Introduction to High Performance Computing

\\

SDS406 – Fall semester, 2024 - 2025

\\

L04: OpenMP and Optimization, 21st October 2024

Outline for today

OpenMP

- Review of OpenMP examples; `axpy` and `xdoty`

Outline for today

OpenMP

- Review of OpenMP examples; `axpy` and `xdoty`

Performance analysis and optimization

- Understanding of program performance
- Understanding of hardware characteristics in terms of performance
- Understanding of *potential* performance of a program given the hardware characteristics
- Optimization steps to approach potential performance

OpenMP Examples

Linear algebra

Operation:

$$z_i = a x_i + y_i$$

OpenMP Examples

Linear algebra

Operation:

$$z_i = a x_i + y_i$$

Number of *floating point operations*

OpenMP Examples

Linear algebra

Operation:

$$z_i = a x_i + y_i$$

Number of *floating point operations*

- For each iteration, i.e. for each element `i`

OpenMP Examples

Linear algebra

Operation:

$$z_i = ax_i + y_i$$

Number of *floating point operations*

- For each iteration, i.e. for each element `i`
  - One multiplication (ax_i)

OpenMP Examples

Linear algebra

Operation:

$$z_i = a x_i + y_i$$

Number of *floating point operations*

- For each iteration, i.e. for each element `i`
  - One multiplication ($a x_i$)
  - One addition ($+y_i$)

OpenMP Examples

Linear algebra

Operation:

$$z_i = a x_i + y_i$$

Number of *floating point operations*

- For each iteration, i.e. for each element `i`
  - One multiplication ($a x_i$)
  - One addition ($+y_i$)
- Therefore, a total of 2 floating point operations per iteration

OpenMP Examples

Linear algebra

Operation:

$$z_i = ax_i + y_i$$

Number of *floating point operations*

- For each iteration, i.e. for each element `i`
  - One multiplication (ax_i)
  - One addition ($+y_i$)
- Therefore, a total of 2 floating point operations per iteration
- For `n` iterations, `2*n` floating point operations (flops) in total

OpenMP Examples

Linear algebra

Operation:

$$z_i = a x_i + y_i$$

Number of *floating point operations*

- For each iteration, i.e. for each element `i`
  - One multiplication ($a x_i$)
  - One addition ($+y_i$)
- Therefore, a total of 2 floating point operations per iteration
- For `n` iterations, `2*n` floating point operations (flops) in total
- If the `2*n` flops require `t1` seconds, the floating point rate is: `2*n/t1` flops/s

OpenMP Examples

Linear algebra

Operation:

$$z_i = a x_i + y_i$$

Number of *floating point operations*

- For each iteration, i.e. for each element `i`
  - One multiplication ($a x_i$)
  - One addition ($+y_i$)
- Therefore, a total of 2 floating point operations per iteration
- For `n` iterations, `2*n` floating point operations (flops) in total
- If the `2*n` flops require `t1` seconds, the floating point rate is: `2*n/t1` flops/s
- Equivalently `2*n/t1/1e9` Gflops/s

OpenMP Examples

Linear algebra

Operation:

$$z_i = ax_i + y_i$$

Number of *floating point operations*

- For each iteration, i.e. for each element `i`
    - One multiplication (ax_i)
    - One addition ($+y_i$)
- Therefore, a total of 2 floating point operations per iteration
- For `n` iterations, `2*n` floating point operations (flops) in total
- If the `2*n` flops require `t1` seconds, the floating point rate is: `2*n/t1` flops/s
- Equivalently `2*n/t1/1e9` Gflops/s

```
double p = 2.0*n/t1/1e9;
printf(" nth = %2d, t0 = %lf sec, t1 = %lf sec, diff z norm = %e, perf = %lf Gflop/s\n", nth, t0, t1, norm, p);
```

OpenMP Examples

Linear algebra

Operation:

$$r = \sum_{i=0}^{n-1} x_i \cdot y_i$$

OpenMP Examples

Linear algebra

Operation:

$$r = \sum_{i=0}^{n-1} x_i \cdot y_i$$

Number of *floating point operations*

OpenMP Examples

Linear algebra

Operation:

$$r = \sum_{i=0}^{n-1} x_i \cdot y_i$$

Number of *floating point operations*

- For each iteration, i.e. for each element `i`

OpenMP Examples

Linear algebra

Operation:

$$r = \sum_{i=0}^{n-1} x_i \cdot y_i$$

Number of *floating point operations*

- For each iteration, i.e. for each element `i`
    - One multiplication ($x_i \cdot y_i$)

OpenMP Examples

Linear algebra

Operation:

$$r = \sum_{i=0}^{n-1} x_i \cdot y_i$$

Number of *floating point operations*

- For each iteration, i.e. for each element `i`
    - One multiplication ($x_i \cdot y_i$)
    - One addition ($r = r + x_i \cdot y_i$)

OpenMP Examples

Linear algebra

Operation:

$$r = \sum_{i=0}^{n-1} x_i \cdot y_i$$

Number of *floating point operations*

- For each iteration, i.e. for each element `i`
  - One multiplication ($x_i \cdot y_i$)
  - One addition ($r = r + x_i \cdot y_i$)
- Therefore, a total of 2 floating point operations per iteration

OpenMP Examples

Linear algebra

Operation:

$$r = \sum_{i=0}^{n-1} x_i \cdot y_i$$

Number of *floating point operations*

- For each iteration, i.e. for each element `i`
  - One multiplication ($x_i \cdot y_i$)
  - One addition ($r = r + x_i \cdot y_i$)
- Therefore, a total of 2 floating point operations per iteration
- For `n` iterations, `2*n/t1/1e9` in Gflops/s

OpenMP for linear algebra

```
[ikoutsou@front02 ex03]$ for ((n=1; n<=16; n++))
> do OMP_NUM_THREADS=${n} srun -N 1 -p p100 -n 1 --cpus-per-task=64 ./axpy $((32*1024*1024))
> done
```

OpenMP for linear algebra

```
[ikoutsou@front02 ex03]$ for ((n=1; n<=16; n++))
> do OMP_NUM_THREADS=${n} srun -N 1 -p p100 -n 1 --cpus-per-task=64 ./axpy $((32*1024*1024))
> done
```

Similarly for `xdoty`

OpenMP for linear algebra

```
[ikoutsou@front02 ex03]$ for ((n=1; n⩽16; n++))
> do OMP_NUM_THREADS=${n} srun -N 1 -p p100 -n 1 --cpus-per-task=64 ./axpy $((32*1024*1024))
> done
```

Similarly for `xdoty`

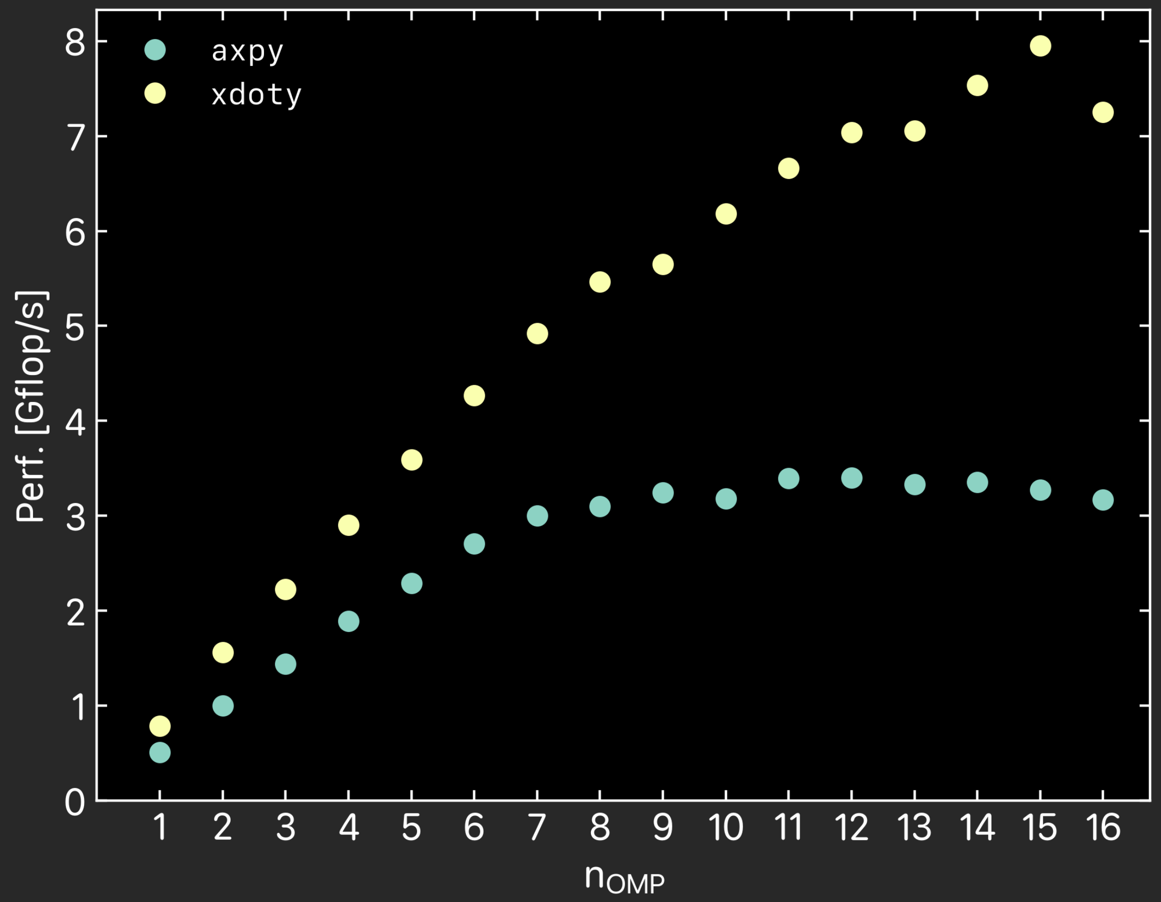
n_{OMP}	`axpy` [Gflop/s]	`xdoty` [Gflop/s]
1	0.509	0.791
2	0.999	1.564
3	1.443	2.226
4	1.890	2.907
5	2.294	3.594
6	2.705	4.268
7	3.000	4.922
8	3.102	5.465
9	3.247	5.651
10	3.179	6.183
11	3.396	6.660
12	3.404	7.039
13	3.334	7.057
14	3.353	7.534
15	3.276	7.953
16	3.169	7.253

OpenMP for linear algebra

```
[ikoutsou@front02 ex03]$ for ((n=1; n≤16; n++))
> do OMP_NUM_THREADS=${n} srun -N 1 -p p100 -n 1 --cpus-per-task=64 ./axpy $((32*1024*1024))
> done
```

Similarly for `xdoty`

n_{OMP}	`axpy` [Gflop/s]	`xdoty` [Gflop/s]
1	0.509	0.791
2	0.999	1.564
3	1.443	2.226
4	1.890	2.907
5	2.294	3.594
6	2.705	4.268
7	3.000	4.922
8	3.102	5.465
9	3.247	5.651
10	3.179	6.183
11	3.396	6.660
12	3.404	7.039
13	3.334	7.057
14	3.353	7.534
15	3.276	7.953
16	3.169	7.253



Performance Analysis

Performance analysis

Understand performance and hardware to evaluate the potential performance of your code

- Performance analysis: analysis of the performance of a code; comparison to what is expected
- Identification of bottlenecks
- Identification of improvements that can be made

Performance analysis

Understand performance and hardware to evaluate the potential performance of your code

- Performance analysis: analysis of the performance of a code; comparison to what is expected
- Identification of bottlenecks
- Identification of improvements that can be made

Hardware characteristics

Performance analysis

Understand performance and hardware to evaluate the potential performance of your code

- Performance analysis: analysis of the performance of a code; comparison to what is expected
- Identification of bottlenecks
- Identification of improvements that can be made

Hardware characteristics

Peak floating point rate

- The theoretical, highest number of floating point operations that can be carried out by a computational unit
- Depends on: clock rate, vector length, FPUs per core, cores per socket

Performance analysis

Understand performance and hardware to evaluate the potential performance of your code

- Performance analysis: analysis of the performance of a code; comparison to what is expected
- Identification of bottlenecks
- Identification of improvements that can be made

Hardware characteristics

Peak floating point rate

- The theoretical, highest number of floating point operations that can be carried out by a computational unit
- Depends on: clock rate, vector length, FPUs per core, cores per socket

Peak bandwidth

- The theoretical, highest number of bytes that can be read/written from/to some level of memory (L1,2,3 cache, RAM, etc.)
- For RAM: data rate, channels, ranks, banks

Performance analysis

Understand performance and hardware to evaluate the potential performance of your code

- Performance analysis: analysis of the performance of a code; comparison to what is expected
- Identification of bottlenecks
- Identification of improvements that can be made

Hardware characteristics

Peak floating point rate

- The theoretical, highest number of floating point operations that can be carried out by a computational unit
- Depends on: clock rate, vector length, FPUs per core, cores per socket

Peak bandwidth

- The theoretical, highest number of bytes that can be read/written from/to some level of memory (L1,2,3 cache, RAM, etc.)
- For RAM: data rate, channels, ranks, banks

When optimizing, it is important to have these numbers in mind for the machine you're running on

`P100` GPU nodes

On Linux, you can obtain processor details via `cat /proc/cpuinfo`:

```
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 85
model name      : Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz
 ...
cpu MHz         : 2100.000
cache size      : 22528 KB
physical id     : 0
siblings        : 32
core id         : 0
cpu cores       : 16
 ...
flags           : fpu vme de pse tsc msr pae mce cx8 apic  ...
```

Look up the processor on https://ark.intel.com

Peak performance

- Peak Floating Point rate
    - 16-cores per socket, 2 sockets per node, clock: 2.1 GHz
    - Best case: two 512-bit multiply-and-add per cycle (AVX-512 FMA)
    - In double precision: $2 \times$ (8 `mul` + 8 `add`) per cycle = 32 flop/cycle
    - Therefore: $2.1\text{x}10^9$ cycles/s $\times$ 32 flop/cycle = 67.2 Gflop/s per core
    - 1,075.2 Gflop/s per socket
- Peak BW
    - Up to 6 channels per socket $\Rightarrow$ 128 GB/s to RAM per socket
    - Assuming all channels filled, two sockets $\Rightarrow$ 256 GB/s
      $\longrightarrow$ `p100` nodes: 4 channels per socket filled
        - Theoretical peak **80 GBytes/s** per socket
- Some (semi-)standard tools
    - On Linux, you can obtain processor details via `cat /proc/cpuinfo`.
    - You can obtain topology and memory info e.g. `hwloc`, `dmidecode` (latter requires access to `/dev/mem`)

Computational kernels

Sustained performance

- Sustained FP-rate: the measured, average number of floating point operations carried out by the kernel per unit time
    - `add`, `sub`, and `mul` count as 1 flop
    - `dev`, `sqrt`, `sin`, etc. count $\geqslant 2$ flops. Depends on architecture
    - Count number of flops in kernel and divide by runtime
    - Alternatively, or for more complex codes, use performance counters

In our examples we will see cases of kernels where the flops are countable

- Sustained BW: the measured, average bytes read/written from main memory per unit time
    - As in the case of FP-rate, count bytes needed to be read and bytes needed to be written to and from RAM and divide by run time
    - *Maximum data reuse* assumption: "local" data, once read from RAM, never needs to be re-read

Maximum data reuse

- A (hopefully) familiar example:

$$y_i = a \cdot x_i + y_i, i = 0, \ldots, L-1$$

- It's implementation in C:

```c
double y[L]; /* Or malloc() */
double x[L]; /* Or malloc() */
double a;

/*  ...  Some initialization of a, x, and y
 *  omitted here ...
 */

for(int i=0; i<L; i++) {
  y[i] = a*x[i] + y[i];
}
```

Maximum data reuse

- A (hopefully) familiar example:

$$y_i = a \cdot x_i + y_i, i = 0, \ldots, L-1$$

- It's implementation in C:

```
double y[L]; /* Or malloc() */
double x[L]; /* Or malloc() */
double a;

/*  ...  Some initialization of a, x, and y
 *  omitted here  ...
 */

for(int i=0; i<L; i++) {
  y[i] = a*x[i] + y[i];
}
```

- Number of flops
  - $N_{fp} = L \cdot 2$
- Number of bytes of I/O
  - $N_{IO} = w \cdot (3 \cdot L)$
    - Read each element of y once
    - Read each element of x once
    - Write an element of y once
  - w: word-length in bytes, e.g.
    - $w = 4$ bytes for single precision,
    - $w = 8$ bytes for double, etc.

Maximum data reuse

- A (hopefully) familiar example:

  $$y_i = a \cdot x_i + y_i, i = 0, \ldots, L-1$$

- It's implementation in C:

```c
double y[L]; /* Or malloc() */
double x[L]; /* Or malloc() */
double a;

/*  ...  Some initialization of a, x, and y
 *  omitted here ...
 */

for(int i=0; i<L; i++) {
  y[i] = a*x[i] + y[i];
}
```

How about the memory accesses of a?

- Number of flops
  - $N_{fp} = L \cdot 2$
- Number of bytes of I/O
  - $N_{IO} = w \cdot (3 \cdot L)$
    - Read each element of y once
    - Read each element of x once
    - Write an element of y once
  - w: word-length in bytes, e.g.
    - $w = 4$ bytes for single precision,
    - $w = 8$ bytes for double, etc.

Maximum data reuse

- A (hopefully) familiar example:

$$y_i = a \cdot x_i + y_i, i = 0, \ldots, L-1$$

- It's implementation in C:

```
double y[L]; /* Or malloc() */
double x[L]; /* Or malloc() */
double a;

/*  ...  Some initialization of a, x, and y
 *  omitted here ...
 */

for(int i=0; i<L; i++) {
  y[i] = a*x[i] + y[i];
}
```

- Number of flops
  - $N_{fp} = L \cdot 2$
- Number of bytes of I/O
  - $N_{IO} = w \cdot (3 \cdot L)$
    - Read each element of y once
    - Read each element of x once
    - Write an element of y once
  - w: word-length in bytes, e.g.
    - $w = 4$ bytes for single precision,
    - $w = 8$ bytes for double, etc.

How about the memory accesses of a?

- Maximum data reuse assumption: a is read only in the first iteration and remains in *cache* for all following iterations

Maximum data reuse

- Another example:

$$y_i = \sigma \cdot (x_i - x_{i-1}), \quad i = 1, \ldots, L-1$$

- It's implementation in C:

```c
double y[L]; /* Or malloc() */
double x[L]; /* Or malloc() */
double sigma;

/*  ...  Some initialization of sigma, x, and y omitted here  ...
 */

for(int i=1; i<L; i++) {
  y[i] = sigma*(x[i] - x[i-1]);
}
```

- Number of flops
  - $N_{fp} = (L-1) \cdot 2$

Maximum data reuse

- Another example:

$$y_i = \sigma \cdot (x_i - x_{i-1}), \quad i = 1, \ldots, L-1$$

- It's implementation in C:

```
double y[L]; /* Or malloc() */
double x[L]; /* Or malloc() */
double sigma;

/*  ...  Some initialization of sigma, x, and y omitted here  ...
 */

for(int i=1; i<L; i++) {
  y[i] = sigma*(x[i] - x[i-1]);
}
```

- Number of flops
  - $N_{fp} = (L-1) \cdot 2$
  - For practical purposes, when $L \gg$, this is $N_{fp} \simeq L \cdot 2$

Maximum data reuse

- Another example:

$$y_i = \sigma \cdot (x_i - x_{i-1}), \quad i = 1, \ldots, L-1$$

- It's implementation in C:

```
double y[L]; /* Or malloc() */
double x[L]; /* Or malloc() */
double sigma;

/*  ...  Some initialization of sigma, x, and y omitted here  ...
 */

for(int i=1; i<L; i++) {
  y[i] = sigma*(x[i] - x[i-1]);
}
```

- Number of flops
  - $N_{fp} = (L-1) \cdot 2$
  - For practical purposes, when $L \gg$, this is $N_{fp} \simeq L \cdot 2$
- Number of bytes of I/O
  - In each iteration, read two elements of x and write one of y

Maximum data reuse

- Another example:

$$y_i = \sigma \cdot (x_i - x_{i-1}), \quad i = 1, \ldots, L-1$$

- It's implementation in C:

```
double y[L]; /* Or malloc() */
double x[L]; /* Or malloc() */
double sigma;

/*  ...  Some initialization of sigma, x, and y omitted here ...
 */

for(int i=1; i<L; i++) {
  y[i] = sigma*(x[i] - x[i-1]);
}
```

- Number of flops
  - $N_{fp} = (L-1)\cdot 2$
  - For practical purposes, when $L \gg$, this is $N_{fp} \simeq L \cdot 2$
- Number of bytes of I/O
  - In each iteration, read two elements of x and write one of y
  - **But**, one element of x has been read during previous iteration

Maximum data reuse

- Another example:

$$y_i = \sigma \cdot (x_i - x_{i-1}), \quad i = 1, \ldots, L-1$$

- It's implementation in C:

```
double y[L]; /* Or malloc() */
double x[L]; /* Or malloc() */
double sigma;

/*  ...  Some initialization of sigma, x, and y omitted here  ...
 */

for(int i=1; i<L; i++) {
  y[i] = sigma*(x[i] - x[i-1]);
}
```

- Number of flops
  - $N_{fp} = (L-1) \cdot 2$
  - For practical purposes, when $L \gg$, this is $N_{fp} \simeq L \cdot 2$
- Number of bytes of I/O
  - In each iteration, read two elements of x and write one of y
  - **But**, one element of x has been read during previous iteration $\Rightarrow N_{IO} = w \cdot (2 \cdot (L-1) + 1) \simeq w \cdot 2 \cdot L$

Maximum data reuse

For counting N_{IO}, looking at the iteration unrolled may help to reveal data reuse

```
y[  1] = sigma*(x[  1] - x[  0]);
y[  2] = sigma*(x[  2] - x[  1]);
y[  3] = sigma*(x[  3] - x[  2]);
 ...
y[L-1] = sigma*(x[L-1] - x[L-2]);
```

`sigma` and elements of `x[]` are required multiple times. However we assume they only need to be read once.

Maximum data reuse

For counting N_{IO}, looking at the iteration unrolled may help to reveal data reuse

```
y[  1] = sigma*(x[  1] - x[  0]);
y[  2] = sigma*(x[  2] - x[  1]);
y[  3] = sigma*(x[  3] - x[  2]);
 ...
y[L-1] = sigma*(x[L-1] - x[L-2]);
```

`sigma` and elements of `x[]` are required multiple times. However we assume they only need to be read once.

- Given some measurement of the run-time $\bar{T}$
  - FP-rate: $\beta_{fp} = \frac{N_{fp}}{\bar{T}}$
  - IO-rate: $\beta_{IO} = \frac{N_{IO}}{\bar{T}}$

Maximum data reuse

For counting N_{IO}, looking at the iteration unrolled may help to reveal data reuse

```
y[  1] = sigma*(x[  1] - x[  0]);
y[  2] = sigma*(x[  2] - x[  1]);
y[  3] = sigma*(x[  3] - x[  2]);
 ...
y[L-1] = sigma*(x[L-1] - x[L-2]);
```

`sigma` and elements of `x[]` are required multiple times. However we assume they only need to be read once.

- Given some measurement of the run-time $\bar{T}$
  - FP-rate: $\beta_{fp} = \frac{N_{fp}}{\bar{T}}$
  - IO-rate: $\beta_{IO} = \frac{N_{IO}}{\bar{T}}$
- This motivates defining their ratio, which is referred to as *intensity*

$$I = \frac{N_{fp}}{N_{IO}}$$

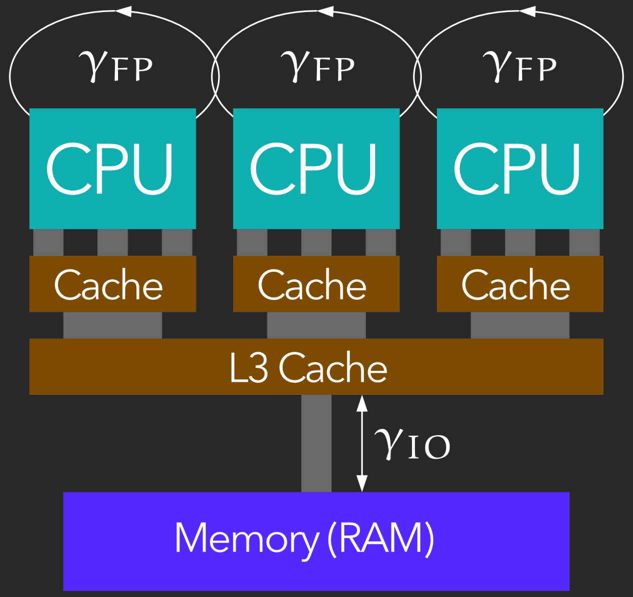
Intensities

Computational kernel intensity

- Ratio of kernel floating point operations to bytes of I/O
- For our previous example:
  - $\mathrm{I_k} = \frac{\mathrm{N_{fp}}}{\mathrm{N_{IO}}} = \frac{2\mathrm{L}}{2w\mathrm{L}} = 1/w$
  - Note how the problem size L drops out $\Rightarrow$ constant $\mathrm{I_k}$ irrespective of problem size
  - E.g. for double precision ($w = 8$), $\mathrm{I_k}$=0.125 flops/byte

Machine flop/byte ratio

- Similarly to $\mathrm{I_k}$, we can define the *machine flop/byte ratio* ($\mathrm{I_m}$)
- $\mathrm{I_m} = \frac{\gamma_{\mathrm{fp}}}{\gamma_{\mathrm{IO}}}$
- E.g. for the nodes of the `p100` partition: $\mathrm{I_m} = 8.4 \frac{\mathrm{flop}}{\mathrm{byte}}$



Intensities

Balance between kernel / hardware intensities

- $I_k \gg I_m$: Kernel is "compute-bound" on this architecture. Higher γ_{fp} would lead to higher performance, but higher γ_{IO} would not necessarily.
- $I_k \ll I_m$: Kernel is "bandwidth-" or "memory-bound" on this architecture. Higher γ_{IO} would lead to higher performance, but higher γ_{fp} would not necessarily.
- $I_k \simeq I_m$: Kernel is balanced on this architecture. Ideal situation.

For the example we have been studying

- $I_k \ll I_m \Rightarrow$ the kernel is memory-bound

Note the assumptions that enter I_k and I_m

- γ_{fp} considers all operations can be a sequence of multiply-and-add
- β_{IO} assumes maximum data reuse
- I_k constant if problem size L drops out

Kernel computational intensity

Another example: matrix-matrix multiplication

- Consider a matrix-matrix multiplication: $C_{M \times K} = A_{M \times N} \cdot B_{N \times K}$

```
double C[M][K];
double A[M][N];
double B[N][K];

for(int m=0; m<M; m++) {
  for(int k=0; k<K; k++) {
    C[m][k] = 0;
    for(int n=0; n<N; n++) {
      C[m][k] += A[m][n]*B[n][k];
    }
  }
}
```

Kernel computational intensity

Another example: matrix-matrix multiplication

- Consider a matrix-matrix multiplication: $C_{M\times K} = A_{M\times N} \cdot B_{N\times K}$

```
double C[M][K];
double A[M][N];
double B[N][K];

for(int m=0; m<M; m++) {
  for(int k=0; k<K; k++) {
    C[m][k] = 0;
    for(int n=0; n<N; n++) {
      C[m][k] += A[m][n]*B[n][k];
    }
  }
}
```

- $N_{fp} = 2 \cdot M \cdot K \cdot N$
- $N_{IO} = w \cdot (M \cdot K + M \cdot N + N \cdot K)$
- $I_k = \frac{2}{w} \frac{1}{\frac{1}{M} + \frac{1}{N} + \frac{1}{K}}$
- E.g. for a square problem $M = N = K, \Rightarrow I_k = \frac{2}{3}\frac{N}{w}$

Kernel computational intensity

Another example: matrix-matrix multiplication

- Consider a matrix-matrix multiplication: $C_{M\times K} = A_{M\times N} \cdot B_{N\times K}$

```
double C[M][K];
double A[M][N];
double B[N][K];

for(int m=0; m<M; m++) {
  for(int k=0; k<K; k++) {
    C[m][k] = 0;
    for(int n=0; n<N; n++) {
      C[m][k] += A[m][n]*B[n][k];
    }
  }
}
```

- $N_{fp} = 2 \cdot M \cdot K \cdot N$
- $N_{IO} = w \cdot (M \cdot K + M \cdot N + N \cdot K)$
- $I_k = \frac{2}{w} \frac{1}{\frac{1}{M} + \frac{1}{N} + \frac{1}{K}}$
- E.g. for a square problem $M = N = K, \Rightarrow I_k = \frac{2}{3} \frac{N}{w}$

- This is an example of a kernel where I_k depends on the problem size.
- On a given architecture with I_m, the kernel transitions from bandwidth-bound to compute-bound as N increases.
- For double precision (w = 8 bytes), the kernel is balanced when $N \simeq 100$.

Optimization

Given a code you wish to optimize, for an architecture with I_m

- What is N_{fp} and N_{IO} and what is the resulting I_k?
- Is the kernel memory or compute bound on this architecture?
- What do you obtain for β_{fp} and β_{IO}
  - For this one requires measuring the performance on the targeted architecture
- What are the ratios $\frac{\beta_{fp}}{\gamma_{fp}}$ and $\frac{\beta_{IO}}{\gamma_{IO}}$?

These are questions you need to answer before considering optimization

- After answering the above, we can start considering targeted optimizations for our kernel on the given machine
  - If your kernel is memory-bound, we should be trying to optimize for memory I/O. Ideally we try to achieve a $\frac{\beta_{IO}}{\gamma_{IO}} \longrightarrow 1$.
  - If your kernel is compute-bound, we should be trying to optimize for a higher FP-rate. Ideally we try to achieve a $\frac{\beta_{fp}}{\gamma_{fp}} \longrightarrow 1$.

Performance analysis

Back to our linear algebra kernels

`axpy`:

$$y_i \leftarrow a \cdot x_i + y_i$$

`xdoty`:

$$r = \sum_i x_i \cdot y_i$$

Performance analysis

Back to our linear algebra kernels

`axpy`:

$$y_i \leftarrow a \cdot x_i + y_i$$

`xdoty`:

$$r = \sum_i x_i \cdot y_i$$

- Run again for 1 to 16 OpenMP threads

Performance analysis

Back to our linear algebra kernels

`axpy`:

$$y_i \leftarrow a \cdot x_i + y_i$$

`xdoty`:

$$r = \sum_i x_i \cdot y_i$$

- Run again for 1 to 16 OpenMP threads
- But now, we care about understanding performance; first of all, enable additional compiler optimizations with `-O3`:

```
[ikoutsou@front02 l04]$ cc -O3 -fopenmp -o axpy axpy.c
```

Performance analysis

Back to our linear algebra kernels

`axpy`:

$$y_i \leftarrow a \cdot x_i + y_i$$

`xdoty`:

$$r = \sum_i x_i \cdot y_i$$

- Run again for 1 to 16 OpenMP threads
- But now, we care about understanding performance; first of all, enable additional compiler optimizations with `-O3`:

```
[ikoutsou@front02 l04]$ cc -O3 -fopenmp -o axpy axpy.c
```

- We would also like to ensure we're running each OpenMP thread on a *physical core*

Performance analysis

Back to our linear algebra kernels

`axpy`:

$$y_i \leftarrow a \cdot x_i + y_i$$

`xdoty`:

$$r = \sum_i x_i \cdot y_i$$

- Run again for 1 to 16 OpenMP threads
- But now, we care about understanding performance; first of all, enable additional compiler optimizations with `-O3`:

  ```
  [ikoutsou@front02 l04]$ cc -O3 -fopenmp -o axpy axpy.c
  ```

- We would also like to ensure we're running each OpenMP thread on a *physical core*

This needs to be ensured at two levels:

- Slurm's `salloc` or `srun`
- With the OpenMP runtime

Performance analysis

Back to our linear algebra kernels

`axpy`:

$$y_i \leftarrow a \cdot x_i + y_i$$

`xdoty`:

$$r = \sum_i x_i \cdot y_i$$

- Run again for 1 to 16 OpenMP threads
- But now, we care about understanding performance; first of all, enable additional compiler optimizations with `-O3`:

```
[ikoutsou@front02 l04]$ cc -O3 -fopenmp -o axpy axpy.c
```

- We would also like to ensure we're running each OpenMP thread on a *physical core*

This needs to be ensured at two levels:

- Slurm's `salloc` or `srun`
- With the OpenMP runtime

This thread placement is also referred to as the *OpenMP thread affinity*

Thread/process affinity

Affinity at the Slurm level

- Slurm sets defaults, but these are usually configured by the system administrators and can therefore be different on each system or even every Slurm partition
  - For example, `-n 16`, depending on this setup, may provide us with 16 cores (one CPU) or with 8 cores and two processes per core
  - Asking for 64 tasks (32 cores, 2 threads per core), ensures we have allocated all CPU resources of a node
  - But, running with `OMP_NUM_THREADS=16` may do one of several things, e.g.:
    - Place 8 OpenMP threads on 8 cores of the first CPU and 8 threads on the 8 cores of the second CPU
    - Place 16 OpenMP threads on the 8 cores of one of the two CPUs, thus "oversubscribing" each core by $\times 2$
    - Place 16 OpenMP threads on the 16 cores of one of the two CPUs

Thread/process affinity

Affinity at the Slurm level

- Slurm sets defaults, but these are usually configured by the system administrators and can therefore be different on each system or even every Slurm partition
    - For example, `-n 16`, depending on this setup, may provide us with 16 cores (one CPU) or with 8 cores and two processes per core
    - Asking for 64 tasks (32 cores, 2 threads per core), ensures we have allocated all CPU resources of a node
    - But, running with `OMP_NUM_THREADS=16` may do one of several things, e.g.:
        - Place 8 OpenMP threads on 8 cores of the first CPU and 8 threads on the 8 cores of the second CPU
        - Place 16 OpenMP threads on the 8 cores of one of the two CPUs, thus "oversubscribing" each core by $\times 2$
        - Place 16 OpenMP threads on the 16 cores of one of the two CPUs
    - You should be in position to:
        - Specify the desired placement of OpenMP threads to cores
        - Verify that the actual placement is indeed the desired placement

Thread/process affinity

Getting CPU info

- You can check which cores have been allocated and their placement with the `hwloc` tool

```
[ikoutsou@front02 ~]$ salloc -N 1 -n 1 --cpus-per-task=6 -p p100
[ikoutsou@cyc01 ~]$ module load gompi
[ikoutsou@cyc01 ~]$ hwloc-ls|more
[ikoutsou@cyc01 ex04]$ hwloc-ls|more
Machine (123GB total)
  Package L#0
    NUMANode L#0 (P#0 63GB)
    L3 L#0 (22MB)
      L2 L#0 (1024KB) + L1d L#0 (32KB) + L1i L#0 (32KB) + Core L#0
        PU L#0 (P#0)
        PU L#1 (P#32)
      L2 L#1 (1024KB) + L1d L#1 (32KB) + L1i L#1 (32KB) + Core L#1
        PU L#2 (P#1)
        PU L#3 (P#33)
      L2 L#2 (1024KB) + L1d L#2 (32KB) + L1i L#2 (32KB) + Core L#2
        PU L#4 (P#2)
        PU L#5 (P#34)
    HostBridge
     ...
```

- A `NUMANode` is one CPU socket. Each `L#` is a core.
- Slurm gave us 3 cores on the same CPU

Thread/process affinity

Getting CPU info

- We will reserve all cores on the node with `--cpus-per-task=64` and determine the placement via OpenMP

Thread/process affinity

Getting CPU info

- We will reserve all cores on the node with `--cpus-per-task=64` and determine the placement via OpenMP

```
[ikoutsou@front02 ~]$ salloc -N 1 -n 1 --cpus-per-task=64 -p p100
[ikoutsou@cyc01 ex04]$ hwloc-ls|more
Machine (123GB total)
  Package L#0
    NUMANode L#0 (P#0 63GB)
    L3 L#0 (22MB)
      L2 L#0 (1024KB) + L1d L#0 (32KB) + L1i L#0 (32KB) + Core L#0
        PU L#0 (P#0)
        PU L#1 (P#32)
      L2 L#1 (1024KB) + L1d L#1 (32KB) + L1i L#1 (32KB) + Core L#1
        PU L#2 (P#1)
        PU L#3 (P#33)
  ...
  Package L#1
    NUMANode L#1 (P#1 60GB)
    L3 L#1 (22MB)
      L2 L#16 (1024KB) + L1d L#16 (32KB) + L1i L#16 (32KB) + Core L#16
        PU L#32 (P#16)
        PU L#33 (P#48)
      L2 L#17 (1024KB) + L1d L#17 (32KB) + L1i L#17 (32KB) + Core L#17
        PU L#34 (P#17)
        PU L#35 (P#49)
```

- You should see two `NUMANode`, 16 cores for each, and each core has two `PU`s

Thread/process affinity

Affinity at the OpenMP level

- Two environment variables:
  - `OMP_PLACES`: Specify what a "slot" is for OpenMP. Options are `sockets`, `cores`, `threads`
    - You can also specify *which* slots to use for OpenMP. I.e., which cores, which threads, which sockets
  - `OMP_PROC_BIND`: Specify whether OpenMP threads are allowed to move between `PLACES`. `OMP_PROC_BIND=false` means **don't bind**, i.e. allow OpenMP threads to move. In addition to `true`, you can specify:
    - `spread`: Spread out consecutive OpenMP threads over `PLACES`
    - `close`: Keep consecutive OpenMP threads over `PLACES` close
- An additional environment variables allows us to check where the OpenMP threads are placed:
  - `OMP_DISPLAY_AFFINITY`: prints out numbers indicating the hardware threads

Thread/process affinity

Affinity at the OpenMP level

- Two environment variables:
  - `OMP_PLACES`: Specify what a "slot" is for OpenMP. Options are `sockets`, `cores`, `threads`
    - You can also specify *which* slots to use for OpenMP. I.e., which cores, which threads, which sockets
  - `OMP_PROC_BIND`: Specify whether OpenMP threads are allowed to move between `PLACES`. `OMP_PROC_BIND=false` means **don't bind**, i.e. allow OpenMP threads to move. In addition to `true`, you can specify:
    - `spread`: Spread out consecutive OpenMP threads over `PLACES`
    - `close`: Keep consecutive OpenMP threads over `PLACES` close
- An additional environment variables allows us to check where the OpenMP threads are placed:
  - `OMP_DISPLAY_AFFINITY`: prints out numbers indicating the hardware threads

Example A:

```
[ikoutsou@front02 l04]$ export OMP_DISPLAY_AFFINITY=true
[ikoutsou@front02 l04]$ export OMP_NUM_THREADS=4
[ikoutsou@front02 l04]$ srun -N 1 -n 1 --cpus-per-task=64 -p p100 ./axpy $((32*1024*1024))
level 1 thread 0x7f96eefc77c0 affinity 0-63
level 1 thread 0x7f96ae60d700 affinity 0-63
level 1 thread 0x7f96ade0c700 affinity 0-63
level 1 thread 0x7f96ad60b700 affinity 0-63
nth =  4   t0 = 0.129383 sec, t1 = 0.035683 sec, diff z norm = 0.000000e+00, perf = 1.880688 Gflop/s
```

Thread/process affinity

Affinity at the OpenMP level

- Two environment variables:
    - `OMP_PLACES`: Specify what a "slot" is for OpenMP. Options are `sockets`, `cores`, `threads`
        - You can also specify *which* slots to use for OpenMP. I.e., which cores, which threads, which sockets
    - `OMP_PROC_BIND`: Specify whether OpenMP threads are allowed to move between `PLACES`. `OMP_PROC_BIND=false` means **don't bind**, i.e. allow OpenMP threads to move. In addition to `true`, you can specify:
        - `spread`: Spread out consecutive OpenMP threads over `PLACES`
        - `close`: Keep consecutive OpenMP threads over `PLACES` close
- An additional environment variables allows us to check where the OpenMP threads are placed:
    - `OMP_DISPLAY_AFFINITY`: prints out numbers indicating the hardware threads

Example A:

```
[ikoutsou@front02 l04]$ export OMP_DISPLAY_AFFINITY=true
[ikoutsou@front02 l04]$ export OMP_NUM_THREADS=4
[ikoutsou@front02 l04]$ srun -N 1 -n 1 --cpus-per-task=64 -p p100 ./axpy $((32*1024*1024))
level 1 thread 0x7f96eefc77c0 affinity 0-63
level 1 thread 0x7f96ae60d700 affinity 0-63
level 1 thread 0x7f96ade0c700 affinity 0-63
level 1 thread 0x7f96ad60b700 affinity 0-63
nth =  4   t0 = 0.129383 sec, t1 = 0.035683 sec, diff z norm = 0.000000e+00, perf = 1.880688 Gflop/s
```

- No affinity specified, so all threads can take all slots — from 0 to 63

Thread/process affinity

Affinity at the OpenMP level

- Two environment variables:
  - `OMP_PLACES`: Specify what a "slot" is for OpenMP. Options are `sockets`, `cores`, `threads`
    - You can also specify *which* slots to use for OpenMP. I.e., which cores, which threads, which sockets
  - `OMP_PROC_BIND`: Specify whether OpenMP threads are allowed to move between `PLACES`. `OMP_PROC_BIND=false` means **don't bind**, i.e. allow OpenMP threads to move. In addition to `true`, you can specify:
    - `spread`: Spread out consecutive OpenMP threads over `PLACES`
    - `close`: Keep consecutive OpenMP threads over `PLACES` close
- An additional environment variables allows us to check where the OpenMP threads are placed:
  - `OMP_DISPLAY_AFFINITY`: prints out numbers indicating the hardware threads

Example B:

```
[ikoutsou@front02 l04]$ export OMP_PROC_BIND=true; export OMP_NUM_THREADS=4; export OMP_DISPLAY_AFFINITY=true
[ikoutsou@front02 l04]$ srun -N 1 -n 1 --cpus-per-task=64 -p p100 ./axpy $((32*1024*1024))
level 1 thread 0x7f8a6703a7c0 affinity 0
level 1 thread 0x7f8a26680700 affinity 32
level 1 thread 0x7f8a25e7f700 affinity 1
level 1 thread 0x7f8a2567e700 affinity 33
 nth =   4   t0 = 0.130506 sec, t1 = 0.061074 sec, diff z norm = 0.000000e+00, perf = 1.098812 Gflop/s
```

Thread/process affinity

Affinity at the OpenMP level

- Two environment variables:
    - `OMP_PLACES`: Specify what a "slot" is for OpenMP. Options are `sockets`, `cores`, `threads`
        - You can also specify *which* slots to use for OpenMP. I.e., which cores, which threads, which sockets
    - `OMP_PROC_BIND`: Specify whether OpenMP threads are allowed to move between `PLACES`. `OMP_PROC_BIND=false` means **don't bind**, i.e. allow OpenMP threads to move. In addition to `true`, you can specify:
        - `spread`: Spread out consecutive OpenMP threads over `PLACES`
        - `close`: Keep consecutive OpenMP threads over `PLACES` close
- An additional environment variables allows us to check where the OpenMP threads are placed:
    - `OMP_DISPLAY_AFFINITY`: prints out numbers indicating the hardware threads

Example B:

```
[ikoutsou@front02 l04]$ export OMP_PROC_BIND=true; export OMP_NUM_THREADS=4; export OMP_DISPLAY_AFFINITY=true
[ikoutsou@front02 l04]$ srun -N 1 -n 1 --cpus-per-task=64 -p p100 ./axpy $((32*1024*1024))
level 1 thread 0x7f8a6703a7c0 affinity 0
level 1 thread 0x7f8a26680700 affinity 32
level 1 thread 0x7f8a25e7f700 affinity 1
level 1 thread 0x7f8a2567e700 affinity 33
 nth =  4   t0 = 0.130506 sec, t1 = 0.061074 sec, diff z norm = 0.000000e+00, perf = 1.098812 Gflop/s
```

- OpenMP threads bound to specific hardware thread

Thread/process affinity

Affinity at the OpenMP level

- Two environment variables:
  - `OMP_PLACES`: Specify what a "slot" is for OpenMP. Options are `sockets`, `cores`, `threads`
    - You can also specify *which* slots to use for OpenMP. I.e., which cores, which threads, which sockets
  - `OMP_PROC_BIND`: Specify whether OpenMP threads are allowed to move between `PLACES`. `OMP_PROC_BIND=false` means **don't bind**, i.e. allow OpenMP threads to move. In addition to `true`, you can specify:
    - `spread`: Spread out consecutive OpenMP threads over `PLACES`
    - `close`: Keep consecutive OpenMP threads over `PLACES` close
- An additional environment variables allows us to check where the OpenMP threads are placed:
  - `OMP_DISPLAY_AFFINITY`: prints out numbers indicating the hardware threads

Example B:

```
[ikoutsou@front02 l04]$ export OMP_PROC_BIND=true; export OMP_NUM_THREADS=4; export OMP_DISPLAY_AFFINITY=true
[ikoutsou@front02 l04]$ srun -N 1 -n 1 --cpus-per-task=64 -p p100 ./axpy $((32*1024*1024))
level 1 thread 0x7f8a6703a7c0 affinity 0
level 1 thread 0x7f8a26680700 affinity 32
level 1 thread 0x7f8a25e7f700 affinity 1
level 1 thread 0x7f8a2567e700 affinity 33
 nth =  4   t0 = 0.130506 sec, t1 = 0.061074 sec, diff z norm = 0.000000e+00, perf = 1.098812 Gflop/s
```

- OpenMP threads bound to specific hardware thread
- The numbering follows `hwloc-ls`, the `P#` entry $\Rightarrow$ `0` and `32` are two hardware threads of the same core. Same for `1` and `33`

Thread/process affinity

Affinity at the OpenMP level

- Two environment variables:
  - `OMP_PLACES`: Specify what a "slot" is for OpenMP. Options are `sockets`, `cores`, `threads`
    - You can also specify *which* slots to use for OpenMP. I.e., which cores, which threads, which sockets
  - `OMP_PROC_BIND`: Specify whether OpenMP threads are allowed to move between `PLACES`. `OMP_PROC_BIND=false` means **don't bind**, i.e. allow OpenMP threads to move. In addition to `true`, you can specify:
    - `spread`: Spread out consecutive OpenMP threads over `PLACES`
    - `close`: Keep consecutive OpenMP threads over `PLACES` close
- An additional environment variables allows us to check where the OpenMP threads are placed:
  - `OMP_DISPLAY_AFFINITY`: prints out numbers indicating the hardware threads

Example C:

```
[ikoutsou@front02 l04]$ export OMP_PROC_BIND=true; export OMP_NUM_THREADS=4; export OMP_DISPLAY_AFFINITY=true
[ikoutsou@front02 l04]$ export OMP_PLACES=cores
[ikoutsou@front02 l04]$ srun -N 1 -n 1 --cpus-per-task=64 -p p100 ./axpy $((32*1024*1024))
level 1 thread 0x7fbb918d87c0 affinity 0,32
level 1 thread 0x7fbb50f1e700 affinity 1,33
level 1 thread 0x7fbb5071d700 affinity 2,34
level 1 thread 0x7fbb4ff1c700 affinity 3,35
nth =  4   t0 = 0.129853 sec, t1 = 0.035487 sec, diff z norm = 0.000000e+00, perf = 1.891086 Gflop/s
```

Thread/process affinity

Affinity at the OpenMP level

- Two environment variables:
  - `OMP_PLACES`: Specify what a "slot" is for OpenMP. Options are `sockets`, `cores`, `threads`
    - You can also specify *which* slots to use for OpenMP. I.e., which cores, which threads, which sockets
  - `OMP_PROC_BIND`: Specify whether OpenMP threads are allowed to move between `PLACES`. `OMP_PROC_BIND=false` means **don't bind**, i.e. allow OpenMP threads to move. In addition to `true`, you can specify:
    - `spread`: Spread out consecutive OpenMP threads over `PLACES`
    - `close`: Keep consecutive OpenMP threads over `PLACES` close
- An additional environment variables allows us to check where the OpenMP threads are placed:
  - `OMP_DISPLAY_AFFINITY`: prints out numbers indicating the hardware threads

Example C:

```
[ikoutsou@front02 l04]$ export OMP_PROC_BIND=true; export OMP_NUM_THREADS=4; export OMP_DISPLAY_AFFINITY=true
[ikoutsou@front02 l04]$ export OMP_PLACES=cores
[ikoutsou@front02 l04]$ srun -N 1 -n 1 --cpus-per-task=64 -p p100 ./axpy $((32*1024*1024))
level 1 thread 0x7fbb918d87c0 affinity 0,32
level 1 thread 0x7fbb50f1e700 affinity 1,33
level 1 thread 0x7fbb5071d700 affinity 2,34
level 1 thread 0x7fbb4ff1c700 affinity 3,35
nth =  4   t0 = 0.129853 sec, t1 = 0.035487 sec, diff z norm = 0.000000e+00, perf = 1.891086 Gflop/s
```

- OpenMP threads are bound to cores, but can "float" between hardware threads of the same core

Thread/process affinity

Affinity at the OpenMP level

- Two environment variables:
  - `OMP_PLACES`: Specify what a "slot" is for OpenMP. Options are `sockets`, `cores`, `threads`
    - You can also specify *which* slots to use for OpenMP. I.e., which cores, which threads, which sockets
  - `OMP_PROC_BIND`: Specify whether OpenMP threads are allowed to move between `PLACES`. `OMP_PROC_BIND=false` means **don't bind**, i.e. allow OpenMP threads to move. In addition to `true`, you can specify:
    - `spread`: Spread out consecutive OpenMP threads over `PLACES`
    - `close`: Keep consecutive OpenMP threads over `PLACES` close
- An additional environment variables allows us to check where the OpenMP threads are placed:
  - `OMP_DISPLAY_AFFINITY`: prints out numbers indicating the hardware threads

Example D:

```
[ikoutsou@front02 l04]$ export OMP_DISPLAY_AFFINITY=true; export OMP_NUM_THREADS=4
[ikoutsou@front02 l04]$ export OMP_PLACES="{0}:4:1"
[ikoutsou@front02 l04]$ srun -N 1 -n 1 --cpus-per-task=64 -p p100 ./axpy $((32*1024*1024))
level 1 thread 0x7f70941da7c0 affinity 0
level 1 thread 0x7f7053820700 affinity 1
level 1 thread 0x7f705301f700 affinity 2
level 1 thread 0x7f705281e700 affinity 3
nth =  4   t0 = 0.130987 sec, t1 = 0.036052 sec, diff z norm = 0.000000e+00, perf = 1.861447 Gflop/s
```

Thread/process affinity

Affinity at the OpenMP level

- Two environment variables:
  - `OMP_PLACES`: Specify what a "slot" is for OpenMP. Options are `sockets`, `cores`, `threads`
    - You can also specify *which* slots to use for OpenMP. I.e., which cores, which threads, which sockets
  - `OMP_PROC_BIND`: Specify whether OpenMP threads are allowed to move between `PLACES`. `OMP_PROC_BIND=false` means **don't bind**, i.e. allow OpenMP threads to move. In addition to `true`, you can specify:
    - `spread`: Spread out consecutive OpenMP threads over `PLACES`
    - `close`: Keep consecutive OpenMP threads over `PLACES` close
- An additional environment variables allows us to check where the OpenMP threads are placed:
  - `OMP_DISPLAY_AFFINITY`: prints out numbers indicating the hardware threads

Example D:

```
[ikoutsou@front02 l04]$ export OMP_DISPLAY_AFFINITY=true; export OMP_NUM_THREADS=4
[ikoutsou@front02 l04]$ export OMP_PLACES="{0}:4:1"
[ikoutsou@front02 l04]$ srun -N 1 -n 1 --cpus-per-task=64 -p p100 ./axpy $((32*1024*1024))
level 1 thread 0x7f70941da7c0 affinity 0
level 1 thread 0x7f7053820700 affinity 1
level 1 thread 0x7f705301f700 affinity 2
level 1 thread 0x7f705281e700 affinity 3
nth =  4   t0 = 0.130987 sec, t1 = 0.036052 sec, diff z norm = 0.000000e+00, perf = 1.861447 Gflop/s
```

- Explicitly specify the placement of OpenMP threads (therefore `OMP_PROC_BIND` is unnecessary)

Thread/process affinity

Affinity at the OpenMP level

- Two environment variables:
  - `OMP_PLACES`: Specify what a "slot" is for OpenMP. Options are `sockets`, `cores`, `threads`
    - You can also specify *which* slots to use for OpenMP. I.e., which cores, which threads, which sockets
  - `OMP_PROC_BIND`: Specify whether OpenMP threads are allowed to move between `PLACES`. `OMP_PROC_BIND=false` means **don't bind**, i.e. allow OpenMP threads to move. In addition to `true`, you can specify:
    - `spread`: Spread out consecutive OpenMP threads over `PLACES`
    - `close`: Keep consecutive OpenMP threads over `PLACES` close
- An additional environment variables allows us to check where the OpenMP threads are placed:
  - `OMP_DISPLAY_AFFINITY`: prints out numbers indicating the hardware threads

Example E:

```
[ikoutsou@front02 l04]$ export OMP_DISPLAY_AFFINITY=true; export OMP_NUM_THREADS=4
[ikoutsou@front02 l04]$ export OMP_PLACES="{0}:4:8"
[ikoutsou@front02 l04]$ srun -N 1 -n 1 --cpus-per-task=64 -p p100 ./axpy $((32*1024*1024))
level 1 thread 0x7fe4deb797c0 affinity 0
level 1 thread 0x7fe49e1bf700 affinity 8
level 1 thread 0x7fe49d9be700 affinity 16
level 1 thread 0x7fe49d1bd700 affinity 24
nth =  4   t0 = 0.131311 sec, t1 = 0.035607 sec, diff z norm = 0.000000e+00, perf = 1.884705 Gflop/s
```

Thread/process affinity

Affinity at the OpenMP level

- Two environment variables:
  - `OMP_PLACES`: Specify what a "slot" is for OpenMP. Options are `sockets`, `cores`, `threads`
    - You can also specify *which* slots to use for OpenMP. I.e., which cores, which threads, which sockets
  - `OMP_PROC_BIND`: Specify whether OpenMP threads are allowed to move between `PLACES`. `OMP_PROC_BIND=false` means **don't bind**, i.e. allow OpenMP threads to move. In addition to `true`, you can specify:
    - `spread`: Spread out consecutive OpenMP threads over `PLACES`
    - `close`: Keep consecutive OpenMP threads over `PLACES` close
- An additional environment variables allows us to check where the OpenMP threads are placed:
  - `OMP_DISPLAY_AFFINITY`: prints out numbers indicating the hardware threads

Example E:

```
[ikoutsou@front02 l04]$ export OMP_DISPLAY_AFFINITY=true; export OMP_NUM_THREADS=4
[ikoutsou@front02 l04]$ export OMP_PLACES="{0}:4:8"
[ikoutsou@front02 l04]$ srun -N 1 -n 1 --cpus-per-task=64 -p p100 ./axpy $((32*1024*1024))
level 1 thread 0x7fe4deb797c0 affinity 0
level 1 thread 0x7fe49e1bf700 affinity 8
level 1 thread 0x7fe49d9be700 affinity 16
level 1 thread 0x7fe49d1bd700 affinity 24
nth =  4   t0 = 0.131311 sec, t1 = 0.035607 sec, diff z norm = 0.000000e+00, perf = 1.884705 Gflop/s
```

- Two OpenMP threads on one CPU socket and two on the other

Performance analysis

- Rerun `xdoty.c` and `axpy.c`
- You can use your own from last lesson or take them from `/onyx/data/sds406f24/l04/ex01/`

Performance analysis

- Rerun `xdoty.c` and `axpy.c`
- You can use your own from last lesson or take them from `/onyx/data/sds406f24/l04/ex01/`
- A subtle difference is that `axpy` now follows: $y \leftarrow a \cdot x + y$

Performance analysis

- Rerun `xdoty.c` and `axpy.c`
- You can use your own from last lesson or take them from `/onyx/data/sds406f24/l04/ex01/`
- A subtle difference is that `axpy` now follows: $y \leftarrow a \cdot x + y$
- Run again for 1 to 16 OpenMP threads. Ensure all physical CPU cores are on **the same** CPU socket. And similarly for `xdoty`

Performance analysis

- Rerun `xdoty.c` and `axpy.c`
- You can use your own from last lesson or take them from `/onyx/data/sds406f24/l04/ex01/`
- A subtle difference is that `axpy` now follows: $y \leftarrow a \cdot x + y$
- Run again for 1 to 16 OpenMP threads. Ensure all physical CPU cores are on **the same** CPU socket. And similarly for `xdoty`

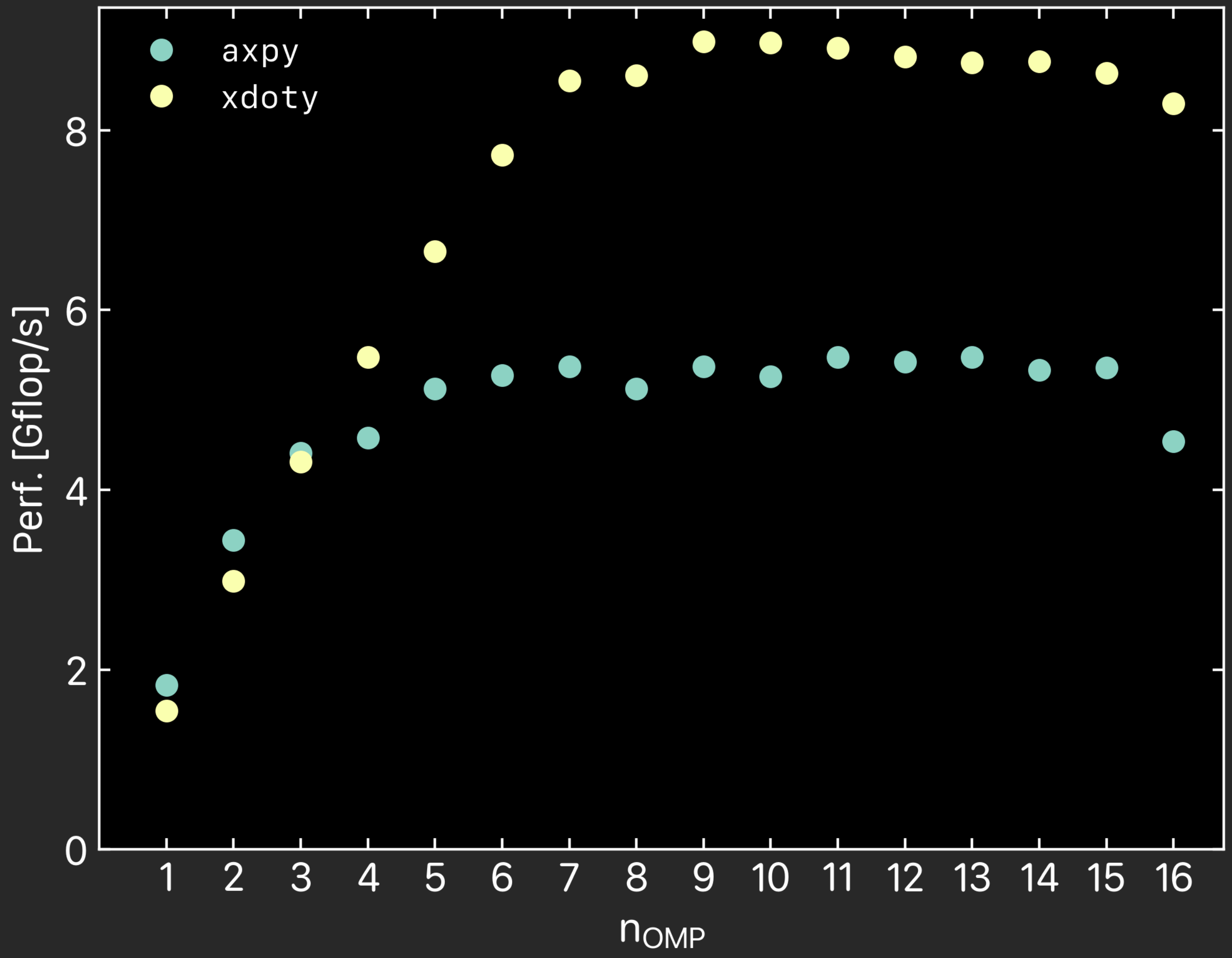
```
[ikoutsou@front02 l04]$ for((n=1; n<=16; n++))
> do
> export OMP_PLACES="{0}:$n:1"
> export OMP_NUM_THREADS=$n
> srun -N 1 -n 1 --cpus-per-task=64 -p p100 ./axpy $((32*1024*1024))
> done
nth =  1   t0 = 0.033659 sec, t1 = 0.036666 sec, diff z norm = 0.000000e+00, perf = 1.830279 Gflop/s
nth =  2   t0 = 0.033601 sec, t1 = 0.019508 sec, diff z norm = 0.000000e+00, perf = 3.440089 Gflop/s
nth =  3   t0 = 0.033593 sec, t1 = 0.015206 sec, diff z norm = 0.000000e+00, perf = 4.413355 Gflop/s
nth =  4   t0 = 0.033526 sec, t1 = 0.014653 sec, diff z norm = 0.000000e+00, perf = 4.579882 Gflop/s
nth =  5   t0 = 0.033566 sec, t1 = 0.013098 sec, diff z norm = 0.000000e+00, perf = 5.123596 Gflop/s
nth =  6   t0 = 0.033436 sec, t1 = 0.012727 sec, diff z norm = 0.000000e+00, perf = 5.272943 Gflop/s
nth =  7   t0 = 0.033536 sec, t1 = 0.012487 sec, diff z norm = 0.000000e+00, perf = 5.374327 Gflop/s
nth =  8   t0 = 0.033525 sec, t1 = 0.013092 sec, diff z norm = 0.000000e+00, perf = 5.125928 Gflop/s
nth =  9   t0 = 0.033648 sec, t1 = 0.012486 sec, diff z norm = 0.000000e+00, perf = 5.374737 Gflop/s
nth = 10   t0 = 0.033825 sec, t1 = 0.012753 sec, diff z norm = 0.000000e+00, perf = 5.262198 Gflop/s
nth = 11   t0 = 0.033825 sec, t1 = 0.012249 sec, diff z norm = 0.000000e+00, perf = 5.478725 Gflop/s
nth = 12   t0 = 0.033886 sec, t1 = 0.012366 sec, diff z norm = 0.000000e+00, perf = 5.426861 Gflop/s
nth = 13   t0 = 0.033876 sec, t1 = 0.012259 sec, diff z norm = 0.000000e+00, perf = 5.474250 Gflop/s
nth = 14   t0 = 0.033799 sec, t1 = 0.012575 sec, diff z norm = 0.000000e+00, perf = 5.336727 Gflop/s
nth = 15   t0 = 0.034108 sec, t1 = 0.012522 sec, diff z norm = 0.000000e+00, perf = 5.359182 Gflop/s
nth = 16   t0 = 0.034084 sec, t1 = 0.014791 sec, diff z norm = 0.000000e+00, perf = 4.537138 Gflop/s
```

OpenMP and controlling for affinity

n_{OMP}	`axpy` [Gflop/s]	`xdoty` [Gflop/s]
1	1.830	1.541
2	3.440	2.985
3	4.413	4.312
4	4.580	5.479
5	5.124	6.652
6	5.273	7.724
7	5.374	8.553
8	5.126	8.609
9	5.375	8.990
10	5.262	8.974
11	5.479	8.914
12	5.427	8.819
13	5.474	8.756
14	5.337	8.765
15	5.359	8.638
16	4.537	8.298

OpenMP and controlling for affinity

n_{OMP}	`axpy` [Gflop/s]	`xdoty` [Gflop/s]
1	1.830	1.541
2	3.440	2.985
3	4.413	4.312
4	4.580	5.479
5	5.124	6.652
6	5.273	7.724
7	5.374	8.553
8	5.126	8.609
9	5.375	8.990
10	5.262	8.974
11	5.479	8.914
12	5.427	8.819
13	5.474	8.756
14	5.337	8.765
15	5.359	8.638
16	4.537	8.298



Performance analysis

Back to our linear algebra kernels

`axpy`:

$$y_i \leftarrow a \cdot x_i + y_i$$

`xdoty`:

$$r = \sum_i x_i \cdot y_i$$

Performance analysis

Back to our linear algebra kernels

`axpy`:

$$y_i \leftarrow a \cdot x_i + y_i$$

`xdoty`:

$$r = \sum_i x_i \cdot y_i$$

- Take `xdoty.c` and `axpy.c` from `/onyx/data/sds406f24/l04/ex01/` and modify them to report sustained bandwidth
- Run again for 1 to 16 OpenMP threads and plot β_{IO} versus n_{OMP} for both programs on the same plot

Performance analysis

Back to our linear algebra kernels

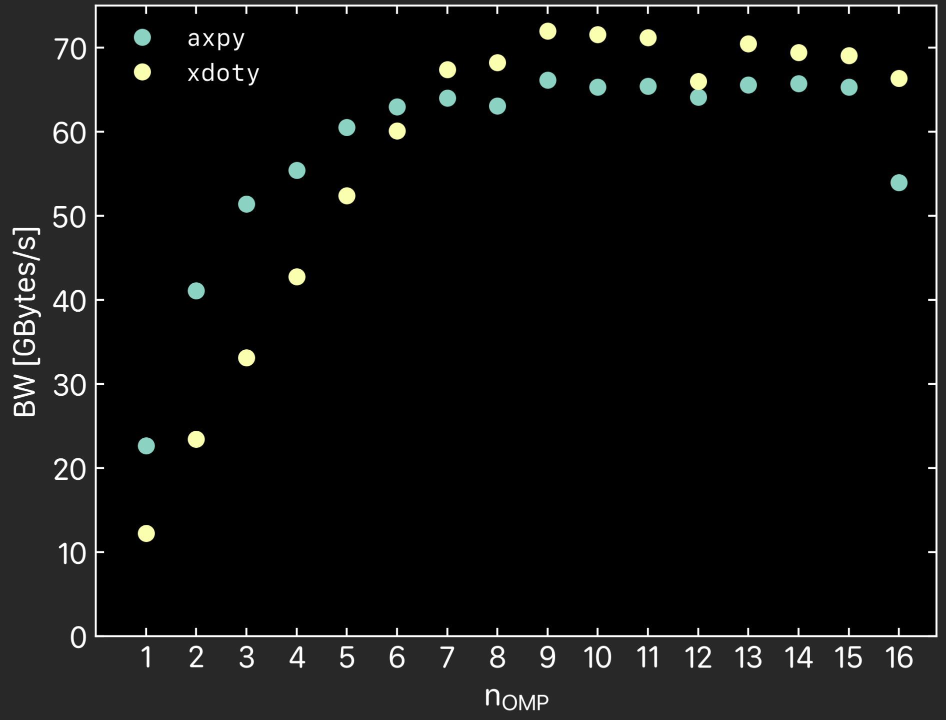
`axpy`:

$$y_i \leftarrow a \cdot x_i + y_i$$

`xdoty`:

$$r = \sum_i x_i \cdot y_i$$

- Take `xdoty.c` and `axpy.c` from `/onyx/data/sds406f24/l04/ex01/` and modify them to report sustained bandwidth
- Run again for 1 to 16 OpenMP threads and plot β_{IO} versus n_{OMP} for both programs on the same plot



Performance analysis

Back to our linear algebra kernels

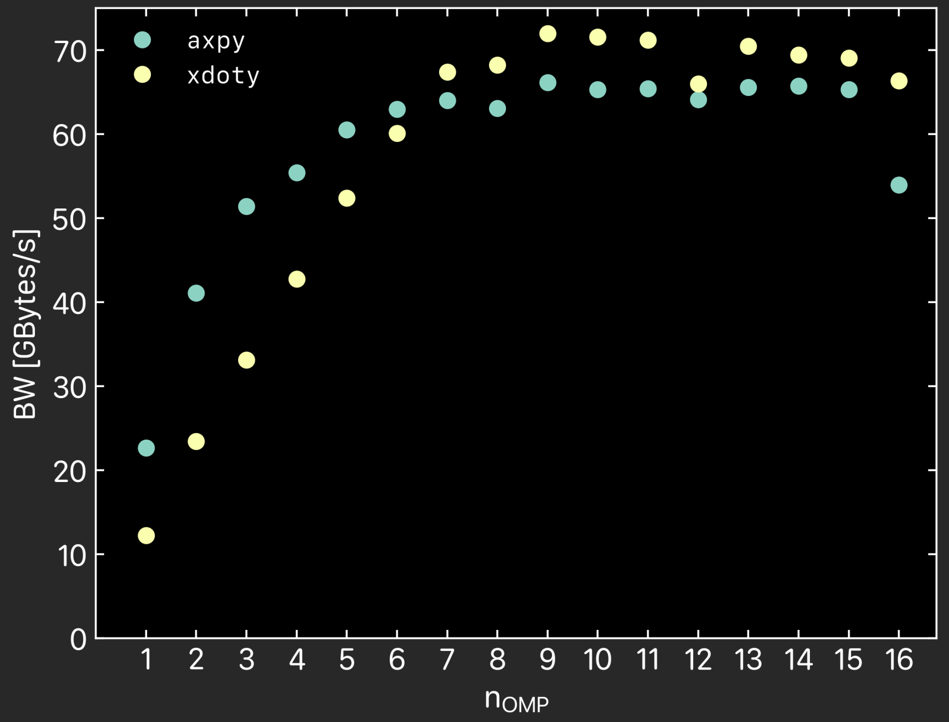
`axpy`:

$$y_i \leftarrow a \cdot x_i + y_i$$

`xdoty`:

$$r = \sum_i x_i \cdot y_i$$

- Take `xdoty.c` and `axpy.c` from `/onyx/data/sds406f24/l04/ex01/` and modify them to report sustained bandwidth
- Run again for 1 to 16 OpenMP threads and plot β_{IO} versus n_{OMP} for both programs on the same plot
- Percentage of peak?



Performance analysis

Back to our linear algebra kernels

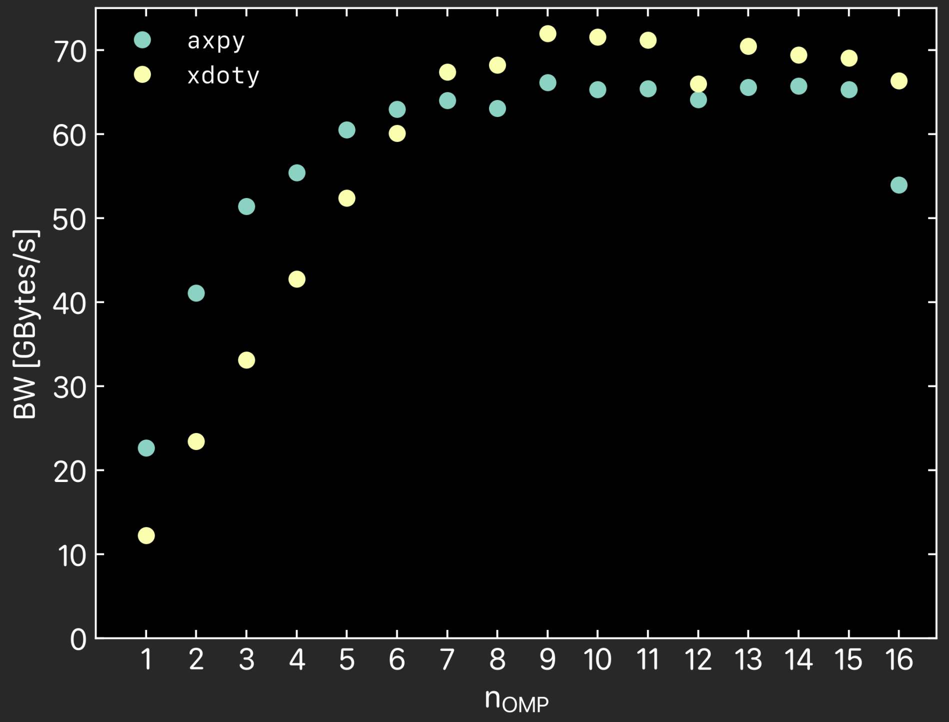
`axpy`:

$$y_i \leftarrow a \cdot x_i + y_i$$

`xdoty`:

$$r = \sum_i x_i \cdot y_i$$

- Take `xdoty.c` and `axpy.c` from `/onyx/data/sds406f24/l04/ex01/` and modify them to report sustained bandwidth
- Run again for 1 to 16 OpenMP threads and plot β_{IO} versus n_{OMP} for both programs on the same plot
- Percentage of peak?
- Compare with *per socket theoretical peak* of $\sim$80 GBytes/s



Performance analysis

Another example using `axpy()`:

$$y_i \leftarrow a \cdot x_i + y_i \quad i = 0, \ldots, L-1$$

Performance analysis

Another example using `axpy()`:

$$y_i \leftarrow a \cdot x_i + y_i \quad i = 0, \ldots, L-1$$

- See `/onyx/data/sds406f24/l04/ex02/.`
- Use `module load gompi` and compile with `cc -O3 -fopenmp -o axpy axpy.c`

Performance analysis

Another example using `axpy()`:

$$y_i \leftarrow a \cdot x_i + y_i \quad i = 0, \ldots, L-1$$

- See `/onyx/data/sds406f24/l04/ex02/.`
- Use `module load gompi` and compile with `cc -O3 -fopenmp -o axpy axpy.c`
- This version is slightly different to what we have been using so far

Performance analysis

Another example using `axpy()`:

$$y_i \leftarrow a \cdot x_i + y_i \quad i = 0, \ldots, L-1$$

- See `/onyx/data/sds406f24/l04/ex02/.`
- Use `module load gompi` and compile with `cc -O3 -fopenmp -o axpy axpy.c`
- This version is slightly different to what we have been using so far
  - Carries out `y[i] = a*x[i] + y[i]`, i.e. no `z[]` array

Performance analysis

Another example using `axpy()`:

$$y_i \leftarrow a \cdot x_i + y_i \quad i = 0, \ldots, L-1$$

- See `/onyx/data/sds406f24/l04/ex02/.`
- Use `module load gompi` and compile with `cc -O3 -fopenmp -o axpy axpy.c`
- This version is slightly different to what we have been using so far
  - Carries out `y[i] = a*x[i] + y[i]`, i.e. no `z[]` array
  - No checking of result, i.e. no comparing of `z0[]` and `z1[]`

Performance analysis

Another example using `axpy()`:

$$y_i \leftarrow a \cdot x_i + y_i \quad i = 0, \ldots, L-1$$

- See `/onyx/data/sds406f24/l04/ex02/.`
- Use `module load gompi` and compile with `cc -O3 -fopenmp -o axpy axpy.c`
- This version is slightly different to what we have been using so far
  - Carries out `y[i] = a*x[i] + y[i]`, i.e. no `z[]` array
  - No checking of result, i.e. no comparing of `z0[]` and `z1[]`
  - Does `y[:] = a*x[:] + y[:]` repeatidly until three seconds have elapsed

Performance analysis

Another example using `axpy()`:

$$y_i \leftarrow a \cdot x_i + y_i \quad i = 0, \ldots, L-1$$

- See `/onyx/data/sds406f24/l04/ex02/.`
- Use `module load gompi` and compile with `cc -O3 -fopenmp -o axpy axpy.c`
- This version is slightly different to what we have been using so far
    - Carries out `y[i] = a*x[i] + y[i]`, i.e. no `z[]` array
    - No checking of result, i.e. no comparing of `z0[]` and `z1[]`
    - Does `y[:] = a*x[:] + y[:]` repeatidly until three seconds have elapsed

```
double t0 = stop_watch(0);
int niter = 0;
for(; stop_watch(t0) < 3; niter++) {
  axpy(n, a, x, y);
}
t0 = stop_watch(t0)/niter;
```

Performance analysis

Another example using `axpy()`:

$$y_i \leftarrow a \cdot x_i + y_i \quad i = 0, \ldots, L-1$$

- See `/onyx/data/sds406f24/l04/ex02/.`
- Use `module load gompi` and compile with `cc -O3 -fopenmp -o axpy axpy.c`
- This version is slightly different to what we have been using so far
  - Carries out `y[i] = a*x[i] + y[i]`, i.e. no `z[]` array
  - No checking of result, i.e. no comparing of `z0[]` and `z1[]`
  - Does `y[:] = a*x[:] + y[:]` repeatidly until three seconds have elapsed

```
double t0 = stop_watch(0);
int niter = 0;
for(; stop_watch(t0) < 3; niter++) {
  axpy(n, a, x, y);
}
t0 = stop_watch(t0)/niter;
```

$\longrightarrow$ *Warm-up*: Add an OpenMP pragma to parallelize the loop over `n` in `axpy()`

Performance analysis

Another example using `axpy()`:

$$y_i \leftarrow a \cdot x_i + y_i \quad i = 0, \ldots, L-1$$

- See `/onyx/data/sds406f24/l04/ex02/.`
- Use `module load gompi` and compile with `cc -O3 -fopenmp -o axpy axpy.c`
- This version is slightly different to what we have been using so far
  - Carries out `y[i] = a*x[i] + y[i]`, i.e. no `z[]` array
  - No checking of result, i.e. no comparing of `z0[]` and `z1[]`
  - Does `y[:] = a*x[:] + y[:]` repeatidly until three seconds have elapsed

```
double t0 = stop_watch(0);
int niter = 0;
for(; stop_watch(t0) < 3; niter++) {
  axpy(n, a, x, y);
}
t0 = stop_watch(t0)/niter;
```

$\longrightarrow$ *Warm-up*: Add an OpenMP pragma to parallelize the loop over `n` in `axpy()`

$\longrightarrow$ *Warm-up*: Correct the calculation of β_{fp} and β_{IO}

Performance analysis

Another example using `axpy()`:

$$y_i \leftarrow a \cdot x_i + y_i \quad i = 0, \ldots, L-1$$

- See `/onyx/data/sds406f24/l04/ex02/.`
- Use `module load gompi` and compile with `cc -O3 -fopenmp -o axpy axpy.c`
- This version is slightly different to what we have been using so far
  - Carries out `y[i] = a*x[i] + y[i]`, i.e. no `z[]` array
  - No checking of result, i.e. no comparing of `z0[]` and `z1[]`
  - Does `y[:] = a*x[:] + y[:]` repeatidly until three seconds have elapsed

```
double t0 = stop_watch(0);
int niter = 0;
for(; stop_watch(t0) < 3; niter++) {
  axpy(n, a, x, y);
}
t0 = stop_watch(t0)/niter;
```

$\longrightarrow$ *Warm-up*: Add an OpenMP pragma to parallelize the loop over `n` in `axpy()`

$\longrightarrow$ *Warm-up*: Correct the calculation of $\beta_{\rm fp}$ and $\beta_{\rm IO}$

$\longrightarrow$ *Warm-up*: Compile and run once using $n = 1024^2$ and 2 OpenMP thread

Performance analysis

Another example using `axpy()`:

$$y_i \leftarrow a \cdot x_i + y_i \quad i = 0, \ldots, L-1$$

```
[ikoutsou@front02 ex02]$ sbatch run.sh
Submitted batch job 181364
[ikoutsou@front02 ex02]$ cat axpy.txt
 nth =  2   t0 = 3.678e-04 sec   L =    1048576   niter =     8158   beta_fp = 5.702e+00 Gflop/s   beta_io = 6.843e+01 GBytes/s
```

Performance analysis

Another example using `axpy()`:

$$y_i \leftarrow a \cdot x_i + y_i \quad i = 0, \ldots, L-1$$

```
[ikoutsou@front02 ex02]$ sbatch run.sh
Submitted batch job 181364
[ikoutsou@front02 ex02]$ cat axpy.txt
 nth =  2   t0 = 3.678e-04 sec   L =     1048576   niter =      8158   beta_fp = 5.702e+00 Gflop/s   beta_io = 6.843e+01 GBytes/s
```

- The program called `axpy()` 8158 times

Performance analysis

Another example using `axpy()`:

$$y_i \leftarrow a \cdot x_i + y_i \quad i = 0, \ldots, L-1$$

```
[ikoutsou@front02 ex02]$ sbatch run.sh
Submitted batch job 181364
[ikoutsou@front02 ex02]$ cat axpy.txt
 nth =   2   t0 = 3.678e-04 sec   L =     1048576   niter =      8158   beta_fp = 5.702e+00 Gflop/s   beta_io = 6.843e+01 GBytes/s
```

- The program called `axpy()` 8158 times
- `t0` is the average time for the 8158 calls

Performance analysis

Another example using `axpy()`:

$$y_i \leftarrow a \cdot x_i + y_i \quad i = 0, \ldots, L-1$$

```
[ikoutsou@front02 ex02]$ sbatch run.sh
Submitted batch job 181364
[ikoutsou@front02 ex02]$ cat axpy.txt
 nth =   2   t0 = 3.678e-04 sec   L =    1048576   niter =     8158   beta_fp = 5.702e+00 Gflop/s   beta_io = 6.843e+01 GBytes/s
```

- The program called `axpy()` 8158 times
- `t0` is the average time for the 8158 calls

Your task now

- Edit the incomplete `run.sh`

Performance analysis

Another example using `axpy()`:

$$y_i \leftarrow a \cdot x_i + y_i \quad i = 0, \ldots, L-1$$

```
[ikoutsou@front02 ex02]$ sbatch run.sh
Submitted batch job 181364
[ikoutsou@front02 ex02]$ cat axpy.txt
 nth =   2   t0 = 3.678e-04 sec   L =     1048576   niter =      8158   beta_fp = 5.702e+00 Gflop/s   beta_io = 6.843e+01 GBytes/s
```

- The program called `axpy()` 8158 times
- `t0` is the average time for the 8158 calls

Your task now

- Edit the incomplete `run.sh`
    - Loop over L for values L=128, 256, ..., 512×1024^2 (double each time)

Performance analysis

Another example using `axpy()`:

$$y_i \leftarrow a \cdot x_i + y_i \quad i = 0, \ldots, L-1$$

```
[ikoutsou@front02 ex02]$ sbatch run.sh
Submitted batch job 181364
[ikoutsou@front02 ex02]$ cat axpy.txt
 nth =   2    t0 = 3.678e-04 sec    L =     1048576    niter =      8158    beta_fp = 5.702e+00 Gflop/s    beta_io = 6.843e+01 GBytes/s
```

- The program called `axpy()` 8158 times
- `t0` is the average time for the 8158 calls

Your task now

- Edit the incomplete `run.sh`
    - Loop over L for values L=128, 256, ..., 512×1024^2 (double each time)
    - Loop over n_{OMP} for values n_{OMP}=1, 2, 4, ..., 16 (double each time)

Performance analysis

Another example using `axpy()`:

$$y_i \leftarrow a \cdot x_i + y_i \quad i = 0, \ldots, L-1$$

```
[ikoutsou@front02 ex02]$ sbatch run.sh
Submitted batch job 181364
[ikoutsou@front02 ex02]$ cat axpy.txt
 nth =   2   t0 = 3.678e-04 sec   L =    1048576   niter =     8158   beta_fp = 5.702e+00 Gflop/s   beta_io = 6.843e+01 GBytes/s
```

- The program called `axpy()` 8158 times
- `t0` is the average time for the 8158 calls

Your task now

- Edit the incomplete `run.sh`
  - Loop over L for values L=128, 256, ..., 512$\times$1024^2 (double each time)
  - Loop over $n_{\rm OMP}$ for values $n_{\rm OMP}$=1, 2, 4, ..., 16 (double each time)
- Plot $\beta_{\rm IO}$ versus L. Use different colors for different $n_{\rm OMP}$

Performance analysis

Another example using `axpy()`:

$$y_i \leftarrow a \cdot x_i + y_i \quad i = 0, \ldots, L-1$$

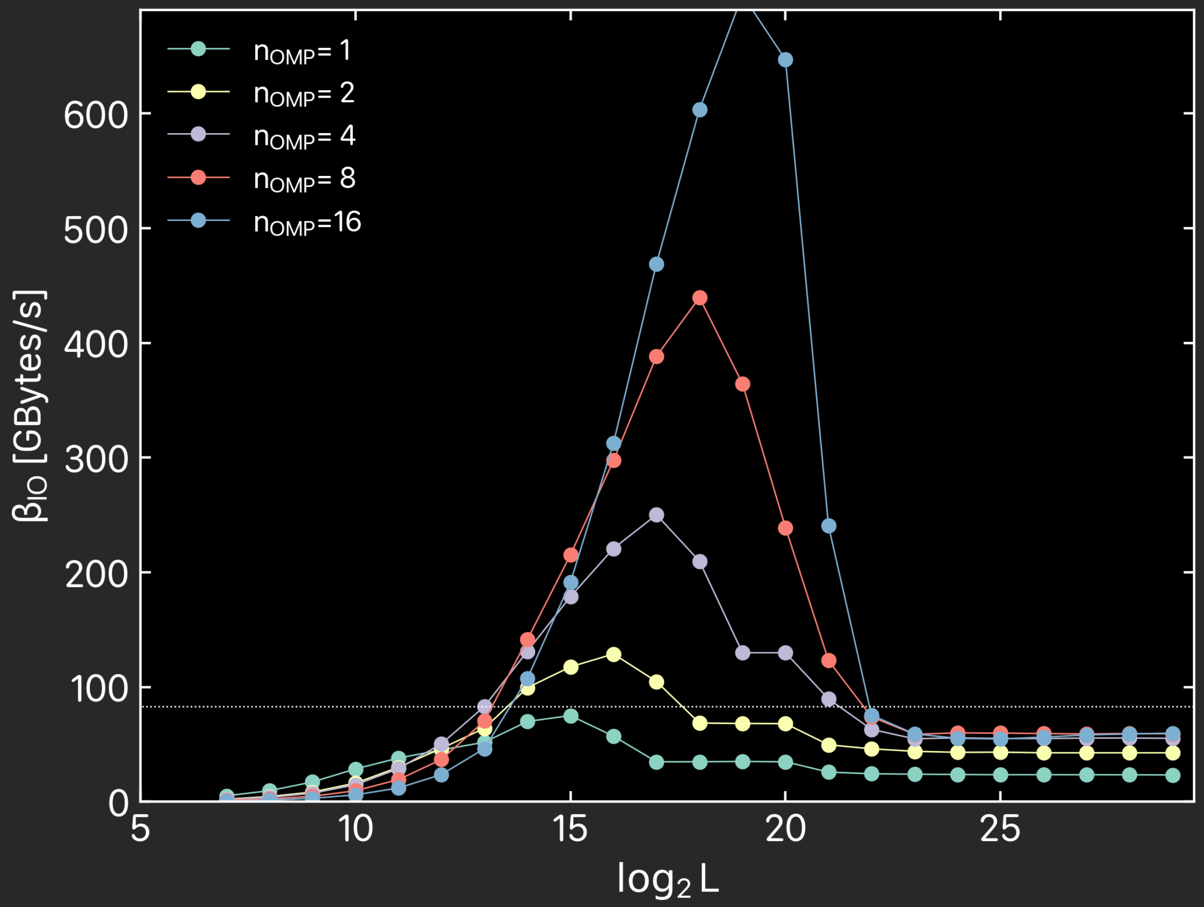
```
[ikoutsou@front02 ex02]$ sbatch run.sh
Submitted batch job 181364
[ikoutsou@front02 ex02]$ cat axpy.txt
 nth =   2    t0 = 3.678e-04 sec    L =     1048576    niter =      8158    beta_fp = 5.702e+00 Gflop/s    beta_io = 6.843e+01 GBytes/s
```

- The program called `axpy()` 8158 times
- `t0` is the average time for the 8158 calls

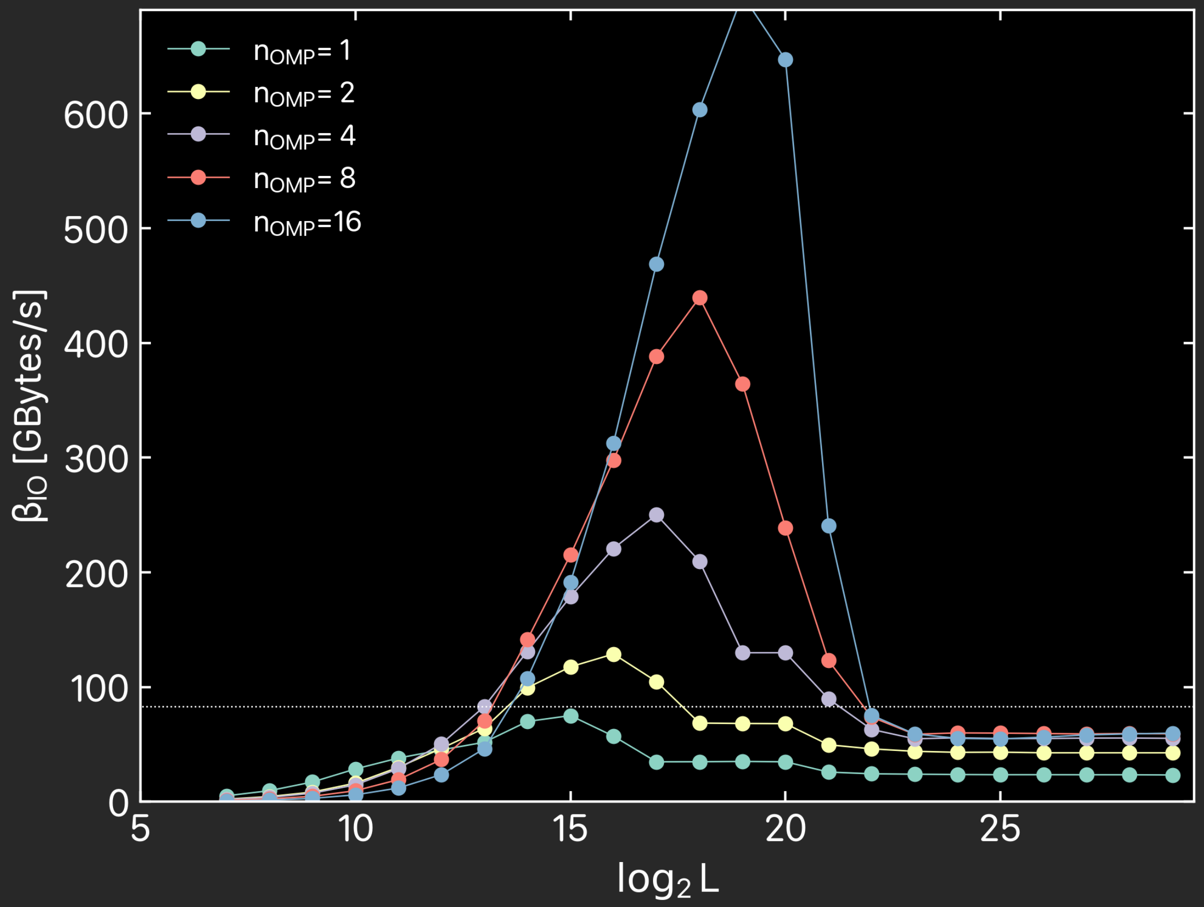
Your task now

- Edit the incomplete `run.sh`
  - Loop over L for values L=128, 256, ..., 512×1024^2 (double each time)
  - Loop over n_{OMP} for values n_{OMP}=1, 2, 4, ..., 16 (double each time)
- Plot β_{IO} versus L. Use different colors for different n_{OMP}
  $\longrightarrow$ best use log-scale on the x-axis

Performance analysis

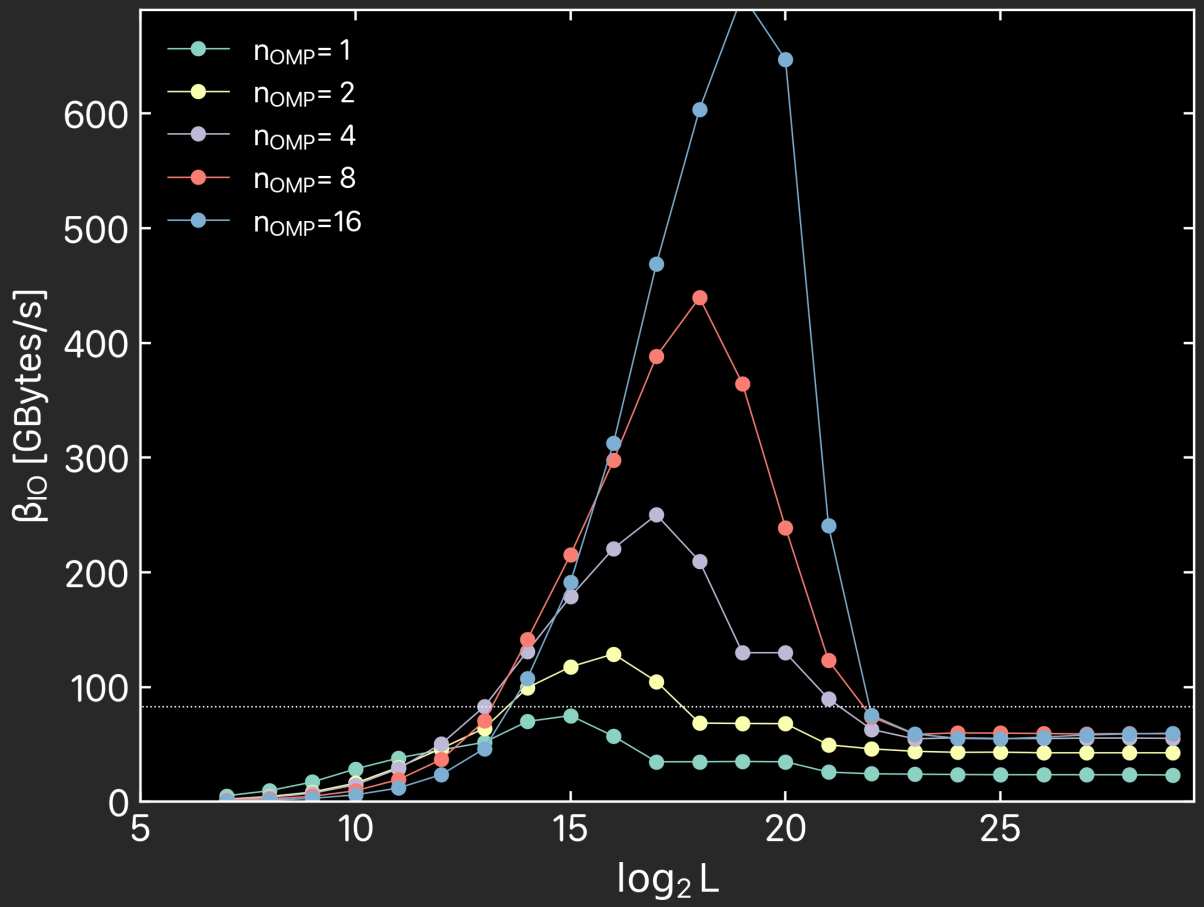


Performance analysis



- Dashed horizontal line is theoretical peak bandwidth (single socket) of 83 GBytes/s

Performance analysis

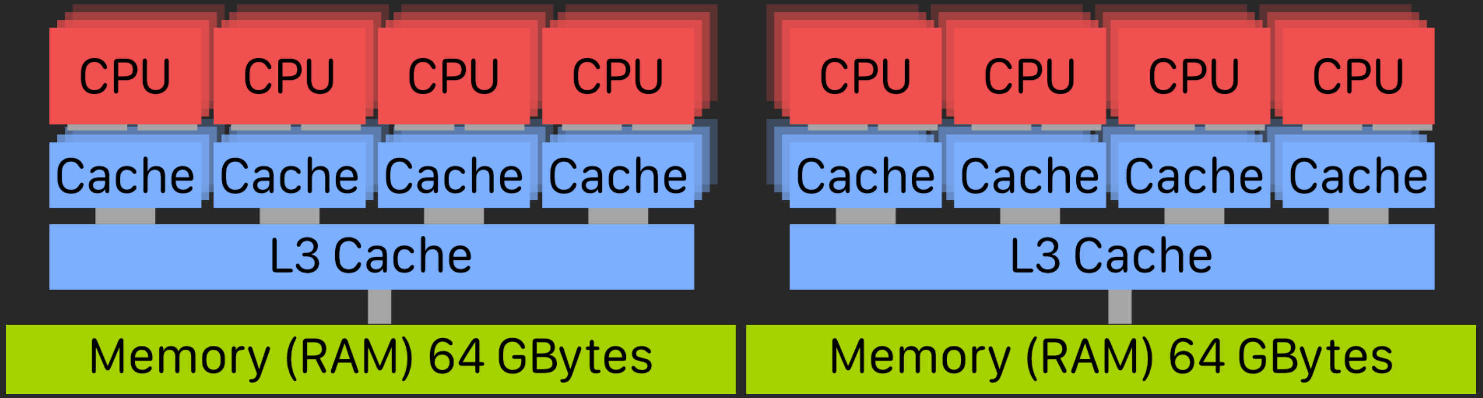


- Dashed horizontal line is theoretical peak bandwidth (single socket) of 83 GBytes/s
- What is happening between $L = 2^{15}$ and 2^{20}?

Performance analysis

`axpy`:

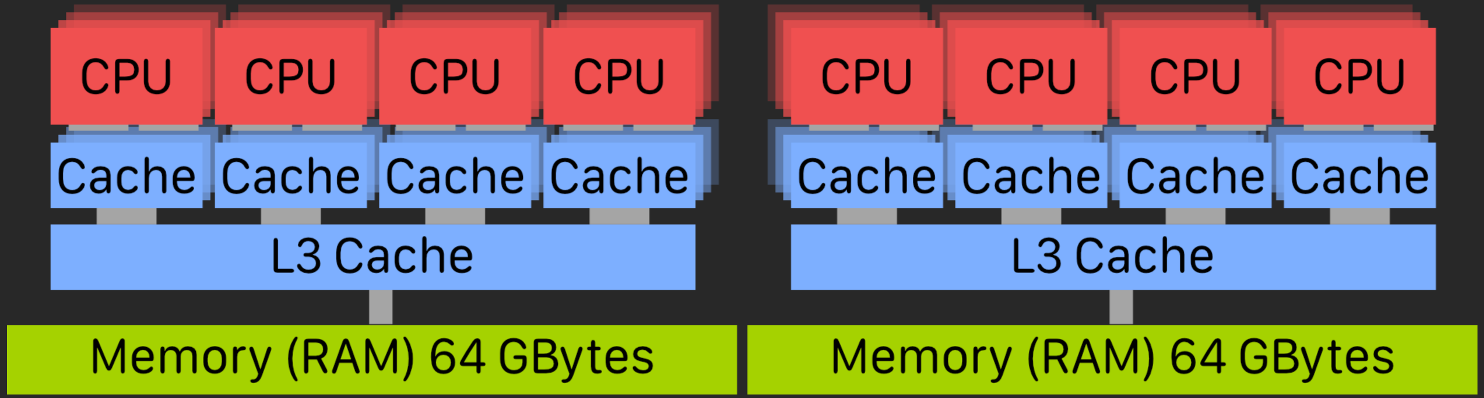
$$y_i \leftarrow a \cdot x_i + y_i \qquad i = 0, \ldots, L-1$$



Performance analysis

`axpy`:

$$y_i \leftarrow a \cdot x_i + y_i \qquad i = 0, \ldots, L-1$$

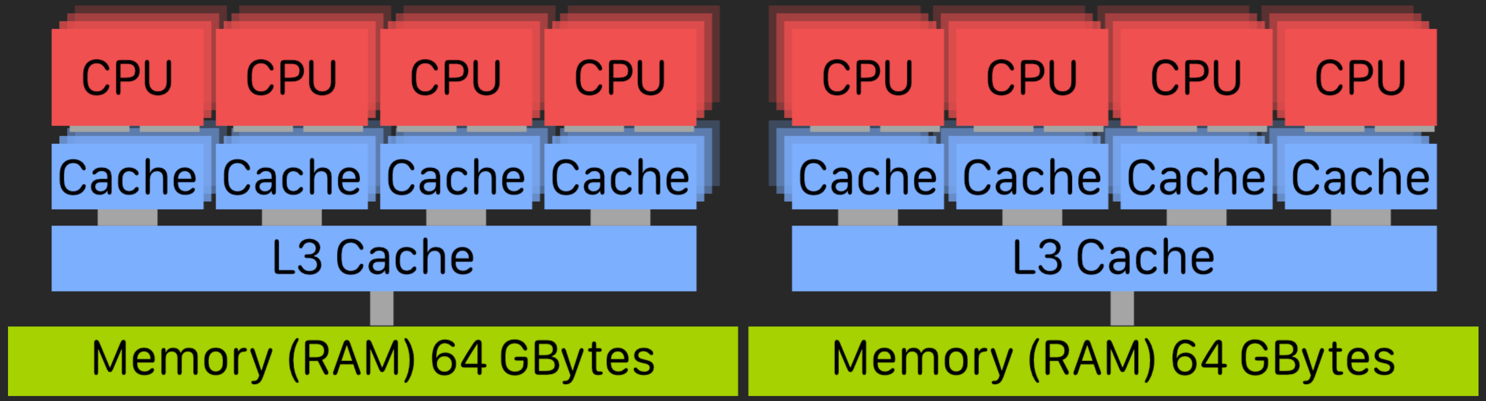


- Smaller $L \Rightarrow$ data is repeatedly read from cache $\Rightarrow$ high bandwidth $\gg \gamma_{IO}$

Performance analysis

`axpy`:

$$y_i \leftarrow a \cdot x_i + y_i \qquad i = 0, \ldots, L-1$$

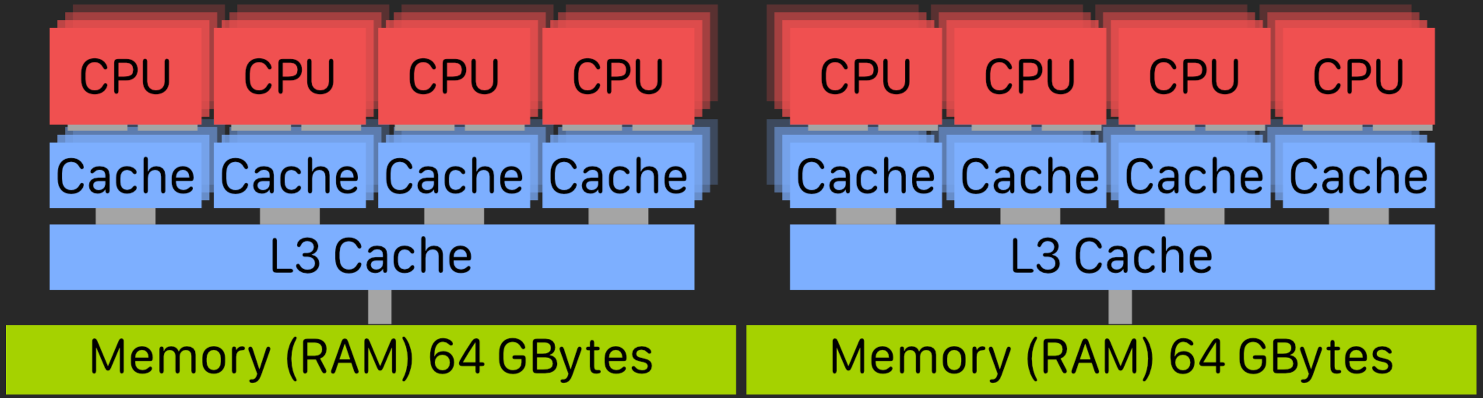


- Smaller $L \Rightarrow$ data is repeatedly read from cache $\Rightarrow$ high bandwidth $\gg \gamma_{IO}$
- Increasing the number of cores used increases the bandwidth that can be sustained

Performance analysis

`axpy`:

$$y_i \leftarrow a \cdot x_i + y_i \qquad i = 0, \ldots, L-1$$

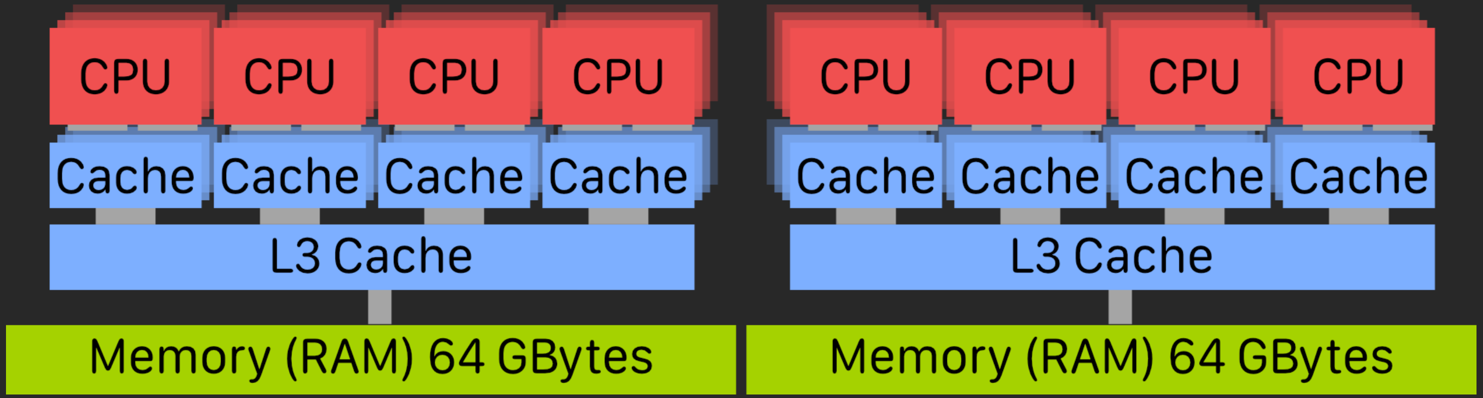


- Smaller $L \Rightarrow$ data is repeatedly read from cache $\Rightarrow$ high bandwidth $\gg \gamma_{IO}$
- Increasing the number of cores used increases the bandwidth that can be sustained
- For L that fits in a single NUMA-node RAM (i.e. $<$ 64 GBytes) $\Rightarrow$ peak bandwidth is 83 GBytes/s

Performance analysis

`axpy`:

$$y_i \leftarrow a \cdot x_i + y_i \qquad i = 0, \ldots, L-1$$

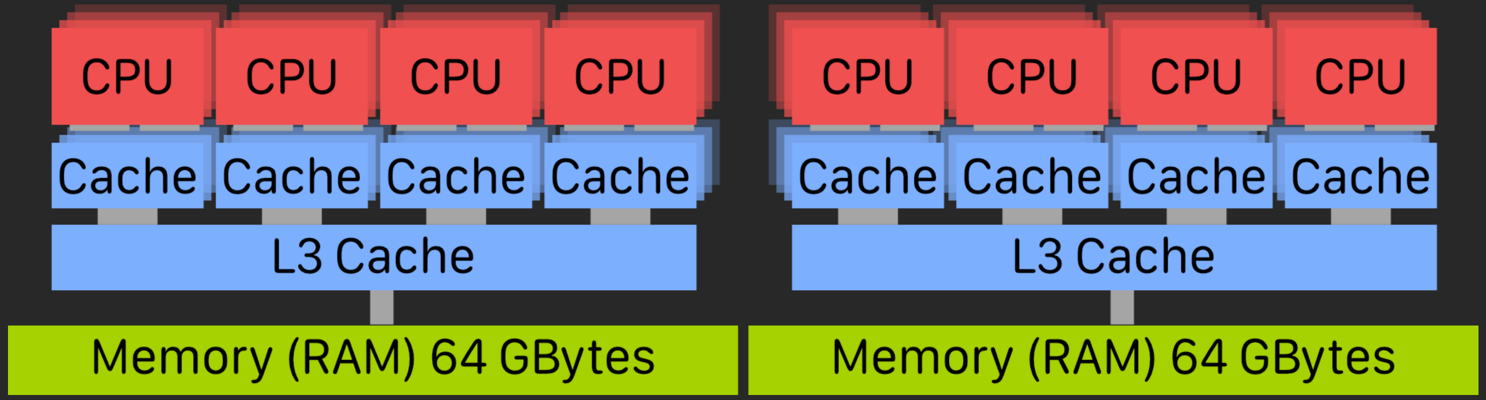


- What is the cache size?

Performance analysis

`axpy`:

$$y_i \leftarrow a \cdot x_i + y_i \qquad i = 0, \ldots, L-1$$

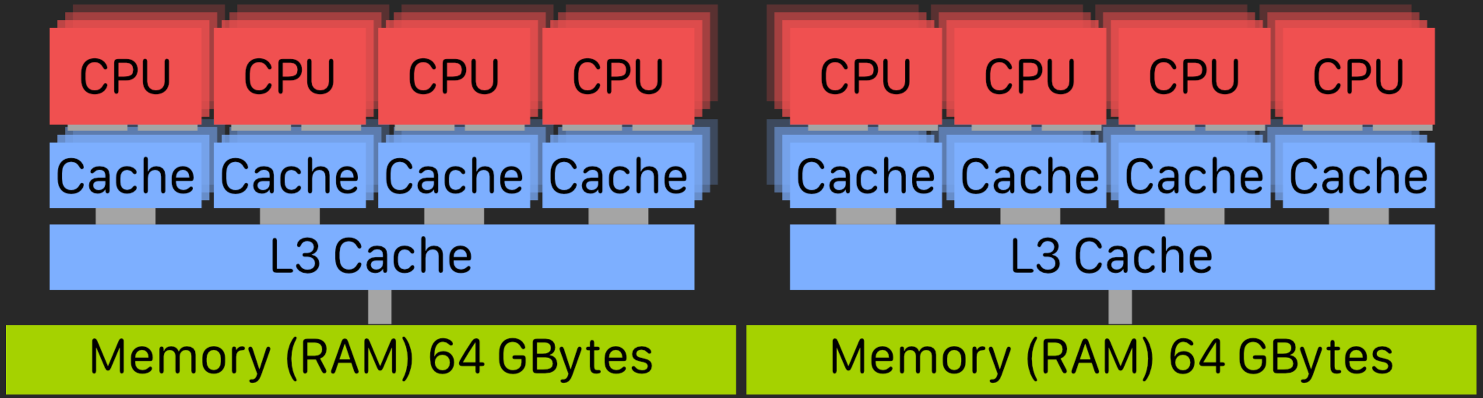


- What is the cache size?
  - 22 MBytes (for each socket)
  - See e.g. `/proc/cpuinfo` or `hwloc-ls`

Performance analysis

`axpy`:

$$y_i \leftarrow a \cdot x_i + y_i \qquad i = 0, \ldots, L-1$$

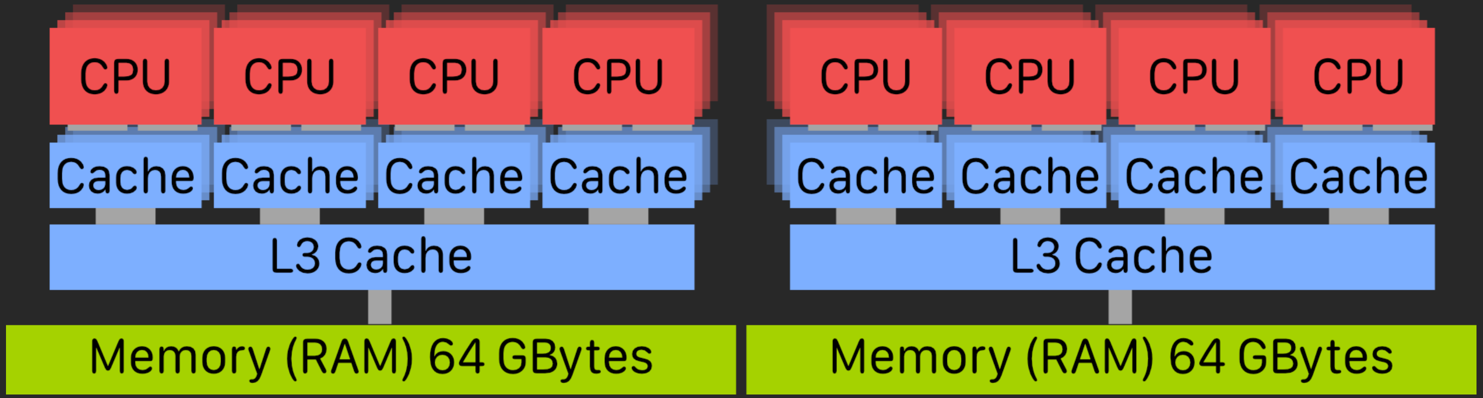


- What is the cache size?
    - 22 MBytes (for each socket)
    - See e.g. `/proc/cpuinfo` or `hwloc-ls`
- For which value of L do we reach the cache size limit

Performance analysis

`axpy`:

$$y_i \leftarrow a \cdot x_i + y_i \quad i = 0, \ldots, L-1$$

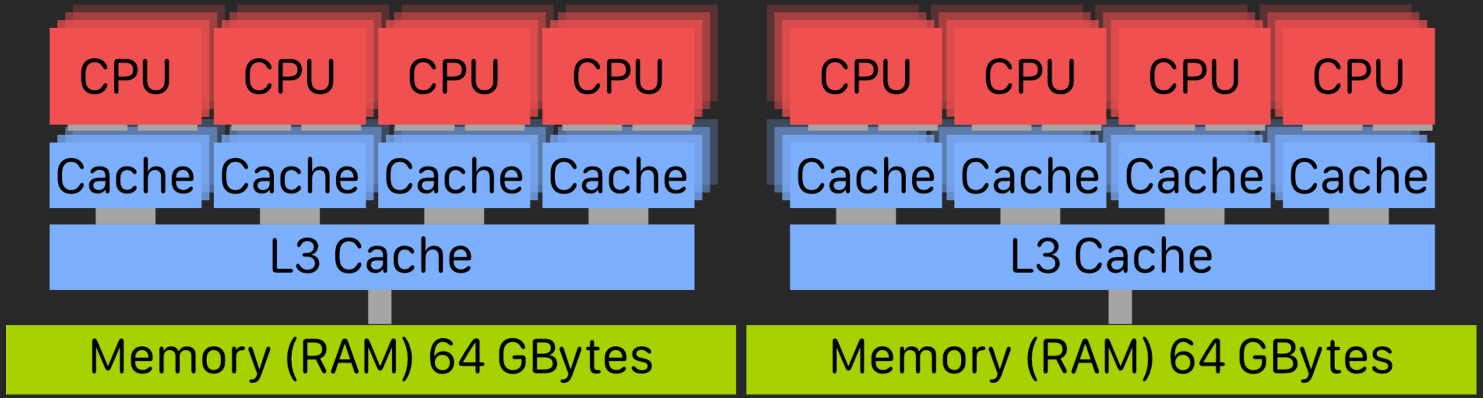


- What is the cache size?
  - 22 MBytes (for each socket)
  - See e.g. `/proc/cpuinfo` or `hwloc-ls`
- For which value of L do we reach the cache size limit
  - Storage requirements: $w \cdot 2 \cdot L$ (one `x[]` and one `y[]`)

Performance analysis

`axpy`:

$$y_i \leftarrow a \cdot x_i + y_i \quad i = 0, \ldots, L-1$$

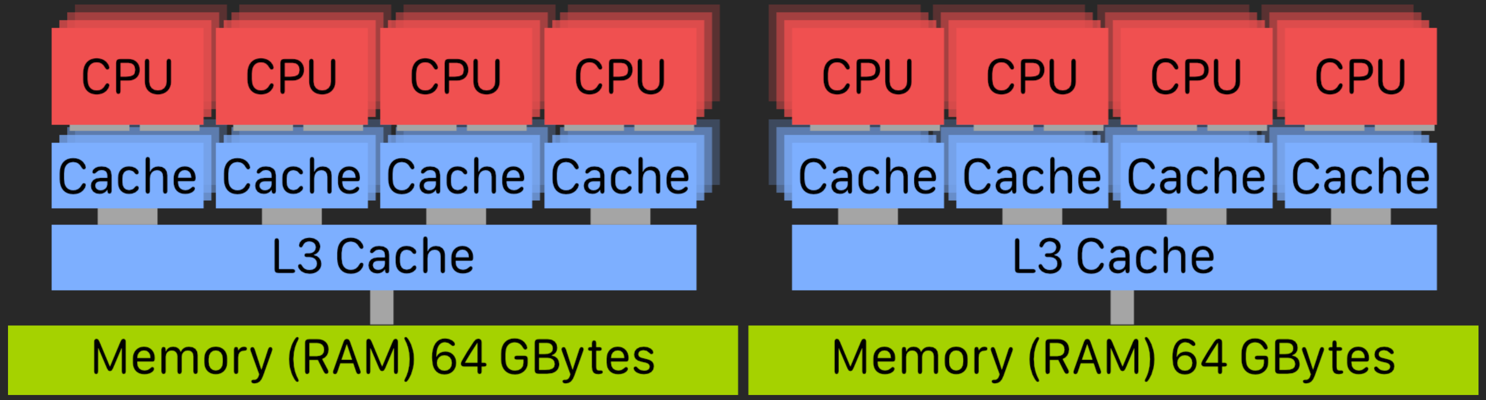


- What is the cache size?
  - 22 MBytes (for each socket)
  - See e.g. `/proc/cpuinfo` or `hwloc-ls`
- For which value of L do we reach the cache size limit
  - Storage requirements: $w \cdot 2 \cdot L$ (one `x[]` and one `y[]`)
  - w=`sizeof(double)`=8 Bytes

Performance analysis

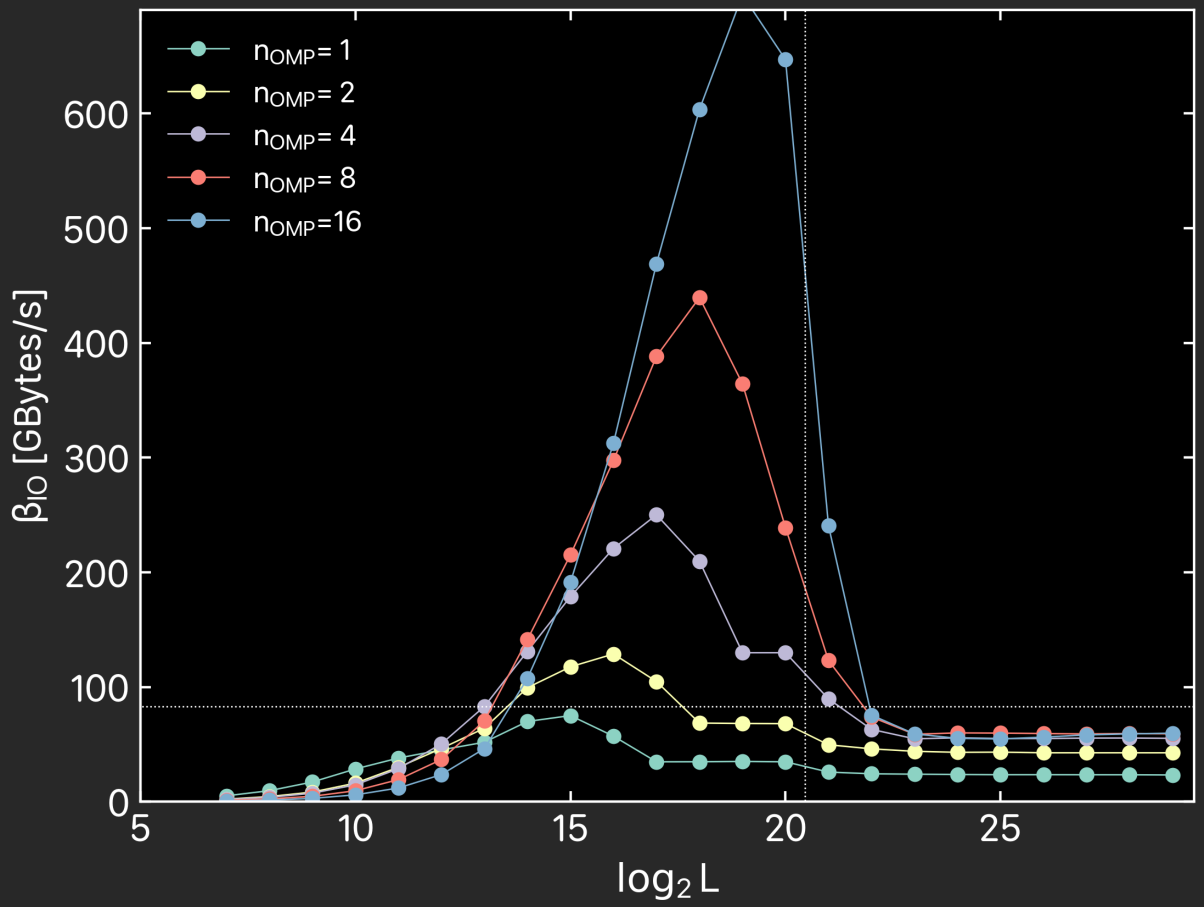
`axpy`:

$$y_i \leftarrow a \cdot x_i + y_i \qquad i = 0, \dots, L-1$$

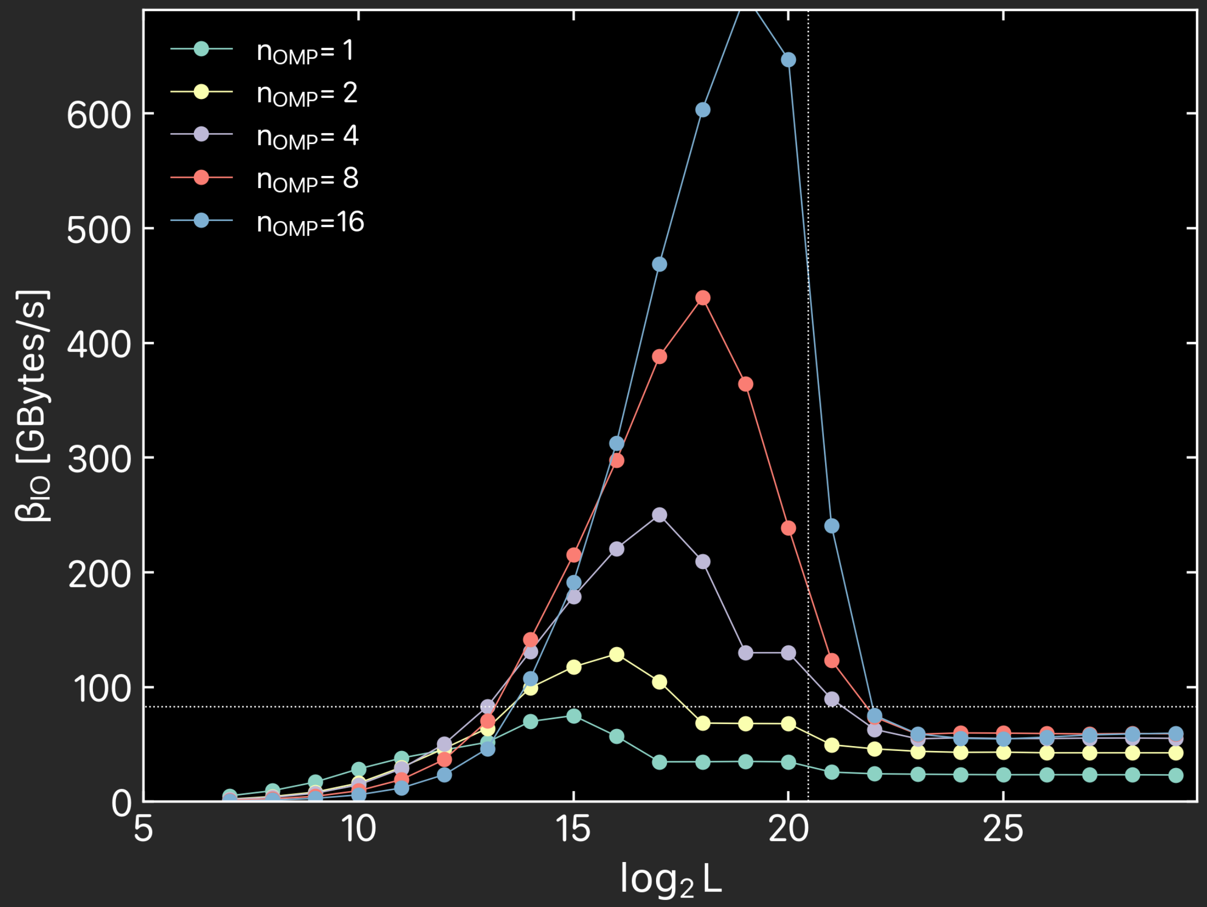


- What is the cache size?
  - 22 MBytes (for each socket)
  - See e.g. `/proc/cpuinfo` or `hwloc-ls`
- For which value of L do we reach the cache size limit
  - Storage requirements: $w \cdot 2 \cdot L$ (one `x[]` and one `y[]`)
  - w=`sizeof(double)`=8 Bytes
  - $\Rightarrow L_c$ =22 MBytes / (8 Bytes$\times 2$) $\simeq 2^{20}$

Performance analysis



Performance analysis



- Dashed vertical line indicates L_c, i.e. the largest value of L for which arrays `x[]` and `y[]` still fit in the L3 cache of 22 MBytes

Optimization

Cache locality

- The `axpy` analysis we just went through is a simple case that helps demonstrate the effect of "staying in cache" or "dropping out of cache"

Optimization

Cache locality

- The `axpy` analysis we just went through is a simple case that helps demonstrate the effect of "staying in cache" or "dropping out of cache"
- For `axpy`, there is not much to be done in terms of data-layout transformations that can help achieve better so-called "data locality"

Optimization

Cache locality

- The `axpy` analysis we just went through is a simple case that helps demonstrate the effect of "staying in cache" or "dropping out of cache"
- For `axpy`, there is not much to be done in terms of data-layout transformations that can help achieve better so-called "data locality"
- One good example though that can help demonstrate this effect it the matrix-matrix multiplication