15-418 Final Project Report

gxt minjeon2

1 Summary

Our project aims to optimize the computation of the self-attention mechanism in Transformer models. We developed a CUDA implementation of the full attention calculation. In particular, we focused on improving the performance of the Softmax kernel, using a parallel tree-based reduction to reduce the row-wise reduction step from $O(N)$ to $O(\log N)$. To mitigate the hardware bottlenecks we identified during profiling, we implemented asynchronous pipelining to parallelize memory- and compute- bound operations at a single time step and hide kernel launch latency.

2 Background

2.1 Transformer Models and Self-Attention

Transformer models are fundamental to many modern machine learning applications. Specifically, the transformer model architecture forms the backbone of high-performing Natural Language Models models such as BERT, and Generative Pre-trained Transformers (GPTs) such as ChatGPT and Gemini. Transformer models are a type of neural network architecture that utilize "self-attention" mechanism to modify weights of sequential input data dynamically. By applying set of mathematical formulas which are specified below, the model learns the ways that distant data elements in a series influence each other[1]. A common input for transformer models are sentences, since the model can learn how some word in the sentence affects the meaning of a word that comes much later or earlier in the sentence. Unlike more traditional models like recurrent neural networks (RNNs) where each word is processed in sequential order, the self-attention mechanism of transformer models looks at every word in the sentence simultaneously to determine the meaning of the sentence.[2]

2.2 Mathematical Overview of Self-Attention

Once the input is received by the model, it first tokenizes the input into a sequence of N separate tokens. Each token is then converted into a vector of size D. This results in the input matrix X with dimensions $N \times D$. Given the input matrix X, the model learns three weight matrices: Queries (Q), Keys (K), and Values (V). All of these matrices also have dimensions $N \times D$. Then, the Scaled Dot-Product Attention is computed as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{D}}\right)V$$

This formula can be broken into the following concrete steps with the following time complexities:

1. **Score Computation(QK^T):** This produces an $N \times N$ "score" matrix which represents the raw attention scores between every pair of tokens. In a sequential implementation, this requires $O(N^2D)$ operations. Since matrix Q has dimensions $N \times D$ and matrix K^T has dimensions $D \times N$, calculating each entry of the output matrix requires calculating the dot product of two vectors of length D. Since the product matrix has dimensions $N \times N$, this step occurs in $O(N^2D)$ calculations are required for this step.

2. **Softmax Activation:** This step applies the Softmax action function to the score matrix to normalize the scores in each row into probabilities . For a single row vector $\mathbf{x}$ of length N, the Softmax operation is:

$$\sigma(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}}$$

   The sequential calculation of Softmax is very memory-intensive, as it requires three passes over the data. First, the softmax function find the maximum value to subtract from each input value. This ensures that the exponent e^{x_i} is in the range $[0, 1]$, removing the issue of computational overflow. Then, the softmax function computes the exponents and their sum, and finally, it normalizes the results based on the sum. The complexity of this step is $O(N^2)$ for the whole matrix, where each row is typically processed serially. **This creates opportunity for parallelization, which is the main focus of our project.**

3. **Weighted Sum (SV):** Finally, the $N \times N$ probability matrix is multiplied by the V matrix to produce the final $N \times D$ output matrix. Since computing the value for each entry in the output matrix requires computing the dot product of two vectors of length N, the total time complexity of this step is in $O(N^2D)$.

2.3 Parallelizing the Softmax Step

While the operations in the first and third step as explained above are easily parallelizable as there are no dependencies in the data, the Softmax parallelization poses a challenge as it contains several layers of dependencies. First, the maximum of all values has to be calculated then subtracted from each term. Then, after computing the e^{x_i} terms, they have to be summed and then divided from each term. Thus, there are two layers of dependencies. A naive Softmax implementation would simply assign one thread to each row, and perform the operations described in parallel. This is very doable since there are no dependencies per row. However, this results in $O(N)$ time complexity per row.

To optimize this, we utilize **Tree Reduction**. This is a common parallelization used in machine learning operations. Instead of a single thread accumulating values sequentially, multiple threads work in parallel to reduce pairs of values. We explain this technique in more depth in the Approach section. By leveraging Tree Reduction, we speed up the process of calculating the max of the elements and the sum of the exponents such that they are in $O(\log n)$ instead of $O(n)$, providing a significant speedup especially for large inputs.

Regarding SIMD suitability, the parallelization in the self-attention calculation is ideal for SIMD because it involves applying the same operations to every element in the vector or matrix, which is exactly what SIMD does in parallel.

3 Approach

Technologies. For performance evaluation, we utilized the NVIDIA GeForce RTX 2080 GPUs available on the GHC cluster machines. The implementation used C++ for the baseline and CUDA for the parallel program. To guide our optimization decisions, we utilized NVIDIA Nsight Compute (ncu) for kernel profiling$_3$.

Baseline. We implemented a sequential baseline of the attention mechanism in C++. First, the score calculation ($QK^\top$) iterates through every query vector and every key vector in the outer loops, and iterates through the embedding dimension D in the inner loop, then scaled by $1/\sqrt{D}$. This has step $O(N^2D)$ time complexity. Next, in the softmax stage, for each row in the score matrix, we perform three linear scans to find the maximum, exponentiate, and normalize. The scans cost $O(N)$ per each of the N rows, producing a total complexity of $O(N^2)$. The last step calculates the values weighted by the attention score, which iterates over N elements for each of the $N \times D$ outputs, giving this step $O(N^2D)$ time complexity.

Evaluation. To validate the correctness of our implementations, we generated random Q, K, V matrices and compute the attention output using PyTorch. We designed the test suite to consist of realistic problem sizes with embedding dimension $D = 64$ and matrix dimension N ranging from 256 to 8192. For pipelining experiments, we generated matrices for both single-inference ($B = 1$) and batched problems ($B = 4$).

Hardware mapping. We decomposed the attention computation into three kernels:

1. **Attention Score Calculation ($QK^\top$):** We mapped the output matrix S to a grid of thread blocks. We implemented tiled matrix multiplication where threads within a block would load a `TILE_SIZE` $\times$ `TILE_SIZE` tile of Q and K into shared memory in order to improve locality and reduce the number of global memory accesses.

   We tuned `TILE_SIZE` by sweeping `TILE_SIZE` $\in \{4, 8, 16, 32\}$ and observed that setting `TILE_SIZE = 8` produced the lowest execution time. This is because a tile size of 8 launches 64 threads, or 2 warps, per block. This configuration allows the SM to schedule the maximum number of concurrent blocks and also hide the latency of global memory accesses by swapping in warps from other blocks while another block waits for data. The main tradeoff between tile sizes is the granularity of thread block scheduling and the amount of memory reuse. With `TILE_SIZE = 8`, the performance gain from higher occupancy and latency hiding outweighs the reduction in data reuse for this specific problem.

2. **Softmax:** Since softmax normalization is applied based the results per row (e.g. row max and row sum), we based our work assignment strategy on the fact that calculations for different rows are independent. Thus, we mapped each thread block to process one row of the matrix. This ensures that the intermediate results from each reduction are

in shared memory and the global write is deferred until the global max and global sum are derived. To handle matrix dimensions that exceed our block dimension, we implemented a grid-stride loop where each thread performs a local sequential reduction over its segment into private memory before reducing at the block-level. While these additions/comparisons are serial per thread, the memory access pattern such that the sequence of reads by all the threads are contiguous, thus reducing global memory accesses due to memory coalescing. Block-level reduction will be discussed in more detail in the following sections covering our linear and tree reduction strategies.

3. **Weighted Sum (SV):** We implemented a similar tiling strategy as the score computation, changing the grid dimensions to be $N \times D$ to match the dimensions of the output matrix.

Linear Reduction. In our initial parallel implementation of the self-attention mechanism, we used a naive linear reduction strategy to compute the row-wise maximums and sums. Once threads write their partial results to shared memory, the algorithm uses a single thread (thread 0) to linearly scan through the shared array. This mean that all other 1023 threads are idle while the single thread performs the 1024 operations serially, which creates critical path of $O(B)$ latency for block size B. During this stage, this workload imbalance causes hardware utilization to drop significantly, creating a significant latency bottleneck.

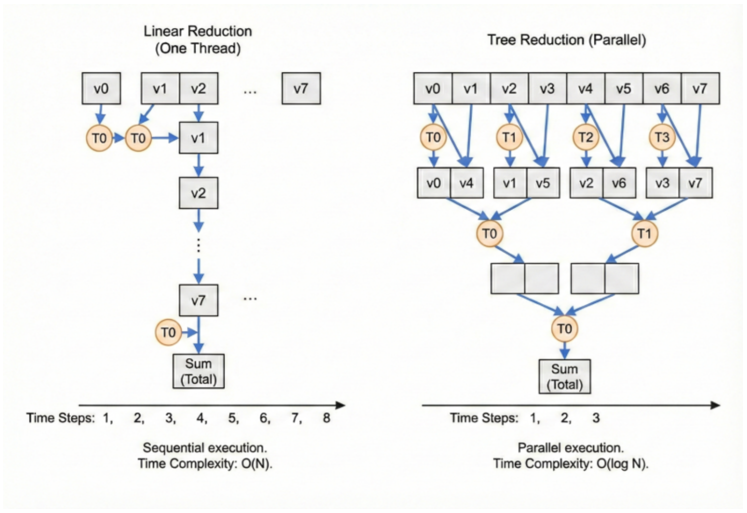

Figure 1: Comparison of linear (sequential) and tree (parallel) reduction approaches.

Tree Reduction. Our second approach attempts to alleviate the synchronization bottleneck by distributing the reduction work across threads in a tree-based manner$_4$. At each step, the reduction work is halved such that active threads effectively "fold" the array in half repeatedly. This reduces the amount of idle time significantly since in the early iterations, every thread in the active warps are doing useful work. In the later steps in the reduction, the amount of time that more threads are sitting idle is drastically less than in linear reduction since they perform fewer reduction steps. Specifically, this reduces the critical path latency from $O(B)$ to $O(\log_2 B)$. Additionally, since our "folding" approach uses contiguous addressing, we ensure than active threads belong to the same warp, which minimizes warp divergence. However, once the number of threads drops below 32, then there is divergence in warp 0.

Another optimization we made was warp unrolling when the number of active threads drops below 32. At this point, explicit barriers are redundant since threads in the same warp are guaranteed to execute instructions simultaneously due to the SIMD principle. Thus, when we are down to the last warp, we remove the `__syncthreads()` barrier and manually unroll the warp. We utilized `volatile` specifiers to force values to be written directly into shared memory so that they are visible to other threads without synchronization.

Pipelining. After we had a functional parallel implementation of the self-attention mechanism, we used NVIDIA Nsight Compute (ncu) to profile the three kernels. As shown in Table 1, the single-batch execution for small workloads (e.g. $N = 256$) showed significant GPU resource under-utilization for all three kernels. Because of the availability of both compute units and memory, we hypothesized that we could fill these idle cycles with useful work from independent instruction streams by implementing asynchronous pipelining. However, our profiling also showed that for larger workloads (e.g. $N = 4096$), GPU resources are saturated by the large amount of thread blocks that are launched for the large grid size. Moreover, rather than vacancy in GPU utilization, the $\sim 50\%$ throughput percentages represent the memory latency limitation. This is the upper bound for the kernel execution time since no additional threads can be scheduled. Thus, we predicted that pipelining would yield diminishing returns for larger workloads.

Kernel	**Duration**	**Compute Throughput**	**Memory Throughput**
Score	24.54 μs	40.03%	42.02%
Softmax	19.42 μs	28.04%	28.04%
Weighted Sum	18.11 μs	38.35%	38.35%

Table 1: Nsight Compute profiling metrics for $N = 256$, $B = 1$.

Kernel	**Duration**	**Compute Throughput**	**Memory Throughput**
Score	5.13 ms	47.57%	49.93%
Softmax	0.63 ms	26.65%	48.58%
Weighted Sum	3.10 ms	53.97%	53.97%

Table 2: Nsight Compute profiling metrics for $N = 4096$, $B = 1$.

To support batched computation ($B > 1$), we implemented pipelined execution model using CUDA Streams. In our single-inference implementation, processing a batch of inputs serializes kernel execution, meaning that GPU resources are unused during kernel launch overheads or memory bus transfers. The opportunity for parallelism comes from the claim that computations and data in each batch are independent.

We created a separate CUDA stream for each batch index. Then, we restructured our host code to loop through the number of batches and asynchronously launch the three kernels into their respective streams. This allows the scheduler to overlap the execution of kernels from different batches. For example, our profiling showed that the Softmax kernel was memory-bound and the Score and Weighted Sum computations were compute-bound, so the pipelined execution would allow the Softmax kernel of batch 0 to execute concurrently with the Score computation of batch 1. Since the workloads had distinct resource bottlenecks, they can run simultaneously and better utilize the GPU's resources. Lastly, we synchronize after all batches are run before validating our output.

4 Results

Evaluation. We measured performance using two main metrics. The first is wall-clock execution time in milliseconds. We measured the total runtime of the attention mechanism as well as the runtime of each step of the self-attention computation using CUDA events for the GPU kernels and the standard C++ chrono library for the CPU baseline. The next metric is the relative speedup of our optimized parallel implementation of the self-attention mechanism and specific kernels compared to the sequential baseline implementation.

Experimental Setup. To simulate realistic transformer workloads, we fixed the embedding dimension at $D = 64$, which is a common head dimension in language models like BERT, and varied sequence length N ranging from 256 to 8192. For our pipelining experiments, we evaluated single inference ($B = 1$) and batched inference ($B = 4$) to evaluate improvements in throughput from pipelining.

To generate the test data, we wrote a Python script to generate test cases and reference output. The Q, K, and V input matrices were initialized using a standard normal distribution with a fixed seed.

4.1 Speedup Graphs

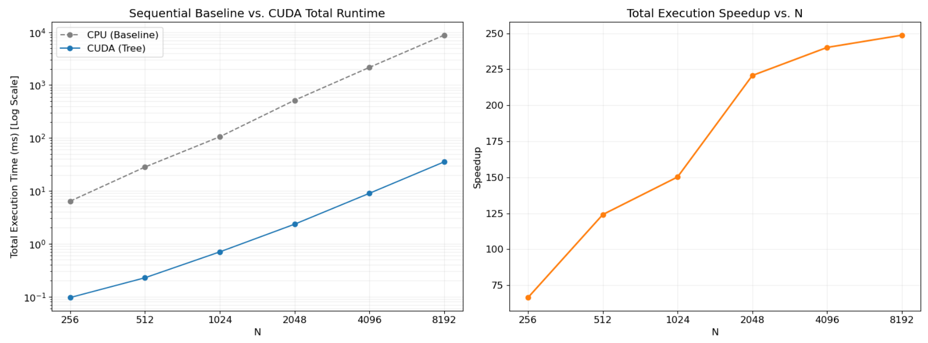

Figure 2: Comparison of the end-to-end runtime of the Sequential CPU baseline vs. optimized CUDA implementation across all N.

Figure 2 plots the total execution time and relative speedup of our CUDA implementation against the sequential baseline across all sequence lengths N with fixed $B = 1$ and $D = 64$. From the total execution time plot, we observe that the gap between the two approaches widens as N increases, indicating that our CUDA implementation produces a substantial performance gain that increases with problem size. For $N = 8192$, our CUDA program reduced the total runtime from 8.7 seconds on the CPU to roughly 35.1 ms on the GPU.

The speedup curve shows the scaling of our implementation, going from $65\times$ at $N = 256$ to nearly $250\times$ at $N = 8192$. We observe an initial increase from $N = 256$ to $N = 1024$ due to improved GPU occupancy and increased parallelism. From $N = 2048$ to $N = 8192$, we observe a second increase and plateau in speedup. Since we fix thread block size to be 1024, for $N = 2048$, each thread now processes 2 elements but with only 1 reduction and constant synchronization overhead, increasing the proportion of useful work done per thread. The speedup plateaus after $N = 4096$, indicating that the program has saturated the device's memory and compute throughput as confirmed by ncu profiling.

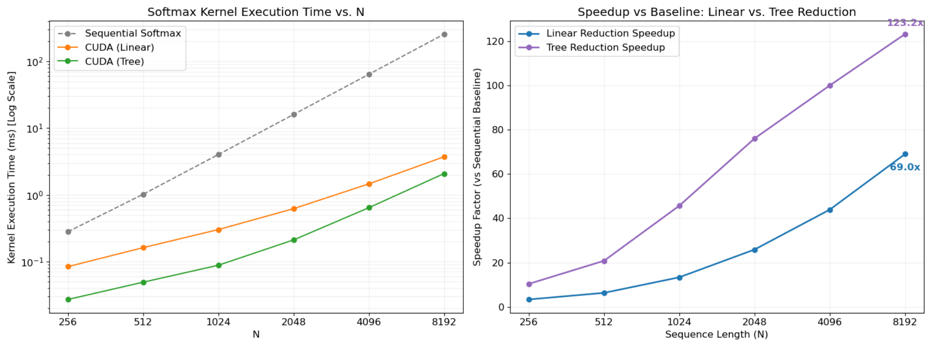

Figure 3: Comparison of the sequential, the linear reduction CUDA, and the tree reduction CUDA implementations across all N.

From Figure 3, we can analyze the evolution of our Softmax kernel and the relative performance of each approach. As shown in the runtime plot, both parallel (CUDA) implementations significantly outperform the sequential baseline, and the tree reduction implementation consistently outperforms linear reduction across all N. The speedup plot on the right shows the speedup achieved by the linear and tree reduction softmax kernels relative to the sequential baseline. The tree reduction implementation consistently outperforms the linear reduction, with both implementations showing strong scaling. For larger N, the performance gap between the two parallel implementations widens, eventually reaching a speedup of 123× and 69× for tree and linear reduction, respectively. The tree reduction speedup also has a steeper slope, indicating that the algorithm is able to increase throughput from the hardware for larger workloads. Overall, both speedup curves show that unlike the total runtime plot, the softmax kernel hasn't plateaued at large values of N.

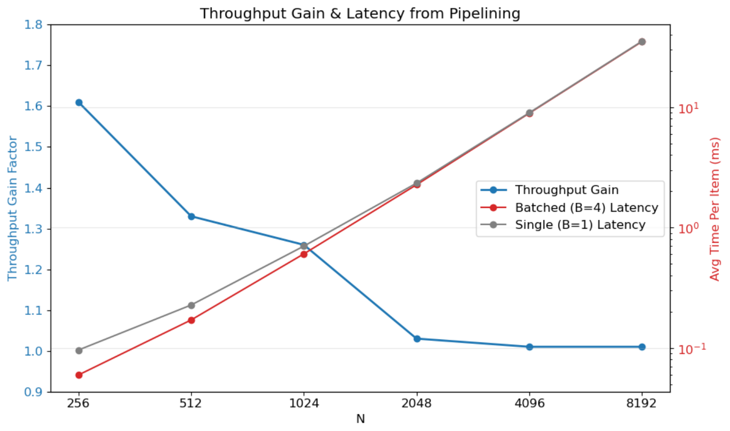

Figure 4: Throughput gain and resource saturation from pipelining by N

As shown in Figure 5, our pipelining strategy was effective for small workloads but hit hardware limits for larger sequence lengths. For small N, we observed a 1.6× throughput gain, reducing the average time per item from 0.1ms to 0.06ms. Profiling with Nsight Compute (ncu) confirmed that this gain was due to successfully hiding kernel launch overhead and increased occupancy of GPU resources via concurrent streams. For N beyond 2048, the throughput gain becomes negligible. For such large workloads, a single batch is large enough to saturate the GPU's compute and memory bandwidth (as shown in Table 2), so concurrent streams are effectively serialized, therefore eliminating any parallelism benefit and incurring some amount of stream management overhead.

4.2 Limitations

Memory Bandwidth. The main limitation for our implementation at large sequence lengths (N) is memory bandwidth. As shown in Figure 4, the score computations kernels (calculating QK^T) dominate the runtime for large N. While the kernel parallelizes the computation of the output matrix S by assigning tiles of S to different blocks, due to the nature of matrix multiplication, every single block still needs access to every single row of the input matrix Q. Similarly, every column of K^T is read by every single block. Thus, since we have a tile size of 8, each element of Q and K is read from global memory $\frac{N}{8}$ times. For large grid sizes, this results in a very large number of global memory accesses. While global memory accesses are parallelized by the hardware, the aggregate bandwidth (the width of the memory bus) is finite, so when thousands of threads simultaneously request data, the memory bus becomes saturated, causing requests to queue up and stalling the compute units. Thus, the large number of read requests reduces the speedup. This issue also exists for the Attention kernel, where reading the input matrix V also requires $\frac{N}{8}$ reads and reading the score matrix S requires $\frac{D}{8}$ reads from global memory. Additionally, global memory traffic is further increased by write operations and repeated reads. The score calculation writes the $N \times N$ matrix S, and the softmax kernel reads this entire matrix three times before overwriting it once as well. Finally, the attention kernel reads the massive S matrix again to produce the $N \times D$ output matrix. While these write operations are parallelized by the GPU hardware, similar to reads to global memory, the volume of data that the kernels are trying to write simultaneously saturates the DRAM write bandwidth, creating a bottleneck. Since global memory is shared by all blocks, this significantly reduces speedup.

Our pipelining results provide further evidence that our speedup is limited by memory bandwidth rather than compute capability. The blue curve in Figure 4 shows the throughput gain of pipelining. As we predicted, the throughput gain decreases as the sequence length N increases, and provides practically no gain at $N = 2048$. Since pipelining relies on overlapping the execution of independent kernels to utilize idle hardware resources, pipelining does not provide throughput gain when a single kernel launch is sufficient to fully saturate the critical resource. Since we observe that the throughput gains diminishes as the sequence length increases, we can infer that our critical resource is the access to the global memory. If our implementation was compute-bound, with higher sequence lengths, the GPU scheduler would successfully overlap execution of independent batches and provide significant throughput gains. Thus, the fact that no overlap occurred for large problem sizes indicates that a single kernel launch is sufficient to fully saturate the device's global memory bus, leaving no "spare" bandwidth for other threads.

Furthermore, while our implementation is mainly memory bound, there are also data dependencies in Softmax that would prevent perfect speedup. As detailed in the Background section, the Softmax calculation requires finding the maximum of all values in a row, then subtracting the maximum from each value. While the subtraction can happen in parallel, the program has to wait for the maximum to be found. Similarly, after all values are exponentiated, the sum of all values have to be calculated so that the row can be normalized. Again, before the normalization occurs, the program has to wait until the sum reduction is complete. Thus, In our Softmax kernel, these dependencies are enforced via the `__syncthreads()` barriers between stages. Because of these unavoidable barriers, the synchronization overhead of waiting for all threads to complete their computations prevents the Softmax kernel from achieving perfect speedup. However, as we mentioned in the previous paragraph, the main bottleneck of our implementation is the limited memory bandwidth.

4.3 Execution Time Breakdown

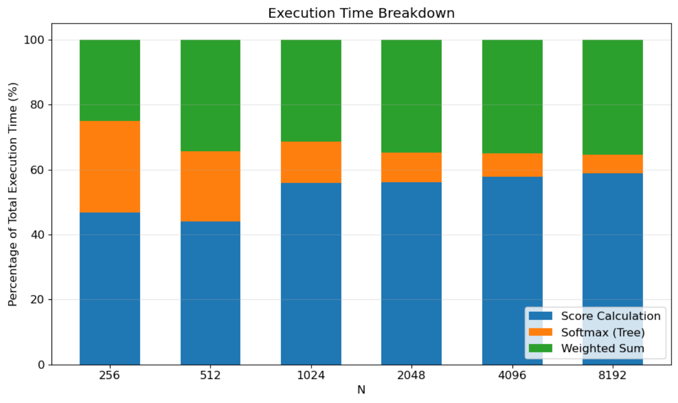

Figure 5: Execution time breakdown by kernel across all N.

The above plot shows the the execution time divided into the three kernels. From our previous analysis, we know the theoretical time complexity of the score and weighted sum calculation to be $O(N^2D)$ and the softmax function to be $O(N^2)$. We observe in Figure 4 that the two matrix multiplication kernels begin to dominate total execution time as N increases, reaching over 90% for $N = 8192$. The softmax kernel, on the other hand, has a diminishing impact on execution time, which is consistent with the increase in speedup of the softmax kernel with larger N. While we mainly focused on the softmax step for this project, this plot clearly shows that the bottleneck is now the matrix multiplication stages.

4.4 Machine Target Analysis

We believe that choosing GPU as the machine target was the correct decision because of its much higher memory bandwidth. CPUs lacks the thousands of cores necessary to parallelize the millions of independent dot products in the QK^T and SV stages. Furthermore, although our GPU kernel saturated its memory bandwidth, high-end GPUs offer significantly higher aggregate bandwidth compared to standard CPU system RAM. Specifically, the NVIDIA RTX 2080 which we used for our performance results has a memory bandwidth of 448 GB/s$_5$, and the NVIDIA A100 offers up ro 1.5 TB/s$_6$. A high-end CPU, such as the Intel 14900K offers only 89.6 GB/s$_7$. Thus, A CPU implementation would have hit the memory limit much earlier and with a much lower performance ceiling. Therefore, despite the memory bandwidth limitations we encountered, the GPU remains the optimal hardware target for this highly data-parallel workload.

5 References

References

[1] ”What Is a Transformer Model? — NVIDIA Blogs.” Accessed December 10, 2025. https://blogs.nvidia.com/blog/what-is-a-transformer-model/.

[2] ”What Are Transformers? - Transformers in Artificial Intelligence Explained - AWS.” Accessed December 10, 2025. https://aws.amazon.com/what-is/transformers-in-artificial-intelligence/.

[3] NVIDIA Developer. ”NVIDIA Nsight Compute.” Accessed December 10, 2025. https://developer.nvidia.com/nsight-compute.

[4] Shyam, Vasudev, et al. ”Tree Attention: Topology-Aware Decoding for Long-Context Attention on GPU Clusters.” ArXiv:2408.04093. Preprint, ArXiv, February 9, 2025. https://doi.org/10.48550/arXiv.2408.04093.

[5] TechPowerUp. ”NVIDIA GeForce RTX 2080 Specs.” December 10, 2025. https://www.techpowerup.com/gpu-specs/geforce-rtx-2080.c3224.

[6] NVIDIA. ”NVIDIA A100 GPUs Power the Modern Data Center.” Accessed December 10, 2025. https://www.nvidia.com/en-us/data-center/a100/.

[7] Intel. ”Intel® Core™ i9 Processor 14900K (36M Cache, up to 6.00 GHz) - Product Specifications.” Accessed December 10, 2025. https://www.intel.com/content/www/us/en/products/sku/236773/intel-core-i9-processor-14900k-36m-cache-up-to-6-00-ghz/specifications.html.

6 Distribution of Work

Given our distribution of the work, we believe that each of us should receive 50% of the credit. We have listed the exact work that each of us have done below. Note that there is some overlap showing the parts we worked on together.

Sunny:

1. Implement serial baseline and CUDA kernel implementations.
2. Design and run experiments on the PSC machines to observe any performance differences.
3. Run final experiments and profile the optimized kernels.
4. Write the “Approach” and “Background” sections of the report.

Grace:

1. Implement serial baseline and CUDA kernel implementations.

2. Utilize tools (e.g., NVIDIA Nsight Compute) to profile the kernels to confirm main bottlenecks of our implementation.
3. Implement pipelining to support concurrent batch and multihead processing.
4. Write the “Summary”, “Approach”, and “Results” sections.