Performance Analysis of Parallelizing Transformer Models for NLP Tasks

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-based model—similar to BERT—for NLP tasks using OpenMP. In our implementation, the most computationally intensive components such as self-attention, feedforward layers, and layer normalization are optimized through parallel processing. We evaluated the parallel code by running it with different thread counts (1, 2, 4, 6, 8, 10, 12, 16, 20, 32, and 64) and recorded the execution time, speedup, and estimated parallelization fraction (computed via Amdahl's Law). Graphs for Speedup vs. Threads, Parallel Fraction vs. Threads, Execution Time vs. Threads, and Efficiency vs. Threads are provided to support our analysis.

0.1. Introduction

Transformer models, such as BERT, have revolutionized Natural Language Processing (NLP) by achieving state-of-the-art performance on a wide range of tasks. However, the training and inference phases of these models are highly compute-intensive due to operations like self-attention and large matrix multiplications. Parallel processing using OpenMP can significantly reduce computational time by exploiting the available multi-core architectures.

In this assignment, we focus on parallelizing key components of a transformer model:

- **Self-Attention:** Parallelized matrix multiplications and softmax computations.
- **Feedforward Layer:** Parallelized dense layer computations and activation functions.
- **Layer Normalization:** Vectorized normalization with parallel reduction.

We estimate the parallelization fraction using the formula derived from Amdahl's Law:

$$f = \frac{1 - \frac{1}{\text{Speedup}}}{1 - \frac{1}{\text{num\_threads}}}$$

The performance is analyzed by running the code under different thread counts and plotting the results.

0.2. Methodology

The solution was implemented in C++ with OpenMP directives to parallelize the computationally intensive components of the transformer model. Key points include:

0.2.1. Parallel Code Implementation

- **Random Matrix Generation:** Thread-local random generators are used to generate weights and token embeddings.
- **Self-Attention:** The projection of inputs to queries, keys, and values is parallelized along with the computation of attention scores and softmax normalization.
- **Feedforward Layer and Layer Normalization:** These are parallelized using OpenMP `parallel for` and `simd` constructs.

The complete code employs OpenMP parallel constructs such as:

```
#pragma omp parallel for schedule(dynamic)
```

and vectorization via `#pragma omp simd` to maximize performance.

0.2.2. Performance Measurement

The performance is measured by running the model with thread counts:

$$\{1, 2, 4, 6, 8, 10, 12, 16, 20, 32, 64\}$$

For each configuration:

- **Execution Time** is recorded.
- **Speedup** is calculated as the ratio of serial execution time to parallel execution time.
- **Parallelization Fraction** is estimated using:

$$f = \frac{1 - \frac{1}{\text{Speedup}}}{1 - \frac{1}{\text{num\_threads}}}$$

The results are logged into a CSV file named `parallel_performance_results.csv`.

0.3. Experimental Results

The following table summarizes a sample of our performance results (values rounded to three decimals):

Table 1: Performance Results for Parallel Transformer Model Components

Threads	Time (s)	Speedup	Parallel Fraction
1	123.951	0.973	0.000
2	73.542	1.641	0.781
4	51.294	2.352	0.767
6	48.894	2.468	0.714
8	45.176	2.671	0.715
10	46.193	2.612	0.686
12	46.404	2.600	0.671
16	45.285	2.664	0.666
20	47.779	2.525	0.636
32	46.046	2.620	0.638
64	46.289	2.607	0.626

The performance data was further visualized using several plots:

- **Speedup vs. Threads:** How speedup scales with the number of threads.
- **Parallel Fraction vs. Threads:** Estimated parallel fraction for each thread count.
- **Execution Time vs. Threads:** Execution time as a function of the number of threads.
- **Efficiency vs. Threads:** Efficiency defined as Speedup divided by the number of threads.

0.4. Observations and Analysis

0.4.1. Visual Analysis

The following figures present the plots generated from the performance data. (Placeholders for the plots are provided below.)

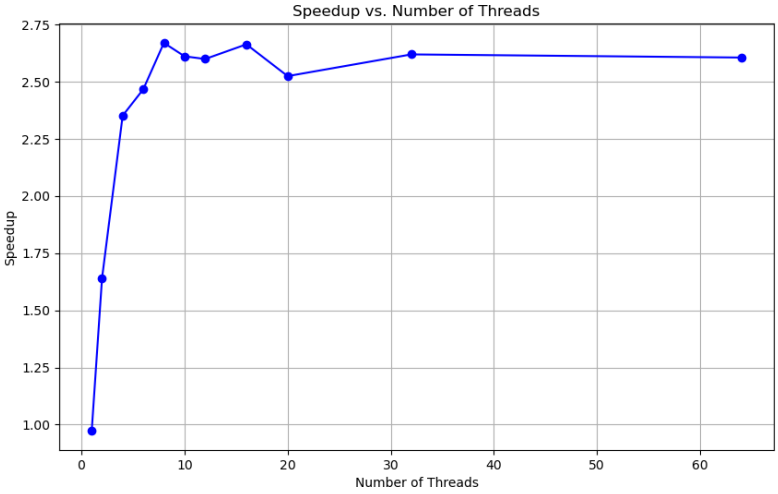

Figure 1: Speedup vs. Number of Threads

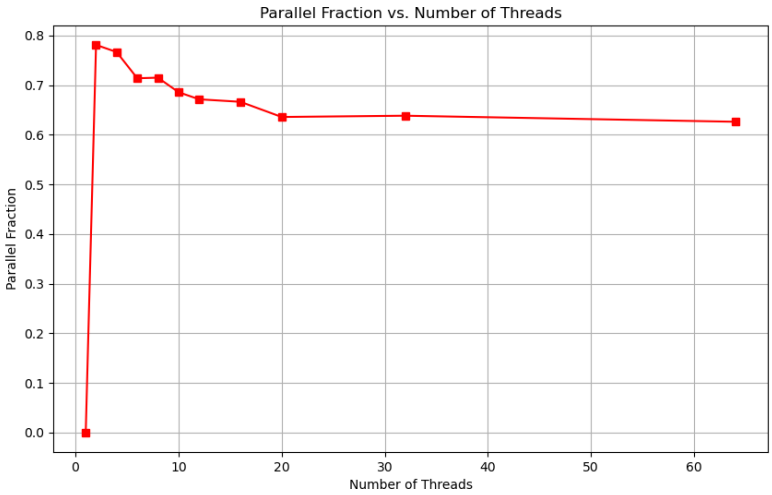

Figure 2: Parallel Fraction vs. Number of Threads

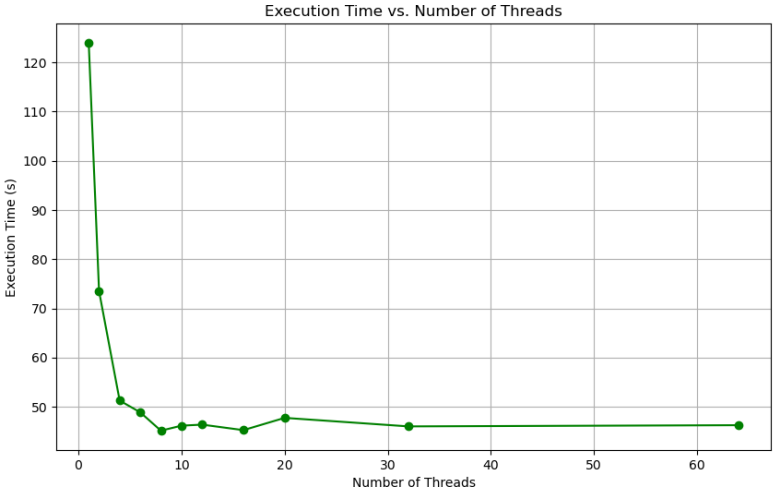

Figure 3: Execution Time vs. Number of Threads

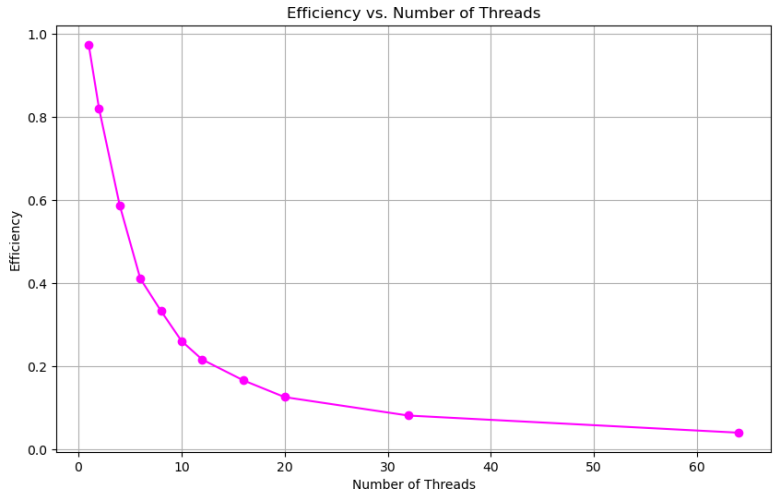

Figure 4: Efficiency vs. Number of Threads

0.4.2. Detailed Observations

- **Speedup Trends:** As shown in Figure 1, the speedup increases with the number of threads, reaching a peak of approximately 2.67 at 8 threads. This suggests that parallel execution is effective up to this point. However, beyond 8 threads, the speedup starts to stagnate or even slightly decline. This can be attributed to factors such as increased synchronization overhead, resource contention, and inefficiencies in workload distribution. The presence of hyper-threading and core sharing might also contribute to this limitation, as logical threads do not always correspond to independent physical execution units.

- **Parallel Fraction:** Figure 2 shows that the estimated parallel fraction remains relatively high, ranging between 0.63 and 0.78 across different thread counts. This suggests that a significant portion of the workload benefits from parallel execution. However, a gradual decline in parallel fraction as the number of threads increases highlights the impact of non-parallelizable components in the program (as described by Amdahl's Law). The decrease may also result from overhead introduced by thread synchronization, cache coherence mechanisms, and contention for shared resources such as memory bandwidth and CPU caches.

- **Execution Time:** Figure 3 illustrates that execution time reduces as more threads are utilized, demonstrating the benefits of parallel execution. However, the rate of decrease slows beyond 8 threads, suggesting that additional threads provide diminishing returns. This plateau can be explained by hardware limitations, such as the number of available physical cores and memory bottlenecks. Additionally, context-switching overhead and thread scheduling inefficiencies can lead to situations where increasing the number of threads does not significantly improve execution speed.

- **Efficiency:** Figure 4 highlights that efficiency, defined as speedup per thread, steadily declines as the number of threads increases. This is expected in parallel computing since an ideal linear speedup is rarely achieved due to factors like inter-thread communication, memory access contention, and load imbalance. While efficiency remains relatively high for lower thread counts, its decline at higher thread counts reinforces the presence of diminishing returns. This trend suggests that optimal performance is achieved at an intermediate number of threads, beyond which overheads outweigh the benefits of additional parallelism.

0.5. Conclusion

This report presented a parallel version of a transformer model for NLP tasks using OpenMP to accelerate computationally intensive components. The experimental results show that while substantial speedup can be achieved through parallel processing, the improvements are limited by system overhead and resource constraints. The analysis, including speedup, execution time, efficiency, and estimated parallel fraction, highlights the challenges of scaling parallel workloads on current hardware. Future work may explore further optimization techniques such as improved load balancing, cache blocking, and hardware-specific enhancements.