Fully accelerating quantum Monte Carlo simulations of real materials on GPU clusters

Ken Esler¹
Jeongnim Kim
David Ceperley

National Center for Supercomputing
University of Illinois at Urbana-Champaign

¹To be Stone Ridge Technology as of 7/19/10
“The underlying physical laws necessary for a large part of physics and the whole of chemistry are thus completely known, and the difficulty is only that the exact applications of these laws lead to equations much too complicated to be soluble.”

– Paul Dirac, 1929
Dirac's Challenge

\[-\frac{1}{2} \sum_i \nabla_i^2 \Psi(r_1, r_2, \ldots) + \left[ \sum_{i \neq j} \frac{1}{|r_i - r_j|} + \sum_{i,k} \frac{-Z_k}{|r_i - I_k|} - E \right] \Psi(r_1, r_2, \ldots) = 0\]

- For N electrons, \( \Psi \) is a function in 3N dimensions
- Exact solution impossible if N > ~5
- Theoretical physics and chemists try to find high-accuracy approximate solutions
Example Systems
Quantum Monte Carlo

- Address high-dimensionality through stochastic sampling
- Exact for bosons, very accurate for fermions (e.g. electrons)
- Computation scales as $N^3$ to $N^4$
- Excellent parallel scalability
- Several methods
  - **Variational Monte Carlo (VMC)**
  - **Diffusion Monte Carlo (DMC)**
    - Path integral Monte Carlo (nonzero temperature)
    - Auxiliary field Monte Carlo
  - Lattice methods
    - Solve model problems
Variational Monte Carlo

- Write down a parametrized form for $\Psi(R)$
  - $R = \{r_1, r_2, r_3, \ldots \}$

- Sample distribution $P(R) = |\Psi(R)|^2$ with Metropolis Monte Carlo
  - Propose move $r_i \rightarrow r'_i$
  - Accept or reject with probability $\propto \left| \frac{\Psi(R')}{\Psi(R)} \right|^2$

- Average energy over distribution, $\langle E \rangle$

- Minimize $\langle E \rangle$ with respect to parameters of $\Psi$
Diffusion Monte Carlo

- Start with optimized $\Psi$ from VMC
- Walker $= R$, point in $3N$-dimensional space
  - i.e. the positions of all electrons
- Start with population of $\sim 1000$ walkers
- Algorithm
  - **Drift/diffuse**: try to move each electron in each walker in sequence
  - **Branch**: make $M$ copies of each walker, where
    \[
    M \propto \exp \{ \tau [E_T - E(R)] \}
    \]
DMC Output: Total Energy

![Graph showing DMC Output: Total Energy with data points and trend lines.](image-url)
Why QMC on GPUs?

- QMC consumes many cycles
  - e.g. ~70 million hours on Jaguar last year
  - 8k cores 24/7
- It affords many levels for parallelization
  - "Natural" Monte Carlo parallelism (walkers)
  - Electrons, orbitals, ions, etc.
- MPI communication requirements are low
  - Primarily for load balancing
- With GPUs
  - Small problems are doable on a workstation
  - Large problems are much cheaper
Why not QMC on GPUs?

- Many kernels to port
  - No single major kernel
  - Wave functions take many forms
  - Subset of functionality:
    - ~100 CUDA kernels, ~10k lines of kernel code
- Algorithms and data need restructuring
  - Need to utilize more parallelism than on CPUs
- Debugging stochastic methods is hard
- No libraries for the kernels we need
QMCPACK

- Principal author: Jeongnim Kim
- Open source C++ code (NCSA license)
- Hybrid OpenMP/MPI programming model
  - Scaled to > 200k cores with > 90% efficiency
- Heavy use of template meta-programming
  - At least as fast as any comparable code
- Object-oriented style for extensibility
Restructuring Algorithms

- Do all walkers in parallel
  - Helps provide enough threads
  - Many threads per walker
  - Need ~256 walkers/GPU to maximize throughput

- Simple in theory, not in practice

- Rewrite all computational kernels

```plaintext
for generation = 1 \cdots N_{MC} do
  for walker = 1 \cdots N_w do
    let \{r_1 \ldots r_N\} = R
    for particle i = 1 \cdots N do
      set \mathbf{r}_i' = \mathbf{r}_i + \delta
      let \{r_1' \ldots r_N'\} = R'
      ratio \rho = \Psi_T(\mathbf{R}')/\Psi_T(\mathbf{R})
      if \mathbf{r} \rightarrow \mathbf{r}' is accepted then
        update inverse matrix, distance tables, etc.
      end if
    end for
  end for
end for

Compute local energy, \[ E_L = \hat{H}(\Psi_T(\mathbf{R})/\Psi_T(\mathbf{R})) \]
Kinetic energy = \(-\frac{1}{2} \nabla^2 \Psi_T(\mathbf{R})\)
Electron-electron energy (Coulomb)
Pseudopotential energy
Reweight and branch walkers
Update \[ E_T \]
if generation > \text{\textbar} N_{eq} then
  Collect properties
end if
end for
```
DMC simulation
DMC simulation
DMC simulation
DMC simulation
DMC simulation
DMC simulation
DMC simulation
Computational Kernels

- Wave function and derivatives:
  - Ratio, gradient, and Laplacian
    \[
    \frac{\psi(R')}{\psi(R)}', \quad \frac{\nabla \psi(R')}{\psi(R)}', \quad \frac{\nabla^2 \psi(R')}{\psi(R)}
    \]
  - Kernels needed for determinants, \( J_1 \), and \( J_2 \)

- Potential energy:
  - Periodic coulomb interaction
    - Part in real-space, part in frequency space
  - Pseudopotential interaction
template<typename T, int BS, int LMAX> __global__ void
evaluateHybridPolyToReal_kernel (HybridJobType *job_types, T *rhats,
   T **Ylm, float **dR, float **dYlm_dTheta, float **dYlm_dphi,
   AtomicOrbitalCuda<T> *orbitals, HybridDataFloat *data,
   T *k_reduced, T **vals, T **grad_lapl, int row_stride, int N)

template<typename T, int BS, int LMAX> __global__ void
evaluateHybridSplineComplexToReal_kernel
   (HybridJobType *job_types, T **Ylm_complex, AtomicOrbitalCuda<T> *orbitals,
   HybridDataFloat *data, T *k_reduced, int *make2copies, T **vals, int N)

template<typename T, int BS, int LMAX> __global__ void
evaluateHybridSplineComplexToReal_knl_NLP_KERNEL
   (HybridJobType *job_types, T **Ylm_complex, int numQuad,
   AtomicOrbitalCuda<T> *orbitals, HybridDataFloat *data,
   T *k_reduced, int *make2copies, T **vals, int N)

template<typename T, int BS, int LMAX> __global__ void
evaluateHybridPolyComplexToReal_kernel
   (HybridJobType *job_types, T **Ylm_complex, AtomicOrbitalCuda<T> *orbitals,
   HybridDataFloat *data, T *k_reduced, int *make2copies, T **vals, int N)

template<typename T, int BS, int LMAX> __global__ void
evaluateHybridPolyComplexToReal_knl
   (HybridJobType *job_types, T *rhats, T **Ylm_complex, float **dYlm_dTheta,
   float **dYlm_dphi, AtomicOrbitalCuda<T> *orbitals, HybridDataFloat *data,
   T *k_reduced, int *make2copies, T **vals, T **grad_lapl, int row_stride, int N)

template<typename T, int BS, int LMAX> __global__ void
CalcYlmComplex (T *rhats, HybridJobType *job_types,
   T **Ylm_ptr, T **dR, T **dYlm_dTheta_ptr, T **dYlm_dphi_ptr, int N)

template<typename T, int BS, int LMAX> __global__ void
CalcYlmReal (T *rhats, HybridJobType *job_types, T **Ylm_ptr, int N)

template<typename T, int BS, int LMAX> __global__ void
CalcYlmReal (T *rhats, HybridJobType *job_types, T **Ylm_ptr, int N)

Template<typename BS> __global__ void
evaluate3DsploeReal_kernel
   (HybridJobType *job_types, float *pos, float *kpoints_reduced, float3 dRinv,
   float *coefs, uint3 dim, uint3 strides, float *Linv, float **vals, int N)

template<typename BS> __global__ void
evaluate3DsploeReal_knl
   (HybridJobType *job_types, float *pos, float *kpoints, int *make2copies,
   float3 dRinv, float *coefs, uint3 dim, uint3 strides, float *Linv,
   float **vals, float **grad_lapl, int row_stride, int N)

template<typename BS> __global__ void
phase_factor_kernel (T *kPoints, int *makeTwo Copies, T *pos, T *phi_in,
   T *phi_out, int num_splines, int num_walkers)

template<typename BS> __global__ void
phase_factor_kernel (T *kPoints, int *makeTwo Copies, T *pos, T *phi_in,
   T *phi_out, int num_splines, int num_walkers)

template<typename BS> __global__ void
phase_factor_kernel (T *kPoints, int *makeTwo Copies, T *pos, T *phi_in,
   T *phi_out, T *grad_lapl_in, T **grad_lapl_out,
   int num_splines, int num_walkers, int row_stride)

template<typename BS> __global__ void
block_inverse (float* A, int N, int stride)

template<typename BS> __global__ void
inverse (T *A, T *work, int N, int stride)

template<typename BS> __global__ void
inverse_many (T **A_list, T **work_list, int N, int stride)

template<typename BS> __global__ void
inverse_many_pivot (T **A_list, T **work_list, int N, int stride)

template<typename BS> __global__ void
inverse_many_pivot (T **A_list, T **work_list, int N, int stride)
template<typename T, int BS> __global__ void
cplex_inverse_many_naiive_pivot (T **A_list, T **work_list, int N, int stride)

template<typename Tdest, typename Tsorc> __global__ void
convert (Tdest **dest_list, Tsorc **src_list, int len)

template<typename Tdest, typename Tsorc> __global__ void
convert (Tdest **dest_list, Tsorc **src_list,
        int dest_rows, int dest_cols, int dest_rowstride,
        int src_rows, int src_cols, int src_rowstride)

template<typename T1, typename T2, int Bs> __global__ void
check_inj (T1 **A, T2 **B, int N, int A_stride, int B_stride)

template<typename T, int BS> __global__ void
update_inverse_cudal1 (updateJob *updateList, int N, int rowstride)

template<typename T, int BS> __global__ void
update_inverse_cudal2 (updateJob *updateList, int N, int rowstride)

template<typename T, int BS> __global__ void
update_inverse_kernel1 (T **data, int *list, int A off, int AinV off,
        int newRow_off, int AinvDelta_off, int AinvColk_off,
        int N, int rowstride)

template<typename T, int BS> __global__ void
update_inverse_kernel2 (T **data, int *list, int A off, int AinV off,
        int newRow_off, int AinvDelta_off, int AinvColk_off,
        int N, int rowstride)

template<typename T, int BS> __global__ void
update_inverse_kernel1 (T **data, int k, int A off, int AinV off,
        int newRow_off, int AinvDelta_off, int AinvColk_off,
        int N, int rowstride)

template<typename T, int BS> __global__ void
update_inverse_kernel2 (T **data, int k, int A off, int AinV off,
        int newRow_off, int AinvDelta_off, int AinvColk_off,
        int N, int rowstride)

template<typename T, int BS> __global__ void
update_inverse_cudal1 (T **A_g, T **AinV_g, T **u_g,
        T **Ainv_delta_g, T **AinvColk_g,
        int N, int rowstride, int k)

template<typename T, int BS> __global__ void
update_inverse_cudal2 (T **A_g, T **AinV_g, T **u_g,
        T **Ainv_delta_g, T **AinvColk_g,
        int N, int rowstride, int k)

template<typename T, int MAX> __global__ void
update_inverse_transpose_cudal2pass (T **A_g, T **AinV_g, T **u_g,
        int N, int rowstride, int elec)

template<typename T, int BS> __global__ void
calc_ratios_transpose (T **AinV_list, T **new_row_list,
        T *ratio_out, int N, int row_stride, int elec,
        int numRats)

template<typename T, int BS> __global__ void
calc_ratios (T **AinV_list, T **new_row_list,
        T *ratio, int N, int row_stride, int elec)

template<typename T, int BS> __global__ void
calc_ratio_grad_lapl1 (T **AinV_list, T **new_row_list, T **grad_lapl_list,
        T *ratio_grad_lapl, int N, int row_stride, int elec)

template<typename T, int BS> __global__ void
calc_ratio_grad_lapl2 (T **AinV_list, T **new_row_list, T **grad_lapl_list,
        T *ratio_grad_lapl, int N, int row_stride, int elec)

template<typename T, int BS> __global__ void
calc_grad_kernel (T **AinV_list, T **grad_lapl_list,
        T *grad, int N, int row_stride, int elec)

template<typename T> __global__ void
all_ratios (T **AinV_list, T **new_mats_list,
        T **ratio_list, int N, int row_stride)

template<typename T, int BS> __global__ void
calc_many_ratios_kernel (T **AinV_list, T **new_row_list,
        T **ratio_list, int *num_ratio_list,
        int N, int row_stride, int *elec_list)

template<typename T> __global__ void
scale_grad_lapl_kernel (T **grad_lapl_list, T **hess_list,
        T **grad_lapl_list, T *Linv, int N)

template<typename T> __global__ void
all_ratios_grad_lapl_kernel (T **AinV_list, T **grad_lapl_list,
        T **out_list, int N, int row_stride)

template<typename T> __global__ void
multi_copy (T **dest, T **src, int len)

template<typename T> __global__ void
multi_copy (T **buff, int dest_off, int src_off, int len)

template<typename T> __global__ void
woodbury_update16 (T ** Ain_trans, T ** delta,
        T ** Ain_delta,
        int N, int rowstride)
Kernel classes

(a) CPU
- B-spline
- Inverse
- Det. ratio
- Phase
- 2-body
- 1-body
- Other

(b) GPU
- B-spline
- Inverse
- Det. ratio
- Phase
- 2-body
- 1-body
- Memcopy
- Other
The Trial Wavefunction

\[ \Psi_T(R) = \det(R^\uparrow) \det(R^\downarrow) e^{J_1 + J_2} \]

\[ \det(R) = \det \begin{vmatrix} \phi_1(r_1) & \phi_2(r_1) & \cdots & \phi_N(r_1) \\ \phi_1(r_2) & \phi_2(r_2) & \cdots & \phi_N(r_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_1(r_N) & \phi_2(r_N) & \cdots & \phi_N(r_N) \end{vmatrix} \]

Pauli exclusion

\[ J_1 = \sum_{i,j} u_1 (|r_i - I_j|) \]

\[ J_2 = \sum_{i \neq j} u_2 (|r_i - r_j|) \]

electron-nucleus electron-electron
Determinant ratios

\[
\det(R') = \det \begin{vmatrix}
\phi_1(r_1) & \phi_2(r_1) & \cdots & \phi_N(r_1) \\
\phi_1(r'_2) & \phi_2(r'_2) & \cdots & \phi_N(r'_2) \\
\vdots & \vdots & \ddots & \vdots \\
\phi_1(r_N) & \phi_2(r_N) & \cdots & \phi_N(r_N)
\end{vmatrix}
\]

\[
\det(R) = \det \begin{vmatrix}
\phi_1(r_1) & \phi_2(r_1) & \cdots & \phi_N(r_1) \\
\phi_1(r_2) & \phi_2(r_2) & \cdots & \phi_N(r_2) \\
\vdots & \vdots & \ddots & \vdots \\
\phi_1(r_N) & \phi_2(r_N) & \cdots & \phi_N(r_N)
\end{vmatrix}
\]

Determinant ratio computed by dotting with corresponding column of the inverse of the orbital matrix
Electronic orbitals

- Orbital: general function of \((x,y,z)\)
- Represented in a linear basis
- \(N\) electrons \(\rightarrow\) \(N/2\) occupied orbitals
- Evaluate all orbitals at the same time for a single \((x,y,z)\)
Evaluating orbitals

- In QMC, we evaluate orbitals at a specific point in space.
- Most other methods evaluate integrals over all space directly.
- We find 3D cubic B-spline basis is fastest for large systems:
  - Basis elements are strictly local.
  - Only 64 elements are nonzero at each point.
  - Per-orbital evaluation time is independent of system size.
Evaluating 3D B-splines

- First, evaluate 64 basis elements
  - \( B_{ijk}(x, y, z) = a_i(x)b_j(y)c_k(z) \)
  - 1D functions are piecewise polynomials

- Then, do matrix-vector product with orbital index fastest

- For value only: 2 FLOPS per load: bandwidth limited, 75 GB/s

- For value, gradient, and Hessian: 20 FLOPS per load
  216 GFLOPS

\(~256\) walkers x
Overcoming memory limitations

- 3D B-splines use a lot of memory
  - 4GB limits system size
  - Spacing defined by smallest feature size of orbitals
- Small features are in atomic cores
- Divide space into two regions
  - Inside cores: Use 1D splines times spherical harmonics ($Y_{lm}$)
  - Outside cores: use coarse 3D B-spline
- Save more that 10x on memory
- A little slower, but less noise in energy
Inverse matrices

- Determinant ratios and derivatives use inverse of the determinant matrix
- Since only one row changes at a time, we can use an update formula:
  \[
  [A + e_k \delta^T]^{-1} = A^{-1} - \frac{A^{-1} e_k \delta^T A^{-1}}{1 + \delta^T A^{-1} e_k}
  \]
- Requires matrix-vector product, then adding an outer product to $A^{-1}$
- Dominant computation only 1 FLOP per load/store → bandwidth limited.
- On CPU, matrices fit in cache, so GPU speedup is relatively poor
Multi-GPU parallelization

- Current model: 1 MPI process per GPU
  - Select devices automatically
  - Want to change to MPI / OpenMP model to allow fast on-node communication

- Main communication is for load balancing
  - Fluctuating DMC population creates imbalance
  - Transfer walker data point-to-point
  - All data is packed in a single buffer in GPU RAM
  - Allows efficient transfers and minimizes fragmentation
Results: Accuracy

- CPU: double precision
- GPU: mixed precision
- Random numbers used in different order
- Can only compare answers statistically
- No statistically significant differences!
- VMC tested to 6 digits
Results: Performance

- Speedup is GPU speed / quad-core Xeon speed
- CPU is in double precision, GPU is in mixed precision
- Optimal throughput with large system, many walkers
- Saturates above 256 walkers
  - May run out of RAM 1st
- With large system, speed is above 15 quad-core processors
Performance analysis

(a) CPU

B-spline

Inv. update  Det. ratio  Phase  2-body  1-body  Other

(b) GPU

B-spline  Inverse
Performance analysis

- Bandwidth limited on CPU and GPU
  - Orbital evaluation
  - Smaller % of total time on GPU
- Bandwidth limited on GPU, but not CPU
  - Determinant update
  - Smaller % of total time on CPU
- Compute limited on CPU and GPU
  - Jastrow / distance evaluation
  - About same % of total time on CPU and GPU
Scaling with system size

- Determinant update, inverse: $O(N^3)$
- Everything else: $O(N^2)$
- For largest systems, update dominates
  - Not true for CPU!
  - L3 cache saves the day
  - Solution: use delayed updates on GPUs
  - On synthetic tests, delayed updates are 4x to 8x faster
Scaling with the number of GPUs

- Higher throughput on GPUs puts greater demand on interconnect
- Down to 88% efficiency on Lincoln at 256 GPUs
- Slightly better on Longhorn?
- PCI-E or IB limited?
Future directions

- Enable larger systems to be simulated
  - Multi-threaded framework to allow distribution of read-only orbital dataset between GPUs on same node
  - Use Fermi to allow smaller # of walkers / GPU
    - Shorten wall time for equilibration
    - Launch several kernels in parallel
    - Fast atomics to communicate between blocks
- More science!
- More documentation → more users
Acknowledgments

- Dr. Luke Shulenburger for invaluable feedback
- Department of Energy Contract No. DOE-DE-FG05-08OR23336
- National Science Foundation No. 0904572
- National Center for Computational Sciences and the Center for Nanophase Materials Sciences
- NSF Teragrid resources under TG-MCA93S030 and TG-MCA07S016
The End

- Dr. Luke Shulenburger for invaluable feedback
- Department of Energy Contract No. DOE-DE-FG05-08OR23336
- National Science Foundation No. 0904572
- National Center for Computational Sciences and the Center for Nanophase Materials Sciences
- NSF Teragrid resources under TG-MCA93S030 and TG-MCA07S016
Quantum Simulation Methods

- Semi-empirical methods
  - Replace true energy operator with parametrized form
  - Fast, but little predictive capability
- Ab initio methods
  - No free parameters
  - Only input is the type and positions of the atoms
  - Many flavors, varying in accuracy and computational expense
Ab initio methods

- Density Functional Theory (DFT)
  - Maps 3N-dimensional problem approximately onto N 3-dimensional problems
  - Mapping not exact because electrons repel each other (correlation)
  - Relatively fast [order(N) to order(N^3)]
  - Often gives very good results
  - Sometimes qualitatively wrong (e.g. predicts some insulators to be metals)
  - Many functionals, no a priori way to choose
Quantum Chemistry Methods

- Expand $\Psi$ in a linear basis
  - For exact solution, size of basis $\sim \exp(N)$
  - Approximate solutions truncate basis intelligently
  - Most accurate method for small molecules
  - Can compute many properties of molecules
  - Different levels of accuracy:
    - CASSCF
    - Coupled-cluster
  - Too expensive for large systems at high-accuracy