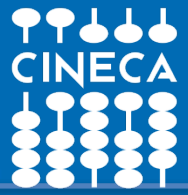
Roofline Model

Fabio Affinito / Marco Celoria



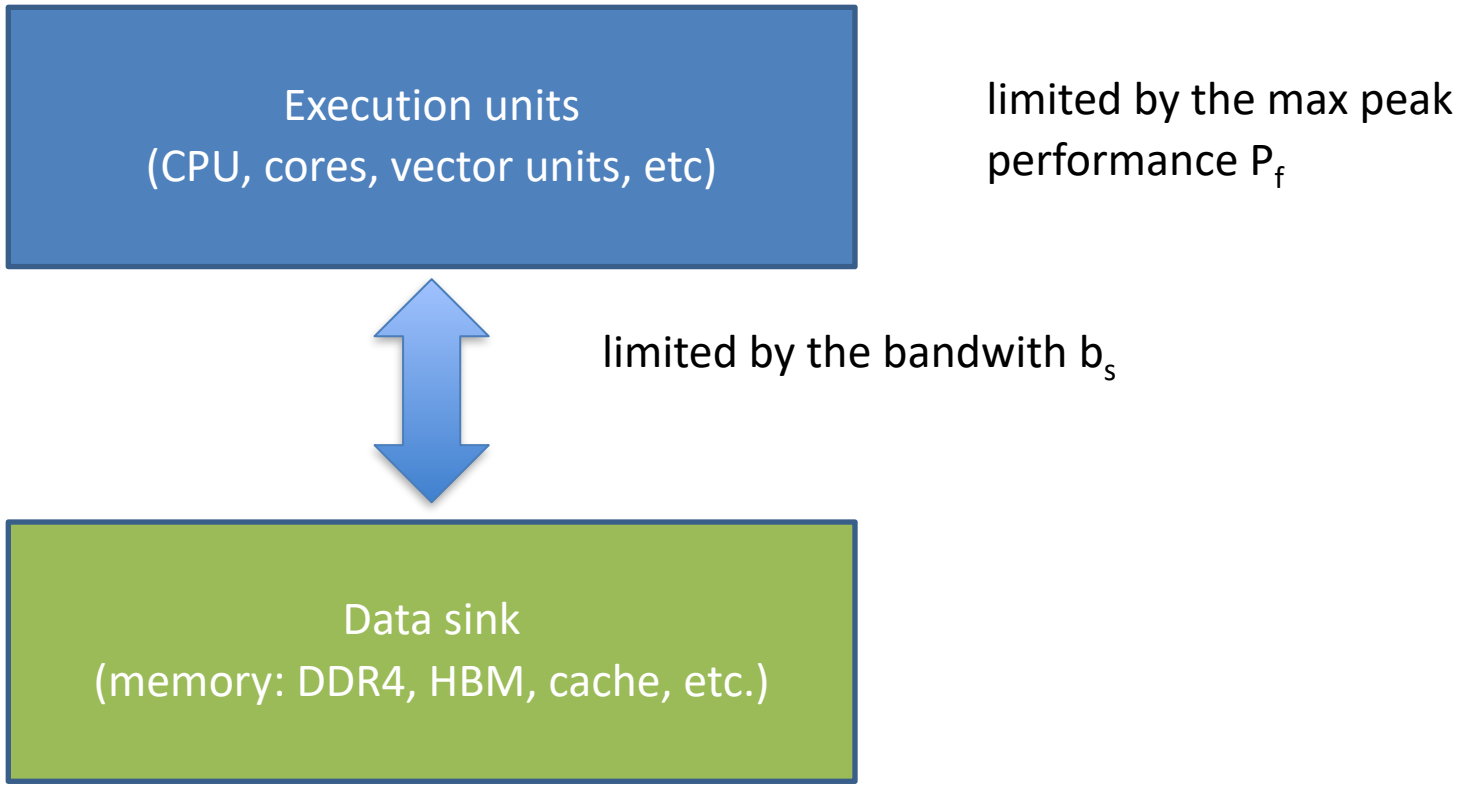
The motivation

CINECA

- How good am I exploiting the resources of my node?
  - number crunching (how much in terms of flop/s)
  - usage of the memory (am I limited by the memory usage?)
- The (empirical) Roofline Model provides a tool to visualize how much of the resources I am using and if (and where) there is room for improvement

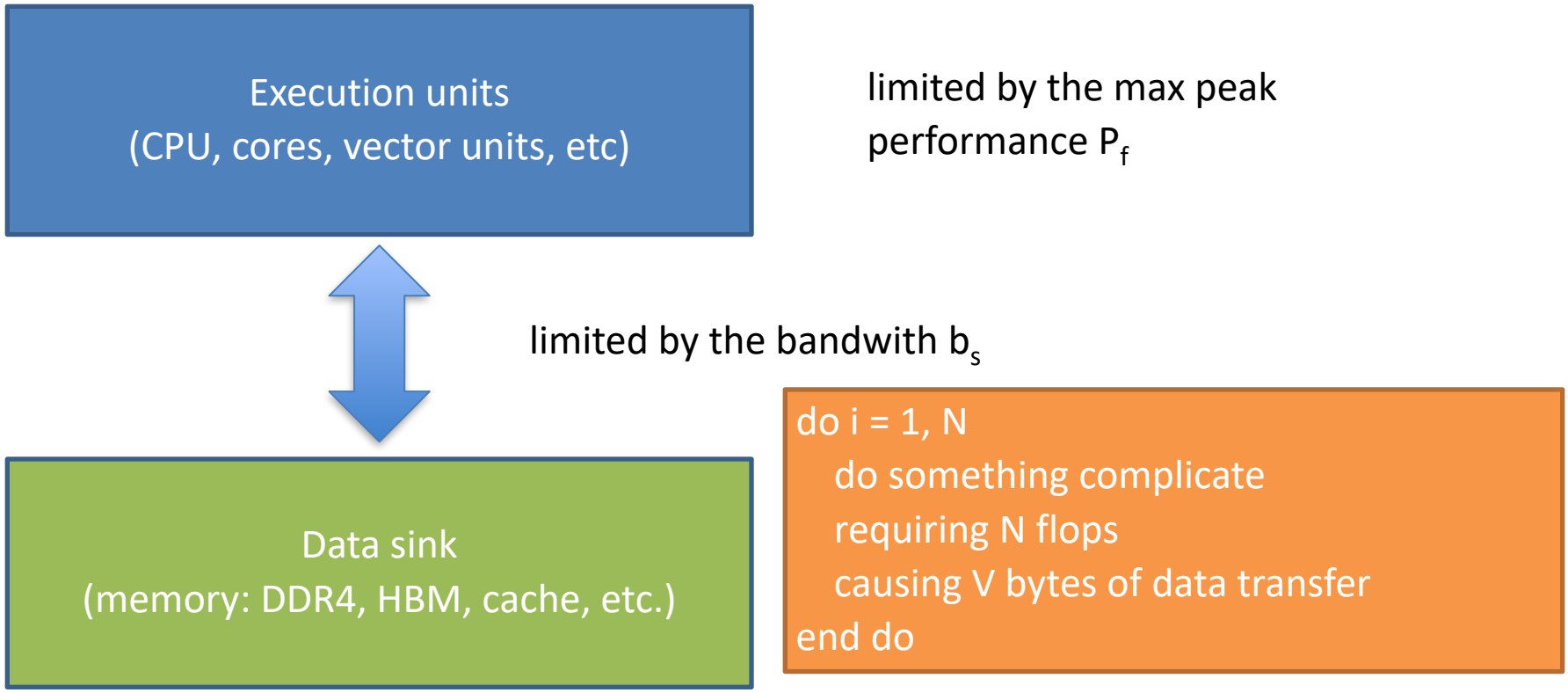
A simple model

- Let’s start with a very simple model



A simple model

- Let’s start with a very simple model



Arithmetic intensity

- Let's start with a very simple model

```
do i = 1, 100000
   do something complicate
   requiring N flops
   causing V bytes of data transfer
end do
```

I cannot directly compare P_f with b_s because they have different dimensions (flops/s and byte/s, respectively)
I must introduce a new quantity which is directly related to the bandwith:

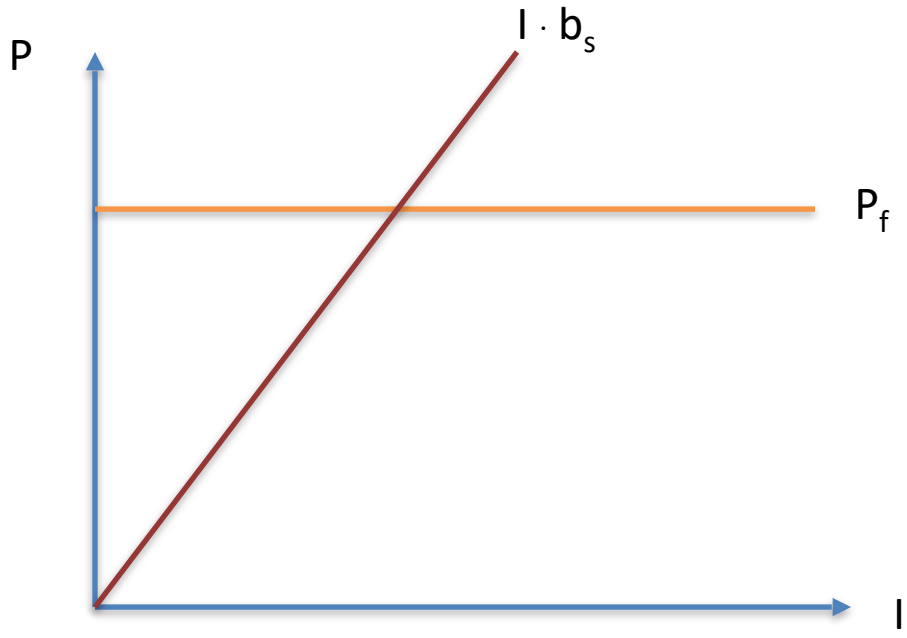
$$I = N / V$$

I is the arithmetic intensity and it is measured in flop/byte

Naive roofline model

- Question:
  - What is the highest performance achievable on our hardware (i.e. on our node?)

$$P = \max(P_f, I \cdot b_s)$$



The performance is limited:
- at high I by the peak performance
- at low I by the memory bandwith

All the values above the roofline are not permitted

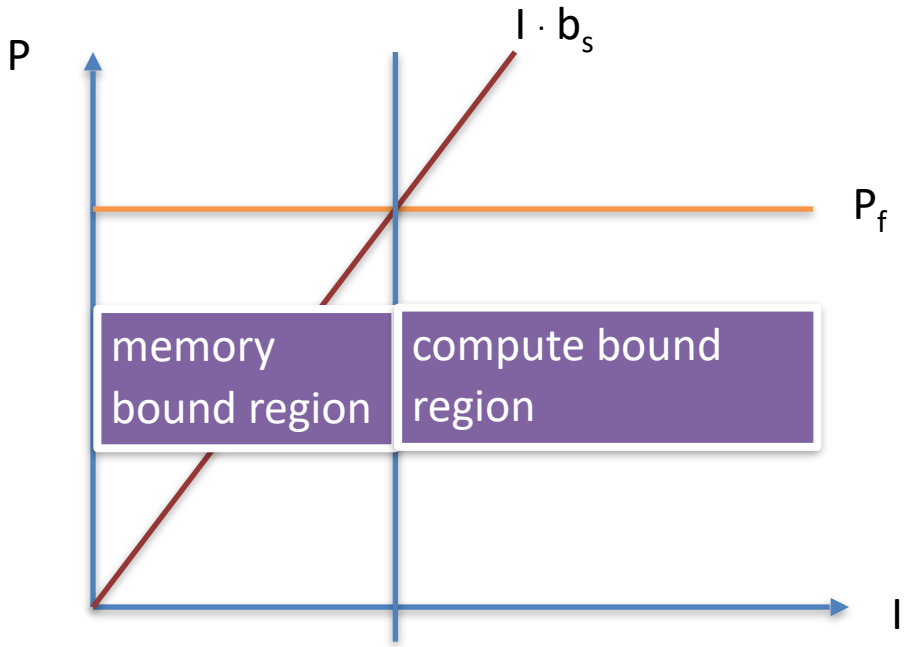
Naive roofline model

- Question:
  - What is the highest performance achievable on our hardware (i.e. on our node?)

$$P = \max(P_f, I \cdot b_s)$$

The performance is limited:
- at high I by the peak performance
- at low I by the memory bandwith

All the values above the roofline are not permitted

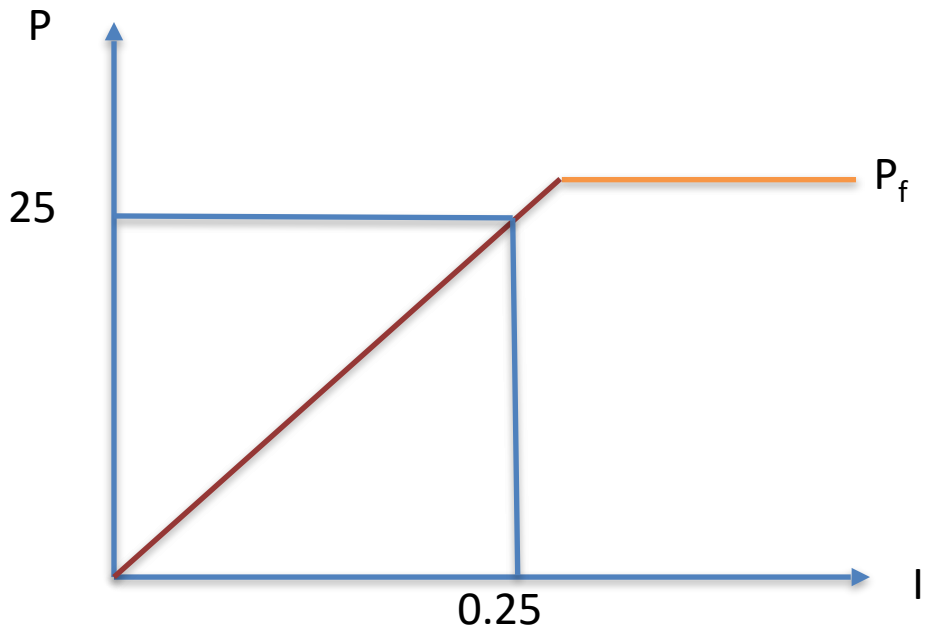


Example

```
double s=0, a[];
for(i=0; i<N; ++i){
 s = s + a[i] * a[i];
}
```

$$I = \frac{2\ flop}{8\ bytes} = 0.25\ \text{flop/bytes}$$

The memory bandwith and the peak performance are machine characteristics: they can be obtained from the specification of the architecture or measured (see later).
In this case, let's suppose to have:
P_f = 4 Gflop/s
b_s = 10 Gbyte/s



Example

```
double s=0, a[];
for(i=0; i<N; ++i){
 s = s + a[i] * a[i];
}
```

$$I = \frac{2\ flop}{8\ bytes} = 0.25\ \text{flop/bytes}$$

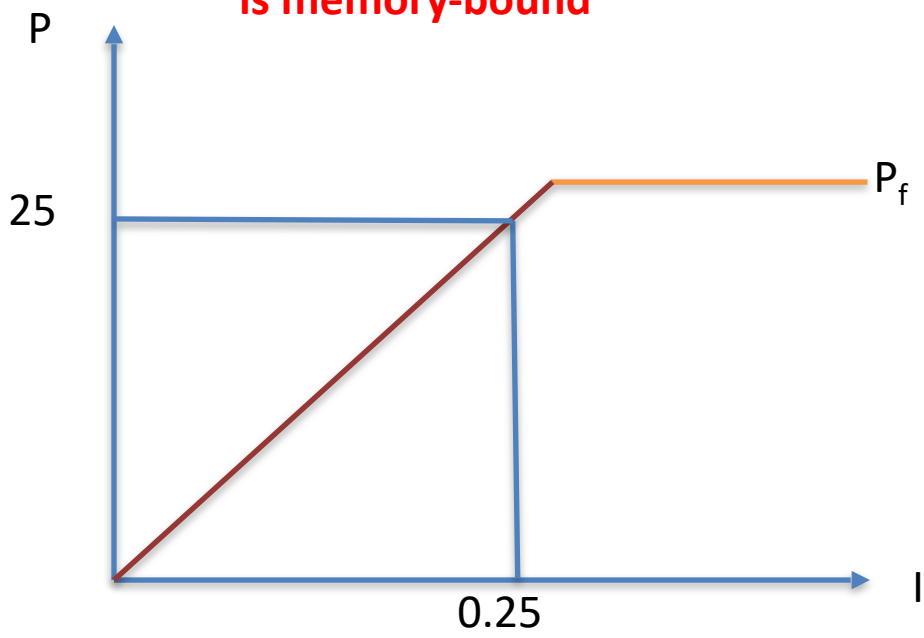
The memory bandwith and the peak performance are machine characteristics: they can be obtained from the specification of the architecture or measured (see later).

In this case, let's suppose to have:

P_f = 4 Gflop/s

b_s = 10 Gbyte/s

This portion of code is memory-bound



Code balance and arithmetic intensity

```
double a[], b[];
for(i=0; i<N; ++i) {
    a[i] = a[i] + b[i];}
```

B_C = 24B / 1F = 24 B/F

I = 0.042 F/B

```
double a[], b[];
for(i=0; i<N; ++i) {
    a[i] = a[i]+ s * b[i];}
```

B_C = 24B / 2F = 12 B/F

I = 0.083 F/B

Scalar – can be kept in register

```
float s=0, a[];
for(i=0; i<N; ++i) {
    s = s + a[i] * a[i];}
```

B_C = 4B / 2F = 2 B/F

I = 0.5 F/B

Scalar – can be kept in register

```
float s=0, a[], b[];
for(i=0; i<N; ++i) {
    s = s + a[i] * b[i];}
```

B_C = 8B / 2F = 4 B/F

I = 0.25 F/B

Scalar – can be kept in register

Theoretical peak performance

- The theoretical peak performance P_f

  $P_f = C_v \times f_c \times I_c$ = Cores × Clock Rate × Flops/Cycle

- To determine I_c we take the vector-width and divide by the word size in bits. We may also include the fused multiply-add (FMA) instruction as another factor of two operations per cycle.

  $I_c = VW/W_{bits} \times F_{ops}$

  $= (256\text{-bit Vector Unit}/64\text{bits}) \times (2 \text{ FMA})$

  $= 8$ Flops/Cycle

Theoretical memory bandwidth

- The theoretical memory bandwidth is

$$b_T = \text{MTR} \times \text{M}_c \times \text{T}_w \times \text{N}_s$$

$$= \text{Data Transfer Rate} \times \text{Memory Channels} \times \text{Bytes Per Access} \times \text{Sockets}$$

- The achievable memory bandwidth is lower than the theoretical bandwidth due to the effects of the rest of the memory hierarchy.
- For this, we will turn to empirical measurements of bandwidth at the CPU.

Limits of the model

- Let's review the assumptions made so far:
  - we accepted the specification values for the peak performance and the bandwith: typically achievable values are much lower
  - we pictured the memory as a single entity, neglecting the hierarchical structure (L1, L2, etc)
  - data transfer and computation overlap perfectly: **either** the limit is core execution **or** data transfer
  - latency is not even considered

Some improvements to our model

- Replace P_f with P_{max}, defined as the maximum achievable peak performance, for example when data are available on the L1
- Let b_S be the applicable (saturated) peak bandwidth of the slowest data path utilized (measure attainable bandwidth using, e.g. STREAM)

Some improvements to our model

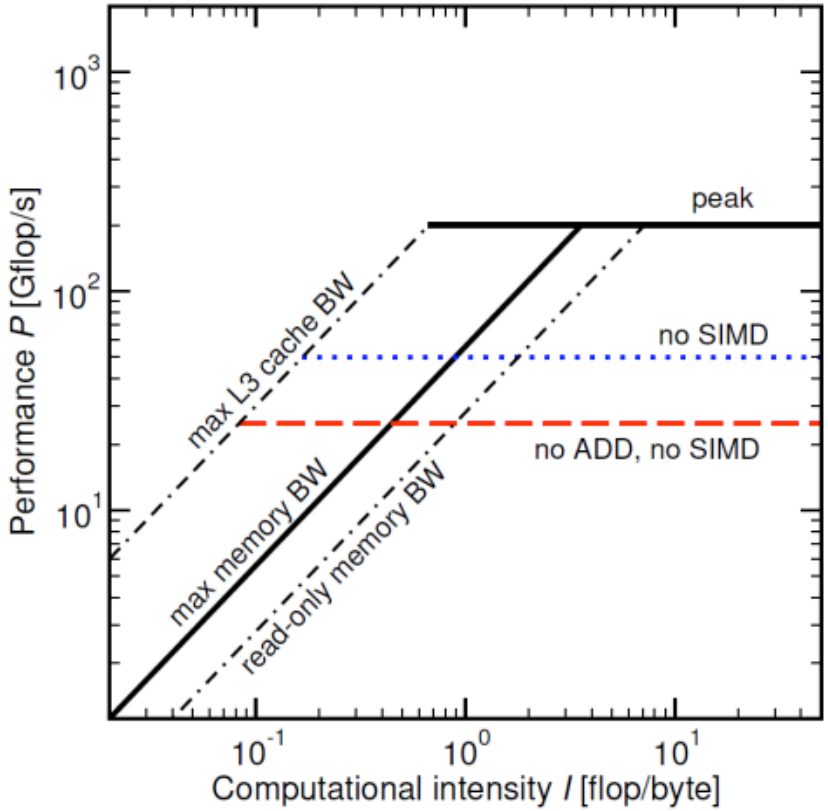
- To run STREAM:

```
git clone https://github.com/jeffhammond/STREAM.git

Modify the Makefile and use the following CFLAGS:

CC = gcc
CFLAGS = -O3 -march=native -fstrict-aliasing -ftree-vectorize -fopenmp
-DSTREAM_ARRAY_SIZE=80000000 -DNTIMES=20


stream_c.exe: stream.c
    $(CC) $(CFLAGS) stream.c -o stream_c.exe


On a compute node, compile and run

make stream_c.exe
./stream_c.exe
```

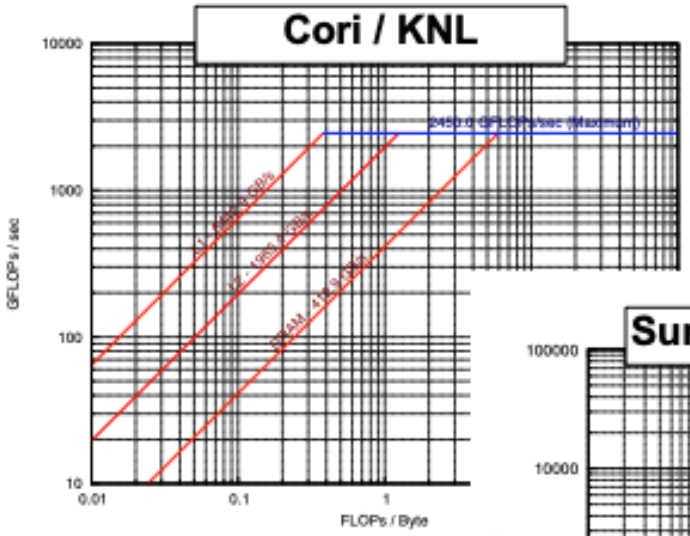
Graphical representations

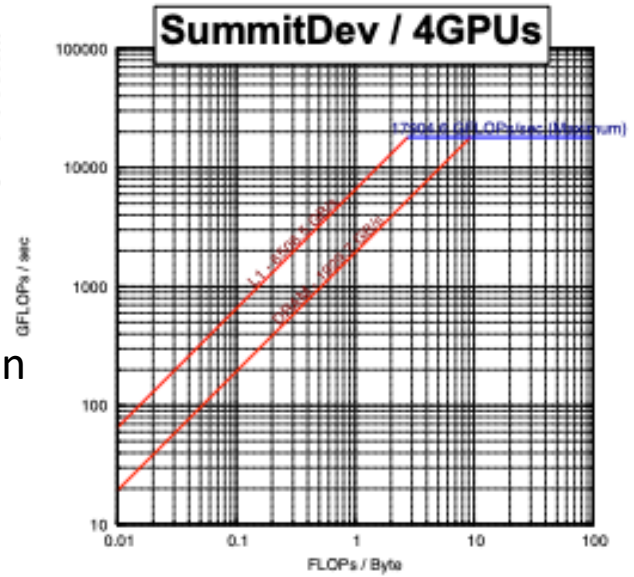
- Reality can be a little more complicate than our model

Where is my application? What can I do to improve the exploitation?
-> application characterization

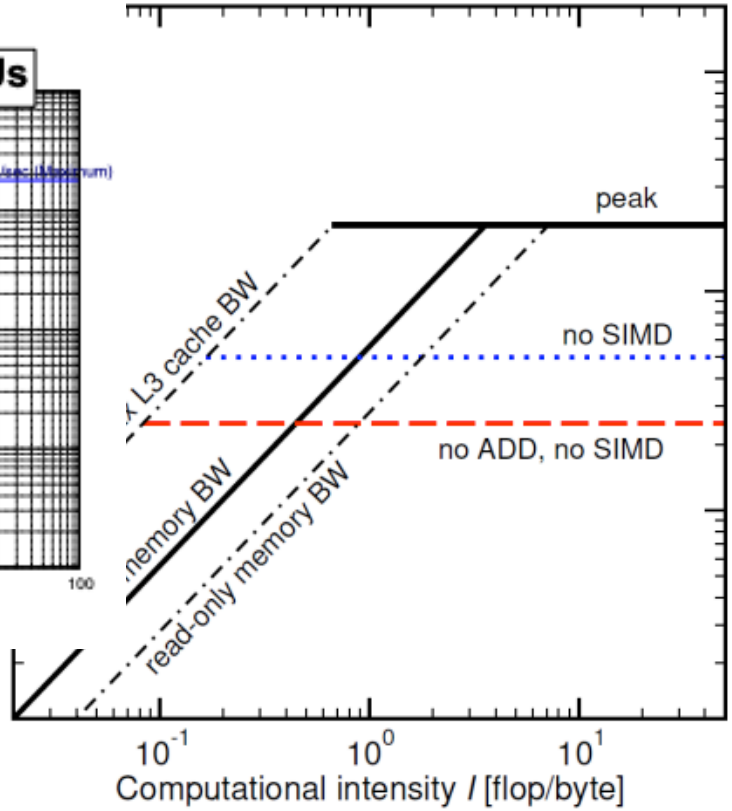


Graphical representations



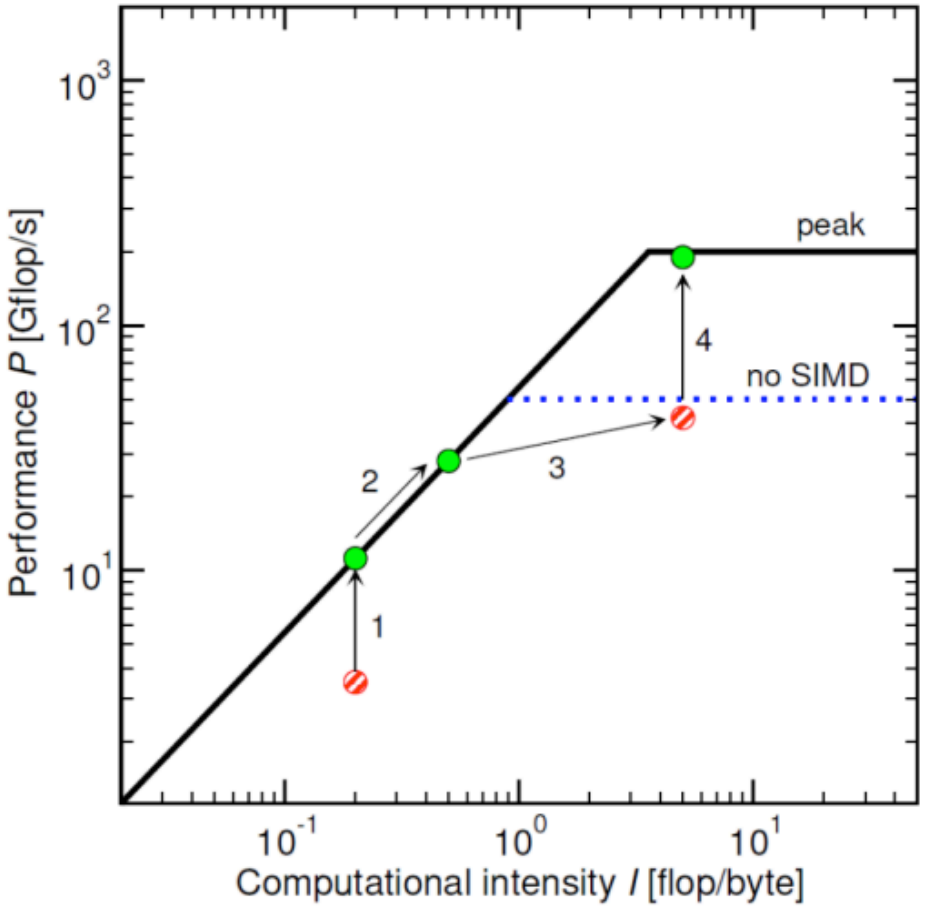


machine characterization



Tracking optimization work

1) improving the serial code to reach the maximum of the available BW
2) increase the intensity (for example using spatial loop blocking)
3) increase intensity and go in the compute bound regime
4) optimize the code and reach the compute performance peak



Counting flop/s

Manual Counting

- Go thru each loop nest and count the number of FP operations
- ✓ Works best for deterministic loop bounds
- ✓ or parameterize by the number of iterations (recorded at run time)
- ✘ Not scalable

Perf. Counters

- Read counter before/after
- ✓ More Accurate
- ✓ Low overhead (<%) == can run full MPI applications
- ✓ Can detect load imbalance
- ✘ Requires privileged access
- ✘ Requires manual instrumentation (+overhead) or full-app characterization
- ✘ Broken counters = garbage
- ✘ May not differentiate FMADD from FADD
- ✘ No insight into special pipelines

Binary Instrumentation

- Automated inspection of assembly at run time
- ✓ Most Accurate
- ✓ FMA-, VL-, and mask-aware
- ✓ Can count instructions by class/type
- ✓ Can detect load imbalance
- ✓ Can include effects from non-FP instructions
- ✓ Automated application to multiple loop nests
- ✘ >10x overhead (short runs / reduced concurrency)

slide courtesy of NERSC

Measure data movement

Manual Counting

- Go thru each loop nest and estimate how many bytes will be moved
- Use a mental model of caches
- ✓ Works best for simple loops that stream from DRAM (stencils, FFTs, spare, …)
- ✘ N/A for complex caches
- ✘ Not scalable

Perf. Counters

- Read counter before/after
- ✓ Applies to full hierarchy (L2, DRAM,
- ✓ Much more Accurate
- ✓ Low overhead (<%) == can run full MPI applications
- ✓ Can detect load imbalance
- ✘ Requires privileged access
- ✘ Requires manual instrumentation (+overhead) or full-app characterization

Cache Simulation

- Build a full cache simulator driven by memory addresses
- ✓ Applies to full hierarchy and multicore
- ✓ Can detect load imbalance
- ✓ Automated application to multiple loop nests
- ✘ Ignores prefetchers
- ✘ >10x overhead (short runs / reduced concurrency)

slide courtesy of NERSC

Intel Advisor

In order to describe a code with the roofline model we use Intel Advisor:

- Root permissions not needed

Getting roofline with Intel Advisor is a two-pass approach

1. We perform a survey with --collect=survey
   - This provides the time in #Secs
   - No overhead

2. We collect flops and bytes with --collect=tripcounts --flops
   - This provides the time in #Flops, #Bytes (also vectorisation informations AVX-512)
   - The overhead is significant

After collecting these data, we plot the roofline model with

Axis X: AI = #FLOP / #Bytes
Axis Y: FLOP/S = #FLOP (vectorization aware) / #Seconds

Intel Advisor - Improving the roofline model

In the original roofline model:

- Ceilings for DDR and theoretical peak performance

We can improve the roofline model by considering the Cache-aware roofline model (CARM)

- Ceilings for cache/memory levels and Vectorized, single or double FP computations

slide courtesy of NERSC

How to generate CARM CPU roofline profile?

To generate a CARM roofline model from command line that is also compatible with MPI, use the following:

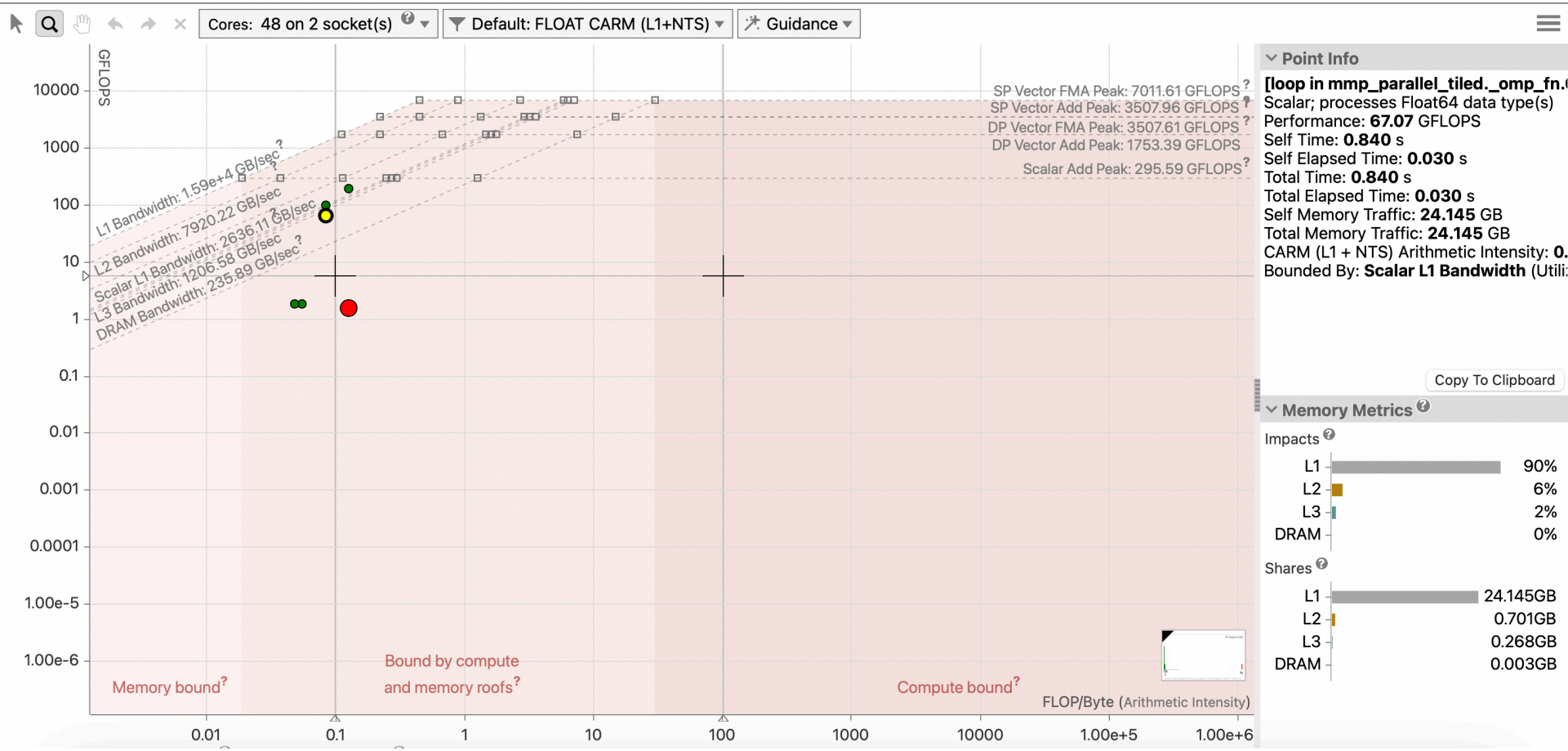
```
advisor --collect=survey --project-dir=./advi_res -- <exe-with-parms>

advisor --collect=tripcounts --flop --project-dir=./advi_res -- <exe-with-parms>
```

Finally, to get the report:

```
advisor --report=roofline --report-output=./roof.html --project-dir=./advi_res
```

slide courtesy of NERSC

Example

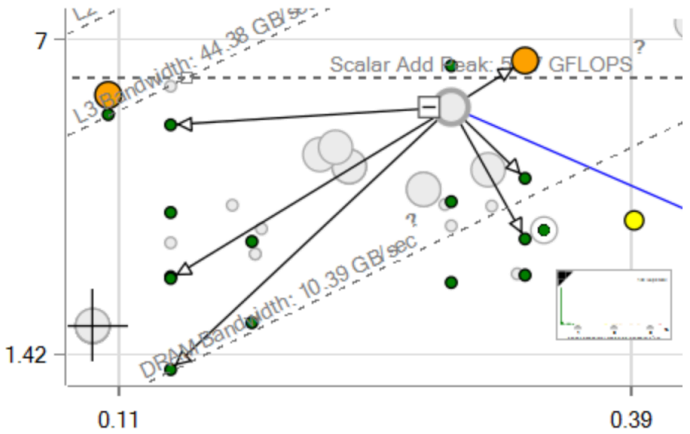


slide courtesy of NERSC

- In order to perform advanced collection of callstack data during Roofline and Trip Counts & FLOP analysis add the following option

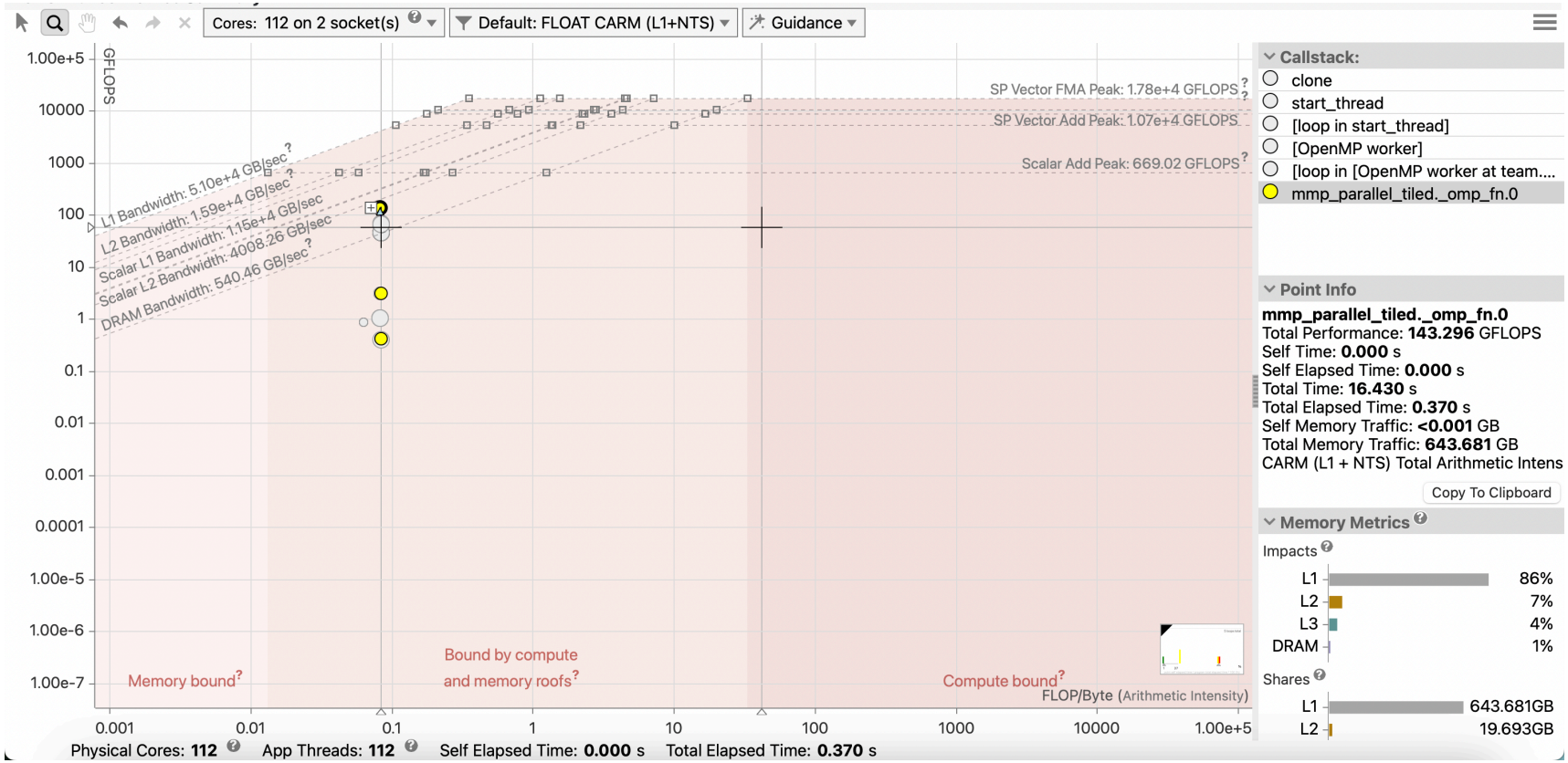
```
advisor --collect= survey --project-dir=./advi_res -- <x-parms>
advisor --collect=tripcounts --flop --stacks --project-dir=./advi_res -- <x-parms>

advisor --report=roofline --with-stack --report-output=./roof.html --project-
dir=./advi_res
```

- The ability to use total data gives the Roofline with Callstacks a means of adjusting the granularity of the data.



In the roofline report Arrows indicate relationships between dots.

Example



slide courtesy of NERSC

Memory Level Roofline

The Roofline Model with Intel Advisor

- First Implementation: Cache-Aware Roofline Model
- CARM represents cumulative (L1+L2+LLC+DRAM) traffic-based AI for application kernels.
- Whether the data is coming from DRAM or any cache level doesn't change anything. Such AI is invariant for the given code/platform combination.

New Implementation: Memory Level Roofline (MLR)

- Based on cache simulation, evaluate traffic between each memory subsystem
- AI combines CARM, Original and L1-L2-L3 only perspectives
- Harder to interpret for multiple kernels at a time

How to generate MLR CPU roofline profile?

To generate a CARM roofline model from command line that is also compatible with MPI, use the following:

```
advisor --collect= survey --project-dir=./advi_res -- <exe-with-parms>

advisor --collect=tripcounts --flop --enable-cache-simulation
--project-dir=./advi_res -- <exe-with-parms>
```

Finally, to get the report:

```
advisor --report=roofline --project-dir=advisor_proj_simp --with-stack --
memory-level=L1_L2_L3_DRAM
```

Memory Level Roofline

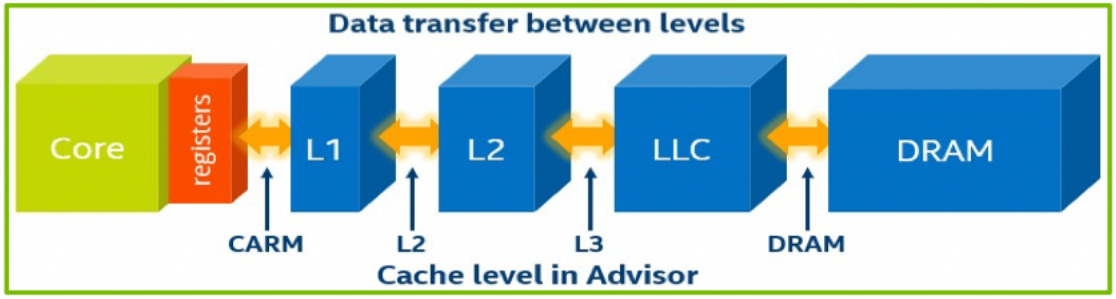
- Same Flops (it's the same loop)
- But with different data transfers

  - 1st dot comes from CARM (L1)
  - 2nd dot comes from traffic L1 <-> L2
  - 3rd dot comes from traffic L2 <-> L3
  - 4th dot comes from traffic L3 <-> DRAM

$$AI = \frac{\text{Flops}}{\text{Bytes transferred}}$$

This allows to detect cache problems in your application.

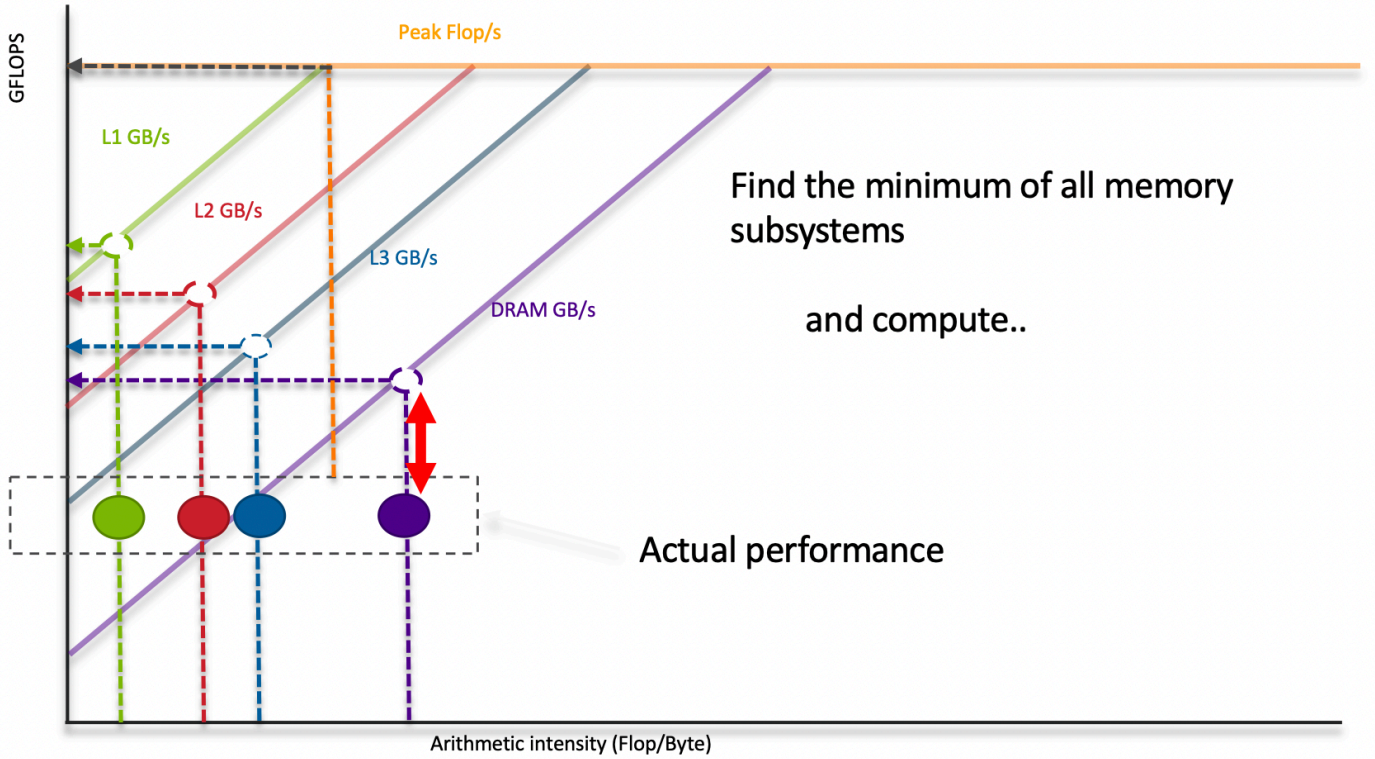
It is even possible to detect which cache level is behaving as a bottleneck.



How to Interpret Your Current Limitation?

Finding what is your current performance upper bound can be done by looking at each memory subsystem and projecting the arithmetic intensity on the appropriate roofline.

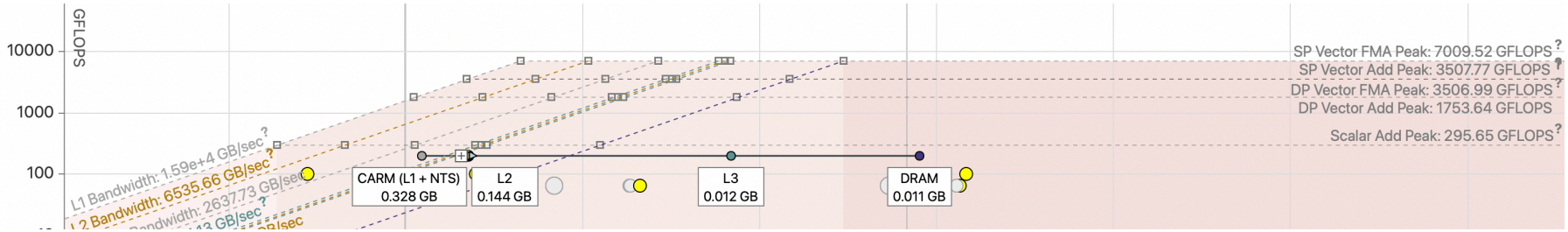
The upper bound performance is the minimum of all your projections.



How to Interpret Your Current Limitation?

Finding what is your current performance upper bound can be done by looking at each memory subsystem and projecting the arithmetic intensity on the appropriate roofline.

The upper bound performance is the minimum of all your projections.



Try on your own

Try to start with a simple matrix matrix multiplication

```
for (int i = 0; i < N; ++i) {
  for (int j = 0; j < M; ++j) {
    for (int k = 0; k < L; ++k) {
       C[i * M + j] += A[i * L + k] * B[k * M + j];
    }
  }
}
```

Profile with Intel Advisor to get a baseline

Try to consider vectorisation and multi-threading with OpenMP

Try to exploit better the cache using blocks or matrix transposition so that can data be traversed by row (data locality)