Best practices for optimizing large language model inference with GPUs on Google Kubernetes Engine (GKE)

Autopilot (/kubernetes-engine/docs/concepts/autopilot-overview) Standard (/kubernetes-engine/docs/concepts/choose-cluster-mode)

Google Kubernetes Engine (GKE) provides fine-grain control for large language model (LLM) inference with optimal performance and cost. This guide describes best practices for optimizing inference and serving of open LLMs with GPUs on GKE using the vLLM (https://github.com/vllm-project/vllm) and Text Generation Inference (https://huggingface.co/docs/text-generation-inference/en/index) (TGI) serving frameworks.

For a summarized checklist of all the best practices, see the Checklist summary (#checklist).

Objectives

This guide is intended for Generative AI customers, new or existing GKE users, ML Engineers, and LLMOps (DevOps) engineers who are interested in optimizing their LLM workloads using GPUs with Kubernetes.

By the end of this guide, you'll be able to:

- Choose post-training LLM optimization techniques, including quantization, tensor parallelism, and memory optimization.
- Weigh the high-level tradeoffs when considering these optimization techniques.
- Deploy open LLM models to GKE using serving frameworks such as vLLM or TGI with optimization settings enabled.

Overview of LLM serving optimization techniques

Unlike non-AI workloads, LLM workloads typically exhibit higher latency and lower throughput due to their reliance on matrix multiplication operations. To enhance LLM inference performance, you can use specialized hardware accelerators (for example, GPUs and TPUs) and optimized serving frameworks.

You can apply one or more of the following best practices to reduce LLM workload latency while improving throughput and cost-efficiency:

- Quantization (#quantization)
- Tensor parallelism (#tensor-parallelism)
- Model memory optimization (#model-memory)

The examples in this guide use the Gemma 7B LLM together with the vLLM or TGI serving frameworks to apply these best practices; however, the concepts and features described are applicable to most popular open LLMs.

Before you begin

Before you try the examples in this guide, complete these prerequisite tasks:

1. Follow the instructions in these guides to get access to the Gemma model, prepare your environment, and create and configure Google Cloud resources:
   - Serve Gemma open models using GPUs on GKE with vLLM (/kubernetes-engine/docs/tutorials/serve-gemma-gpu-vllm)
   - Serve Gemma open models using GPUs on GKE with Hugging Face TGI (/kubernetes-engine/docs/tutorials/serve-gemma-gpu-tgi)

   Make sure to save the Hugging Face access token to your Kubernetes secret.
2. Clone the https://github.com/GoogleCloudPlatform/kubernetes-engine-samples/ (https://github.com/GoogleCloudPlatform/kubernetes-engine-samples/) samples repository to your local development environment.

3. Change your working directory to `/kubernetes-engine-samples/ai-ml/llm-serving-gemma/`.

Best practice: Quantization

Quantization is a technique analogous to lossy image compression that reduces model size by representing weights in lower precision formats (8-bit or 4-bit), thus lowering memory requirements. However, like image compression, quantization involves a trade-off: decreased model size can lead to reduced accuracy.

Various quantization methods exist, each with its own unique advantages and disadvantages. Some, like AWQ and GPTQ, require pre-quantization and are available on platforms like Hugging Face (https://huggingface.co) or Kaggle (https://www.kaggle.com). For example, if you apply GPTQ on the Llama-2 13B model and AWQ on the Gemma 7B model, you can serve the models on a single L4 GPU instead of two L4 GPUs without quantization.

You can also perform quantization using tools like AutoAWQ (https://github.com/casper-hansen/AutoAWQ) and AutoGPTQ (https://github.com/AutoGPTQ/AutoGPTQ). These methods can improve latency and throughput. In contrast, techniques using EETQ (https://huggingface.co/docs/transformers/main/en/quantization/eetq) and the **`bitsandbytes`** (https://github.com/bitsandbytes-foundation/bitsandbytes) library for quantization don't require pre-quantized models so can be a suitable choice when pre-quantized versions aren't available.

The best quantization technique to use depends on your specific goals, and the technique's compatibility with the serving framework you want to use. To learn more, see the Quantization guide (https://huggingface.co/docs/optimum/en/concept_guides/quantization) from Hugging Face.

Select one of these tabs to see an example of applying quantization using the TGI or vLLM frameworks:

TGI (#tgi) vLLM (#vllm)

GKE supports these quantization options with TGI:

- `awq`
- `gptq`
- `eetq`
- `bitsandbytes`
- `bitsandbytes-nf4`
- `bitsandbytes-fp4`

AWQ and GPTQ quantization methods require pre-quantized models, while EETQ and `bitsandbytes` quantization can be applied to any model. To learn more about these options, see this Hugging Face article (https://huggingface.co/blog/4bit-transformers-bitsandbytes).

To use quantization, set the `--quantize` (https://huggingface.co/docs/text-generation-inference/en/basic_tutorials/launcher#quantize) parameter when starting the model server.

The following snippet shows how to optimize Gemma 7B with `bitsandbytes` quantization using TGI on GKE.

Note: Gemma 7B requires a minimum of two GPUs to run on GKE.

```
args:
- --model-id=$(MODEL_ID)
- --num-shard=2
- --quantize=bitsandbytes
```

To apply this configuration, use the following command:

```
kubectl apply -f tgi/tgi-7b-bitsandbytes.yaml
```

Best practice: Tensor parallelism

Tensor parallelism is a technique that distributes computational load across multiple GPUs, which is essential when you run large models that exceed single GPU memory capacity. This approach can be more cost-effective as it lets you use multiple affordable GPUs instead of a single expensive one. It can also enhance model inference throughput. Tensor parallelism leverages the fact that tensor operations can be performed independently on smaller data chunks.

To learn more about this technique, see the Tensor Parallelism guide (https://huggingface.co/docs/transformers/main/en/perf_train_gpu_many#tensor-parallelism) from Hugging Face.

Select one of these tabs to see an example of applying tensor parallelism using the TGI or vLLM frameworks:

TGIvLLM (#vllm)
(#tgi)

With TGI, the serving runtime will use all GPUs available to the Pod by default. You can set the number of GPUs to use by specifying the `--num-shard` parameter with the number of GPUs as the value.

See the Hugging Face documentation for the list of models (https://huggingface.co/docs/text-generation-inference/supported_models) supported for tensor parallelism.

The following snippet shows how to optimize the Gemma 7B instruction-tuned model using tensor parallelism and two L4 GPUs:

```
args:
- --model-id=$(MODEL_ID)
- --num-shard=2
```

To apply this configuration, use the following command:

```
kubectl apply -f tgi/tgi-7b-it-tensorparallelism.yaml
```

In GKE Autopilot clusters, running this command creates a Pod with minimum resource requirements (/kubernetes-engine/docs/concepts/autopilot-resource-requests#hardware-min-max) of 21 vCPU and 78 GiB memory.

Best practice: Model memory optimization

Optimizing the memory usage of LLMs is crucial for efficient inference. This section introduces attention layer optimization strategies, such as paged attention and flash attention. These strategies enhance memory efficiency, allowing for longer input sequences and reduced GPU idle time. This section also describes how you can adjust model input and output sizes to fit memory constraints and optimize for specific serving frameworks.

Attention layer optimization

Self-attention layers (https://sebastianraschka.com/blog/2023/self-attention-from-scratch.html) allow models to understand context in language processing tasks, as word meanings can shift depending on the context. However, these layers store input token weights, keys (K), and values (V) in GPU vRAM. Thus, as the input sequence lengthens, this leads to quadratic growth in size and computation time.

Using KV caching (https://huggingface.co/blog/kv-cache-quantization) is particularly useful when you're dealing with long input sequences, where the overhead of self-attention can become significant. This optimization approach reduces computational processing to linear complexity.

Specific techniques for optimizing attention mechanisms in LLMs include:

- **Paged attention**: Paged attention (https://docs.vllm.ai/en/latest/design/kernel/paged_attention.html) improves memory management for large models and long input sequences by using paging techniques, similar to OS virtual memory. This effectively reduces fragmentation and duplication in the KV cache, allowing for longer input sequences without running out of GPU memory.
- **Flash attention**: Flash attention (https://arxiv.org/abs/2205.14135) reduces GPU memory bottlenecks by minimizing data transfers between GPU RAM and L1 cache during token generation. This eliminates idle time for computing cores, significantly improving inference and training performance for GPUs.

Model input and output size tuning

Memory requirements depend on input and output size. Longer output and more context require more resources, while shorter output and less context can save costs by using a smaller, cheaper GPU.

Select one of these tabs to see an example of tuning the model input and output memory requirements in the TGI or vLLM frameworks:

TGIvLLM (#vllm)
(#tgi)

The TGI serving runtime checks memory requirements during startup and doesn't start if the maximum possible model memory footprint doesn't fit into the available GPU memory. This check eliminates out-of-memory (OOM) crashes on memory-intensive workloads.

GKE supports the following TGI parameters for optimizing model memory requirements:

- `MAX_CONCURRENT_REQUESTS` (https://huggingface.co/docs/text-generation-inference/en/basic_tutorials/launcher#maxconcurrentrequests)
- `MAX_TOTAL_TOKENS` (https://huggingface.co/docs/text-generation-inference/en/basic_tutorials/launcher#maxtotaltokens)
- `MAX_BATCH_PREFILL_TOKENS` (https://huggingface.co/docs/text-generation-inference/en/basic_tutorials/launcher#maxbatchprefilltokens)
- `MAX_INPUT_LENGTH` (https://huggingface.co/docs/text-generation-inference/en/basic_tutorials/launcher#maxinputlength)

The following snippet shows how you can serve a Gemma 7B instruction-tuned model with a single L4 GPU, with parameter settings `--max-total-tokens=3072, --max-batch-prefill-tokens=512, --max-input-length=512`:

```
args:
- --model-id=$(MODEL_ID)
- --num-shard=1
- --max-total-tokens=3072
- --max-batch-prefill-tokens=512
- --max-input-length=512
env:
- name: MODEL_ID
  value: google/gemma-7b
```

To apply this configuration, use the following command:

```
kubectl apply -f tgi/tgi-7b-token.yaml
```

Checklist summary

Optimization Goal	Practice
Latency	☐ **Prioritize powerful GPUs**: Consider upgrading to GPUs with higher computational capabilities and memory I/O throughput. ☐ **Explore quantization** (/kubernetes-engine/docs/best-practices/machine-learning/inference/llm-optimization#quantization): Techniques like AWQ can improve latency, but be mindful of potential accuracy trade-offs.
Throughput	☐ **Scale horizontally**: Increase the number of serving replicas (Pods) to distribute the workload. ☐ **Use tensor parallelism** (/kubernetes-engine/docs/best-practices/machine-learning/inference/llm-optimization#tensor-parallelism): For large models that exceed single GPU capacity, use tensor parallelism to spread computations across multiple GPUs. For smaller models, consider multiple replicas with tensor parallelism of `1` to avoid overhead. ☐ **Batch requests and quantize** (/kubernetes-engine/docs/best-practices/machine-learning/inference/llm-optimization#quantization): Combine requests and explore quantization techniques that maintain acceptable accuracy.
Cost-efficiency	☐ **Choose smaller models**: Select models within a family that fit your resource constraints and budget. ☐ **Apply quantization** (/kubernetes-engine/docs/best-practices/machine-learning/inference/llm-optimization#quantization): Use quantization to reduce memory requirements, particularly when dealing with larger models. ☐ **Limit context length** (/kubernetes-engine/docs/best-practices/machine-learning/inference/llm-optimization#model-io-adjustments): Constrain the context length to further decrease memory usage and enable execution on smaller, more cost-effective GPUs.

What's next

- For an end-to-end guide that covers container configuration, refer to Serve an LLM with multiple GPUs on GKE (/kubernetes-engine/docs/tutorials/serve-multiple-gpu).
- If you need a cloud-managed LLM serving solution, deploy your model through Vertex AI Model Garden (/vertex-ai/docs/start/explore-models).

Except as otherwise noted, the content of this page is licensed under the Creative Commons Attribution 4.0 License (https://creativecommons.org/licenses/by/4.0/), and code samples are licensed under the Apache 2.0 License (https://www.apache.org/licenses/LICENSE-2.0). For details, see the Google Developers Site Policies (https://developers.google.com/site-policies). Java is a registered trademark of Oracle and/or its affiliates.

Last updated 2025-12-02 UTC.