OpenMPCon

Argonne NATIONAL LABORATORY

THE PAST, PRESENT AND FUTURE OF QMCPACK WITH OPENMP

QMCPACK

Ye Luo[1], Anour Benali[1], Jeongnim Kim[2], Paul R.C. Kent[3]

1, Argonne Leadership computing facility
2, Intel Corporation
3, Oak Ridge National Laboratory

Stony Brook, Sep 19， 2017

OUTLINE

- QMC basics and QMCPACK
- OpenMP introduced from the beginning
- Performance portable on CPUs via OpenMP 4.0 simd
- Experimenting OpenMP 4.5 offloading for the future

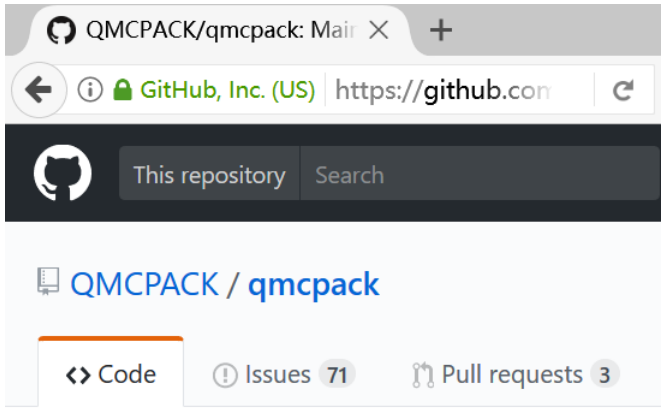
- This work is supported by Intel Corporation to establish the Intel Parallel Computing Center at Argonne National Laboratory.
- This research has used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.
- This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.
- This research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation's exascale computing imperative.

QUANTUM MONTE CARLO BASICS

Argonne NATIONAL LABORATORY

QMC AND QMCPACK

- Quantum Monte Carlo is not a single method.
- Quantum mechanics + Monte Carlo algorithms
- QMCPACK, is a modern high-performance open-source Quantum Monte Carlo (QMC) simulation code. Its main applications are electronic structure calculations of molecular, quasi-2D and solid-state systems.
- QMCPACK is C/C++, MPI+X(OpenMP, CUDA)
- At qmcpack.org and public @ github
- ECP application development award



VARIATIONAL MONTE CARLO (VMC)

From the variational principle.

- The goal is to solve Schrodinger equation with Monte Carlo technique.

$$\langle E \rangle = \frac{\int \mathrm{d}R \; \Psi^*_{\mathrm{T},\alpha} H \Psi_{\mathrm{T},\alpha}}{\int \mathrm{d}R \; |\Psi_{\mathrm{T},\alpha}|^2} \; ,$$

- Monte Carlo methods can be used to evaluate multi-dimensional integrals much more efficiently than deterministic methods.

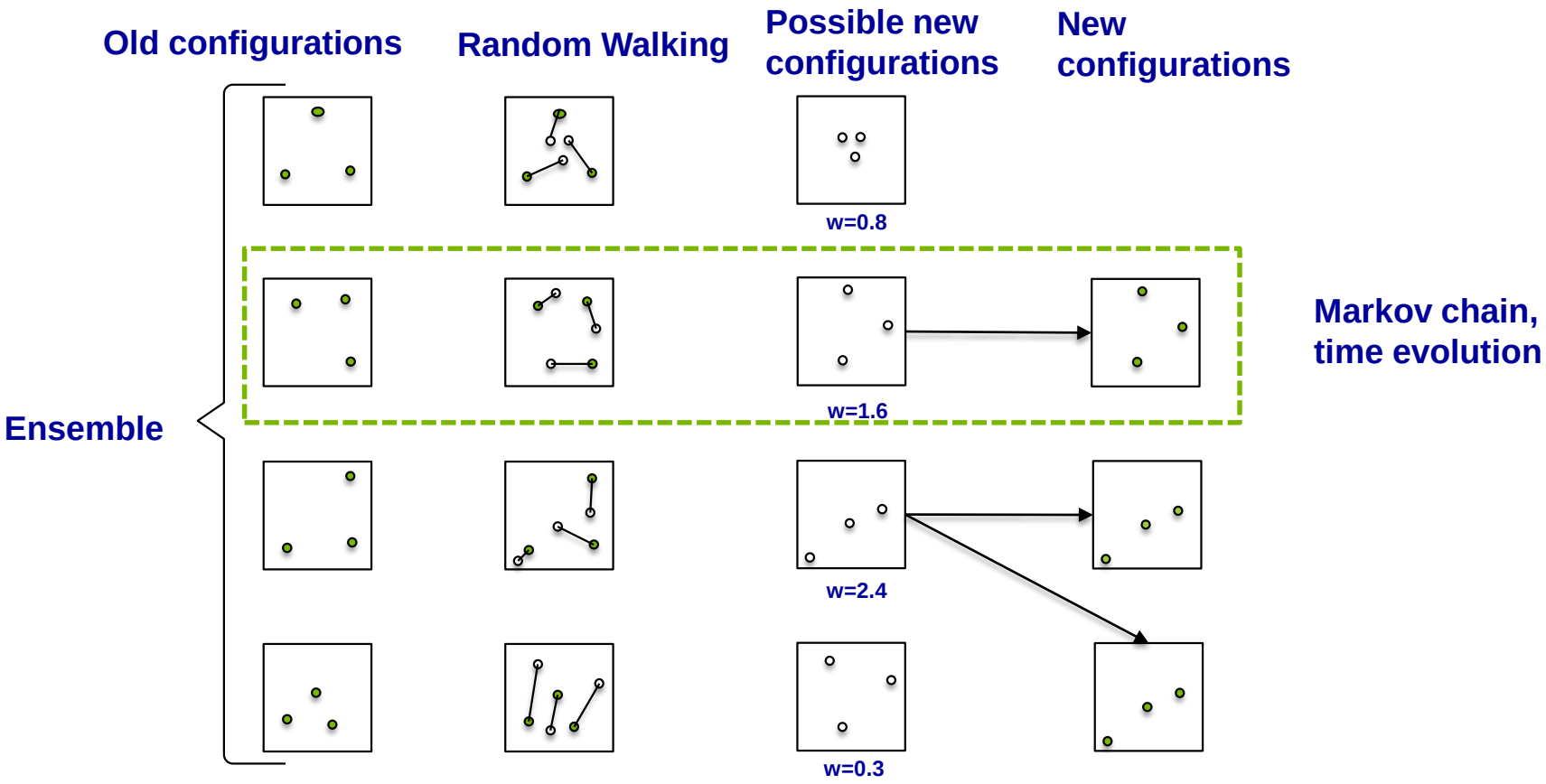
$$\rho(R) = \frac{|\Psi_{\mathrm{T},\alpha}(R)|^2}{\int \mathrm{d}R \; |\Psi_{\mathrm{T},\alpha}|^2} \; .$$

$$E_L(R) = \frac{\hat{H}\Psi_T(R)}{\Psi_T(R)}$$

- The random walking is performed by many walkers on individual Markov Chains.

$$\langle E \rangle = \sum_R E_L(R)$$

DIFFUSION MONTE CARLO(DMC) SCHEMATICS

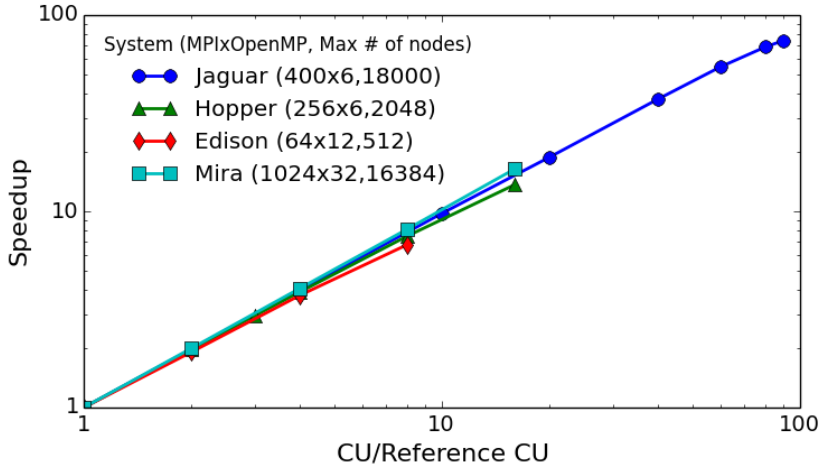


Argonne NATIONAL LABORATORY

DMC ALGORITHM

Multiple levels of parallelism can be exploited.

1: **for** MC generation $= 1 \cdots M$ **do**
2: **for** walker $= 1 \cdots N_w$ **do**
3: let $\mathbf{R} = \{\mathbf{r}_1 \ldots \mathbf{r}_N\}$
4: **for** particle $i = 1 \cdots N$ **do**
5: set $\mathbf{r}_i^{'} = \mathbf{r}_i + \delta$
6: let $\mathbf{R}^{'} = \{\mathbf{r}_1 \ldots \mathbf{r}_i^{'} \ldots \mathbf{r}_N\}$
7: **ratio** $\rho = \Psi_T(\mathbf{R}^{'})/\Psi_T(\mathbf{R})$
8: **derivatives** $\nabla_i \Psi_T, \nabla_i^2 \Psi_T$
9: **if** $\mathbf{r} \rightarrow \mathbf{r}'$ is accepted **then**
10: update state of a walker
11: **end if**
12: **end for**{particle}
13: **local energy** $E_L = \hat{H}\Psi_T(\mathbf{R})/\Psi_T(\mathbf{R})$
14: reweight and branch walkers
15: **end for**{walker}
16: update E_T and load balance
17: **end for**{MC generation}

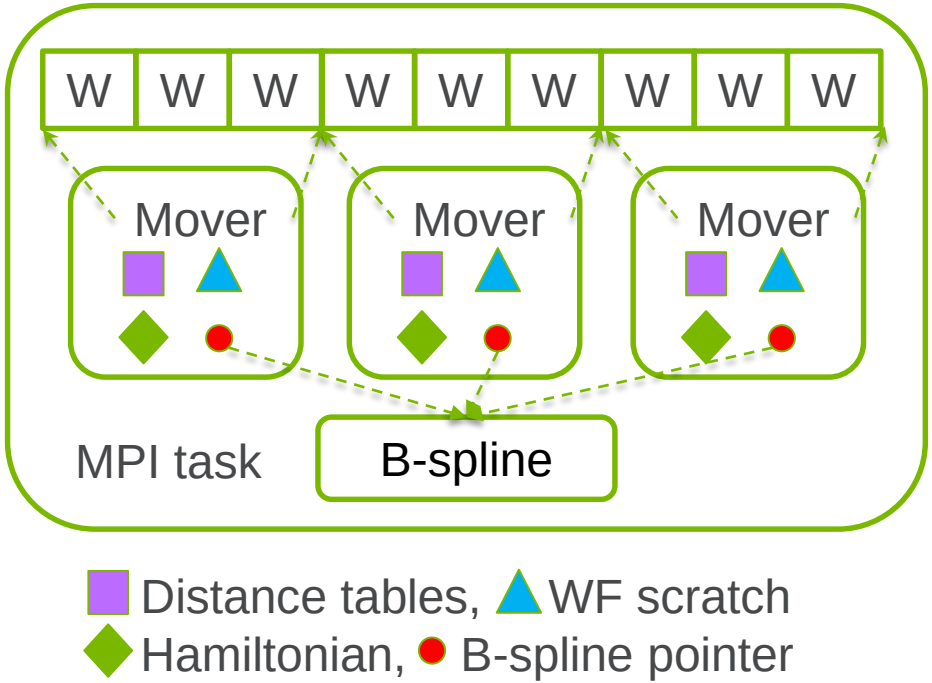


- Loop 2: walkers (~10 per node) are distributed both over MPI and cores/SMs using OpenMP and CUDA.
- Loop 2 and 4 are interchanged on GPU.
- Steps 6,7,8 have extra particle (~1k) concurrency, exposed to GPU threads and CPU SIMD.

WALKER LEVEL PARALLELISM (COARSE)

SINGLE-NODE MANAGEMENT

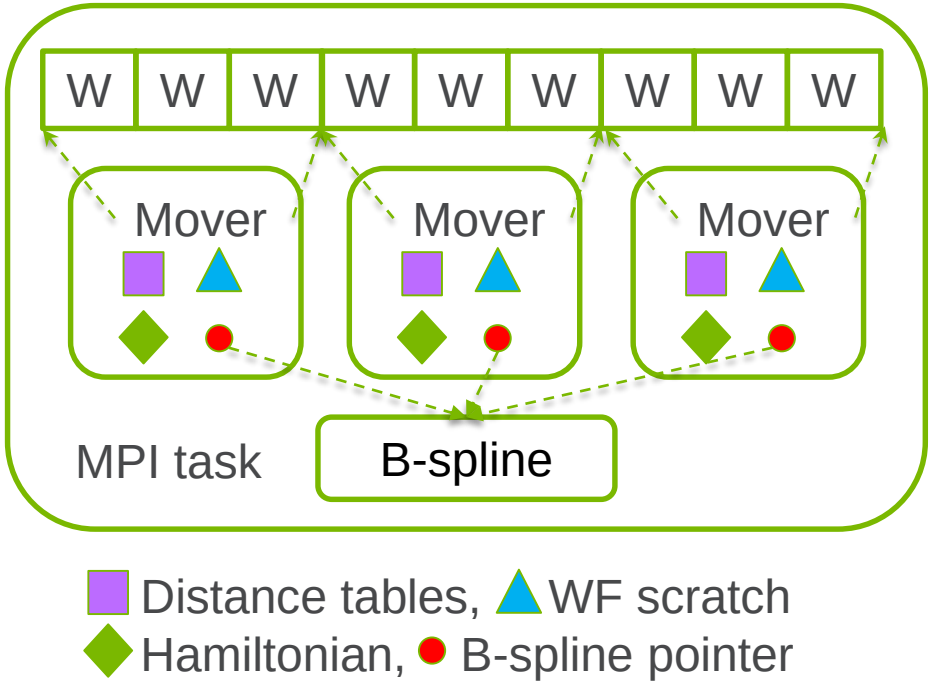
Ensure data locality



- Walkers carry minimum data
- A high level #omp parallel
- One mover on each thread holding scratch data
- #omp for over the walker loop
- Only one synchronization at the end of parallel region
- Near perfect on-node weak scaling

SINGLE-NODE MANAGEMENT

Need OpenMP for large shared data



- Single particle orbitals are stored in memory and replicated on every node for fast evaluation.
- 20-40% computational cost.
- Large size up to 1~100x GB memory
- Read-only, initialized once.
- Frequent random access.
- OpenMP shared memory model

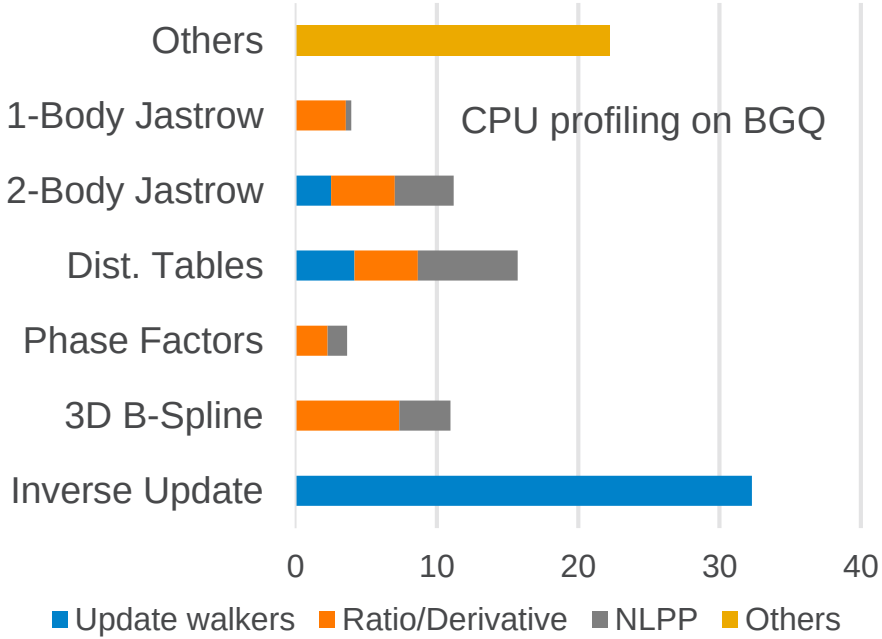
PARTICLE LEVEL PARALLELISM (FINE)

PERFORMANCE ISSUE (BGQ/KNL)

A lot of inefficiency

- Numerical light kernels (Jastrows and Distance tables) are taking a lot of time.
- Vectorization is needed to be reworked for KNL

percentage	KNL	BGQ
Dist. Tables	26.9	15.7
3D B-Spline	12.4	10.9
Inv. Update	22.0	32.3



WHY PERFORMANCE IS LOW

SIMD efficiency is low

- All in double precision but 3D B-Spline.
- SIMD efficiency low
  - Array of Structure (AoS) for D-dim particle attributes, e.g., R (N,3), Gradients, Hessians, Matrices
  - Good OOP but not ideal for the high performance on Xeon Phi
  - Basically scalar performance with few exceptions
    - Einspline – SSE/SSE2/QPX
    - Distance tables with QPX

SSE:

```
 r2 = _mm_shuffle_pd (tmp0, tmp1,
_MM_SHUFFLE2(0, 0));
tmp0 = _mm_load_pd (P(1,2));
_MM_DDOT4_PD(r0, r1, r2, r3,     b01,    b23,
b01,    b23,    bP01r);
_MM_DDOT4_PD(r0, r1, r2, r3,    db01,   db23,
db01,   db23,   dbP01r);
```

QPX:

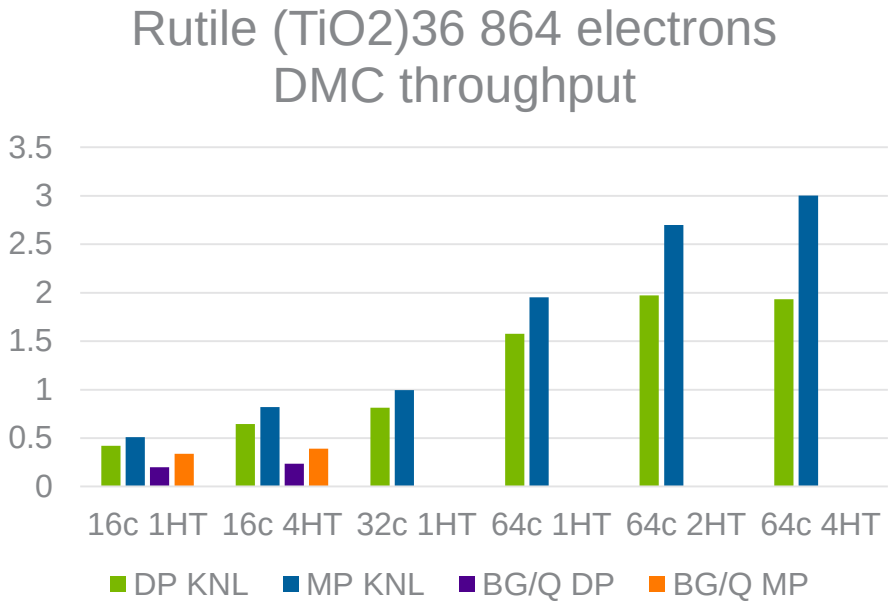
```
 __dcbt(&gx    [n+8]);
 __dcbt(&vals  [n+8]);
vector4double coef0 = vec_ld(0, &coefs0[n]);
vector4double coef1 = vec_ld(0, &coefs1[n]);
vector4double sum0, sum1, sum2;
sum0 = vec_mul (vec_c0, coef0);
sum0 = vec_madd(vec_c1, coef1, sum0);
```

MIXED PRECISION (TOP DOWN)

Gain performance not only on KNL but also on BG/Q.

- 20-55% gain on KNL
- 70% gain on BG/Q

- Should gain more with good vectorization.
- Need to handle double/float with a performance portable code.



Argonne NATIONAL LABORATORY

VECTORIZATION (BOTTOM UP)

Requirements and tools

We desire

- Old codes
  - Minimized changes
- New implementation
  - Single source
  - Both float/double types
  - No intrinsics
  - Any vector length
  - Any alignment requirement
  - Any compiler
  - Any CPU vendor

We have

- C++
  - Template
  - Operator overloading
- OpenMP 4.0
  - #pragma omp simd with aligned clause

arXiv: 1708.02645, to be published at SC17
10.1145/3126908.3126952

SOA VECTOR CLASS

Solve the compatibility issue and enables SoA data layout

```
template <typename T, unsigned D>
struct VectorSoaContainer
{
  // (size+padding)*D elements
  aligned_vector<T> X;
  // access a single struct
  TinyVector <T,D> operator[](size_t i) const;
  // access an array
  T * data(size_t i);
};
```

- Alignment is handled by aligned_vector
- Code changes

```
Vector<TinyVector<T, 3> > R, G;
Vector<TinyVector<T, 6> > H;

VectorSoaContainer<T, 3> Rsoa, Gsoa;
VectorSoaContainer<T, 6> Hsoa;
```

OPENMP SIMD CONSTRUCT

Elegant way of expressing SIMD

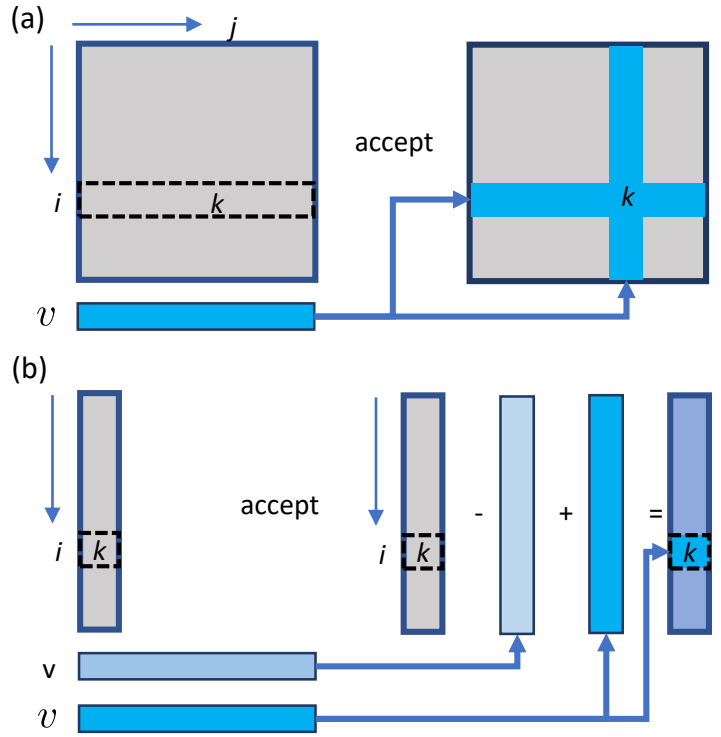
```
for(int idim=0; idim<3; ++idim)
{
  const valT* restrict new_dX=new_dr.data(idim);
  const valT* restrict old_dX=old_dr.data(idim);
  const valT* restrict cur_du_pt=cur_du.data();
  const valT* restrict old_du_pt=old_du.data();
  valT* restrict save_g=dUat.data(idim);
  #pragma omp simd aligned(old_dX,new_dX,…)
  for(int jat=0; jat<N; jat++)
  {
    const valT newg = cur_du_pt[jat] * new_dX[jat];
    const valT dg   = newg - old_du_pt[jat]*old_dX[jat];
    save_g[jat]  -= dg;
  }
}
```

- No alignment detail needed
- Only vector load/store in assembly
- Flexible to vector length

TWO BODY JASTROW FACTOR

Use an update algorithm

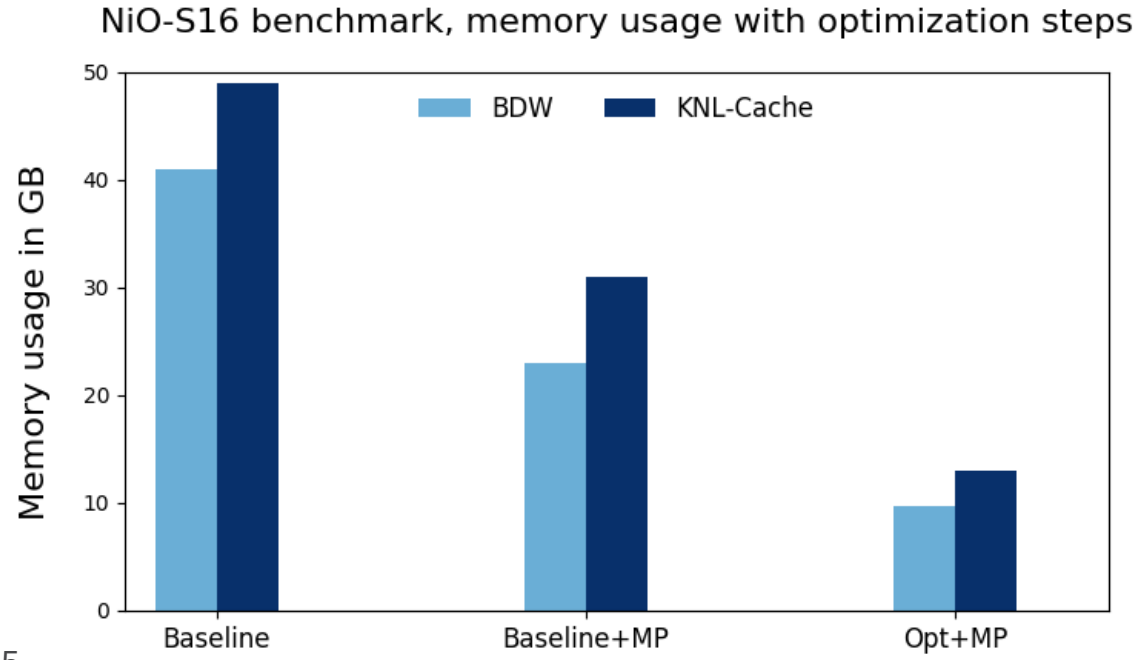
- Two body potential $U(i,j)$
- Only need $\sum_j U(i,j)$
- Memory footprint N^2=>N
- Still N^2 complexity but smaller prefactor
- All vectorizable computation



Argonne NATIONAL LABORATORY

NIO 64 ATOM BENCHMARK

A huge save in memory footprint, 49GB=>13GB

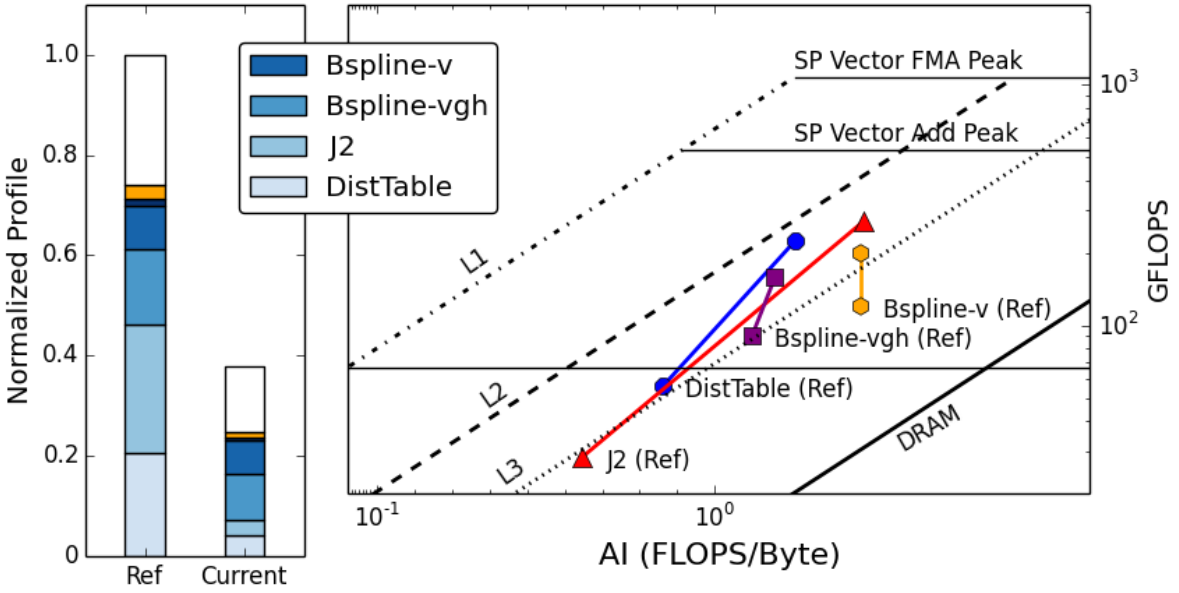


arXiv: 1708.02645,
to be published at SC17 10.1145/3126908.3126952

ROOFLINE ANALYSIS

Significant gain in the distance table and the Jastrow

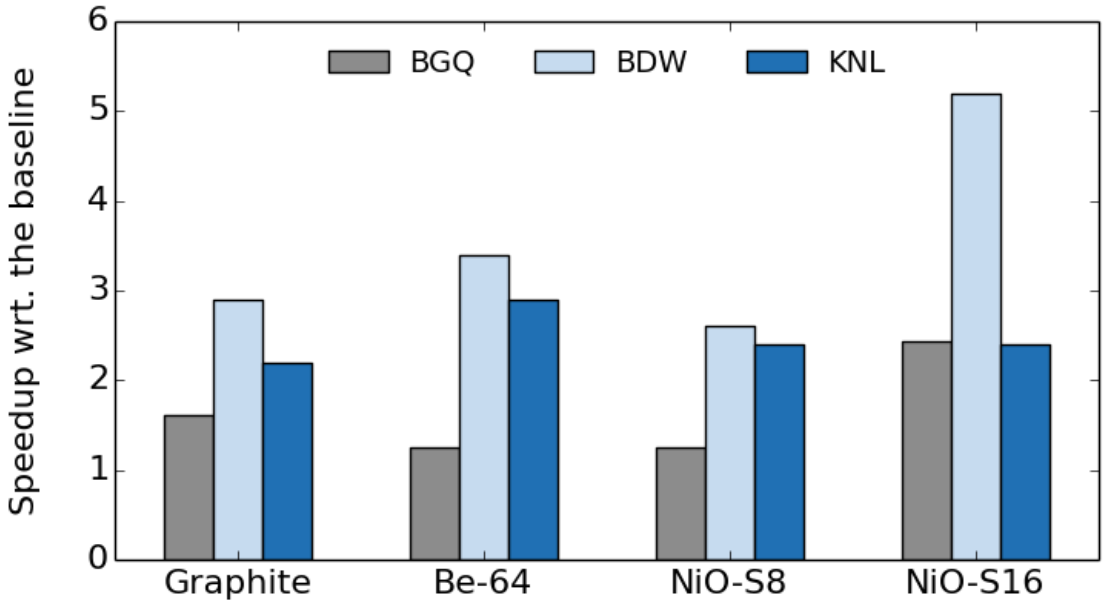
- On BDW, only DDR
- Drop from 47% to 8%
- Huge improvement in algorithmic intensity
- Everything in L3



arXiv: 1708.02645,
to be published at SC17 10.1145/3126908.3126952

BENCHMARK

Significant speed on all the platforms



arXiv: 1708.02645,
to be published at SC17 10.1145/3126908.3126952

SUMMARY

- OpenMP solves memory issue
- It expresses our on-node parallelization including threading and vectorization
- It enables very clean and understandable code
- It gives perfect thread scaling and SIMD efficiency

- How about accelerators?

EXPERIMENTING OPENMP OFFLOAD

WHY WE NEED OMP OFFLOAD

- In the past
  - Two large fork for CPU and GPU(CUDA)
  - Largely incompatible, datatypes/dataflow
  - Lack of developers and hard to implement both
  - GPU code features only parts of functionality
- In long term
  - Need a portable solution
  - Not depend on proprietary solutions
  - Portable performance is desired

PERFORMANCE PORTABILITY

Assess OpenMP

On multiple architectures including CPU and GPU.

1. Capability: Can we express the required parallelism?
2. Performance: Can we achieve good performance?
3. Portability: What is the extent of required changes?
4. Support: How are compilers, libraries, tools?

MINIQMC

Via miniapp route

- We have accumulated a set of miniapps during SoA optimization.
- They are
  - Stand-alone separated from QMCPACK.
  - Expressing the same concurrency as QMCPACK.
  - Using state-of-art algorithms.
- To collaborate with non-QMCPACK developers
- Become public on github by the end of Sep. 2017

3D CUBIC B-SPLINE KERNEL

Initial version (v0)

```
#pragma omp parallel
{
 T x,y,z;  T *v, *g, *h;
 #pragma omp for
 for(size_t iw=0; iw<nw; iw++)
  for(size_t iel; iel<Nel; iel++)
    MultiBspline ::VGH(x,y,z,v,g,h);
}
```

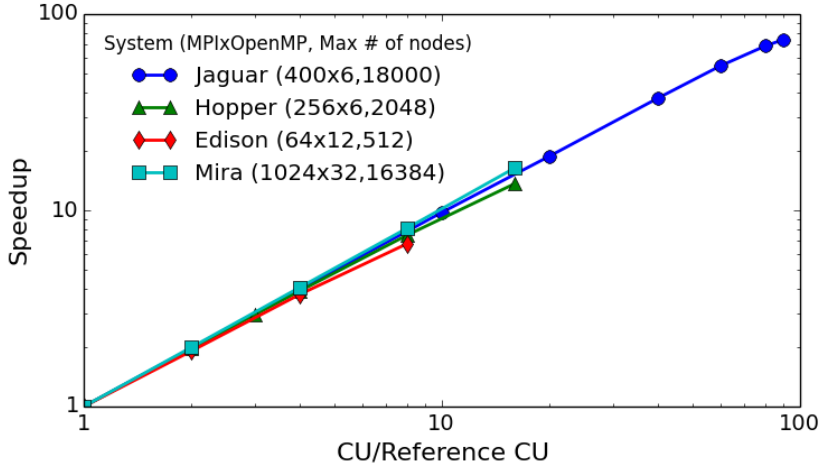
- Two level parallelism.
- MultiBspline computes N orbital values at a given electron coordinates
- Inner loop iel is serial

```
Class MultiBspline {
 T b[Nx][Ny][Nz][N]
 void VGH(T x, T y, T z, T* v, T* g, T *h)
 {
   // compute the lower-bound x0, y0, z0
   // compute prefectors using (x-x0, y-y0, z-z0)
   for(size_t i,j,k=0; i,j,k<4; i,j,k++)
     #pragma omp simd
     for(size_t n=0; n<N; n++) {
       v[n]+=F(b[n]);
       gx[n]+=Gx(b[n]); …
       hxx[n]+=Hxx(b[n]); …
     } } }
```

DMC ALGORITHM

Multiple levels of parallelism can be exploited.

1: **for** MC generation $= 1 \cdots M$ **do**
2: **for** walker $= 1 \cdots N_w$ **do**
3: let $\mathbf{R} = \{\mathbf{r}_1 \ldots \mathbf{r}_N\}$
4: **for** particle $i = 1 \cdots N$ **do**
5: set $\mathbf{r}_i^{'} = \mathbf{r}_i + \delta$
6: let $\mathbf{R}^{'} = \{\mathbf{r}_1 \ldots \mathbf{r}_i^{'} \ldots \mathbf{r}_N\}$
7: **ratio** $\rho = \Psi_T(\mathbf{R}^{'})/\Psi_T(\mathbf{R})$
8: **derivatives** $\nabla_i \Psi_T, \nabla_i^2 \Psi_T$
9: **if** $\mathbf{r} \rightarrow \mathbf{r}'$ is accepted **then**
10: update state of a walker
11: **end if**
12: **end for**{particle}
13: **local energy** $E_L = \hat{H}\Psi_T(\mathbf{R})/\Psi_T(\mathbf{R})$
14: reweight and branch walkers
15: **end for**{walker}
16: update E_T and load balance
17: **end for**{MC generation}



- Loop 2: walkers (~10 per node) are distributed both over MPI and cores/SMs using OpenMP and CUDA.
- Loop 2 and 4 are interchanged on GPU.
- Steps 6,7,8 have extra particle (~1k) concurrency, exposed to GPU threads and CPU SIMD.

3D CUBIC B-SPLINE KERNEL

Loop interchanged (v1)

```
for(size_t iel; iel<Nel; iel++)
{
 T* x,y,z;  T **v, **g, **h;
 #pragma omp target teams distribute \
  num_teams(nw)
 //#pragma omp parallel for
 for(size_t iw=0; iw<nw; iw++)
  #pragma omp parallel num_threads(N)
  MultiBspline ::VGH(x[iw],y[iw],z[iw],
                     v[iw],g[iw],h[iw]);
}
```

```
Class MultiBspline {
 T b[Nx][Ny][Nz][N]
 void VGH(T x, T y, T z, T* v, T* g, T *h)
 {
   for(size_t i,j,k=0; i,j,k<4; i,j,k++)
    #pragma omp for nowait
    //#pragma omp simd
    for(size_t n=0; n<N; n++) {
     v[n]+=F(b[n]);
     gx[n]+=Gx(b[n]); …
     hxx[n]+=Hxx(b[n]); …
    } } }
```

DATA MAPPING TO DEVICE

Hide with class

- Vector type is the most used datatype in QMCPACK.

Each mover has
OMPVector<T> val;

mover 0
collects pointers and
handles the offload.
OMPVector<T *> val_shadows;

```
template<typename T, class Container = std::vector<T>>
class OMPVector:public Container
{
 T * vec_ptr;
 inline OMPVector(size_t size = 0): vec_ptr(nullptr) {  resize(size);  }
 inline void resize(size_t size) {
   vec_ptr = Container::data();
   #pragma omp target enter data map(alloc:vec_ptr[0:size])
 }
 inline void update_to_device() const
 {  #pragma omp target update to …  }
 inline void update_from_device() const
 {  #pragma omp target update from …  }
 inline ~OMPVector()
 {  #pragma omp target exit data map(delete:vec_ptr)  }
};
```

3D CUBIC B-SPLINE KERNEL

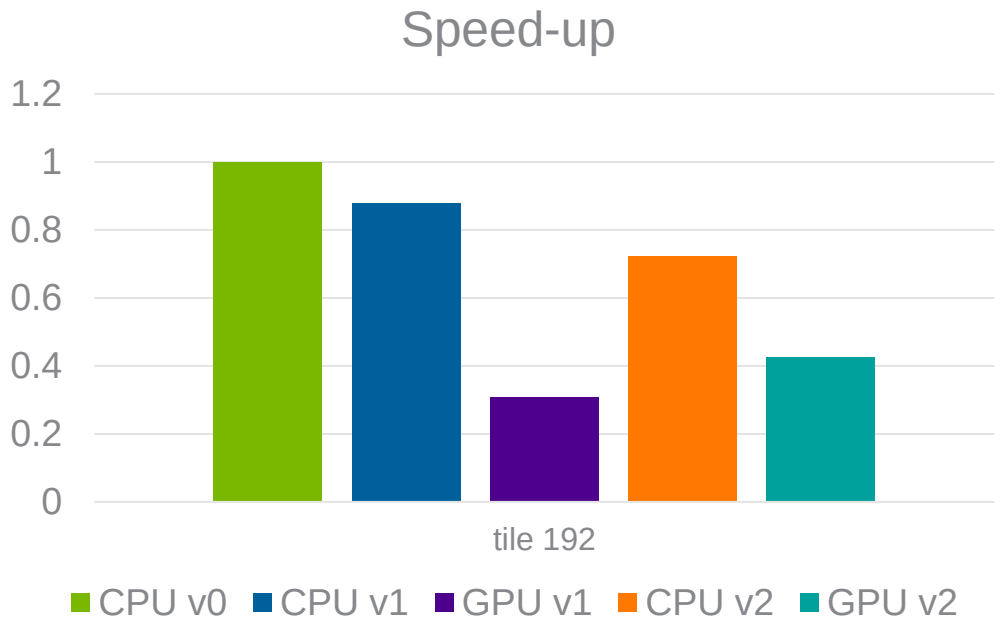
Second loop interchanged inside device region (v2)

```
Class MultiBspline {
 T b[Nx][Ny][Nz][N]
 void VGH(T x, T y, T z, T* v, T* g, T *h)
 {
   for(size_t i,j,k=0; i,j,k<4; i,j,k++)
     #pragma omp for nowait
     //#pragma omp simd
     for(size_t n=0; n<N; n++) {
       v[n]+=F(b[n]);
       gx[n]+=Gx(b[n]); …
       hxx[n]+=Hxx(b[n]); …
     } } }
```

```
Class MultiBspline {
 T b[Nx][Ny][Nz][N]
 void VGH(T x, T y, T z, T* v, T* g, T *h)
 {
   #pragma omp for nowait
   //#pragma omp simd
   for(size_t n=0; n<N; n++)
   {
    T v, gx, hxx …;
    for(size_t i,j,k=0; i,j,k<4; i,j,k++) {
      v+=F(b[n]);     gx+=Gx(b[n]); …
      hxx+=Hxx(b[n]); …
    }
    v[n]=v; gx[n]=gx …
   } } }
```

PERFORMANCE NIO 1X1X1 (SMALL)

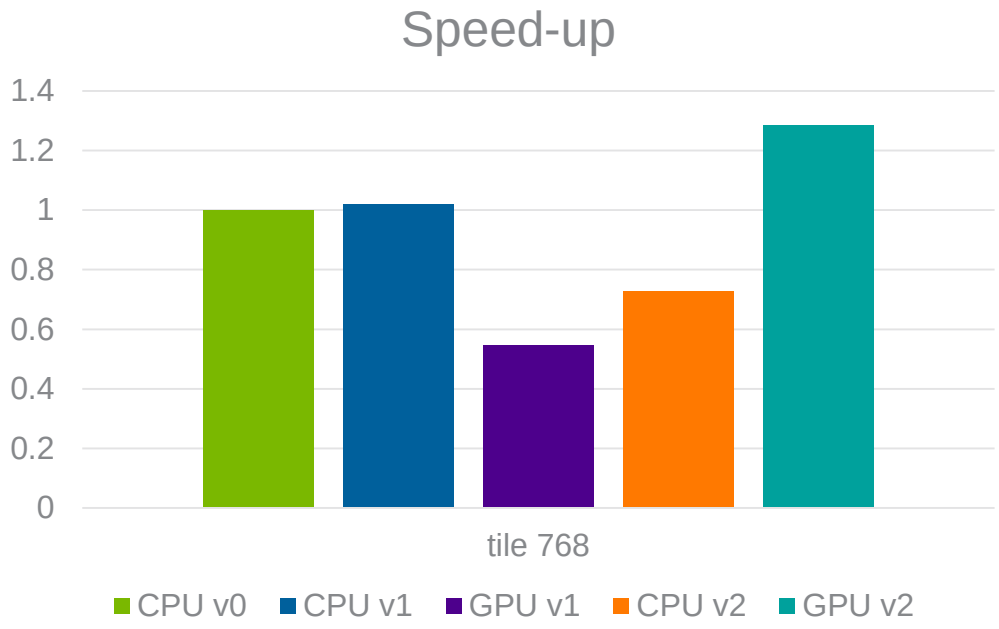
192 spline SPOs using 160 walkers, 160 threads



- IBM power 8 + pascal, Clang
- v1 is a bit slower than v0 due to fork/join overhead
- v2 on CPU is slower than v1
- v2 on GPU is faster than v1

PERFORMANCE NIO 2X2X1 (MEDIUM)

768 spline SPOs using 160 walkers, 160 threads



- IBM power 8 + pascal, Clang
- v1 fork/join overhead is negligible
- v2 on CPU is slower than v1
- v2 on GPU is faster than v1

REMARKS

- GPU performance has potential improvement
  - 142 register/thread and low occupancy 18.8%.
  - Little use of shared memory
  - Measured HBM bandwidth is only ~80GB/s far from peak.
  - How to improve?
- Single source with portable performance is not achieved on this kernel at the moment. Maybe compiler can do more?
- Complier quality is improving but takes time.
  - Application developer can help finding bugs.
  - More accessible info via compiler report.
- Limited performance tool.

PERSPECTIVE

PERSPECTIVE

OpenMP is promising

- More kernels will be attempted using OpenMP offloading. We will have better understanding of the situation.
- 80/20 rule.
  - 80% routines take 20% time.
    - Old way. Write codes for both CPU and GPU, GPU code is good enough just to avoid data transfer.
    - New way. Write a single code.
  - 20% routines take 80% time.
    - Performance portable code
    - Architecture-specific code if really necessary.