Blocked Dynamic Sparse Attention

Mehul Goel (mehulg), Saransh Agrawal (saransh2)
Website: https://mehulgoel873.github.io/15418-final-project/

Summary

We are going to implement optimized parallel versions of the attention mechanism for the transformer architecture (Vaswani et al., 2017) on the NVIDIA GPUs in the lab. We want to compare and analyze a variety of implementations on inference speed, accuracy, and resource usage.

Background

The core computation in transformer-based models is the *scaled dot-product attention* mechanism, defined as:

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

where $Q, K, V \in \mathbb{R}^{n \times d_k}$ are the query, key, and value matrices derived from input embeddings, n is the sequence length, and d_k is the head dimension. The diagram below illustrates the dataflow of this computation:

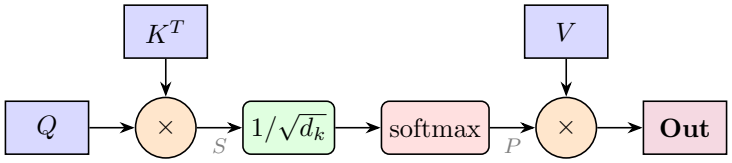

Figure 1: Dataflow of scaled dot-product attention.

This computation involves two matrix multiplications (QK^T and the product with V) separated by an element-wise scaling and a row-wise softmax. While the matrix multiplications are straightforward to parallelize on NVIDIA GPUs – each output element can be computed independently – the softmax operation makes parallelizing this computation challenging.

The softmax function, $\text{softmax}(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$, requires a reduction over the entire row to compute the normalization denominator. This creates a data dependency: no output element can be finalized until every element in its row has been read. In a naive implementation, this forces the full $n \times n$ attention matrix $S = QK^T/\sqrt{d_k}$ to be moved into GPU global memory before the softmax can proceed, resulting in $O(n^2)$ memory usage that becomes prohibitive for long sequences.

Also notice that computing attention requires $O(n^2 d_k)$ FLOPs but $O(n^2)$ memory traffic, leading to a low arithmetic intensity, implying that attention is often bottlenecked by memory bandwidth rather than compute.

Further, the attention matrix S is often approximately sparse: many entries contribute negligibly after softmax. This motivates the use of sparse attention variants that prune or mask out small

entries based on heuristics or learned patterns, which can be static (e.g., local attention) or dynamic (e.g., top-k). However, exploiting this sparsity on GPU hardware is non-trivial, as discussed below.

Challenge

Workload Characteristics

The attention computation $\text{softmax}(QK^T/\sqrt{d_k})\,V$ presents a workload with conflicting parallel characteristics:

- **Dependencies.** The two matrix multiplications are separated by a row-wise softmax, creating a sequential dependency chain: the second matmul ($P \times V$) cannot begin until softmax completes, and softmax cannot begin until the entire row of $S = QK^T$ is available. Within softmax itself, computing the normalization denominator $\sum_j e^{z_j}$ is a global reduction across each row, meaning every element depends on every other element in its row.
- **Memory access patterns.** The naive algorithm materializes the full $n \times n$ attention matrix S in HBM (high-bandwidth memory), reads it back to compute softmax, writes P back to HBM, then reads P again for the second matmul. This results in $O(n^2)$ memory traffic dominated by repeated HBM reads/writes, making the computation *memory-bound* rather than compute-bound. There is strong row-wise locality in the softmax (each row is accessed sequentially), but the two matmuls access their operands along different dimensions, creating conflicting access patterns.
- **Communication-to-computation ratio.** For a sequence of length n and head dimension d_k, the attention block performs $O(n^2 d_k)$ FLOPs but moves $O(n^2)$ data through the memory hierarchy. Since d_k is typically small (64 or 128), the arithmetic intensity is low, and performance is bottlenecked by memory bandwidth rather than ALU throughput.
- **Divergent execution.** In sparse attention variants, different regions of the attention matrix may be masked or pruned, leading to irregular computation. Threads within the same warp may follow different code paths depending on whether their assigned block is sparse or dense, reducing SIMT efficiency.

System Constraints

Mapping this workload onto GPU hardware introduces several challenges:

- **SRAM capacity.** GPU shared memory (SRAM) per streaming multiprocessor is limited. Tiled algorithms like FlashAttention must partition Q, K, and V into blocks that fit in SRAM, but tile sizes affect both occupancy and the number of passes required over the data. Choosing optimal tile dimensions requires balancing SRAM usage, register pressure, and warp occupancy.
- **Sparsity and load imbalance.** In practice, attention matrices are often approximately sparse: many entries contribute negligibly after softmax. Exploiting this sparsity can reduce computation from $O(n^2)$ to $O(n \cdot k)$ for some $k \ll n$, but introduces challenges: sparse block layouts destroy the regular memory access patterns that dense tiled matmuls rely on, and non-uniform sparsity patterns cause load imbalance across thread blocks.

- **Static vs Dynamic sparsity and memory irregularity.** Static sparsity patterns (e.g., local attention) can be efficiently implemented with fixed block layouts, but may sacrifice accuracy by ignoring long-range dependencies. Dynamic sparsity patterns (e.g., top-k) can adapt to the input data and potentially improve accuracy, but require additional computation to determine which blocks to keep or prune, as well as more complex indexing, coalescing, and load balancing strategies to handle irregular memory accesses.

Through this project, we hope to learn how to navigate these tradeoffs: understanding when tiling, fusion, and sparsity provide gains, and how to compose them into an implementation that is both fast and numerically convergent with the standard attention mechanism.

Resources

We will begin with a naive CUDA implementation of the attention block, drawing on reference code from prior coursework and public repositories. From there, we will follow the FlashAttention paper (Dao et al., 2022) and related work on tiled, fused attention kernels to guide our memory-efficient implementations, followed by exploration of static sparse attention patterns (e.g., local, strided) and dynamic sparse attention patterns (e.g., top-k) using heuristics guided by literature on sparse attention.

We have access to the GHC machines with the NVIDIA GeForce RTX 2080 for development and benchmarking, as well as an NVIDIA Blackwell RTX 6000 (with much higher memory) through a shared club compute cluster for testing in a different environment at larger scales.

Deliverables

- **Baseline implementation.** A naive CUDA implementation of the attention block that materializes the full $n \times n$ attention matrix in global memory, serving as a correctness and performance baseline.
- **Tiled and fused implementations.** Optimized CUDA kernels that implement tiled and fused attention computations (a la, FlashAttention), reducing memory usage and improving performance by keeping intermediate results in shared memory.
- **Sparse attention implementations.** Implementations of static sparse attention patterns (e.g., local, strided) and dynamic sparse attention patterns (e.g., top-k) that exploit sparsity to reduce computation while maintaining accuracy. Stretch goals: a flexible implementation that can switch between different sparsity patterns at runtime; more efficient heuristics for dynamic sparsity that balance accuracy and performance on test workloads.
- **Performance analysis.** A comprehensive evaluation comparing the different implementations on inference speed, memory usage, and numerical accuracy across varying sequence lengths and sparsity levels, with insights into the tradeoffs involved. Stretch goal: integration of our custom kernels into a PyTorch workflow using C++ extensions for testing against existing implementations in the PyTorch ecosystem.

Platform Choice

We will implement our attention kernels aimed at NVIDIA GPUs. The choice of GPUs is motivated by their power and versatility for parallel workloads, especially those involving matrix operations;

the choice of NVIDIA is driven by the availability of CUDA, which provides a mature programming model and ecosystem for GPU development. We will implement our kernels in CUDA C++, as it provides low-level control over GPU resources and allows us to optimize for memory access patterns, thread scheduling, and synchronization. CUDA's rich ecosystem of libraries (e.g., cuBLAS for dense matmuls) and profiling tools (e.g., NVIDIA Nsight) will facilitate both development and performance analysis. If time permits and we believe it might be constructive, we will leverage PyTorch's C++ extensions to integrate our custom kernels into a Python workflow for testing and benchmarking against existing implementations.

Schedule

Over 5 weeks, we plan to follow this schedule:

- **Week 1:** Implement a naive CUDA version of the attention block that materializes the full attention matrix in global memory. Verify correctness against a CPU implementation and establish a performance baseline.
- **Week 2:** Implement a tiled and fused version of the attention block (a la, FlashAttention), keeping intermediate results in shared memory to reduce global memory traffic. Benchmark against the naive implementation.
- **Week 3:** Implement static sparse attention patterns (e.g., local, strided) that prune blocks of the attention matrix based on fixed heuristics. Evaluate accuracy and performance tradeoffs.
- **Week 4:** Implement dynamic sparse attention patterns (e.g., top-k) that adaptively prune blocks based on input data. This will involve additional computation to determine which blocks to keep, as well as handling irregular memory access patterns.
- **Week 5:** Flex room for prior weeks' goals. Perform a comprehensive performance analysis comparing all implementations across varying sequence lengths and sparsity levels. Prepare final deliverables and documentation.