EIC Coding Test 3 - FFN CUDA Optimization

Jai Kashyap

1 Goal

This coding test focuses on optimizing a GEGLU FFN with hidden size 4096 and intermediate size 12288 using CUDA kernel. The goal is to outperform a PyTorch baseline across batch sizes {4, 8, 16, 32, 64, 128} by using optimization methods.

2 Baseline Analysis

The baseline GEGLU FFN computes:

$$\text{out} = (GELU(xW_u) \odot (xW_v)) \cdot W_o$$

This design uses two separate GEMMs with the same input, creation of intermediate tensors u, v, and h, and multiple kernel launches.

3 Approach

The optimization approach is based on three main ideas aimed at improving efficiency on the GPU. First, redundant computation and memory traffic are reduced by avoiding repeated reads of the input data and by minimizing the creation of intermediate tensors in global memory. This helps lower memory bandwidth usage, which is often a performance bottleneck. Second, highly optimized GEMM kernels are used for matrix multiplications, since libraries like cuBLASLt are specifically tuned for the GPU architecture and can achieve much higher performance than custom implementations. Instead of reimplementing GEMMs, the computation is structured to take advantage of these optimized kernels. Finally, simple element-wise operations, such as the GEGLU activation, are fused into custom CUDA kernels. This reduces kernel launch overhead and avoids extra memory reads and writes, leading to improved overall performance.

4 Fused Input Projection via cuBLASLt

4.1 Weight Concatenation

To eliminate redundant input reads, the two projection matrices are concatenated:

$$W_{\text{combined}} = [W_u \mid W_v] \in \mathbb{R}^{4096 \times 24576}$$

A single GEMM computes both projections:

$$[u \mid v] = xW_{\text{combined}}$$

This takes away one full GEMM call, one read of the input tensor x and reduces kernel launch overhead.

4.2 Why cuBLASLt Instead of cuBLAS

cuBLASLt is chosen over cuBLAS because it supports heuristic based algorithm selection. The implementation queries cuBLASLt for multiple candidate algorithms and caches the best-performing one. This avoids repeated tuning overhead and ensures optimal kernel selection for the fixed FFN dimensions. Additionally it provides better performance for non-square and fused GEMM shapes.

5 Custom GEGLU CUDA Kernel

5.1 Kernel Design

After the fused GEMM, the output tensor is laid out as:

$$\text{d\_u}[b] = [u \mid v]$$

A custom CUDA kernel computes the GEGLU activation directly from this layout:

$$h = u \odot \text{GELU}(v)$$

5.2 GELU Approximation Choice

The formula for GELU is:

$$GELU(x) = x \cdot \Phi(x)$$

where $\Phi(x)$ is the cumulative distribution function for Gaussian distribution. When not specified PyTorch calculates it exactly. However, PyTorch also has a tanh approximation, where:

$$GELU(x) = 0.5x(1 + Tanh(\sqrt{2/\pi} \cdot (x + 0.044715 \cdot x^3)$$

Initially this was the approximation used, but since even tanh is expensive a sigmoid approximation is faster.

$$\text{GELU}(x) \approx x \cdot \sigma(1.702x)$$

This approximation was chosen because it avoids calculating expensive functions.

6 Output Projection

The final projection:

$$\text{out} = hW_o$$

is computed using a second cuBLASLt GEMM. As with the first GEMM, the optimal algorithm is selected once and reused across iterations.

7 Correctness Verification

Correctness was verified by comparing the CUDA output against a CPU reference implementation for all batch sizes. A tolerance of 10^{-4} was used to account for FP16 precision and activation approximations. All tested configurations pass verification.

8 Results

Both the Python and CUDA implementation were given 5 warmup runs, after which performance was measured as the average execution time over 50 trials.

Batch Size	Baseline (ms)	Optimized (ms)	Speedup
4	3.686	1.818	∼2.028×
8	4.072	1.821	∼2.236×
16	4.368	1.863	∼2.344×
32	4.803	1.92	∼2.502×
64	7.851	2.040	∼3.849×
128	11.911	3.306	∼3.603×

Table 1: Performance comparison between PyTorch baseline and optimized CUDA implementation.

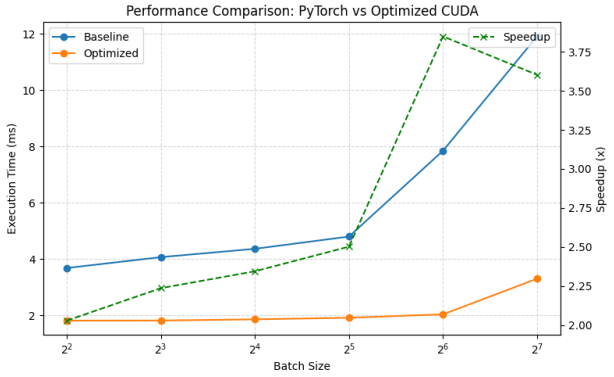

Figure 1: Performance Comparison

9 Discussion

The optimized implementation achieves noticeable speedups over the baseline, particularly for larger batch sizes where kernel launch overhead is less significant. However, reaching a consistent 3× speedup across all batch sizes proved challenging. At smaller batch sizes, performance is limited by kernel launch overhead and reduced parallelism, while at larger batch sizes the implementation approaches the performance of highly optimized cuBLASLt kernels. For the batch size of 64, the speedup factor was 3.849x whereas for the batch size of 128, the speedup factor was less, at 3.603x. Possible reasons for this could be because of the cuBLASLt algorithm selection that is dependent on shape, so B=64 uses an algorithm with great tile utilization, but B=128 uses larger tiles. The speedup ratio dropped, although the runtime still improved.

9.1 Attempts at Further Optimization

Additional optimizations were explored, including experiments with lower-level tensor core interfaces such as WMMA and template-based libraries like CUTLASS.

Specifically, a custom warp-tiled GEMM kernel was implemented using nvcuda::wmma fragments, where each warp computes a 16×16 output tile using Tensor Cores. The kernel manually tiles the input matrices along the K dimension and uses WMMA matrix_a, matrix_b, and accumulator fragments to perform warp-level matrix multiply–accumulate operations. This approach was intended to enable greater data reuse and allow for potential fusion of the GEGLU activation directly into the epilogue by modifying accumulator fragments before storing results to global memory. However, achieving competitive performance proved challenging due to the complexity of correctly managing tile sizes, memory layouts, and warp-level synchronization. Without careful

shared memory staging and extensive tuning, the WMMA-based kernel did not outperform the cuBLASLt implementation, and in some cases incurred additional overhead.

While these approaches have the potential for higher performance, they require detailed knowledge of GPU architecture, warp-level tiling, and memory layout, and the initial attempts did not outperform the cuBLASLt-based solution. As a result, the final implementation focuses on optimizations that are effective and reliable within the scope of this project, while more advanced techniques are identified as future work.

10 Conclusion

This work demonstrates that meaningful FFN performance gains can be achieved through projection fusion, cuBLASLt algorithm selection, and custom CUDA kernel design. The optimized GEGLU FFN significantly reduces memory traffic and kernel overhead while maintaining correctness. The remaining performance gap highlights the difficulty of surpassing vendor-optimized tensor core kernels without extensive architecture-specific tuning.