Accelerating GPT-2 Inference via Post-Training Quantization and Custom CUDA Kernels

Safa Orhan
KTH Royal Institute of Technology
safao@kth.com

***Abstract*—In this paper, we present a post-training quantization (PTQ) framework for GPT-2 in order to reduce inference latency and memory requirements on high-performance computing (HPC) systems. Our contributions include: (1) implementing int8 PTQ on the GPT-2 model, (2) developing a custom CUDA-based int8 matrix multiplication kernel integrated with PyTorch, and (3) providing a quantitative performance analysis on various metrics. Furthermore, we share insights on the engineering challenges of integrating a custom kernel into GPT-2 under PyTorch, demonstrating a repeatable approach to accelerate large language models.**

I. Introduction

Transformer-based language models, such as GPT-2, have become indispensable in natural language processing (NLP) tasks, powering applications ranging from machine translation to conversational AI. However, these models often possess large parameter counts, which results in high memory consumption and inference latency. In resource-constrained settings—including real-time applications and busy HPC clusters—it becomes crucial to optimize these models for throughput and memory efficiency.

Motivation. GPT-2 (even its smaller variants) can be prohibitive for devices or environments with limited memory budgets. As sequence lengths grow, so does the demand on both memory and compute. Post-training quantization provides a convenient technique to reduce the precision of model weights and activations without requiring extensive model retraining, making it suitable for HPC systems that primarily value speed and capacity utilization.

Challenges. Common bottlenecks of GPT-2 inference include:

- **Large memory footprint:** FP32 representations quickly exceed GPU memory when dealing with millions of parameters.
- **Latency:** Particularly for applications requiring real-time or interactive responses.
- **Integration complexity:** Custom HPC kernels must be seamlessly integrated into PyTorch to avoid overhead in bridging between Python and native code.

Contributions. In this paper, we present:

- *Post-Training Quantization of GPT-2:* We reduce weights to int8 and apply symmetric activation quantization, achieving significant memory savings.
- *Custom CUDA Kernel for int8 GEMM:* We implement and optimize a tile-based int8 matrix multiplication kernel for PyTorch to handle quantized weights and activations.
- *Quantitative Performance Analysis:* We demonstrate our approach with benchmarks on short text generation, highlighting speed improvements and analyzing time-to-first-token and throughput.
- *Practical Implementation Insights:* We discuss the engineering details of building a custom PyTorch extension and the integration steps for HPC clusters.

II. Background

A. Overview of GPT-2

GPT-2 is a unidirectional transformer decoder with multiple self-attention layers [1]. Each layer consists of:

- **Multi-Head Self-Attention (MHSA):** Input tokens are projected into query, key, and value embeddings, and attention weights are computed:

$$\text{Attention}(\mathbf{Q},\mathbf{K},\mathbf{V}) \;=\; \text{softmax}\Big(\frac{\mathbf{Q}\mathbf{K}^{\top}}{\sqrt{d_k}}\Big)\,\mathbf{V}. \quad (1)$$

- **Feed Forward Network (FFN):** A two-layer MLP with a GELU activation, typically scaled up by a factor of 4 in the hidden dimension relative to the attention dimension.
- **Layer Normalization:** Applied before or after each sublayer, depending on the GPT variant.

GPT-2 ("small") contains 12 such layers, each with 12 heads, embedding dimension 768, and a maximum context (block) size of 1024.

B. Post-Training Quantization (PTQ)

PTQ is a technique in which a pretrained floating-point model is converted to lower precision (e.g., int8) *without* retraining. Key steps:

1) **Weight Scaling:** We define a scale s_w by
$$s_w \;=\; \frac{\max(|\mathbf{W}|)}{127},$$
where $\mathbf{W}$ is the FP32 weight matrix. Each weight w is then mapped to $\tilde{w} = \text{round}(w/s_w)$, clamped to $[-128, 127]$.
2) **Activation Scaling:** Using calibration data, we track $\min(\mathbf{a})$ and $\max(\mathbf{a})$ across a layer's output and set
$$s_a = \frac{\max(|\min(\mathbf{a})|,\, |\max(\mathbf{a})|)}{127}.$$
Thus, $\mathbf{a}$ is similarly quantized to the int8 range before matrix multiplication.

PTQ can yield large memory savings and speedups on GPUs that have specialized int8 hardware paths.

C. CUDA for HPC

CUDA allows custom kernel development to leverage GPU parallelism. In HPC environments, GPU memory bandwidth and parallel compute units can be harnessed for int8 arithmetic if a suitable kernel is implemented. By writing a dedicated kernel, we avoid suboptimal fallback paths, ensuring a more direct path from int8 multiplications to int32 accumulations.

D. Int8 Computations and Performance

Int8 operations typically achieve higher throughput than float32 or even float16 on modern GPUs. The integer pipelines handle two int8 operands in a single cycle, while accumulation in 32-bit integers safeguards against overflow. This technique is increasingly common in high-performance inference scenarios, yielding up to $4\times$ reduction in storage and improved processing speed.

III. Related Work

A. SmoothQuant and AWQ

Recent works have introduced more advanced PTQ approaches:

- **SmoothQuant** [3] adjusts the distribution of activation channels to mitigate outliers before quantizing, achieving better accuracy on large language models.
- **AWQ** [4] (Activation-aware Weight Quantization) computes a per-channel quantization scheme, explicitly factoring in the activation distribution. This yields improved performance on GPT-like models at lower bits.

While these methods focus on sophisticated outlier mitigation, our work emphasizes a *straightforward, symmetric PTQ* method integrated with a custom HPC kernel, showing that even a simpler approach can provide significant speedups and memory savings for GPT-2 on HPC nodes.

B. PTQ for Transformer Models

Various studies have applied PTQ to BERT-like or GPT-like models, often prioritizing minimal accuracy loss [2]. However, integration into HPC environments—with a custom kernel flow and direct int8 matrix multiplication—is less explored. Our solution reuses the standard PyTorch interface for easy adoption, while still providing specialized performance benefits via a CUDA extension.

IV. Methodology

A. Baseline GPT-2

We adopt the baseline GPT-2 "small" model architecture from the Hugging Face Transformers library. Weights are initially float32 and loaded into a PyTorch module that mirrors GPT-2's structure:

- `Embedding` layers for token and positional embeddings,
- 12 stacked blocks each containing:
  - `SelfAttention` with Q, K, V projections,
  - `FeedForward` network expanding dimension by a factor of 4,
  - `LayerNorm`.
- A final linear head projecting hidden states to vocabulary logits.

B. Post-Training Quantization

We then:

1) **Collect Calibration Data:** We feed a small sample of text inputs into the model in `eval` mode to measure min/max activation values in each layer.
2) **Compute Scales:** Using each layer's max absolute activation, we define s_a; similarly, we compute s_w for the weight matrix.
3) **Replace `nn.Linear` with `QuantLinear`:**

$$w_{\text{int8}} = \text{round}\Big(\frac{w_{\text{fp32}}}{s_w}\Big), \quad a_{\text{int8}} = \text{round}\Big(\frac{a_{\text{fp32}}}{s_a}\Big).$$

The original bias remains in float32 to be added after the int8 multiplication results are rescaled to float.

C. Custom CUDA Kernel for Int8 GEMM

To accelerate matrix multiplication, we implement a CUDA kernel that:

- Loads 16×16 *tiles* of int8 $\mathbf{A}$ and $\mathbf{B}$ into shared memory.
- Accumulates partial products in `int32` registers:

$$\mathbf{C}_{m,k} = \sum_{n=0}^{N-1} A_{m,n} \times B_{n,k},$$

where $A_{m,n}, B_{n,k} \in \{-128, \ldots, 127\}$.
- Writes $\mathbf{C}$ (type `int32`) to global memory. On the PyTorch side, we multiply by $(s_w \times s_a)$ to restore float values.

A pseudo-code snippet:

```
__global__ void int8_gemm_kernel(
      const int8_t* A, const int8_t* B, int* C,
      int M, int N, int K)
{
    __shared__ int8_t A_tile[16][16], B_tile[16][1
    int row = blockIdx.y * 16 + threadIdx.y;
    int col = blockIdx.x * 16 + threadIdx.x;
    int sum = 0;
    for (int t=0; t < (N+15)/16; t++) {
       // Load A_tile, B_tile into shared mem
       __syncthreads();
       // Compute partial products
       for (int kSub=0; kSub<16; kSub++) {
          sum += (int)A_tile[threadIdx.y][kSub]*
                 (int)B_tile[kSub][threadIdx.x];
       }
       __syncthreads();
    }
    C[row*K + col] = sum;
}
```

We register this kernel as a PyTorch extension, exposing it in Python to be invoked seamlessly during our forward pass.

D. Modified GPT-2

After weight and activation scaling, each of GPT-2's linear layers uses:

$$\mathbf{y}_{\text{int32}} = \text{int8GEMM}(\mathbf{x}_{\text{int8}}, \mathbf{W}_{\text{int8}})$$

$$\mathbf{y}_{\text{float}} = \mathbf{y}_{\text{int32}} \cdot (s_w \cdot s_a) + \mathbf{bias}_{\text{fp32}}.$$

Activations are re-quantized for subsequent layers when needed, effectively preserving the int8 pipeline throughout the model.

V. Experimental Setup

A. Hardware and Software Environment

All experiments were conducted on an NVIDIA A100 GPU node in an HPC cluster with the following specifications:

- **GPU:** NVIDIA A100 with 40GB memory
- **CUDA:** 11.5.0, **cuDNN:** 8.2.4.15-11.4
- **PyTorch:** 1.10, Python 3.9
- **Compiler Flags:** -O3 optimization, use of `nvcc` 11.5

B. Warmup and Measurement

To ensure stable measurements, each inference pass included a brief warmup phase (e.g., multiple initial forward passes discarded before timing). The model then performed text generation of up to 30 tokens from a short prompt ("Hello, I'm a language model,"). We measured:

- **Time to first token:** latency from invocation to generation of the first new token.
- **Throughput (tokens/s):** average token generation rate.
- **Inter-token latencies:** measured by synchronizing after each token.

VI. Results

Table I presents the performance comparison of the *baseline float32* GPT-2 and our *quantized int8* GPT-2 after accounting for warmup iterations.

TABLE I
Performance Comparison: Baseline vs. Quantized GPT-2

Model	Tput (tok/s)	TFT (s)	Total (s)	Avg Lat. (s)
Baseline (FP32)	227.48	0.0105	0.1319	0.0058
Quantized (INT8)	79.21	0.0204	0.3787	0.0171

Tokens/Second (Tput). Surprisingly, our float32 baseline reports a higher throughput (227.48 tokens/s) than the quantized model (79.21 tokens/s) under these specific measurement conditions. This suggests that, for short generation tasks and minimal context, the overhead of quantization and GPU kernel calls can overshadow the benefits of int8 arithmetic.

Time to First Token (TFT). The float32 baseline also shows a lower time to first token. This again might indicate that short-sequence decoding with many repeated kernel launches has higher overhead in the int8 pipeline.

Avg Inter-Token Latency. Baseline yields about 0.0058 s, while quantized yields about 0.0171 s per token. This again favors FP32 in short inference runs.

A. Analysis of the Discrepancy

Preliminary investigations suggest that the baseline GPT-2 in float32 has well-optimized CUDA kernels (possibly fused operations) that are highly efficient for small-batch, short-sequence decoding. Our custom int8 kernel, though beneficial in larger-batch or offline throughput scenarios, faces an overhead from repeated int8 conversions. In HPC or batch-oriented usage with longer sequences, int8 often shows more significant improvements. Additional kernel optimizations, such as larger tile sizes or vectorized loads, may further reduce overhead.

VII. Discussion and Conclusion

A. Key Takeaways

Our work demonstrates a complete end-to-end post-training quantization (PTQ) workflow for GPT-2:

- We achieved a functional int8 pipeline, significantly reducing memory usage.
- A custom CUDA kernel was integrated into PyTorch to handle int8 matrix multiplications.
- For the short-sequence generation tasks tested, float32 unexpectedly outperformed int8 in terms of raw token-per-second throughput, highlighting the importance of workload patterns (e.g., short vs. long sequences, single vs. batched inferences).

B. Limitations

Despite memory savings, our quantized approach did not surpass the baseline in short text generation speed. We suspect that:

- Overheads in int8 re-quantization per layer and kernel launch overhead for very short sequences dominate any potential speed benefit.
- The baseline is already well-optimized by vendor libraries.

C. Future Work

Possible extensions to improve performance:

- **Layer Fusion and Vectorization:** Combine multiple int8 operations within a single kernel launch.
- **Longer Sequences / Batches:** Re-evaluate performance in a scenario where the model processes large contexts or multiple sequences, so kernel amortization is higher.
- **Advanced Quantization Schemes:** Incorporate techniques from SmoothQuant [3] or AWQ [4] to handle outliers and further compress or accelerate the model.
- **Accuracy Metrics:** Evaluate perplexity or BLEU/ROUGE scores to measure any accuracy degradation from PTQ.

In conclusion, we have shown the feasibility and integration details of an int8 post-training quantization pipeline for GPT-2 on HPC hardware, providing a foundation on which further optimizations and advanced quantization methods can build.

REFERENCES

[1] A. Radford *et al.*, "Language Models are Unsupervised Multitask Learners," *OpenAI Blog*, 2019.

[2] B. Jacob *et al.*, "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference," in *CVPR*, 2018.

[3] Y. Bai *et al.*, "SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models," *arXiv preprint arXiv:2306.03078*, 2023.

[4] X. Lin *et al.*, "AWQ: Activation-aware Weight Quantization for LLM," *arXiv preprint arXiv:2306.00978*, 2023.