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

Ken Esler[1]
Jeongnim Kim
David Ceperley

National Center for Supercomputing
University of Illinois at Urbana-Champaign

[1]To be Stone Ridge Technology as of 7/19/10

NCSA

Dirac's Challenge

"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

NCSA

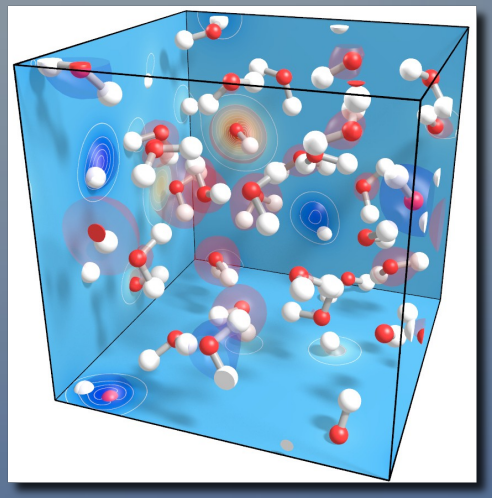
Dirac's Challenge

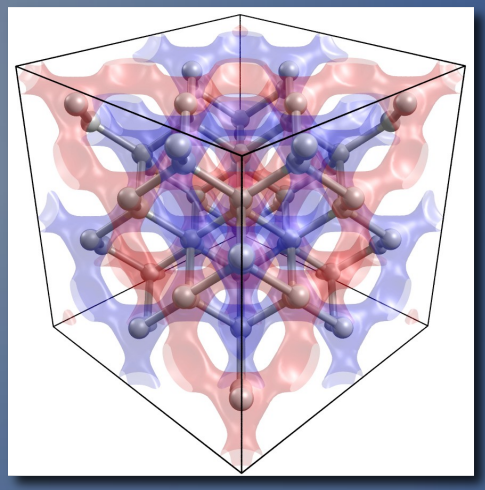
$$-\frac{1}{2}\sum_i \nabla_i^2 \Psi(\mathbf{r}_1, \mathbf{r}_2, \dots) + \left[\sum_{i \neq j} \frac{1}{|\mathbf{r}_i - \mathbf{r}_j|} + \sum_{i,k} \frac{-Z_k}{|\mathbf{r}_i - \mathbf{I}_k|} - E\right] \Psi(\mathbf{r}_1, \mathbf{r}_2, \dots) = 0$$

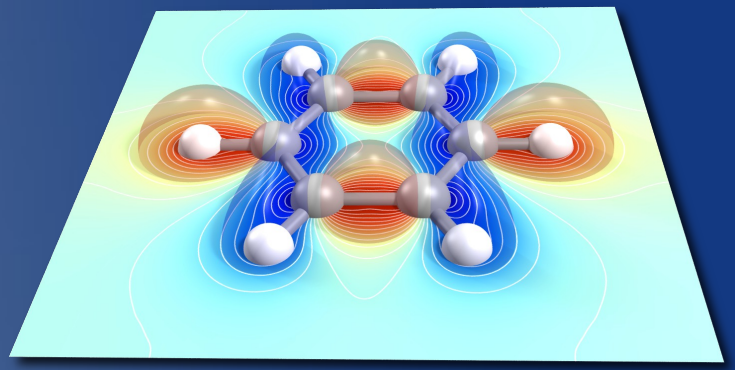
- For N electrons, Ψ is a function in 3N dimensions
- Exact solution impossible if N > ~5
- Theoretical physics and chemists try to find high-accuracy approximate solutions

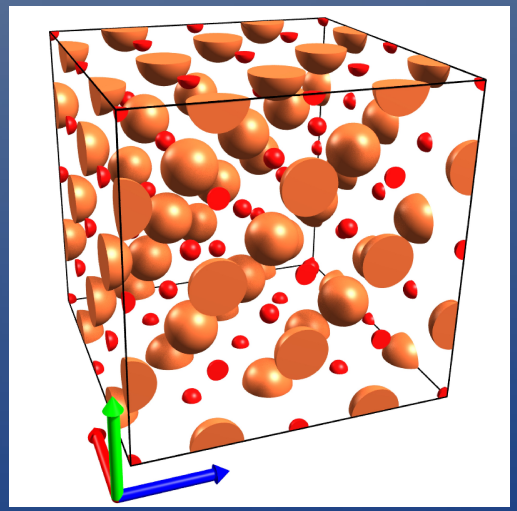
NCSA

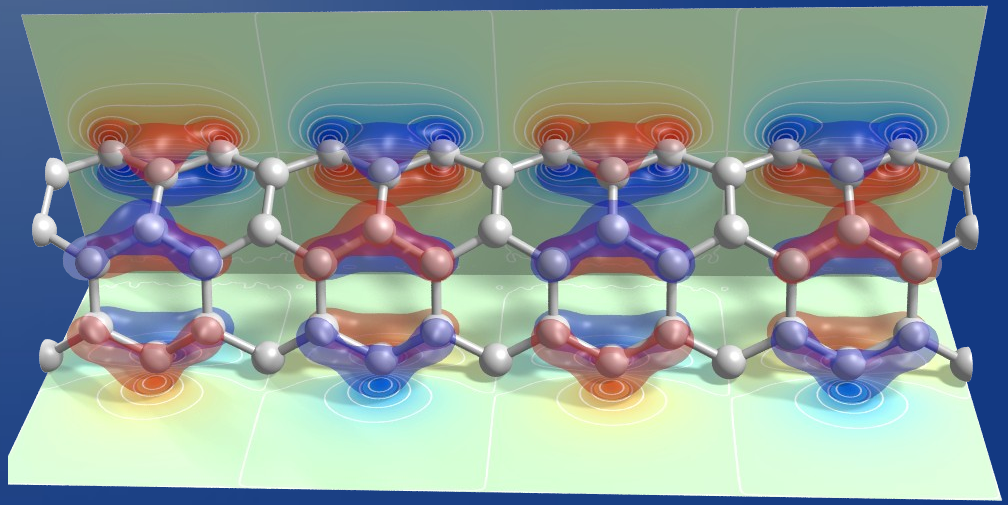
Example Systems











NCSA

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)
  - Auxilliary field Monte Carlo
  - Lattice methods
    - Solve model problems

NCSA

Variational Monte Carlo

- Write down a parametrized form for $\Psi(\mathbf{R})$
  - $\mathbf{R} = \{\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3, \ldots\}$
- Sample distribution P(R) = $|\Psi(R)|^2$ with Metropolis Monte Carlo
  - Propose move $r_i \rightarrow r_i'$
  - Accept or reject with probability $\propto \left| \frac{\Psi(\mathbf{R}')}{\Psi(\mathbf{R})} \right|^2$
- Average energy over distribution, $\langle E \rangle$
- Minimize $\langle E \rangle$ with respect to parameters of Ψ

NCSA

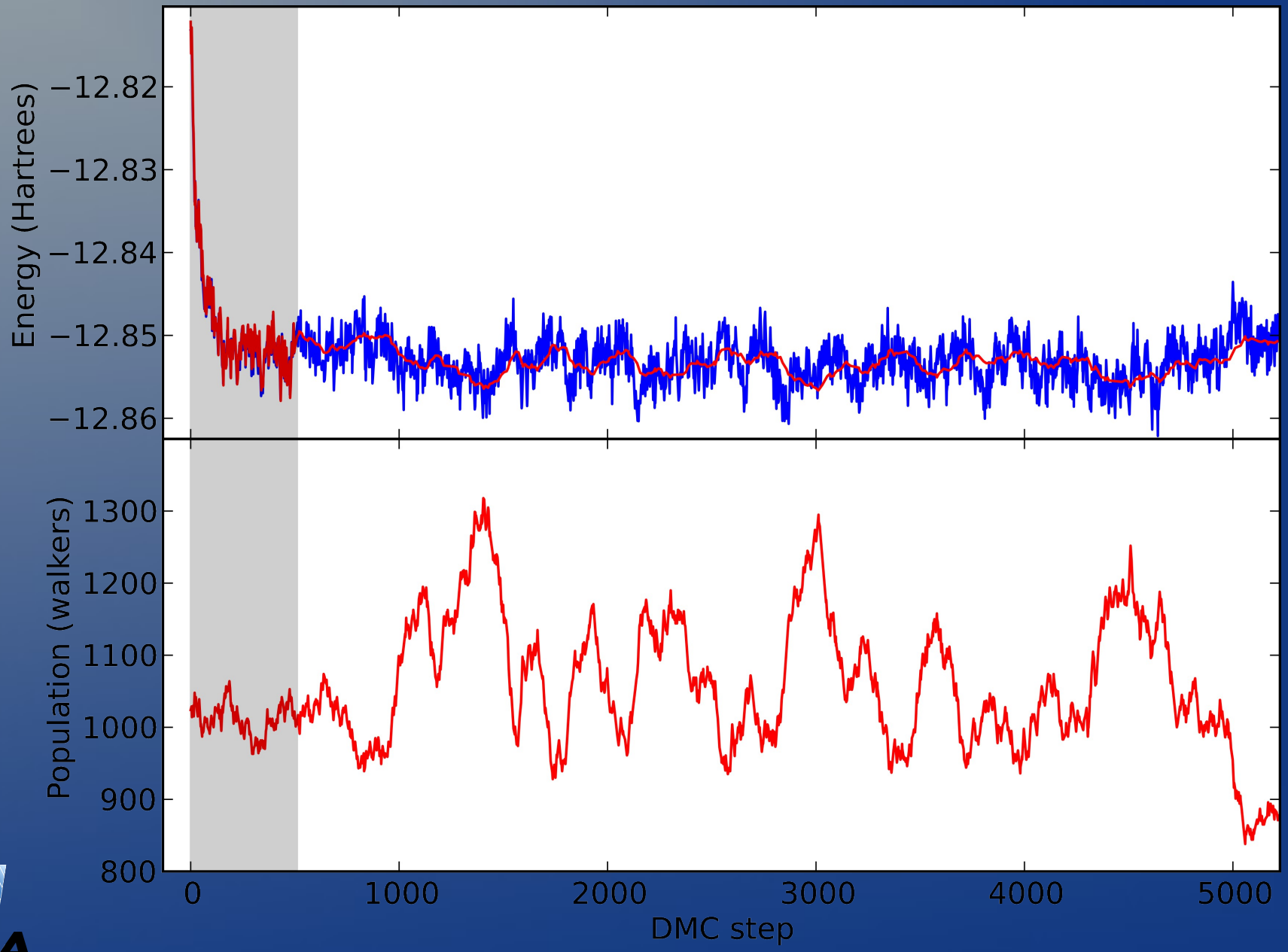
Diffusion Monte Carlo

- Start with optimized Ψ from VMC
- Walker = R, point in 3N-dimensional space
  - i.e. the positions of all electrons
- Start with population of ~1000 walkers
- Algorithm
  - Drift/diffuse: try to move each electron in each walker in sequence
  - Branch: make M copies of each walker, where

$$M \propto \exp\{\tau\,[E_T - E(R)]\}$$

NCSA

DMC Output: Total Energy



NCSA

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

NCSA

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

NCSA

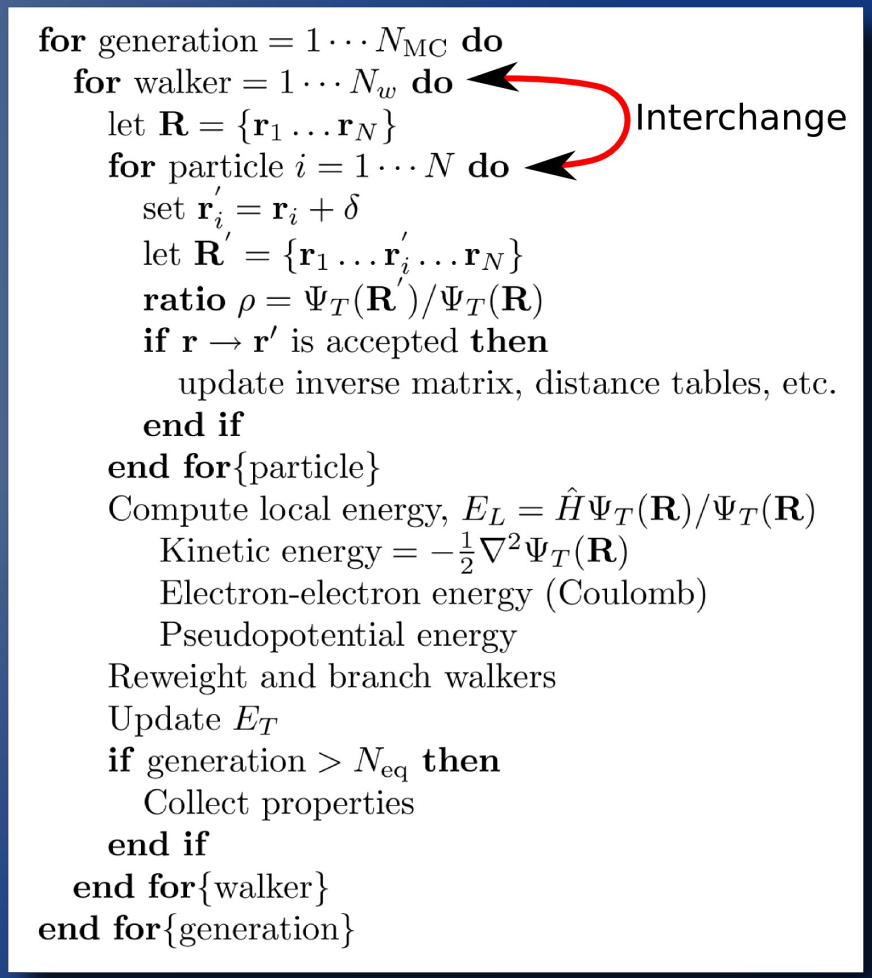
QMCPACK

- 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

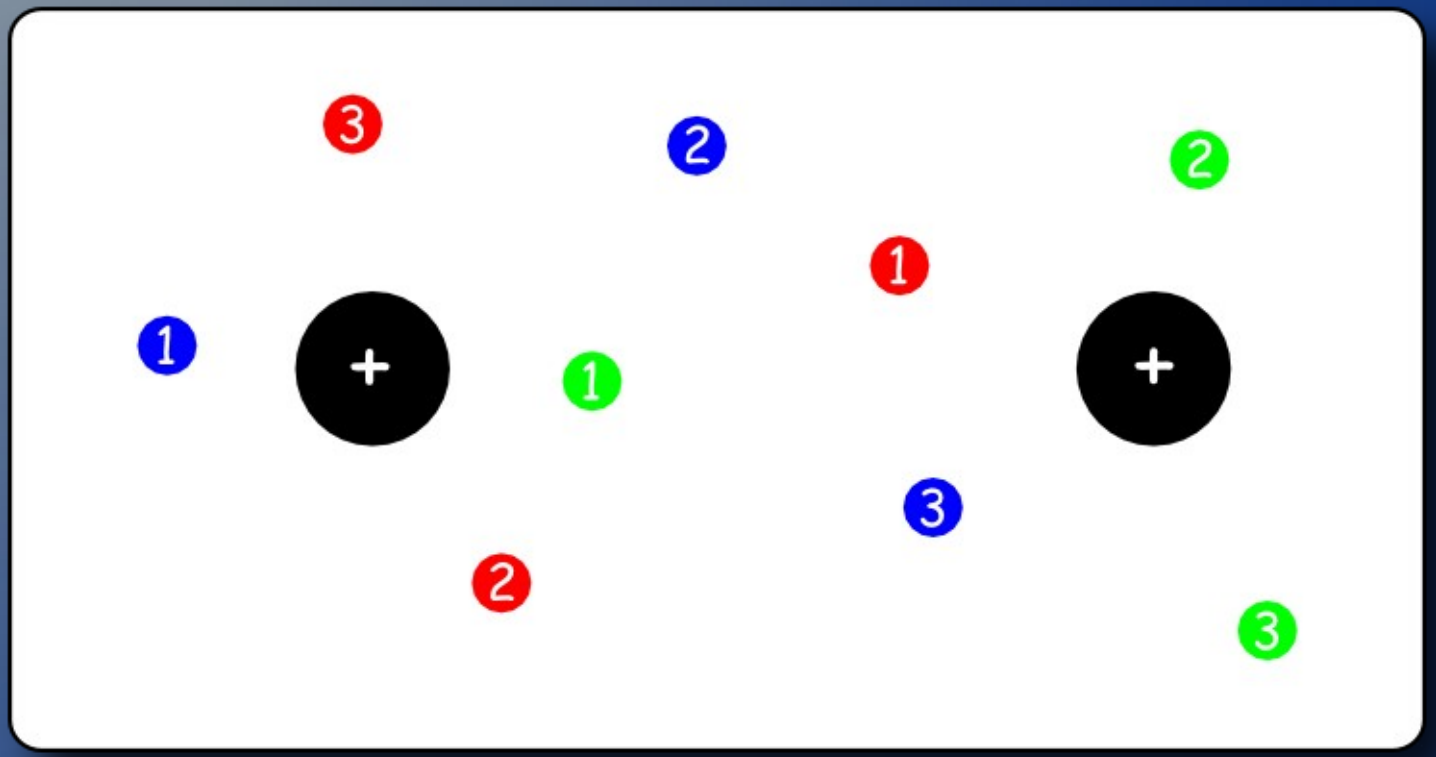
NCSA

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

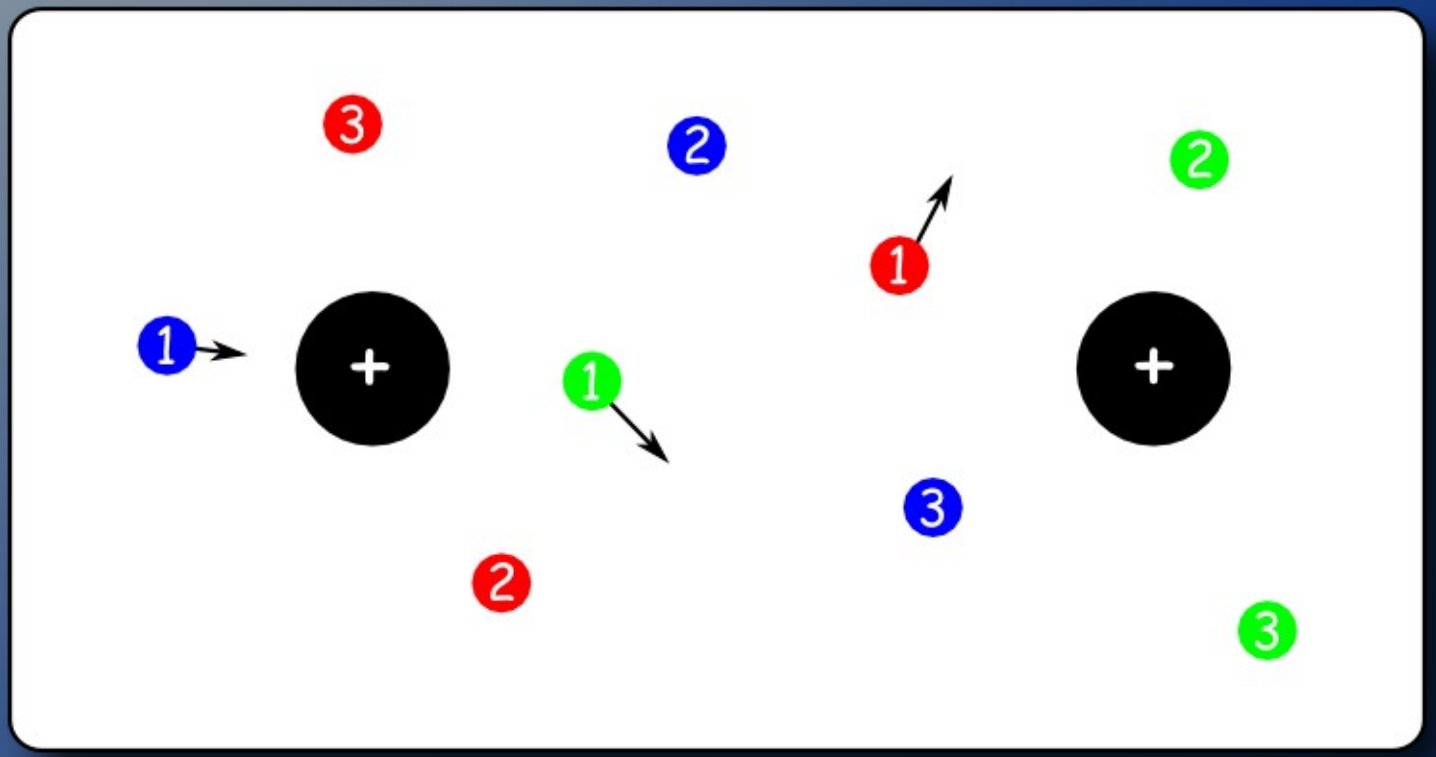


DMC simulation



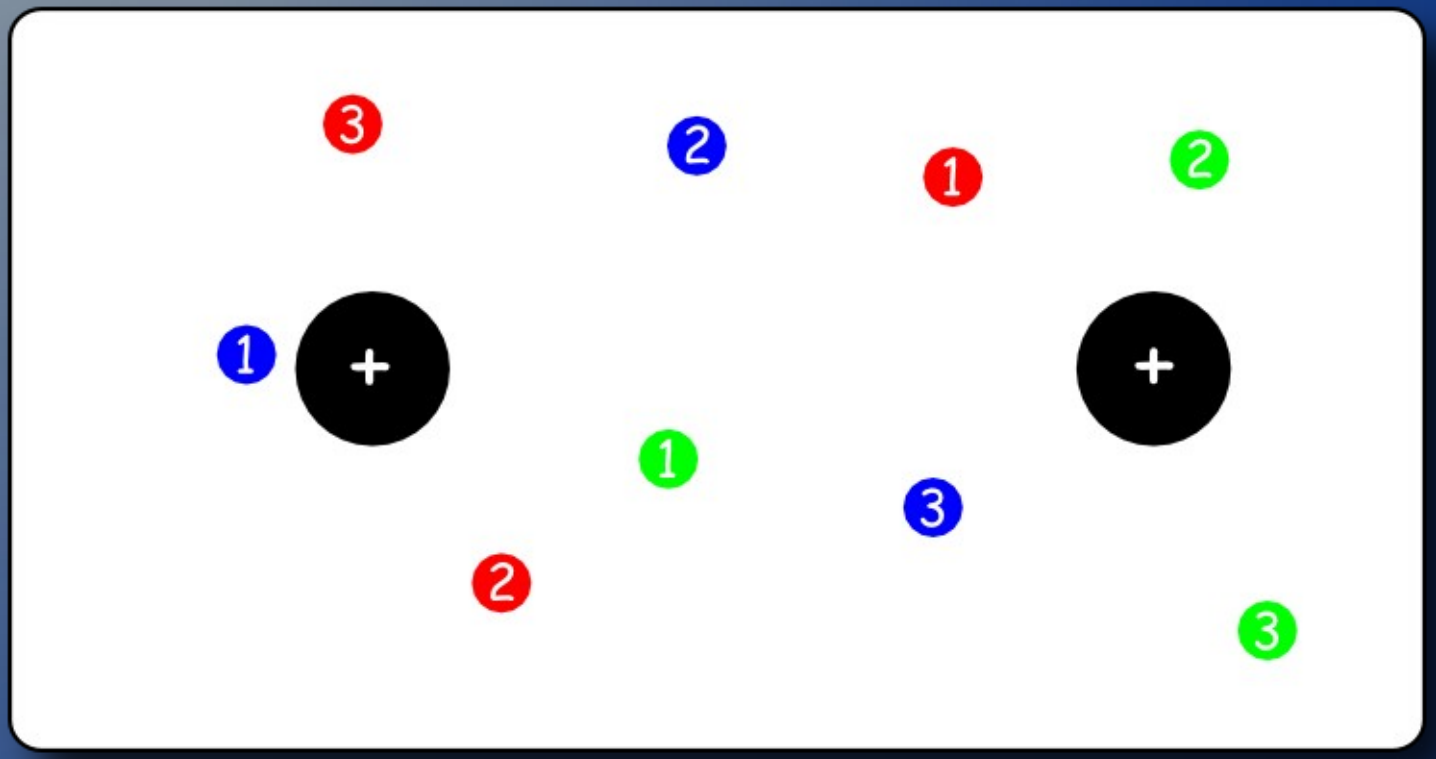
NCSA

DMC simulation



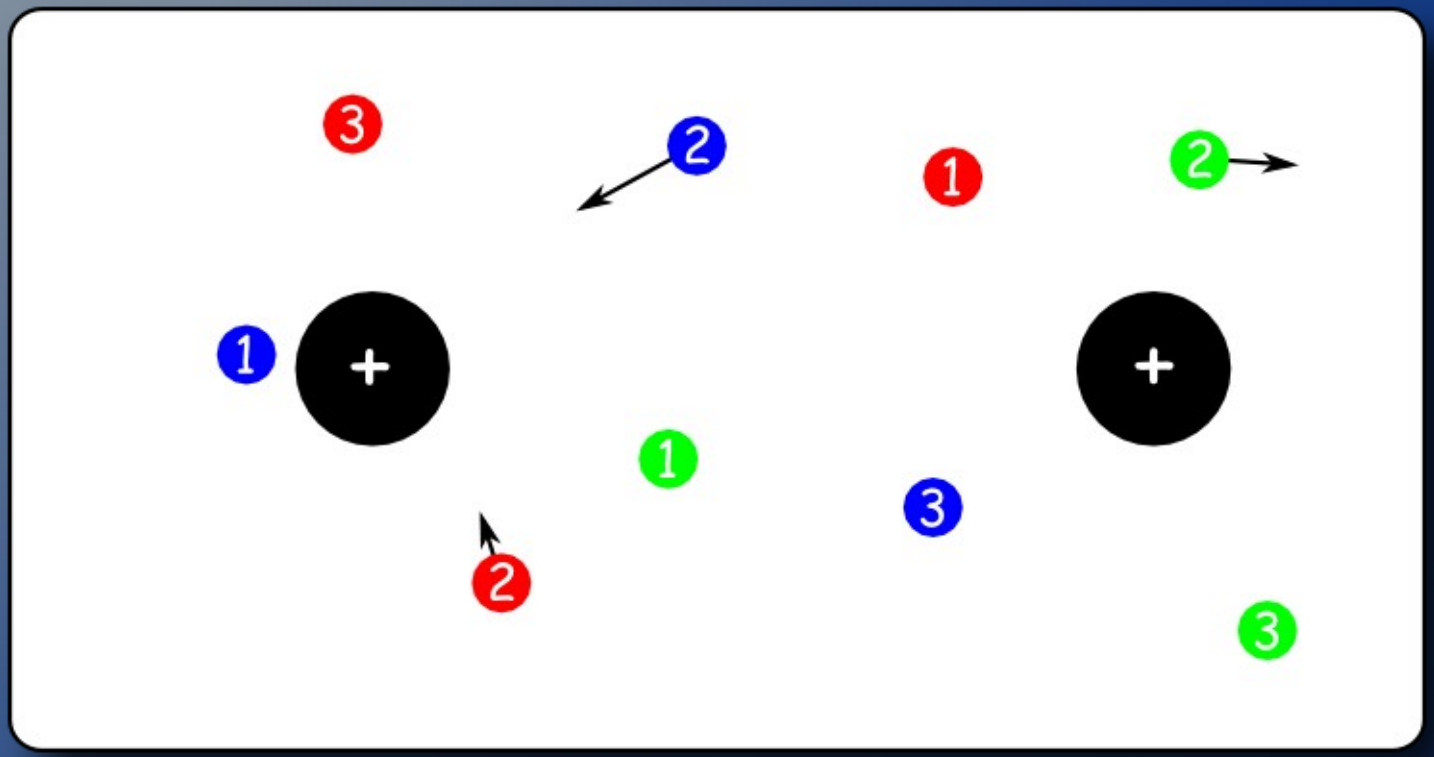
NCSA

DMC simulation



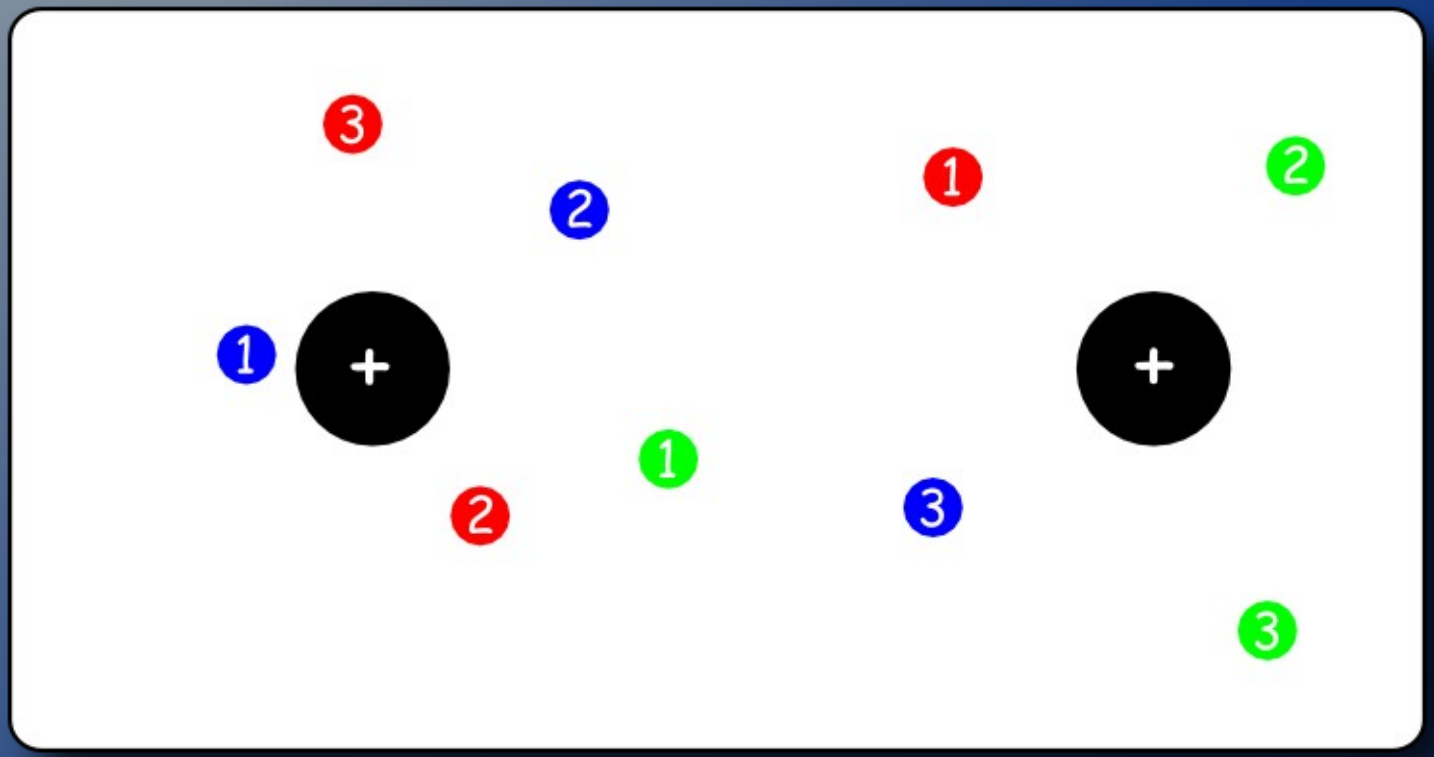
NCSA

DMC simulation



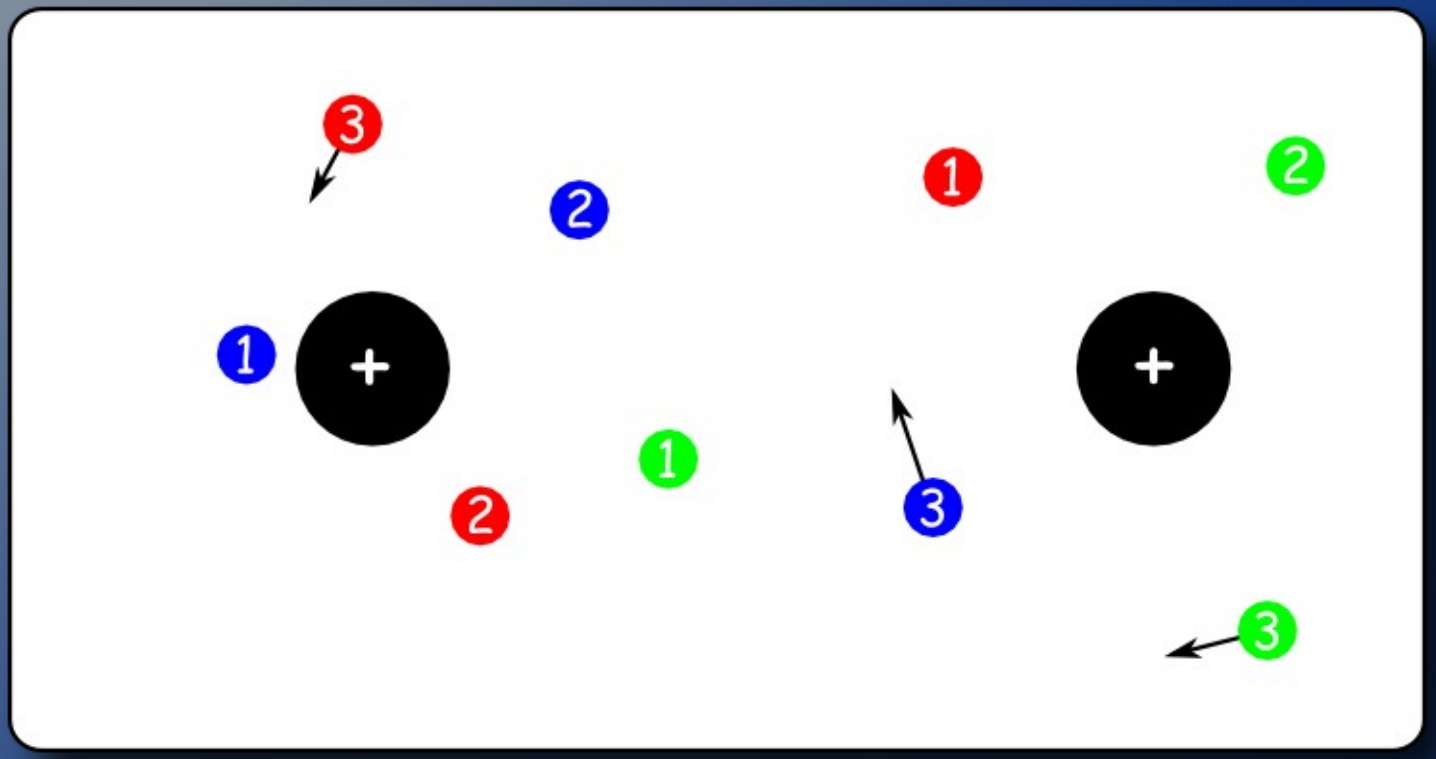
NCSA

DMC simulation



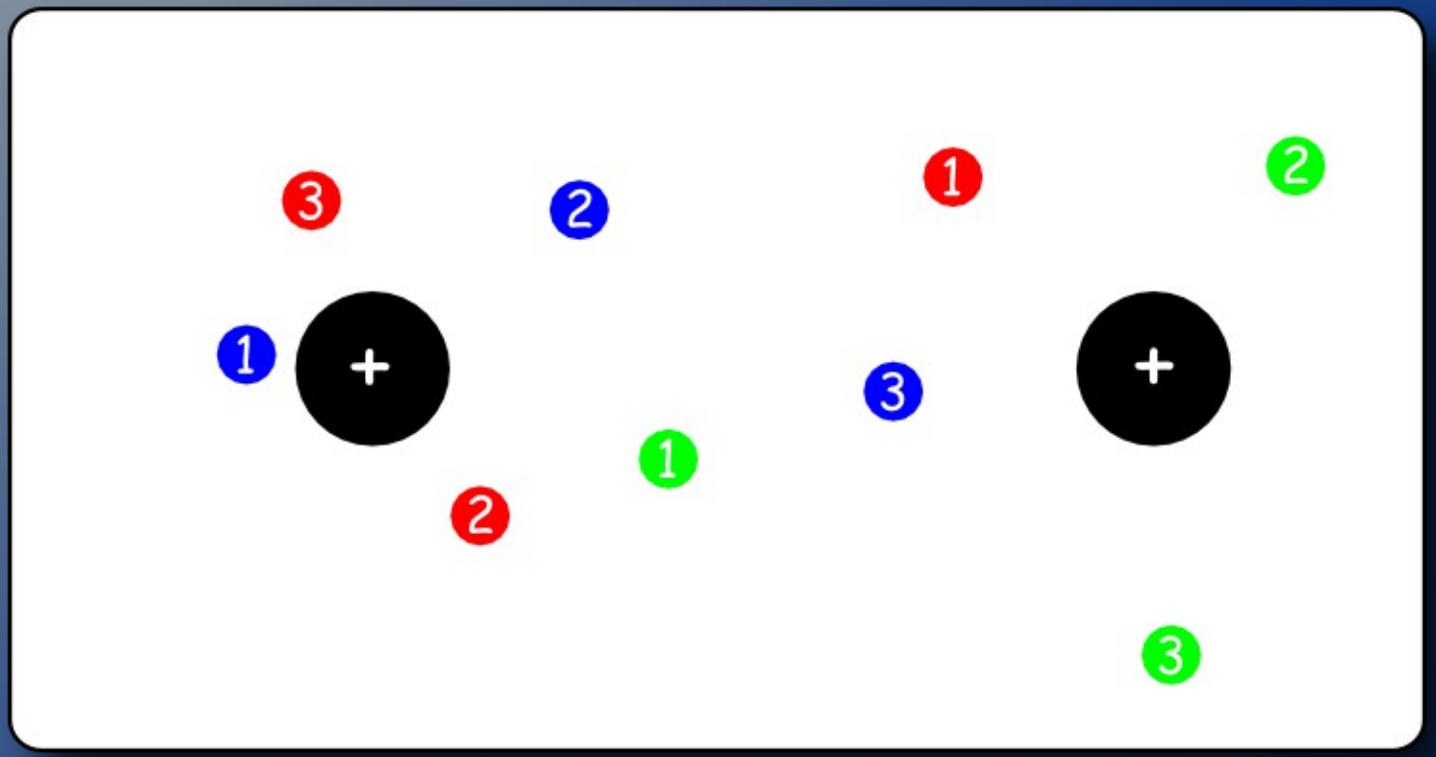
NCSA

DMC simulation



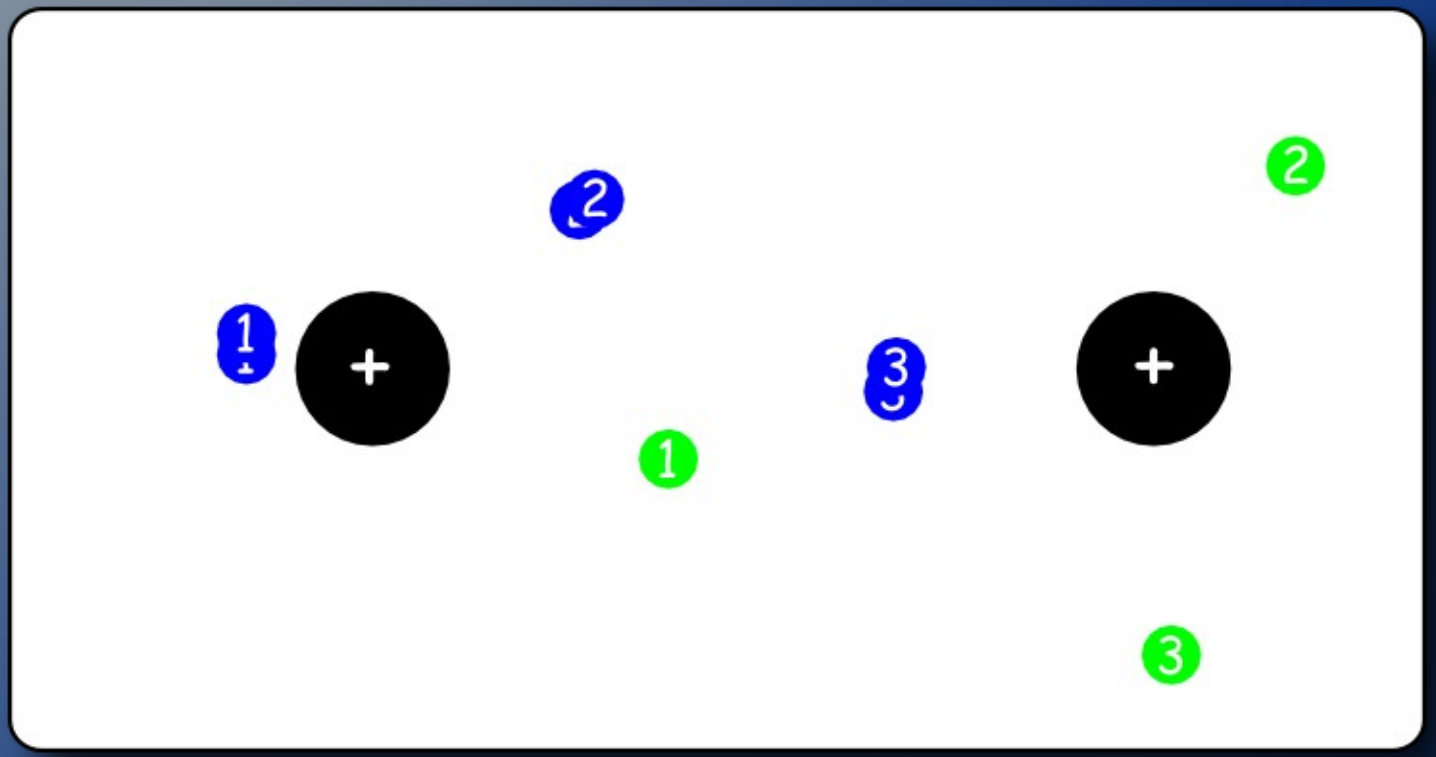
NCSA

DMC simulation



NCSA

DMC simulation



Computational Kernels

- Wave function and derivatives:
  - Ratio, gradient, and Laplacian

$$\frac{\Psi(\mathbf{R}')}{\Psi(\mathbf{R})}, \quad \frac{\nabla\Psi(\mathbf{R}')}{\Psi(\mathbf{R})}, \quad \frac{\nabla^2\Psi(\mathbf{R}')}{\Psi(\mathbf{R})}$$

  - Kernels needed for determinants, J_1, and J_2

- Potential energy:
  - Periodic coulomb interaction
    - Part in real-space, part in frequency space
  - Pseudopotential interaction

NCSA

```
template<typename T, int BS> __global__ void
accept_kernel (T** Rlist, T* Rnew, int* toAccept, int iat, int N)

template<typename T, int BS> __global__
void NL_move_kernel (T** Rlist, T* Rnew, int N)

template<typename T, int BS> __global__ void
coulomb_AA_PBC_kernel(T **R, int N, T rMax, int Ntex, int textureNum,
                      T *lattice, T *latticeInv, T *sum)

template<typename T, int BS> __global__
void coulomb_AA_kernel(T **R, int N, T *sum)

template<typename T, int BS> __global__ void
MPC_SR_kernel(T **R, int N, T *lattice, T *latticeInv, T *sum)

template<typename T, int BS> __global__ void
MPC_LR_kernel(T **R, int N, T* coefs, typename Three<T>::type gridInv,
              uint3 dim, uint3 strides, T *latticeInv, T *sum)

template<typename T, int BS> __global__ void
coulomb_AB_PBC_kernel(T **R, int Nelec, T *I, int Ifirst, int Ilast,
                      T rMax, int Ntex, int textureNum,
                      T *lattice, T *latticeInv, T *sum)

template<typename T, int BS> __global__ void
local_ecp_kernel(T **R, int Nelec, T *I, int Ifirst, int Ilast,
                 T rMax, int Ntex, int textureNum, T *sum)

template<typename T, int BS> __global__ void
coulomb_AB_kernel(T **R, int Nelec, T *I, T *Zion, int Nion, T *sum)

template<typename T, int BS> __global__ void
eval_rhok_kernel (T **R, int numr, T *kpoints, int numk, T **rhok)

template<typename T, int BS> __global__ void
eval_rhok_kernel (T **R, int first, int last,
                  T *kpoints, int numk, T **rhok)

template<typename T, int BS>
__global__ void vk_sum_kernel(T **rhok, T *vk, int numk, T *sum)

template<typename T, int BS> __global__ void
vk_sum_kernel2(T **rhok1, T **rhok2, T *vk, int numk, T *sum)

template<typename T, int BS> __global__ void
vk_sum_kernel2(T **rhok1, T *rhok2, T *vk, int numk, T *sum)

template<typename T, int BS> __global__ void
find_core_electrons_PBC_kernel(T **R, int numElec,
                               T *I, int firstIon, int lastIon,
                               T rcut, T *L_global, T *Linv_global,
                               int2 **pairs, T **dist, int *numPairs)
```

```
template<typename T, int BS> __global__ void
find_core_electrons_PBC_kernel(T **R, int numElec,
                               T *I, int firstIon, int lastIon,
                               T rcut, T *L_global, T *Linv_global,
                               T *quadPoints, int numQuadPoints,
                               int **elecs, T **ratioPos,
                               T **dist_list, T **cosTheta_list, int *numPairs)


template<typename T, int BS> __global__ void
find_core_electrons_kernel(T **R, int numElec,
                           T *I, int firstIon, int lastIon,
                           T rcut, int2 **pairs, T **dist, int *numPairs)

template<typename T, int BS> __global__ void
find_core_electrons_kernel(T **R, int numElec,
                           T *I, int firstIon, int lastIon,
                           T rcut, T *quadPoints, int numQuadPoints,
                           int **elecs, T **ratioPos,
                           T **dist_list, T **cosTheta_list, int *numPairs)

template<typename T, int BS> __global__ void
make_work_list_kernel (int2 **pairs, T **dist, int *numPairs,
                       T *I, int numIons, T *quadPoints, int numQuadPoints,
                       T **ratio_pos)

template<typename T, int BS> __global__ void
MakeHybridJobList_kernel (T* elec_list, int num_elecs, T* ion_list,
                          T* poly_radii, T* spline_radii,
                          int num_ions, T *L, T *Linv,
                          HybridJobType *job_list, T *rhat_list,
                          HybridDataFloat *data_list)

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

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

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

NCSA

```
template<typename T,int BS,int LMAX> __global__ void
evaluateHybridPolyReal_kernel (HybridJobType *job_types, T* rhats,
                               T **YlmReal, 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 **YlmComplex, AtomicOrbitalCuda<T> *orbitals,
   HybridDataFloat *data, T *k_reduced, int *make2copies,
   T **vals, int N)


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

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

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

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

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

```
template<int BS> __global__ void
evaluate3DSplineReal_kernel (HybridJobType *job_types, float *pos, float *k_reduced,
                             float3 drInv, float *coefs, uint3 dim, uint3 strides,
                             float *Linv, float **vals, float **grad_lapl,
                             int row_stride, int N)

template<int BS> __global__ void
evaluate3DSplineReal_kernel
  (HybridJobType *job_types, float *pos, float *kpoints_reduced, float3 drInv,
   float *coefs, uint3 dim, uint3 strides, float *Linv, float **vals, int N)

template<int BS> __global__ void
evaluate3DSplineComplexToReal_kernel
  (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<int BS> __global__ void
evaluate3DSplineComplexToReal_kernel
  (HybridJobType *job_types, float *pos, float *kpoints, int *make2copies,
   float3 drInv, float *coefs, uint3 dim, uint3 strides,
   float *Linv, float **vals, int N)

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

template<typename T, int BS> __global__ void
phase_factor_kernel_new (T *kPoints, int *makeTwoCopies, T *pos, T **phi_in,
                         T **phi_out, int num_splines, int num_walkers)

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

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

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

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

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

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

NCSA

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

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

template<typename Tdest, typename Tsrc> __global__ void
convert (Tdest **dest_list, Tsrc **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_inv (T1 **A, T2 **B, int N, int Astride, int Bstride)

template<typename T, int BS> __global__ void
update_inverse_cuda1 (updateJob *updateList, int N, int rowstride)

template<typename T, int BS> __global__ void
update_inverse_cuda2 (updateJob *updateList, int N, int rowstride)

template<typename T, int BS> __global__ void
update_inverse_kernel1 (T **data, int *iat, 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 *iat, 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_cuda1 (T **A_g, T **Ainv_g, T **u_g,
                      T **Ainv_delta_g, T **Ainv_colk_g,
                      int N, int rowstride, int k)

template<typename T, int BS> __global__ void
update_inverse_cuda2 (T **A_g, T **Ainv_g, T **u_g,
                      T **Ainv_delta_g, T **Ainv_colk_g,
                      int N, int rowstride, int k)

template<typename T, int BS, int MAXN> __global__ void
update_inverse_transpose_cuda(T **A_g, T **AinvT_g, T **u_g,
                              int N, int row_stride, int elec)
```

```
template<typename T, int BS, int MAXN> __global__ void
update_inverse_transpose_cuda_2pass(T **A_g, T **AinvT_g, T **u_g,
                                    int N, int row_stride, int elec)

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

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_lapl (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_lapl (T **Ainv_list, T **new_row_list, T **grad_lapl_list,
                      T *ratio_grad_lapl, int N, int row_stride, int *elec_list)

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_kernel (T **Ainv_list, T **new_mat_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_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_update_16 (T** Ainv_trans, T** delta,
                    T** Ainv_delta,
                    int N, int rowstride)
```

NCSA

```
template<typename T> __global__ void
woodbury_update_16a (T** Ainv_trans, T** delta, T** Ainv_delta, T** inv_block,
                     int N, int rowstride, int kblock)

template<typename T> __global__ void
woodbury_update_16b (T** Ainv_trans, T** delta, T** Ainv_delta, T** inv_block,
                     int N, int rowstride, int kblock)

template<typename T> __global__ void
woodbury_update_32 (T** Ainv_trans, T** delta, T** Ainv_delta,
                    int N, int rowstride)

template<typename T, int BS > __global__ void
two_body_sum_kernel(T **R, int e1_first, int e1_last, int e2_first, int e2_last,
                    T *spline_coefs, int numCoefs, T rMax, T* sum)

template<typename T, int BS> __global__ void
two_body_ratio_kernel(T **R, int first, int last, T *Rnew, int inew,
                      T *spline_coefs, int numCoefs, T rMax, T* sum)

template<typename T, int BS> __global__ void
two_body_ratio_grad_kernel(T **R, int first, int last, T *Rnew, int inew,
                           T *spline_coefs, int numCoefs, T rMax,
                           bool zero, T *ratio_grad)

template<int BS> __global__ void
two_body_NLratio_kernel(NLjobGPU<float> *jobs, int first, int last,
                        float** spline_coefs, int *numCoefs, float *rMaxList)

template<int BS> __global__ void
two_body_NLratio_kernel(NLjobGPU<double> *jobs, int first, int last,
                        double **spline_coefs, int *numCoefs, double *rMaxList)

template<typename T, int BS> __global__ void
two_body_grad_lapl_kernel(T **R, int e1_first, int e1_last, int e2_first, int e2_last,
                          T *spline_coefs, int numCoefs, T rMax,
                          T *gradLapl, int row_stride)

template<typename T, int BS> __global__ void
two_body_grad_kernel(T **R, int first, int last, int iat, T *spline_coefs,
                     int numCoefs, T rMax, bool zeroOut, T *grad)

template<typename T, int BS> __global__ void
two_body_derivs_kernel(T **R, T **gradLogPsi, int e1_first, int e1_last,
                       int e2_first, int e2_last, int numCoefs, T rMax, T **derivs)

template<typename T, int BS > __global__ void
one_body_sum_kernel(T *C, T **R, int cfirst, int clast, int efirst, int elast,
                    T *spline_coefs, int numCoefs, T rMax, T *sum)

template<typename T, int BS> __global__ void
one_body_ratio_kernel(T *C, T **R, int cfirst, int clast, T *Rnew, int inew,
                      T *spline_coefs, int numCoefs, T rMax, T* sum)
```

```
template<typename T, int BS> __global__ void
one_body_ratio_grad_kernel(T *C, T **R, int cfirst, int clast, T *Rnew, int inew,
                           T *spline_coefs, int numCoefs, T rMax,
                           bool zero, T *ratio_grad)

template<typename T, int BS> __global__ void
one_body_grad_lapl_kernel(T *C, T **R, int cfirst, int clast, int efirst, int elast,
                          T *spline_coefs, int numCoefs, T rMax,
                          T *gradLapl, int row_stride)

template<int BS> __global__ void
one_body_NLratio_kernel(NLjobGPU<float> *jobs, float *C, int first, int last,
                        float *spline_coefs, int numCoefs, float rMax)

template<int BS> __global__ void
one_body_NLratio_kernel(NLjobGPU<double> *jobs, double *C, int first, int last,
                        double *spline_coefs, int numCoefs, double rMax)

template<typename T, int BS> __global__ void
one_body_grad_kernel(T **R, int iat, T *C, int first, int last, T *spline_coefs,
                     int numCoefs, T rMax, bool zeroOut, T* grad)

template<typename T, int BS> __global__ void
one_body_derivs_kernel(T* C, T **R, T **gradLogPsi, int cfirst, int clast,
                       int efirst, int elast, int numCoefs, T rMax, T **derivs)

template<typename T, int BS > __global__ void
two_body_sum_PBC_kernel(T **R, int e1_first, int e1_last, int e2_first, int e2_last,
                        T *spline_coefs, int numCoefs, T rMax,
                        T *lattice, T* latticeInv, T* sum)

template<typename T, int BS> __global__ void
two_body_ratio_PBC_kernel(T **R, int first, int last, T *Rnew, int inew, T *spline_coefs,
                          int numCoefs, T rMax, T *lattice, T* latticeInv, T* sum)

template<typename T, int BS> __global__ void
two_body_ratio_grad_PBC_kernel(T **R, int first, int last, T *Rnew, int inew,
                               T *spline_coefs, int numCoefs, T rMax,
                               T *lattice, T* latticeInv, bool zero, T *ratio_grad)

template<typename T, int BS> __global__ void
two_body_ratio_grad_PBC_kernel_fast
  (T **R, int first, int last, T *Rnew, int inew, T *spline_coefs, int numCoefs,
   T rMax, T *lattice, T* latticeInv, bool zero, T* ratio_grad)

template<int BS> __global__ void
two_body_NLratio_PBC_kernel(NLjobGPU<float> *jobs, int first, int last,
                            float** spline_coefs, int *numCoefs, float *rMaxList,
                            float* lattice, float* latticeInv, float sim_cell_radius)

template<int BS> __global__ void
two_body_NLratio_PBC_kernel(NLjobGPU<double> *jobs, int first, int last,
                            double **spline_coefs, int *numCoefs, double *rMaxList,
                            double *lattice, double *latticeInv, double sim cell radius)
```

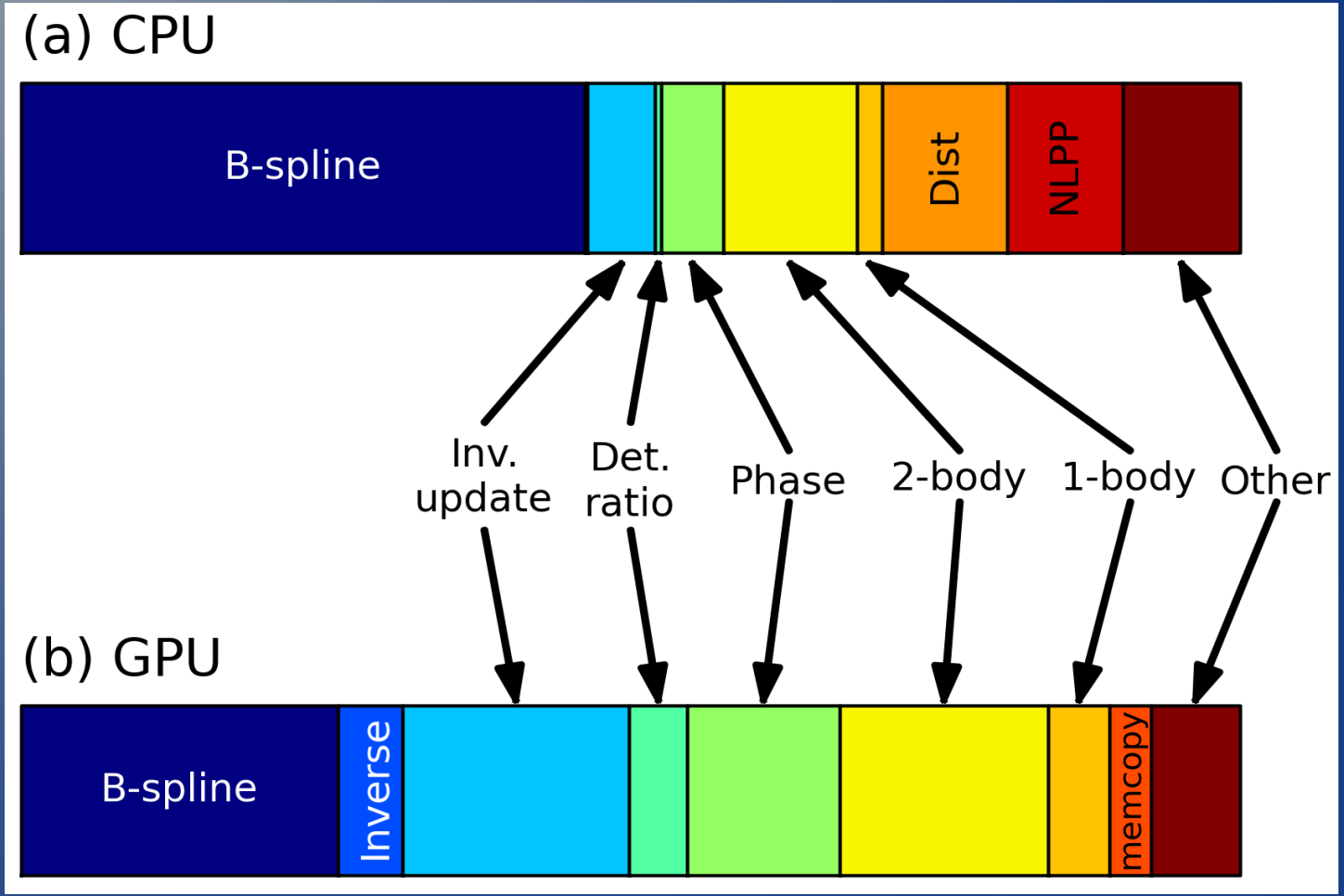
NCSA

```
template<typename T> __global__ void
two_body_update_PBC_kernel (T **R, int N, int iat)

template<typename T, int BS> __global__ void
two_body_grad_lapl_PBC_kernel(T **R, int e1_first, int e1_last, int e2_first,
                              int e2_last, T *spline_coefs, int numCoefs, T rMax,
                              T *lattice, T *latticeInv, T *gradLapl, int row_stride)

template<typename T, int BS> __global__ void
two_body_grad_lapl_PBC_kernel_fast(T **R, int e1_first, int e1_last, int e2_first,
                                   int e2_last, T *spline_coefs, int numCoefs, T rMax,
                                   T *lattice, T *latticeInv, T *gradLapl,
                                   int row_stride)

template<typename T, int BS> __global__ void
two_body_grad_PBC_kernel(T **R, int first, int last, int iat,
                         T *spline_coefs, int numCoefs, T rMax,
                         T *lattice, T *latticeInv, bool zeroOut, T *grad)

template<typename T, int BS> __global__ void
two_body_grad_PBC_kernel_fast(T **R, int first, int last, int iat,
                              T *spline_coefs, int numCoefs, T rMax,
                              T *lattice, T *latticeInv, bool zeroOut, T *grad)

template<typename T, int BS> __global__ void
two_body_derivs_PBC_kernel(T **R, T **gradLogPsi, int e1_first, int e1_last,
                           int e2_first, int e2_last, int numCoefs, T rMax,
                           T *lattice, T *latticeInv, T **derivs)

template<typename T, int BS > __global__ void
one_body_sum_PBC_kernel(T *C, T **R, int cfirst, int clast, int efirst, int elast,
                        T *spline_coefs, int numCoefs, T rMax,
                        T *lattice, T *latticeInv, T *sum)

template<typename T, int BS> __global__ void
one_body_ratio_PBC_kernel(T *C, T **R, int cfirst, int clast, T *Rnew, int inew,
                          T *spline_coefs, int numCoefs, T rMax,
                          T *lattice, T *latticeInv, T *sum)

template<typename T, int BS>
__global__ void one_body_ratio_grad_PBC_kernel(T *C, T **R, int cfirst, int clast,
                                               T *Rnew, int inew, T *spline_coefs,
                                               int numCoefs, T rMax, T *lattice,
                                               T* latticeInv, bool zero, T *ratio_grad)

template<typename T, int BS> __global__ void
one_body_ratio_grad_PBC_kernel_fast(T *C, T **R, int cfirst, int clast, T *Rnew,
                                    int inew, T *spline_coefs, int numCoefs, T rMax,
                                    T *lattice, T *latticeInv, bool zero, T *ratio_grad)

template<typename T> __global__ void
one_body_update_kernel (T **R, int N, int iat)
```

```
template<typename T, int BS> __global__ void
one_body_grad_lapl_PBC_kernel
(T *C, T **R, int cfirst, int clast, int efirst, int elast, T *spline_coefs,
 int numCoefs, T rMax, T *lattice, T* latticeInv, T *gradLapl, int row_stride)

template<int BS> __global__ void
one_body_NLratio_PBC_kernel
  (NLjobGPU<float> *jobs, float *C, int first, int last, float *spline_coefs,
   int numCoefs, float rMax, float *lattice, float *latticeInv)

template<int BS> __global__ void
one_body_NLratio_PBC_kernel_fast
  (NLjobGPU<float> *jobs, float *C, int first, int last, float *spline_coefs,
   int numCoefs, float rMax, float *lattice, float *latticeInv)

template<int BS> __global__ void
one_body_NLratio_PBC_kernel
  (NLjobGPU<double> *jobs, double *C, int first, int last, double *spline_coefs,
   int numCoefs, double rMax, double *lattice, double *latticeInv)

template<typename T, int BS> __global__ void
one_body_grad_PBC_kernel(T **R, int iat, T *C, int first, int last,
                         T *spline_coefs, int numCoefs, T rMax,
                         T *lattice, T *latticeInv, bool zeroOut, T* grad)

template<typename T, int BS> __global__ void
one_body_grad_PBC_kernel_fast(T **R, int iat, T *C, int first, int last,
                              T *spline_coefs, int numCoefs, T rMax,
                              T *lattice, T *latticeInv, bool zeroOut, T *grad)

template<typename T, int BS> __global__ void
one_body_derivs_PBC_kernel
  (T* C, T **R, T **gradLogPsi, int cfirst, int clast,  int efirst, int elast,
   int numCoefs, T rMax, T *lattice, T *latticeInv,  T **derivs)
```

NCSA

Kernel classes



NCSA

The Trial Wavefunction

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

$$\det(\mathbf{R}) = \det\begin{vmatrix} \phi_1(\mathbf{r}_1) & \phi_2(\mathbf{r}_1) & \cdots & \phi_N(\mathbf{r}_1) \\ \phi_1(\mathbf{r}_2) & \phi_2(\mathbf{r}_2) & \cdots & \phi_N(\mathbf{r}_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_1(\mathbf{r}_N) & \phi_2(\mathbf{r}_N) & \cdots & \phi_N(\mathbf{r}_N) \end{vmatrix}$$

Pauli exclusion

$$J_1 = \sum_{i,j} u_1\left(|\mathbf{r}_i - \mathbf{I}_j|\right)$$

electron-nucleus

$$J_2 = \sum_{i \neq j} u_2\left(|\mathbf{r}_i - \mathbf{r}_j|\right)$$

electron-electron

NCSA

Determinant ratios

$$\det(\mathbf{R}') = \det \begin{vmatrix} \phi_1(\mathbf{r}_1) & \phi_2(\mathbf{r}_1) & \cdots & \phi_N(\mathbf{r}_1) \\ \phi_1(\mathbf{r}'_2) & \phi_2(\mathbf{r}'_2) & \cdots & \phi_N(\mathbf{r}'_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_1(\mathbf{r}_N) & \phi_2(\mathbf{r}_N) & \cdots & \phi_N(\mathbf{r}_N) \end{vmatrix}$$

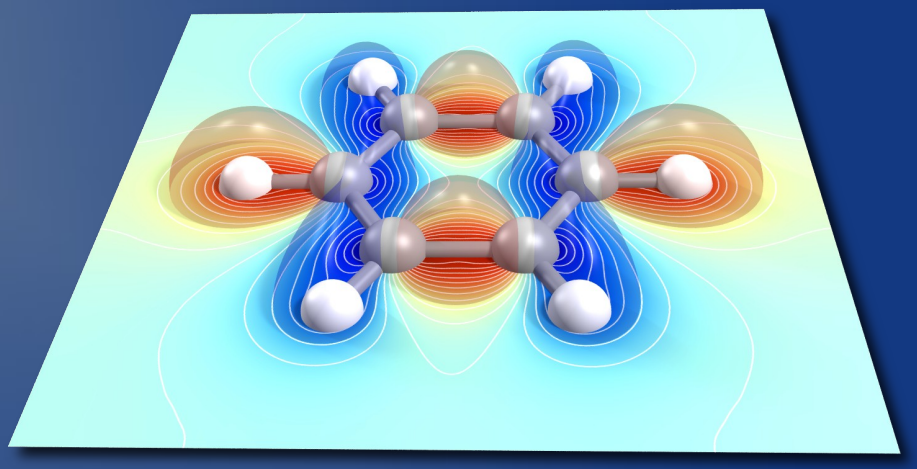
$$\det(\mathbf{R}) = \det \begin{vmatrix} \phi_1(\mathbf{r}_1) & \phi_2(\mathbf{r}_1) & \cdots & \phi_N(\mathbf{r}_1) \\ \phi_1(\mathbf{r}_2) & \phi_2(\mathbf{r}_2) & \cdots & \phi_N(\mathbf{r}_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_1(\mathbf{r}_N) & \phi_2(\mathbf{r}_N) & \cdots & \phi_N(\mathbf{r}_N) \end{vmatrix}$$

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

NCSA

Electronic orbitals

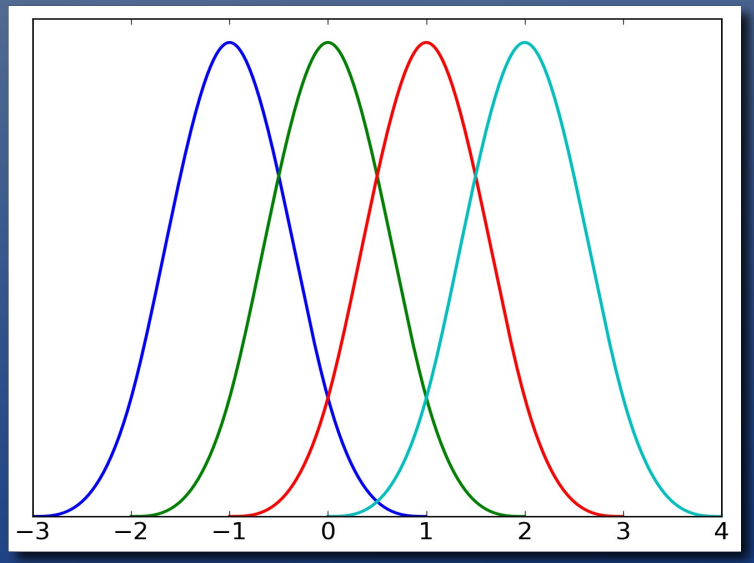
- Orbital: general function of (x,y,z)
- Represented in a linear basis
- N electrons → N/2 occupied orbitals
- Evaluate all orbitals at the same time for a single (x,y,z)



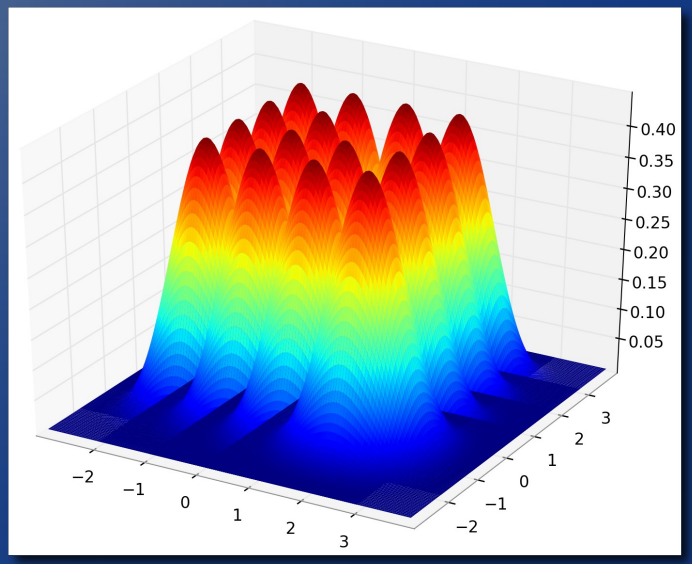
NCSA

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



1D



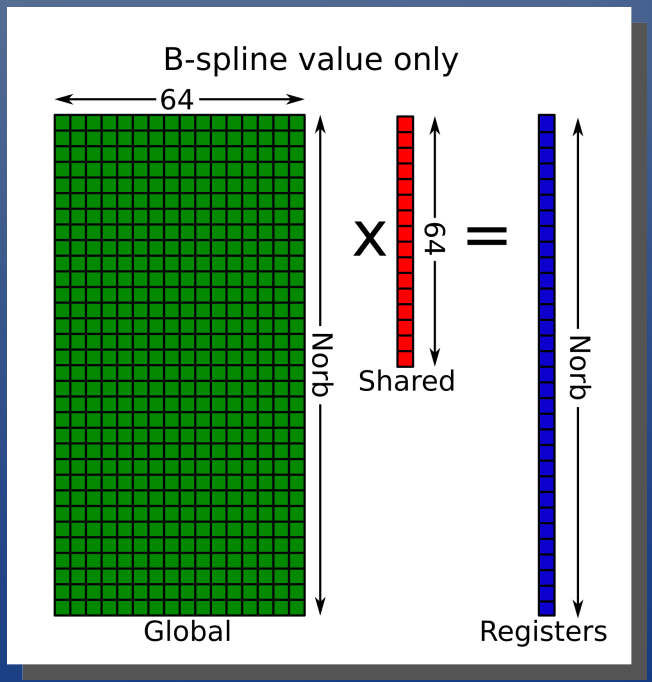
2D

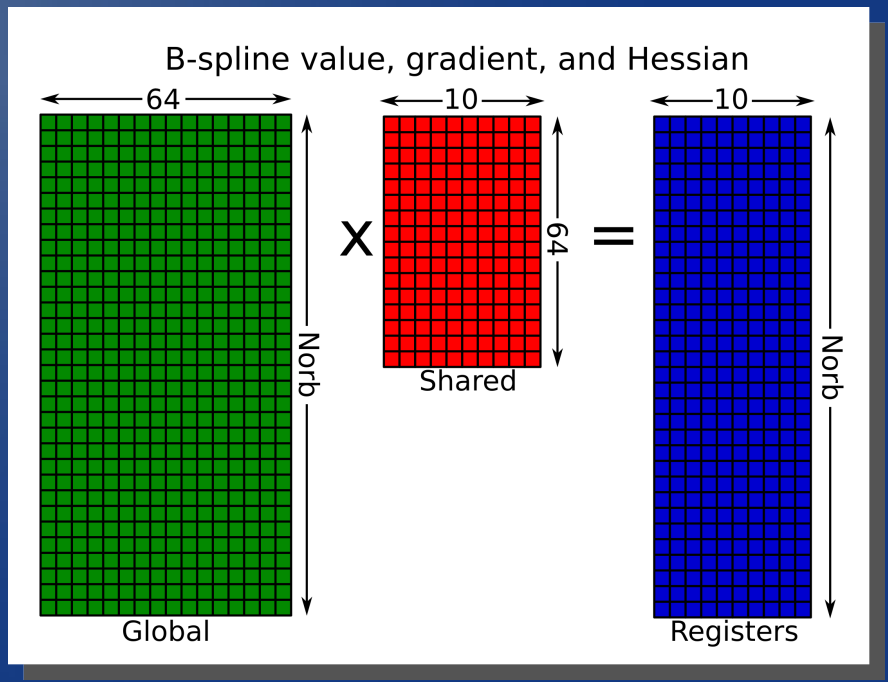
NCSA

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

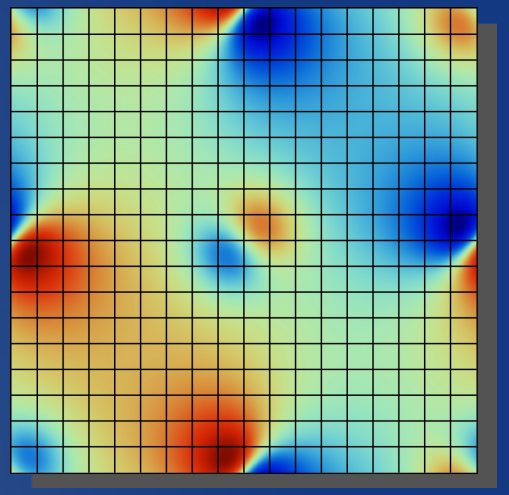
~256 walkers x



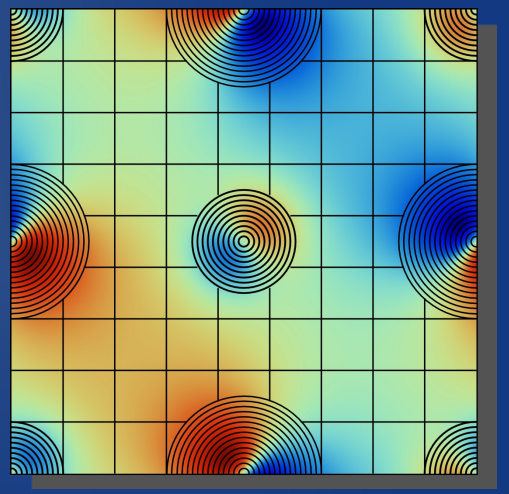


Overcoming memory limitations

- 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



Uniform B-splines



Hybrid storage

NCSA

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

$$\left[A + e_k \delta^T\right]^{-1} = A^{-1} - \frac{A^{-1} e_k \delta^T A^{-1}}{1 + \delta^T A^{-1} e_k},$$

matvec

outer product

- 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

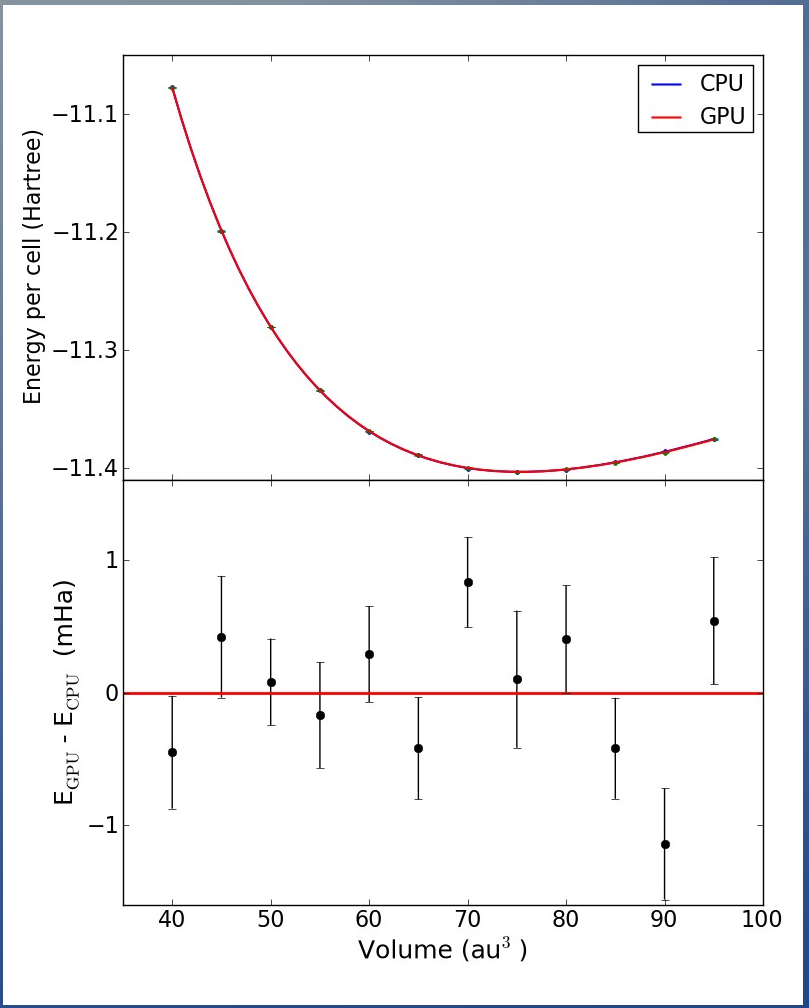
NCSA

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

NCSA

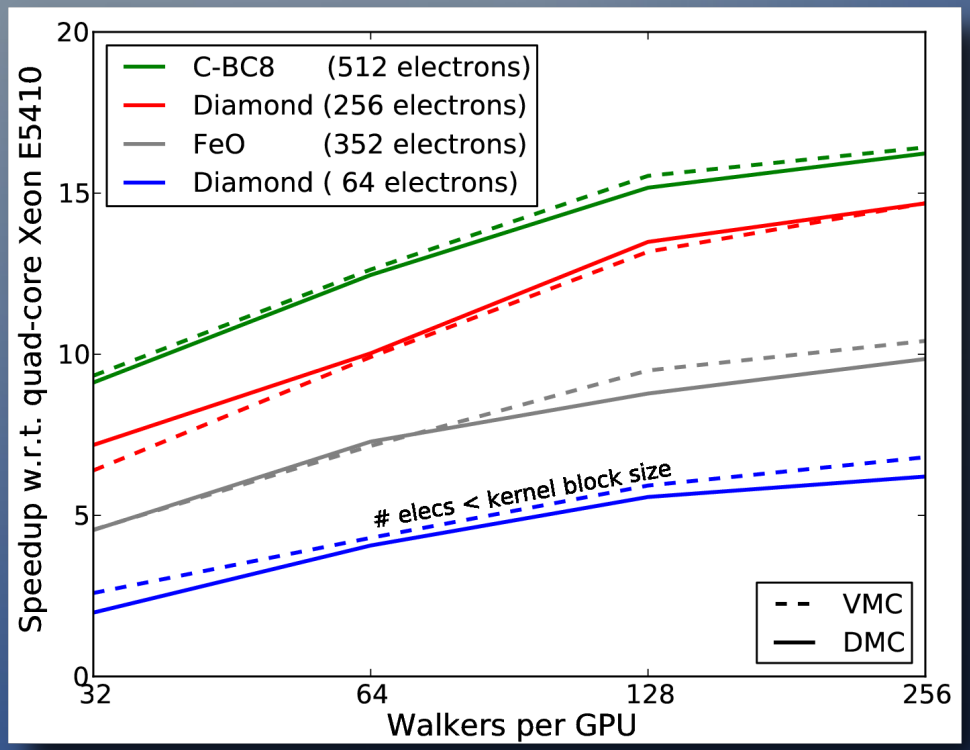
Results: Accuracy



- 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

NCSA

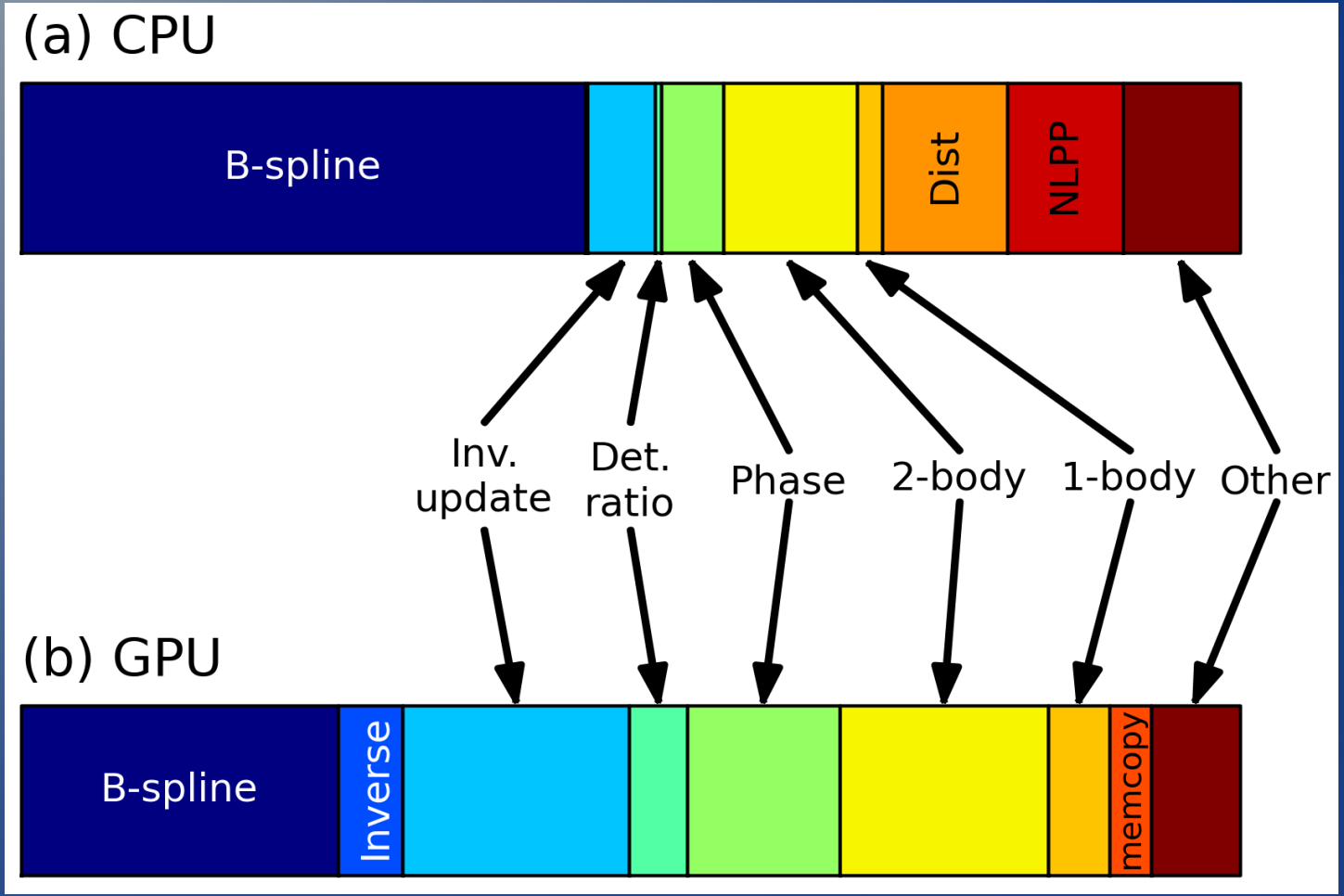
Results: Performance



- 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

NCSA

Performance analysis



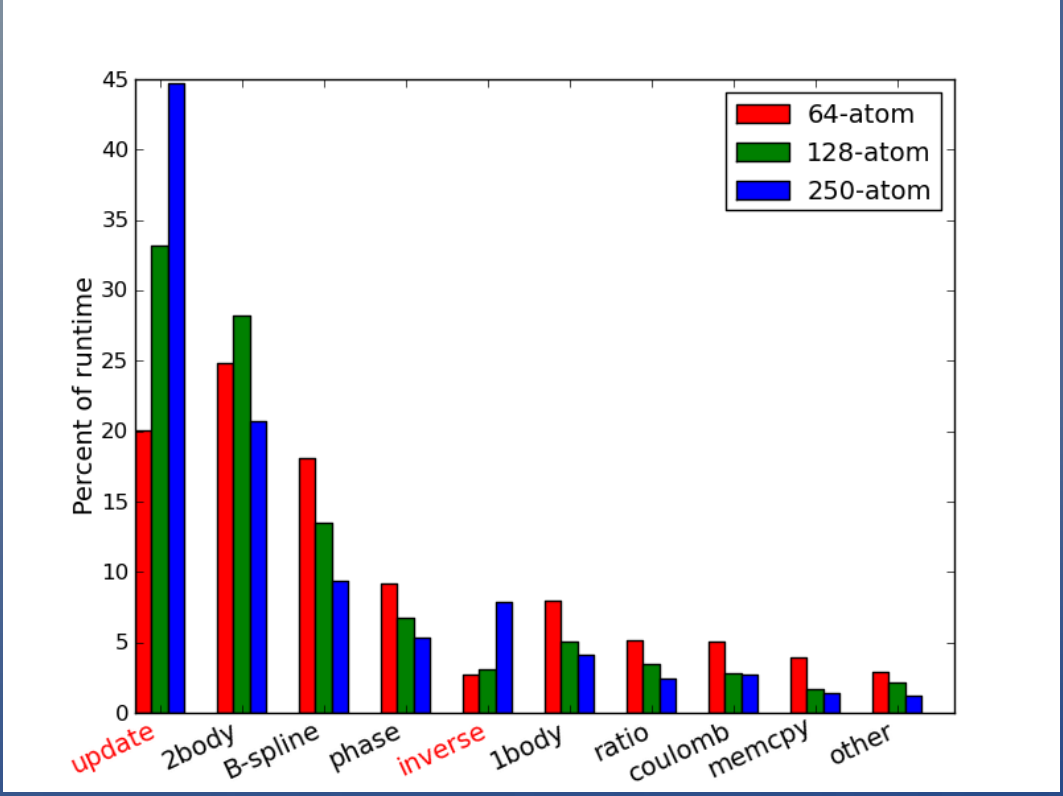
NCSA

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

NCSA

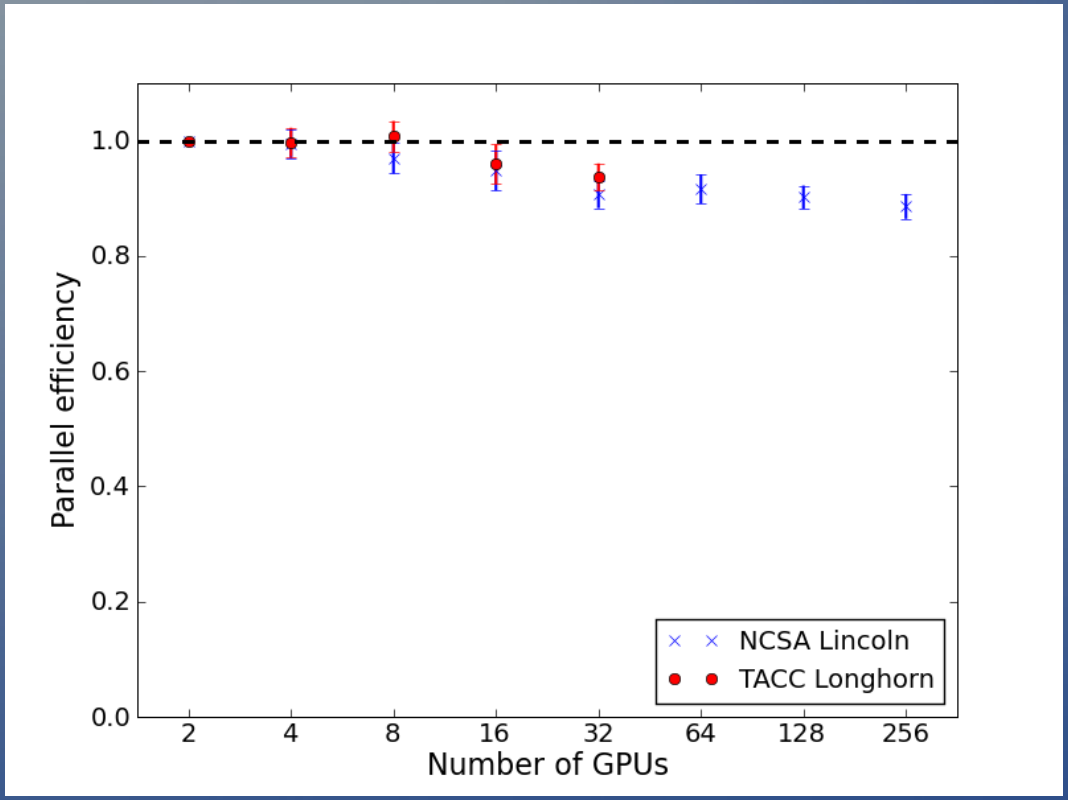
Scaling with system size



- 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

NCSA

Scaling with the number of GPUs

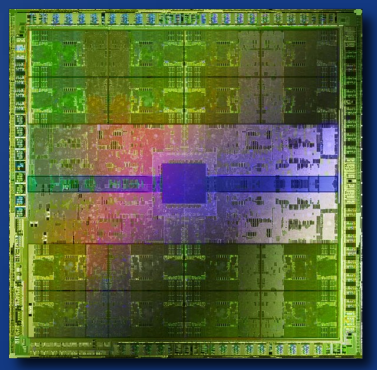


- Higher throughput on GPUs puts greater demand on interconnect
- Down to 88% efficiency on Lincoln at 256 GPUs
- Slightly better on Longhorn?
- PCI-E or IB limited?

NCSA

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



NCSA

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

NCSA

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

NCSA

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

NCSA

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

NCSA

Quantum Chemistry Methods

- Expand Ψ in a linear basis
  - For exact solution, size of basis ~ 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

NCSA