Measuring/Evaluating the performance of a computer system using benchmarks

Computer Architecture project

[ISTY]

- UVSQ, Paris Saclay -

Olivier Benaben

1 February, 2021

1 Introduction

This project is about CPU memory benchmark for UNIX systems.

The purpose was to run a collection of benchmarks on the cleanest possible system so that the results would not be bloated by other processes.

In this way, we may want to kill all unnecessary processes like network deamon, destkop manager & X server, ...

A good way to boot on a clean system is to request the `Grub/rEFInd` to start in text mode.

Otherwise, it is possible to kill all GUI instance after the boot : after accessing a new `tty` by pressing `Ctrl + At + F2/F3/..`, we are able to terminate the window manager with a `sudo systemctl stop gdm # (or kvm, lightdm, ...)`.

Contents

2 README

2.1 Instructions

2.1.1 Compilation

Simply run the `./compile_all.sh` script, it will compile all the benchmarks for you.

USAGE : `./compile_all.sh` : build all benchmarks in the current directory `./compile_all.sh [-c | --clean]` : clean all builds (*ie* remove the executables)

2.1.2 Run the benchmarks

The `./run_benchs.sh` script run all the benchs previously compiled (it may take a few moment). It generates a `.dat` file for each benchmark in it's directory, containing the perfs.

USAGE : `./run_bench.sh [BENCH_EXECUTABLE PATH]` : run only the executable passed in argument. Otherwise, all the benchmarks are run.

Each benchmark is run 3 times (for the `L1`, `L2`, `L3`and `DRAM` caches).

Be sure to configure the caches sizes according to you architecture at the top of the script. (cf. `lshw -C memory` command). Also you may want to configure the core on witch the tests are performed (you may want a physical core id, cf `egrep -e "core id" -e ^physical /proc/cpuinfo | xargs -l2 echo | sort -u`).

2.1.3 Generating charts

The `./gen_charts.sh` script generates a chart for each `.dat` generated with the previous cammand.

How it works : For each subdirectory containing a benchmarck executable, it calls the `gnuplot` script `template.gp` with according arguments to generate chart for the benchmark (avaliable in the `./charts/` directory)

3 Benchmarks results

3.1 copy

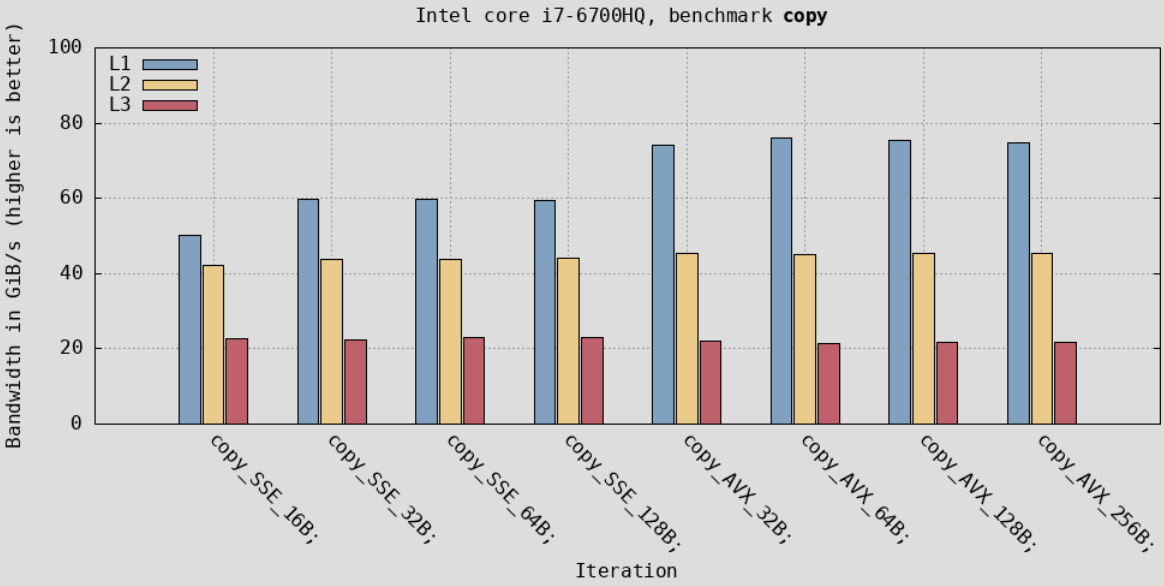

Figure 1: copy benchmark results (for L1, L2 and L3 caches)

The result of the copy benchmark are very straightforward : the farther the memory used is from the precursor, the lower is the bandwidth. The measured values are for load and write in memory.

The blue bars represent the run where only the L1 cache is used, and the bandwith we measure is in average 70GiB/s for my processor. The yellow bars shows a run where the L1 is filled and half of the L2 cache, the difference is striking : only 40GiB/s ; and on the last run, where we filed both L1, L2, and half of the L3 cache, we have 20GiB/s.

We notice a slight change in the L1 results : the bandwidth is a bit better when `AVX` instructions are used (almost 20GiB/s). It may be the processor which preferes those instructions than `SSE`. Anyway, we do not detect this change in the other runs where the others caches are involved.

3.2 dotprod

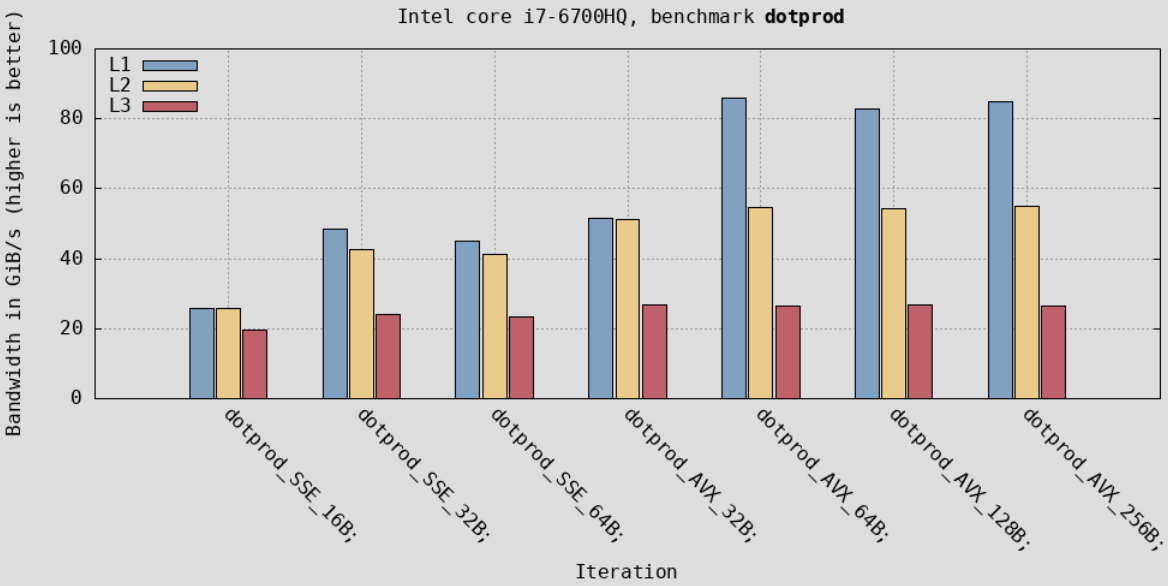

Figure 2: dotprod benchmark results (for L1, L2 and L3 caches)

The interpretation for this benchmark is even as the previous previous one : the farther the memory used is from the precursor, the lower is the bandwidth. The measured values are also for load and store in the memory, but this time there is an extra operation between registers.

The difference of bandwith for this benchmark between the L1 and L2 caches is lower compared to the previous test.

We measure a bandwidth of 70GiB/s in average for the L1 cache. The L2 performances are a bit lower : only 40GiB/s ; and on the last run, where we filed both L1 and L2, and half of the L3 cache, we have 20GiB/s.

We notice the same evolution we found in the previous benchmark, but this time for both the L1 and L2 caches : the bandwidth is a bit better when `AVX` instructions are used. This changes are more visible for the L1 cache : the difference is more than 20GiB/s, where it's only 10GiB/s for the L2.

3.3 load

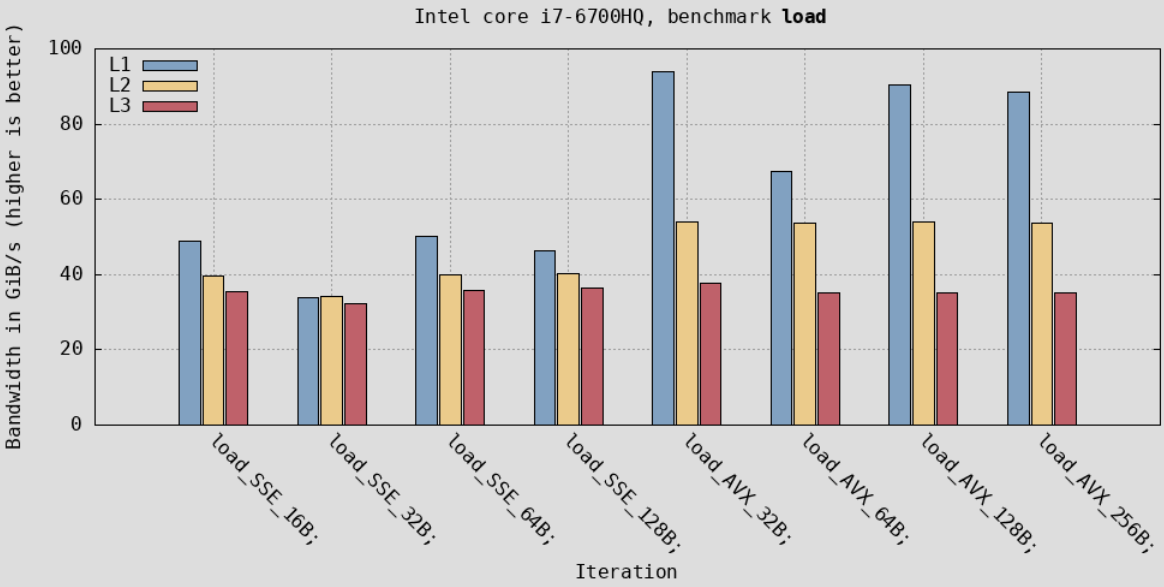

Figure 3: load benchmark results (for L1, L2 and L3 caches)

On this benchmark, the bandwidth measured is only for a load operation, and the values beetween runs are surprisingly very close - and low - when we use `SSE` instructions.

On the other hand, when `AVX` instructions are used, L1 cache is about twice more efficient, and a small gap between L2 and L3 bandwidth come up.

The L3 bandwidth seems fixed at 30GiB/s, independently of the benchmark variant.

3.4 memcpy

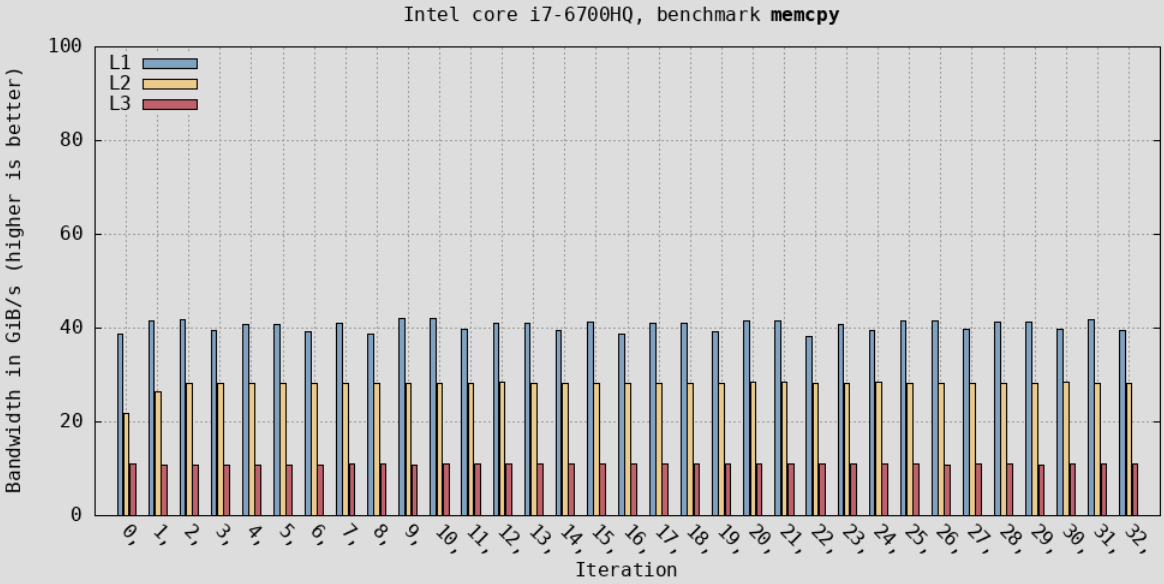

Figure 4: memcpy benchmark results (for L1, L2 and L3 caches)

This benchmark measure the performance of the libc's memcpy primitive.

The results are pretty clear : the L1's bandwidth is better than L2's, and L2 is better than L3 one.

3.5 ntstore

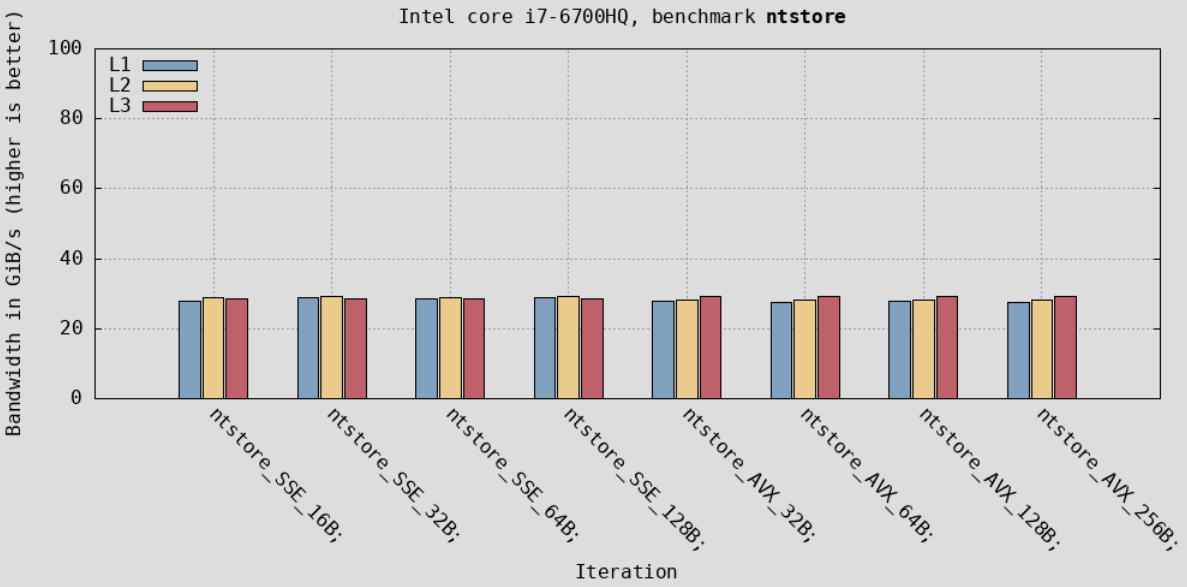

Figure 5: ntstore benchmark results (for L1, L2 and L3 caches)

As long as this benchmark uses **non-temporal stores** (*ie* does not uses cache), the bandwidths computed does not depend on the L* caches, and the result for all runs is actually the bandwidth of the `DRAM` of my computer - which is about 28GiB/s.

3.6 pc

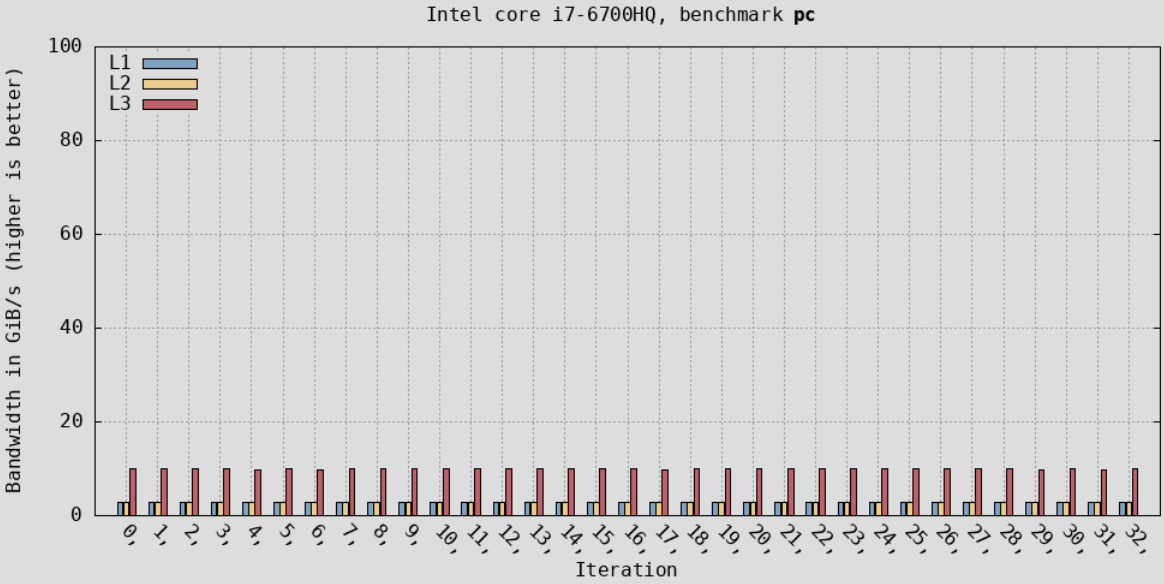

Figure 6: pc benchmark results (for L1, L2 and L3 caches)

This benchmarks measure the bandwidth of the DRAM loads, without CPU interferences by bypassing prefetchers thanks to a 'random pointer chasing' method.

As long as we benchmark the DRAM only, it makes sense that the result is not dependant on L* caches. However I don't explain why the L3's results are way higher than the others...

3.7 reduc

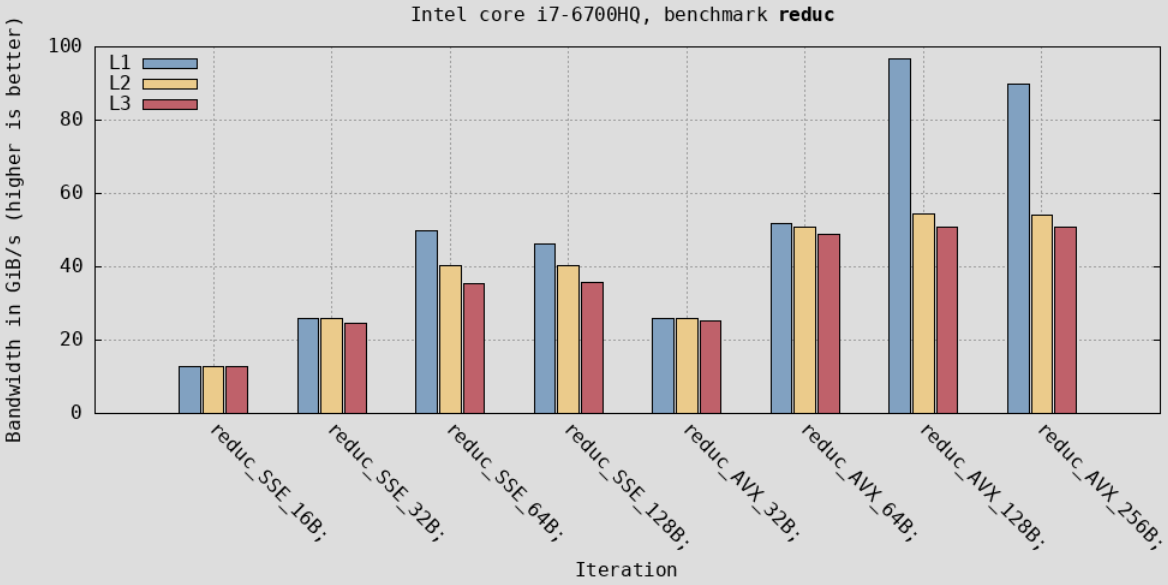

Figure 7: reduc benchmark results (for L1, L2 and L3 caches)

In this benchmark again, the most important result is that the more large the vector instruction is, the higher the bandwidth is, and so for all the caches used.

For instance, using `SSE 128B` instead of `SSE 16B` instructions triples the bandwidth, whatever the caches used.

Also, we can see that the `AVX` instructions are much more effective than `SSE` ones.

The improvement of "maximizing the length of the vector instructions" is here limited by the caches themselves. L2 and L3 caches reach their maximum bandwidth for the variations `AVX 64B` to `256B` around 50GiB/s.

The L1 cache show its bandwidth superiority, and continue rising to almost a 100GiB/s, thanks to the large vectors instructions.

3.8 store

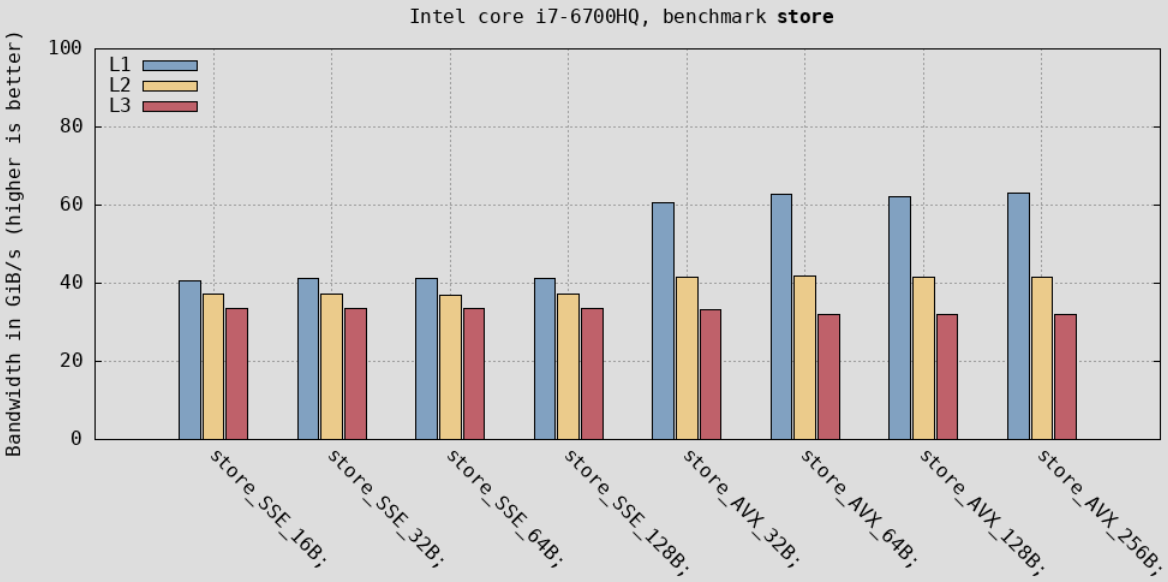

Figure 8: store benchmark results (for L1, L2 and L3 caches)

These results shows once again that L1 is faster than L2, which is faster than L3, this time for store operation only. The results are less impressive than the load operations : the differences between the differences between the runs are more similar than for the load test.

An interesting point is here that the `AVX` instructions are more effective than the `SSE` ones ; and takes better benefits of the storing advantages of the different caches : the results for the L1 cache are far better than the others with `AVX` instructions than they are with `SSE`.

3.9 triad

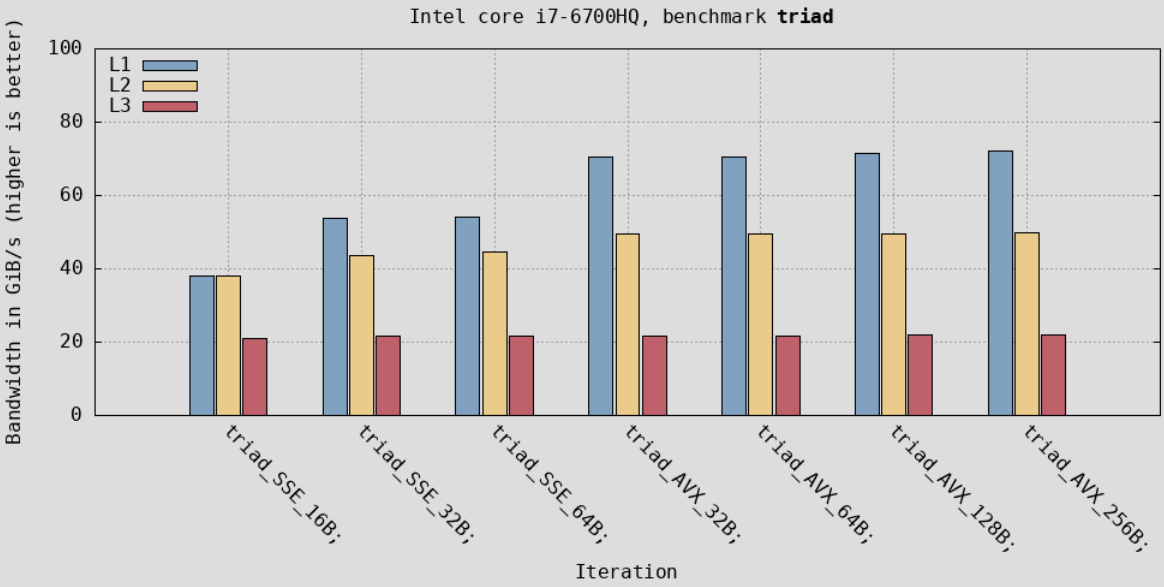

Figure 9: triad benchmark results (for L1, L2 and L3 caches)

On this benchmark, once again, we can see that L1 is faster than L2, which is faster than L3. Also, the larger the vector instructions are, the best are the results.

We can notice that there is no differences between the four variations of the `AVX` instructions size.

4 Conclusion

This project was about benchmarking the CPU memeory on a computer system. All the tests have been run on a linux system.

As predicted, we measured that the L1 bandwidth is faster than the L2's, which is faster than the L3's.

Also, we have noticed two trends in the instructions used :

- The larger the instructions register is, the faster the bandwidth will be. This is because vector operations processes multiple data in once, and the more data the CPU can handle in one time, the less operations will be needed.
- `AVX` instructions are more efficient than `SSE` ones, especially for the L1-contained data.