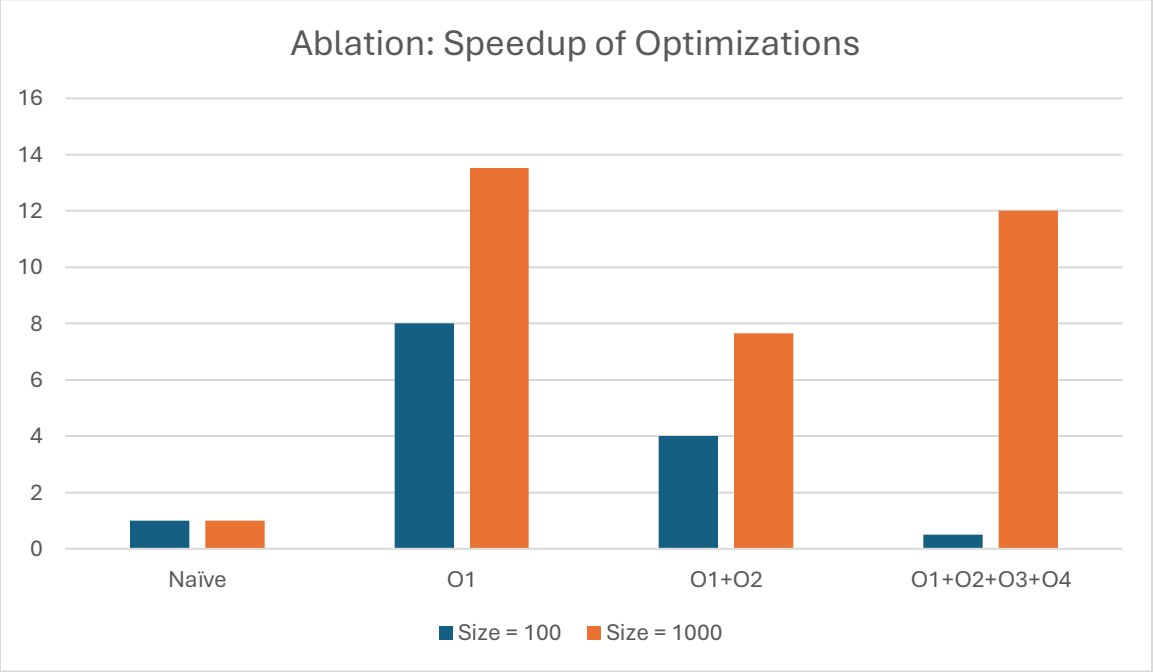
CS 521 ML & Compilers Spring 2025 – MP1

Student Name: **Joyce Au**

NetID: **joyceau2**

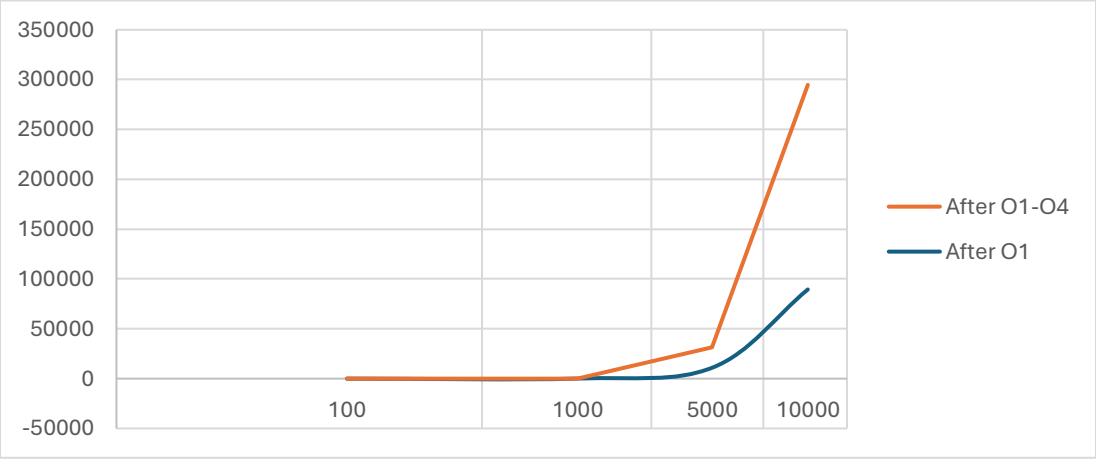
Part 1 CPU

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.

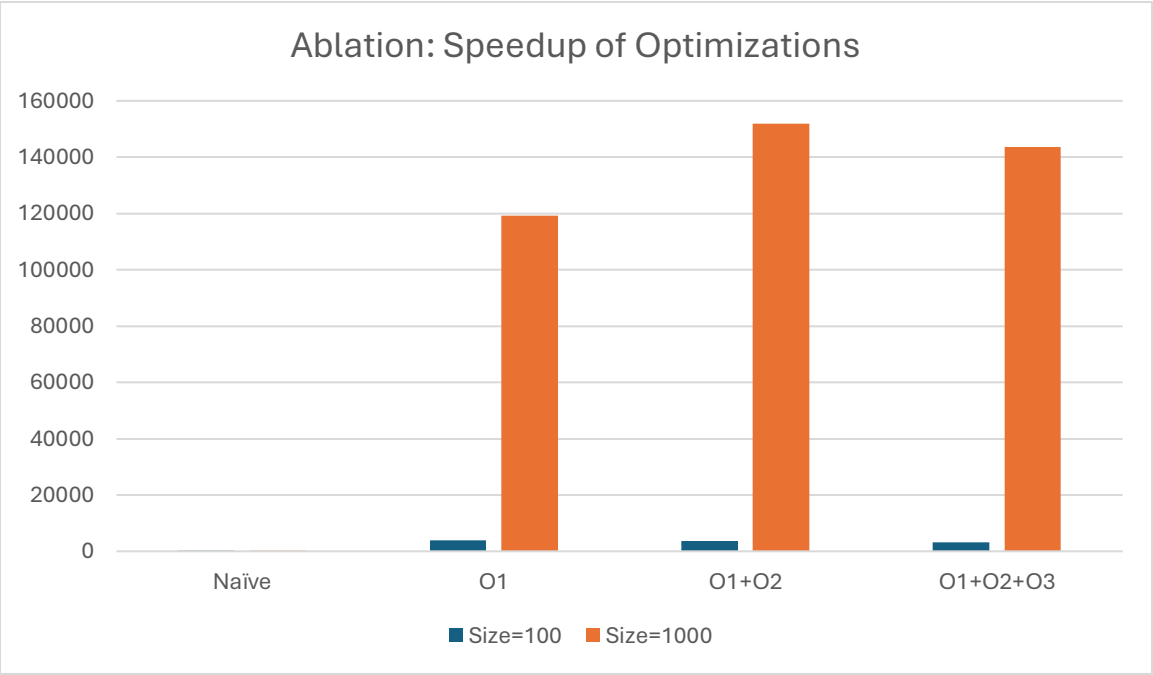
1.3) **Scaling Study:**



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

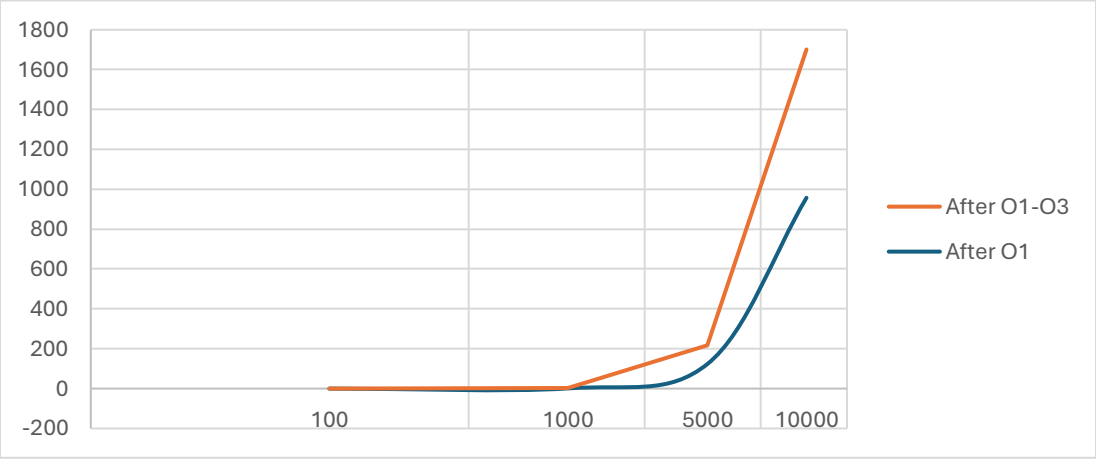
2.2) **Ablation Study:**



**note: my o2 is the most optimized and not o3*

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.

Part 3 NPU:

3.1) **Implementation**:

The overall algorithm of my implementation is as followed:

1) Reshape and tile weights into a 6D tensor
2) Allocate space for weights & load from HBM to SBUF
3) Process the images in batches
4) Allocate space for all chunk rows in SBUF
5) Loop over chunks and load each chunk's rows from HBM, using nl.mgrid to generate row/column indices for the chunk and masking for boundaries
6) Accumulate partial sums by looping over input channel tiles and filter height/width offsets.
7) Add bias and store output in HBM using masked stores to handle leftover rows

Chunk-based convolution:

In my initial iteration, I followed the overall pseudocode for the algorithm in the readme and processed the images in one pass without chunking. However, this did not work for larger images due to on-chip memory constraints. Thus, chunk-based convolution was implemented, using the readme's chunking section and AWS documentation on NKI tensor indexing as guidance. I divided the input image into row-chunks that fit into the SBUF. After loading each chunk into the SBUF, partial-sum convolution was performed on that chunk before moving to the next.

Weight Tiling:

My convolution filter weights are reshaped and loaded into SBUF as a 6D tensor of shape:

(n_tiles_c_out, nl.par_dim(c_out_pmax), n_tiles_c_in, c_in_pmax, filter_height, filter_width)

This format enforces a maximum of 128 for the partition dimension required by Tranium's architecture.

Matrix Multiplication for Convolution:

Each chunk is processed via the matmul operation, and partial sums are computed via repeated matrix multiplications. Results are accumulated in PSUM memory to avoid frequent write backs to HBM.

Bias Addition and Storing Results

Once partial sums are computed, the bias vector is added, and final results are written into HBM. Masking is used to handle leftover rows that do not fill the chunk, following the AWS documentation on NKI masking.

3.2) **Optimization of implementation:**

1. Adaptive Chunk Size: As aforementioned, chunking was the first optimization needed to be able to process larger images. In my first iteration of chunking, I used a chunk size of 16. However, as I ran the tests, I got an error and realized that the chunk size was too large, and decreased if to 8, then 4 and finally settled at 2. Later, I decided to try to implement adaptive chunk sizes based on the input height. I introduced an if/else condition with varying heights which I tinkered with until they seemed about right. While doing this, I realized that there were cases where it did not fit the PSUM dimension. Thus, my final iteration was to do an adaptive chunk size calculation, with a final clamp based on the PSUM dimension which is 512/ out_width. This approach balances overhead from too many small chunks while also avoiding PSUM buffer overflow.

2. Accumulating PSUM On-Chip: The partial sums from each input-channel are accumulated in the PSUM buffer to reduce the number of times data is written back to HBM. This helps to lower memory traffic and keeps the matrix pipelines active.

3. Affine Loops and weight space allocation: I had originally thought it would be better to allocate the space for weights per chunk. With this, I needed to use sequential_range when looping over the chunks. However, this was causing a dip in performance. After much trouble trying to optimize my performance and reading more on the difference between the two loops, I realized it would be better to allocate space for all the weights in SBUF ahead of time and used affine_range to process the chunks, enabling more parallelism.

4. **mGrid Indexing and Masking:** To load each chunk's row efficiently and solve out-of-bounds errors I was getting initially, I read up more on matrix multiplication on the AWS documentation and implemented mgrid to generate the row and column indexes. I then applied a mask derived from global_row < input_height to enforce boundaries.

3.3) **Profile Report**

```
(aws_neuronx_venv_pytorch_2_5) ubuntu@ip-172-31-57-249:~/cs521_mp1/npu$ python test_harness.py --profile part3_profile
Running correctness test for conv2d kernel with smaller images...Passed 😎
Running correctness test for conv2d kernel with larger images...Passed 😇
Running correctness test for conv2d kernel with larger images + bias...Passed 🤩
Comparing performance with reference kernel (float32)...

file_float32
+---+----+---------+---------+---------+-------+-------+------+------+------+-------+-------+-------+-------+
  B   NC   NC USED   WEIGHTS   MODE      INF/S   IRES/S   L(1)   L(50)   L(99)   NCL(1)   NCL(50)   NCL(99)   %USER
  1   1    1         dynamic   LIBMODE   76.05   76.05    3812   3843    3875    3767     3769      3771      N/A
+---+----+---------+---------+---------+-------+-------+------+------+------+-------+-------+-------+-------+


Execution Time for student implementation: 3771 µs
Performance test passed 🤩


NEFF / NTFF files generated with names: part3_profile_float32.neff, part3_profile_float32.ntff
Comparing performance with reference kernel (float16)...

file_float16
+---+----+---------+---------+---------+-------+-------+------+------+------+-------+-------+-------+-------+
  B   NC   NC USED   WEIGHTS   MODE      INF/S   IRES/S   L(1)   L(50)   L(99)   NCL(1)   NCL(50)   NCL(99)   %USER
  1   1    1         dynamic   LIBMODE   203.10  203.10   1095   1115    1126    1046     1047      1049      N/A
+---+----+---------+---------+---------+-------+-------+------+------+------+-------+-------+-------+-------+


Execution Time for student implementation: 1049 µs
Performance test passed 🤩


NEFF / NTFF files generated with names: part3_profile_float16.neff, part3_profile_float16.ntff
```

Layer Summary for float32:

Name	Start	End	Duration	Tensor Engine MFU	Tensor Engine HFU	Tensor Engine Active Percent	Tensor Engine Active Time	Tensor Engine Instruction Time	Tensor Engine Instruction Count	Tensor Engine Flop Count
/sg00	58 ns	3.78 ms	3.78 ms	33.55 %	43.22 %	98.94 %	3.74 ms	9.04 ms	33046	149.797 GFLOPS
/sg00/Unknown	12.42 us	3.77 ms	3.75 ms	33.8 %	43.54 %	99.56 %	3.74 ms	9.03 ms	32966	149.797 GFLOPS

Layer Summary for bfloat16:

Name	Start	End	Duration	Tensor Engine MFU	Tensor Engine HFU	Tensor Engine Active Percent	Tensor Engine Active Time	Tensor Engine Instruction Time	Tensor Engine Instruction Count	Tensor Engine Flop Count	Tensor Engine Model Flop Count
/sg00	58 ns	1.06 ms	1.06 ms	59.8 %	77.03 %	93.45 %	0.99 ms	2.86 ms	16920	74.898 GFLOPS	58.138 GFLOPS
/sg00/Unknown	12.18 us	1.05 ms	1.04 ms	61.34 %	79.03 %	95.54 %	0.99 ms	2.86 ms	16840	74.898 GFLOPS	58.138 GFLOPS

Insight: MFU measures how effectively the kernel is saturating the Trainium's matrix multiplication pipelines. Under float32, my MFU utilization was around 33-34%. This

shows that the hardware's matrix pipelines were actively busy one-third of the time, while the remaining two-thirds was likely spend waiting on data transfers, or processing overhead from chunking or tiling. Conversely, with bfloat16, my MFU utilization was significantly higher at around 60-61%. This shows that the matrix pipelines were doing more work each cycle and stalled less often, illustrating a more effective pipeline saturation. The half-precision mode enabled the kernel to handle more parallel operations, resulting in a higher percentage of cycles where matrix units are fully busy.

References:

Works Cited

Cabrera, Fang. "The CUDA Parallel Programming Model - 5. Memory Coalescing - Fang's Notebook." *The CUDA Parallel Programming Model - 5. Memory Coalescing - Fang's Notebook*, nichijou.co/cuda5-coalesce. Accessed 24 Feb. 2025.

"CS 3410 Cache Optimization." *Cache Optimization - CS 3410*, Cornell University, www.cs.cornell.edu/courses/cs3410/2024fa/assignments/cacheblock/instructions.html#li. Accessed 24 Feb. 2025.

"Locality of Reference." *Code Guidelines for Correctness, Modernization, Security, Portability, and Optimization*, open-catalog.codee.com/Glossary/Locality-of-reference. Accessed 24 Feb. 2025.

"Loop Tiling." *Code Guidelines for Correctness, Modernization, Security, Portability, and Optimization*, open-catalog.codee.com/Glossary/Loop-tiling/. Accessed 24 Feb. 2025.

"Matrix Multiplication." *Matrix Multiplication - AWS Neuron Documentation*, AWS, awsdocs-neuron.readthedocs-hosted.com/en/latest/general/nki/tutorials/matrix_multiplication.html. Accessed 24 Feb. 2025.

"NKI API Common Fields." *NKI API Common Fields - AWS Neuron Documentation*, AWS, awsdocs-neuron.readthedocs-hosted.com/en/latest/general/nki/api/nki.api.shared.html#nki-mask. Accessed 24 Feb. 2025.

"NKI Programming Model." *NKI Programming Model - AWS Neuron Documentation*, AWS, awsdocs-neuron.readthedocs-hosted.com/en/latest/general/nki/programming_model.html#nki-tensor-indexing. Accessed 24 Feb. 2025.

"NKI Programming Model." *NKI Programming Model - AWS Neuron Documentation*, AWS, awsdocs-neuron.readthedocs-hosted.com/en/latest/general/nki/programming_model.html#tile-size-considerations. Accessed 24 Feb. 2025.

Nvidia. "NVIDIA/Cuda-Samples/matrixMul." *GitHub*, NVIDIA, github.com/NVIDIA/cuda-samples/blob/master/Samples/0_Introduction/matrixMul/matrixMul.cu. Accessed 24 Feb. 2025.

Nvidia. "Nvidia/Cuda-Samples/matrixMulCUBLAS." *GitHub*, NVIDIA, github.com/NVIDIA/cuda-samples/blob/master/Samples/4_CUDA_Libraries/matrixMulCUBLAS/matrixMulCUBLAS.cpp. Accessed 24 Feb. 2025.