Report OpenMP

Authors:
Riccardo Isola - S4943369
Gabriele Dellepere - S4944557
Kevin Cattaneo - S4944382

Index

Head of analysis

Algorithm: the code provided has quadratic N^2 complexity.
Tools: we used the *icx* compiler, the Intel one, and Intel Advisor GUI to perform the most of the analysis.
Machine: the machine on which we run the code has 20 processors:

- 12 core
    - 8 core with hyperthreading up to 2
    - 4 performance core

Hotspot identification

In first place we compiled the program:
- **compiling line:** `icx -g -fopenmp omp_homework.c`
- **data size:** `30000`
- **time taken:** `67.684 seconds`

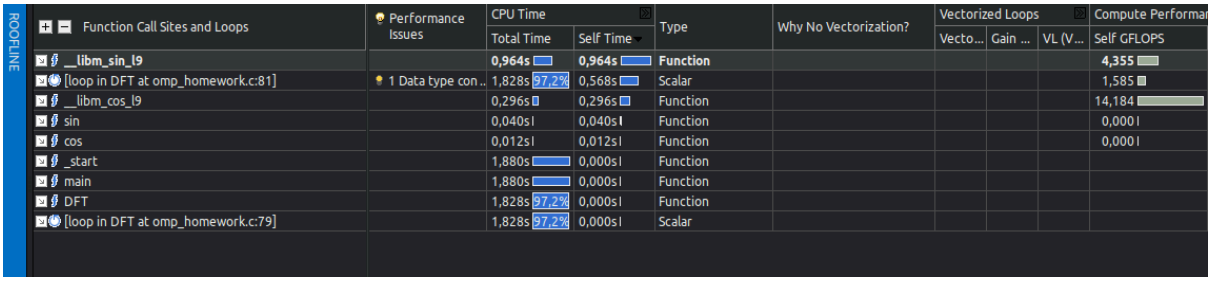
Then we put the executable into intel advisor to perform a detailed analysis, where we identified the following hotspots.

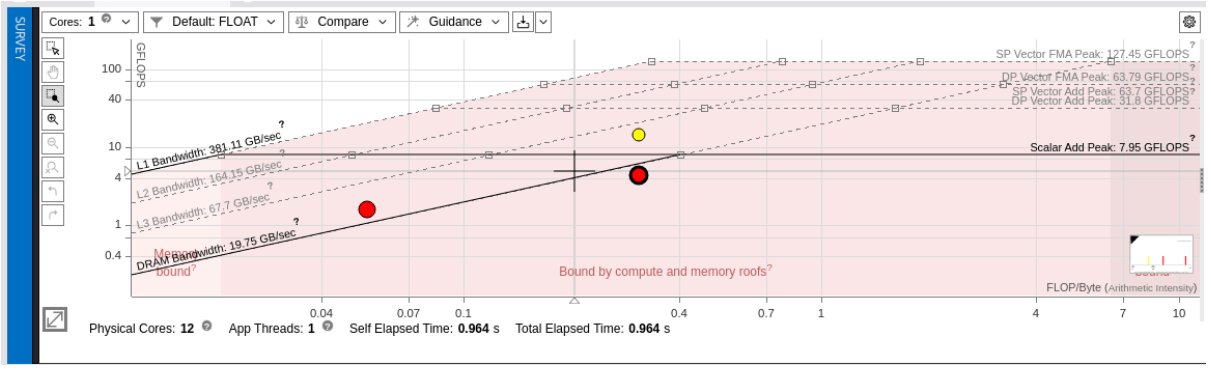
Name	Time taken (seconds)
Overall program	67.684242
DFT loops (lines 71-73)	24.496
sin function	31.982
cos function	9.452

It was clear (being the sin and cos function called only inside the DFT inner loop) that the region of code that took the most amount of time during the computation were the lines 71-80, containing the two loops needed to compute the DFT.

To delve into more details of the hotspots, we computed the roofline model, and we lowered the datasize (for a matter of analysis time available):

- **compiling line:** `icx -g -fopenmp omp_homework.c`
- **data size:** `5000`
- **time taken:** `1.869 seconds`





As seen before we have three hotspots represented by the red (took more time) and yellow points. In particular on the left we have the DFT loop hotspot. We observed that our performances are near the corresponding rooflines: our arithmetic intensity is low, so the limit is represented by the memory.
We could reach better performances if we run a different implementation of our algorithm that exploits the caches better.

First optimization (manual)

We provided a first optimization by swapping the for loops in line 71 with 73. The main objective was, by having n fixed in the inner loop, to reuse cache lines when using xr[n] xi[n] in the computation, where k was used continuously to perform write operations.

- **compiling line:** `icx -g -fopenmp omp_homework.c`
- **data size:** `30000`
- **time taken:** `30.452 seconds`

Name	Time taken (seconds)
Overall program	30.451946
DFT loops (lines 71-73)	6.932
sin function	17.780
cos function	4,936

The manual swap reduced the overall program time by half.

Vectorization issues

To obtain the report about vectorization infos, we specified the level 3:

First
- **compiling line:** `icx -fopenmp -O3 -xHost -qopt-report=3 omp_homework.c`
- **data size:** `30000`
- **time taken:** `3.410 seconds`
- **note:** `no manual swap of loops`

Second
- **compiling line:** `icx -fopenmp -O3 -xHost -qopt-report=3 omp_homework.c`
- **data size:** `30000`
- **time taken:** `3.440 seconds`
- **note:** `applied manual swap of loops`

We observed that:

- no vectorization issues reported
- all the inner loops, apart the one where we print the result, are successfully vectorized (this includes the signal generation loop, however this has minimal impact on the performance)
- the only difference is that in the non-swapped case the report shows a message telling that a vectorization reduction has been performed, still in the second case with the manual loop swapping we obtained the same time results

Best sequential time

Now we perform different runs to find the best sequential time, to be used as reference for the further parallel application, by passing several arguments and flags to the intel compiler.

- **compiling line:** `icx -fopenmp -O3 -xHost -ipo -ffast-math omp_homework.c`
- **data size:** 30000
- **time taken:** 3.501 seconds
- **note:** no manual swap of loops

Name	Time taken (seconds)
Overall program	3.500807
DFT loops (lines 71-73)	0.470
sin function	0.556
cos function	2.044

Overall we are applying the best optimizations arguments:

- **O3**: to allow the compiler to optimize the code, so that it can reorder the operations in the most efficient way, perform hoisting and use sincos instead of computing the sin and cosine in separate operations
- **xHost**: to achieve the best assembly instructions specifically on the current machine architecture
- **ipo**: to enable interprocedural optimizations; we observed that with this flag the program didn't achieve better performances: the program slows by a fraction of time (also by increasing data size)
- **fast-math**: to reduce the time needed for mathematical operations, altough reducing the precision of our calculations.

Then another run:

- **compiling line:** `icx -fopenmp -fopenmp-O3 -xHost -ipo -ffast-math omp_homework.c`
- **data size:** 30000
- **time taken:** 3.501 seconds
- **note:** applied manual swap of loops

In this case Intel advisor software suggested to use specifically the correct FMA instruction, so we specified the flag -axCORE-AVX2 as suggested, instead of xHost.
But the result in time was the same, so xHost already does the work well.
At the end we decided to discard the usage of the manual swap.

Then we performed several runs to reach at least 30 seconds of execution, and to do so we increased the data size:

- **compiling line**: icx -O3 -xHost -ipo -ffast-math omp_homework.c
- **data size**: 90000
- **time taken**: 30.454 seconds
- **note**: no manual swap of loops

OpenMP Parallelization

The following analysis is again performed with N = 90000 and the optimal number of threads decided by the scheduler.

On line 71 we put the following line:
 # pragma omp parallel for collapse(2) default(none) private(k,n) shared(Xr_o,xr,xi,Xi_o,N,idft)

The usage of *collapse(2)* would collapse both the DFT loops into a single longer one, allowing us to theoretically parallelize every iteration.

- **compiling line**: icx -fopenmp -O3 -xHost -ipo -ffast-math omp_homework.c
- **data size**: 90000
- **number of threads**: 20 (decided by scheduler)
- **time taken**: 26.731 seconds
- **note**: no manual swap of loops

We assumed the poor results were caused by the excessive dimensions of the collapsed loop, which caused too much overhead for the “small” number of threads we could assign jobs to.

We then tried parallelizing **only the outer** loop:

 # pragma omp parallel for default(none) private(k,n) shared(Xr_o,xr,xi,Xi_o,N,idft)

- **compiling line**: icx -fopenmp -O3 -xHost -ipo -ffast-math omp_homework.c
- **data size**: 90000
- **number of threads**: 20 (decided by scheduler)
- **time taken**: 4.615 seconds
- **note**: no manual swap of loops

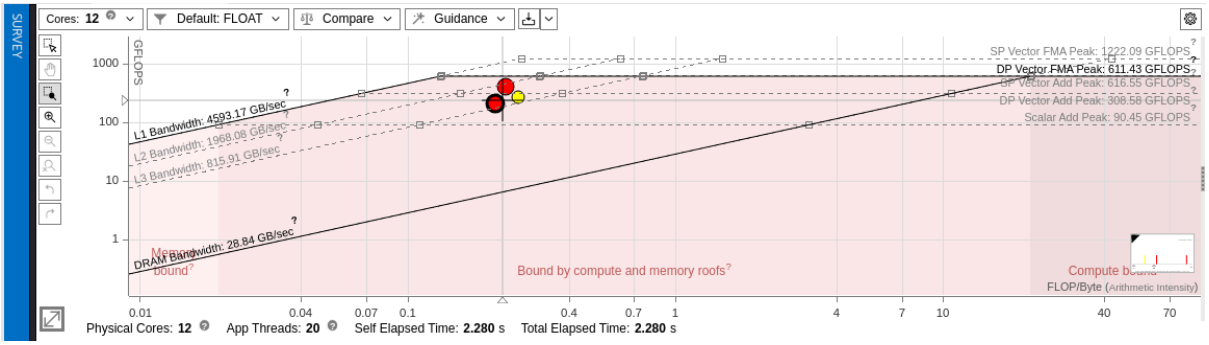
As a result the program took **4.615 seconds.** So just the parallelization of the first and most external loop seems to be sufficient and doesn’t provide the overhead of the previous defined *pragma*.

We proceeded to do the same for **just the inner loop** as well, with some precautions to avoid breaking the parallelization:

On line 71 we removed the “for” from the pragma:
 # pragma omp parallel default(none) private(k,n)
`shared(Xr_o,xr,xi,Xi_o,N,idft)`
And on line 73 (inner loop in line 74) we added:
 # pragma omp for nowait
The nowait was needed to avoid having a barrier being waited at after each iteration of the loop.

- **compiling line:** `icx -fopenmp -O3 -xHost -ipo -ffast-math omp_homework.c`
- **data size:** `90000`
- **number of threads:** `20 (decided by scheduler)`
- **time taken:** `4.849 seconds`
- **note:** `no manual swap of loops`

As a result the program took **4.849 seconds.**



Also we can see from Intel Advisor that our performances have increased as expected. In that case we are very near to the roofline line of L1 cache exploitations with the current set of instructions and optimizations.
We just found the best combination of flags for parallelization using the optimal number of threads decided by the scheduler.
Now we increase the data size to test the limits by trying different combinations of the usage of the *pragma* instructions.

First
- **compiling line:** `icx -fopenmp -O3 -xHost -ipo -ffast-math omp_homework.c`
- **data size:** `230000`
- **number of threads:** `20 (decided by scheduler)`
- **time taken:** `34.311 seconds`
- **note:** `both pragma instruction applied (just the internal loop is parallelized)`

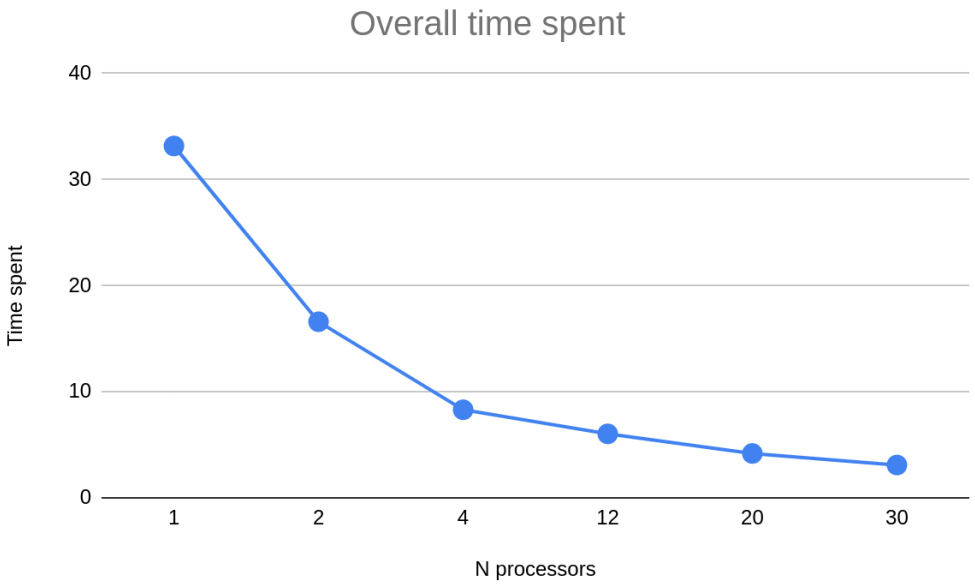
Second
- **compiling line:** `icx -fopenmp -O3 -xHost -ipo -ffast-math omp_homework.c`
- **data size:** 230000
- **number of threads:** 20 (decided by scheduler)
- **time taken:** 29.045 seconds
- **note:** just one *pragma* on the external for loop (just the external one is parallelized)

Speedup and efficiency

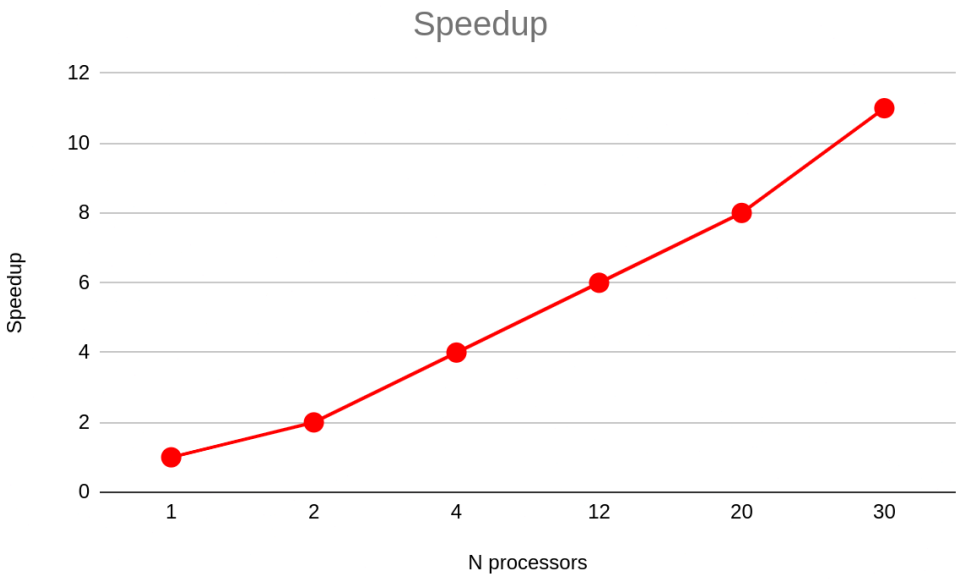
As possible number of processors, we chose:

- 1: sequential run
- 2: enable parallelization
- 4: equal to the number of performance cores
- 12: equal to the number of all cores without considering hyperthreading
- 20: the maximum value of processors considering hyperthreading up to 2
- 30: to observe any overhead

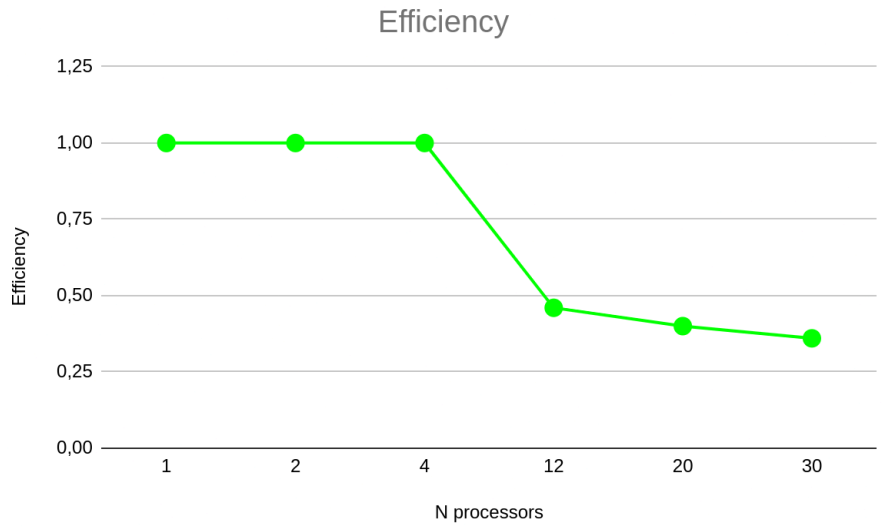
Data size	**90000**		
N processors	Time spent	Speedup	Efficiency
1	33,17	1	1,00
2	16,60	2	1,00
4	8,30	4	1,00
12	6,03	6	0,46
20	4,18	8	0,40
30	3,09	11	0,36



We observe that after enabling the parallelization with 2 processors, the time spent drastically decreases, while remaining more or less stable after 12 processors.



Here we see more or less a linear speedup increment; then after 12 obtaining a bigger speedup when exploiting the maximum number of processors.



On the other hand, the efficiency remains more or less stable up to 4 processors, then starts decreasing, observing that the percentage of decrement is bigger between the measures of 4 and 12 processors rather than the 12 and 20 measures. As expected continues to decrease also after that number.

Conclusions

We observed that, in the best sequential case, our code performs the run in about 30 seconds, on an amount of data that has to be triplicated from the original unoptimized sequential run (that performed in about 60 seconds for N=30000). So the compilation optimization flags increased by a lot the performance of our code given also by the quadratic complexity, so the result obtained with the new amount of data indeed shows a huge improvement.
By applying the *pragma* instructions to achieve better performance exploiting our processors via the OpenMP library, we observed an improvement that follows the increase of the number of threads. By just using 2 processors, we halved the time run, and by using 4 we halved again.
The optimal number of threads to be used, without losing too much in terms of efficiency is around 10: so 12 in our choices. We can get better performance by increasing that number to the maximum number of processors, that is 20, but the gain is of just a few seconds, not so relevant, also because the efficiency dramatically decreases.
Finally, by applying a number of processors greater than the one owned in the machine, we don't observe any major overhead that significantly delays the time run of our code.