CS3210 Lab 2 Report

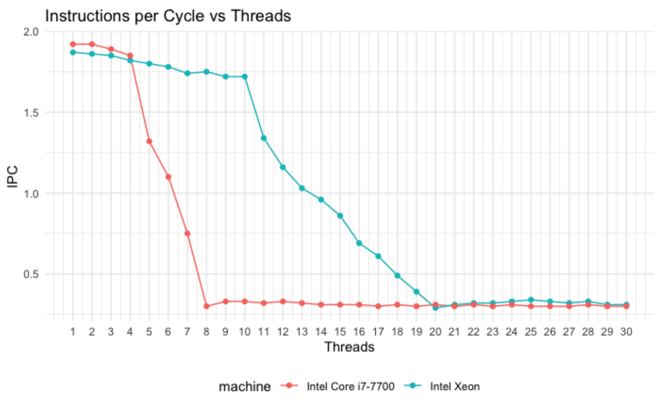
Wang Xiyu (A0282500R)

September 11, 2025

Ex11 Performance Metrics

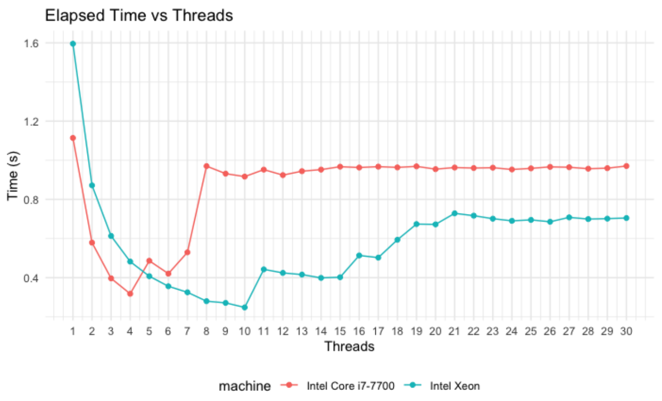
This sections shows the comparision of i7-7700(4 cores 8 threads) and Xeon Silver 4114 (10 cores 20 cores) in performance measured by instruction per cycle(IPC), wall clock time(elapsed time) and MFLOPS(Million floating point operations). Both processors have the same core architecture.

Instruction per Cycle



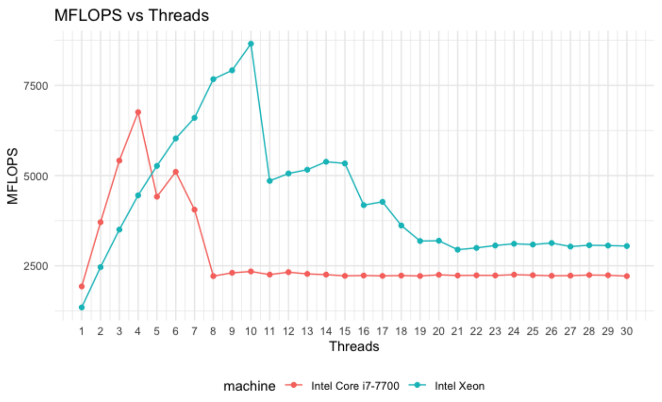
We can observe that for i7-7700 IPC moderately decreases from 1 to 4 threads, and IPC on Xeon 4114 also moderately decreases at similar rate from 1 to 10 threads, indicating that there exist overhead of thread scheduling

Wall clock time



aaa

MFLOPS



For both processors we can observe that MFLOPS peaks when the number of threads = the number of cores, this is intuitive as there are more processing units are being utilized and the number of floating point operation increases. MFLOPS starts to decrease when number of cores $\leq$ the number of threads $\leq$ the number of processing units, this is due to overheads such as memory contention, as both PUs residing in the same core share the same LLC (last level cache), causing the overall

Ex12&13 Optimization & Performance Analysis

As we can observe from the perf statistics, there are only scalar floating point operations being performed, which is a wasted opportunity for SIMD parallelism as we are accessing array elements and we can exploit the spatial locality if we can access contiguous cache in consecutive instructions. As 2D arrays are stored row-wise, wwe can access the first matrix by rows, but the second matrix is accessed by columns first, ie jumping by `B.size()`. If we can access matrix B row-wise we can utilize AVX instructions to perform packed floating point operations.

$$AB[i][j] = \sum_{k=0}^{\texttt{A[0].size()}} A[i][k] \times B[k][j]$$

We can observe that the order of the accessing order does not matter so long as we ensure in the innermost loop:

1. All elements from `A[i]` and column `i` of B...

Original

```
for (i = 0; i < size; i++)
    for (j = 0; j < size; j++)
        for (k = 0; k < size; k++)
            AB[i][j] += A[i][k] * B[k][j];
```

Optimized

```
for (i = 0; i < size; i++)
    for (k = 0; k < size; k++)
        for (k = 0; k < size; k++)
            AB[i][j] += A[i][k] * B[k][j];
```

Reproduce result

compile & perf sampling options:

```
g++ mmomp.cpp -o mmomp -fopenmp -O3
srun perf stat -r 3 -e fp_arith_inst_retired.scalar_single, fp_arith_inst_retired.128b_packed_single,
    fp_arith_inst_retired.256b_packed_single, cycles,instructions ./mmomp
g++ rowbase.cpp -o rowbase -fopenmp -O3
srun perf stat -r 3 -e fp_arith_inst_retired.scalar_single, fp_arith_inst_retired.128b_packed_single,
    fp_arith_inst_retired.256b_packed_single, cycles, instructions, ./rowbase
```