LLM inference optimization

Large language models (LLMs) have pushed text generation applications, such as chat and code completion models, to the next level by producing text that displays a high level of understanding and fluency. But what makes LLMs so powerful - namely their size - also presents challenges for inference.

Basic inference is slow because LLMs have to be called repeatedly to generate the next token. The input sequence increases as generation progresses, which takes longer and longer for the LLM to process. LLMs also have billions of parameters, making it a challenge to store and handle all those weights in memory.

This guide will show you how to use the optimization techniques available in Transformers to accelerate LLM inference.

> Hugging Face also provides Text Generation Inference (TGI), a library dedicated to deploying and serving highly optimized LLMs for inference. It includes deployment-oriented optimization features not included in Transformers, such as continuous batching for increasing throughput and tensor parallelism for multi-GPU inference.

Static kv-cache and torch.compile

During decoding, a LLM computes the key-value (kv) values for each input token and since it is autoregressive, it computes the same kv values each time because the generated output becomes part of the input now. This is not very efficient because you're recomputing the same kv values each time.

To optimize this, you can use a kv-cache to store the past keys and values instead of recomputing them each time. However, since the kv-cache grows with each generation step and is dynamic, it prevents you from taking advantage of `torch.compile`, a powerful optimization tool that fuses PyTorch code into fast and optimized kernels. We have an entire guide dedicated to kv-caches here.

The *static kv-cache* solves this issue by pre-allocating the kv-cache size to a maximum value which allows you to combine it with `torch.compile` for up to a 4x speed up. Your speed up may vary depending on the model size (larger models have a smaller speed up) and hardware.

> Currently, only Llama and a few other models support static kv-cache and `torch.compile`. Check this issue for a live model compatibility list.

There are three flavors of static kv-cache usage, depending on the complexity of your task:

1. Basic usage: simply set a flag in `generation_config` (recommended);
2. Advanced usage: handle a cache object for multi-turn generation or a custom generation loop;
3. Advanced usage: compile the entire `generate` function into a single graph, if having a single graph is relevant for you.

Select the correct tab below for further instructions on each of these flavors.

> Regardless of the strategy used with `torch.compile`, you can avoid shape-related recompilations if you left-pad your LLM inputs to a limited set of values. The `pad_to_multiple_of` tokenizer flag is your friend!

basic usage: generation_config | advanced usage: control Static Cache | advanced usage: end-to-end generate compilation

For this example, let's use the Gemma model. All we need to do is to:

1. Access the model's `generation_config` attribute and set the `cache_implementation` to "static";
2. Call `torch.compile` on the model to compile the forward pass with the static kv-cache.

And that's it!

```
from transformers import AutoTokenizer, AutoModelForCausalLM
import torch
import os
os.environ["TOKENIZERS_PARALLELISM"] = "false"  # To prevent long warnings :)

tokenizer = AutoTokenizer.from_pretrained("google/gemma-2b")
model = AutoModelForCausalLM.from_pretrained("google/gemma-2b", device_map="auto")

model.generation_config.cache_implementation = "static"

model.forward = torch.compile(model.forward, mode="reduce-overhead", fullgraph=True)
input_text = "The theory of special relativity states "
input_ids = tokenizer(input_text, return_tensors="pt").to("cuda")

outputs = model.generate(**input_ids)
print(tokenizer.batch_decode(outputs, skip_special_tokens=True))
['The theory of special relativity states 1. The speed of light is constant in all inertial reference']
```

Under the hood, `generate` will attempt to reuse the same cache object, removing the need for re-compilation at each call. Avoiding re-compilation is critical to get the most out of `torch.compile`, and you should be aware of the following:

1. If the batch size changes or the maximum output length increases between calls, the cache will have to be reinitialized, triggering a new compilation;
2. The first couple of calls of the compiled function are slower, as the function is being compiled.

> For a more advanced usage of the static cache, such as multi-turn conversations, we recommend instantiating and manipulating the cache object outside generate(). See the advanced usage tab.

Speculative decoding

> For a more in-depth explanation, take a look at the Assisted Generation: a new direction toward low-latency text generation blog post!

Another issue with autoregression is that for each input token you need to load the model weights each time during the forward pass. This is slow and cumbersome for LLMs which have billions of parameters. Speculative decoding alleviates this slowdown by using a second smaller and faster assistant model to generate candidate tokens that are verified by the larger LLM in a single forward pass. If the verified tokens are correct, the LLM essentially gets them for “free” without having to generate them itself. There is no degradation in accuracy because the verification forward pass ensures the same outputs are generated as if the LLM had generated them on its own.

To get the largest speed up, the assistant model should be a lot smaller than the LLM so that it can generate tokens quickly. The assistant and LLM model must also share the same tokenizer to avoid re-encoding and decoding tokens.

> Speculative decoding is only supported for the greedy search and sampling decoding strategies, and it also doesn’t support batched inputs.

Enable speculative decoding by loading an assistant model and passing it to the generate() method.

greedy search | sampling

```
from transformers import AutoModelForCausalLM, AutoTokenizer
import torch

device = "cuda" if torch.cuda.is_available() else "cpu"

tokenizer = AutoTokenizer.from_pretrained("facebook/opt-1.3b")
inputs = tokenizer("Einstein's theory of relativity states", return_tensors="pt").to(device)

model = AutoModelForCausalLM.from_pretrained("facebook/opt-1.3b").to(device)
assistant_model = AutoModelForCausalLM.from_pretrained("facebook/opt-125m").to(device)
outputs = model.generate(**inputs, assistant_model=assistant_model)
tokenizer.batch_decode(outputs, skip_special_tokens=True)
["Einstein's theory of relativity states that the speed of light is constant.    "]
```

Prompt lookup decoding

Prompt lookup decoding is a variant of speculative decoding that is also compatible with greedy search and sampling. Prompt lookup works especially well for input-grounded tasks - such as summarization - where there is often overlapping words between the prompt and output. These overlapping n-grams are used as the LLM candidate tokens.

To enable prompt lookup decoding, specify the number of tokens that should be overlapping in the `prompt_lookup_num_tokens` parameter. Then you can pass this parameter to the generate() method.

greedy decoding | sampling

```
from transformers import AutoModelForCausalLM, AutoTokenizer
import torch

device = "cuda" if torch.cuda.is_available() else "cpu"

tokenizer = AutoTokenizer.from_pretrained("facebook/opt-1.3b")
inputs = tokenizer("The second law of thermodynamics states", return_tensors="pt").to(device)

model = AutoModelForCausalLM.from_pretrained("facebook/opt-1.3b").to(device)
assistant_model = AutoModelForCausalLM.from_pretrained("facebook/opt-125m").to(device)
outputs = model.generate(**inputs, prompt_lookup_num_tokens=3)
print(tokenizer.batch_decode(outputs, skip_special_tokens=True))
['The second law of thermodynamics states that entropy increases with temperature.      ']
```

Attention optimizations

A known issue with transformer models is that the self-attention mechanism grows quadratically in compute and memory with the number of input tokens. This limitation is only magnified in LLMs which handles much longer sequences. To address this, try FlashAttention2 or PyTorch's scaled dot product attention (SDPA), which are more memory efficient attention implementations and can accelerate inference.

FlashAttention-2

FlashAttention and FlashAttention-2 break up the attention computation into smaller chunks and reduces the number of intermediate read/write operations to GPU memory to speed up inference. FlashAttention-2 improves on the original FlashAttention algorithm by also parallelizing over sequence length dimension and better partitioning work on the hardware to reduce synchronization and communication overhead.

To use FlashAttention-2, set `attn_implementation="flash_attention_2"` in the from_pretrained() method.

```
from transformers import AutoModelForCausalLM, BitsAndBytesConfig

quant_config = BitsAndBytesConfig(load_in_8bit=True)
model = AutoModelForCausalLM.from_pretrained(
    "google/gemma-2b",
    quantization_config=quant_config,
    torch_dtype=torch.bfloat16,
    attn_implementation="flash_attention_2",
)
```

PyTorch scaled dot product attention

Scaled dot product attention (SDPA) is automatically enabled in PyTorch 2.0 and it supports FlashAttention, xFormers, and PyTorch's C++ implementation. SDPA chooses the most performant attention algorithm if you're using a CUDA backend. For other backends, SDPA defaults to the PyTorch C++ implementation.

> SDPA supports FlashAttention-2 as long as you have the latest PyTorch version installed.

Use the torch.backends.cuda.sdp_kernel context manager to explicitly enable or disable any of the three attention algorithms. For example, set `enable_flash=True` to enable FlashAttention.

```
import torch
from transformers import AutoModelForCausalLM
```

```
model = AutoModelForCausalLM.from_pretrained(
    "google/gemma-2b",
    torch_dtype=torch.bfloat16,
)

with torch.backends.cuda.sdp_kernel(enable_flash=True, enable_math=False, enable_mem_efficient=False):
    outputs = model.generate(**inputs)
```

Quantization

Quantization reduces the size of the LLM weights by storing them in a lower precision. This translates to lower memory usage and makes loading LLMs for inference more accessible if you're constrained by your GPUs memory. If you aren't limited by your GPU, you don't necessarily need to quantize your model because it can incur a small latency cost (except for AWQ and fused AWQ modules) due to the extra step required to quantize and dequantize the weights.

> There are many quantization libraries (see the Quantization guide for more details) available, such as Quanto, AQLM, AWQ, and AutoGPTQ. Feel free to try them out and see which one works best for your use case. We also recommend reading the Overview of natively supported quantization schemes in 🤗 Transformers blog post which compares AutoGPTQ and bitsandbytes.

Use the Model Memory Calculator below to estimate and compare how much memory is required to load a model. For example, try estimating how much memory it costs to load Mistral-7B-v0.1.

To load Mistral-7B-v0.1 in half-precision, set the `torch_dtype` parameter in the from_pretrained() method to `torch.bfloat16`. This requires 13.74GB of memory.

```
from transformers import AutoTokenizer, AutoModelForCausalLM
import torch

model = AutoModelForCausalLM.from_pretrained(
    "mistralai/Mistral-7B-v0.1", torch_dtype=torch.bfloat16, device_map="auto",
)
```

To load a quantized model (8-bit or 4-bit) for inference, try bitsandbytes and set the `load_in_4bit` or `load_in_8bit` parameters to `True`. Loading the model in 8-bits only requires 6.87 GB of memory.

```
from transformers import AutoTokenizer, AutoModelForCausalLM, BitsAndBytesConfig
import torch

quant_config = BitsAndBytesConfig(load_in_8bit=True)
model = AutoModelForCausalLM.from_pretrained(
    "mistralai/Mistral-7B-v0.1", quantization_config=quant_config, device_map="auto"
)
```