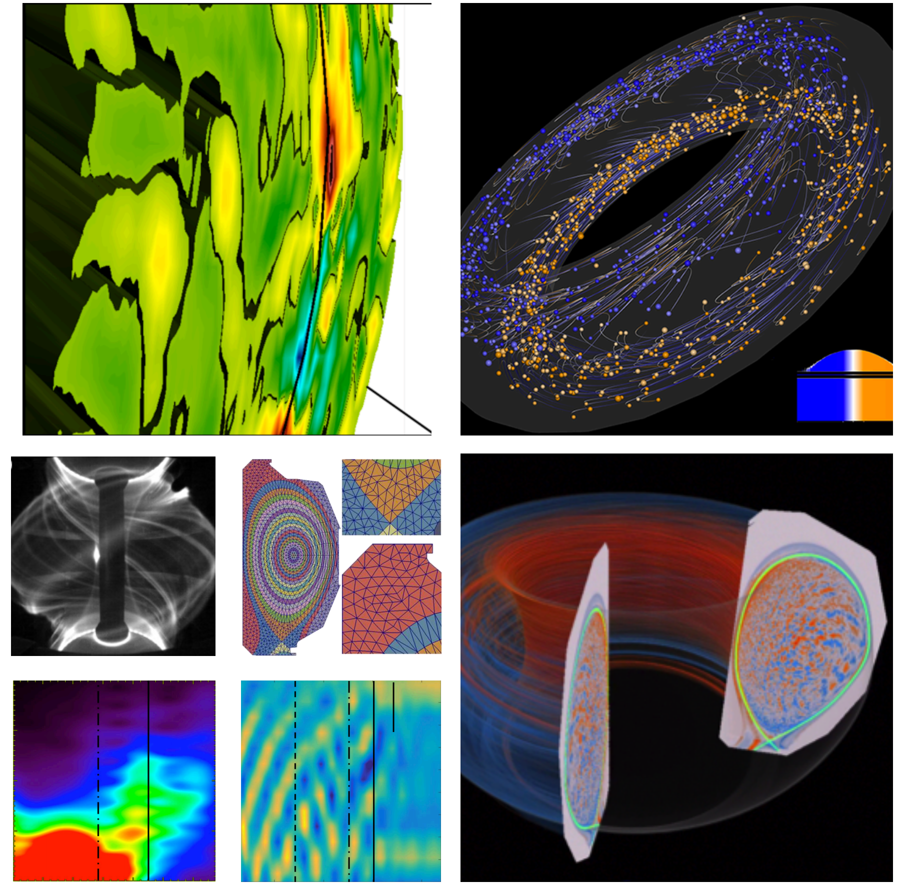
Performance Optimization of XGC1 on Cori KNL



Tuomas Koskela

NESAP postdoc

NERSC / LBNL

tkoskela@lbl.gov

February 27, 2018

U.S. DEPARTMENT OF ENERGY | Office of Science

BERKELEY LAB

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

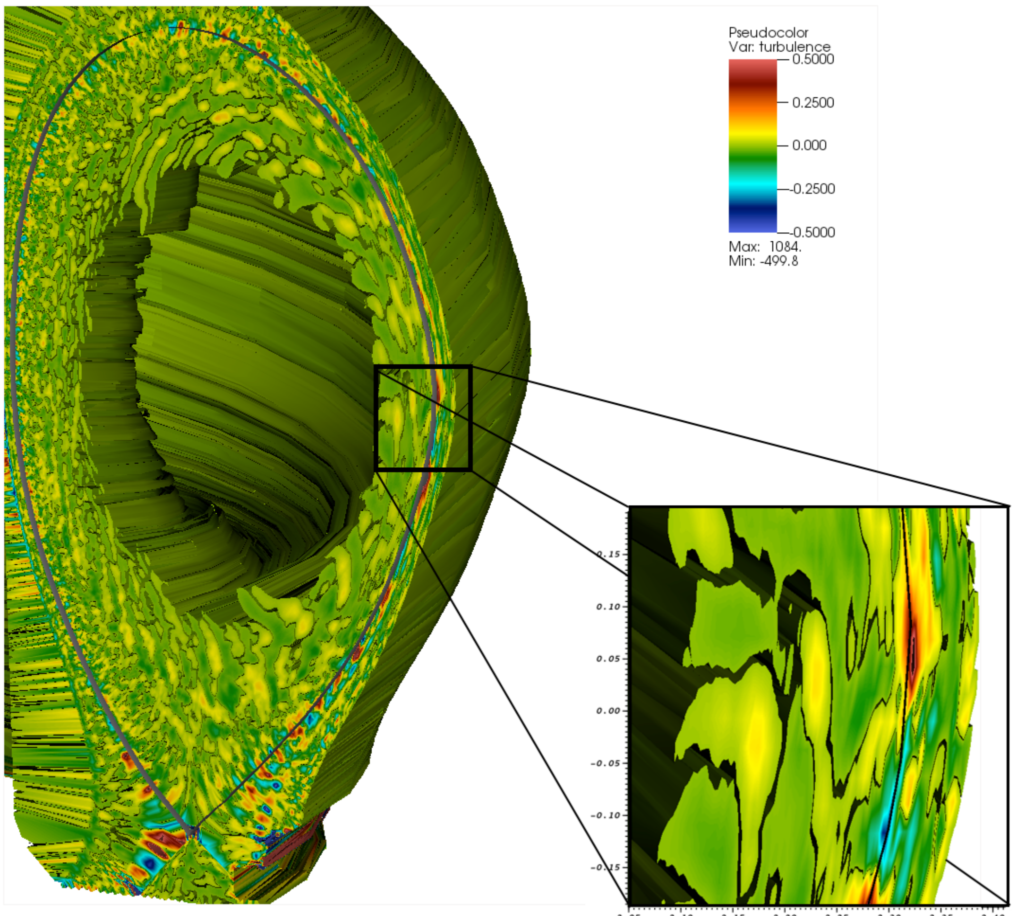
- Introduction to XGC1
- Particle Push Vectorization and Data Structure Reordering Optimizations
- Toypush mini-app
- Charge Deposition Threading Optimizations
- Conclusions

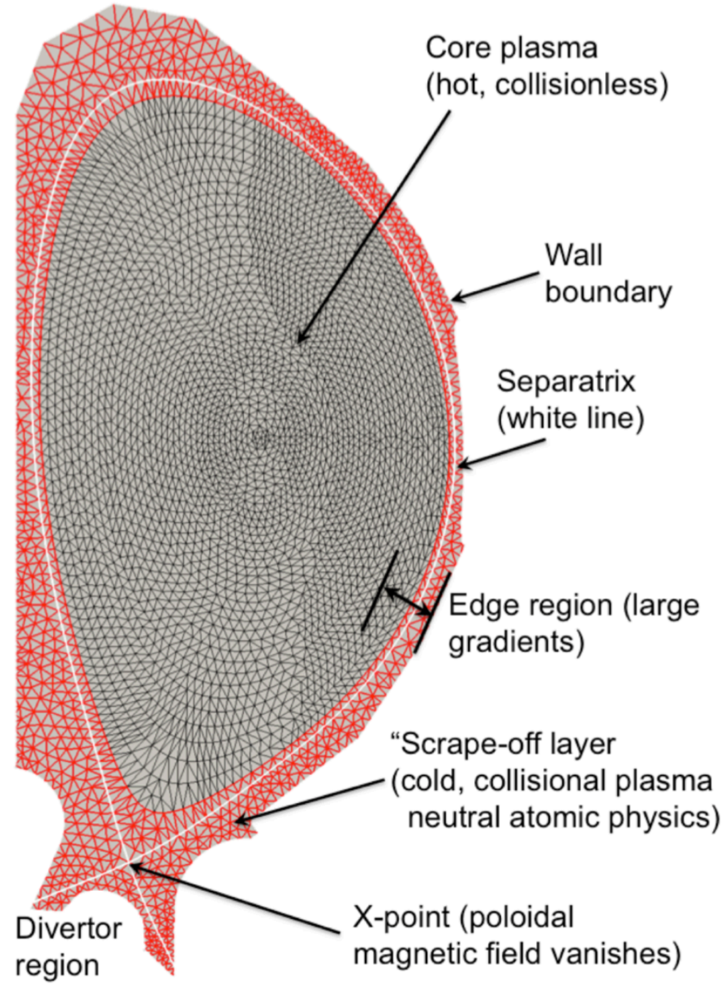
Cori at NERSC

- **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

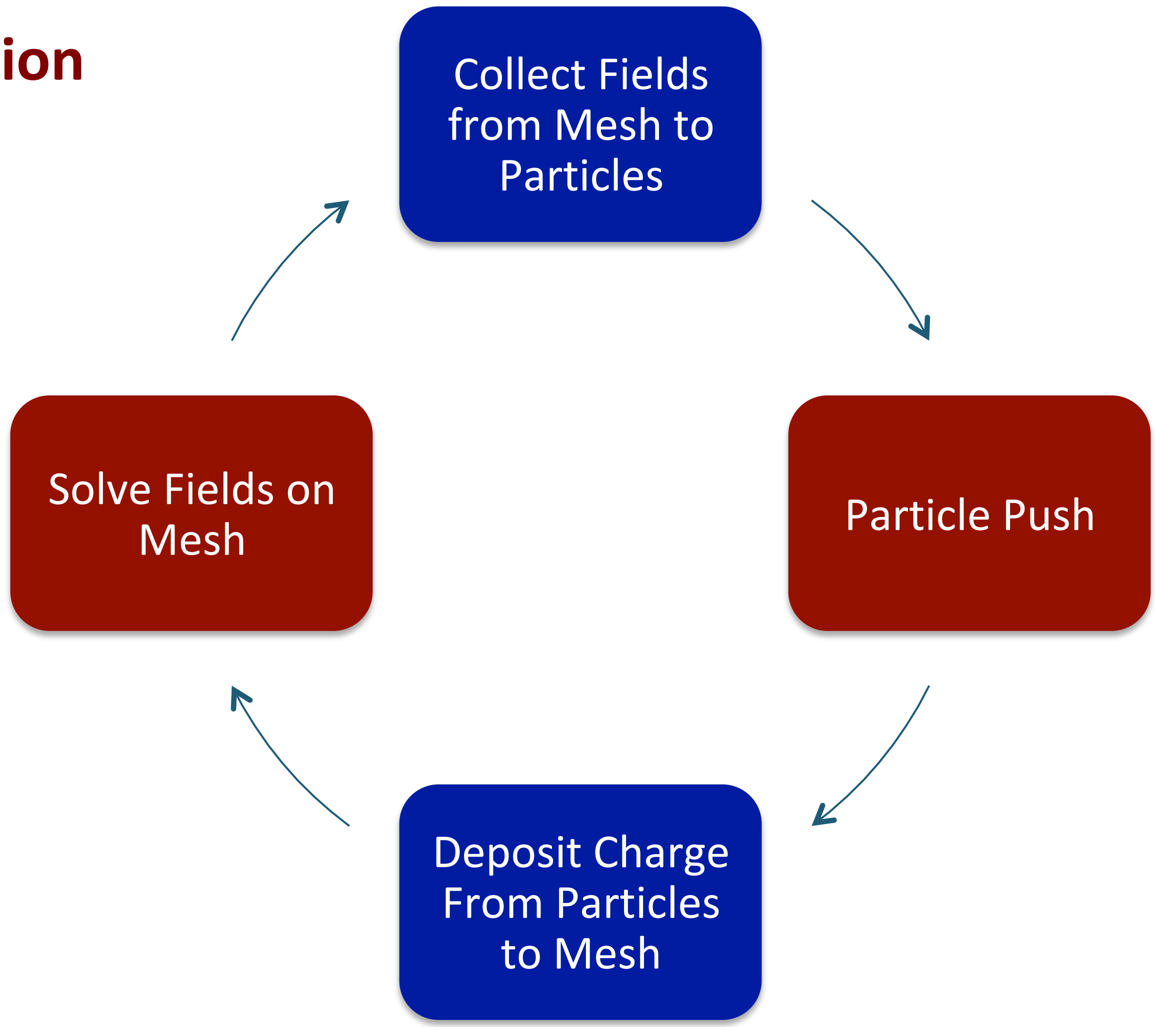




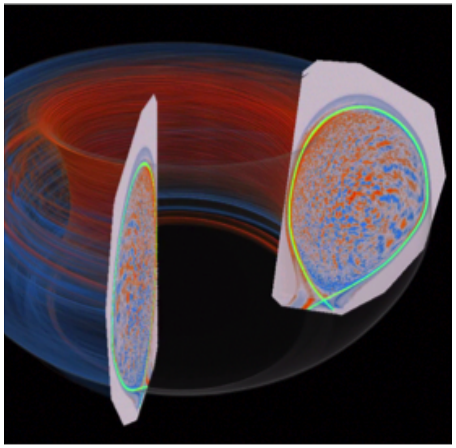
PI: CS Chang (PPPL) | ECP: High-Fidelity Whole Device Modeling of Magnetically Confined Fusion Plasma

Basic Plasma PIC Code Flowchart

Computation
Mapping



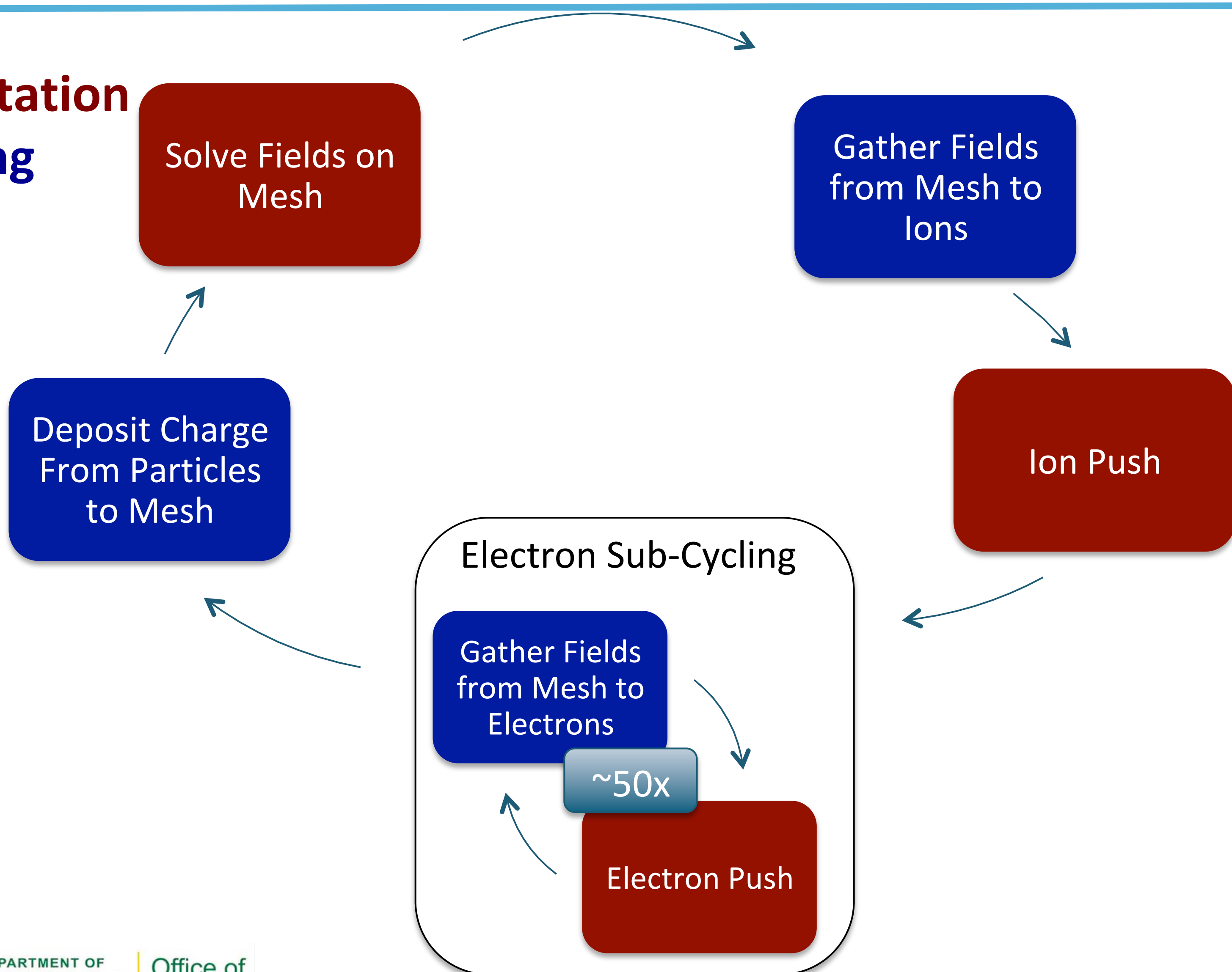
XGC1 Unique Optimization Challenges



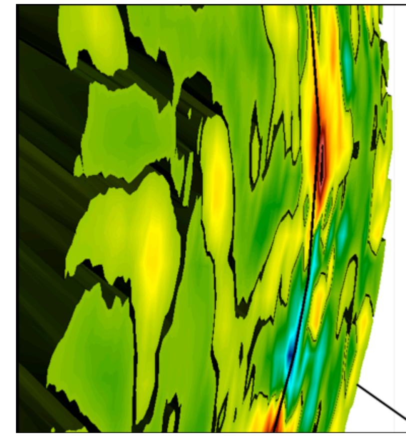
- **Complicated Tokamak Geometry**
  - Unstructured gridin 2D (poloidal) plane(s)
  - Nontrivial field-following (toroidal) mapping between planes
  - Full-f model, exascale simulations will have 10 000 particles per cell, 1 000 000 cells per domain, 100 toroidal domains.
- **Gyrokinetic Equation of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

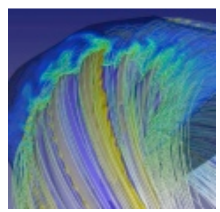
In XGC1 Electron Time Scale is Separated From the Ion Push in a Sub-Cycling Loop

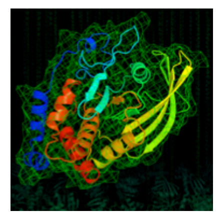
Computation Mapping

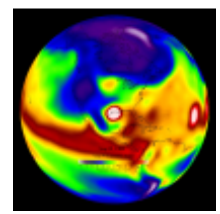


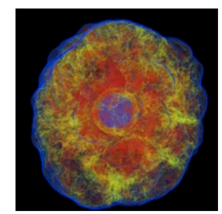
Electron Push Sub-Cycling

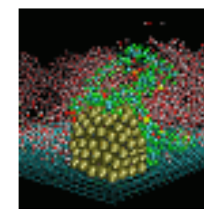




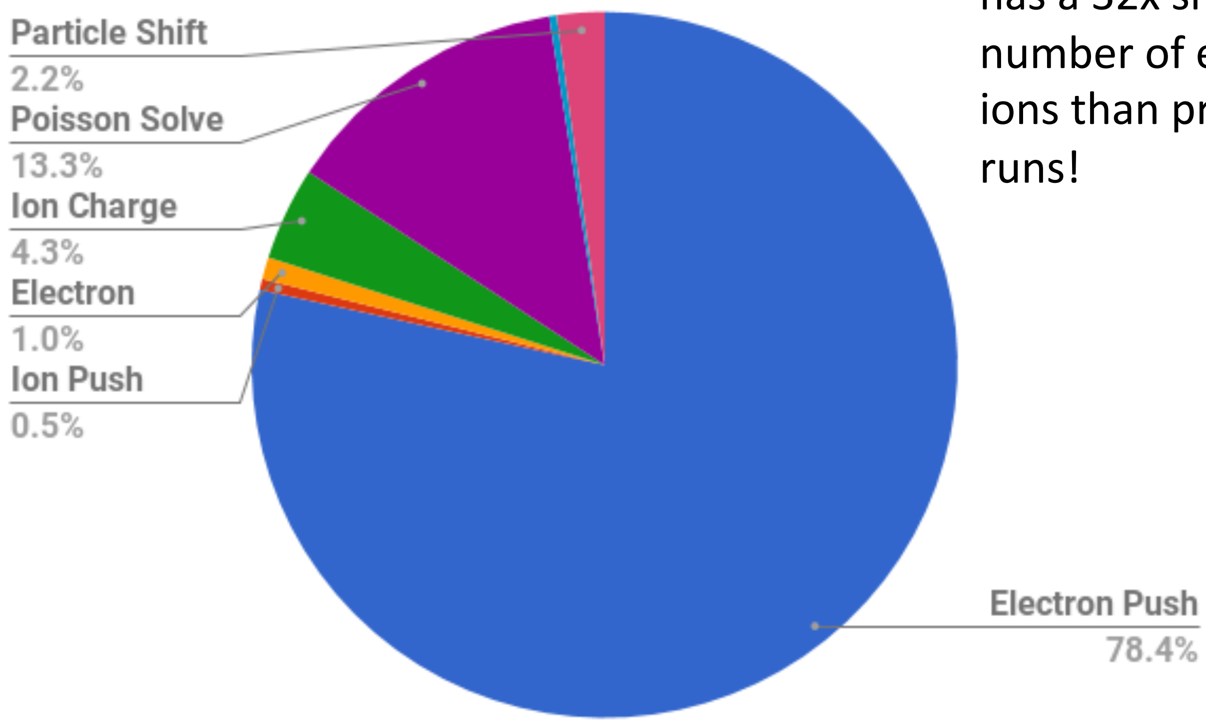








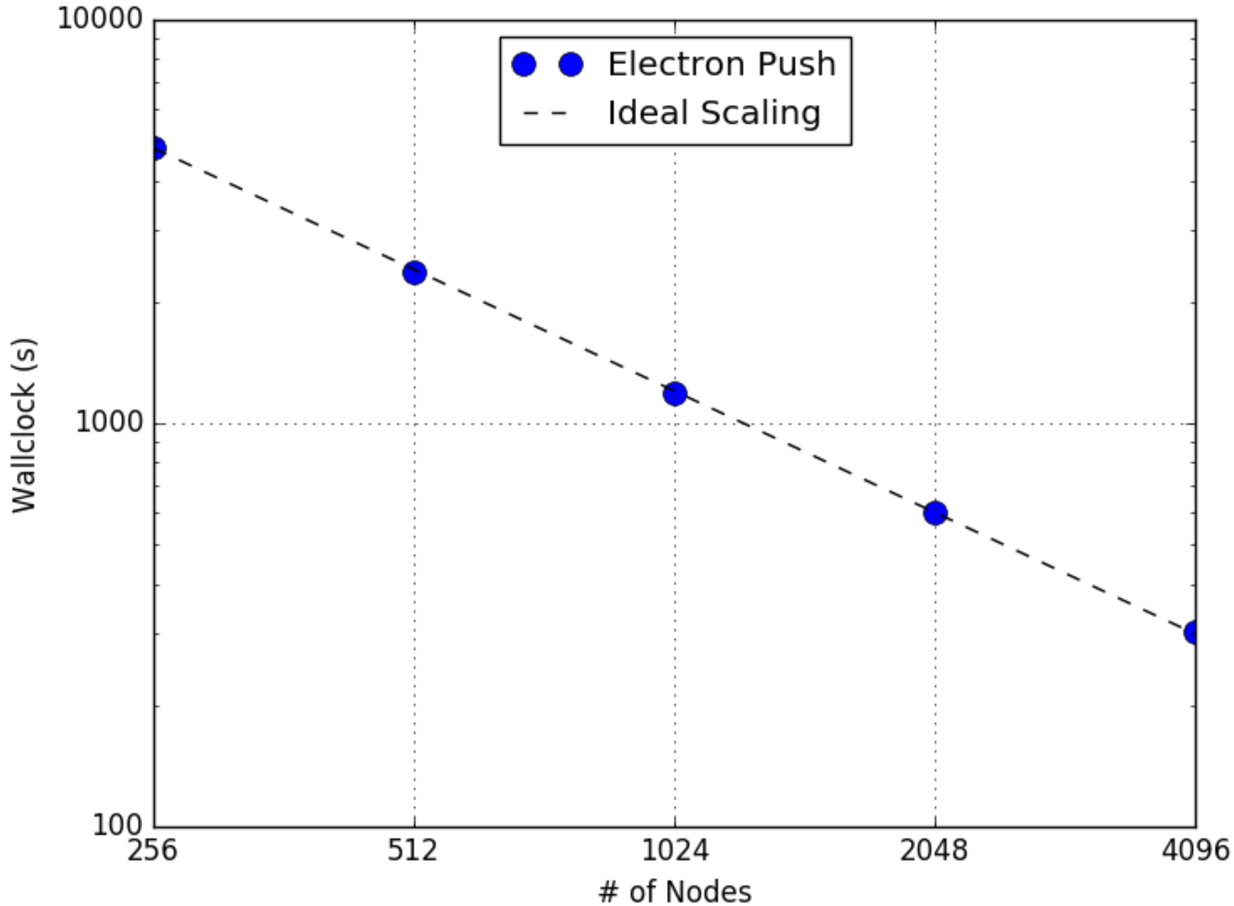
Motivation: XGC1 CPU time is dominated by electron push sub-cycle



Note: This run actually has a 32x smaller number of electrons & ions than production runs!

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

Motivation: Ideal Strong Scaling* of Electron Sub-Cycling On Cori



KNL, quadrant cache

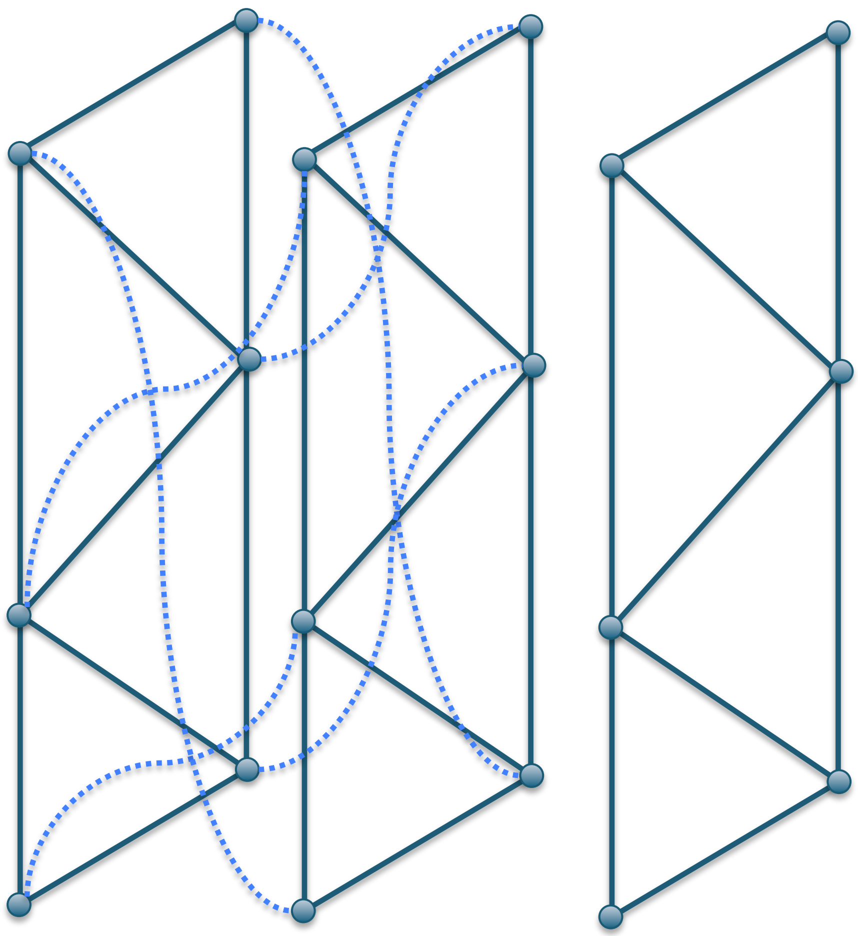
Hybrid MPI/OpenMP

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

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

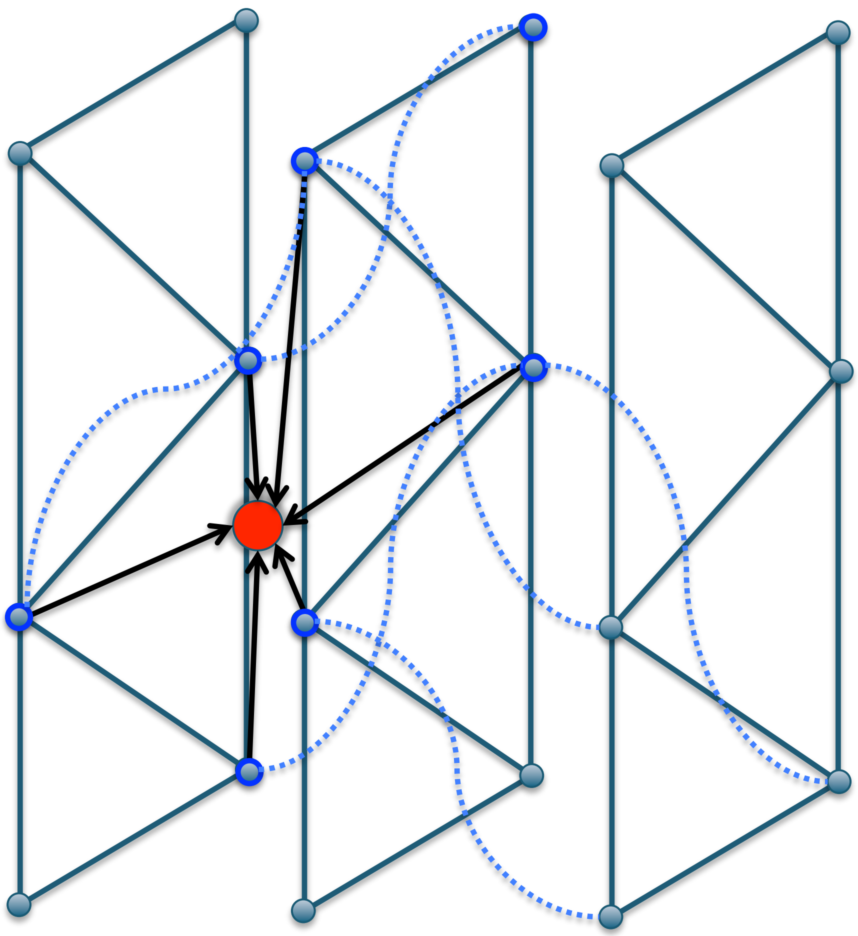
*Requires good load balancing

(Simplified) Field following node mapping



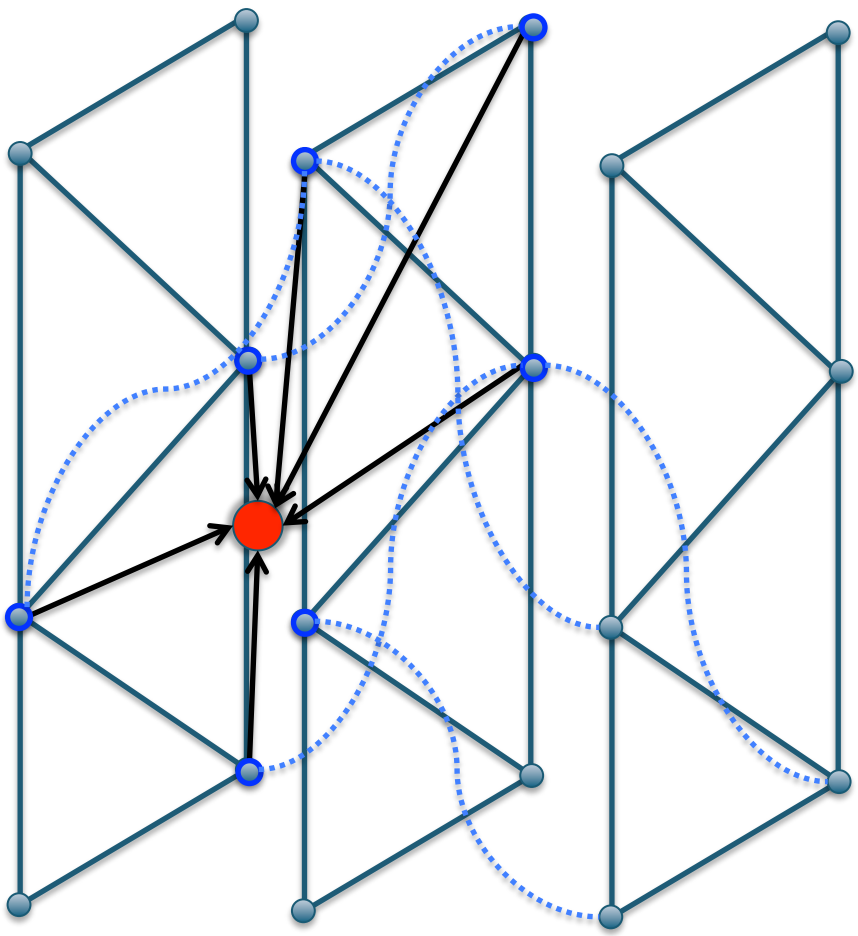
- **Grid consists of poloidal (2D) planes that have an identical set of nodes each.**
- **Nodes connect to neighboring planes by (approximately) following the magnetic field**

(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

(Simplified) Particle Push Algorithm



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

Program metrics

Elapsed Time	16.88s	Paused Time	8.10s
Vector Instruction Set	AVX512, AVX2, AVX	Number of CPU Threads	16
Total GFLOP Count	20.35	Total GFLOPS	1.21
Total Arithmetic Intensity	0.08005		

Loop metrics

Metrics	Total	
Total CPU time	136.46s	100.0%
Time in **1** vectorized loop	0.02s	
Time in scalar code including time in **19** vectorized completely unrolled loops	136.44s	100.0%
Total GFLOP Count	20.35	100.0%
Total GFLOPS	1.21	

Vectorization Gain/Efficiency

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

Elapsed Time: 16.264s

CPU Time:	138.169s	
Memory Bound:		
L2 Hit Rate:	89.6%	
L2 Hit Bound:	6.4%	of Clockticks
L2 Miss Bound:	**10.0%**	**of Clockticks**
MCDRAM Bandwidth Bound:	0.0%	
DRAM Bandwidth Bound:	0.0%	of Elapsed Time
L2 Miss Count:	90,002,700	
MCDRAM Hit Rate:	100.0%	
MCDRAM HitM Rate:	84.9%	
Total Thread Count:	17	
Paused Time:	7.490s	

Top time-consuming loops

Loop	Self Time	Total Time
[loop in search_tr2 at search.F90:736]	14.815s	14.815s
[loop in derivs_elec_vec at derivs_elec_vec.F90:118]	3.380s	3.380s
[loop in efield_gk_elec at pushe.F90:1089]	2.780s	2.780s
[loop in pushe_1step2_vec_ompparallel_for@39 at pushe_1step2_vec.F90:49]	2.280s	110.906s
[loop in derivs_single_with_e_elec_vec at derivs_single_with_e_elec_vec.F90:47]	2.100s	40.902s

Search

Force

E-Field

Main Bottlenecks in Electron Push

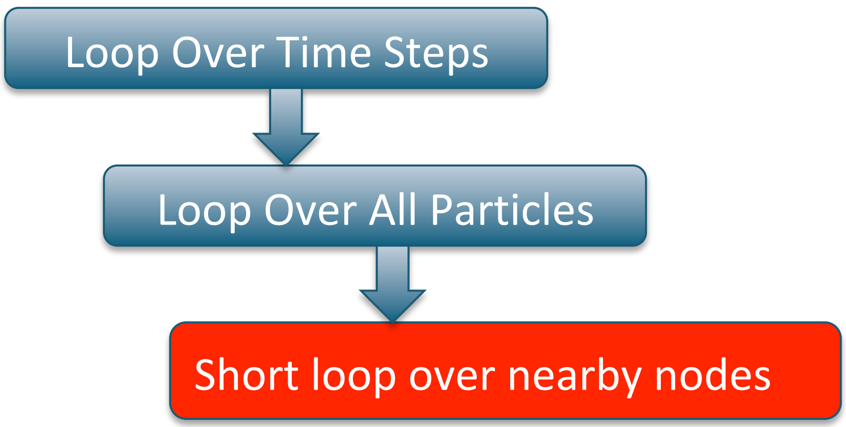
- **E and B Field Interpolation**
  - Inner loops in function calls over nearby grid nodes with short trip counts make auto-vectorization ineffective
  - Indirect grid access produces gather/scatter instructions
- **Search on Unstructured Mesh**
  - Multiple exit conditions
- **Force Calculation**
  - Strided memory access in complicated data types
  - Cache unfriendly

Main Optimizations in Electron Push

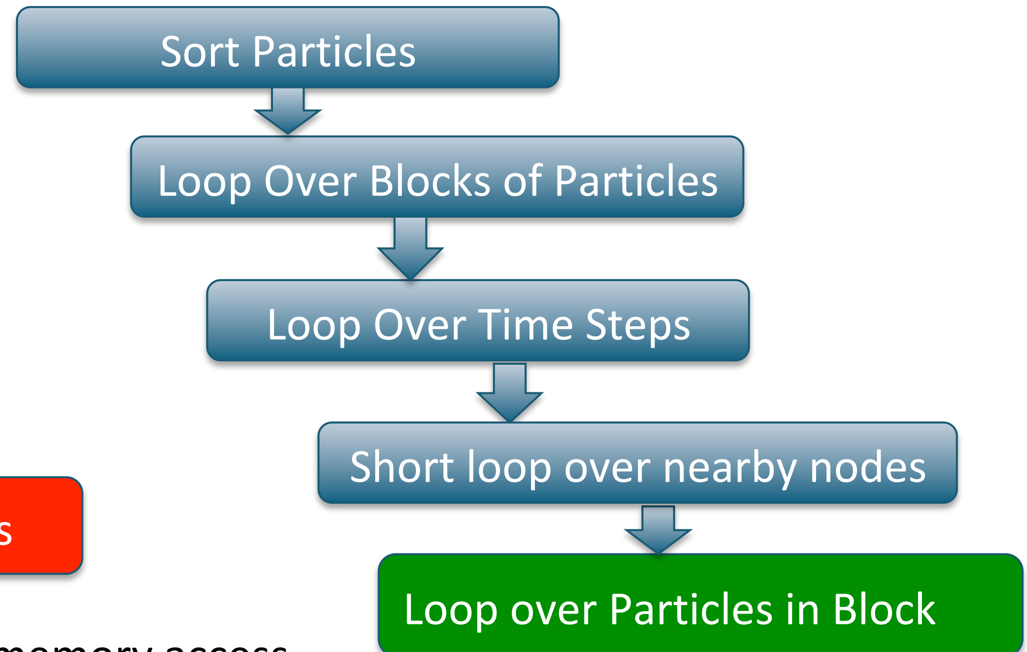
- **Enabling Vectorization**
  - Insert loops over blocks of particles inside short trip count loops to enable automatic vectorization
  - Sort particles to reduce random memory accesses
  - Tile particle loop to improve cache reuse
- **Data Structure Reordering**
  - Store field and particle data in 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



Vectorized code



- 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
  → 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

Number of particles per block: 32

x_1	y_1	z_1	B_{x1}	B_{y1}	B_{z1}	...
x_2	y_2	z_2	B_{x2}	B_{y2}	B_{z2}	...
:	:	:	:	:	:	...
x_N	y_N	z_N	B_{xN}	B_{yN}	B_{zN}	...

SoA

x_1	x_2	...	x_N
y_1	y_2	...	y_N
z_1	z_2	...	z_N
B_{x1}	B_{x2}	...	B_{xN}
B_{y1}	B_{y2}	...	B_{yN}
B_{z1}	B_{z2}	...	B_{zN}
:	:	:	:

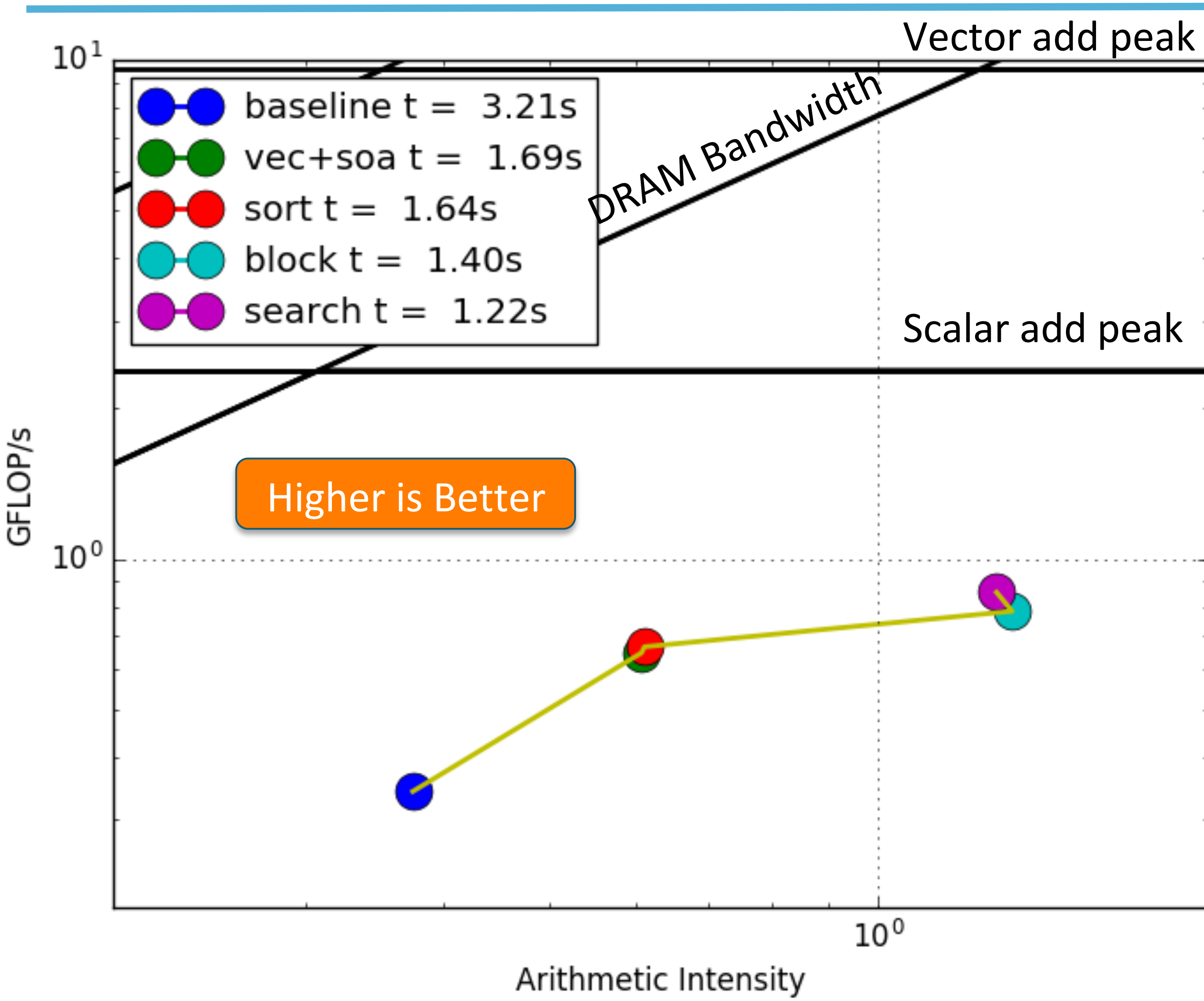
Reorder Particle and Field Data Structures

- 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
- AoSoA→ Unit stride when accessing 3 components of a vector field of multiple particles

AoSoA/ SoAoS?

x_1	y_1	z_1	x_2	y_2	z_2	...	x_N	y_N	z_N
B_{x1}	B_{y1}	B_{z1}	B_{x2}	B_{y2}	B_{z2}	...	B_{xN}	B_{yN}	B_{zN}
:	:	:	:	:	:	x	:	:	:
M_{x1}	M_{y1}	M_{z1}	M_{x2}	M_{y2}	M_{z2}	...	M_{xN}	M_{yN}	M_{zN}

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

Elapsed Time	38.75s	Paused Time	34.05s
Vector Instruction Set	AVX512, AVX2, AVX, SSE2, SSE	Number of CPU Threads	16
Total GFLOP Count	33.81	Total GFLOPS	0.87
Total Arithmetic Intensity	0.07553		

Loop metrics

Metrics	Total	
Total CPU time	69.04s	100.0%
Time in **51** vectorized loops	24.30s	35.2%
Time in scalar code including time in **21** vectorized completely unrolled loops	44.74s	64.8%
Total GFLOP Count	33.81	100.0%
Total GFLOPS	0.87	

Vectorization Gain/Efficiency

Vectorized Loops Gain/Efficiency	3.78x	28%
Program Approximate Gain	1.98x	

Elapsed Time: 37.198s

CPU Time:	68.089s	
Memory Bound:		
L2 Hit Rate:	100.0%	
L2 Hit Bound:	12.5%	of Clockticks
L2 Miss Bound:	0.0%	of Clockticks
MCDRAM Bandwidth Bound:	0.0%	
DRAM Bandwidth Bound:	0.0%	of Elapsed Time
L2 Miss Count:	0	
MCDRAM Hit Rate:	100.0%	
MCDRAM HitM Rate:	83.2%	
Total Thread Count:	16	
Paused Time:	32.798s	

Top time-consuming loops

Loop	Self Time	Total Time
[loop in get_acoef_vec at bicub_mod.F90:1423]	5.040s	5.040s
[loop in eval_bicub_1_vec at bicub_mod.F90:737]	3.360s	3.360s
[loop in i_interpol_wo_pspline_vec at one_d_cub_mod.F90:295]	3.080s	3.080s
[loop in derivs_elec_vec at pushe_vec.F90:750]	2.360s	2.360s
[loop in efield_gk_elec2_vec at efield_gk_elec2_vec.F90:152]	2.340s	2.340s

Memory Access Patterns Remain an Issue

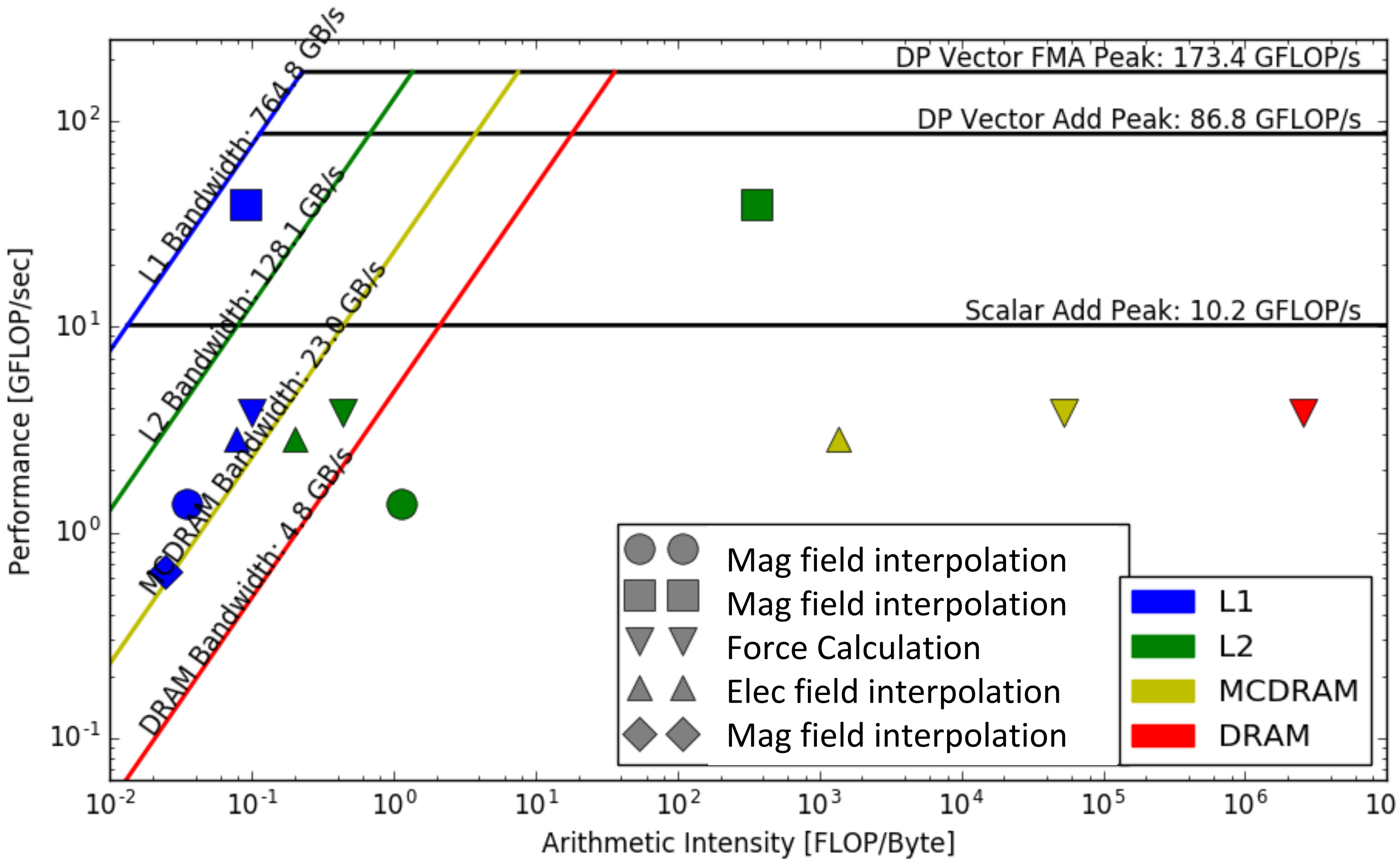
Site Location	Loop-Carried Dependencies	Strides Distribution	Access Pattern
[loop in efield_gk_elec2_vec at efield_gk_elec2_vec.F90:1…	No information available	79% / 4% / 18%	Mixed strides
[loop in get_acoef_vec at bicub_mod.F90:1424]	No information available	75% / 0% / 25%	Mixed strides

Memory Access Patterns Report | Dependencies Report | Recommendations

ID		Stride	Type	Source	Nested Function	Variable references	Max. Site Footpr
P1		2	Constant stride	efield_gk_elec2_vec.F90:192			320B
P2			Gather stride	bicub_mod.F90:1424			431KB
P3			Gather stride	efield_gk_elec2_vec.F90:155			2MB
P4			Gather stride	efield_gk_elec2_vec.F90:156			560B
P5			Gather stride	efield_gk_elec2_vec.F90:192			394KB
P6			Gather stride	efield_gk_elec2_vec.F90:195			394KB
P7			Gather stride	efield_gk_elec2_vec.F90:238			394KB
P8			Parallel site information	bicub_mod.F90:1424			
P9			Parallel site information	efield_gk_elec2_vec.F90:153			
P12		0	Uniform stride	bicub_mod.F90:1424			8B
P13		0	Uniform stride	efield_gk_elec2_vec.F90:152			8B
P14		0	Uniform stride	efield_gk_elec2_vec.F90:155			8B
P15		0	Uniform stride	efield_gk_elec2_vec.F90:155			4B
P16		0	Uniform stride	efield_gk_elec2_vec.F90:156			64B
P17		0	Uniform stride	efield_gk_elec2_vec.F90:156			4B
P18		0	Uniform stride	efield_gk_elec2_vec.F90:161			4B
P19		0	Uniform stride	efield_gk_elec2_vec.F90:192			64B

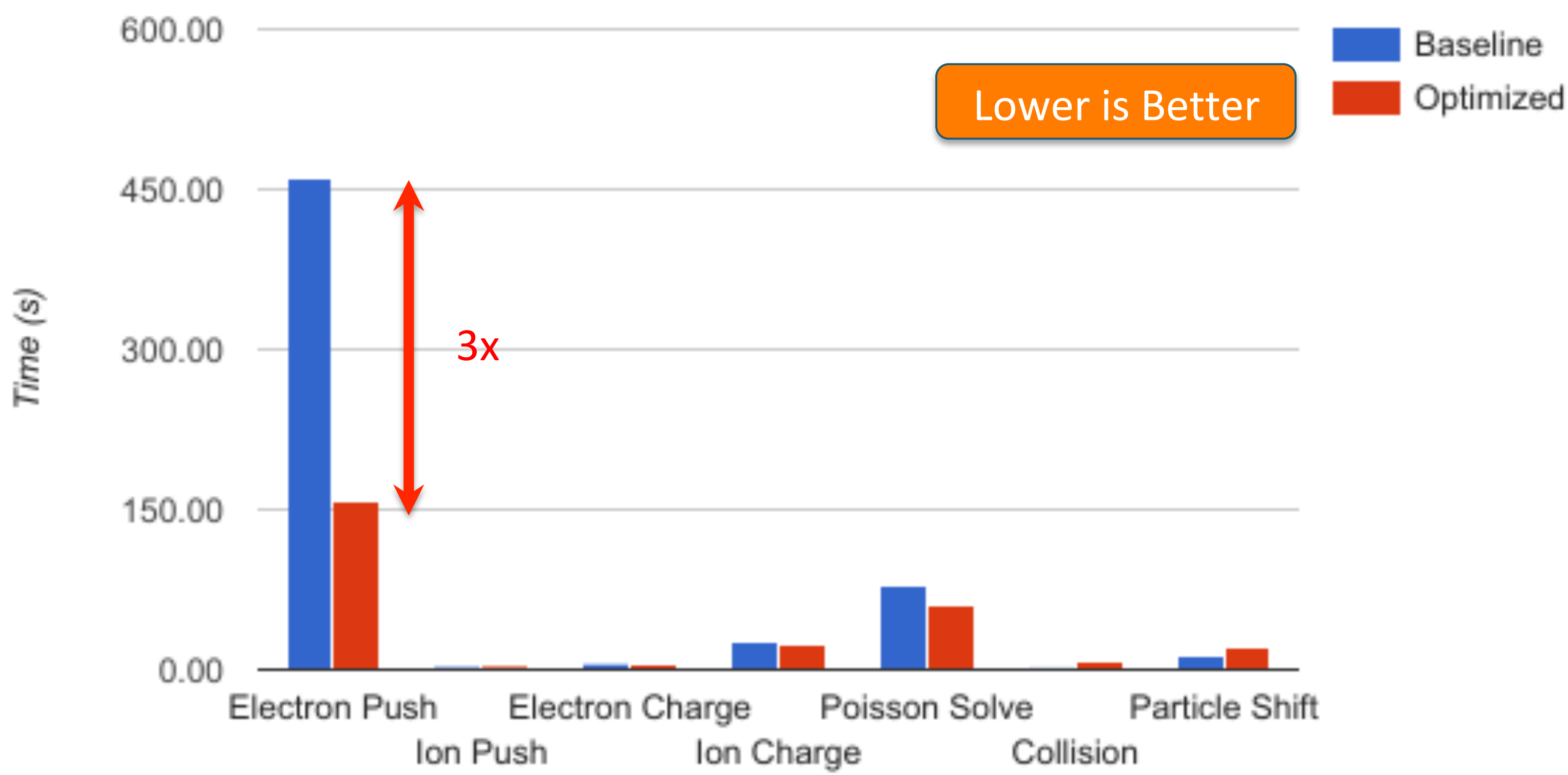
Intel Advisor Integrated Roofline for Five Hottest Loops, KNL quad cache

KNL, 16 threads



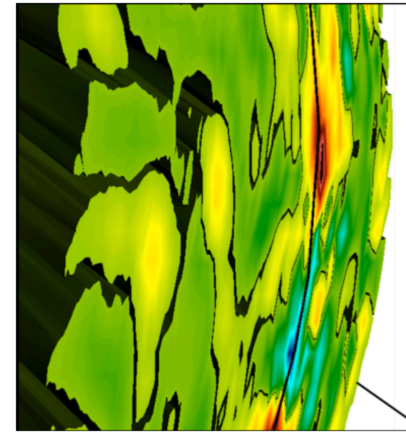
[T. Koskela et al, submitted to ISC'18]

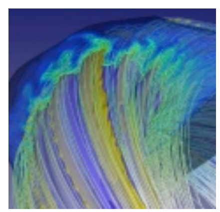
Electron Push Speedup

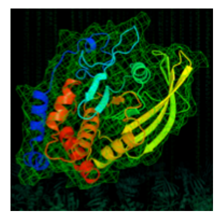


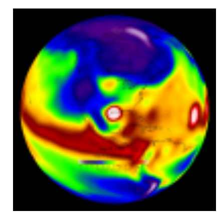
XGC1 Timing on 1024 Cori KNL nodes in quadrant flat mode.

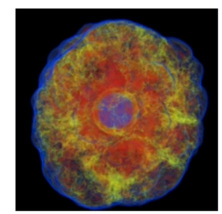
Toypush Mini-App

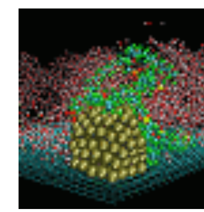










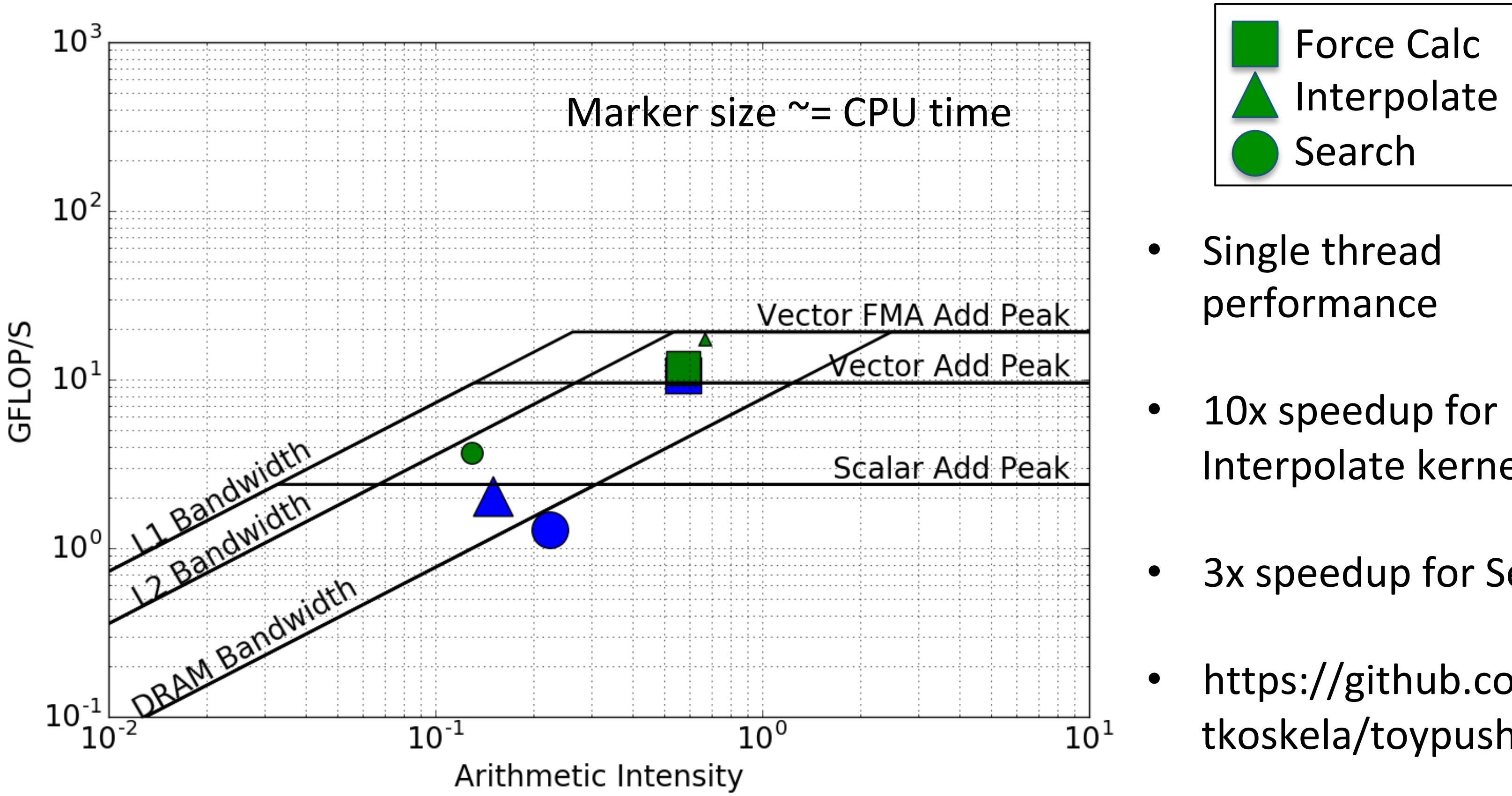


Toypush: Introduction/Motivation

- **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

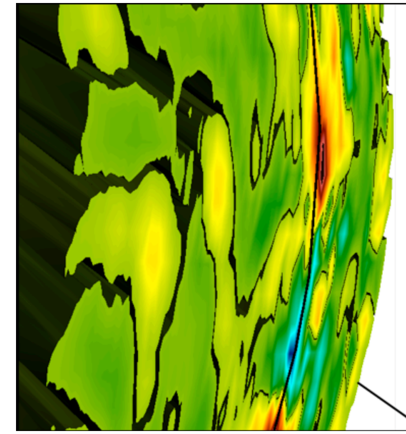


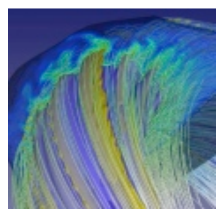
- Single thread performance
- 10x speedup for Interpolate kernel
- 3x speedup for Search
- https://github.com/tkoskela/toypush

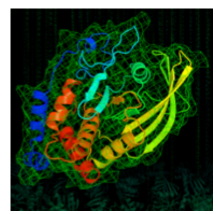
Toypush Conclusions

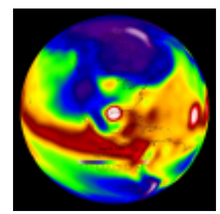
- **We optimized a mini-app to attain peak on-node performance in the electron push algorithm on KNL.**
  - Main bottlenecks are search and interpolation
  - We were successful in vectorizing and pushing them close to maximum attainable performance based on the roofline model
- **Porting optimizations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

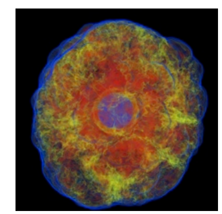
Charge Deposition

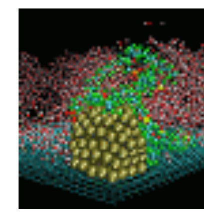






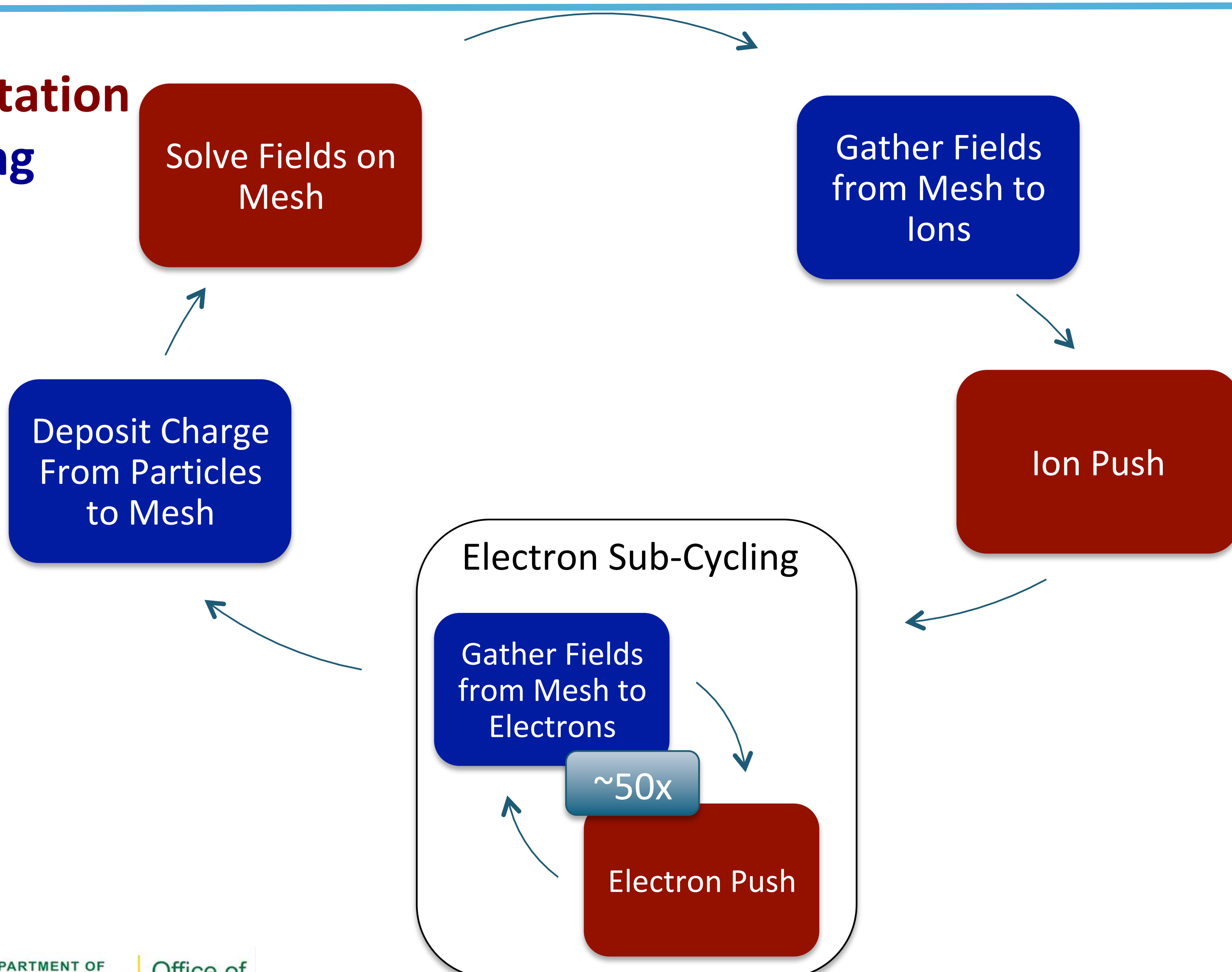






In XGC1 Electron Time Scale is Separated From the Ion Push in a Sub-Cycling Loop

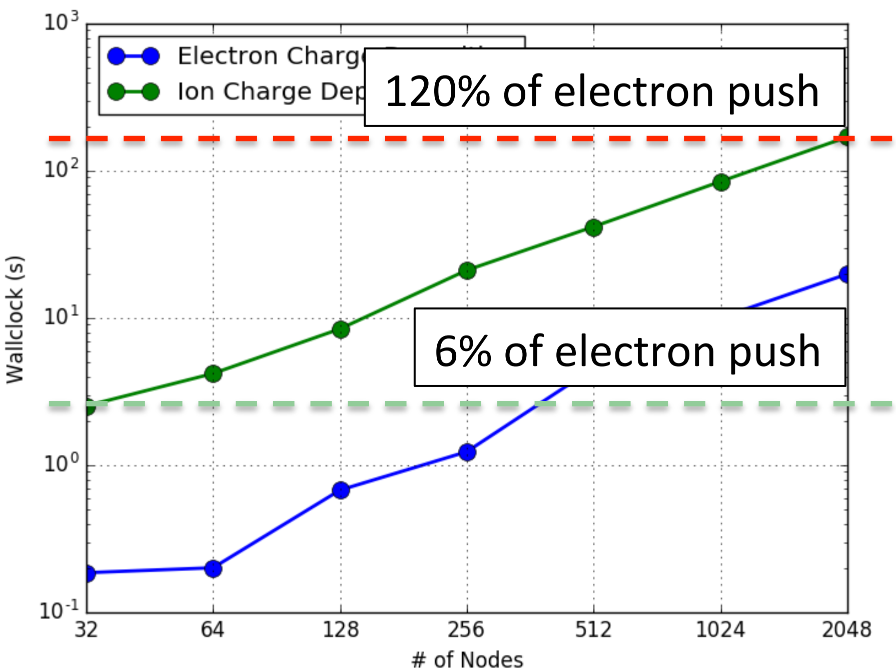
Computation Mapping



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

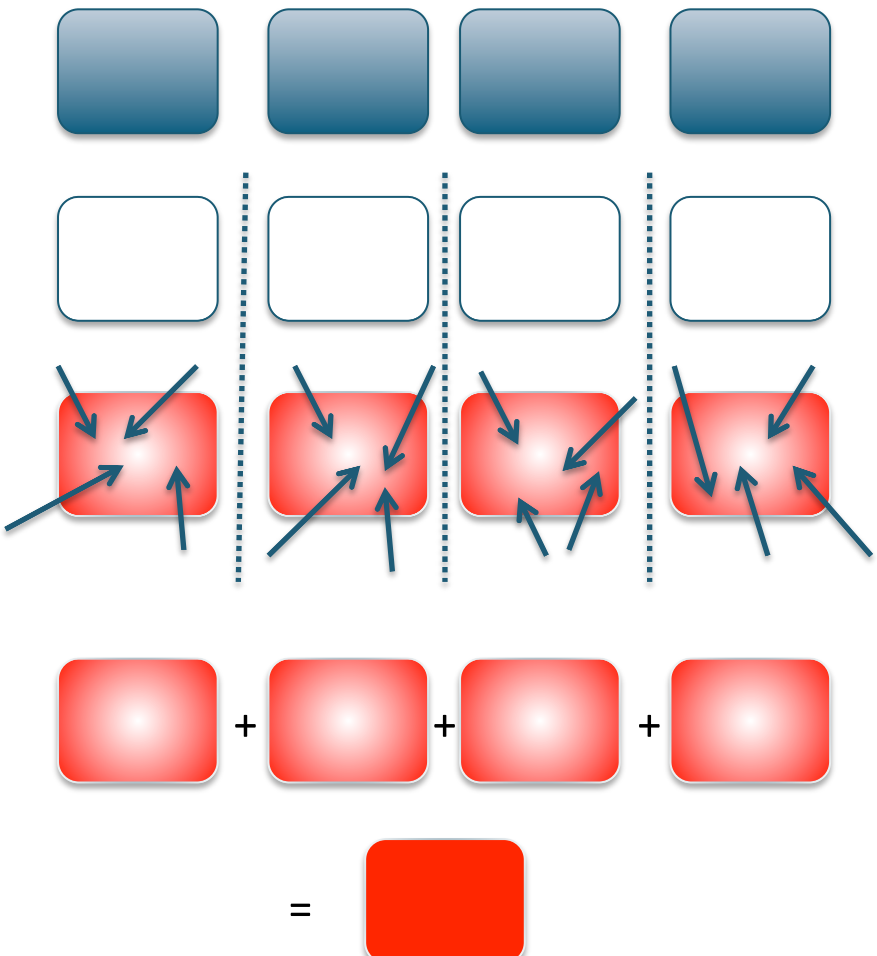
Initial State: Poor Weak Scaling of Charge Deposition



Compute Nodes	Total Grid Nodes	Total Particles
32	7 500	200 M
64	15 000	400 M
128	30 000	800 M
256	60 000	1.6 Bn
512	120 000	3.2 Bn
1024	240 000	6.4 Bn
2048	480 000	12.8 Bn

- 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

Original Charge Deposition



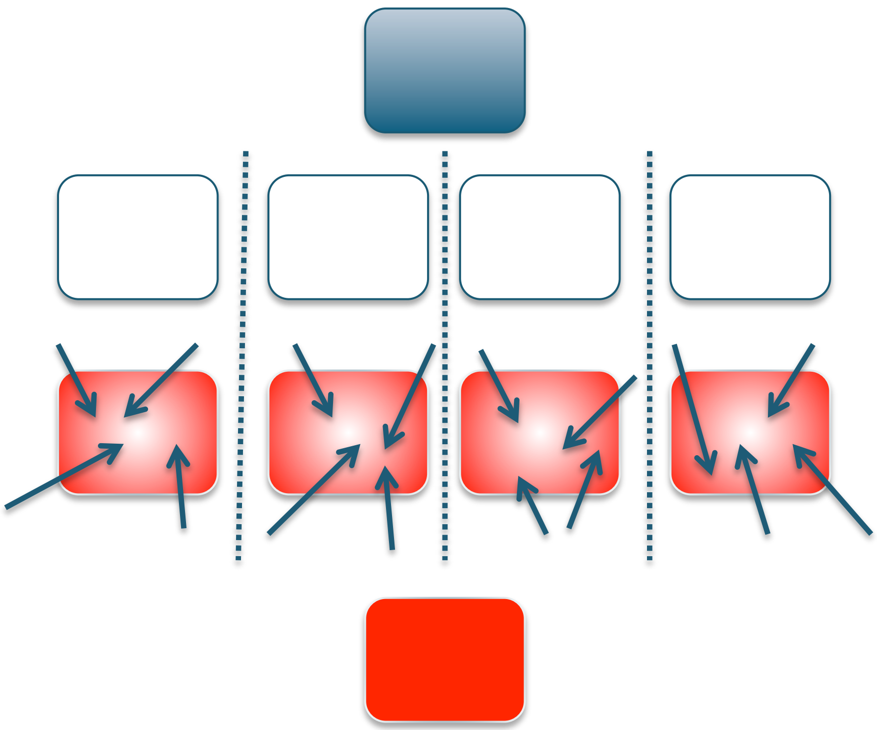
Allocate private arrays for each thread

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



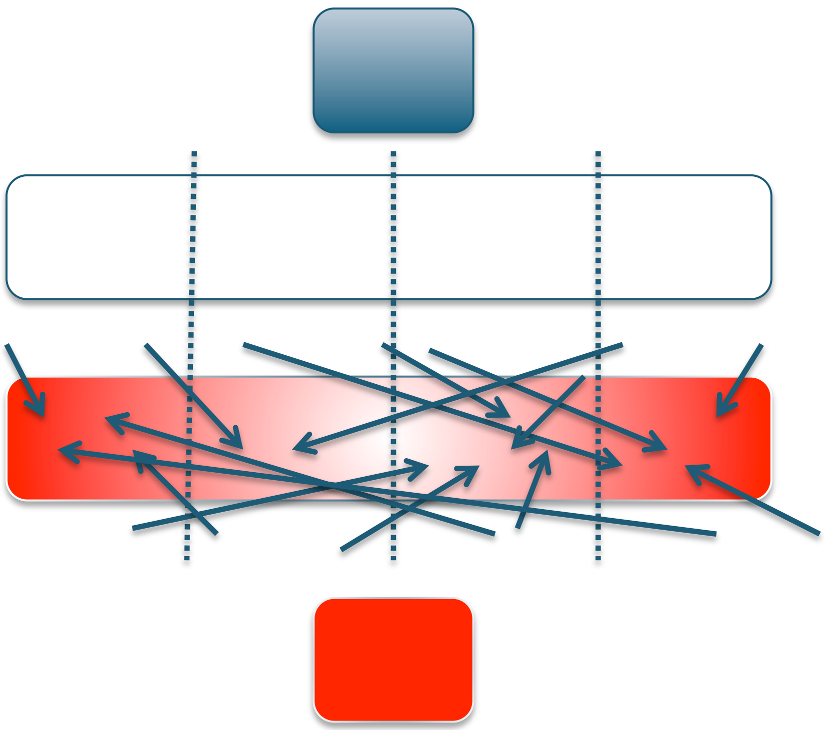
Allocate single array
→ 64x smaller memory footprint

!$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



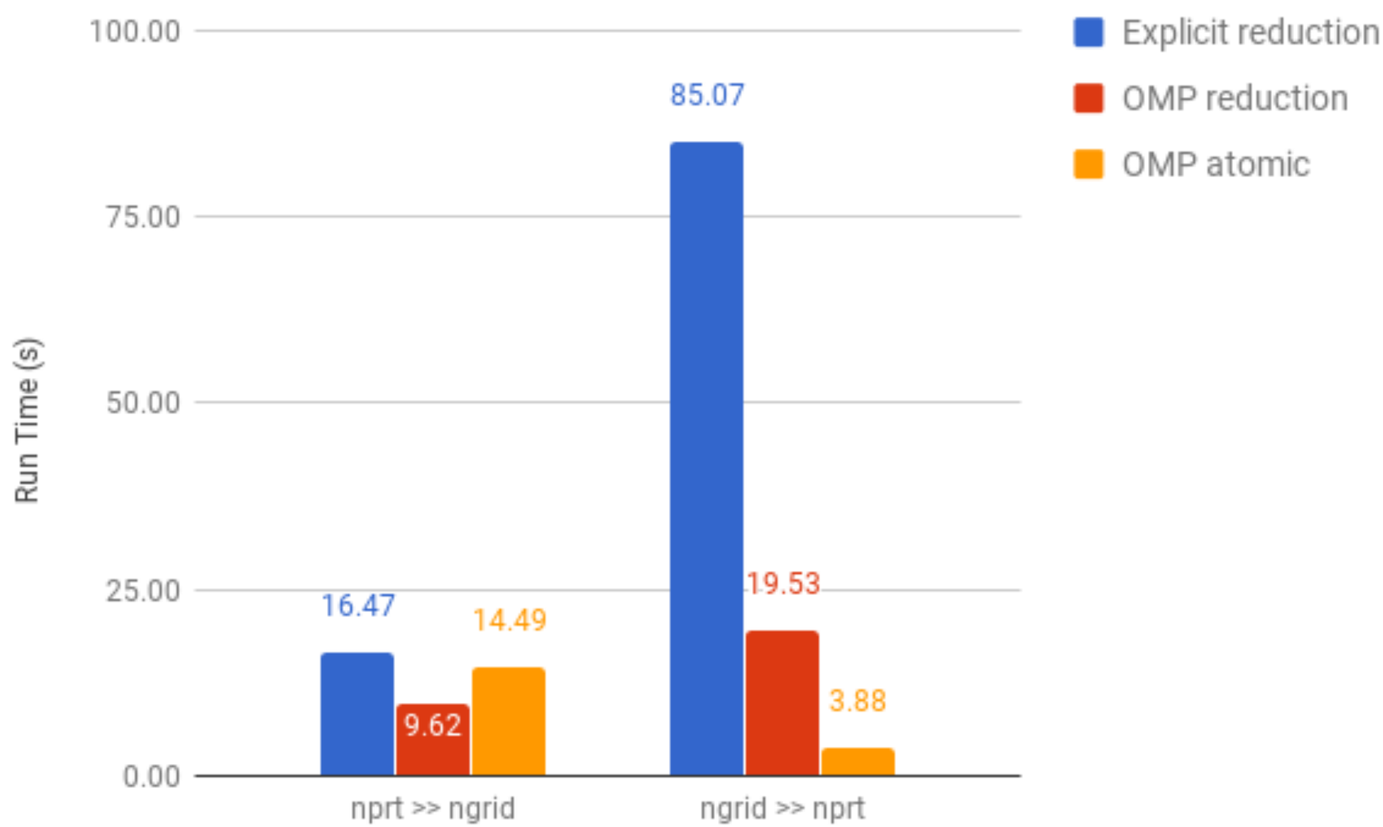
Allocate single array
→ 64x smaller memory footprint

Initialize single array to 0
→ 64x faster with threads

Deposit particles atomically
→ Avoid data races

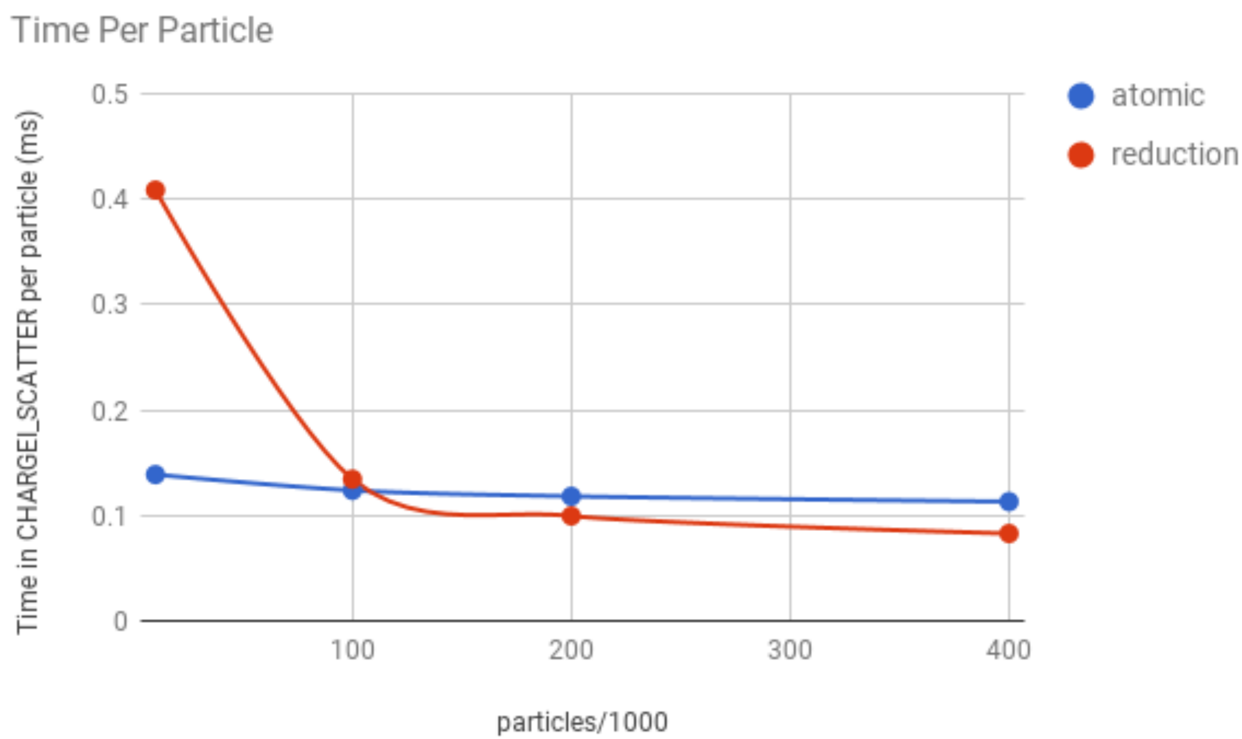
No need for reduction

KNL Performance Results



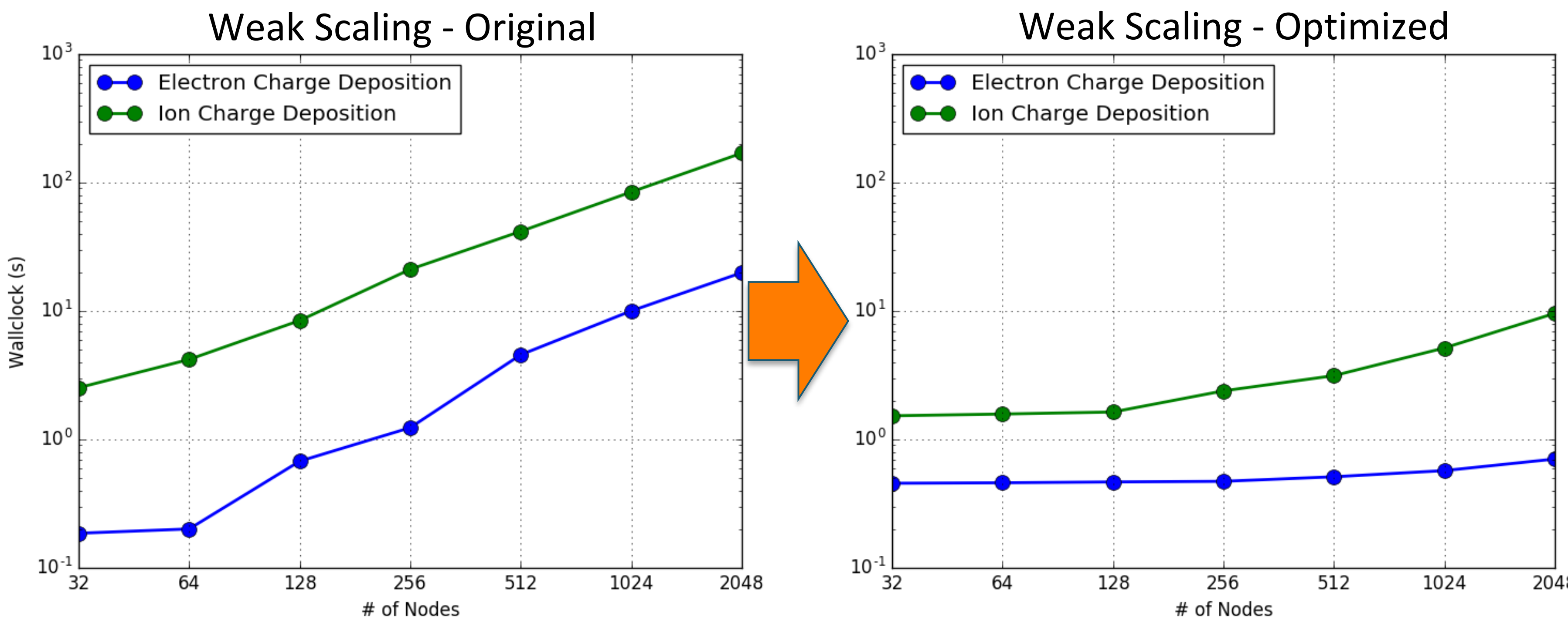
"Electrons" "Ions"

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

Weak Scaling of Charge Deposition with Atomic Updates



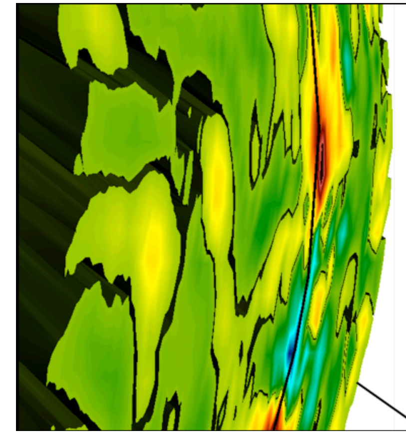
- Ideal scaling of electron charge deposition
- Some performance degradation in ion charge deposition, but > 10x faster than before at 2048 nodes.
    - "Fast enough" to be insignificant compared to particle push

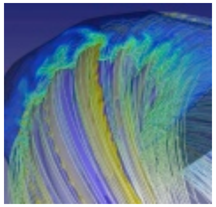
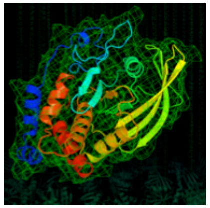
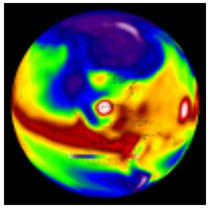
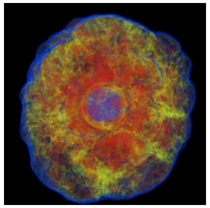
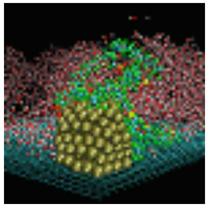
Summary And Conclusions

- **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")

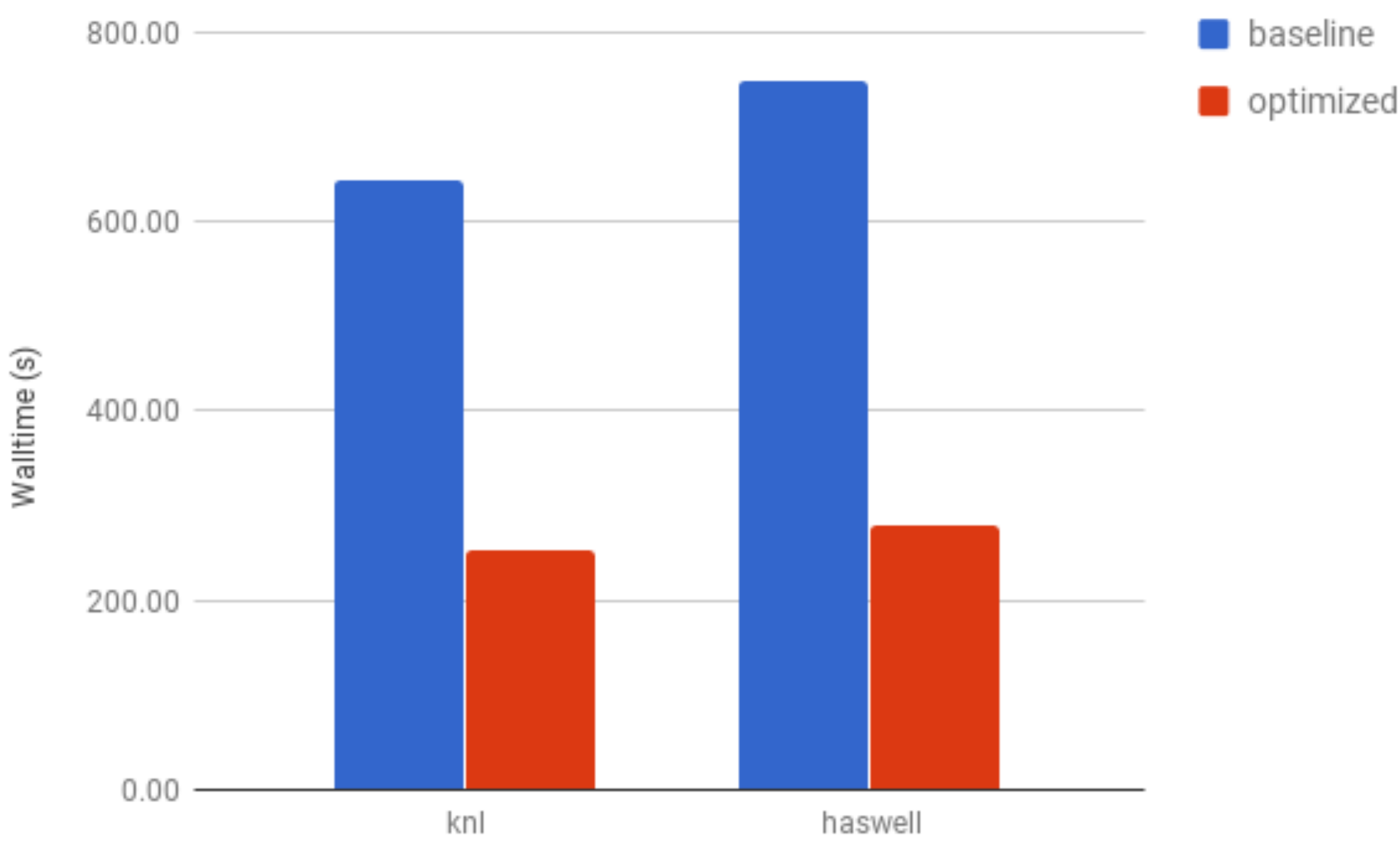
Thank you!

Performance Comparison

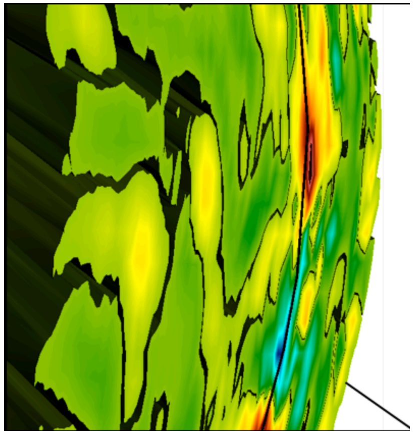


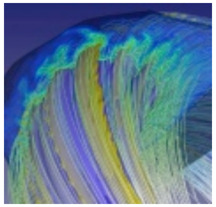






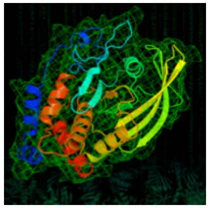
Performance Comparison

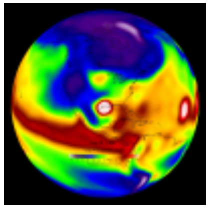


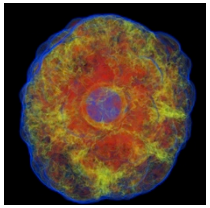
Scaling Studies

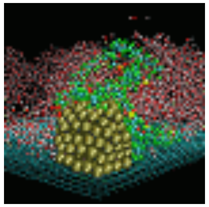










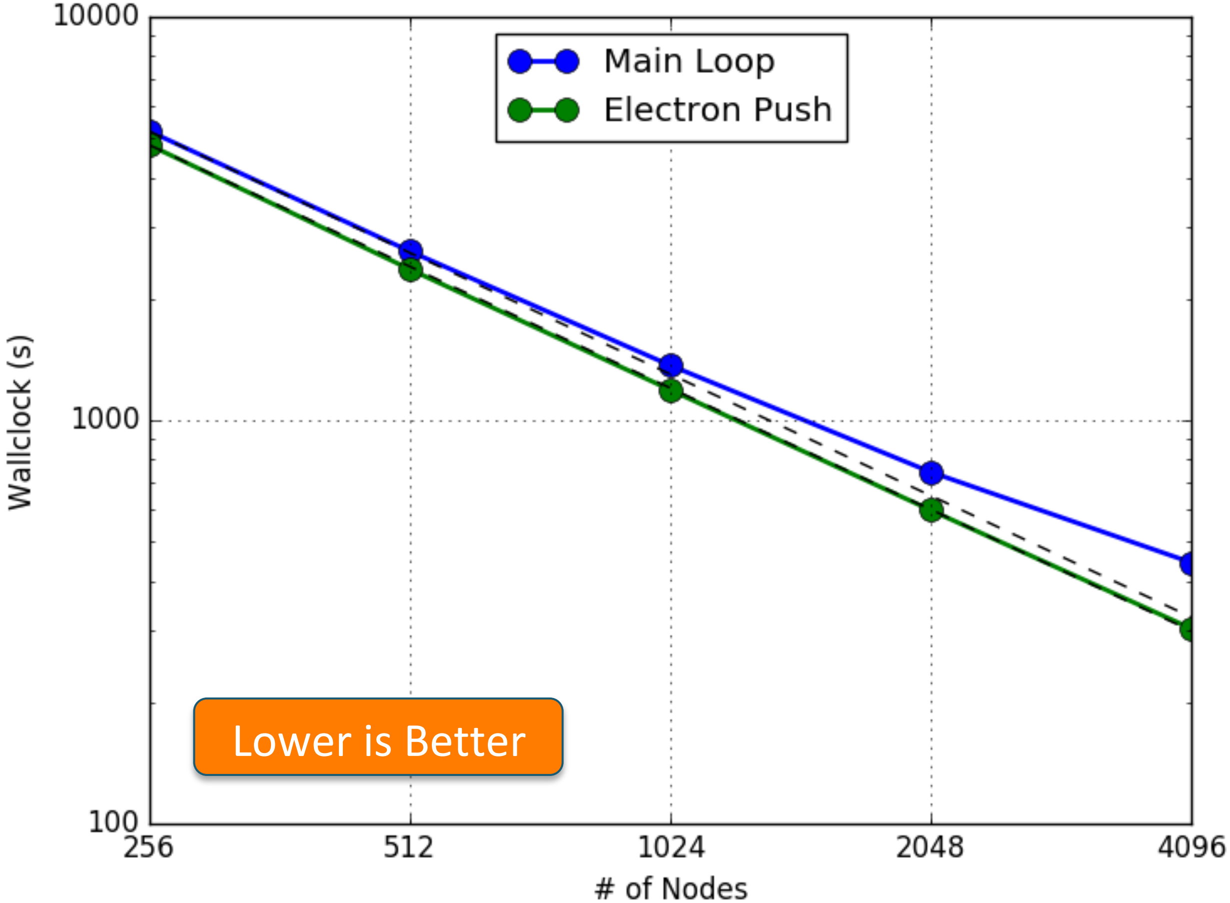


Strong Scaling Parameters

Compute Nodes	Grid Nodes Per Rank	Particles Per Rank
256	448	12.2 M
512	224	6.1 M
1024	112	3.1 M
2048	56	1.5 M
4096	28	0.75 M

- 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

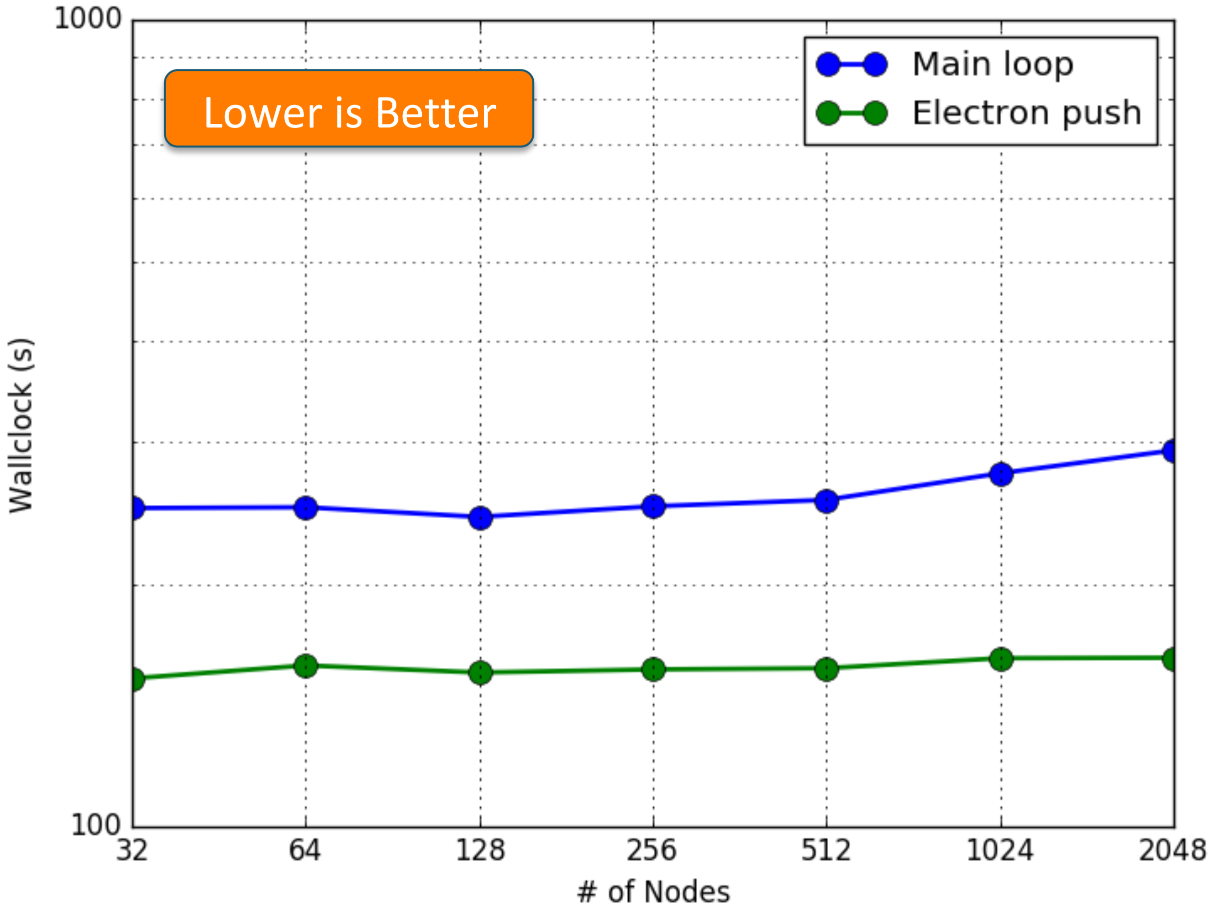
30% scaling deficit in main loop at 4096 nodes (half machine size)

Particle Weak Scaling Parameters

Compute Nodes	Grid Nodes Per Rank	Particles Per Rank
32	3584	0.4 M
64	1792	0.4 M
128	896	0.4 M
256	448	0.4 M
512	224	0.4 M
1024	112	0.4 M
2048	56	0.4 M

- 16 MPI ranks per Node, 16 OpenMP Threads per rank
- 57 000 total grid nodes per plane, 32 planes
- Quadrant Cache mode

XGC1 "Weak Scaling" Up to 2048 KNL Nodes



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%

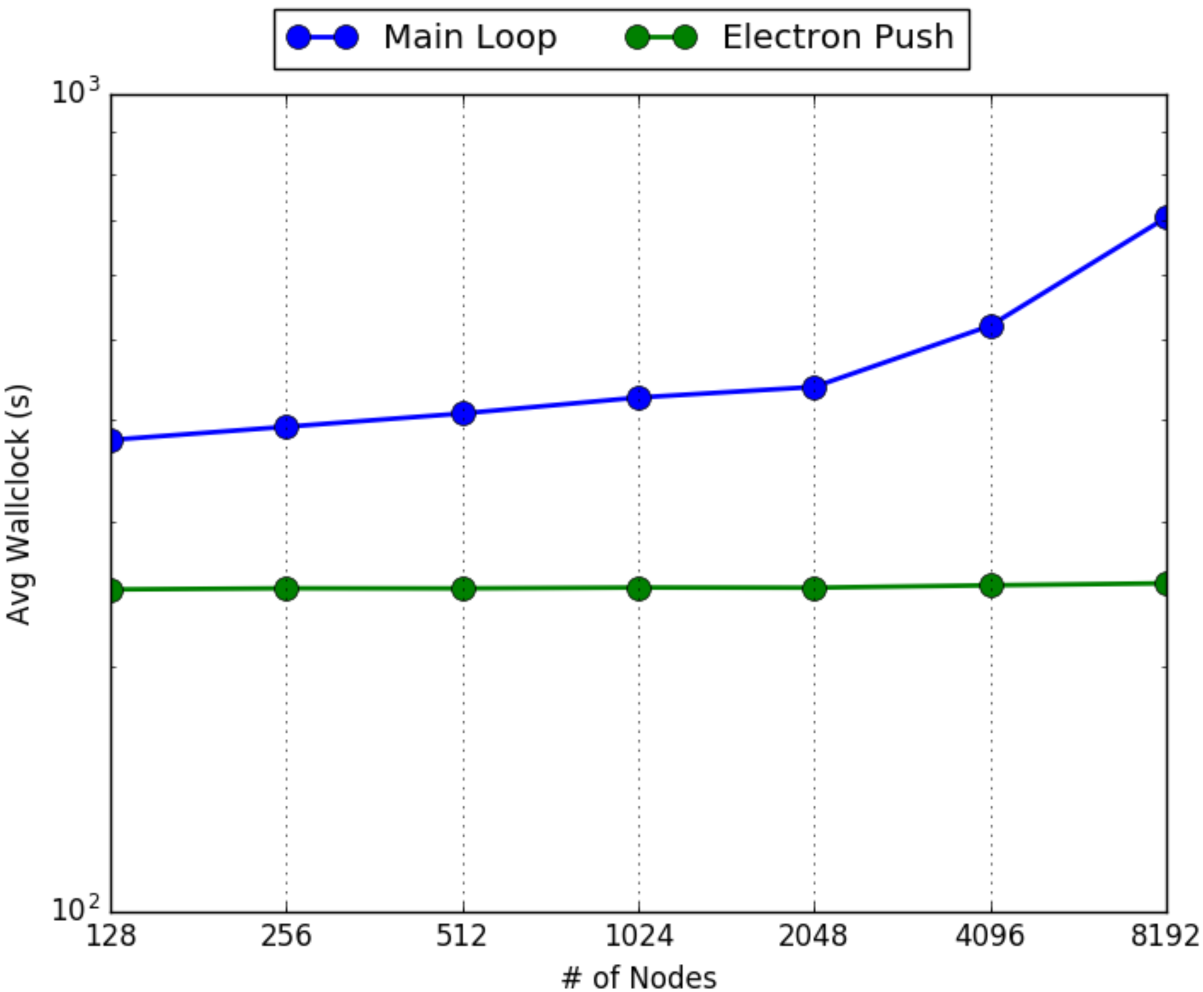
~50% slowdown at full machine size (9600 nodes) by extrapolation

Weak Scaling Parameters

Compute Nodes	Grid Nodes Per Rank	Total Grid Nodes	Particles Per Rank	Total Particles
128	117	3 750	1.75 M	900 M
256	117	7 500	1.75 M	1.8 Bn
512	117	15 000	1.75 M	3.6 Bn
1024	117	30 000	1.75 M	7.2 Bn
2048	117	60 000	1.75 M	14.4 Bn
4096	117	120 000	1.75 M	28.8 Bn
8192	117	240 000	1.75 M	57.6 Bn

- 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

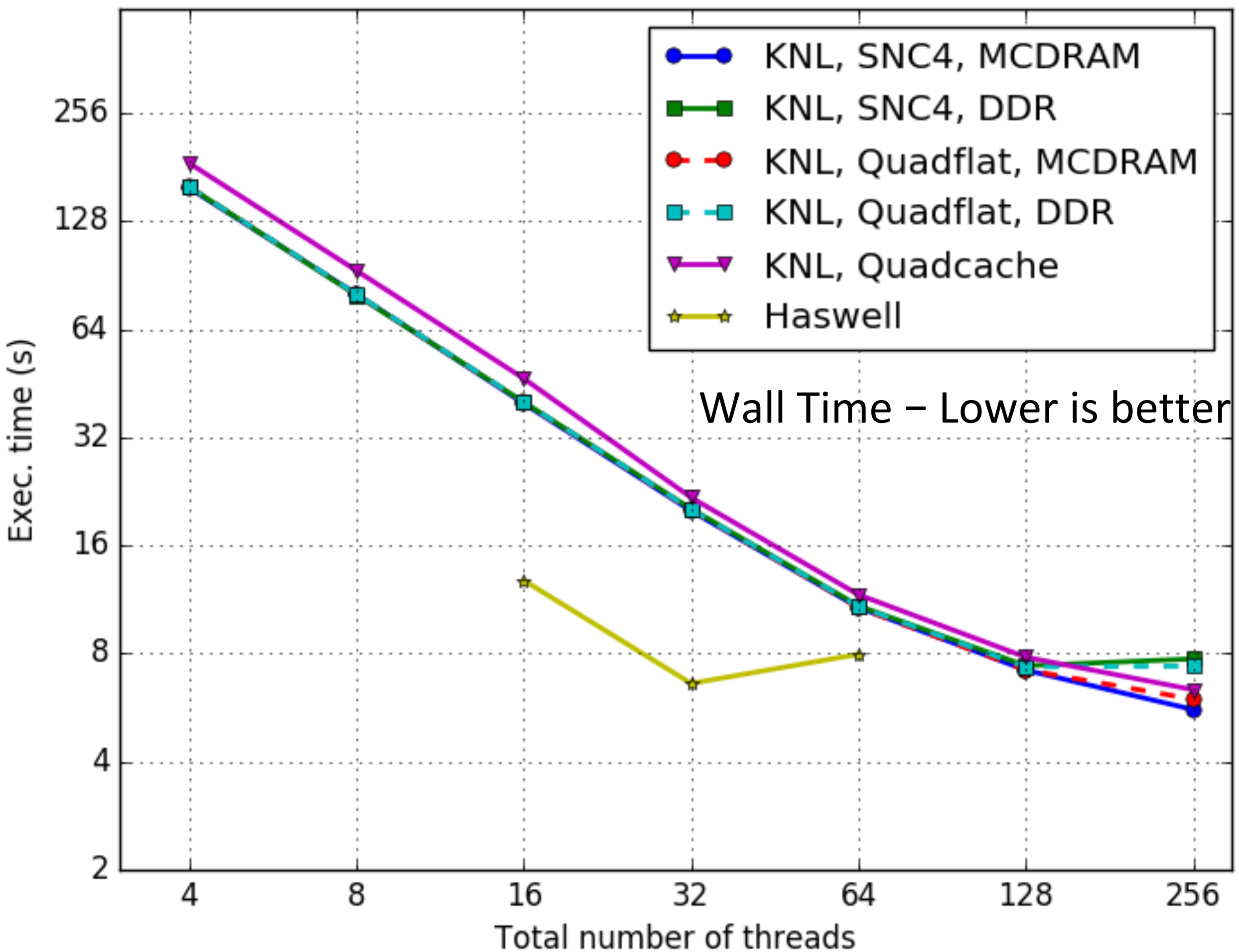
Slowdown from 128 to 2048 nodes: 16%

~90% slowdown at 8192 nodes.

Poor Weak Scaling at large scale caused by load imbalance

Single node thread scaling of electron push kernel

1 Node, 4 MPI ranks per node



Performance gain from MCDRAM only when using more than 2 threads/core → 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=1T (N <= 64)
2T (N == 128)
4T (N == 256)
OMP_NUM_THREADS=N

Original Ion Charge Deposition Pseudo Code

Legend:
OpenMP | Loops | Instructions

```
allocate(density(nnode,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

Optimized code I: Omp reduction

Legend:
OpenMP | Loops | Instructions

```
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

Optimized code II: Omp atomic

Legend:
OpenMP | Loops | Instructions

```
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
→ 64x smaller memory footprint

Initialize single copy to 0
→ 64x faster with threads

Deposit particles atomically
→ Avoid data races