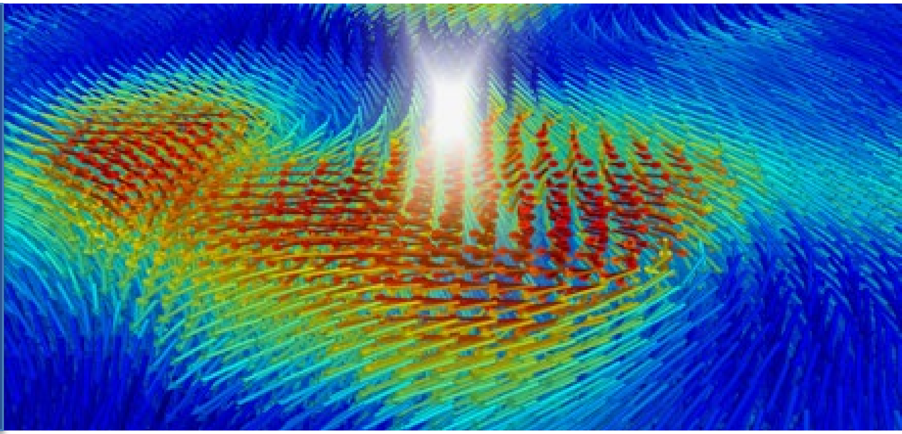
Porting RT-TDDFT codes for GPU-accelerated architectures



Taufeq Razakh
University of Southern California

Ye Luo
Argonne National Lab

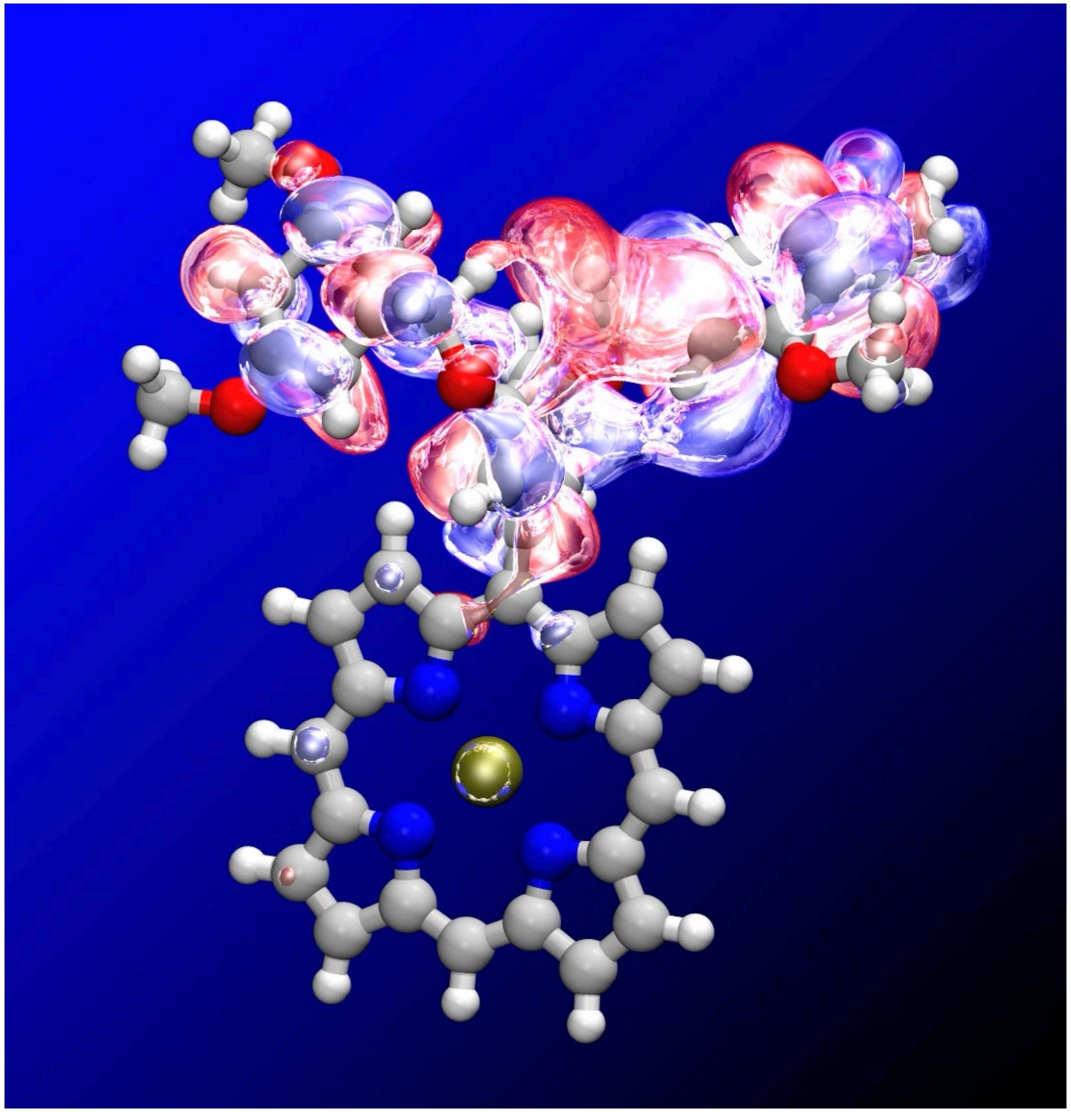
Aiichiro Nakano
University of Southern California

Outline

- Application: Quantum Dynamics Simulation
- Code Layout
- Porting Challenges
- Kernel Optimization: Case of Kinetic propagation
  - Loop Re-ordering
  - Offload Strategy
- BLASification: Non-local exchange-correlation computation
- Future Directions

- Achieved offload on Polaris and Sunspot with Clang/icpx
- Achieved 640X Speed-up on Polaris Blades at ANL

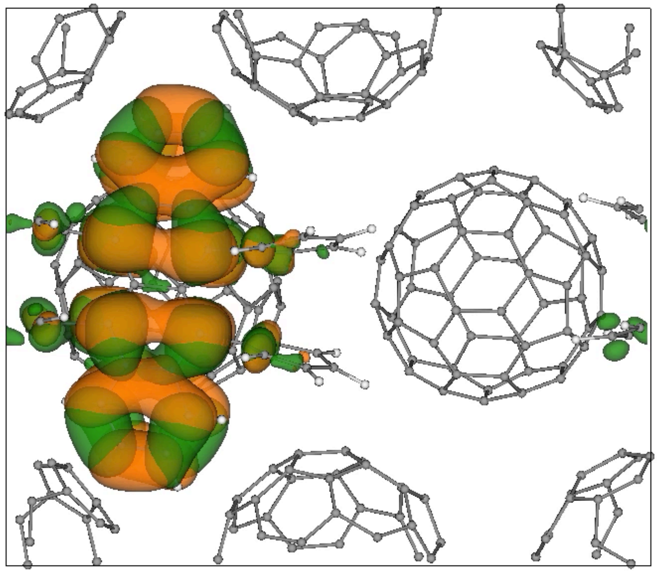
Non-adiabatic Quantum Molecular Dynamics



Appl. Phys. Lett. **98**, 113301 ('11); *ibid*. **100**, 203306 ('12); *J. Chem. Phys*. **136**, 184705 ('12); *Comput. Phys. Commun*. **184**, 1 ('13); *Appl. Phys. Lett*. **102**, 093302 ('13); *ibid*. **102**, 173301 ('13); *J. Chem. Phys*. **140**, 18A529 ('14); *IEEE Computer* **48(11)**, 33 ('15); *Sci. Rep*. **5**, 19599 ('16); *Nature Commun*. **8**, 1745 ('17); *Nano Lett*. **18**, 4653 ('18); *Nature Photon.* **13**, 425 ('19)

Zn porphyrin

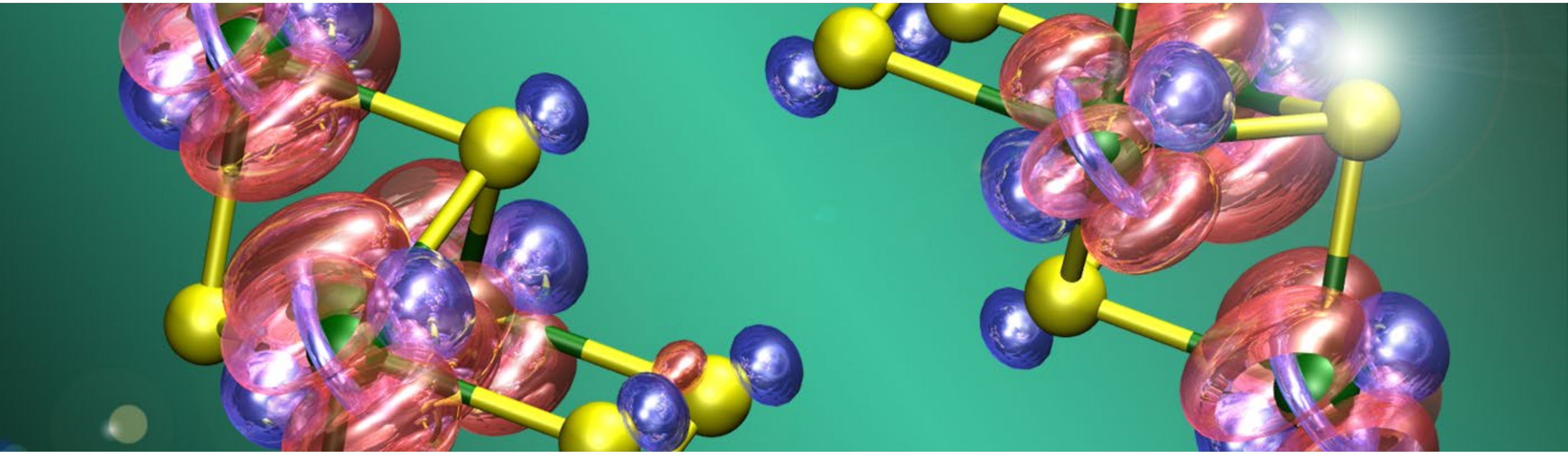
Rubrene/C_{60}



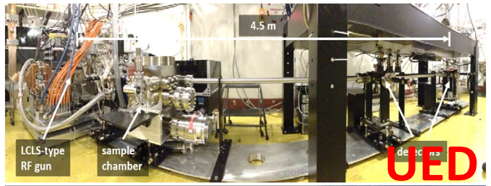
quasi-electron; quasi-hole

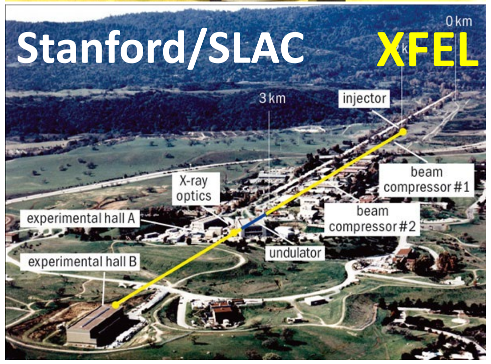
- **Excited states:** Linear-response time-dependent density functional theory [Casida, '95]
- **Interstate transitions:** Surface hopping [Tully, '90; Jaeger, Fisher & Prezhdo, '12]

Simulation-Experiment Synergy



- In ultrafast ‘electron & X-ray cameras,’ laser light hitting a material is almost completely converted into nuclear motions — key to switching material properties on & off at will for future electronics applications.
- High-end nonadiabatic quantum molecular dynamics simulations reproduce the ultrafast energy conversion at exactly the same space & time scales, and explain it as a consequence of photo-induced phonon softening.

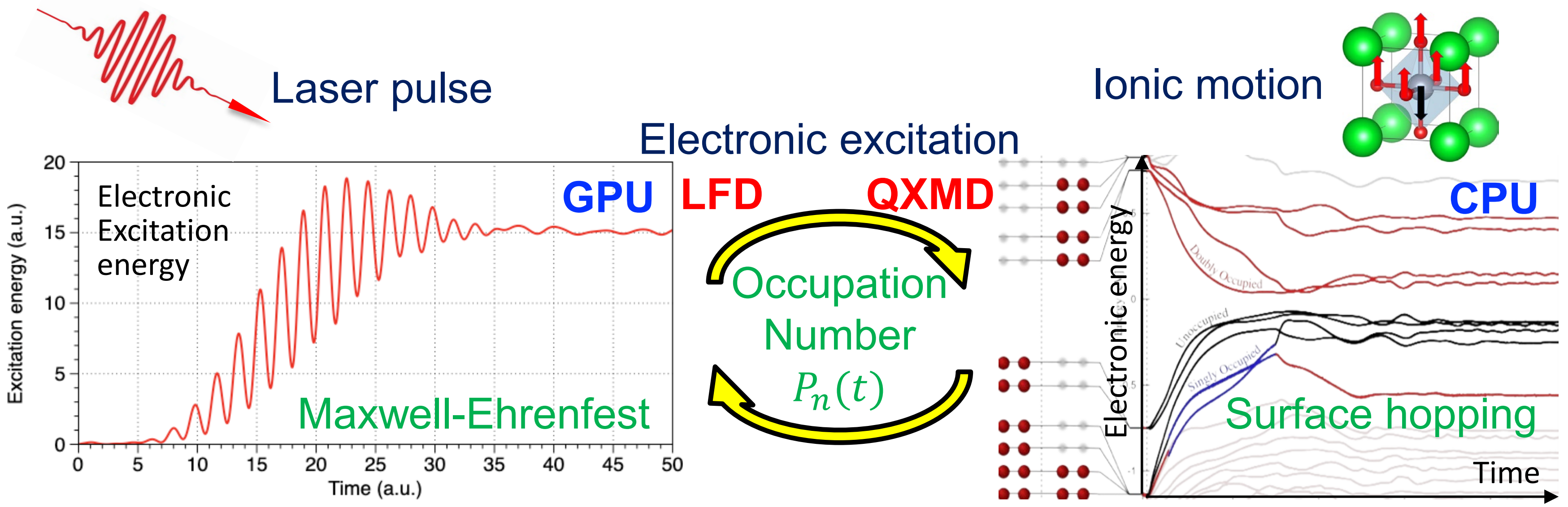




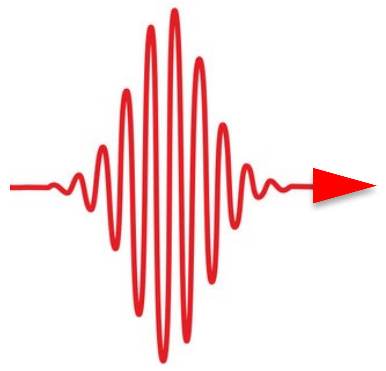
Ultrafast electron diffraction: M.F. Lin *et al.*, *Nature Commun.* **8**, 1745 (’17)
X-ray free-electron laser: I. Tung *et al.*, *Nature Photon.* **13**, 425 (’19)

Nonadiabatic Quantum MD: DC-MESH

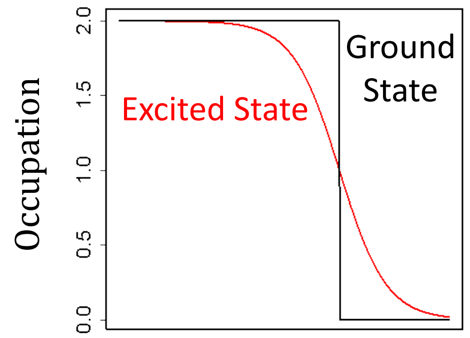
- **DC-MESH (divide-&-conquer Maxwell + Ehrenfest + surface-hopping):** O(N) algorithm to simulate photo-induced quantum materials dynamics
- **LFD (local field dynamics):** Maxwell equations for light & real-time time-dependent density functional theory equations for electrons to describe light-matter interaction
- **QXMD (quantum molecular dynamics with excitation):** Nonadiabatic coupling of excited electrons & ionic motions based on surface-hopping approach
- **“Shadow” LFD (GPU)-QXMD (CPU) handshaking** via electronic occupation numbers with minimal CPU-GPU data transfer
- **GSLD:** Globally sparse (interdomain Hartree coupling via multigrid) & locally dense (intradomain nonlocal exchange-correlation computation via BLAS) solver



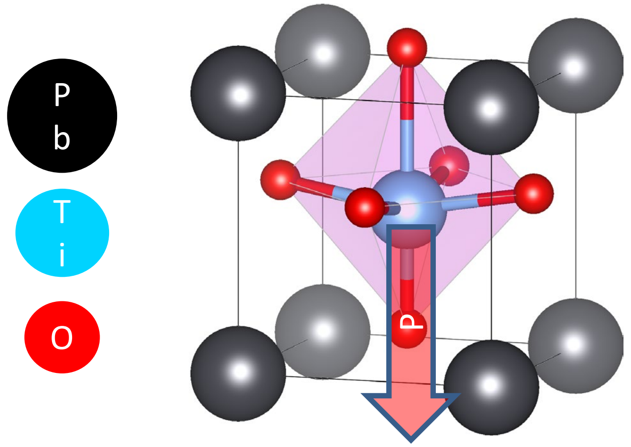
Multiscaling from DC-MESH to XS-NNQMD



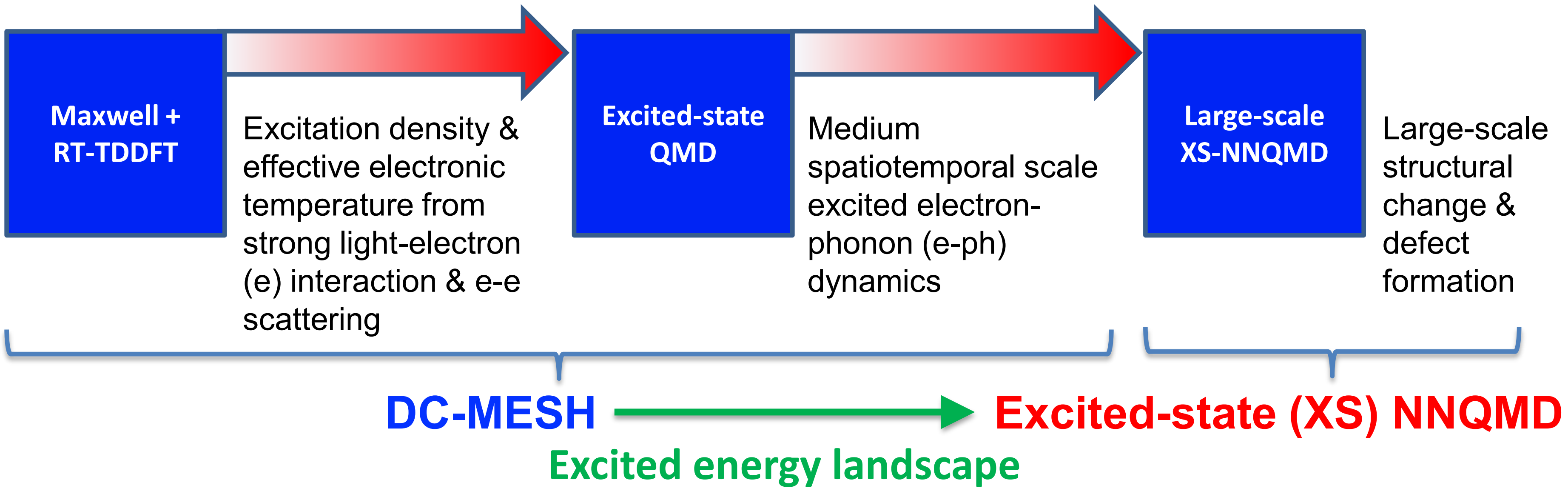
Electronic excitation by ultrafast laser pulse



Change in electronic occupation due to electron-electron interaction



Change in polarization dynamics due to electron-ion interaction



Excitation density & effective electronic temperature from strong light-electron (e) interaction & e-e scattering

Medium spatiotemporal scale excited electron-phonon (e-ph) dynamics

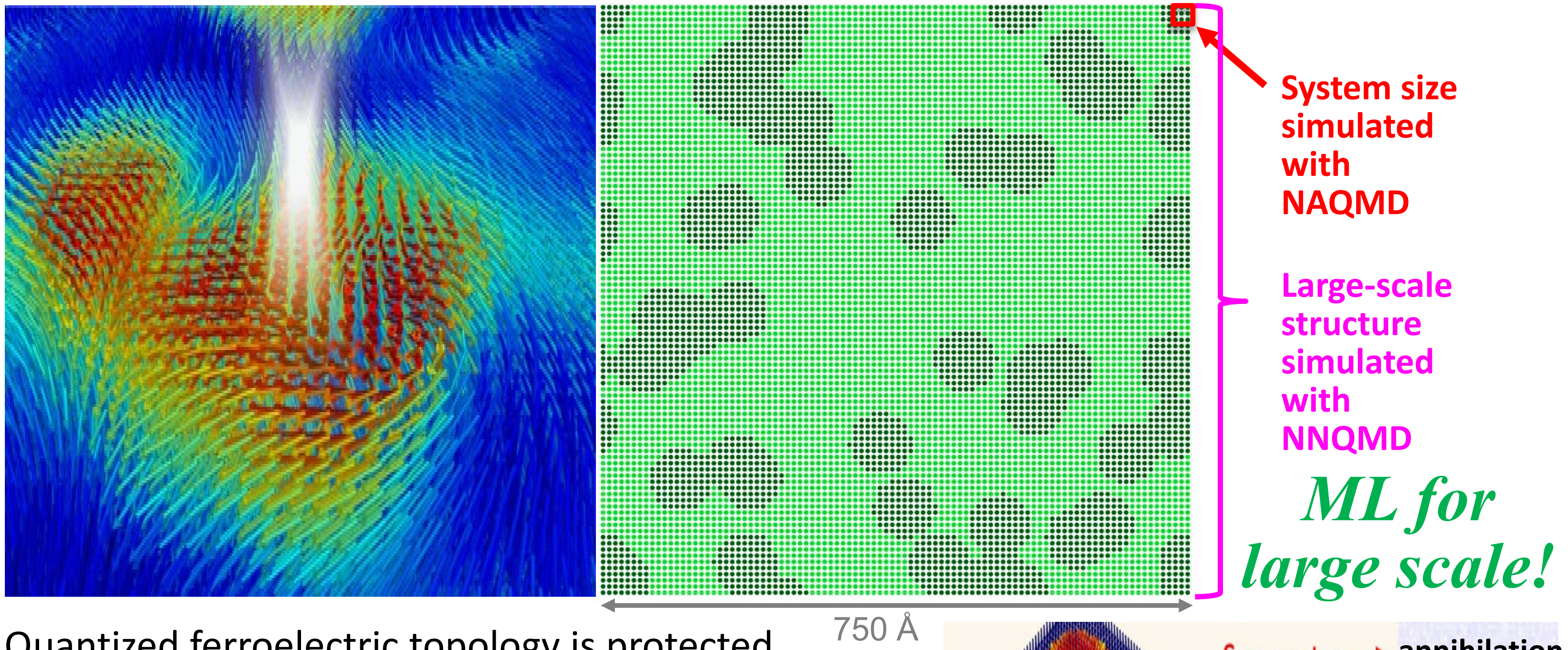
Large-scale structural change & defect formation

DC-MESH → **Excited-state (XS) NNQMD**

Excited energy landscape

Linker *et al.*, *Science Advances* **8**, eabk2625 ('22)

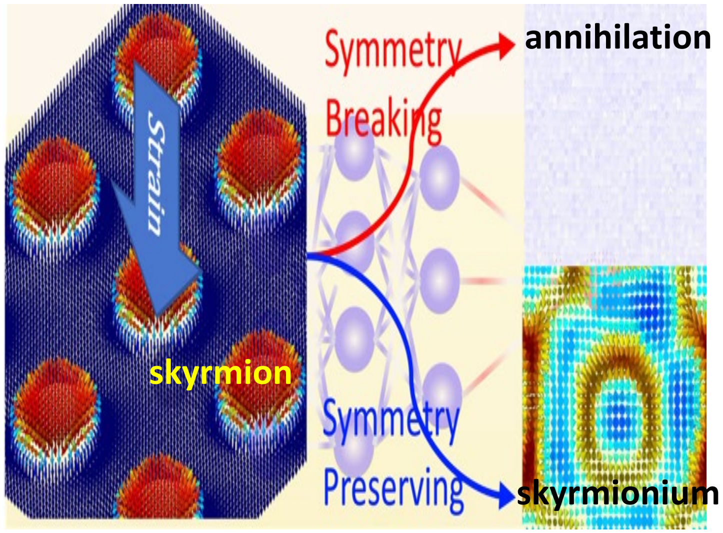
Application: Ferroelectric Opto-Topotronics


System size simulated with NAQMD
Large-scale structure simulated with NNQMD
ML for large scale!
750 Å

- Quantized ferroelectric topology is protected against thermal noise → future ultralow-power opto-electronics applications
- Billion-atom NNQMD revealed photo-induced topological phase-transition dynamics (*cf*. Kibble-Zurek mechanism in cosmology)
- Symmetry-controlled skyrmion-to-skyrmionium* switching

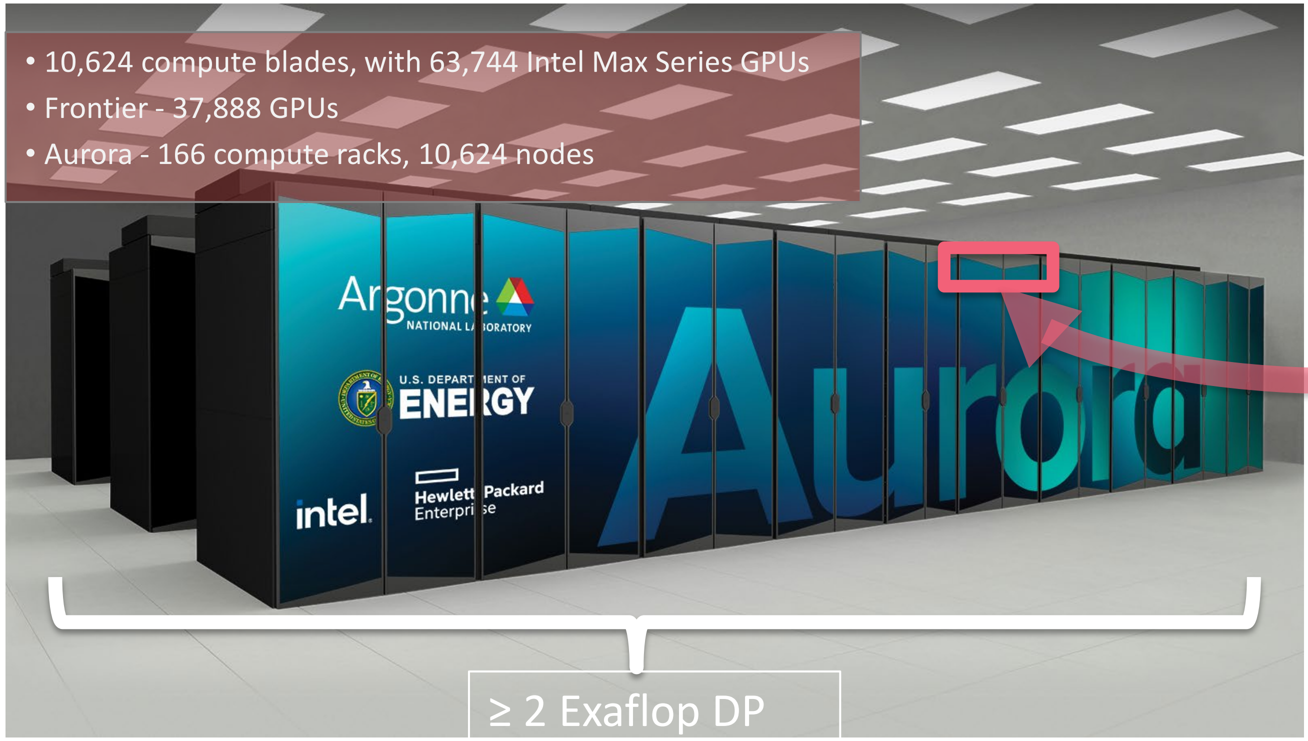
*Composite of skyrmions with opposite topological charges

Linker *et al., Science Adv.* **8**, eabk2625 ('22); *JPCL* **13**, 11335 ('22); *Nano Lett*., article ASAP ('23)



Aurora is coming, prepare!

- DCMESH to be used for Aurora ESP
- ALCF test machines to restructure and evaluate the porting process
- Exciting Application, Exciting Machines!


Argonne NATIONAL LABORATORY
U.S. DEPARTMENT OF ENERGY
intel
Hewlett Packard Enterprise
Aurora

- 10,624 compute blades, with 63,744 Intel Max Series GPUs
- Frontier - 37,888 GPUs
- Aurora - 166 compute racks, 10,624 nodes

≥ 130 TeraFlop DP Per Node

≥ 2 Exaflop DP

https://www.top500.org/lists/top500/2023/06/

Overview of Code Structure

- DC-MESH (divide-&-conquer Maxwell + Ehrenfest + surface-hopping) has extensively used MPI
  - QXMD(Fortran) + LFD(C++)
- Local Field Dynamics : LFD
  - Completely new extensions written from scratch in C++
  - Enabling GPU offload for kernels using OpenMP
  - Wave function initialization, computing electron density, local potential, nonlocal pseudopotential propagator, energy correction, transition probability are ported.

OpenMP

Overview of Code Structure (contd.)

- LFD initially developed as a C++ mini-app (by Pankaj Rajak while a PostDoc at ANL) now imported as a library.
- **Isolation:** lfd_main.cpp serving fake simulations for development needs
- **Interoperability:** Initialize and update occupation number in C++

```
// flexible precision
template <typename Real>
class LFD;

// expose C APIs
extern "C“ {
void init_lfd();
void occ_by_lfd();
}

// iso_c_binding in Fortran
```

Header for Fortran facing functions for Maxwell + Ehrenfest + surface-hopping)

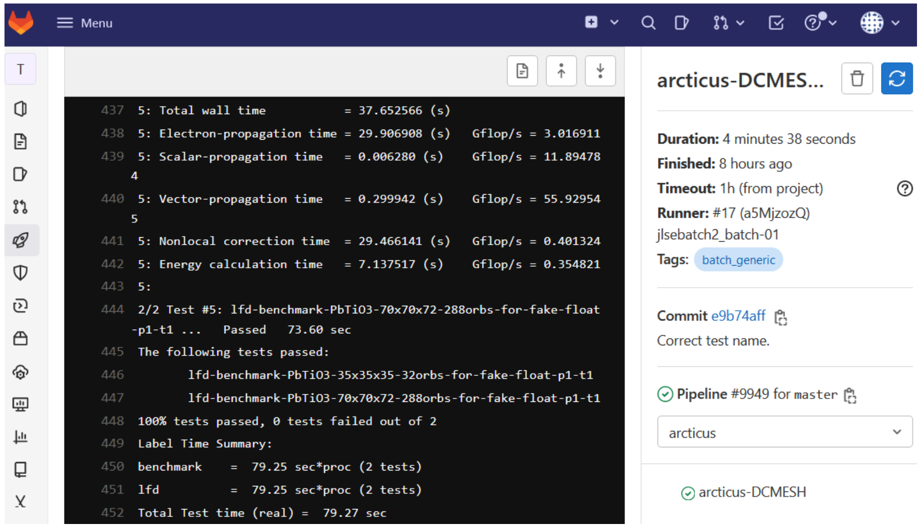
Linking Offload Code and Mixing Compilers

- Some C++ compilers complain about missing main() when Fortran has the main program, but it is not compiled with gfortran
- Linking GPU code device code requires extra steps inside the compiler driver
  - When not mixing compiler vendors, use Fortran compiler as the linker with offload flag added
- Flang doesn’t work and need to use gfortran. “-fPIE” on some Linux distros
  - Mixing is not supported when not using gfortran
- Static library linking rules does not extract device image for input object files
  - Adding a dummy library with target offload to an existing object file

https://github.com/llvm/llvm-project/issues/63158

CMake/CTest

- MakeFile
  - Original QXMD, Mini-App
- Switch to CMake
  - Modular libraries with CMake targets
  - Can handle Fortran and C++ runtime libraries
- CTest
  - Fully automated validation and benchmarking
  - JLSE gitlab



```
5: Total wall time          = 37.652566 (s)
5: Electron-propagation time = 29.906908 (s)   Gflop/s = 3.016911
5: Scalar-propagation time   = 0.006280 (s)    Gflop/s = 11.89478
4
5: Vector-propagation time   = 0.299942 (s)    Gflop/s = 55.92954
5
5: Nonlocal correction time  = 29.466141 (s)   Gflop/s = 0.401324
5: Energy calculation time   = 7.137517 (s)    Gflop/s = 0.354821
5:
2/2 Test #5: lfd-benchmark-PbTiO3-70x70x72-288orbs-for-fake-float
-p1-t1 ...   Passed   73.60 sec
The following tests passed:
        lfd-benchmark-PbTiO3-35x35x35-32orbs-for-fake-float-p1-t1
        lfd-benchmark-PbTiO3-70x70x72-288orbs-for-fake-float-p1-t1
100% tests passed, 0 tests failed out of 2
Label Time Summary:
benchmark    =  79.25 sec*proc (2 tests)
lfd          =  79.25 sec*proc (2 tests)
Total Test time (real) =  79.27 sec
```

arcticus-DCMES...

Duration: 4 minutes 38 seconds
Finished: 8 hours ago
Timeout: 1h (from project)
Runner: #17 (a5MjzozQ) jlsebatch2_batch-01
Tags: batch_generic

Commit e9b74aff
Correct test name.

Pipeline #9949 for `master`

arcticus

arcticus-DCMESH

Correctness and Reproducibility

- ❑ Make CPU code solid
  - Scrutinize the code with address sanitizer
  - Address all warnings when compiling with -Wall
- ❑ The initial offload port was buggy
  - Missing transfers when using target enter/exit data optimization
  - target task missing dependencies
    - Segfault at random places in the backtrace
    - Issue: Synchronization with nowait
    - Disable them
- ✓ Testing while enabling offload to host
- ✓ Validate the code with NVIDIA GPU gfortran + clang++

Time Propagation of Electronic Wave Function is Well-Suited for GPUs

1. Given a wave function ψ for mesh S, quantum time step Δt_{QD}, compute diagonal α, and lower-upper diagonal coefficients β_l, β_u
2. Pick a point $i \in I$ from the $S = \{I, J, K\}$
3. Apply stencil to the grid points $(j, k) \in (J, K)$ to evolve ψ_i over Δt_{QD}
4. Update new wave function at i

$$\psi_i \leftarrow \beta_l \psi_{i-1} + \alpha \psi_i + \beta_u \psi_{i+1}$$

5. Repeat **2** $\forall \{I\}$ with next grid point i_{new}

Electron field solver: kin_prop( )

```
void kin_prop (int d, int p) {
float wrk[Nx+2][Ny+2][Nz+2][2], w[2];
for (int n=0; n < Norb; n++) {
  for (int i=1; I <= Nr[0]; i++)
    for (int j=1; j <= Nr[1]; j++)
      for (int k=1; k <= Nr[2]; k++) {
        w[0]  = al[d][p][0]*psi[n][i][j][k][0] - al[d][p][1]*psi[n][i][j][k][1] ;
        w[1]  = al[d][p][0]*psi[n][i][j][k][1] + al[d][p][1]*psi[n][i][j][k][0] ;

         ...
        for (int s=0; s<2; s++) wrk[i][j][k][s] = w[s] ;
      }
    # update psi[n][i][j][k][s] ← wrk[i][j][k][s]
} }
```

- Inefficient memory usage and loop structure
- By loop re-ordering we can eliminate wrk
- al can be cached since it is independent of n, j, k

Electron field solver: kin_prop( )

Update #1

```
void kin_prop (int d, int p) {
float w[2];
for (int j=1; j < Nr[1]; j++)
  for (int k=1; l <= Nr[2]; k++)
    for (int i=1; j <= Nr[0]; i++)
        for (int n=0; n < Norb; n++) {
            w[0]  = al_0*psi[i][j][k][n][0] – al_1*psi[i][j][k][n][1] ;
            w[1]  = al_1*psi[i][j][k][n][1] + al_0*psi[i][j][k][n][0] ;
             ...
            # update psi[n][i][j][k][s] ← w[s]
        }
}
```

✗

- Inefficient memory usage and loop structure
- By loop re-ordering we can eliminate wrk
- al can be cached since it is independent of n, j, k

✓ Better memory usage and data locality by changing data layout
psi[n,i,j,k,s]→psi[i,j,k,n,s]

Electron field solver: kin_prop( )

Update #1 Caveat!

```
void kin_prop (int d, int p) {
for (int j=1; j < Nr[1]; j++)
  for (int k=1; l <= Nr[2]; k++)
    for (int i=1; j <= Nr[0]; i++)
         for (int n=0; n < Norb; n++) {
              w[0]  = al_0*psi[i][j][k][n][0] – al_1*psi[i][j][k][n][1] ;
              w[1]  = al_1*psi[i][j][k][n][1] + al_0*psi[i][j][k][n][0] ;
               w[0]  += bl_0[i]*psi[i-1][j][k][n][0] – bl_1[i]*psi[i-1][j][k][n][1] ;
               w[1]  += bl_0[i]*psi[i-1][j][k][n][1] – bl_1[i]*psi[i-1][j][k][n][0] ;
               …
              # update psi[n][i][j][k][s] ← w[s]
         }
}
```

Upon loop re-ordering

- No longer using old value of psi.
- psi[i-1][j][k] underwent an unnecessary update for all orbitals

Electron field solver: kin_prop( )

Update #2

```
void kin_prop (int d, int p) {
for (int j=1; j < Nr[1]; j++)
  for (int k=1; l <= Nr[2]; k++) {
    for (int n=0; n <= Norb; n++) {
         psi_old0[n] = psi[0][j][k][n][0];
         psi_old1[n] = psi[0][j][k][n][1];  }
     for (int i=1; j <= Nr[0]; i++)
     for (int n=0; n < Norb; n++) {
              w[0]  = al_0*psi[i][j][k][n][0] – al_1*psi[i][j][k][n][1] ;
              w[1]  = al_1*psi[i][j][k][n][1] + al_0*psi[i][j][k][n][0] ;
               w[0]  += bl_0[i]*psi_old0[n] – bl_1[i]*psi_old1[n] ;
               w[1]  += bl_0[i]*psi_old1[n] – bl_1[i]*psi_old0[n] ;
               …
              # update psi_old0 ← psi[n][i][j][k][0] and psi_old1 ← psi[n][i][j][k][s]
             # update psi[n][i][j][k][s] ← w[s]
         }
} }
```

Copy old psi for correctness

Electron field solver: kin_prop( )

Update #2 Caveat!

```
void kin_prop (int d, int p) {
for (int j=1; j < Nr[1]; j++)
  for (int k=1; l <= Nr[2]; k++) {
    for (int n=0; n <= Norb; n++) {
        psi_old0[n] = psi[0][j][k][n][0];
        psi_old1[n] = psi[0][j][k][n][1];  }
    for (int i=1; j <= Nr[0]; i++)
    for (int n=0; n < Norb; n++) {
            w[0]  = al_0*psi[i][j][k][n][0] – al_1*psi[i][j][k][n][1] ;
            w[1]  = al_1*psi[i][j][k][n][1] + al_0*psi[i][j][k][n][0] ;
             w[0]  += bl_0[i]*psi_old0[n] – bl_1[i]*psi_old1[n] ;
             w[1]  += bl_0[i]*psi_old1[n] – bl_1[i]*psi_old0[n] ;
            …
            # update psi_old0 ← psi[n][i][j][k][0] and psi_old1 ← psi[n][i][j][k][s]
           # update psi[n][i][j][k][s] ← w[s]
        }
} }
```

Multi-Dimensional Arrays do not work on GPU

- Complex operations
- Convert psi, psi_old into *1D* complex variable

Electron field solver: kin_prop( )

Update #3

```
void kin_prop (int d, int p) {
for (int j=1; j < Nr[1]; j++)
  for (int k=1; l <= Nr[2]; k++) {
    for (int n=0 ; n <  Norb; n++)
        psi_old[i] = psi[yz_stride+n];
        for (int i=1; j <= Nr[0]; i++)
        for (int n=0; n < Norb; n++) {
            w  = al*psi[stride+n] + bl[i]*psi_old[n] + …;
          # update psi_old0[n] ← psi[stride+n]
          # update psi[stride+n]← w
        }
  }
}
```

Multi-Dimensional Arrays do not work on GPU

- Complex operations
- Convert psi, psi_old into *1D* complex variable

Before

- std:: float psi[Nx+2][Ny+2][Nz+2][Norb]
- std:: float psi_old[Ny+2][Nz+2][Norb]

After

- std::complex<float> psi
- std::complex<float> psi_old

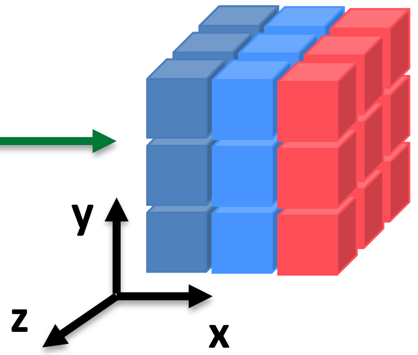
Electron field solver: kin_prop( )

Offload Strategy

```
void kin_prop (int d, int p) {
#pragma omp teams distribute collapse(2)
for (int j=1; j < Nr[1]; j++)
  for (int k=1; l <= Nr[2]; k++) {
   #pragma omp parallel for simd nowait
   for (int n=0 ; n <  Norb; n++)
        psi_old[i] = psi[yz_stride+n];
   for (int i=1; j <= Nr[0]; i++)
      #pragma omp parallel for simd nowait
       for (int n=0; n < Norb; n++) {
              w  = al*psi[stride+n] + bl[i]*psi_old[n] + …;
             # update psi_old0[n] ← psi[stride+n]
             # update psi[stride+n]← w
          }
  } }
```

Hierarchical parallelism

- Coarse grain parallelism via *omp teams distribute* on outer loops
- Fine grain parallelism on inner Norb loop via *omp parallel for*
- Typical size of Nr is 256 and Norb 100

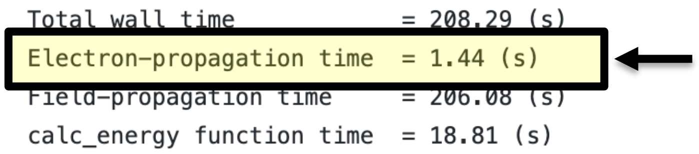


Electron field solver: kin_prop( )

Offload Timing

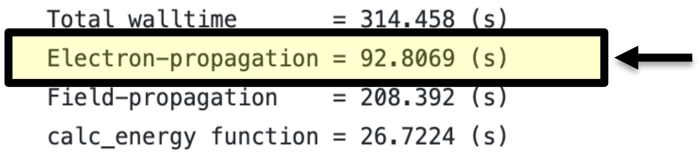
```
void kin_prop (int d, int p) {
#pragma omp teams distribute  collapse(2)
for (int j=1; j < Nr[1]; j++)
  for (int k=1; l <= Nr[2]; k++) {
   #pragma omp parallel for simd nowait
    for (int n=0 ; n <  Norb; n++)
        psi_old[i] = psi[yz_stride+n];
        for (int i=1; j <= Nr[0]; i++)
       #pragma omp parallel for simd nowait
        for (int n=0; n < Norb; n++) {
             w  = al*psi[stride+n] + bl[i]*psi_old[n] + …;
           # update psi_old0[n] ← psi[stride+n]
           # update psi[stride+n]← w
        }
  }}
```

Updated timing



```
Total wall time              = 208.29 (s)
Electron-propagation time    = 1.44 (s)
Field-propagation time       = 206.08 (s)
calc_energy function time    = 18.81 (s)
```

Original



```
Total walltime         = 314.458 (s)
Electron-propagation = 92.8069 (s)
Field-propagation      = 208.392 (s)
calc_energy function = 26.7224 (s)
```

Using xlr complier

Time Propagation of Electromagnetic Wave Function: field_prop()

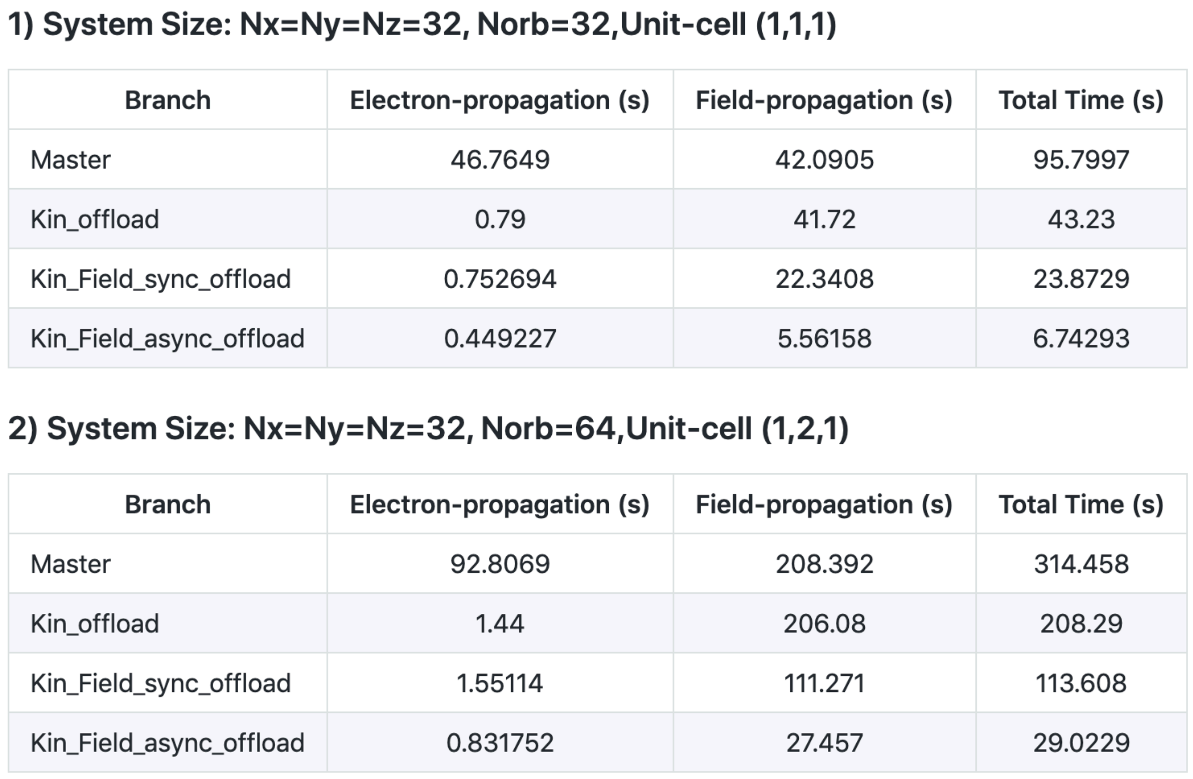
1. Given a Hartree field υ for mesh S and field dynamics time step Δt_{FD}, compute electron density ρ
2. Pick a point $i \in I$ from the $S = \{I, J, K\}$
3. Apply stencil to the grid points $(j, k) \ \in \ (J, K)$ to evolve υ_i over Δt_{FD}
4. Update new Hartree field at i

$$\upsilon_i \leftarrow \ e^{i\widehat{L}[\rho]t_{QD}}\upsilon_i$$

5. Repeat **2** $\forall \ \{I\}$ with next grid point i_{new}

Follows same loop-reordering, array flattening and parallelism strategy

Timing Summary



1) System Size: Nx=Ny=Nz=32, Norb=32,Unit-cell (1,1,1)

Branch	Electron-propagation (s)	Field-propagation (s)	Total Time (s)
Master	46.7649	42.0905	95.7997
Kin_offload	0.79	41.72	43.23
Kin_Field_sync_offload	0.752694	22.3408	23.8729
Kin_Field_async_offload	0.449227	5.56158	6.74293

2) System Size: Nx=Ny=Nz=32, Norb=64,Unit-cell (1,2,1)

Branch	Electron-propagation (s)	Field-propagation (s)	Total Time (s)
Master	92.8069	208.392	314.458
Kin_offload	1.44	206.08	208.29
Kin_Field_sync_offload	1.55114	111.271	113.608
Kin_Field_async_offload	0.831752	27.457	29.0229

Workaround/Optimization for Intel icpx

Issues with “this”

```
class LFD {
  size_t N;
  void do_something(){
    #pragma omp target
    { // read N }
    // N becomes garbage
    // due to map(tofrom:this[:1])
  }
}
```

```
class LFD {
  size_t N;
  void do_something(){
    size_t N_local = N;
    #pragma omp target
    { // read N_local }
    // N_local is first private
  }
}
```

Use BLAS GEMM

- Initial CPU run takes 20 minutes and a lot of time in GEMM like codes
- The dot product over the mesh is not contiguous. 0 and Nx+1, Ny+1, Nz+1 are ghost elements.
- We extract the core elements to psi_core array and then call zgemm
- Now <1 minute on 1 CPU core.
- Use vendor optimized libraries

```
for (int m = 0; m < Norb; m++) {
  for (int n = 0; n < Norb; n++) {
    sum = {0.0, 0.0};
    for (int k = 1; k <= Nz; k++)
      for (int j = 1; j <= Ny; j++)
        for (int i = 1; i <= Nx; i++) {
          const size_t indxm = m + …;
          const size_t indxn = n + …;
          sum += std::conj(psi0_ptr[indxm]) * psi_ptr[indxn];
        }
    sum *= Dvol; } // End inner for KS orbitals n
 }  // End outer for KS orbitals m
```

Non-local Potential Propagation

- For $Norb$ orbitals solve: $0 < \text{nlumo} < nhomo \leq Norb$

$$|\psi_n\rangle -= \frac{i\Delta_{sci}\Delta_{QD}}{2} \underbrace{\sum_{nlumo}^{Norb} |m\rangle\langle m|\psi_{n,}\rangle}_{\text{GEMM operation}} \quad n \in [0, nhomo]$$

- Where:

$$\langle m|\psi_n\rangle = \Delta_x\Delta_y\Delta_z \overbrace{\sum_{ijk} \psi^T_{[0:nlumo],t=0}\psi_{[nlumo:]}}$$

Re-write as two BLAS Level 3 calls

Energy Correction

- To calculate total potential energy E_{pot}:

$0 < \mathrm{nlumo} < nhomo \leq Norb$

$$E_{pot} \mathrel{+}= \Delta_{sci} \underbrace{\sum_{n=o}^{nhomo} f_n \underbrace{\sum_{m=nlumo}^{Norb} |\langle m|\psi_n\rangle|^2}_{\textbf{GEMM operation 1}}}_{\textbf{GEMM operation 2}}$$

Re-write as two BLAS Level 3 calls

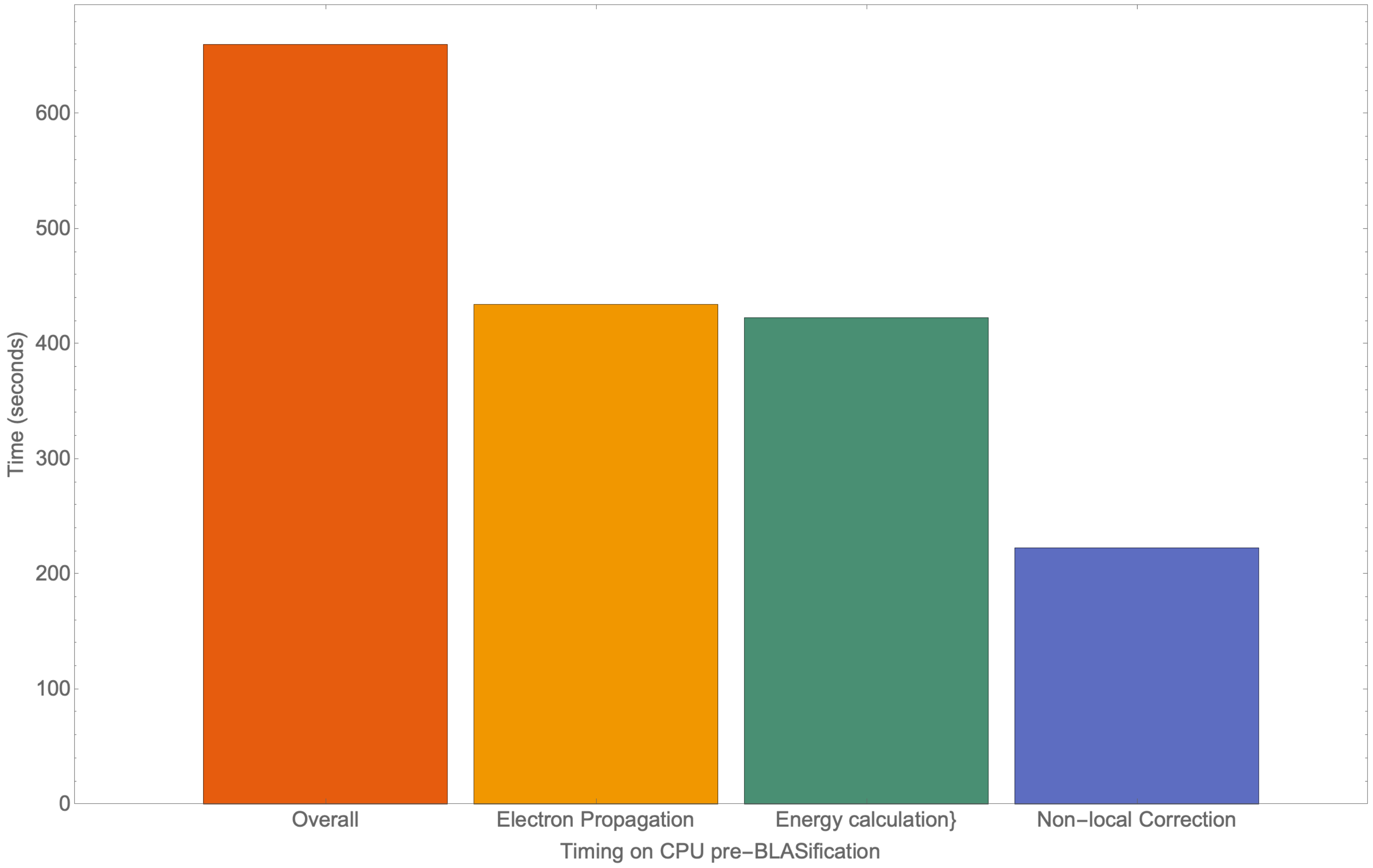
Performance bottleneck

Profiling Insights: “Look around the GEMM calls”

- > 80% hot spots have recurring pattern
- Utilize vendor BLAS call
- Take all the codes from QMCPACK platforms abstraction CUDA/HIP/SYCL
  - Device management
  - GPU runtime abstraction
  - GPU BLAS abstraction

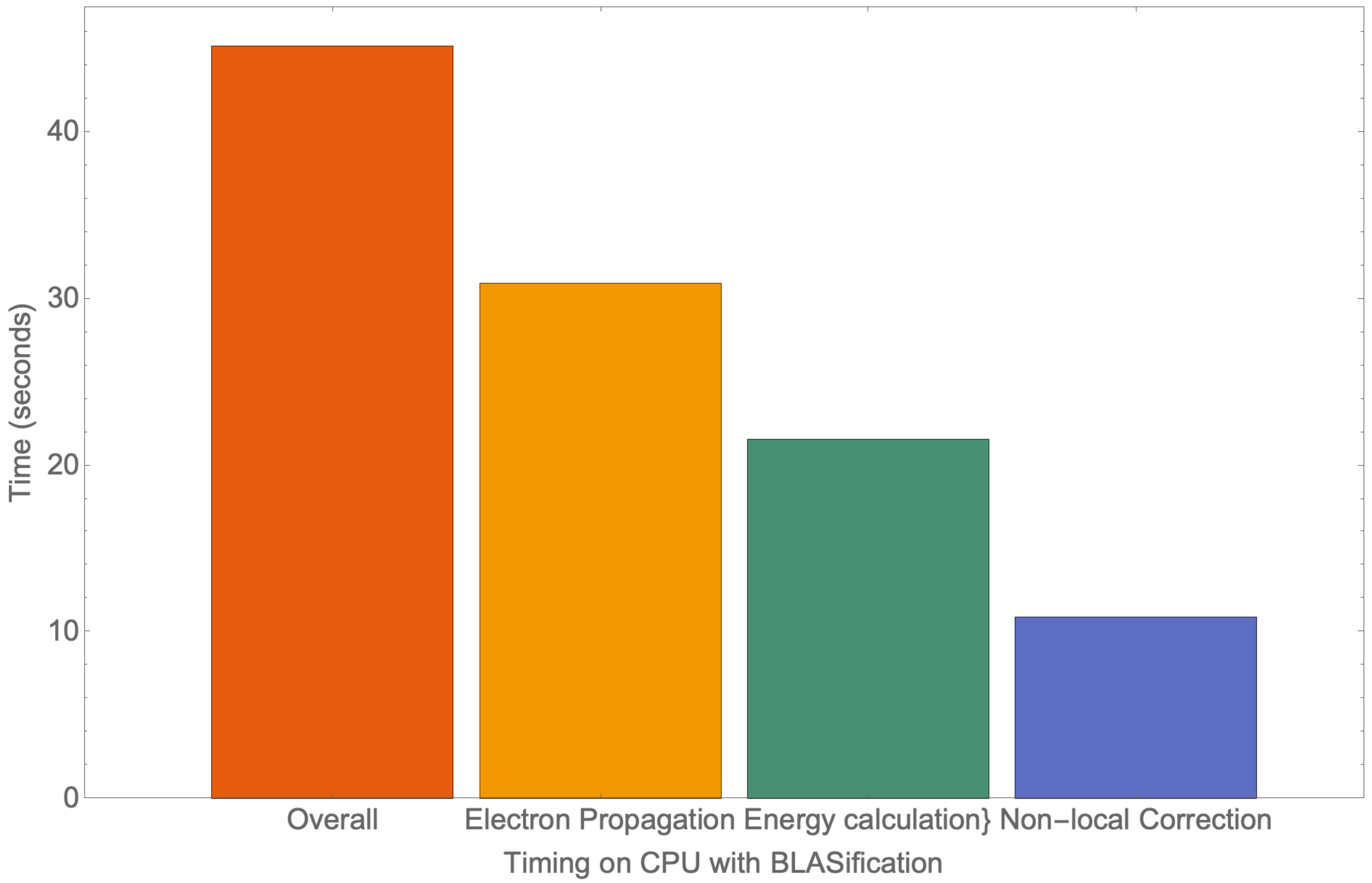
Time in Benchmark with Loop-Reordering

clang version 15.0.0 Nx=70 Ny=70 Nz= 72 Norb = 288orbs



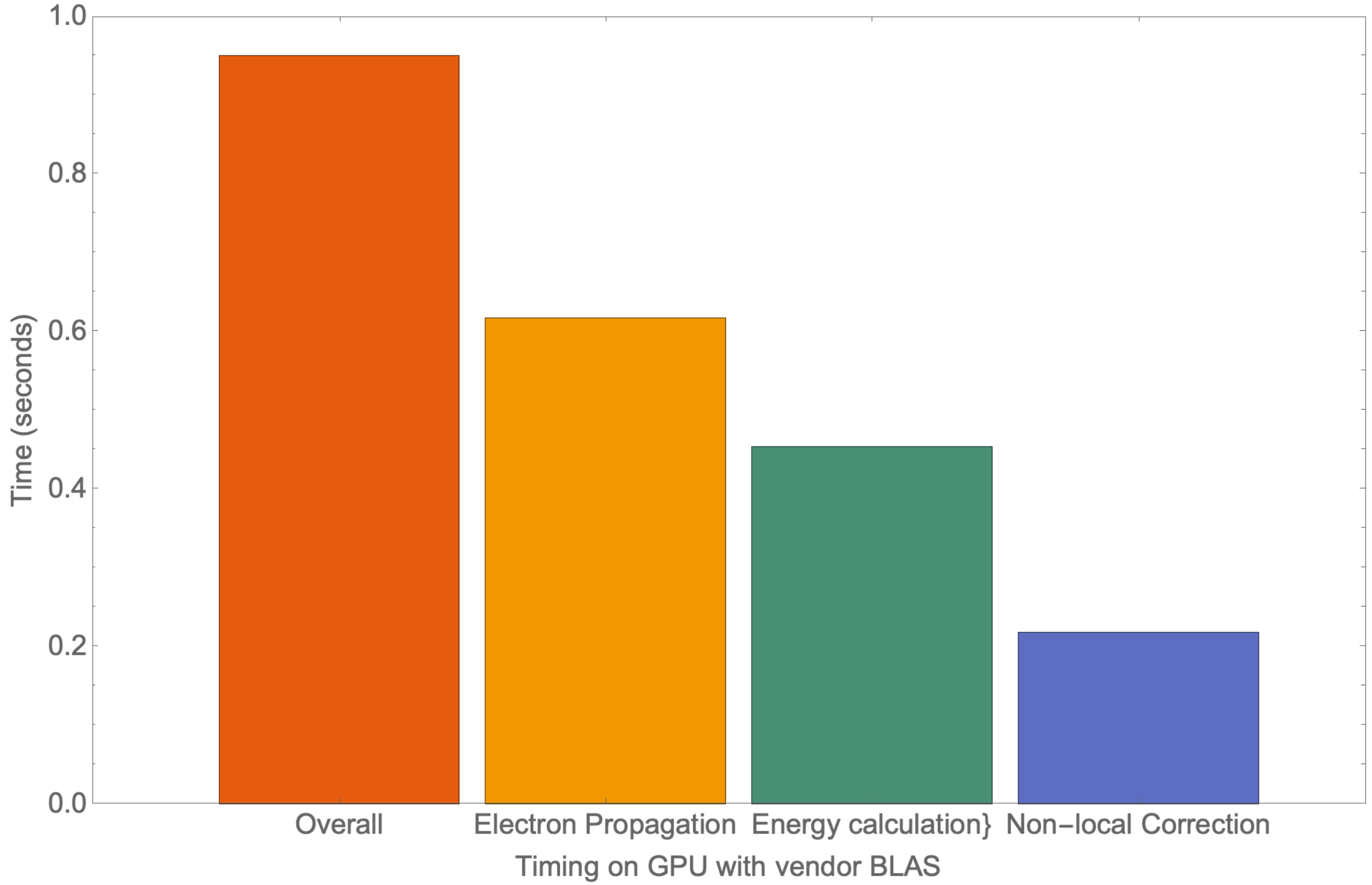
Time in Benchmark with BLAS

clang version 15.0.0 Nx=70 Ny=70 Nz= 72 Norb = 288orbs

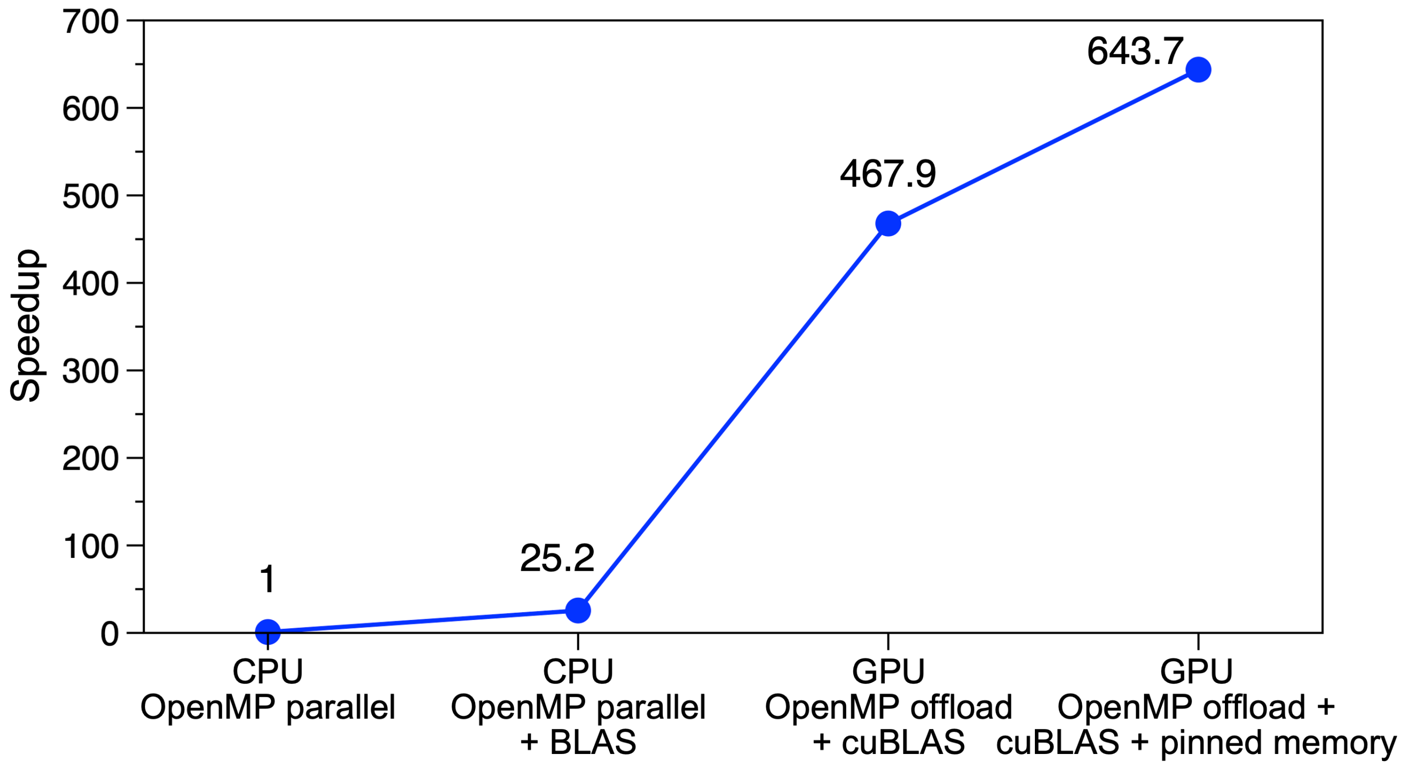


Time in Benchmark with Offload

clang version 15.0.0 Nx=70 Ny=70 Nz= 72 Norb = 288orbs



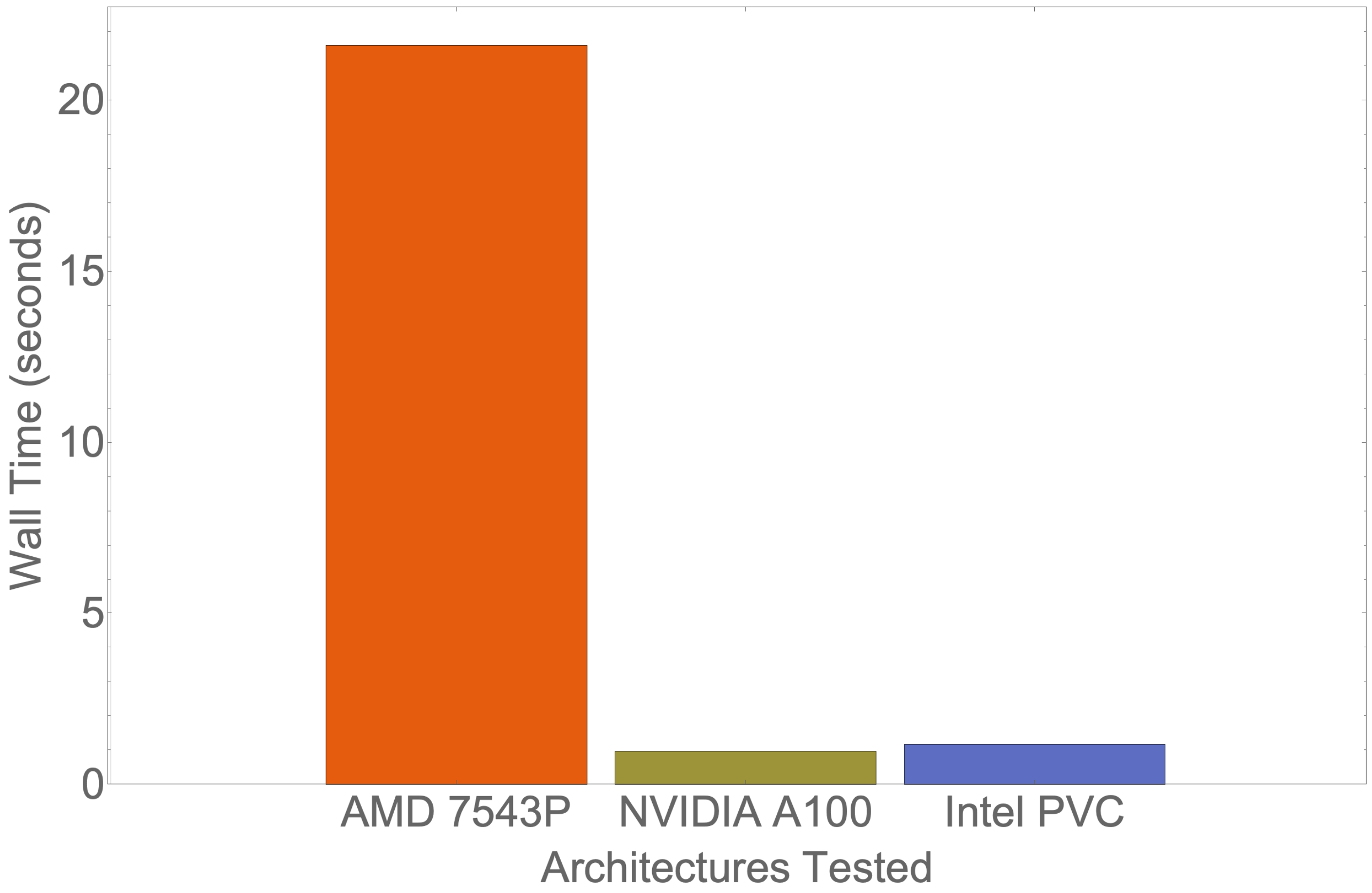
Speed-up



Plotted over the baseline QXMD DC-MESH code on a single Polaris node resulting from a series of code optimizations

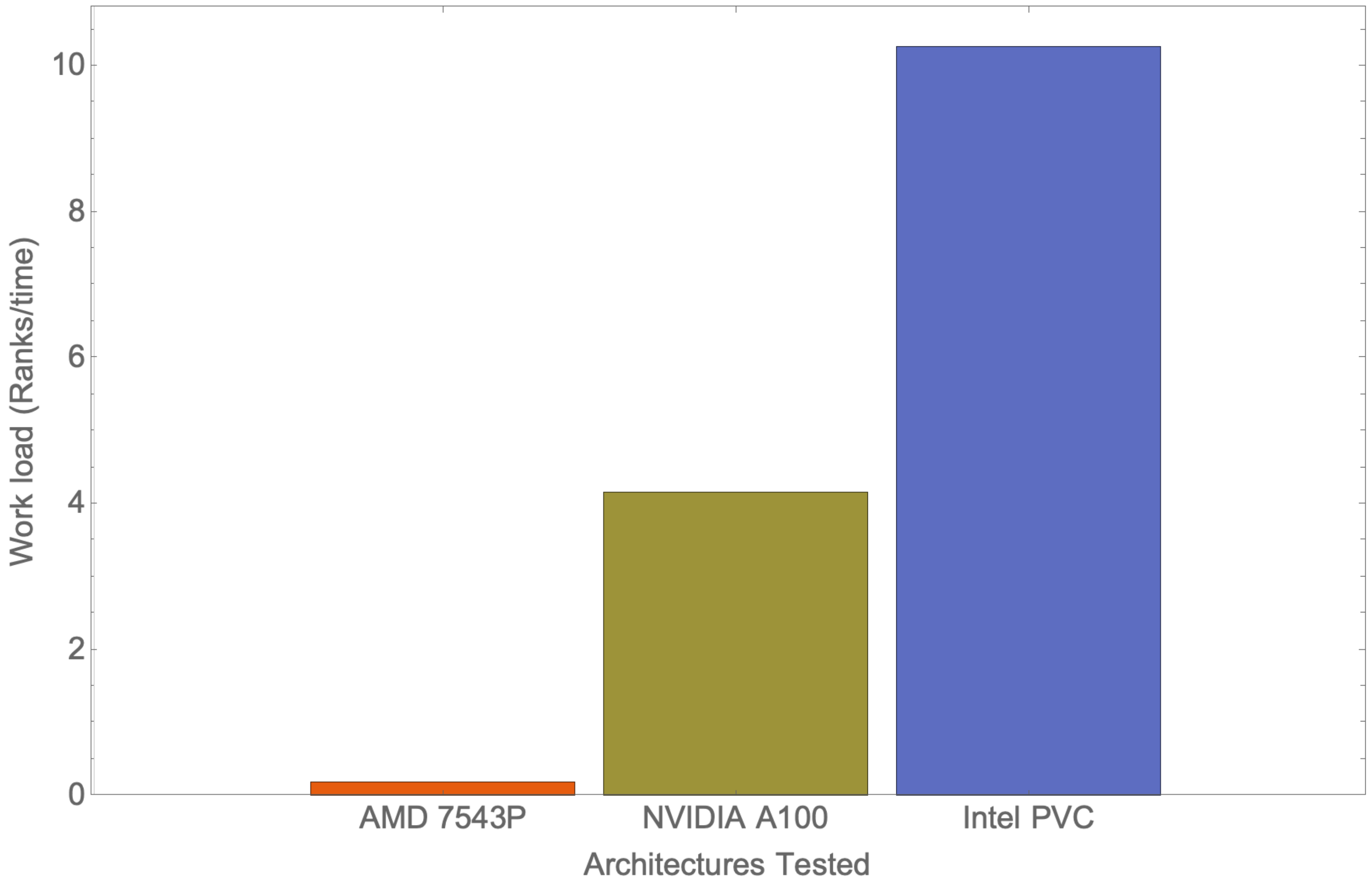
Completion time across Architectures

Benchmarks carried out on PbTiO3-70x70x72-288orbs, float



Workload across Architectures

Benchmarks carried out on PbTiO3-70x70x72-288orbs, float



Summary

- A data layout for propagators that facilitates SIMD parallelism to a high extent
- Utilized SYCL kernels for Intel GPU offload
- Carried out LFD execution on Sunspot blades (PVC)