15-618 Project Milestone Report

15-618 Parallel Programming Final Project

Milestone Report

Project Title: Accelerating Sparse Attention with Triton Kernels and Tensor Parallelism

Team Members: Chuqi Zhang, Likeer Xu

Date: Dec 1, 2024

Summary of Work Completed

We have successfully completed the core components of our hybrid Transformer implementation and validated their correctness. Our progress aligns well with the proposed timeline, with both major components (Sparse Attention and Tensor Parallel MLPs) now implemented and tested.

1. Sparse Attention Triton Kernel Implementation

We implemented a mask-specific sparse attention kernel using Triton that supports arbitrary sparsity patterns defined by boolean masks. The kernel takes as input:

- Query, Key, Value matrices (Q, K, V)
- A boolean attention mask specifying which token pairs to compute
- Standard attention parameters (scaling factor, optional dropout)

Key implementation details:

- Block-based computation for efficient GPU memory usage
- Implemented using a sparse grid data structure instead of the original rectangular block
- Proper handling of masked positions (setting to $-\infty$ before softmax)
- Support for batch processing and multiple attention heads
- Fused softmax computation within the kernel

Correctness validation:

- Implemented reference PyTorch implementation for comparison
- Tested on multiple sparsity patterns with different masks
- Numerical error is controlled within threshold

2. Tensor Parallel MLP Implementation

We implemented Megatron-style tensor parallel linear layers:

ColumnParallelLinear:

- Partitions weight matrix column-wise across GPUs
- Each GPU computes a slice of the output independently
- No communication required in forward pass

RowParallelLinear:

- Partitions weight matrix row-wise across GPUs
- Requires AllReduce communication to sum partial results in forward pass

Correctness validation:

- Tested on multiple L40 GPUs
- Compared outputs with single-GPU dense linear layer
- All numerical tests passed with error threshold

Progress Against Original Goals

Plan to Achieve

1. Implement Sparse Attention Triton Kernels

- Status: **Completed**
- We have a working mask-based sparse attention kernel
- Correctness validated against PyTorch reference
- Initial performance measurements completed

2. Implement Tensor Parallel MLP Layers

- Status: **Completed**
- Both ColumnParallelLinear and RowParallelLinear implemented
- Forward passes working correctly
- Tested on multi-GPU setup

3. End-to-End Integration

- Status: **In Progress**
- Individual components are ready
- Integration planned for next week

Hope to Achieve (Optional Goals)

1. Advanced Kernel Optimization

- Status: **Partially Started**
- Basic block-level optimization implemented
- Load balancing for uneven sparsity patterns deferred to next week

2. Comprehensive Performance Analysis

- Status: **Preliminary Results Available**
- Initial benchmarks on sparse attention completed
- Full scaling analysis scheduled for next week

3. Comparison with Other Parallelism Strategies

- Status: **Not Started**
- This remains a stretch goal if time permits

Revised Schedule and Next Steps

Updated Schedule

Week 4 (Nov 27 - Dec 1): Component Implementation ✅ Completed

- **Chuqi:** Implemented and validated Sparse Attention Triton kernel
- **Likeer:** Implemented Tensor Parallel MLP layers (Column and Row)
- **Deliverable:** ✅ Both core components working with correctness tests passing

Final Week (Dec 2 - Dec 8): Integration, Testing & Report ⏳ Current Week (FINAL DEADLINE: Dec 8)

- **Dec 2-3 (Mon-Tue):**
  - **Chuqi:** Integrate sparse attention with tensor parallel MLPs into Hybrid Transformer Block
  - **Likeer:** Set up end-to-end testing framework and multi-GPU benchmarking infrastructure
  - **Deliverable:** Integrated Hybrid Transformer Block with basic functionality
- **Dec 4-5 (Wed-Thu):**
  - **Chuqi:** Run comprehensive performance benchmarks (sparse vs dense, scaling tests)
  - **Likeer:** Optimize communication patterns, profile bottlenecks
  - **Joint:** Analyze results, generate performance graphs
  - **Deliverable:** Complete performance data and analysis
- **Dec 6-7 (Fri-Sat):**
  - **Both:** Write final report following course guidelines
  - Prepare poster materials and visualizations
  - **Deliverable:** Draft final report complete
- **Dec 8 (Sun):**
  - **Both:** Finalize and submit final report (Due 11:59pm)

Immediate Next Steps

1. **Integrate Components** (Priority 1)
   - Combine sparse attention with tensor parallel MLPs
   - Implement Hybrid Transformer Block class
   - Ensure proper device placement and communication
2. **End-to-End Testing** (Priority 2)

- Test on small model configuration (e.g., hidden_dim=1024, num_heads=8)
- Verify numerical correctness against reference implementation
- Validate gradients with PyTorch autograd checker

3. **Initial Benchmarking** (Priority 3)

- Measure end-to-end latency for different configurations
- Profile to identify bottlenecks (computation vs. communication)
- Prepare baseline for optimization efforts

Preliminary Results

Sparse vs. Dense Attention Performance

Our sparse attention implementation uses a **mask-controlled** design where only tiles (blocks) corresponding to `True` positions in the attention mask are processed as computational tasks. This allows flexible sparsity patterns while maintaining efficient GPU utilization.

Benchmark Sparsity Pattern: Band Diagonal (Local Attention Window)

We evaluated performance using a band diagonal sparse pattern, similar to Longformer's local attention mechanism:

```
Pattern characteristics:
- Each token attends to itself + neighboring tokens within a window
- Window size: configurable (tested with window_size = 32)
- Computational complexity: O(N × window_size) = O(N) instead of O(N²)
- Sparsity: ~75% (only 25% of attention pairs computed)
```

Why this pattern:

1. **Practical relevance:** Widely used in state-of-the-art models (Longformer, BigBird, etc.)
2. **Sequential modeling:** Most NLP/sequence tasks benefit from local context (neighboring tokens are more relevant)
3. **Scalability:** Linear complexity enables processing much longer sequences
4. **Validation:** Regular pattern makes correctness testing straightforward

Sparse Triton Kernel Design

We implement sliding-window attention as a **block-sparse Triton kernel** instead of materializing the full dense (N N) score matrix. Given a dense boolean mask ([B, H, N, N]) that encodes the local window pattern, we first convert it into block metadata with `_build_block_metadata`. The sequence is partitioned into tiles of size `BLOCK_M × BLOCK_N` (64×64 in our experiments). For each ((b, h, q_)) we store (1) `BlockCounts[b,h,q_block]`: how many key blocks are actually used, and (2) `BlockIndices[b,h,q_block,*]`: the IDs of those key blocks. This converts the dense mask into a compact block-sparse representation and lets the kernel completely skip tiles that are fully masked out.

The Triton launch grid is two-dimensional: `grid = (num_q_blocks, batch_size * num_heads)`. Each program instance handles one ((b, h, q_)). Inside the kernel we

- load the corresponding Q tile `[BLOCK_M, BLOCK_DMODEL]` once into registers;
- loop over `idx in range(MAX_BLOCKS_PER_Q)` and, for each valid `idx < BlockCounts`, read the `key_block_id` from `BlockIndices`;
- compute the key positions for this block and load the corresponding sub-mask `mask_block` **before** loading K/V;
- if the block index is invalid or `mask_block` is entirely false, we skip loading K/V and move to the next block (block-level pruning);
- otherwise, load the K and V tiles, compute scaled scores `qk = tl.dot(q, k.T) * scale`, and apply three guards by setting scores to `-inf`:
  - invalid blocks (`process_block == False`),
  - masked positions (`mask_block == False`),
  - out-of-range positions (`offs_n_cur >= seq_len`);
- run an **online softmax across blocks**, maintaining running max `m_i`, normalization `l_i`, and accumulator `acc`, and update `acc = acc * alpha[:, None] + tl.dot(p.to(v.dtype), v)` for each active block.

After the loop we normalize with `acc = acc / l_i[:, None]` (where `l_i > 0`) and write the result back to `Out`. This design achieves good performance because:

- the sliding-window pattern is treated as block sparsity, so we only touch tiles that contain at least one valid entry, reducing both computation and memory traffic from (O(N^2)) to (O(N));
- **block-level pruning** avoids loading K/V for fully masked tiles, which is especially beneficial at high sparsity (e.g., ~75% unused pairs);
- the kernel **fuses score computation, masking, softmax, and value aggregation** into a single pass over active tiles, avoiding intermediate tensors and extra kernel launches.

In microbenchmarks (e.g., `seq_len = 128`, `window_size = 32`, 8 heads), this kernel delivers roughly **2× speedup** over a dense Triton attention baseline while matching its numerical output within (10^{-5}) relative error.

Performance Results:

Implementation	Time (ms)	Speedup	Pattern
Dense Blocks	0.23	1.0x (baseline)	Full attention
Pruned Sparse	0.13	**~2x**	Band diagonal (window=32)

Key Observations:

- **~2x speedup** achieved with 75% sparsity
- **Good numerical accuracy** (no difference from masked dense implementation)
- Only tiles within the attention window are scheduled and computed
- Unused tiles are pruned completely, saving both computation and memory

Note: Measurements on NVIDIA GeForce RTX 5090 (Laptop GPU), batch_size=4, num_heads=8, seq_len=128

Next Steps:

- Scale to longer sequences (1024, 2048, 4096) where O(N) vs O(N^2) advantages become dramatic
- Test additional patterns (random sparse, block-structured, strided)

Concerns and Remaining Challenges

Technical Challenges

1. **Integration Complexity:**
    - Ensuring proper synchronization between sparse attention (local to GPU) and tensor parallel MLPs (distributed across GPUs)
    - Need to carefully manage device placement and data movement
2. **Performance Optimization:**

- Current sparse attention speedup is below theoretical maximum
- Need to investigate memory access patterns and occupancy issues
- Potential for further optimization through better load balancing

3. **Scalability Testing:**

    - Limited access to multi-GPU nodes (L40 cluster availability)
    - Need to secure extended access for comprehensive scaling experiments or try AWS as a back-up plan

Confidence Level

We are **confident** that we will meet all "Plan to Achieve" goals by the poster session. The core implementations are complete and working correctly. The remaining work is primarily:

- Integration (well-defined task, ~2 days)
- Testing and benchmarking (straightforward, ~2 days)
- Performance analysis and final report (~2 days)

The "Hope to Achieve" goals (advanced optimization, comprehensive comparison) remain stretch goals depending on available time after completing the primary deliverables.