LLM Deployment

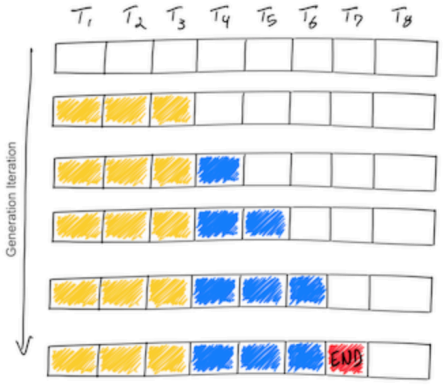
LLM Inference

LLM inference is memory-IO bound, not compute bound. In other words, it currently takes more time to load 1MB of data to the GPU's compute cores than it does for those compute cores to perform LLM computations on 1MB of data. This means that LLM inference throughput *is largely determined by how large a batch you can fit into high-bandwidth GPU memory*.

This is why approaches such as model quantization strategies such as AutoGPTQ are potentially so powerful; if you could halve the memory usage by moving from 16-bit to 8-bit representations, you could double the space available for larger batch sizes. However, not all strategies require modifications to the model weights. For example, FlashAttention found significant throughput improvements by reorganizing the attention computation to require less memory-IO.

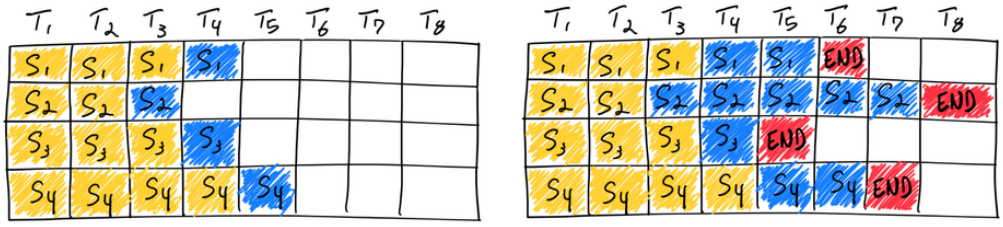
Continuous batching is another memory optimization technique which does not require modification of the model.

- You start with a sequence of tokens (called the "prefix" or "prompt").
- The LLM produces a sequence of completion tokens, stopping only after producing a stop token or reaching a maximum sequence length.



Starting from the prompt tokens (yellow), the iterative process generates a single token at a time (blue). Once the model generates an end-of-sequence token (red), the generation loop stops. This example shows a batch of only one input sequence, so the batch size is 1.

Static Batching

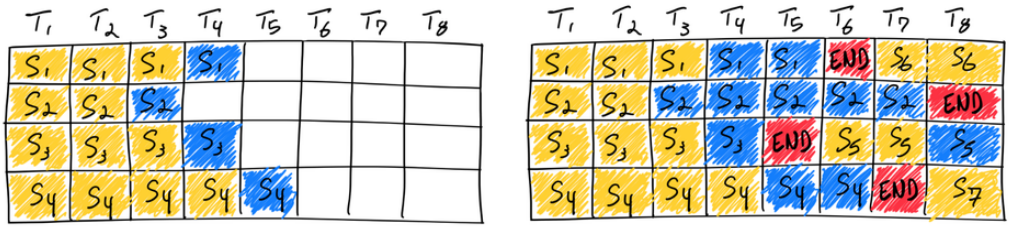


Completing four sequences using static batching. On the first iteration (left), each sequence generates one token (blue) from the prompt tokens (yellow). After several iterations (right), the completed sequences each have different sizes because each emits their end-of-sequence-token (red) at different iterations. Even though sequence 3 finished after two iterations, static batching means that the GPU will be underutilized until the last sequence in the batch finishes generation (in this example, sequence 2 after six iterations).

How often does static batching under-utilize the GPU? It depends on the generation lengths of sequences in a batch. If the input sequences are also the same size (say, 512 tokens) and all output token sequences are also the same size (say, 1 token), then each static batch will

achieve the best possible GPU utilization. However, this is not always the case e.g. a LLM-powered chatbot service cannot assume fixed-length input sequences, nor assume fixed-length output sequences.

Continuous Batching



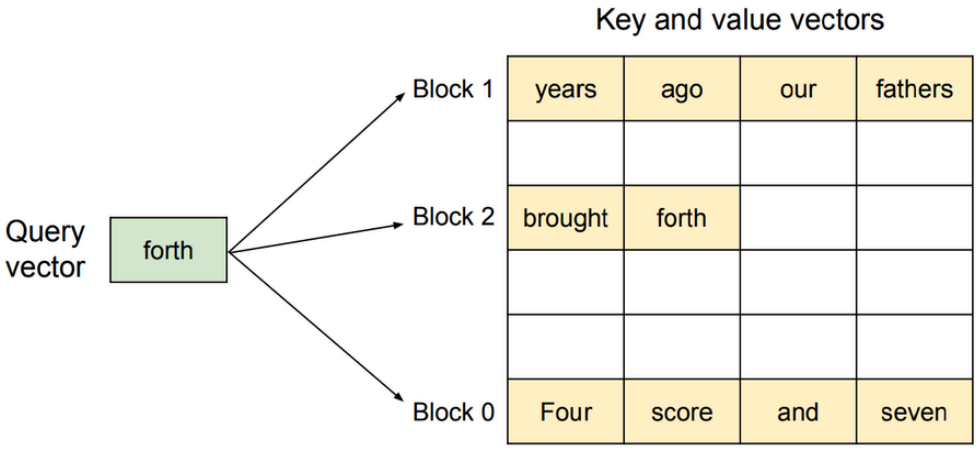
Once a sequence emits an end-of-sequence token, we insert a new sequence in its place (i.e. sequences S5, S6, and S7). This achieves higher GPU utilization since the GPU does not wait for all sequences to complete before starting a new one.

Why is there a Need for PagedAttention?

LLMs like GPT-4 can have trillions of parameters, making them extremely powerful but also incredibly memory-hungry when inferencing during serving. The main bottleneck of memory is due to the KV cache.

During the decoding process of transformer-based LLMs, as each input token is processed, the model generates corresponding attention key and value tensors. These key and value tensors encode important contextual information about the current input and its relationship to the broader context. Rather than recomputing these attention-related tensors from scratch for each step of the decoding process, the model stores them in GPU memory. This stored collection of key and value tensors is commonly referred to as KV cache.

By maintaining the KV cache, LLMs can retrieve and reuse the pre-computed contextual information when generating the next output token during inference. The cache acts as a sort of "memory" for the model to draw upon. The core idea behind Paged Attention is to partition the KV cache of each sequence into smaller, more manageable "pages" or blocks. Each block contains key-value vectors for a fixed number of tokens. This way, the KV cache can be loaded and accessed more efficiently during attention computation.



Ray

Ray is an open-source unified compute framework that makes it easy to scale AI and Python workloads — from reinforcement learning to deep learning to tuning, and model serving.

```
# On head node
ray start --head --dashboard-host "0.0.0.0"

# On worker nodes
```

```
ray start --address=<ray-head-address>:6379

serve run vllm_serve:app
```

VLLM

vLLM is a fast and easy-to-use library for LLM inference and serving.

vLLM is fast with:

- State-of-the-art serving throughput
- Efficient management of attention key and value memory with **PagedAttention**
- Continuous batching of incoming requests
- Fast model execution with CUDA/HIP graph
- Quantization: GPTQ, AWQ, SqueezeLLM, FP8 KV Cache
- Optimized CUDA kernels

Mixtral-instruct

```
python3 -u -m vllm.entrypoints.openai.api_server \
       --host 0.0.0.0 \
       --model casperhansen/mixtral-instruct-awq \
       --tensor-parallel-size 4 \
       --enforce-eager \
       --quantization awq \
       --gpu-memory-utilization 0.96 \
       --kv-cache-dtype fp8
```

```
url = "http://localhost:8000/v1/completions"
```

[23]:
```
max_tokens = 32768 - len(query.split())
max_tokens
```

[23]:
```
32130
```

[26]:
```
len(query)
```

[26]:
```
4703
```

[31]:
```
from transformers import AutoTokenizer

max_tokens = 32768
tokenizer = AutoTokenizer.from_pretrained("casperhansen/mixtral-instruct-awq")
tokens = tokenizer.tokenize(query)
len(tokens)
```

[31]:
```
1441
```

[36]:
```
available_tokens = max_tokens - len(tokens) - 2
available_tokens
```

[36]:
```
31325
```

[37]:
```
data = {
    "model": "casperhansen/mixtral-instruct-awq",
    "prompt": query,
    "max_tokens": available_tokens,
    "temperature": 0
}

headers = {'Content-Type': 'application/json'}

resp = requests.post(url, json=data, headers=headers)
resp
```

[37]:
```
<Response [200]>
```

References:

- Distributed Inference and Serving — vLLM
- Quickstart — vLLM
- GitHub - vllm-project/vllm: A high-throughput and memory-efficient inference and serving engine for LLMs

TensorRT-LLM and Triton Inference Server

Triton Inference Server is an open source inference serving software that streamlines AI inferencing. Triton enables teams to deploy any AI model from multiple deep learning and machine learning frameworks, including TensorRT, TensorFlow, PyTorch, ONNX, OpenVINO, Python, RAPIDS FIL, and more. Triton Inference Server supports inference across cloud, data center, edge and embedded devices on NVIDIA GPUs, x86 and ARM CPU, or AWS Inferentia. Triton Inference Server delivers optimized performance for many query types, including real time, batched, ensembles and audio/video streaming.

Setting up

1. Download model of your choice, we will use `Mixtral-8x7B-Instruct-v0.1` You can you `huggingface-cli` to download

```
huggingface-cli download mistralai/Mixtral-8x7B-Instruct-v0.1 --repo-type model --local-dir-use-symlinks False
```

2. Clone GitHub - NVIDIA/TensorRT-LLM: TensorRT-LLM provides users with an easy-to-use Python API to define Large Language Models (LLMs) and build TensorRT engines that contain state-of-the-art optimizations to perform inference efficiently on NVIDIA GPUs. TensorRT-LLM also contains components to create Python and C++ runtimes that execute those TensorRT engines. This tool is built on top of TensorRT, allowing you to optimize LLMs in various data types e.g. float16, int8, int4, etc.

```
pip install tensorrt-llm
git clone -b v0.8.0 https://github.com/NVIDIA/TensorRT-LLM.git
cd TensorRT-LLM/examples/mixtral
cp ../llama/convert_checkpoint.py .

python3 convert_checkpoint.py --model_dir ~/models/Mixtral-8x7B-Instruct-v0.1 \
                              --output_dir trt_engines/Mixtral-8x7B-Instruct-v0.1/int8_wo_4gpu_ckpt \
                              --dtype float16 \
                              --use_weight_only \
                              --weight_only_precision int8 \
                              --pp_size 4 \
                              --load_model_on_cpu

trtllm-build --checkpoint_dir trt_engines/Mixtral-8x7B-Instruct-v0.1/int8_wo_4gpu_ckpt \
             --output_dir trt_engines/Mixtral-8x7B-Instruct-v0.1/int8_weight_only/4gpu \
             --gemm_plugin float16 --gpt_attention_plugin float16 --context_fmha enable \
             --use_paged_context_fmha enable --remove_input_padding enable

# run
mpirun -n 4 python3 ../run.py --engine_dir trt_engines/Mixtral-8x7B-Instruct-v0.1/int8_weight_only/4gpu \
                  --tokenizer_dir ~/models/Mixtral-8x7B-Instruct-v0.1 \
                  --max_output_len 512 \
                  --input_text "tell me about AI"
```

3. Once the above works, we will set up Triton Inference Server to deploy Mixtral as an API. For this, we need GitHub - triton-inference-server/backend: Common source, scripts and utilities for creating Triton backends. as a backend

```
git clone https://github.com/triton-inference-server/backend.git
cd tensorrtllm_backend
mkdir triton_model_repo
cp inflight_batcher_llm/* triton_model_repo/

# copy tokenizer
cp ~/models/Mixtral-8x7B-Instruct-v0.1/tokenizer* tensorrt_llm/mixtral

# copy TensorRT models (engines)
cp TensorRT-LLM/examples/mixtral/trt_engines/Mixtral-8x7B-Instruct-v0.1/int8_weight_only/4gpu/* triton_model_

# setup Triton configurations
python3 tools/fill_template.py -i triton_model_repo/preprocessing/config.pbtxt tokenizer_dir:/tensorrtllm_bac
python3 tools/fill_template.py -i triton_model_repo/postprocessing/config.pbtxt tokenizer_dir:/tensorrtllm_ba
python3 tools/fill_template.py -i triton_model_repo/tensorrt_llm_bls/config.pbtxt triton_max_batch_size:64,bl
python3 tools/fill_template.py -i triton_model_repo/ensemble/config.pbtxt triton_max_batch_size:64
python3 tools/fill_template.py -i triton_model_repo/tensorrt_llm/config.pbtxt triton_max_batch_size:64,engine

# run server
docker run --rm -it --net host --shm-size=2g --ulimit memlock=-1 --ulimit stack=67108864 --gpus all -v ~/tens
cd /tensorrtllm_backend/
python3 scripts/launch_triton_server.py --model_repo=/tensorrtllm_backend/triton_model_repo/ --world_size=4
```

[53]:
```
query = f"""
{PROMPT}

Resume:
{resume}

Answer: ```json"""

url = "http://localhost:8000/v2/models/mixtral/generate"
```

[54]:
```
data = {
    "max_tokens": available_tokens,
    "temperature": 0,
    "text_input": query,
}

headers = {'Content-Type': 'application/json'}

resp = requests.post(url, json=data, headers=headers)
resp
```

[54]:
```
<Response [200]>
```

[56]:
```
print(resp.json()["text_output"])
```

```
{
  "personal": {
    "gender": "",
    "full_name": "Peeranat Fupongsiripan",
    "birthplace": "",
    "first_name": "Peeranat",
    "family_name": "Fupongsiripan",
    "middle_name": "",
    "nationality": [],
    "date_of_birth": "",
    "marital_status": ""
  },
  "contact": {
    "email": ["peeranat85@gmail.com"],
    "phone": ["(+66) 83 008 9383"],
    "address": [],
```

References:

- GitHub - triton-inference-server/backend: Common source, scripts and utilities for creating Triton backends.
- GitHub - NVIDIA/TensorRT-LLM: TensorRT-LLM provides users with an easy-to-use Python API to define Large Language Models (LLMs) and build TensorRT engines that contain state-of-the-art optimizations to perform inference efficiently on NVIDIA GPUs. TensorRT-LLM also contains components to create Python and C++ runtimes that execute those TensorRT engines.
- Deploying a Large Language Model (LLM) with TensorRT-LLM on Triton Inference Server: A Step-by-Step…

Cost and Hardware Configuration

Monthly estimate

$3,605.00

That's about $4.94 hourly

Pay for what you use: no upfront costs and per second billing

Item	Monthly estimate
48 vCPU + 192 GB memory	$1,586.41
4 NVIDIA L4	$2,017.49
10 GB balanced persistent disk	$1.10
Total	**$3,605.00**

Machine configuration

Machine type	g2-standard-48
CPU platform	Intel Cascade Lake
Minimum CPU platform	None
Architecture	x86/64
vCPUs to core ratio	—
Custom visible cores	—
Display device	Disabled Enable to use screen capturing and recording tools
GPUs	4 x NVIDIA L4
Resource policies	