Optimization of Scaled Dot-Product Attention with FlashAttention3

Gül Oymak

Abstract

This report presents the implementation and evaluation of FlashAttention3 in a transformer-style self-attention layer. I compare performance metrics between standard PyTorch scaled dot-product attention (which already integrates FlashAttention2 optimizations) and my FlashAttention3 implementation across training and inference tasks. The evaluation demonstrates runtime improvements and resource utilization comparisons.

1 Introduction

Self-attention is a core component of transformer models, where the scaled dot-product attention computes weighted sums of value vectors based on query-key similarities. Recent advancements such as FlashAttention algorithms optimize memory access patterns and computational efficiency on GPUs. PyTorch's native `scaled_dot_product_attention` leverages FlashAttention2 under the hood for improved performance. In this project, I integrate FlashAttention3 (Hopper) into a custom transformer block and evaluate its impact on throughput and GPU utilization.

2 Implementation

2.1 Model Architecture

I built a transformer network using:

- A custom `Attention` module with `wq, wk, wv, wo` linear projections.
- Rotary positional embeddings applied to queries and keys.
- Conditional execution of FlashAttention3 via `flash_attn_func` when `use_flash_attn=True`, otherwise falling back to `F.scaled_dot_product_attention` (FlashAttention2).

My training code includes learning rate warmup, gradient clipping, and optional TorchDynamo compilation (skipped due to incompatibility with FlashAttention3).

3 Experimental Setup

- **Hardware**: Single NVIDIA Hopper GPU.
- **Dataset**: Parquet text dataset, sequence length 4096, batch size 1.
- **Training**: 1000 steps with logging every 5 steps.
- **Metrics**: Tokens/sec, training token ratio (%), MFU (%), TFLOPs.
- **Inference**: 50-step average latency measurement.

4 Correctness Tests

I verified that all attention backends produce equivalent results:

- **Output Consistency**: I compare logits from `flash2` and `flash3` against `standard` using `torch.allclose(..., rtol=1e-3, atol=1e-3)`.
- **Loss Alignment**: I confirm losses match within 0.5% relative error on the same batch and seed.
- **Gradient Check**: I optionally use `torch.autograd.gradcheck()` and ensure gradients align via `allclose()`.
- **Stability**: I check for no `NaN`/`Inf` in outputs or gradients and test both `bf16` without overflow.
- **Cross-Mode Loading**: I save a checkpoint in one mode and reload it in another, confirming outputs remain consistent.

5 Results

Table 1 summarizes the average metrics over the training phases.

Table 1: Average Performance Metrics (1000 steps)

Model	Tokens/sec	Train Token %	MFU %	TFLOPs
Baseline (standard)	7893.10	22.59	41.13	406.82
FlashAttention3	8127.71	22.59	42.36	418.91

Inference latency decreased from 150.28 ms (baseline) to 141.08 ms (FlashAttention3), yielding a speedup of approximately 1.06x.

6 Discussion

The integration of FlashAttention3 yielded a 2.95% increase in training throughput and a 6.2% reduction in inference latency compared to the baseline. Memory utilization remained constant due to similar allocation patterns. I observe that FlashAttention3's optimized kernel improves overall efficiency beyond the enhancements already provided by FlashAttention2 within PyTorch.