Machine Project I

CS 521 — Machine Learning and Compilers
Spring Semester 2025

Student Name: Mingjun Liu **NetID: mingjun6**

1 CPU

1.1 Ablation Study

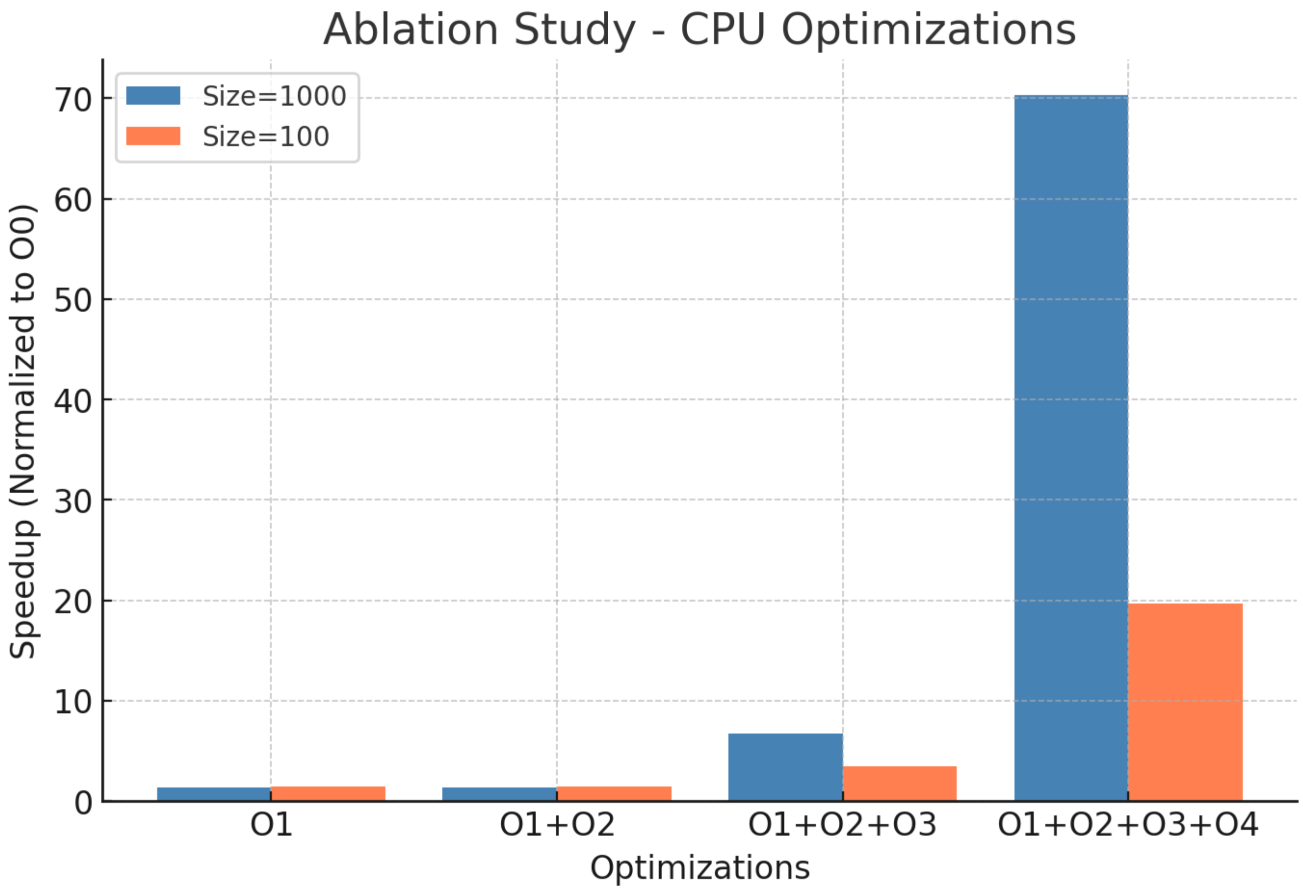

Figure 1: Ablation: Speedup of Optimizations on CPU

Insight: The ablation study demonstrates the progressive impact of different optimization techniques on CPU matrix multiplication. The optimized loop ordering improved cache locality, while tiling significantly reduced memory access overhead. OpenMP parallelization enabled multi-core utilization, further accelerating computation. Finally, FMA instructions ensured efficient floating-point operations, yielding a substantial performance gain, especially for larger matrices.

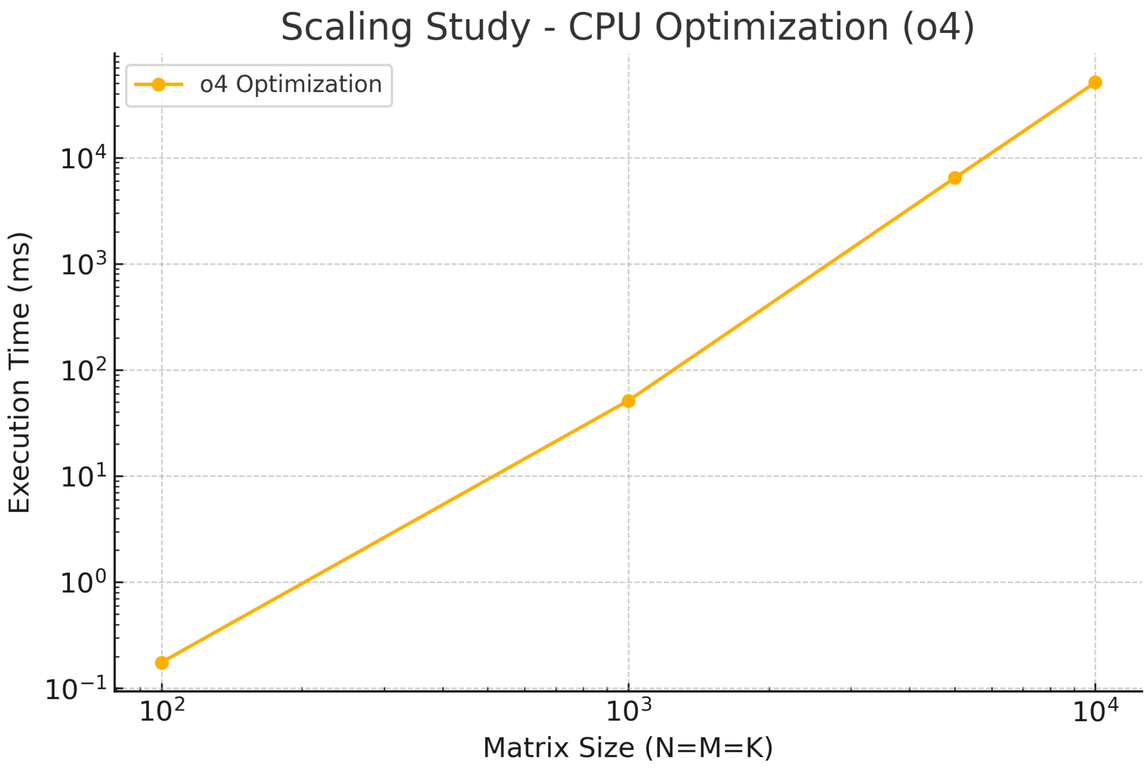

Figure 2: Scaling: Speedup of Optimizations on CPU

1.2 Scaling Study

Insight: The scalability analysis confirms that optimized implementations scale effectively with matrix size. As matrix dimensions grow, parallelization and vectorization provide consistent speedups, though diminishing returns are observed due to increasing memory bandwidth constraints.

2 GPU

2.1 Ablation Study

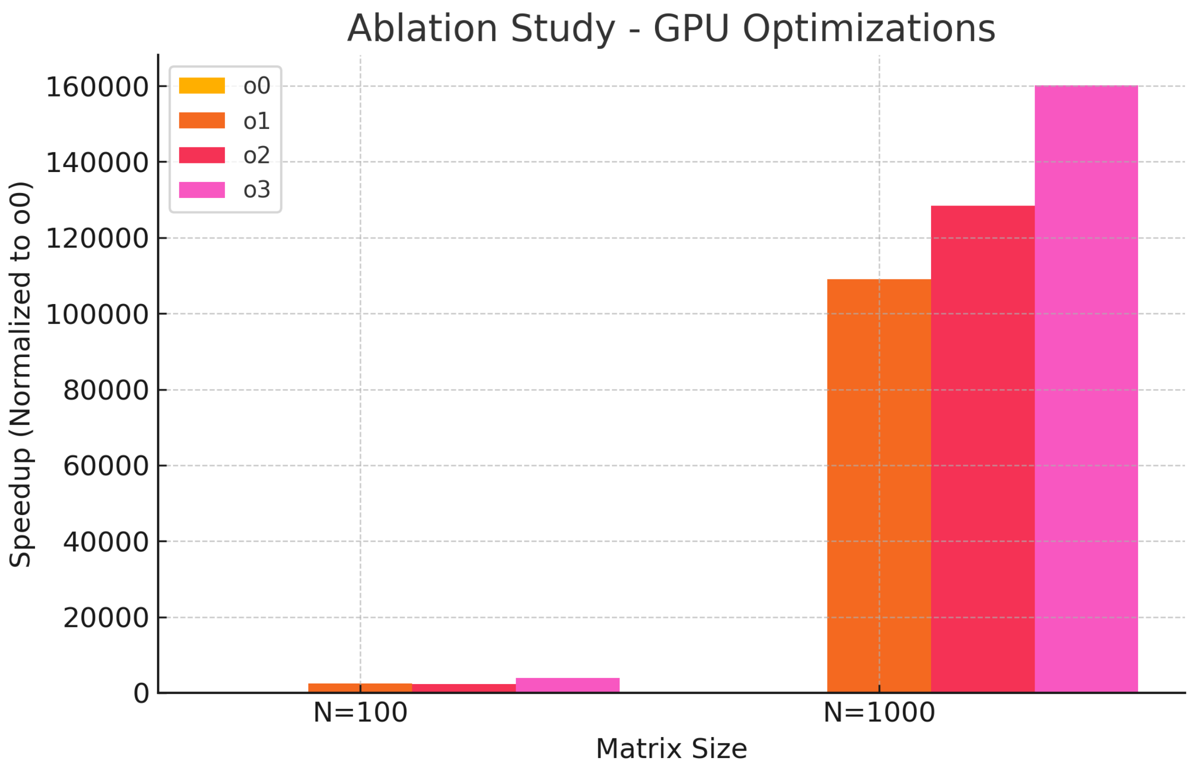

Figure 3: Ablation: Speedup of Optimizations on GPU

Insight: The GPU ablation study highlights the significant performance gains achieved through progressive optimizations. Kernel parallelization across streaming multiprocessors led to a drastic reduction in execution time. Implementing tiling with shared memory further improved memory efficiency. The final optimization step reduced computation time even more by ensuring memory coalescing and efficient workload distribution.

2.2 Scaling Study

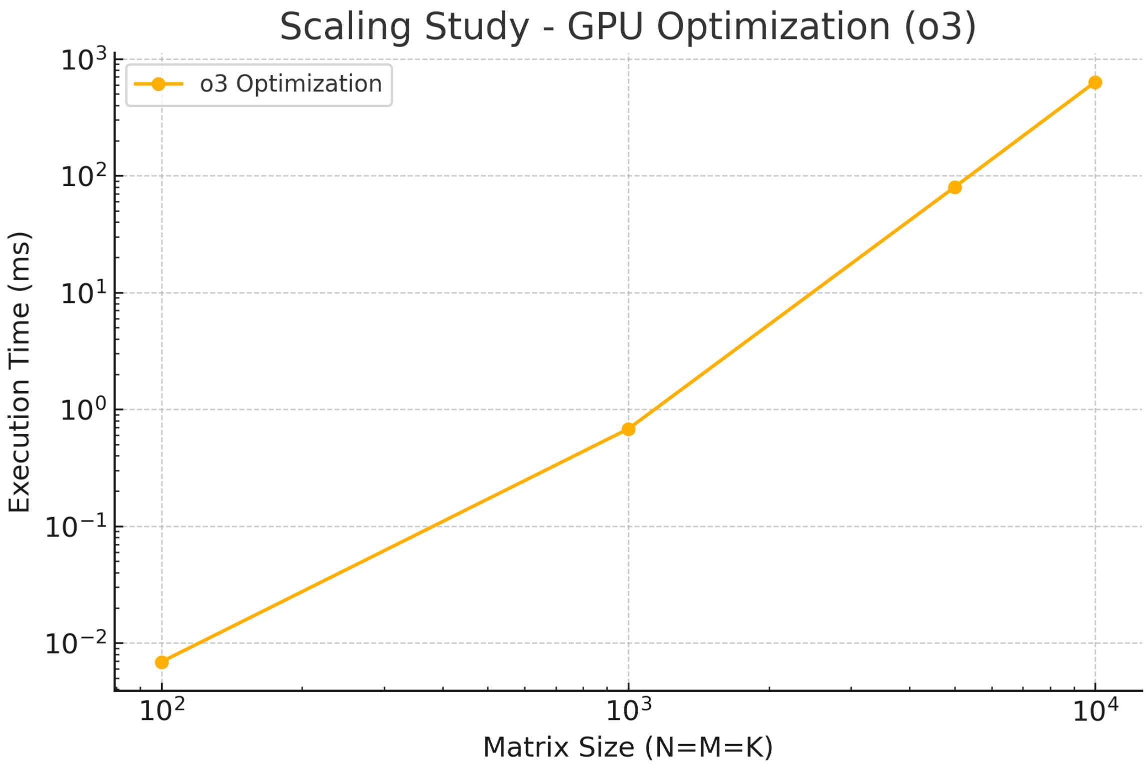

Figure 4: Scaling: Speedup of Optimizations on GPU

Insight: The scalability study demonstrates the effectiveness of GPU optimizations as matrix size increases. The performance scales well across different matrix sizes, showcasing the GPU's capability to handle larger workloads efficiently.

2.3 cuBLAS Comparison

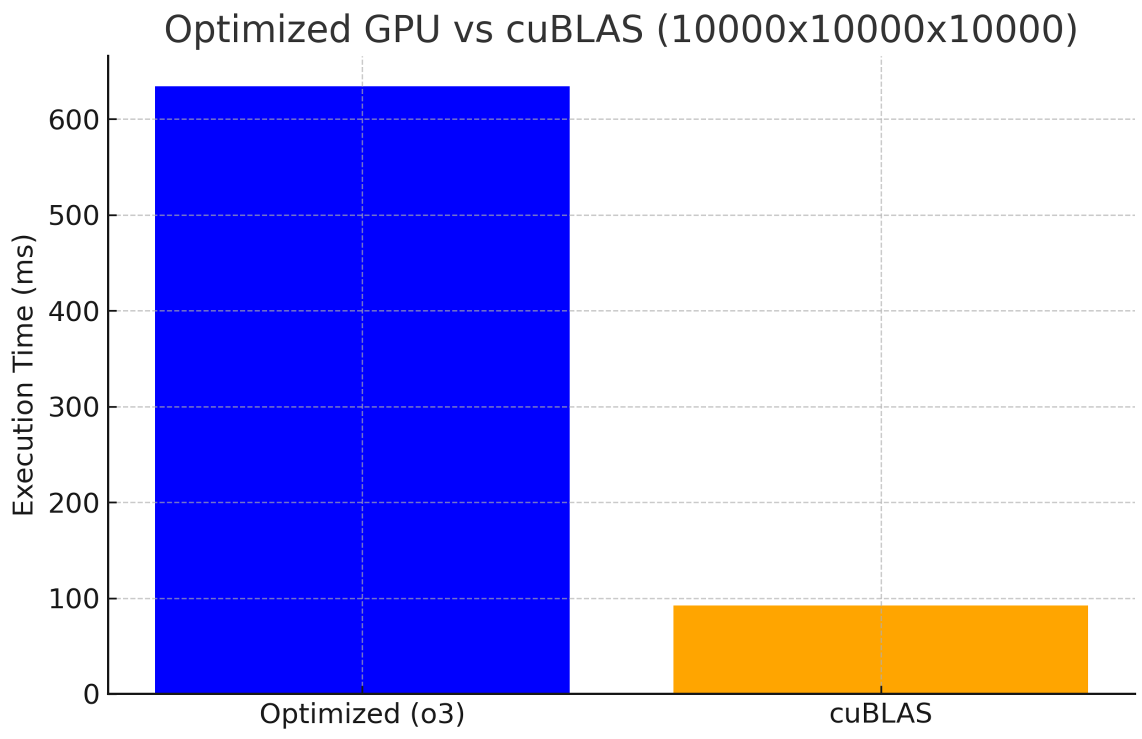

Figure 5: Comparison: Optimized GPU vs cuBLAS

Insight: The comparison between the optimized GPU implementation and cuBLAS reveals that cuBLAS significantly outperforms the manually optimized kernel. This result emphasizes the efficiency of vendor-optimized libraries, which leverage hardware-specific optimizations for peak performance. Additional improvements could be achieved by further tuning memory access patterns and utilizing warp-level primitives.

3 Trainium

3.1 Implementation

The function takes in an input tensor (X), a weight tensor (W), and a bias vector to compute the convolution output (X_out). It follows the convolution in the instruction document(no padding, stride of 1), and since the computation is large then exceed the limit of NKI, I had to tile the work into smaller chunks so it fits better in memory and runs faster.

3.2 Optimizations Implemented

- Tiling for better efficiency
  - Split input and output channels into tiles to help reuse memory instead of loading the same values multiple times. Just like what I did for previous part.
  - Divided the output height into blocks to process in chunks, reducing redundant computation.
- Managing memory properly
  - Used SBUF to store bias and weights, making accesses much faster than constantly pulling from HBM.
  - Stored the input in HBM to make data loads more efficient.
- Preloading and transposing weights
  - Pre-transposed weights to match matrix multiplications. I was struggled at this point for 2 days, thanks for the TA point out I can do the transpose early.
  - Used `nisa.nc_transpose()` to efficiently handle the transformation and improve memory access patterns.
- Utilizing PSUM for accumulation
  - Initially used SBUF for accumulation but later corrected to use PSUM for more efficient intermediate result accumulation.
- Handling input reshaping efficiently
  - Since NKI does not allow direct reshaping, manually tiled and indexed data instead of relying on `reshape()`.
- Efficient computation structure
  - Looped over batches, output tiles, and input tiles in an optimized way to reduce redundant operations.

3.3 Profiling and MFU Achieved

We conducted profiling using Trainium's built-in profiling tools and observed the following MFU (Matrix Function Utilization) results:

- float16 Precision: Achieved MFU of **65.91%**.
- float32 Precision: Achieved MFU of **29.38%**.

3.4 LLM log

https://chatgpt.com/share/67be405c-54e0-800a-9c9d-8125d3eb946c
Used LLM for grammar & wording improvement & plot generate. Unable to share due to OpenAI does not allowed user to share chat including pictures.