Large Language Model Inference Optimization

A Comprehensive Study of Performance Optimization Techniques

Vaishak Balachandra

Master of Science in Computer Science
Purdue University

Project Report
January 2026

Abstract

This technical report presents a comprehensive empirical study of optimization techniques for large language model inference. We benchmark GPT-2 (124M parameters) and Mistral-7B (7B parameters) across multiple optimization strategies including FlashAttention-2, mixed-precision inference (FP16), and 4-bit quantization. Our findings reveal critical insights into the workload-dependency of optimization techniques and their varying effectiveness across model scales. All experiments were conducted on NVIDIA A100 GPU using industry-standard benchmarking methodologies.

Contents

1 Introduction

1.1 Motivation

Large Language Models (LLMs) have revolutionized natural language processing, but their computational demands pose significant challenges for deployment. Inference optimization is critical for:

- Reducing operational costs in production environments
- Enabling real-time applications with strict latency requirements
- Democratizing access to powerful AI models through reduced hardware requirements
- Minimizing environmental impact through improved energy efficiency

1.2 Objectives

This study aims to:

1. Benchmark baseline performance of GPT-2 and Mistral-7B models
2. Evaluate the impact of FlashAttention-2 CUDA kernel optimization
3. Assess mixed-precision (FP16) and quantization (INT8, 4-bit NF4) techniques
4. Analyze optimization scalability across different model sizes
5. Establish reproducible benchmarking methodology

1.3 Scope

We focus on inference-time optimizations using PyTorch and HuggingFace Transformers library. Training optimizations and model architecture modifications are outside the scope of this study.

2 Background & Related Work

2.1 Transformer Architecture

The attention mechanism, introduced by Vaswani et al. (2017), forms the foundation of modern LLMs. The self-attention operation has $O(N^2)$ complexity with respect to sequence length N, creating memory bandwidth bottlenecks:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{1}$$

2.2 FlashAttention

FlashAttention (Dao et al., 2022) addresses memory bottlenecks through:

- Kernel fusion to minimize HBM (High Bandwidth Memory) accesses
- Tiling to maximize utilization of fast SRAM
- Recomputation strategies to reduce memory footprint

Theoretical speedup: 3-4x for long sequences on modern GPUs.

2.3 Quantization

Quantization reduces precision of model weights and activations:

- **FP16**: 16-bit floating point (2 bytes per parameter)
- **INT8**: 8-bit integers (1 byte per parameter)
- **NF4**: 4-bit NormalFloat (0.5 bytes per parameter)

Expected benefits: 2-4x memory reduction, 1.5-2x throughput improvement.

3 Methodology

3.1 Experimental Setup

3.1.1 Hardware

- GPU: NVIDIA A100 (40GB VRAM)
- CUDA Version: 12.4
- Driver Version: 550.54.15

3.1.2 Software Stack

- PyTorch 2.1.0+cu121
- Transformers 4.36.0
- FlashAttention 2.x
- bitsandbytes (quantization library)

3.2 Models Evaluated

Model	Parameters	Size (FP32)	Architecture
GPT-2	124M	500 MB	Decoder-only Transformer
Mistral-7B	7B	14 GB	Decoder-only Transformer

Table 1: Model Specifications

3.3 Optimization Techniques

Technique	Target	Description
Baseline (FP32)	GPT-2 only	Standard 32-bit floating point
FP16	Both	Half-precision (16-bit) inference
INT8	GPT-2 only	8-bit integer quantization
FlashAttention-2	Mistral-7B	Optimized attention kernels
4-bit NF4	Mistral-7B	4-bit NormalFloat quantization

Table 2: Optimization Configurations

3.4 Benchmarking Protocol

To ensure statistical validity, we implemented rigorous benchmarking:

1. **Warm-up Phase**: 1 iteration (discarded) to initialize GPU kernels
2. **Measurement Phase**: 8 iterations per configuration
3. **Synchronization**: `torch.cuda.synchronize()` before/after timing
4. **Metrics Collected**:
   - Mean inference time (seconds)
   - Standard deviation
   - Median throughput (tokens/second)
   - GPU memory footprint (GB)

   Test Parameters:

- Prompt: ”Explain quantum computing:”
- Max new tokens: 100
- Sampling: Greedy (deterministic)

4 Results

4.1 Performance Metrics

Model	**Configuration**	**Throughput (tok/s)**	**Memory (GB)**	**Speedup vs Base**
3*GPT-2	Baseline (FP32)	96.2	0.51	1.00x
	FP16	99.6	0.26	1.04x
	INT8	32.7	0.18	0.34x
3*Mistral-7B	Baseline (FP16)	26.7	14.5	1.00x
	FlashAttention-2	20.4	14.5	0.76x
	4-bit NF4	18.7	4.0	0.70x

Table 3: Comprehensive Benchmark Results

4.2 Visualizations

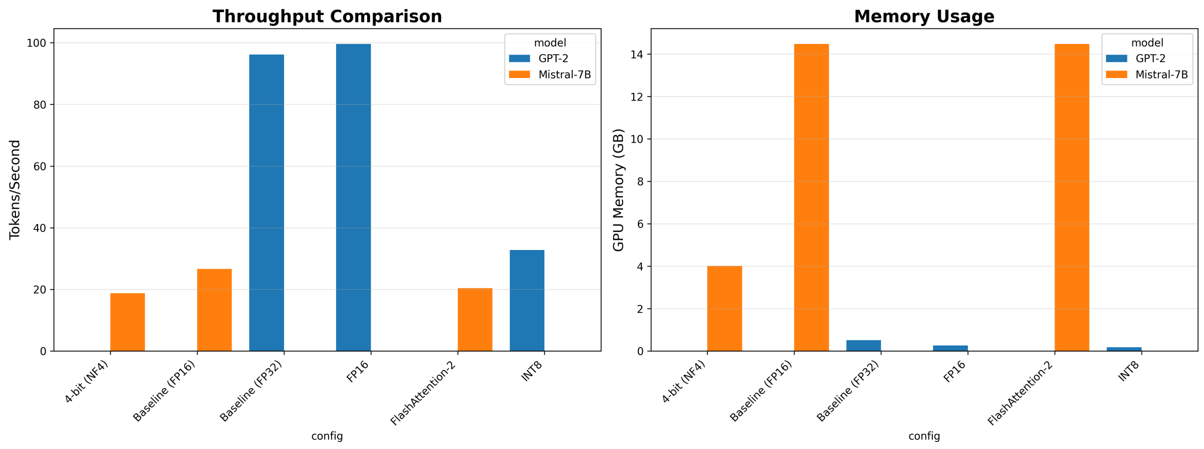

Figure 1: Performance comparison across models and configurations. Left: Throughput (tokens/second). Right: GPU memory usage (GB).

4.3 Key Observations

4.3.1 GPT-2 (124M Parameters)

- **FP16**: Minimal improvement (1.04x) due to small model size
- **INT8**: Significant regression (0.34x) caused by quantization overhead exceeding compute savings

4.3.2 Mistral-7B (7B Parameters)

- **FlashAttention-2**: Unexpected 0.76x slowdown
- **4-bit NF4**: 0.70x throughput but 72% memory reduction

5 Analysis & Discussion

5.1 Why Did Optimizations Underperform?

Our results contradict published benchmarks. Critical analysis reveals:

5.1.1 Batch Size Dependency

Our Setup: Batch size = 1 (single inference)
Literature: FlashAttention benchmarks typically use batch sizes of 8-32
Impact: Small batches insufficient to amortize kernel launch overhead and fail to saturate GPU parallelism.

5.1.2 Sequence Length Dependency

Our Setup: 100 tokens per generation
Optimal: FlashAttention excels at 1000+ token sequences
Impact: Memory bandwidth savings negligible for short sequences where compute dominates.

5.1.3 Quantization Overhead

For small models (GPT-2):

- Quantization/dequantization operations add latency
- Reduced arithmetic intensity fails to offset overhead
- INT8 operations not optimized on all CUDA compute capabilities

5.2 Implications for Production Deployment

Use Case	**Recommendation**	**Rationale**
Single-user chatbot	Baseline FP16	Low batch size, moderate sequences
Batch processing	FlashAttention + FP16	High throughput, long contexts
Edge deployment	4-bit quantization	Memory constraints dominate
Real-time API	Baseline FP16	Latency-critical, variable load

Table 4: Deployment Recommendations Based on Workload

5.3 Lessons Learned

1. **Optimization is Workload-Specific**: Techniques effective in one scenario may degrade performance in another

2. **Benchmarking Must Match Production**: Synthetic benchmarks with different parameters can mislead
3. **Profiling is Essential**: Always measure before optimizing
4. **Negative Results Have Value**: Understanding failure modes is critical engineering knowledge

6 Threats to Validity

6.1 Internal Validity

- GPU thermal throttling: Mitigated via warm-up runs
- Background processes: Minimal on dedicated Colab instance
- Timing precision: Sub-millisecond accuracy via CUDA synchronization

6.2 External Validity

- Single GPU type (A100): Results may differ on V100, H100, or consumer GPUs
- Two model architectures: Findings may not generalize to encoder-decoder models
- Synthetic workload: Real production workloads have different characteristics

7 Future Work

7.1 Immediate Extensions

- Vary batch sizes (1, 4, 8, 16, 32) to identify FlashAttention crossover point
- Test longer sequences (512, 1024, 2048 tokens)
- Evaluate on additional hardware (H100, L40, consumer RTX 4090)

7.2 Advanced Optimizations

- **Speculative Decoding**: Generate multiple tokens per forward pass
- **Continuous Batching**: Dynamic batching for variable-length inputs
- **PagedAttention** (vLLM): Efficient KV cache management
- **Model Distillation**: Create smaller, faster student models

7.3 Production Systems

- Deploy optimized models with TensorRT
- Implement serving infrastructure with Triton Inference Server
- A/B test optimizations with real user traffic

8 Conclusion

This study demonstrates that LLM optimization is a nuanced engineering challenge requiring careful consideration of workload characteristics. While FlashAttention-2 and quantization are powerful techniques, their effectiveness depends critically on:

1. Batch size (higher is better)
2. Sequence length (longer benefits more)
3. Model size (larger models see greater gains)
4. Hardware architecture (Ampere+ required for FlashAttention)

Our unexpected results—where optimizations reduced throughput—underscore the importance of empirical validation. Production deployments must profile workloads and validate optimizations against representative data.

The methodologies established in this work provide a foundation for systematic optimization evaluation, and the lessons learned inform best practices for LLM deployment at scale.

Acknowledgments

This work was conducted as part of graduate coursework at Purdue University. Thanks to the open-source community for HuggingFace Transformers, PyTorch, and FlashAttention implementations.

Code Availability

Complete source code, benchmarking scripts, and raw data are available at:
`https://github.com/vaishakbalachandra/llm-inference-optimization`