ENSC 453 SFU

Lab 1: Optimizing Matrix Multiplication Using Parallelization

Liam Feng

301 463 567

Provided C++ code for matrix multiplication was parallelized it to run on the 14-core CPU (20 hyper-threads) in our FAS-RLA lab using OpenMP. **FAS-RLA lab machines** use the Intel(R) Core(TM) i5-13500 CPU. The i5-13500 CPU has 6 performance cores (P-cores) which are hyper-threaded and 8 efficient cores (E-cores) which are not hyper-threaded, resulting in 20 hyper-threads in total.

Code:

The core computation of matrix multiplication is: **C = alpha*A*B + beta*C**, where A, B, and C are **2048*2048 size matrices**, and alpha and beta are two constants. The core code is in the following function in mm.cpp file:

```
#pragma omp parallel for num_threads(8) private(i,j,k)
  for (i = 0; i < NI; i++) {
    // #pragma omp parallel for num_threads(20)
    for (j = 0; j < NJ; j++) {
      C[i*NJ+j] *= beta;
      for (k = 0; k < NK; ++k) {
        C[i*NJ+j] += alpha * A[i*NK+k] * B[k*NJ+j];
      }
    }
  }
}
```

Parallelization of the outer loop (for (i = 0; i< *NI; i++)):*

"#pragma omp parallel for num_threads(x) private(i,j,k)" was added right before the outer loop of the matrix multiplication, with x being the number of threads used. Results for the usage of 2,4,6…20 threads were obtained by running the program with each number of threads 5 times and then recording the average execution time.

Two metrics are of importance: the speedup and parallelization efficiency. The speed up is the raw performance gain from adding more threads, and is found with the following equation:

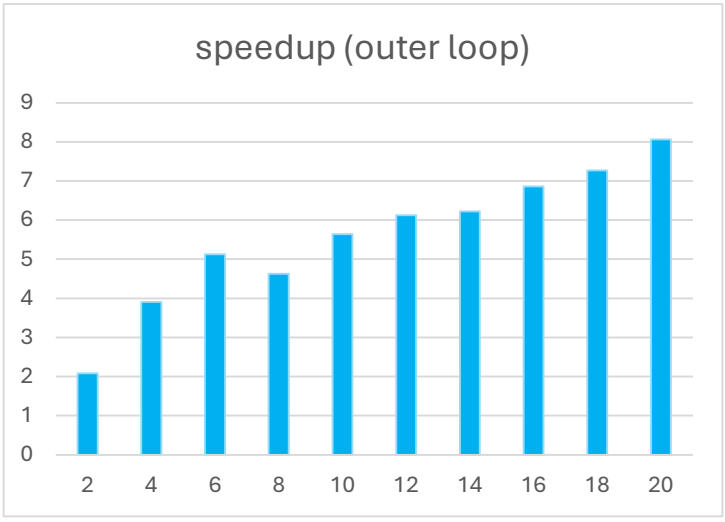
$$Speedup(p) = \frac{T(1)}{T(p)}$$

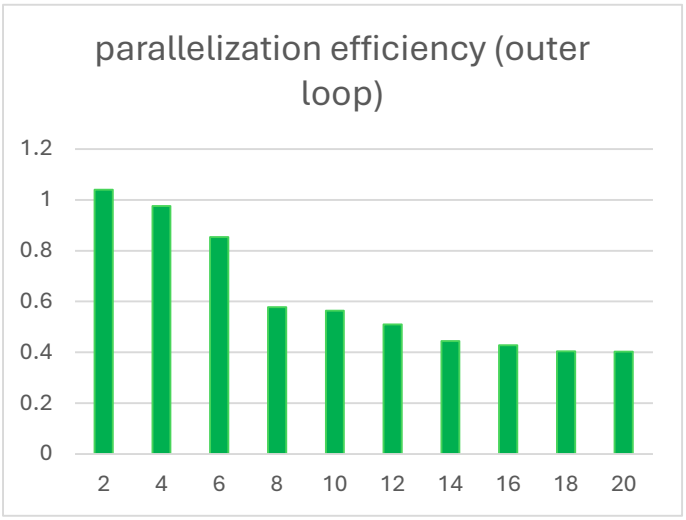
Where T(1) = time taken with one thread and T(p) = time taken with ***p*** threads

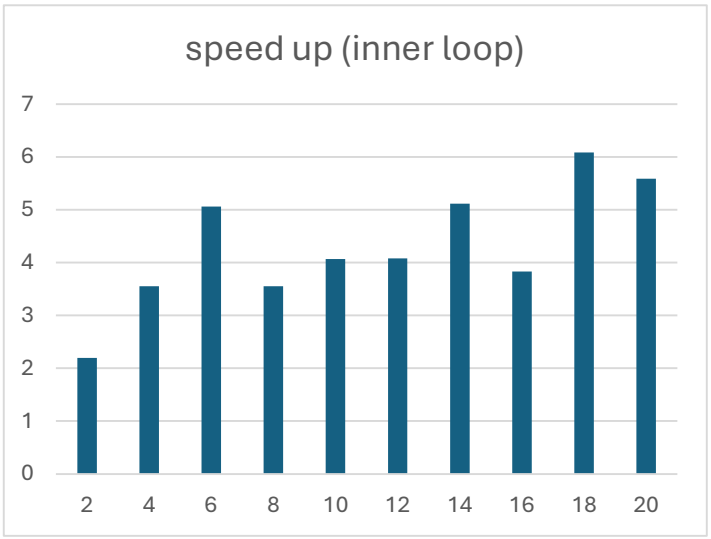
The parallelization efficiency is the *normalization* of the speedup by the number of threads and measures how effectively the threads are being used:

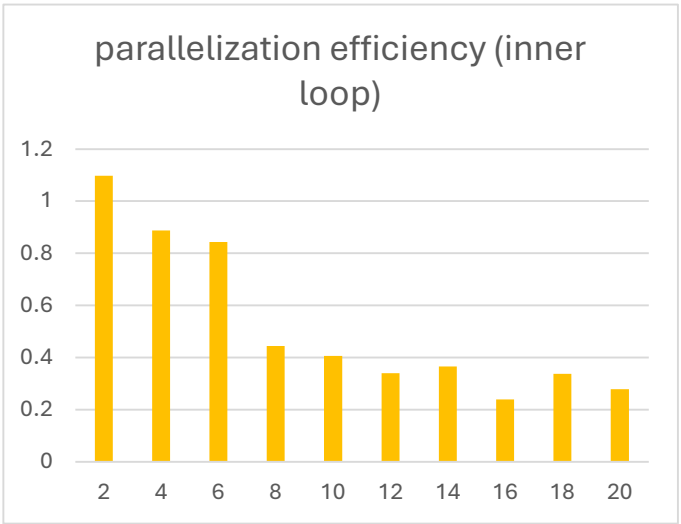
$$Efficiency(p) = \frac{Speedup(p)}{p}$$

Recorded results are shown in the bar graphs below:









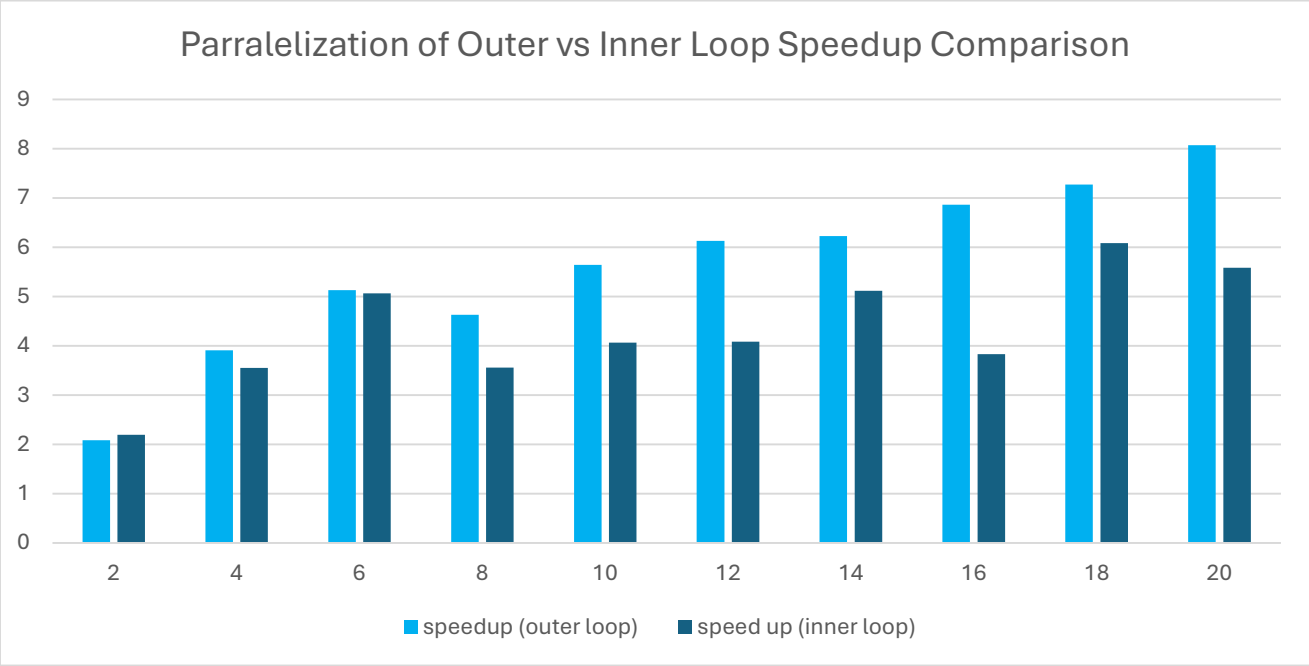
Parallelization of the outer loop:

From the results collected, we see a consistent inverse relationship between the speedup of adding more threads and the parallelization efficiency of each thread. As more threads are added, the total amount of work required in the matrix multiplication remains the same, and so each thread will be responsible for a smaller number of tasks. The constant overhead cost associated with adding a thread will take an increasingly greater portion of the total run time, which will lower the parallelization efficiency as more threads are added.

Parallelization of the inner loop:

While the overall effects of adding threads are *like* the parallelization of the outer loop, the speedup effects are noticeably less consistent. Firstly, certain thread counts are noticed to cost a HIGHER runtime than a lower number of threads (example: 6 vs 8 threads). This likely relates to the hardware differences between the performance/efficiency cores. When 6 threads are called, it is likely that all 6 threads are run on the performance cores, allowing computations to be quite fast. But when 8 threads are called, the 7^{th} and 8^{th} threads will land on the efficiency cores, which may be slower, narrower and have a lower memory bandwidth, causing them to be the bottlenecks. Even with the performance cores finishing quickly, they must wait for the efficiency cores to finish, slowing down the total execution time.

Final Speedup Comparison



There is evidence to suggest that parallelization of the outer loop yields stronger speedup results compared to adding threads in the inner loop. There are a few likely explanations for why this is the case:

- When the #pragma omp parallel for command is called inside the loop, the forking/joining of threads is required to be done for every iteration of the outer loop, which runs NI = 2048 times. This explicitly raises the overhead costs of adding threads significantly compared to parallelizing the outer loop, which only requires threads to be forked once at the beginning, and joined once at the end when the whole computation is finished.
- Parallelizing the inner loop divides the j iterations among threads, so each thread works on different columns of matrix B while sharing the same row of A. Because the columns of B are stored in a striped pattern in row-major layout, each thread ends up reading memory that is far apart, which hurts cache locality and increases memory-bandwidth pressure.
- Parallelizing the outer loop, on the other hand, assigns different rows of A and C to different threads. All threads still read the same columns of B, but they do so in a consistent, sequential pattern that the cache hierarchy handles

well. This improves spatial locality, reduces contention, and leads to faster runtimes.

Additional Notes:

It was realized that the two inner j loops could be merged into one for loop to prevent the program from making two entire scans of the C matrix. This improves the runtime for free without changing the results. Therefore, all experiments done in this lab were done with this added change.

Base Code:

```
for (i = 0; i < NI; i++) {
    for (j = 0; j < NJ; j++) {
      C[i*NJ+j] *= beta;
    }
    for (j = 0; j < NJ; j++) {
      for (k = 0; k < NK; ++k) {
        C[i*NJ+j] += alpha * A[i*NK+k] * B[k*NJ+j];
      }
    }
  }
}
```

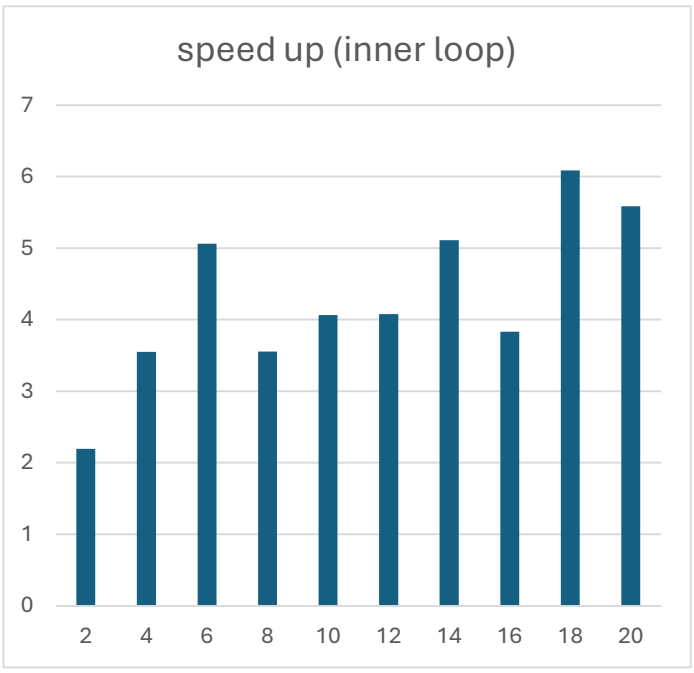
With the j loop merge:

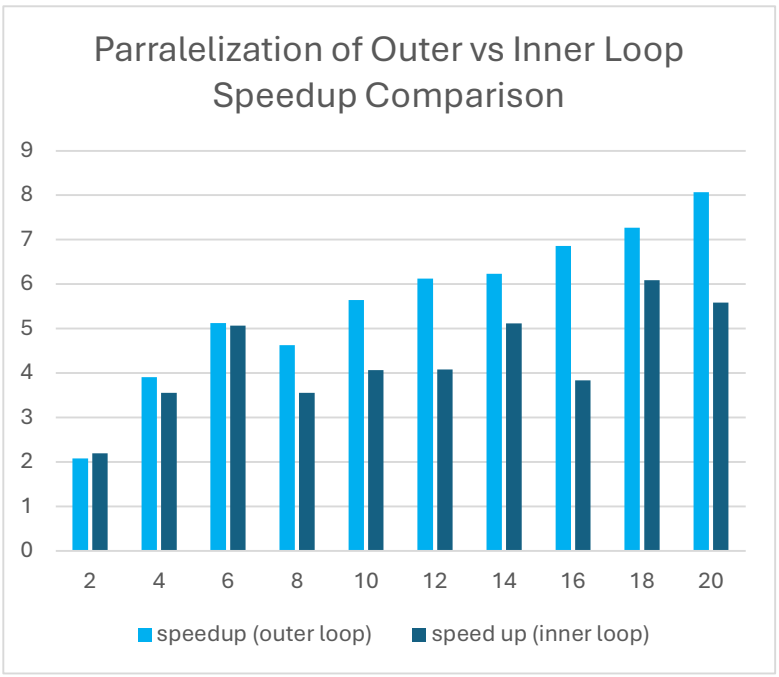
```
for (i = 0; i < NI; i++) {
    for (j = 0; j < NJ; j++) {
      C[i*NJ+j] *= beta;
      for (k = 0; k < NK; ++k) {
        C[i*NJ+j] += alpha * A[i*NK+k] * B[k*NJ+j];
      }
    }
  }
}
```

It should also be noted that in this lab, parallelizing the inner loop involved running the **#pragma omp parallel for num_threads ()** command right before the j loops, because this was the method that imitates the one presented in our second lecture slides. However, it was noticed that runtime could be improved on the inner loop parallelization by initializing one team of threads outside the outer loop, and then distributing the j loops among them afterwards:

```
#pragma omp parallel num_threads(20)
{
    for (int i = 0; i < NI; i++) {
        #pragma omp for nowait
        for (int j = 0; j < NJ; j++) {
            C[i*NJ+j] *= beta;
            for (int k = 0; k < NK; k++) {
                C[i*NJ+j] += alpha * A[i*NK+k] * B[k*NJ+j];
            }
        }
    }
}
```

The improved results with this change are shown below:





Changes to the main program:

To provide convenience and accurate results, the main program was changed to have the user select whether the outer/inner loop should be parallelized, along with prompting the user to enter the number of threads to use. There will then be 5 execution runs, and the average execution time will be computed by the end of every run.

```
Choose parallelization strategy:
1 = Parallelize OUTER loop (i-loop)
2 = Parallelize INNER loop (j-loop)
Enter choice: 2
Enter number of threads: 8
kernel execution: 4.792908451
kernel execution: 4.827058815
kernel execution: 4.874923339
kernel execution: 5.022675402
kernel execution: 5.030066492
sum of C array = 3212133376.000000
average time cost: 4.909527
[lfa32@fas-rla-06 lfa32]$
```