Performance Optimization of XGC1 on Cori KNL

Tuomas Koskela
NESAP postdoc
NERSC / LBNL
tkoskela@lbl.gov

February 27, 2018
Thank you to all collaborators!

- **LBNL**
  - Brian Friesen, Ankit Bhagatwala, Mark Adams, Mathieu Lobet, Tareq Malas, Andrey Ovsyannikov, Kevin Gott, Rahul Gayatri, Zahra Ronaghi

- **PPPL**
  - CS Chang, Robert Hager, Seung-Hoe Ku, Stephane Ethier

- **ORNL**
  - Ed D’Azevedo, Stephen Abbott, Pat Worley

- **Intel**
  - Thanh Phung, Zakhar Matveev, John Pennycook, Martyn Corden, Karthik Raman

- **RPI**
  - Eisung Yoon, Mark Shephard
Outline

- Introduction to XGC1
- Particle Push Vectorization and Data Structure Reordering Optimizations
- Toypush mini-app
- Charge Deposition Threading Optimizations
- Conclusions
Cori at NERSC

- **2388 Haswell nodes**
  - 2x 16 core @ 2.3 GHz
  - 40 MB shared L3
  - 128 GB DDR

- **Cray Aries Interconnect**
  - dragonfly topology

- **9688 Xeon Phi (KNL) nodes**
  - 68 cores @ 1.4 GHz
  - 34 MB distributed L2
  - 96 GB DDR
  - 16 GB MCDRAM (on-package)
XGC1 is a Particle-In-Cell Simulation Code for Tokamak (Edge) Plasmas

PI: CS Chang (PPPL) | ECP: High-Fidelity Whole Device Modeling of Magnetically Confined Fusion Plasma
Basic Plasma PIC Code Flowchart

Computation Mapping

1. Collect Fields from Mesh to Particles
2. Solve Fields on Mesh
3. Deposit Charge From Particles to Mesh
4. Particle Push
XGC1 Unique Optimization Challenges

• **Complicated Tokamak Geometry**
  – Unstructured grid in 2D (poloidal) plane(s)
  – Nontrivial field-following (toroidal) mapping between planes
  – Full-f model, exascale simulations will have 10 000 particles per cell, 1 000 000 cells per domain, 100 toroidal domains.

• **Gyrokinetic Equation of Motion in Cylindrical Coordinates**
  – $+ 6D$ to $5D$ problem
  – $+ O(100)$ longer time steps
  – -- Higher (2nd) order derivative terms in force calculation
  – -- Averaging scheme in field gather

• **Electron Sub-Cycling**
In XGC1 Electron Time Scale is Separated From the Ion Push in a Sub-Cycling Loop

**Computation Mapping**

- **Solve Fields on Mesh**
- **Deposit Charge From Particles to Mesh**
- **Gather Fields from Mesh to Ions**
- **Gather Fields from Mesh to Electrons**
- **~50x**
- **Electron Push**
- **Electron Sub-Cycling**
- **Ion Push**
Electron Push Sub-Cycling
Motivation: XGC1 CPU time is dominated by electron push sub-cycle

Baseline XGC1 Timing distribution on 1024 Cori KNL nodes in quadrant flat mode.

Note: This run actually has a 32x smaller number of electrons & ions than production runs!
Motivation: Ideal Strong Scaling* of Electron Sub-Cycling On Cori

KNL, quadrant cache

Hybrid MPI/OpenMP

16 MPI ranks per node/16 OpenMP threads per rank.

25 Bn total electrons, decomposed to MPI ranks and OpenMP threads

*Requires good load balancing
(Simplified) Field following node mapping

- Grid consists of poloidal (2D) planes that have an identical set of nodes each.
- Nodes connect to neighboring planes by (approximately) following the magnetic field.
(Simplified) Particle Push Algorithm

1. **Search** for nearest 3 mesh nodes to the particle position & map to neighbor plane. Calculate neighbor node indices

2. **Interpolate** fields from neighbor mesh nodes to particle position

3. **Calculate force on** particle from fields

4. **Push** particle for time step $dt$
1. Search for nearest 3 mesh nodes to the particle position, map to neighbor plane and Calculate neighbor node indices

2. Interpolate fields from neighbor mesh nodes to particle position

3. Calculate force on particle from fields

4. Push particle for time step \( dt \)
Main Bottlenecks in Electron Push: Advisor/Vtune view before

<table>
<thead>
<tr>
<th>Program metrics</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Elapsed Time</td>
<td>16.88s</td>
<td>Paused Time</td>
</tr>
<tr>
<td>Vector Instruction Set</td>
<td>AVX512, AVX2, AVX</td>
<td>Number of CPU Threads</td>
</tr>
<tr>
<td>Total GFLOP Count</td>
<td>20.35</td>
<td>Total GFLOPS</td>
</tr>
<tr>
<td>Total Arithmetic Intensity</td>
<td>0.08005</td>
<td></td>
</tr>
</tbody>
</table>

Loop metrics

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Total</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total CPU time</td>
<td>136.46s</td>
<td>100.0%</td>
</tr>
<tr>
<td>Time in 1 vectorized loop</td>
<td>0.02s</td>
<td></td>
</tr>
<tr>
<td>Time in scalar code including time in 19 vectorized completely unrolled loops</td>
<td>136.44s</td>
<td>100.0%</td>
</tr>
<tr>
<td>Total GFLOP Count</td>
<td>20.35</td>
<td>100.0%</td>
</tr>
<tr>
<td>Total GFLOPS</td>
<td>1.21</td>
<td></td>
</tr>
</tbody>
</table>

Vectorization Gain/Efficiency

| Vectorized Loops Gain/Efficiency | 1.59x | 20%  |
| Program Approximate Gain | 1.00x |

Top time-consuming loops

<table>
<thead>
<tr>
<th>Loop</th>
<th>Self Time</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>[loop in search_tr2 at search_F90:736]</td>
<td>14.815s</td>
<td>14.815s</td>
</tr>
<tr>
<td>[loop in derivs_elec_vec at derivs_elec_vec_F90:118]</td>
<td>3.380s</td>
<td>3.380s</td>
</tr>
<tr>
<td>[loop in efield_gk_elec at pushe_F90:1089]</td>
<td>2.780s</td>
<td>2.780s</td>
</tr>
<tr>
<td>[loop in pushe_1step2_vec_omp$parallel_for@39 at pushe_1step2_vec_F90:49]</td>
<td>2.280s</td>
<td>110.906s</td>
</tr>
<tr>
<td>[loop in derivs_single_with_e_elec_vec at derivs_single_with_e_elec_vec_F90:49]</td>
<td>2.100s</td>
<td>40.902s</td>
</tr>
</tbody>
</table>
Main Bottlenecks in Electron Push

• **E and B Field Interpolation**
  – Inner loops in function calls over nearby grid nodes with short trip counts make auto-vectorization ineffective
  – Indirect grid access produces gather/scatter instructions

• **Search on Unstructured Mesh**
  – Multiple exit conditions

• **Force Calculation**
  – Strided memory access in complicated data types
  – Cache unfriendly
Main Optimizations in Electron Push

• **Enabling Vectorization**
  – Insert loops over blocks of particles inside short trip count loops to enable automatic vectorization
  – Sort particles to reduce random memory accesses
  – Tile particle loop to improve cache reuse

• **Data Structure Reordering**
  – Store field and particle data in SoAoS format to reduce number of gathers and improve vectorization efficiency

• **Algorithmic Improvements**
  – Sort particles by the mesh element index instead of local coordinates
  – Reduce number of unnecessary calls to the search routine
Re-Ordering Loops to Enable Vectorization

Baseline code

1. Loop Over Time Steps
2. Loop Over All Particles
3. Short loop over nearby nodes

Vectorized code

1. Sort Particles
2. Loop Over Blocks of Particles
3. Loop Over Time Steps
4. Short loop over nearby nodes
5. Loop over Particles in Block

- Sort particles to reduce random memory access
- Swap the order of time step and particle loops to improve cache reuse
- Insert vectorizeable loop over blocks of particles inside short trip count loop
- Near-ideal vectorization in compute-heavy loops
  \(\rightarrow\) Indirect memory access becomes the bottleneck
Reorder Particle and Field Data Structures

- Stores field data at particle location between field gather and particle push
- During push, each particle stores 12 doubles + 2 integers + a field structure with 27 doubles. Common access pattern is accessing 3 components of a vector field \((x, y, z)\)
- AoS → Strided when accessing one data type of multiple particles
- SoA → Strided when accessing multiple data types of a one particle

**AoS**

Number of fields: 27

<table>
<thead>
<tr>
<th>(x_1)</th>
<th>(y_1)</th>
<th>(z_1)</th>
<th>(B_{x1})</th>
<th>(B_{y1})</th>
<th>(B_{z1})</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x_2)</td>
<td>(y_2)</td>
<td>(z_2)</td>
<td>(B_{x2})</td>
<td>(B_{y2})</td>
<td>(B_{z2})</td>
<td>...</td>
</tr>
</tbody>
</table>
| ... | ... | ... | ... | ... | ... | ...

| \(x_N\) | \(y_N\) | \(z_N\) | \(B_{xN}\) | \(B_{yN}\) | \(B_{zN}\) | ... |

**SoA**

<table>
<thead>
<tr>
<th>(x_1)</th>
<th>(x_2)</th>
<th>...</th>
<th>(x_N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(y_1)</td>
<td>(y_2)</td>
<td>...</td>
<td>(y_N)</td>
</tr>
<tr>
<td>(z_1)</td>
<td>(z_2)</td>
<td>...</td>
<td>(z_N)</td>
</tr>
<tr>
<td>(B_{x1})</td>
<td>(B_{x2})</td>
<td>...</td>
<td>(B_{xN})</td>
</tr>
<tr>
<td>(B_{y1})</td>
<td>(B_{y2})</td>
<td>...</td>
<td>(B_{yN})</td>
</tr>
<tr>
<td>(B_{z1})</td>
<td>(B_{z2})</td>
<td>...</td>
<td>(B_{zN})</td>
</tr>
</tbody>
</table>
| ... | ... | ... | ...

Number of particles per block: 32
Reorder Particle and Field Data Structures

- Stores field data at particle location between field gather and particle push
- During push, each particle stores 12 doubles + 2 integers + a field structure with 27 doubles. Common access pattern is accessing 3 components of a vector field (x,y,z)
- AoS $\rightarrow$ Strided when accessing one data type of multiple particles
- SoA $\rightarrow$ Strided when accessing multiple data types of a one particle
- AoSoA $\rightarrow$ Unit stride when accessing 3 components of a vector field of multiple particles

**AoSoA/ SoAoS?**

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$y_1$</th>
<th>$z_1$</th>
<th>$x_2$</th>
<th>$y_2$</th>
<th>$z_2$</th>
<th>...</th>
<th>$x_N$</th>
<th>$y_N$</th>
<th>$z_N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B_{x1}$</td>
<td>$B_{y1}$</td>
<td>$B_{z1}$</td>
<td>$B_{x2}$</td>
<td>$B_{y2}$</td>
<td>$B_{z2}$</td>
<td>...</td>
<td>$B_{xN}$</td>
<td>$B_{yN}$</td>
<td>$B_{zN}$</td>
</tr>
<tr>
<td>$M_{x1}$</td>
<td>$M_{y1}$</td>
<td>$M_{z1}$</td>
<td>$M_{x2}$</td>
<td>$M_{y2}$</td>
<td>$M_{z2}$</td>
<td>...</td>
<td>$M_{xN}$</td>
<td>$M_{yN}$</td>
<td>$M_{zN}$</td>
</tr>
</tbody>
</table>
Intel Advisor Classical Roofline for Electron Push Kernel, KNL quad cache

Single thread performance on KNL for **entire application**

3x Speedup achieved

Large increase in AI from blocking/sorting

Optimized performance still 10x below vector peak, AI would be high enough to reach it.

Lack of flops mainly due to gather/scatters

[T. Koskela et al, IXPUG @ ISC’17]
Main Optimizations in Electron Push: Advisor/Vtune view after

**Program metrics**

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elapsed Time</td>
<td>38.75s</td>
</tr>
<tr>
<td>Paused Time</td>
<td>34.05s</td>
</tr>
<tr>
<td>Vector Instruction Set</td>
<td>AVX512, AVX2, AVX, SSE2, SSE</td>
</tr>
<tr>
<td>Number of CPU Threads</td>
<td>16</td>
</tr>
<tr>
<td>Total GFLOP Count</td>
<td>33.81</td>
</tr>
<tr>
<td>Total GFLOPS</td>
<td>0.87</td>
</tr>
<tr>
<td>Total Arithmetic Intensity</td>
<td>0.07553</td>
</tr>
</tbody>
</table>

**Loop metrics**

<table>
<thead>
<tr>
<th>Metric</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total CPU time</td>
<td>69.04s</td>
</tr>
<tr>
<td>Time in 51 vectorized loops</td>
<td>24.30s</td>
</tr>
<tr>
<td>Time in scalar code</td>
<td>44.74s</td>
</tr>
<tr>
<td>Time including time in 21 vectorized completely unrolled loops</td>
<td></td>
</tr>
<tr>
<td>Total GFLOP Count</td>
<td>33.81</td>
</tr>
<tr>
<td>Total GFLOPS</td>
<td>0.87</td>
</tr>
</tbody>
</table>

**Vectorization Gain/Efficiency**

- Vectorized Loops Gain/Efficiency: $3.78x$ ($28\%$)
- Program Approximate Gain: $1.98x$

**Top time-consuming loops**

<table>
<thead>
<tr>
<th>Loop</th>
<th>Self Time</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>[loop in get_coef_vec at bicub_mod.F90:1423]</td>
<td>5.040s</td>
<td>5.040s</td>
</tr>
<tr>
<td>[loop in eval_bicub_1_vec at bicub_mod.F90:737]</td>
<td>3.360s</td>
<td>3.360s</td>
</tr>
<tr>
<td>[loop in i_interpol_wo_pspline_vec at one_d_cub_mod.F90:295]</td>
<td>3.080s</td>
<td>3.080s</td>
</tr>
<tr>
<td>[loop in derivs_elec_vec at pushe_vec.F90:750]</td>
<td>2.360s</td>
<td>2.360s</td>
</tr>
<tr>
<td>[loop in efield_gk_elec2_vec at efield_gk_elec2_vec.F90:152]</td>
<td>2.340s</td>
<td>2.340s</td>
</tr>
</tbody>
</table>
## Memory Access Patterns Report

<table>
<thead>
<tr>
<th>Site Location</th>
<th>Loop-Carried Dependencies</th>
<th>Strides Distribution</th>
<th>Access Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td># [loop in efield_gk_elec2_vec at efield_gk_elec2_vec.F90:1...</td>
<td>No information available</td>
<td>79% / 4% / 16%</td>
<td>Mixed strides</td>
</tr>
<tr>
<td># [loop in get_acosf_vec at bicub_mod.F90:1424]</td>
<td>No information available</td>
<td>75% / 0% / 25%</td>
<td>Mixed strides</td>
</tr>
</tbody>
</table>

### Memory Access Patterns Report

<table>
<thead>
<tr>
<th>ID</th>
<th>Stride</th>
<th>Type</th>
<th>Source</th>
<th>Nested Function</th>
<th>Variable references</th>
<th>Max. Site Footprint</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>2</td>
<td>Constant stride</td>
<td>efield_gk_elec2_vec.F90:192</td>
<td></td>
<td></td>
<td>320B</td>
</tr>
<tr>
<td>P2</td>
<td></td>
<td>Gather stride</td>
<td>bicub_mod.F90:1424</td>
<td></td>
<td></td>
<td>431KB</td>
</tr>
<tr>
<td>P3</td>
<td></td>
<td>Gather stride</td>
<td>efield_gk_elec2_vec.F90:155</td>
<td></td>
<td></td>
<td>2MB</td>
</tr>
<tr>
<td>P4</td>
<td></td>
<td>Gather stride</td>
<td>efield_gk_elec2_vec.F90:156</td>
<td></td>
<td></td>
<td>560B</td>
</tr>
<tr>
<td>P5</td>
<td></td>
<td>Gather stride</td>
<td>efield_gk_elec2_vec.F90:192</td>
<td></td>
<td></td>
<td>394KB</td>
</tr>
<tr>
<td>P6</td>
<td></td>
<td>Gather stride</td>
<td>efield_gk_elec2_vec.F90:195</td>
<td></td>
<td></td>
<td>394KB</td>
</tr>
<tr>
<td>P7</td>
<td></td>
<td>Gather stride</td>
<td>efield_gk_elec2_vec.F90:238</td>
<td></td>
<td></td>
<td>394KB</td>
</tr>
<tr>
<td>P8</td>
<td>1</td>
<td>Parallel site information</td>
<td>bicub_mod.F90:1424</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P9</td>
<td>1</td>
<td>Parallel site information</td>
<td>efield_gk_elec2_vec.F90:153</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P12</td>
<td>0</td>
<td>Uniform stride</td>
<td>bicub_mod.F90:1424</td>
<td></td>
<td></td>
<td>8B</td>
</tr>
<tr>
<td>P13</td>
<td>0</td>
<td>Uniform stride</td>
<td>efield_gk_elec2_vec.F90:152</td>
<td></td>
<td></td>
<td>8B</td>
</tr>
<tr>
<td>P14</td>
<td>0</td>
<td>Uniform stride</td>
<td>efield_gk_elec2_vec.F90:155</td>
<td></td>
<td></td>
<td>8B</td>
</tr>
<tr>
<td>P15</td>
<td>0</td>
<td>Uniform stride</td>
<td>efield_gk_elec2_vec.F90:155</td>
<td></td>
<td></td>
<td>4B</td>
</tr>
<tr>
<td>P16</td>
<td>0</td>
<td>Uniform stride</td>
<td>efield_gk_elec2_vec.F90:156</td>
<td></td>
<td></td>
<td>64B</td>
</tr>
<tr>
<td>P17</td>
<td>0</td>
<td>Uniform stride</td>
<td>efield_gk_elec2_vec.F90:156</td>
<td></td>
<td></td>
<td>4B</td>
</tr>
<tr>
<td>P18</td>
<td>0</td>
<td>Uniform stride</td>
<td>efield_gk_elec2_vec.F90:161</td>
<td></td>
<td></td>
<td>4B</td>
</tr>
<tr>
<td>P19</td>
<td>0</td>
<td>Uniform stride</td>
<td>efield_gk_elec2_vec.F90:192</td>
<td></td>
<td></td>
<td>64B</td>
</tr>
</tbody>
</table>
Intel Advisor Integrated Roofline for Five Hottest Loops, KNL quad cache

KNL, 16 threads

![Graph showing performance vs. arithmetic intensity for different operations and cache levels.](image-url)
Electron Push Speedup

XGC1 Timing on 1024 Cori KNL nodes in quadrant flat mode.
Toypush Mini-App
Toypush: Introduction/Motivation

- The electron push in XGC1 is practically embarrassingly parallel → only on-core optimizations matter, scaling is almost perfect
- The electron push “kernel” is still rather complex, ~ 20k lines of F90 code, with a deep subroutine call tree, which makes it hard to analyze and optimize
- To determine a “speed of light” for a particle pusher on KNL, we wrote Toypush, a small kernel with <1k lines of code with the same main loops as the XGC1 electron push
  - Triangle interpolation
  - Triangle search
  - Force calculation
  - RK4 push
- Toypush was optimized in an Intel dungeon session, with encouraging results [T. Koskela, CUG’17]
ToyPush Performance on Roofline

- Intel Advisor, cache-aware roofline, single thread on KNL
- Good vector performance from the Force Calculation kernel
- Interpolate kernel close to theoretical peak, Search close to by L2 bandwidth

Marker size $\approx$ CPU time

- Single thread performance
- 10x speedup for Interpolate kernel
- 3x speedup for Search
- https://github.com/tkoskela/toypush
Toypush Conclusions

• We optimized a mini-app to attain peak on-node performance in the electron push algorithm on KNL.
  – Main bottlenecks are search and interpolation
  – We were successful in vectorizing and pushing them close to maximum attainable performance based on the roofline model

• Porting optimizations to XGC1 not as easy as we had hoped, however a 3x speedup in electron push has been achieved
  – Electron push remains the most expensive kernel, followed by Poisson solver (PETSc linear algebra)

• Toypush is a useful mini-app benchmark for particle pushing applications on unstructured meshes
Charge Deposition
In XGC1 Electron Time Scale is Separated From the Ion Push in a Sub-Cycling Loop.
Charge Deposition Algorithm

• Charge deposition bins particle charge density from the particles onto the grid nodes

• In XGC1 grid is only decomposed into planes → each MPI process deposits charge from its particles on entire plane.
  – Aim to run with 200,000 grid element planes on KNL
  – Best code performance (overall) with 4 ranks per node, aim to run ~2,000,000 particles per rank
  – Electron binning array size = grid elements per plane * 2 planes → number of electrons >> array size
  – Ion binning array size = electron binning array size * O(10) velocity space grid. → number of ions << array size

• Deposition is threaded with OpenMP (64 threads)
  – Need to avoid data races when writing to binning array
Initial State: Poor Weak Scaling of Charge Deposition

- At small scale the cost of charge deposition is small compared to electron push. Need to scale it up at that level.
- Ions 5x more expensive than electrons because of gyro-averaging
- Nearly linear slowdown with problem size
Allocate private arrays for each thread

Each thread initializes its private array to 0

Each thread deposits particles to private array → avoids data races

Reduce private arrays manually on master thread
Optimization I: OMP reduction

Allocate single array
→ 64x smaller memory footprint

!$omp reduction(+) → Creates private arrays and initializes to 0

Deposit particles to private arrays
→ Avoids data races

Reduce private arrays at the end of parallel region
Optimization II: Atomic update

Allocate single array  
→ 64x smaller memory footprint

Initialize single array to 0  
→ 64x faster with threads

Deposit particles atomically  
→ Avoid data races

No need for reduction
KNL Performance Results

"Electrons" vs "Ions"
Atomic Updates Beat Reduction Only When the Number of Updates is Relatively Small

- Atomic overhead is constant/particle while reduction overhead is constant/grid
- Note: Atomic code does not vectorize ➔ not significant as long as it scales well
• Ideal scaling of electron charge deposition
• Some performance degradation in ion charge deposition, but > 10x faster than before at 2048 nodes.
  • “Fast enough” to be insignificant compared to particle push
Summary And Conclusions

• Optimizations have improved vectorization and memory access patterns in XGC1 electron push kernel
  – 3x gained in total performance
  – Optimized electron push kernel has roughly equal per-node performance on KNL and Haswell
  – Not memory bandwidth bound → Focus on enabling vectorization, improving memory access patterns
  – Theoretically still room for ~10x improvement. Limited by Gather/Scatter latency, Memory alignment, Integer operations, Type conversions, ...

• Lessons learned from optimization
  – Achieving good vectorization can require major code refactoring, especially if the code has long subroutine call chains
  – Memory latency is hard to analyze
  – Large array initializations are expensive
  – When writing OpenMP code, take advantage of OpenMP features (Besides “omp parallel do”)
Thank you!
Performance Comparison
Performance Comparison

![Bar chart showing performance comparison between baseline and optimized versions for knl and haswell.](image)
Scaling Studies
## Strong Scaling Parameters

<table>
<thead>
<tr>
<th>Compute Nodes</th>
<th>Grid Nodes Per Rank</th>
<th>Particles Per Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>448</td>
<td>12.2 M</td>
</tr>
<tr>
<td>512</td>
<td>224</td>
<td>6.1 M</td>
</tr>
<tr>
<td>1024</td>
<td>112</td>
<td>3.1 M</td>
</tr>
<tr>
<td>2048</td>
<td>56</td>
<td>1.5 M</td>
</tr>
<tr>
<td>4096</td>
<td>28</td>
<td>0.75 M</td>
</tr>
</tbody>
</table>

- 16 MPI ranks per Node, 16 OpenMP Threads per rank
- 5 Bn total particles
- 57 000 total grid nodes per plane, 32 planes
- Quadrant Cache mode
XGC1 Strong Scaling up to 4096 KNL Nodes

16 MPI ranks per node, 16 OpenMP threads per rank.

Strong scaling for problem size of 25 Bn ions and electrons, grid representative of present production runs (DIII-D tokamak)

Ideal Scaling in electron push

30% scaling deficit in main loop at 4096 nodes (half machine size)
### Particle Weak Scaling Parameters

<table>
<thead>
<tr>
<th>Compute Nodes</th>
<th>Grid Nodes Per Rank</th>
<th>Particles Per Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>3584</td>
<td>0.4 M</td>
</tr>
<tr>
<td>64</td>
<td>1792</td>
<td>0.4 M</td>
</tr>
<tr>
<td>128</td>
<td>896</td>
<td>0.4 M</td>
</tr>
<tr>
<td>256</td>
<td>448</td>
<td>0.4 M</td>
</tr>
<tr>
<td>512</td>
<td>224</td>
<td>0.4 M</td>
</tr>
<tr>
<td>1024</td>
<td>112</td>
<td>0.4 M</td>
</tr>
<tr>
<td>2048</td>
<td>56</td>
<td>0.4 M</td>
</tr>
</tbody>
</table>

- 16 MPI ranks per Node, 16 OpenMP Threads per rank
- 57 000 total grid nodes per plane, 32 planes
- Quadrant Cache mode
Weak Scaling in particle structure size for fixed grid size

Grid representative of present production runs (DIII-D tokamak)

60-70% of time in electron push

Slowdown from 32 to 2048 nodes: 20%

~50% slowdown at full machine size (9600 nodes) by extrapolation
## Weak Scaling Parameters

<table>
<thead>
<tr>
<th>Compute Nodes</th>
<th>Grid Nodes Per Rank</th>
<th>Total Grid Nodes</th>
<th>Particles Per Rank</th>
<th>Total Particles</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>117</td>
<td>3,750</td>
<td>1.75 M</td>
<td>900 M</td>
</tr>
<tr>
<td>256</td>
<td>117</td>
<td>7,500</td>
<td>1.75 M</td>
<td>1.8 Bn</td>
</tr>
<tr>
<td>512</td>
<td>117</td>
<td>15,000</td>
<td>1.75 M</td>
<td>3.6 Bn</td>
</tr>
<tr>
<td>1024</td>
<td>117</td>
<td>30,000</td>
<td>1.75 M</td>
<td>7.2 Bn</td>
</tr>
<tr>
<td>2048</td>
<td>117</td>
<td>60,000</td>
<td>1.75 M</td>
<td>14.4 Bn</td>
</tr>
<tr>
<td>4096</td>
<td>117</td>
<td>120,000</td>
<td>1.75 M</td>
<td>28.8 Bn</td>
</tr>
<tr>
<td>8192</td>
<td>117</td>
<td>240,000</td>
<td>1.75 M</td>
<td>57.6 Bn</td>
</tr>
</tbody>
</table>

- 16 MPI ranks per Node, 16 OpenMP Threads per rank
- Quadrant Cache mode
XGC1 Weak Scaling

Weak Scaling in particle structure size and grid size

Grid representative of production runs for Cori (JET tokamak)

60-70% of time in electron push

Slowdown from 128 to 2048 nodes: 16%

~90% slowdown at 8192 nodes.

Poor Weak Scaling at large scale caused by load imbalance
Single node thread scaling of electron push kernel

Performance gain from MCDRAM only when using more than 2 threads/core \( \rightarrow \) KNL outperforms Haswell node when all logical threads are used.

KNL: 64 physical cores/4 hyper threads
Haswell: 32 physical cores/2 hyper threads

KMP_AFFINITY=compact
KMP_PLACE_THREADS=1
T \( (N \leq 64) \)
2T \( (N = 128) \)
4T \( (N = 256) \)
OMP_NUM_THREADS=N
allocate(density(nnnode,2,nvel,nthreads))

!$omp parallel do ...
do ith = 1,nThreads
  density(:,:,ith) = 0
  do iprt = 1,nParticles_per_thread
    call deposit_charge(iprt,density(:,:,ith))
  end do
end do

!$omp parallel do ...
do ith = 1,nThreads
  density(:,:,1) = density(:,:,1) + density(:,:,ith)
end do

Allocate private copy for each thread

Initialize all private copies to 0

Deposit particles to private copy – avoids data races

Reduce private copies
allocate(density(nnode,2,nvel))

!$omp parallel do reduction(+:density) ...
do iprt = 1,nParticles_per_thread
call deposit_charge(iprt,density)
end do

Allocate single copy
→ 64x smaller memory footprint

Declare reduction(+) → Creates private copies and initializes to 0

Deposit particles

Reduce private copies at the end of parallel region
allocate(density(nnode,2,nvel))

!$omp parallel do ...
do inode = 1,nNodes  
density(inode,:,:) = 0
end do

!$omp parallel do shared(density) ...
do iprt = 1,nParticles_per_thread  
!$omp atomic  
call deposit_charge(iprt,density)
end do

Allocate single copy  
$\rightarrow$ 64x smaller memory footprint

Initialize single copy to 0  
$\rightarrow$ 64x faster with threads

Deposit particles atomically  
$\rightarrow$ Avoid data races