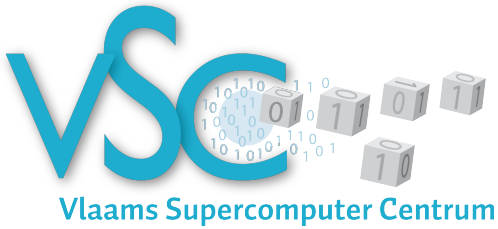
Code modernization a practical approach

HPC-TNT-1.2 fall 2016



Annie Cuyt ☐ Stefan Becuwe ☐ Franky Backeljauw ☐ Kurt Lust ☐ Engelbert Tijskens

Goal

- A practical approach
  - To build efficient applications
  - To understand performance issues
  - To maximize code flexibility
  - To minimize coding efforts

Overview

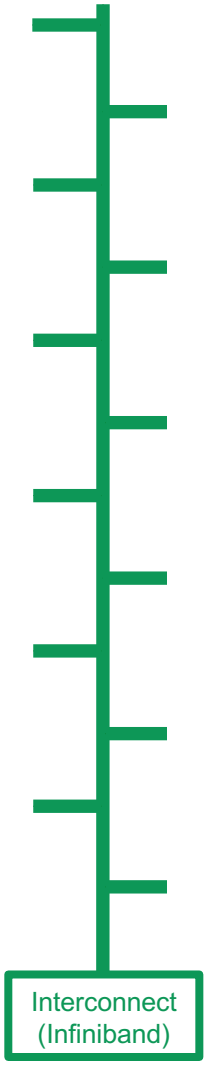
- Some computer architecture concepts related to performance and machine limits
  - Levels of parallelism → peak performance
  - Memory on modern CPUs → bandwidth, latency
- Example1 [toy problem]
  - Atomic system interacting through a Lennard-Jones potential
  - Monte Carlo setting
- Example 2
  - Atomic system interacting through Lennard-Jones potential
  - Molecular dynamics setting
- Optimizing code ⇔ optimizing data access
  - Spatial sorting using space filling curve
- Choosing a programming language

Levels of parallelism

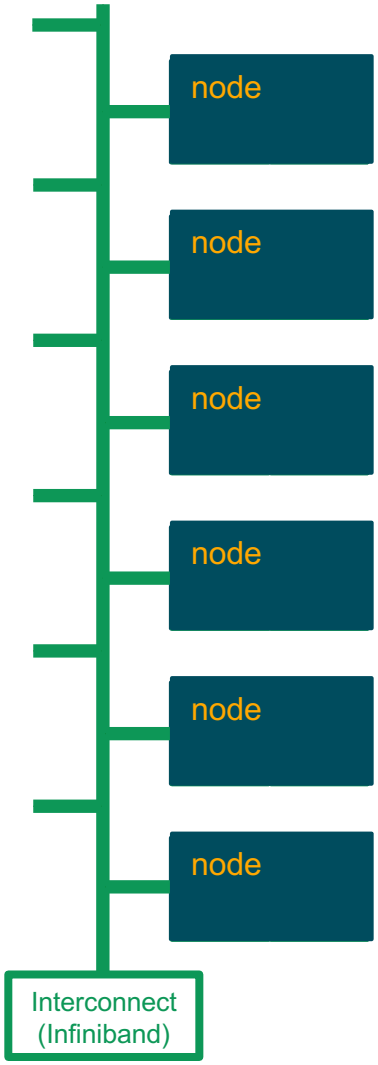
A little terminology

- A (**compute**) **node** in a cluster is basically a PC without all the peripheral devices
- Nodes are connected through a fast network: **interconnect**
- Every node has several **socket**s, each of which contains a processor
- Every processor contains many **core**s
- Each core can simultaneously execute one task: **thread**
  - Simultaneous multi-threading : more than one thread per core
  - Usually switched off on clusters
- Thread executes a sequential stream of instructions

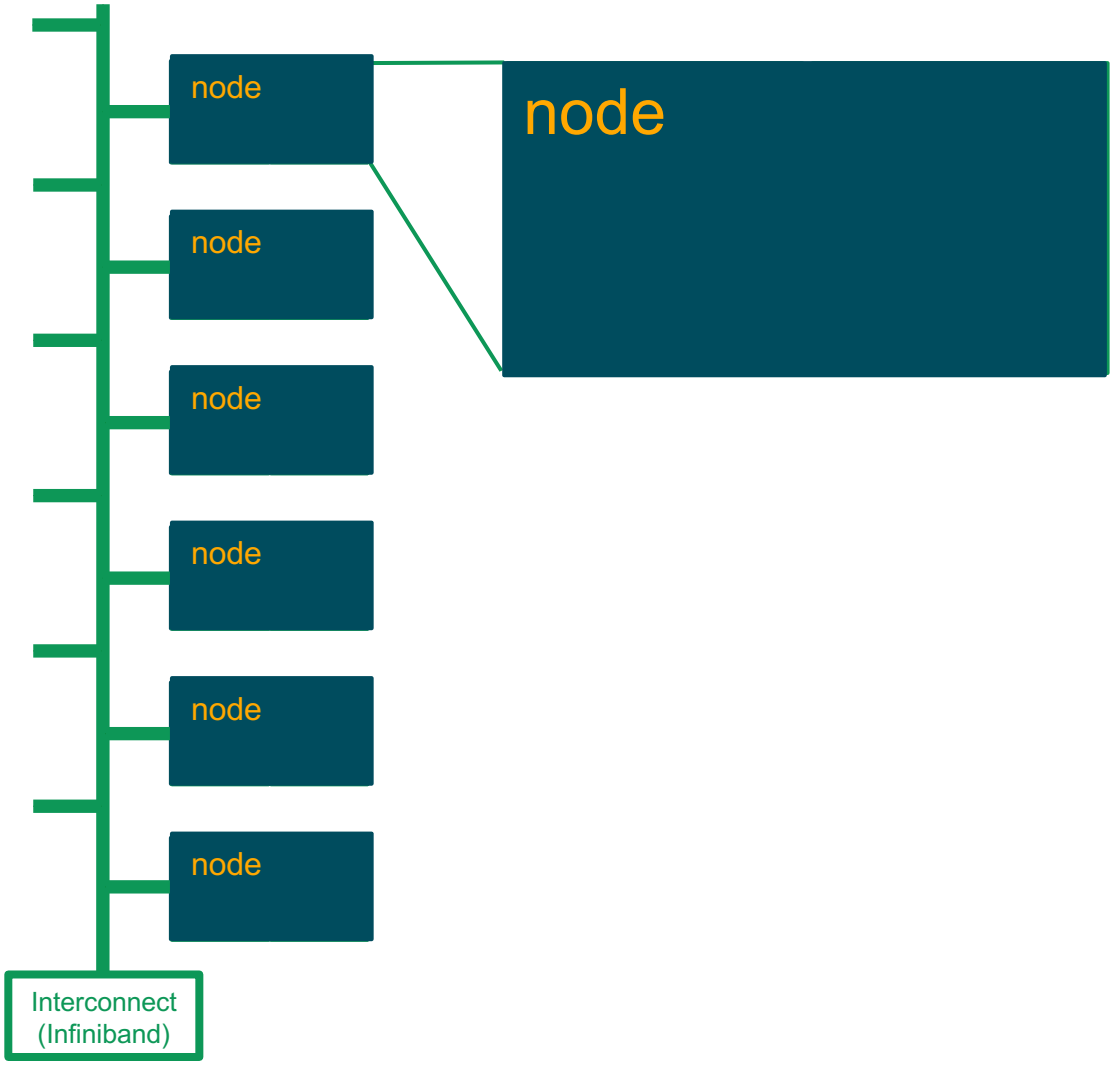
Terminology



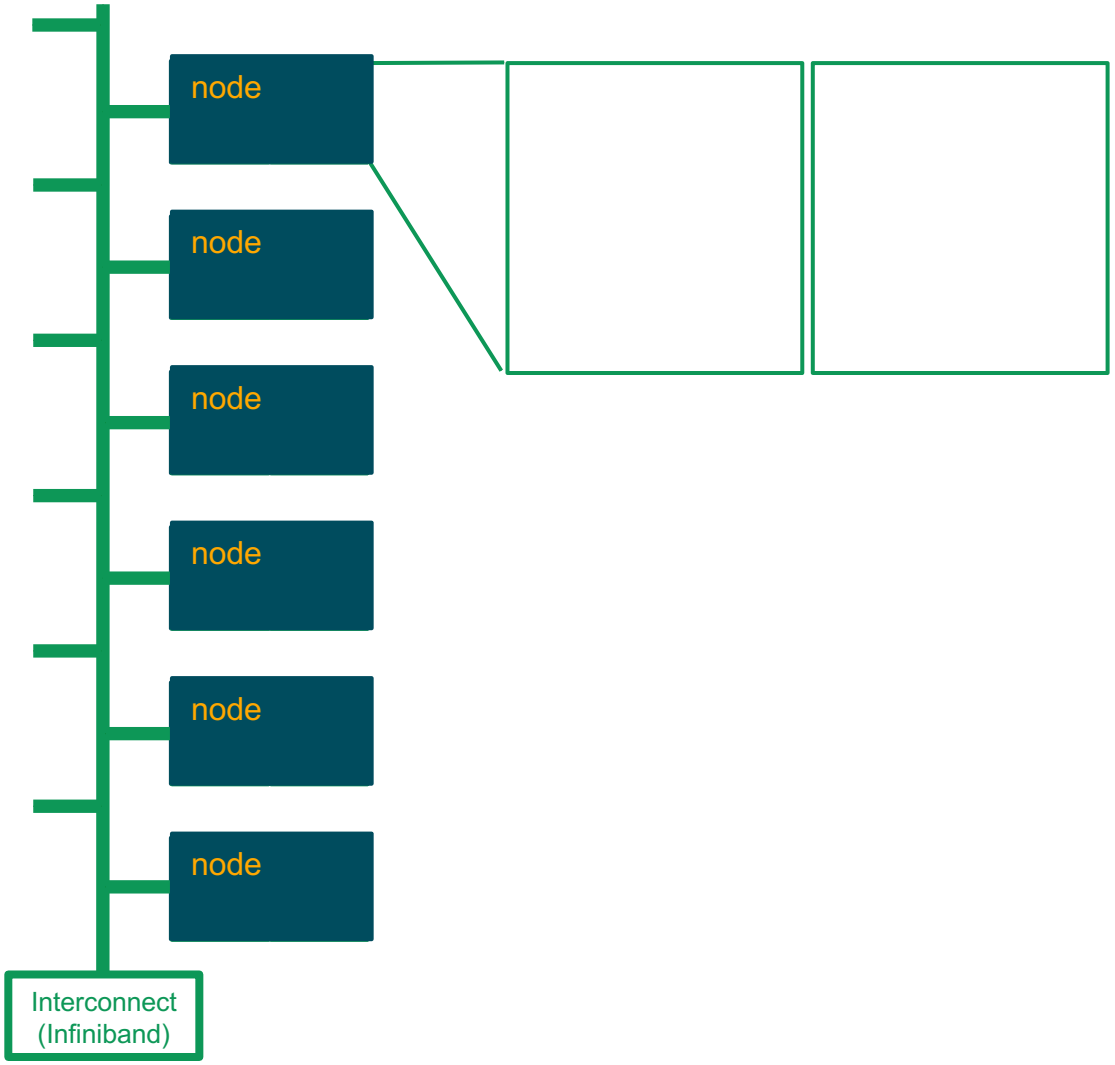
Terminology



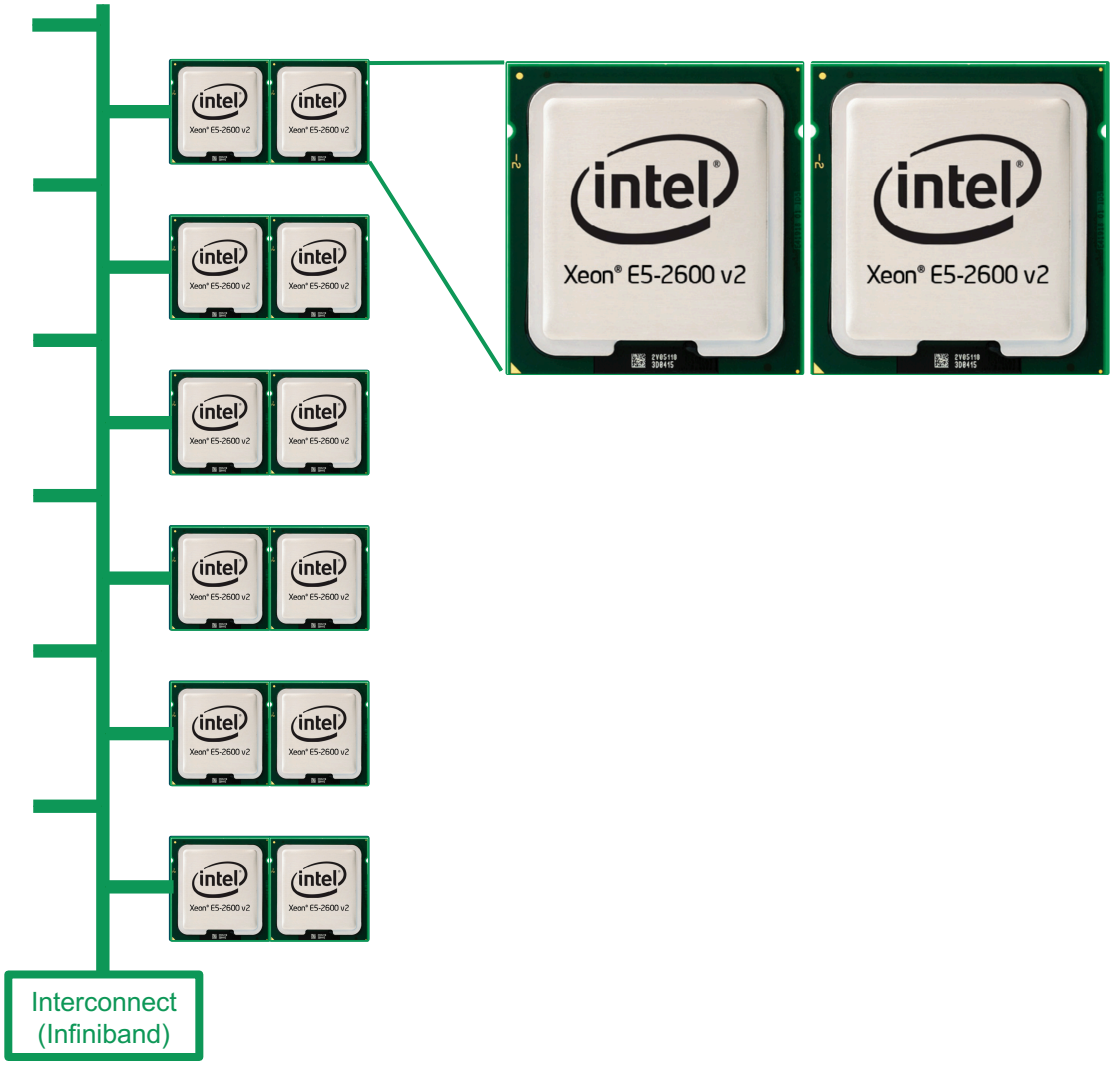
Terminology



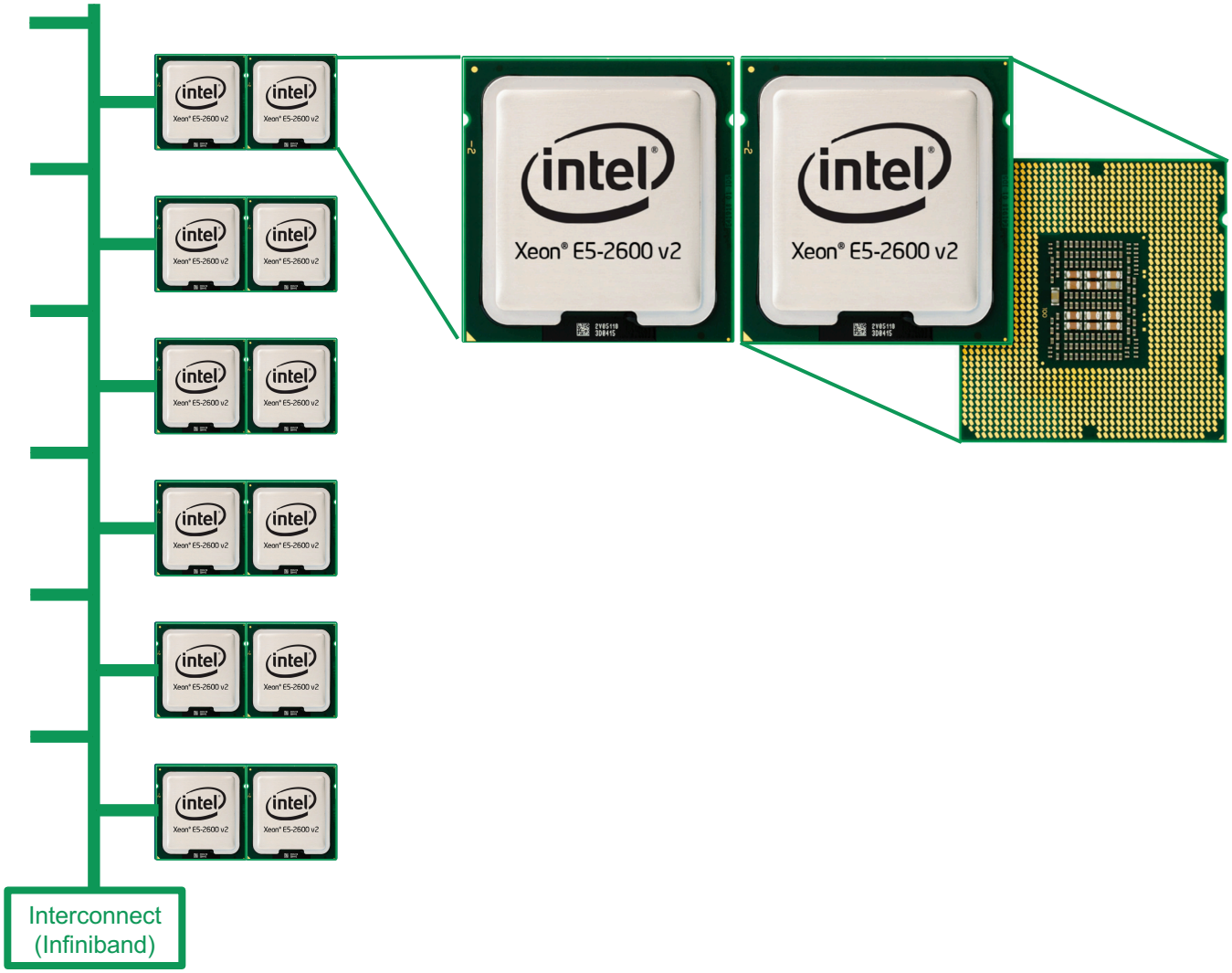
Terminology



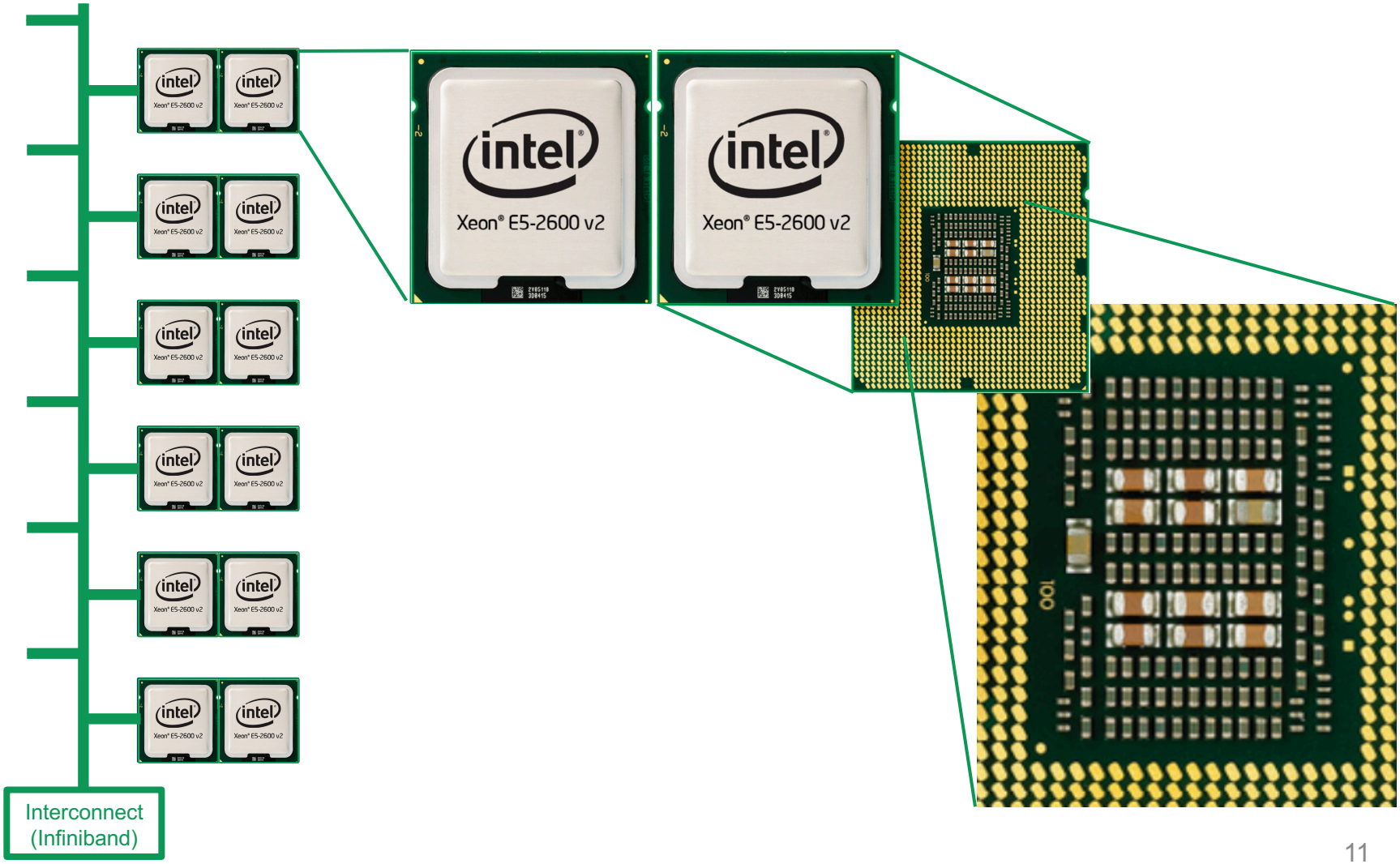
Terminology



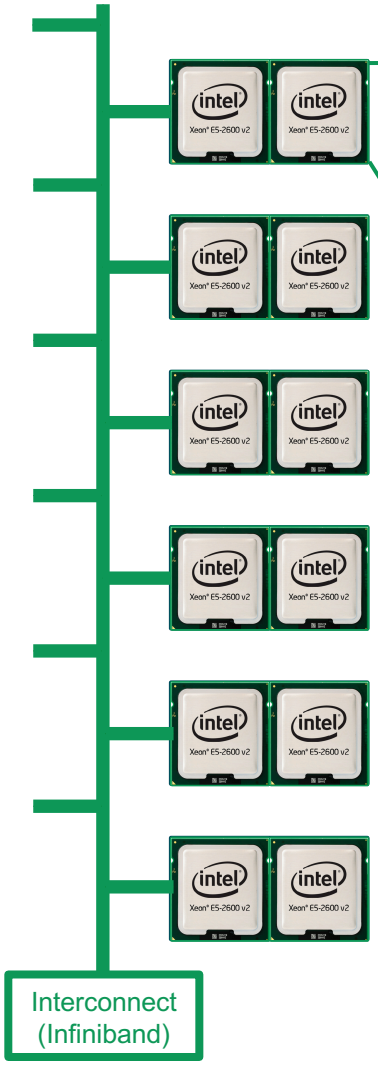
Terminology

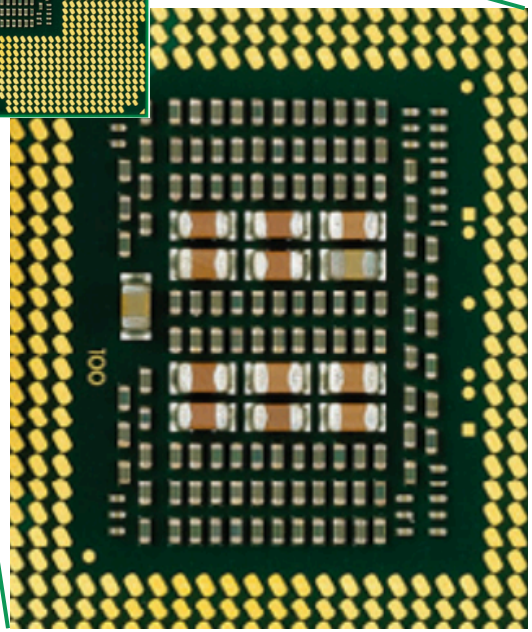


Terminology



Terminology





Hopper

- 168 nodes
- 2 sockets x E5-2680 v2
- 10 cores per CPU
- 20 cores per node
- 1 thread per core
- 3360 cores
- 3360 threads

Levels of parallelism

1. Several nodes can cooperate on a task
   - **Distributed** memory parallellization
   - Nodes communicate information via interconnect
     - Typically using MPI: Message Passing Interface
2. Several cores can cooperate on a task
   - **Shared** memory parallellization
   - Cores communicate information via memory
     - Often using OpenMP, but also MPI, …

Levels of parallellism

3. A single core may exploit **pipelining** and **vectorisation** to execute instructions in parallel

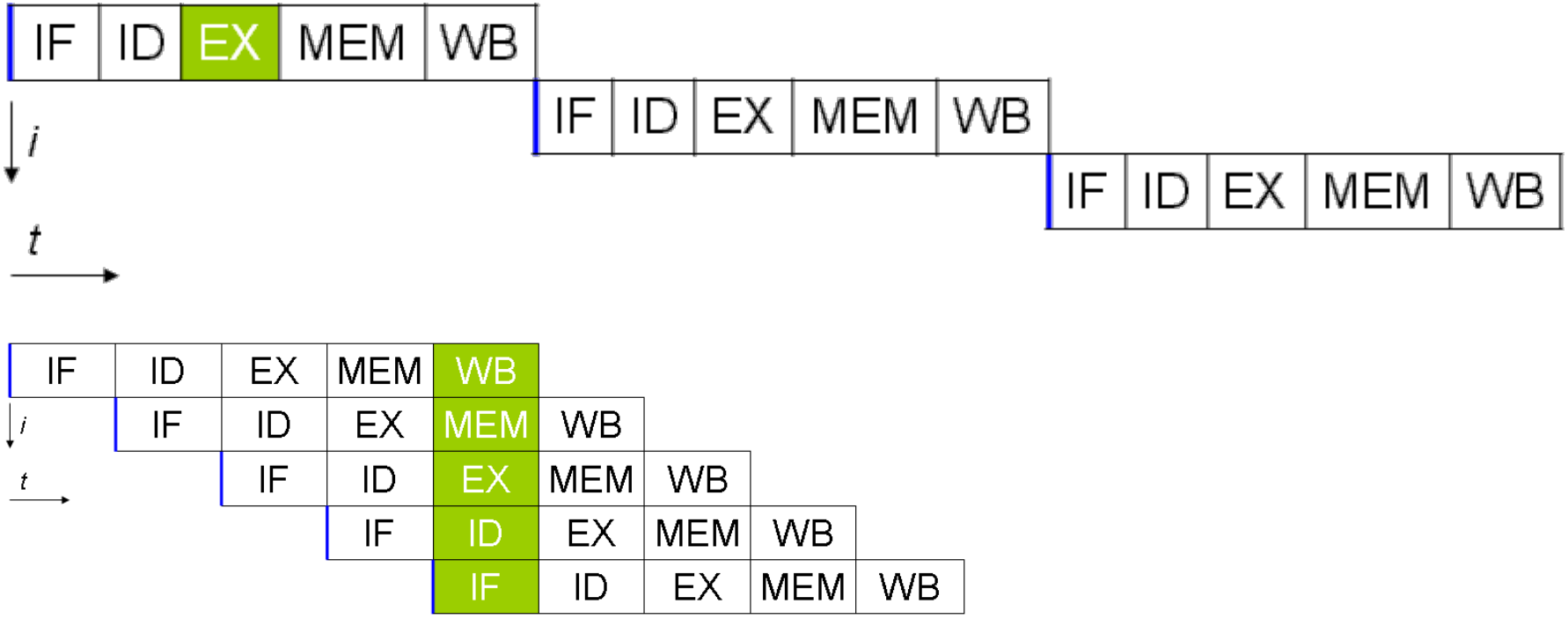
Pipelining

- Instructions break down in micro-instructions
- Each micro-instruction uses a distinct part of the hardware
  1. Instruction fetch (IF)
  2. Instruction Decode (ID)
  3. Execution (EX)
  4. Memory Read/Write (MEM)
  5. Result Writeback (WB)

[https://en.wikibooks.org/wiki/Microprocessor_Design/Pipelined_Processors]

Pipelining

Pipelining

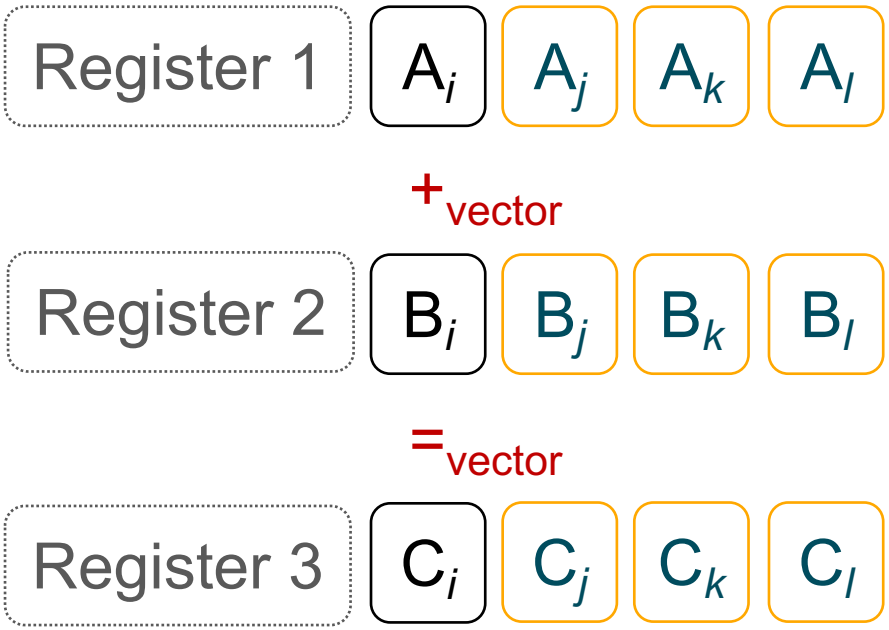

IF ID EX MEM WB
IF ID EX MEM WB
IF ID EX MEM WB
i
t
IF ID EX MEM WB
IF ID EX MEM WB
IF ID EX MEM WB
IF ID EX MEM WB
IF ID EX MEM WB
i
t
5 instructions executing simultaneously

Vectorization

Vectorization

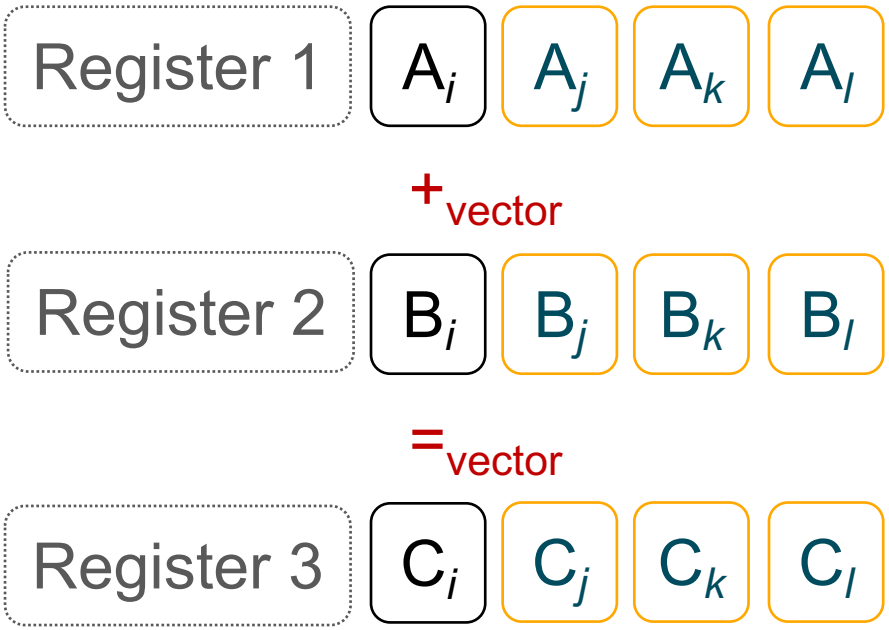
Load 2 operands in register
Execute 1 add instruction
Store 1 result

Vectorization



Load 2x4 operands in register
Execute 1 vadd instruction
Store 1x4 results

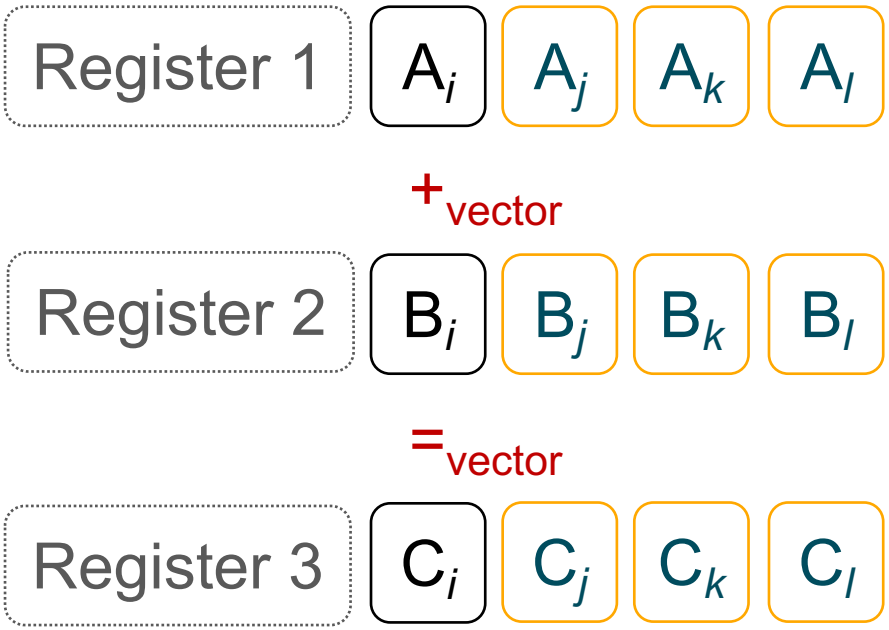
Vectorization



Load 2x4 operands in register
Execute 1 vadd instruction
Store 1x4 results

Single Instruction Multiple Data (SIMD)

Vectorization



Potentially 4x faster
[*if the loads and stores can be executed fast enough*]

Load 2x4 operands in register
Execute 1 vadd instruction
Store 1x4 results

Single Instruction Multiple Data (SIMD)

Vectorization

- Fused instructions: Fused Multiply-Add executes
  y = a*x+b (in vector mode
  in one cycle
- Vector register width on Hopper is 256 bits
  - 8 single precision numbers
  - 4 double precision numbers

Levels of parallellism

- 3 levels of parallellism:
  - Intra-core: pipelining and SIMD
  - Multi-core: shared memory
  - Multi-node: distributed memory

Levels of parallellism

- 3 levels of parallellism:
  - Intra-core: pipelining and SIMD
  - Multi-core: shared memory
  - Multi-node: distributed memory
- Who does the work?
  - Compiler
    - but it appreciates/needs your help
  - You and the compiler
    - OpenMP = directives
    - Relatively simple
  - You only
    - MPI
    - Harder

Case study – Lennard-Jones system

1. Monte Carlo setting:
   - compute the energy of configurations
   - ensemble averages
   - no (individual) forces, accelerations or velocities
   - no time integration

2. Molecular Dynamics setting:
   - compute the time evolution of a collection of atoms
   - Individual forces, accelerations, velocities
   - [time integration]

MC experiment

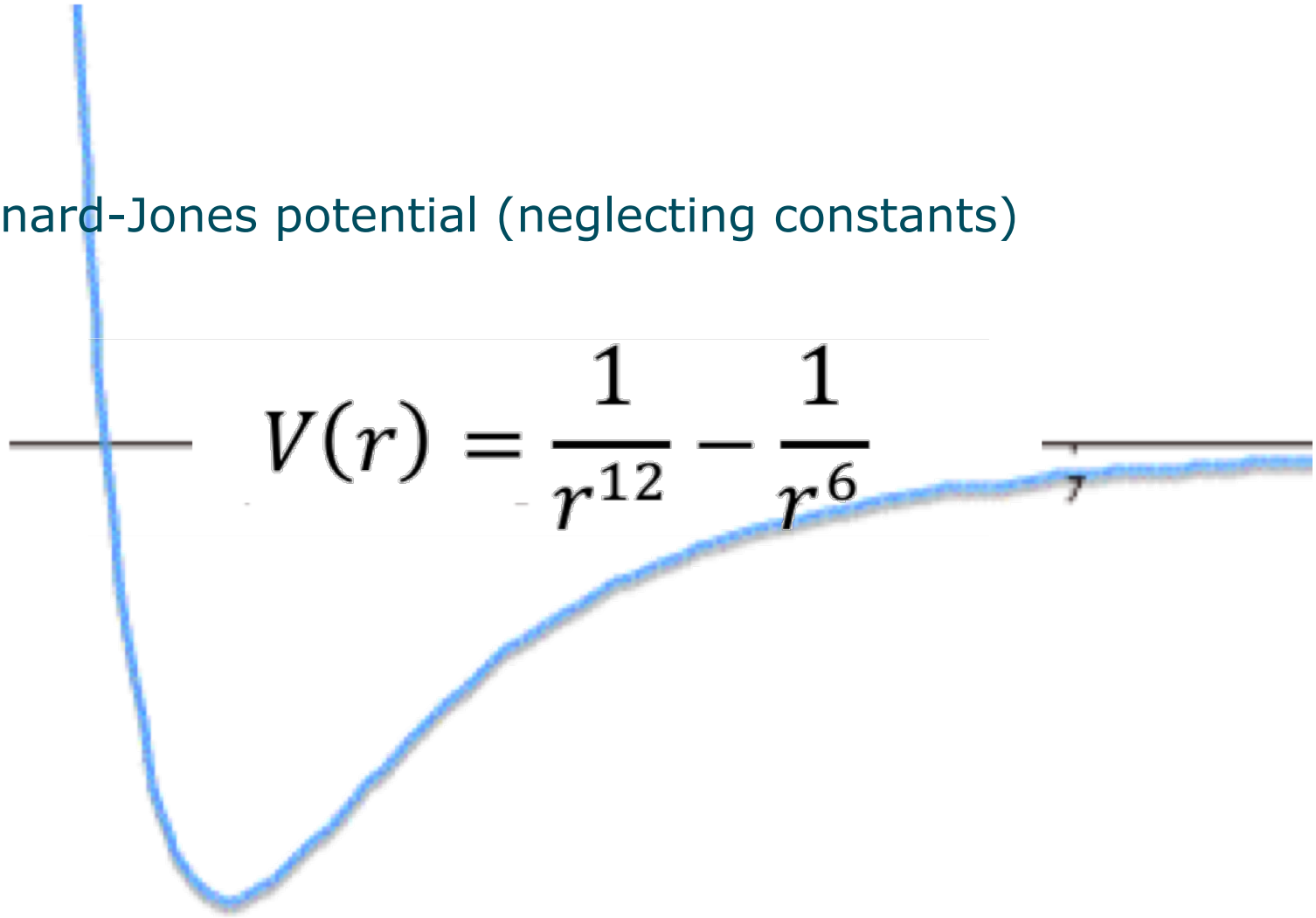
- Compute the potential energy of a system of N atoms: $\sum_{i<j} V_{LJ}(r_{ij})$
- Doing this for all atoms is a $O(N^2)$ approach
  - we do it only for a single atom just for the purpose of illustrating the behavior of the hardware (toy problem)
  - $\sum_j V_{LJ}(r_{ij})$
- Techniques to reduce to $O(N)$ will be discussed in MD setting

- We will only consider **single core** performance
  - That is the first thing to optimize anyway
  - Already complex enough for a single lecture
  - SIMD and pipelining are the only level of parallelism

Lennard-Jones interactions

- Lennard-Jones potential (neglecting constants)

$$V(r) = \frac{1}{r^{12}} - \frac{1}{r^{6}}$$



Lennard-Jones potential
(sequential code)

```
double VLJ0( double r ) {
    return 1./pow(r,12) - 1./pow(r,6);
}                                                    // 18.0 x slower
double VLJ1( double r )_{
    return std::pow(r,-12) - std::pow(r,-6);
}                                                    // 14.9 x slower
double VLJ2( double r ) {
    double tmp = std::pow(r,-6);
    return tmp*(tmp-1.0);
}                                                     // 7.8 x slower
double VLJ3( double r ) {
    double tmp = 1.0/(r*r*r*r*r*r);
    return tmp*(tmp-1.0);
}                                                    // 1.01 x slower
double VLJ( Real_t r ) {
    double rr = 1./r;
    rr *= rr;
    double rr6 = rr*rr*rr;
    return rr6*(rr6-1);
}                                                      // 1 x slower
```

Lennard-Jones potential

```
double VLJ( Real_t r ) {
    double rr = 1./r;
    rr *= rr;
    double rr6 = rr*rr*rr;
    return rr6*(rr6-1);
}                                                    // 1 x slower
double VLJ( Real_t r2 ) {
    double rr = 1./r2;
 // rr *= rr;
    double rr6 = rr*rr*rr;
    return rr6*(rr6-1);
}          // avoid one sqrt per function call (to compute the distance r)
```

Cost of instructions

• cheap instructions	+,-,*	~1 cycle
• Rather expensive	/	~10-20 cycles
• Expensive	sqrt	~35 cycles
• Very expensive	trigonometric/ logarithmic/ exponential functions	~100-200 cycles

- Things get better if pipelining can be exploited
- Relative cost remains

MC experiment

- x0, y0, z0 : coordinates of our central atom
- x1[1:m], y1[1:m], z1[1:m] : coordinates of m neighbouring atoms
- Let m=512, 1012, …, $2^{29} \sim 0.5*10^9$
- surround by outer loop iterating $2^{29}/m$ times
  - every m-case executes 2^{29} evaluations of $V_{LJ}(r^2)$
- Variations
  - Loop over all m neighbouring atoms **contiguously**
    - Structure of arrays (SoA) : x x x ... y y y ... z z z ...
    - Array of structures (AoS) : x y z ... x y z ... x y z ...
  - Pick atoms in the arrays x1, y1, z1 with a random permutation

MC - Contiguous - SoA

```
integer :: m ! # of neighbour atoms
integer :: k ! # of iterations, m*k=cst, same amount of work per iteration
real(wp) :: x0,y0,z0, p(3*m)
! Contiguous access, SoA: p=[xxx…yyy…zzz…]
do ik=1,k
        do im=1,m
            r2 = (p(im)-x0)**2 +(p(m+im)-y0)**2 +(p(2*m+im)-z0)**2
            v = v + lj_pot2(r2)
        enddo
enddo
```

1	x					y					z				
2		x					y					z			
3			x					y					z		

MC - Contiguous - AoS

```
integer :: m ! Number of neighbour atoms
real(wp) :: x0,y0,z0, p(3*m)
! ordered access, AoS: p=[xyzxyzxyz…]
do ik=1,k
    do im=1,m
        r2 = (p(im)-x0)**2 +(p(1+im)-y0)**2 +(p(2+im)-z0)**2
        v = v + lj_pot2(r2)
    enddo
enddo
```

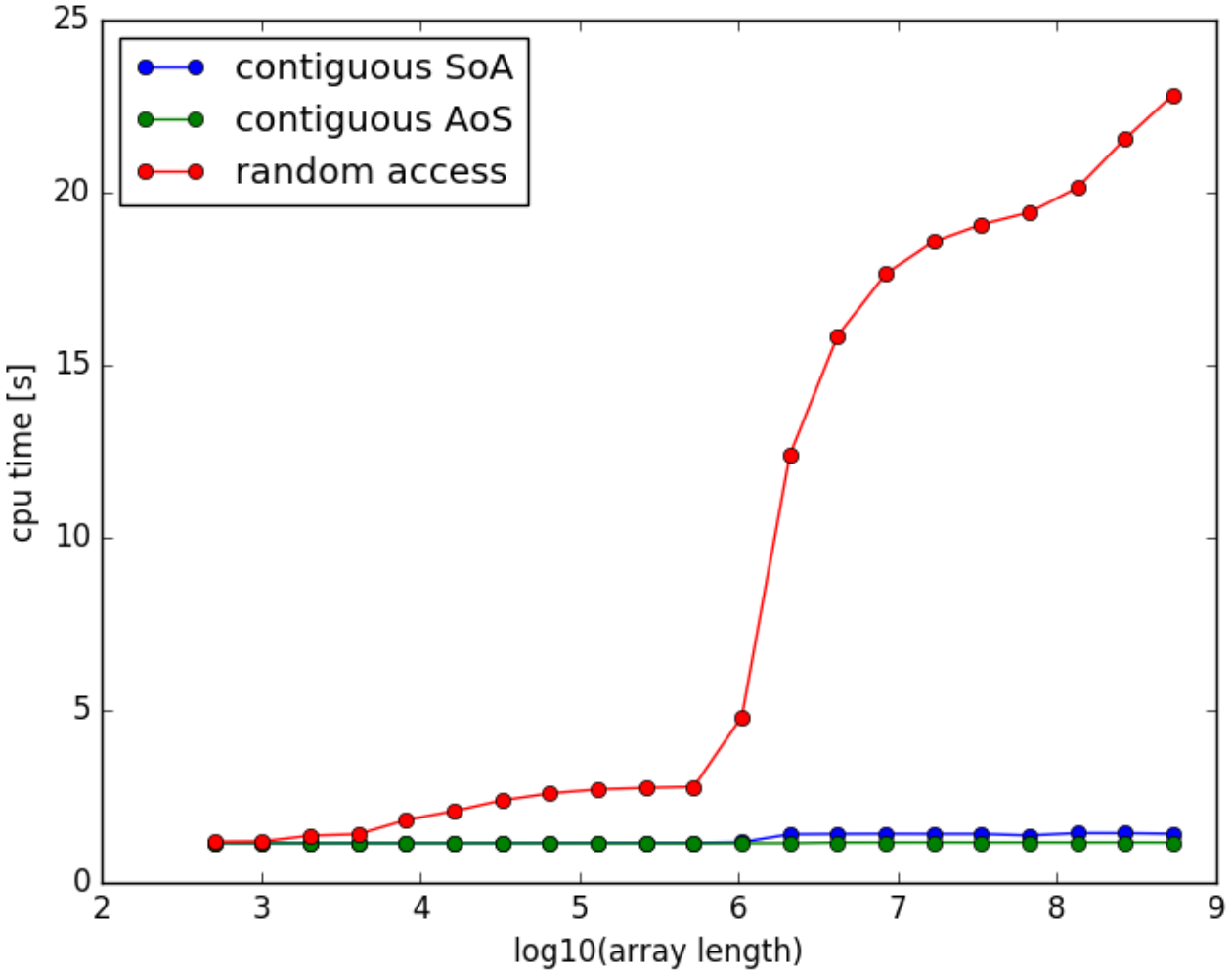
1	x	y	z												
2				x	y	z									
3							x	y	z						

MC - Random access

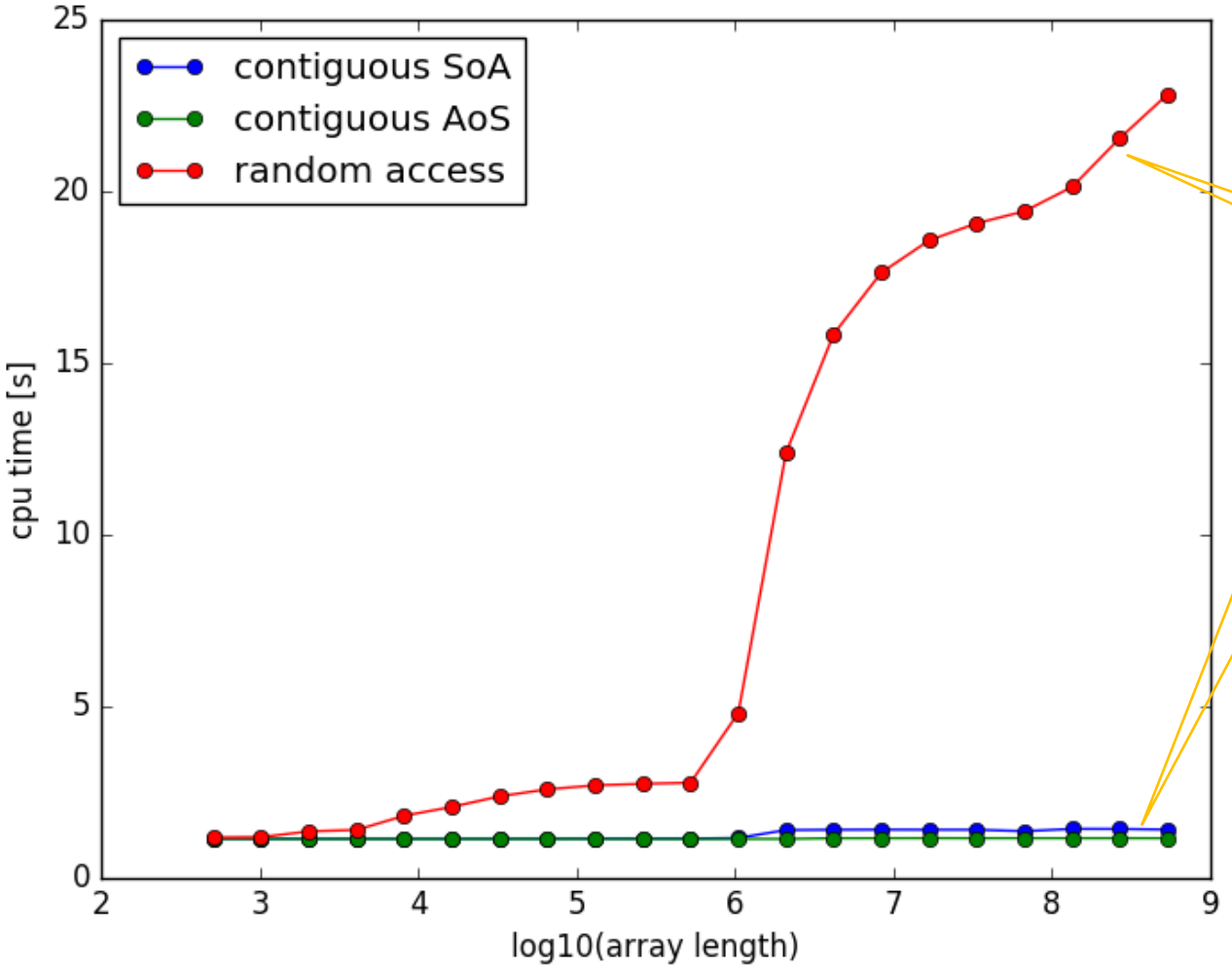
```
integer :: m ! Number of neighbour atoms
real(wp) :: x0,y0,z0, p(3*m)

integer :: j(m) ! random permutation of [1:m]

! random access
do ik=1,k
    do im=1,m
        r2 = (p(j(im))-x0)**2 +(p(j(im)+m)-y0)**2 +(p(j(im)+2*m)-z0)**2
        v = v + lj_pot2(r2)
    enddo
enddo
```

1			x					y					z		
2		x					y					z			
3					x					y					z

Results

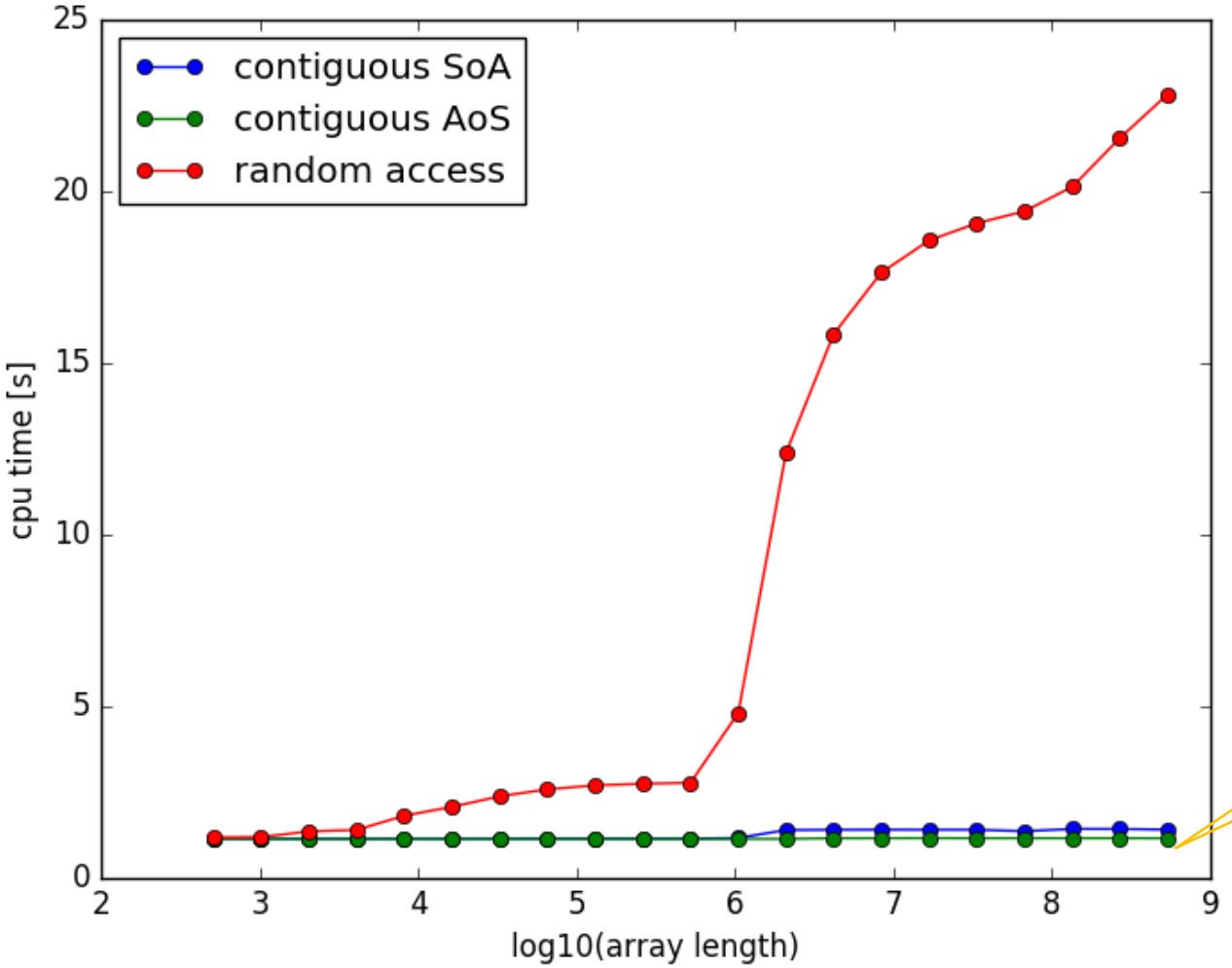

contiguous SoA
contiguous AoS
random access
cpu time [s]
log10(array length)
0
5
10
15
20
25
2
3
4
5
6
7
8
9

Results



Q1: What is going on?

Results



Q2: is this the best we can get?

Performance

- Which factors influence performance of a code?
- Machine limits

Peak performance

- $\stackrel{\text{def}}{=}$ Maximum # floating point operations per second

- For a single core the peak performance =
  - 2*8 instructions per cycle in SP
  - 2*4 instructions per cycle in DP
    - The 2 comes from the fused multiply and add
    - The 8, resp. 4 come from the vector register width

- Peak performance per node
  - (1 cycle = 1/clock_frequency)
  - Assuming 1 hardware thread per core:
  - (#cores=20) * 2*8(SP) * (f=2.8Ghz) = 896 Gflop/s
  - (#cores=20) * 2*4(DP) * (f=2.8Ghz) = 448 Gflop/s

Performance

- Peak performance is not the only limiting factor…
- It is not the most common limiting factor
- Instructions operate on data, ...
- Data resides in memory
- Accessing data takes time (and energy)
  - Data has to be moved from memory to cpu register before it can be processed
- Peak performance has increased much faster than the speed at which data can be moved between memory and cpu

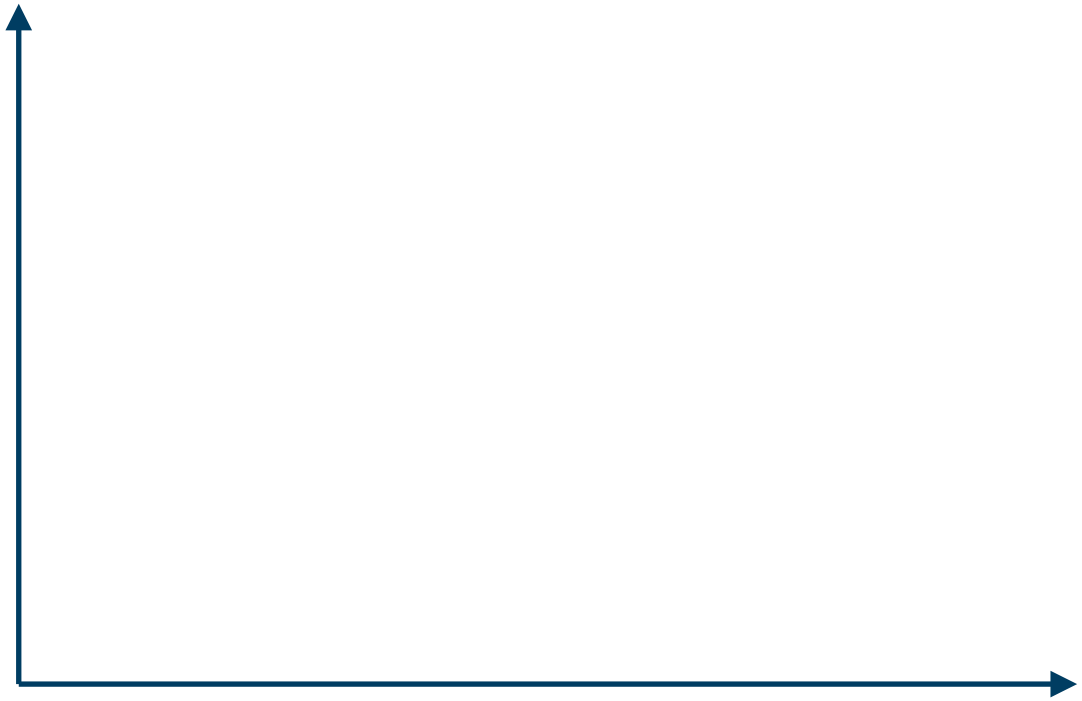
Speed of moving data

- Memory **bandwidth**
  - The maximum number of bytes that can be moved per second between main memory and the cores
  - Hopper
    - 92-110 GB/s (varies depending on read:write ratio)
- Memory **latency**
  - The number of cycles (or the time) needed to fetch a single item from main memory
  - Hopper
    - ~180 cycles (within socket)
    - ~350 cycles (across sockets)

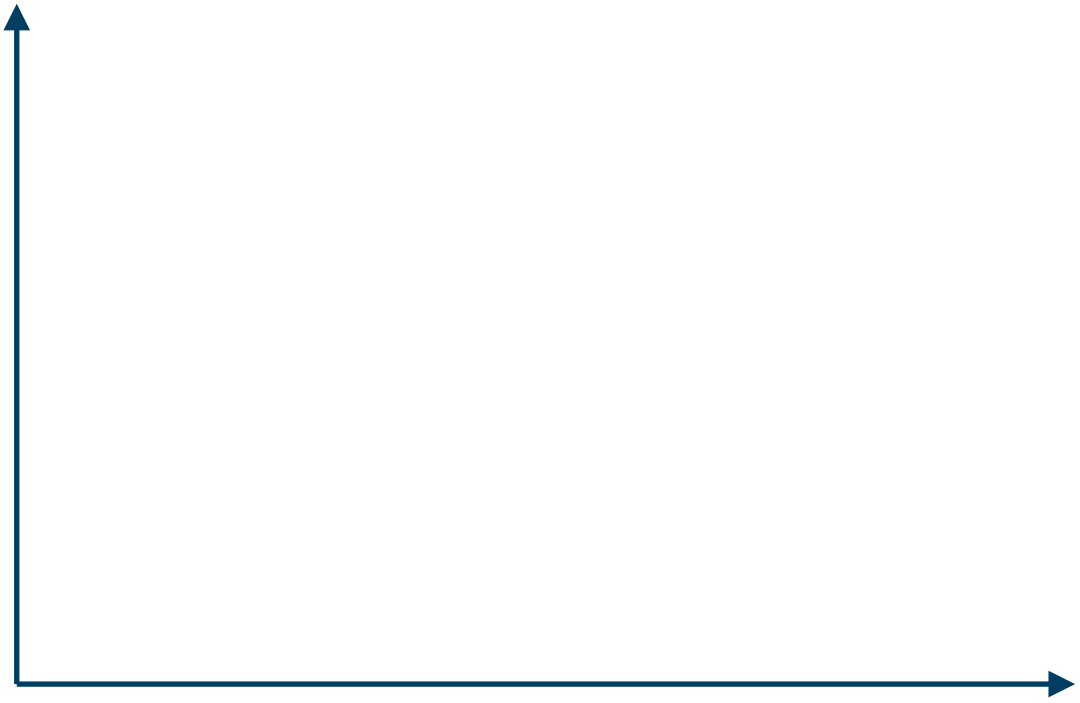
Performance

- Code is **compute bound** if
  - The cpu can execute its compute instructions without having to wait for data
  - The limit is the theoretical peak performance
  - [Used to be the common case – not any more]
- Code is **memory bound** if
  - A considerable amount of cycles is spent waiting for data
  - Too much data requested:
    - **Bandwidth saturation** = machine limit
  - Too distant data requested:
    - If data is not in the cache: latency penalty
    - Latency problem = machine limit
  - [most common situation]

Roofline model

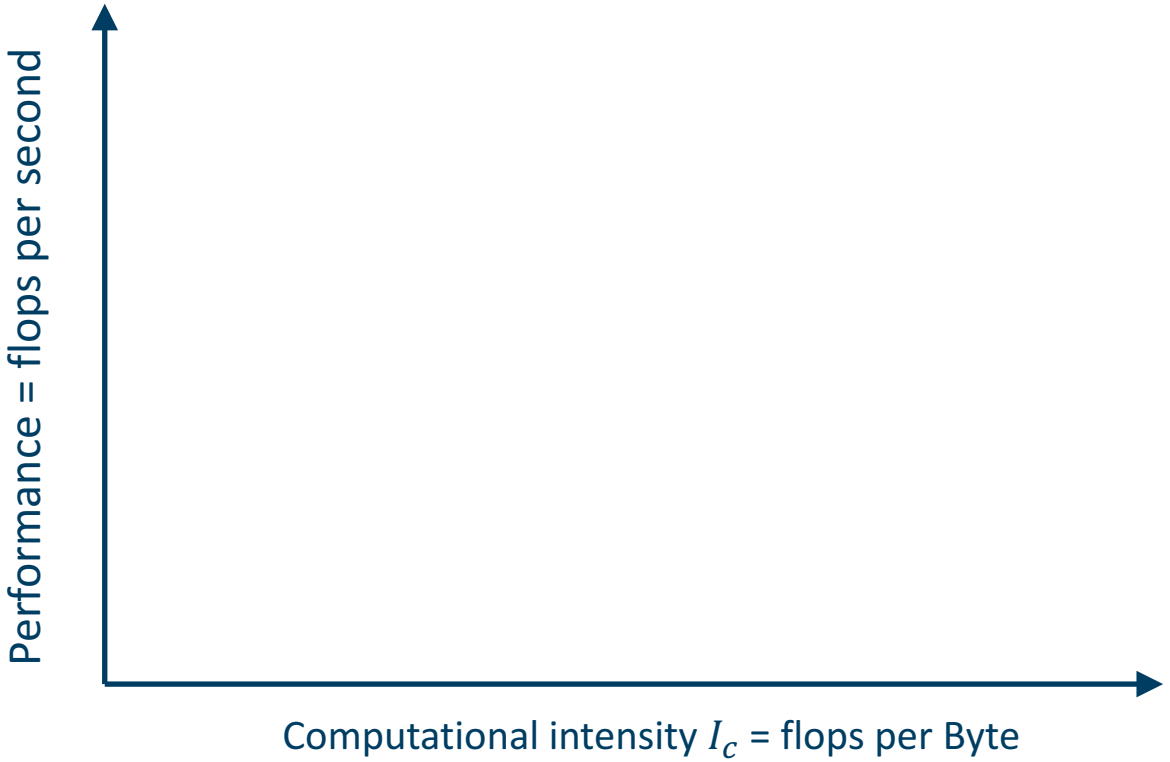


Roofline model

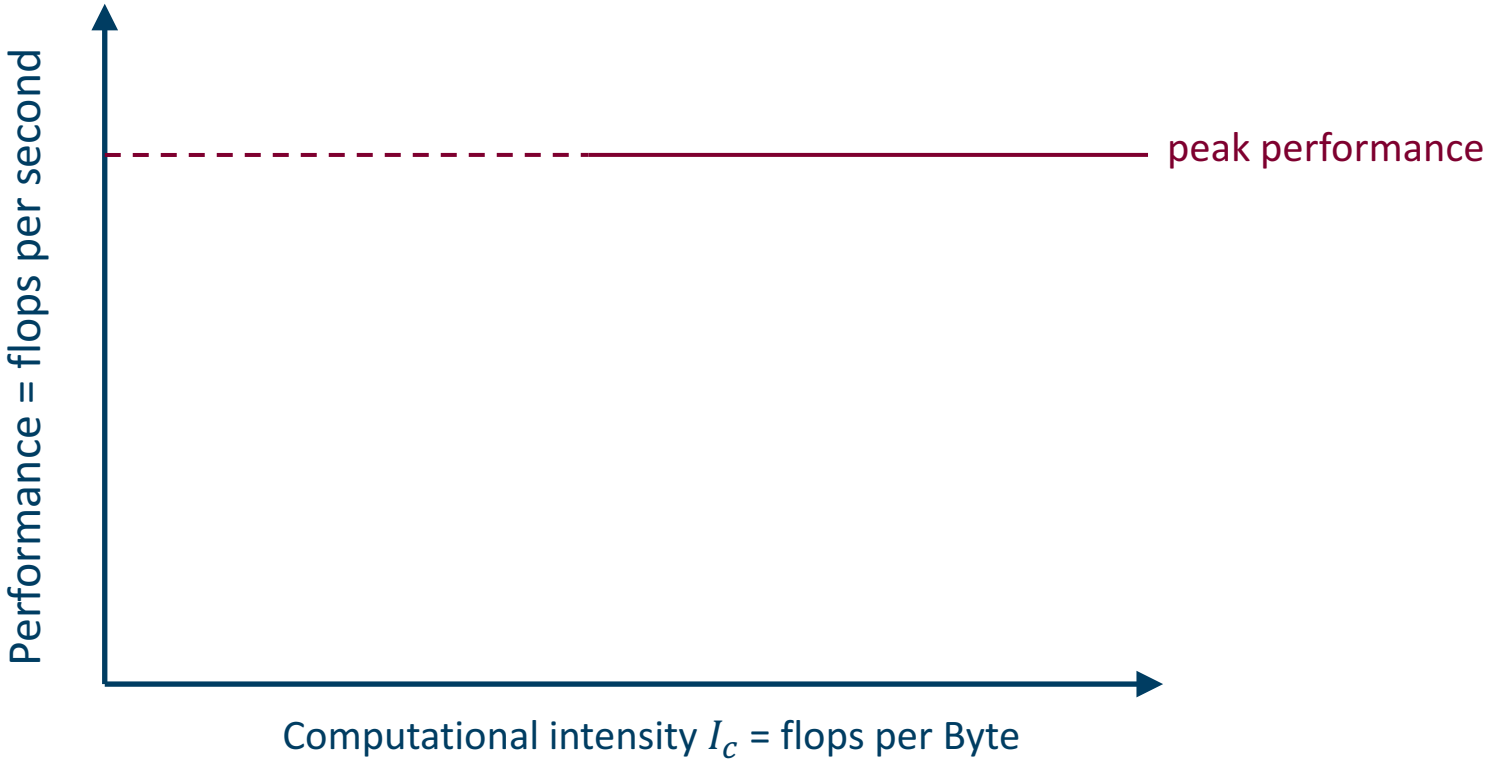


Computational intensity I_c = flops per Byte

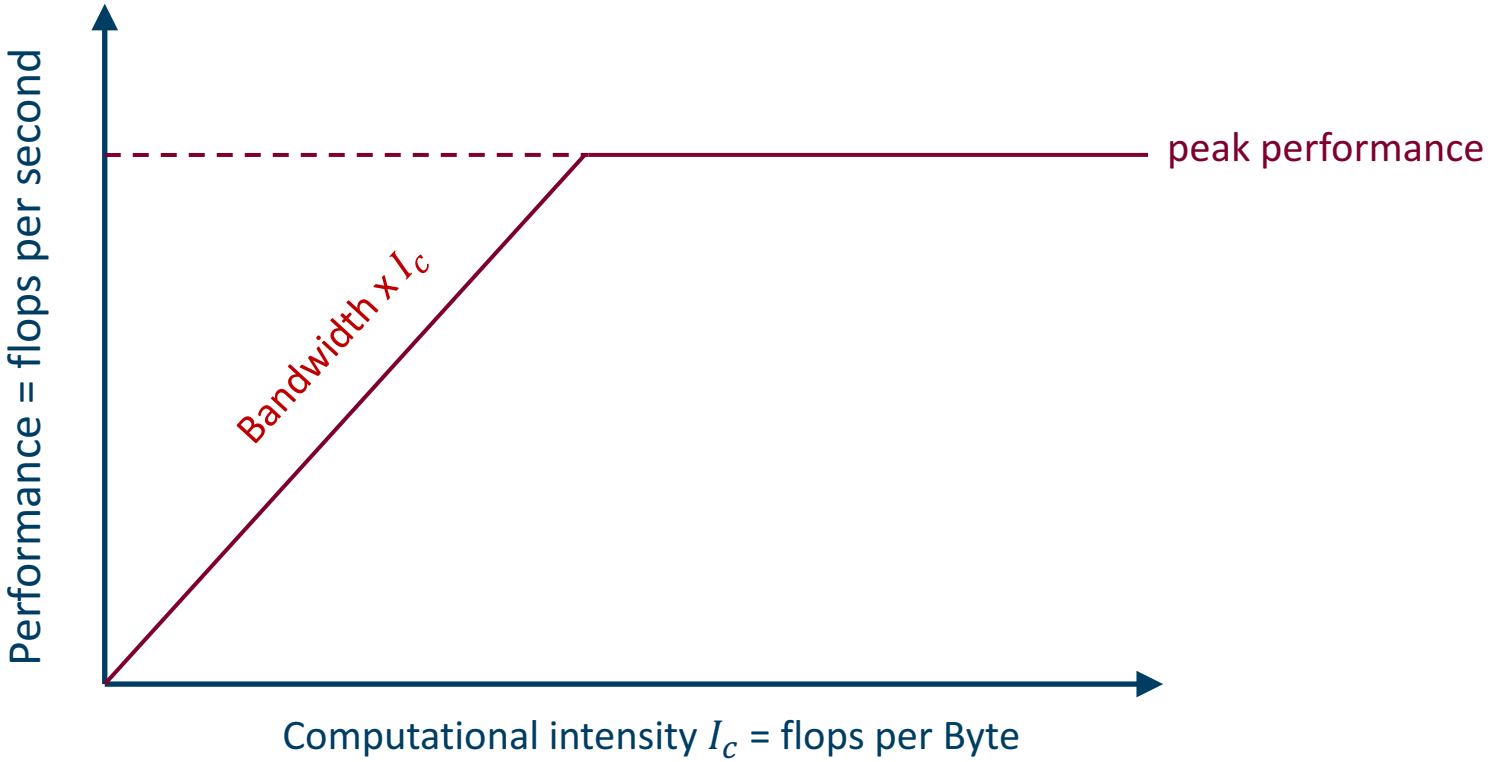
Roofline model



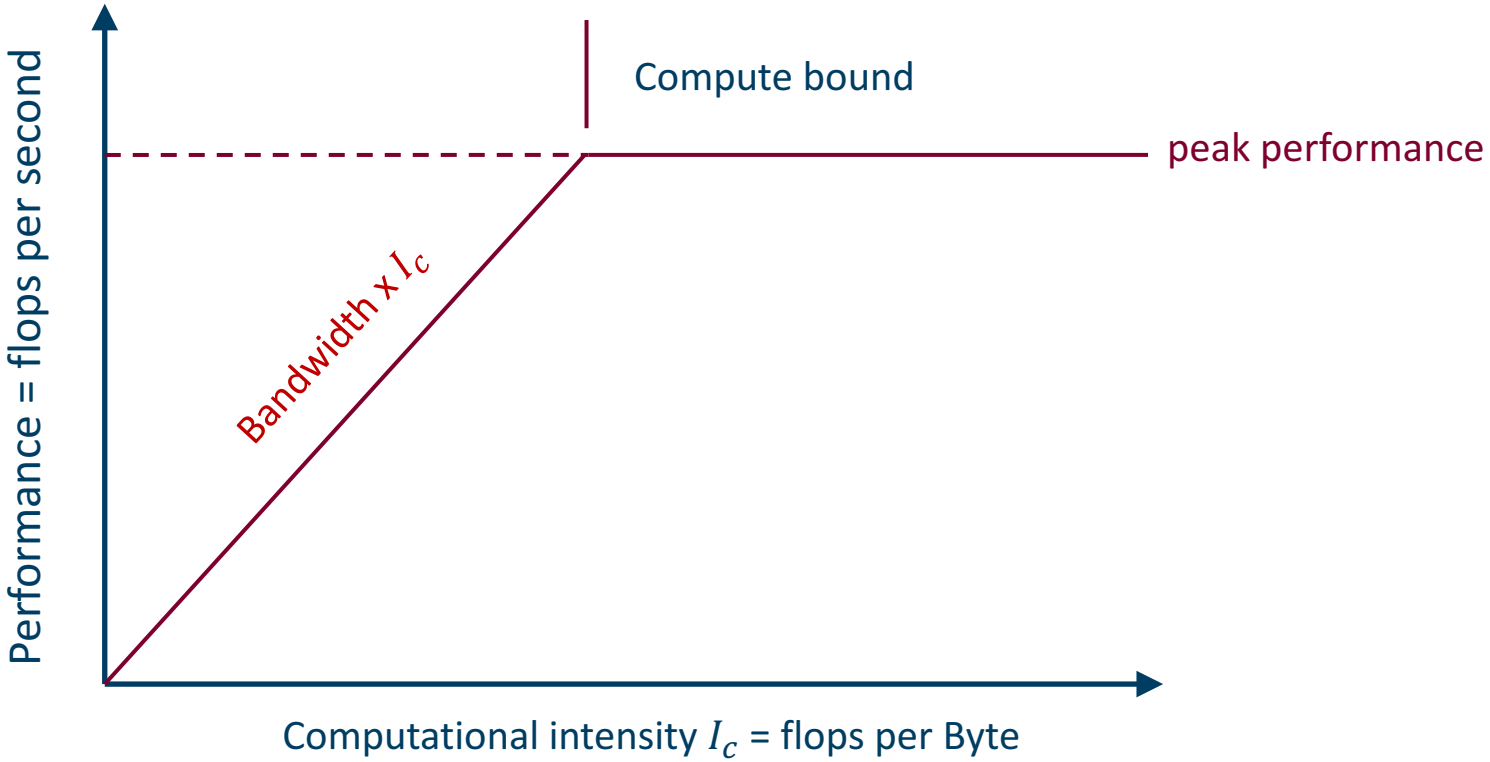
Roofline model



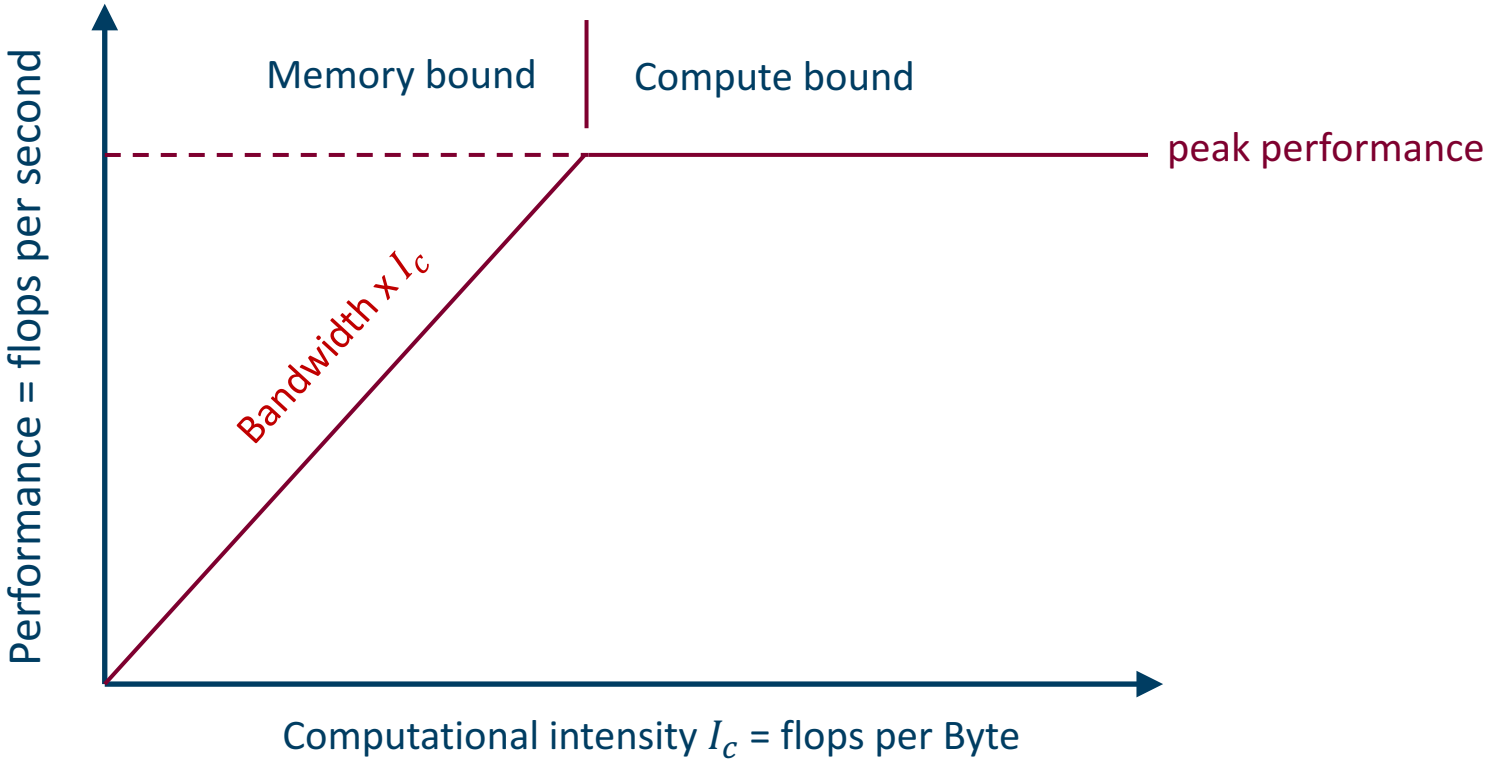
Roofline model



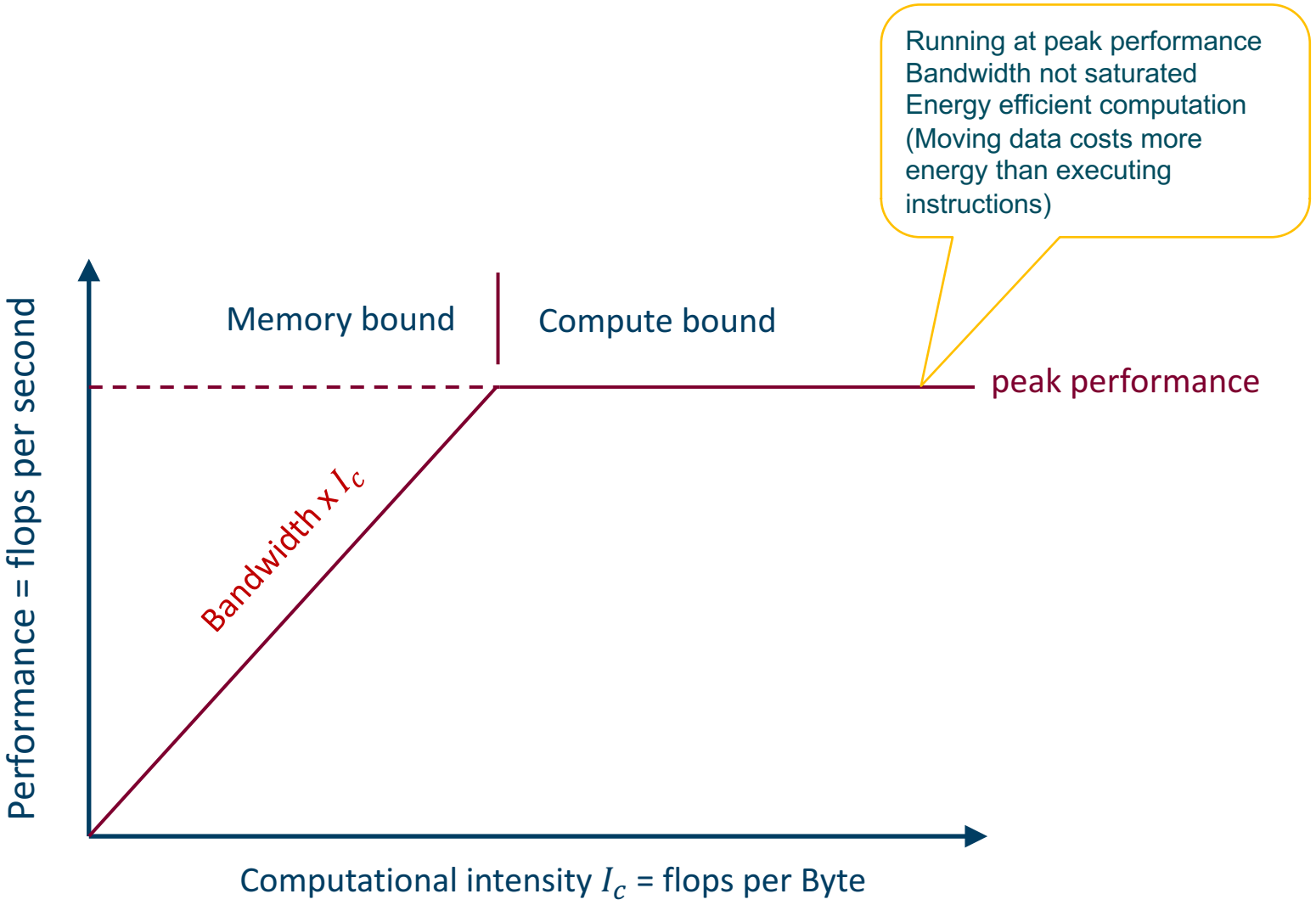
Roofline model



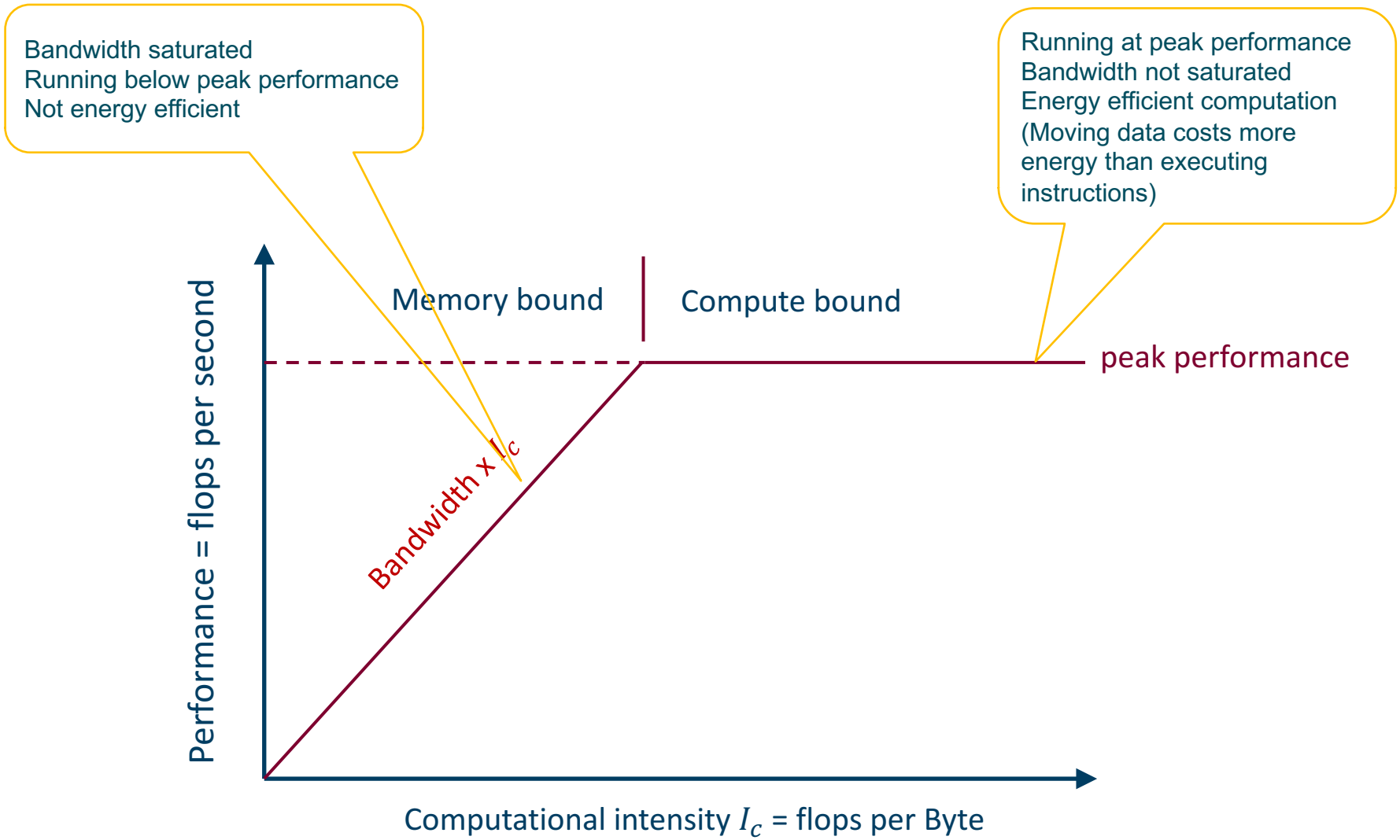
Roofline model



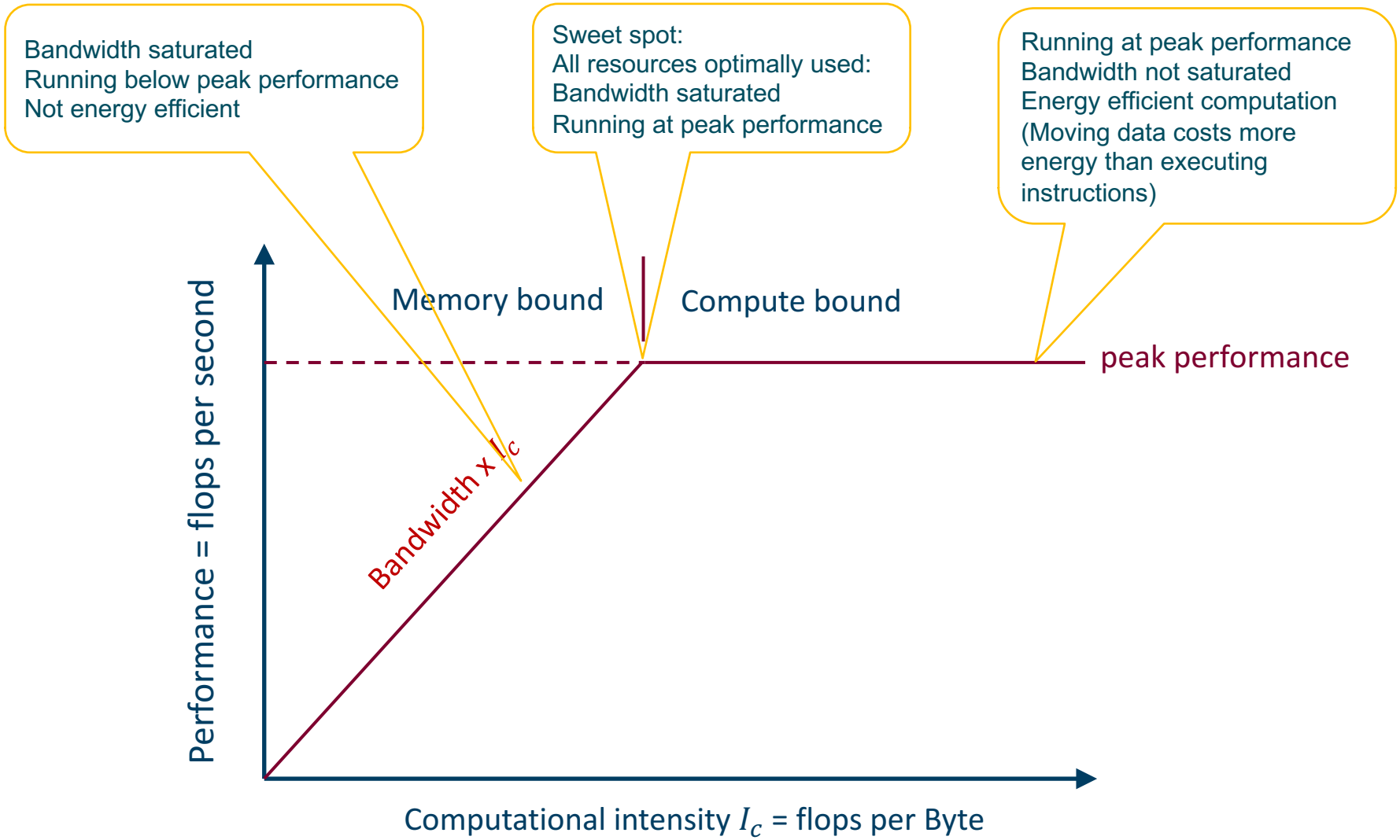
Roofline model



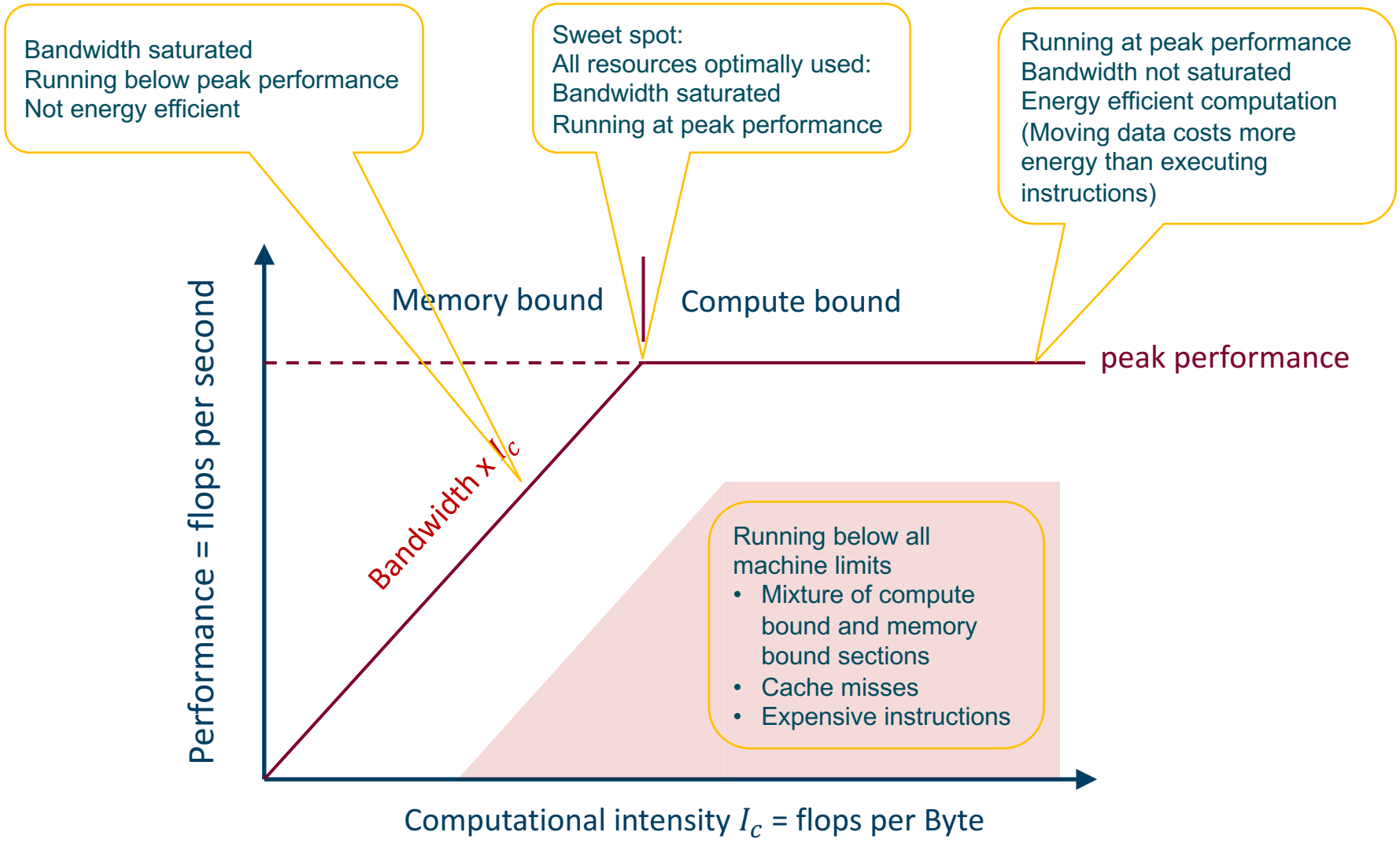
Roofline model



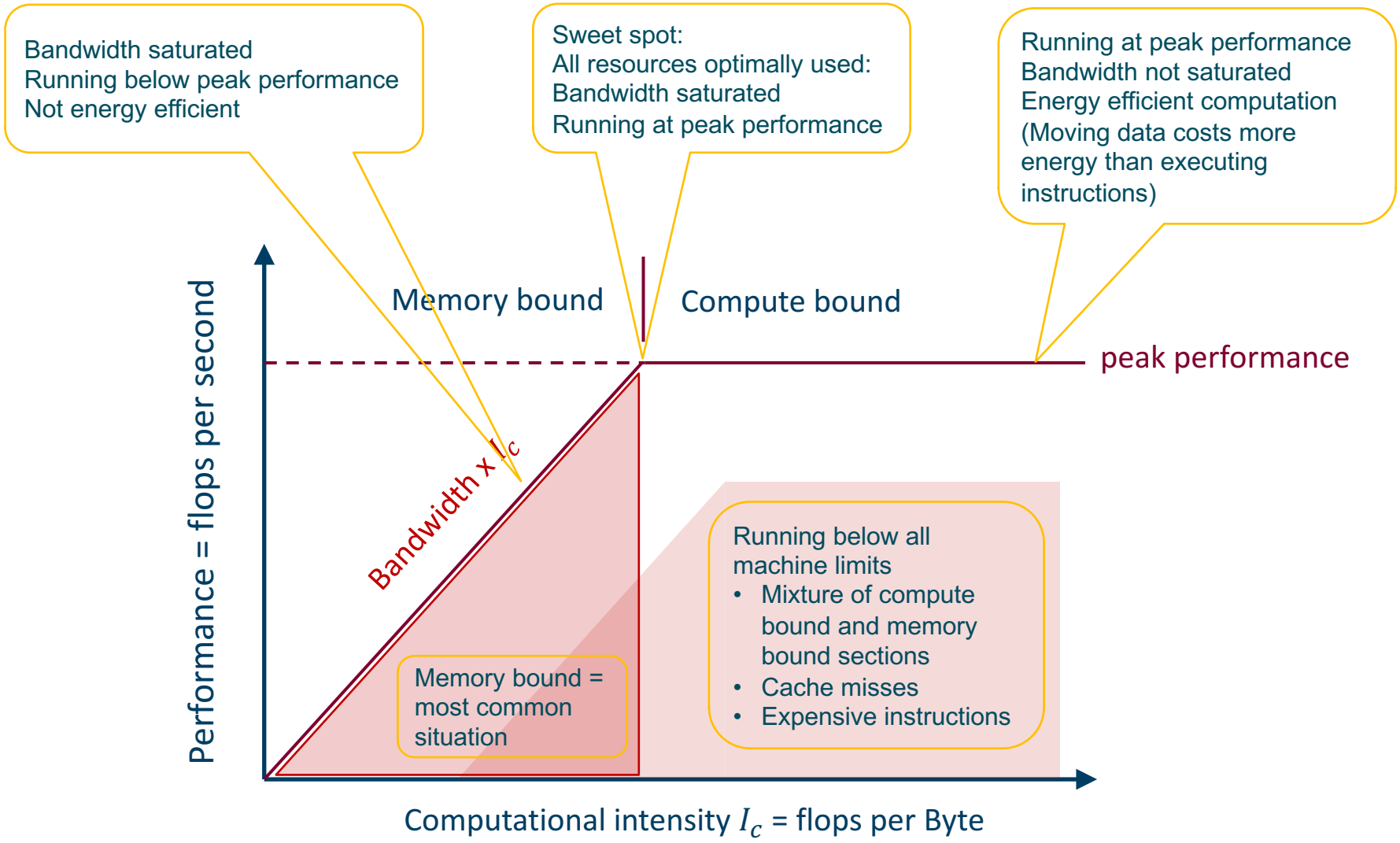
Roofline model



Roofline model



Roofline model



Memory bound - consequences

- Optimizing code **was** about organizing compute instructions
  - Pretty straightforward: less compute cycles is less cputime
  - Algorithmic complexity was important guideline
- Optimizing code is optimizing data access
  - To keep the processor busy doing useful stuff
  - **Algorithmic complexity is no longer a guarantee** for optimal performance
    - E.g. linear search (as in a map) often faster than binary or other search algorithms, also sorting
    - For large N low order complexity wins, but hardware caching takes an early lead
- Understanding how memory works is necessary
- Experimenting and measuring is necessary

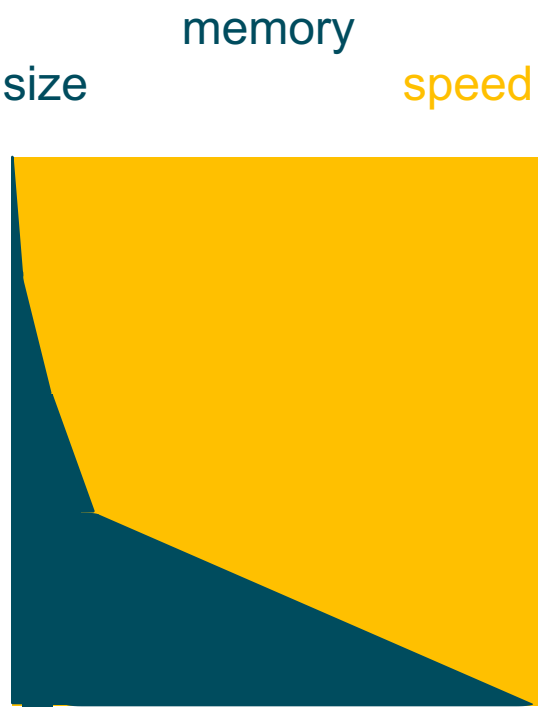
Hierarchical memory organisation

On-chip (E5-2680 v2)

ALU	
Registers: ~1kB per core	0 cycles
L1 Cache: 32 kB per core	~1 cycles
L2 Cache: 256 kB per core	~10 cycles
L3 Cache: 25 MB per socket	~50 cycles

Off-chip

DRAM: 64-256 GB per node	~200 cycles



Memory organization

- Memory is not fetched on a per item basis
- But in chunks called **cache lines**
  - typically 64 Bytes long
    - 16 single precision items
    - 8 double precision items

Memory organisation

- **Linear search** of array A[i], i=1..n
  - A[1] is not in cache, wait time before cache line is loaded, dram latency (~200 cycles) and before item A[1] can be examined
  - Once the cache line is loaded, A[2..16] are also in L1 cache and are ready to be examined without delay
  - Hardware recognizes your loop over the array and keeps loading next (or previous) cache lines into the L1 cache, so that the delay is vanishing
  - Depending on how much work it takes to examine each item, as soon as item A[16], the next cache line A[17:32] may have been loaded already or not
  - In any case, the wait time is now less than the dram latency (~200 cycles)
  - The limitation becomes memory bandwidth of the machine

Memory organisation

- **Binary search** of array A[i], i=1..n
  - A[n/2] is not in cache, wait time before cache line is loaded, dram latency
  - Next item needed is A[n/4] or A[3n/4], which is not in the cache, dram latency hits you again
  - In fact, the dram latency keeps on hitting you until the search range is reduced to one or two cache lines,
  - You do only one examination/dram latency, as opposed to 16/dram latency in linear search.

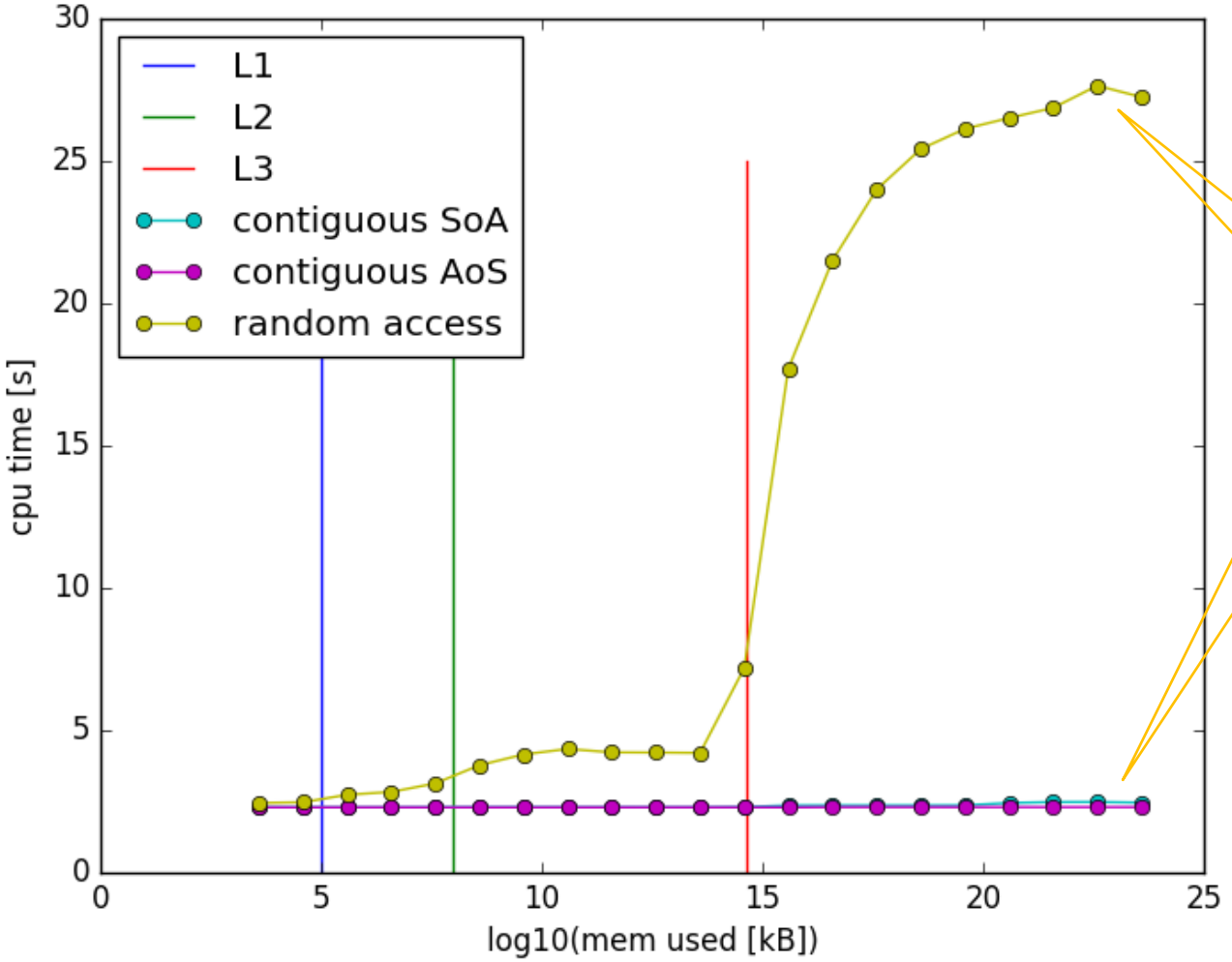
Memory organization

- Instructions are also data stored in memory
- Branching instructions can cause cache misses too!
  - Instruction cache misses
- Avoid unpredictable branches in loops

Need more insight?

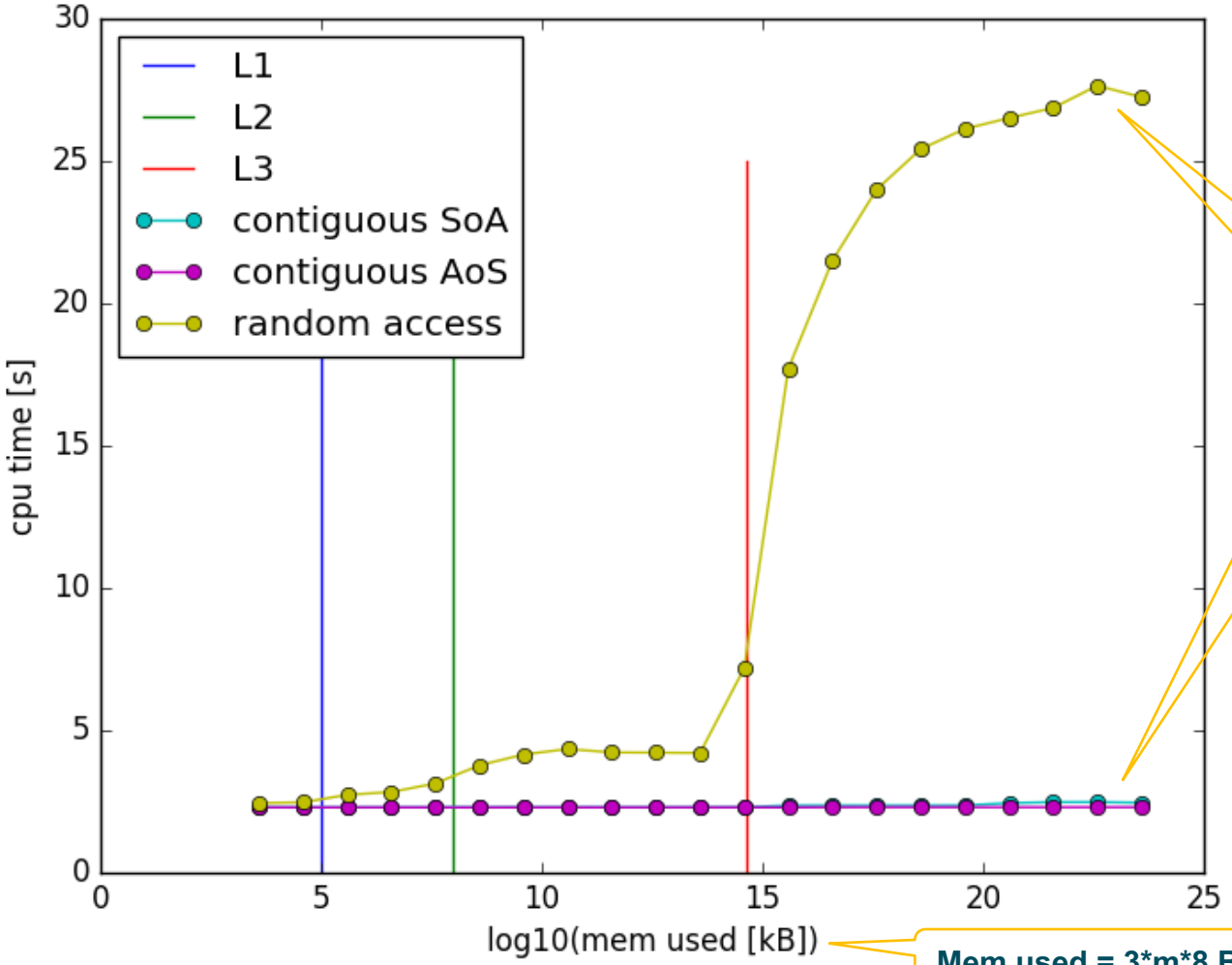
- code::dive conference 2014 - Scott Meyers: Cpu Caches and Why You Care
- https://www.youtube.com/watch?v=WDIkqP4JbkE

Experiment



Q1: What is going on? 15x slower

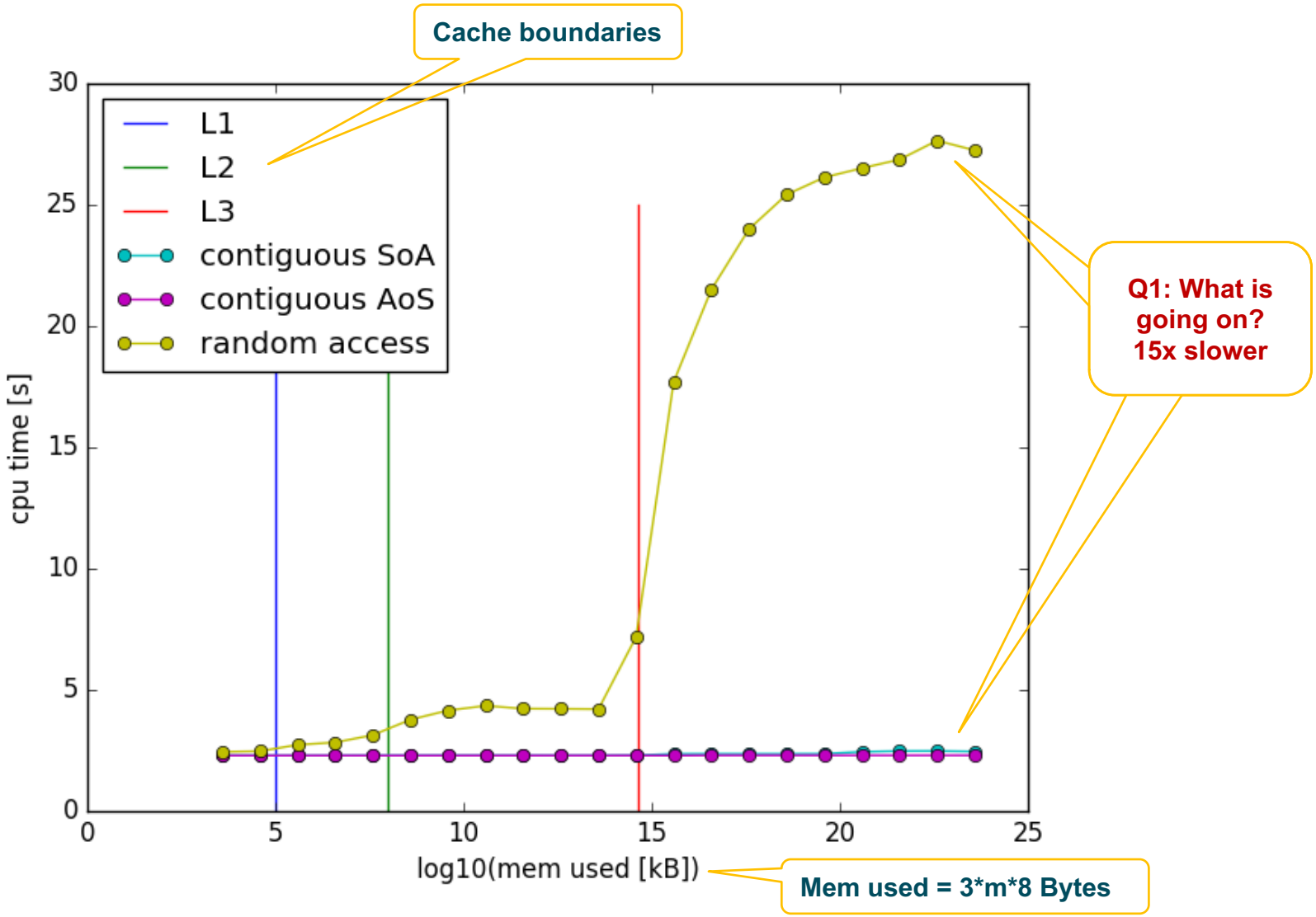
Experiment



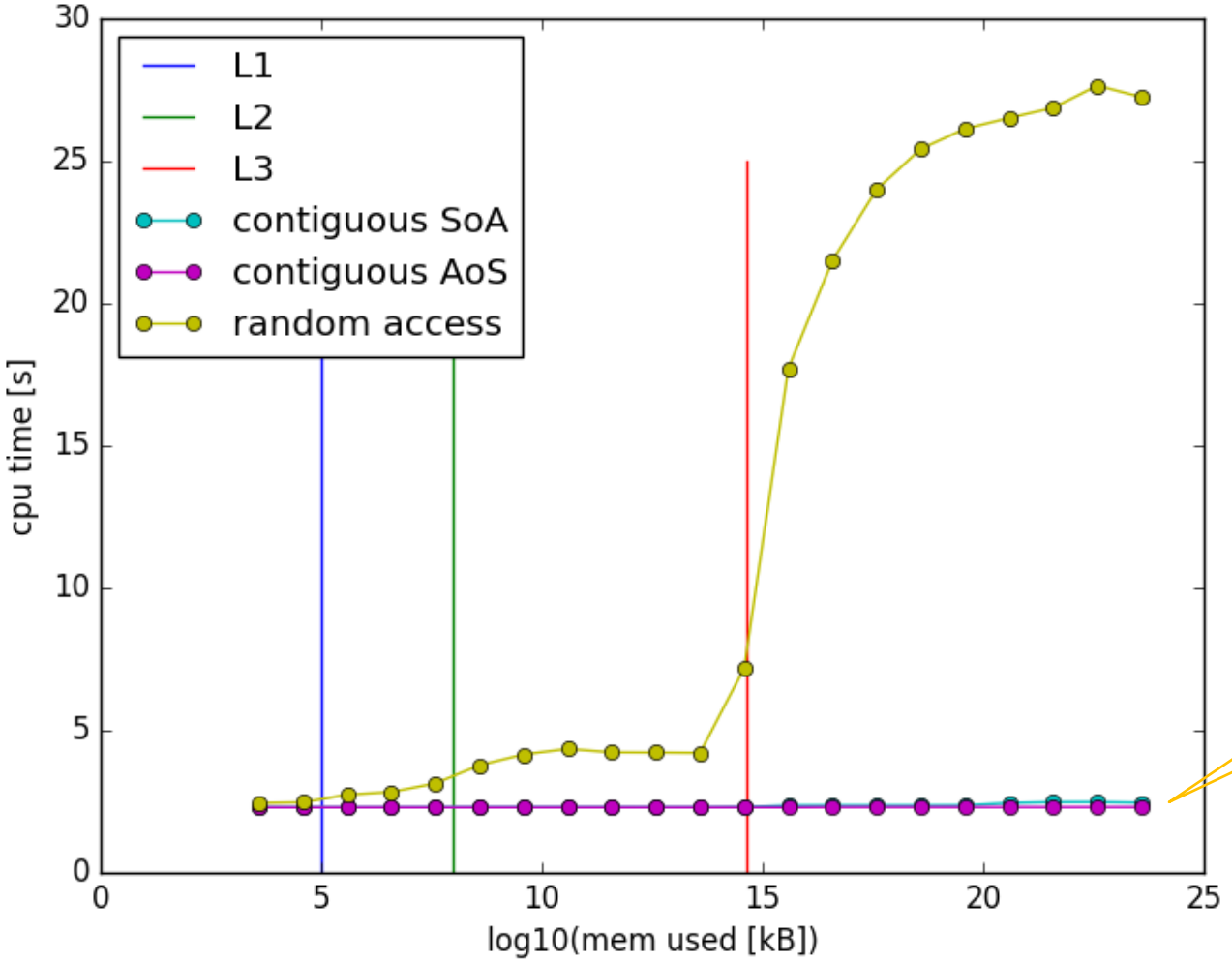
Q1: What is going on? 15x slower

Mem used = 3*m*8 Bytes

Experiment


Cache boundaries
L1
L2
L3
contiguous SoA
contiguous AoS
random access
cpu time [s]
log10(mem used [kB])
0
5
10
15
20
25
30
Q1: What is going on? 15x slower
Mem used = 3*m*8 Bytes

Experiment



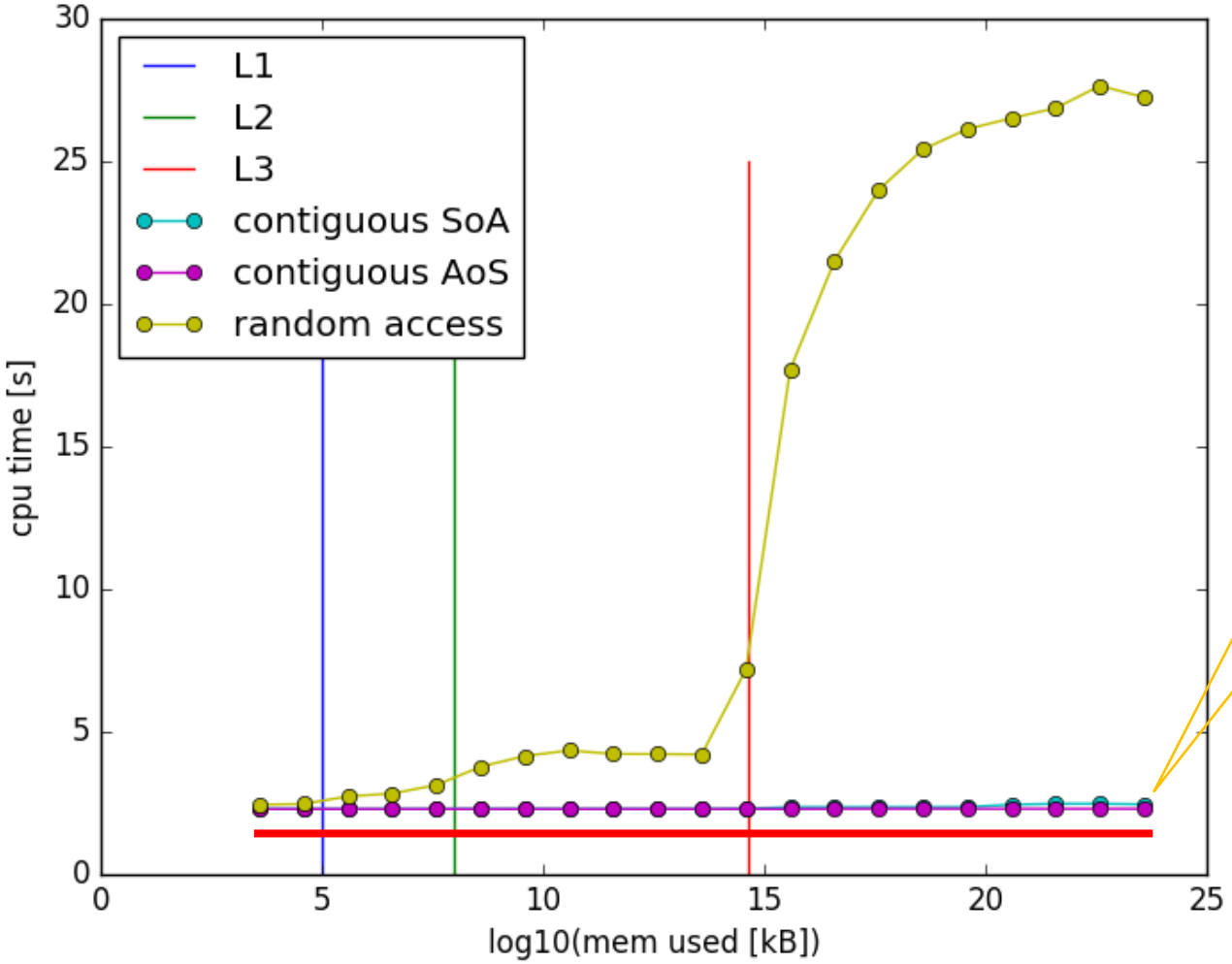
Q2: is this the best we can get?

Flops per second

```
! Contiguous access, SoA: p=[xxx…yyy…zzz…]
do ik=1,k
      do im=1,m
         r2 = (p(im)-x0)**2
             +(p(m+im)-y0)**2
             +(p(2*m+im)-z0)**2          · 3-, 2+, 3*
!        r = lj_pot2(r)
         r2i = 1.0d0/r2                  · 1/
         rr6i = r2i*r2i*r2i;             · 2*
         lj_pot2 = 4.0d0*rr6*(rr6-1.0d0); · 2*, 1-
      enddo                              · ----------
enddo                                    · 14 flops
```

- 14 flops * 2^{29} iterations in 1.2 s = 6.26 10^9 flops/s
- peak performance:
  1*1*4*2.8 GHz = 11.2 Gcycles/s = 11.2 Gflops/s
- We are running at 55.9 % of peak performance

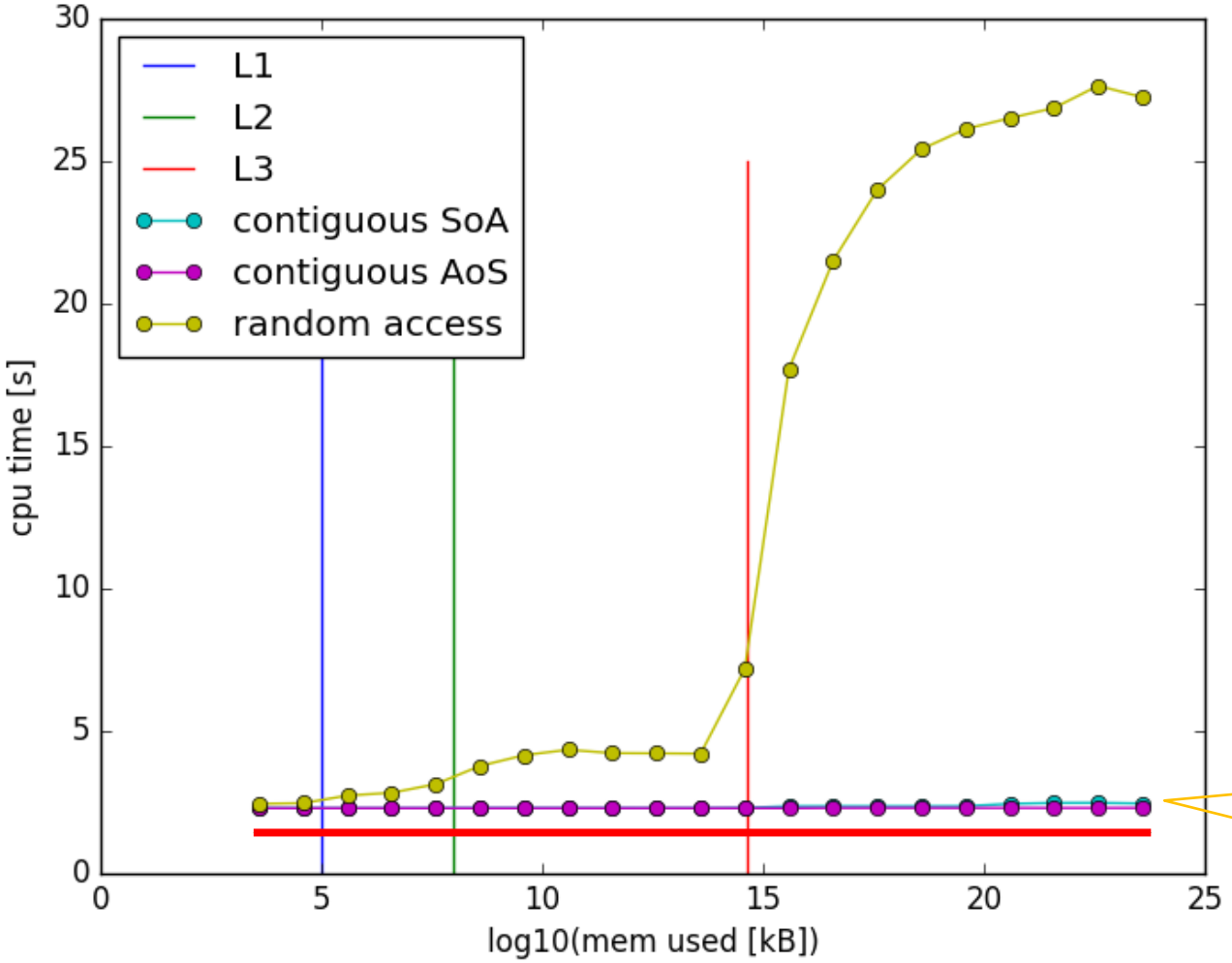
Experiment



Q2: is this the best we can get?

Not really, there is 44% room for improvement

Experiment



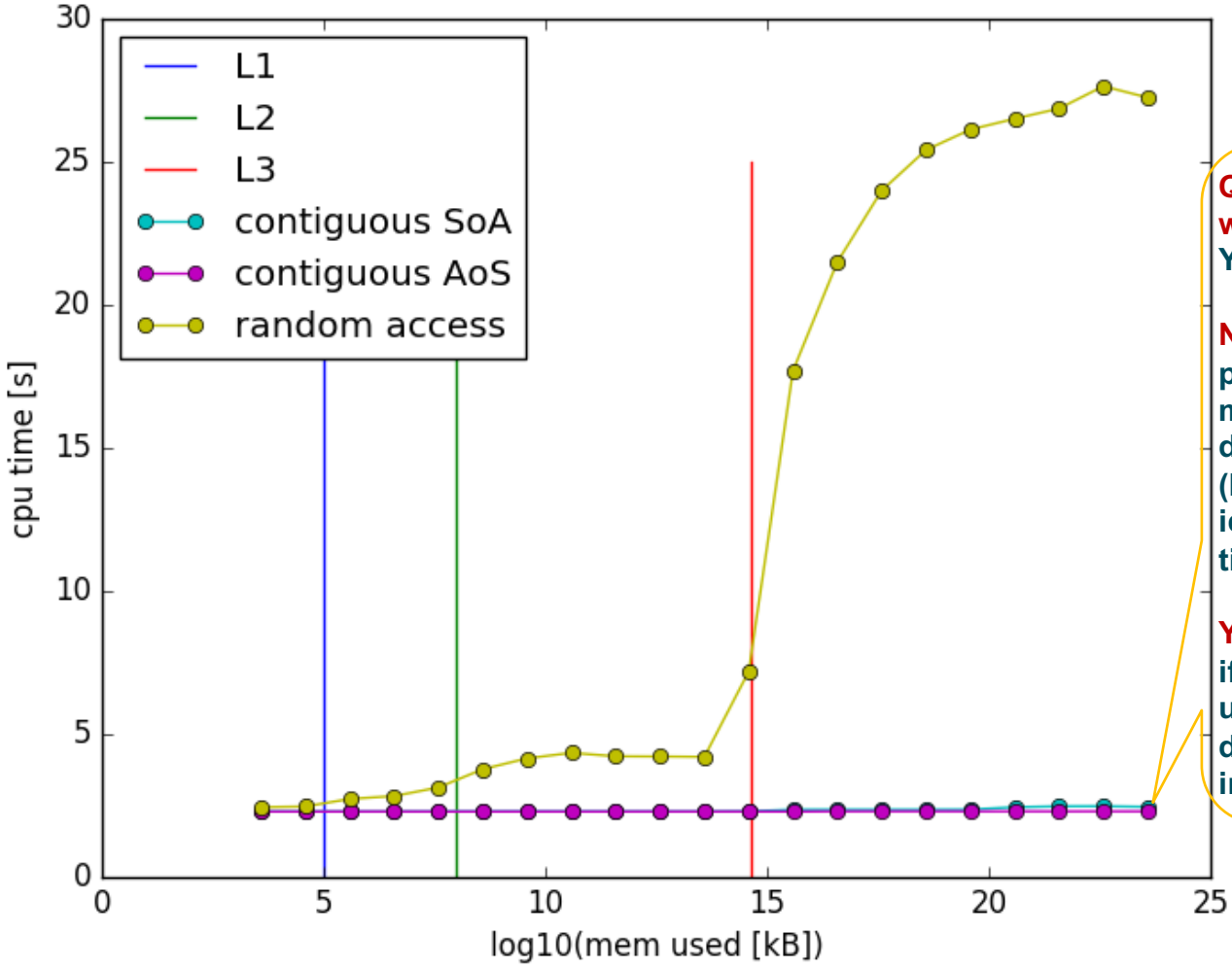
Q3: why aren't we running faster

Bytes per second

```
! Contiguous access, SoA: p=[xxx…yyy…zzz…]
do ik=1,k
      do im=1,m
         r2 = (p(im)-x0)**2
             +(p(m+im)-y0)**2
             +(p(2*m+im)-z0)**2          · 3-, 2+, 3*   · 3 DP
!        r = lj_pot2(r)
         r2i = 1.0d0/r2                  · 1/
         rr6i = r2i*r2i*r2i;             · 2*
         lj_pot2 = 4.0d0*rr6*(rr6-1.0d0); · 2*, 1-
      enddo                              · ----------   · ----------
enddo                                    · 14 flops     · 24 B
```

- 24 B *2^{29} iterations in 1.2 s = 10.7 GB/s
- Bandwidth measured by Intel mlc:
  - 109 GB/s for 10 threads (all reads)
  - 10.9 GB/s for 1 thread
- We are running at maximal bandwidth
  - Bandwidth saturation

Experiment

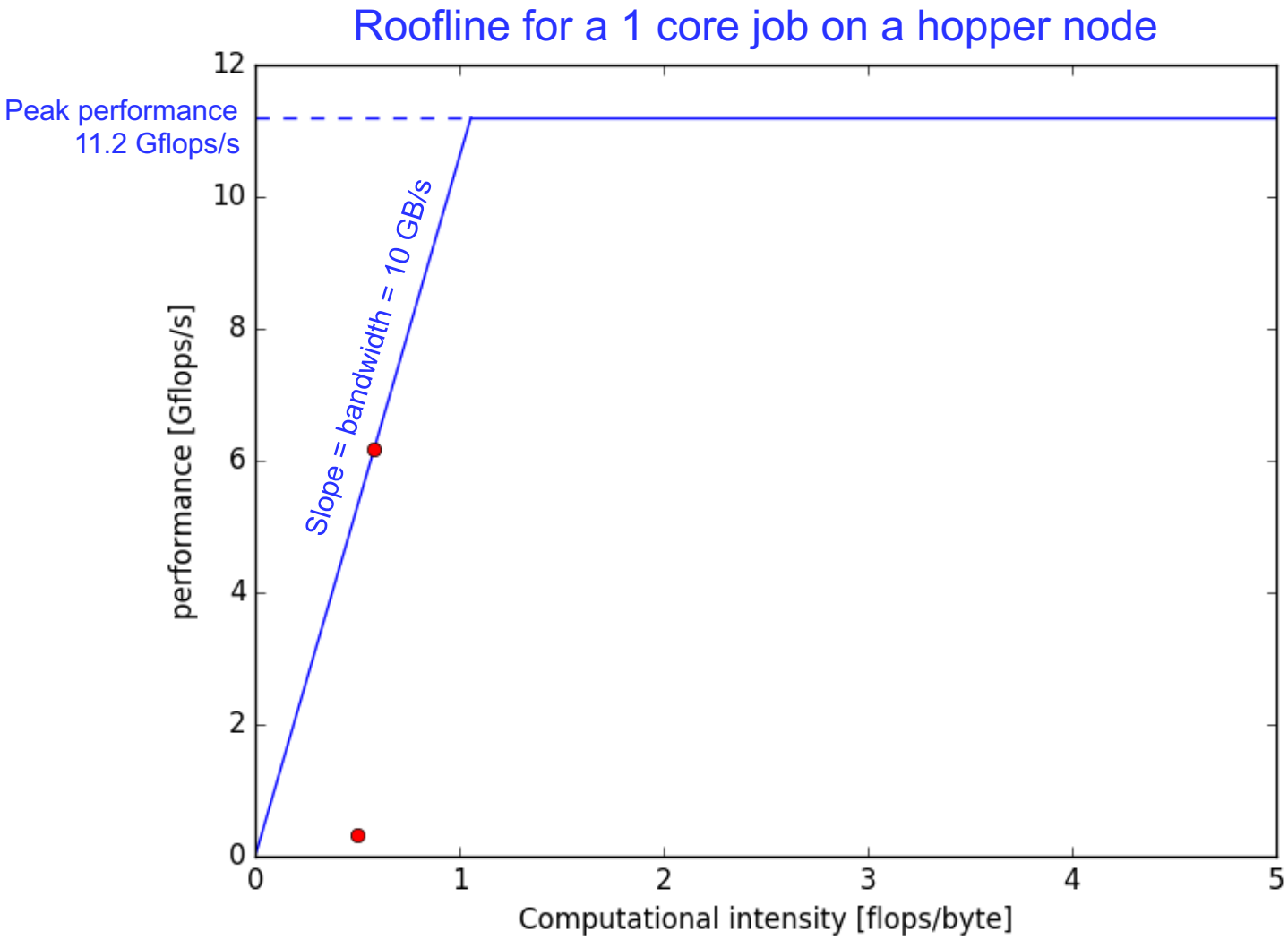


Q2: is this the best we can get?
Yes and No

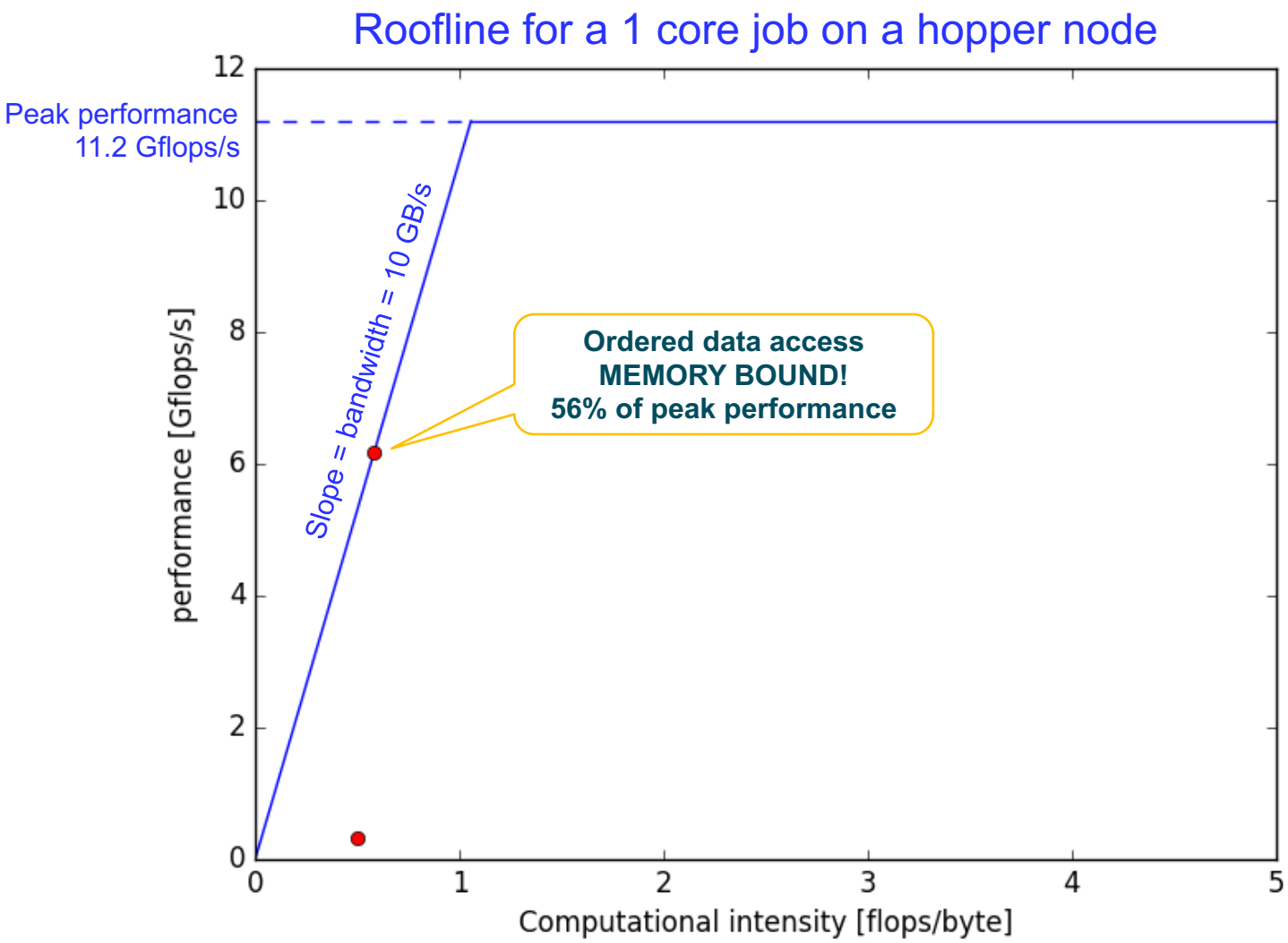
No: We are processing the maximum amount of data in the give time (but the processor is idle for 44% of the time).

Yes: We can improve if we can do more useful work on the data while they are in cache.

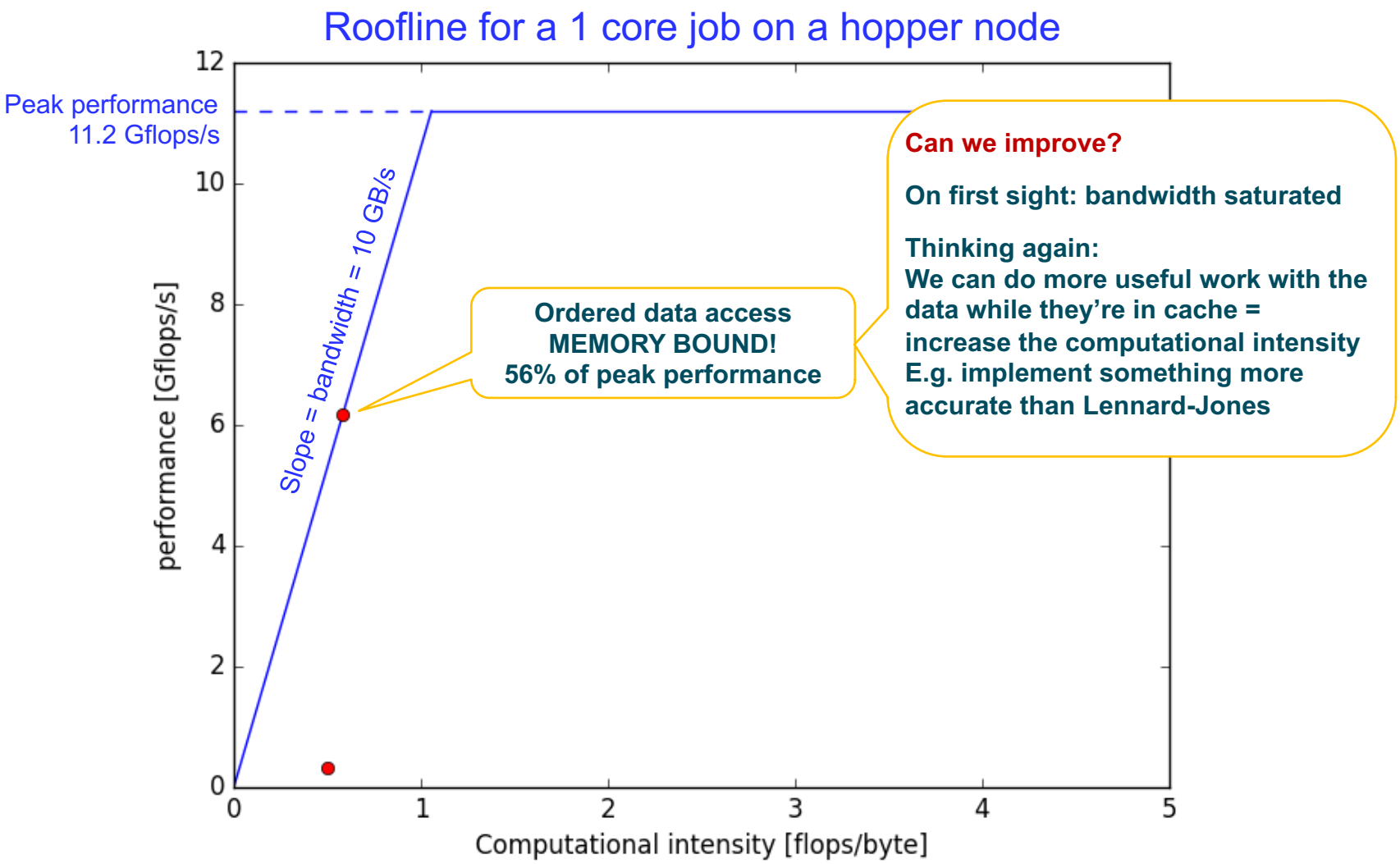
Roofline model



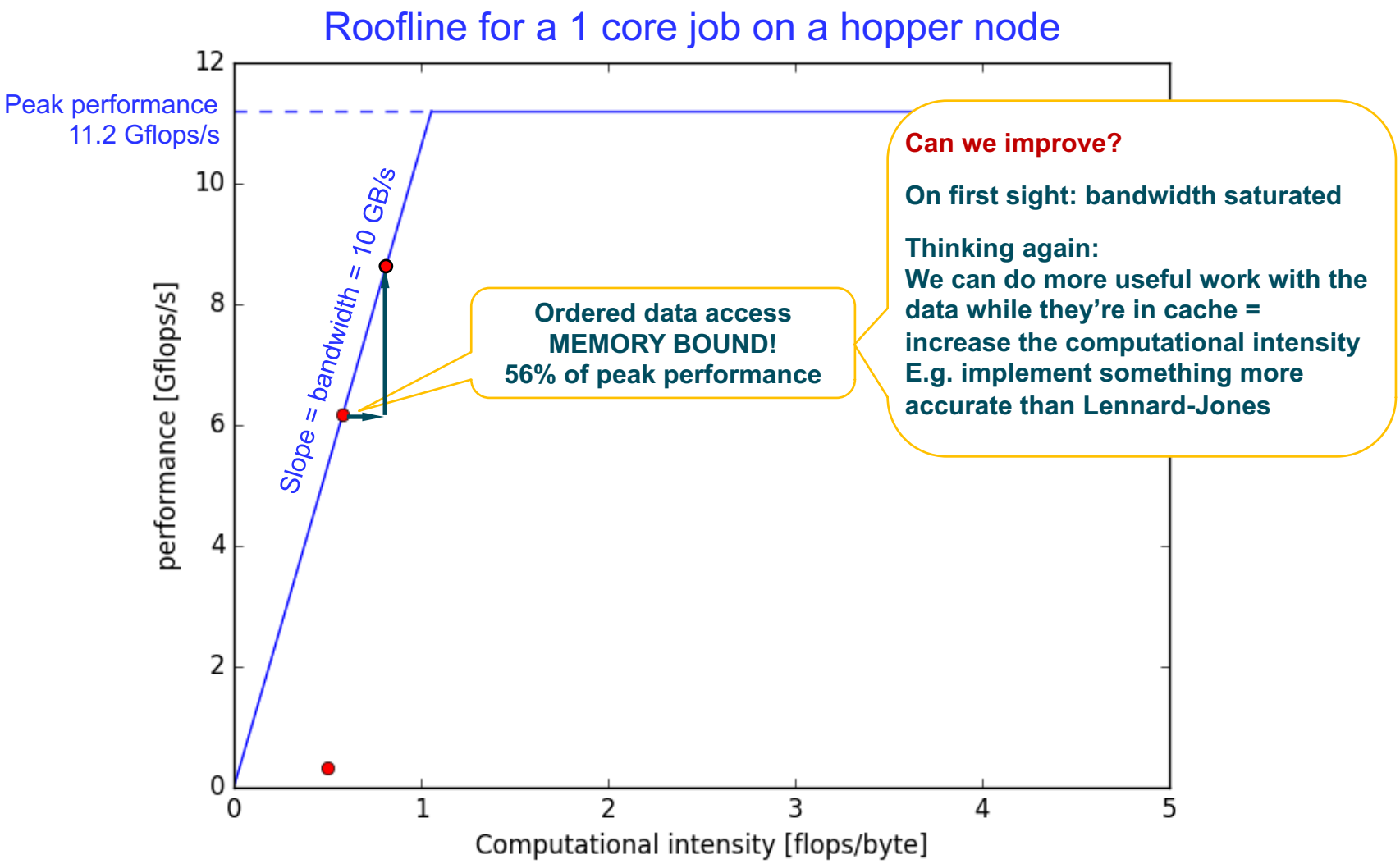
Roofline model



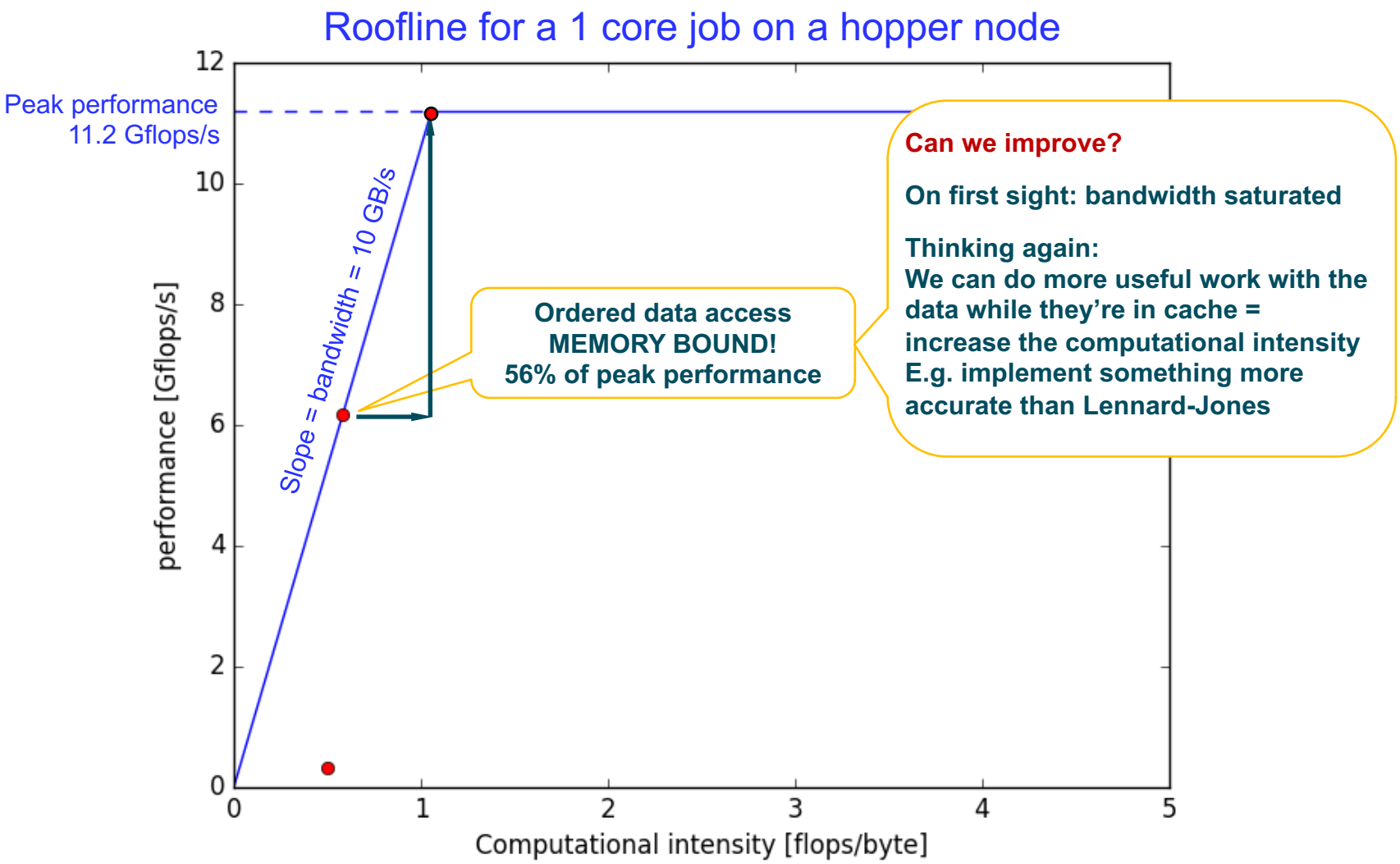
Roofline model



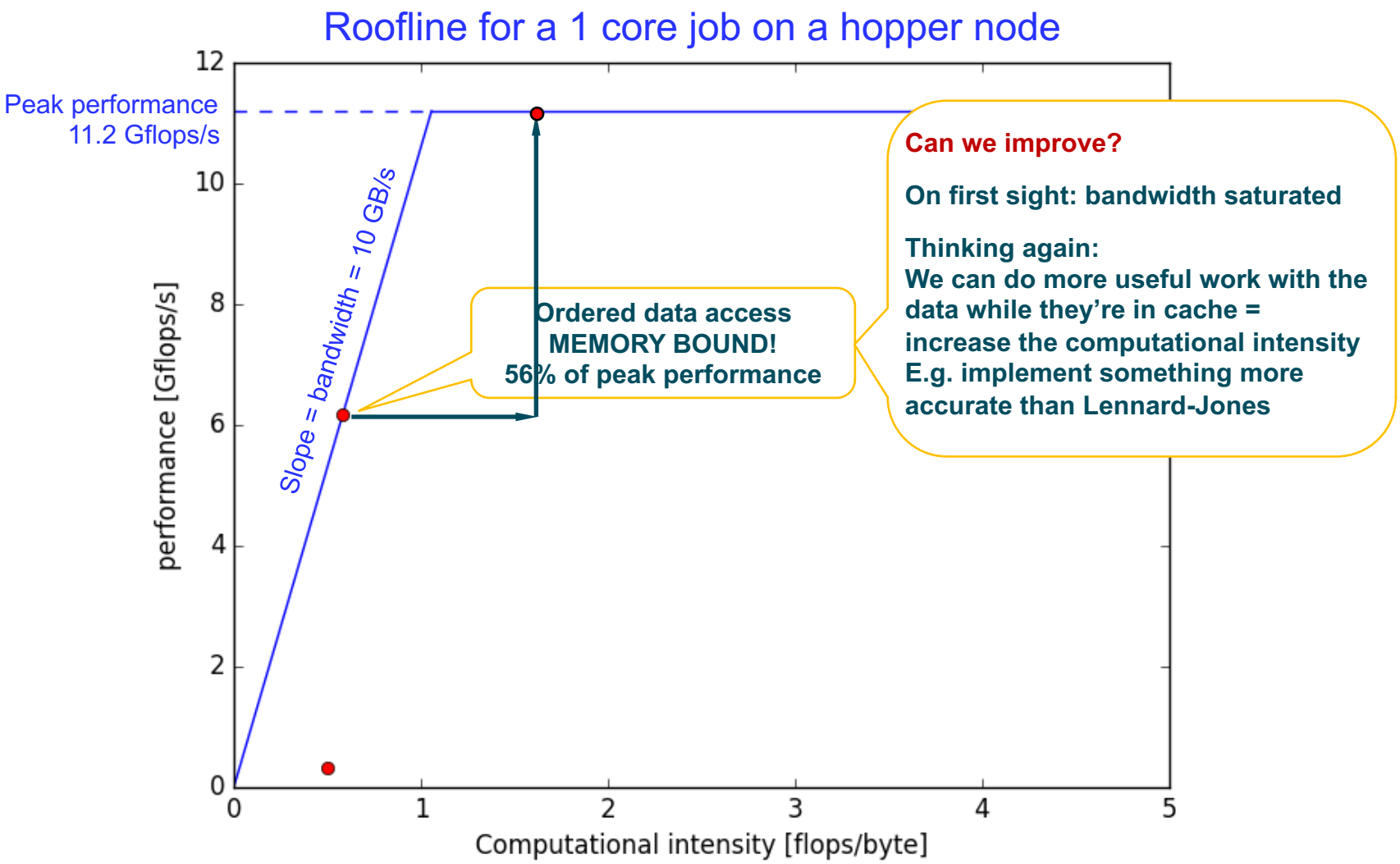
Roofline model



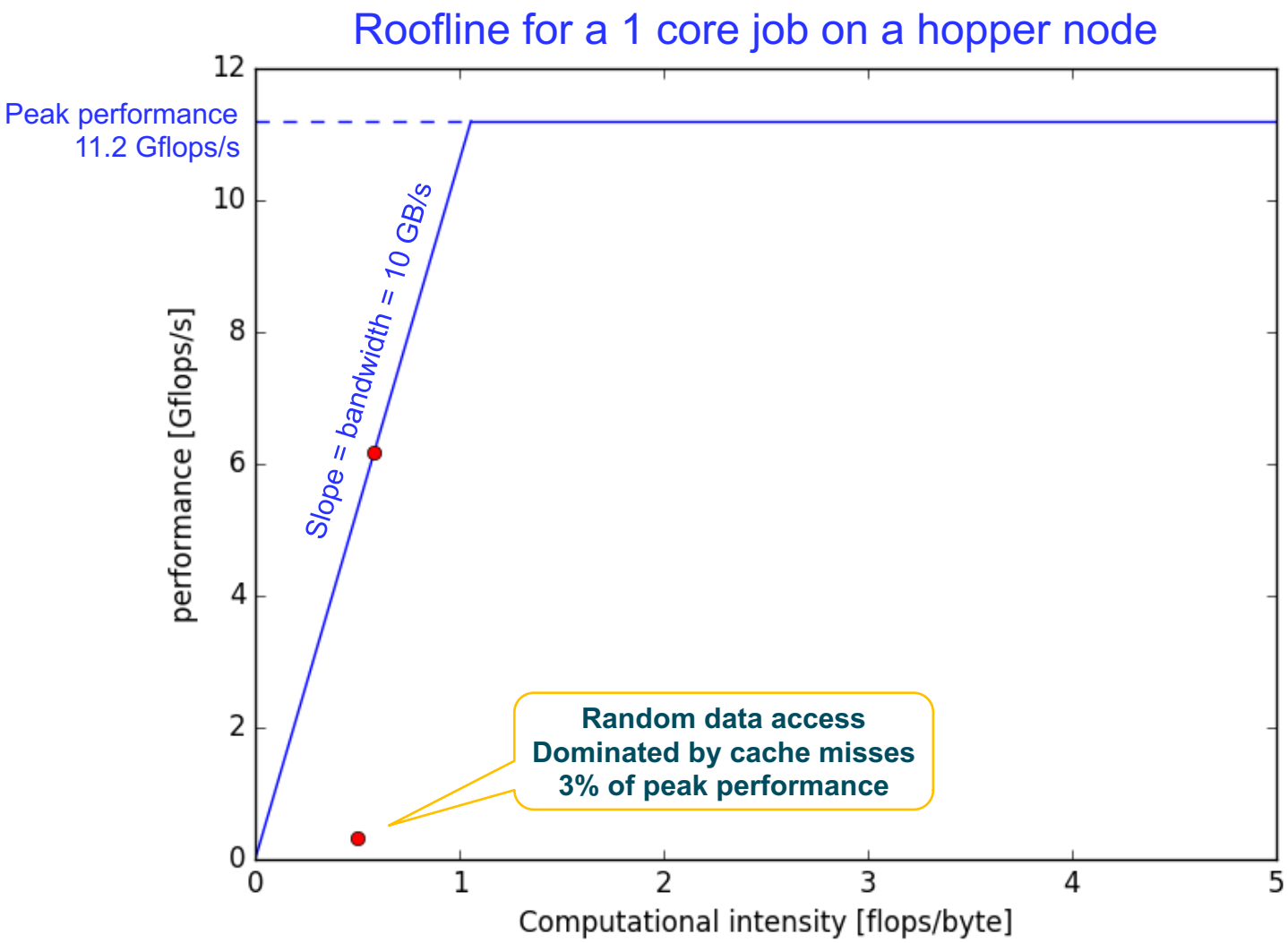
Roofline model



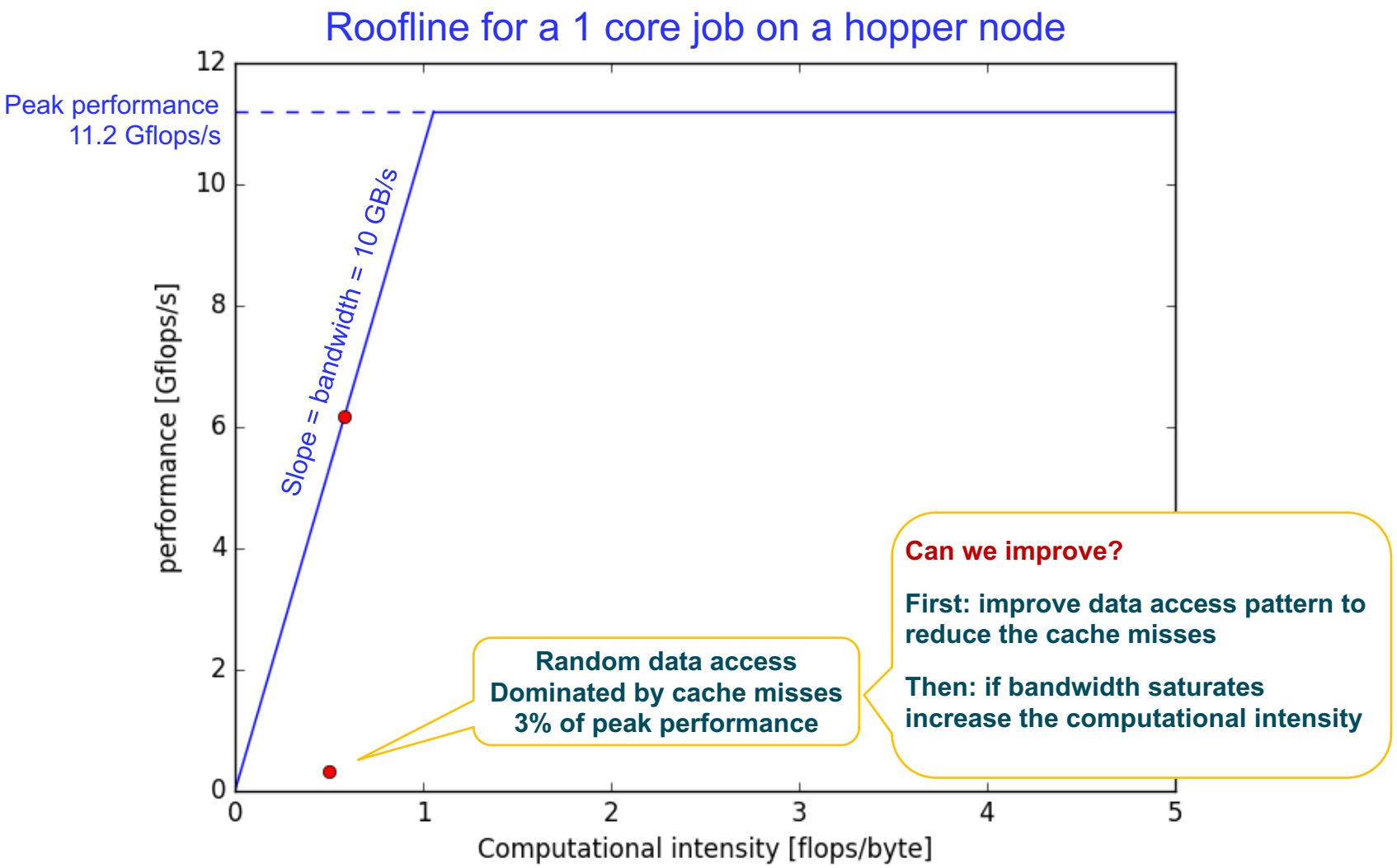
Roofline model



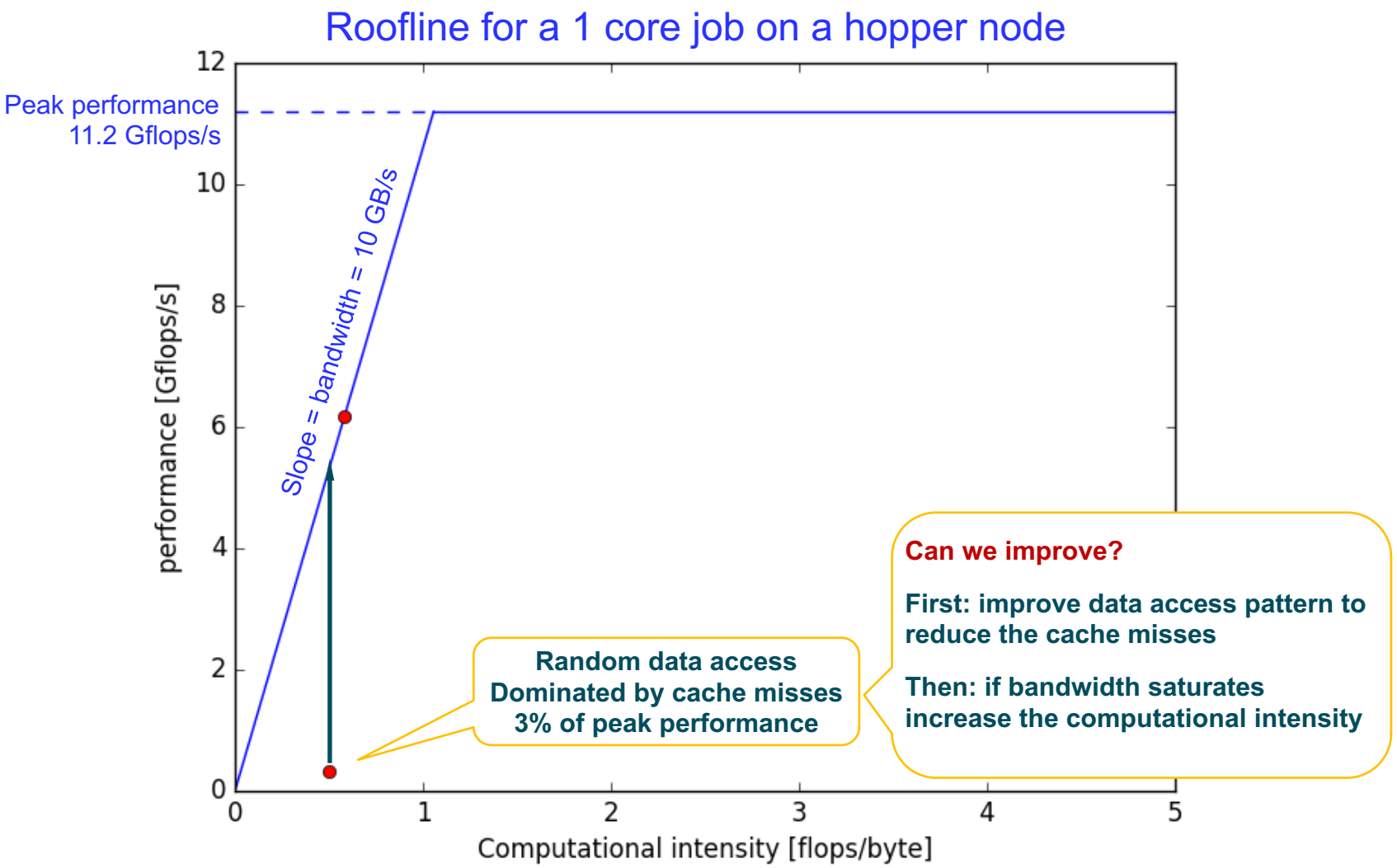
Roofline model



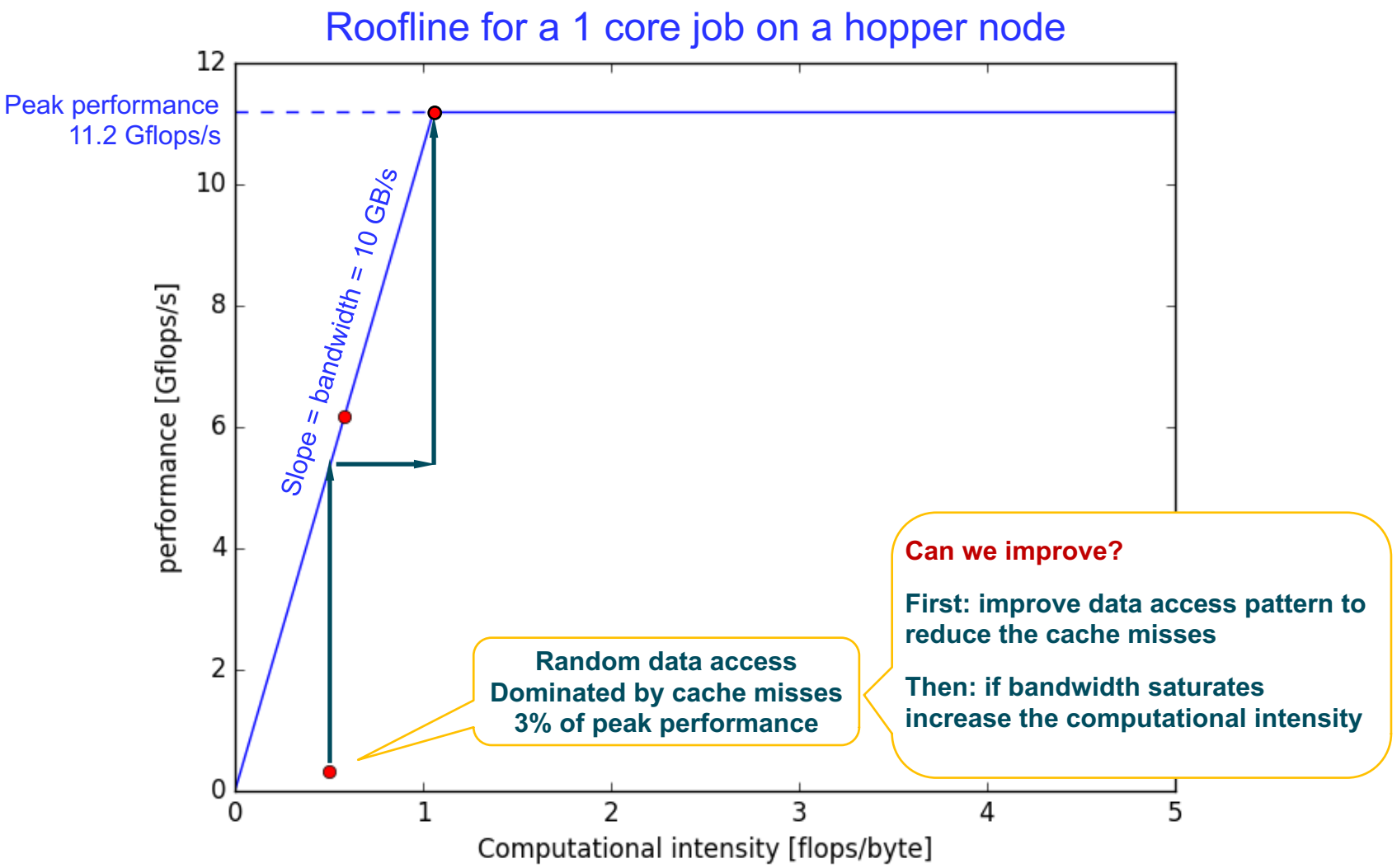
Roofline model



Roofline model



Roofline model



- For simple cases
  - a back of the envelope calculation like this
  - and an understanding of how memory works

  can guide you to more efficient code

- For real cases we need something more sophisticated

Intel Parallel Cluster xe

- Intel Advisor xe
  - Vectorization and threading
- Intel VTune Analyzer xe
  - Data access and cpu utilization
- Intel Inspector
  - Thread performance analysis (OpenMP, Intel TBB)
- Intel Cluster Inspector
  - MPI process performance analysis

Intel Parallel Cluster xe

- Intel Advisor xe
  - Vectorization and threading
- Intel VTune Analyzer xe
  - Data access and cpu utilization
- Intel Inspector
  - Thread performance analysis (OpenMP, Intel TBB)
- Intel Cluster Inspector
  - MPI process performance analysis

For a later session

Intel advisor

- https://software.intel.com/en-us/get-started-with-advisor
- https://software.intel.com/en-us/get-started-with-vtune

Intel Advisor

```
$ssh –X vsc20170@login.hpc.uantwerpen.be
Last login: Thu Sep  8 16:38:25 2016 from 143.169.185.55
------------------------------------------------------------
Welcome to Hopper!
...
vsc20170@ln02 ~$

vsc20170@ln02 ~$ qsub –I –X
vsc20170@r5c6cn05 ~$
vsc20170@r5c6cn05 ~$ module load Advisor
vsc20170@r5c6cn05 ~$ module list
Currently Loaded Modulefiles:
  1) Advisor/2016_update4
vsc20170@r5c6cn05 ~$
vsc20170@r5c6cn05 ~$ advixe-gui &
```

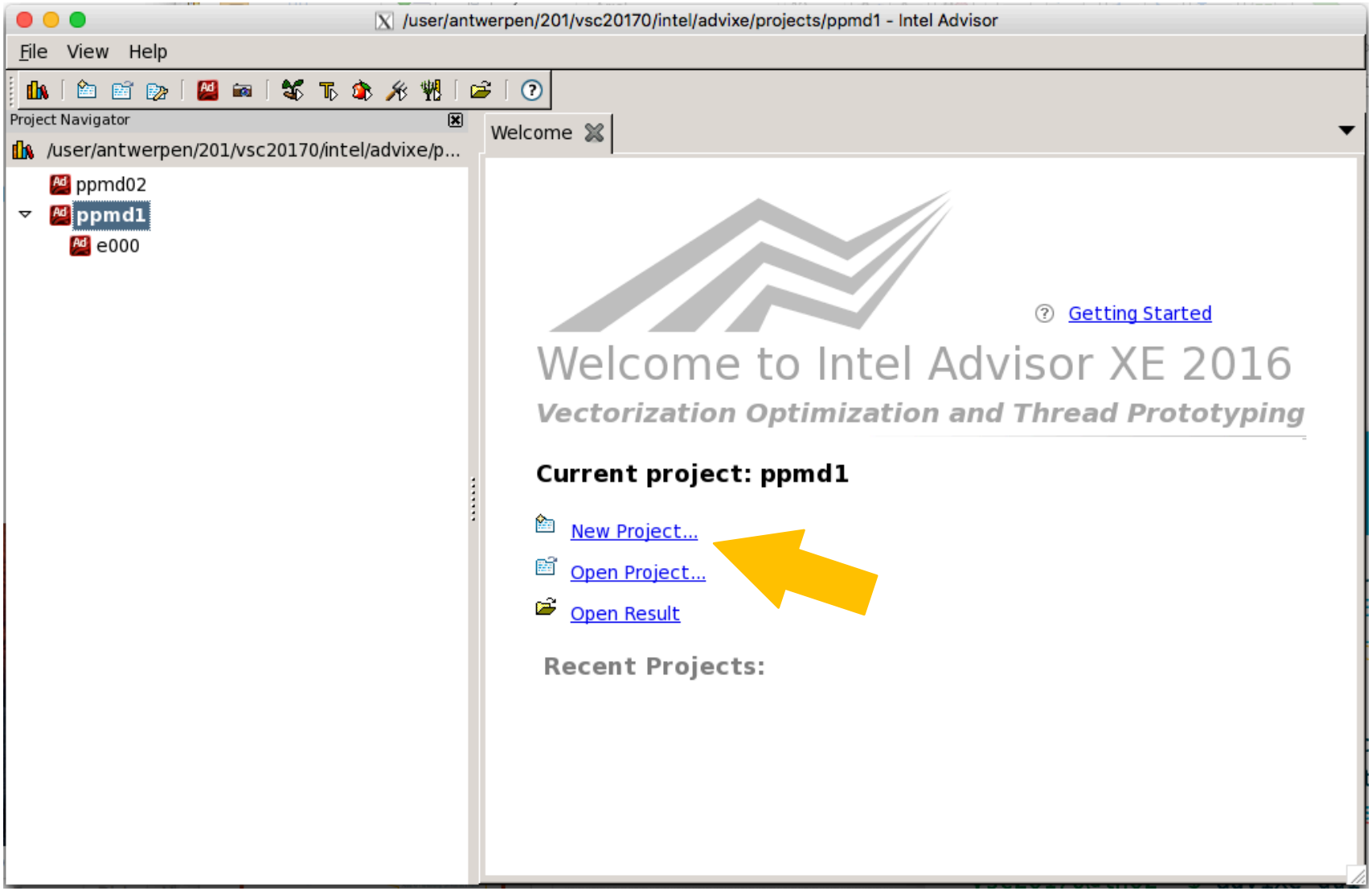
Allow X11 forwarding. On macOS install XQuartz, On Windows install Xming

Start interactive job with X11 forwarding.

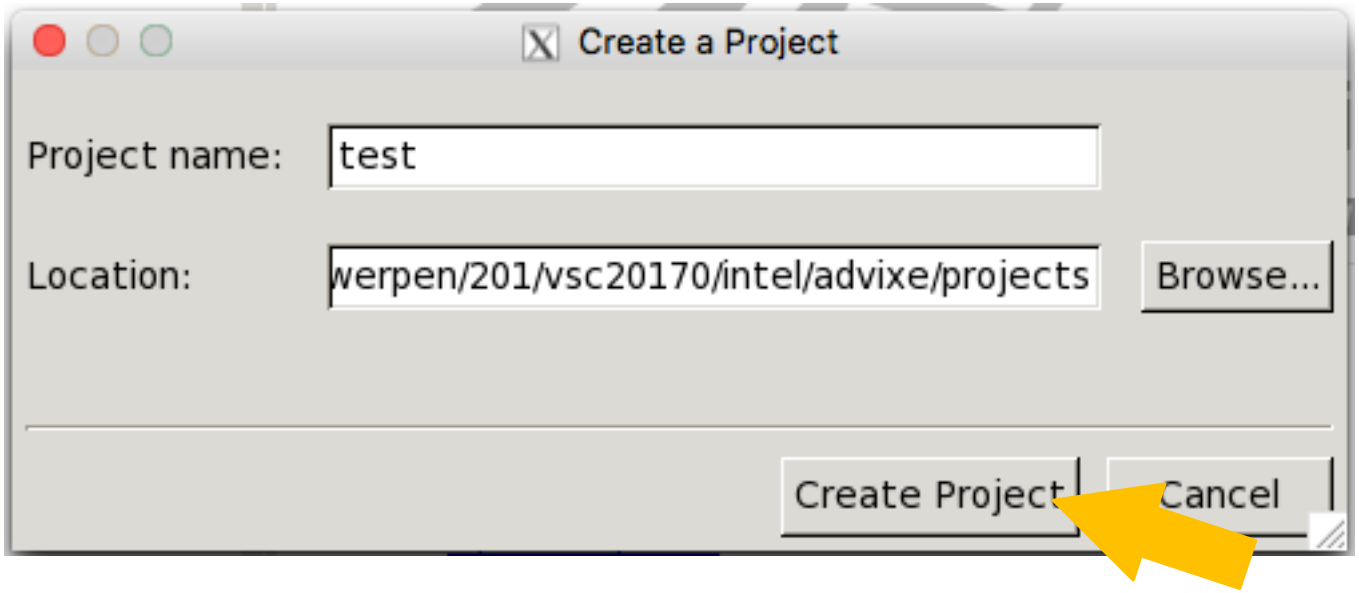
The compute node we are running on

& = Run in background, so the terminal remains functional

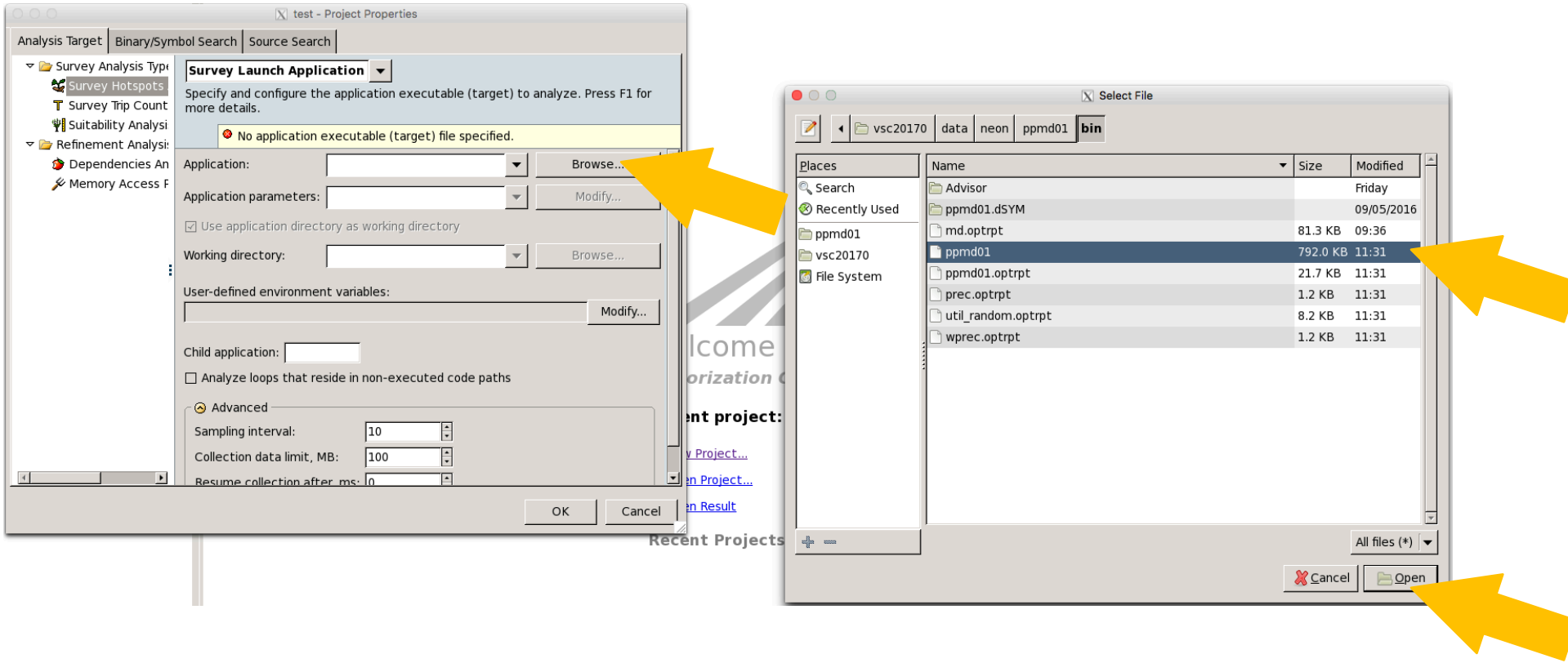
Intel Advisor



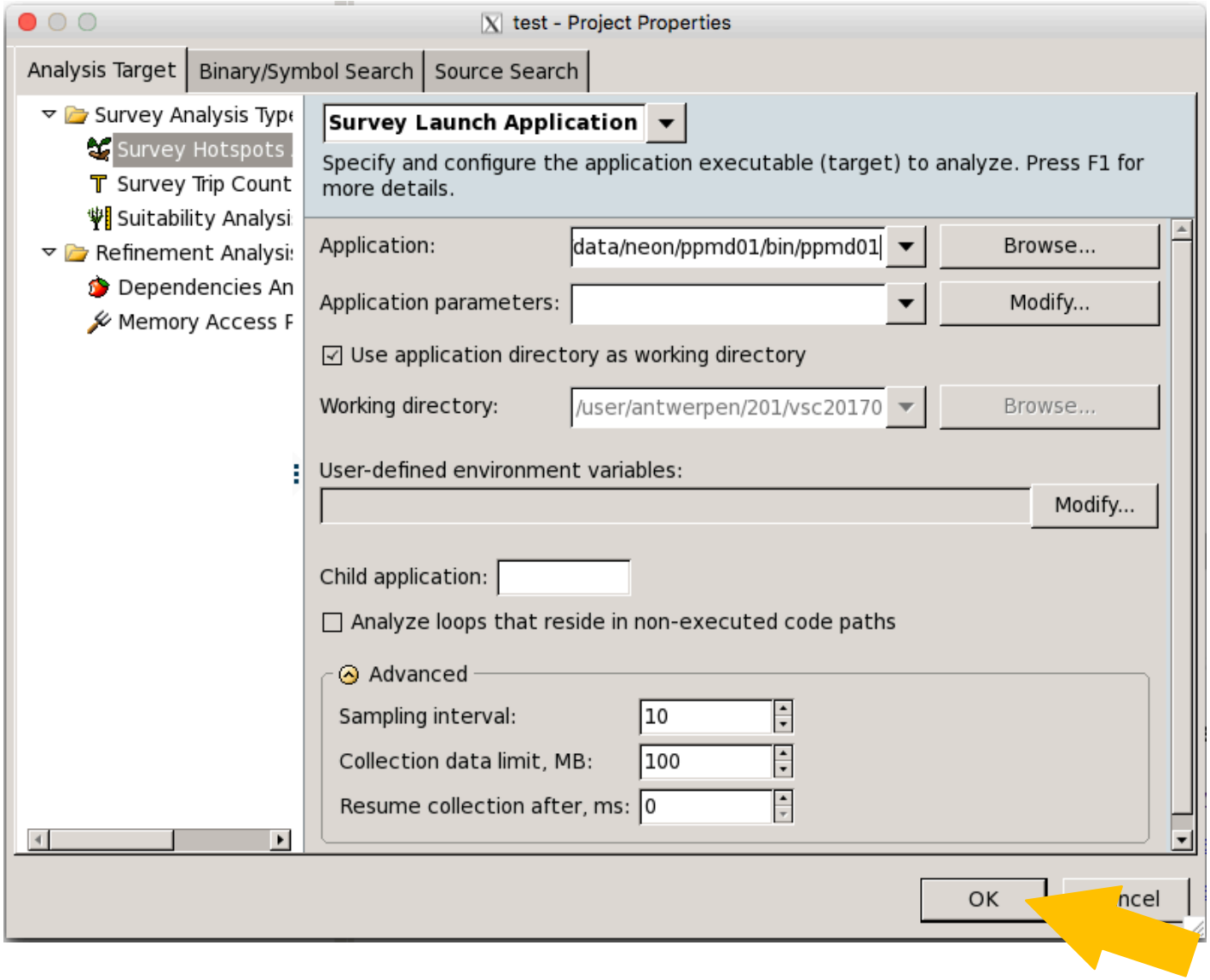
Intel Advisor



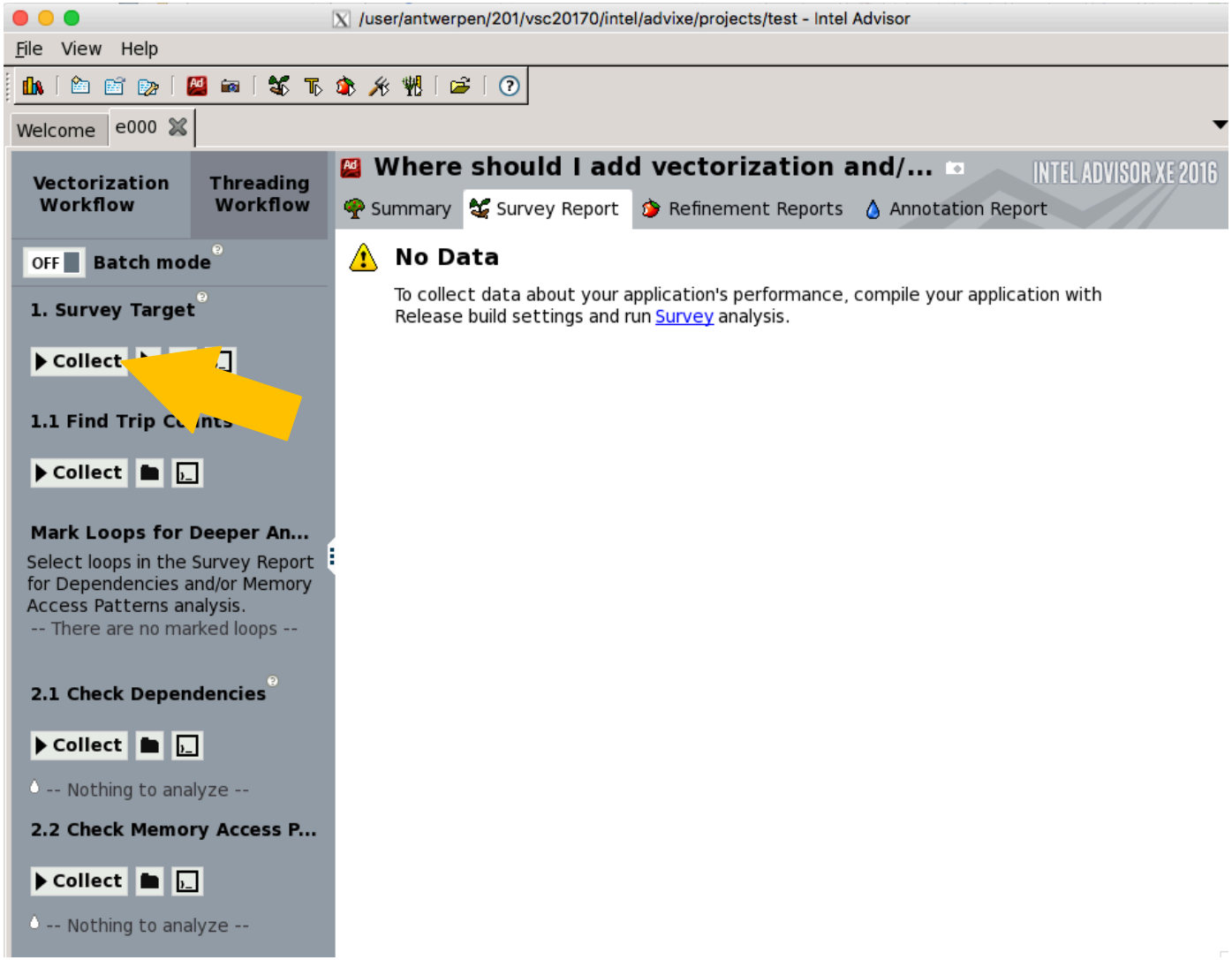
Intel Advisor



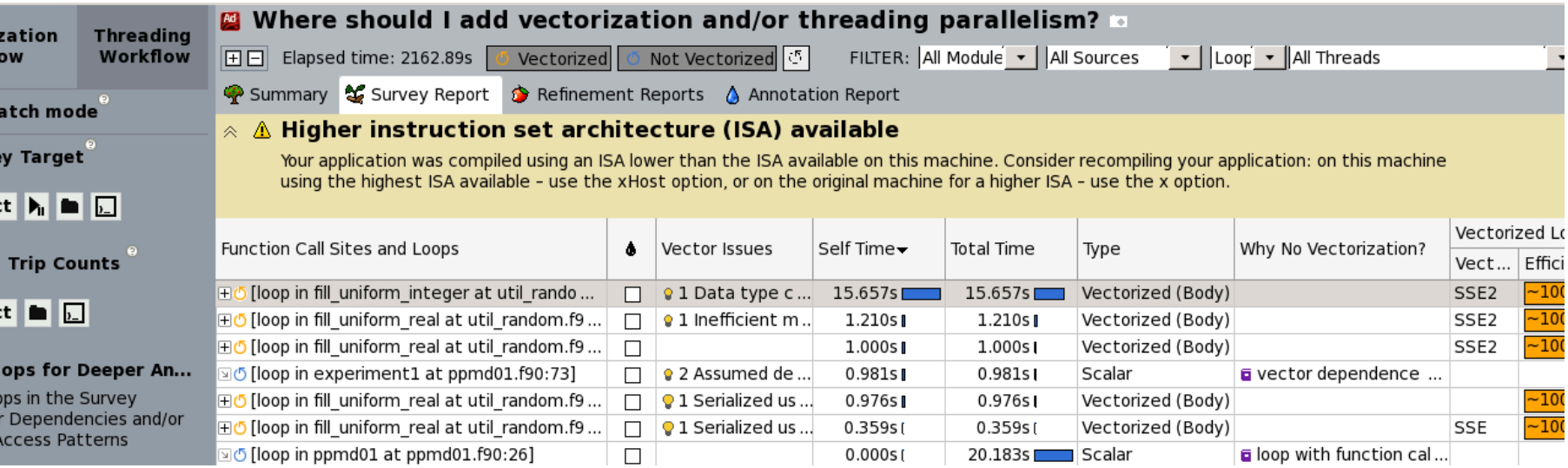
Intel Advisor



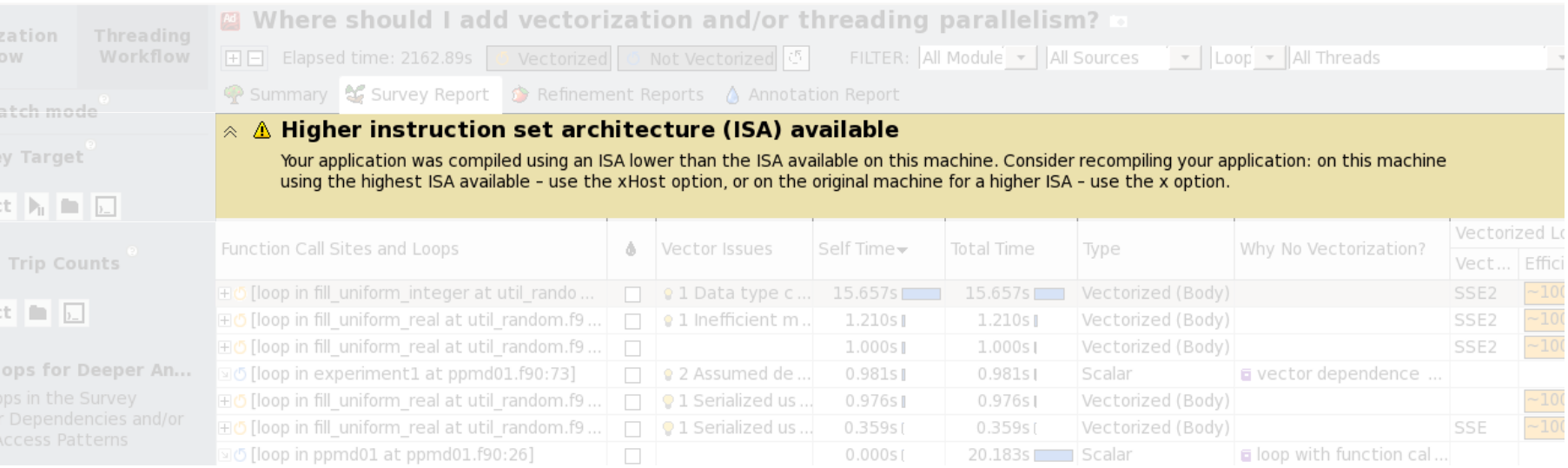
Intel Advisor



Intel Advisor

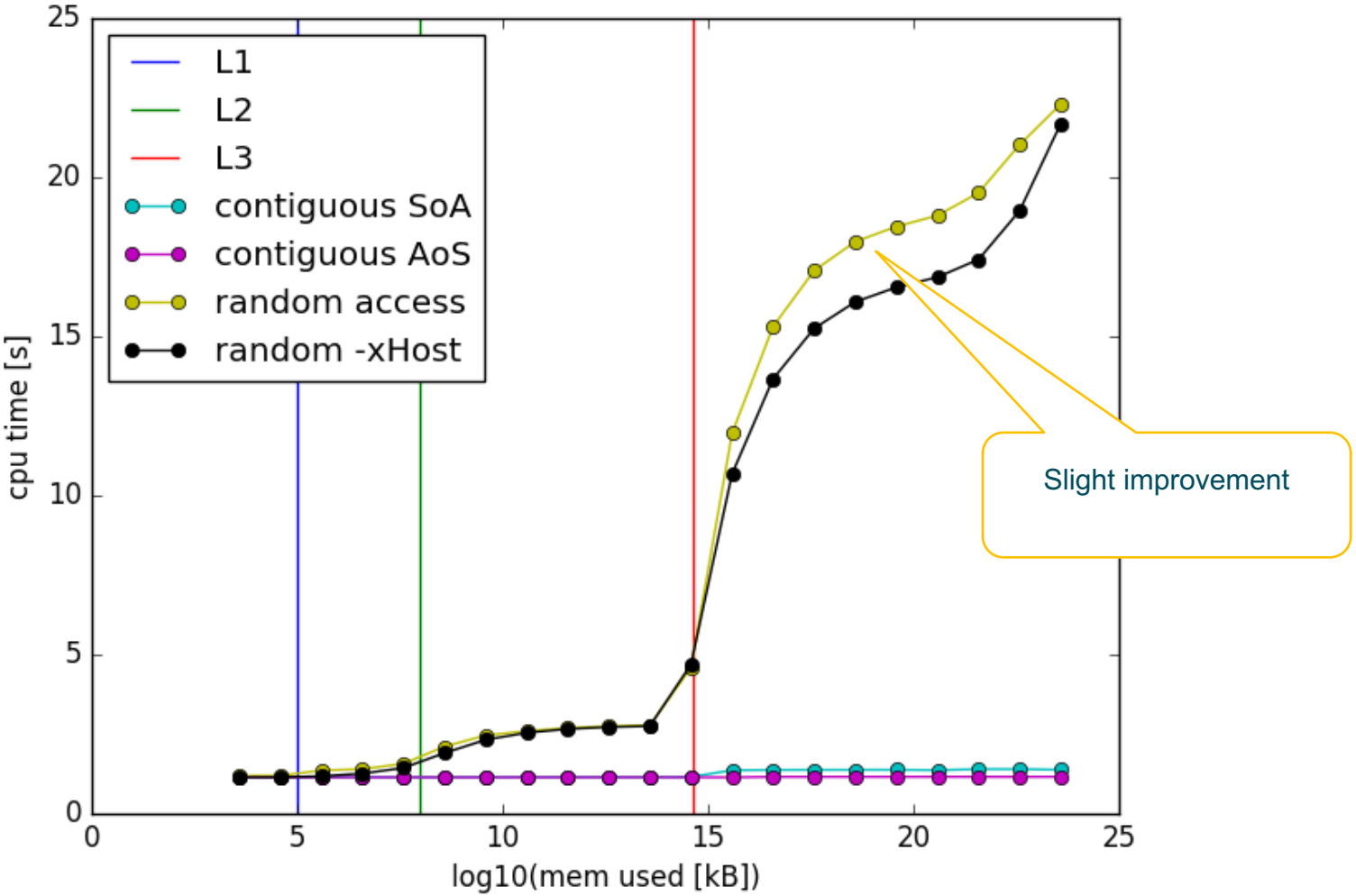


Intel Advisor

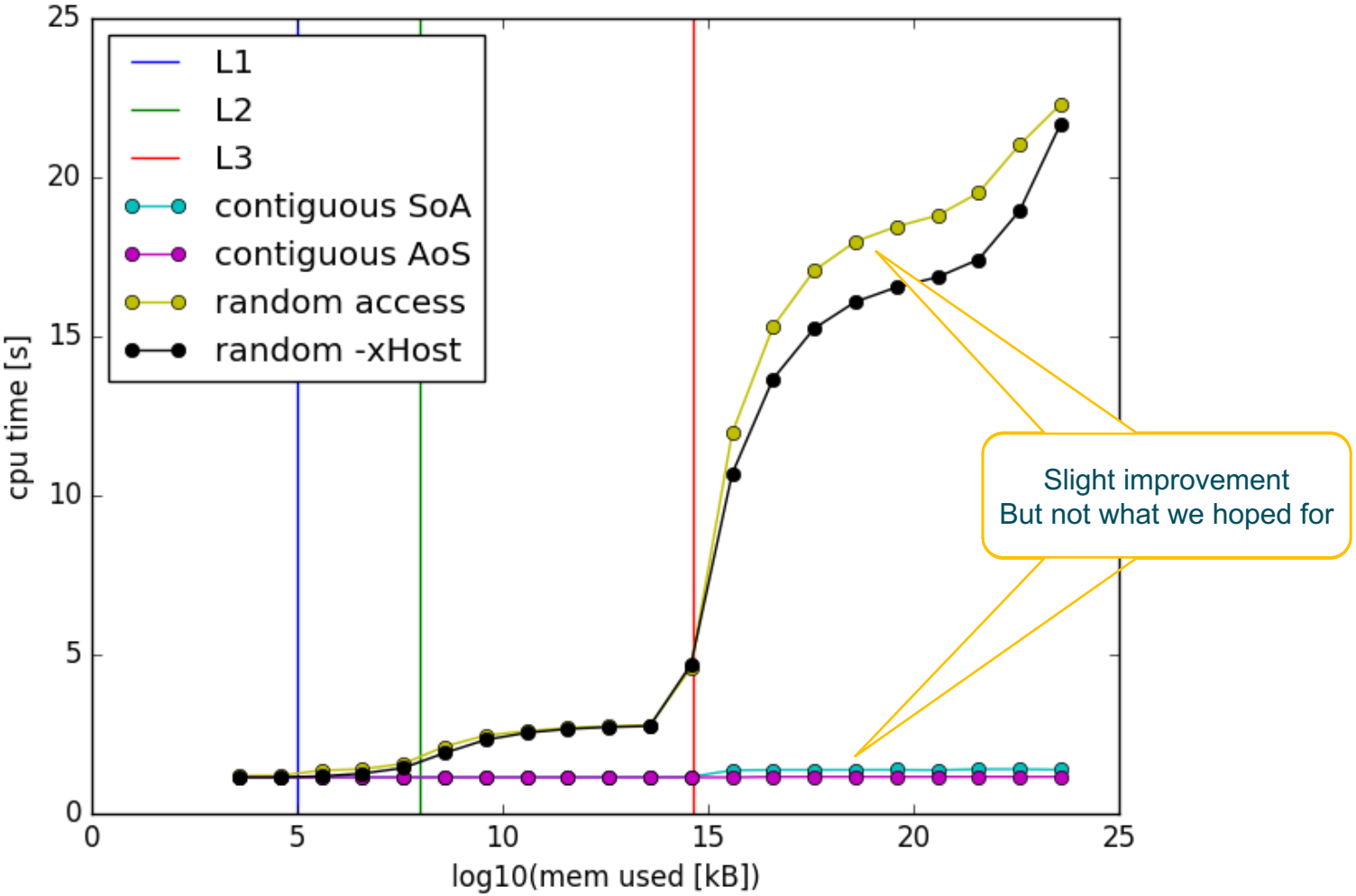


- Add compiler option –xHost

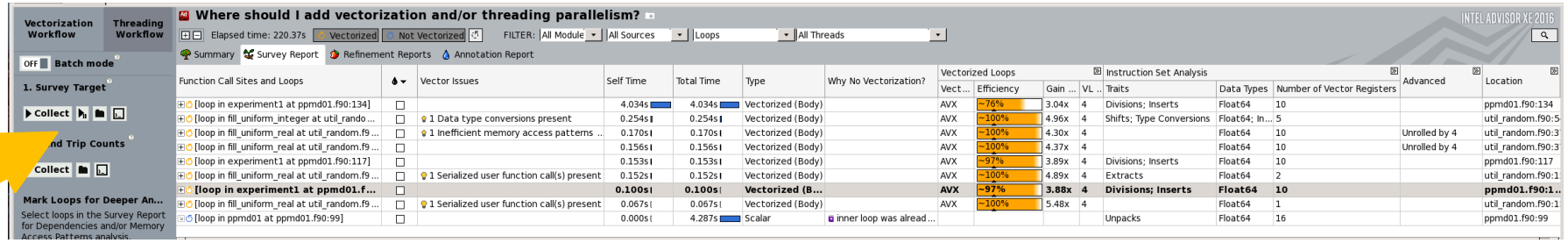
Intel Advisor



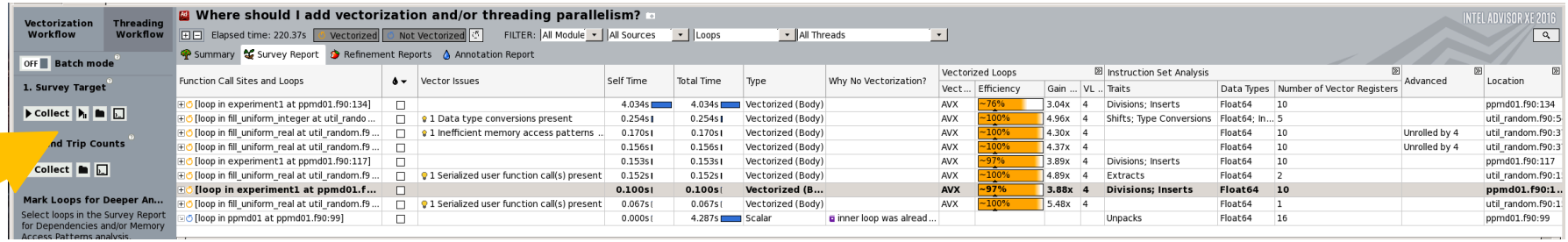
Intel Advisor



Intel Advisor



- Run Intel Advisor again

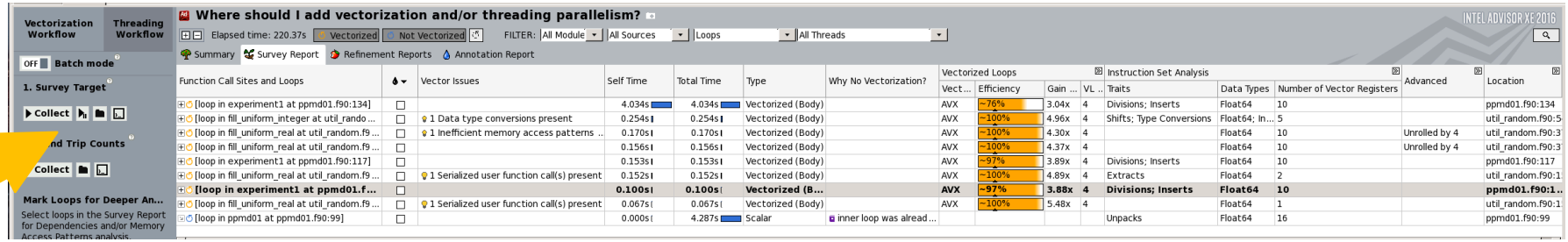


- List of hot spots

Function Call Sites and Loops		Vector Issues	Self Time	Total Time	Type	Why No Vectorization?
[loop in experiment1 at ppmd01.f...			**4.034s**	**4.034s**	**Vectorized (B...**	
[loop in fill_uniform_integer at util_rando ...		1 Data type conversions present	0.254s	0.254s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...		1 Inefficient memory access patterns ..	0.170s	0.170s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...			0.156s	0.156s	Vectorized (Body)	
[loop in experiment1 at ppmd01.f90:117]			0.153s	0.153s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...		1 Serialized user function call(s) present	0.152s	0.152s	Vectorized (Body)	
[loop in experiment1 at ppmd01.f90:100]			0.100s	0.100s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...		1 Serialized user function call(s) present	0.067s	0.067s	Vectorized (Body)	
[loop in ppmd01 at ppmd01.f90:99]			0.000s	4.287s	Scalar	inner loop was alread...

Intel Advisor

- Run Intel Advisor again

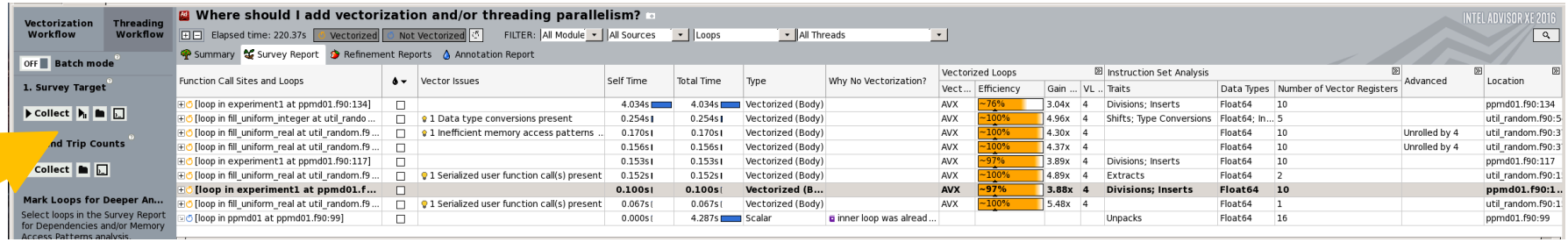


- List of hot spots

Function Call Sites and Loops		Vector Issues	Self Time	Total Time	Type	Why No Vectorization?
[loop in experiment1 at ppmd01.f...			4.034s	4.034s	Vectorized (B...	
[loop in fill_uniform_integer at util_rando ...		1 Data type conversions present	0.254s	0.254s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...		1 Inefficient memory access patterns ..	0.170s	0.170s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...			0.156s	0.156s	Vectorized (Body)	
[loop in experiment1 at ppmd01.f90:117]			0.153s	0.153s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...		1 Serialized user function call(s) present	0.152s	0.152s	Vectorized (Body)	
[loop in experiment1 at ppmd01.f90:100]			0.100s	0.100s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...		1 Serialized user function call(s) present	0.067s	0.067s	Vectorized (Body)	
[loop in ppmd01 at ppmd01.f90:99]			0.000s	4.287s	Scalar	inner loop was alread ...

Intel Advisor

- Run Intel Advisor again



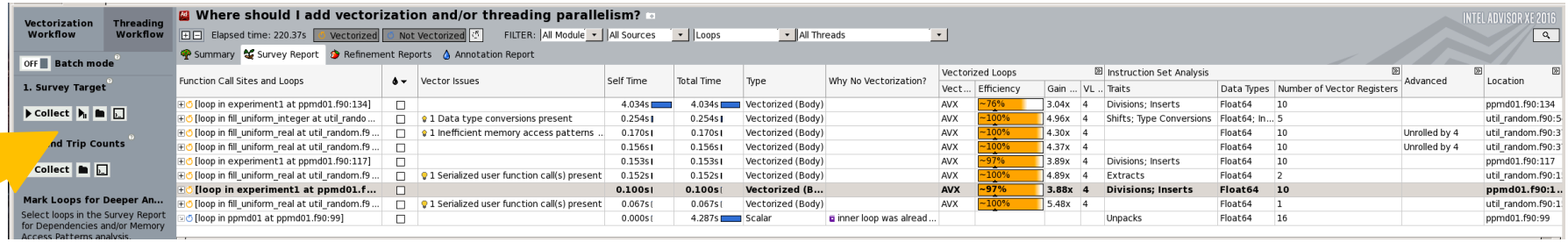
- List of hot spots

Random access

Function Call Sites and Loops		Vector Issues	Self Time	Total Time	Type	Why No Vectorization?
[loop in experiment1 at ppmd01.f...			4.034s	4.034s	Vectorized (B...	
[loop in fill_uniform_integer at util_rando ...		1 Data type conversions present	0.254s	0.254s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...		1 Inefficient memory access patterns ..	0.170s	0.170s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...			0.156s	0.156s	Vectorized (Body)	
[loop in experiment1 at ppmd01.f90:117]			0.153s	0.153s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...		1 Serialized user function call(s) present	0.152s	0.152s	Vectorized (Body)	
[loop in experiment1 at ppmd01.f90:100]			0.100s	0.100s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...		1 Serialized user function call(s) present	0.067s	0.067s	Vectorized (Body)	
[loop in ppmd01 at ppmd01.f90:99]			0.000s	4.287s	Scalar	inner loop was alread...

Intel Advisor

- Run Intel Advisor again



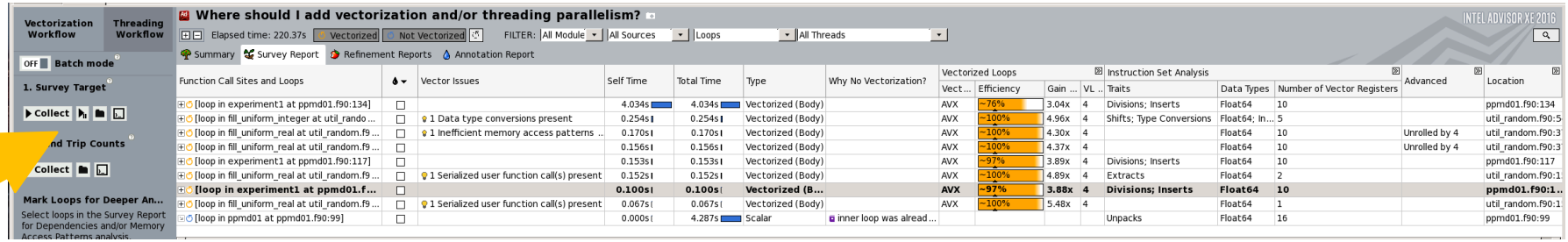
- List of hot spots

Random access

Function Call Sites and Loops		Vector Issues	Self Time	Total Time	Type	Why No Vectorization?
[loop in experiment1 at ppmd01.f...			4.034s	4.034s	Vectorized (B...	
[loop in fill_uniform_integer at util_rando ...		1 Data type conversions present	0.254s	0.254s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...		1 Inefficient memory access patterns ..	0.170s	0.170s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...			0.156s	0.156s	Vectorized (Body)	
[loop in experiment1 at ppmd01.f90:117]			0.153s	0.153s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...		1 Serialized user function call(s) present	0.152s	0.152s	Vectorized (Body)	
[loop in experiment1 at ppmd01.f90:100]			0.100s	0.100s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...		1 Serialized user function call(s) present	0.067s	0.067s	Vectorized (Body)	
[loop in ppmd01 at ppmd01.f90:99]			0.000s	4.287s	Scalar	inner loop was alread ...

Intel Advisor

- Run Intel Advisor again



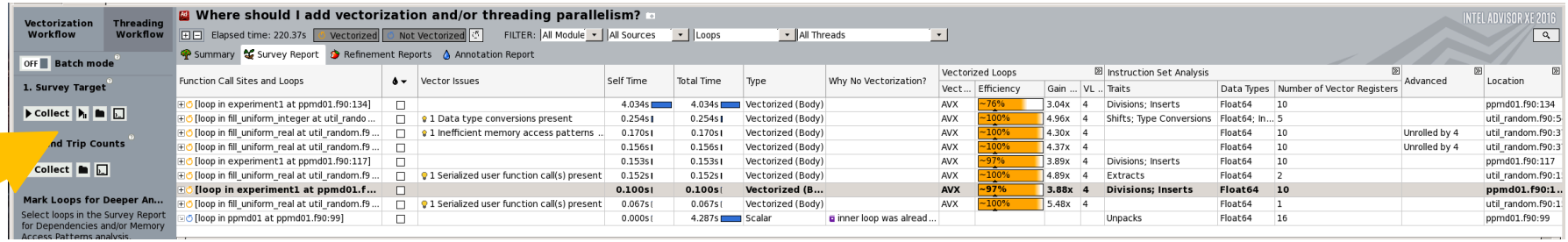
- List of hot spots

Random access

Function Call Sites and Loops		Vector Issues	Self Time	Total Time	Type	Why No Vectorization?
[loop in experiment1 at ppmd01.f...			4.034s	4.034s	Vectorized (B...	
[loop in fill_uniform_integer at util_rando ...		1 Data type conversions present	0.254s	0.254s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...		1 Inefficient memory access patterns ..	0.170s	0.170s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...			0.156s	0.156s	Vectorized (Body)	
[loop in experiment1 at ppmd01.f90:117]			0.153s	0.153s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...		1 Serialized user function call(s) present	0.152s	0.152s	Vectorized (Body)	
[loop in experiment1 at ppmd01.f90:100]			0.100s	0.100s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...		1 Serialized user function call(s) present	0.067s	0.067s	Vectorized (Body)	
[loop in ppmd01 at ppmd01.f90:99]			0.000s	4.287s	Scalar	inner loop was alread ...

Intel Advisor

- Run Intel Advisor again



- List of hot spots

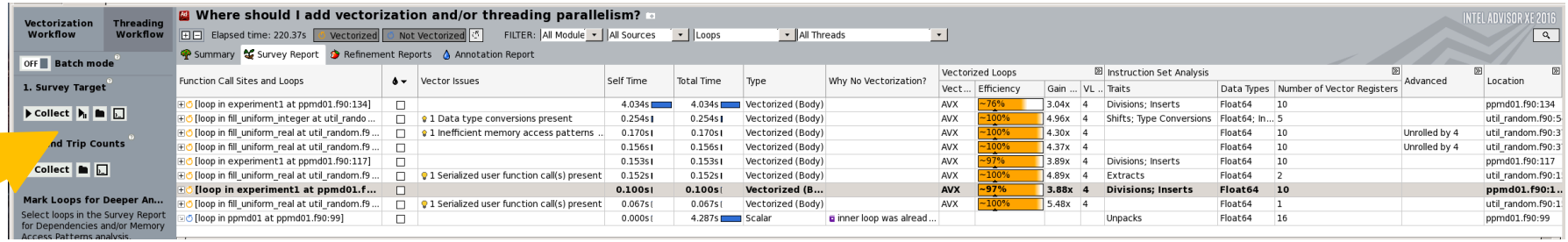
Random access

Contiguous-AoS

Function Call Sites and Loops		Vector Issues	Self Time	Total Time	Type	Why No Vectorization?
[loop in experiment1 at ppmd01.f...			4.034s	4.034s	Vectorized (B...	
[loop in fill_uniform_integer at util_rando ...		1 Data type conversions present	0.254s	0.254s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...		1 Inefficient memory access patterns ..	0.170s	0.170s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...			0.156s	0.156s	Vectorized (Body)	
[loop in experiment1 at ppmd01.f90:117]			0.153s	0.153s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...		1 Serialized user function call(s) present	0.152s	0.152s	Vectorized (Body)	
[loop in experiment1 at ppmd01.f90:100]			0.100s	0.100s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...		1 Serialized user function call(s) present	0.067s	0.067s	Vectorized (Body)	
[loop in ppmd01 at ppmd01.f90:99]			0.000s	4.287s	Scalar	inner loop was alread ...

Intel Advisor

- Run Intel Advisor again



- List of hot spots

Random access

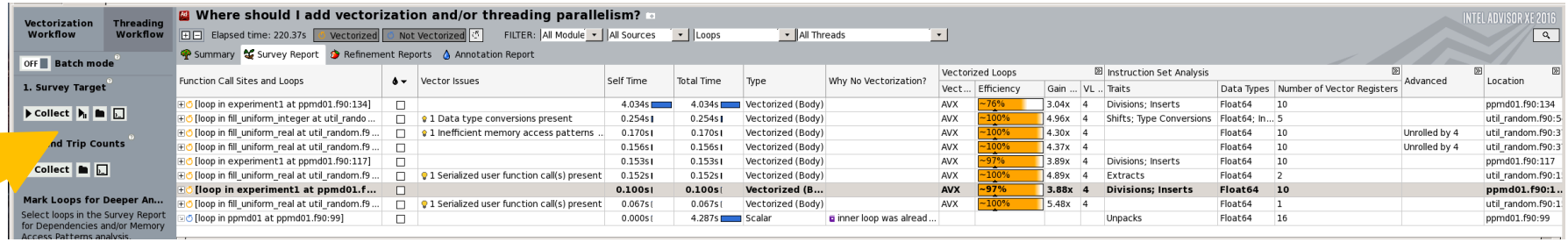
Contiguous-AoS

Contiguous-SoA

Function Call Sites and Loops		Vector Issues	Self Time	Total Time	Type	Why No Vectorization?
[loop in experiment1 at ppmd01.f...			4.034s	4.034s	Vectorized (B...	
[loop in fill_uniform_integer at util_rando ...		1 Data type conversions present	0.254s	0.254s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...		1 Inefficient memory access patterns ..	0.170s	0.170s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...			0.156s	0.156s	Vectorized (Body)	
[loop in experiment1 at ppmd01.f90:117]			0.153s	0.153s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...		1 Serialized user function call(s) present	0.152s	0.152s	Vectorized (Body)	
[loop in experiment1 at ppmd01.f90:100]			0.100s	0.100s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...		1 Serialized user function call(s) present	0.067s	0.067s	Vectorized (Body)	
[loop in ppmd01 at ppmd01.f90:99]			0.000s	4.287s	Scalar	inner loop was alread ...

Intel Advisor

- Run Intel Advisor again



- List of hot spots

Random access

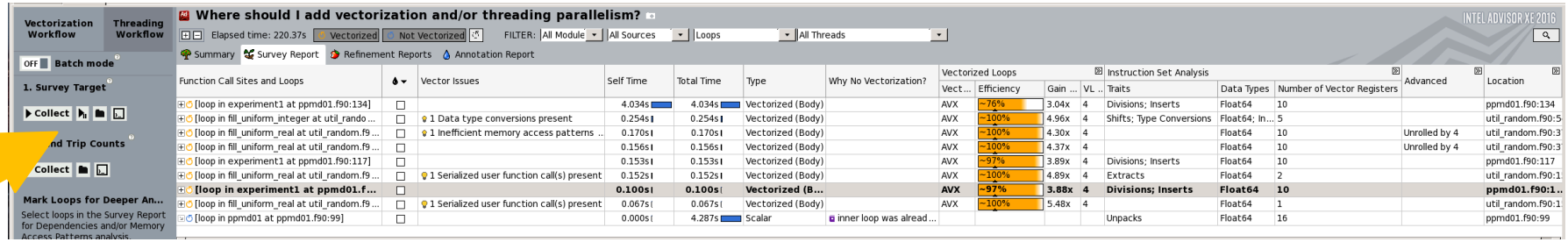
Contiguous-AoS

Contiguous-SoA

Function Call Sites and Loops		Vector Issues	Self Time	Total Time	Type	Why No Vectorization?
[loop in experiment1 at ppmd01.f...			4.034s	4.034s	Vectorized (B...	
[loop in fill_uniform_integer at util_rando ...		1 Data type conversions present	0.254s	0.254s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...		1 Inefficient memory access patterns ..	0.170s	0.170s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...			0.156s	0.156s	Vectorized (Body)	
[loop in experiment1 at ppmd01.f90:117]			0.153s	0.153s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...		1 Serialized user function call(s) present	0.152s	0.152s	Vectorized (Body)	
[loop in experiment1 at ppmd01.f90:100]			0.100s	0.100s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...		1 Serialized user function call(s) present	0.067s	0.067s	Vectorized (Body)	
[loop in ppmd01 at ppmd01.f90:99]			0.000s	4.287s	Scalar	inner loop was alread ...

Intel Advisor

- Run Intel Advisor again



- List of hot spots

Random access

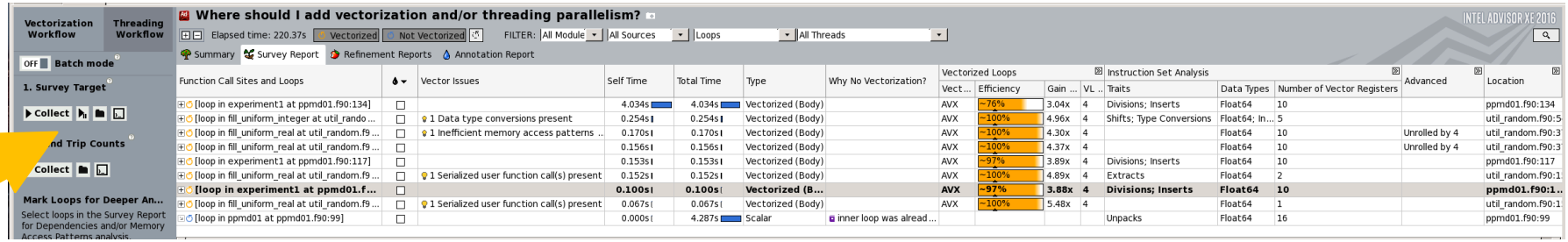
Contiguous-AoS

Contiguous-SoA

Function Call Sites and Loops		Vector Issues	Self Time	Total Time	Type	Why No Vectorization?
[loop in experiment1 at ppmd01.f...			4.034s	4.034s	Vectorized (B...	
[loop in fill_uniform_integer at util_rando ...		1 Data type conversions present	0.254s	0.254s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...		1 Inefficient memory access patterns ..	0.170s	0.170s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...			0.156s	0.156s	Vectorized (Body)	
[loop in experiment1 at ppmd01.f90:117]			0.153s	0.153s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...		1 Serialized user function call(s) present	0.152s	0.152s	Vectorized (Body)	
[loop in experiment1 at ppmd01.f90:100]			0.100s	0.100s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...		1 Serialized user function call(s) present	0.067s	0.067s	Vectorized (Body)	
[loop in ppmd01 at ppmd01.f90:99]			0.000s	4.287s	Scalar	inner loop was alread ...

Intel Advisor

- Run Intel Advisor again



- List of hot spots

Random access

Contiguous-AoS

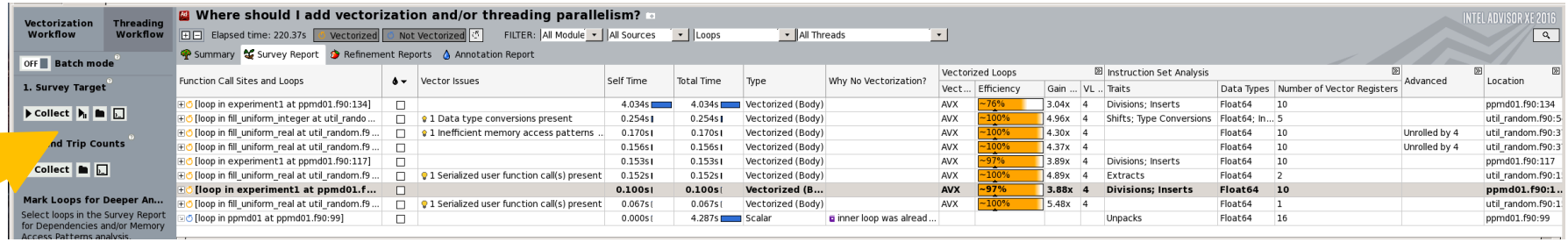
Contiguous-SoA

Function Call Sites and Loops		Vector Issues	Self Time	Total Time	Type	Why No Vectorization?
[loop in experiment1 at ppmd01.f...			4.034s	4.034s	Vectorized (B...	
[loop in fill_uniform_integer at util_rando ...		1 Data type conversions present	0.254s	0.254s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...		1 Inefficient memory access patterns ..	0.170s	0.170s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...			0.156s	0.156s	Vectorized (Body)	
[loop in experiment1 at ppmd01.f90:117]			0.153s	0.153s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...		1 Serialized user function call(s) present	0.152s	0.152s	Vectorized (Body)	
[loop in experiment1 at ppmd01.f90:100]			0.100s	0.100s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...		1 Serialized user function call(s) present	0.067s	0.067s	Vectorized (Body)	
[loop in ppmd01 at ppmd01.f90:99]			0.000s	4.287s	Scalar	inner loop was alread...

Loop over m

Intel Advisor

- Run Intel Advisor again

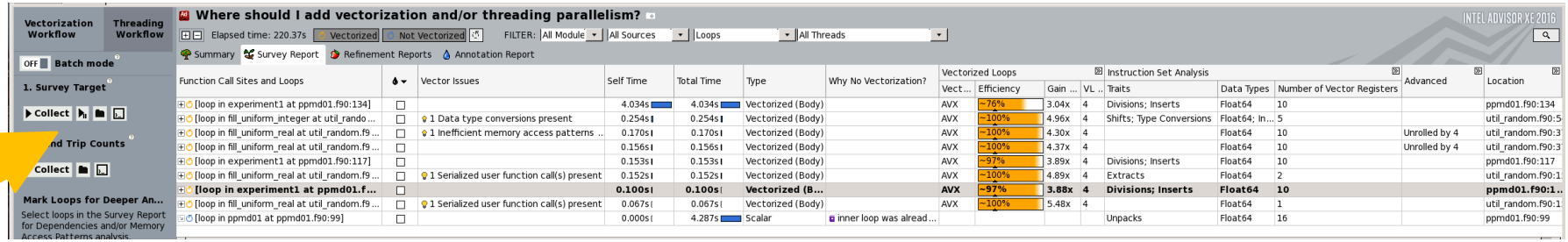


- List of hot spots

Function Call Sites and Loops	Vectorized Loops				Instruction Set Analysis			Advanced	Lo
	Vect...	Efficiency	Gain ...	VL ..	Traits	Data Types	Number of Vector Registers		
[loop in experiment1 at ppmd01.f...	**AVX**	**~76%**	**3.04x**	**4**	**Divisions; Inserts**	**Float64**	**10**		**pp**
[loop in fill_uniform_integer at util_rando ...	AVX	~100%	4.96x	4	Shifts; Type Conversions	Float64; In...	5		ut
[loop in fill_uniform_real at util_random.f9 ...	AVX	~100%	4.30x	4		Float64	10	Unrolled by 4	ut
[loop in fill_uniform_real at util_random.f9 ...	AVX	~100%	4.37x	4		Float64	10	Unrolled by 4	ut
[loop in experiment1 at ppmd01.f90:117]	AVX	~97%	3.89x	4	Divisions; Inserts	Float64	10		pp
[loop in fill_uniform_real at util_random.f9 ...	AVX	~100%	4.89x	4	Extracts	Float64	2		ut
[loop in experiment1 at ppmd01.f90:100]	AVX	~97%	3.88x	4	Divisions; Inserts	Float64	10		pp
[loop in fill_uniform_real at util_random.f9 ...	AVX	~100%	5.48x	4		Float64	1		ut
[loop in ppmd01 at ppmd01.f90:99]					Unpacks	Float64	16		pp

Intel Advisor

- Run Intel Advisor again

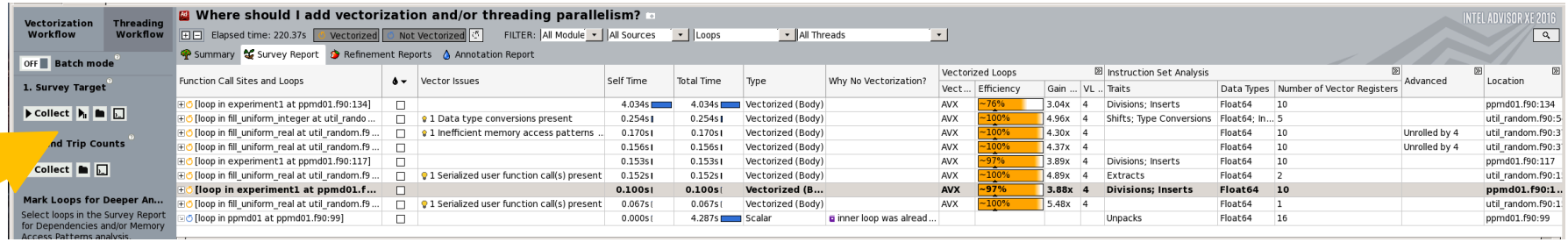


- List of hot spots

Function Call Sites and Loops		Vector Issues	Self Time	Total Time	Type	Why No Vectorization?
[loop in experiment1 at ppmd01.f...			4.034s	4.034s	Vectorized (B...	
[loop in fill_uniform_integer at util_rando ...		1 Data type conversions present	0.254s	0.254s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...		1 Inefficient memory access patterns ..	0.170s	0.170s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...			0.156s	0.156s	Vectorized (Body)	
[loop in experiment1 at ppmd01.f90:117]			0.153s	0.153s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...		1 Serialized user function call(s) present	0.152s	0.152s	Vectorized (Body)	
[loop in experiment1 at ppmd01.f90:100]			0.100s	0.100s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...		1 Serialized user function call(s) present	0.067s	0.067s	Vectorized (Body)	
[loop in ppmd01 at ppmd01.f90:99]			0.000s	4.287s	Scalar	inner loop was alread...

Intel Advisor

- Run Intel Advisor again

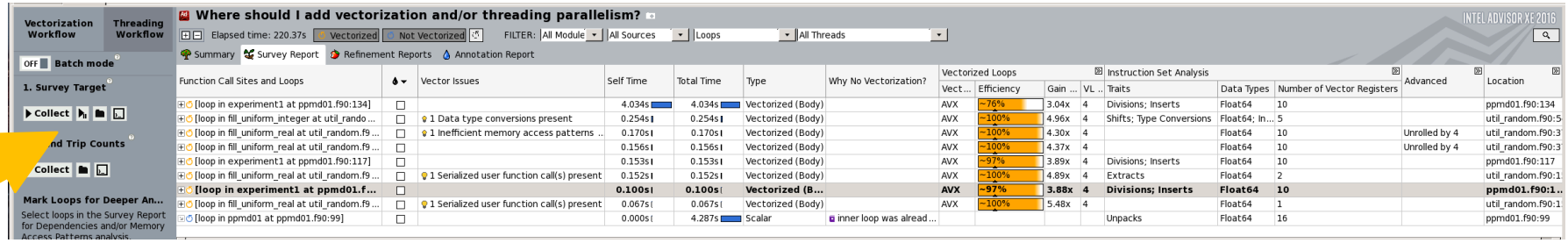


- List of hot spots

Function Call Sites and Loops		Vector Issues	Self Time	Total Time	Type	Why No Vectorization?
[loop in experiment1 at ppmd01.f...	☐		**4.034s**	**4.034s**	**Vectorized (B...**	
[loop in fill_uniform_integer at util_rando ...	☐	1 Data type conversions present	0.254s	0.254s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...	☐	1 Inefficient memory access patterns ..	0.170s	0.170s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...	☐		0.156s	0.156s	Vectorized (Body)	
[loop in experiment1 at ppmd01.f90:117]	☐		0.153s	0.153s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...	☐	1 Serialized user function call(s) present	0.152s	0.152s	Vectorized (Body)	
[loop in experiment1 at ppmd01.f90:100]	☐		0.100s	0.100s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...	☐	1 Serialized user function call(s) present	0.067s	0.067s	Vectorized (Body)	
[loop in ppmd01 at ppmd01.f90:99]	☐		0.000s	4.287s	Scalar	inner loop was alread...

Intel Advisor

- Run Intel Advisor again

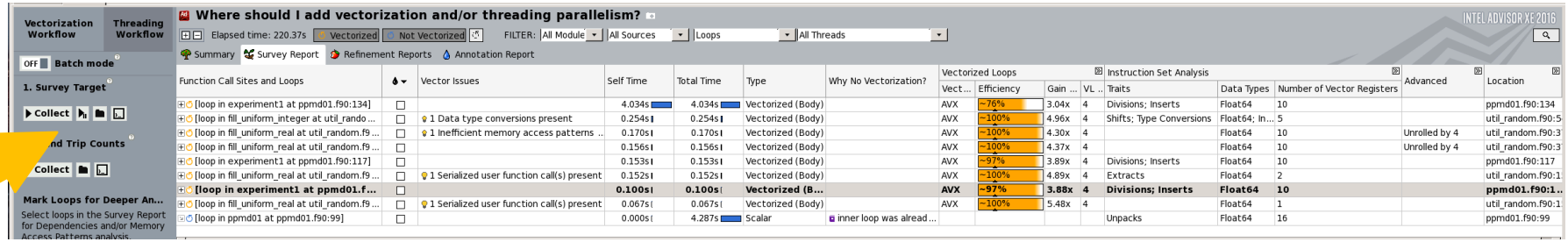


- List of hot spots

Function Call Sites and Loops		Vector Issues	Self Time	Total Time	Type	Why No Vectorization?
[loop in experiment1 at ppmd01.f...			4.034s	4.034s	Vectorized (B...	
[loop in fill_uniform_integer at util_rando ...		1 Data type conversions present	0.254s	0.254s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...		1 Inefficient memory access patterns ..	0.170s	0.170s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...			0.156s	0.156s	Vectorized (Body)	
[loop in experiment1 at ppmd01.f90:117]			0.153s	0.153s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...		1 Serialized user function call(s) present	0.152s	0.152s	Vectorized (Body)	
[loop in experiment1 at ppmd01.f90:100]			0.100s	0.100s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...		1 Serialized user function call(s) present	0.067s	0.067s	Vectorized (Body)	
[loop in ppmd01 at ppmd01.f90:99]			0.000s	4.287s	Scalar	inner loop was alread...

Intel Advisor

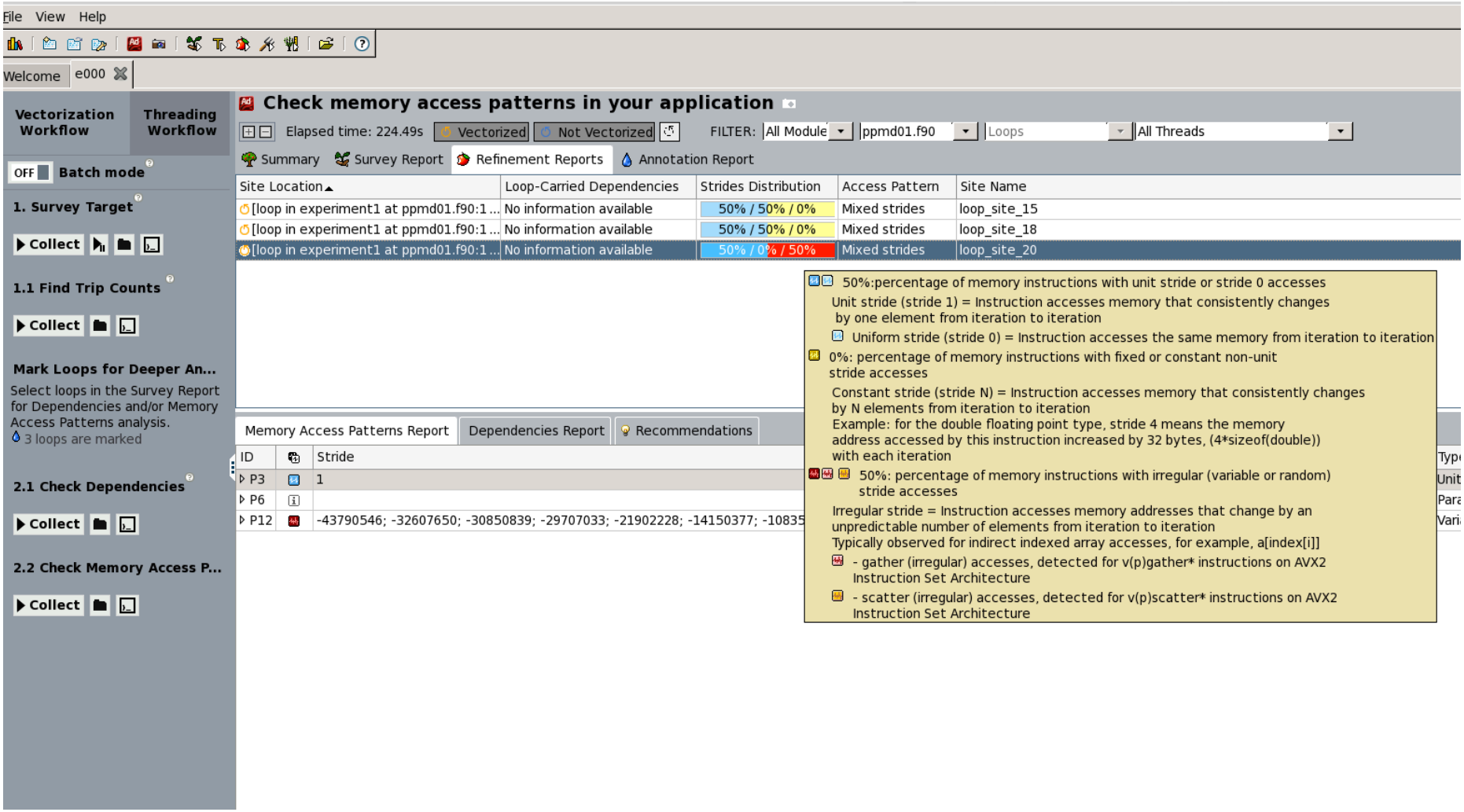
- Run Intel Advisor again



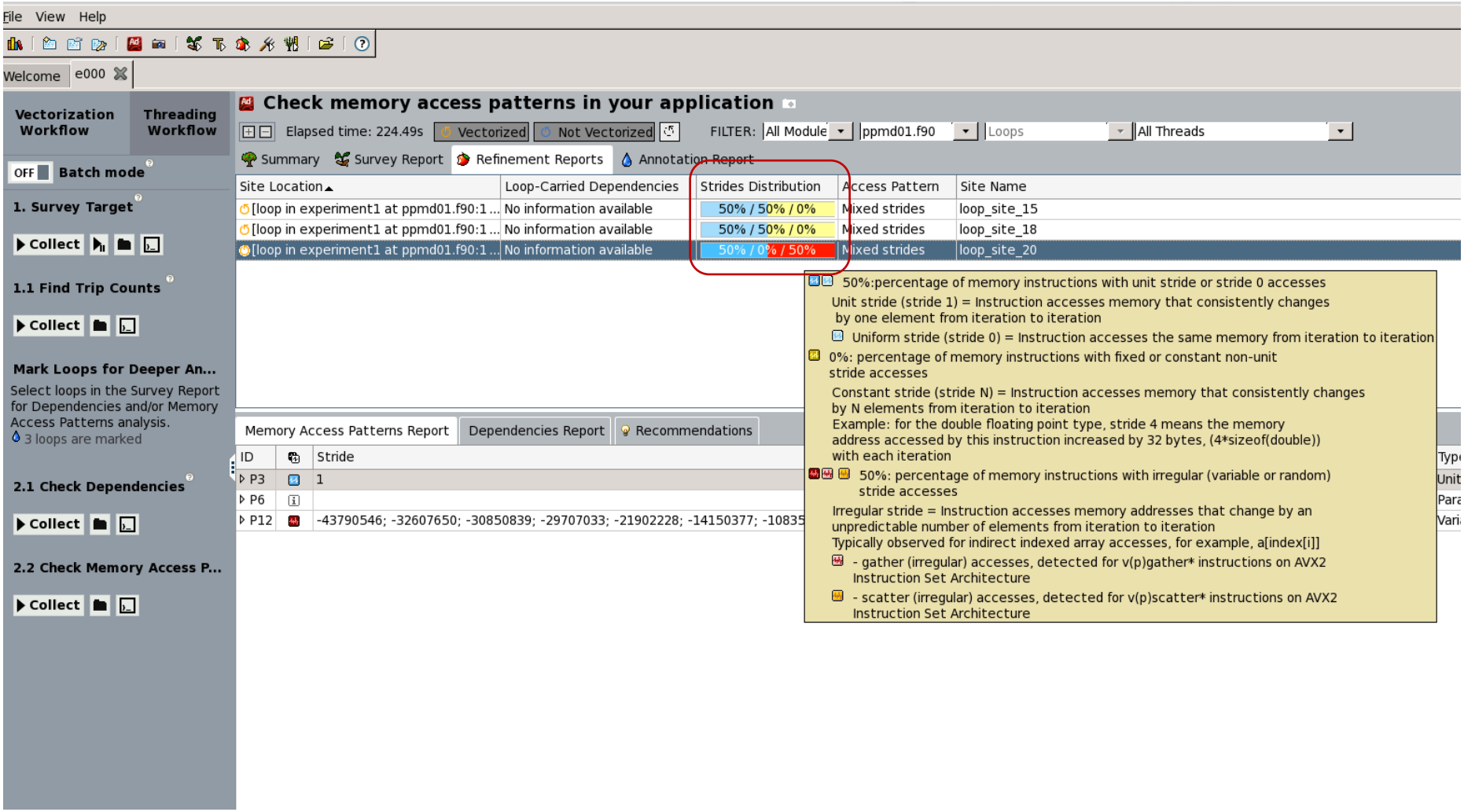
- List of hot spots

Function Call Sites and Loops		Vector Issues	Self Time	Total Time	Type	Why No Vectorization?
[loop in experiment1 at ppmd01.f...	☑		4.034s	4.034s	Vectorized (B...	
[loop in fill_uniform_integer at util_rando ...	☐	1 Data type conversions present	0.254s	0.254s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...	☐	1 Inefficient memory access patterns ..	0.170s	0.170s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...	☐		0.156s	0.156s	Vectorized (Body)	
[loop in experiment1 at ppmd01.f90:117]	☑		0.153s	0.153s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...	☐	1 Serialized user function call(s) present	0.152s	0.152s	Vectorized (Body)	
[loop in experiment1 at ppmd01.f90:100]	☑		0.100s	0.100s	Vectorized (Body)	
[loop in fill_uniform_real at util_random.f9 ...	☐	1 Serialized user function call(s) present	0.067s	0.067s	Vectorized (Body)	
[loop in ppmd01 at ppmd01.f90:99]	☐		0.000s	4.287s	Scalar	inner loop was alread ...

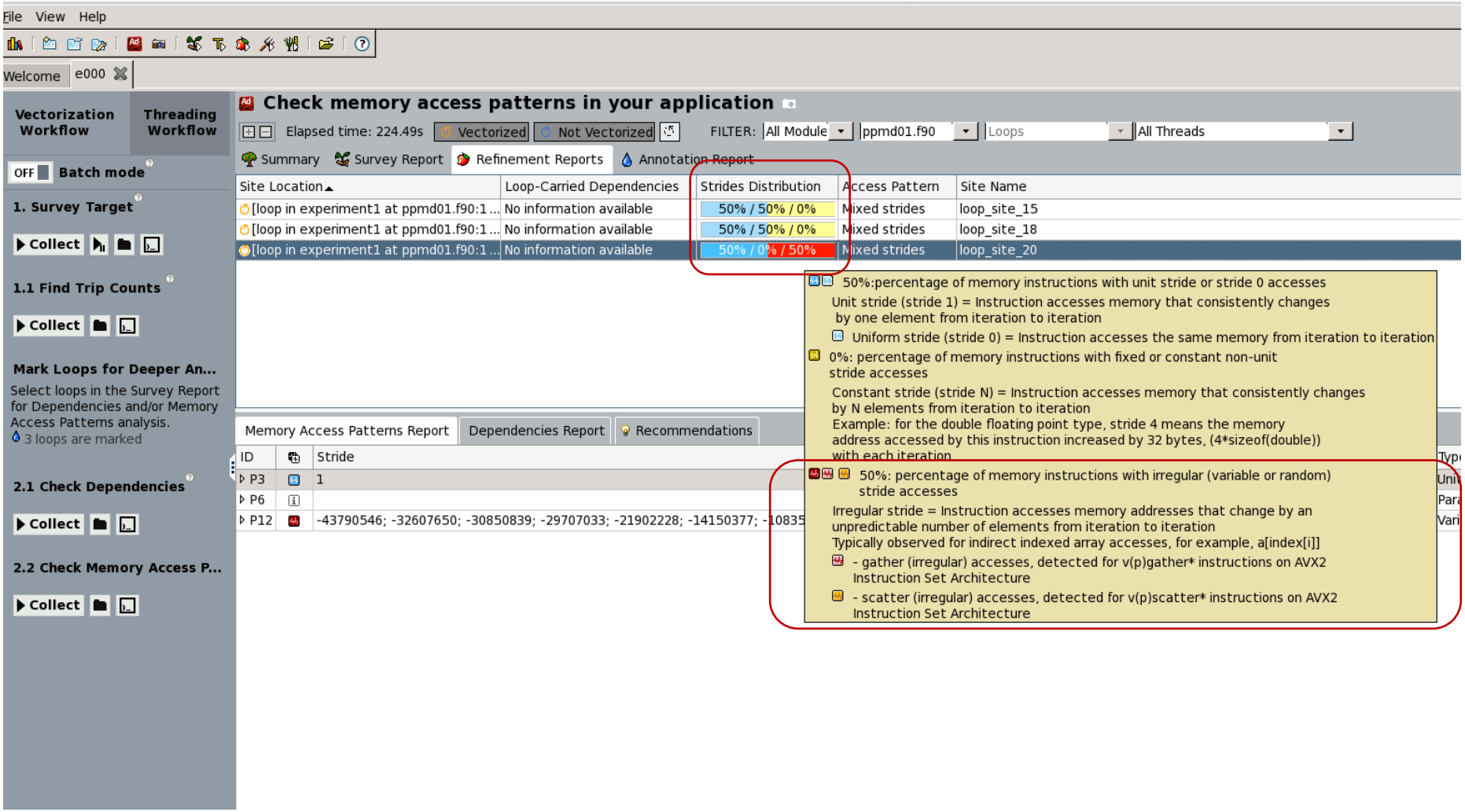
Intel Advisor



Intel Advisor



Intel Advisor

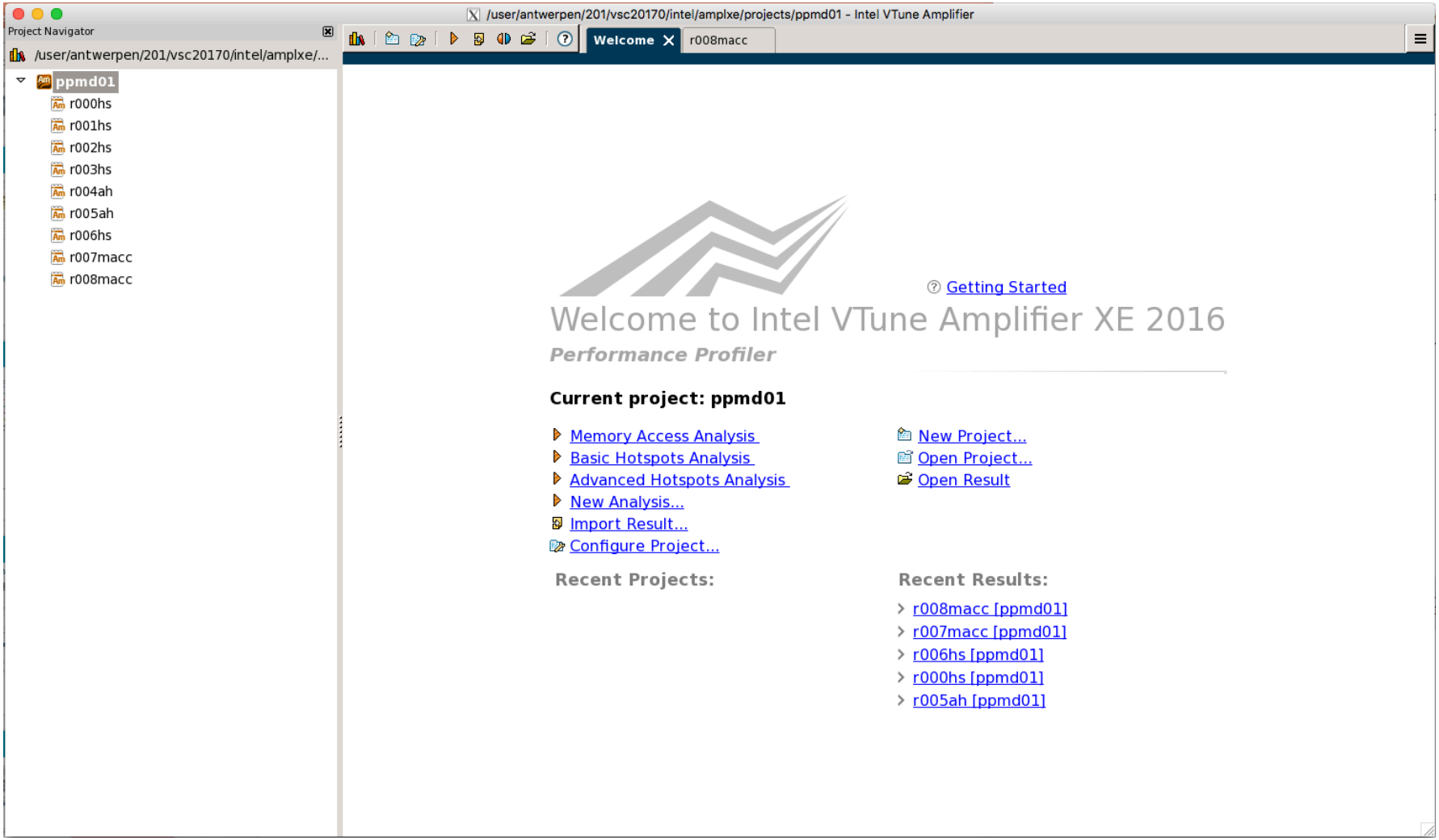


- Advisor tells us
  - Vectorization is ok
  - Strided memory access in the random access loop is a problem
- Let’s run a **memory access** analysis in VTune

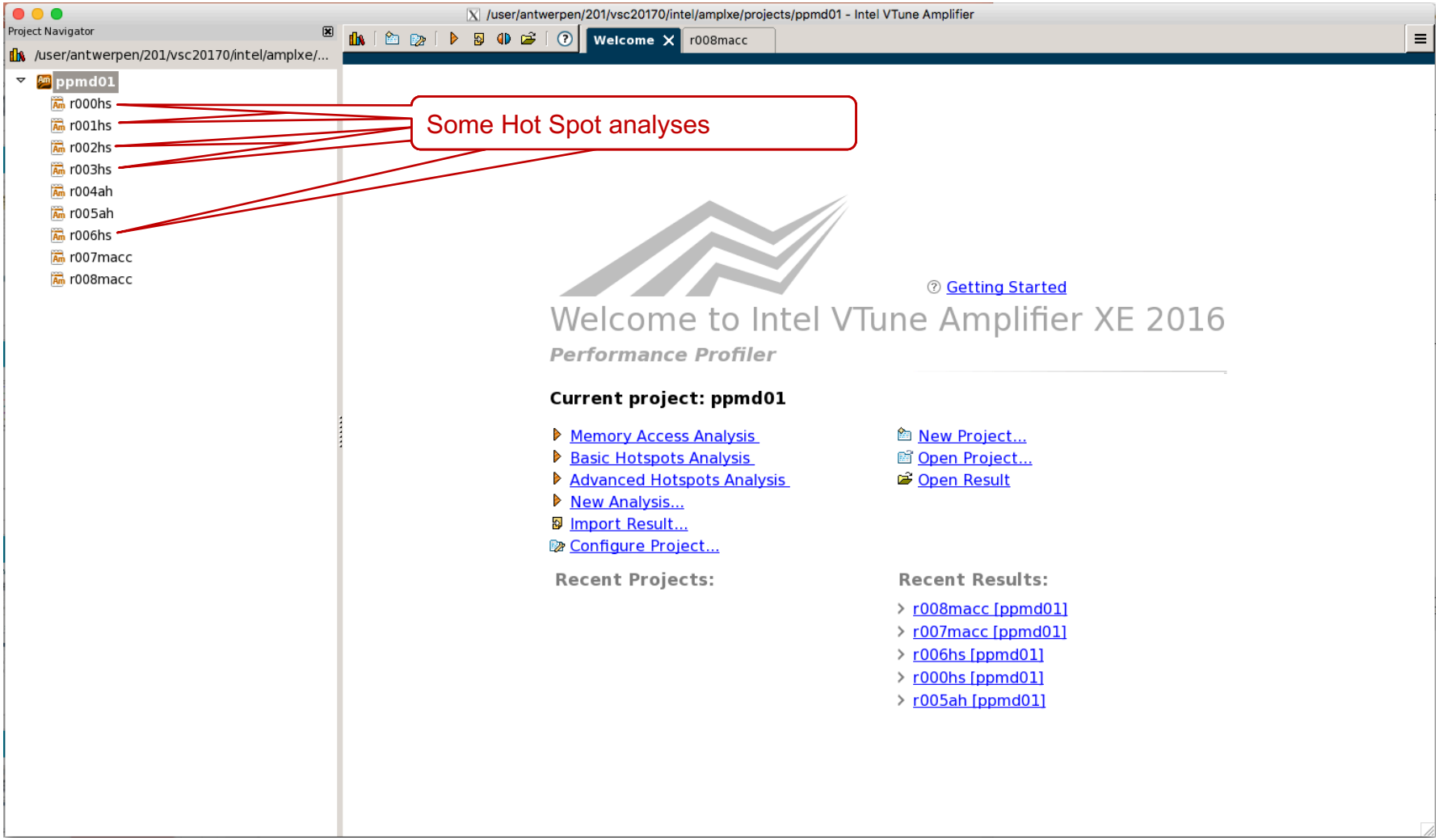
Intel Vtune Amplifier

```
$ssh –X vsc20170@login.hpc.uantwerpen.be
Last login: Thu Sep  8 16:38:25 2016 from 143.169.185.55
------------------------------------------------------------
Welcome to Hopper!
...
vsc20170@ln02 ~$ module load VTune
vsc20170@ln02 ~$ module list
Currently Loaded Modulefiles:
  1) GCCcore/5.4.0                      4) ifort/2016.3.210-GCC-5.4.0-2.26      7) VTune/2016_update3
  2) binutils/2.26-GCCcore-5.4.0        5) iccifort/2016.3.210-GCC-5.4.0-2.26
  3) icc/2016.3.210-GCC-5.4.0-2.26      6) Advisor/2016_update4
vsc20170@ln02 ~$
vsc20170@ln02 ~$ amplxe-gui &
```

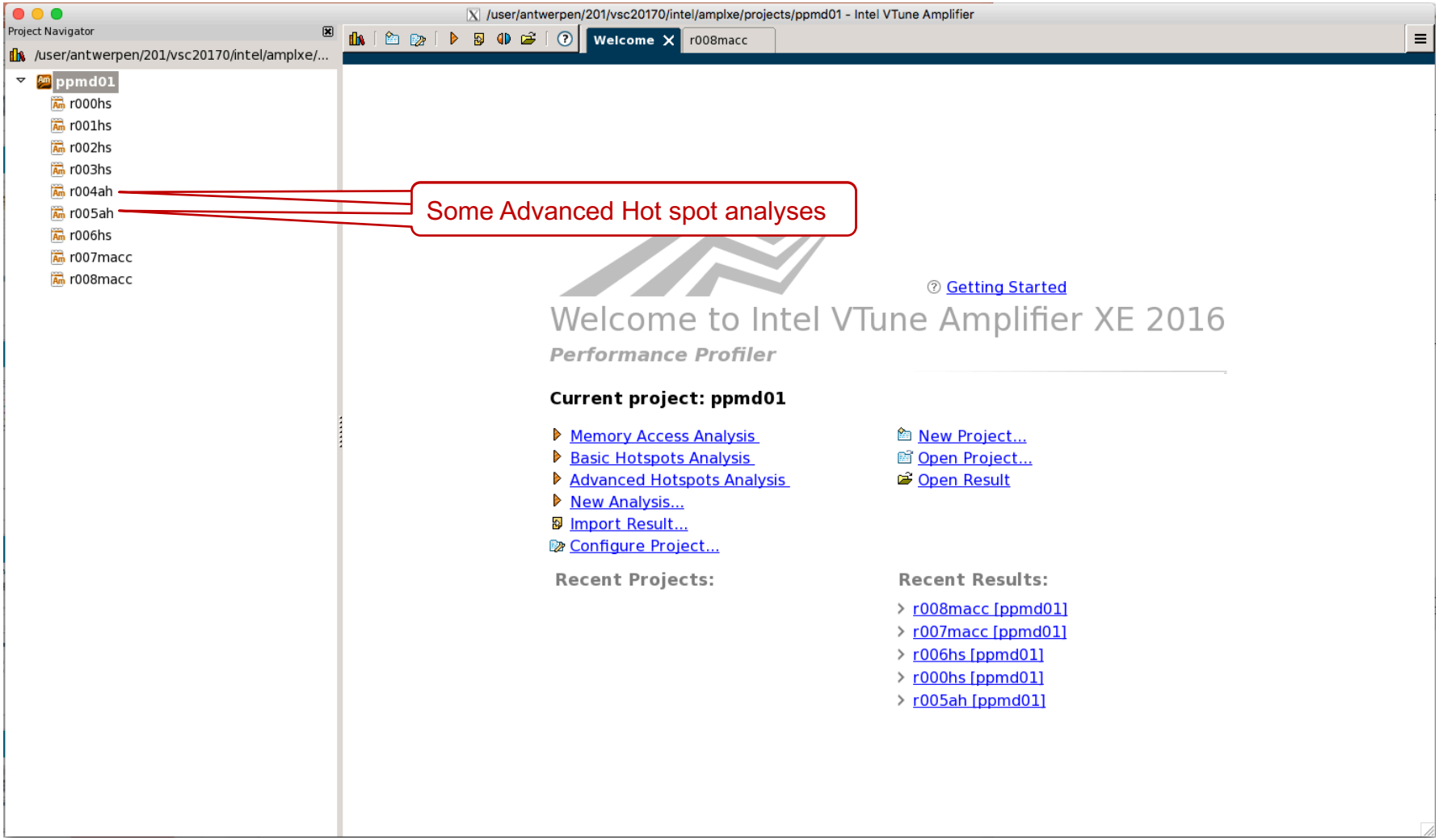
Intel Vtune Amplifier



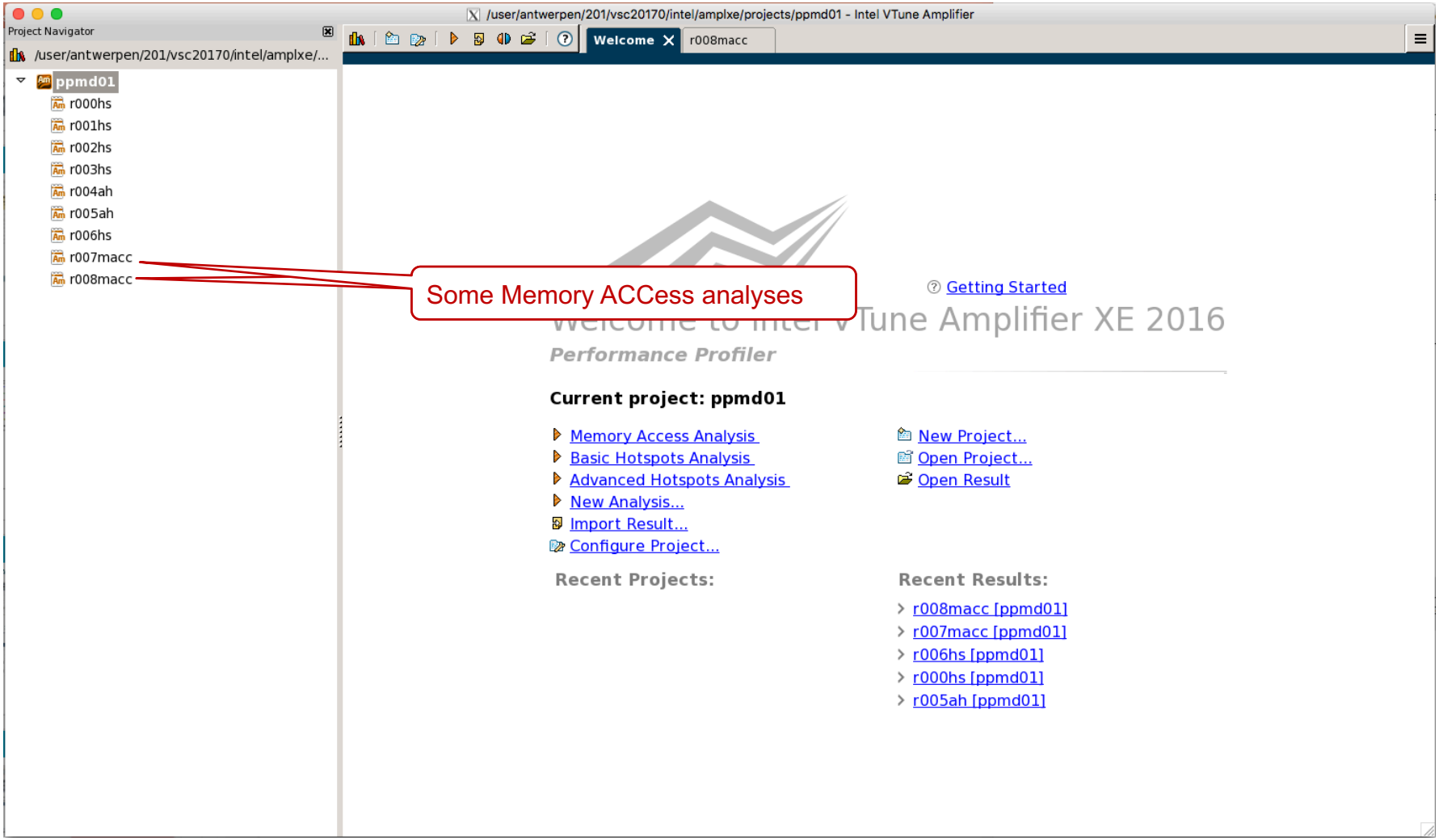
Intel Vtune Amplifier



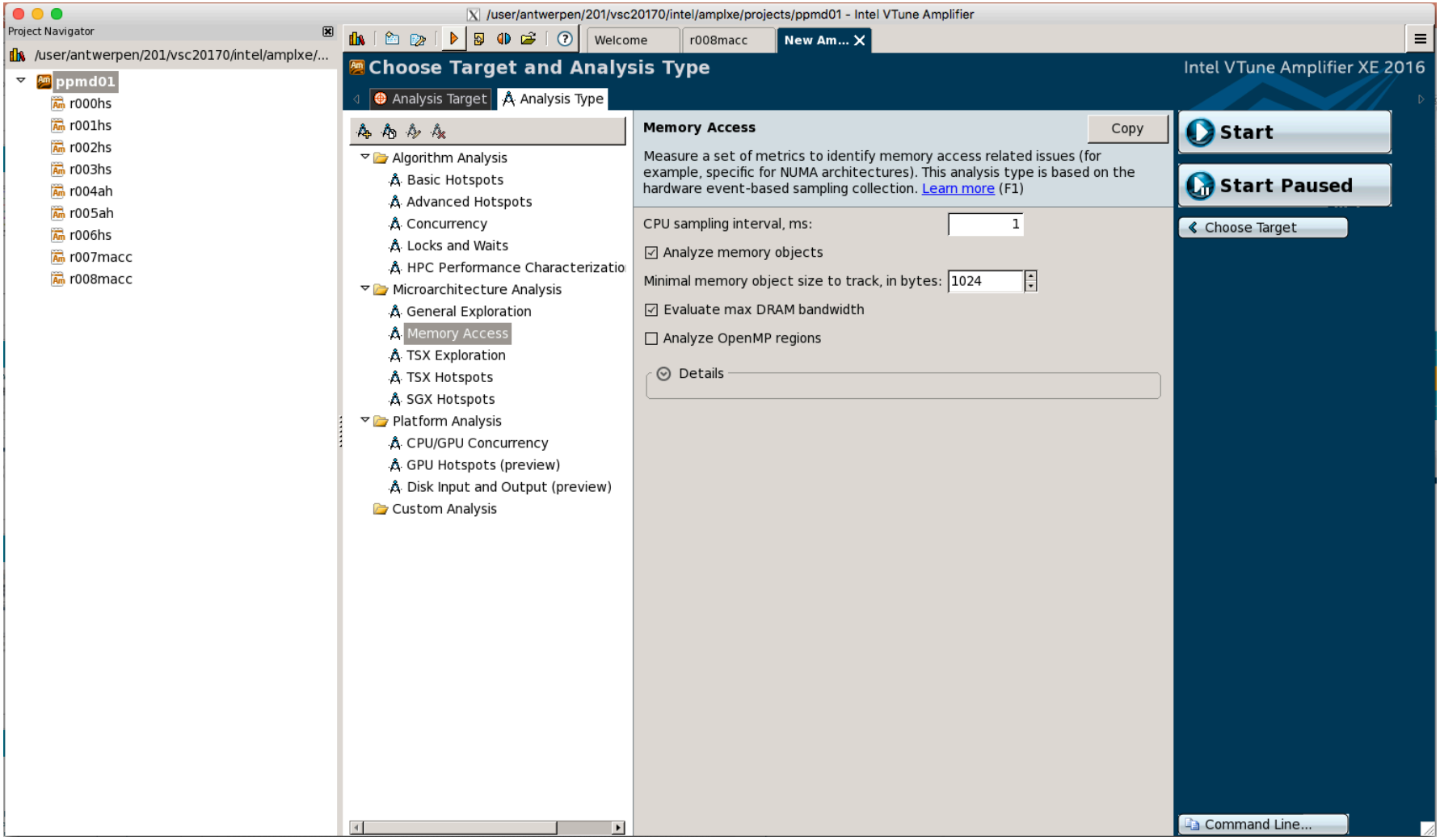
Intel Vtune Amplifier



Intel Vtune Amplifier

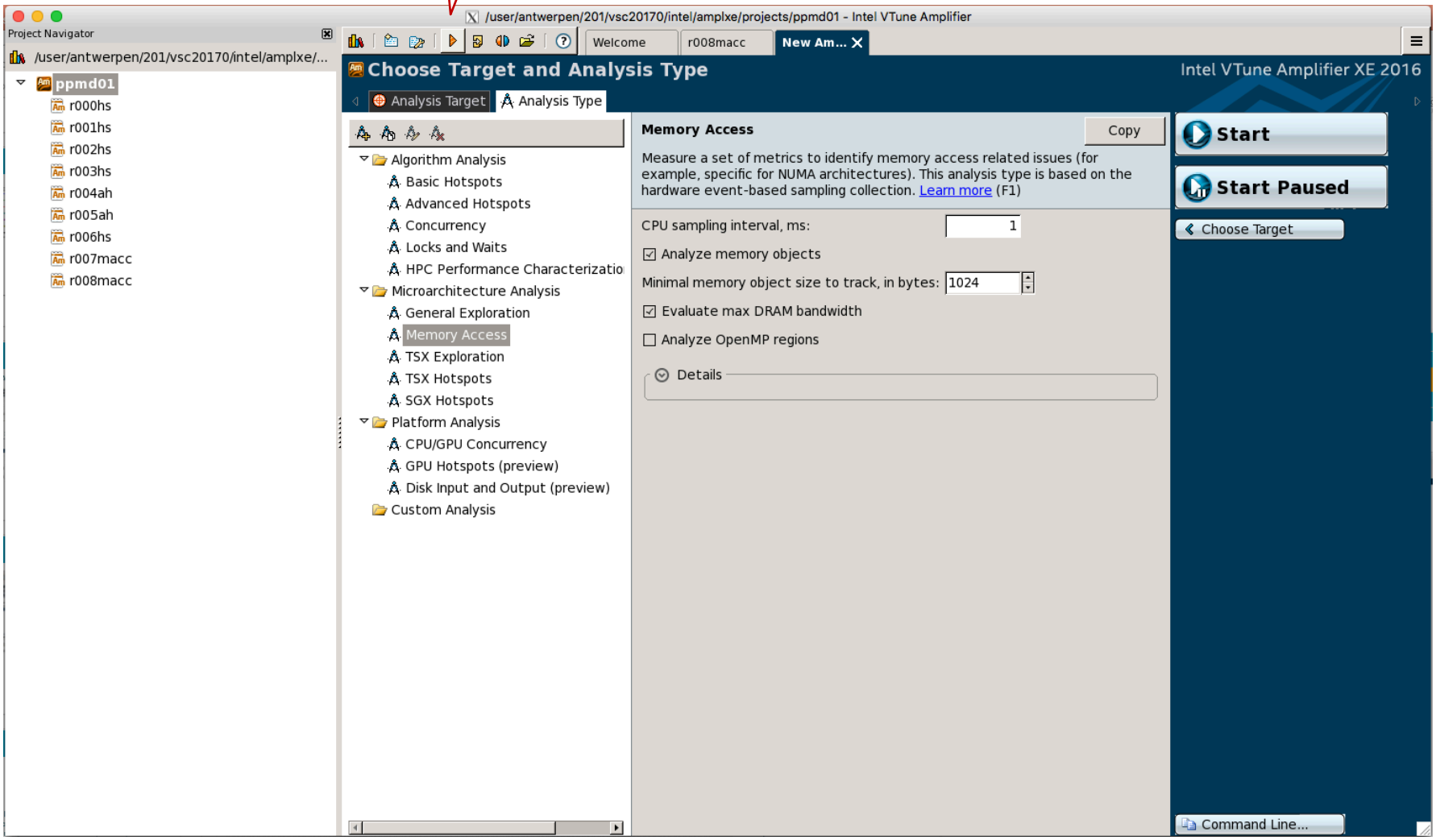


Intel Vtune Amplifier

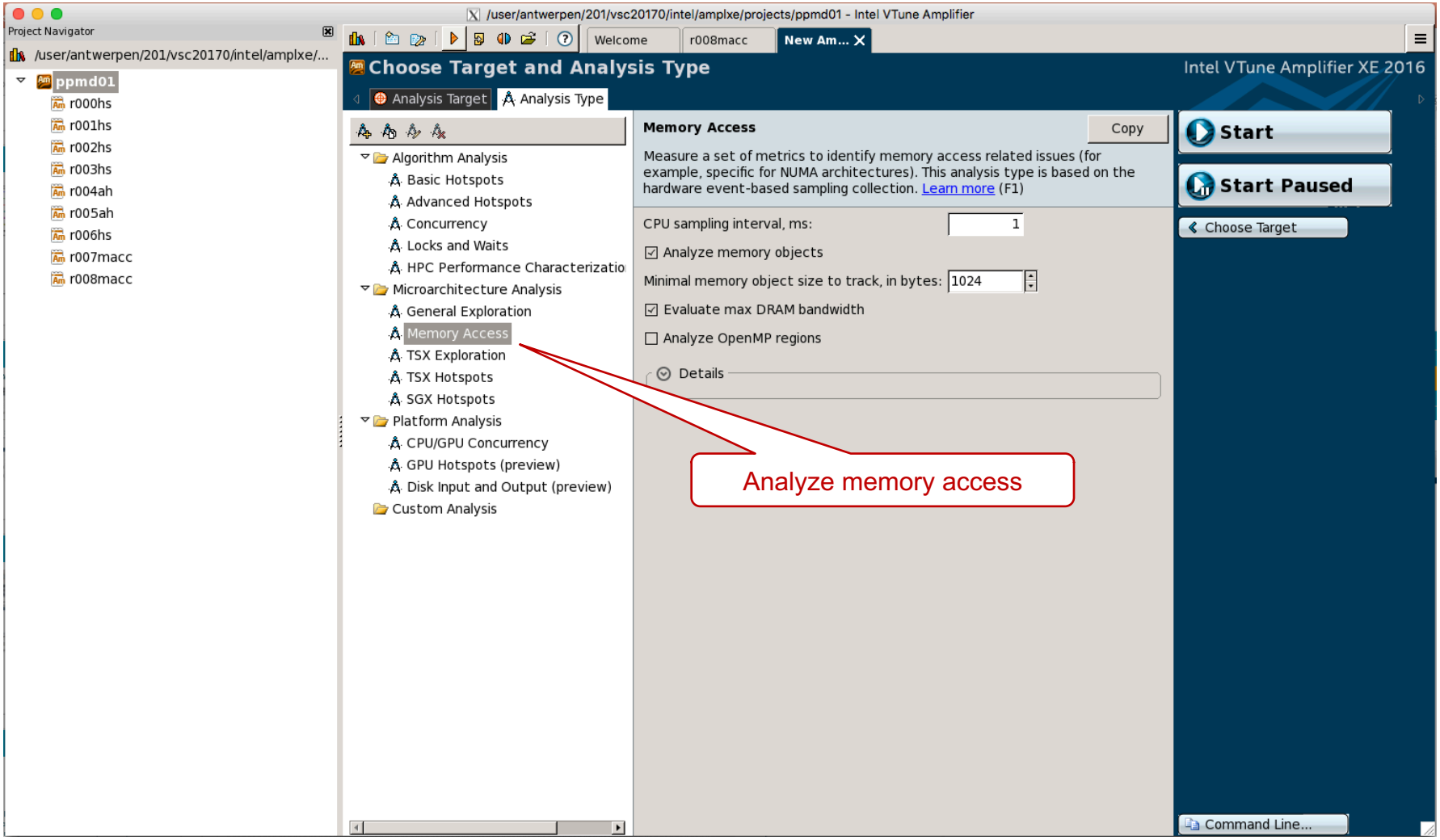


Intel Vtune Amplifier

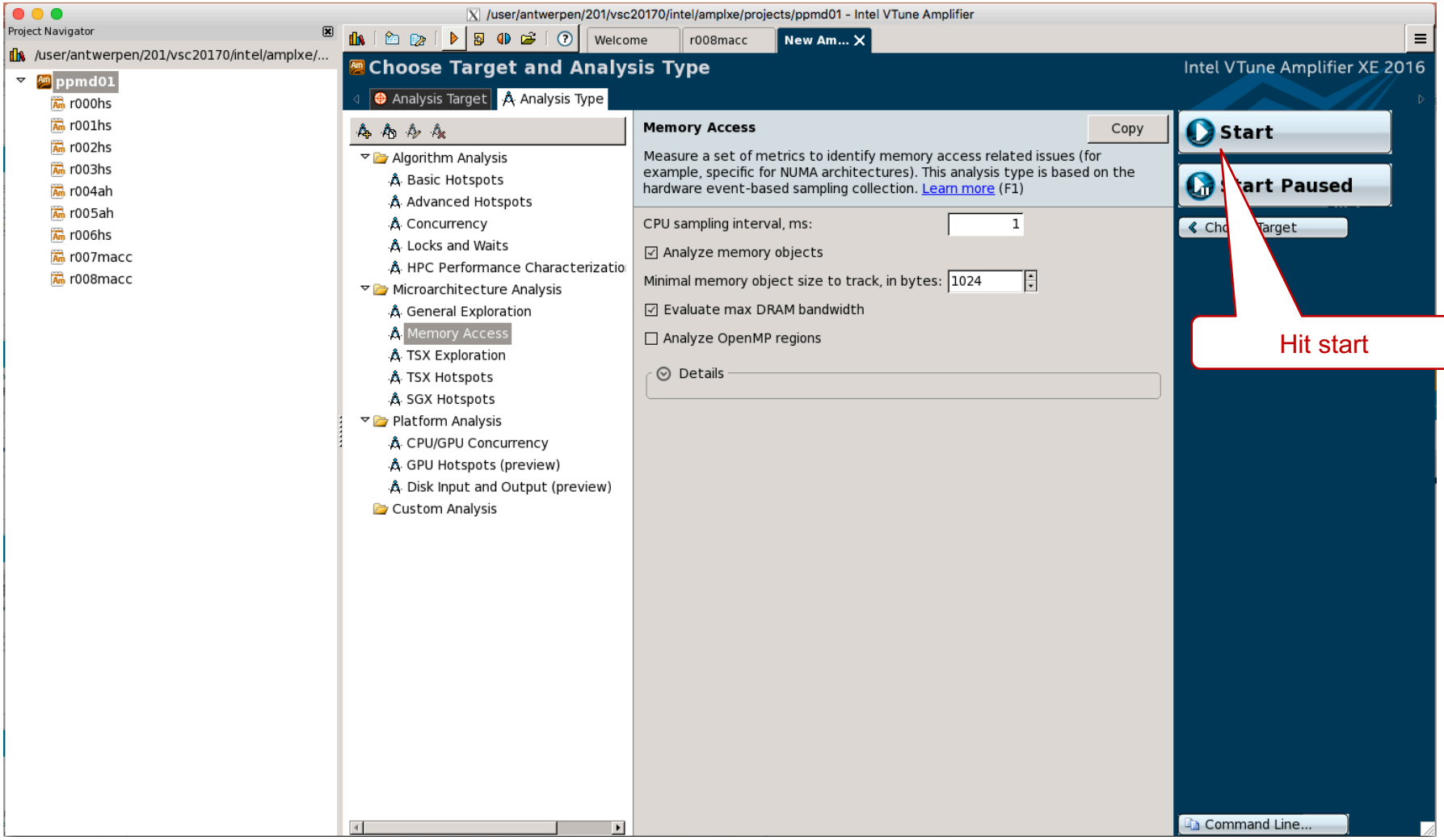
New analysis



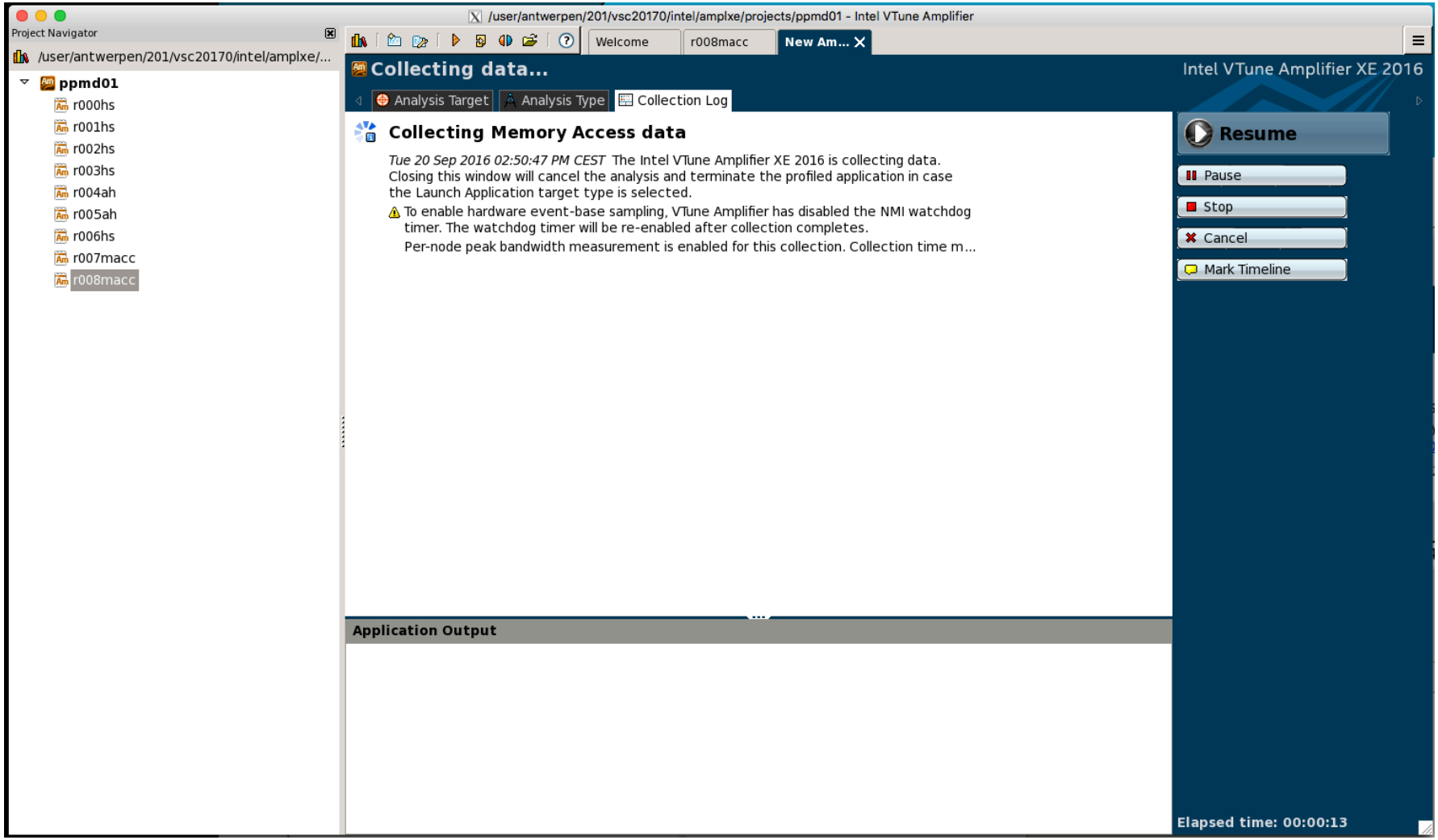
Intel Vtune Amplifier



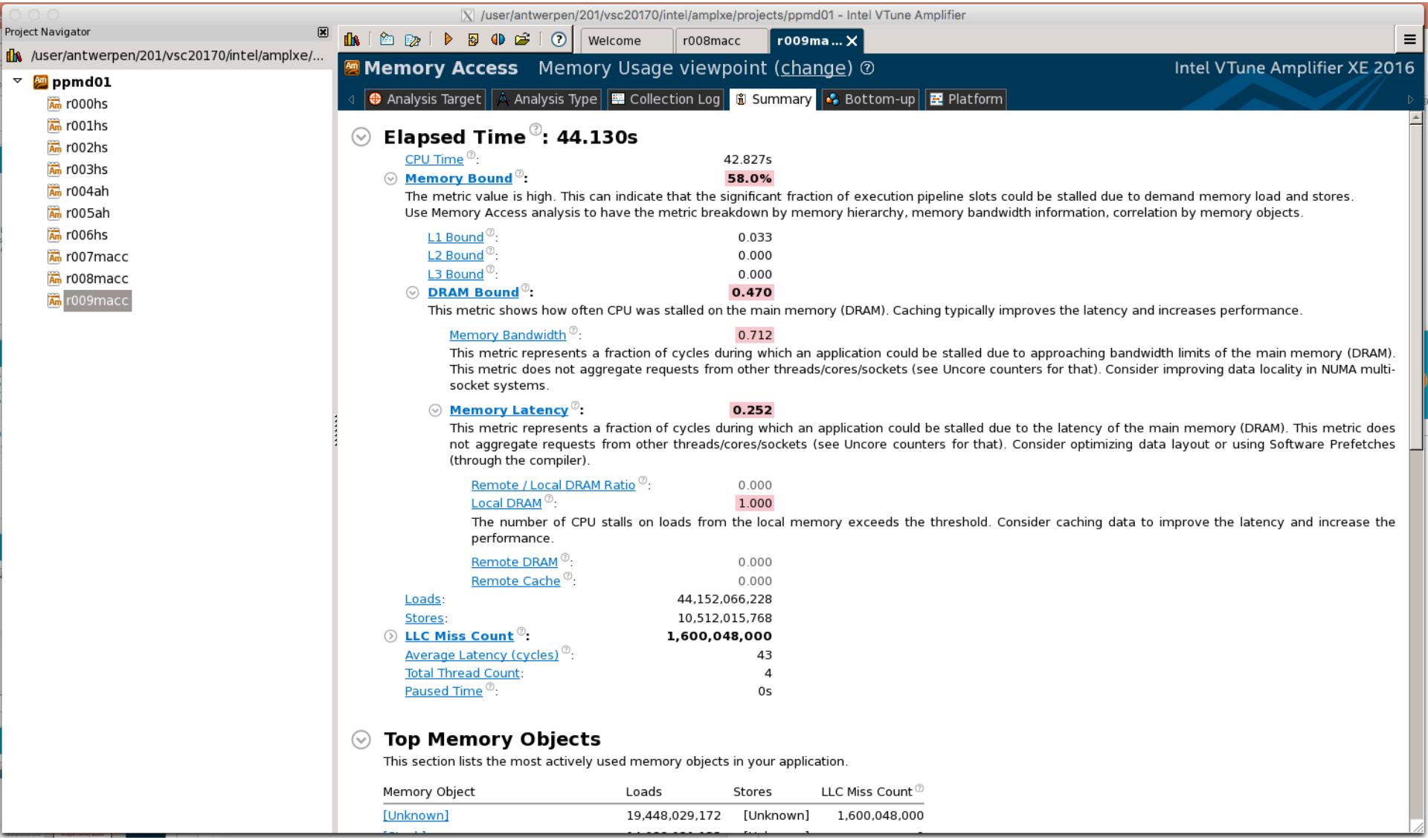
Intel Vtune Amplifier



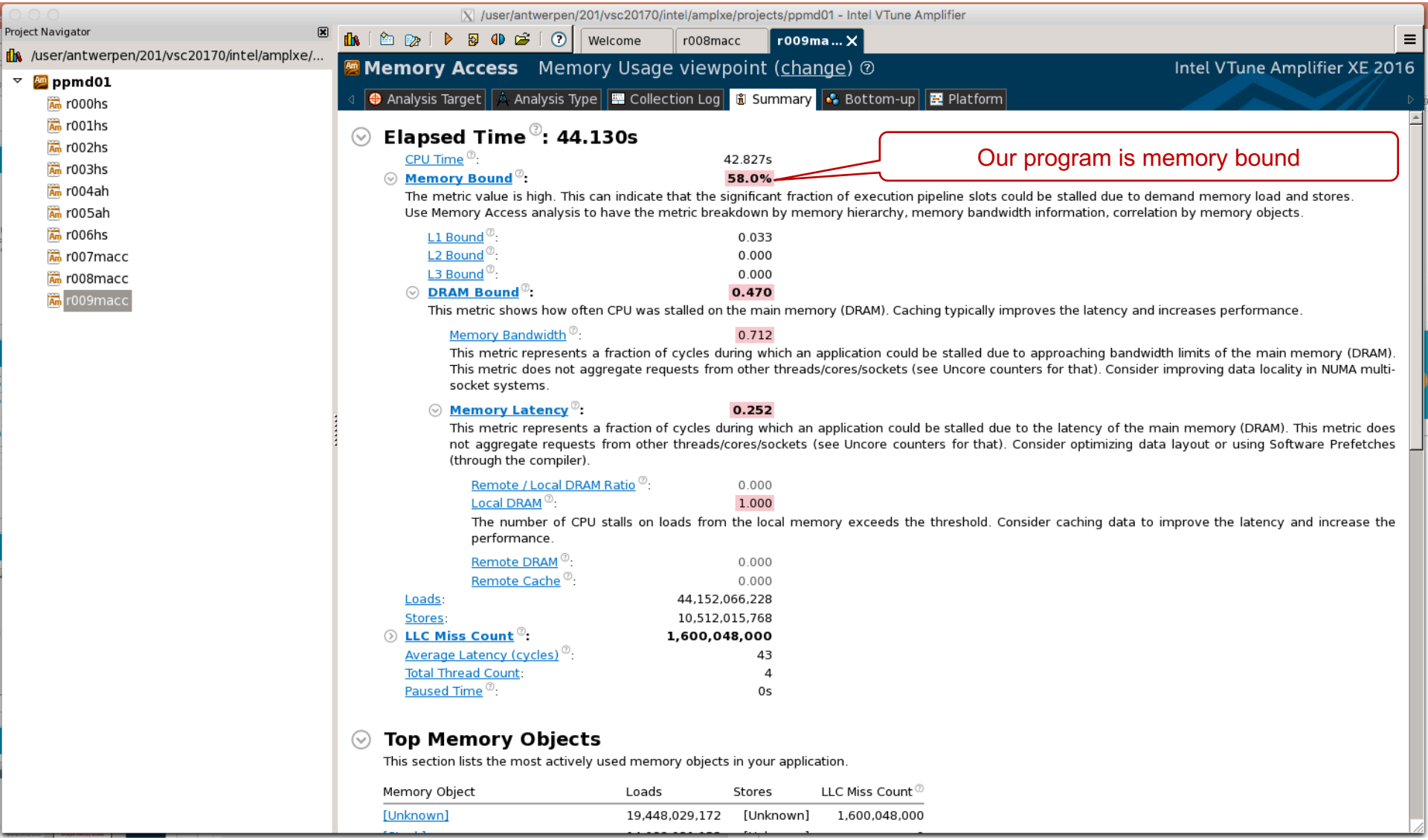
Intel Vtune Amplifier



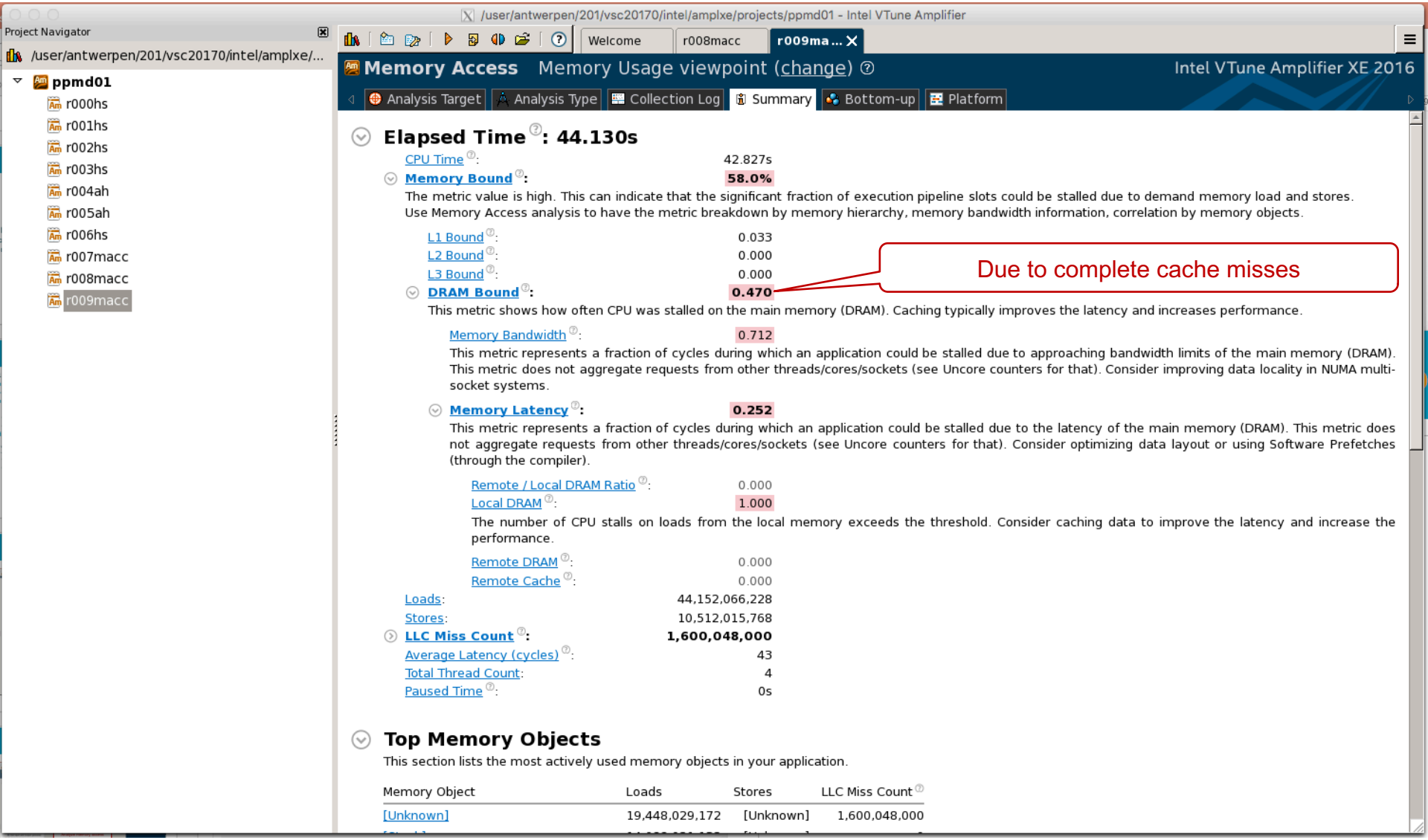
Intel Vtune Amplifier



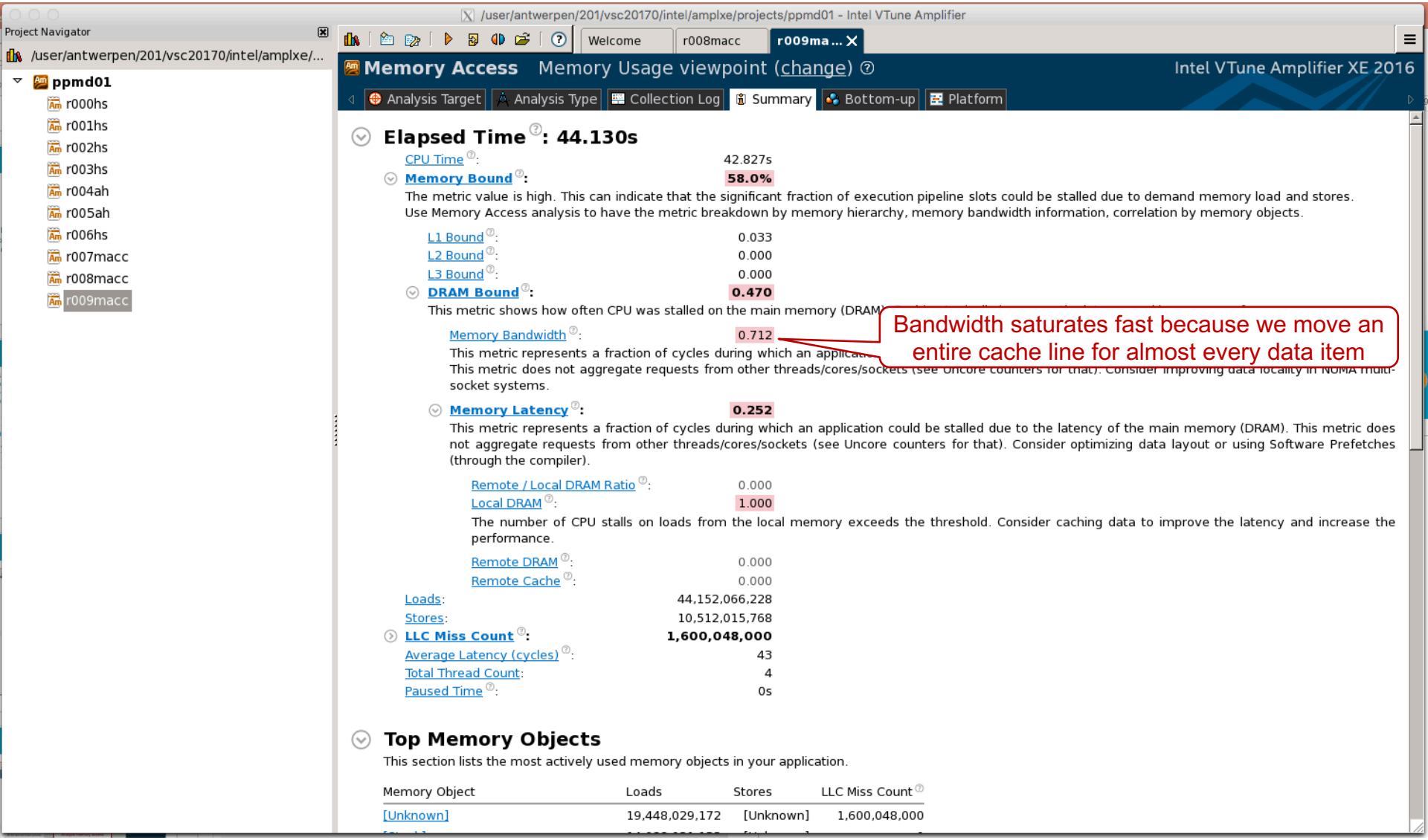
Intel Vtune Amplifier



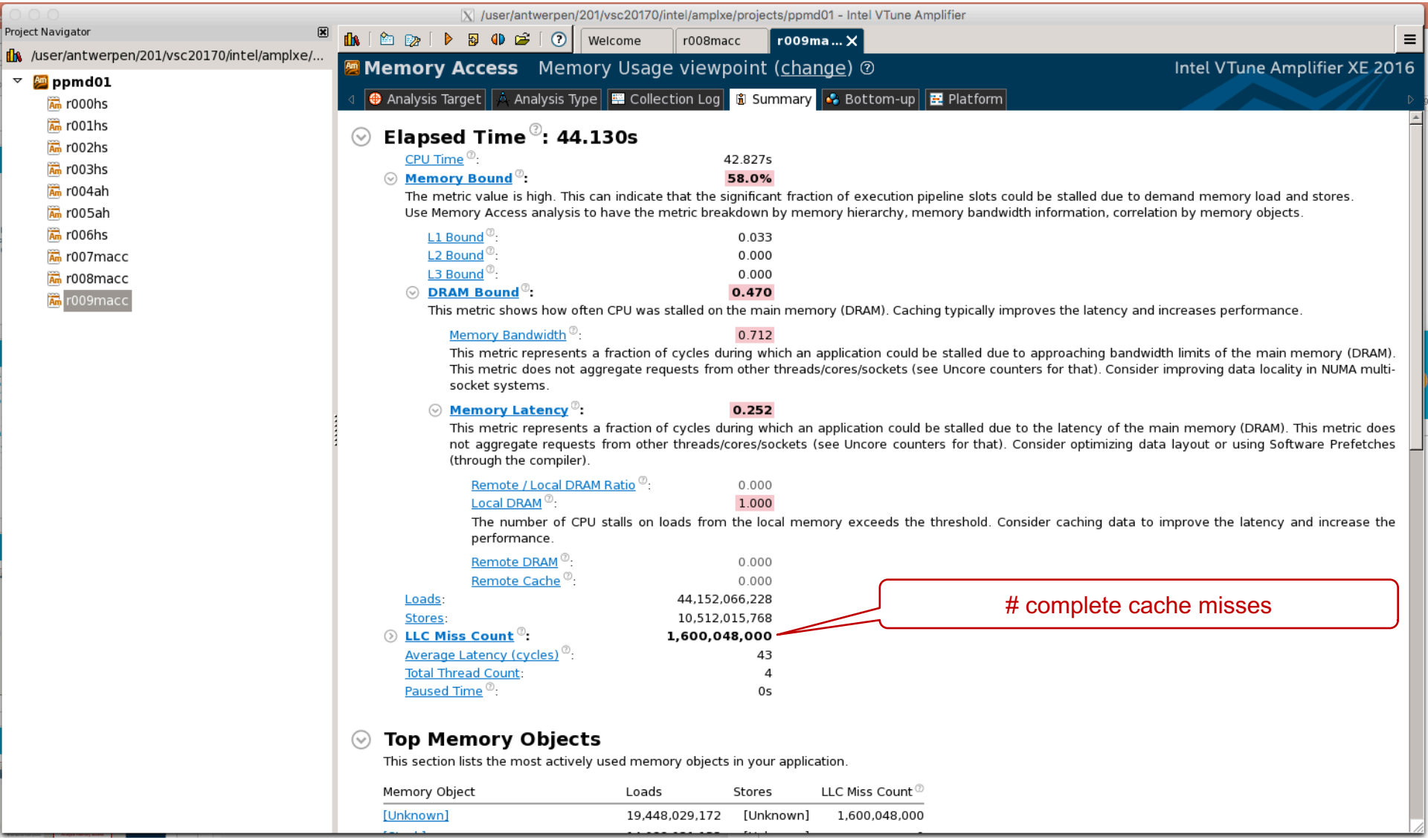
Intel Vtune Amplifier



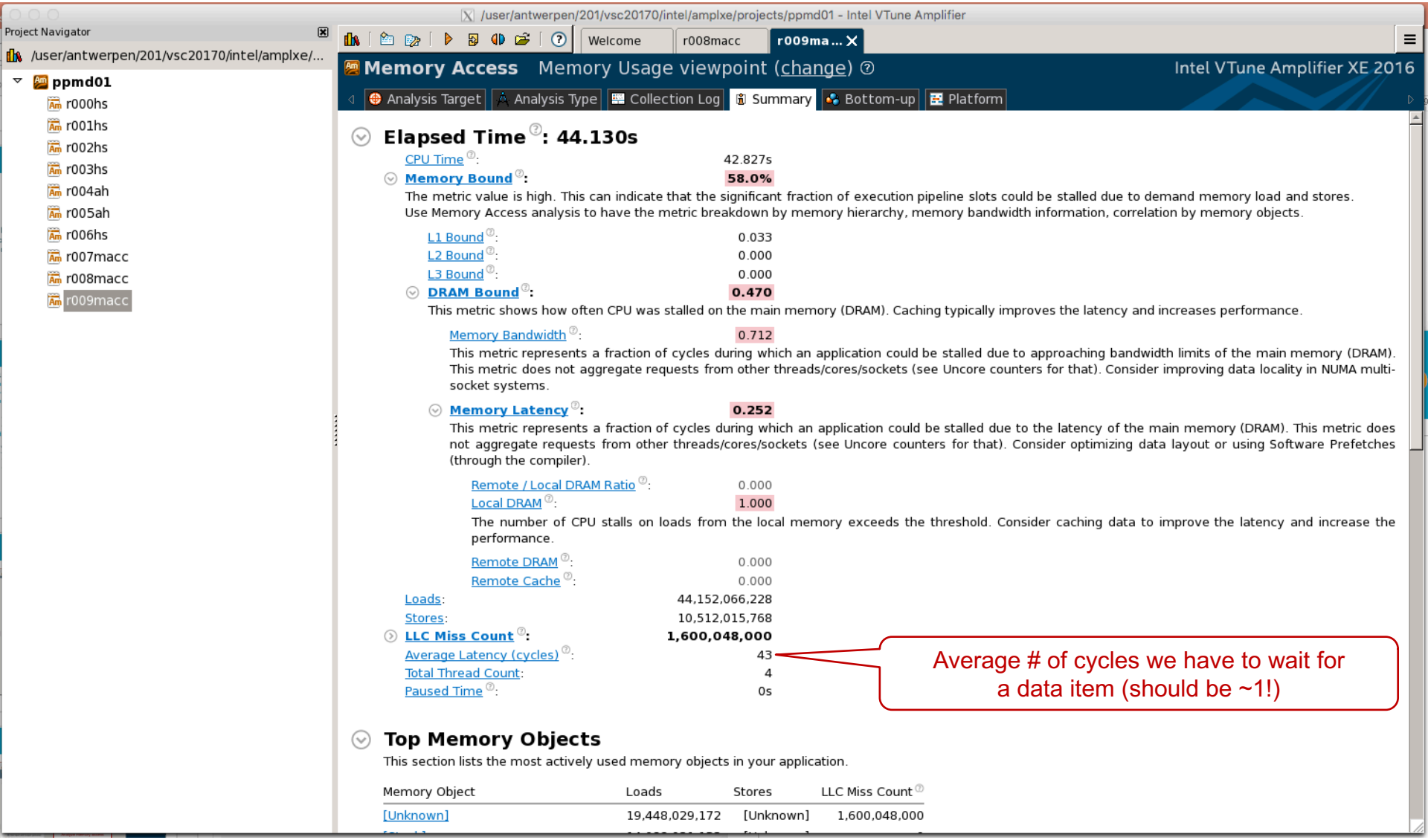
Intel Vtune Amplifier



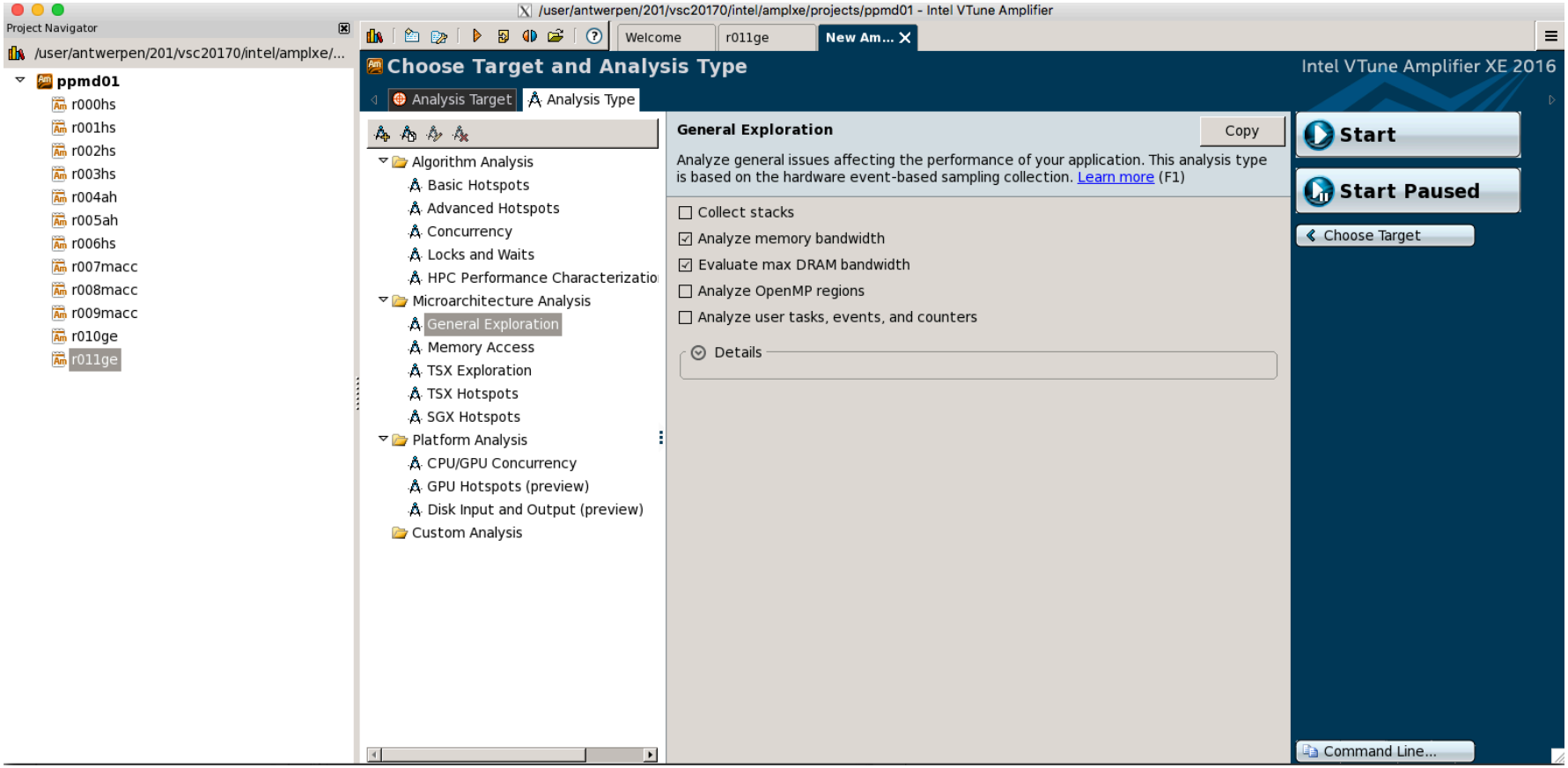
Intel Vtune Amplifier



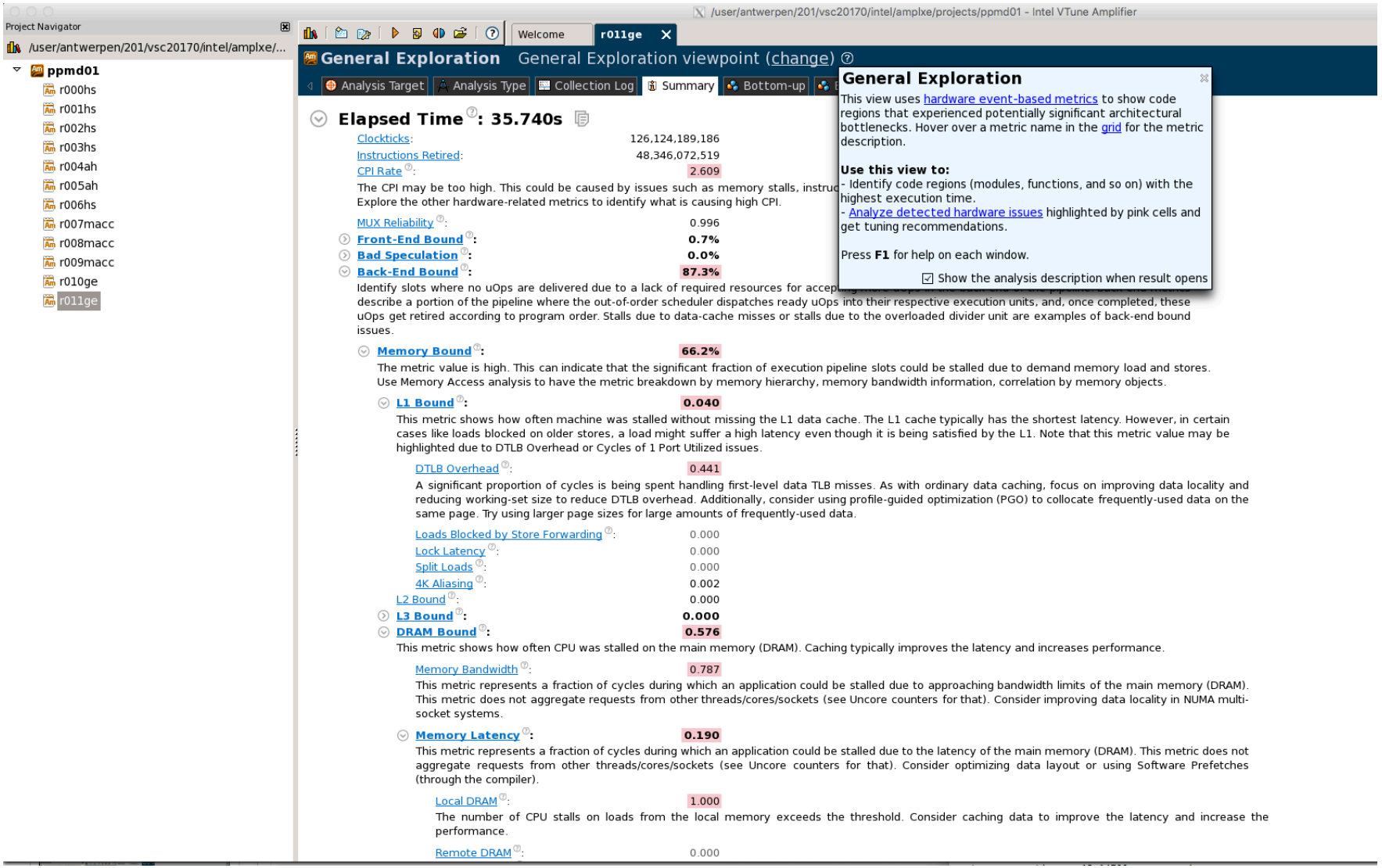
Intel Vtune Amplifier



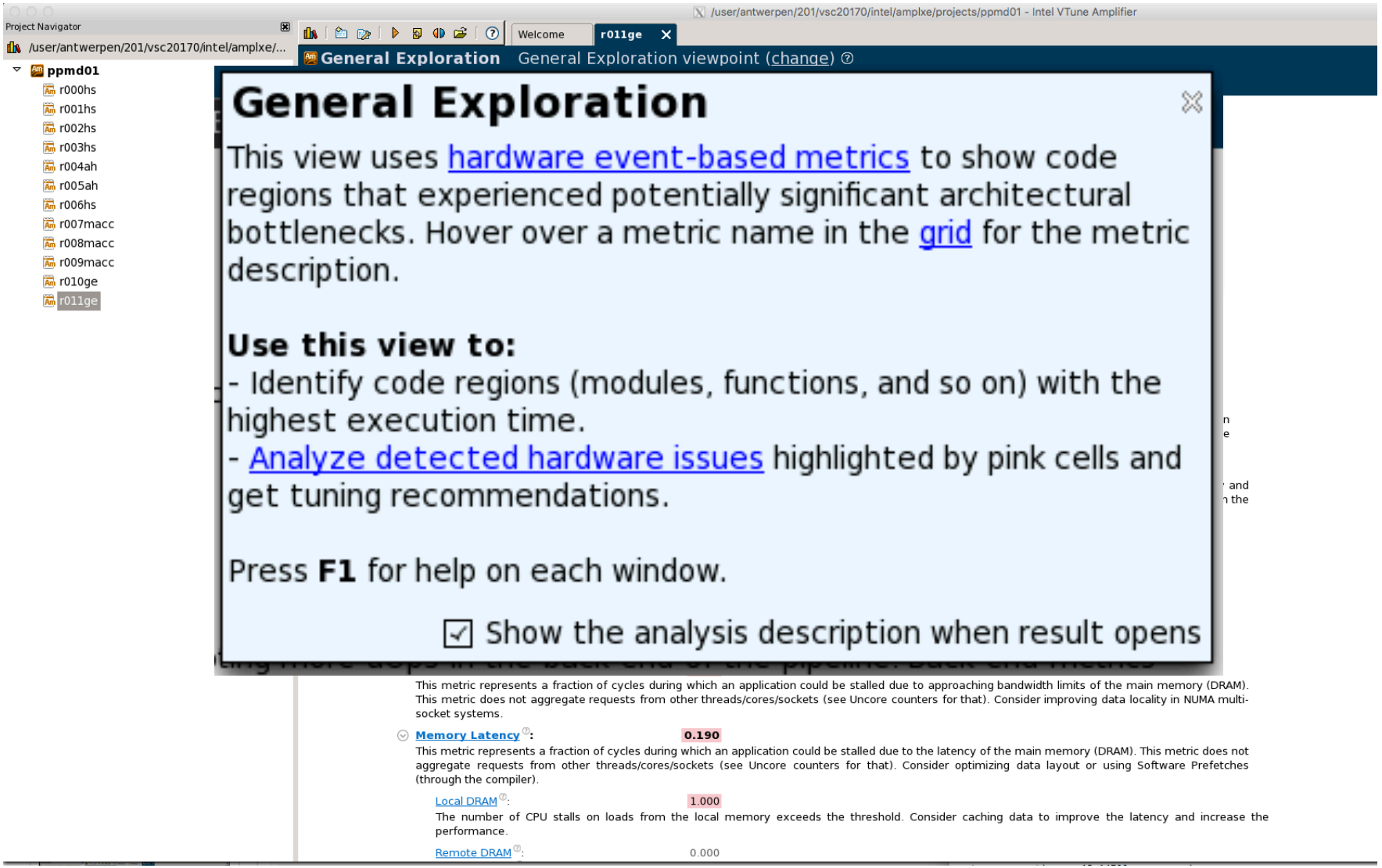
Intel Vtune Amplifier



Intel Vtune Amplifier



Intel Vtune Amplifier



Elapsed Time: 35.740s

Clockticks:	126,124,189,186
Instructions Retired:	48,346,072,519
CPI Rate:	2.609

The CPI may be too high. This could be caused by issues such as memory stalls, instruction starvation, branch misprediction or long latency instructions. Explore the other hardware-related metrics to identify what is causing high CPI.

MUX Reliability:	0.996
Front-End Bound:	**0.7%**
Bad Speculation:	**0.0%**
Back-End Bound:	**87.3%**

Identify slots where no uOps are delivered due to a lack of required resources for accepting more uOps in the back-end of the pipeline. Back-end metrics describe a portion of the pipeline where the out-of-order scheduler dispatches ready uOps into their respective execution units, and, once completed, these uOps get retired according to program order. Stalls due to data-cache misses or stalls due to the overloaded divider unit are examples of back-end bound issues.

Memory Bound: 66.2%

The metric value is high. This can indicate that the significant fraction of execution pipeline slots could be stalled due to demand memory load and stores. Use Memory Access analysis to have the metric breakdown by memory hierarchy, memory bandwidth information, correlation by memory objects.

L1 Bound: 0.040

This metric shows how often machine was stalled without missing the L1 data cache. The L1 cache typically has the shortest latency. However, in certain cases like loads blocked on older stores, a load might suffer a high latency even though it is being satisfied by the L1. Note that this metric value may be highlighted due to DTLB Overhead or Cycles of 1 Port Utilized issues.

DTLB Overhead: 0.441

A significant proportion of cycles is being spent handling first-level data TLB misses. As with ordinary data caching, focus on improving data locality and reducing working-set size to reduce DTLB overhead. Additionally, consider using profile-guided optimization (PGO) to collocate frequently-used data on the same page. Try using larger page sizes for large amounts of frequently-used data.

Loads Blocked by Store Forwarding:	0.000
Lock Latency:	0.000
Split Loads:	0.000
4K Aliasing:	0.002
L2 Bound:	0.000
L3 Bound:	**0.000**
DRAM Bound:	**0.576**

This metric shows how often CPU was stalled on the main memory (DRAM). Caching typically improves the latency and increases performance.

Memory Bandwidth: 0.787

This metric represents a fraction of cycles during which an application could be stalled due to approaching bandwidth limits of the main memory (DRAM). This metric does not aggregate requests from other threads/cores/sockets (see Uncore counters for that). Consider improving data locality in NUMA multi-socket systems.

Memory Latency: 0.190

This metric represents a fraction of cycles during which an application could be stalled due to the latency of the main memory (DRAM). This metric does not aggregate requests from other threads/cores/sockets (see Uncore counters for that). Consider optimizing data layout or using Software Prefetches (through the compiler).

Local DRAM: 1.000

The number of CPU stalls on loads from the local memory exceeds the threshold. Consider caching data to improve the latency and increase the performance.

Remote DRAM:	0.000
Remote Cache:	0.000
Store Bound:	**0.134**

Intel Vtune Amplifier

Elapsed Time: 35.740s

Clockticks:	126,124,189,186
Instructions Retired:	48,346,072,519
CPI Rate:	2.609

The CPI may be too high. This could be caused by issues such as memory stalls, instru
Explore the other hardware-related metrics to identify what is causing high CPI.

MUX Reliability:	0.996
Front-End Bound:	**0.7%**
Bad Speculation:	**0.0%**
Back-End Bound:	**87.3%**

Identify slots where no uOps are delivered due to a lack of required resources for accepting more uOps in the back-end of the pipeline. Back-end metrics describe a portion of the pipeline where the out-of-order scheduler dispatches ready uOps into their respective execution units, and, once completed, these uOps get retired according to program order. Stalls due to data-cache misses or stalls due to the overloaded divider unit are examples of back-end bound issues.

Memory Bound: 66.2%

The metric value is high. This can indicate that the significant fraction of execution pipeline slots could be stalled due to demand memory load and stores. Use Memory Access analysis to have the metric breakdown by memory hierarchy, memory bandwidth information, correlation by memory objects.

L1 Bound: 0.040

This metric shows how often machine was stalled without missing the L1 data cache. The L1 cache typically has the shortest latency. However, in certain cases like loads blocked on older stores, a load might suffer a high latency even though it is being satisfied by the L1. Note that this metric value may be highlighted due to DTLB Overhead or Cycles of 1 Port Utilized issues.

DTLB Overhead: 0.441

A significant proportion of cycles is being spent handling first-level data TLB misses. As with ordinary data caching, focus on improving data locality and reducing working-set size to reduce DTLB overhead. Additionally, consider using profile-guided optimization (PGO) to collocate frequently-used data on the same page. Try using larger page sizes for large amounts of frequently-used data.

Loads Blocked by Store Forwarding:	0.000
Lock Latency:	0.000
Split Loads:	0.000
4K Aliasing:	0.002
L2 Bound:	0.000
L3 Bound:	**0.000**
DRAM Bound:	**0.576**

This metric shows how often CPU was stalled on the main memory (DRAM). Caching typically improves the latency and increases performance.

Memory Bandwidth: 0.787

This metric represents a fraction of cycles during which an application could be stalled due to approaching bandwidth limits of the main memory (DRAM). This metric does not aggregate requests from other threads/cores/sockets (see Uncore counters for that). Consider improving data locality in NUMA multi-socket systems.

Memory Latency: 0.190

This metric represents a fraction of cycles during which an application could be stalled due to the latency of the main memory (DRAM). This metric does not aggregate requests from other threads/cores/sockets (see Uncore counters for that). Consider optimizing data layout or using Software Prefetches (through the compiler).

Local DRAM: 1.000

The number of CPU stalls on loads from the local memory exceeds the threshold. Consider caching data to improve the latency and increase the performance.

Remote DRAM:	0.000
Remote Cache:	0.000
Store Bound:	**0.134**

CPI = cycles per instruction
Peak performance corresponds to 4 instructions per cycle in DP vectorized code. Hence CPI should be between 0.25 and 0.5.

Elapsed Time: 35.740s

Clockticks:	126,124,189,186
Instructions Retired:	48,346,072,519
CPI Rate:	2.609

The CPI may be too high. This could be caused by issues such as memory stalls, instruction starvation, branch misprediction or long latency instructions. Explore the other hardware-related metrics to identify what is causing high CPI.

MUX Reliability:	0.996
Front-End Bound:	**0.7%**
Bad Speculation:	**0.0%**
Back-End Bound:	**87.3%**

Identify slots where no uOps are delivered due to a lack of required resources for acce
describe a portion of the pipeline where the out-of-order scheduler dispatches ready uOp
uOps get retired according to program order. Stalls due to data-cache misses or stalls due to the overloaded divider unit are examples of back-end bound issues.

Pipeline stalls (because the data is not arriving in time)

Memory Bound:	**66.2%**

The metric value is high. This can indicate that the significant fraction of execution pipeline slots could be stalled due to demand memory load and stores. Use Memory Access analysis to have the metric breakdown by memory hierarchy, memory bandwidth information, correlation by memory objects.

L1 Bound:	**0.040**

This metric shows how often machine was stalled without missing the L1 data cache. The L1 cache typically has the shortest latency. However, in certain cases like loads blocked on older stores, a load might suffer a high latency even though it is being satisfied by the L1. Note that this metric value may be highlighted due to DTLB Overhead or Cycles of 1 Port Utilized issues.

DTLB Overhead:	0.441

A significant proportion of cycles is being spent handling first-level data TLB misses. As with ordinary data caching, focus on improving data locality and reducing working-set size to reduce DTLB overhead. Additionally, consider using profile-guided optimization (PGO) to collocate frequently-used data on the same page. Try using larger page sizes for large amounts of frequently-used data.

Loads Blocked by Store Forwarding:	0.000
Lock Latency:	0.000
Split Loads:	0.000
4K Aliasing:	0.002
L2 Bound:	0.000
L3 Bound:	**0.000**
DRAM Bound:	**0.576**

This metric shows how often CPU was stalled on the main memory (DRAM). Caching typically improves the latency and increases performance.

Memory Bandwidth:	0.787

This metric represents a fraction of cycles during which an application could be stalled due to approaching bandwidth limits of the main memory (DRAM). This metric does not aggregate requests from other threads/cores/sockets (see Uncore counters for that). Consider improving data locality in NUMA multi-socket systems.

Memory Latency:	**0.190**

This metric represents a fraction of cycles during which an application could be stalled due to the latency of the main memory (DRAM). This metric does not aggregate requests from other threads/cores/sockets (see Uncore counters for that). Consider optimizing data layout or using Software Prefetches (through the compiler).

Local DRAM:	1.000

The number of CPU stalls on loads from the local memory exceeds the threshold. Consider caching data to improve the latency and increase the performance.

Remote DRAM:	0.000
Remote Cache:	0.000
Store Bound:	**0.134**

Elapsed Time: 35.740s

Clockticks:	126,124,189,186
Instructions Retired:	48,346,072,519
CPI Rate:	2.609

The CPI may be too high. This could be caused by issues such as memory stalls, instruction starvation, branch misprediction or long latency instructions. Explore the other hardware-related metrics to identify what is causing high CPI.

MUX Reliability:	0.996
Front-End Bound:	**0.7%**
Bad Speculation:	**0.0%**
Back-End Bound:	**87.3%**

Identify slots where no uOps are delivered due to a lack of required resources for accepting more uOps in the back-end of the pipeline. Back-end metrics describe a portion of the pipeline where the out-of-order scheduler dispatches ready uOps into their respective execution units, and, once completed, these uOps get retired according to program order. Stalls due to data-cache misses or stalls due to the overloaded divider unit are examples of back-end bound issues.

Memory Bound:	**66.2%**

The metric value is high. This can indicate that the significant fraction of execution p
Use Memory Access analysis to have the metric breakdown by memory hierarchy, memory bandwidth information, correlation by memory objects.

Our program is memory bound

L1 Bound:	**0.040**

This metric shows how often machine was stalled without missing the L1 data cache. The L1 cache typically has the shortest latency. However, in certain cases like loads blocked on older stores, a load might suffer a high latency even though it is being satisfied by the L1. Note that this metric value may be highlighted due to DTLB Overhead or Cycles of 1 Port Utilized issues.

DTLB Overhead:	0.441

A significant proportion of cycles is being spent handling first-level data TLB misses. As with ordinary data caching, focus on improving data locality and reducing working-set size to reduce DTLB overhead. Additionally, consider using profile-guided optimization (PGO) to collocate frequently-used data on the same page. Try using larger page sizes for large amounts of frequently-used data.

Loads Blocked by Store Forwarding:	0.000
Lock Latency:	0.000
Split Loads:	0.000
4K Aliasing:	0.002
L2 Bound:	0.000
L3 Bound:	**0.000**
DRAM Bound:	**0.576**

This metric shows how often CPU was stalled on the main memory (DRAM). Caching typically improves the latency and increases performance.

Memory Bandwidth:	0.787

This metric represents a fraction of cycles during which an application could be stalled due to approaching bandwidth limits of the main memory (DRAM). This metric does not aggregate requests from other threads/cores/sockets (see Uncore counters for that). Consider improving data locality in NUMA multi-socket systems.

Memory Latency:	**0.190**

This metric represents a fraction of cycles during which an application could be stalled due to the latency of the main memory (DRAM). This metric does not aggregate requests from other threads/cores/sockets (see Uncore counters for that). Consider optimizing data layout or using Software Prefetches (through the compiler).

Local DRAM:	1.000

The number of CPU stalls on loads from the local memory exceeds the threshold. Consider caching data to improve the latency and increase the performance.

Remote DRAM:	0.000
Remote Cache:	0.000
Store Bound:	**0.134**

Elapsed Time: 35.740s

Clockticks:	126,124,189,186
Instructions Retired:	48,346,072,519
CPI Rate:	2.609

The CPI may be too high. This could be caused by issues such as memory stalls, instruction starvation, branch misprediction or long latency instructions. Explore the other hardware-related metrics to identify what is causing high CPI.

MUX Reliability:	0.996
Front-End Bound:	**0.7%**
Bad Speculation:	**0.0%**
Back-End Bound:	**87.3%**

Identify slots where no uOps are delivered due to a lack of required resources for accepting more uOps in the back-end of the pipeline. Back-end metrics describe a portion of the pipeline where the out-of-order scheduler dispatches ready uOps into their respective execution units, and, once completed, these uOps get retired according to program order. Stalls due to data-cache misses or stalls due to the overloaded divider unit are examples of back-end bound issues.

Memory Bound:	**66.2%**

The metric value is high. This can indicate that the significant fraction of execution pipeline slots could be stalled due to demand memory load and stores. Use Memory Access analysis to have the metric breakdown by memory hierarchy, memory bandwidth information, correlation by memory objects.

L1 Bound:	**0.040**

This metric shows how often machine was stalled without missing the L1 data cache. The L1 cache typically has the shortest latency. However, in certain cases like loads blocked on older stores, a load might suffer a high latency even though it is being satisfied by the L1. Note that this metric value may be highlighted due to DTLB Overhead or Cycles of 1 Port Utilized issues.

DTLB Overhead:	0.441

A significant proportion of cycles is being spent handling first-level data TLB misses. As with ordinary data caching, focus on improving data locality and reducing working-set size to reduce DTLB overhead. Additionally, consider using profile-guided optimization (PGO) to collocate frequently-used data on the same page. Try using larger page sizes for large amounts of frequently-used data.

Loads Blocked by Store Forwarding:	0.000
Lock Latency:	0.000
Split Loads:	0.000
4K Aliasing:	0.002
L2 Bound:	0.000
L3 Bound:	**0.000**
DRAM Bound:	**0.576**

This metric shows how often CPU was stalled on the main memory (DRAM). Caching typically improves the latency and increases performance.

Memory Bandwidth:	0.787

This metric represents a fraction of cycles during which an application could be stalled due to approaching bandwidth limits of the main memory (DRAM). This metric does not aggregate requests from other threads/cores/sockets (see Uncore counters for that). Consider improving data locality in NUMA multi-socket systems.

Memory Latency:	**0.190**

This metric represents a fraction of cycles during which an application could be stalled due to the latency of the main memory (DRAM). This metric does not aggregate requests from other threads/cores/sockets (see Uncore counters for that). Consider optimizing data layout or using Software Prefetches (through the compiler).

Local DRAM:	1.000

The number of CPU stalls on loads from the local memory exceeds the threshold. Consider caching data to improve the latency and increase the performance.

Remote DRAM:	0.000
Remote Cache:	0.000
Store Bound:	**0.134**

Our program is memory bound

Elapsed Time: 35.740s

Clockticks:	126,124,189,186
Instructions Retired:	48,346,072,519
CPI Rate:	2.609

The CPI may be too high. This could be caused by issues such as memory stalls, instruction starvation, branch misprediction or long latency instructions. Explore the other hardware-related metrics to identify what is causing high CPI.

MUX Reliability:	0.996
Front-End Bound:	**0.7%**
Bad Speculation:	**0.0%**
Back-End Bound:	**87.3%**

Identify slots where no uOps are delivered due to a lack of required resources for accepting more uOps in the back-end of the pipeline. Back-end metrics describe a portion of the pipeline where the out-of-order scheduler dispatches ready uOps into their respective execution units, and, once completed, these uOps get retired according to program order. Stalls due to data-cache misses or stalls due to the overloaded divider unit are examples of back-end bound issues.

Memory Bound: 66.2%

The metric value is high. This can indicate that the significant fraction of execution pipeline slots could be stalled due to demand memory load and stores. Use Memory Access analysis to have the metric breakdown by memory hierarchy, memory bandwidth information, correlation by memory objects.

L1 Bound: 0.040

This metric shows how often machine was stalled without missing the L1 data cache. The L1 cache typically has the shortest latency. However, in certain cases like loads blocked on older stores, a load might suffer a high latency even though it is being satisfied by the L1. Note that this metric value may be highlighted due to DTLB Overhead or Cycles of 1 Port Utilized issues.

DTLB Overhead:	0.441

A significant proportion of cycles is being spent handling first-level data TLB misses. As with ordinary data caching, focus on improving data locality and reducing working-set size to reduce DTLB overhead. Additionally, consider using profile-guided optimization (PGO) to collocate frequently-used data on the same page. Try using larger page sizes for large amounts of frequently-used data.

Loads Blocked by Store Forwarding:	0.000
Lock Latency:	0.000
Split Loads:	0.000
4K Aliasing:	0.002
L2 Bound:	0.000
L3 Bound:	**0.000**
DRAM Bound:	**0.576**

This metric shows how often CPU was stalled on the main memory (DRAM). Caching typically improves the latency and increases performance.

Memory Bandwidth:	0.787

This metric represents a fraction of cycles during which an application cou
This metric does not aggregate requests from other threads/cores/sockets (
socket systems.

Bandwidth saturates fast because we move an entire cache line for almost every data item

Memory Latency: 0.190

This metric represents a fraction of cycles during which an application could be stalled due to the latency of the main memory (DRAM). This metric does not aggregate requests from other threads/cores/sockets (see Uncore counters for that). Consider optimizing data layout or using Software Prefetches (through the compiler).

Local DRAM:	1.000

The number of CPU stalls on loads from the local memory exceeds the threshold. Consider caching data to improve the latency and increase the performance.

Remote DRAM:	0.000
Remote Cache:	0.000
Store Bound:	**0.134**

Store Bound:	**0.134**
Core Bound:	**21.1%**

This metric represents how much Core non-memory issues were of a bottleneck. Shortage in hardware compute resources, or dependencies software's instructions are both categorized under Core Bound. Hence it may indicate the machine ran out of an OOO resources, certain execution units are overloaded or dependencies in program's data- or instruction- flow are limiting the performance (e.g. FP-chained long-latency arithmetic operations).

Divider:	0.099
Port Utilization:	**0.373**

This metric represents a fraction of cycles during which an application was stalled due to Core non-divider-related issues. For example, heavy data-dependency between nearby instructions, or a sequence of instructions that overloads specific ports. Hint: Loop Vectorization - most compilers feature auto-Vectorization options today - reduces pressure on the execution ports as multiple elements are calculated with same uop.

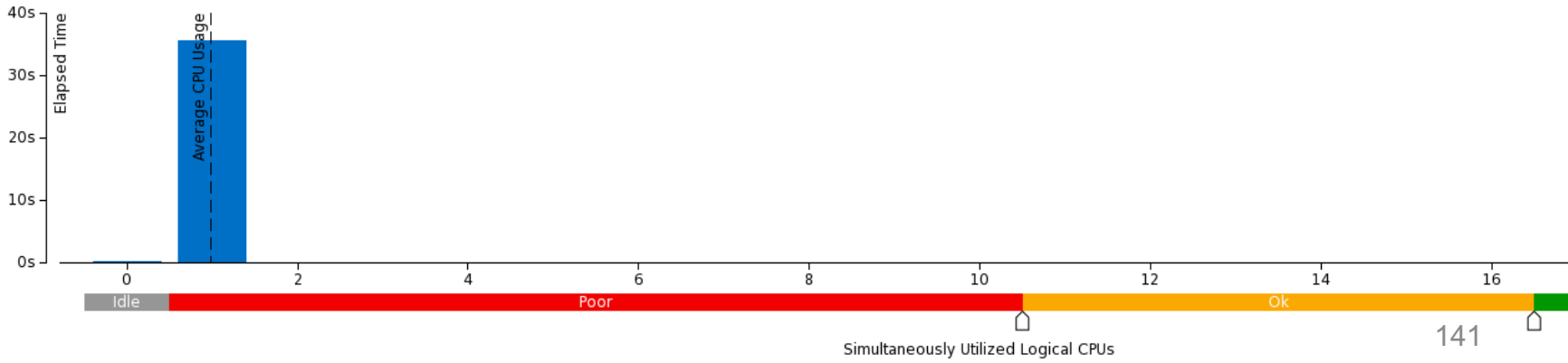
Cycles of 0 Ports Utilized:	0.627

This metric represents cycles fraction CPU executed no uops on any execution port.

Cycles of 1 Port Utilized:	0.132
Cycles of 2 Ports Utilized:	0.089
Cycles of 3+ Ports Utilized:	**0.068**
Retiring:	**12.0%**
Total Thread Count:	2
Paused Time:	0s

CPU Usage Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU usage value.



Store Bound:	**0.134**
Core Bound:	**21.1%**

This metric represents how much Core non-memory issues were of a bottleneck. Shortage in hardware compute resources, or dependencies software's instructions are both categorized under Core Bound. Hence it may indicate the machine ran out of an OOO resources, certain execution units are overloaded or dependencies in program's data- or instruction- flow are limiting the performance (e.g. FP-chained long-latency arithmetic operations).

Divider:	0.099
Port Utilization:	**0.373**

This metric represents a fraction of cycles during which an application was stalled due to Core non-divider-related issues. For example, heavy data-dependency between nearby instructions, or a sequence of instructions that overloads specific ports. Hint: Loop Vectorization - most compilers feature auto-Vectorization options today - reduces pressure on the execution ports as multiple elements are calculated with same uop.

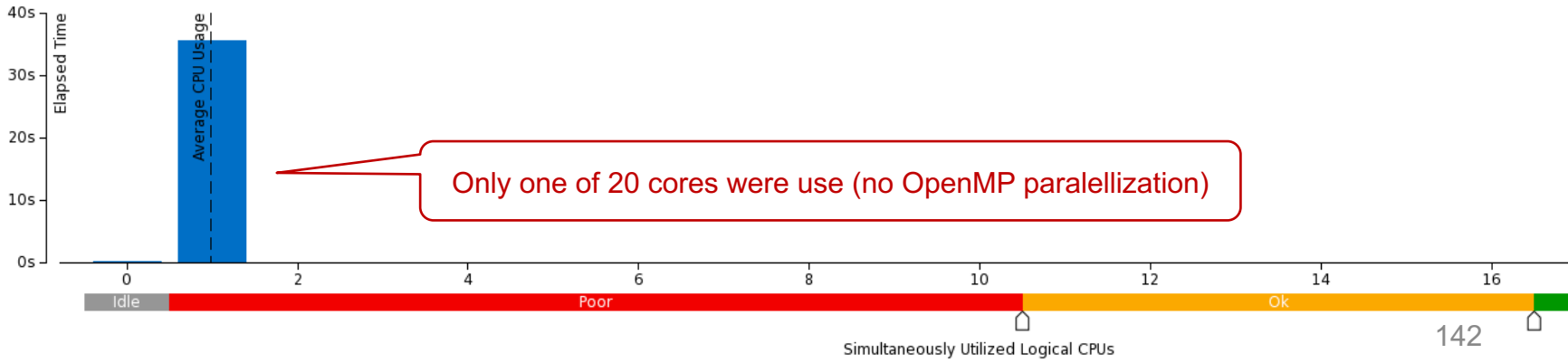
Cycles of 0 Ports Utilized:	0.627

This metric represents cycles fraction CPU executed no uops on any execution port.

Cycles of 1 Port Utilized:	0.132
Cycles of 2 Ports Utilized:	0.089
Cycles of 3+ Ports Utilized:	**0.068**
Retiring:	**12.0%**
Total Thread Count:	2
Paused Time:	0s

CPU Usage Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU usage value.



Store Bound:	**0.134**
Core Bound:	**21.1%**

This metric represents how much Core non-memory issues were of a bottleneck. Shortage in hardware compute resources, or dependencies software's instructions are both categorized under Core Bound. Hence it may indicate the machine ran out of an OOO resources, certain execution units are overloaded or dependencies in program's data- or instruction- flow are limiting the performance (e.g. FP-chained long-latency arithmetic operations).

Divider:	0.099
Port Utilization:	**0.373**

This metric represents a fraction of cycles during which an application was stalled due to Core non-divider-related issues. For example, heavy data-dependency between nearby instructions, or a sequence of instructions that overloads specific ports. Hint: Loop Vectorization - most compilers feature auto-Vectorization options today - reduces pressure on the execution ports as multiple elements are calculated with same uop.

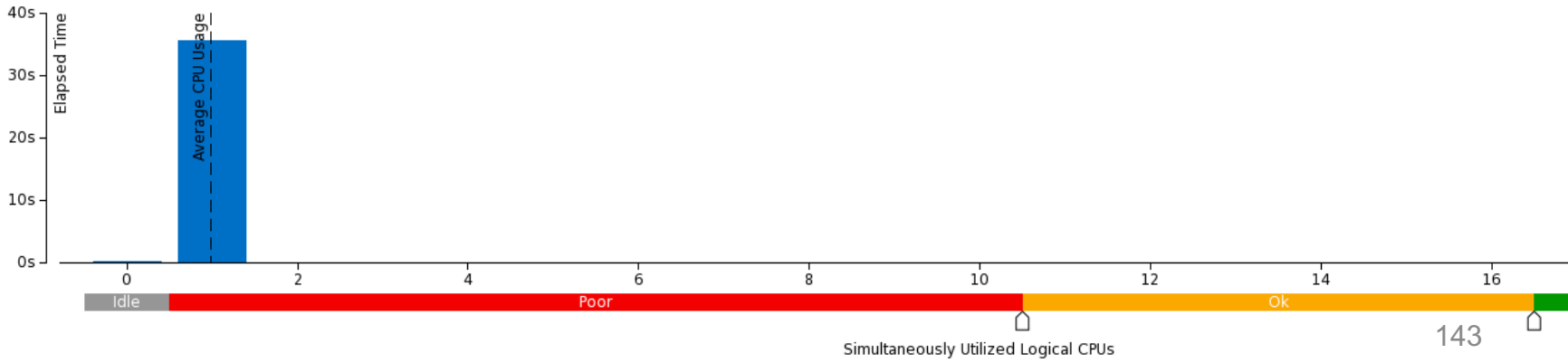
Cycles of 0 Ports Utilized:	0.627

This metric represents cycles fraction CPU executed no uops on any execution port.

Cycles of 1 Port Utilized:	0.132
Cycles of 2 Ports Utilized:	0.089
Cycles of 3+ Ports Utilized:	**0.068**
Retiring:	**12.0%**
Total Thread Count:	2
Paused Time:	0s

CPU Usage Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU usage value.



Advisor vs Vtune amplifier

- Advisor is profiler
  - Analyzes your code on a per statement basis
  - Looks at the assembly code to analyze vectorization
  - Hints to the location of the problem
- Vtune Amplifier accumulates statistics on hardware events such as expensive instructions, vector instructions, cache misses, …
  - Statistics accumulated on a per subprogram (function, subroutine) basis, not per statement
  - Hints to the nature of the problem
- Both are complementary

What have we learned so far

- Compiler does good job at producing vectorized code
- Advisor will tell you if and why the compiler is sometimes not able to produce vectorized code, and will suggest solutions
- Advisor tells you which parts of your code consume the most cputime and are candidates for optimization

What have we learned so far

- Most often performance problems on modern cpus are due to memory access problems (DRAM latency hits you)
- VTune amplifier gives you clues on how and where to fixes the issues
  - CPI and Cache Misses
- Optimize
  1. If there are cache misses, try to reduce them
     - Easier said than done (we'll come to that in the next section)
  2. If you are memory bound and CPI is high,
     1. Verify vectorization
     2. Increase the computational complexity (do more useful work on the data while it is in cache)

Molecular dynamics settings

- Suppose we have 10^9 atoms
- Computing all interactions in single precision
  - $10^9(10^9\text{-}1)/2 \sim 0.5\ 10^{18}$
  - complexity $O(N^2)$– not a good idea
    - Adding 1 atom increases the work by a factor $N = 10^9$
    - Adding 2 atoms increases the work by a factor $N \cdot N = 10^{18}$
    - …
  - Lennard-Jones is short range
    - $\lim_{r\to\infty} 4\pi r^2\, V_{LJ}(r) \to 0$
    - In practice cut-off $r_c \cong 2.5$

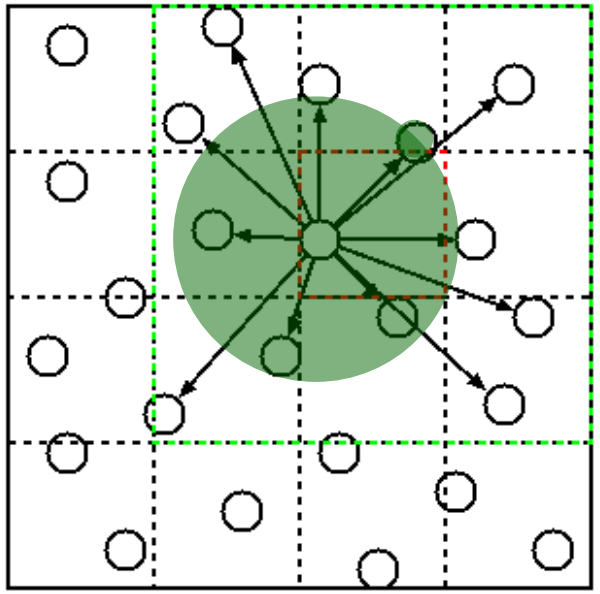
Implementing cut-off

```
forces = 0
do i=1,N
   do j=1,N-1
      r2 = squared_distance(i,j)
      if r2<rcutoff2
         force_ij = ljforce(r2)
         force(i) = force(i) + force_ij
         force(j) = force(j) - force_ij
      endif
   enddo
enddo
integrate forces to update atom positions
```

- Still $O(N^2)$ ☹
- Might be ok for small N

Implementing cut-off

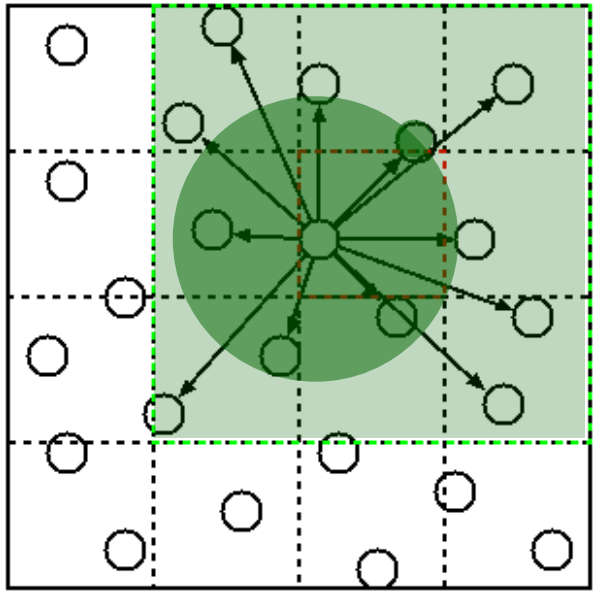
- Verlet lists
  - Verlet list of atom i is list of all atoms j for which $j < i$ and $r_{ij} < r_c$
  - Increase cutoff slightly so that we do not have to update the Verlet lists at every timestep (depending on how vigorously the atoms move)
  - Verlet list construction is amortized



- Construction of Verlet lists is still $O(N^2)$☹
- Is dominant data structure: typically between 50 and 100 neighbour atoms/atom

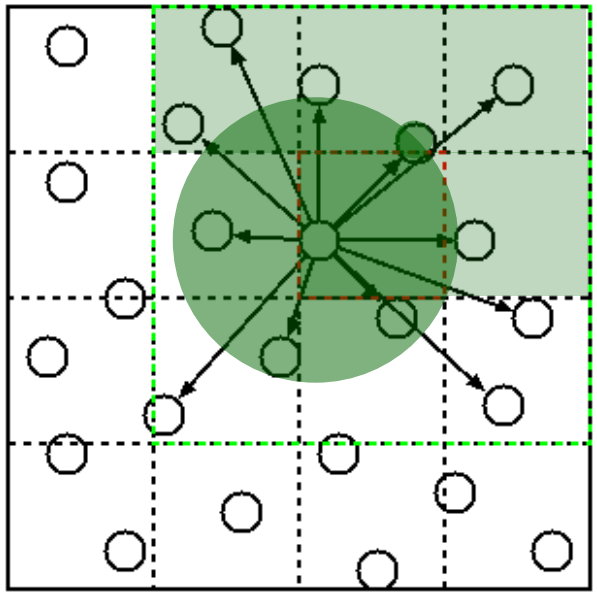
Implementing cut-off

- Put atoms in cells of width r_c : $O(N)$☺
- Only atoms in neigbouring cells can satisfy $r_{ij} < r_c$
- Because of symmetry only half of the neighbouring cells must be examined
- Construct Verlet lists as follows
  - Loop over all cells $[O(N)]$
    - Loop over all neighbours of the current cell using the neighbour stencil  $[O(1)]$
      - Construct the Verlet list of all atoms in the current cell $[O(1)]$
- Now our MD algorithm is $O(N)$☺



Implementing cut-off

- Put atoms in cells of width r_c : $O(N)$☺
- Only atoms in neigbouring cells can satisfy $r_{ij} < r_c$
- Because of symmetry only half of the neighbouring cells must be examined
- Construct Verlet lists as follows
  - Loop over all cells $[O(N)]$
    - Loop over all neighbours of the current cell using the neighbour stencil  $[O(1)]$
      - Construct the Verlet list of all atoms in the current cell $[O(1)]$
- Now our MD algorithm is $O(N)$☺



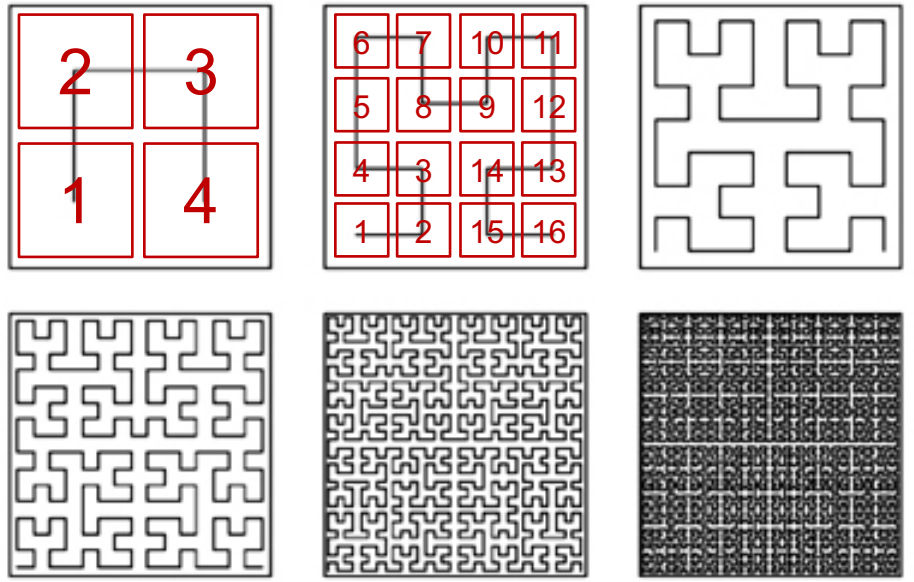
Implementing cut-off

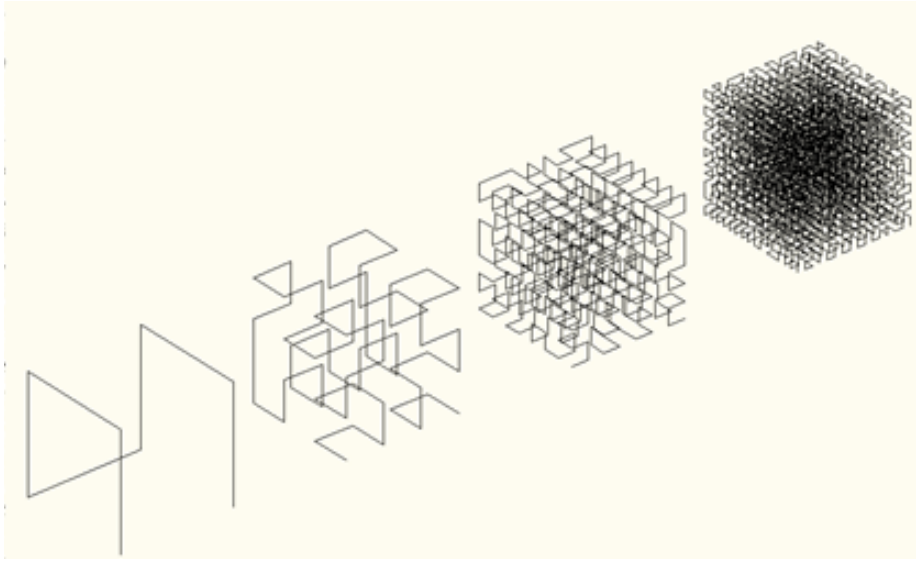
- Atoms move!
- Iterating over the Verlet lists to compute the interactions will soon jump randomly through memory
- Performance evolves naturally to the random access case

- Fix data access pattern using spatial sorting
  - Spatial sort = ensure that atoms which are close in space are also close in memory
  - This reduces cache misses

Fixing the data access pattern

- **Space filling curve**
- Linearize a space of dimension >1
- Hilbert curve
- Hilbert index: coordinate of a cell along the Hilbert curve
- Locality guarantee: points close in space are also close along the space filling curve (on average)





Fixing the data access pattern

1. Sort atom property arrays (rx,ry,rz,vx,vy,vz,…) based on the Hilbert index h of the cell of the atoms (**spatial sort**).
   Atoms which are close in space (and hence will interact) will be close in memory (and hence will be in the cache with high probability)
2. Build a table containing the index of the first atom in each cell, and the number of atoms in the cell (Hilbert list)
3. Build Verlet list from the Hilbert list (discard the latter)
4. Compute the interactions by looping over the Verlet list
5. Integrate forces, updating velocities and positions and time
6. If *need_to_rebuild_verlet_list* is true
   jump back to step 1.
   else
   continue at step 4.

Fixing the data access pattern

1. Sort atom property arrays (rx,ry,rz,vx,vy,vz,…) based on the Hilbert index h of the cell of the atoms (**spatial sort**).
   Atoms which are close in space (and hence will interact) will be close in memory (and hence will be in the cache with high probability)
2. Build a table containing the index of the first atom in each cell, and the number of atoms in the cell (Hilbert list)
3. Build Verlet list from the Hilbert list (discard the latter)
4. Compute the interactions by looping over the Verlet list
5. Integrate forces, updating velocities and positions and time
6. If *need_to_rebuild_verlet_list* is true
   jump back to step 1.
   else
   continue at step 4.

Fixing the data access pattern

- We need to
  - Compute Hilbert indices
  - Sort atom property arrays
  - Build Hilbert list and Verlet list
- Fixing data access patterns can be a lot of work

Choosing a programming language

- **Implementation in Fortran? C? C++?**
- Arguments
  - C++ is inefficient
    - Modern compilers good enough to generate efficient code
    - After all your are using the same hardware

Choosing a programming language

- **Implementation in Fortran? C? C++?**
- Arguments
  - ~~C++ is inefficient~~ Lie #1
    - Modern compilers good enough to generate efficient code
    - After all your are using the same hardware

Choosing a programming language

- **Implementation in Fortran? C? C++?**
- Arguments
  - C++ is inefficient
    - Modern compilers good enough to generate efficient code
    - After all your are using the same hardware
  - Fortran is efficient

Lie #1

Choosing a programming language

- **Implementation in Fortran? C? C++?**
- Arguments
  - ~~C++ is inefficient~~ Lie #1
    - Modern compilers good enough to generate efficient code
    - After all your are using the same hardware
  - ~~Fortran is efficient~~ Lie #2
    - Also fortran has constructs that sometimes come in handy, but can kill performance
  - But C++ has quite a bit more features which can kill performance than Fortran
    - Because C++ is a general purpose language and Fortran is meant for scientific computing
  - Yet these features can be extremely useful if you use them wisely
    - For computational kernels where performance is an issue you generally need to stay close to the C subset and far away from the C++ features such as classes, inheritance, virtual functions, etc. (templates are an exception)

Choosing a programming language

- **Implementation in Fortran? C? C++?**
- Arguments
  - ~~C++ is inefficient~~ Lie #1
    - Modern compilers good enough to generate efficient code
    - After all your are using the same hardware
  - ~~Fortran is efficient~~ Lie #2
    - Also fortran has constructs that sometimes come in handy, but can kill performance
  - But C++ has quite a bit more features which can kill performance than Fortran
    - Because C++ is a general purpose language and Fortran is meant for scientific computing
  - Yet these features can be extremely useful if you use them wisely
    - For computational kernels where performance is an issue you generally need to stay close to the C subset and far away from the C++ features such as classes, inheritance, virtual functions, etc. (templates are an exception)

Choosing a programming language

- I'll use C++ because I know it better

Choosing a programming language

- ~~I'll use C++ because I know it better~~

Most often a lie too!

Choosing a programming language

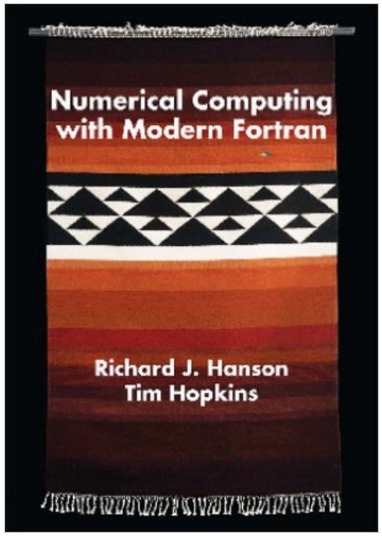
- ~~I'll use C++ because I know it better~~ Most often a lie too!
  - Unless you have read and understood all the C++ books by Scott Meyers, Herb Sutter, Andrei Alexandrescu, Nicolai Josuttis
  - In which case you probably also understand which C++ features can kill performance and when they should be used to your advantage
  - For number-crunching I find myself advancing faster using Fortran than using C++ (which I do know better!)

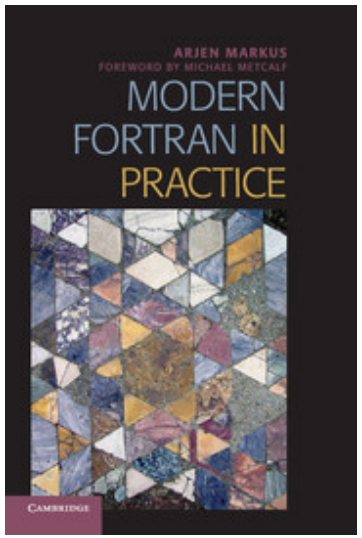
Choosing a programming language

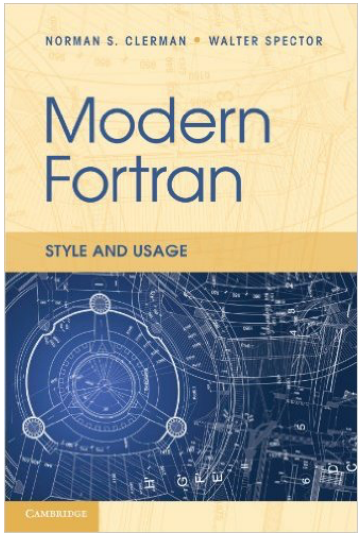
- I'll use C++ because it is better documented

Not a lie

  - There aren't too many books on Fortran like the above ones on C++
  - There is no website of the same quality as cplusplus.com or cppreference.com for Fortran (imho)
  - But still it is much harder to learn and to learn to use efficiently
  - Not a valid argument







Choosing a programming language

- I'll use C because that is the language in which Python was written and I want too integrate my code with Python Good point!
  - Python integration leverages your code with
    - A high level programming interface:
      - Providing initial data for your simulation is much easier and flexible through a Python script than having to parse input files…
      - Compose and customize high level solution schemes with ease (e.g. choosing another solver for a subproblem)
    - Hundreds of very useful open source Python libraries: Numpy, Scipy, Pandas, matplotlib, …
  - But …

Choosing a programming language

- I'll use C because that is the language in which Python was written and I want too integrate my code with Python [continued]
  - …
  - But the easiest way to create your own module that can be imported in Python is through **f2py**
    - Automatically turns your Fortran code into a Python module
    - As simple as
      - `F2py –c mysource.f90 –m myPythonModule`
    - Automatically integrates with **Numpy**! Pass Numpy arrays to your own Fortran library with no effort and no copying of data!
    - Much harder in C
    - Also feasible in C++ with the help of boost.python and boost.multi_array, easier than C but not as easy as f2py

∴ stick to Fortran

Choosing a programming language

- I'll use C/C++ because I don't want to mess with storage orders

Good point!

  - Fortran uses row-major ordering indices start at 1
  - C/C++ use column-major ordering indices start at 0
  - Plenty of ways to mess up!
  - Inadvertent copying of the array when passing to fortran!
  - Trivial for 1D arrays (but mind the indexing)
  - Numpy arrays by default use the C convention, but arrays can be easily made to follow the Fortran convention:
    - `A=numpy.empty( (2,2), dtype=np.float32, order='F')`
  - A little bit of experimentation will take away the confusion
  - Simplest way to avoid problems:
    - If you use fortran modules adhere to fortran convention in numpy arrays
    - If you use C/C++ modules adhere to C/C++ convention in numpy arrays
    - If you use both pay attention..

Choosing a programming language

- I'll use X because I need to use some library Y that is written in X
  - Point taken, often the easiest way
  - Valid argument for libraries that deal with parallelization issues:
    - TBB, for shared memory parallelization (C++)
    - Libraries abstracting vector instructions Vc, boost.simd (C++)
  - However, there is a lot of support for mixed language programming
  - If you do not master X you might end up writing inefficient code and loosing the advantage of using Y
  - it may be worthwhile to find out how to tackle the mixed language challenge
  - Once done, you will proceed faster and write efficient code

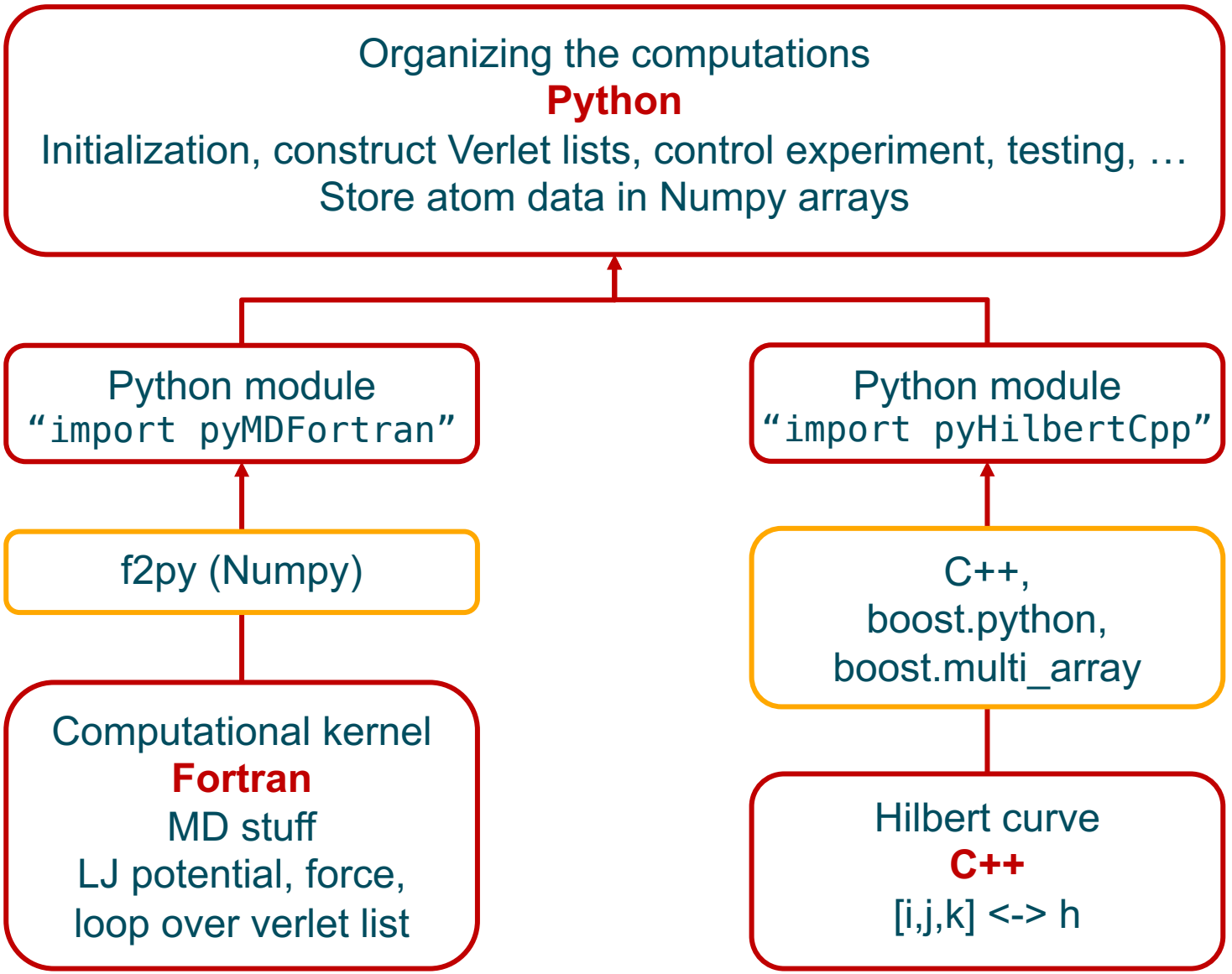
To conclude

- Stick to Fortran (unless you are a seasoned C++ programmer)
- Using Python and Numpy for high level programming and Fortran (C++) for your own number-crunching routines is a very practical approach
  - Use f2py to turn your fortran routines into a Python module that is compatible with Numpy

Choosing a programming language

- Details of Python+Numpy+Fortran/C++ is topic of another talk
- Including shared memory parallelism (multi-threading) and distributed memory parallelism (multi-node)

Back to Lennard-Jones MD



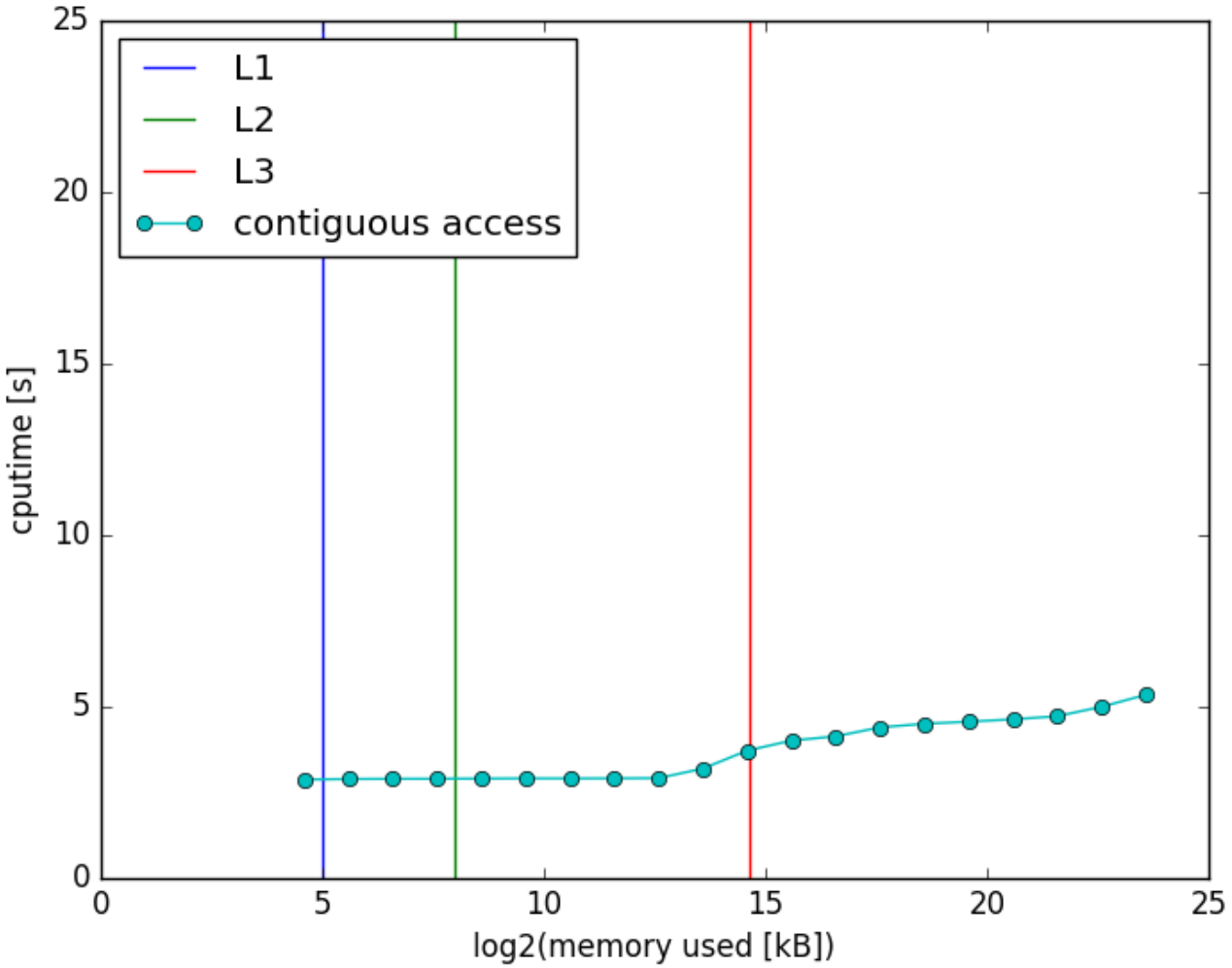
MD interaction forces

- Take derivative of (Lennard-Jones) potential with respect to interatomic distance vector = force exerted on the atoms
- $\frac{dV(r)}{d\vec{r}} = \frac{dV_2(r^2)}{dr^2}\frac{dr^2}{d\vec{r}} = \frac{dV_2(r^2)}{dr^2}\,2\,\vec{r} = f(r^2)\vec{r}$
- Loop over i
  - Loop over $j \in VL_i$
    - $\vec{a}_i \mathrel{+}= f\left(r_{ij}^2\right)\vec{r}_{ij}$
    - $\vec{a}_j \mathrel{-}= f\left(r_{ij}^2\right)\vec{r}_{ij}$
- 3 x load ($\vec{r}_j$)
- 3 x load ($\vec{a}_j$)
- 3 x store ($\vec{a}_j$)

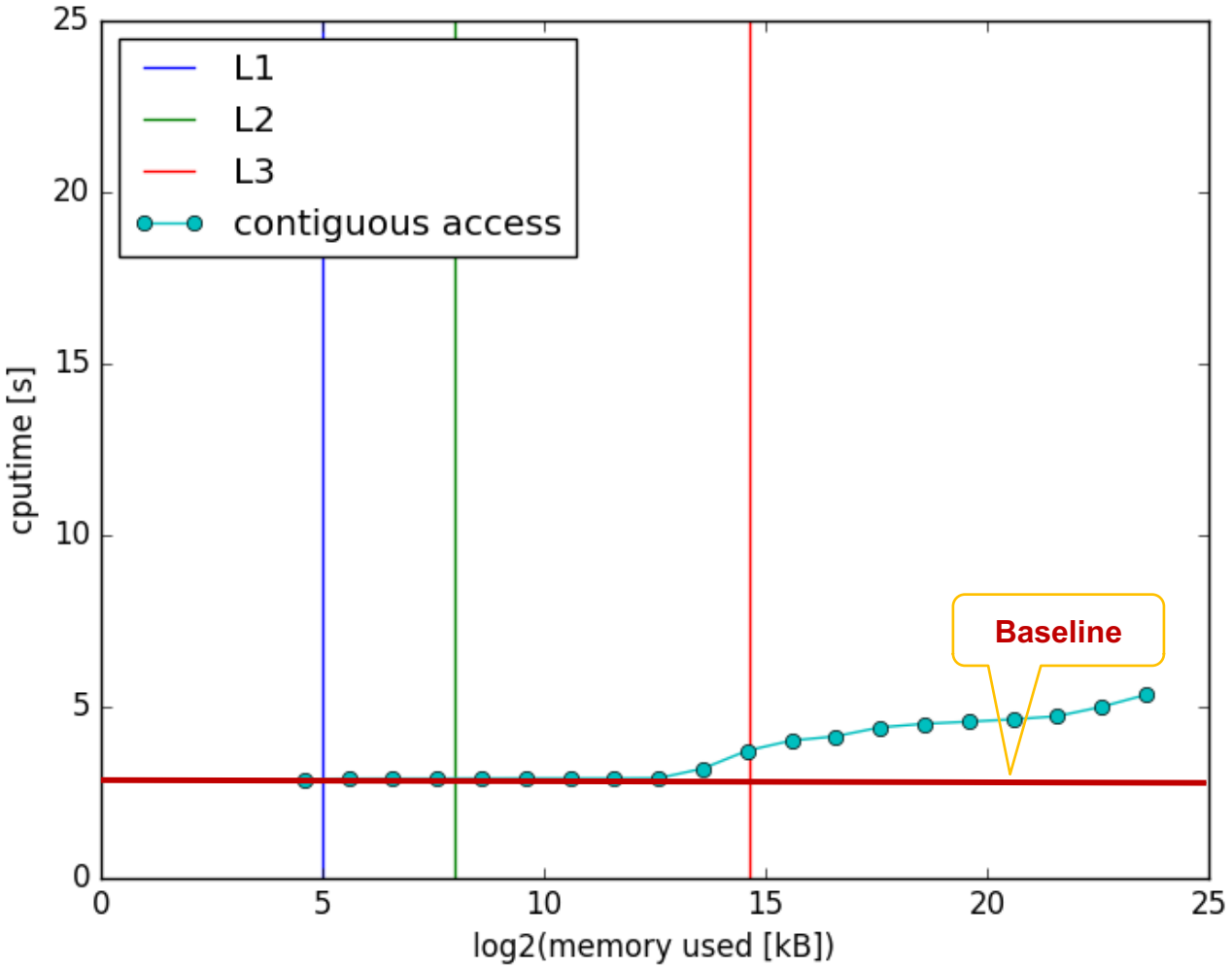
Results

- Baseline case:
  - N atoms
  - Compute interaction forces of atom 0 with all other atoms
  - Contiguous memory access
  - Bandwidth saturated

Baseline

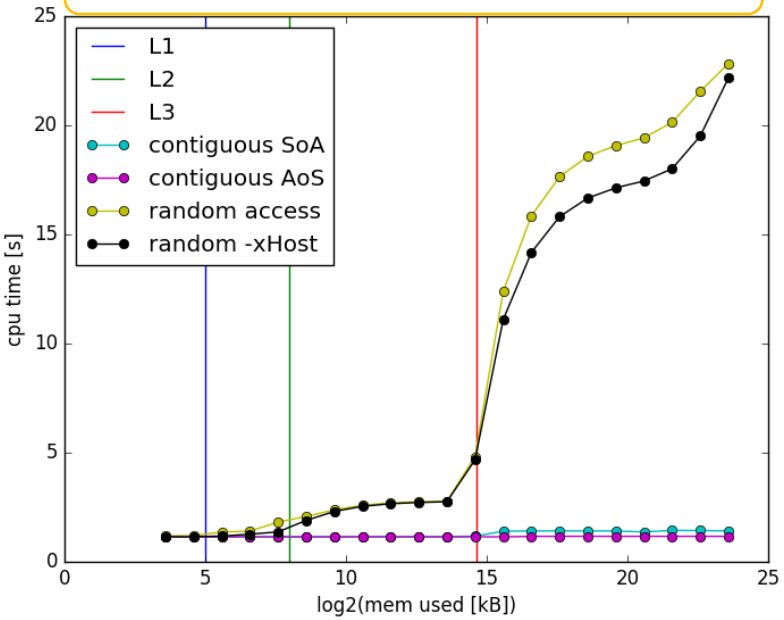

L1
L2
L3
contiguous access
cputime [s]
log2(memory used [kB])
0
5
10
15
20
25

Baseline

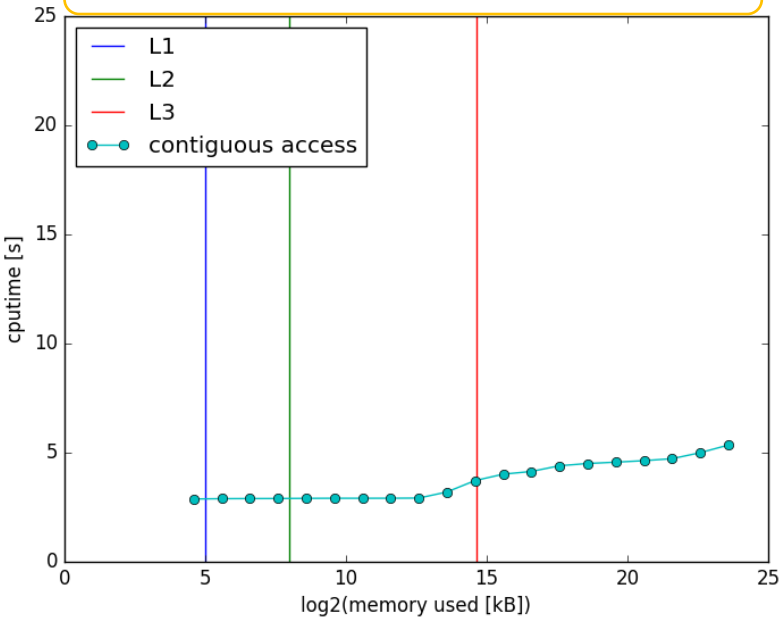

L1
L2
L3
contiguous access
Baseline
cputime [s]
log2(memory used [kB])
0
5
10
15
20
25

Results

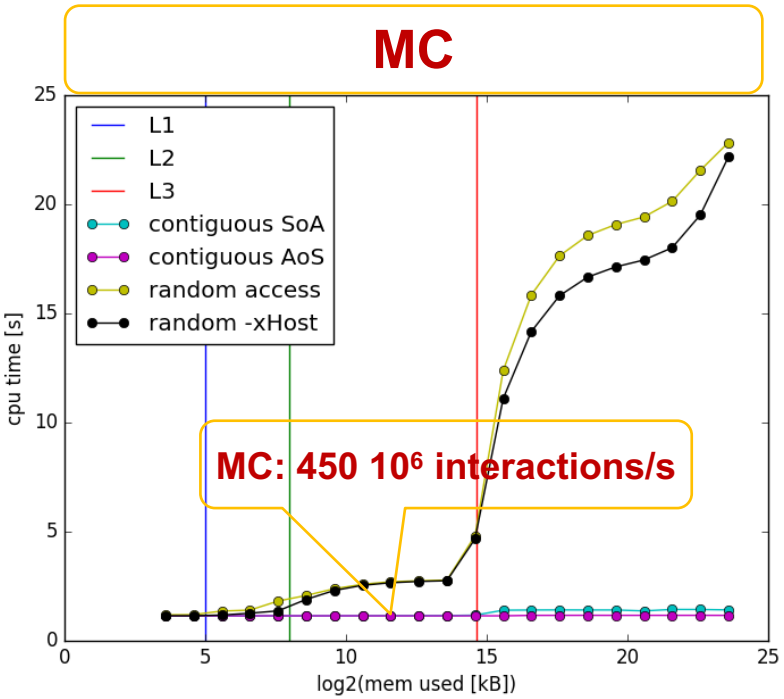
MC

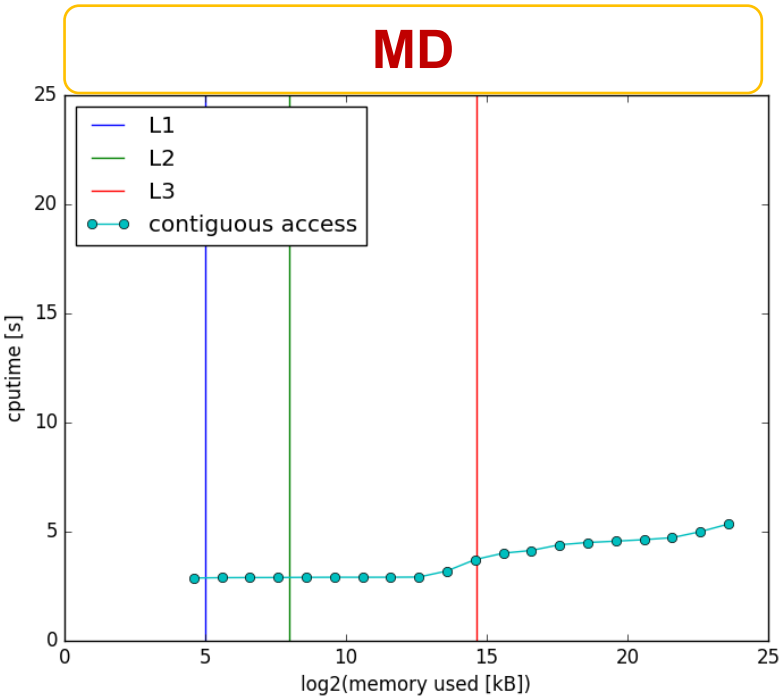


MD

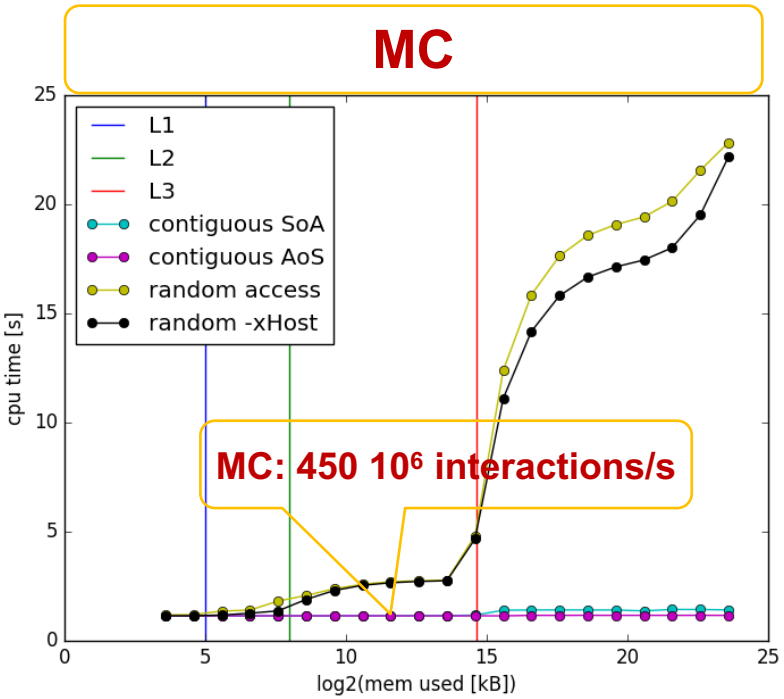


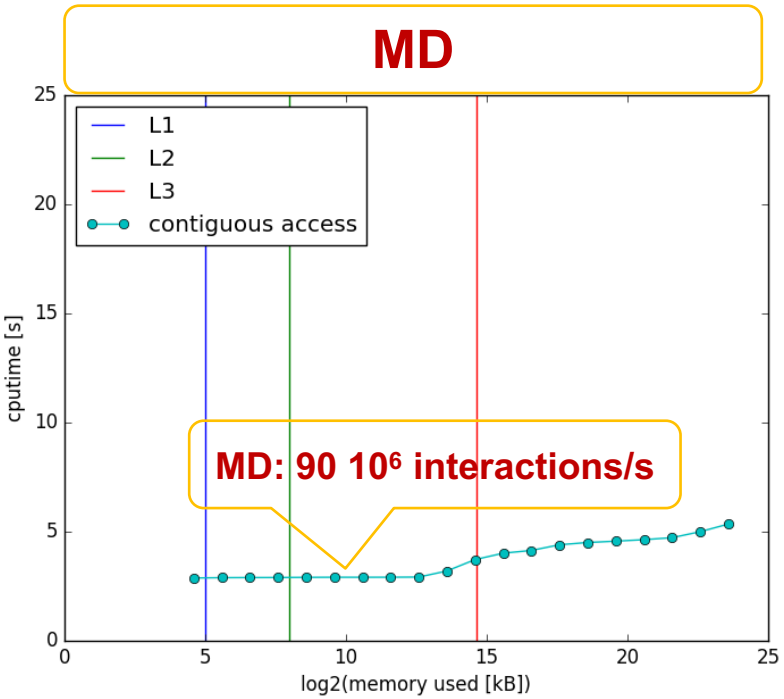
Results


MC
L1
L2
L3
contiguous SoA
contiguous AoS
random access
random -xHost
MC: 450 10⁶ interactions/s
cpu time [s]
log2(mem used [kB])
0
5
10
15
20
25

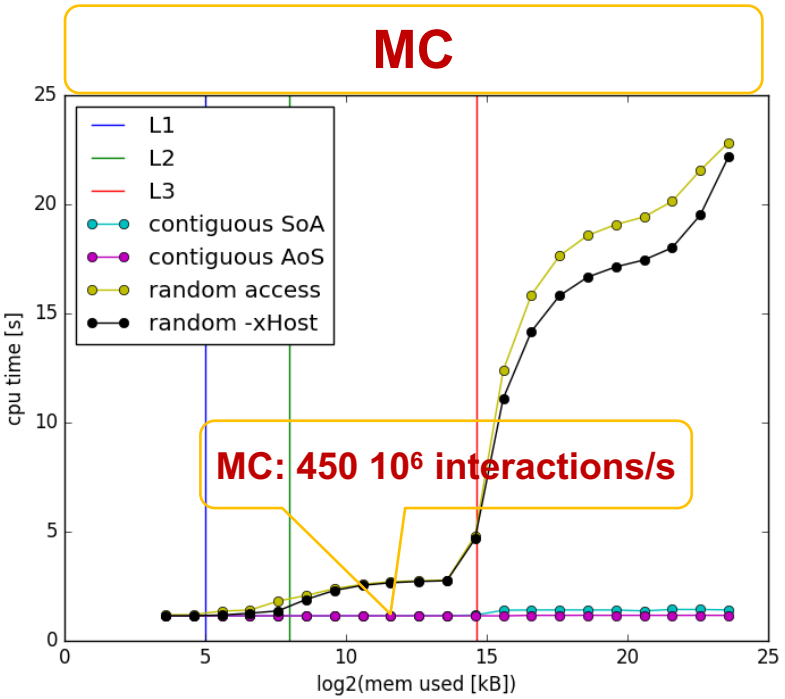

MD
L1
L2
L3
contiguous access
cputime [s]
log2(memory used [kB])
0
5
10
15
20
25

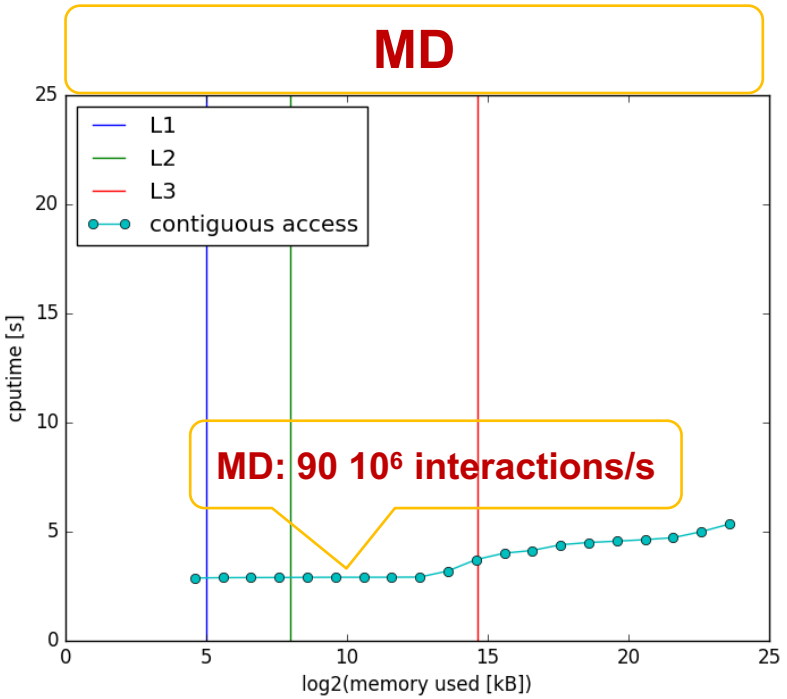
Results


MC
L1
L2
L3
contiguous SoA
contiguous AoS
random access
random -xHost
MC: 450 10⁶ interactions/s
cpu time [s]
log2(mem used [kB])
0
5
10
15
20
25


MD
L1
L2
L3
contiguous access
MD: 90 10⁶ interactions/s
cputime [s]
log2(memory used [kB])
0
5
10
15
20
25

Results





In terms of interactions/s MD is about 5 times slower than MC

- A bit more instructions per interaction, but MC is memory bound, that should not matter
- 3 times more memory access
- With a read:write ratio of 2:1 the bandwidth drops from 11 GB/s to 9.5 GB/s
- 3x11/9.5 = 3.47
- Still factor 1.44 slower than expected

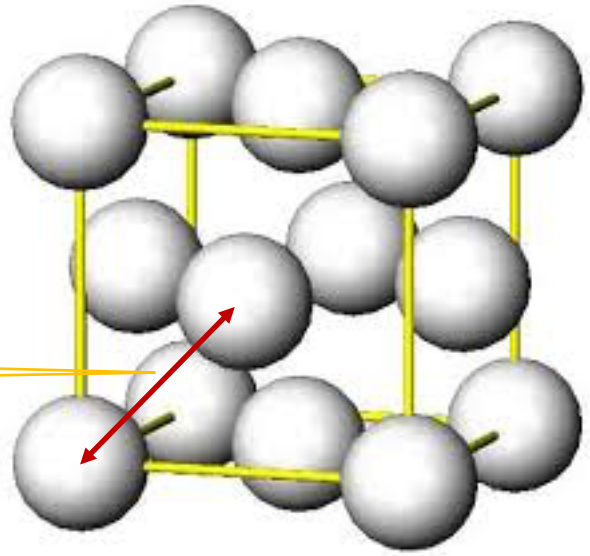
Experiments

- Three cases:
  1. Atoms on FCC lattice
  2. Permute the atoms (=random memory access)
  3. Spatial sort by hilbert index

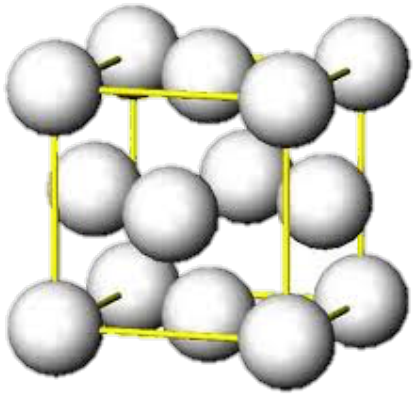
- Every experiment build the Verlet list and computes the interactions
- CPUtime is measured only for computing the interactions
- Plot result relative to baseline
  - 90 10^6 interactions/s

Case 1

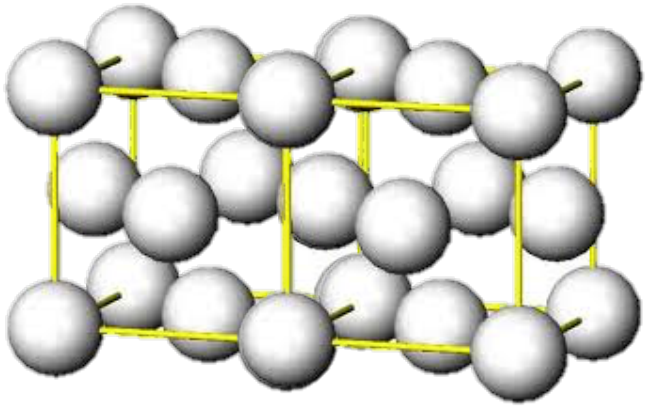
- Put atoms on FCC lattice
- 4 atoms per unit cell
- Closest neighbor distance = LJ r_{min}
- $r_{cutoff} = 3r_{min}$



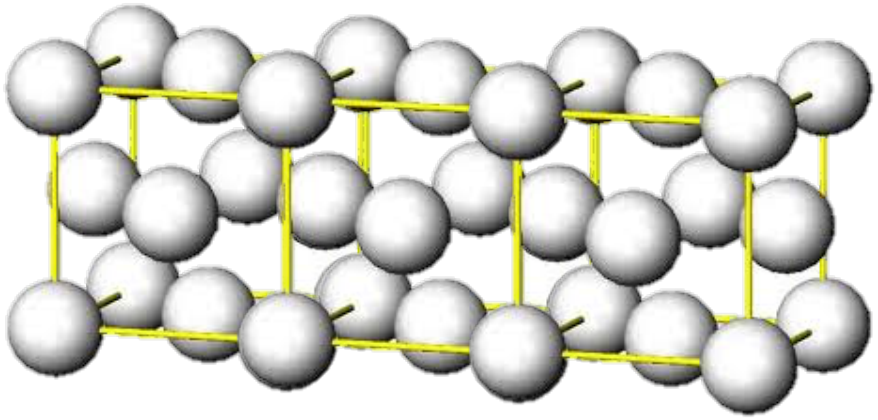
Case 1



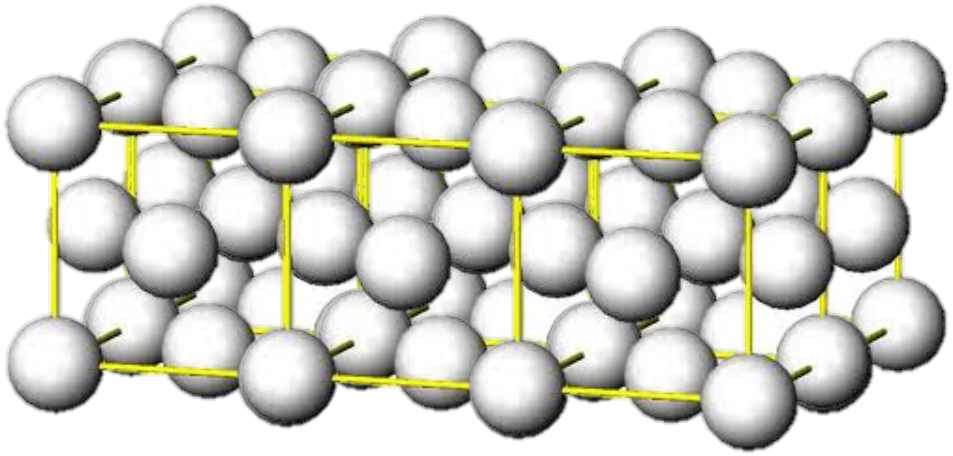
Case 1



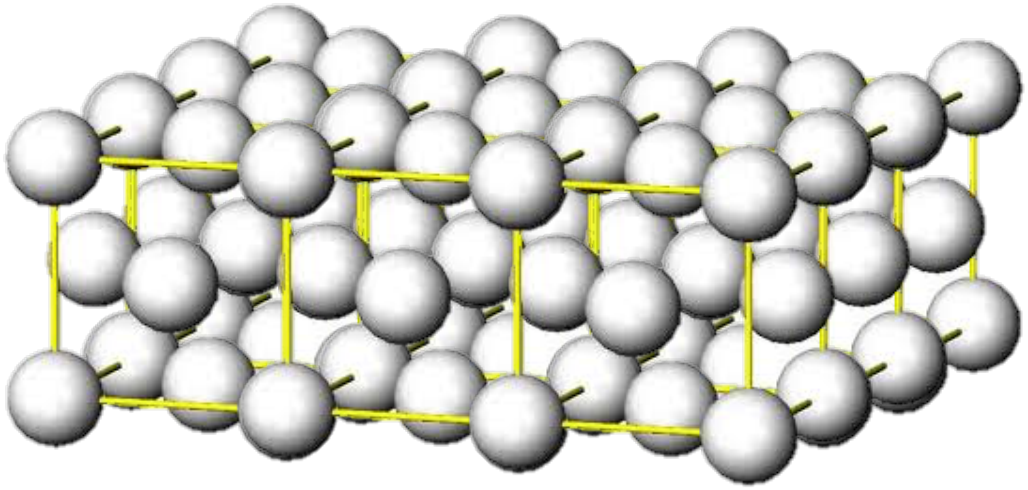
Case 1



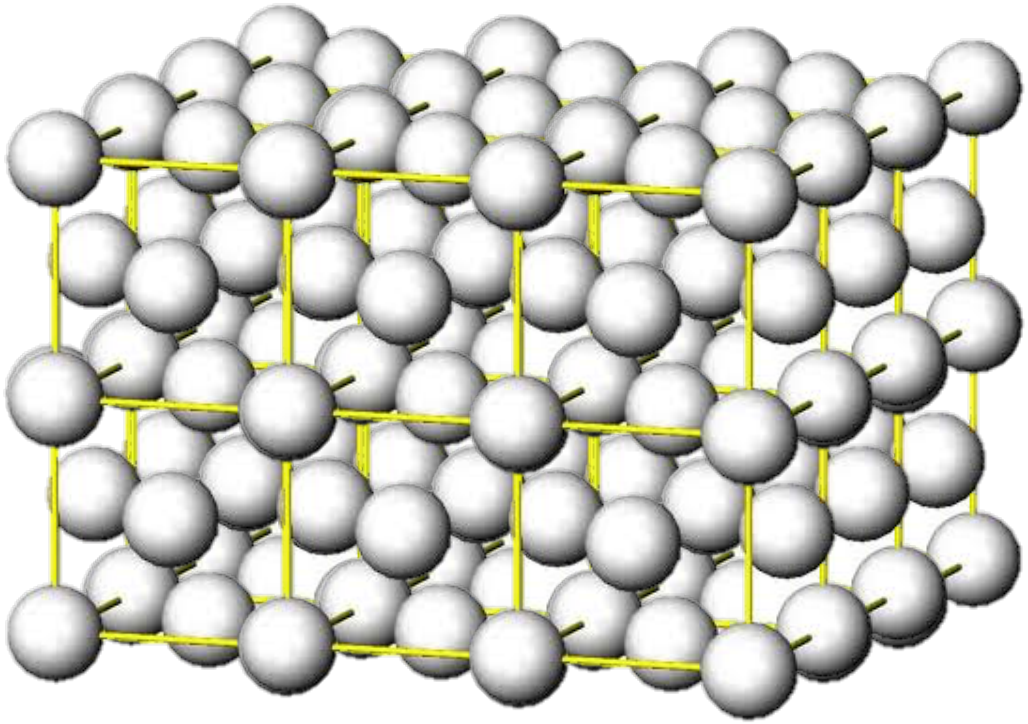
Case 1



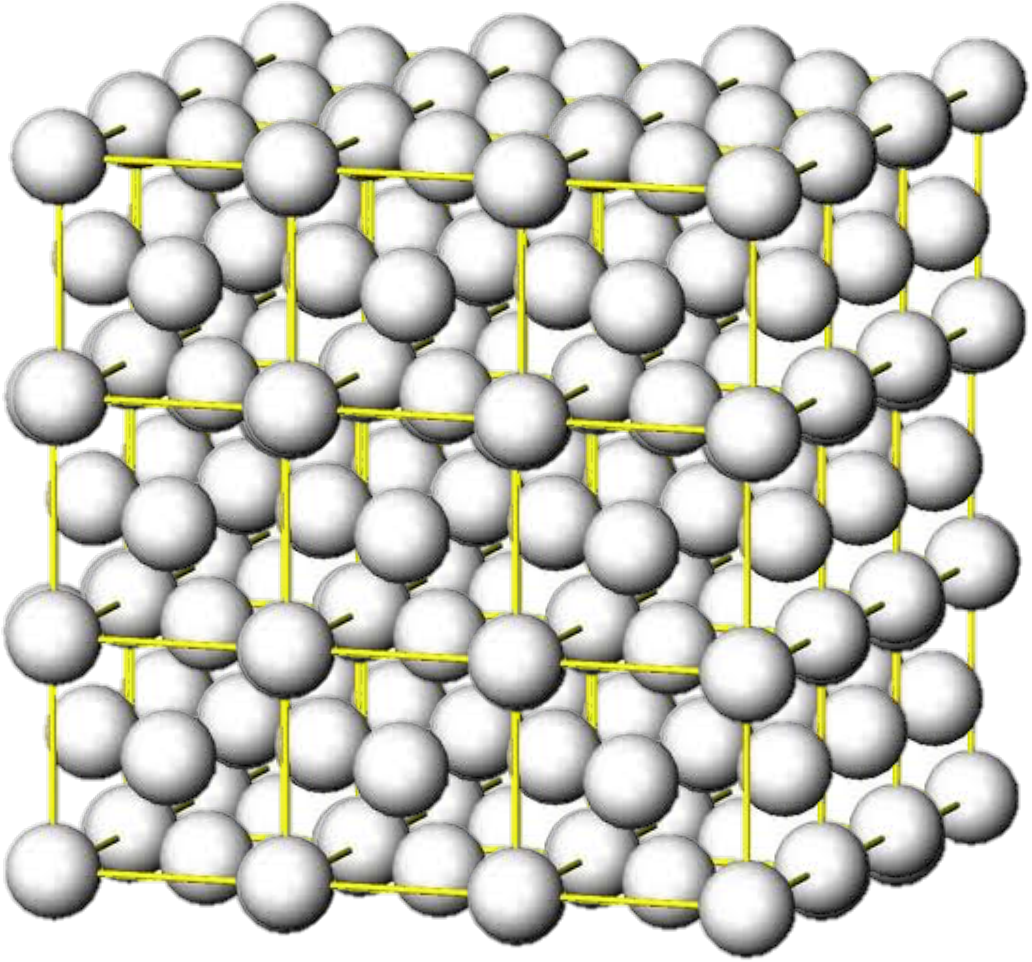
Case 1



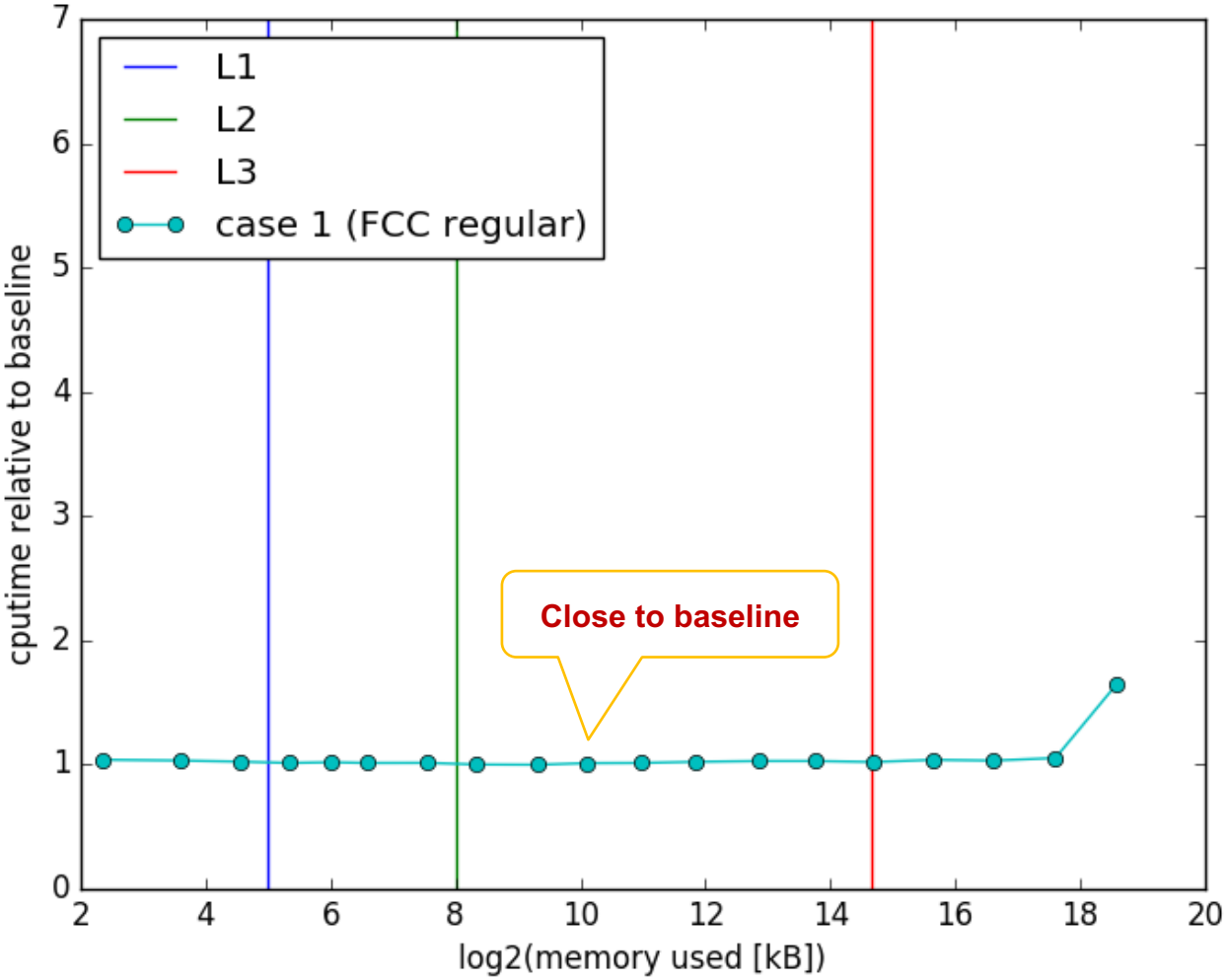
Case 1



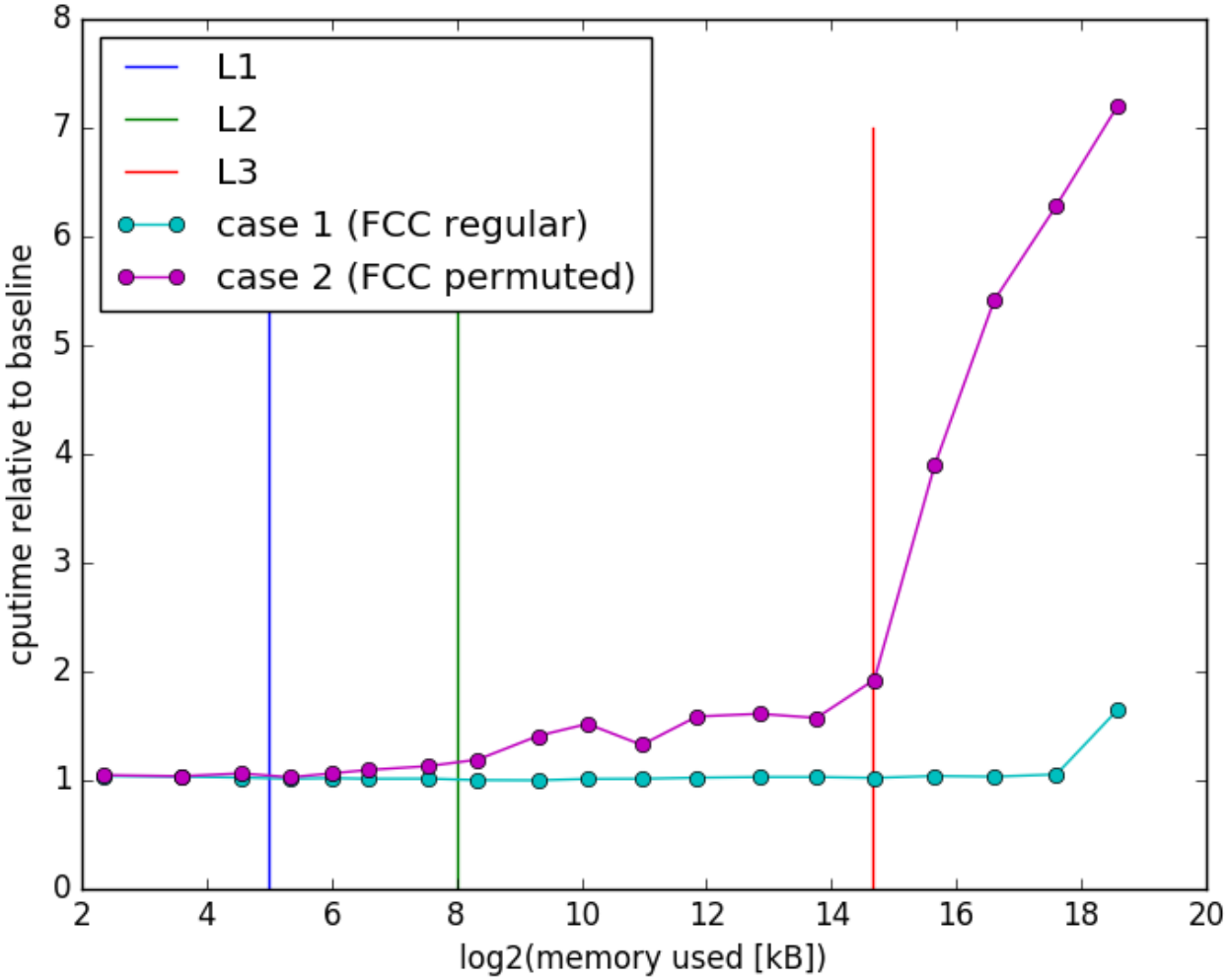
Case 1



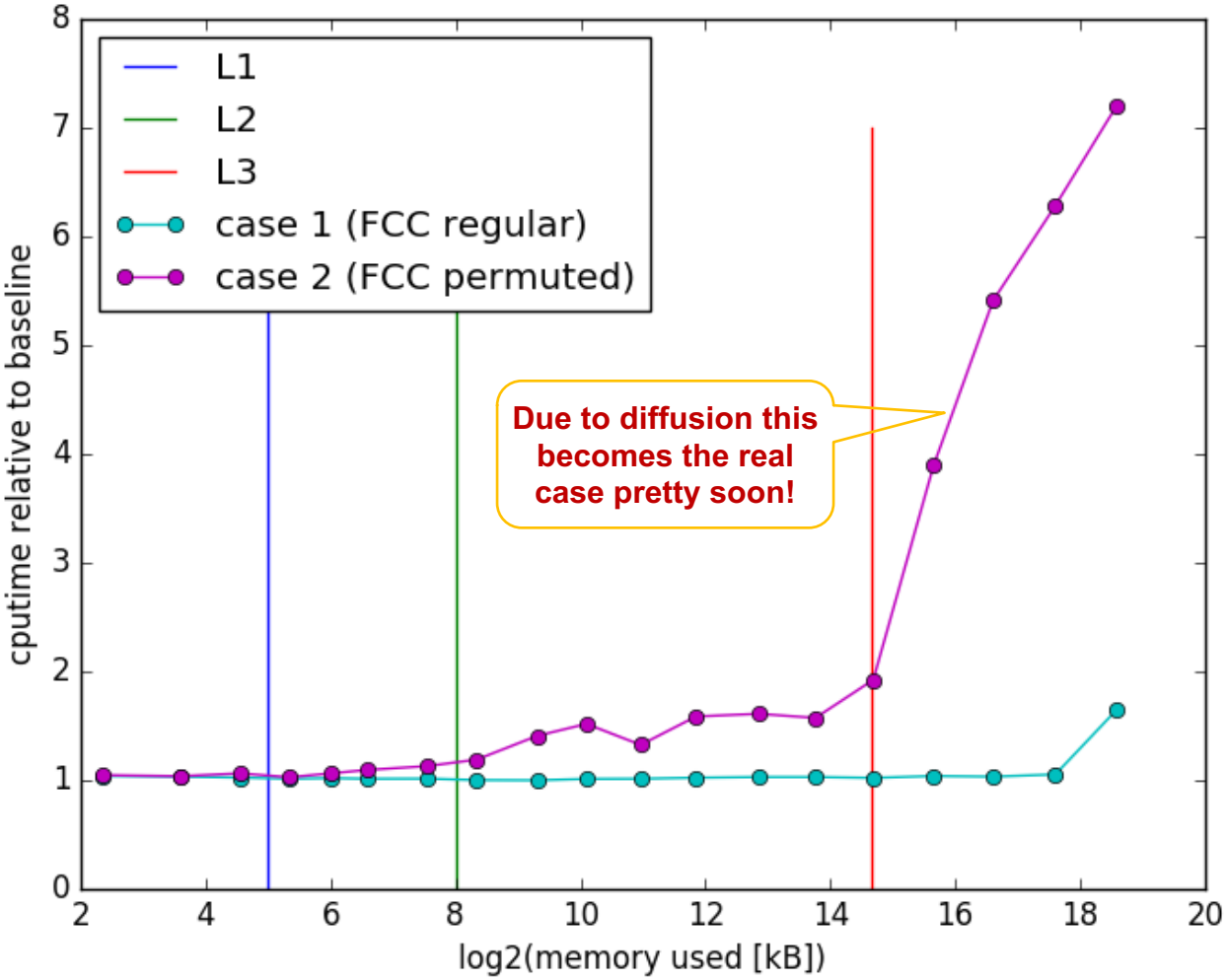
Case 1


L1
L2
L3
case 1 (FCC regular)
Close to baseline
cputime relative to baseline
log2(memory used [kB])
0
1
2
3
4
5
6
7
2
4
6
8
10
12
14
16
18
20

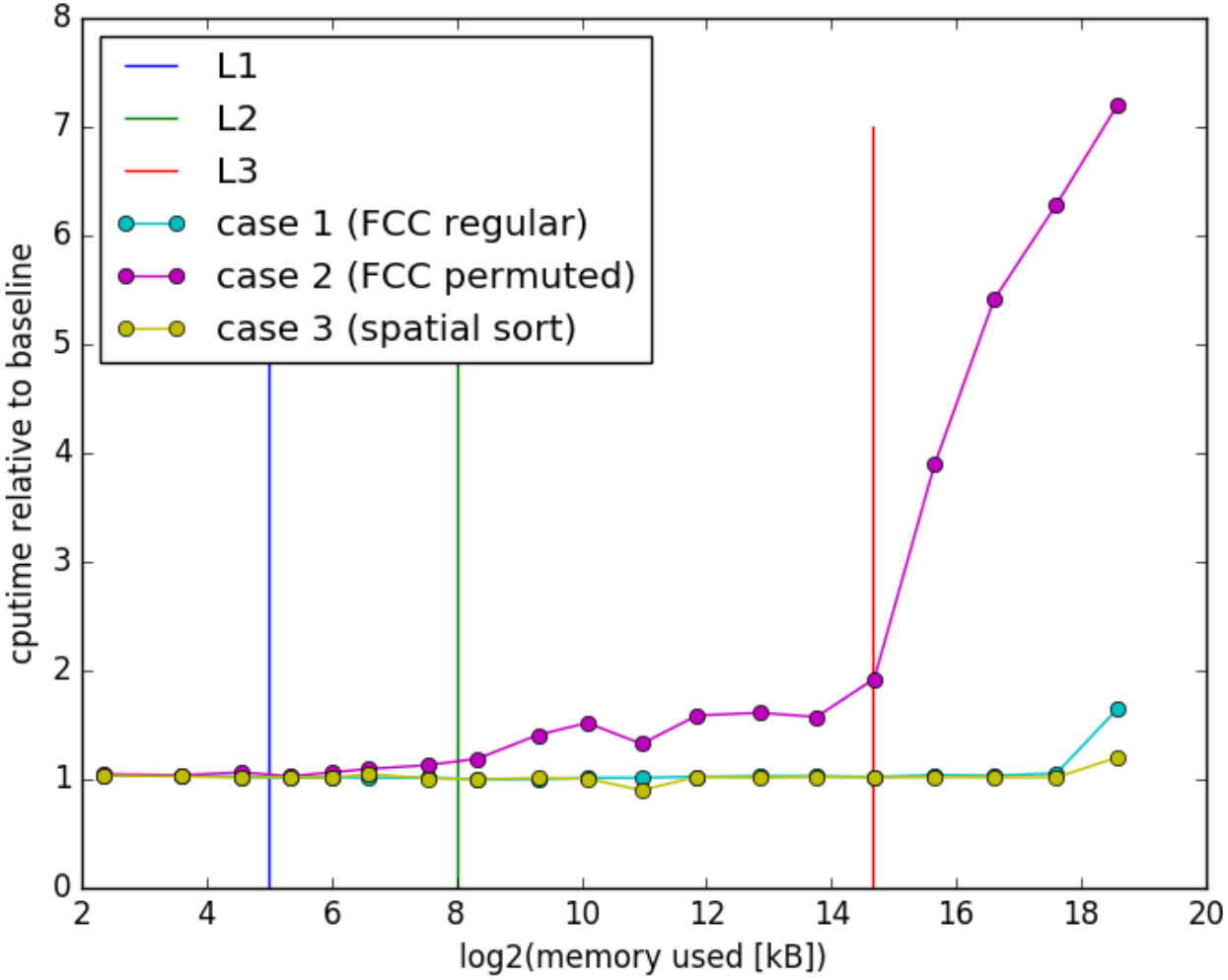
Case 2


L1
L2
L3
case 1 (FCC regular)
case 2 (FCC permuted)
cputime relative to baseline
log2(memory used [kB])
0
1
2
3
4
5
6
7
8
2
4
6
8
10
12
14
16
18
20

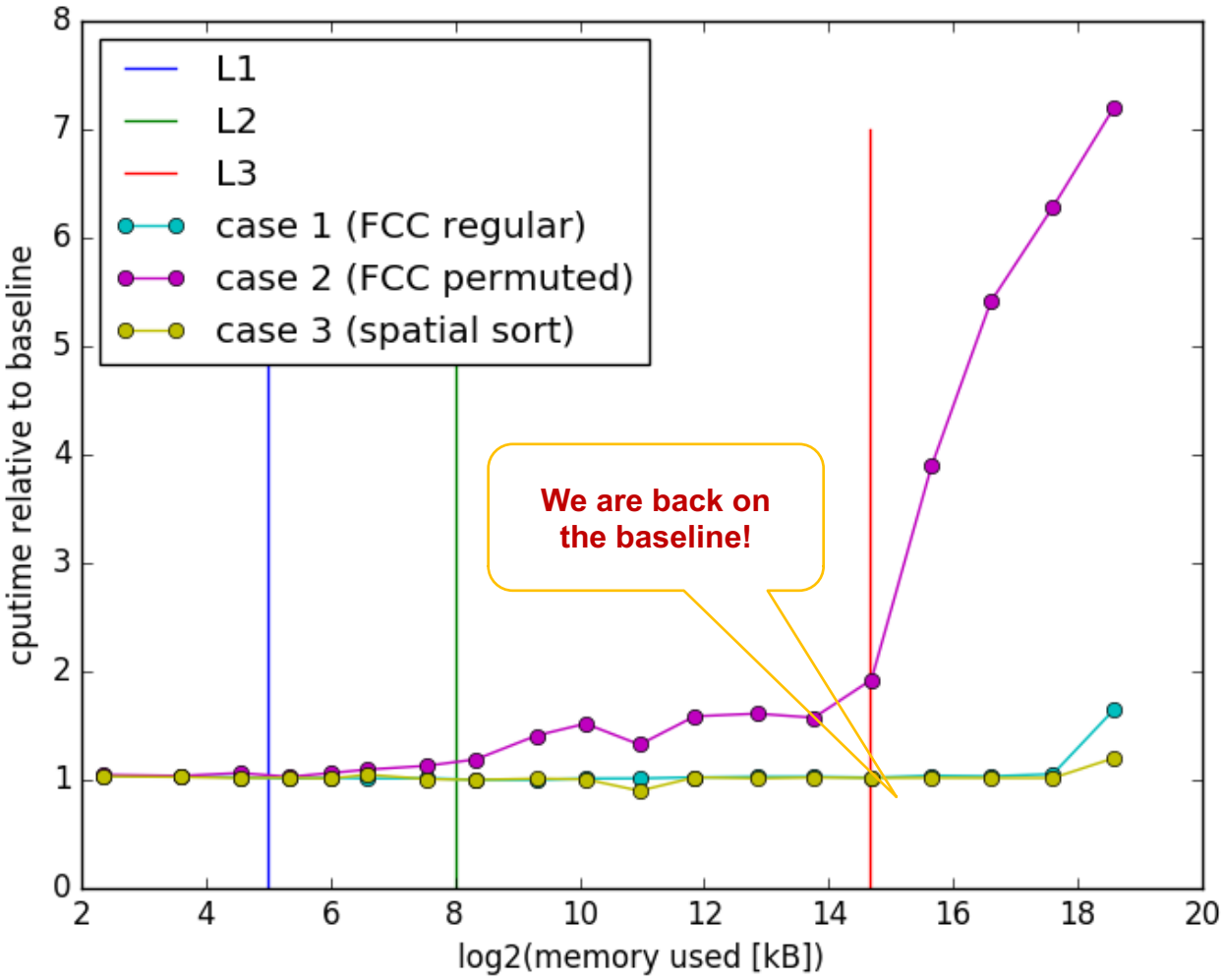
Case 2


L1
L2
L3
case 1 (FCC regular)
case 2 (FCC permuted)
Due to diffusion this becomes the real case pretty soon!
cputime relative to baseline
log2(memory used [kB])
0
1
2
3
4
5
6
7
8
2
4
6
8
10
12
14
16
18
20

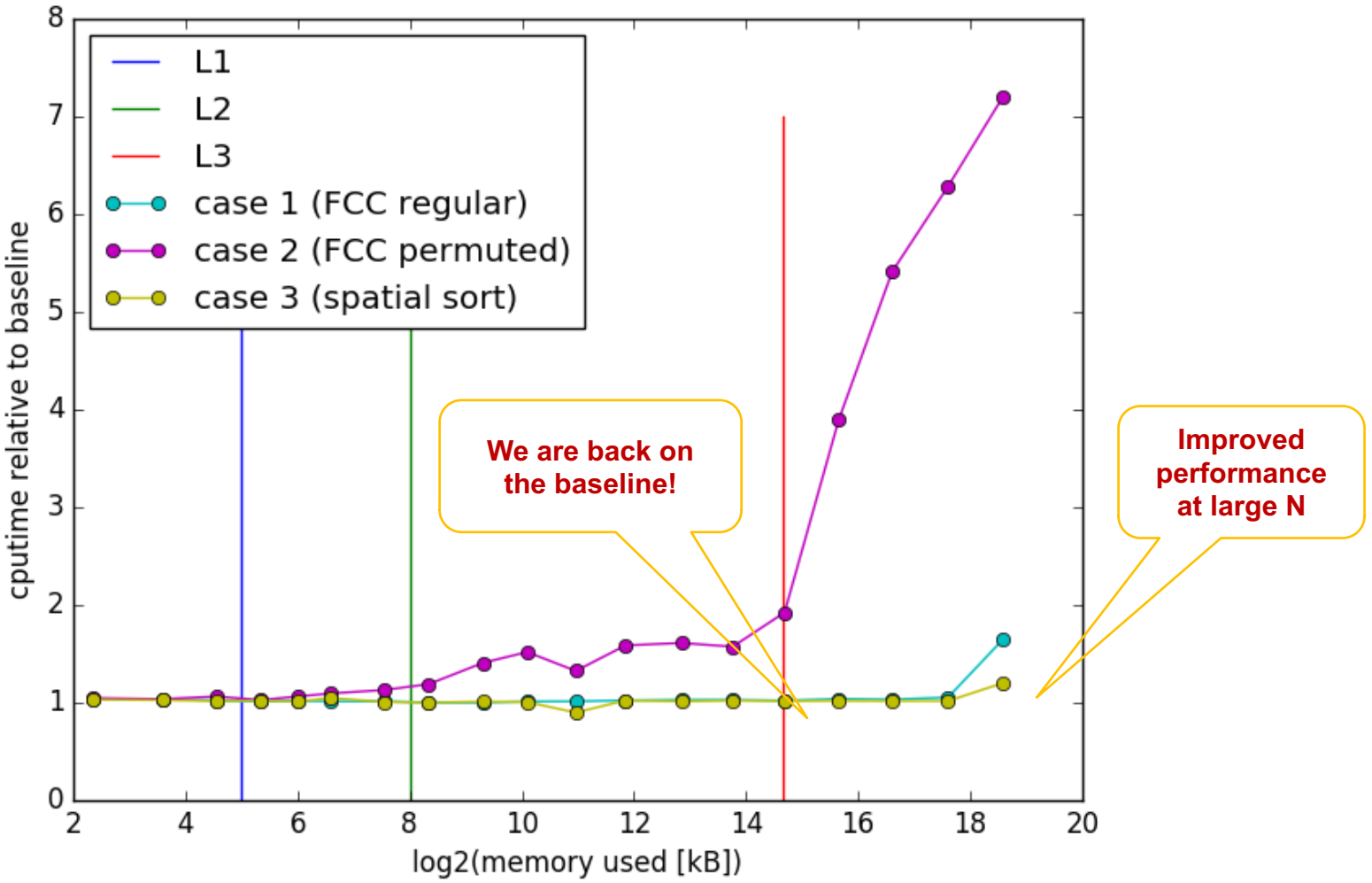
Case 3


L1
L2
L3
case 1 (FCC regular)
case 2 (FCC permuted)
case 3 (spatial sort)
cputime relative to baseline
log2(memory used [kB])
0
1
2
3
4
5
6
7
8
2
4
6
8
10
12
14
16
18
20

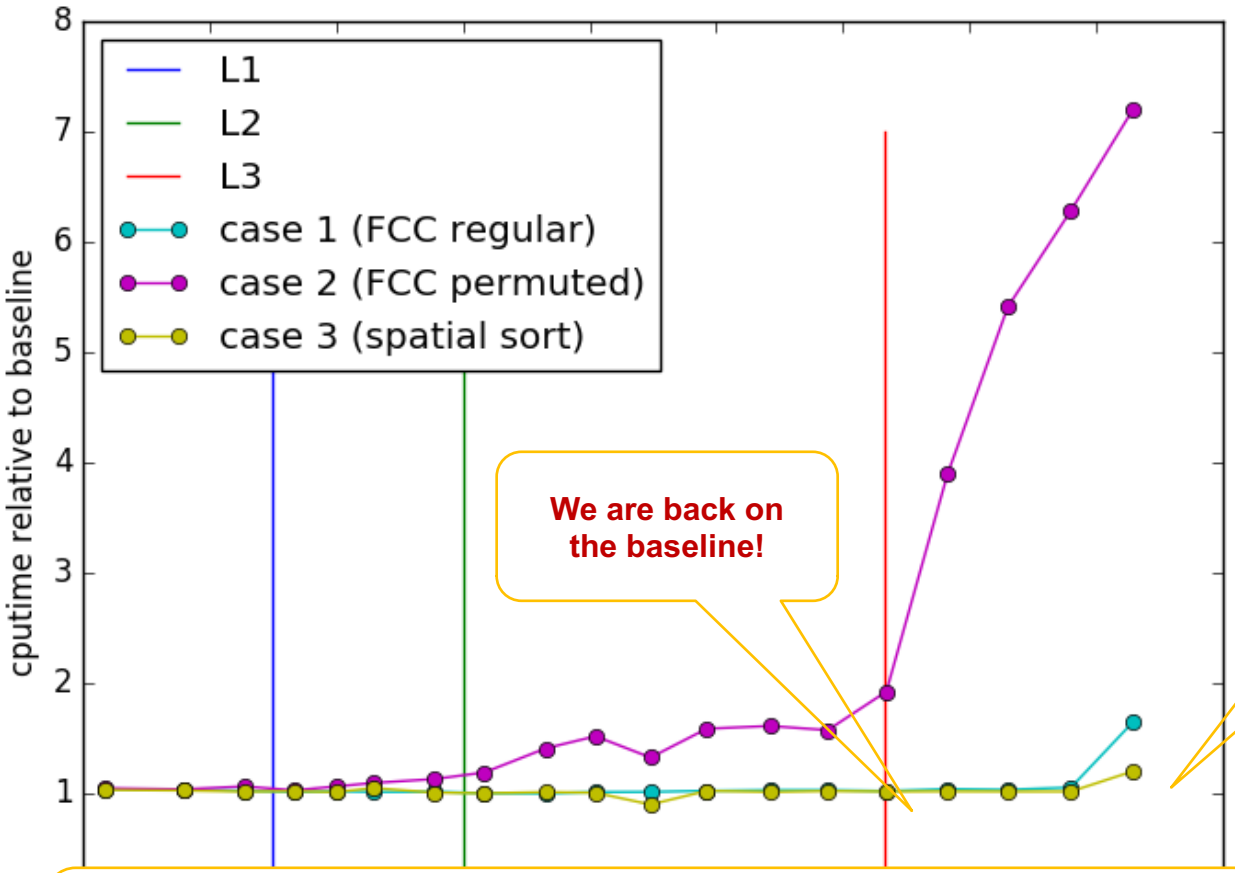
Case 3


L1
L2
L3
case 1 (FCC regular)
case 2 (FCC permuted)
case 3 (spatial sort)
We are back on the baseline!
cputime relative to baseline
log2(memory used [kB])
0
1
2
3
4
5
6
7
8
2
4
6
8
10
12
14
16
18
20

Case 3


L1
L2
L3
case 1 (FCC regular)
case 2 (FCC permuted)
case 3 (spatial sort)
cputime relative to baseline
log2(memory used [kB])
0
1
2
3
4
5
6
7
8
2
4
6
8
10
12
14
16
18
20
We are back on the baseline!
Improved performance at large N

Case 3



Spatial sort is relatively expensive
but the timestep and displacements are small
Cost can be amortized over many timesteps

Fixing the data access pattern

1. Sort atom property arrays (rx,ry,rz,vx,vy,vz,…) based on the Hilbert index h of the cell of the atoms (**spatial sort**).
   Atoms which are close in space (and hence will interact) will be close in memory (and hence will be in the cache with high probability)
2. Build a table containing the index of the first atom in each cell, and the number of atoms in the cell (Hilbert list)
3. Build Verlet list from the Hilbert list (discard the latter)
4. Compute the interactions by looping over the Verlet list and measure the performance (e.g. interactions/s)
5. Integrate forces, updating velocities and positions and time
6. If performance degrades
   jump back to step 1.
   else
   continue at step 4.

Fixing the data access pattern

1. Sort atom property arrays (rx,ry,rz,vx,vy,vz,…) based on the Hilbert index h of the cell of the atoms (**spatial sort**).
   Atoms which are close in space (and hence will interact) will be close in memory (and hence will be in the cache with high probability)
2. Build a table containing the index of the first atom in each cell, and the number of atoms in the cell (Hilbert list)
3. Build Verlet list from the Hilbert list (discard the latter)
4. Compute the interactions by looping over the Verlet list and measure the performance (e.g. interactions/s)
5. Integrate forces, updating velocities and positions and time
6. If performance degrades
   jump back to step 1.
   else
   continue at step 4.

Performance analysis

- atoms : 2,13 10^6
- pairs : 162 10^6
- Ratio : 76
- Pairs computed per second : 88.6 10^6

- B/atom: 376,5 (320B is in the verlet list)
- Bandwidth = 9,5 GB/s (measured by mlc 2 reads : 1 write)
- maximum atoms per second: 27.000.000
- actual atoms per second: 1.160.000
- ratio: 0,04

- flops_per_pair : 27
- flops_per_atom : 2055,5
- gflops_per_second: 2,39
- peak performance : 11,2
- ratio : 0,21

Performance analysis

- atoms : 2,13 10^6
- pairs : 162 10^6
- Ratio : 76
- Pairs computed per second : 88.6 10^6

- B/atom: 376,5 (320B is in the verlet list)
- Bandwidth = 9,5 GB/s (measured by mlc 2 reads : 1 write)
- maximum atoms per second: 27.000.000
- actual atoms per second: 1.160.000
- ratio: 0,04

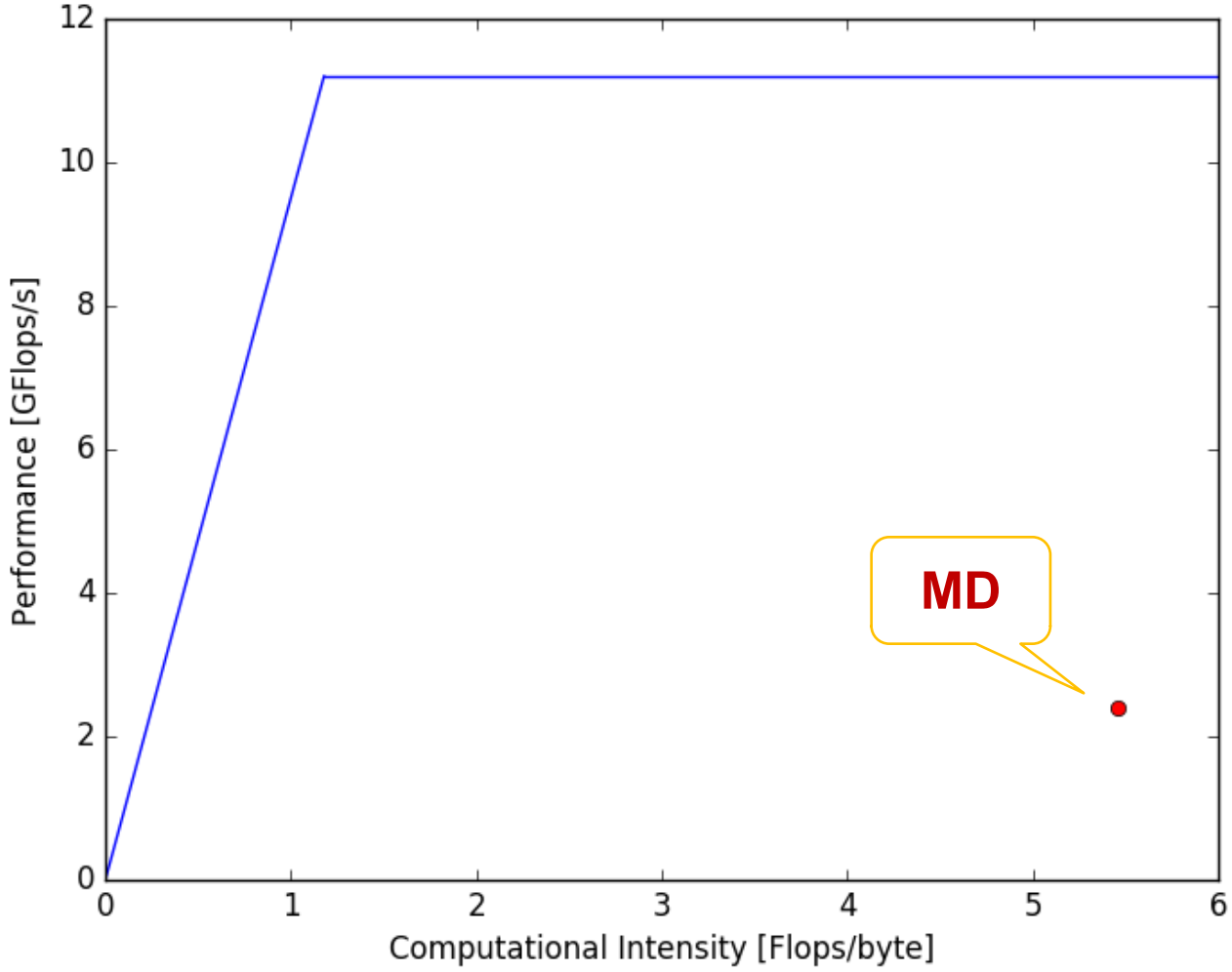
Not memory bound

- flops_per_pair : 27
- flops_per_atom : 2055,5
- gflops_per_second: 2,39
- peak performance : 11,2
- ratio : 0,21

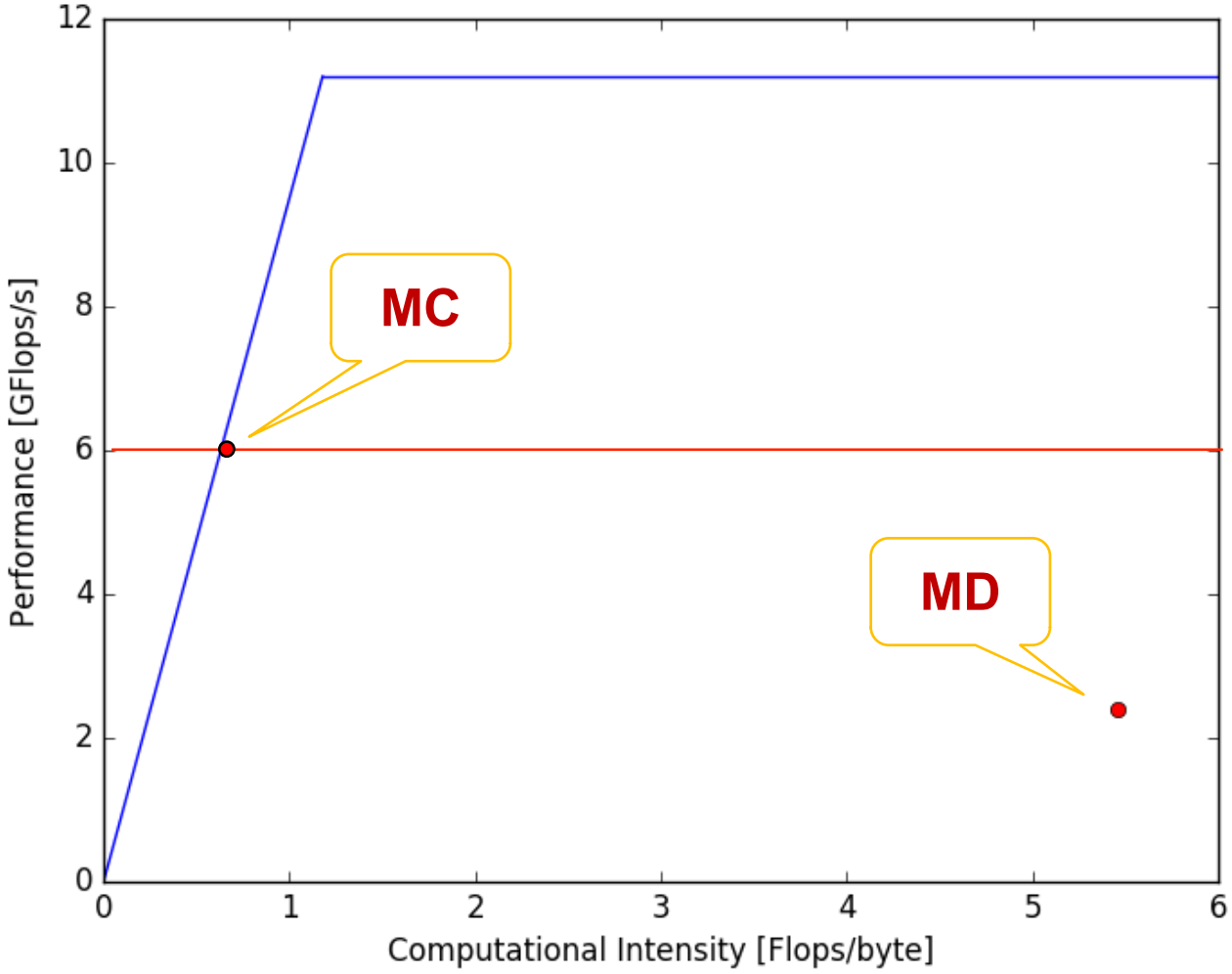
Lot of flops per atom!

Not compute bound either!

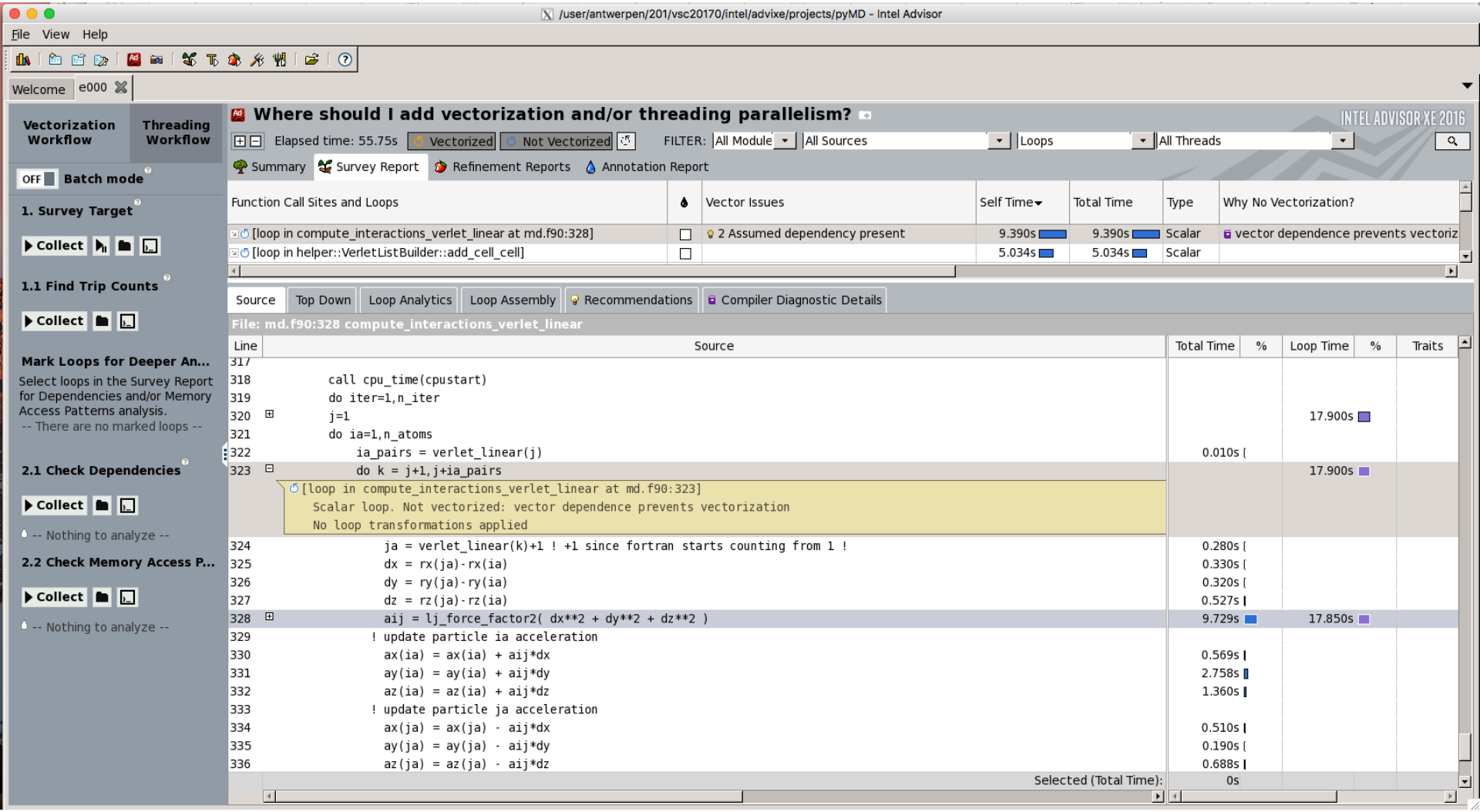
Roofline MD setting



Roofline MD setting



What do the intel tools tell?
Advisor



What do the intel tools tell?

```
j=1
do ia=1,n_atoms
    ia_pairs = verlet_linear(j) ! Size of the Verlet list of atom ia

    do k = j+1,j+ia_pairs
        ja = verlet_linear(k)+1 ! +1 since Fortran starts counting from 1 !
        dx = rx(ja)-rx(ia)
        dy = ry(ja)-ry(ia)
        dz = rz(ja)-rz(ia)
        aij = lj_force_factor2( dx**2 + dy**2 + dz**2 )
      ! update particle ia acceleration
        ax(ia) = ax(ia) + aij*dx
        ay(ia) = ay(ia) + aij*dy
        az(ia) = az(ia) + aij*dz
      ! update particle ja acceleration
        ax(ja) = ax(ja) - aij*dx
        ay(ja) = ay(ja) - aij*dy
        az(ja) = az(ja) - aij*dz
    enddo
    j = j + 1 + ia_pairs
enddo
```

What do the intel tools tell?

```
j=1
do ia=1,n_atoms
    ia_pairs = verlet_linear(j) ! Size of the Verlet list of atom ia

    do k = j+1,j+ia_pairs
        ja = verlet_linear(k)+1 ! +1 since Fortran starts counting from 1 !
        dx = rx(ja)-rx(ia)
        dy = ry(ja)-ry(ia)
        dz = rz(ja)-rz(ia)
        aij = lj_force_factor2( dx**2 + dy**2 + dz**2 )
      ! update particle ia acceleration
        ax(ia) = ax(ia) + aij*dx
        ay(ia) = ay(ia) + aij*dy
        az(ia) = az(ia) + aij*dz
      ! update particle ja acceleration
        ax(ja) = ax(ja) - aij*dx
        ay(ja) = ay(ja) - aij*dy
        az(ja) = az(ja) - aij*dz
    enddo
    j = j + 1 + ia_pairs
enddo
```

- SIMD vectorization means that you update `ax(aj)` for 4 successive `ja` values (also `ay(aj)` and `az(ja)`)
- The compiler cannot know that the `ja` are different
- Assumed dependency
- We know that the `ja` are different by construction of the Verlet list
- We must tell the compiler to ignore assumed dependencies

What do the intel tools tell?

```
j=1
do ia=1,n_atoms
    ia_pairs = verlet_linear(j) ! Size of the Verlet list of atom ia
    !DIR$ SIMD
    do k = j+1,j+ia_pairs
        ja = verlet_linear(k)+1 ! +1 since Fortran starts counting from 1 !
        dx = rx(ja)-rx(ia)
        dy = ry(ja)-ry(ia)
        dz = rz(ja)-rz(ia)
        aij = lj_force_factor2( dx**2 + dy**2 + dz**2 )
      ! update particle ia acceleration
        ax(ia) = ax(ia) + aij*dx
        ay(ia) = ay(ia) + aij*dy
        az(ia) = az(ia) + aij*dz
      ! update particle ja acceleration
        ax(ja) = ax(ja) - aij*dx
        ay(ja) = ay(ja) - aij*dy
        az(ja) = az(ja) - aij*dz
    enddo
    j = j + 1 + ia_pairs
enddo
```

Ignore assumed dependencies and vectorize the loop

What do the intel tools tell?

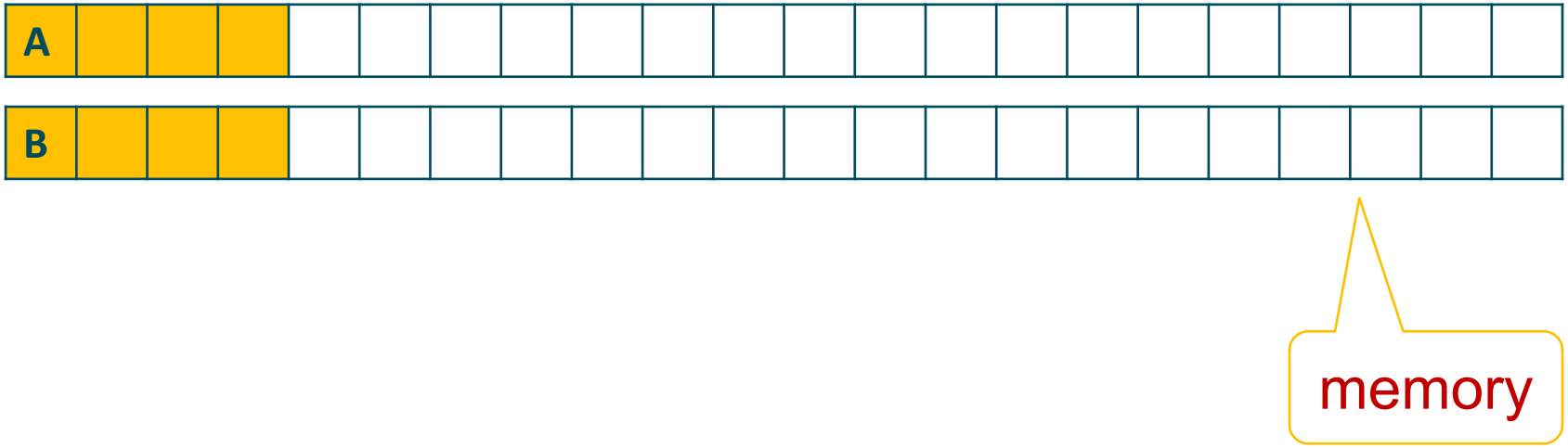
- "inserts present" = hint for gather/scatter
- Filling a vector register element per element (in the case of non-contiguous elements)
- AVX (highest SIMD extension available on Hopper) has no built-in support for gather/scatter
- AVX2 has special instructions for gather/scatter
  - Available on BrENIAC
  - Available on successor of Turing

```
j=1
do ia=1,n_atoms
    ia_pairs = verlet_linear(j) ! Size of the Verlet list of atom ia
    !DIR$ SIMD
    do k = j+1,j+ia_pairs
        ja = verlet_linear(k)+1 ! +1 since Fortran starts counting from 1 !
        dx = rx(ja) - rx(ia)
        dy = ry(ja) - ry(ia)
        dz = rz(ja) - rz(ia)
        aij = lj_force_factor2( dx**2 + dy**2 + dz**2 )
      ! update particle ia acceleration
        ax(ia) = ax(ia) + aij*dx
        ay(ia) = ay(ia) + aij*dy
        az(ia) = az(ia) + aij*dz
      ! update particle ja acceleration
        ax(ja) = ax(ja) - aij*dx
        ay(ja) = ay(ja) - aij*dy
        az(ja) = az(ja) - aij*dz
    enddo
    j = j + 1 + ia_pairs
enddo
```

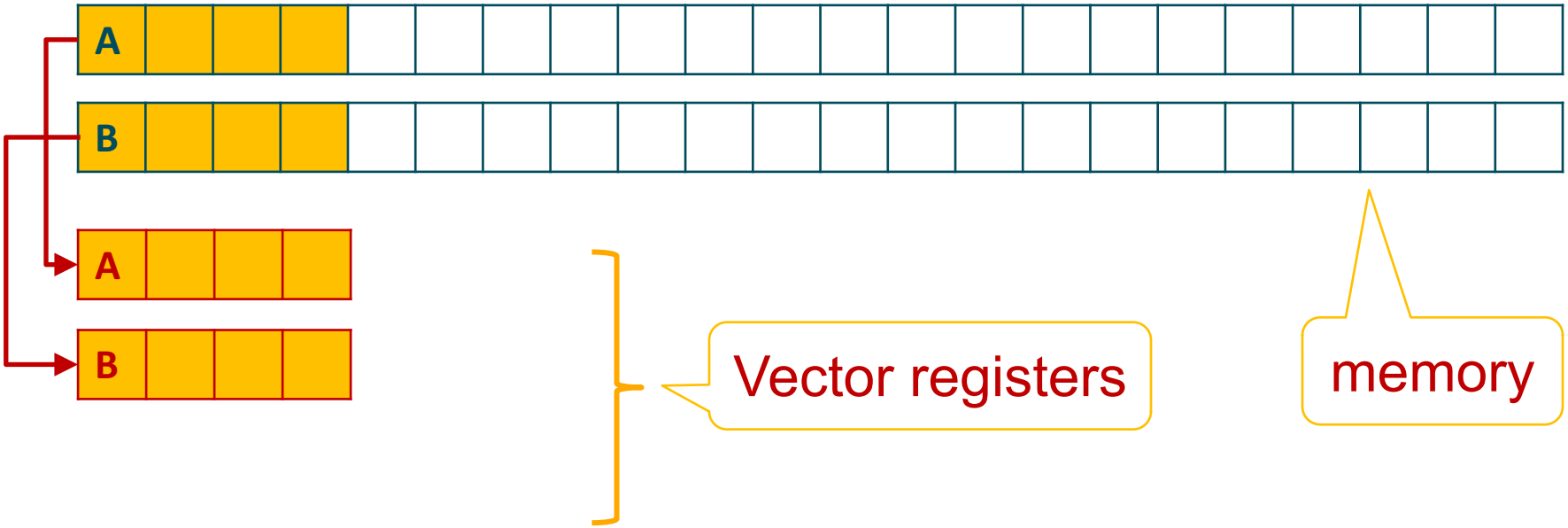
Gather operation moving rx(ja), ry(ja), rz(ja) for 4 successive ja values into the vector registers

Scatter operation moving ax(ja), ay(ja), az(ja) for 4 successive ja values out of the vector registers.

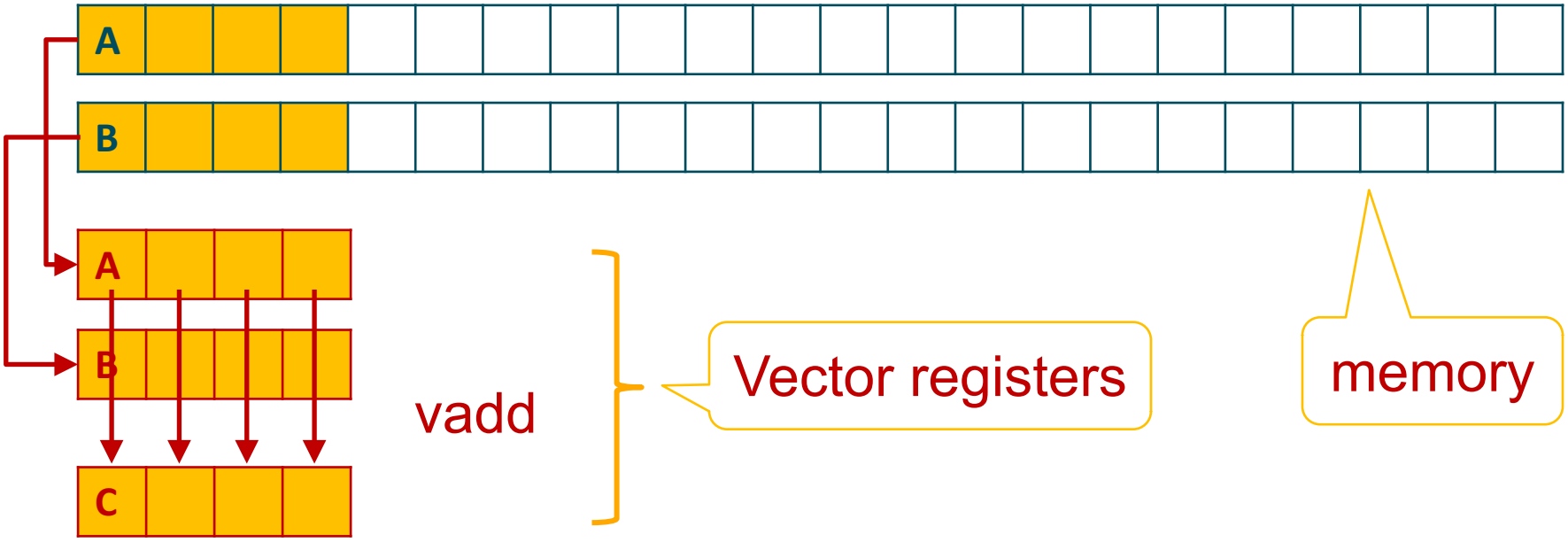
- C = A+B
- No gather/scatter, 4 successive items moved as a block into/out of vector registers



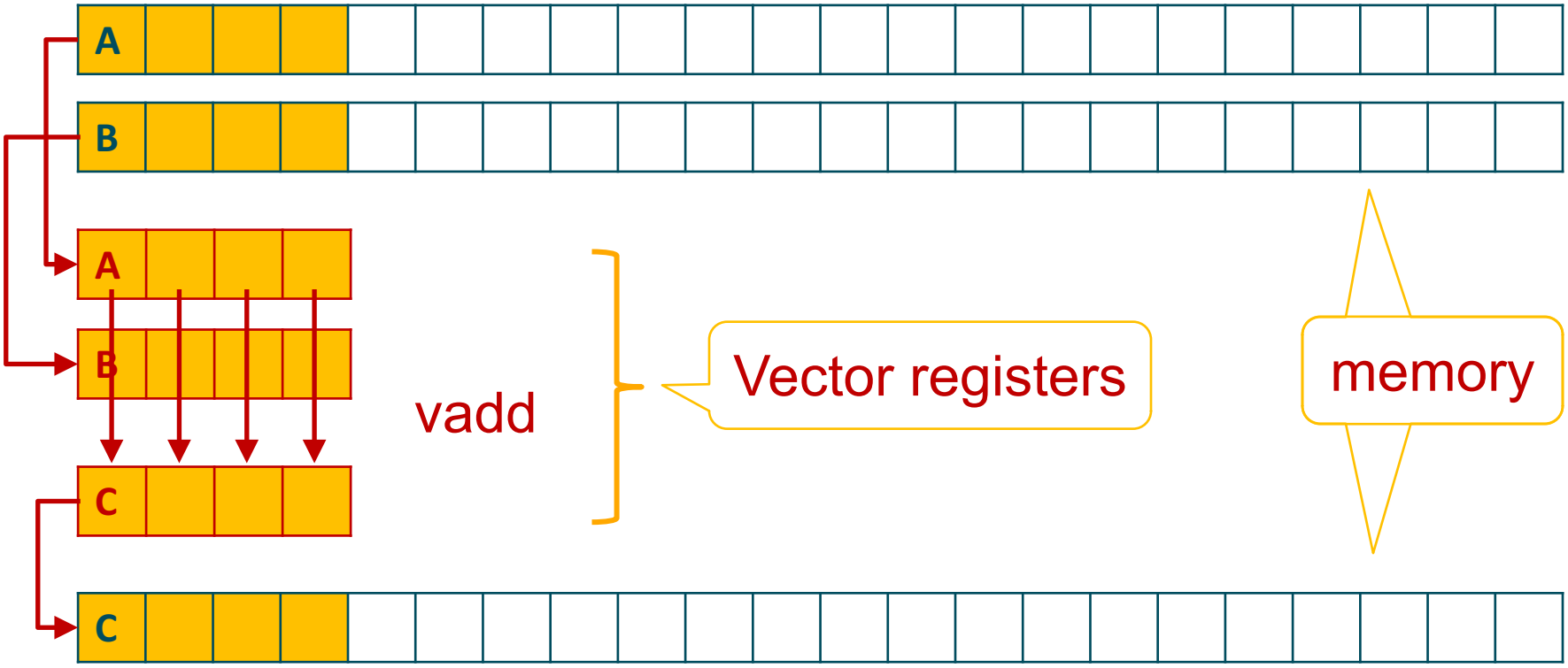
- C = A+B
- No gather/scatter, 4 successive items moved as a block into/out of vector registers



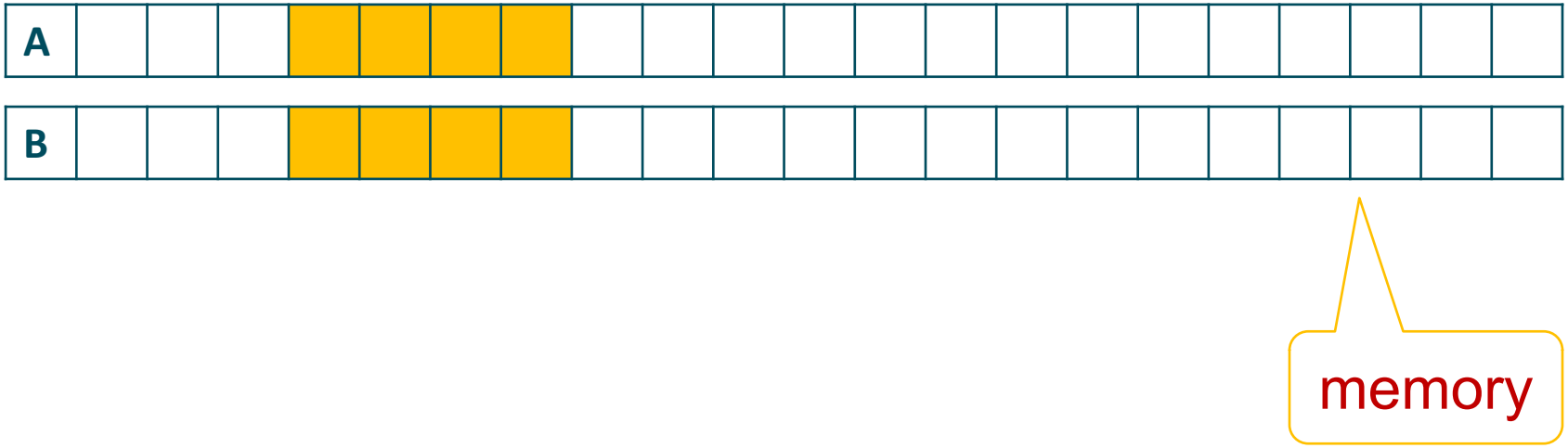
- C = A+B
- No gather/scatter, 4 successive items moved as a block into/out of vector registers



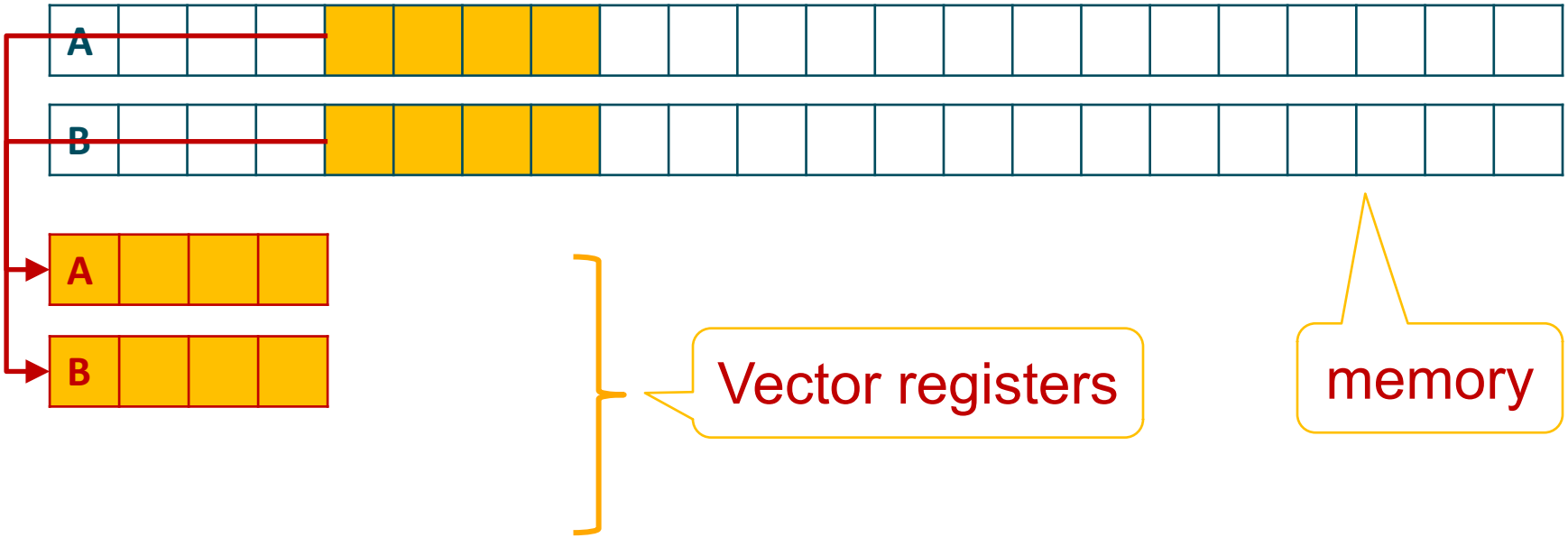
- C = A+B
- No gather/scatter, 4 successive items moved as a block into/out of vector registers



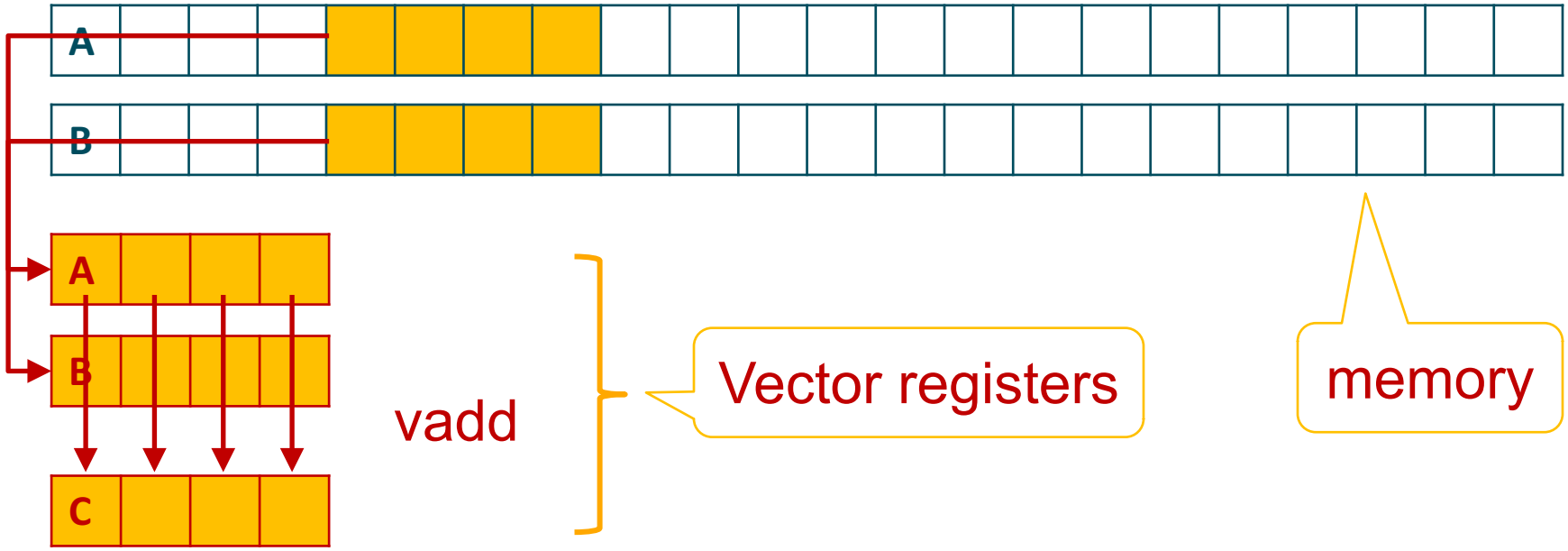
- C = A+B
- No gather/scatter, 4 successive items moved as a block into/out of vector registers



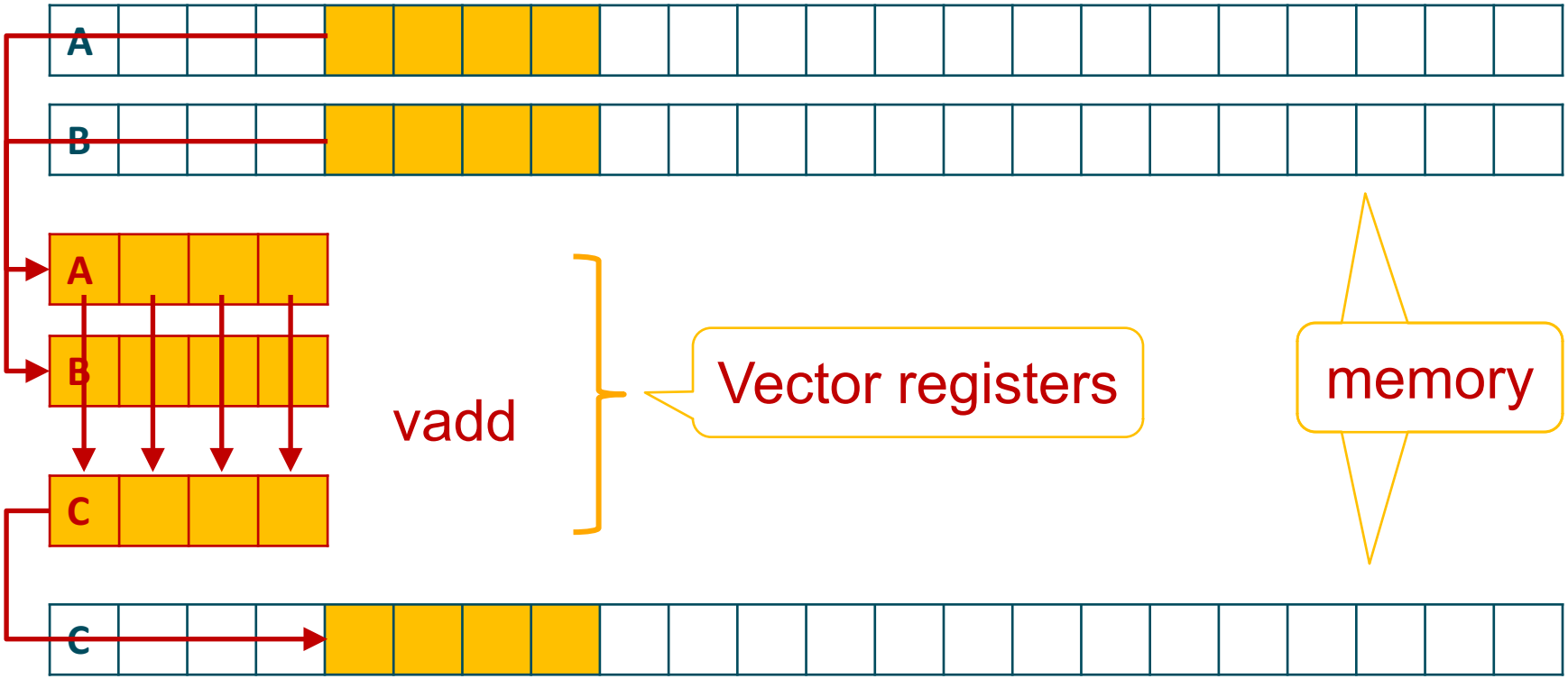
- C = A+B
- No gather/scatter, 4 successive items moved as a block into/out of vector registers



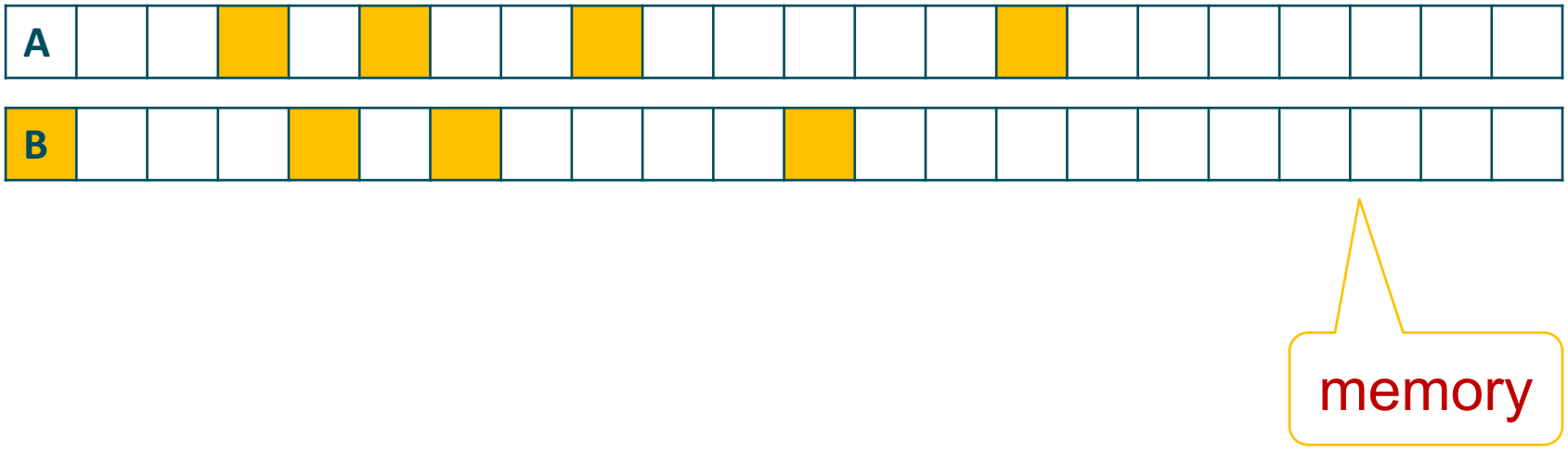
- C = A+B
- No gather/scatter, 4 successive items moved as a block into/out of vector registers



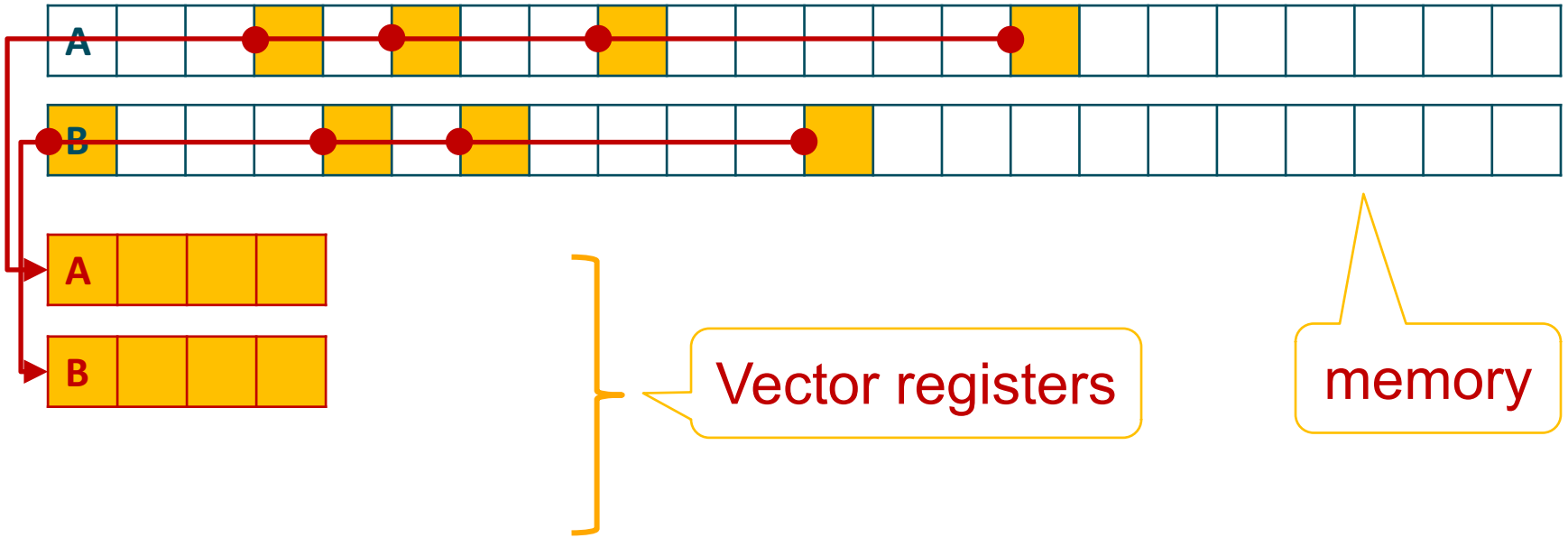
- C = A+B
- No gather/scatter, 4 successive items moved as a block into/out of vector registers



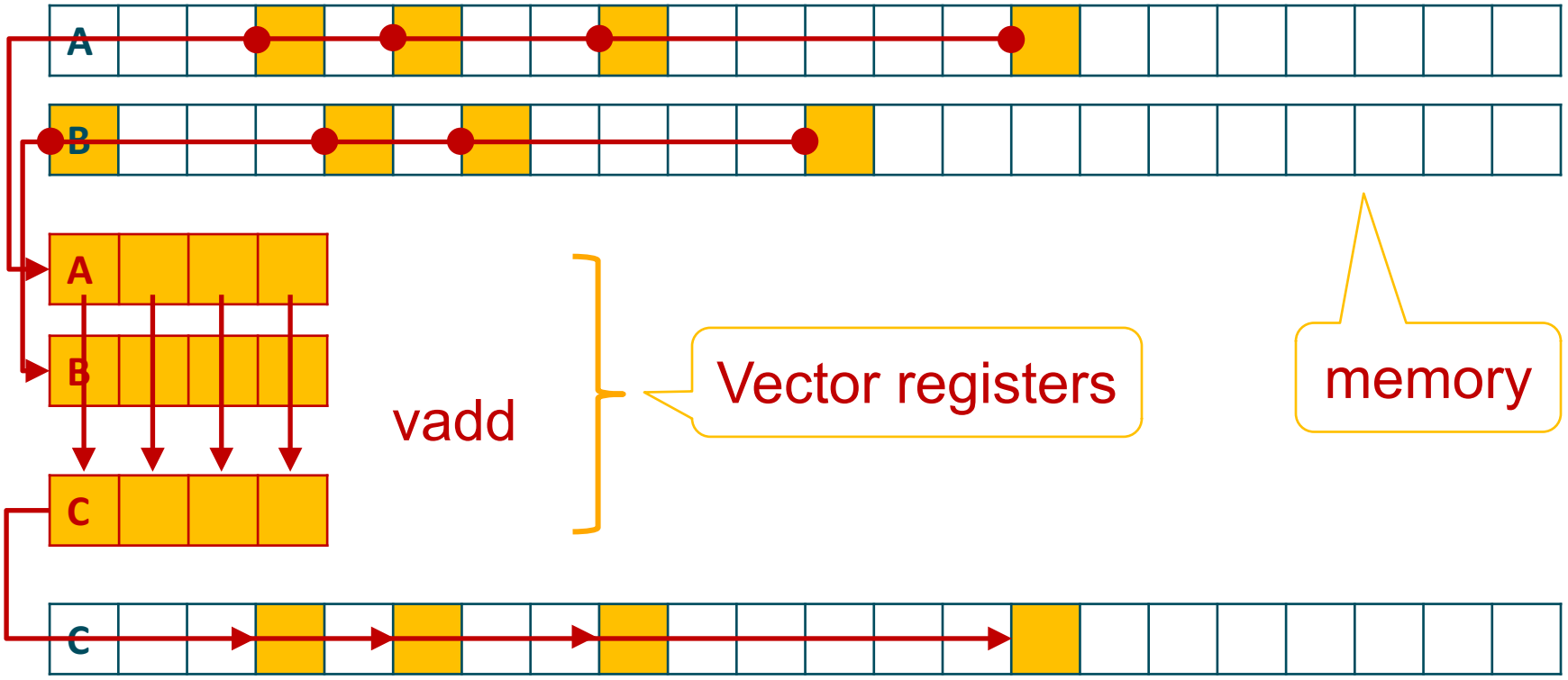
- C = A+B
- gather/scatter, items moved one by one into/out of vector registers



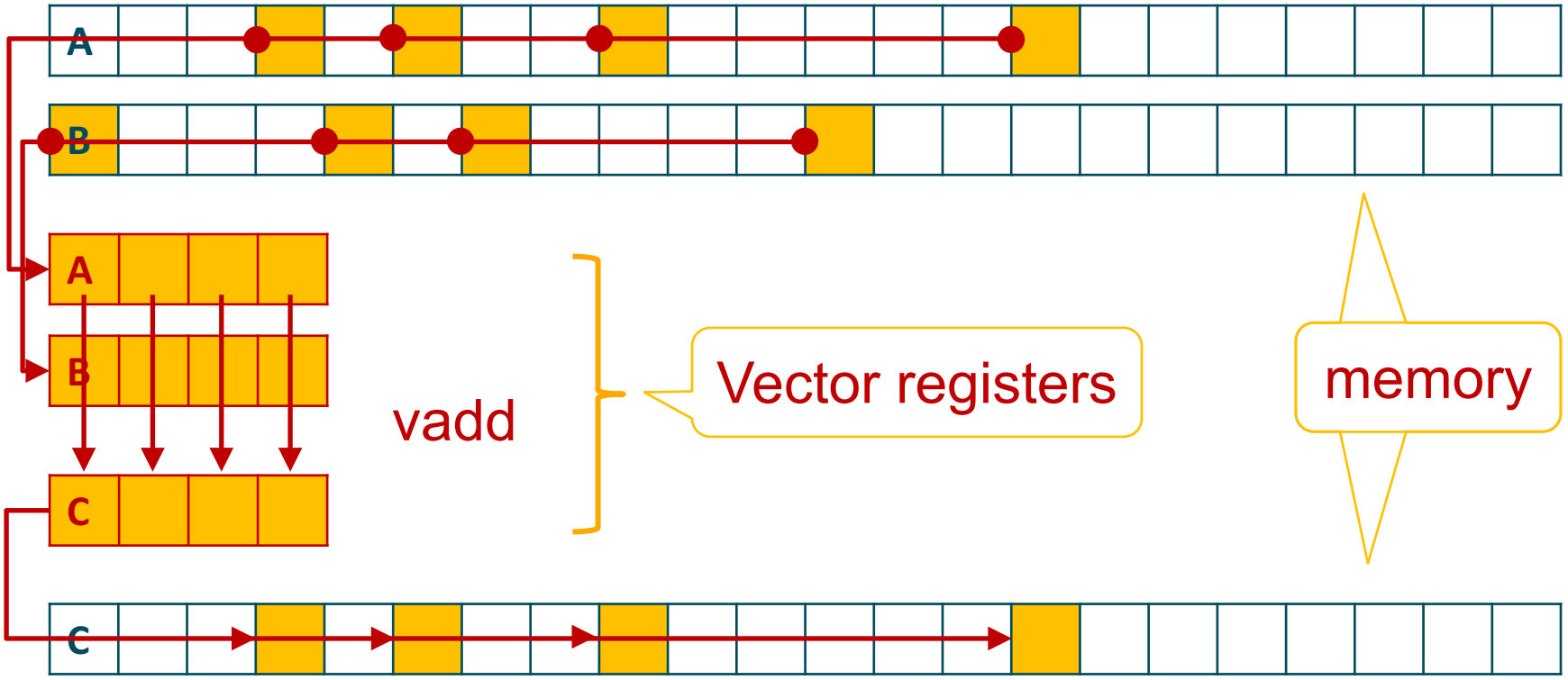
- C = A+B
- gather/scatter, items moved one by one into/out of vector registers



- C = A+B
- gather/scatter, items moved one by one into/out of vector registers



- C = A+B
- gather/scatter, items moved one by one into/out of vector registers



What do the intel tools tell?

- LLC misses 0.008 %
- LJ_force_factor2 0.59 cpi
- Compute_interactions 0.77 cpi

- cpi at best 0.25
- cpi ≤ 1 considered acceptable in HPC

- There is still room for improvement
  - No more low hanging fruit, though

Beyond auto-vectorisation

- Assembly code
  - Architecture dependent code
- Libraries for writing explicit vector code
  - Architecture independent code
  - C++ only
  - **Vc** (github.com/VcDevel/Vc)
  - **Boost.simd** (github.com/NumScale/boost.simd)
  - Similar performance
- Rewrite algorithm in terms of vector operations rather than scalar operations

Beyond auto-vectorisation

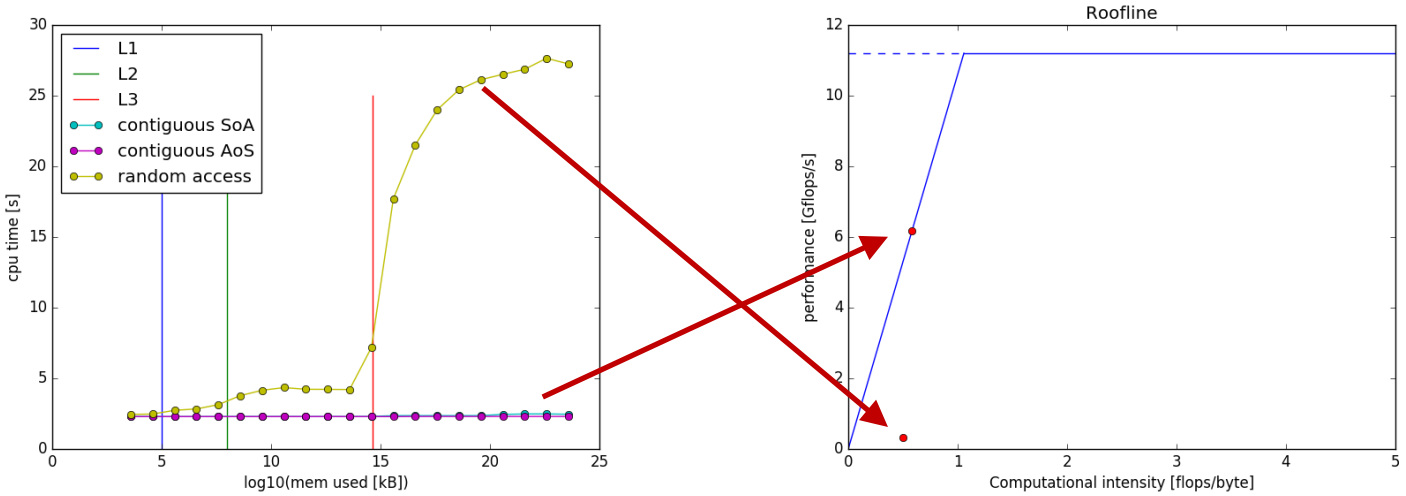
- Gain:
  - 120–127 10^6 interactions per second (instead of 90 10^6)
  - 30% improvement
  - Not bad, but not effortless (took me about 1½ week)
  - AVX2 will probably do even better
- Coding details in other talk

Conclusions

1. Data access pattern is crucial to performance

Conclusions

- Monte Carlo case (3N reads, no writes)
- Contiguous data access:
  - 450 10^6 interactions/s
  - Bandwidth saturation (machine limit)
- Random access:
  - Performance drops by factor 15 (cache misses, gather/scatter)

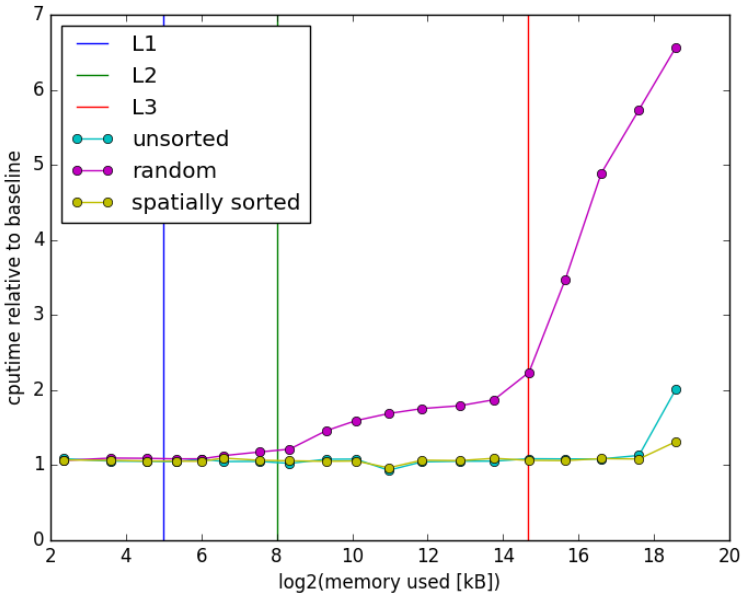


Conclusions

- Bandwidth saturation means that the CPU is waiting for the data to arrive
  - Try to do more computations with the data that is available, e.g.:
    - Additional computations
    - A more complex model
  - Program will not run faster but will do more work in the same time

Conclusions

- Molecular dynamics case (6N reads, 3N writes + VL)
- Contiguous data access:
  - 90 10^6 interactions/s
  - No machine limits hit
  - gather/scatter
- Random access:
  - Performance drops by factor 7
- Spatial sort fixes the problem



- Fixing the data access pattern is not always easy
- Involves usually some form of sorting the data

Conclusions

1. Data access pattern is crucial to performance
2. **Spatial sort using space filling curves is useful technique for fixing data access patterns**

Conclusions

1. Data access pattern is crucial to performance
2. Spatial sort using space filling curves
3. **Intel tools (Advisor, VTune) provide useful clues to optimizing your code**
   - **Hot spots**
   - **The nature of hot spots (data access, expensive instructions, …)**
   - **Issues with vectorization**

Conclusions

1. Data access pattern is crucial to performance
2. Spatial sort using space filling curves is useful
3. Intel tools (Advisor, VTune) provide useful clues
4. **Fortran has many advantages for programming number-crunching routines**
   - **F2py for producing python modules**

Python/Fortran/C++

Python/Numpy

- verify code correctness
- generate FCC lattice
- generate arrays filled with random numbers
- generate permutations
- zero accelerations between time steps
- build Verlet list
- spatial sort of atom property arrays
- define coarse computational strategy and data structures
- control and initialize the experiments
- plot results

code **80**%

cputime 5%

Fortran

- Lennard-Jones potential
- Lennard-Jones forces
- iterate over Verlet list and compute interactions
- iterate over array and compute interactions (baselines)

code 10%

cputime **90**%

C++

- compute hilbert indices
- build Verlet list
- iterate over Verlet list and compute inter–actions using simd libraries

10%

cputime 5%

Speedup of 1200x !

Conclusions

1. Data access pattern is crucial to performance
2. Spatial sort using space filling curves is useful
3. Intel tools (Advisor, VTune) provide useful clues
4. Fortran has many advantages
5. **Use a simple but relevant baseline that you understand to judge the performance of your code and direct your efforts**

Conclusions

1. Data access pattern is crucial to performance
2. Spatial sort using space filling curves is useful
3. Intel tools (Advisor, VTune) provide useful clues
4. Fortran has many advantages
5. Use a simple relevant baseline
6. **Be aware of the machine limits**
   - **Bandwidth**
   - **Peak performance**
   - **Roofline model**

Conclusions

1. Data access pattern is crucial to performance
2. Spatial sort using space filling curves is useful
3. Intel tools (Advisor, VTune) provide useful clues
4. Fortran has many advantages
5. Use a simple relevant baseline
6. Be aware of the machine limits
7. **SIMD libraries are useful**
   - **Vc, Boost.simd**

Conclusions

1. Data access pattern is crucial to performance
2. Spatial sort using space filling curves is useful
3. Intel tools (Advisor, VTune) provide useful clues
4. Fortran has many advantages
5. Use a simple relevant baseline
6. Be aware of the machine limits
7. Simd libraries are useful

Thank you

You are always welcome to discuss your (computational) problems