**CS 521 ML & Compilers Spring 2025 – MP1**

**Student Name: Joyce Au**

**NetID: joyceau2**

**Part 1 CPU**

1.2) **Ablation Study:**

***Insight:*** For the larger matrices (Size =1000), each optimization yields a greater speedup over the naïve version, reflecting that tilting and parallelization pay off more at a significantly greater scale.

1.3) **Scaling Study:**

***Insight:*** For smaller matrix sizes such as 100, there is not much of a speedup difference between After O1 and O1-O4. However, as the size grows larger beyond 1000, the fully optimized code (O1-O4) scales a lot better. This shows that utilizing parallelization and vectorization can significantly lower runtime than just loop reordering due to its optimizations in cache-blocking, parallel loop scheduling and vectorization.

**Part 2 GPU**

2.2) **Ablation Study:**

***Insight:*** In contrast to the CPU test, the kernel optimizations for the GPU in O1-O3 are much more pronounced compared to the naïve version, as evident in dramatic speedups in the chart above. The naïve version is so slow for both matrix sizes that any GPU optimized kernel results in a large speedup factor. For size = 100, the speedup for O1 alone is slightly larger than O1 + O2 and O1 + O2 + O3 at 3796ms vs 3604ms vs 3313ms respectively. This shows that the overhead for the optimizations such as tiling and parallelization outweighs the benefits when the matrix is small. Conversely, for size = 1000, the speedup in O1 + O2 is better than O1 + O2 + O3, which is in line with the time results obtained when experimenting in O3, where I discovered that my initial implementation of a block size of 16x16 is superior to 16 x 32 (which is what is recorded in O3), as well as 8 x 8 and 32 x 32 (results are in the comments for O3). In general, this indicates that with larger matrix sizes, the benefits of the optimizations outweigh the overhead.

2.3) **Scaling Study:**

***Insight:*** For smaller matrices (e.g. 100 x 100 x 100), the extra overhead from tiling optimizations performs less optimally compared to after O1-O3 and is slightly slower. However, as the matrix size grows beyond 1000, the tiling and parallelization becomes significantly better than just O1 alone. This shows that the overhead of the optimizations is negligible at larger matrix sizes when compared to the better data reuse and parallel efficiency.

2.4) **Extra Credit:**

N = M = K = 125

Time taken for GEMM (GPU, gemm\_gpu\_o1): 0.0091136ms

Time taken for GEMM (GPU, gemm\_gpu\_o2): 0.0089056ms

Time taken for GEMM (GPU, gemm\_gpu\_o3): 0.0103488ms

Time taken for GEMM (GPU, gemm\_cublas): 0.0119808ms

N = M = K = 1000

Time taken for GEMM (GPU, gemm\_gpu\_o1): 0.951552ms

Time taken for GEMM (GPU, gemm\_gpu\_o2): 0.748576ms

Time taken for GEMM (GPU, gemm\_gpu\_o3): 0.791059ms

Time taken for GEMM (GPU, gemm\_cublas): 0.130374ms

My o2 kernel is the more optimized version. Based on these results, my version is slightly faster for smaller matrices, likely due to the overhead that cuBLAS has. However, for larger matrices, cuBLAS outperforms my approach as it is more optimized for the GPU. To improve my solution for larger matrices, I could consider more advanced tiling strategies, such as splitting tiles further or multiple shared levels of memory/cache.