Gyrokinetic Toroidal Code (GTC) on nVidia GPUs and MultiCore Clusters

Khaled Ibrahim

Future Technologies Group, LBNL

with:

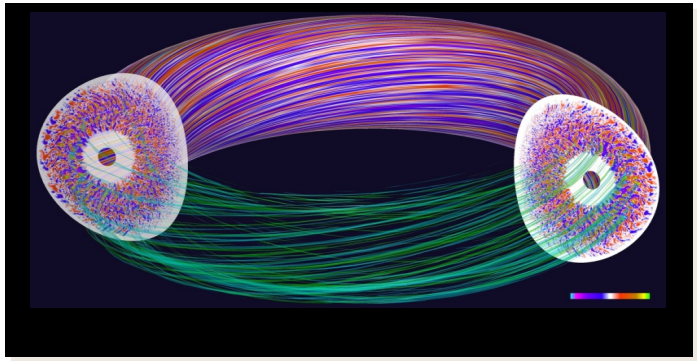
Kamesh Madduri , Samuel Williams, Bei Wang, Stephane Ethier , John Shalf , Leonid Oliker

Layout

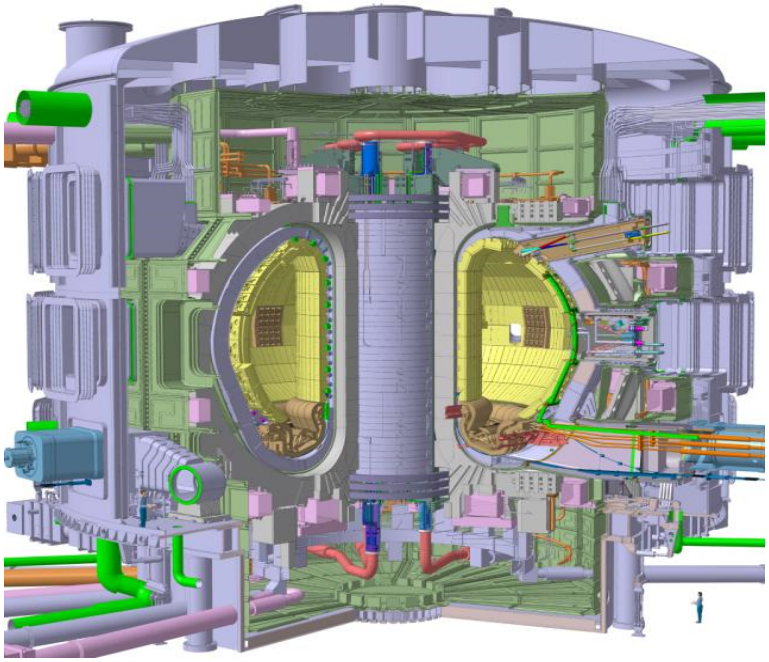
- Brief overview of GTC using Particle in Cell Method
- Optimization strategies on multicore clusters vs. GPU Clusters
- Performance analysis of GTC on GPUarchitecture.
  - Locality and Fermi Architecture
  - New performance results on Kepler architecture and comparison with Fermi.
- Comparison with CPU based architectures.

Gyrokinetic Toroidal Code (GTC)

- GTC (Gyrokinetic Toroidal Code) is numerical tool for studying microturbulence in burning plasma.
- GTC uses particle-in-cell (PIC) method for modeling plasma within magnetic confinement.
- ITER: International collaboration to build tokamak nuclear fusion reactor. GTC is one of the numerical tools to predict efficiency of plasma burning in ITER.

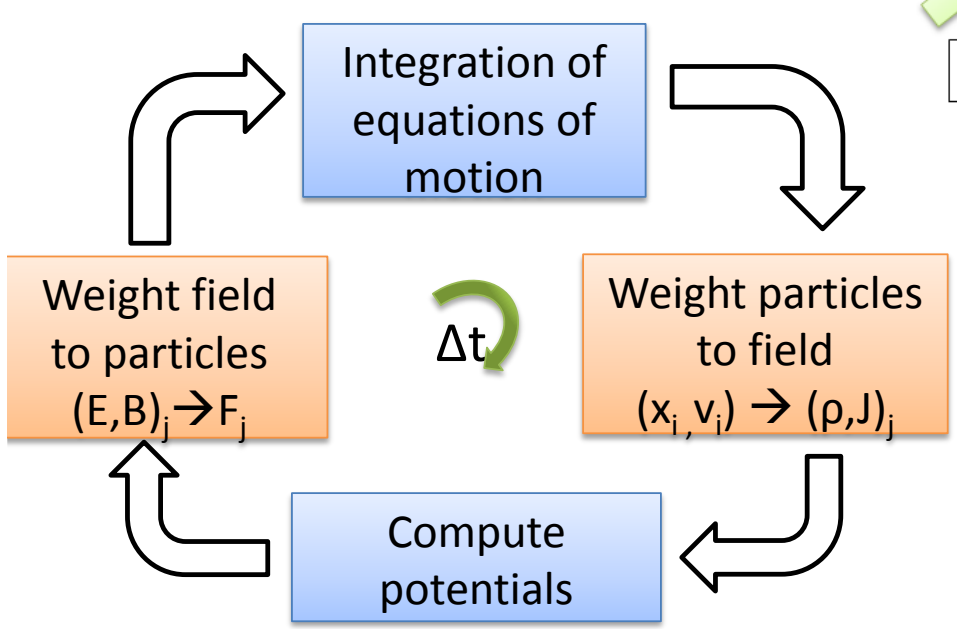


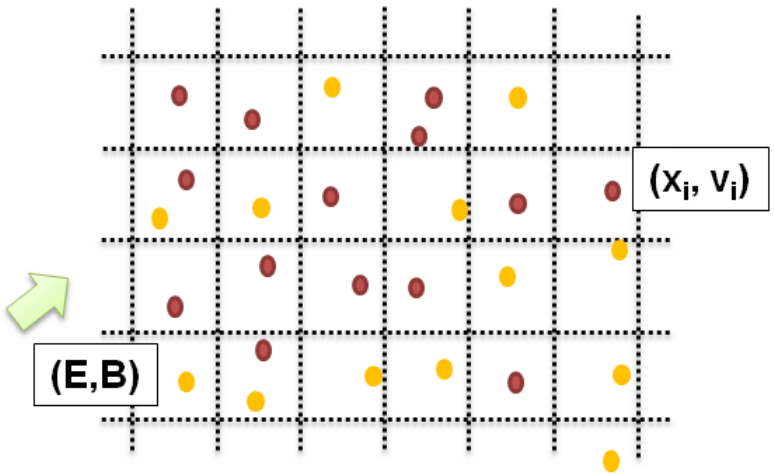
Donut-shaped ("tokamak") reactor



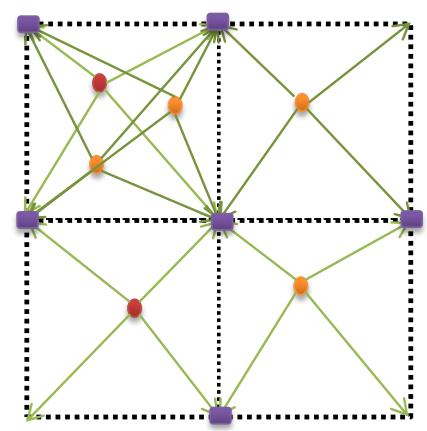
Particle in Cell Method

Instead of solving the PDE for each particle, we consider the aggregate effect on fewer cells.



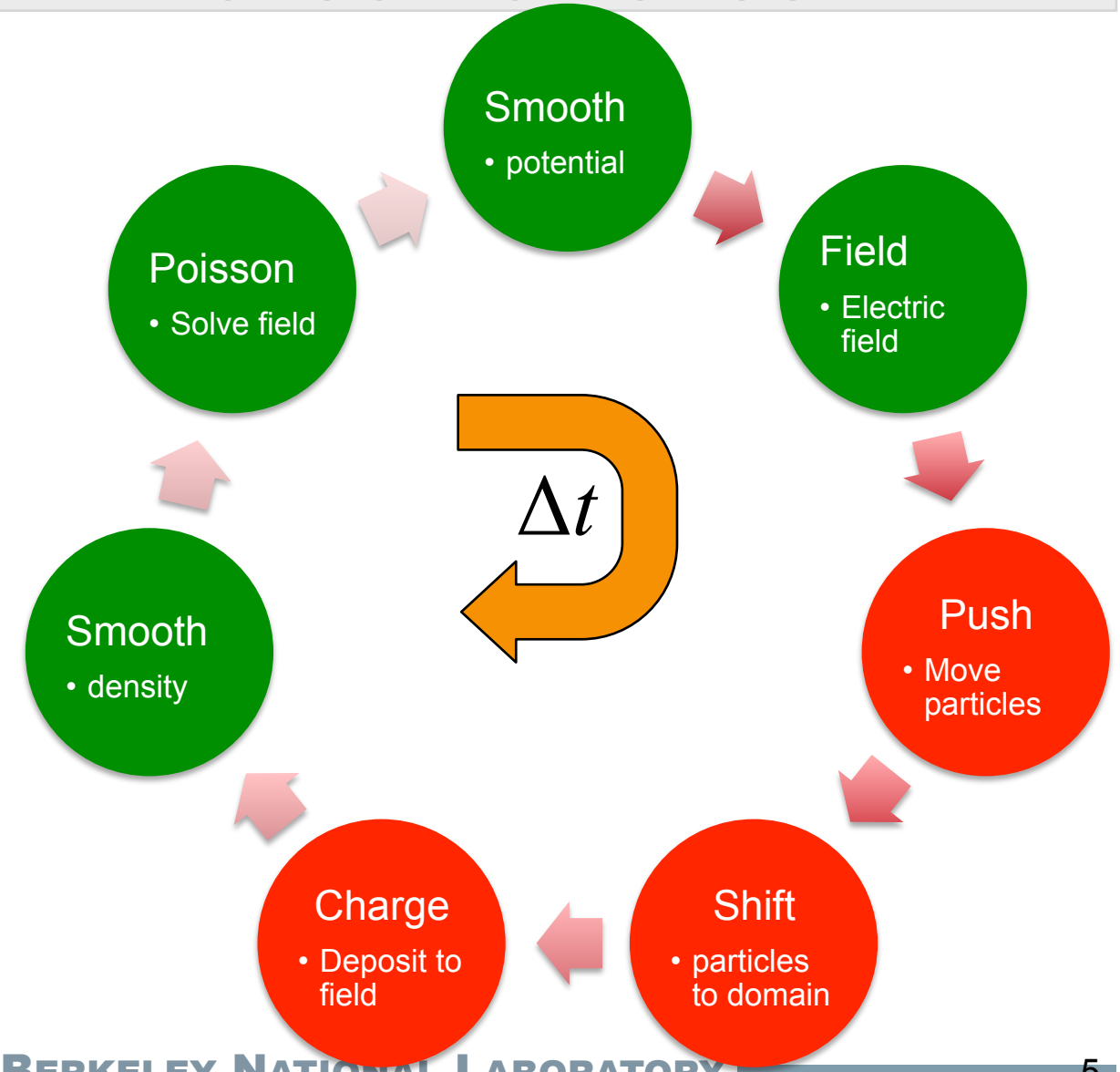


Particle-grid interpolation



GTC Computation - PIC based

- **Work on electric field or charge density**
  - Perfectly parallel
  - Proportional to the grid
- Work on particles and grid
  - Push: gather pattern.
  - Charge: scatter pattern.

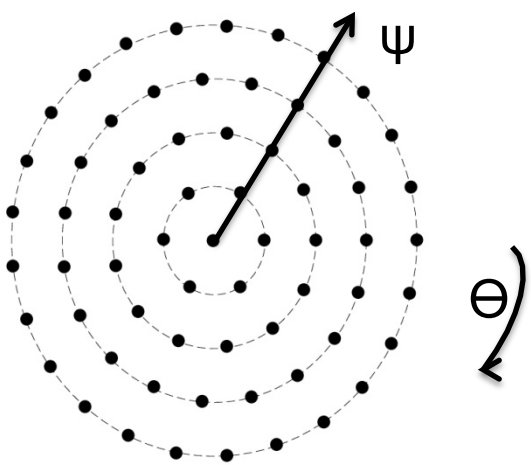


Parallel GTC

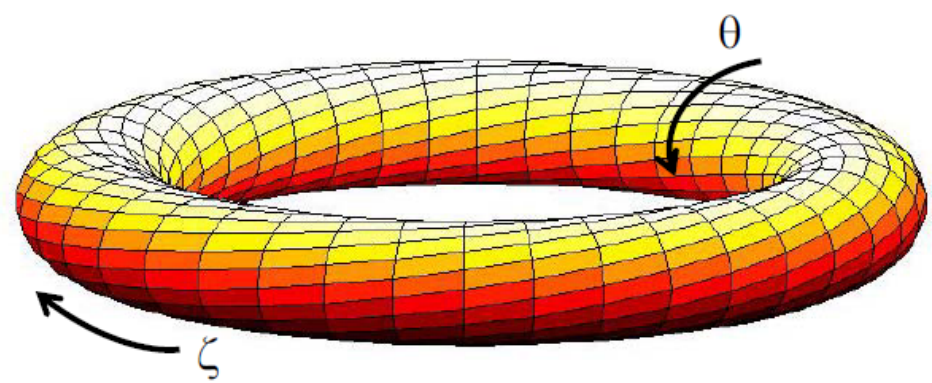
- Two levels of parallelization in the GTC MPI Implementation.
  - 1D domain decomposition along the torus.
  - Particle decomposition in each toroidal domain (require plan replication)

Cross-section: poloidal plane

Uniform in ψ-θ

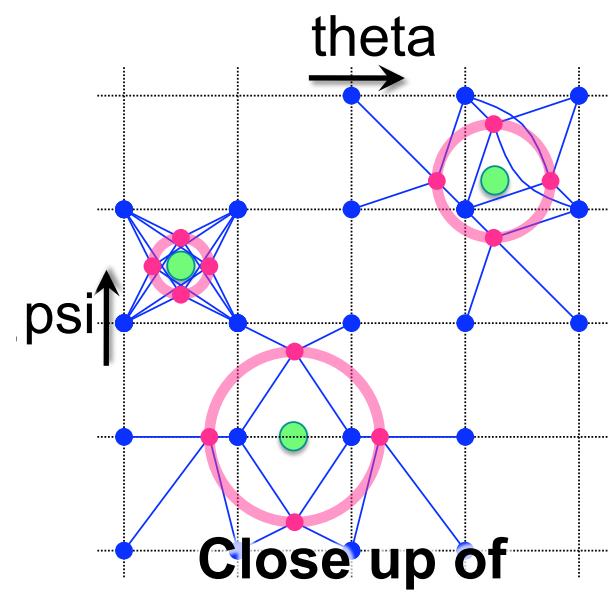


3D toroidal mesh



Data Structures & Access Pattern

- Particles Data
  - Position and speed of particles
  - Large percentage of dataset
- Grid data
  - Electric field: external and charge
  - Charge Density: particles related
  - Small percentage of the data set
- Particle to cell ratio
  - Typically 5-100 for problems of interest
- Access Pattern
  - Grid alone or particles alone
    - Streaming access
  - Charge grid and particle, simultaneously (charge, push)
    - Make one regular (Particles-larger)
    - The other irregular (Grid-smaller)
      - Scatter/Gather
      - 1 particle affects up to 8 cells (32 doubles)



Close up of Poloidal Plane

Application Setup

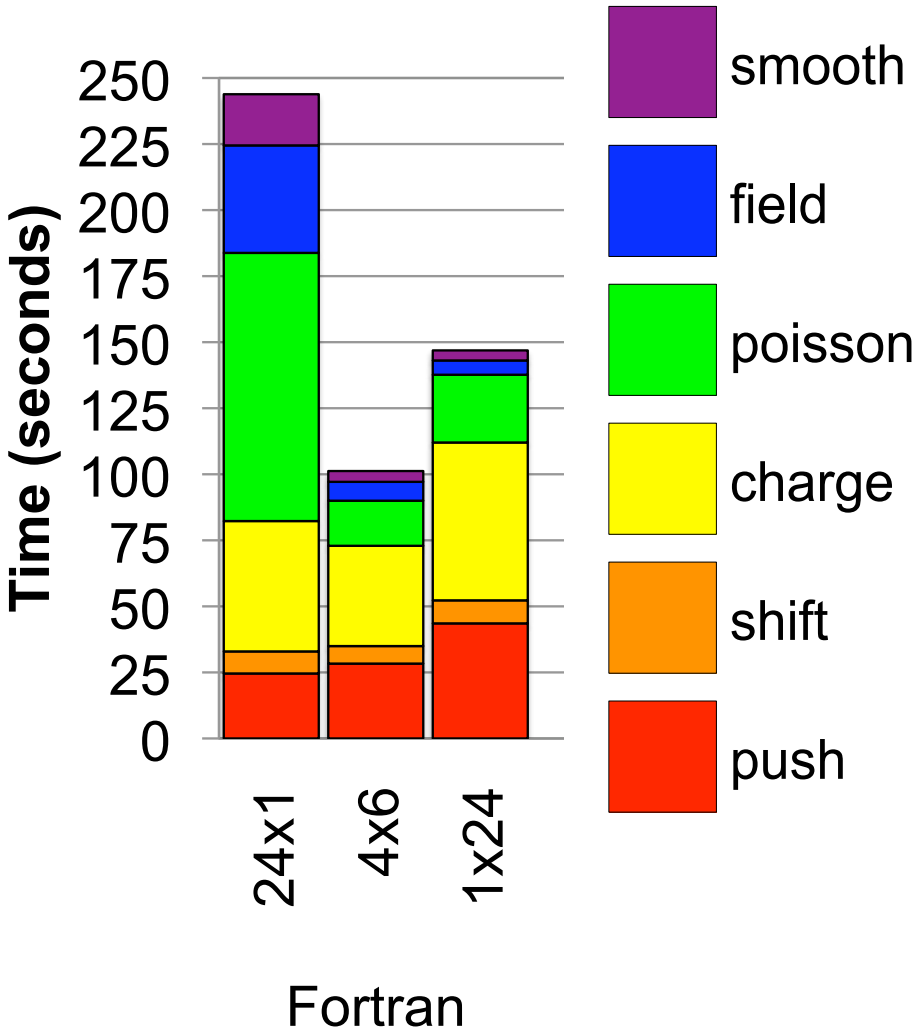
- We assume a **radially-binned** particle distribution, experiment with several different **grid sizes** and particles-per-cell configurations.

Problem Instance	A	B	C
# ζ	1-16	1-16	1-16
# ψ	90	192	384
# θmax	640	1408	2816
# poloidal grid points	32449	151161	602695
# particles (micell = 5)	0.16M	0.76M	3M
# particles (micell = 100)	3M	12M	60M

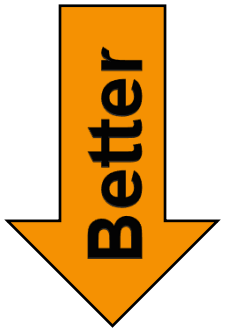
Studied Clusters

	AMD Opteron	Intel SNBe	Fermi	K20x	IBM PPC450d	IBM A2
Clock (GHz)	2.1	2.6	1.33	0.73	0.85	1.6
Gflops	201.6	166.4	665	1310	13.6	204.8
$/core (KB)	64+512	32+256	16-48	16-48	32	16
Last $/chip	6 MB	20MB	768KB	1500KB	8MB	32MB
Cores x chips	6x4	8x2	32x16	192x14	4x1	16x1
Memory (GB)	32	32	6	6	2	16
System	Hopper	Edison	Fermi	Titan	Intrepid	Vesta

Execution Time Decomposition on Multicore (base)



- **B20: average 20 particles per grid cell**
- **Hopper 16 nodes - 24 cores per node**
- **Configuration (processes x threads)**



24x1: excessive grid replication
(maximum mem footprint)

1x24 excessive conflict
(minimal mem. Footprint)

Challenges

- **Conventional**
  - Data hazards in the charge deposition computation.
  - Irregular access to the cells (temporal locality challenge).
    - Complexity of mapping particle to cells.
    - Particle to cell mapping evolve with time.
  - Vectorization.
- **Multicore related**
  - Parallelization amid hazards?
    - Revisiting locality with shared memory.
  - Load balancing work across cores
  - Stress on memory.
  - NUMA awareness.
- GPU Related
  - All the above + explicit memory management.
    - CPU to GPU
    - GPU to shared memory

CPU vs. GPU Optimization

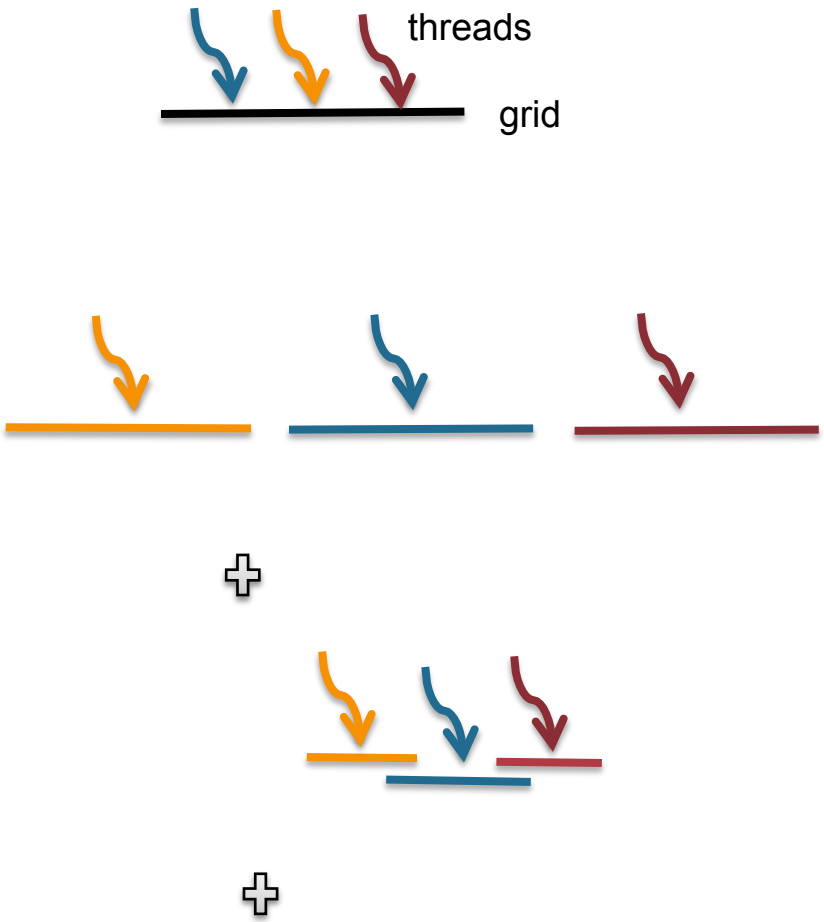
Objectives	CPU Optimizations	GPU Optimization
Conflict avoidance	Replication (per thread or socket)	Not applicable (too many threads)
Serialization (correctness)	♦ pthread locks ♦ Atomics	♦ Use of atomics ♦ Mixed precision
Improving locality (common)	Sorting particles to improve locality	
Locality (cont.)	Binning particles to threads	♦ Cooperative threading ♦ Reconfigurable L1 cache/ shared memory ♦ Memory coalescing ♦ Texture memory
Reducing stress on memory	♦ Redundant computation of auxiliary arrays ♦ Loop fusion and removal of intermediate arrays	
Numa handling	Thread pinning	Not applicable
Code vectorization	Use of intrinsics	Implicit vectorization (data layout, avoid control flow)
Load balancing	OpenMP guided scheduling	Implicit balancing by creating many thread blocks.

Handling Grid Updates on (shared memory) Multicore

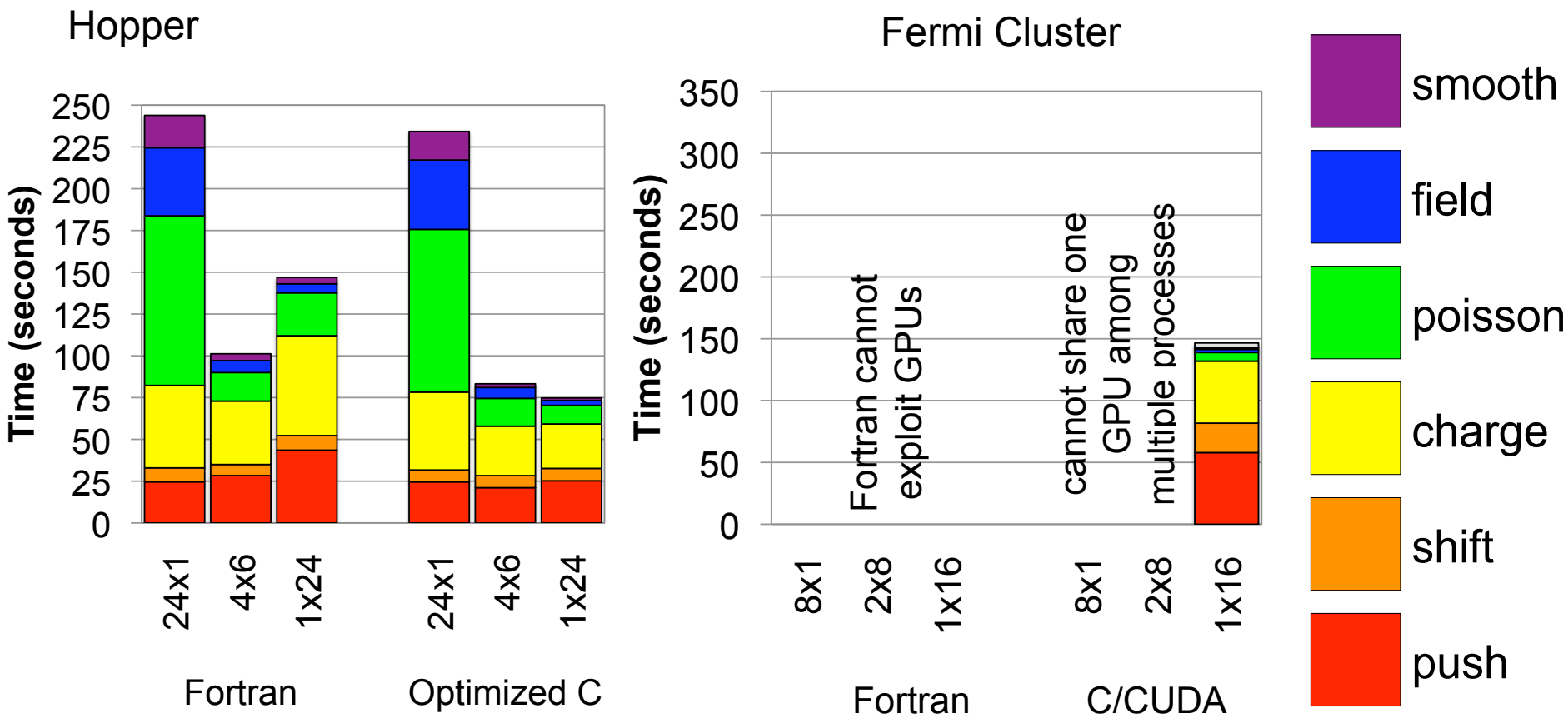
- Shared grid
  - The Greatest memory conflict.

  The most memory-efficient
- Fully-replicated
  - Less conflicts
  - Memory intensive
- Geometric partitioning of grid with ghost
  - Exploits particle locality, reduces shared grid updates

1D illustration



Performance Hopper Vs. Fermi (B20)



GTC Performance Challenge for GPUs

- GPU strength is in handling spatial locality
  - GTC performance on GPU is dependant on grid access
- Grid access
  - Irregular - relies on temporal locality
  - Involves data hazards
  - Evolves with time as particles move
- GPUs have Limited memory and many cores
  - Difficult to replicate grids
- GPUs' Atomics are executed in the L2 cache level
  - Very expensive-slow – no locality
- GPUs have limited caching
  - Difficult to capture temporal locality in the L1/L2

Grid Temporal Locality and Performance on GPUs

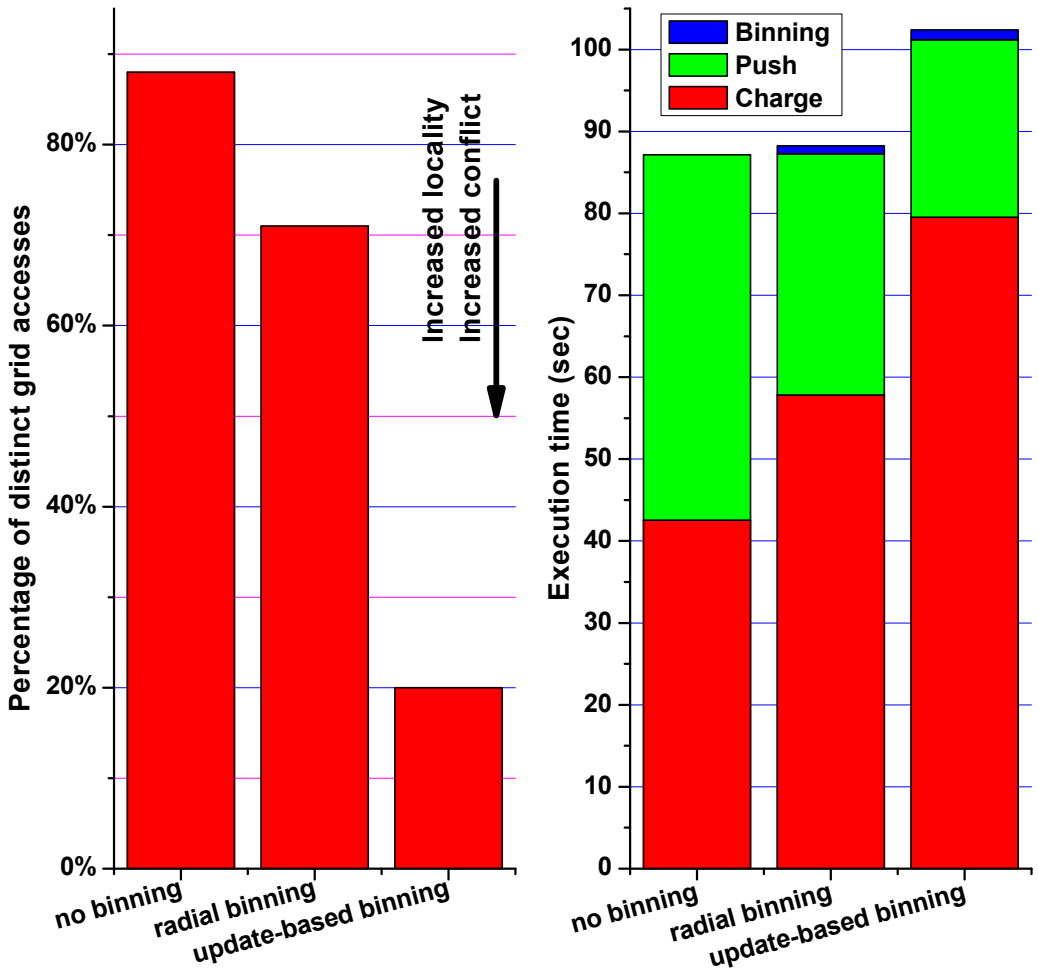
- Visiting particle is arbitrary but need to be streamed
  - Particles sorting to improve grid access locality
- Earlier results are without sorting (will know why in a minute)
- Particle Sorting techniques
  - Based on radial binning
    - Good enough for Multicore
  - Based on the position of the update cells (majority vote!).
    - Needed for GPUs – small locality window
- Can we exploit shared memory to avoid expensive global updates?

Locality Improvement Impact on Fermi

Locality

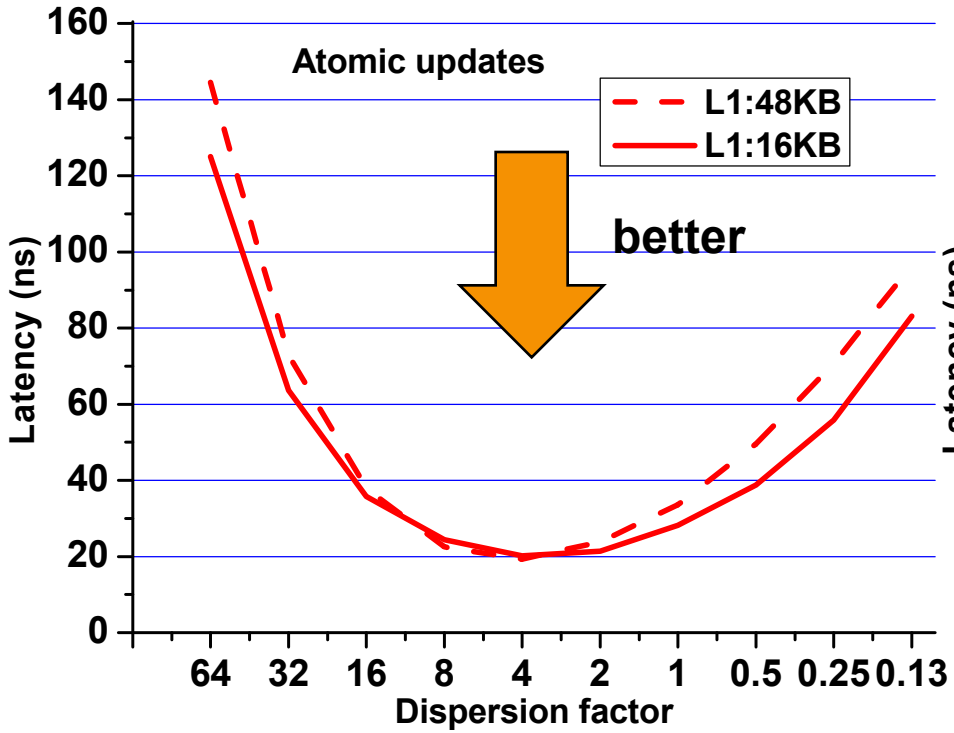
- Improvement in locality with sorting
  - (fewer distinct addresses in a 32KB window)
- Performance impact
  - Push
    - Irregular read (gather) from the electrical grid.
  - Charge
    - Irregular writes (scatter) to the charge grid.

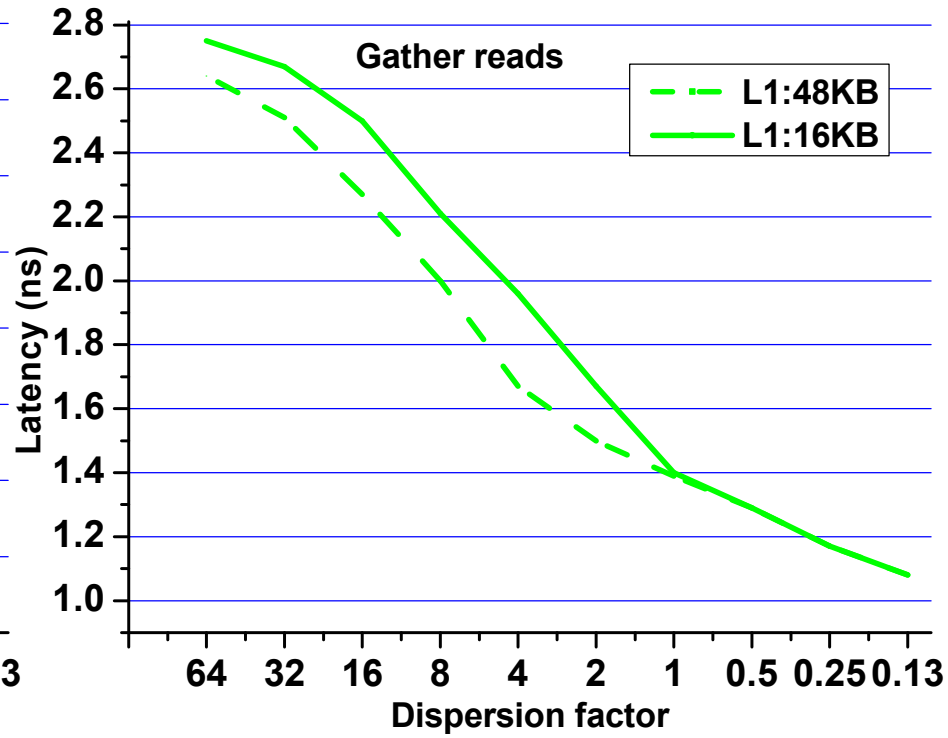
Impact of Improved Locality



Fermi Scatter Atomics & Gather Reads (Fermi M2090)

Microbenchmark to measure the latency with different access clustering.





Exploitation of Locality using Shared Memory

- Early attempts
  - Use shared memory to filter atomic update to global memory.
    - Irregular access due to four point updates
    - Cannot analyze statically (particles keep moving)
    - 2x slowdown
- Four Point Approach (Bei Wang et. al)

(Split charge & push into two stages.)

- Stage 1

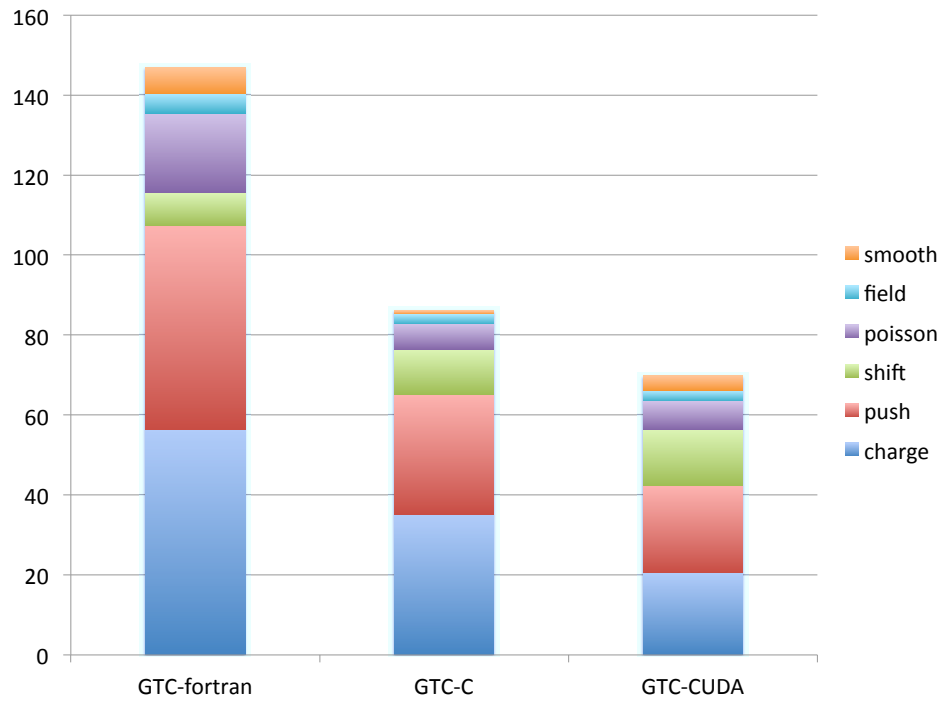
Visit particles and create auxiliary array to store updates to the charge grid into bins.

- Stage 2

Visit the auxiliary arrays, use shared memory for update reductions before sending updates to the global memory.

Four Point Approach on Fermi M2090

- Objective:
  - Avoid Atomic by using shared memory
- Technique:
  - Visit all particle and put 4 updates into cell sorted auxiliary arrays
  - Visit sorted arrays and updated shared memory
  - Commit update to the global grid.
- The first GPU version to match hopper performance.
- Measurements done on Titan-dev
  - M2090 GPUs

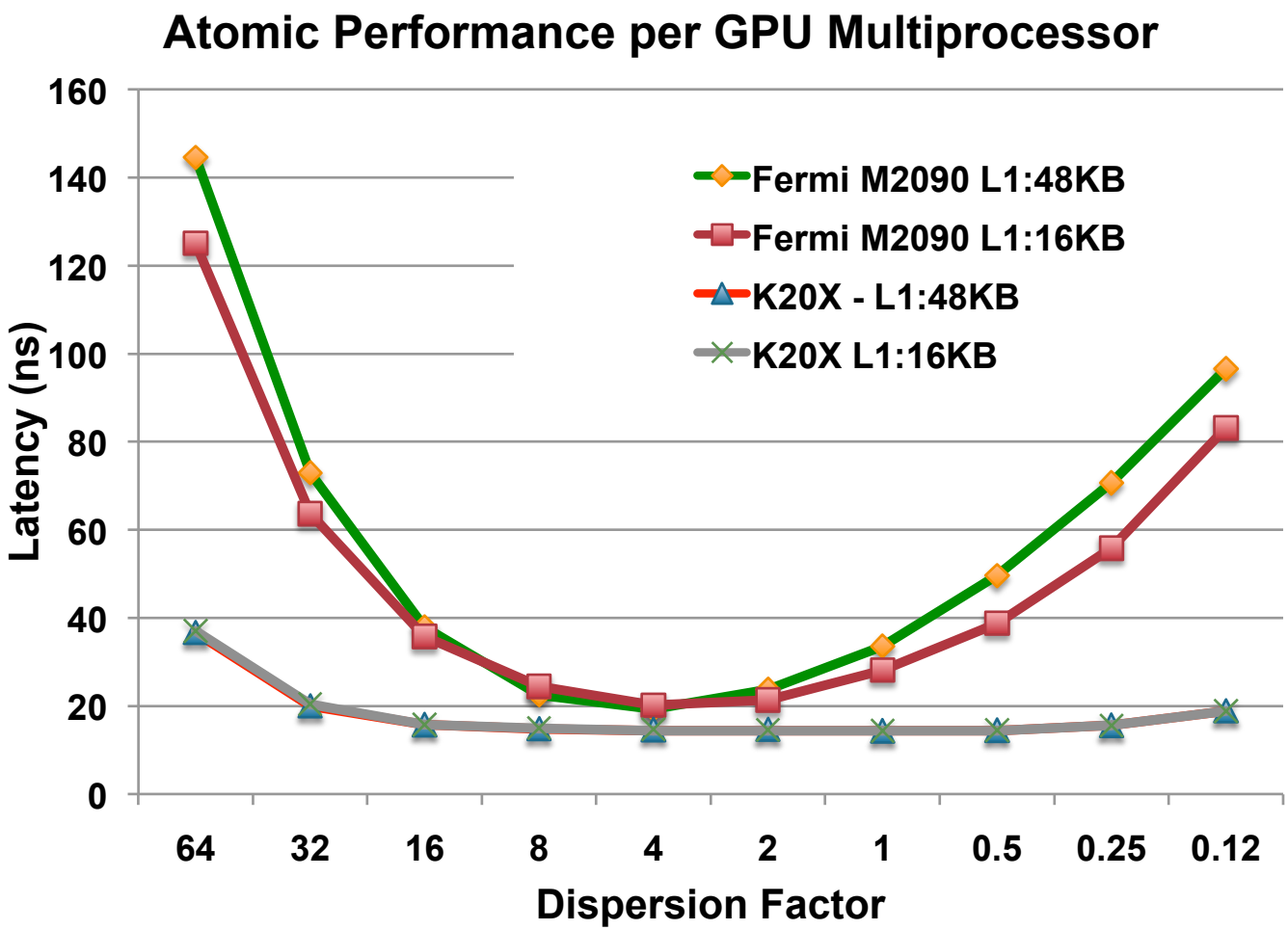


Kepler vs. Fermi

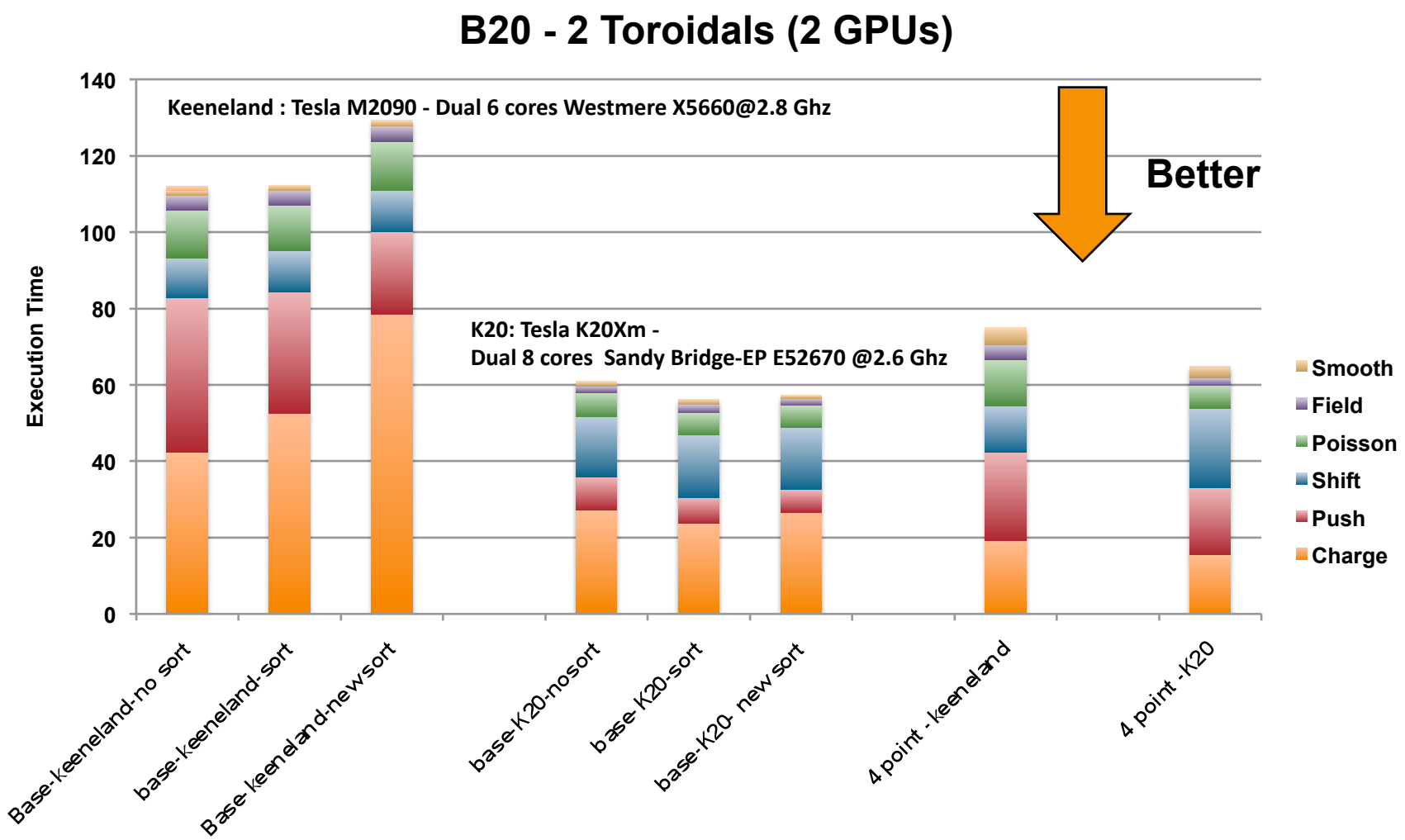
- Key Attributes
  - Larger L2 cache
  - Improved atomic Performance
  - More cores per multiprocessor
  - Higher overall flop rate
  - Lower frequency

	Fermi M2090	K20x
Clock (GHz)	1.33	0.73
Gflops	665	1310
$/core (KB)	16-48	16-48
Last $/chip	768KB	1500KB
Cores x chips	32x16	192x14
Memory (GB)	6	6
System	Titan-Dev Keeneland	Early access

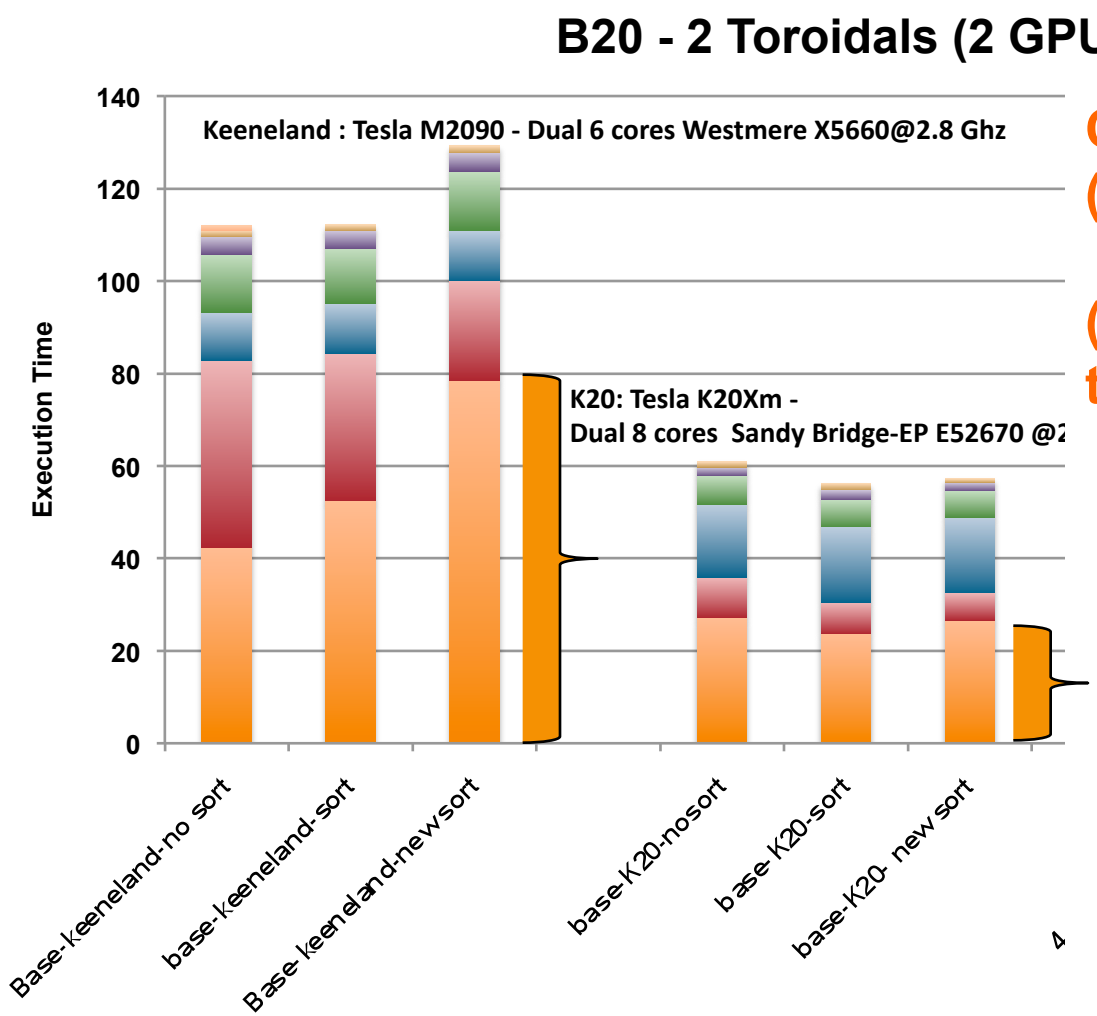
Atomic on Kepler vs. Fermi



Kepler Vs. Fermi



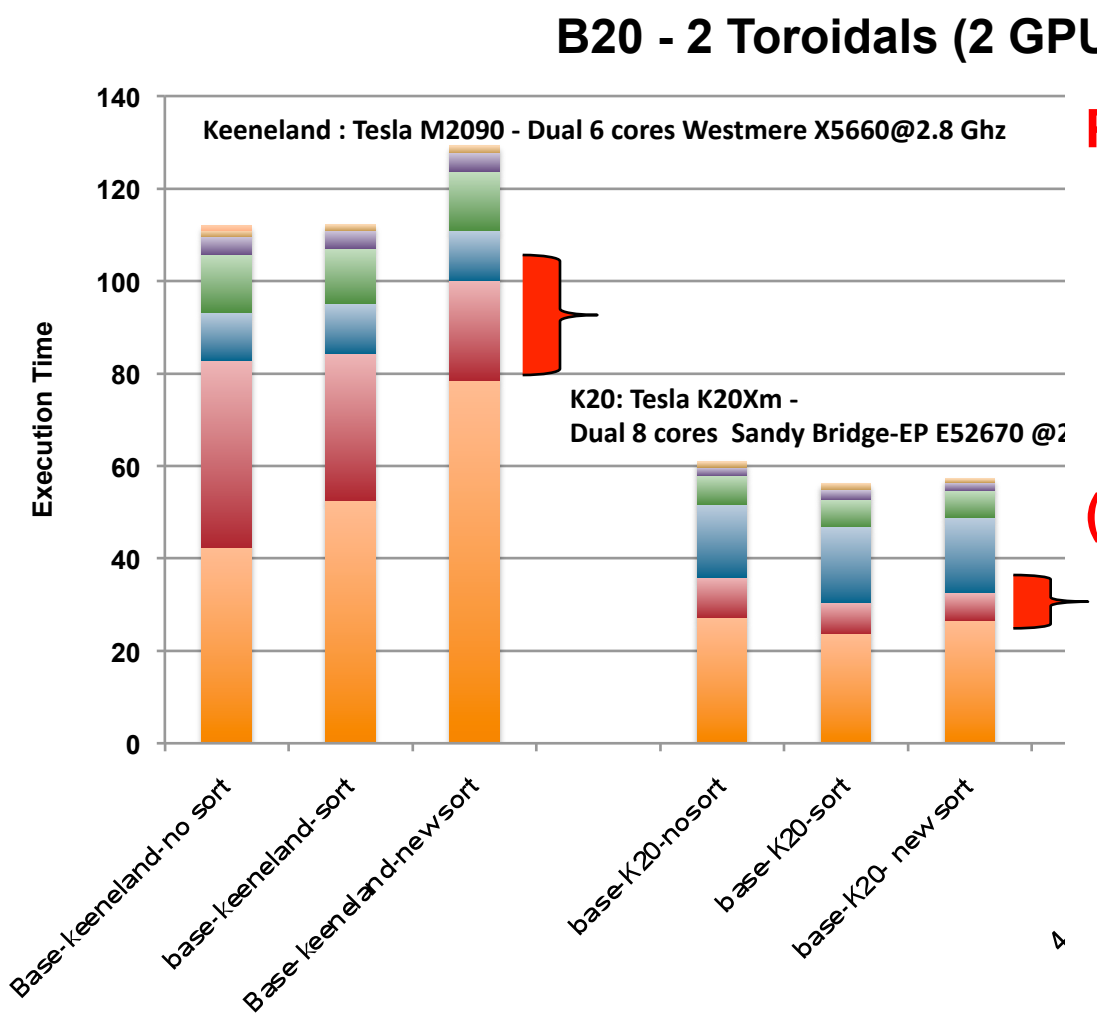
Kepler Vs. Fermi



Charge: Atomic intensive. (1.5X-3X improvement)

(more if we considered the low frequency SMX).

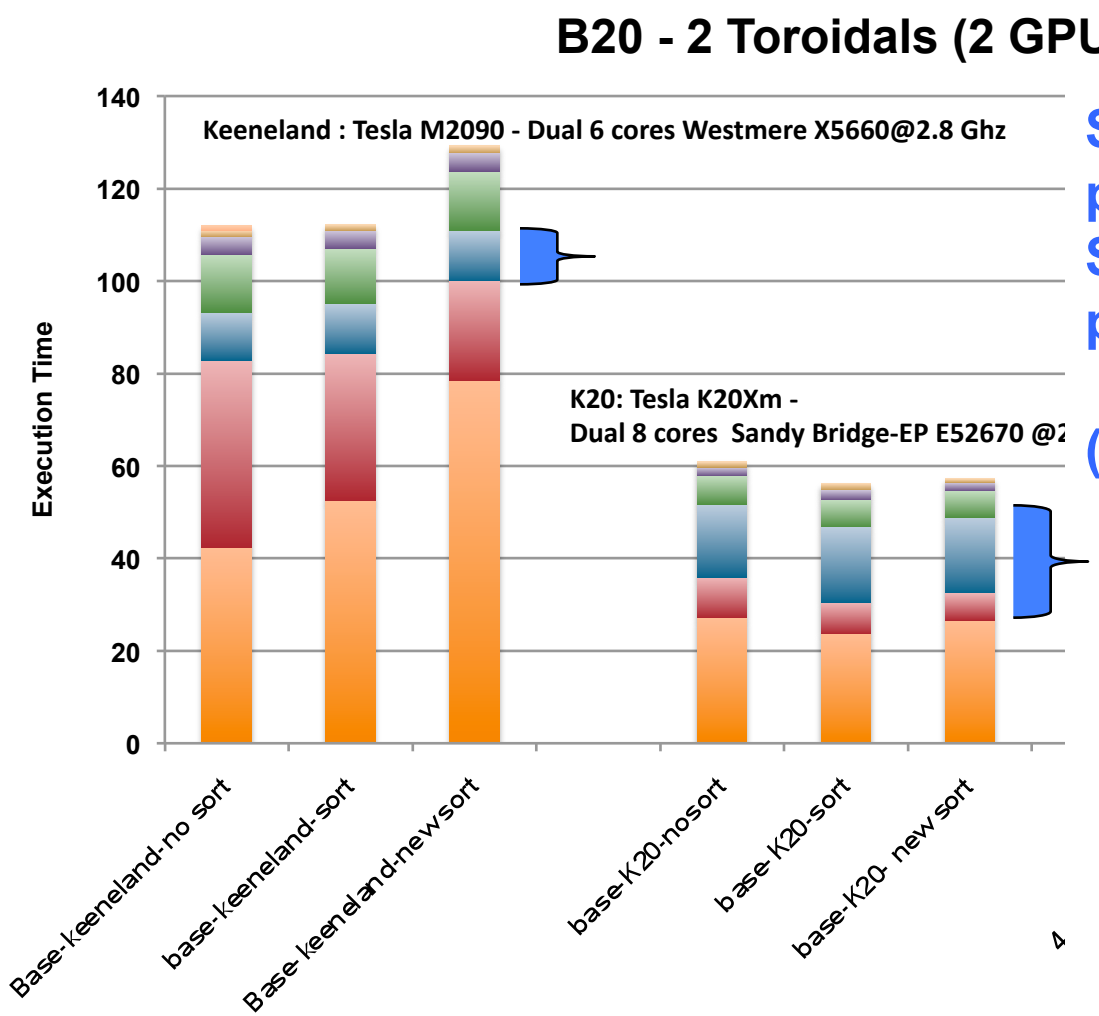
Kepler Vs. Fermi



Push: many thread blocks
- **Gather access pattern**
- **Larger L2**
- **Use of texture**

(3.5X-4.7X improvement)

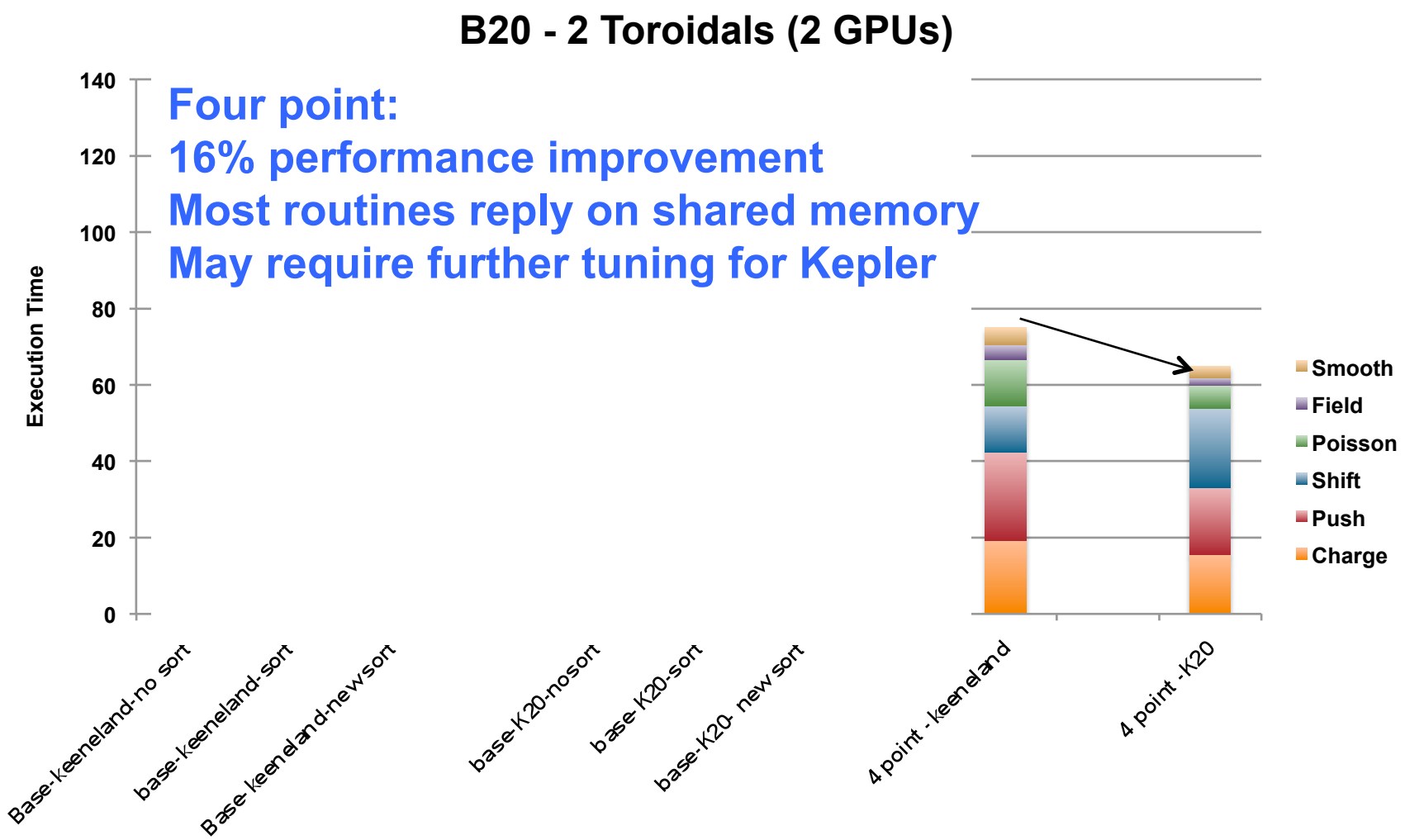
Kepler Vs. Fermi



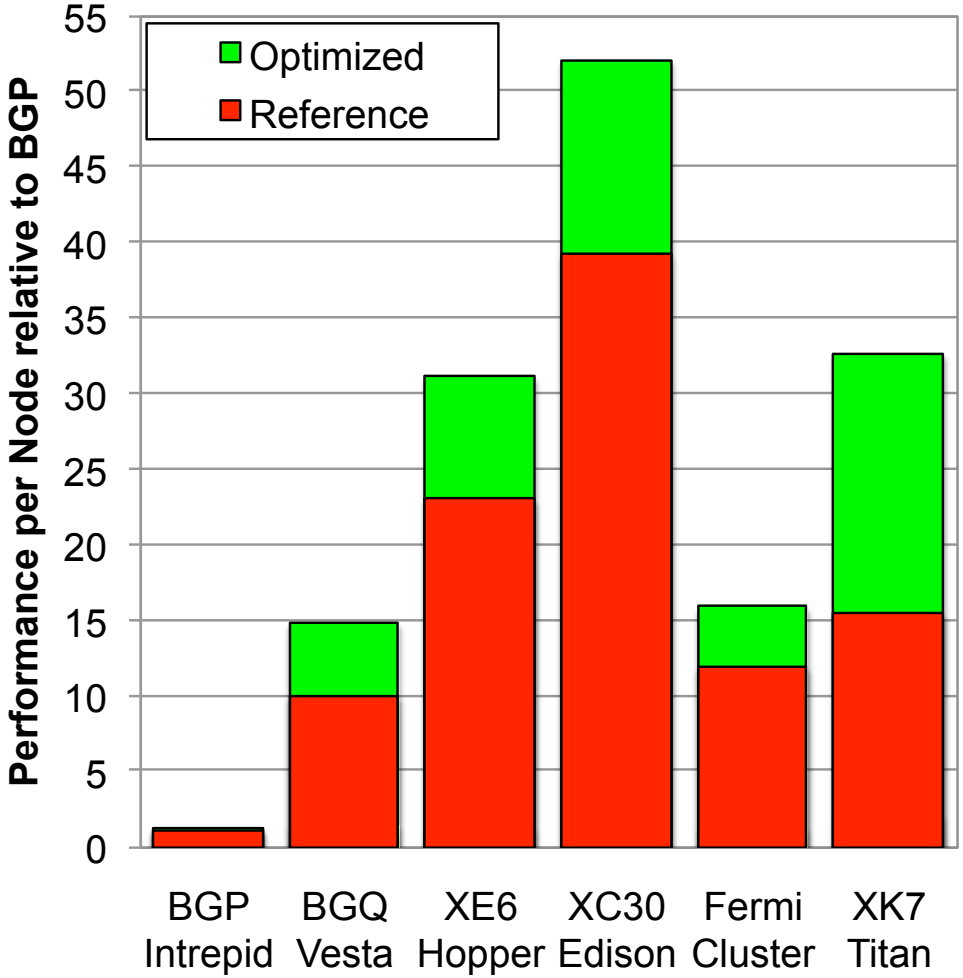
Shift: One thread block per multiprocessor. Shared memory 36KB per thread block-64 threads.

(51% increase in exec time)

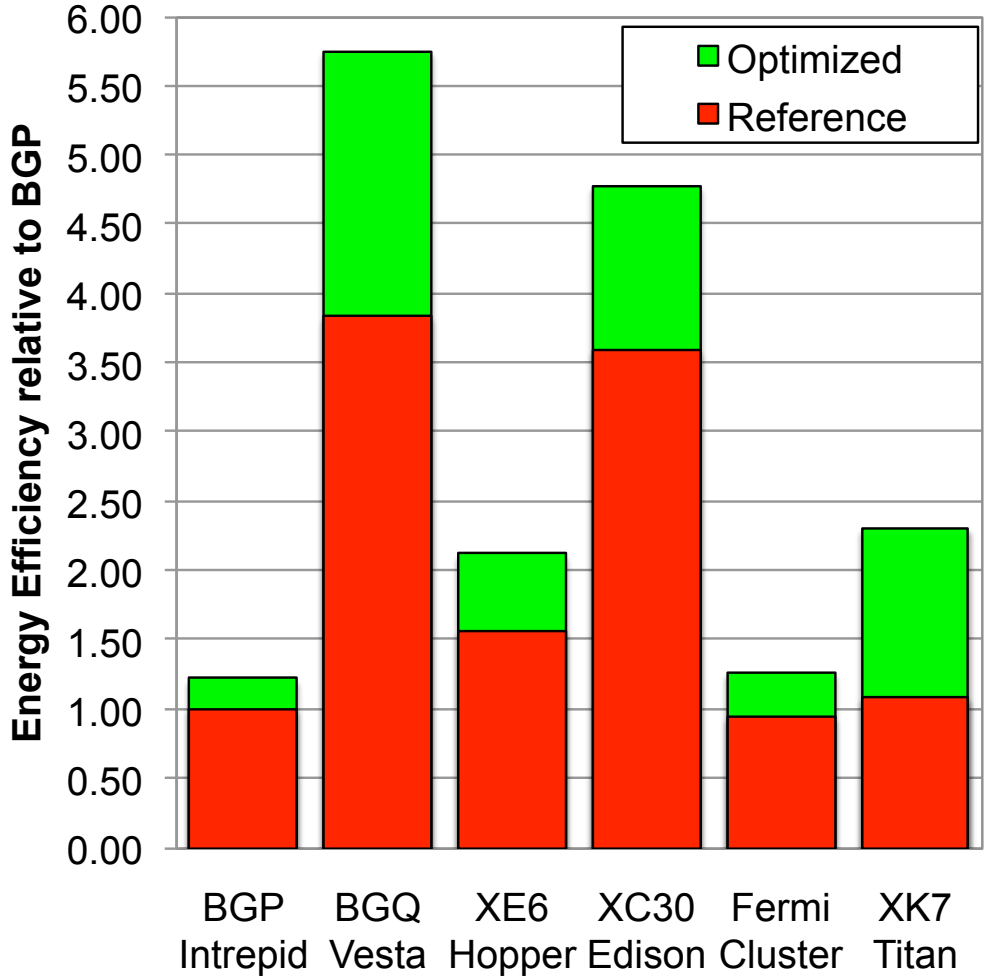
Kepler Vs. Fermi



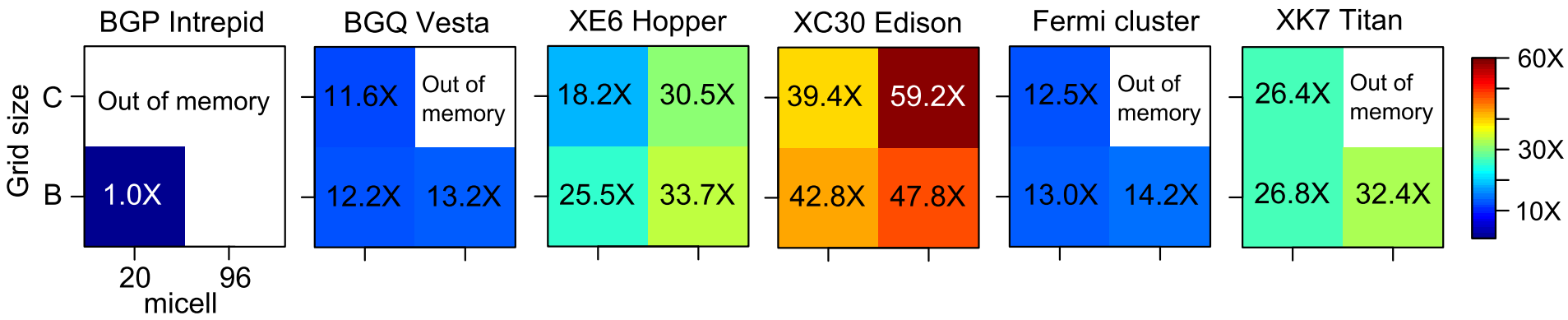
Comparison with Other Architectures B20 16-node experiments



Energy Efficiency



Dataset Impact on Performance



Summary

- GPU Implementation is less performing compared with CPU based implementations
  - Two radically different implementations for performance
    - one for Fermi architecture and the other is for K20.
  - Kepler has better support for atomics, improved the performance significantly 2.2x (up to 2.9x) for charge based on global atomics.
    - Kepler improves locality and thread block occupancy
    - Improvement up to 4.7x for the push phase (fastest across all platforms).
    - Possible improvement for the push is anticipated avoid dependency on shared memory.
- CPU implementations benefits from larger caches
  - Best performance is for Edison, best energy-efficiency is BG/Q.
  - Better temporal locality of accessing the grid
  - Avoiding atomic using limited replication.
  - Less complicated shift routine

Thanks

Your Questions!