EPFL

Parallel and High Performance Computing

Dr. Pablo Antolín

Solution Series 2

March 6 2025

Measuring CPU performances

Exercise 1: ***Theoretical analysis: Amdhal's and Gustafson's laws***

a) What is the definition of the Amdahl's law (with the serial part $1-\alpha$)? What does this law measure?

The Amdahl's law defines the maximum speedup as

$$S(p,\alpha) = \frac{T_1}{(1-\alpha)T_1 + \frac{\alpha}{p}T_1} = \frac{1}{(1-\alpha)+\frac{\alpha}{p}}.$$

It measures the maximum speedup a parallelized code can achieve by keeping the amount of work constant and by increasing the number of processes. This is the so called *strong scaling*.

b) What is the definition of the Gustafson's law (with the non-parallelizable part $1-\alpha(N)$ and problem size N)? What does this law measure?

The Gustafson's law defines the maximum speedup as

$$S(p,N) = 1 + (p-1)\alpha(N),$$

where p is the number of processes. It measures the maximum speedup a parallelized code can achieve by keeping the amount of work constant *per process* and by increasing the number of processes. This is the so called *weak scaling*.

c) Considering the provided algorithm and information, provide an estimation of $1-\alpha$, the serial part of the code that can not be parallelized.

According to the provided algorithm, the non-parallelizable part corresponds to the initialization stage. By computing $t_{\text{init}}/t_{\text{total}}$ we find $1-\alpha \simeq 1\%$ which is the serial part of the code.

d) What is the upper bound of the speedup according to Amdahl's law?

The speedup is bounded as

$$S(p,\alpha) = \frac{1}{(1-\alpha)+\frac{\alpha}{p}} < \frac{1}{1-\alpha}.$$

Therefore, with approximatively 1% of serial part, the speedup will be always below 100.

e) What would be the maximum efficiency with 128 processors?

The efficiency is

$$E(p, \alpha) = \frac{S(p, \alpha)}{p} = \frac{1}{p(\alpha - 1) + \alpha},$$

and for $\alpha - 1 \simeq 1\%$ and $p = 128$ the efficiency will be around 44%.

Exercise 2: ***Theoretical roofline***

On helvetios `cat /proc/cpuinfo` shows that the processors are `Intel(R) Xeon(R) Gold 6140`:

```
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 85
model name      : Intel(R) Xeon(R) Gold 6140 CPU @ 2.30GHz
stepping        : 4
microcode       : 0x2006f05
cpu MHz         : 2881.881
cache size      : 25344 KB
physical id     : 0
siblings        : 18
core id         : 0
cpu cores       : 18
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 22
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
 ↪  pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb
 ↪  rdtscp lm constant_tsc art arch_perfmon pebs bts rep_good nopl xtopology
 ↪  nonstop_tsc cpuid aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx smx est
 ↪  tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid dca sse4_1 sse4_2 x2apic movbe popcnt
 ↪  tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch
 ↪  cpuid_fault epb cat_l3 cdp_l3 invpcid_single pti intel_ppin ssbd mba ibrs
 ↪  ibpb stibp fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm cqm
 ↪  mpx rdt_a avx512f avx512dq rdseed adx smap clflushopt clwb intel_pt avx512cd
 ↪  avx512bw avx512vl xsaveopt xsavec xgetbv1 xsaves cqm_llc cqm_occup_llc
 ↪  cqm_mbm_total cqm_mbm_local dtherm ida arat pln pts pku ospke md_clear
 ↪  flush_l1d arch_capabilities
bugs            : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf mds
 ↪  swapgs taa itlb_multihit
bogomips        : 4600.00
clflush size    : 64
cache_alignment : 64
address sizes   : 46 bits physical, 48 bits virtual
```

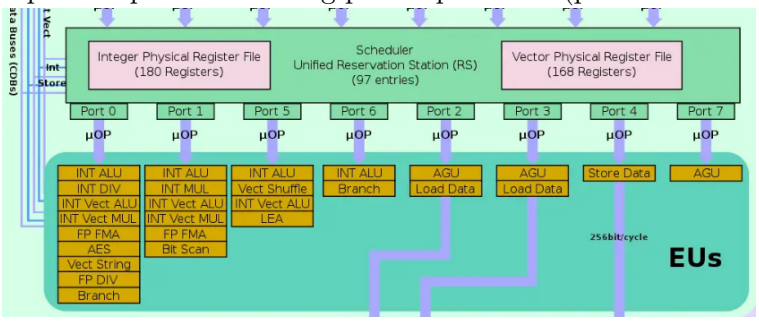
```
power management:
```

You can find all the information about that processor in the links below:

- https://en.wikichip.org/wiki/intel/xeon_gold/6140
- https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client)

a) From the micro-architecture page we can see:

- 3.5 GHz (turbo frequency for AVX512 and 1 single core).
- 2 ports capable of floating point operations (ports 0 and 1 in the image below).



- 2 operations per cycle if we consider FP FMA (fused multiple add): one multiplication and one addition.
- AVX512 FMA units which gives a vector size of 512 bit (8 double precision floating point). Intel vector extensions being in order of increasing vector sizes: MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX, AVX2, AVX512.

Then, the peak performance is computed as $3.5 \cdot 10^9 \times 2 \times 2 \times 512/64 = 112\,\text{GFLOP/(s core)}$.

b) In the processor webpage we can read that the bandwidth single (max for 1 core) is 19.87 GiB/s and max 119.21 GiB/s. This can be also computed manually. As specified in the first link, this microprocessor has a memory of type DDR4-2666 with 6 channels. This is a memory with data transfer rate of 2666 MT/c and bus width 64 bit (inherent to DDR4 memory). Then, the bandwidth for one single channel is $2666\,\text{MT/c} \times 8\,\text{B/T} = 21.33\,\text{GB/s} = 19.87\,\text{GiB/s}$. And considering the 6 memory channels, the maximum bandwidth is $21.33\,\text{GB/s} \times 6 = 127.98\,\text{GB/s} = 119.21\,\text{GiB/s}$.

Remark: Notice the difference between gibibytes and gigabytes units (see https://en.wikipedia.org/wiki/Byte#Multiple-byte_units).

c) Thus, the *ridge point* is $112 \cdot 10^9/21.33 \cdot 10^9 = 5.25\,\text{FLOP/B}$ for 1 core.

Exercise 3: *Measured roofline*

For `OMP_NUM_THREADS=1` we obtained (taking the minimum of the four values for the sustained memory performance):

a) `Stream`: 10.10 GB/s on 1 core.

b) `Dgemm`: 99.7 GFLOP/s on 1 core.

c) Ridge point: 9.87 FLOP/B on 1 core.

Instead, for `OMP_NUM_THREADS=8` we got:

a) `Stream`: 72.48 GB/s on 8 cores.

b) `Dgemm`: 651.01 GFLOP/s on 8 cores.

c) Ridge point: 8.98 FLOP/B on 8 core.

Exercise 4: ***Jacobi stencil***

a) Arithmetic intensity: $4\,\text{FLOP}/(5 \times 8\ \text{B}) = 0.1\,\text{FLOP/B}$
4 operations (3 additions and one 1 multiplications)
5 data access on double precision FP (4 read and 1 write)

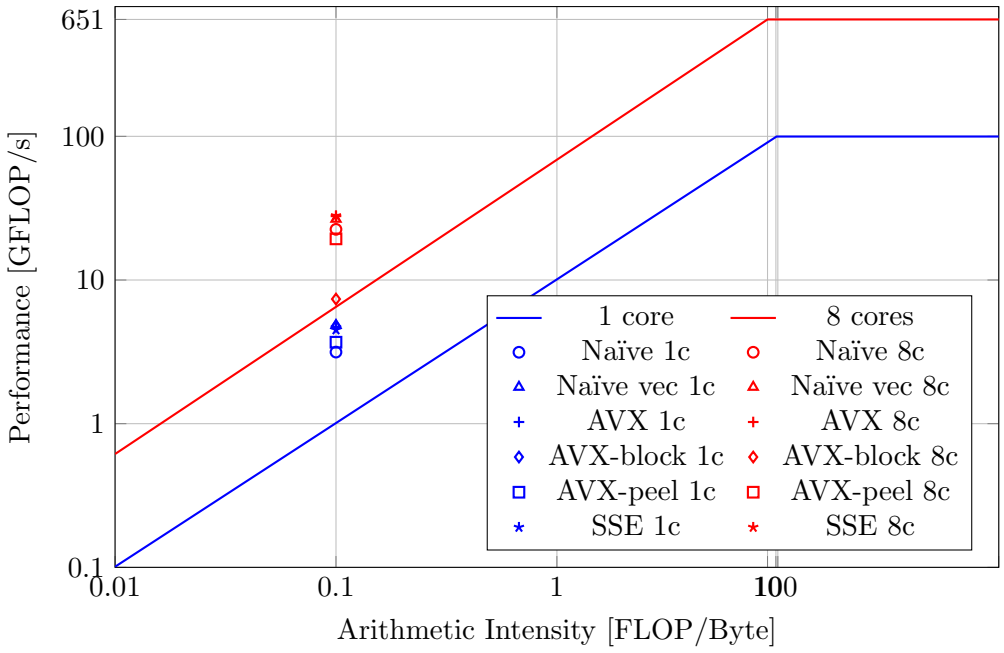
b) Targeted performance. Because we are on the left of the ridge point, the Jacobi stencil is memory bounded (see Fig. 1):
$10.10\,\text{GB/s} \times 0.1\,\text{FLOP/B} = 1.00\,\text{GFLOP/s}$ on 1 core
$72.48\,\text{GB/s} \times 0.1\,\text{FLOP/B} = 7.24\,\text{GFLOP/s}$ on 8 cores.

c) The obtained GFLOP/s are reported in the table below (and in Fig. 1)

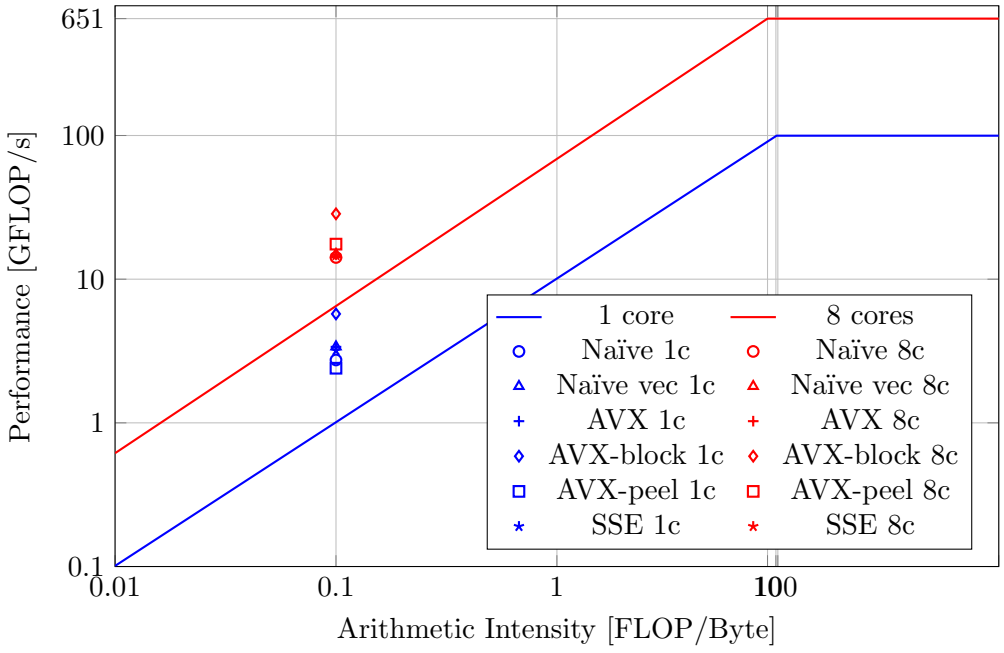
	1000 points		10000 points	
	1 thread	8 threads	1 thread	8 threads
`jacobi-naive`	3.15	22.52	2.74	14.16

d) Focusing first in the case with 1000 points, we observe that for 1 single thread, the obtained performance was 3.15 GFLOP/s, while, according to b), a maximum theoretical value 1.00 GFLOP/s should be expected: In Fig. 1, the result of the naïve implementation should be below the roofline model, but it is above. This difference comes from compiler optimizations and the fact that our roofline model is based on DRAMM bandwidth. However, in this problem we have cache hits, i.e., the required data is not retrieved from the main memory every time, but most of the times is accessible through the different cache levels (spatial and temporal localities), what is much faster. In order to do a more precise estimation, the caches bandwidths should be considered in a refined version of the roofline model. The same applies to the case with 7 threads: 22.52 GFLOP/s versus 7.24 GFLOP/s (expected).

On the other hand, when we consider 10000 stencil points, the obtained performance is significantly worse. Assuming that the data in `u` is stored in a rowwise way, the values `u(i,j-1)`, `u(i,j)`, and `u(i,j+1)` are contiguous in memory (spatial locality). However, the values `u(i-1,j)` and `u(i+1,j)` are at a distance of $2 \times 10000 \times 8 = 160\,\text{kB}$, that is larger than the cache size (L1), see exercise 2. Therefore, at every iteration, cache misses occur, requiring to retrieve the data from lower cache levels and the main memory, what slows the access to the data. Still, the computed performance is superior to the expected one, as L2 and L3 cache levels are still use, and they present a higher bandwidth than the main memory.



(a) $n = 1000$



(b) $n = 10000$

Figure 1: Peformance of different Jacobi stencil implementations (and optimizations) for $n = 1000$ and $n = 10000$.

Exercise 5: ***Optimized Jacobi stencil***

a) In this exercise we explore Data Level Parallelism (DLP): in `jacobi-naive-auto-vec`, we let the compiler full freedom for optimizing as much as possible the code[1]; the files `jacobi-sse.c` and `jacobi-avx*.c` contain different implementations using DLP with intrinsics. All the computed peformances in GFLOP/s are reported in the table below (as well as in Fig. 1).

We expect the AVX block version to be the most efficient since it subdivides the problem into blocks of 512×512 points, maximiming the L1 cache reuse (hits). However, when 8 threads are used, `jacobi-avx-block` is only able to reach a high performance when enough points per direction are provided, as every thread works on a 512×512 points block. On the other hand, the version `jacobi-naive-auto-vec` achieves very high performances for 1 and 8 threads and 1000 points just by using the compiler automatic vectorization. However, when the number of points increases, that performance is bounded by the time to access memory (cache misses), achieving the same performance as `jacobi-naive`.

	1000 points		10000 points	
	1 thread	8 threads	1 thread	8 threads
`jacobi-naive-auto-vec`	4.84	26.56	3.39	14.92
`jacobi-avx`	4.73	27.85	3.38	14.83
`jacobi-avx-block`	4.78	7.37	5.73	28.50
`jacobi-avx-peel`	3.69	19.34	2.40	17.54
`jacobi-sse`	4.50	28.03	3.18	14.63

b) To beat those performances it is difficult in a general case. But for a fixed number of points in the stencil, and knowing the cache architecture of a target machine, it may be possible to tune the `jacobi-avx-block` version for that specific case and squeeze a few more GFLOP/s, but not much more, as the problem is memory bounded.

[1] Notice that in the `Makefile` of the Jacobi stencil, automatic vectorization was disabled for the executable `jacobi-naive`.