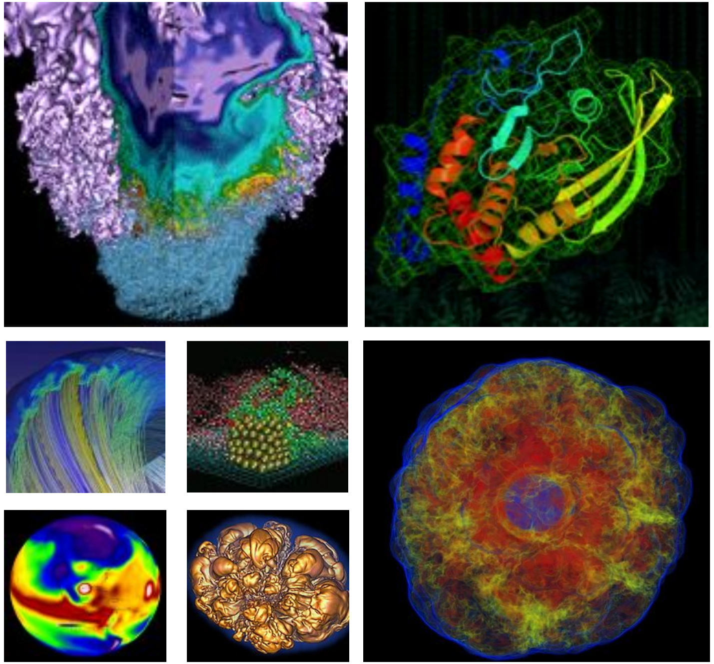
Optimization Use Cases with the Roofline Model



January, 2019

U.S. DEPARTMENT OF ENERGY | Office of Science

BERKELEY LAB

What was different about Cori?

Edison ("Ivy Bridge):

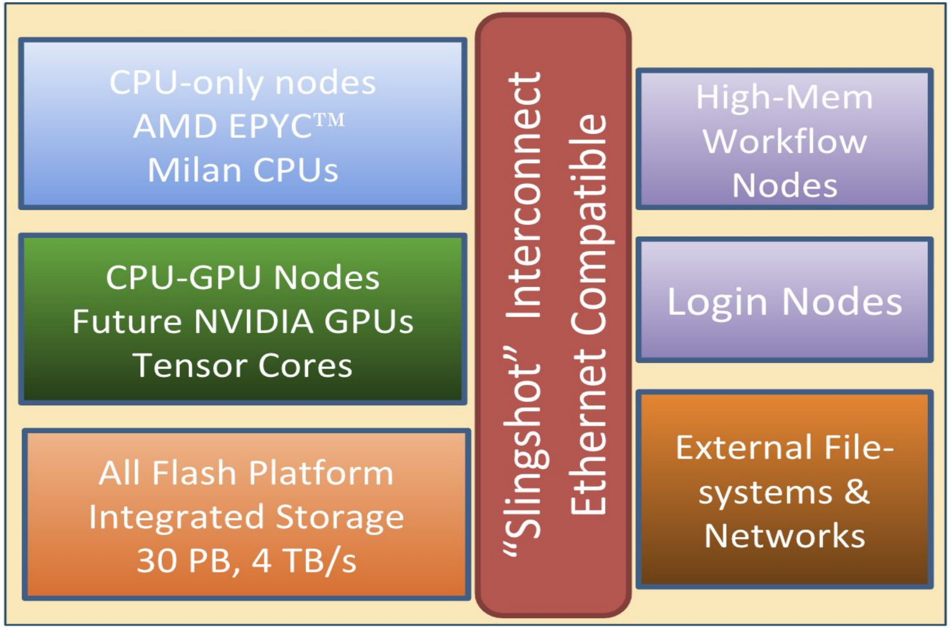
- 5576 nodes
- 24 physical cores per node
- 48 virtual cores per node
- 2.4 - 3.2 GHz
- 8 double precision ops/cycle
- 64 GB of DDR3 memory (2.5 GB per physical core)
- ~100 GB/s Memory Bandwidth

Cori ("Knights Landing"):

- 9304 nodes
- 68 physical cores per node
- 272 virtual cores per node
- 1.4 - 1.6 GHz
- 32 double precision ops/cycle
- 16 GB of fast memory
96GB of DDR4 memory
- Fast memory has 400 - 500 GB/s
- No L3 Cache

Perlmutter: A System Optimized for Science

- **GPU-accelerated and CPU-only nodes meet the needs of large scale simulation and data analysis from experimental facilities**
- **Cray "Slingshot" - High-performance, scalable, low-latency Ethernet-compatible network**
- **Single-tier All-Flash Lustre based HPC file system, 6x Cori's bandwidth**
- **Dedicated login and high memory nodes to support complex workflows**



U.S. DEPARTMENT OF ENERGY | Office of Science

BERKELEY LAB

Optimization Challenges For Scientists

Science teams need a simple way to wrap their heads around performance when main focus is scientific productivity:

1. Need a sense of absolute performance when optimizing applications.
   - How Do I know if My Performance is Good?
   - Why am I not getting peak performance advertised
   - How Do I know when to stop?

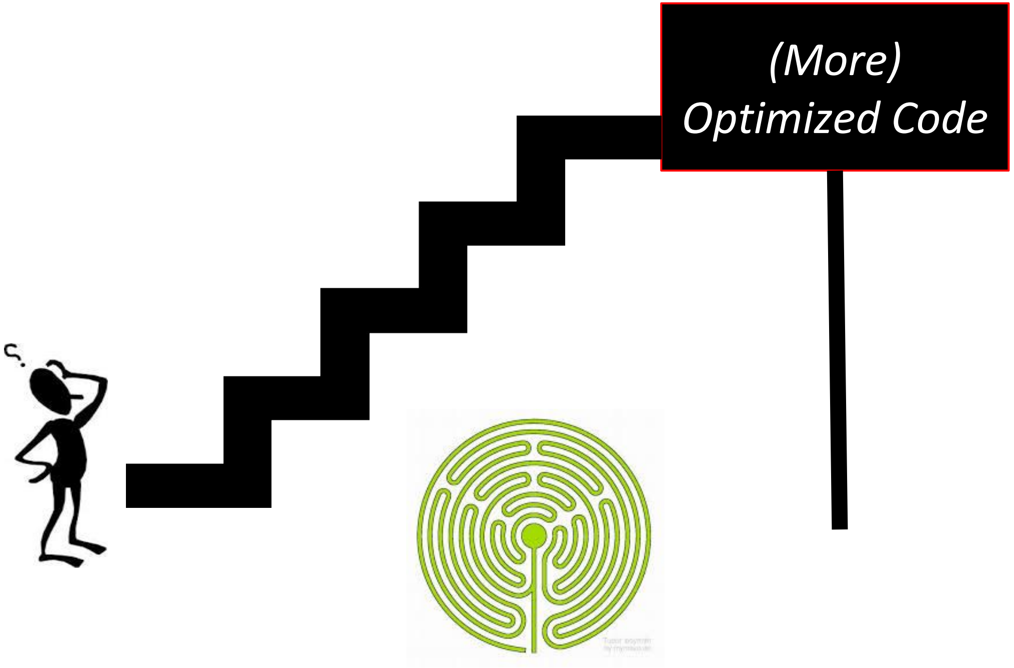
2. Many potential optimization directions:
   - How do I know which to apply?
   - What is the limiting factor in my app's performance?
   - Again, how do I know when to stop?

Optimizing Code For Cori is like:

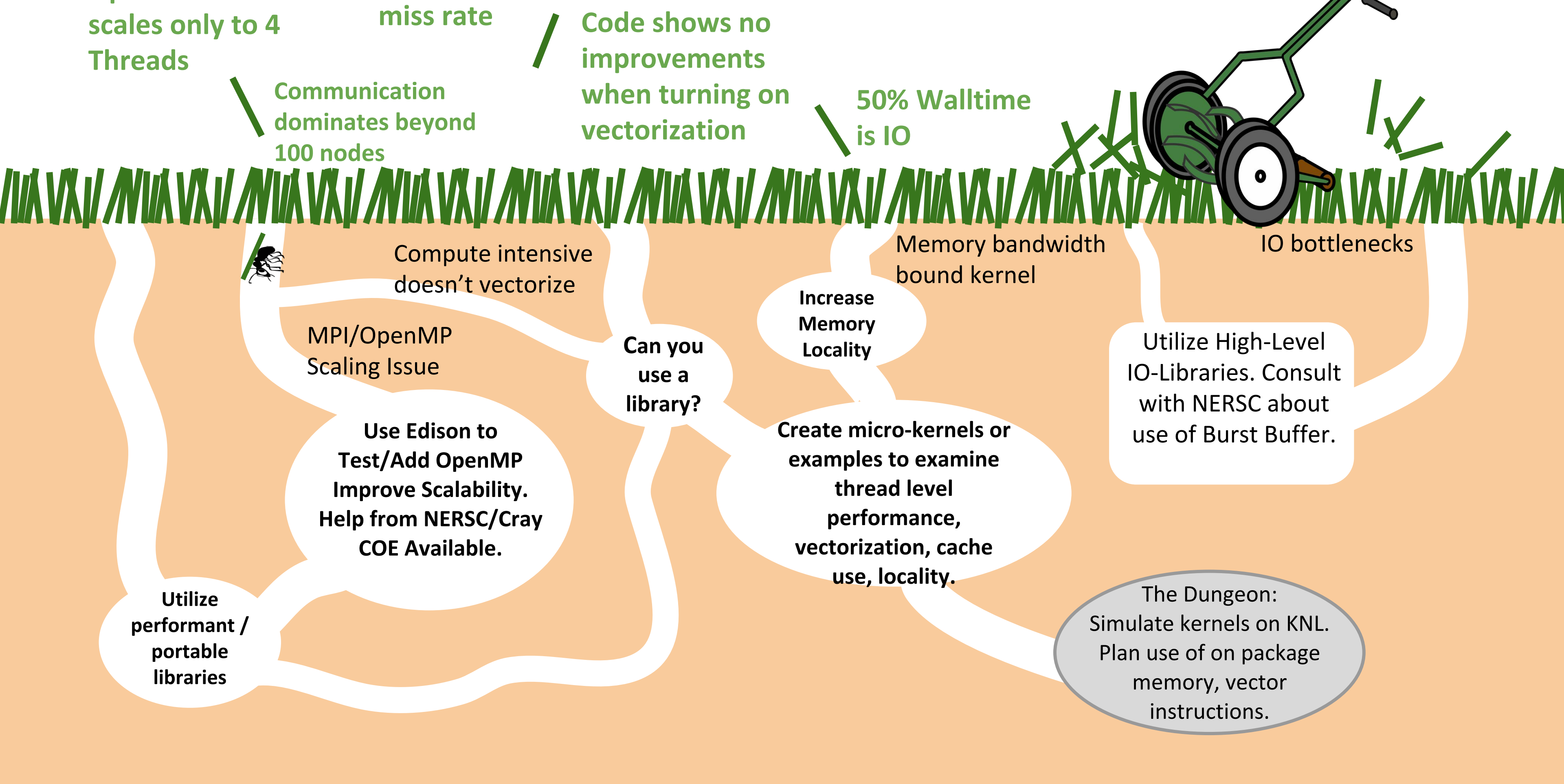
A. **A Staircase ?**

B. **A Labyrinth ?**

C. **A Space Elevator?**



The Ant Farm!



OpenMP scales only to 4 Threads
large cache miss rate
Communication dominates beyond 100 nodes
Code shows no improvements when turning on vectorization
50% Walltime is IO
Compute intensive doesn't vectorize
MPI/OpenMP Scaling Issue
Memory bandwidth bound kernel
IO bottlenecks
Increase Memory Locality
Can you use a library?
Use Edison to Test/Add OpenMP Improve Scalability. Help from NERSC/Cray COE Available.
Create micro-kernels or examples to examine thread level performance, vectorization, cache use, locality.
Utilize High-Level IO-Libraries. Consult with NERSC about use of Burst Buffer.
Utilize performant / portable libraries
The Dungeon: Simulate kernels on KNL. Plan use of on package memory, vector instructions.

Are you memory or compute bound? Or both?

Run Example in "Half Packed" Mode

If you run on only half of the cores on a node, each core you do use has access to more bandwidth

```
aprun -n 24 -N 12 - S 6 ...
```

VS

```
aprun -n 24 -N 24 -S 12 ...
```

```
srun -N 2 -n 24 -c 2 - S 6 ...
```

VS

```
srun -N 1 -n 24 -c 1 ...
```

If your performance changes, you are at least partially memory bandwidth bound

Are you memory or compute bound? Or both?

Run Example at “Half Clock” Speed

Reducing the CPU speed slows down computation, but doesn’t reduce memory bandwidth available.

```
aprun --p-state=2400000 ...
```

VS

```
aprun --p-state=1900000 ...
```

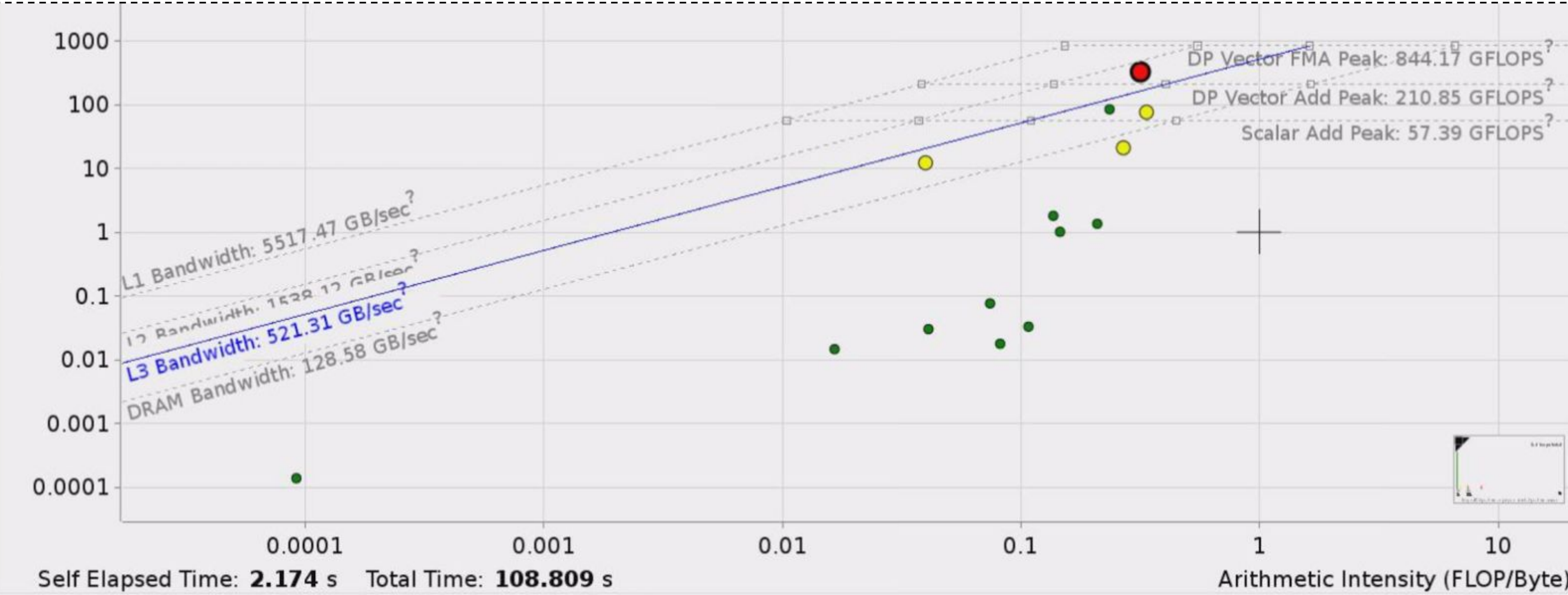
```
srun --cpu-freq=2400000 ...
```

VS

```
srun --cpu-freq=1900000 ...
```

If your performance changes, you are at least partially compute bound

Tools CoDesign

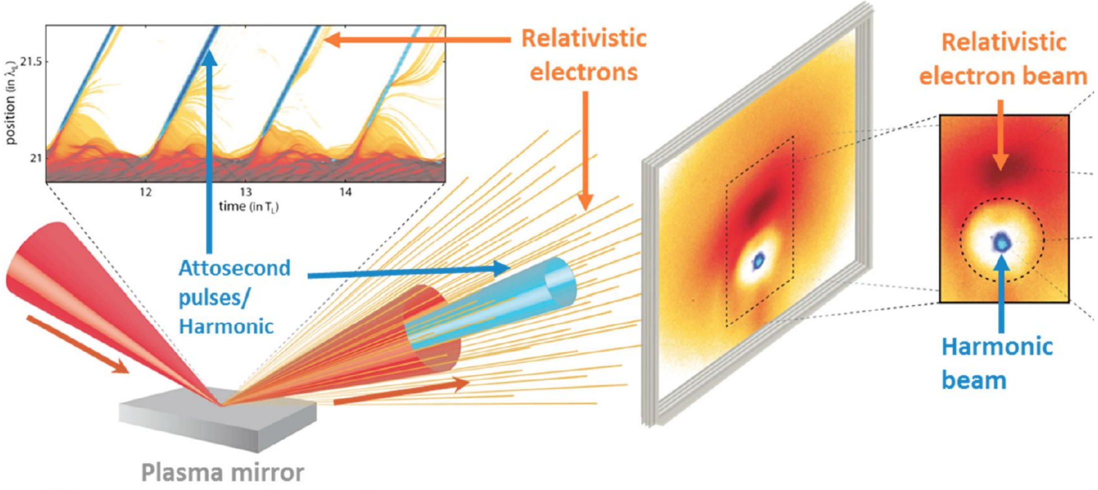


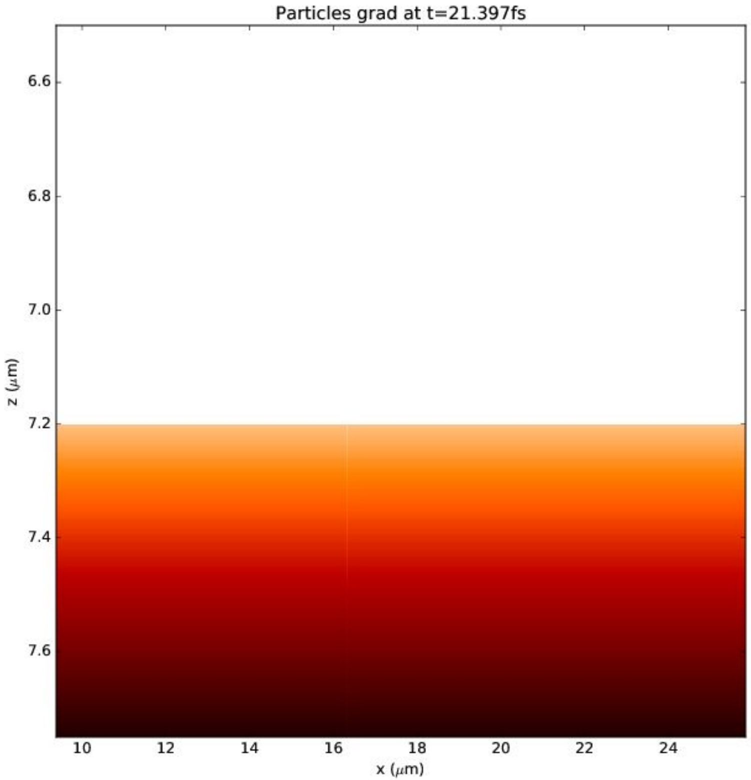
Intel Vector-Advisor Co-Design - Collaboration between NERSC, LBNL Computational Research, Intel

Example: WARP (Accelerator Modeling)

- Particle in Cell (PIC) Applicaion for doing accelerator modeling and related applications.

- **Example Science**: Generation of high-frequency attosecond pulses is considered as one of the best candidates for the next generation of attosecond light sources for ultrafast science.



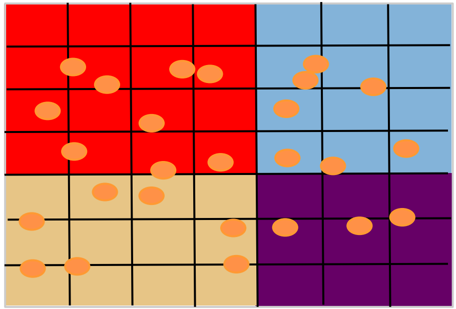


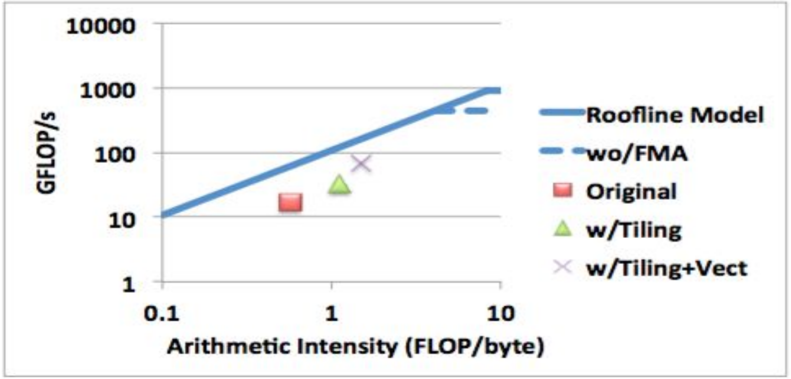
Animation from Plasma Mirror Simulations

Roofline helps visualize this information! Guides optimizations

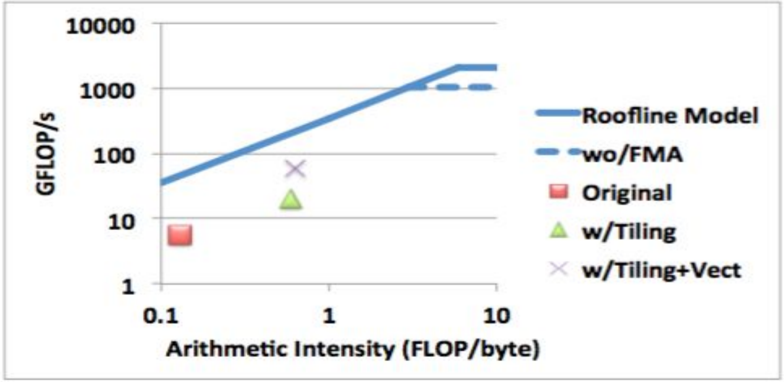
WARP Optimizations:

1. Add tiling over grid targeting L2 cache on both Xeon-Phi Systems
2. Add particle sorting to further improve locality and memory access pattern
3. Apply vectorization over particles



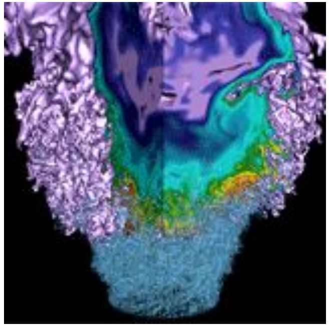


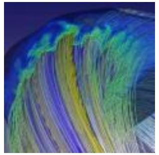
(a) Haswell Roofline

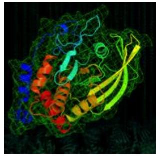


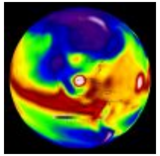
(b) KNL Roofline

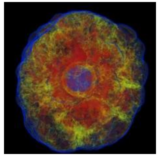
NESAP Example

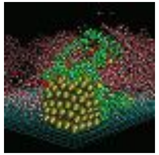












NERSC 40 YEARS at the FOREFRONT

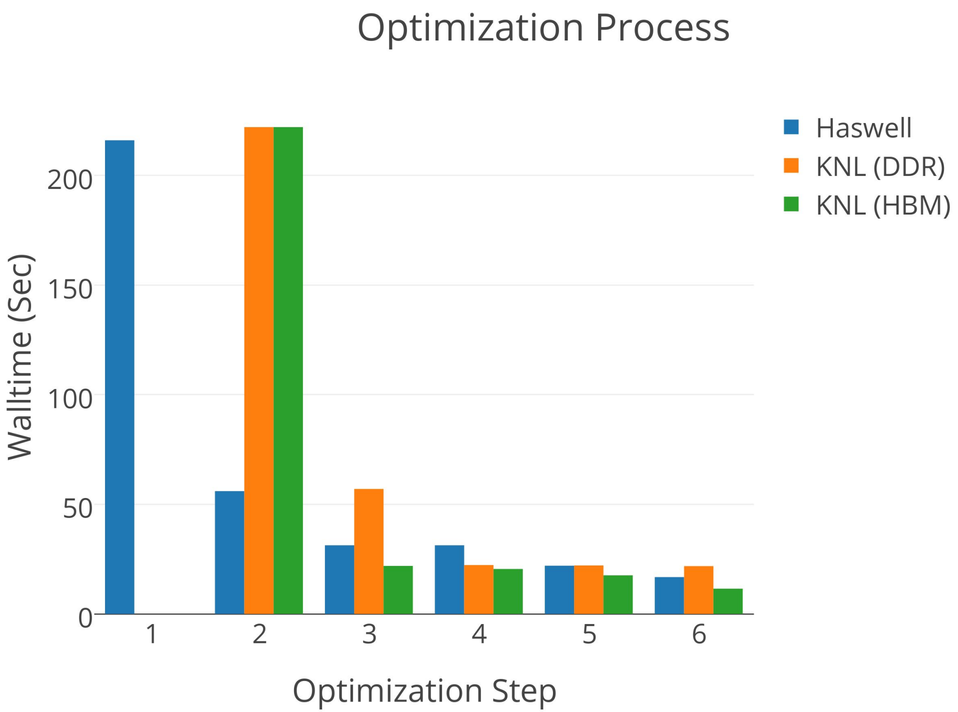
U.S. DEPARTMENT OF ENERGY | Office of Science

BERKELEY LAB

BerkeleyGW NESAP Project Optimization Path

Optimization process Sigma code:

1. Add OpenMP
2. Initial Vectorization (loop reordering, conditional removal)
3. Cache-Blocking
4. Improved Vectorization (Divides)
5. Hyper-threading



Vectorization

```
!$OMP DO reduction(+:achtemp)
  do my_igp = 1, ngpown
    ...
    do iw=1,nfreq ! nfreq is 3

      scht=0D0
      wxt = wx_array(iw)

      do ig = 1, ncouls

        !if (abs(wtilde_array(ig,my_igp) * eps(ig,my_igp)) .lt. TOL) cycle

        wdiff = wxt - wtilde_array(ig,my_igp)
        delw = wtilde_array(ig,my_igp) / wdiff
        ...
        scha(ig) = mygpvar1 * aqsntemp(ig) * delw * eps(ig,my_igp)
        scht = scht + scha(ig)

      enddo ! loop over g
      sch_array(iw) = sch_array(iw) + 0.5D0*scht

    enddo

    achtemp(:) = achtemp(:) + sch_array(:) * vcoul(my_igp)

  enddo
```

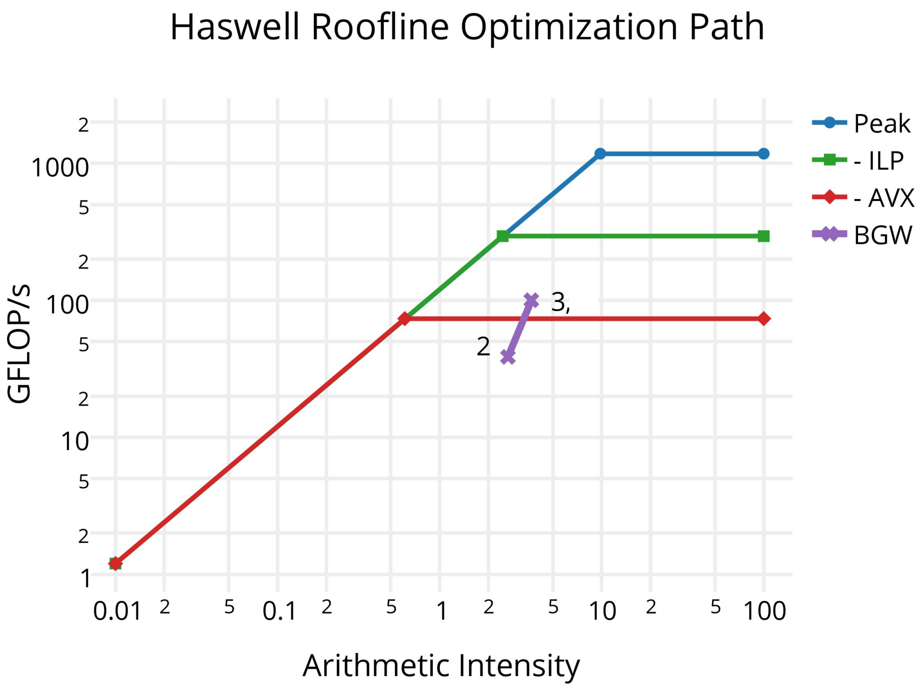
ngpown typically in 100's to 1000s. Good for many threads.

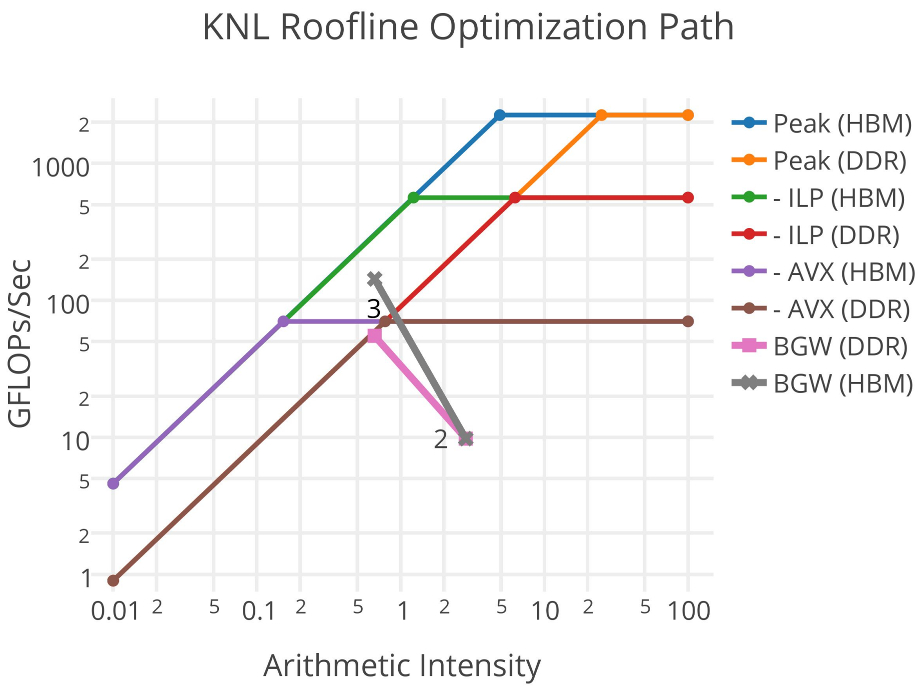
Original inner loop. Too small to vectorize!

ncouls typically in 1000s - 10,000s. Good for vectorization.

Attempt to save work breaks vectorization and makes code slower.

Change in Roofline





The loss of L3 on KNL makes locality more important.

Why KNC worse than Haswell for GPP Kernel?

```
!$OMP DO
do my_igp = 1, ngpown
        do iw = 1 , 3
            do ig = 1, igmax
                load wtilde_array(ig,my_igp) 819 MB, 512KB per row
                load aqsntemp(ig,n1) 256 MB, 512KB per row
                load I_eps_array(ig,my_igp) 819 MB, 512KB per row
                do work (including divide)
```

Required Cache size to reuse 3 times:

1536 KB

L2 on KNL is 512 KB per core
L2 on Has. is 256 KB per core

L3 on Has. is 3800 KB per core

Without blocking we spill out of L2 on KNL and Haswell. But, Haswell has L3 to catch us.

Why KNC worse than Haswell for GPP Kernel?

```
!$OMP DO
do my_igp = 1, ngpown
  do igbeg = 1, igmax, igblk
    do iw = 1 , 3
      do ig = igbeg, min(igbeg + igblk,igmax)
        load wtilde_array(ig,my_igp) 819 MB, 512KB per row
        load aqsntemp(ig,n1) 256 MB, 512KB per row
        load I_eps_array(ig,my_igp) 819 MB, 512KB per row
        do work (including divide)
```

Required Cache size to reuse 3 times:

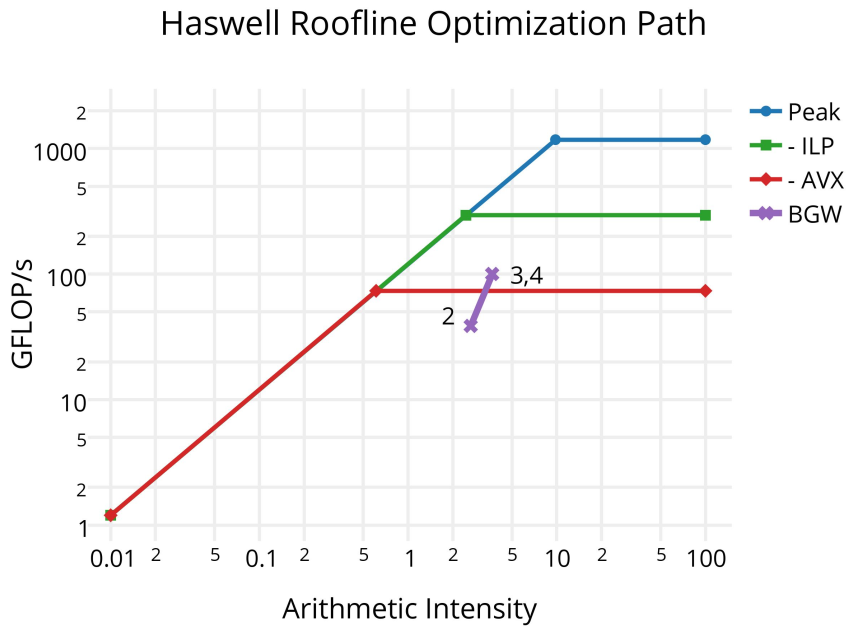
1536 KB

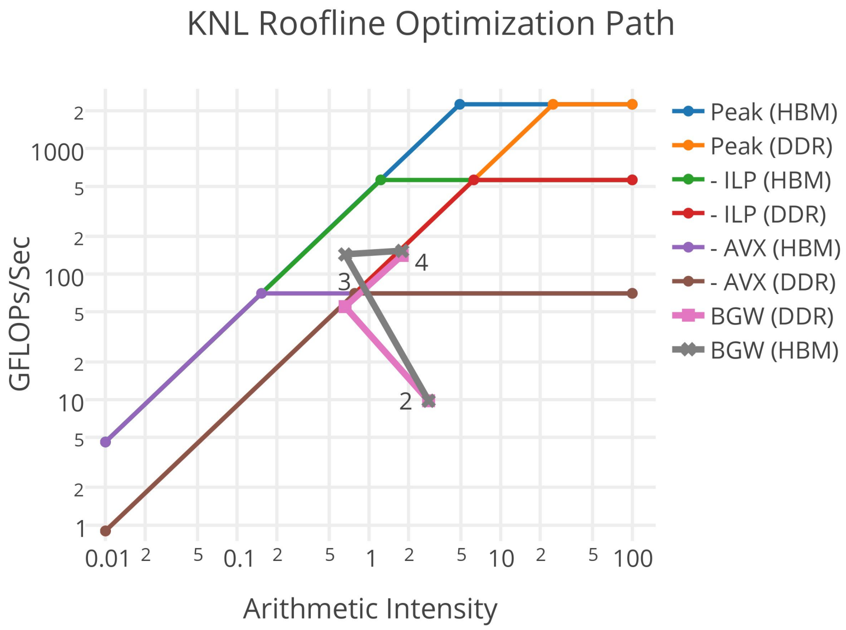
L2 on KNL is 512 KB per core
L2 on Has. is 256 KB per core

L3 on Has. is 3800 KB per core

Without blocking we spill out of L2 on KNL and Haswell. But, Haswell has L3 to catch us.

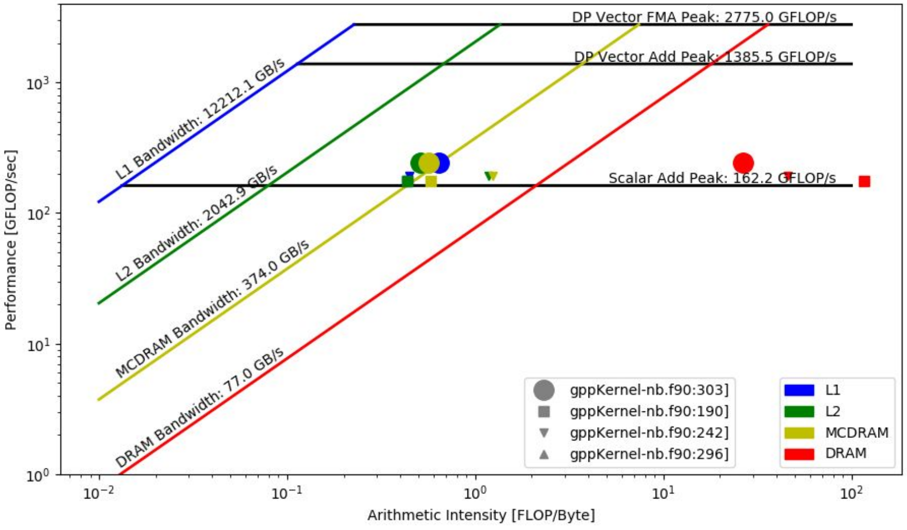
Cache Blocking Optimization

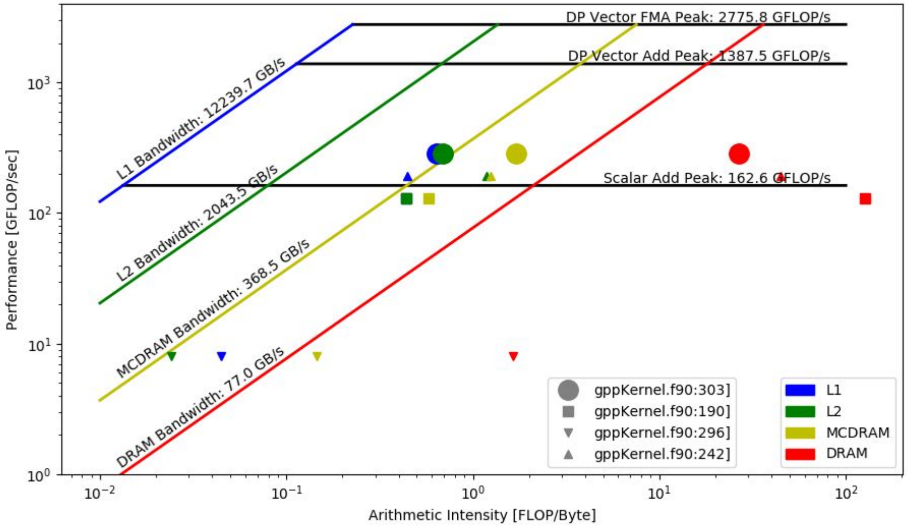




Cache Blocking Optimization (Hierarchical Roofline)

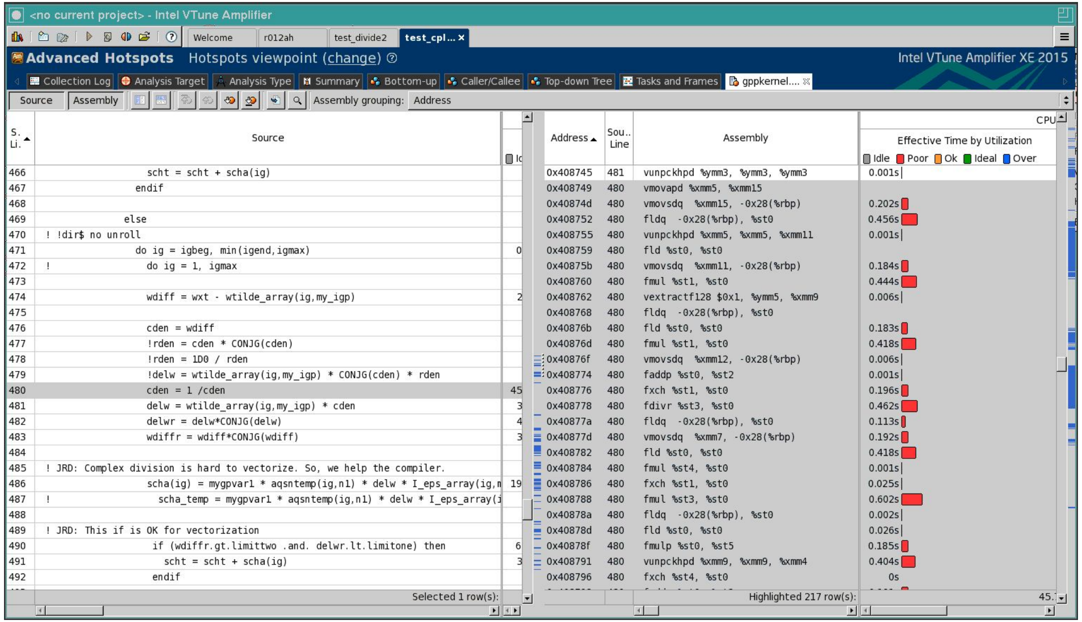
Original Code





Why Complex Divides so Slow?

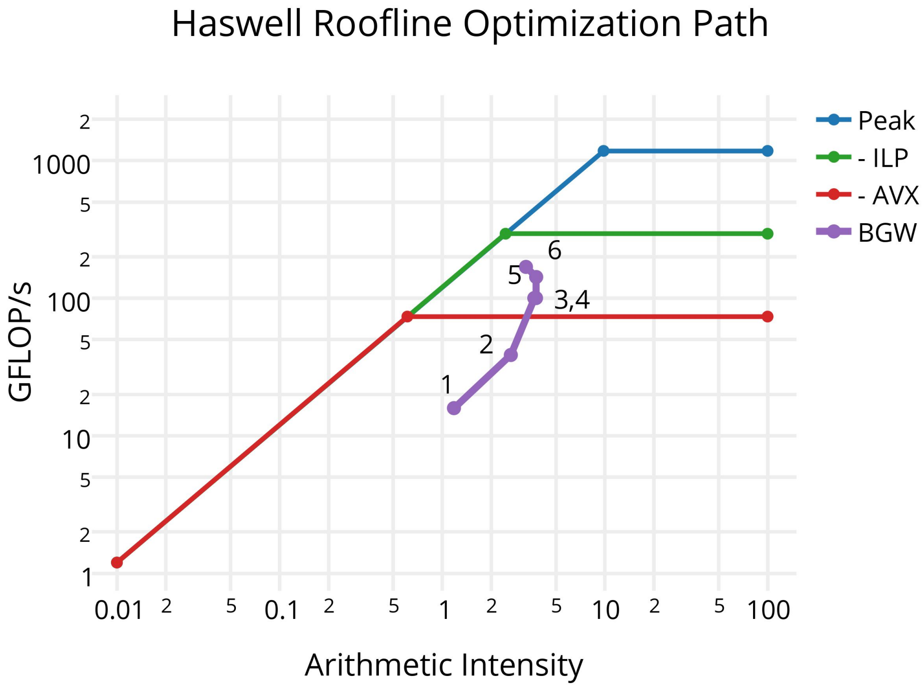
Found significant x87 instructions from 1/complex_number instead of AVX/AVX-512

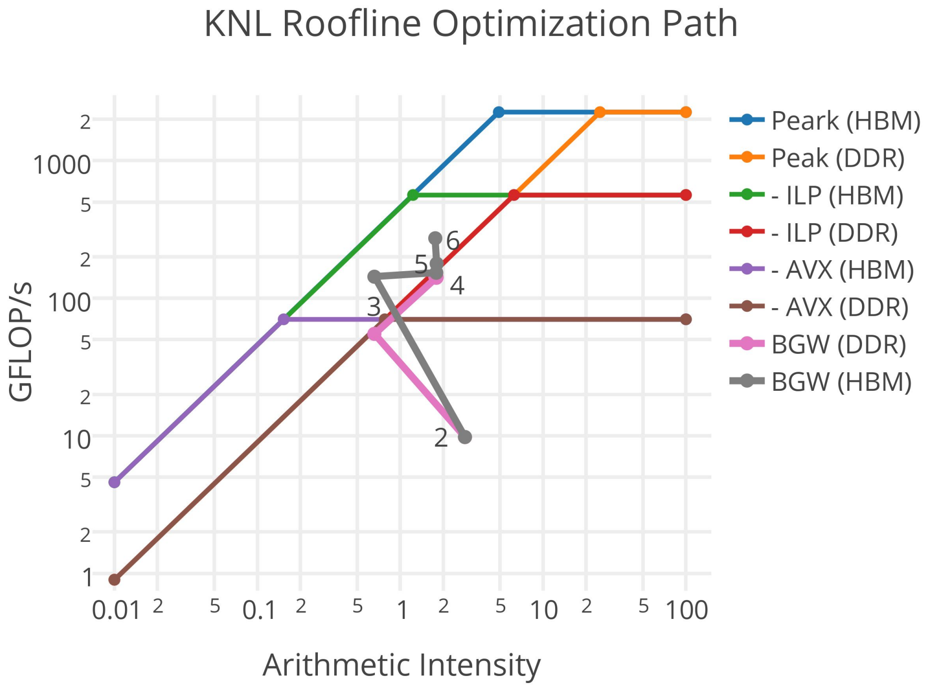


Can significantly speed up by using

-fp-model fast=2

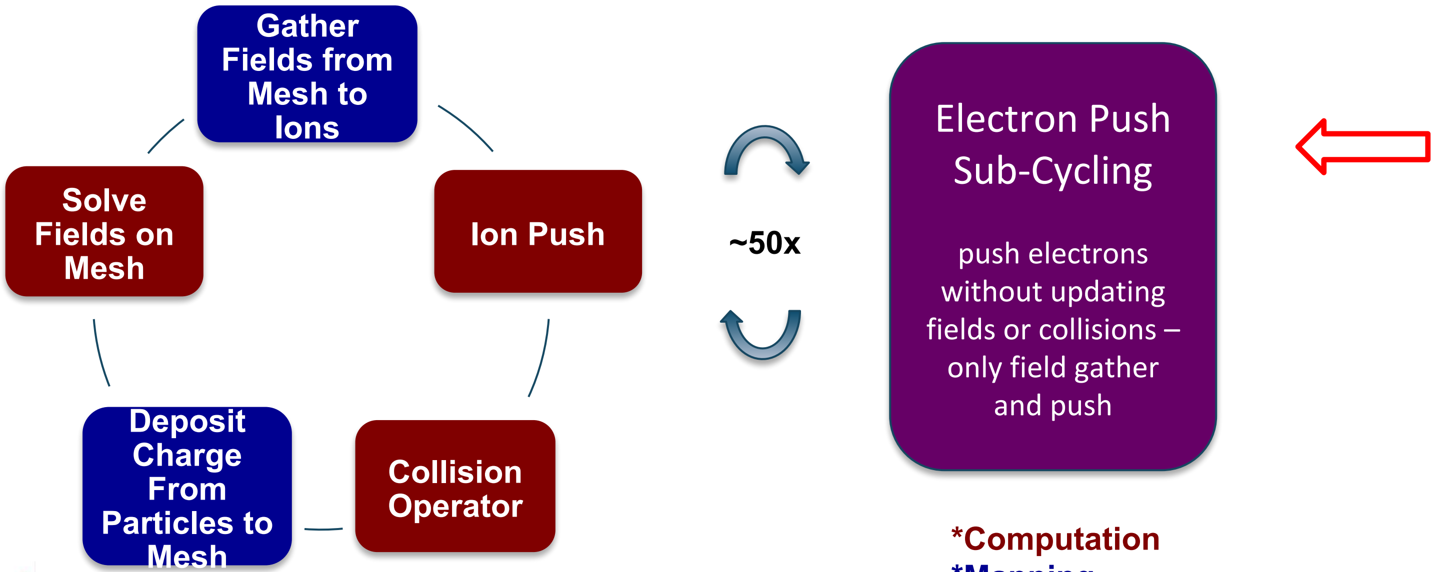
Additional Speedups from Hyperthreading





XGC1: Particle-In-Cell (PIC)

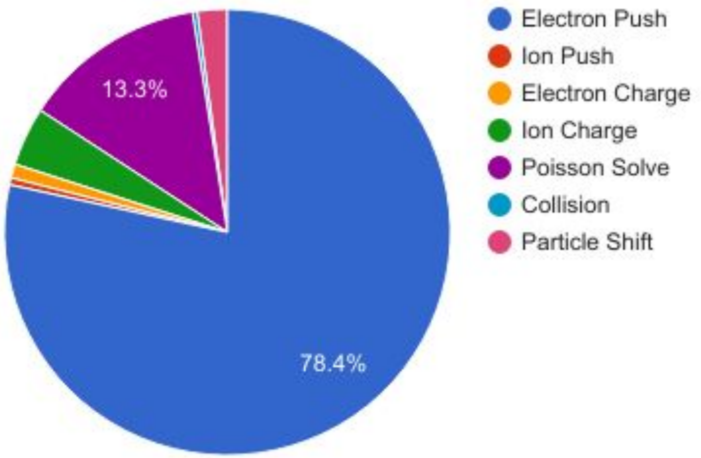
- PIC code to simulate edge plasmas for Tokamak fusion reactor
- Written in Fortran 90, parallelized with MPI and OpenMP
- **Code analysis:**

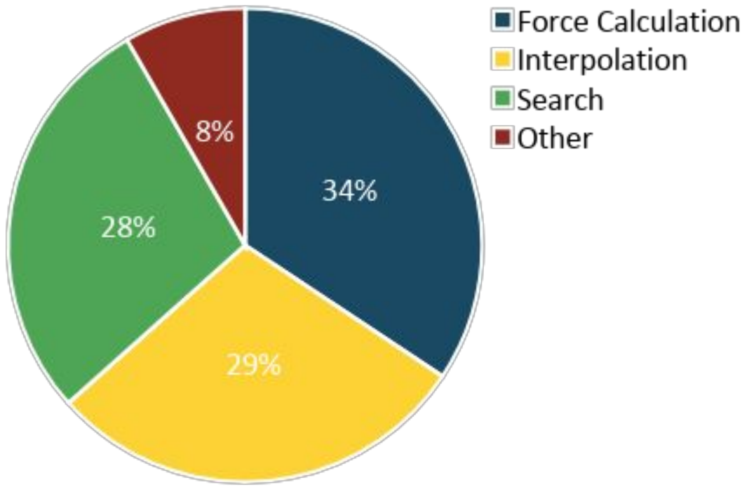


*Computation
*Mapping

XGC1 - ToyPush

- **Hotspot analysis:**

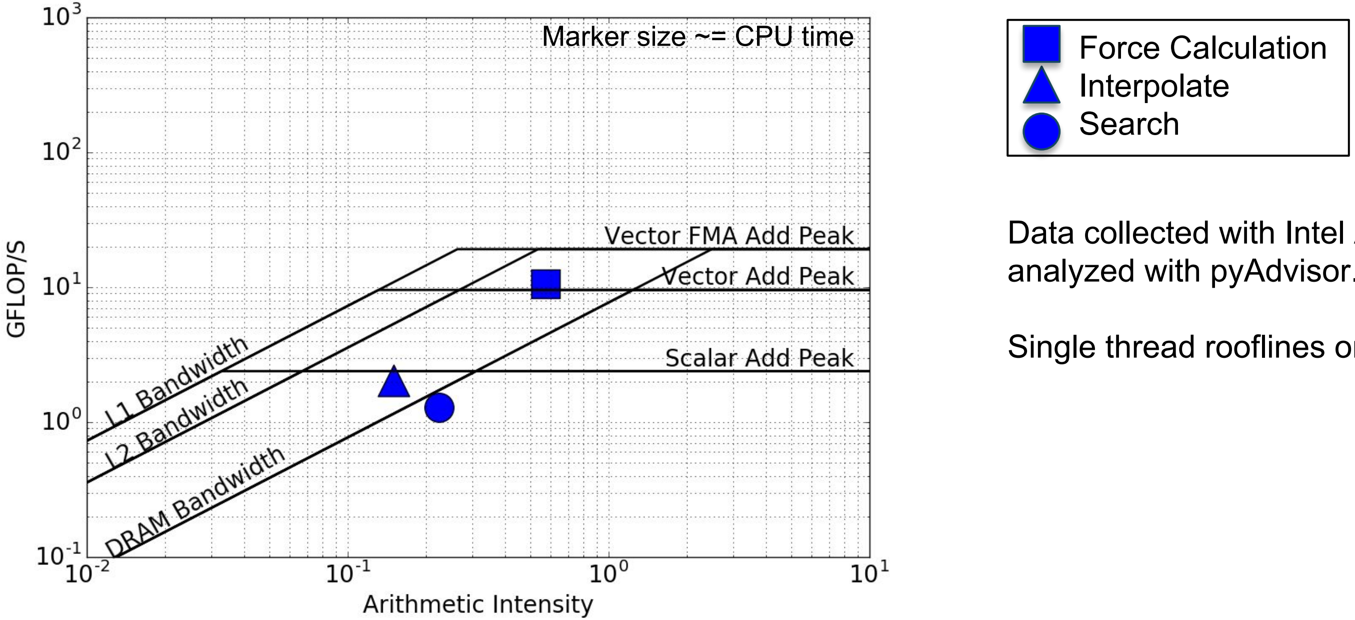




Left: Unoptimized XGC1 timings on 1024 Cori KNL nodes in Quad-Flat mode
Right: Unoptimized ToyPush timings on Cori KNL in Quad-Cache mode
***ToyPush is the proxy app for electron push part of XGC1.**

ToyPush: Baseline Profile

- Force Calculation: close to vector peak
- Interpolate and Search: less than scalar peak



Data collected with Intel Advisor and analyzed with pyAdvisor.

Single thread rooflines on Cori KNL.

ToyPush - Interpolation

- **Compiler vectorization report**
- Indirect access/gathers -> group particles together that access the same triangle

```
efield(j,tri(i,itri(iv)))
```

- Unaligned access -> align at compile time
- Improved vectorization efficiency

LOOP BEGIN at interpolate_aos.F90(67,48)
reference itri(iv) has unaligned access
reference y(iv,1) has unaligned access
reference y(iv,3) has unaligned access
reference evec(iv,icomp) has unaligned access
reference evec(iv,icomp) has unaligned access
…..
irregularly indexed load was generated for the variable <grid_mapping_(1,3,itri(iv))>, 64-bit indexed, part of index is read from memory
…..

LOOP WAS VECTORIZED
unmasked unaligned unit stride loads: 6
unmasked unaligned unit stride stores: 3
unmasked indexed (or gather) loads: 18
…..

ToyPush - Interpolation

- **Use Advisor to examine cache behavior**
- L1 hit rate low -> shorten veclength from 2^9 to 2^6 to achieve L1 blocking

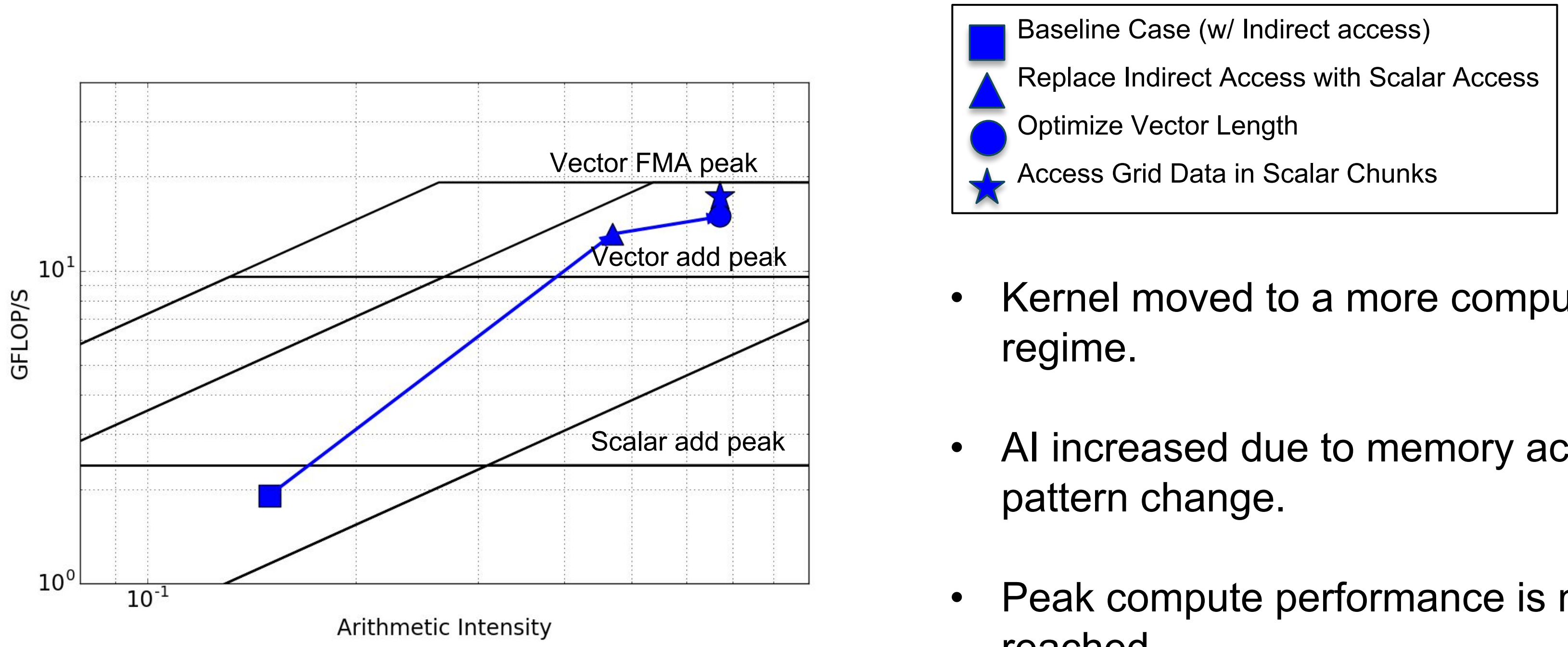
Grouping: Function / Call Stack

Function / Call Stack	Clockticks ▼	Instructions Retired	L1 Hit Rate	L2 Hit Rate	L2 Hit Bound	L2 Miss Bound
e_interpol_tri	105,271,600,000	64,954,400,000	80.8%	94.4%	36.7%	29.5%
eom_eval	73,858,400,000	65,283,400,000	67.3%	99.9%	100.0%	0.8%
b_interpol_analytic	60,141,200,000	23,109,800,000	90.3%	100.0%	4.2%	0.0%
__intel_mic_avx512f_memset	35,288,400,000	3,441,200,000	42.1%	100.0%	0.8%	0.0%
rk4_push	20,528,200,000	14,898,800,000	31.9%	100.0%	100.0%	0.0%

Grouping: Function / Call Stack

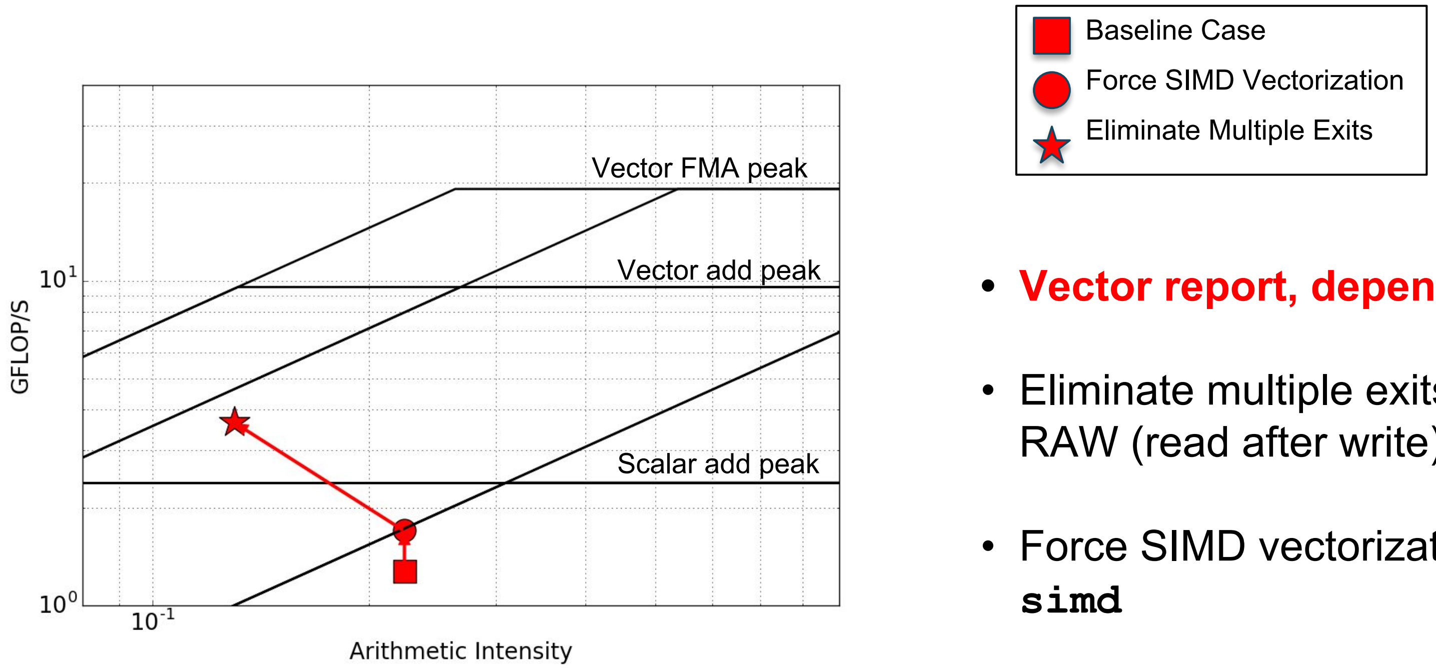
Function / Call Stack	Clockticks ▼	Instructions Retired	L1 Hit Rate	L2 Hit Rate	L2 Hit Bound	L2 Miss Bound
e_interpol_tri	97,042,400,000	76,687,800,000	99.4%	100.0%	0.9%	0.0%
eom_eval	66,556,000,000	67,110,400,000	99.0%	100.0%	3.3%	0.0%
b_interpol_analytic	16,360,400,000	23,641,800,000	99.3%	100.0%	0.3%	0.0%
proc_reg_read	14,984,200,000	75,600,000	100.0%	0.0%	0.0%	0.0%
rk4_push	14,954,800,000	19,702,200,000	98.5%	100.0%	24.8%	0.0%

ToyPush - Interpolation



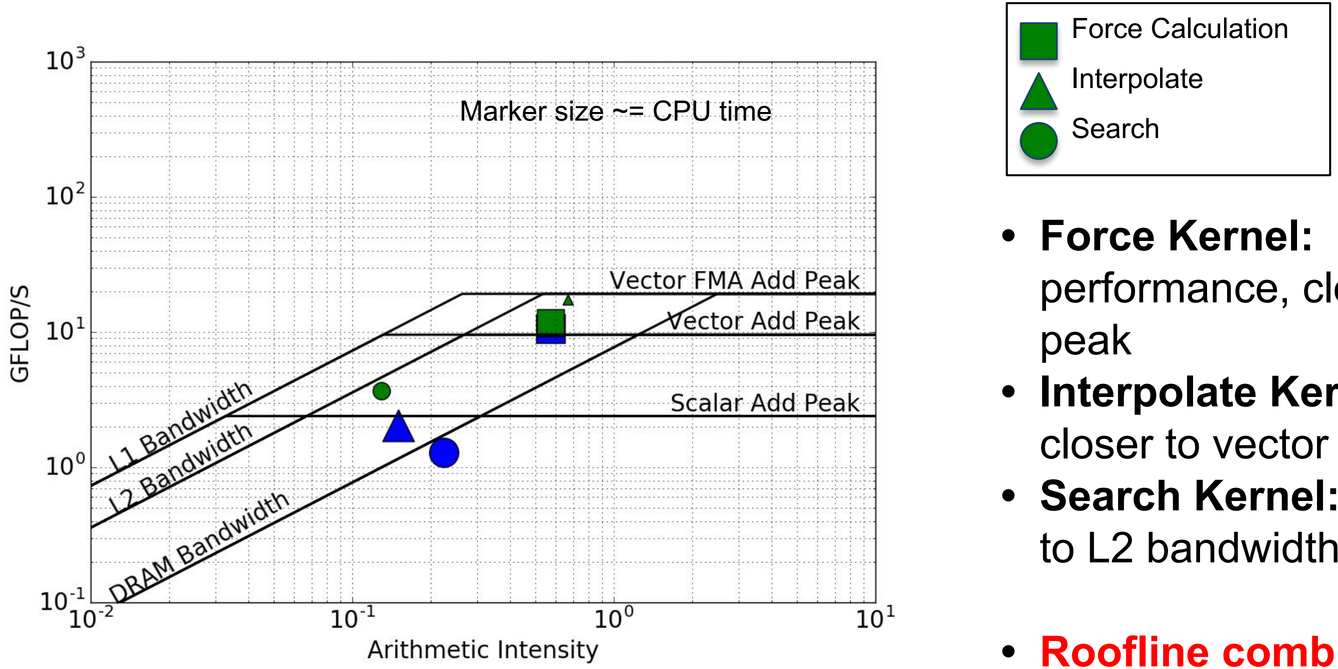
- Kernel moved to a more compute bound regime.
- AI increased due to memory access pattern change.
- Peak compute performance is nearly reached.

ToyPush - Search



- **Vector report, dependency report**
- Eliminate multiple exits, ‘cycle’, and RAW (read after write) dependency
- Force SIMD vectorization with **`omp simd`**

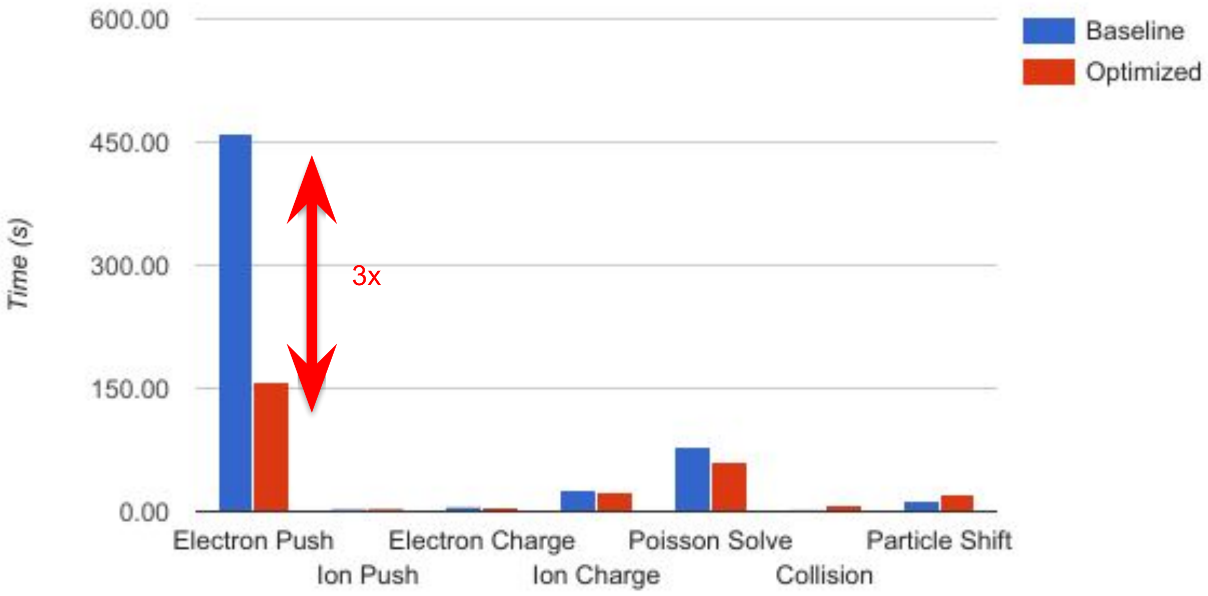
ToyPush: Optimized Profile



- **Force Kernel:** still good performance, close to vector add peak
- **Interpolate Kernel:** 10x speedup, closer to vector FMA peak
- **Search Kernel:** 3x speedup, closer to L2 bandwidth roof

- **Roofline combined with other analysis/tools**

XGC1: Merge ToyPush Changes (WIP)



XGC1 Timings on 1024 Cori KNL nodes in Quad-Flat mode

Summary

- Showcased three scientific applications, and their performance analysis and/or optimization process: Warp, BerkeleyGW, and XGC1.

- Roofline model can help identify performance bottlenecks, prioritize optimization efforts (e.g. routines, vectorization, memory access), and tell when to stop (e.g. attainable performance, distance to roofs).

- Complement Roofline with generic code analysis, compiler reports, binary analysis to confirm details and ways to implement optimizations.
  - vectorization, dependency, memory access pattern, cache locality, cache hit rate, instruction mix

- Tools such as Intel Advisor, Intel VTune, NVProf are very useful!
- Something about Perlmutter