophysics Group, NIH Resource for Macromolecular Modeling and Bioinformatics
  - NVIDIA CUDA Center of Excellence
  - Wen-mei Hwu and the IMPACT Group
  - NCSA Innovative Systems Laboratory:
    - Jeremy Enos, Guochun Shi, Volodymyr Kindratenko
- AMD, IBM, NVIDIA
  - Access to alpha+beta OpenCL toolkits and drivers
  - Many timely bug fixes
  - Consultation and guidance
- NIH support: P41-RR05969

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

- Probing Biomolecular Machines with Graphics Processors. J. Phillips, J. Stone. *Communications of the ACM,* 52(10):34-41, 2009.
- GPU Clusters for High Performance Computing. V. Kindratenko, J. Enos, G. Shi, M. Showerman, G. Arnold, J. Stone, J. Phillips, W. Hwu. *Workshop on Parallel Programming on Accelerator Clusters (PPAC),* IEEE Cluster 2009. In press.
- Long time-scale simulations of in vivo diffusion using GPU hardware. E. Roberts, J. Stone, L. Sepulveda, W. Hwu, Z. Luthey-Schulten. In *IPDPS'09: Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Computing*, pp. 1-8, 2009.
- High Performance Computation and Interactive Display of Molecular Orbitals on GPUs and Multi-core CPUs. J. Stone, J. Saam, D. Hardy, K. Vandivort, W. Hwu, K. Schulten, *2nd Workshop on General-Purpose Computation on Graphics Pricessing Units (GPGPU-2), ACM International Conference Proceeding Series*, volume 383, pp. 9-18, 2009.
- Multilevel summation of electrostatic potentials using graphics processing units. D. Hardy, J. Stone, K. Schulten. *J. Parallel Computing*, 35:164-177, 2009.

Publications (cont)
http://www.ks.uiuc.edu/Research/gpu/

- Adapting a message-driven parallel application to GPU-accelerated clusters. J. Phillips, J. Stone, K. Schulten. *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, IEEE Press, 2008.
- GPU acceleration of cutoff pair potentials for molecular modeling applications. C. Rodrigues, D. Hardy, J. Stone, K. Schulten, and W. Hwu. *Proceedings of the 2008 Conference On Computing Frontiers*, pp. 273-282, 2008.
- GPU computing. J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, J. Phillips. *Proceedings of the IEEE*, 96:879-899, 2008.
- Accelerating molecular modeling applications with graphics processors*.* J. Stone, J. Phillips, P. Freddolino, D. Hardy, L. Trabuco, K. Schulten. *J. Comp. Chem.*, 28:2618-2640, 2007.
- Continuous fluorescence microphotolysis and correlation spectroscopy. A. Arkhipov, J. Hüve, M. Kahms, R. Peters, K. Schulten. *Biophysical Journal*, 93:4006-4017, 2007.