Accelerating Sparse Attention with Triton Kernels and Tensor Parallelism

Chuqi Zhang, Likeer Xu

Project page: https://aoiryo.github.io/pp-finalproject/

Summary

This project attacks the two primary bottlenecks in large Transformer models: the $O(N^2)$ attention compute cost and the massive parameter count. We will implement a "hybrid" Transformer block that (1) uses a custom-written Sparse Attention Triton kernel to reduce the attention complexity, and (2) uses Megatron-style Tensor Parallelism (TP) to partition the large (dense) MLP layers across multiple GPU processes. Our focus is on the low-level implementation of the sparse attention operator using the Pythonic Triton language and integrating it with PyTorch's distributed (NCCL) framework.

Background

Standard Transformer models (e.g., in CV and NLP) are prohibitively expensive due to two main scaling issues. First, the self-attention mechanism performs a dense $N \times N$ matrix computation (where N is sequence length), which is an $O(N^2)$ bottleneck. Sparse Attention (e.g., Longformer, BigBird) proposes a solution by computing attention only over a sparse, pre-defined subset of token pairs, reducing the complexity to $O(N)$ or $O(N \log N)$.

Second, to achieve state-of-the-art results, the Feed-Forward (MLP) layers of these models have grown to billions of parameters, exceeding single-GPU memory. Megatron-LM's Tensor Parallelism (TP) solves this by splitting the large, dense weight matrices of the MLP layers across multiple GPUs.

Our project will implement a hybrid model that uses both techniques.

The Challenge

This project presents two distinct and significant parallel systems challenges:

1. **On-GPU Kernel Challenge (Sparse Attention):** Naively implementing sparse attention in PyTorch (e.g., using gather operations) is extremely inefficient. The core challenge is to write a high-performance kernel that can handle the irregular memory access and workload imbalance inherent in sparse operations. For example, if the sparsity pattern is dynamic or uneven, a naive thread-per-query mapping will lead to massive GPU thread divergence and idle time (load imbalance), which we hope to learn how to solve.

2. **Cross-GPU System Challenge (Megatron TP for MLPs):** The large MLP layers still need to be parallelized. The challenge here is to implement the Megatron-style ColumnParallelLinear and RowParallelLinear modules that manage the distributed computation and synchronized ncclAllReduce communication correctly and efficiently.

Resources

- **Codebase:** We will start from scratch using Python and PyTorch.
- **Languages/APIs:**
    1. **Triton Language:** To write the high-performance sparse attention kernel. Triton allows us to write Pythonic code that JIT-compiles into high-performance CUDA, avoiding all C++ build system and binding complexity.
    2. **Python/PyTorch:** For the overall model structure, automatic differentiation, and to implement the Tensor Parallel MLP layers using torch.distributed (which wraps NCCL).
- **Hardware:** We will use torchrun to simulate a 4-process environment on a single-GPU machine for development. We plan to use the GHC cluster or PSC machines (requesting a node with 4+ GPUs) for final performance analysis.
- **References:**

    1. Triton Language Documentation (OpenAI)
    2. Shoeybi, et al. (2019). Megatron-LM: Training Multi-Billion Parameter Language Models.
    3. Child, R., et al. (2019). Generating Long Sequences with Sparse Transformers.

Goals and Deliverables

Plan to Achieve (Must-haves for a successful project)

1. **Implement Sparse Attention Triton Kernels:**
    - Develop custom Triton kernels for multiple sparse attention patterns:
        - Fixed sparse patterns (e.g., block-diagonal, strided)
        - Local window attention (similar to Longformer)
        - Random sparse attention patterns
    - Each kernel will take Q, K, V matrices and a sparse mask/index map as input and produce correct attention output
    - Validate correctness against dense PyTorch attention implementation
2. **Implement Tensor Parallel MLP Layers:**
    - Build `ColumnParallelLinear` module that partitions weight matrices column-wise across GPUs
    - Build `RowParallelLinear` module that partitions weight matrices row-wise with AllReduce synchronization
    - Support both forward and backward passes with proper gradient handling
3. **End-to-End Integration:**
    - **Deliverable:** A complete "Hybrid Transformer Block" combining sparse attention and tensor-parallel MLPs
    - Run successfully in a multi-process environment (4 processes via torchrun)
    - Demonstrate correctness by comparing outputs with a reference PyTorch implementation
    - Document the API and usage examples

Hope to Achieve (Stretch Goals)

1. **Advanced Kernel Optimization:**
    - Implement load-balancing techniques for handling uneven sparsity patterns
    - Optimize memory access patterns using shared memory tiling
    - Benchmark different block sizes and thread configurations
    - **Deliverable:** Performance analysis showing kernel optimization impact (2-5x speedup target)
2. **Comprehensive Performance Analysis:**
    - **Sparse Attention Benchmarks:**

        - Compare our Triton kernels vs. naive PyTorch gather/scatter implementations
        - Test on varying sequence lengths (512, 1024, 2048, 4096 tokens)
        - Measure speedup across different sparsity levels (50%, 75%, 90% sparse)
    - **Tensor Parallelism Scaling:**
        - Strong scaling analysis: fixed model size, varying GPU count (1, 2, 4, 8 GPUs)
        - Measure communication overhead (AllReduce latency)
        - Compare with baseline implementations (Data Parallel, Pipeline Parallel)
    - **Deliverable:** Performance graphs and analysis report demonstrating:
        - Sparse attention speedup vs. dense attention
        - Tensor Parallel efficiency and communication costs
        - Comparison with alternative parallelism strategies
3. **Comparison with Other Parallelism Strategies (if time permits):**
    - Implement baseline Data Parallel version (DDP) for comparison
    - Compare memory efficiency: Tensor Parallel vs. Data Parallel vs. Pipeline Parallel
    - Analyze trade-offs between computation efficiency and communication overhead

Platform Choice

This hybrid problem requires a hybrid platform. The chosen platform is ideal because:

1. **Triton:** We cannot solve the sparse attention bottleneck with PyTorch/NCCL alone, as they are for dense operations. Triton is the perfect tool as it gives us the fine-grained, kernel-level control (like CUDA C++) needed to manage irregular parallelism, but with a high-level Pythonic syntax and seamless, zero-overhead integration with PyTorch Tensors.

2. **PyTorch/NCCL:** We use this for its robust and highly-optimized library for distributed parallelism, which is perfect for the large, dense MLP layers.

Schedule

- **Week 1 (Nov 13 - Nov 17):** Finalize proposal.
- **Week 2 (Nov 18 - Nov 24):**

    - A: Implement the naive (but functional) Sparse Attention Triton kernel. Focus on correctness.
    - B: Implement the ColumnParallelLinear module (forward/backward) for the MLP.
- **Week 3 (Nov 25 - Dec 1):**
    - A: Implement RowParallelLinear (MLP part 2) and integrate with Partner B's work.
    - B: Begin optimizing the Triton kernel, focusing on memory access patterns.
- **Milestone (Dec 1):** Deliverable: A functional, integrated Hybrid Transformer Block that passes correctness tests in a multi-process environment.
- **Week 4 (Dec 2 - Dec 8):**
    - A: Focus on optimizing the Triton kernel for load imbalance (the stretch goal).
    - B: Secure cluster access. Run all benchmarks and generate performance graphs for both components.
- **Final Report (Dec 8):** Write up the final report.