High Performance LLM Serving on Nvidia GPUs – Sharan Chetlur
Summary notes

Introduction:

- Large Language Models (LLMs) are one of the defining workloads of high-performance computing today.
- There is an increasing need for deep learning in language applications such as chat, translation, summarization, search, and generation.
- Accuracy is crucial as it directly translates to how helpful the model is to the user.
- Online deployments require end-user acceptable latencies to ensure a great experience.
- Multifunctional, accurate models are large, making them slow during inference and expensive to deploy, making cost-effective deployments challenging.

LLM Ecosystem:

- There is a rapid increase in the release of new foundational LLMs (Llama, Falcon, Starcoder, ChatGLM, MPT, etc.).
- New operators and customization techniques make optimization a moving target.
- The latest models continue to be very large (70-200 billion parameters or more) for the best accuracy.
- A performant, robust, and extensible solution is needed for cost-effective, real-time LLM deployments.

TensorRT-LLM:

- TensorRT-LLM aims to provide state-of-the-art performance for large language models for production deployments on NVIDIA GPUs.
- It offers ease of extension, allowing developers to add new operators or models in Python to support new LLMs with optimized performance.
- TensorRT-LLM leverages TensorRT compilation and custom kernels.

Quantization:

- BFloat16 is the default data precision in the world of LLMs.
- Inference can be done in lower bit-width precisions (FP8, INT8, INT4, or lower) for better performance and cost savings.
- Post-training quantization (PTQ) is a common technique to convert BF16 trained checkpoints into lower precision, serviceable models without additional training or weight updates.
- Quantization-aware training (QAT) can also be used for better accuracy by retraining the model on a smaller dataset.

- TensorRT Model Optimizer offers techniques like PTQ, QAT (with FP8, INT8, and INT4), and sparsity.

Phases of an LLM Request:

- There are two phases: prefill and generate.
- In the prefill phase, the user supplies a prompt, and the first output token and a KV cache (intermediate activations for reuse) are produced.
- In the generate phase, starting from the previous state (KV cache and last generated token), the next token is generated, and the KV cache is updated.
- The prefill phase has more parallelism and is compute-bound, while the generate phase is more memory-bandwidth-bound.
- Attention operations straddle the line between compute-bound and memory-bound, depending on the implementation.

Request Scheduling:

- Traditional inference treats a request as the atomic unit of scheduling, accumulating requests over a time window and running them in a batch.
- This approach is problematic for LLMs due to variable completion lengths.
- In-flight batching treats a single LLM iteration as the atomic unit of scheduling.
- Requests are dynamically admitted and processed in an online setting, maximizing throughput and GPU utilization.

Token Concatenation:

- Transformer models consist of token-parallel operations (MatMul, LayerNorm) and sequence-parallel operations (Attention).
- Iteration runtime is dominated by weight matrix multiplies, which are token-parallel.
- Token concatenation across requests in the prefill and generate stages allows for higher matrix multiplication efficiency and throughput.

Paged KV Cache:

- Traditionally, KV caches were viewed as contiguous regions of memory, leading to memory waste.
- Paged KV caches represent the KV cache as a linked list of pages, minimizing unused memory.
- Requests can share physical memory regions, reducing duplication and computation for shared tokens (e.g., system prompts).

Speculative Decoding:

- At low batch sizes, the generation phase significantly underutilizes GPU resources.
- Speculative decoding allows guessing and validating multiple future tokens (n, n+1, n+2, ...) at approximately zero additional time.
- This technique statistically leads to lower latency gains by advancing the generation state by more than one request.