CS698L Assignment-3

Hritvik Taneja(160300)

September 17, 2019

Machine Specifications

```
CPU(s):                12
On-line CPU(s) list:   0-11
Thread(s) per core:    2
Core(s) per socket:    6
Socket(s):             1
Model name:            Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz
CPU MHz:               900.009
CPU max MHz:           4600.0000
CPU min MHz:           800.0000
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              12288K
```

Compiling and Running Instructions

All the problems have a README.txt file in their corresponding folder, which has all the compiling and running instructions.

Problem 1

Optimizations	**Time**	**Speedup**
Reference	0.949 sec	1
Loop Permutation(*p1_per.cpp*)	0.122 sec	7.77
Loop Permutation and Tiling(*p1_per_tile.cpp*)	0.111 sec	8.54
Loop Permutation and Unroll(*p1_per_unroll.cpp*)	0.089 sec	10.66

Table 1: Problem 1

```
for(int t = 0; t < Niter; t++) {
    for(int i = 0; i < N; i++) {
        for(int j = 0; j < N; j++) {
S1:             y[j] = y[j] + A[i][j]*x[i];
S2:             z[j] = z[j] + A[j][i]*x[i];
        }
    }
}
```

Statement **S2** is the main reason for poor performance, because it is accessing array **A** in a column major order. So, the first optimization for this code snippet which we refer as **Loop Permutation** is following:

```
S2:             z[j] = z[j] + A[i][j]*x[j];
```

This row major access improves the performance the most in all three optimizations. The second that optimization we try is **Loop Tiling**, we tile both the loops with a tile size of 2. And, the third that optimization we try is **Loop Unroll**, we unroll 2 iterations the ith loop.

Size of array **y** and **z** is 32K each which is same as the size of L1 cache. So, we will have a mandatory miss for every access of a cache block in each iteration of loop **i**. So, if we unroll or tile loop **i**, then the number of misses for both these array decrease by a factor of unroll/tile size. This is the major reason for increase in performance in loop tiling and unrolling. We get better performance in unrolling because tiling loop **j** does not give us any benefit and only adds overhead.

Problem 2

Optimizations	**Time**	**Speedup**
Reference	0.539 sec	1
Loop Permutation(*p2_per.cpp*)	0.440 sec	1.225
Loop Tiling(*p2_tile.cpp*)	0.345 sec	1.56
Loop Permutation and Tiling(*p2_per_tile.cpp*)	0.471 sec	1.144

Table 2: Problem 2

```
for(int i = 0; i < N; i++) {
    for(int j = 0; j < N; j++) {
        for(int k = 0; k < i+1; k++) {
S1:             C[i][j] += A[k][i]*B[j][k];
        }
    }
}
```

The main reason for poor performance of this code snippet are the cache misses because of column major access and poor reuse. So, the first optimization we tried was loop permutations, so that we could make the maximum amount of accesses column major. But since the way this product is defined we cannot make more than 2 accesses column major.

The second optimization we tried was 3D loop tiling, to improve the reuse and we indeed got a performance improvement.

In the third optimization we tried to combine both these but it lead to performance decrease due to loop overheads.

Problem 3

The code is in the file *p3.cpp*.

(a)

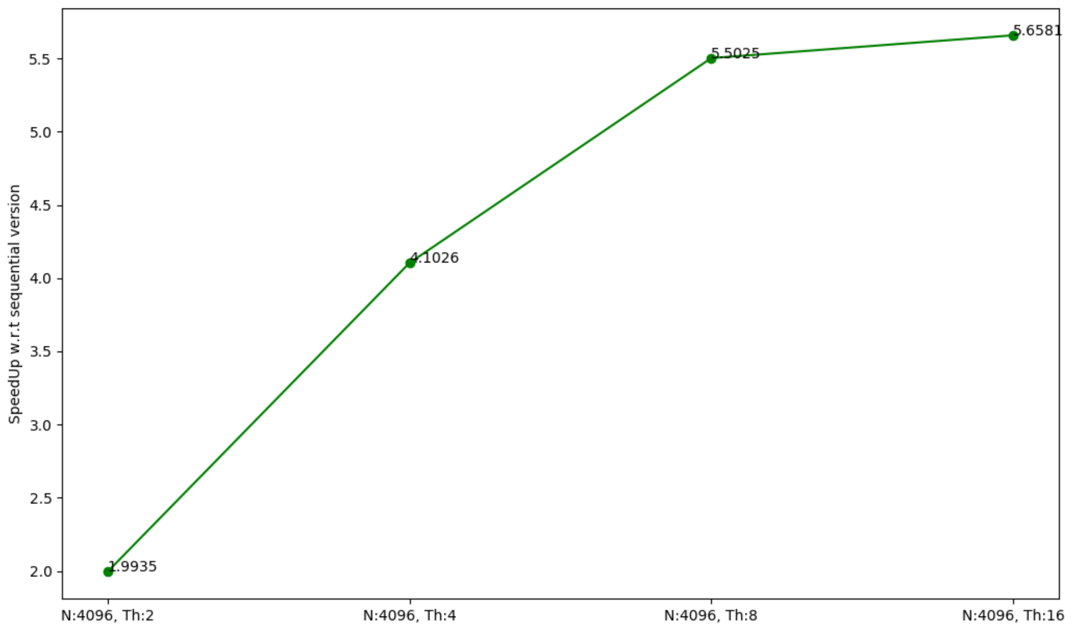

Figure 1: Strong Scaling

(b)

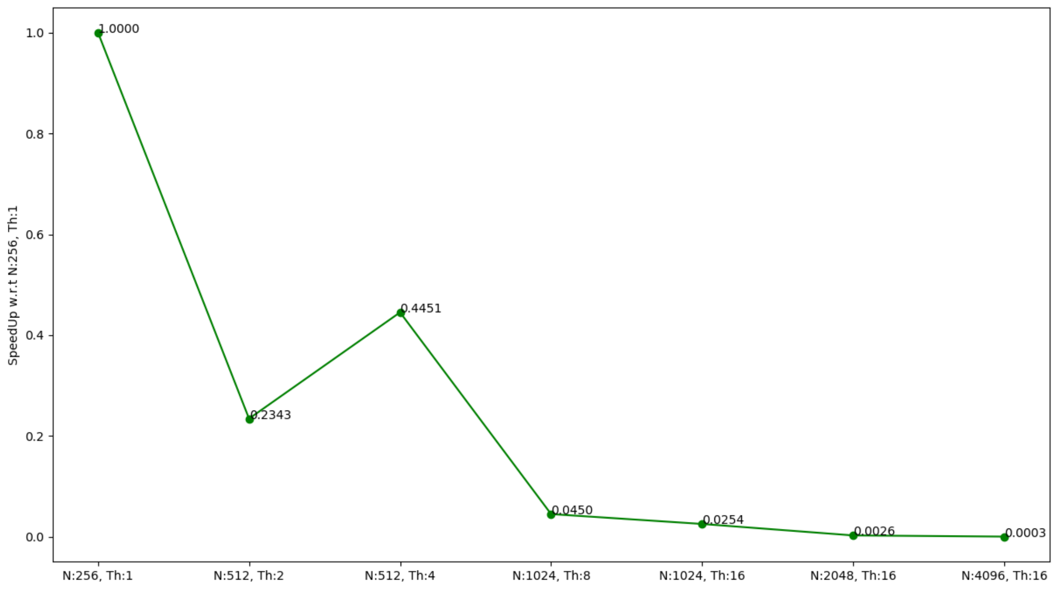

Figure 2: Weak Scaling

Problem 4

The code is in the file *p4.cpp*.