Week 4: Real GPU Kernels for Machine Learning

Abhishek Upadhya, 24B1309

1 Task 1: Naive CUDA Kernel

We implement a numerically stable softmax operation:

$$\text{softmax}(x_i) = \frac{e^{x_i - \max(x)}}{\sum_j e^{x_j - \max(x)}}$$

The naive kernel uses:

- Global memory accesses
- Atomic operations for reductions
- One block per row

Observations:
The kernel was validated by the python script. The output matches the pytorch reference within $\sim 10^{-10}$ precision.

2 Task 2: Optimized Kernel

The optimized kernel improves performance by:

- Reusing data via shared memory
- Eliminating global atomics
- Performing tree-based reductions

Shared Memory Usage:
All row elements are loaded once and reused for max, sum, and normalization.

Observations:

The optimized kernel achieved a higher performance. The kernel was validated by the correctness script. Shared memory reuse and the elimination of atomic operations significantly reduce memory latency, resulting in improved throughput.

3 Task 3: Benchmarking and Performance Comparison

3.1 Benchmarking Methodology

Benchmarking was performed using CUDA events to obtain accurate GPU execution times. Each implementation was run for 200 iterations after warm-up runs, with CUDA synchronization enforced to ensure correct timing. All experiments used identical input tensors of size 1024×512 with single-precision floating-point values.

3.2 Benchmark Results

Table 1 reports the average execution time per iteration for each implementation.

Implementation	Time (ms)	Relative Speed
PyTorch Softmax	0.026	1.0×
Naive CUDA Kernel	0.321	0.08×
Optimized CUDA Kernel	0.068	0.38×

Table 1: Performance comparison of softmax implementations

3.3 Performance Analysis

The naive CUDA kernel performs poorly due to repeated global memory accesses and the use of atomic operations, which introduce serialization and high memory latency. As a result, it underperforms the PyTorch baseline despite parallel execution.

The optimized kernel achieves a 4.7× speedup over the naive version by reusing data through shared memory and eliminating atomic operations. PyTorch remains faster due to its highly optimized, architecture-specific kernels that use warp-level primitives and additional low-level optimizations.

3.4 Bottleneck Identification

The naive kernel is primarily memory-bound and limited by atomic serialization. The optimized kernel reduces global memory traffic, shifting the bottleneck toward computation and block-level synchronization. PyTorch further reduces these overheads through advanced kernel design.

4 Task 4: Reading Production GPU Code

4.1 tiny-cuda-nn

tiny-cuda-nn emphasizes kernel fusion and fine-grained tiling to maximize arithmetic intensity and minimize global memory accesses. Its kernels are highly specialized for fixed tensor shapes, whereas our implementation prioritizes generality and clarity.

4.2 FlashAttention

FlashAttention improves performance through algorithmic restructuring that avoids materializing large intermediate tensors, allowing computation to remain within on-chip memory. This contrasts with our kernel-level optimizations, which focus on memory reuse within a single kernel.

5 Conclusion

This assignment highlights the impact of memory-aware kernel design on GPU performance. While the naive kernel prioritizes correctness, it suffers from inefficient memory access patterns. The optimized kernel demonstrates how shared memory reuse and reduction restructuring can significantly improve performance while maintaining correctness.