Parallelizing Transformer Models for NLP Tasks

(BERT Optimization)

Assignment Report

Abishek Chakravarthy
Roll Number: CS22B2054
IIITDM

February 2025

Abstract

This report presents the design, implementation, and performance analysis of a parallel version of a Transformer encoder (akin to BERT) for NLP tasks. The implementation focuses on optimizing computationally intensive components, including self-attention, feedforward layers, layer normalization, dropout, GELU activation, softmax, and matrix multiplication. The serial CPU implementation takes approximately 70 seconds, while the parallel CUDA implementation achieves execution times as low as approximately 1.69 seconds (with optimal thread configurations), resulting in speedups up to about 42x. Detailed performance data, including execution time and speedup, are presented and analyzed.

0.1. Introduction

Transformer models, such as BERT, have become a cornerstone in modern NLP due to their superior performance on a wide range of tasks. However, training and inference with these models are highly computationally intensive because of operations like multi-head self-attention, large-scale matrix multiplications, normalization, and non-linear activations.

The goal of this project is to parallelize the most compute-intensive components of a Transformer encoder using CUDA and OpenMP techniques to optimize performance. By accelerating operations such as:

- **Matrix Multiplication:** Essential for projecting input embeddings in self-attention and feedforward layers.
- **Self-Attention:** Including the computation of query, key, and value projections, attention score calculation (with softmax), and head concatenation.
- **Feedforward Layers:** Two linear transformations with a GELU activation function.
- **Layer Normalization:** Stabilizing activations for efficient training.
- **Dropout and Positional Encoding:** Providing regularization and injecting sequential information, respectively.

we aim to significantly reduce execution times, enabling efficient training and inference for large-scale NLP tasks.

0.2. Methodology

The implementation is divided into two major parts: a serial (CPU) implementation and a parallel (GPU) implementation. The parallel version leverages CUDA kernels for core operations and uses OpenMP for CPU-side data preparation and benchmarking.

0.2.1. Serial Implementation

The serial version implements all key operations on the CPU:

- **Matrix Multiplication:** Computes the product of two matrices using triple nested loops.
- **Layer Normalization:** Calculates the per-row mean and variance to normalize the inputs.
- **Positional Encoding:** Generates sinusoidal encoding values to embed positional information.
- **Self-Attention:** Projects inputs to query, key, and value matrices, computes attention scores via dot products and scaling, applies softmax for normalization, and concatenates the outputs from multiple attention heads.
- **Feedforward Layers:** Applies two sequential linear transformations with a GELU (or ReLU) activation in between, followed by residual connection and normalization.

The serial execution time for the entire Transformer encoder layer is measured at approximately 70 seconds and serves as the baseline for performance comparison.

0.2.2. Parallel Implementation Using CUDA

The parallel implementation accelerates the Transformer encoder components with CUDA kernels:

- **Matrix Multiplication:** A tiled CUDA kernel is implemented to perform matrix multiplication efficiently by loading sub-matrices (tiles) into shared memory, thus reducing global memory accesses.
- **Layer Normalization:** A dedicated kernel calculates per-row statistics (mean and variance) and normalizes each row using these values.

- **Positional Encoding:** A CUDA kernel generates the sinusoidal positional encoding directly on the GPU.
- **Self-Attention:** Multiple kernels handle the projection of inputs to queries, keys, and values; compute attention scores via matrix multiplication; apply softmax normalization; and concatenate head outputs.
- **Feedforward Layers:** The matrix multiplication kernel is re-used along with kernels for GELU activation and dropout, followed by a residual addition and layer normalization.

The parallel execution is benchmarked by varying the thread configuration (e.g., {2, 4, 8, 16, 32, 64, 128, 256, 512, 1024}). For each configuration, we measure:

- **Parallel Time:** Using CUDA events and C++ chrono for overall timing.
- **Speedup:** Calculated as

$$\text{Speedup} = \frac{\text{Serial Time}}{\text{Parallel Time}}$$

Results are logged in a CSV file for further analysis.

0.3. Experimental Results

The serial execution time for the Transformer encoder layer was measured at 70 seconds. The parallel implementation was benchmarked using various thread configurations, and the results were recorded. An excerpt of the CSV results is shown in Table 1.

Table 1: Transformer Parallelization Performance Results

Threads	Parallel Time (s)	Speedup
2	2.824552	23.5495
4	2.200601	31.7999
8	2.029940	34.3100
16	2.197080	32.4300
32	2.156424	32.5471
64	2.001009	35.0571
128	1.793162	39.2142
256	1.688517	42.3110
512	1.694823	41.4731
1024	1.767937	39.8733

Additionally, performance was visualized using the following plots:

- **Execution Time vs. Threads:** (Figure 1)
- **Speedup vs. Threads:** (Figure 2)

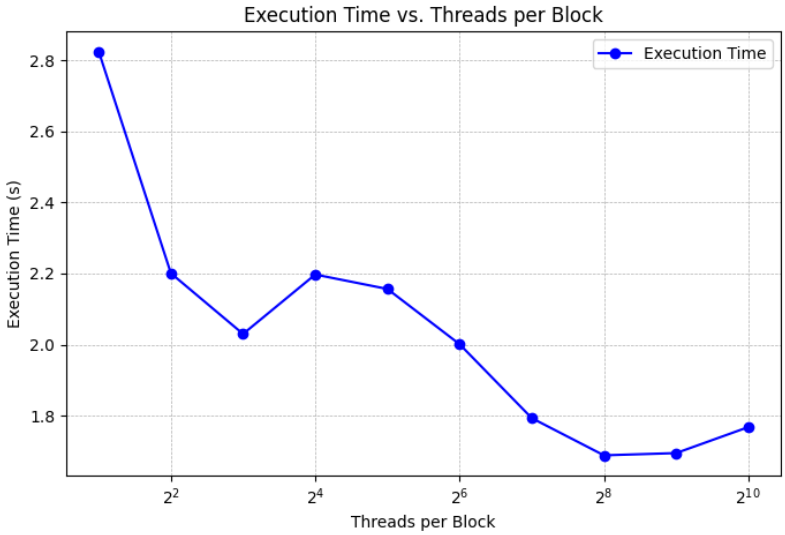

Figure 1: Execution Time vs. Threads for Transformer Encoder

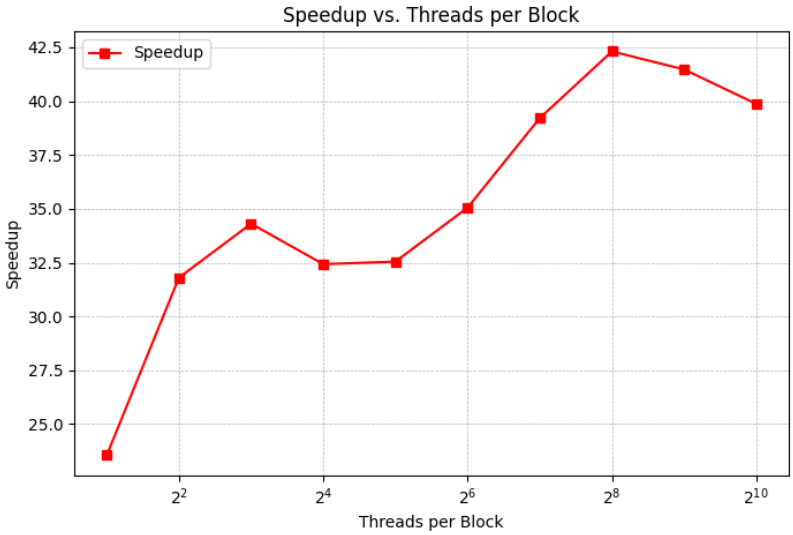

Figure 2: Speedup vs. Threads for Transformer Encoder

0.4. Observation and Analysis

This section provides an in-depth analysis of the performance of the parallel Transformer encoder.

0.4.1. Execution Time Analysis

The parallel execution time decreases significantly as the number of threads increases. As illustrated in Figure 1, the execution time drops from approximately 2.82 seconds with 2 threads to around 1.69 seconds with 256 threads. This dramatic reduction demonstrates the benefit of GPU acceleration for the highly data-parallel operations within the Transformer model. However, beyond 256 threads, the performance improvements are marginal, indicating that the hardware limits—such as memory bandwidth, synchronization overhead, and resource constraints—begin to constrain further gains.

0.4.2. Speedup Analysis

Figure 2 shows that the speedup increases with the number of threads, reaching a peak of about 42x at 256 threads. This significant speedup confirms that a substantial portion of the Transformer's computations (including self-attention, feedforward, and

normalization) are amenable to parallelization. The slight decline in speedup at higher thread counts suggests that additional threads introduce overheads—such as increased synchronization costs and memory contention—that limit scalability.

0.4.3. Detailed Observations

- **Computational Intensity:** The Transformer encoder comprises numerous matrix operations, non-linear activations, and normalization steps that are computationally heavy. The serial implementation's 70-second runtime highlights the critical need for parallelization. By offloading these tasks to the GPU, we can exploit the inherent data parallelism in operations like self-attention and feedforward processing.
- **Scalability and Efficiency:** The observed speedup, peaking at around 42x with 256 threads, indicates excellent scalability for the bulk of the Transformer computations. This efficiency is achieved by mapping matrix operations onto CUDA kernels that leverage shared memory, optimized memory access, and proper thread-block configurations.
- **Bottlenecks and Overheads:** Although the parallel implementation yields substantial speedup, the benefits taper off beyond 256 threads. This plateau is likely due to the overhead associated with thread synchronization, the finite bandwidth of global memory, and the limits of CUDA core availability. These factors suggest that further performance improvements would require advanced optimization techniques such as kernel fusion, loop unrolling, and more efficient memory management.
- **Overall Impact:** The high speedup values validate the effectiveness of parallelizing the Transformer encoder components. The ability to reduce the overall runtime from 70 seconds to as low as 1.69 seconds demonstrates the potential for GPU acceleration to significantly impact real-world NLP tasks, making large-scale training and inference more feasible.

0.5. Conclusion

This report presented a parallelized Transformer encoder model (similar to BERT) implemented using CUDA and OpenMP techniques. The serial execution time was measured at 70 seconds, while the parallel implementation achieved execution times as low as approximately 1.69 seconds with an optimal configuration of 256 threads,

resulting in a maximum speedup of around 42x. Detailed analysis reveals that although increasing the number of threads leads to significant performance improvements initially, further increases yield diminishing returns due to hardware limitations such as memory bandwidth and synchronization overhead. Future work may focus on optimizing memory access patterns, reducing overheads, and exploring advanced scheduling strategies to further enhance parallel performance on larger datasets.