Comparing the sequential version with the two parallel versions

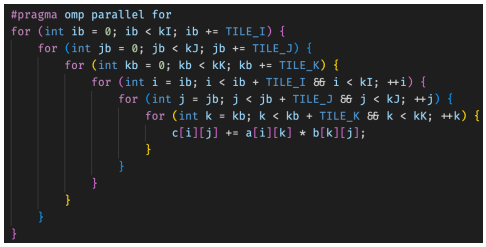
Performance	sequential	parallel	parallel blocked
1024^3	0.58 GFlops	60 GFlops	171 GFlops
2048^3	0.27 GFlops	72 GFlops	175 GFlops
4096^3	0.19 GFlops	37 GFlops	177 GFlops

for the parallel implementation, the big difference in the performance mainly come from the overhead of the cache when fetching large block. This cause a high miss rates thus lower performance.

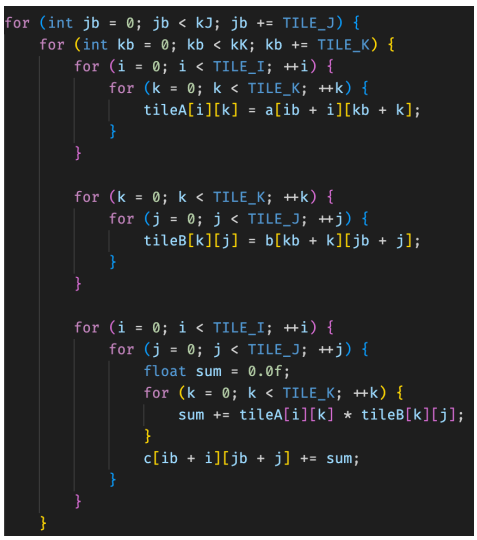
Impact of each optimization

`omp-blocked` optimizations:

1. Add 3 outer loops for traverse block by block 4 GFlops with block size 64x64x64



2. explicit caching into small tile size array of each tile is 64x64, which then result into 71 GFlops



3. Brute force on the permutation 3 of list (number power of 2 ≤ 1024) to find the best combination. Which then result in Tile for (i,j,k) = (256, 512,8) with the performance 162 GFlops
4. since the tile of k is only 8, use loop unrolling and remove unnecessary `for` statement which result in the performance of 165 GFlops

```
for (int kb = 0; kb < kK; kb += TILE_K) {
    for (i = 0; i < TILE_I; ++i) {
        tileA[i][0] = a[ib + i][kb];
        tileA[i][1] = a[ib + i][kb + 1];
        tileA[i][2] = a[ib + i][kb + 2];
        tileA[i][3] = a[ib + i][kb + 3];
        tileA[i][4] = a[ib + i][kb + 4];
        tileA[i][5] = a[ib + i][kb + 5];
        tileA[i][6] = a[ib + i][kb + 6];
        tileA[i][7] = a[ib + i][kb + 7];
    }

    for (k = 0; k < TILE_K; ++k) {
        for (j = 0; j < TILE_J; j+= 8) {
            tileB[k][j] = b[kb + k][jb + j];
            tileB[k][j + 1] = b[kb + k][jb + j + 1];
            tileB[k][j + 2] = b[kb + k][jb + j + 2];
            tileB[k][j + 3] = b[kb + k][jb + j + 3];
            tileB[k][j + 4] = b[kb + k][jb + j + 4];
            tileB[k][j + 5] = b[kb + k][jb + j + 5];
            tileB[k][j + 6] = b[kb + k][jb + j + 6];
            tileB[k][j + 7] = b[kb + k][jb + j + 7];
        }
    }
```

5. Also create a tile for tileC with loop unrolling to cache the value to write back to c afterwards, this results in 173 GFlops

```
for (i = 0; i < TILE_I; ++i) {
    for (j = 0; j < TILE_J; ++j) {
        tileC[i][j] += tileA[i][0] * tileB[0][j];
        tileC[i][j] += tileA[i][1] * tileB[1][j];
        tileC[i][j] += tileA[i][2] * tileB[2][j];
        tileC[i][j] += tileA[i][3] * tileB[3][j];
        tileC[i][j] += tileA[i][4] * tileB[4][j];
        tileC[i][j] += tileA[i][5] * tileB[5][j];
        tileC[i][j] += tileA[i][6] * tileB[6][j];
        tileC[i][j] += tileA[i][7] * tileB[7][j];
    }
}
```

```
for (i = 0; i < TILE_I; ++i) {
    for (j = 0; j < TILE_J; j+= 8) {
        c[ib + i][jb + j] += tileC[i][j];
        c[ib + i][jb + j + 1] += tileC[i][j + 1];
        c[ib + i][jb + j + 2] += tileC[i][j + 2];
        c[ib + i][jb + j + 3] += tileC[i][j + 3];
        c[ib + i][jb + j + 4] += tileC[i][j + 4];
        c[ib + i][jb + j + 5] += tileC[i][j + 5];
        c[ib + i][jb + j + 6] += tileC[i][j + 6];
        c[ib + i][jb + j + 7] += tileC[i][j + 7];
    }
}
```

6. set the number of threads to be 4, the performance increases to 177 GFlops

Scalability of optimized code with different numbers of threads

threads	Performance - GFlops	Speedup
1	44	1
2	89	2
4	176	4
8	175	4

The speed up for doubling the number of thread is roughly x2. However, for the m5.2xlarge, the speedup doesnt change even with the number of thread is 8 most likely due to the fact that it only has 4 cores. so that ultilizing more threads in with the same number of cores doesnt extends the cache. Thus the performance doesnt increases.

Discussion

- Parallelism increase the throughput for such problem of large problems, even naiive parallelism.
- Pseudo big O sometimes not a good measurement to evaluate the practical speed.
- Each problem has different approaches, the biggest optimization for this problem is adding tile to fit into the cache and choose its dimensions to fit the cache perfectly to increase cache hit.

- optimized scalability is limited by cache size on the cores.