LINMA2710 - Scientific Computing Shared-Memory Multiprocessing

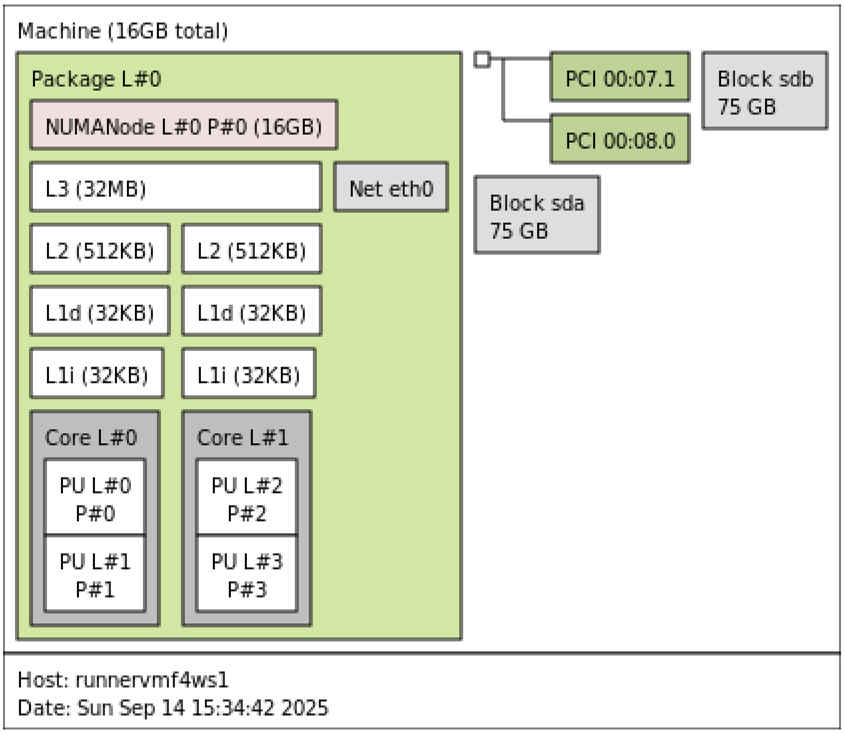
P.-A. Absil and B. Legat

☐ Full Width Mode ☐ Present Mode

Table of Contents

[Eij10] V. Eijkhout. *Introduction to High Performance Scientific Computing*. 3 Edition, Vol. 1 (Lulu.com, 2010).

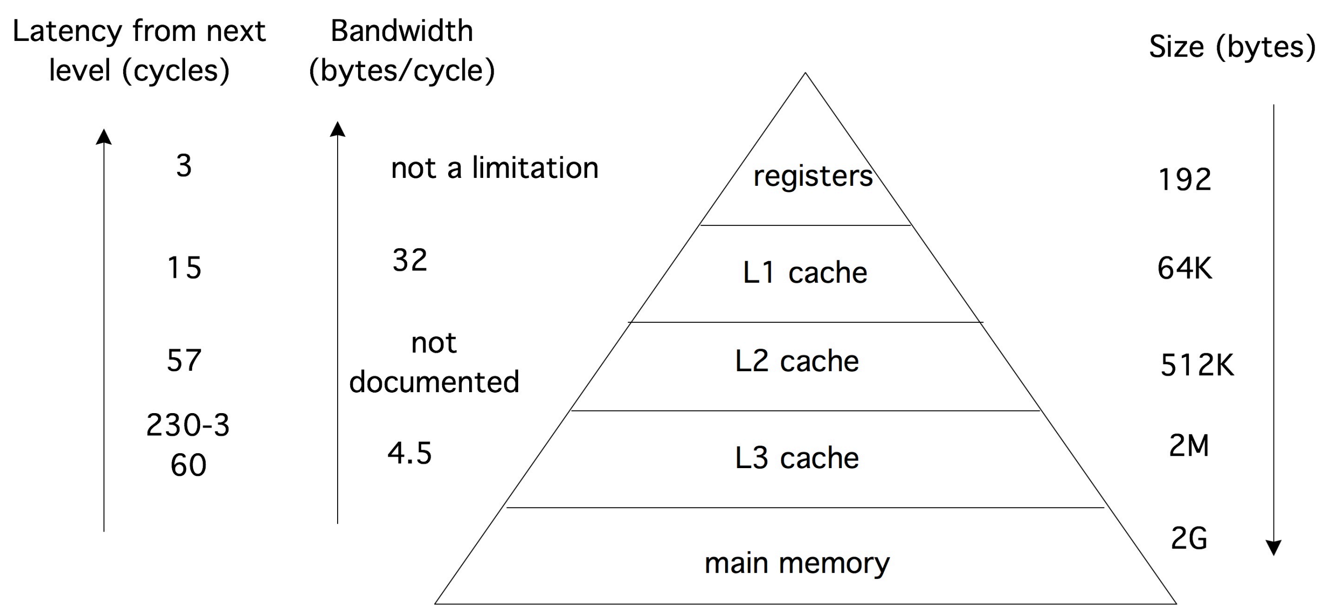
Memory layout



Try it on your laptop!

```
$ lstopo
```

Hierarchy



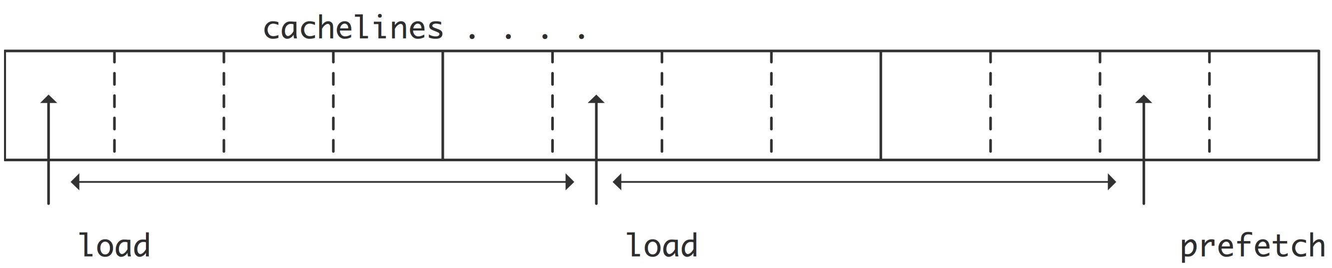
Latency of n bytes of data is given by

$$\alpha + \beta n$$

where α is the start up time and β is the inverse of the bandwidth.

[Eij10; Figure 1.5]

Cache lines and prefetch



- Accessing value not in the cache → *cache miss*
- This value is then loaded along with a whole cache line (e.g., 64 or 128 contiguous bytes)

- Following cache lines may also be anticipated and prefetched

This shows the importance of *data locality*. An algorithm performs better if it accesses data close in memory and in a predictable pattern.

[Eij10; Figure 1.11]

Illustration with matrices

```
32785.992f0
```

```julia
@btime c_sum($mat)
```

```
79.299 μs (0 allocations: 0 bytes)
```

```
mat = 256×256 Matrix{Float32}:
 0.205975   0.994568   0.842984   0.187235  …  0.302202   0.825757   0.108567
 0.375978   0.439683   0.427547   0.249569     0.409877   0.240031   0.740804
 0.59244    0.497382   0.926129   0.678407     0.302835   0.256302   0.879073
 0.348943   0.322328   0.386487   0.528854     0.131483   0.699491   0.342406
 0.592269   0.908697   0.636241   0.156057     0.919066   0.656222   0.038157
 0.984551   0.886692   0.825468   0.407443  …  0.960975   0.0742702  0.94217
 0.539956   0.10128    0.017572   0.154033     0.112219   0.132854   0.526577
 ⋮                                          ⋱                        ⋮
 0.0849119  0.138736   0.199132   0.526747  …  0.3481     0.524391   0.941522
 0.522502   0.217047   0.148168   0.564668     0.611634   0.853078   0.872231
 0.1774     0.871258   0.393734   0.853646     0.718816   0.554816   0.973876
 0.908187   0.0363892  0.0226635  0.319006     0.185189   0.223988   0.138812
 0.686104   0.912607   0.27206    0.608727     0.538763   0.545832   0.5254
 0.577607   0.600008   0.465188   0.223893  …  0.0741135  0.69968    0.144402
```

```julia
mat = rand(Cfloat, 2^8, 2^8)
```

```julia
c_sum(x::Matrix{Cfloat}) = ccall(("sum", sum_matrix_lib), Cfloat, (Ptr{Cfloat}, Cint, Cint), x, size(x, 1), size(x, 2));
```

```c
#include <stdio.h>

float sum(float *mat, int n, int m) {
  float total = 0;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
      total += mat[i + j * n];
    }
  }
  return total;
}
```

▶ **What is the performance issue of this code ?**

Arithmetic intensity

Consider a program requiring `m` load / store operations with memory for `o` arithmetic operations.

- The *arithmetic intensity* is the ratio $a = o/m$.
- The arithmetic time is $t_{\text{arith}} = o/\text{frequency}$
- The data transfer time is $t_{\text{mem}} = m/\text{bandwidth} = o/(a \cdot \text{bandwidth})$

As arithmetic operations and data transfer are done in parallel, the time per iteration is

$$\max(t_{\text{arith}}/o, t_{\text{mem}}/o) = 1/\min(\text{frequency}, a \cdot \text{bandwidth})$$

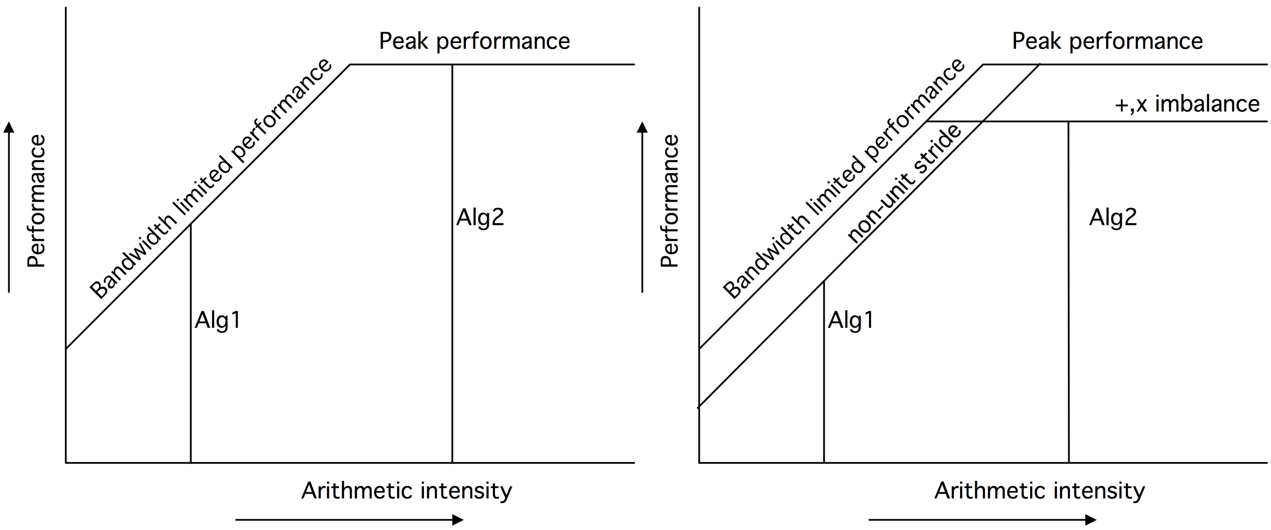
So the number of operations per second is $\min(\text{frequency}, a \cdot \text{bandwidth})$.

This piecewise linear function in a gives the *roofline model*.

Tip

See examples in [Eij10; Section 1.6.1].

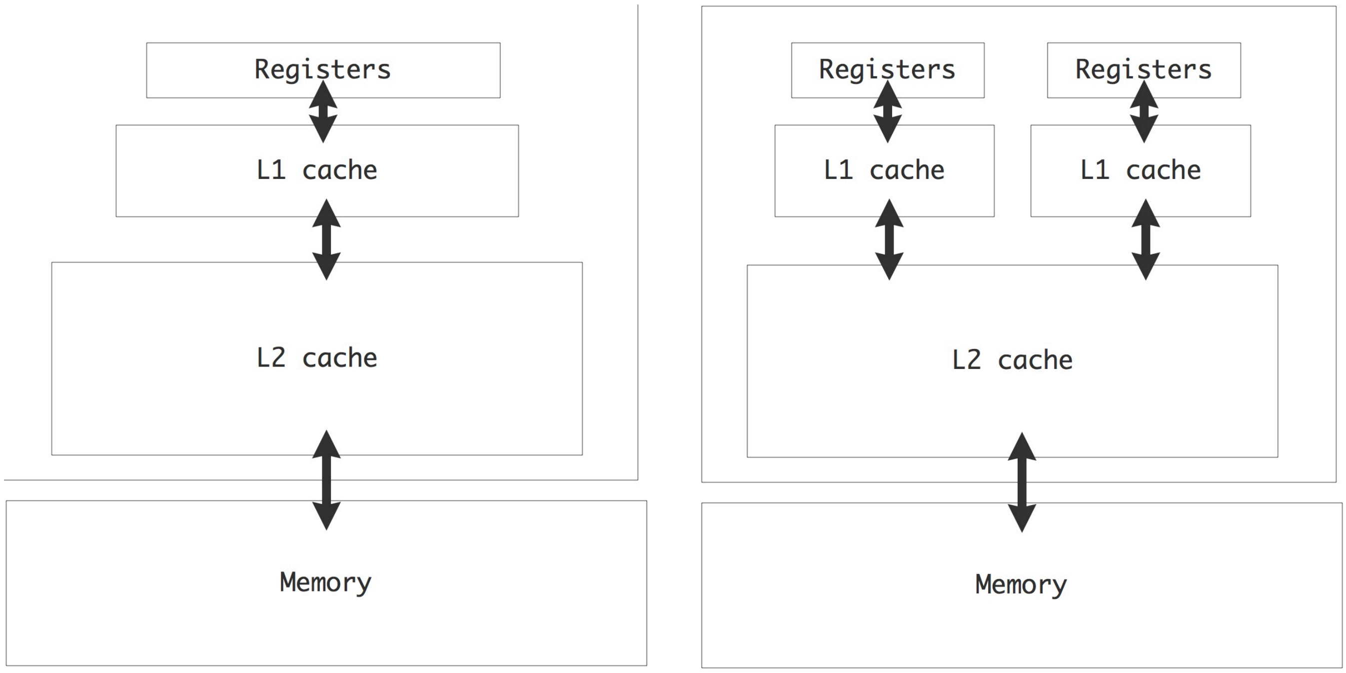
The roofline model



- *compute-bound* : For large arithmetic intensity (Alg2 in above picture), performance determined by processor characteristics
- *bandwidth-bound* : For low arithmetic intensity (Alg1 in above picture), performance determined by memory characteristics
- Bandwidth line may be lowered by inefficient memory access (e.g., no locality)
- Peak performance line may be lowered by inefficient use of CPU (e.g., not using SIMD)

[Eij10; Figure 1.16]

Cache hierarchy for a multi-core CPU



Cache coherence : Update L1 cache when the corresponding memory is modified by another core.

[Eij10; Figure 1.13]

Parallel sum

```
#include <vector>
#include <stdint.h>
#include <omp.h>
#include <stdio.h>

extern "C" {
float sum(float *vec, int length, int num_threads, int verbose) {
  float total = 0;
  omp_set_dynamic(0); // Force the value `num_threads`
  omp_set_num_threads(num_threads);
  #pragma omp parallel
  {
    int thread_num = omp_get_thread_num();
    int stride = length / num_threads;
    int last = stride * (thread_num + 1);
    if (thread_num + 1 == num_threads)
      last = length;
    if (verbose >= 1)
      fprintf(stderr, "thread id : %d / %d %d:%d\n", thread_num, omp_get_num_thre
ads(), stride * thread_num, last - 1);
    #pragma omp simd
    for (int i = stride * thread_num; i < last; i++)
      total += vec[i];
  }
  return total;
}
}
```

32674.21f0

```
@btime c_sum($vec, num_threads = 1, verbose = 0)
```

```
2.748 μs (0 allocations: 0 bytes)
```

32674.223f0

```
@btime c_sum($vec; num_threads, verbose = 0)
```

```
2.163 μs (3 allocations: 80 bytes)
```

32674.223f0

```
@time c_sum(vec; num_threads, verbose = 1)
```

```
thread id : 0 / 2 0:32767
thread id : 1 / 2 32768:65535
  0.000047 seconds
```

vec =

▸ [0.692107, 0.492174, 0.0687251, 0.563652, 0.891212, 0.295056, 0.601932, 0.240832, 0.01353

```
vec = rand(Cfloat, 2^log_size)
```

```
c_sum(x::Vector{Cfloat}; num_threads = 1, verbose = 0) = ccall(("sum", sum_lib),
Cfloat, (Ptr{Cfloat}, Cint, Cint, Cint), x, length(x), num_threads, verbose);
```

Low level implementation using POSIX Threads (pthreads) covered in "LEPL1503 : Projet 3". We use the high level OpenMP library in this course.

log_size = 16

num_threads = 2

▸ **Can you spot the issue in the code ?**

Many processors

```
#include <omp.h>
#include <stdio.h>

extern "C" {
void sum_to(float *vec, int length, float *local_results, int num_threads, int ve
rbose) {
  omp_set_dynamic(0); // Force the value `num_threads`
  omp_set_num_threads(num_threads);
  #pragma omp parallel
  {
    int thread_num = omp_get_thread_num();
    int stride = length / num_threads;
    int last = stride * (thread_num + 1);
    if (thread_num + 1 == num_threads)
      last = length;
    if (verbose >= 1)
      fprintf(stderr, "thread id : %d / %d %d:%d\n", thread_num, omp_get_num_thre
ads(), stride * thread_num, last - 1);
    float no_false_sharing = 0;
    #pragma omp simd
    for (int i = stride * thread_num; i < last; i++)
      no_false_sharing += vec[i];
    local_results[thread_num] = no_false_sharing;
  }
}

float sum(float *vec, int length, int num_threads, int factor, int verbose) {
  float* buffers[2] = {new float[num_threads], new float[num_threads / factor]};
  sum_to(vec, length, buffers[0], num_threads, verbose);
  int prev = num_threads, cur;
  int buffer_idx = 0;
  for (cur = num_threads / factor; cur > 0; cur /= factor) {
    sum_to(buffers[buffer_idx % 2], prev, buffers[(buffer_idx + 1) % 2], cur, ver
bose);
    prev = cur;
    buffer_idx += 1;
  }
  if (prev == 1)
    return buffers[buffer_idx % 2][0];
  sum_to(buffers[buffer_idx % 2], prev, buffers[(buffer_idx + 1) % 2], 1, verbos
e);
  return buffers[(buffer_idx + 1) % 2][0];
}
}
```

Benchmark

If we have many processors, we may want to speed up the last part as well:

32674.219f0

```
@time many_sum(vec; base_num_threads, factor, verbose = 1)
```

```
thread id : 0 / 2 0:32767
thread id : 1 / 2 32768:65535
thread id : 0 / 1 0:1
  0.000073 seconds
```

32859.277f0

```
@btime c_sum($many_vec)
```

```
  2.767 μs (0 allocations: 0 bytes)
```

32859.277f0

```
@btime many_sum($many_vec; base_num_threads, factor)
```

```
  6.236 μs (3 allocations: 80 bytes)
```

many_vec =
[0.258847, 0.689505, 0.141772, 0.410118, 0.375045, 0.110171, 0.322878, 0.842619, 0.458156

```
many_vec = rand(Cfloat, 2^many_log_size)
```

```
many_sum(x::Vector{Cfloat}; base_num_threads = 1, factor = 2, verbose = 0) =
ccall(("sum", many_sum_lib), Cfloat, (Ptr{Cfloat}, Cint, Cint, Cint, Cint), x,
length(x), base_num_threads, factor, verbose);
```

many_log_size = 16

base_num_threads = 2

factor = 2

Amdahl's law

Speed-up and efficency

Def: Speed-up

$$S_p = \frac{T_1}{T_p}$$

Def: Efficiency

$$E_p = \frac{S_p}{p}$$

Let T_p bet the time with p processes

- $E_p > 1$ → Unlikely
- $E_p = 1$ → Ideal
- $E_p < 1$ → Realistic

Amdahl's law

- F_s : Fraction of T_1 that is sequential
- $F_p = 1 - F_s$: Fraction of T_1 that is parallelizable

$$T_p = T_1 F_s + T_1 F_p / p$$

$$S_p = \frac{1}{F_s + F_p / p} \qquad E_p = \frac{1}{p F_s + F_p}$$

$$\lim_{p \to \infty} S_p = \frac{1}{F_s}$$

Application to parallel sum

The first `sum_to` takes n/p operations. Assuming `factor` is `2`, there is one operation for each of the $\log_2(p)$ subsequent `sum_to`.

$$
\begin{aligned}
T_1 &= n \\
T_p &= n/p + \log_2(p) \\
S_p &= \frac{1}{1/p + \log_2(p)/n} \qquad E_p = \frac{1}{1 + p\log_2(p)/n}
\end{aligned}
$$

▶ **How to get $1/F_s = \lim_{p\to\infty} S_p$?**

```
Activating project at `~/work/LINMA2710/LINMA2710/Lectures`
```

```
biblio =
▶ CitationBibliography("/home/runner/work/LINMA2710/LINMA2710/Lectures/references.bib", Alp
```

```
Loading bibliography from `/home/runner/work/LINMA2710/LINMA2710/Lectures/references.bib`...
```

```
Loading completed.
```

0.2

```
BenchmarkTools.DEFAULT_PARAMETERS.seconds = 0.2
```