U.S. DEPARTMENT OF ENERGY

Some developments in QMC for solids

QMC in the Apuan Alps
July 27, 2009

Ken Esler, Jeongnim Kim, R.E. Cohen,
Luke Shulenburger, David Ceperley

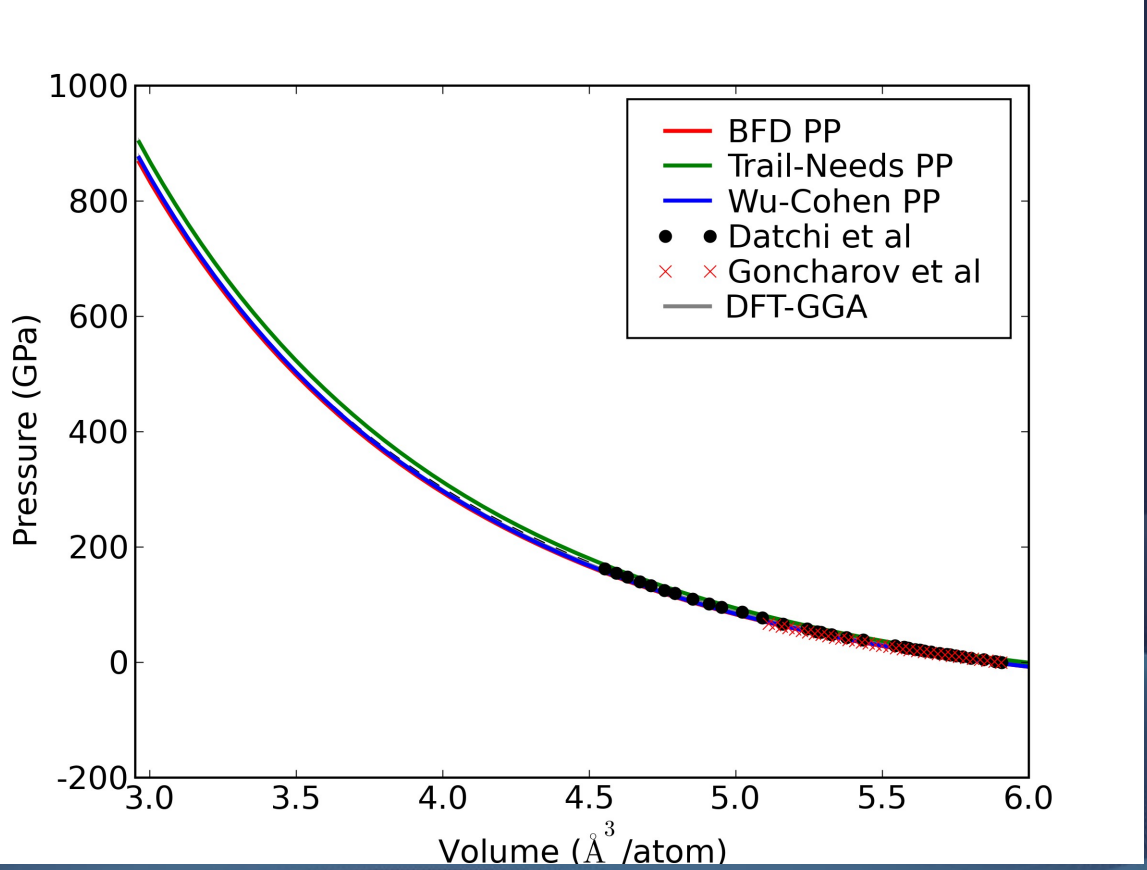
Outline

- All-electron vs. PP calculations in solids
  - Pseudopotential discrepancies
  - All-electron simulation in solids
  - Combining AE and PP data
- Mixed-basis representation for orbitals
  - Storage bottlenecks
  - Older approaches
  - Mixed-basis representation
- QMC simulations on GPUs
  - Intro to GPU computing
  - QMCPACK on GPUs

Pseudopotentials

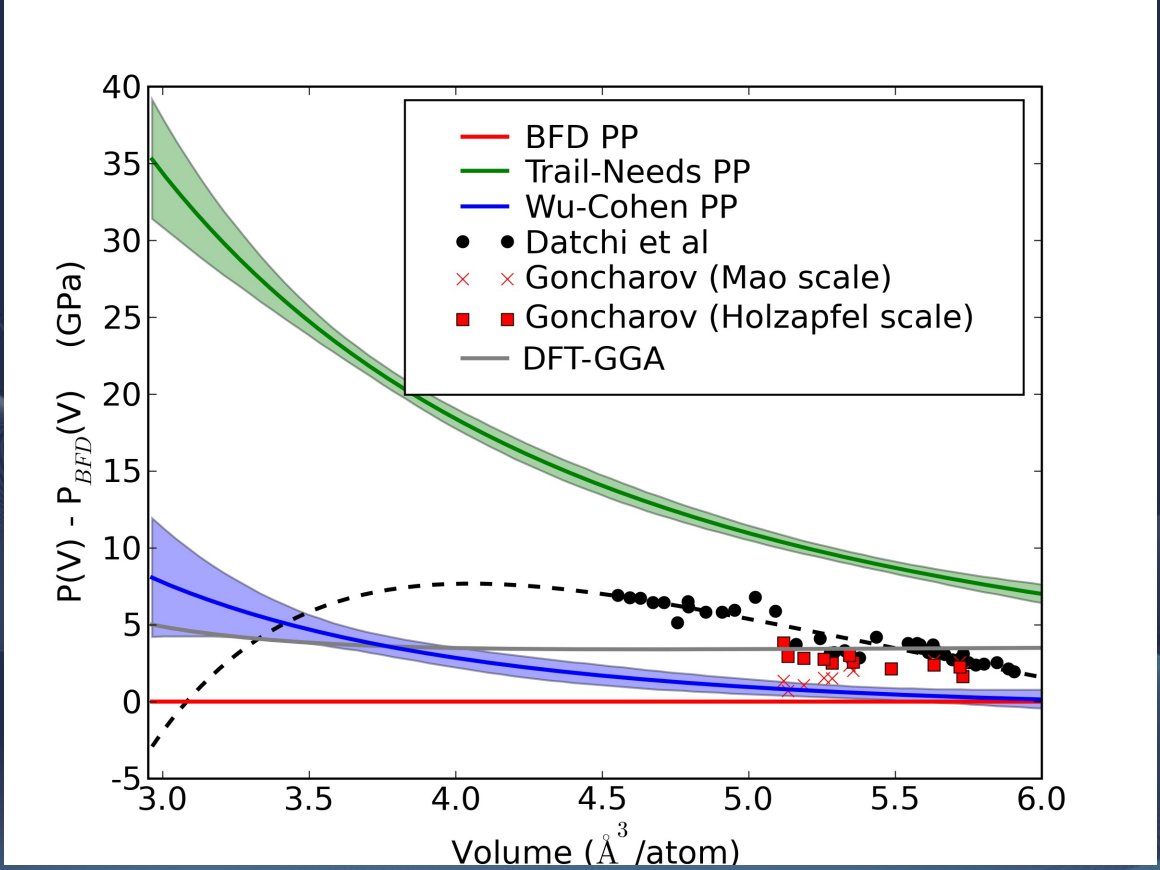
- QMC is only as good as the Hamiltonian
  - Early days: all-electron for small molecules
  - Pseudopotentials grudgingly adopted later
  - Very good work in constructing PPs for DFT and QMC
- DFT has good ways to evaluate PPs
  - All-electron calculations for small systems verify transferability
- Can we do the same for QMC?

Pseudopotential bias Example: EOS of c-BN



- 3 PPs give 3 different equations of state
- HF vs. DFT does not appear to be the difference
- No a priori way to know which is the “best”
- PPs were each constructed well, but with different theory.
- Can we do better?

Pseudopotential bias



- 3 PPs give 3 different equations of state
- HF vs. DFT does not appear to be the difference
- No a priori way to know which is the “best”
- PPs were each constructed well, but with different theory.
- Can we do better?

All-electron QMC for solids

- In DFT, we test transferability against LAPW or PAW calculations
- Can we do the same in QMC?
  - Expensive?
  - Not too bad for first-row solids
  - Still too expensive to large supercells
  - Combine PP data with AE data?
  - Where do we get our trial wave functions?

AE QMC for solids: LAPW methodology

- Inside the “muffin tins”, orbitals are expanded in spherical harmonics

$$\phi_{\mathrm{MT}}^{n\mathbf{k}\alpha}(r) = \sum_{\ell=0}^{\ell_{\max}} \sum_{m=-\ell}^{\ell} Y_\ell^m(\hat{\Omega}) \sum_G c_{\mathbf{G}}^{n\mathbf{k}} \sum_{j=1}^{M_\ell^\alpha} A_{j\ell m}^\alpha (\mathbf{G}+\mathbf{k}) f_{j\ell}^\alpha(r)$$

- Outside, use plane-waves

$$\phi_{\mathrm{int.}}^{n\mathbf{k}} = \sum_{\mathbf{G}} c_G^{n\mathbf{k}} e^{i\mathbf{k}\cdot\mathbf{r}}$$

- PWs and Y_l^ms are matched at MT boundary
- LAPW order, M, gives the order of matching
  - Order 1 matches value only (allows “kinks”)
  - Order 2 gives smooth matching
  - Matching only exact as $\ell_{\max} \to \infty$

AE QMC for solids: QMC methodology

Do sums over G and j offline:

$$\phi_{\mathrm{MT}}^{n\mathbf{k}\alpha}(r) = \sum_{\ell,m} u_{\ell m}^{n\mathbf{k}\alpha}\left(|\mathbf{r}-\mathbf{I}_\alpha|\right) Y_\ell^m\left(\hat{\Omega}\right)$$

$$u_{\ell m}^{n\mathbf{k}\alpha}(r) = \sum_G c_{\mathbf{G}}^{n\mathbf{k}} \sum_{j=1}^{M_\ell^\alpha} A_{j\ell m}^{\alpha}(\mathbf{G}+\mathbf{k}) f_{j\ell}^{\alpha}(r)$$

- Use super-LAPW (M=2) to ensure WF is smooth
- Use lmax = ~10 for good continuity!
- Optionally use blending function to ensure variational WF
- Represent u_{lm} as 1D splines (expect near origin)
- Evaluate all splines at once for efficiency
- Use 3D B-splines (BLIPS) for interstitial region

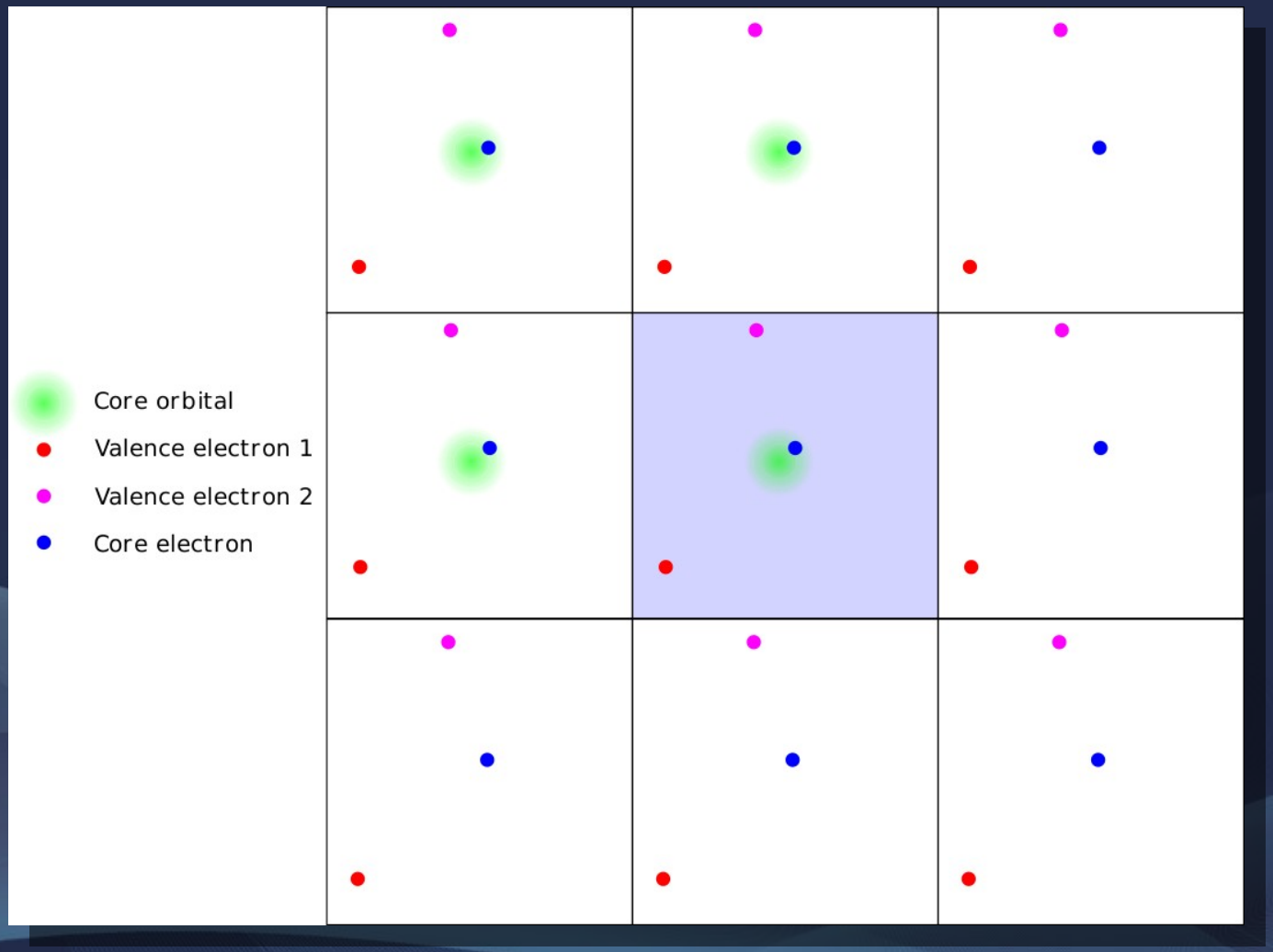
AE QMC for solids: Practical details

- Modified EXCITING (now Elk) FP-LAPW code
  - Remove relativistic corrections
  - Export orbitals in HDF5 format
- Requires smaller time-step than PP calculations
- Too expensive to do large supercell
- Can we combine AE simulation with PP data?
  - Yes, but only if the cell is big enough that core states do not contribute to the finite-size errors

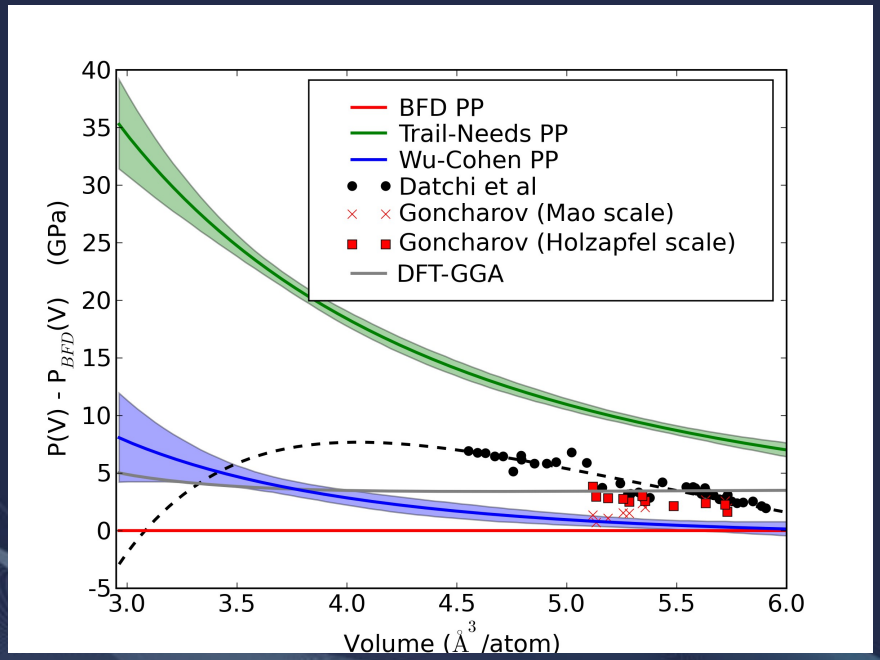
$$E = E_{64}^{\mathrm{PP}} + \left[E_{8}^{\mathrm{AE}} - E_{8}^{\mathrm{PP}}\right]$$

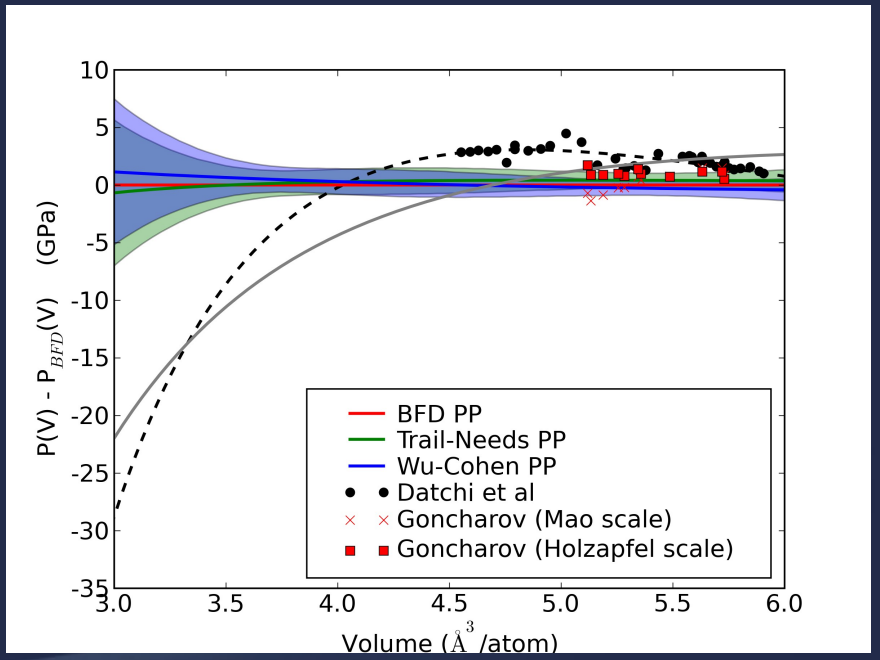
  - Core bands are flat: no momentum quantization error
  - Periodicity of exchange-correlation hole?

Combining AE and PP data



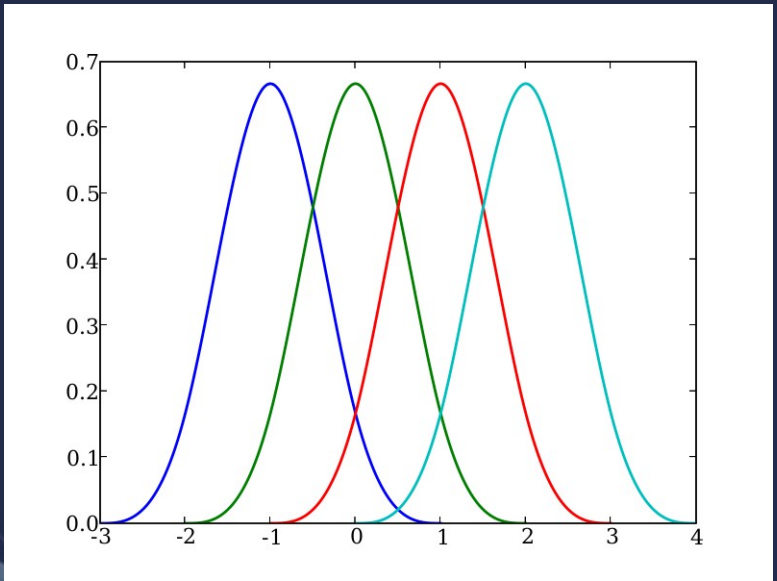
Combining AE and PP data: Results





- Once AE correction is applied, data from 3 PPs come into agreement
- Corrected EOS gives good agreement with experiment at low pressure

Mixed basis representation for orbitals 3D cubic B-spline basis (blips)



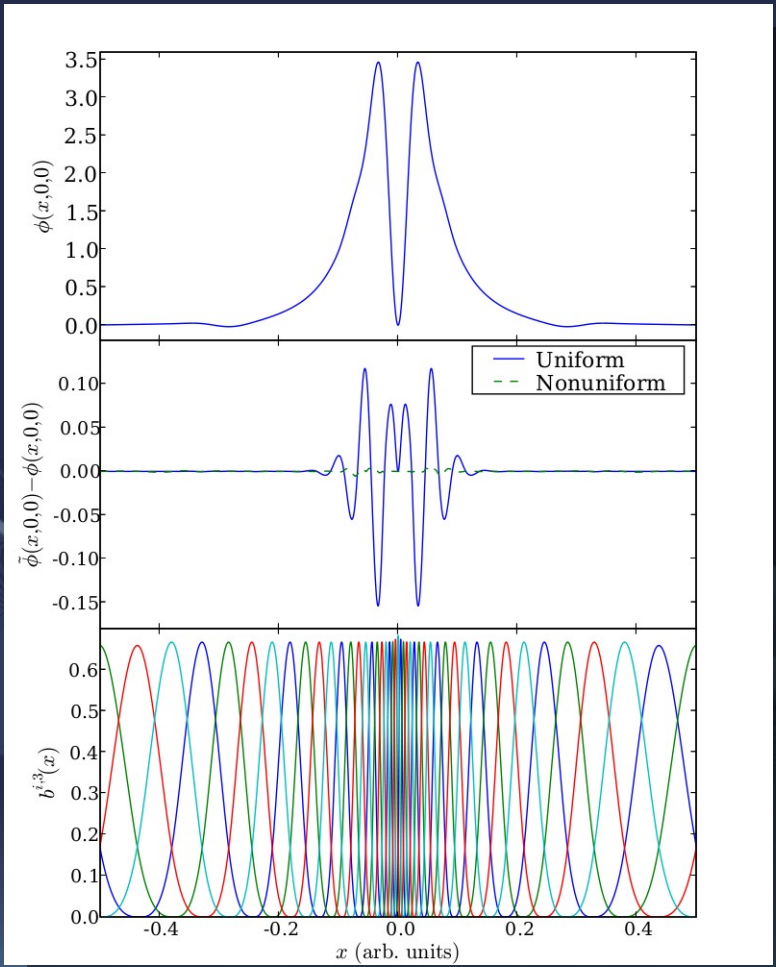
- Local basis from piecewise continuous, smooth polynomials
- Functions centered on grid points
- Strictly local: only 4 nonzero elements at any point in 1D

- Generalize to 3D with tensor product of 1D functions
  - 64 nonzero elements at any point in space
  - Very fast to evaluate
- Store 1 coefficient per grid point
- Required grid spacing determined by hardness of PPs
- Storage goes as N^2
- May have insufficient RAM for otherwise doable problems

Previous solutions to the storage problem

- Share the orbitals on an SMP node
  - Allows 8-16 GB on current clusters
  - Can get to fairly large systems for perfect crystals
  - Disordered systems require more storage
    - e.g. 32 water molecules = 11 GB
  - In QMCPACK and in latest CASINO
- Distribute the orbitals with MPI
  - Requires frequent small messages
  - Can impose a performance penalty

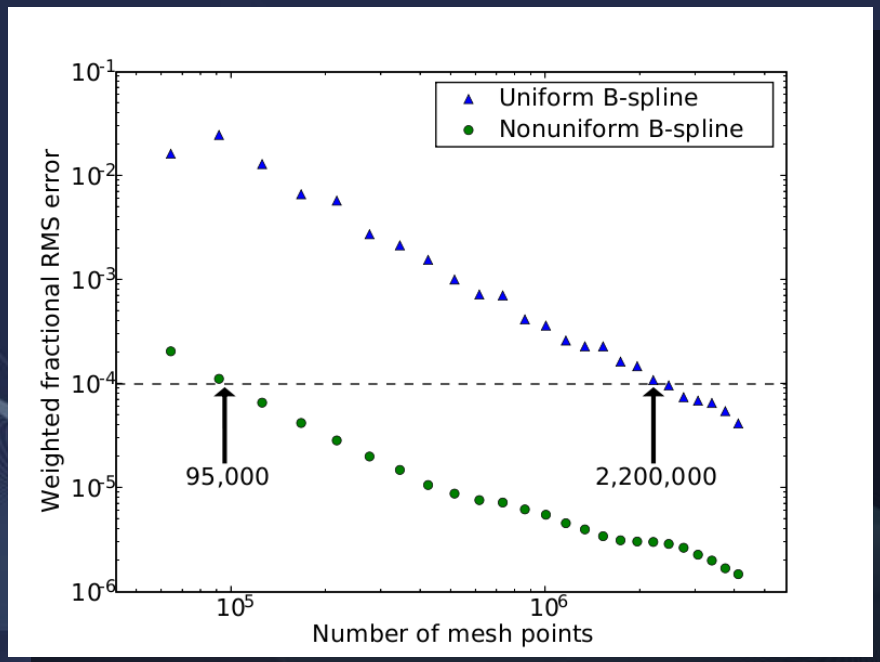
Previous solutions to the storage problem: Nonuniform splines



Fe orbital in FeO solid

- Short-wavelength oscillations are only around ions
- First, localize the orbitals around the ions
  - Alfe and Gillan
  - Reboredo and Williamson
- Use nonuniform B-spline with basis concentrated around ions
- Nonuniform error much smaller for the same number of points

Previous solutions to the storage problem: Nonuniform splines



- Appears to work for systems that can be well-localized
- Each localized orbital has a different center
- Cannot amortize the basis function cost
- Band index cannot be fastest index in memory
- Slower to evaluate that uniform, extended B-splines

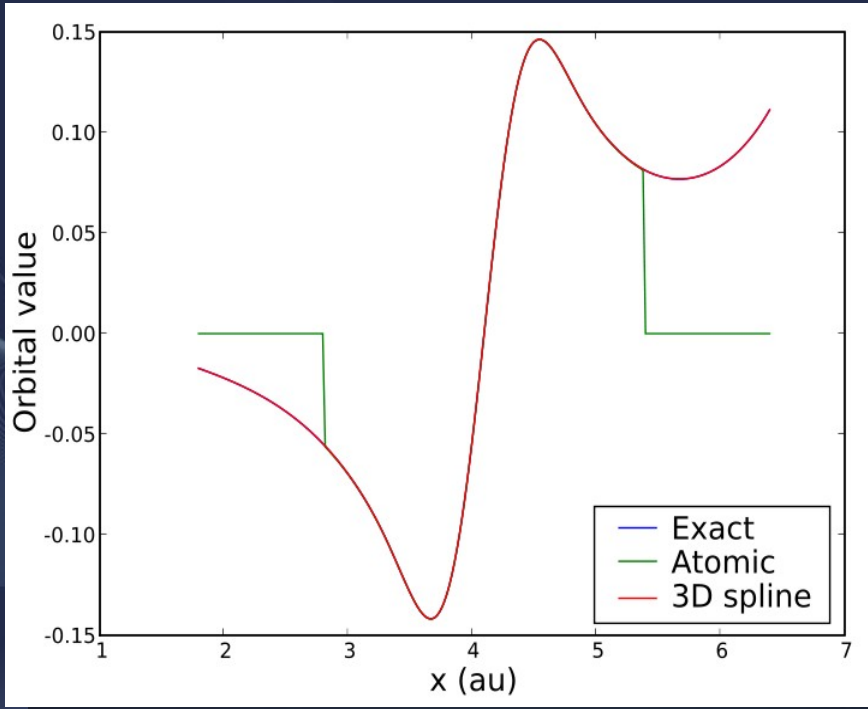
Mixed-basis representation

- Use same form inside muffin tins as with LAPW orbitals
- Projection onto spherical harmonics can be done analytically:

$$u_{\ell m}^{n\mathbf{k}\alpha}(r) = 4\pi i^{\ell} \sum_{\mathbf{G}} c_{\mathbf{G}}^{n\mathbf{k}} e^{-i(\mathbf{G}+\mathbf{k})\cdot\mathbf{I}} j_{\ell}\left(r|\mathbf{G}+\mathbf{k}|\right) \left[Y_{\ell}^{m}(\mathbf{G}\hat{+}\mathbf{k})\right]^{*}$$

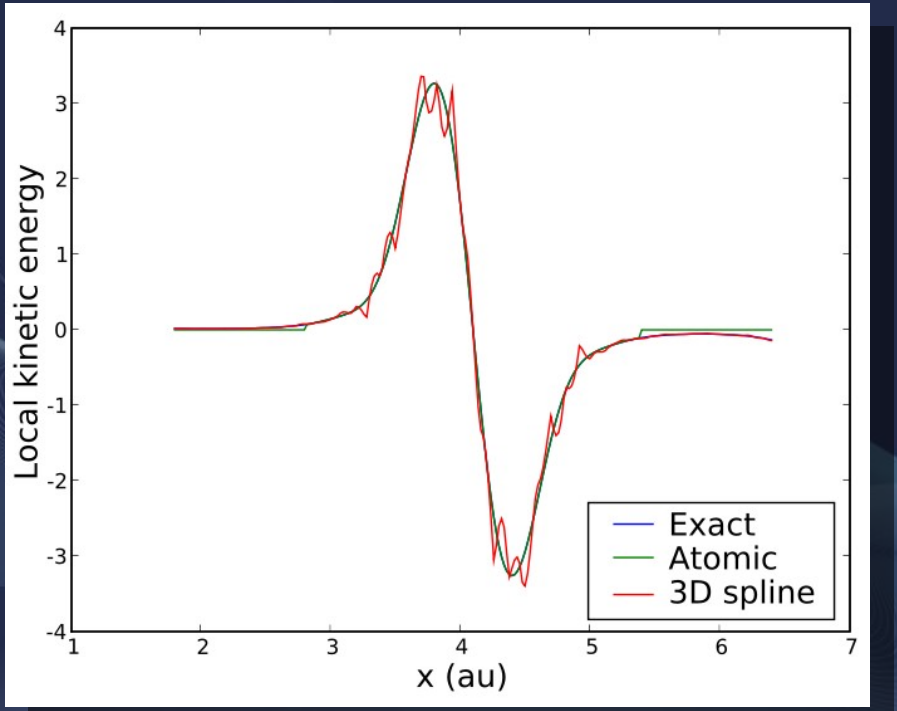
- Must choose radii and l_{max}
- For FeO, use l_{max}=5
  - Yields 144 coefficients reads per evaluation
  - More than 3D B-spline (64),
  - Radial splines only evaluated once per PP quadrature
  - Runs at about the same speed as standard approach

Mixed basis representation: FeO example



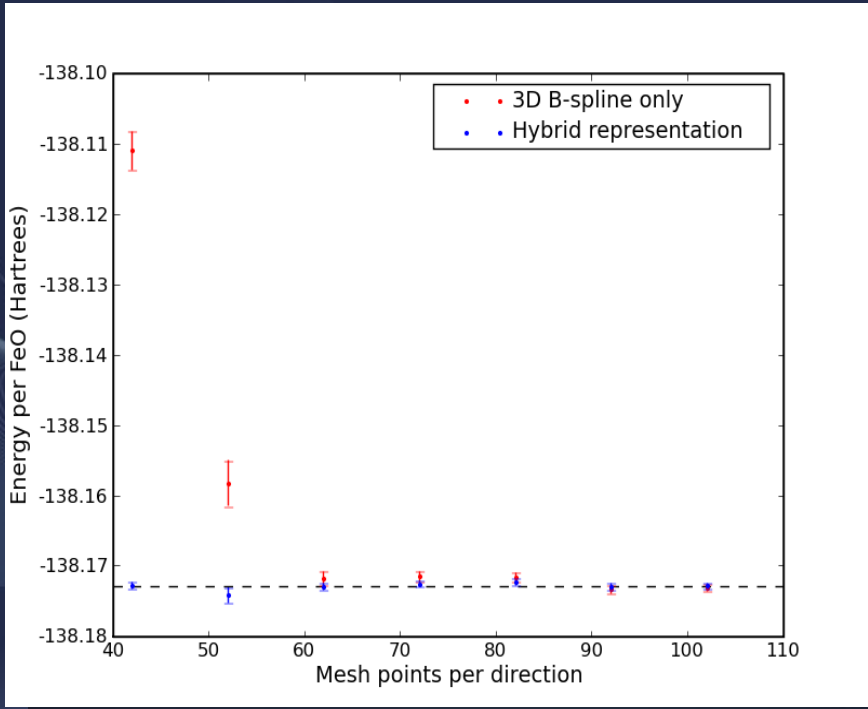
- Near ions:
  - 3D splines handle function value well,
  - but not the Laplacian.
  - Laplacian is continuous, but not smooth.
  - Atomic orbitals handle both well with a fine 1D grid

Mixed basis representation: FeO example



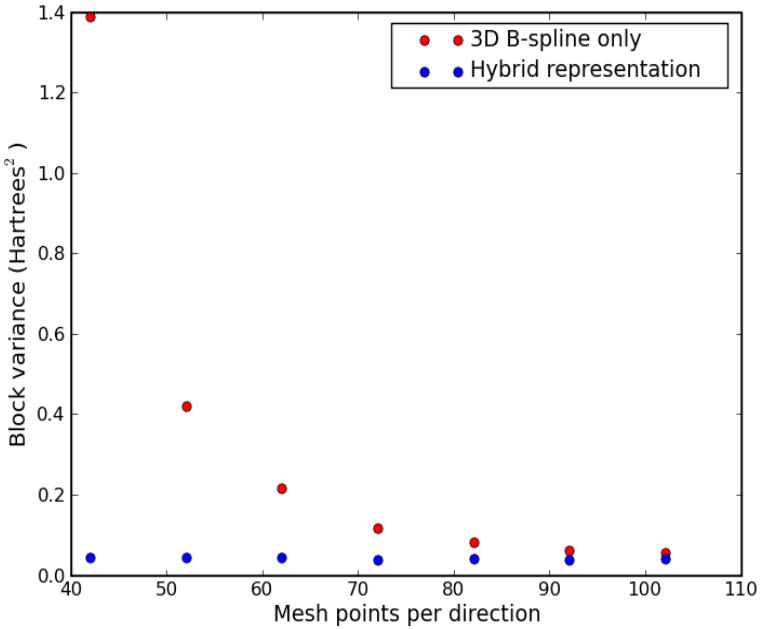
- Near ions:
  - 3D splines handle function value well,
  - but not the Laplacian
  - Laplacian is continuous, but not smooth
  - Atomic orbitals handle both well with a fine 1D grid

Mixed basis representation: Energy



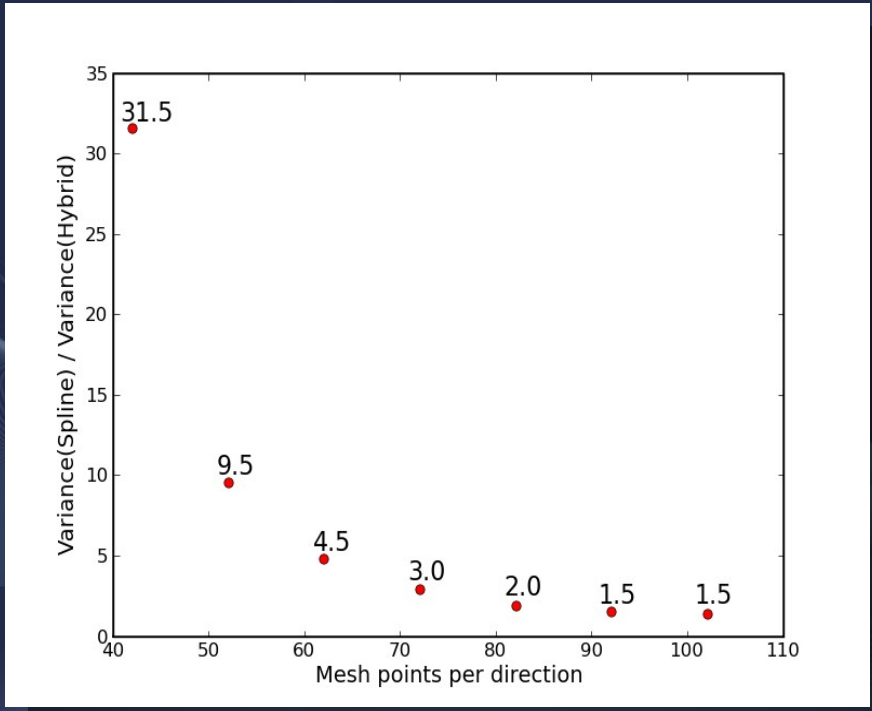
- DMC energy
  - 3D B-spline okay down to ~60 points per dir.
- DMC variance
  - Always lower for hybrid representation
  - Many times lower for very coarse grids

Mixed basis representation: Variance



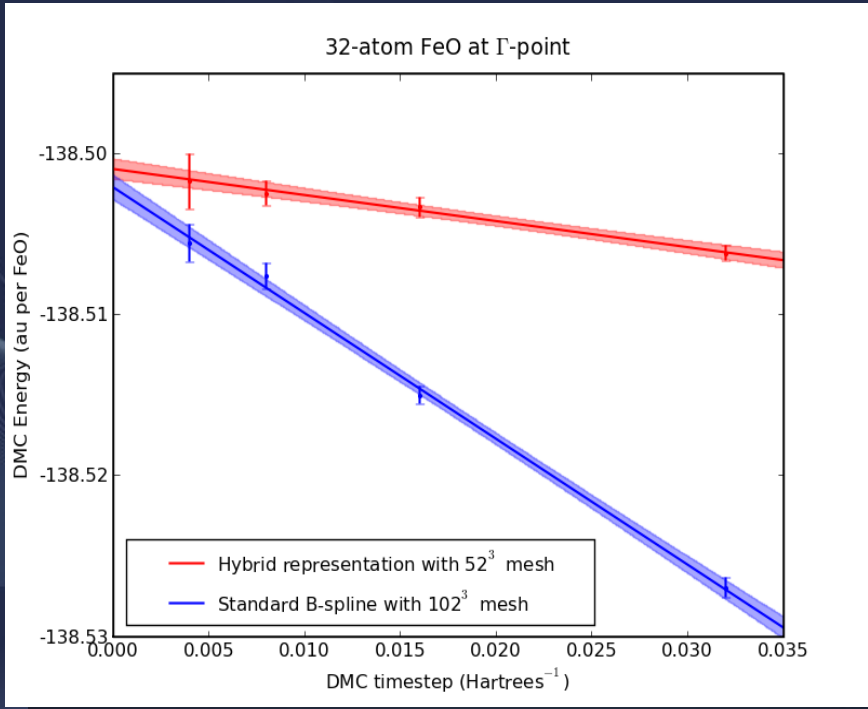
- DMC energy
  - 3D B-spline okay down to ~60 points per dir.
- DMC variance
  - Always lower for hybrid representation
  - Many times lower for very coarse grids

Mixed basis representation: Variance ratio



- DMC energy
  - 3D B-spline okay down to ~60 points per dir.
- DMC variance
  - Always lower for hybrid representation
  - Many times lower for very coarse grids

Mixed basis representation: Time step error

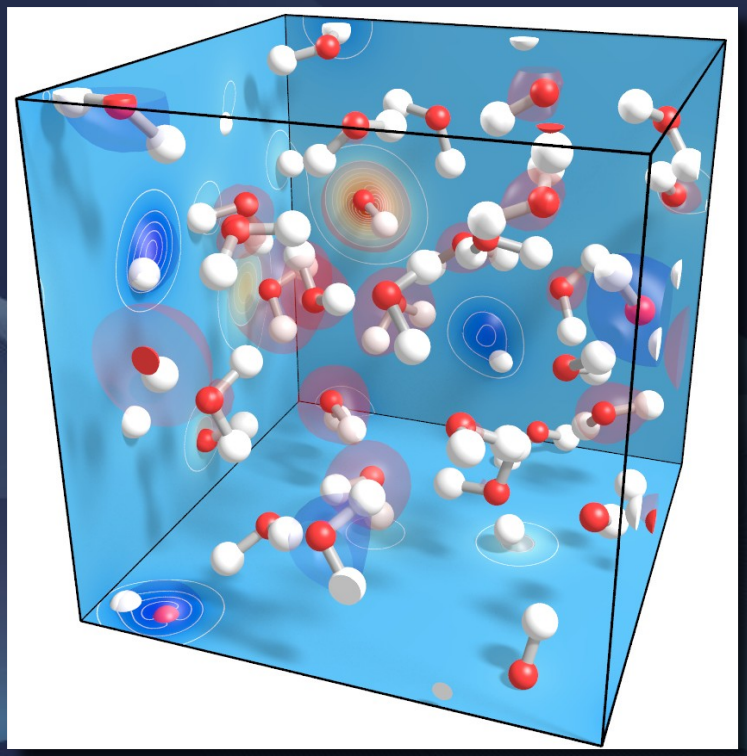


- Lower variance of local energy also gives smaller time step error
- Hybrid representation with coarse mesh gives lower time step error than B-spline only with fine mesh

Mixed basis summary

- With mixed basis, we appear to get the same dt->0 answer, in about the same time, with lower variance, with about 1/8 the memory requirement
- Very effective for hard PPs (e.g. those with semicore states)
- Not as effective with light, molecular systems (e.g. water)

QMC using graphics processors (GPUs)



Why are GPU's interesting?

- CPU
  - 40-60 GFLOPS DP
  - 80-120 GFLOPS SP
  - 10-30 GB/s memory bandwidth
  - Optimized for single-thread performance
- GPU
  - 85 GFLOPS DP
  - 1 TFLOP SP
  - 100-150 GB/s memory bandwidth
  - Technology on steeper trajectory than CPUs
  - Optimized for throughput

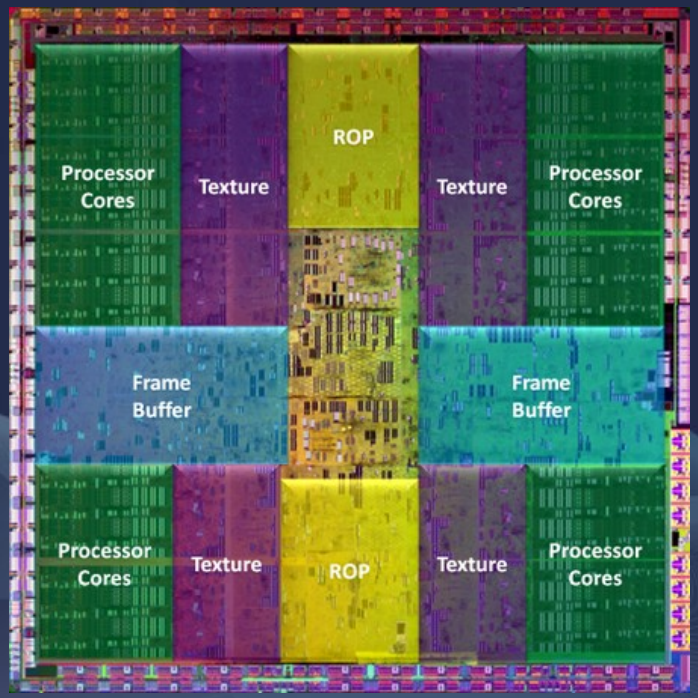
Why are GPU's interesting?

- CPU
  - Hide memory latency with large cache
  - A lot of logic for:
    - Instruction reorder
    - Branch prediction
    - Prefetching
  - Relatively few FPUs
- GPU
  - Hide memory latency through simultaneous multithreading (SMT)
  - In-order execution - little control logic
  - Many more FPUs
  - Very wide memory bus (e.g. 512 bit)

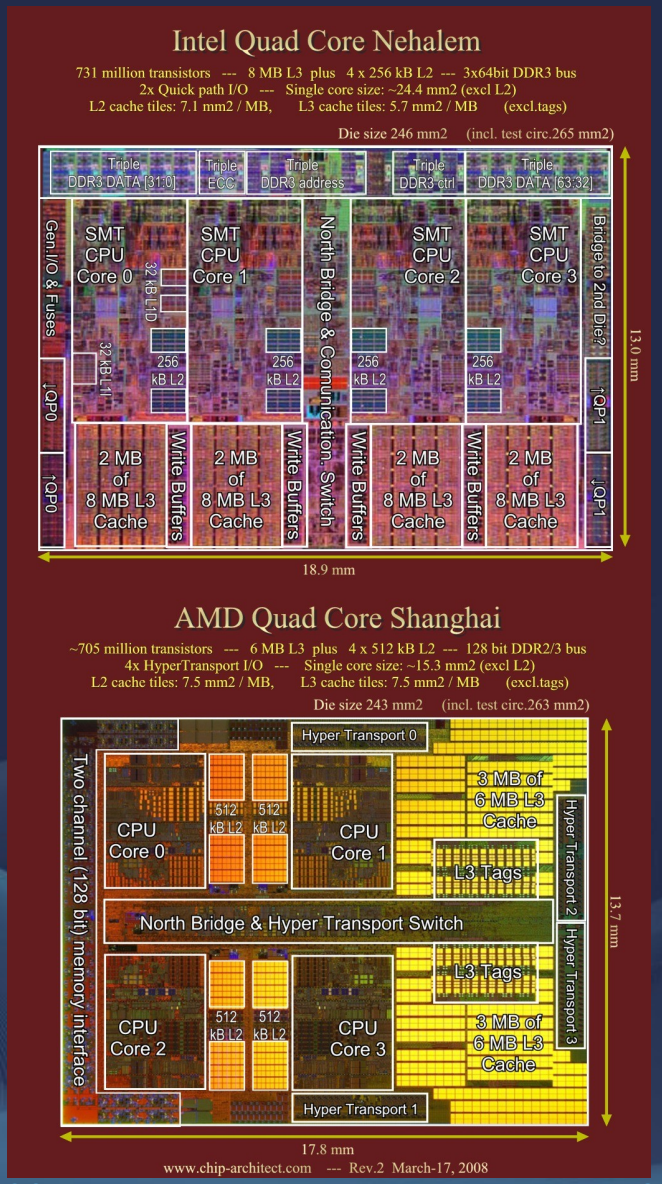
Market drivers

- GAMES!
  - Gamers buy a new card each year
  - Always want better performance
  - FLOPS/Bandwidth makes better graphics
  - Game designers want flexibility of GPGPU
- Media
  - Content producers using GPUs
  - E.g. Adobe CS4
- Finance
- Medical
- Oil
- HPC for science

NVIDIA G200

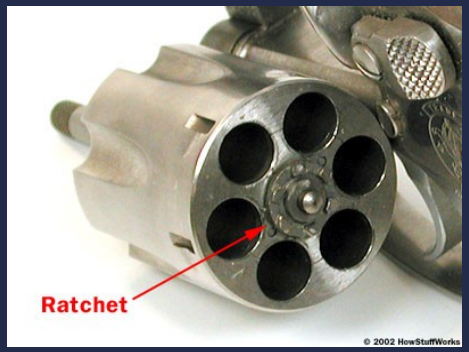


1.4 billion transistors



And now for something completely different...

- GPU architecture
  - Is it a vector processor?
  - Or is it a barrel processor?
  - Or is it a multicore processor?
- YES!



Barrel processors

- Alternative solution to memory latency problem
- Send many independent threads to processor
- Processor executes instructions in a thread sequentially until it blocks
- Whenever a thread blocks, move on to the next thread
- With enough threads in the barrel, memory latency can be completely hidden
- Requires many independent tasks which can run in parallel

Vector processors

- Execution logic is expensive
- Perform the same instruction on multiple data (SIMD)
- x86 has short vector instructions (SSE)
  - 4 for single prec.; 2 for double precision
- GPU variation: SIMT
  - Each thread has its own data
  - Each thread executes same inst., but writing result can be masked with conditionals
  - Vector length is variable, but multiple of 32

Multicore processors

- GPUs are separated into independent units
  - Data can be exchanged on the same unit for a given kernel (shared memory)
  - Data can be exchanged between units through DRAM
  - Data can be exchanged with CPU through API calls (slow!)
  - NVIDIA G200: 30 processor cores
- Need hundreds of independent threads to keep GPU busy

GPU memory bandwidth

- NVIDIA Tesla: 100 GB/s
- AMD 4870: 115 GB/s
- How?
  - Very fast DRAM (GDDR3-GDDR5)
  - Very wide bus (256-512 bits)
- Requires very wide, aligned reads
  - Read 16 floats or 8 doubles at time
  - Reads must be sequential and 64-byte aligned

GPU programming

- “Host” (CPU) does I/O, complicated processing, etc.
- GPU executes “kernels”: small functions to do one task many times
  - Limited number of registers and shared memory
  - All cores execute the same kernel
  - Single precision is very fast
  - Double precision is faster than CPU, but much slower than single precision
- Data is passed between CPU and GPU memory across PCI bus with API calls
  - Currently 1.5 – 8 GB/s: SLOW!

CUDA

- NVIDIA's extensions to C/C++ to allow GPU execution
- Mix CPU and GPU code in the same file
- A few language extensions for device code
- A few API calls for memory allocation, data transfer, etc.
- Challenges:
  - Debugging: sychronoization bugs,
    no “printf” on GPU!
  - Memory layout and access patterns!
  - Pointer book-keeping
  - Exposing parallelism
- Very good forum (CUDA zone)

```
__global__ void
two_body_sum_kernel
(float R[], int e1_first, int e1_last, int e2_first, int e2_last, float spline_coefs[],
 int numCoefs, float rMax, float lattice[], float latticeInv[], float sum[])
{
  // Some setup goes here...
  int N1 = e1_last - e1_first + 1;
  int N2 = e2_last - e2_first + 1;
  int NB1 = (N1+BS-1)//BS;
  int NB2 = (N2+BS-1)/BS;

  float mysum = (float)0.0;
  for (int b1=0; b1 < NB1; b1++) {
    // Load block of positions from global memory
    for (int i=0; i<3; i++)
      if ((3*b1+i)*BS + tid < 3*N1)
        r1[0][i*BS + tid] = myR[3*e1_first + (3*b1+i)*BS + tid];
    __syncthreads();
    int ptcl1 = e1_first+b1*BS + tid;
    for (int b2=0; b2 < NB2; b2++) {
      // Load block of positions from global memory
      for (int i=0; i<3; i++)
        if ((3*b2+i)*BS + tid < 3*N2)
          r2[0][i*BS + tid] = myR[3*e2_first + (3*b2+i)*BS + tid];
      __syncthreads();
      // Now, loop over particles
      int end = (b2+1)*BS < N2 ? BS : N2-b2*BS;
      for (int j=0; j<end; j++) {
        int ptcl2 = e2_first + b2*BS+j;
        float dx, dy, dz;
        dx = r2[j][0] - r1[tid][0];
        dy = r2[j][1] - r1[tid][1];
        dz = r2[j][2] - r1[tid][2];
        float dist = min_dist(dx, dy, dz, L, Linv);
        if (ptcl1 != ptcl2 && (ptcl1 < (N1+e1_first) ) && (ptcl2 < (N2+e2_first)))
          mysum += eval_1d_spline (dist, rMax, drInv, A, coefs);
      }
      synchthreads();
    }
  }
  // Sum result over threads
  __shared__ float shared_sum[BS];
  shared_sum[tid] = mysum;
  __syncthreads();
  for (int s=BS>>1; s>0; s >>=1) {
    if (tid < s)
      shared_sum[tid] += shared_sum[tid+s];
    __syncthreads();
  }
  // Avoid double-counting
  float factor = (e1_first == e2_first) ? 0.5 : 1.0;
  if (tid==0)
    sum[blockIdx.x] += factor*shared_sum[0];
}
```

QMCPACK

- Developed at UIUC
  - Principal developer:
    - Jeongnim Kim
- C++ code with extensive template optimizations
- Hybrid OpenMP/MPI parallelization model
- Uses XML and HDF5 standards
- Object-oriented design for extensibility
- Open source and freely available

- Orbitals
  - Gaussian, STOs, PW, B-spline, mixed-basis, LAPW
  - Real or complex WFs
- VMC, DMC, and RMC
- Standard variance and energy opt. methods
- Scaled to 100k cores
- Works with ABINIT, Pwscf, Qbox, Gaussian, etc.

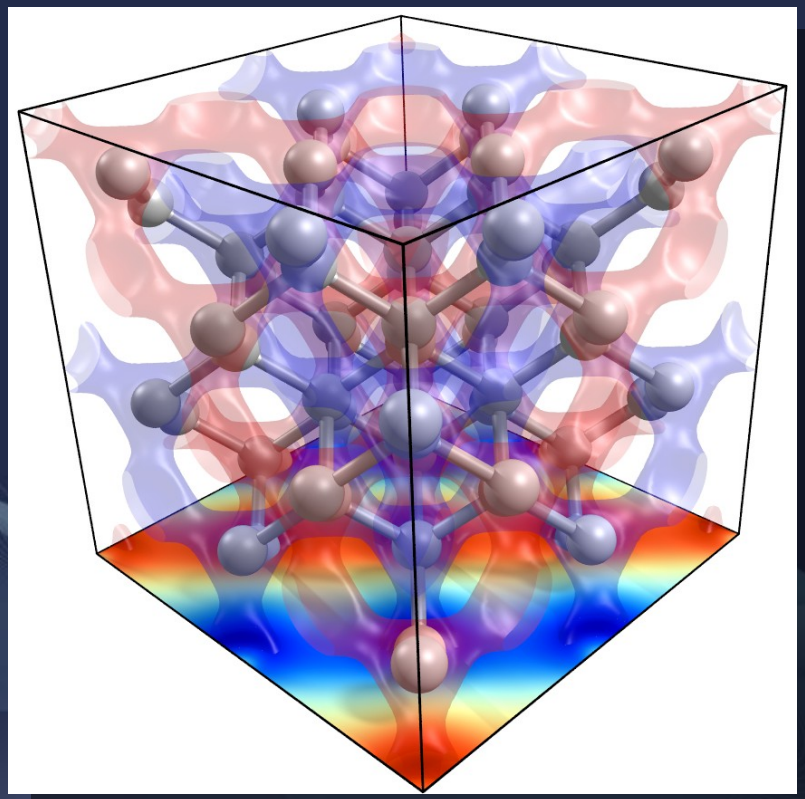
QMC on a GPU

- Advantages:
  - Walkers provide plenty of work for a GPU kernel
- Challenges:
  - Orbital storage: orbitals must fit in 4 GB GPU memory. Distribution on multiple cards could be expensive.
  - Many kernels
    - Orbital evaluation
    - Determinant evaluation, ratios, and update
    - Jastrow evaluation and ratios
    - Coulomb interaction, Ewald sums
    - Pseudopotential ratios
    - Other observables

QMCPACK on GPU

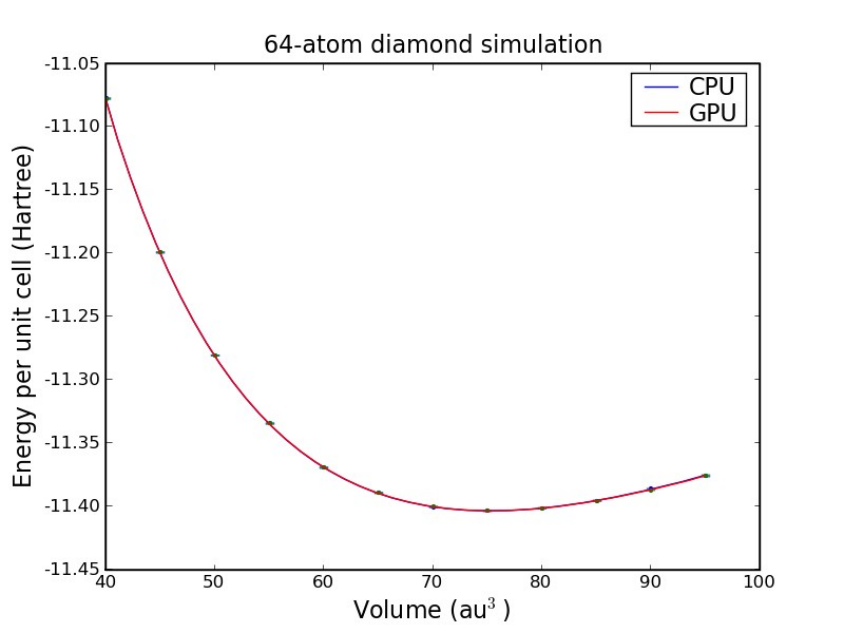
- Reimplemented VMC & DMC drivers to be walker-parallel, but still particle by particle
- Implemented walker-parallel kernels for:
  - B-spline orbital evaluation
  - Determinant updates, ratios, inverses, gradients, laplacians, etc.
  - One-body and two-body B-spline Jastrows
  - Periodic coulomb interaction
  - Nonlocal pseudopotentials
- Single precision for everything but occasional inverse recomputation
- Use texture units for 1D potential interpolation
- Keep all walker-related data in GPU memory
- CPU proposes, accepts/rejects moves; branches; collects averages and does I/O.

Test: 64-atom diamond



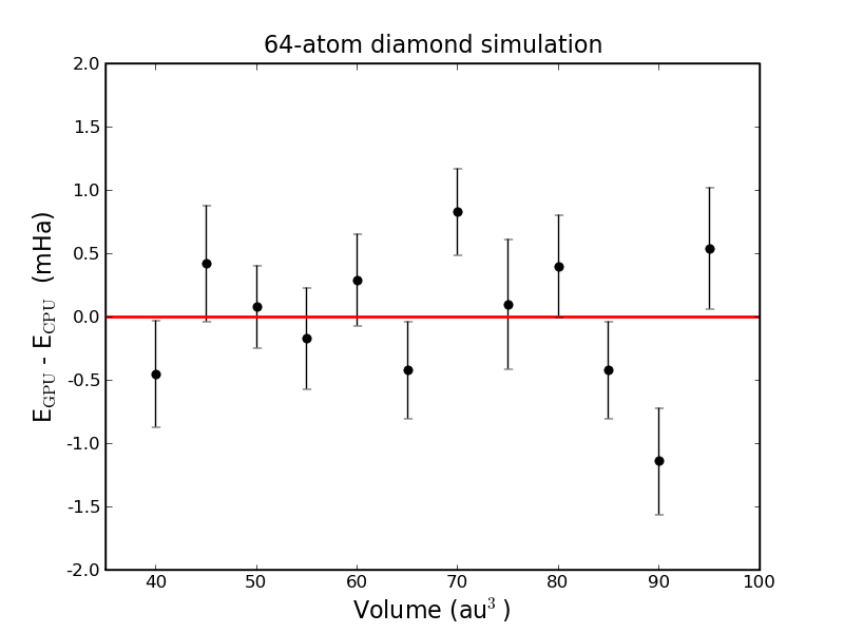
- 64-atom DMC simulation of diamond
- Burkaztki *et al*. PP
- LDA orbitals
- 1+2-body Jastrow
  - Optimized on CPU
- dt=0.01/Hartree
- Simple EOS calculation
  - No phonons
  - No finite-size corrections

CPU vs. GPU: Accuracy



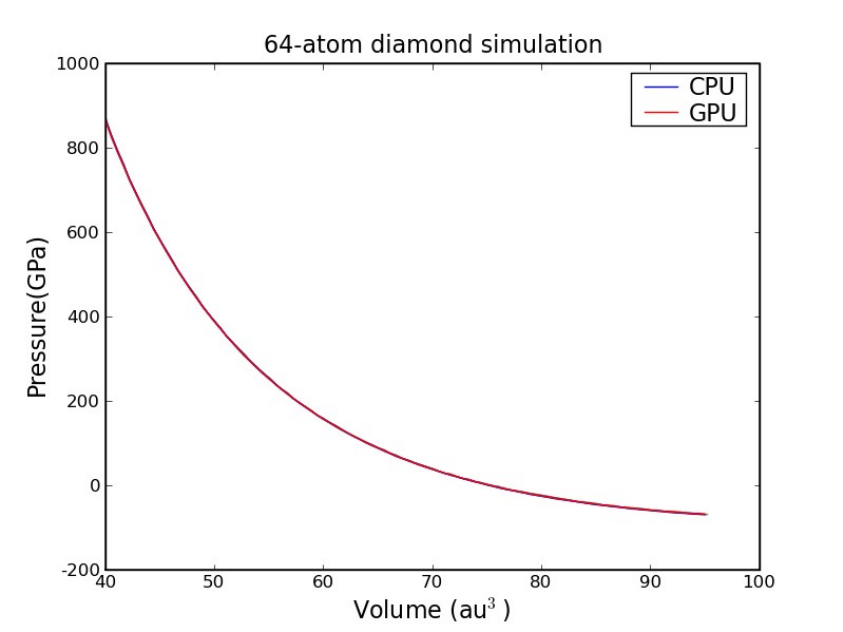
- CPU and GPU give same results within statistical error
- GPU uses single precision for all but recomputing inverse

CPU vs. GPU: Accuracy



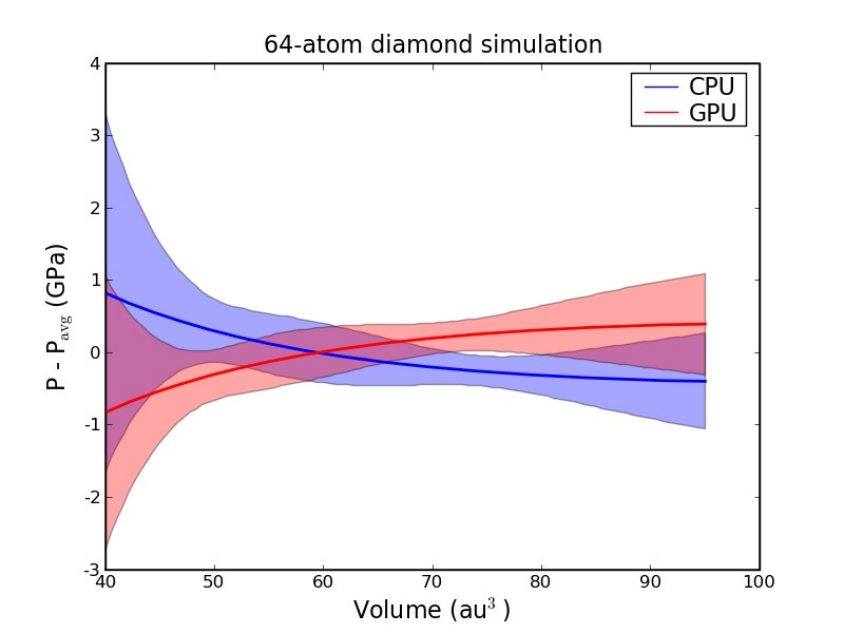
- CPU and GPU give same results within statistical error
- GPU uses single precision for all but recomputing inverse

CPU vs. GPU: Accuracy



- CPU and GPU give same results within statistical error
- GPU uses single precision for all but recomputing inverse

CPU vs. GPU: Accuracy



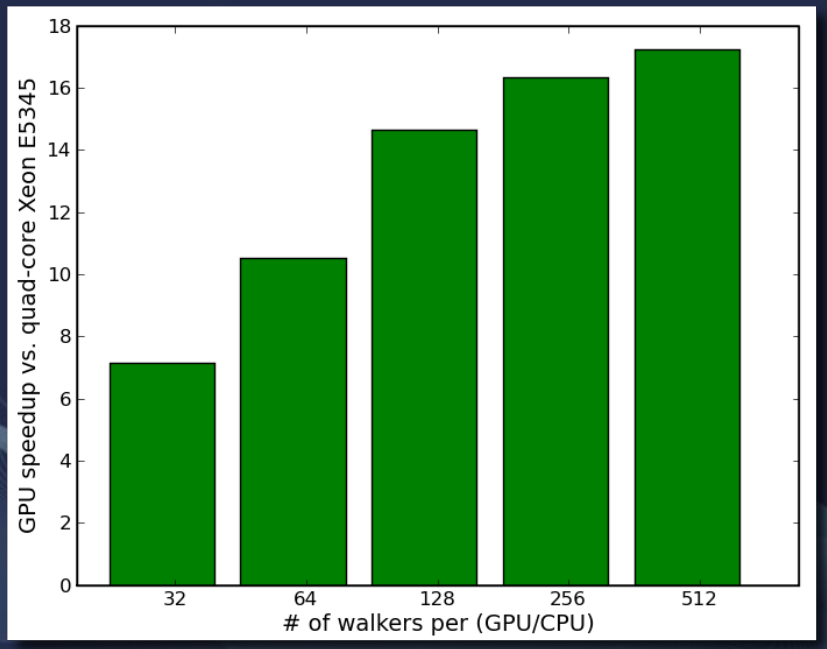
- CPU and GPU give same results within statistical error
- GPU uses single precision for all but recomputing inverse

CPU vs. GPU: Speed

- CPU run
  - Cray XT5 (Kraken)
  - 3072 Opteron cores
  - 1280 walkers per V
  - Double precision
  - Block time:
    - 21.0 seconds
- CPU + GPU run
  - Dell cluster (Lincoln)
  - 48 G200 GPUs (+48 Xeon cores)
  - 1280 walkers per V
  - Mixed precision
  - Block time:
    - 24.4 seconds

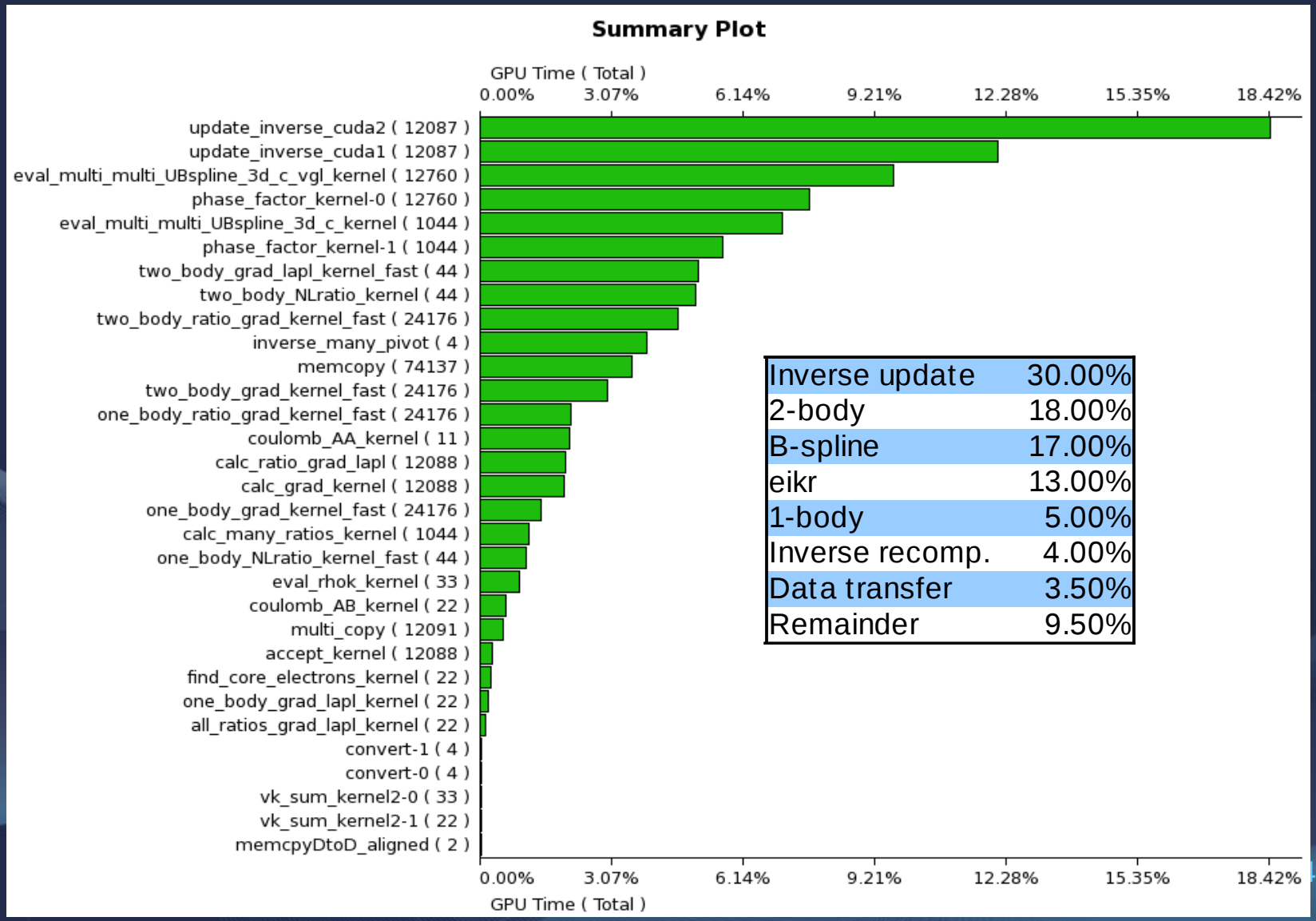
1 G200 GPU = ~14 quad-core Opterons

CPU vs. GPU: Speed



- 32-atom MnO simulation
- We need many walkers per GPU to saturate speed
- May run out of GPU memory first
- May require long wall clock time

GPU kernel breakdown



Inverse update	30.00%
2-body	18.00%
B-spline	17.00%
eikr	13.00%
1-body	5.00%
Inverse recomp.	4.00%
Data transfer	3.50%
Remainder	9.50%

Future work

- WF optimization
  - Derivatives w.r.t. WF parameters
- Larger systems?
- Generalize
  - Nonperiodic BC
- More WFs
  - Hybrid orbitals
  - Atomic bases
  - 3-body Jastrows
  - Multideterminant?
- More estimators
  - MPC
  - Pair correlation

Future speculation

- NVIDIA
  - Continue to support CUDA/OpenCL
  - Next gen. (GT300)
    - Due 4th quarter
    - About 2.5x faster?
    - Faster double precision
  - FORTRAN
    - PGI GPU extensions
- AMD
  - OpenCL out soon
  - Next gen (RV870)
    - Due October?
    - 2.1 TFLOPS (SP)

Future speculation: Intel Larrabee

- 32-48 Pentium-class cores
  - In order execution
  - Vector unit
    - 16-wide single prec.
    - 8-wide double prec.
    - Vector complete
  - Traditional cache-coherent arch.
- Will be supported by Intel compiler
  - Should be able to vectorize well
  - Easier than CUDA/OpenCL?
- First half of 2010
- Performance for QMC?

The End

Memory performance: Bandwidth

- 50 GFLOPS / (10GB/s / 8 bytes/double) = 40 FLOPs per load/store
- If algorithm has low compute/fetch, you get bad performance
- CPUs again use cache to increase effective bandwidth

CPU vs. GPU: Accuracy 32-atom MnO VMC no Jastrow

Energy	CPU	Error	GPU	Error	Difference	Diff/sigma
Kinetic	61.31310	0.00180	61.31057	0.00081	0.00253	1.3
e-e	17.68930	0.00095	17.68698	0.00049	0.00232	2.2
Local PP	-105.98920	0.00230	-105.58310	0.00110	-0.00600	2.4
Nonlocal PP	-8.20833	0.00072	8.20878	0.00042	0.00045	0.5
Total	-118.09989	0.00027	-118.09926	0.00015	-0.00063	2.0

CPU vs. GPU: Speed

- Jaguar:
  - 6.71 walker “steps” per proc sec.
- Lincoln
  - 67.3 walker “steps” per GPU sec.

OpenCL Working Group

- **Diverse industry participation**
  - Processor vendors, system OEMs, middleware vendors, application developers
- **Open, vendor-neutral royalty-free standard**
  - Developed under standard Khronos IP framework
- **Apple will use OpenCL**
  - Performance-enhancing technology will be used in Mac OS X Snow Leopard

CPU vs. GPU: Speed

- CPU
  - 8 Abe nodes (64 cores)
  - Double precision
  - 8 hours
  - 8.63 million “moves”
  - 18.73 moves/(proc sec)
- GPU
  - 1 NVIDA GTX260
    - $250
    - 896 MB
  - 320 walkers
  - Mixed prec.
  - 9.7 hours
  - 8.00 million moves
  - 228.8 moves/(proc sec)

12x faster than quad-core Xeon

Why do QMC on a GPU?

- Walker parallelism is ideally suited for massively multithreaded model
- Decent speedups can be attained
- The future of HPC looks like it will include GPUs as a big part
- It's not nearly as hard as it used to be
- Looks like it will be easier in the future

Why not do QMC on a GPU?

- No single-point of entry, many kernels need to be written and tuned
- Need to rethink code structure
- No REALLY big machines exist yet
- Debugging can be a challenge
- Making predictions is hard, especially about the future: I could be wrong.

Memory performance: Latency

- DRAM is slow
  - A fetch from RAM can take 100-200 clock cycles.
- CPUs hide latency with cache
  - 3 levels:
    - L1: 64 KB
    - L2: 256 KB
    - 8 MB L3
- It's up to the programmer to make good use of cache, but not always possible