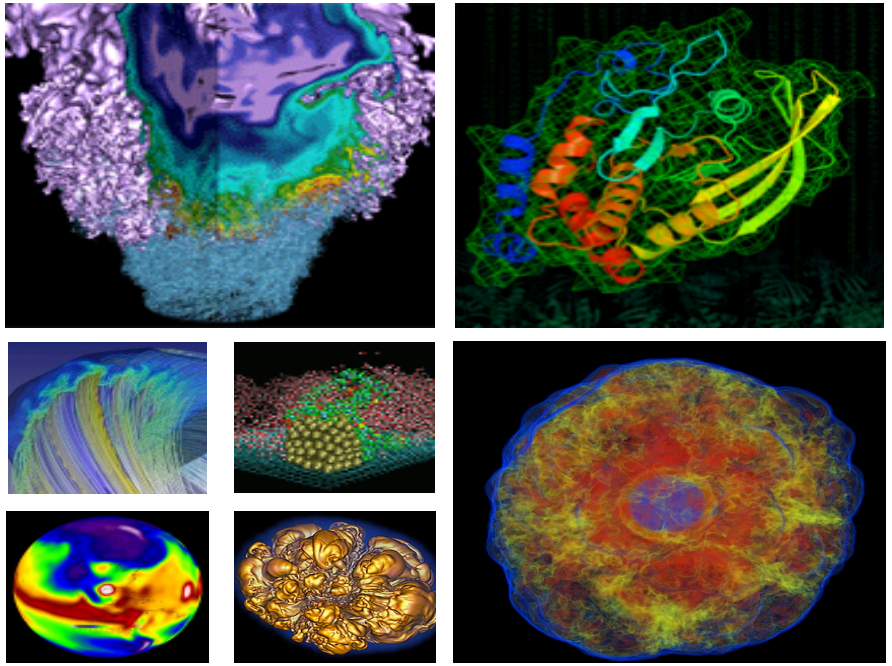
Utilizing Roofline Analysis in the Intel® Advisor to Deliver Optimized Performance for applications on Intel® Xeon Phi™ Processor (Code named Knights Landing)



Tuomas Koskela, Mathieu Lobet, Jack Deslippe, Zakhar Matveev

NESAP postdoctoral fellow

November 12, 2016

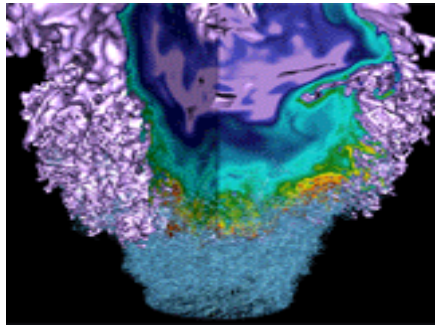
Why Do We Need the Roofline Model?

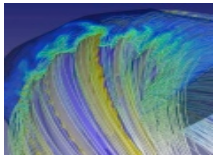
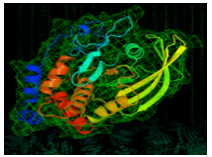
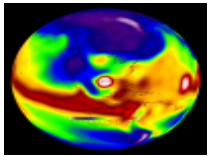
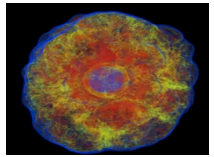
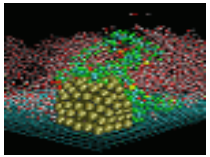
- Need a sense of absolute performance when optimizing applications
  - How do I know if my performance is good?
  - Why am I not getting peak performance of the platform?
- Many potential optimization directions
  - How do I know which one to apply?
  - What is the limiting factor in my app's performance?
  - How do I know when to stop?

Outline

1. Description and purposes of the roofline performance model
2. Building the roofline model for your codes
3. Case studies
    1. PICSAR, a high-performance PIC library for MIC architectures
    2. XGC1, a PIC code for fusion plasmas
4. Conclusions

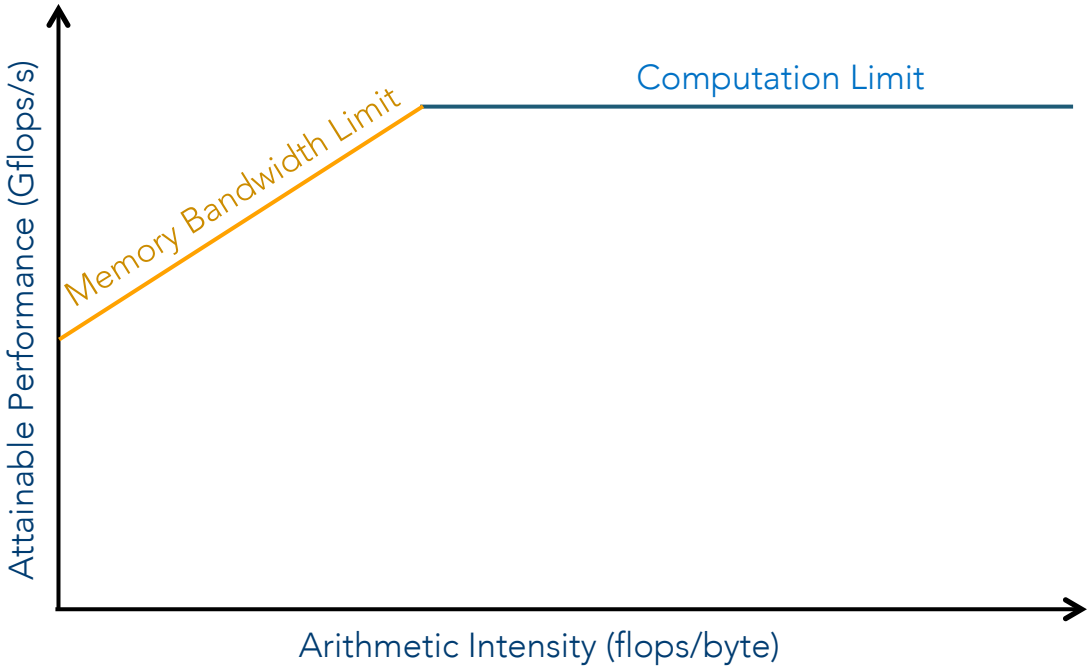
Part 1:
Description and purposes of the roofline performance model



1. General description
2. How to read the roofline model
3. Classical versus cache-aware roofline model

Roofline [1] is a visual performance model



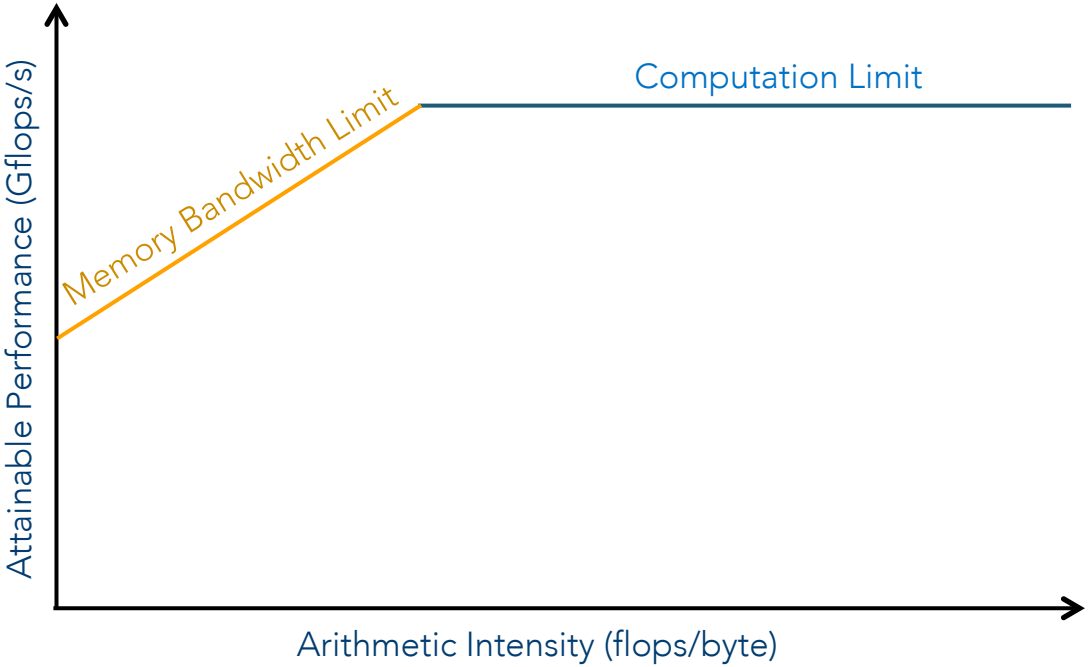
"Roofline is a visually intuitive performance model used to bound the performance of various numerical methods and operations running on multicore, manycore, or accelerator processor architectures."

Roofline:

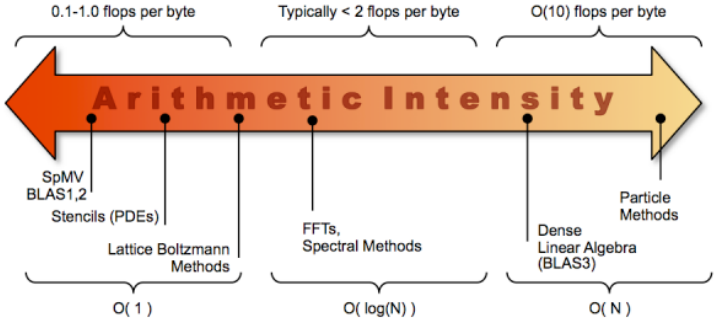
- Reflects a performance bound (Gflops/s) as a function of Arithmetic Intensity (AI).
- Is a performance envelope under which kernel or application performance exists.

[1] S. Williams et al. *CACM* (2009), crd.lbl.gov/departments/computer-science/PAR/research/roofline

Arithmetic Intensity is a ratio of Flops to Bytes

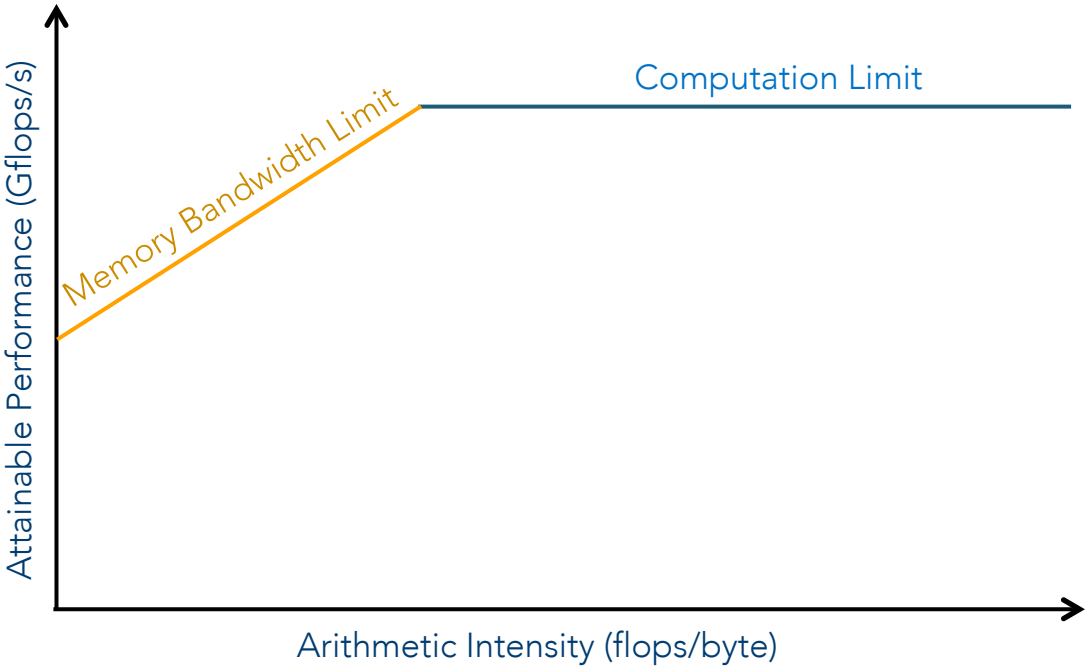


$$\text{Arithmetic Intensity} = \frac{\text{Total Flops computed}}{\text{Total Bytes transferred from DRAM}}$$



[1] S. Williams et al. *CACM* (2009), crd.lbl.gov/departments/computer-science/PAR/research/roofline

Peak Performance is Bounded by the Roofline

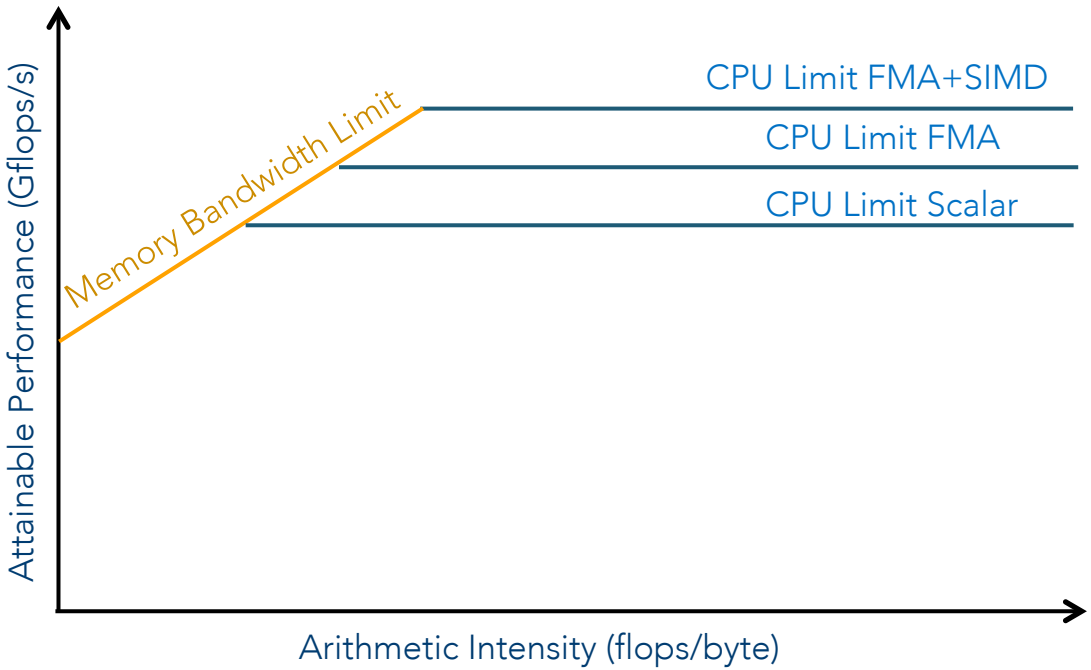


The attainable system performance is the maximal performance that can be reached by an application:

Attainable perf Gflops/s = min { Peak performace Gflops/s ; Peak Memory Bandwidth x Arithmetic Intensity }

[1] S. Williams et al. *CACM* (2009), crd.lbl.gov/departments/computer-science/PAR/research/roofline

There Are Different Ceilings



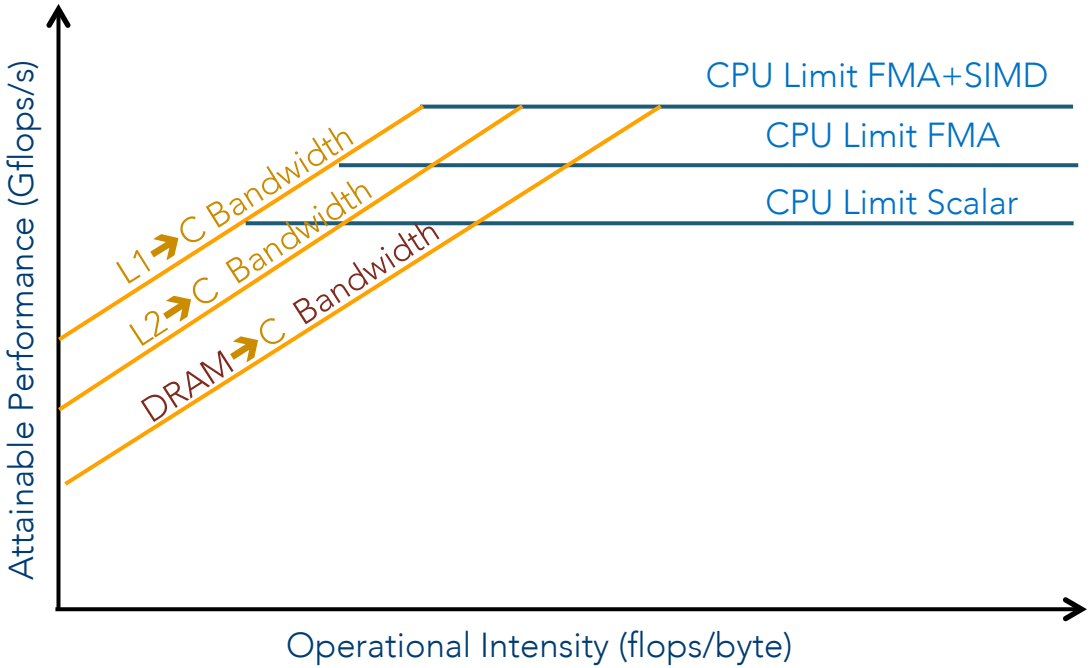
KNL peak performance = AVX Frequency (1.4 Ghz) x 8 (vector width) x 2 (dual vpus) x 2 (FMA inst.) x number of Cores

Bandwidth can be taken as STREAM TRIAD value

Bandwidth and peak performance can be computed, e.g., using the Empirical roofline toolkit from LBNL [2]

[1] S. Williams et al. *CACM* (2009), crd.lbl.gov/departments/computer-science/PAR/research/roofline

The Cache-Aware Roofline Model



Total volume of bytes transferred across all memory hierarchies to the core:

Attainable perf Gflops/s = min { Peak performace Gflops/s ; Bandwidths to Core x Operational Intensity }

Bandwidths to Core {
- Bandwidth from L1 to Core
- Bandwidth from L2 to Core
- Bandwidth from L3 to Core
- Bandwidth from DRAM to Core
}

[1] S. Williams et al. *CACM* (2009), crd.lbl.gov/departments/computer-science/PAR/research/roofline

Classical Roofline vs Cache-Aware Roofline

Classical Roofline Model [1]

AI = # FLOP/
BYTES (DRAM →)

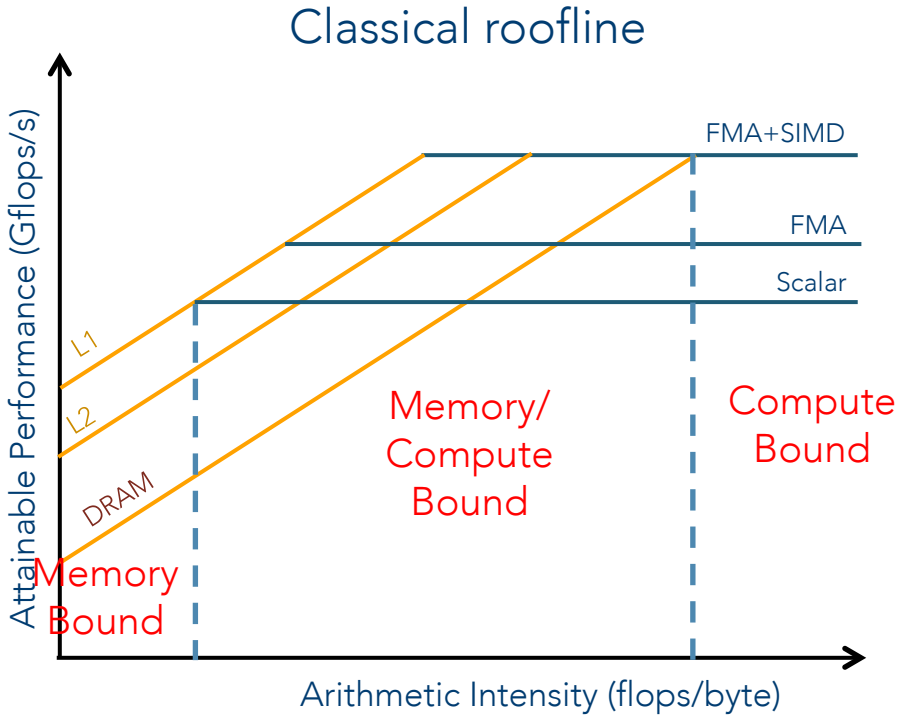
- Bytes are measured out of a given level in memory hierarchy
  - DRAM is a common choice
- AI depends on problem size
- AI is platform dependent
- Memory optimizations will change AI

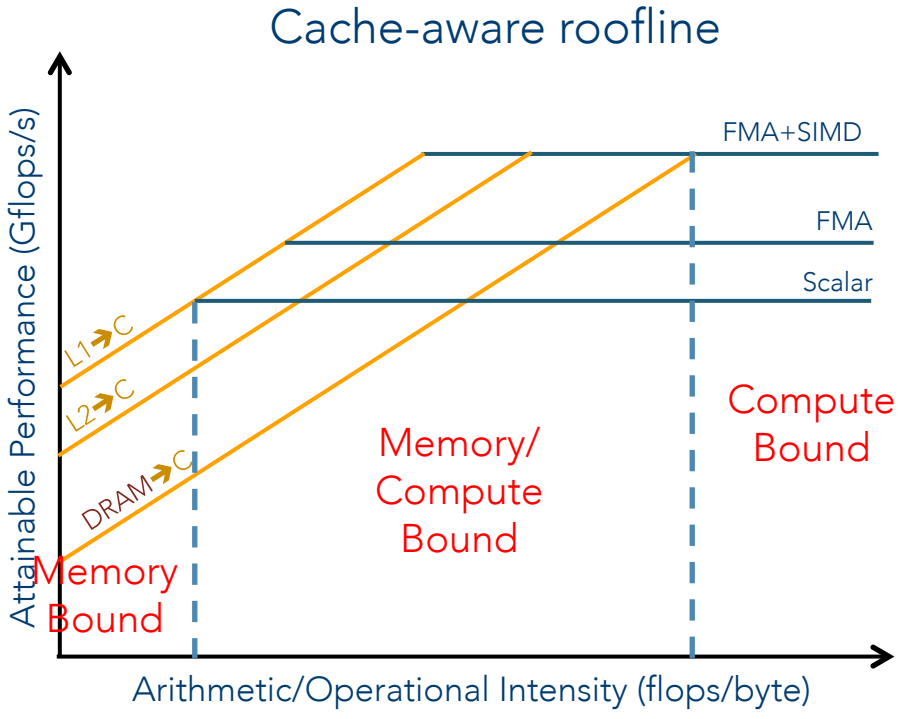
Cache-Aware Roofline Model [2]

AI = # FLOP /
# BYTES (→ CPU)

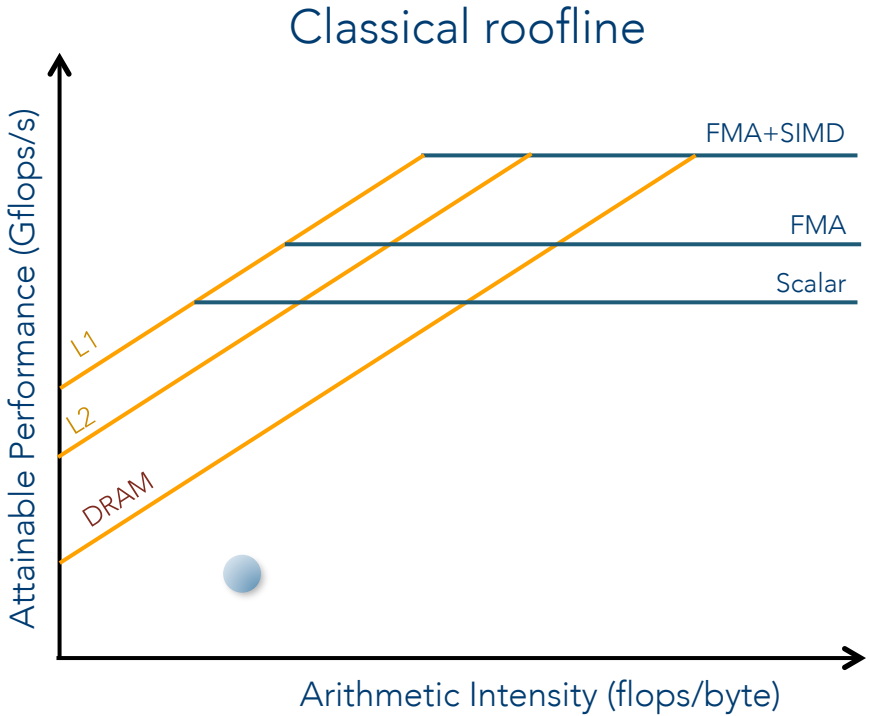
- Bytes are measured into the cpu from all levels in memory hierarchy
- AI is independent of problem size
- AI is independent of platform
- AI is constant for given algorithm

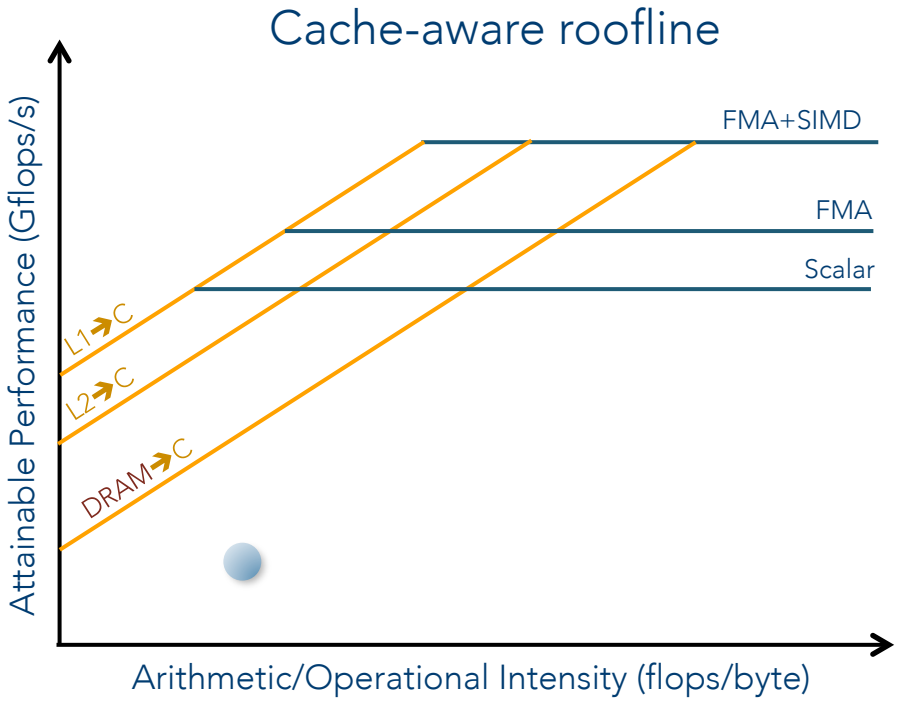
Am I Bound By Memory or CPU?





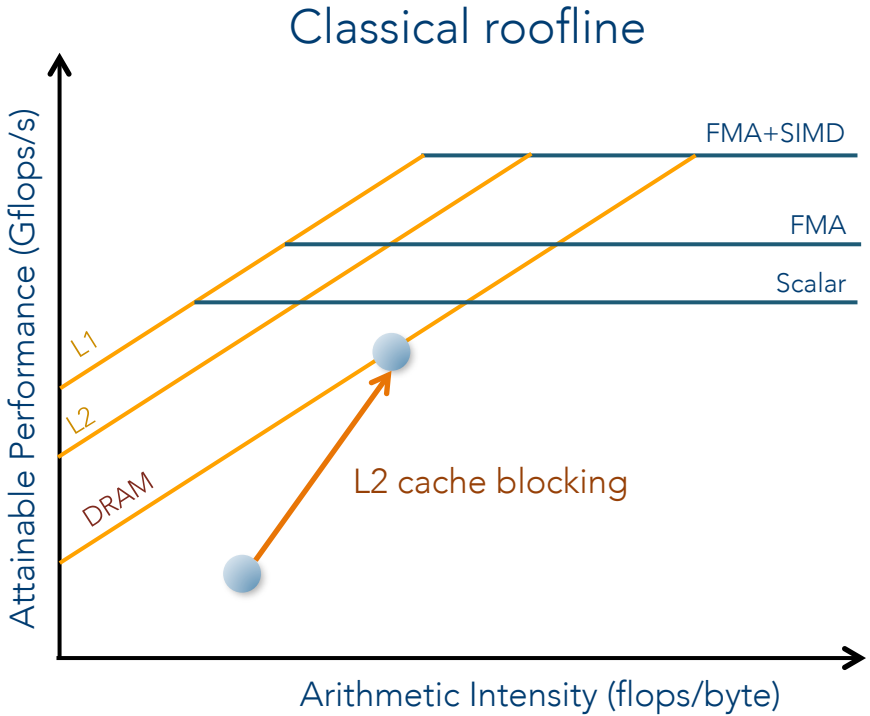
Example 1: Memory Bound Application

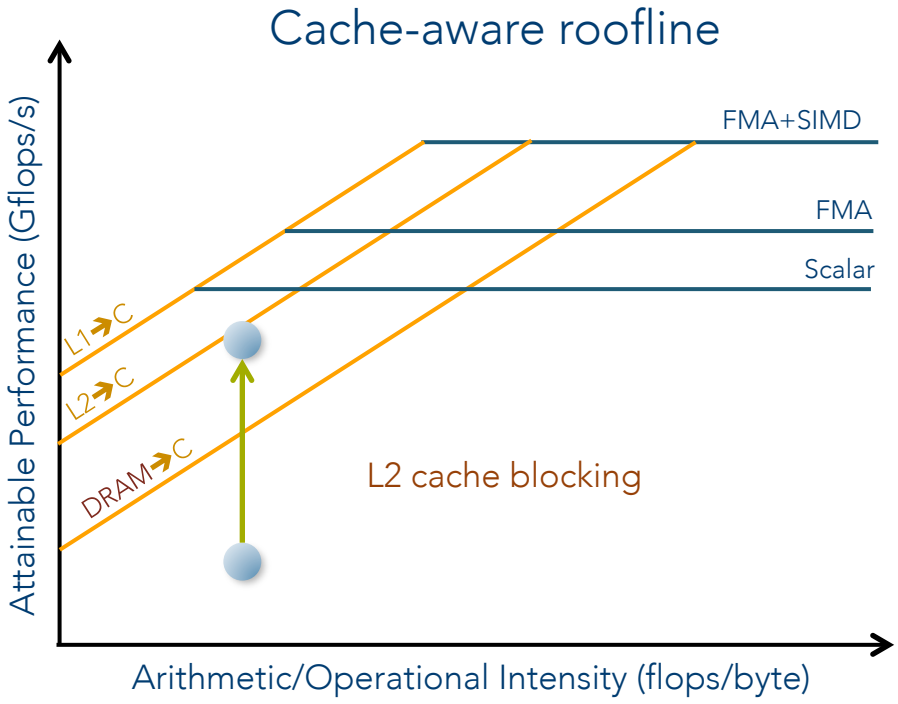




Low AI, "Stream-like", no cache reuse → We are DRAM bandwidth bound

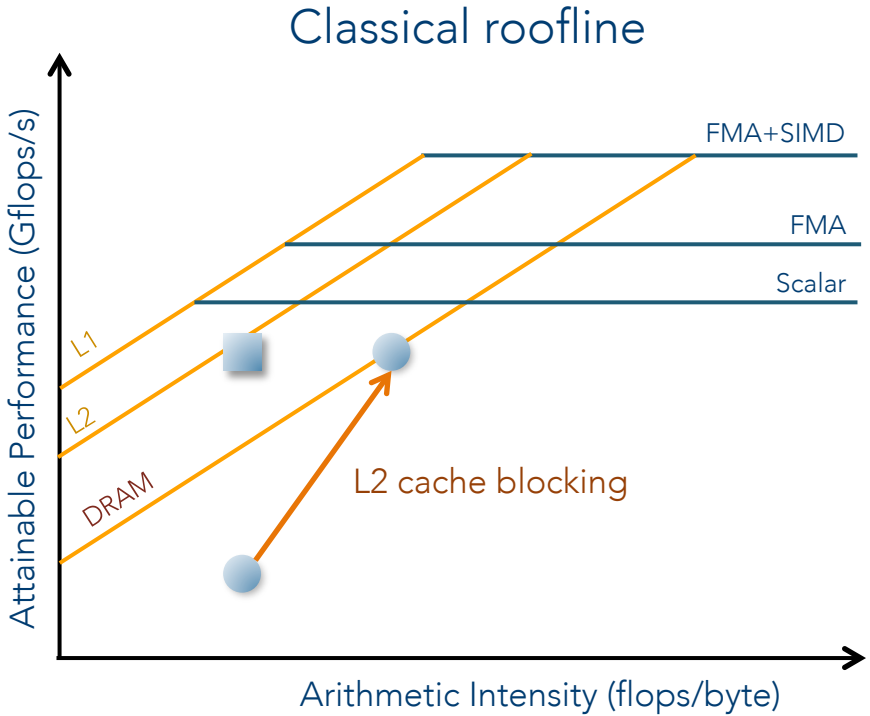
Example 1: Effect of L2 Cache Optimization

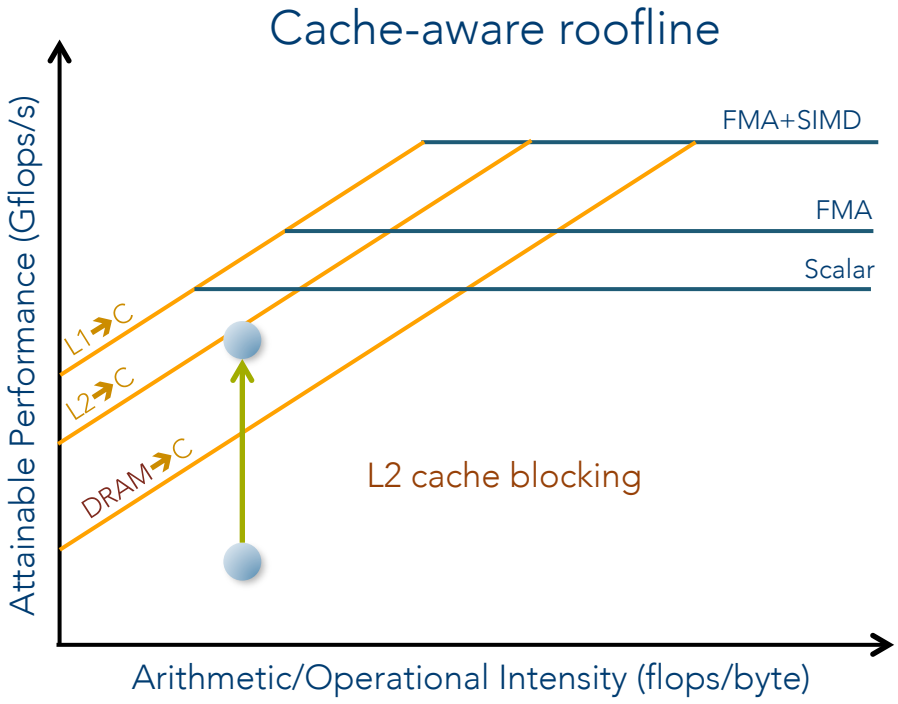




We are L2 bandwidth bound, clearly shown by the cache-aware roofline

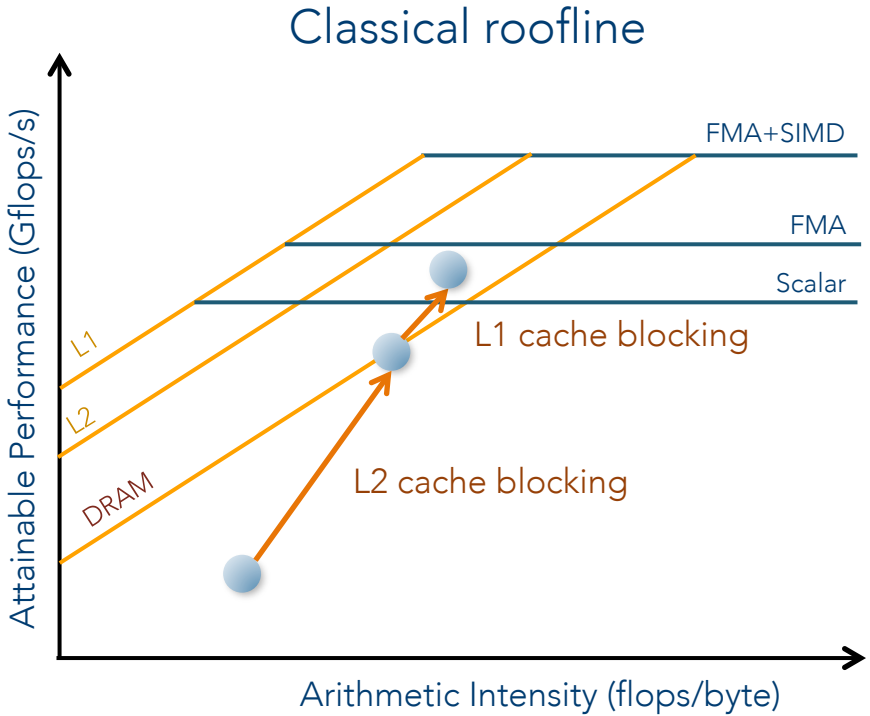
Example 1: Effect of L2 Cache Optimization

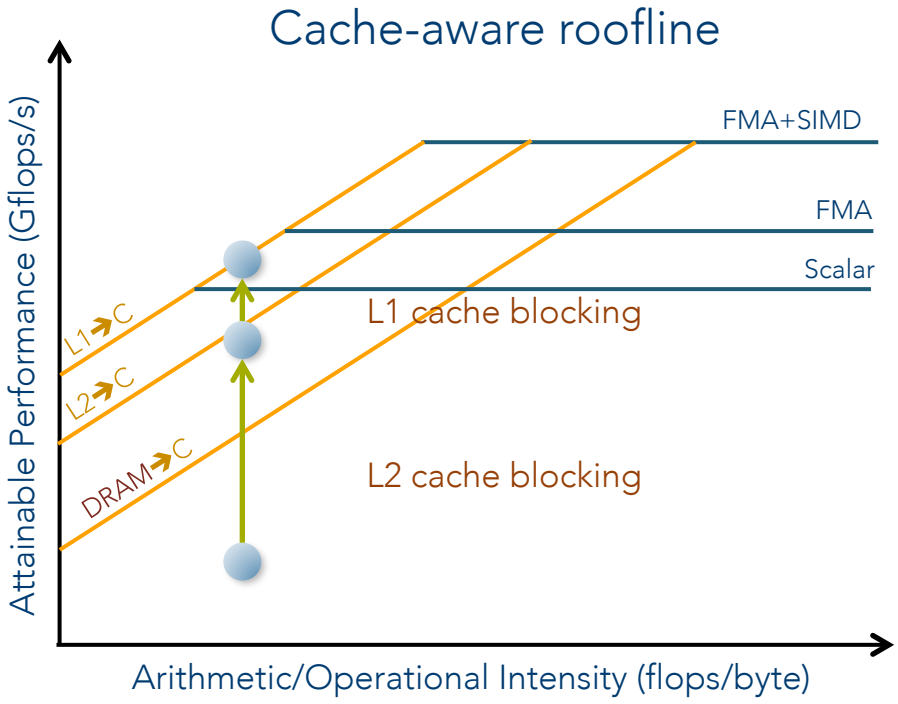




Shown by the classical roofline if we compute the L2 arithmetic intensity

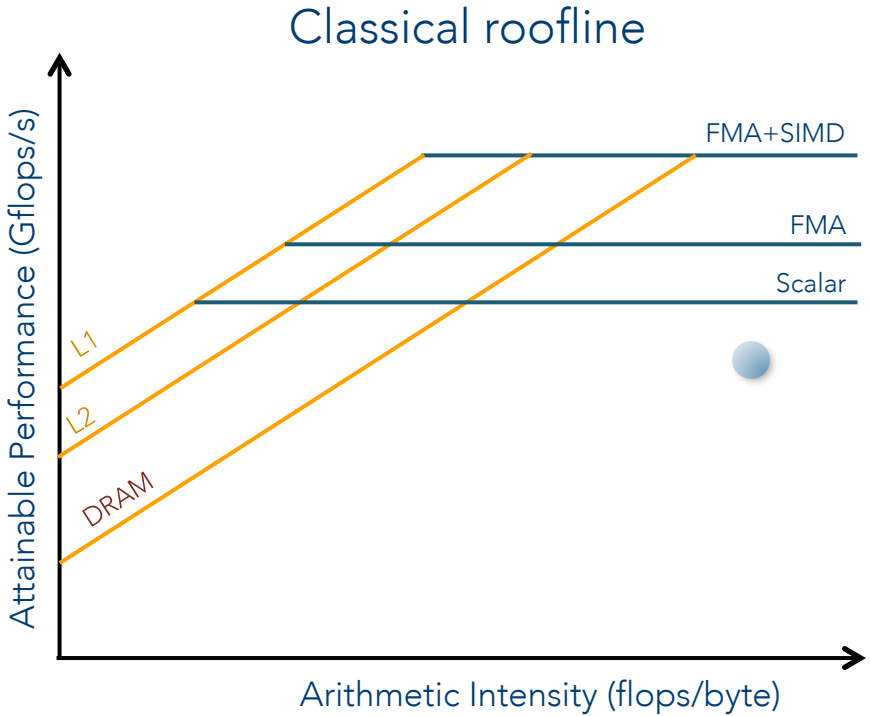
Example 1: Effect of L1 Cache Optimization

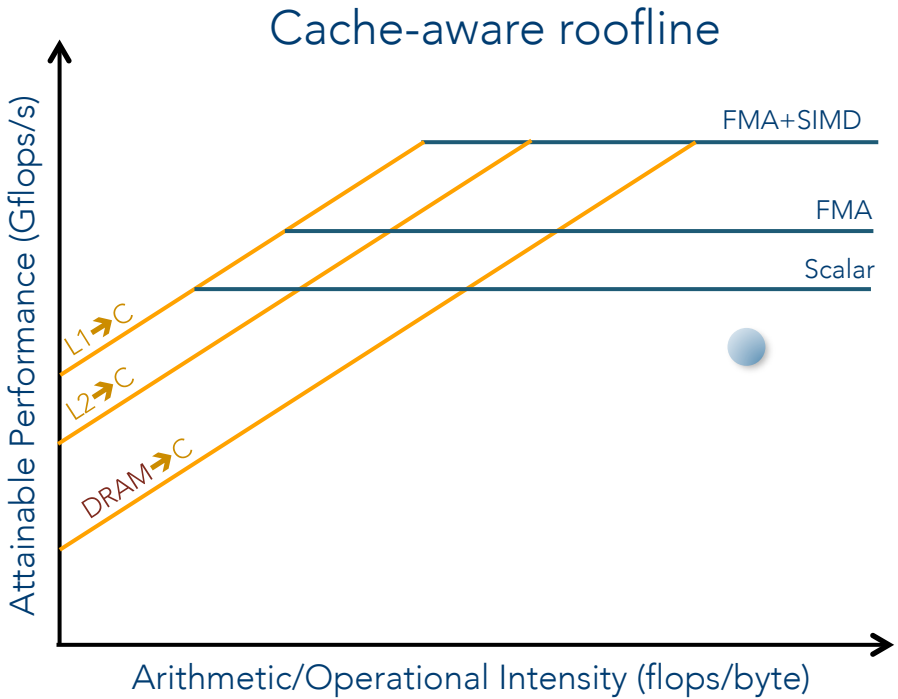




L1 or L2 cache blocking effects can directly be seen with the cache-aware roofline

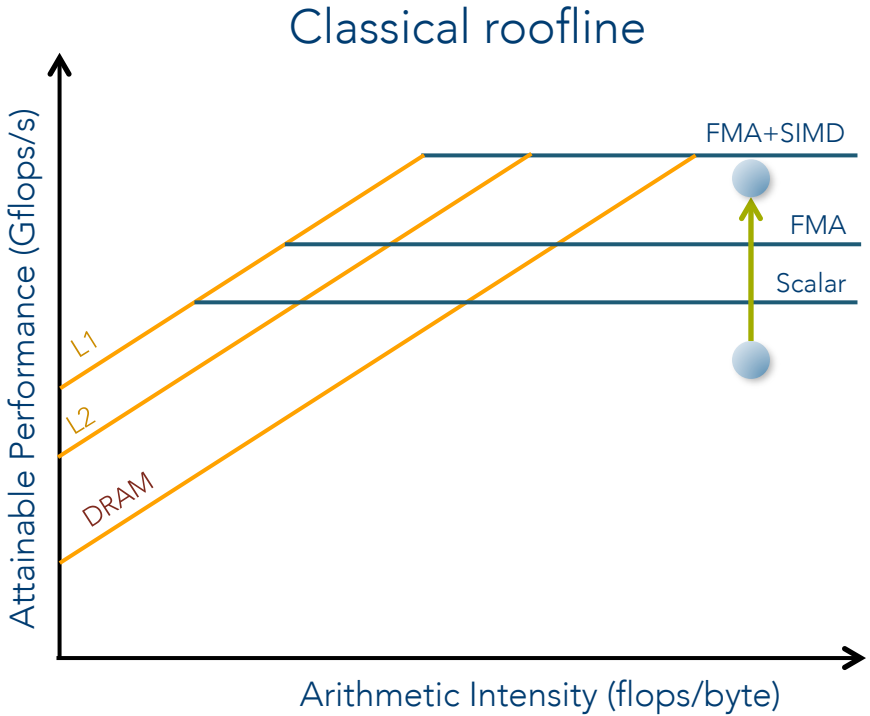
Example 2: Compute Bound Application

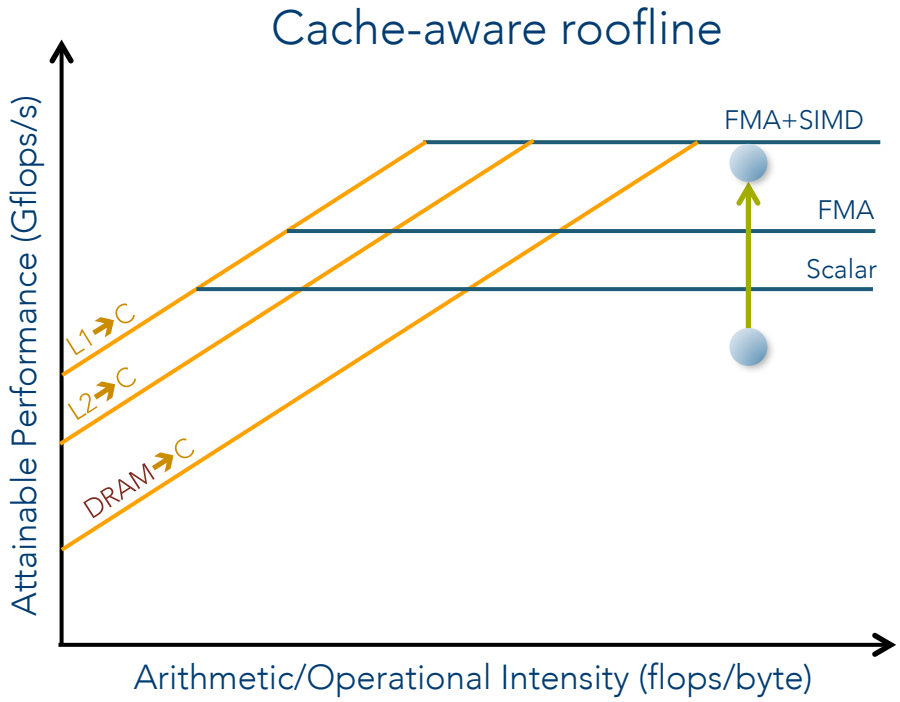




High AI: we are compute bound but not using vectorization/FMA/both VPUs

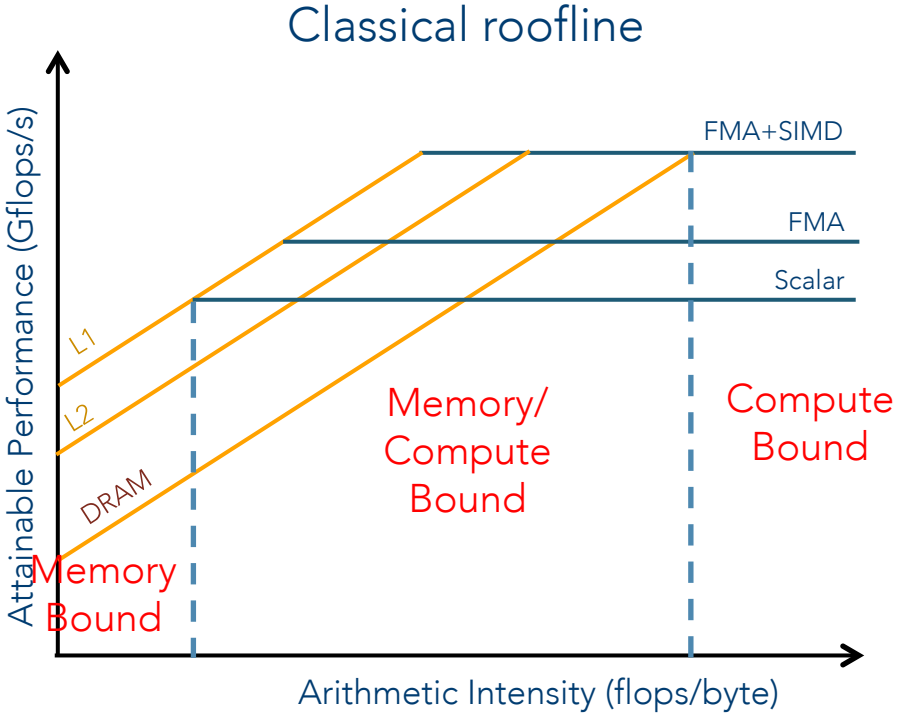
Example 2: Compute Bound Application

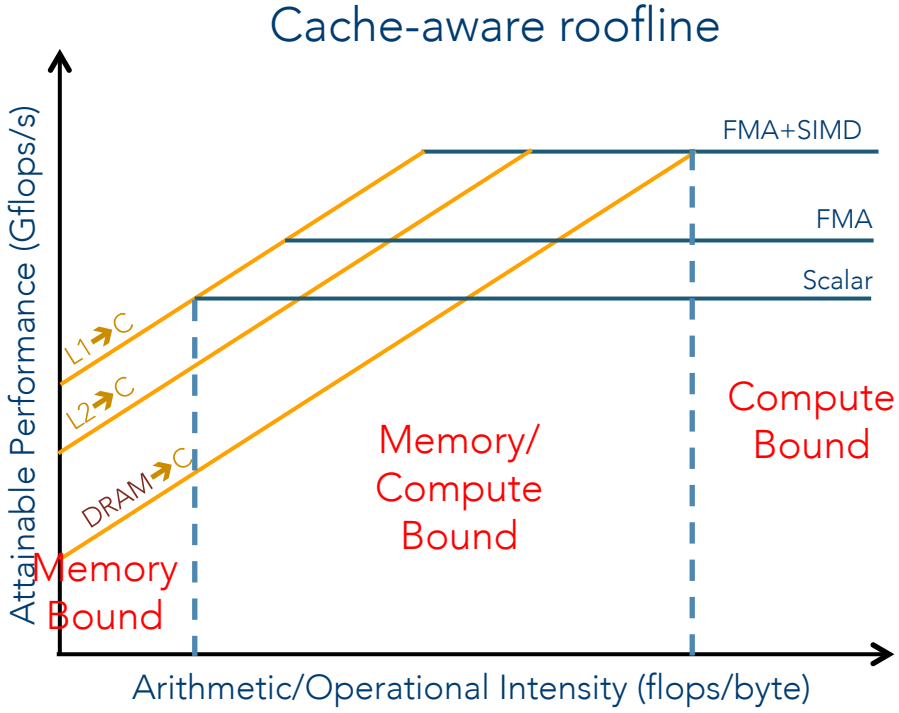




Effect of vectorization/FMA: vertical increase in both models

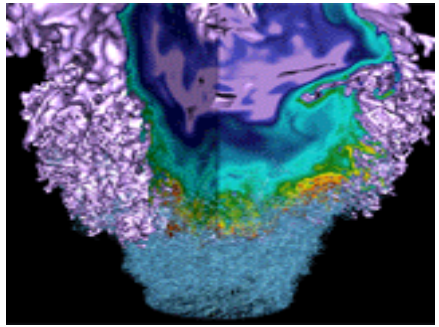
Ask "Why Here?" and "Where To?"

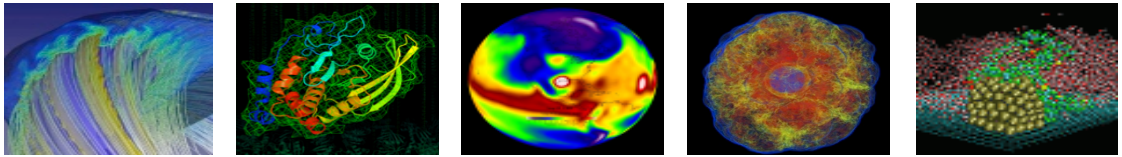




Usually, it is more complicated… You won't be on any ceiling. Or if you are, it is coincidence. BUT - asking the question, why am I not on a higher ceiling is always productive.

Part 2: Building the roofline model for your code





1. Computing the classical roofline model

2. Computing the cache-aware roofline model

The Classical Roofline Model: 3 Ingredients

1. Number of Flops

2. Number of Bytes from DRAM

3. Computation time

The LBNL method to compute the classical roofline model:
www.nersc.gov/users/application-performance/measuring-arithmetic-intensity/

The classical roofline model: computing the number of flops/s

- Run the code with the Intel Software Development Emulator toolkit (SDE) [1]

```
mpirun -n 4 sde -knl -d -iform 1 -omix my_mix.out -i -global_region -start_ssc_mark 111:repeat -stop_ssc_mark 222:repeat -- code.exe
```

- Get the number of flops by parsing all outputs using the script *parse-sde.sh* provided at www.nersc.gov/users/application-performance/measuring-arithmetic-intensity/

The classical roofline model: computing the arithmetic intensity

- Run the code with Intel Vtune in command line by doing a memory access collection

```
mpirun -n 4 amplxe-cl -start-paused –r vtune_results -collect memory-access -no-auto-finalize -trace-mpi -- code.exe
```

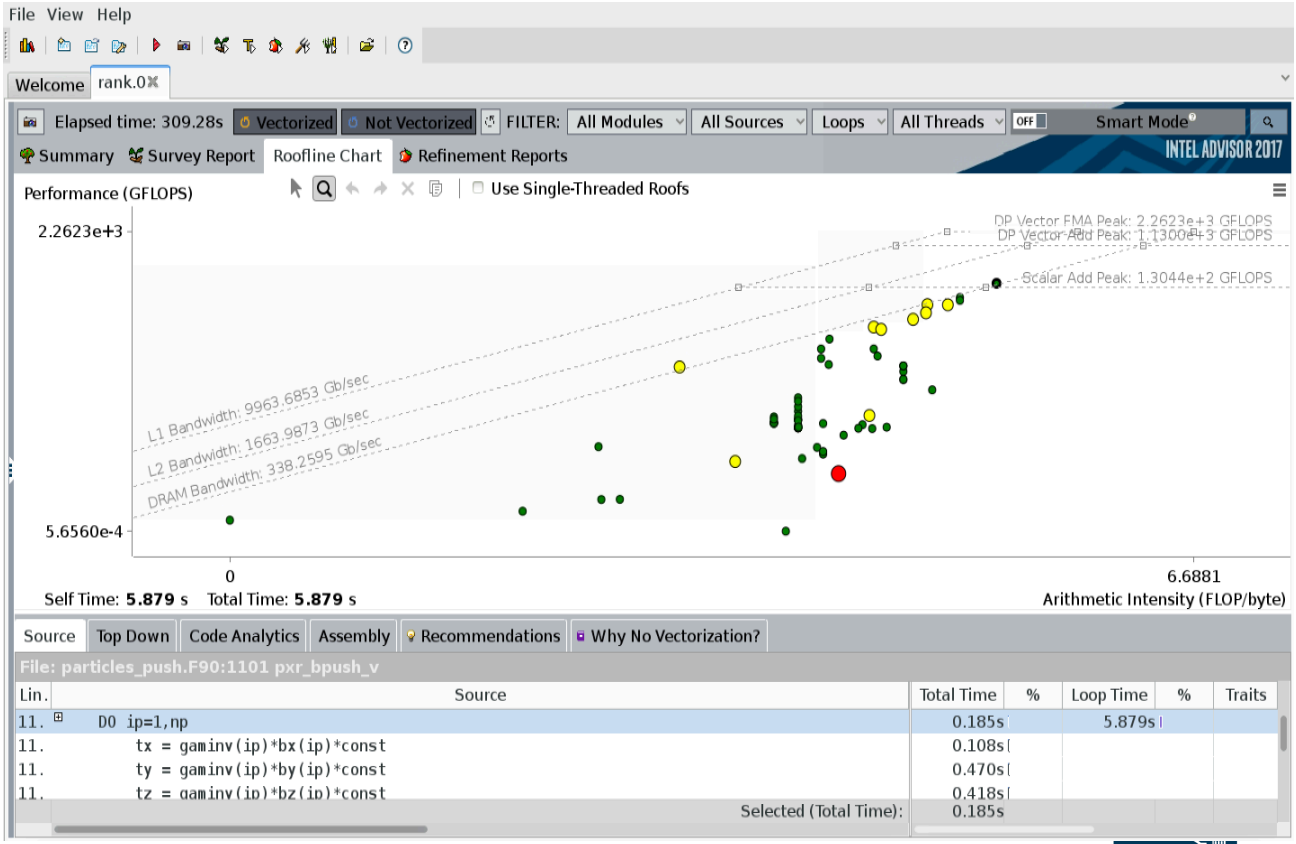
- Create a summary from the Vtune profiling results

```
amplxe-cl –report summary –r vtune_results > summary
```

- Get transferred bytes using the script parse-vtune.sh provided at www.nersc.gov/users/application-performance/measuring-arithmetic-intensity/

The cache-aware roofline model: Roofline automation in Intel Advisor :

- Computation of the roofline: u-bench-based
- #FLOPs, seconds, Bytes
- AVX-512 mask-aware
- Break-down by loops and/ or functions
- Measure L1 <-> Register traffic: what CPU demands from memory sub-system to make a computation →Cumulative traffic through L1/L2/LLC/DRAM/ MCDRAM
- GUI for quick viewing



Computing the cache-aware roofline with Intel Advisor

- Survey collection by command line with advisor

```
mpirun -n 1 advixe-cl -collect survey --trace-mpi --  ./$APPNAME
```

- Trip count collection by command line with advisor

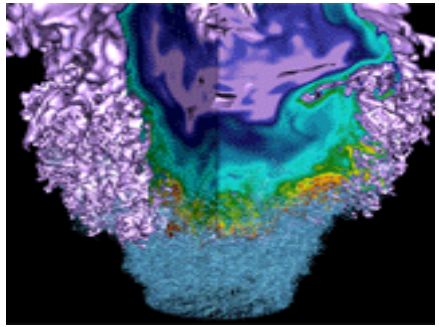
```
mpirun -n 1 advixe-cl -collect tripcounts -flops-and-masks --trace-mpi -- ./$APPNAME
```

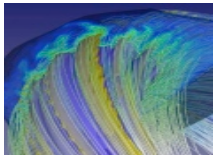
- Visualization with advixe-gui or extraction of the data in a report:

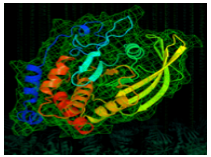
```
advixe-cl --report survey --show-all-columns --format=csv --report-output advixe_report.csv
```

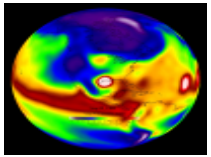
- Python library developed at LBNL used to analyze the csv reports and plots the rooflines:
  - Enable to compare different Advisor survey
  - Enable to select and sort interesting loops in order to have custom plots
  - https://github.com/tkoskela/pyAdvisor

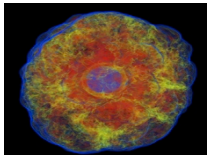
Part 3: Case Studies

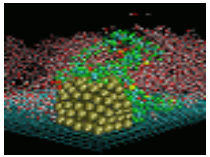












Particle-In-Cell (PIC) Code

In plasma physics:

A way to solve kinetically the intertwined collective interactions between **plasmas/charged beams** described as charged particles and **electromagnetic fields**

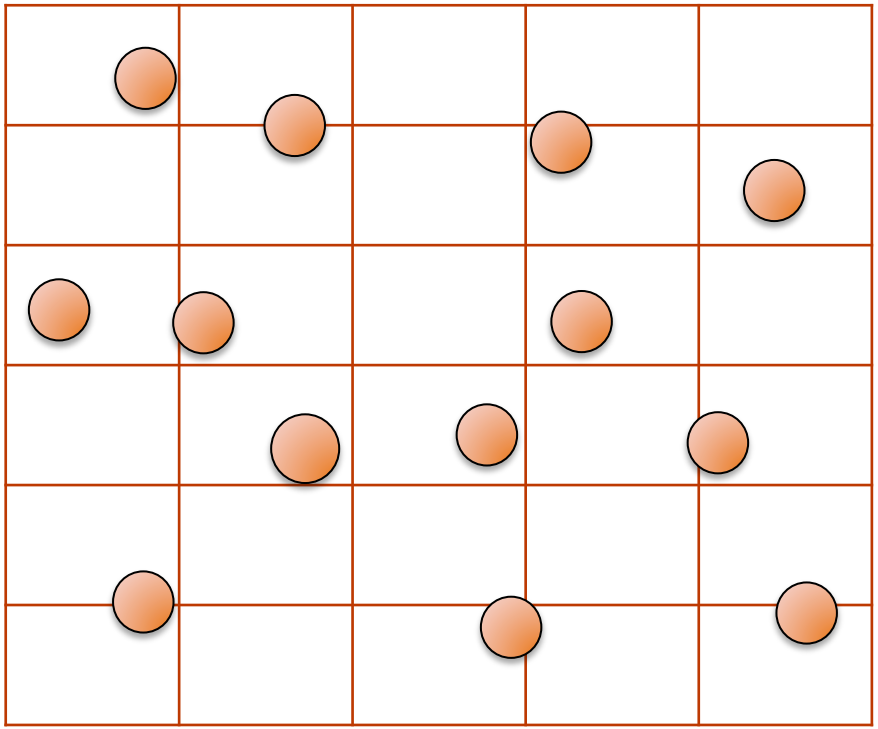
Two Kinds of Data Structure

FIELDS

MATTER/PLASMAS

Field grids for the main discretized electromagnetic field solvers

(3D arrays)

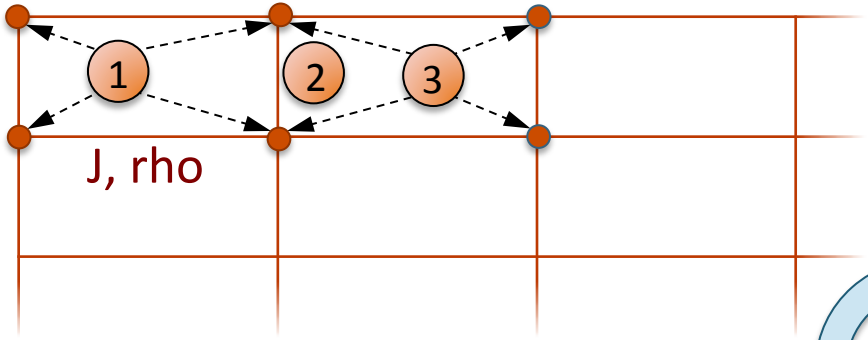


Macro-particles: Group of real particles sharing the same kinetic properties

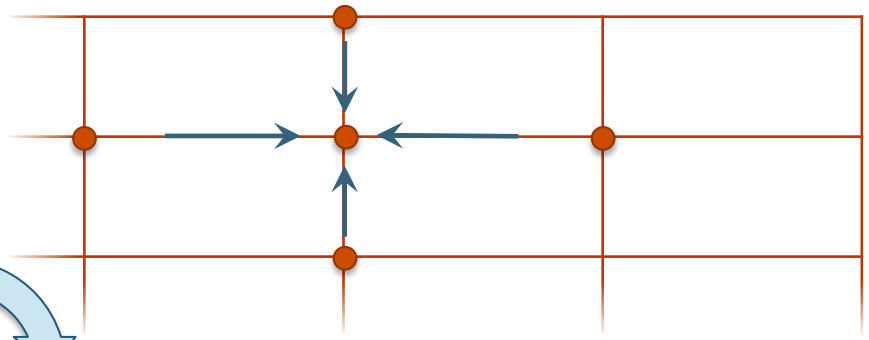
(SoA or AoS)

Four Main Steps in a PIC Loop

1) Charge/current deposition from the particles to the grids (Vectorization hotspot)

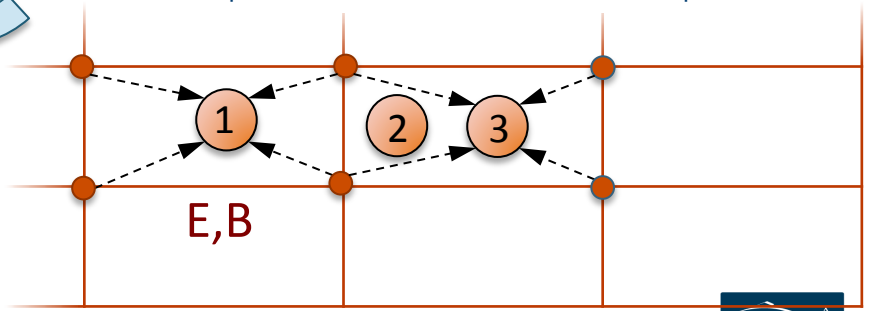


2) Maxwell solver: evolution of the field grids (Stencil problem)

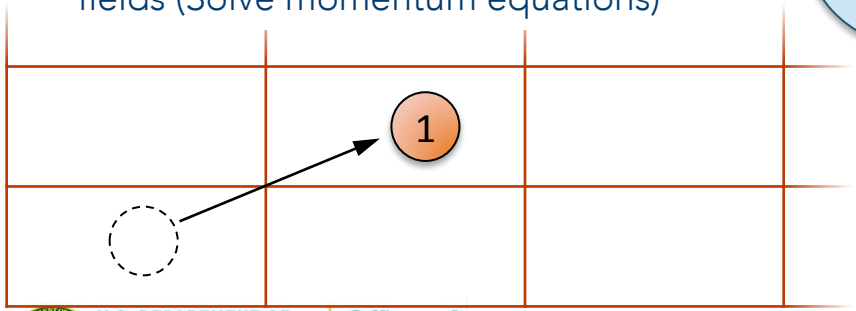


Time

3) Field gathering from the grid to the particles (Vectorization hotspot)



4) Particle pusher using interpolated fields (Solve momentum equations)



Part 3.1: First case study: PICSAR A high-performance PIC library for MIC architectures

1. Presentation
2. Implemented optimizations
3. Roofline plots

PICSAR: a high-performance Particle-In-Cell library optimized for Many-Integrated Core Architectures

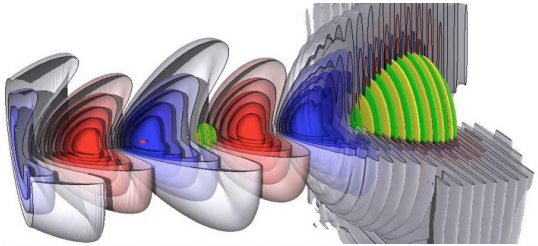
- Designed to be interfaced with the existing PIC code WARP [1] developed at LBNL and LLNL

- Provides high-performance PIC routines to the community (soon release as an Open-Source project)

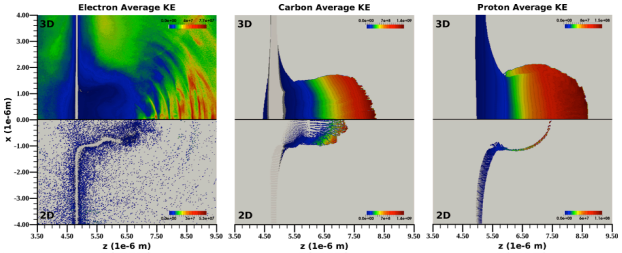
PICSAR: a high-performance Particle-In-Cell library optimized for Many-Integrated Core Architectures

Application domains:

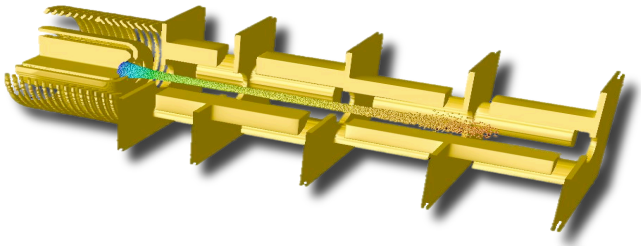
- Plasma physics,
- Laser-matter interaction
- Conventional particle accelerators



WARP Simulation: electron acceleration in the laser wakefield acceleration regime



WARP Simulation: Ion acceleration in the interaction with a thin foil



WARP Simulation: Conventional beam accelerators

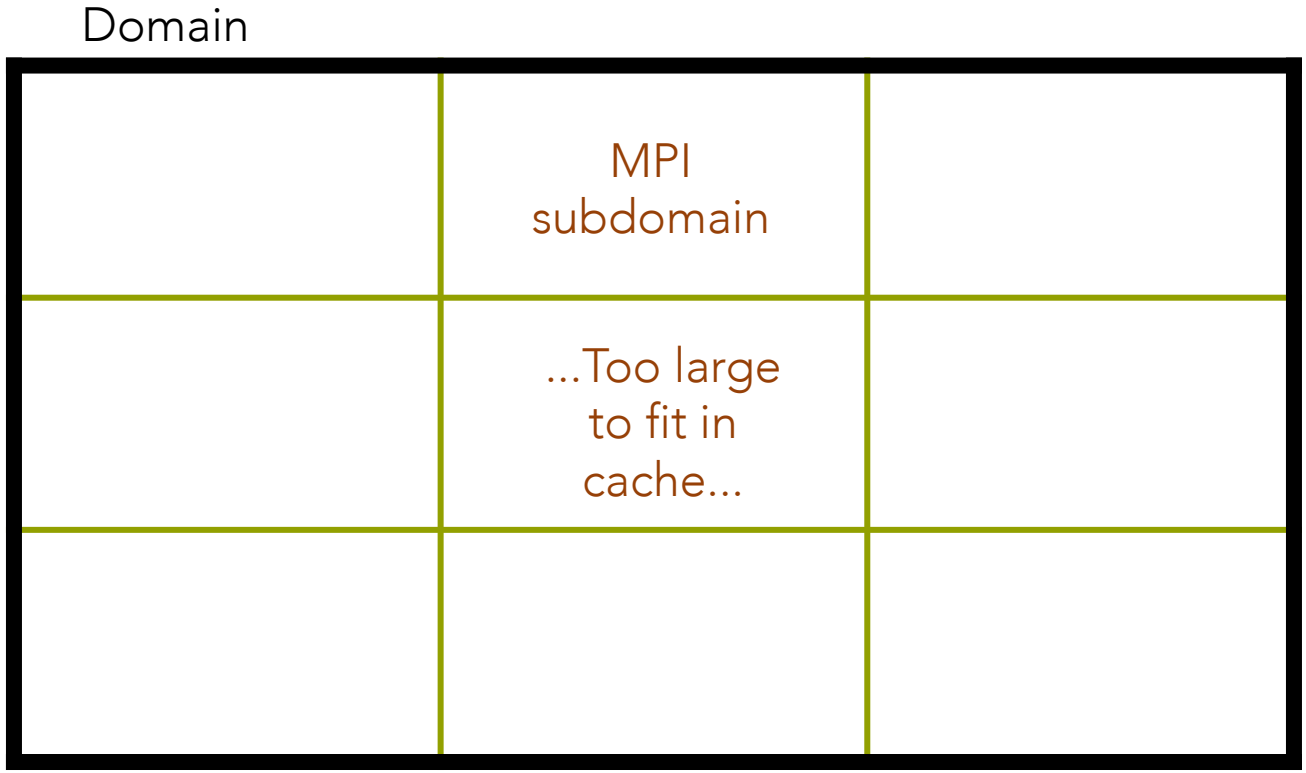
[1] WARP website: warp.lbl.gov/

PICSAR Implemented Optimizations

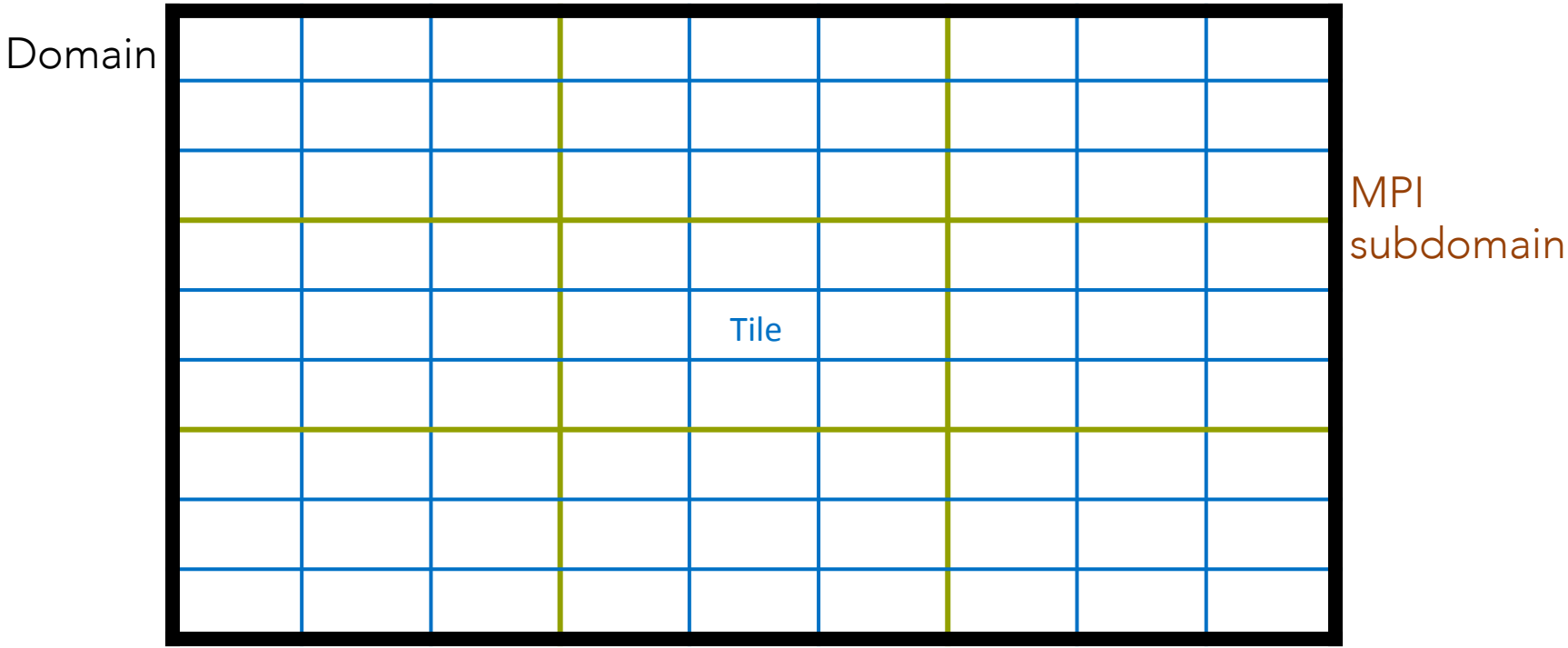
- L2 field cache-blocking: MPI domain decomposition into tiles
- Hybrid parallelization: OpenMP to handle tiles (inner-node parallelism)
- New data structures to enable efficient vectorization (current/charge deposition)
- An efficient parallel particle exchange algorithm between tiles
- Parallel optimized pseudo spectral Maxwell solver
- Particle sorting algorithm (memory locality)

[1] WARP website: warp.lbl.gov/

Space Decomposition Into MPI Domains



L2 Cache-blocking: new decomposition into tiles [1]

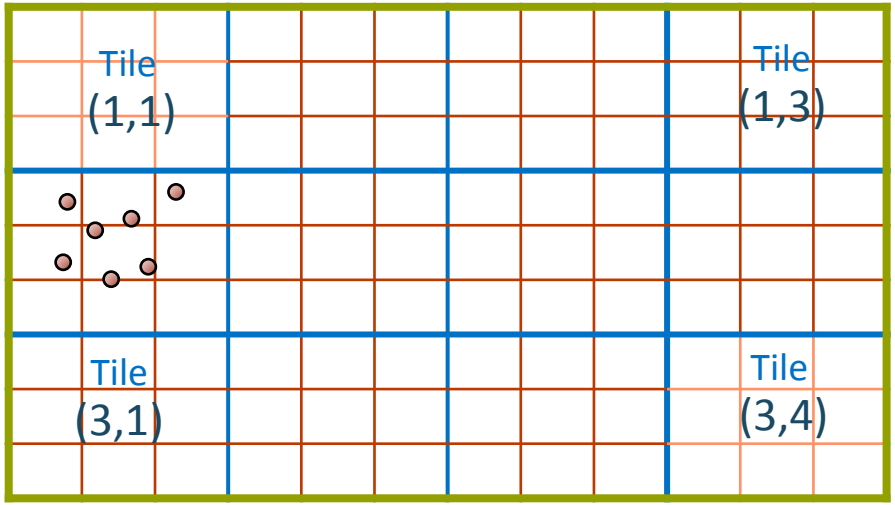


[1] H. Vincenti et al, ArXiv 1601.02056 (2016)

L2 Cache-blocking: new decomposition into tiles [1]

Zoom in a MPI subdomain

Portion of the grid local to the tile



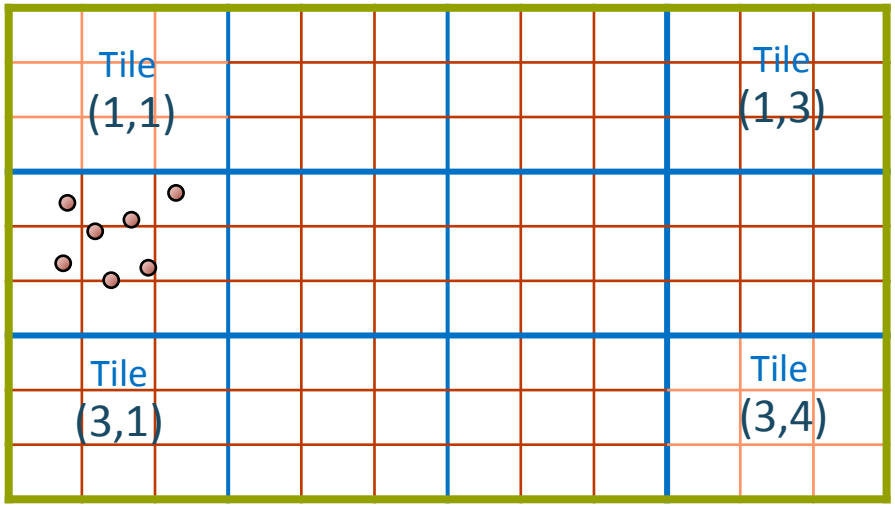
- New subdivision into tile inside MPI subdomains (tiling): **local field grids + guard cells from the global grids**, local particle property arrays

[1] H. Vincenti et al, ArXiv 1601.02056 (2016)

L2 Cache-blocking: new decomposition into tiles (tiling)

Zoom in a MPI subdomain

Portion of the grid local to the tile



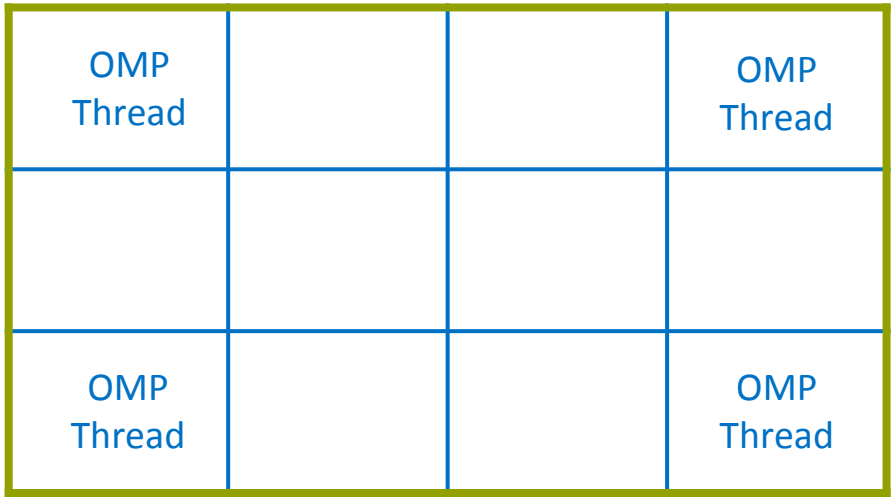
- Tile size:
  - field grids can fit in L2 cache (main constraint)
  - Particle arrays can partially or entirely fit in L3 on Haswell

Better memory locality and cache reuse.

[1] H. Vincenti et al, ArXiv 1601.02056 (2016)

Hybrid parallelization using OpenMP [1]

Zoom in a MPI subdomain



Tiles are handled by OpenMP

- Number of tiles >> number of OpenMP threads = load balancing between the tiles using a dynamic scheduler

[1] H. Vincenti et al, ArXiv 1601.02056 (2016)

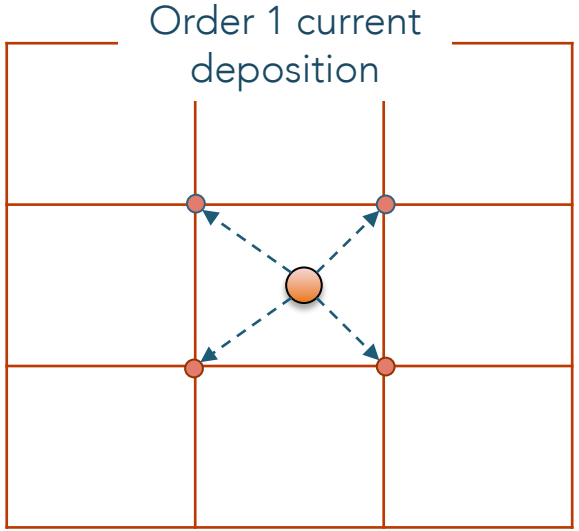
Vectorization issue of the classical current deposition algorithm

Loop on all particles:

1) Determine nearby nodes on the current grid

2) Compute the current generated by each particle

3) Deposit contributions on the nodes



Vectorization issue of the classical current deposition algorithm

Loop on all particles:

1) Determine nearby nodes on the current grid

2) Compute the current generated by each particle

3) Deposit contributions on the nodes

Can be vectorized despite roundings and type conversions

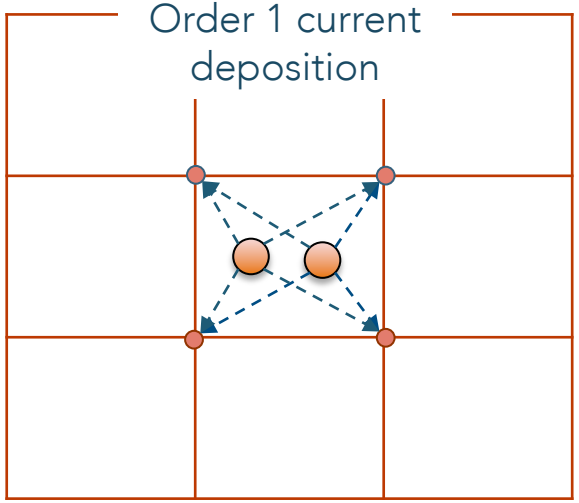
Vectorization issue of the classical current deposition algorithm

Loop on all particles:

1) Determine nearby nodes on the current grid

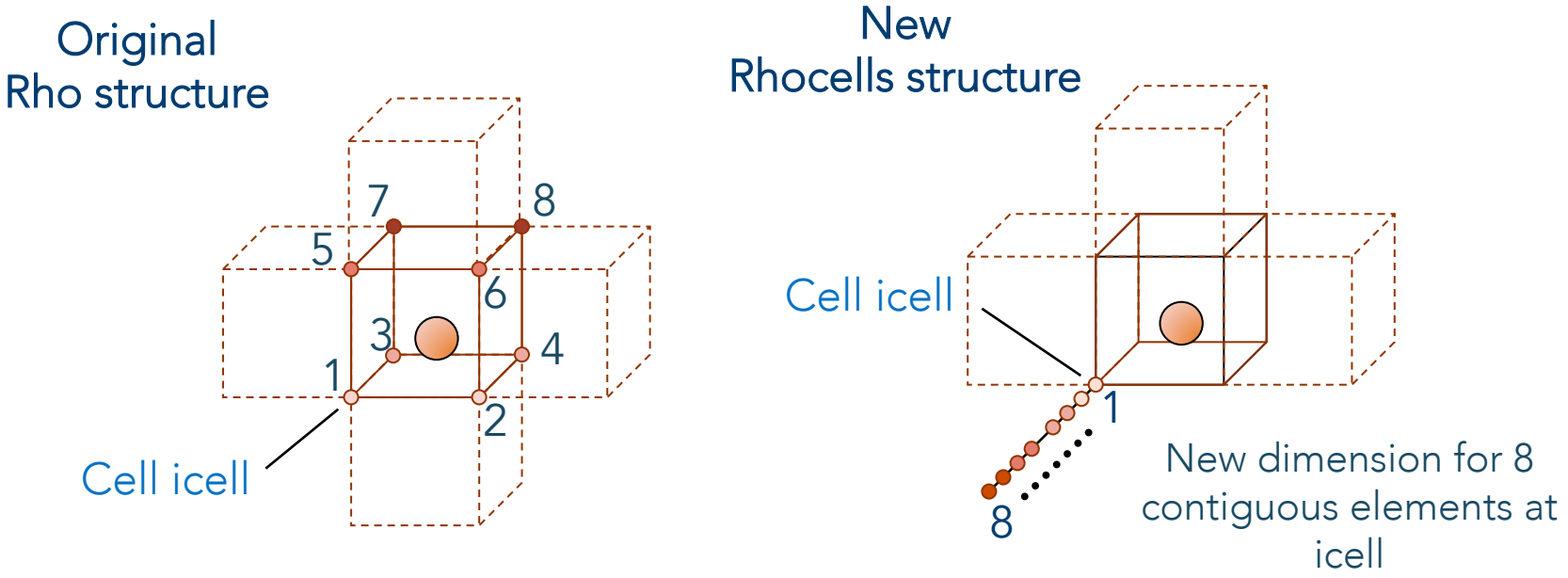
2) Compute the current generated by each particle

3) Deposit contributions on the nodes



prevents vectorization due to memory races when 2 particles are in the same cell

New data structure to enable vectorization of the current deposition [1]

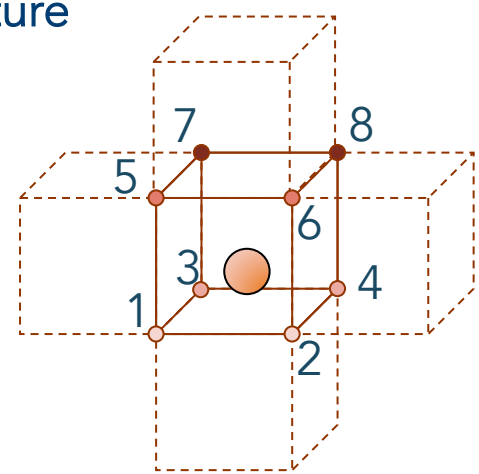


- New dimension in the current arrays to access vertices of a cell in a contiguous way

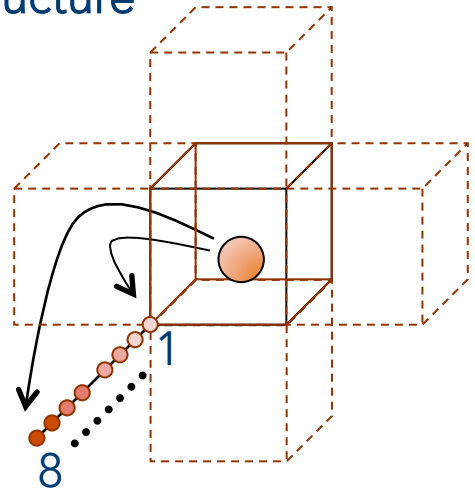
[1] H. Vincenti et al, ArXiv 1601.02056 (2016)

New data structure to enable vectorization of the current deposition [1]

Original
Rho structure



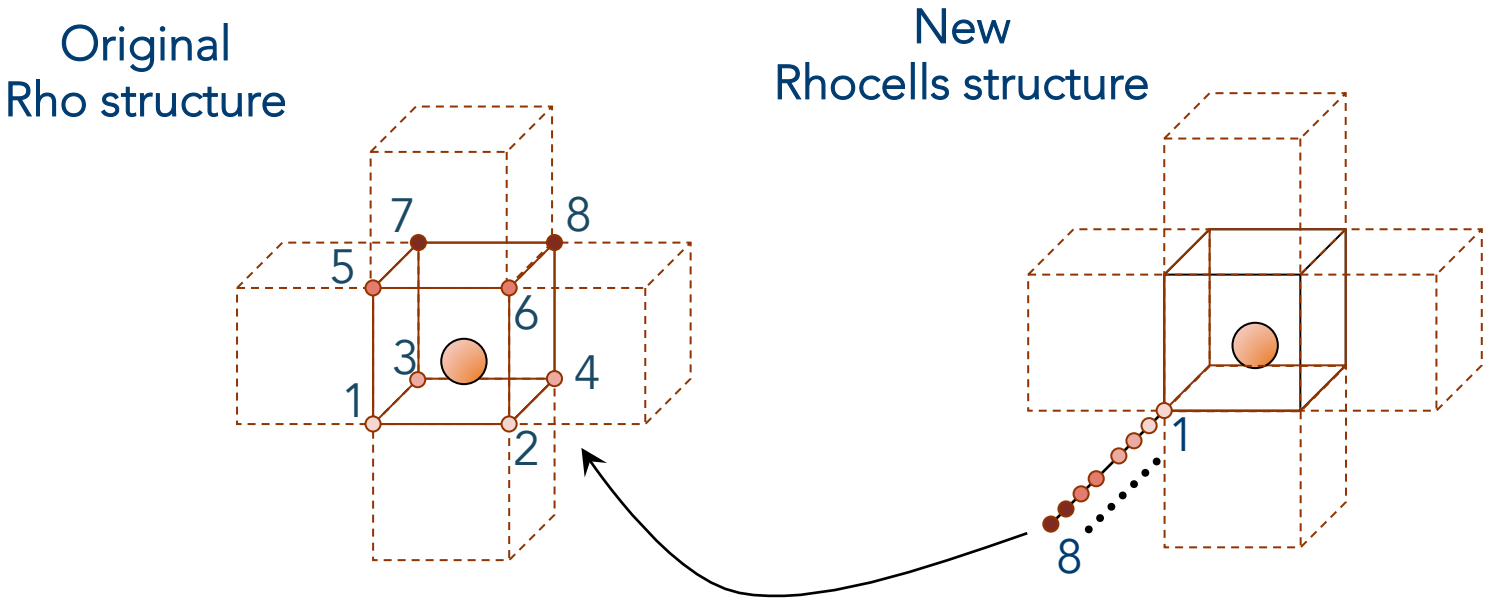
New
Rhocells structure



- Vectorization directly applied on the deposition process itself, not the particles

[1] H. Vincenti et al, ArXiv 1601.02056 (2016)

New data structure to enable vectorization of the current deposition [1]



- Reduction into the original data structure
- Low complexity O(Ncells) with Ncells << Nparticles => Negligible time

[1] H. Vincenti et al, ArXiv 1601.02056 (2016)

Application of the roofline model to 3 configurations

Homogeneous thermalized plasma (load balanced)
Domain of 100x100x100 cells
2 species, 40 particles per cell
Fit in MCDRAM

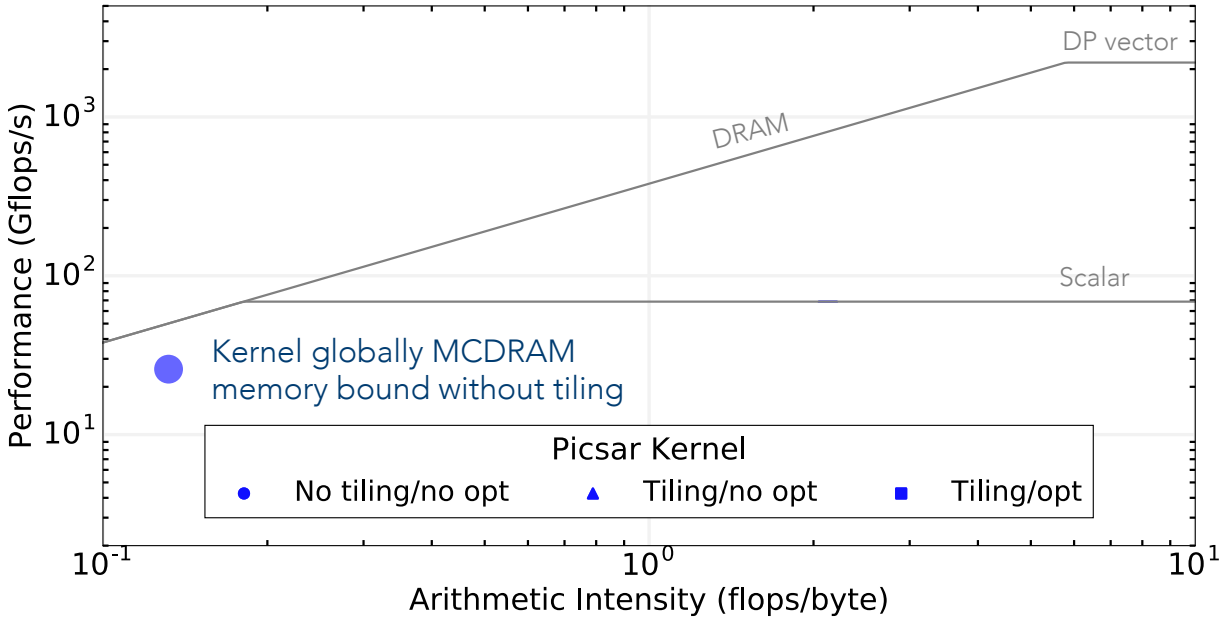
Configuration 1: No Tiling and no vectorization

Configuration 2: Tiling (L2 cache blocking) and no vectorization

Configuration 3: Tiling (L2 cache blocking) and vectorization

Classical Roofline Model applied to the "non-optimized" kernel

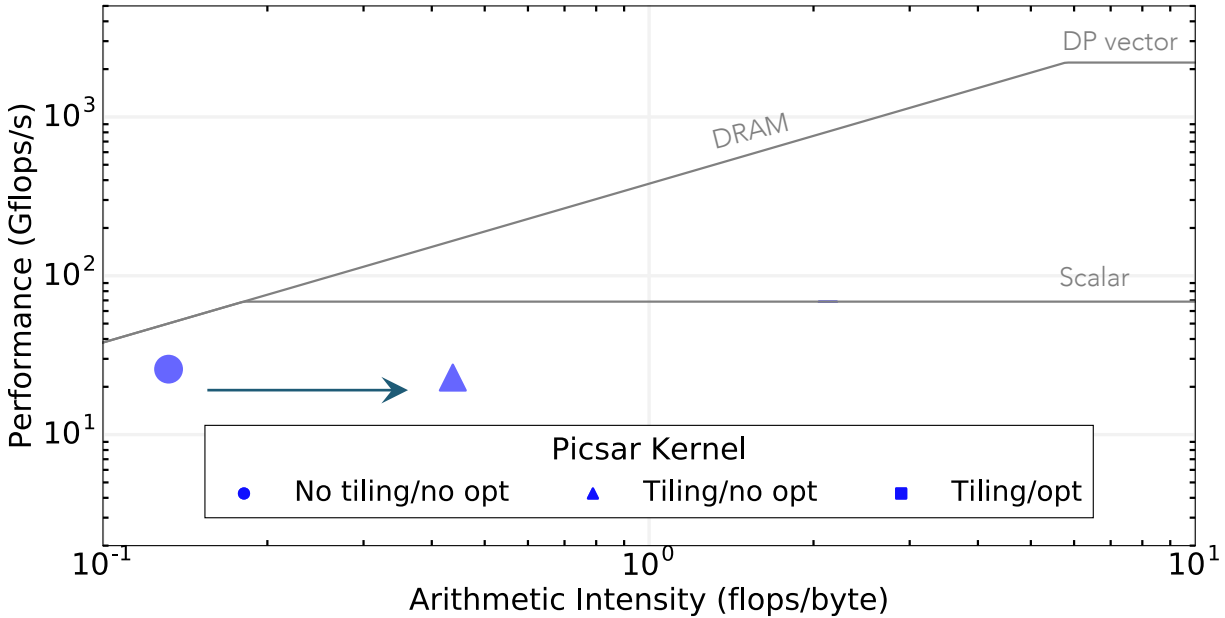
- KNL SNC4 flat mode, 64 MPI tasks
- 1 Tile per MPI task without vectorization



Marker sizes proportional to the simulation time

Classical Roofline Model applied to the kernel with tiling

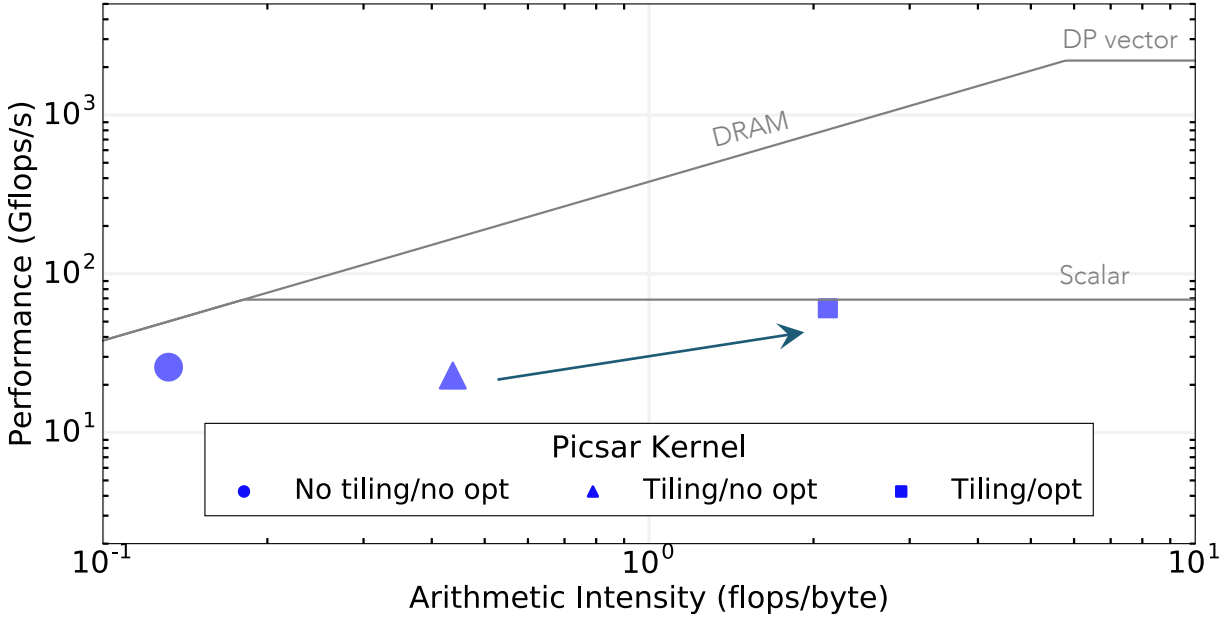
- KNL SNC4 flat mode, 4 MPI tasks, 32 OPM threads
- 432 tiles per MPI task without vectorization



Diminishing of the number of bytes loaded from DRAM by 2.3 (cache reuse)

Classical Roofline Model applied to the fully optimized kernel

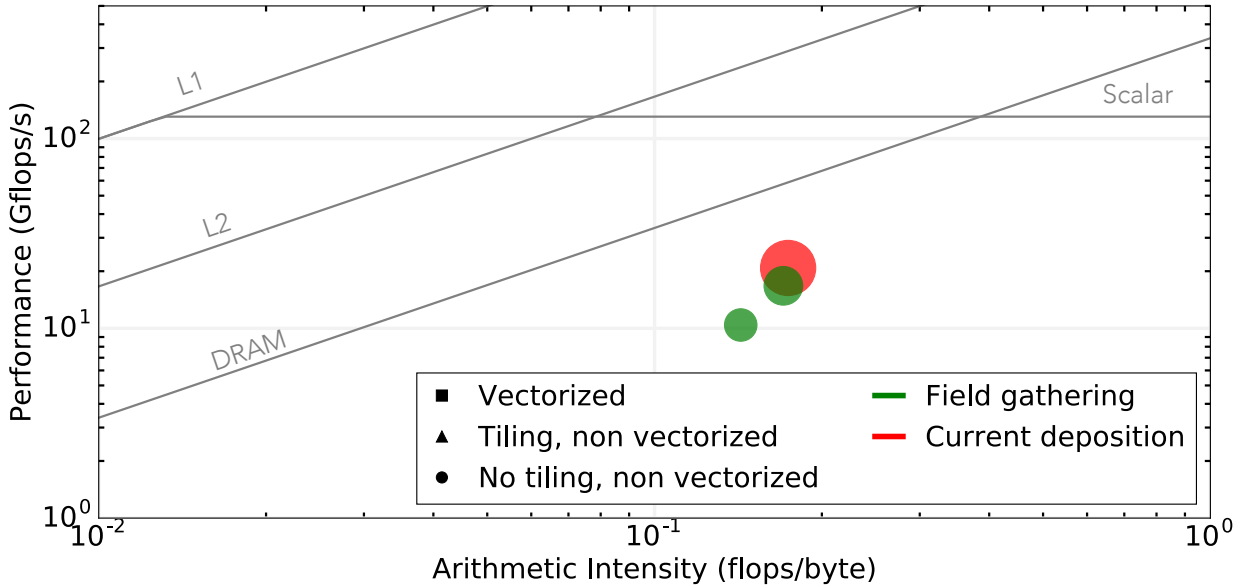
- KNL SNC4 flat mode, 4 MPI tasks, 32 OPM threads
- 432 tiles per MPI task without vectorization



Diminish the computational time (/2.3) and the flops/byte ratio (/3.2) with other memory optimizations

Cache-aware roofline model (Intel Advisor) on the non-optimized loops

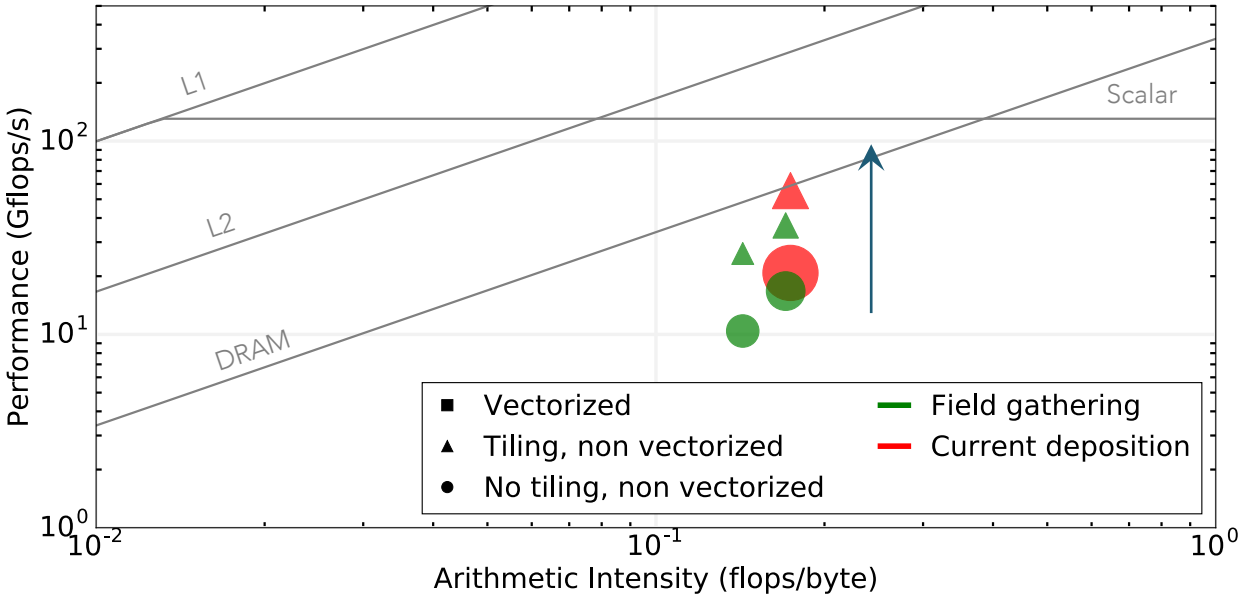
- Quadrant cache mode, 1 MPI, 64 OMP threads
- 1 tile per OMP thread without vectorization



Main loops in field gathering and current deposition are DRAM memory bound

Cache-aware roofline model (Intel Advisor) on the current deposition and field gathering loops with tiling

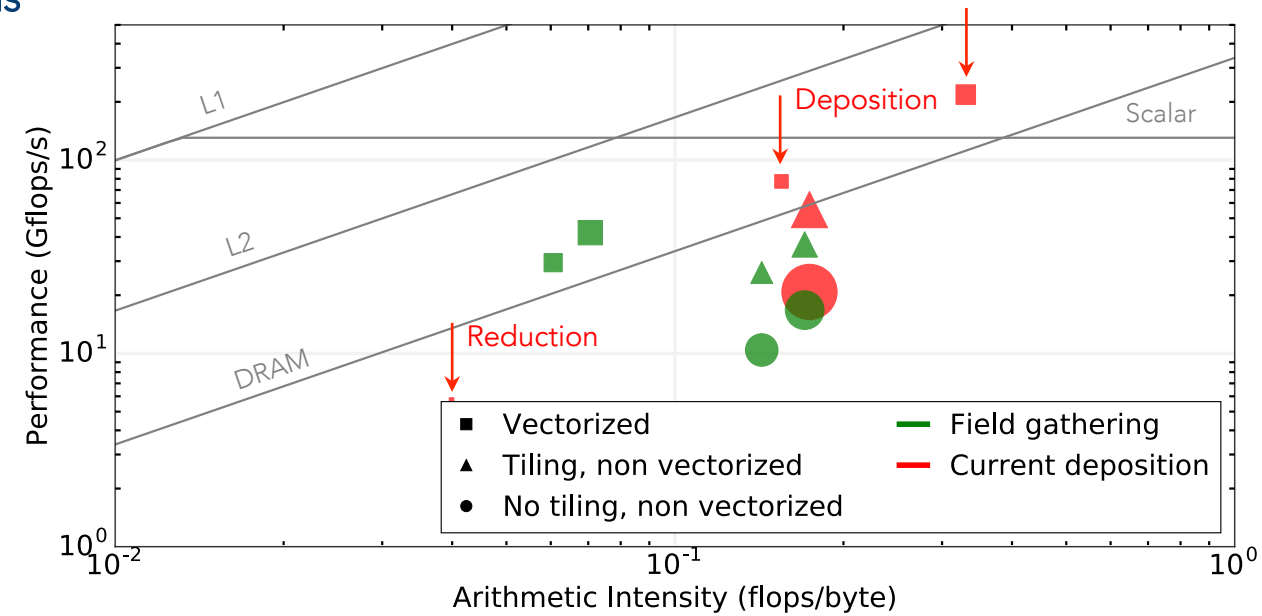
- Quadrant cache mode, 1 MPI, 64 OMP threads
- 27 tiles per OMP thread without vectorization



Diminish the computational time of each loop. Not sufficient to be L2 bound ?

Cache-aware roofline model (Intel Advisor) on the current deposition and field gathering vectorized loops

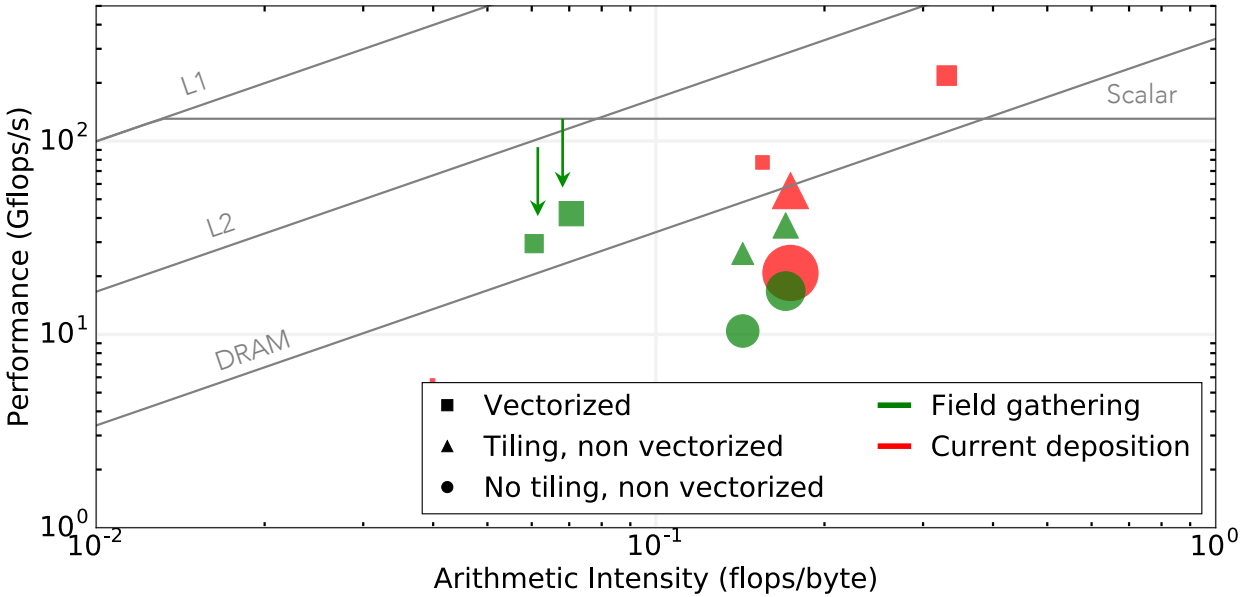
- Quadrant cache mode, 1 MPI, 64 OMP threads
- 27 tiles per OMP thread without vectorization



The vectorized subroutine for the current deposition is divided into 3 vectorized loops that uses the new data structure

Cache-aware roofline model (Intel Advisor) on the current deposition and field gathering vectorized loops

- Quadrant cache mode, 1 MPI, 64 OMP threads



The field gathering vectorized subroutine exhibits 2 vectorized loops with a lower simulation time but a lower arithmetic intensity.

Conclusions for PICSAR

The classical roofline and the cache-aware roofline models using Intel Advisor bring a new vision of the optimization efforts.

PICSAR, even optimized, is still mainly memory bound and far from the peak performance:

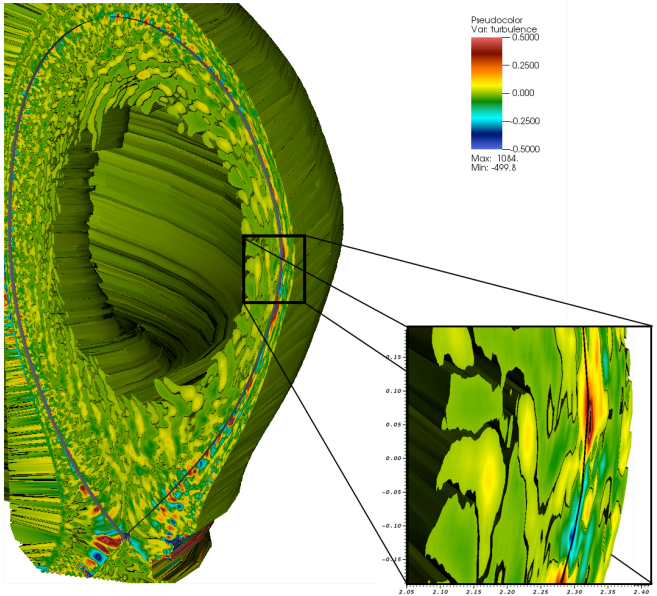
- No efficient cache strategy for the particles (No L3 on KNL), No L1 optimization
- Vectorization efficiency not at 100%: gather/scatter, register spilling, no FMA, type mixing

Is the roof attainable?

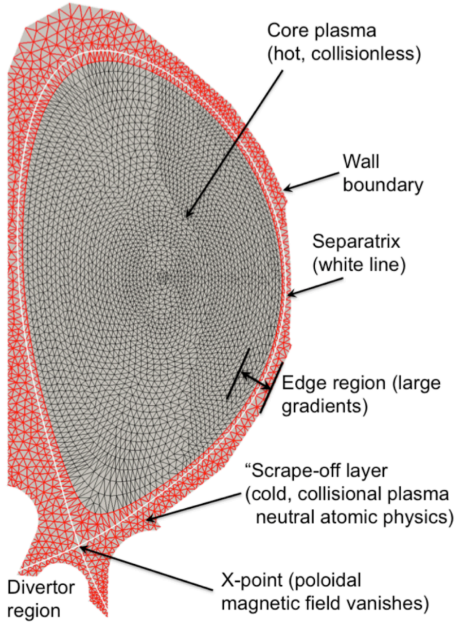
Part 3.2: Second case study: XGC1 A Magnetic Fusion PIC code

1. Introduction
2. Main Bottlenecks and Optimizations
3. Roofline Analysis

XGC1 is a PIC Code for Tokamak (Edge) Fusion Plasmas



XGC1 Simulation of edge turbulence in the DIII-D tokamak

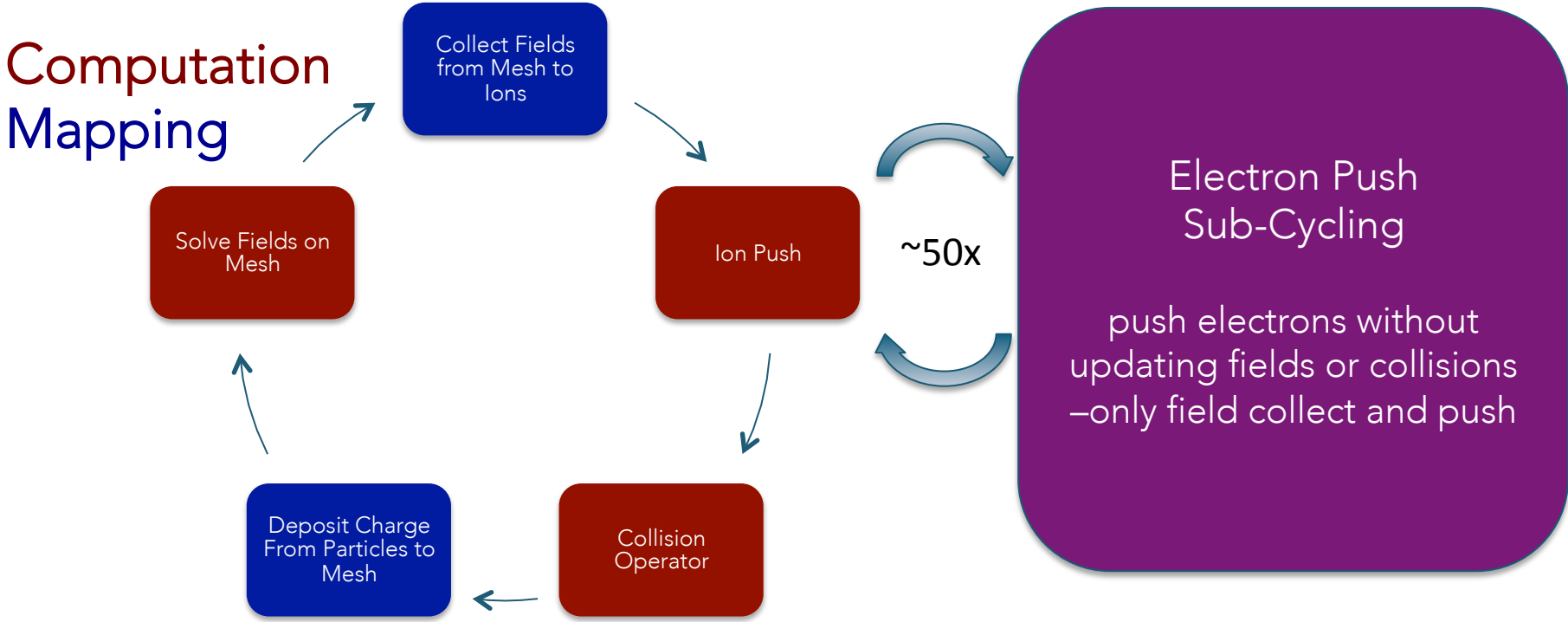


Unstructured field-aligned mesh in a poloidal domain

Different PIC Application → Different Optimization Challenges

- Complicated Geometry
  - Unstructured mesh in 2D (poloidal) plane(s)
  - Nontrivial field-following (toroidal) mapping between meshes
  - Typical simulation has 1 000 particles per cell, 100 000 cells per domain, 32 toroidal domains.
- Gyrokinetic Equation of Motion
  - + 6D to 5D problem
  - + O(100) longer time steps
  - -- Higher (2^{nd}) order derivative terms in EoM
  - -- Averaging scheme in field gather
- Electron Sub-Cycling

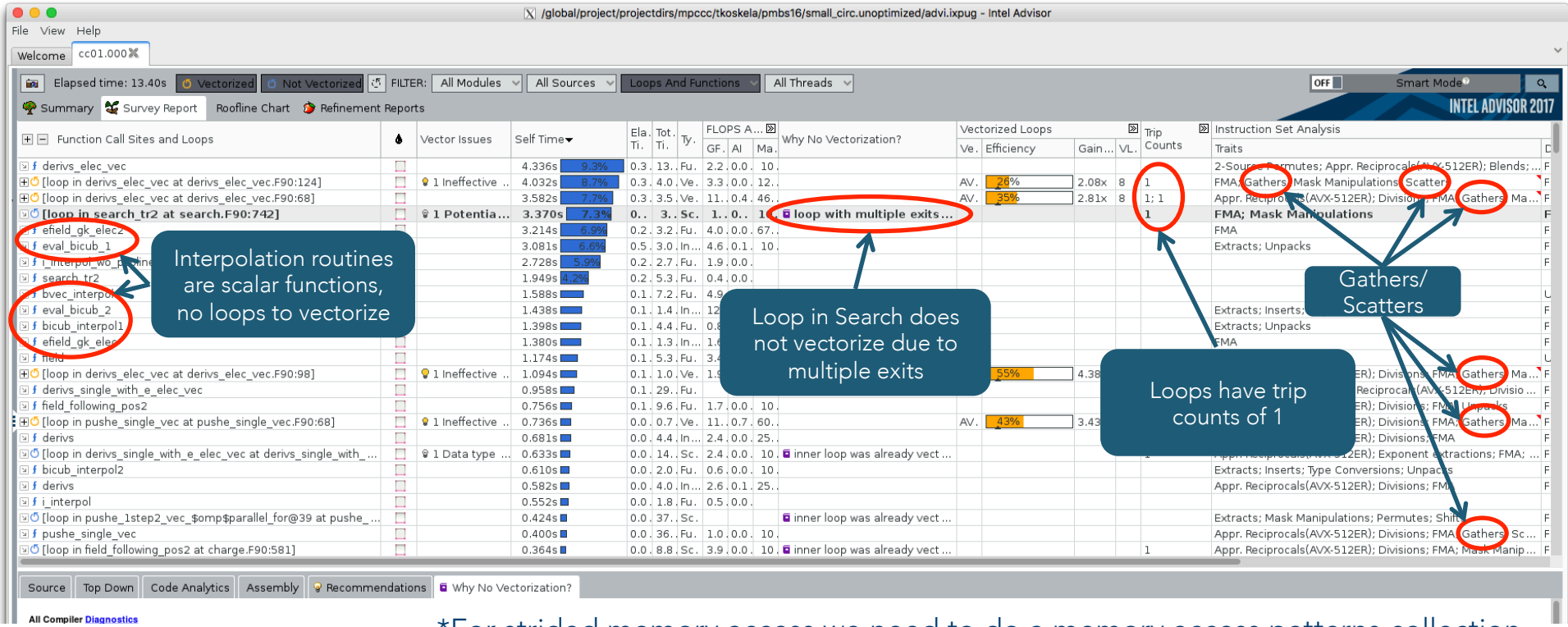
Most of The Computation Time in XGC1 is Spent in Electron Sub-Cycling



Main Bottlenecks in Electron Push

- Field Interpolation to Particle Position in Field Gather
  - Indirect grid access produces gather/scatter instructions
  - Inner loops with short trip counts prevent vectorization
- Element Search on the Unstructured Mesh after Push
  - Indirect grid access produces gather/scatter instructions
  - Multiple exit conditions prevent vectorization
- Computation of High-Order Terms in Equations of Motion in Push
  - Strided memory access in complicated derived data types
  - Cache unfriendly

A Single Advisor Survey Discovers Most* Of Them

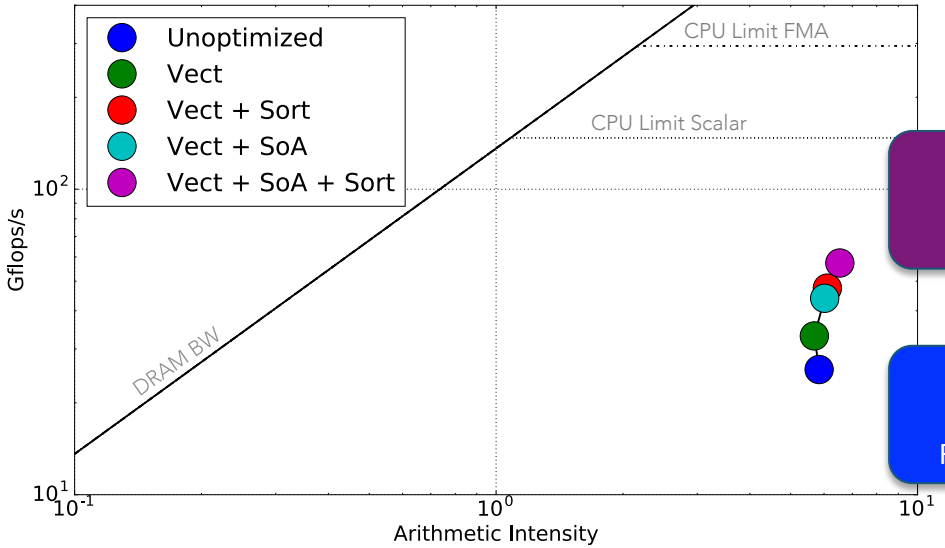


*For strided memory access we need to do a memory access patterns collection

Main Optimizations in Electron Push

- Enabling Vectorization
  - Insert loops over blocks of particles inside short trip count loops
  - Sort particles to reduce random memory accesses
- Data Structure Reordering
  - Store field and particle data in SoA(oS) format.
  - SoAoS best when accessing multiple components with a gather instruction
- Algorithmic Improvements
  - Reduce number of unnecessary calls to the search routine
  - Sort particles by the element index instead of local coordinates

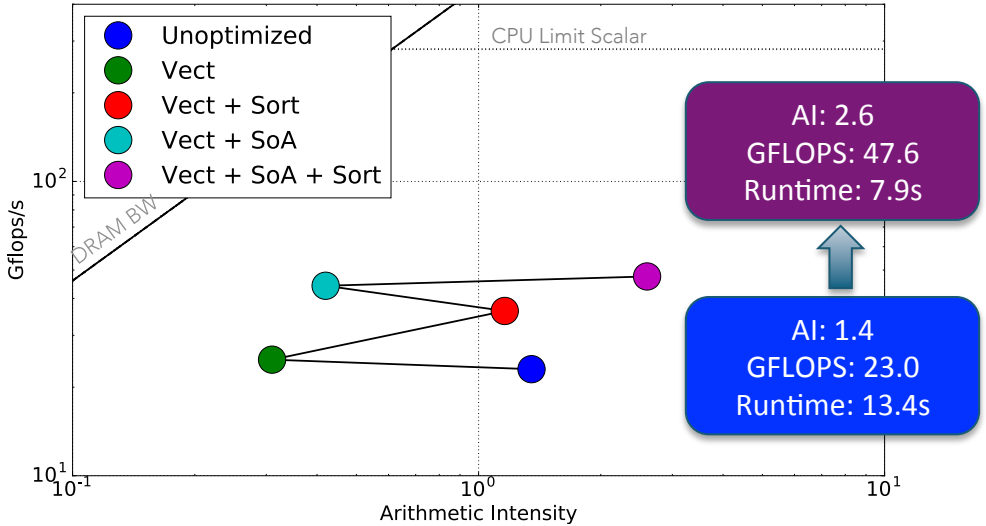
Optimizations to Vectorization and Memory Layout
Move the Push Kernel Up on Classical Roofline (1)



- Single *Haswell* node (Cori phase I)
- 4 MPI ranks, 8 threads rank
- DRAM Memory accesses collected with l Vtune
- OPS collected with sde

AI and GFLOPS are often coupled.
Here optimizations to memory layout lead to larger improvement in GFLOPS than AI.

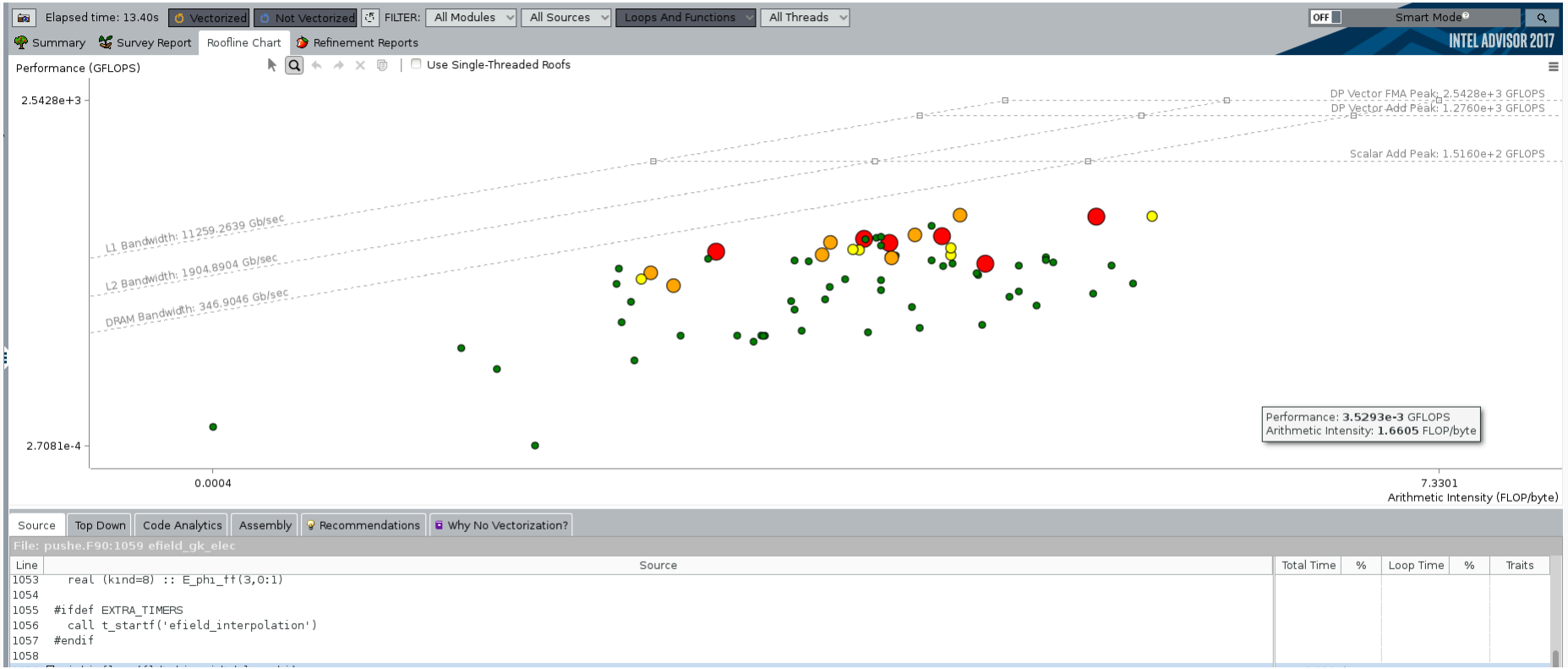
Optimizations to Vectorization and Memory Layout
Move the Push Kernel Up on Classical Roofline (2)



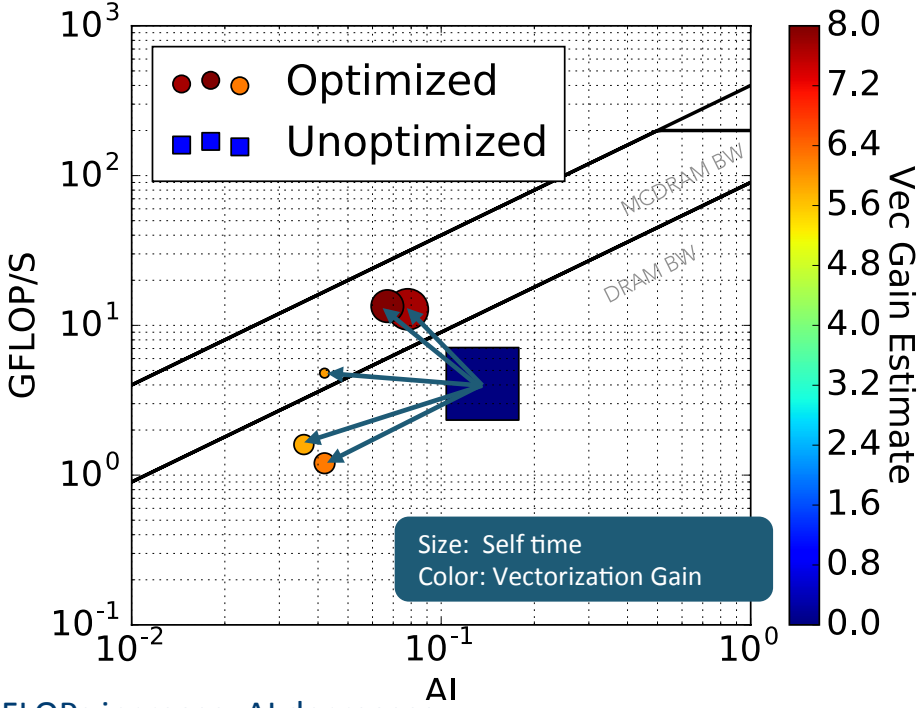
- Single *KNL* node w/ 64 cores, Quadrant flat mode
- 4 MPI ranks, 16 threads per rank
- DRAM Memory accesses collected with Intel Vtune
- FLOPS collected with sde

The penalty for L2 cache misses from unsorted data is higher on KNL than Haswell due to the absence of a low-latency L3 cache.

Roofline Breakdown by Loops and Functions with Intel Advisor – A Zoo of Data Points



Effect of Optimizations on 1st Order B Interpolation

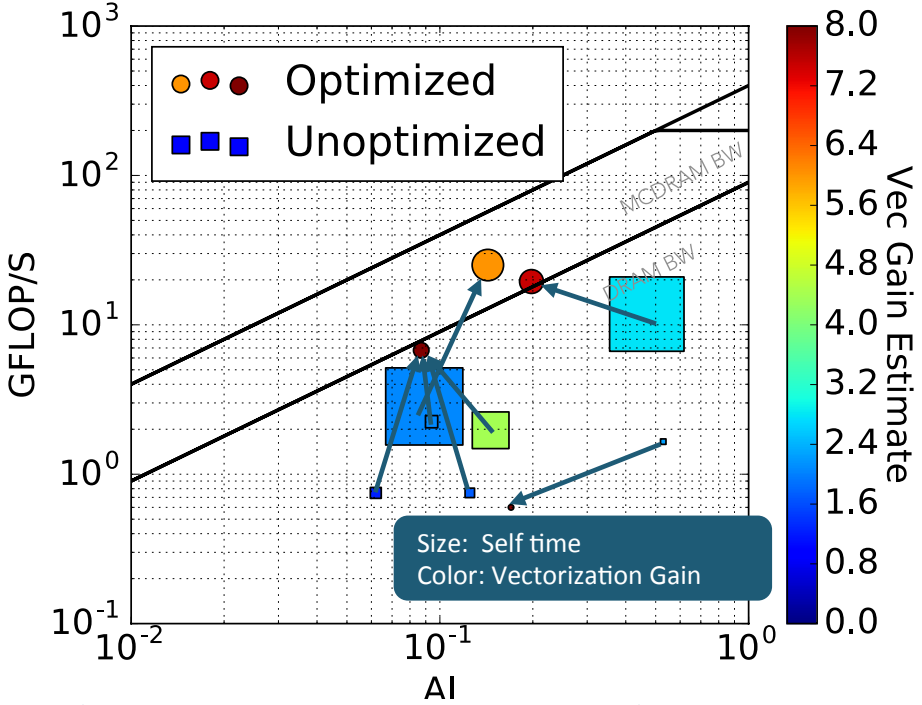


- Single KNL quadcache node 1 rank, 64 threads.
- Data collected with Advisor survey + tripcounts
- Inner loops over blocks of particles added
  - Scalar function → vectorized loops
- Most time-consuming loops above DRAM bandwidth limit

GFLOPs increase, AI decreases
→ Data alignment should be next optimization target

Total time: 3.5s → 2.1s
Peak GFLOPS: 4.0 → 16.0

Effect of Optimizations on Equations of Motion



- Single KNL quadcache node, 1 rank, 64 threads.
- Data collected with Advisor survey + tripcounts
- SoA data structure for field data added
- Large number of implicit loops merged into simd loops

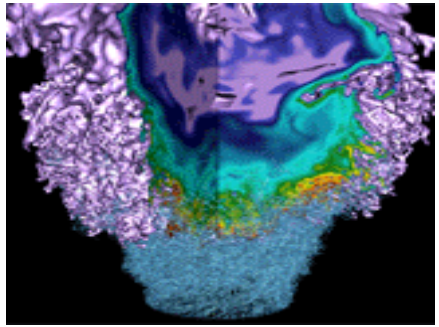
Large reduction in run time. Vectorization and GFLOPs improve. AI roughly constant on average although decrease seen in parts.

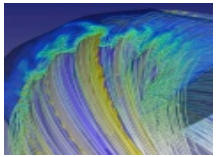
Total time: 8.8s → 1.2s
Peak GFLOPS: 12.6 → 25.1

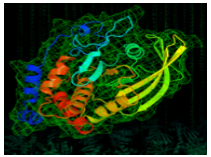
Conclusions and Insights from XGC1

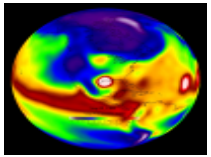
- Optimizations have improved vectorization and memory access patterns
  - Approximately 2x gained in total performance
  - Optimized version has roughly equal per-node performance on KNL and Haswell
- Roofline analysis has been a useful tool in focusing optimization efforts
  - High AI → focus on enabling vectorization, do not worry about memory *bandwidth*
  - Theoretically still room for 10x-20x improvement, what is limiting performance?
    - Memory latency, Memory alignment, Integer operations, Type conversions, ...
  - Advisor tools are a great help in automating detailed analysis

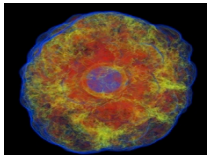
Part 4: Conclusions

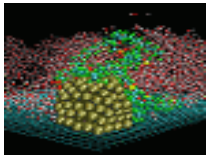












Conclusions

- Roofline analysis helps to determine the gap between applications and peak performance of a computing platform
- Advisor provides an all-in-one tool for cache-aware roofline analysis
  - Automates a lot of tedious analysis from the developer
  - Helps to find hotspots in your code and analyze effect of optimizations
  - Work on classical roofline is in progress
- Python library for postprocessing Advisor data
  - https://github.com/tkoskela/pyAdvisor

National Energy Research Scientific Computing Center