Memory-Efficient Training of Large Language Models Using Liger Kernel and Other Techniques in 2024

- **Midterm Report** for the course *Artificial Intelligence* (2024 Fall)
- **Submitted on:** 2024/10/23
- **Author:** Yong-Cheng Liaw (312581029 廖永誠), Master's Student at National Yang Ming Chiao Tung University
- **Group:** 10

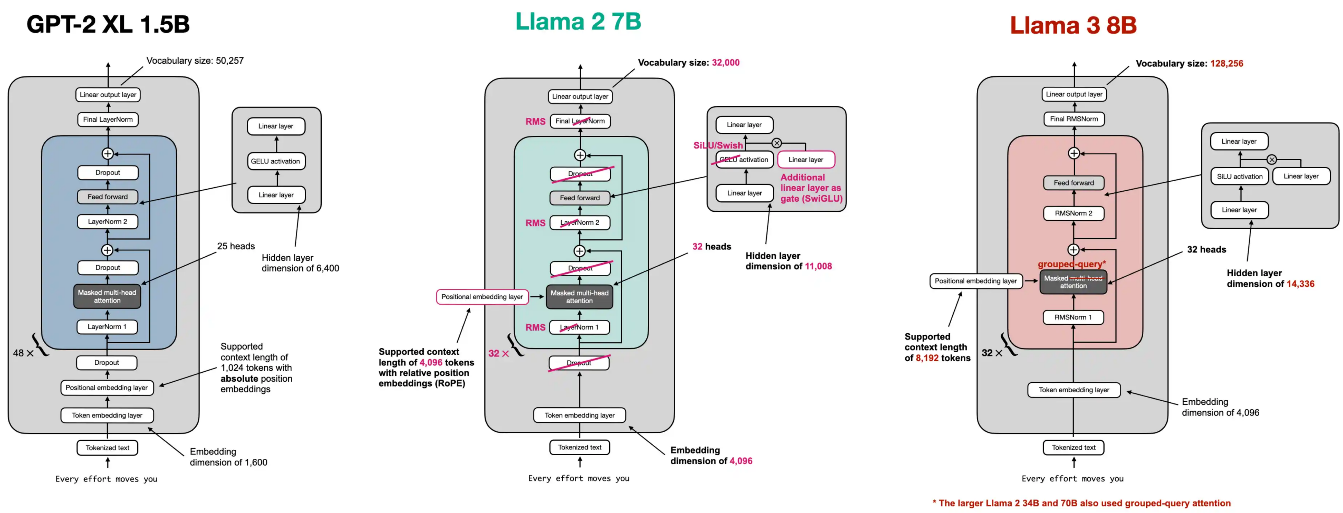
Title and Authors

- **Title:** Liger Kernel: Efficient Triton Kernels for LLM Training
- **Authors:** Pin-Lun Hsu, Yun Dai, Vignesh Kothapalli (LinkedIn Inc.)
- **Publication:** Technical Report published as a preprint on Arxiv (2024/10/14)
- **Code:** 3.3K Stars, Open-source Project on GitHub (2024/08/07)

Background and Motivation

Recent LLM Model Architecture

In recent Large Language Model Architecture, the model architecture has undergone some changes compared to the original transformer architecture. The original transformer architecture is a stack of encoder and decoder, with self-attention and cross-attention [Attention is all you need]. However, after the OpenAI GPT series, the state-of-the-art architecture became a decoder-only model [GPT1,2,3]. Nowadays, the state-of-the-art open-source model, Llama 3.2, also utilizes a dense decoder-only architecture, similar to GPT-2. However, they have some small differences compared to GPT-2.



- **Figure1: From GPT Architecture to Llama3 Architecture [https://github.com/rasbt/LLMs-from-scratch/blob/main/ch05/07_gpt_to_llama/converting-gpt-to-llama2.ipynb].**

The key differences are:

1. Convert absolute positional embedding into RoPE (Rotary Positional Embedding) [https://arxiv.org/abs/2104.09864].
2. Convert Layer Normalization into RMS Norm [RMSnorm].
3. Convert GELU into SwiGLU [SwiGLU].
4. Remove Dropout.
5. Convert multi-head attention into Group Query Attention (GQA) [GQA].

The GPU VRAM Usage During Training

Static Memory

When training Large Language Models (LLMs), a significant portion of GPU memory is consumed by static memory usage, which primarily consists of three main components: model weights, gradients, and optimizer states. Understanding these components is crucial for efficient memory management during the training process.

Model Weights

Model weights represent the parameters of the neural network and are a fundamental component of any LLM. The memory required for storing model weights is directly proportional to the number of parameters in the model. For instance, a GPT-2 model with 1.5 billion parameters requires approximately 3GB of memory for its weights when using 16-bit precision (2 bytes per parameter).

Gradients

Gradients are essential for updating the model weights during the training process. They typically require the same amount of memory as the model weights themselves. For a 1.5 billion parameter model, this would translate to about 3GB of memory for gradients.

Optimizer States

Optimizer states often consume the largest portion of static memory during training. The memory requirements for optimizer states like momentum, variance, master weights in popular optimizers like Adam can be significant. For instance, using mixed-precision training with Adam [?] or AdamW [?] can require up to 12 bytes per parameter just for optimizer states. Because it requires storing two additional tensors for each parameter: the momentum and variance of the gradients. When using mixed-precision training with Adam, the memory consumption includes:

1. An fp32 copy of the parameters (4 bytes per parameter)
2. Momentum (4 bytes per parameter)
3. Variance (4 bytes per parameter)

This results in a total of 12 bytes per parameter just for optimizer states.

Total Static Memory Usage

For a model with P parameters trained using mixed-precision Adam/AdamW, the total static memory usage can be approximated as:

```
Static Memory = (2P + 2P + 12P) bytes
              = 16P bytes
```

Where:

- 2P bytes for fp16 model weights
- 2P bytes for fp16 gradients
- 12P bytes for optimizer states (fp32 weights copy, momentum, and variance)

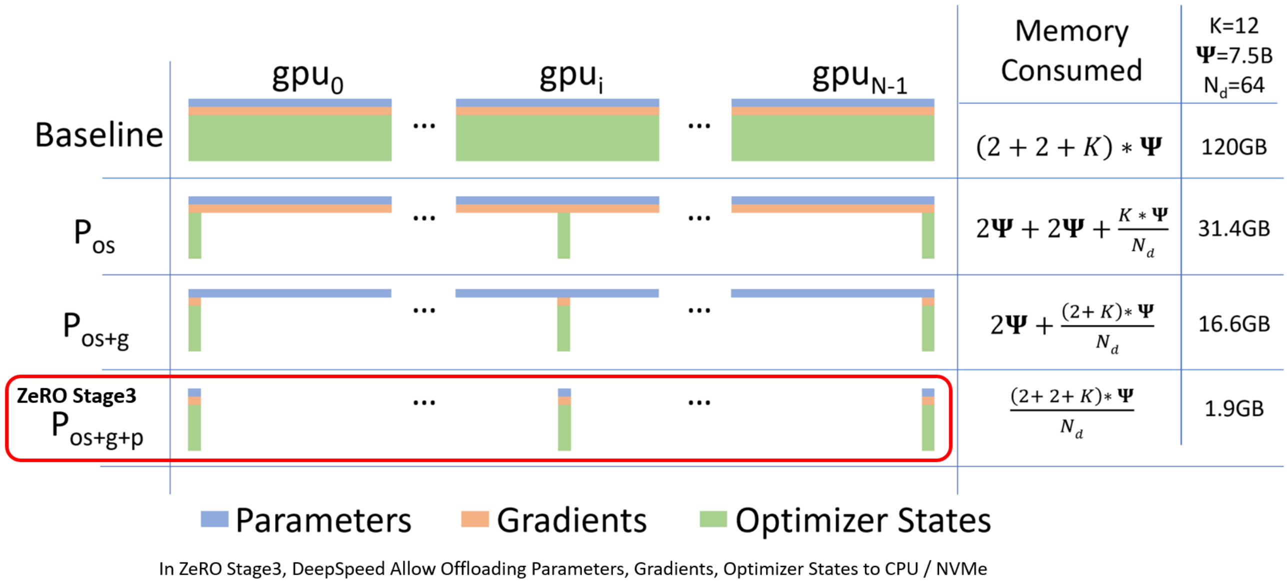
For example, a 1.5 billion parameter model would require at least 24GB of static memory during training.

Static Memory Optimization Techniques

To reduce static memory usage, several techniques can be employed:

1. **ZeRO**: This technique distributes optimizer states across multiple GPUs, reducing the memory required on each device. The popular implementation of ZeRO [zero paper] is available in DeepSpeed [GitHub] or PyTorch FSDP [FSDP doc or paper].

2. **DeepSpeed Offloading**: DeepSpeed allows offloading optimizer states to CPU memory or even SSD memory, freeing up GPU memory for model weights, gradients, and optimizer states [Zero Offload, Zero Infinity].



- Figure 2: ZeRO Memory Consumption [Zero Infinity paper]

3. **Mixed Precision Training**: Using lower precision (e.g., bfloat16, float16) for weights and gradients can significantly reduce memory consumption and improve training speed if supported by the hardware [NVIDIA Mixed Precision Training blog].

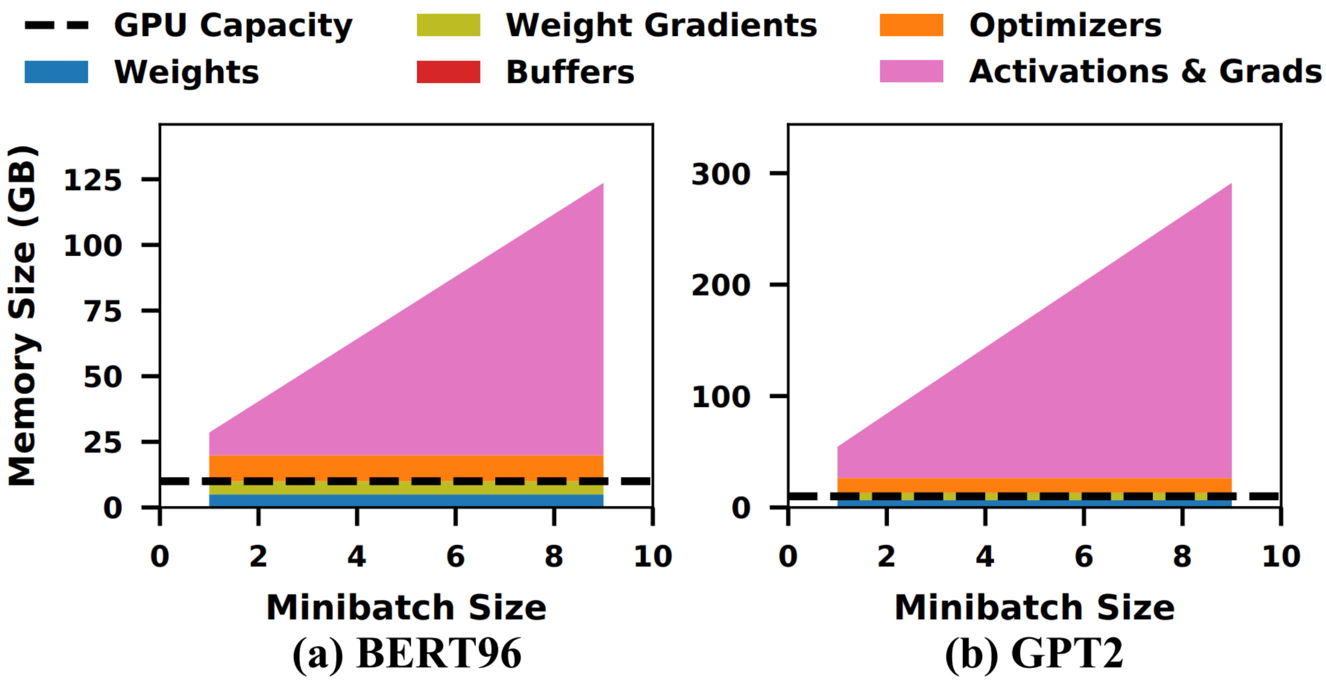
4. **8-bit Optimizers**: This focuses on quantizing optimizer states, such as momentum and variance, into 8-bit by utilizing block-wise quantization to prevent precision degradation [https://arxiv.org/abs/2110.02861].

5. **Gradient Low-Rank Projection (GaLore)**: This approach can reduce optimizer state memory usage by up to 65.5% while maintaining performance by utilizing Low-Rank Projection for optimizer states [https://arxiv.org/abs/2403.03507].

6. **LoRA and QLoRA**: LoRA is a technique that reduces static memory usage by decreasing the number of trainable weights. It projects the same weight matrix using two low-rank matrices, which reduces memory, gradients, and optimizer states. Since the frozen part of the model still requires complete model weights, QLoRA works by further quantizing that part to reduce the memory consumption of model weights [LoRA paper, QLoRA paper].

Activation Memory

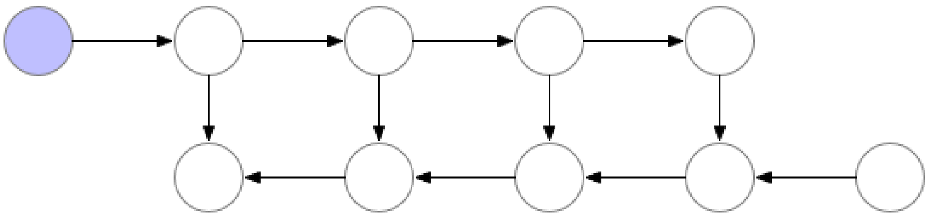
Activation memory refers to the memory used to store intermediate tensors (activations) generated during the forward pass of a neural network. These activations are essential for the backward pass, where gradients are computed for updating model parameters. Activation memory is dynamic and varies with the batch size, model architecture, and sequence length. It tends to dominate GPU memory usage during training because activations need to be stored until they are used in backpropagation.



- **Figure 3: Memory footprint statistics for training massive models [https://arxiv.org/abs/2202.01306].**

Gradient Checkpointing (Activation Checkpointing)

Gradient checkpointing is a technique that trades compute for memory by recomputing intermediate activations during the backward pass instead of storing them in memory. This can significantly reduce the memory footprint during training, especially for models with large memory requirements. However, gradient checkpointing can introduce additional computation overhead due to recomputing activations, which may impact training speed.



- **Figure 4: Example of Gradient Checkpointing [https://github.com/cybertronai/gradient-checkpointing].**

GPU VRAM Bottleneck

So after we have optimized the static memory like using DeepSpeed CPU Offloading to move all static memory from GPU to CPU, enable Gradient Checkpointing to discard all activation. The peak GPU memory usage caused by following two parts:

- The actual **checkpointing value** (which is the stored checkpoint).
- The highest temporary activation usage (occurring between two checkpoints or in sections that aren't checkpointed).
- In the Huggingface Transformers library, a checkpoint is added to the input of each decoder layer. Therefore, the checkpointing value only needs to store the following:
  - `batch * seq * hidden_size * dtype_size * num_layers` For example, using **LLaMA 3.2B** with `batch = 4`, `seq = 1024`, `hidden_size = 2048`, `dtype_size = 2` (fp16), and `num_layers = 16`, the checkpointing value is calculated as:
  - `4 * 1024 * 2048 * 2 * 16 = 268,435,456 bytes = 256MB`
- For the **temporary activation memory**, the largest factor comes from **cross-entropy-related activations**, which include logits, shifted logits, and intermediate values during the cross-entropy calculation.

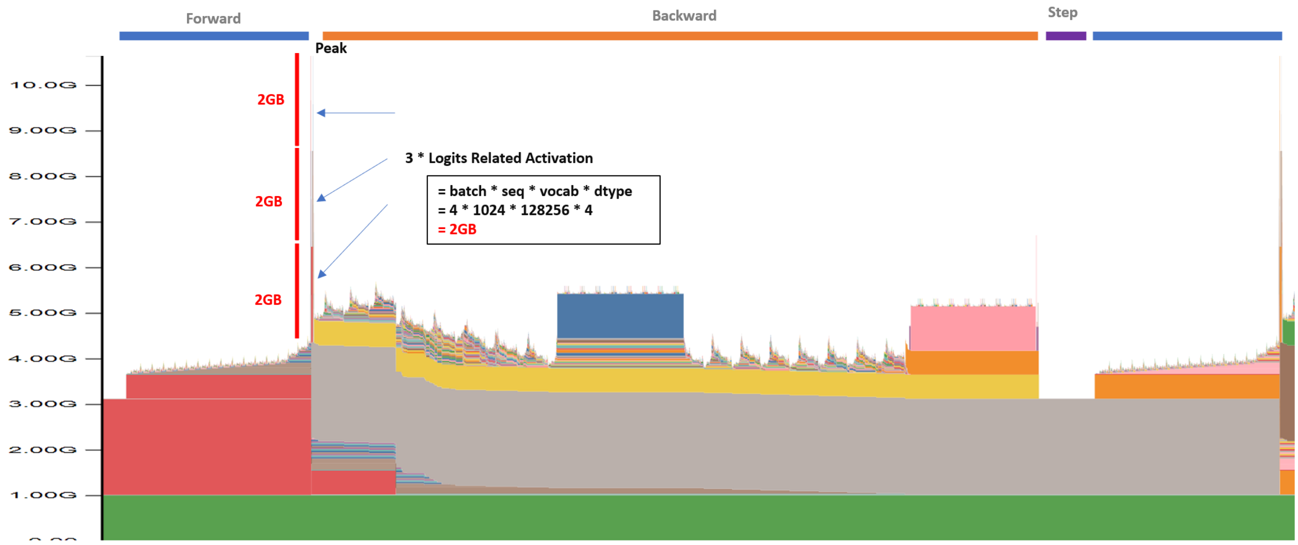
```
        hidden_states = outputs[0]
        if self.config.pretraining_tp > 1:
            lm_head_slices = self.lm_head.weight.split(self.vocab_size // self.config.pretraining_tp, dim=0)
            logits = [F.linear(hidden_states, lm_head_slices[i]) for i in range(self.config.pretraining_tp)]
            logits = torch.cat(logits, dim=-1)
        else:
            # Only compute necessary logits, and do not upcast them to float if we are not computing the loss
            logits = self.lm_head(hidden_states[:, -num_logits_to_keep:, :])
                           Shape: [batch * seq_length * vocab_size]
        loss = None
        if labels is not None:
            # Upcast to float if we need to compute the loss to avoid potential precision issues
            logits = logits.float()
            # Shift so that tokens < n predict n   Shape: [batch * (seq_length-1) * vocab_size]
            shift_logits = logits[..., :-1, :].contiguous()
            shift_labels = labels[..., 1:].contiguous()
            # Flatten the tokens
            loss_fct = CrossEntropyLoss()
            shift_logits = shift_logits.view(-1, self.config.vocab_size)
            shift_labels = shift_labels.view(-1)
            # Enable model parallelism
            shift_labels = shift_labels.to(shift_logits.device)
            loss = loss_fct(shift_logits, shift_labels)

        if not return_dict:
            output = (logits,) + outputs[1:]
            return (loss,) + output if loss is not None else output

        return CausalLMOutputWithPast(
            loss=loss,
            logits=logits,
            past_key_values=outputs.past_key_values,
            hidden_states=outputs.hidden_states,
            attentions=outputs.attentions,
        )
```

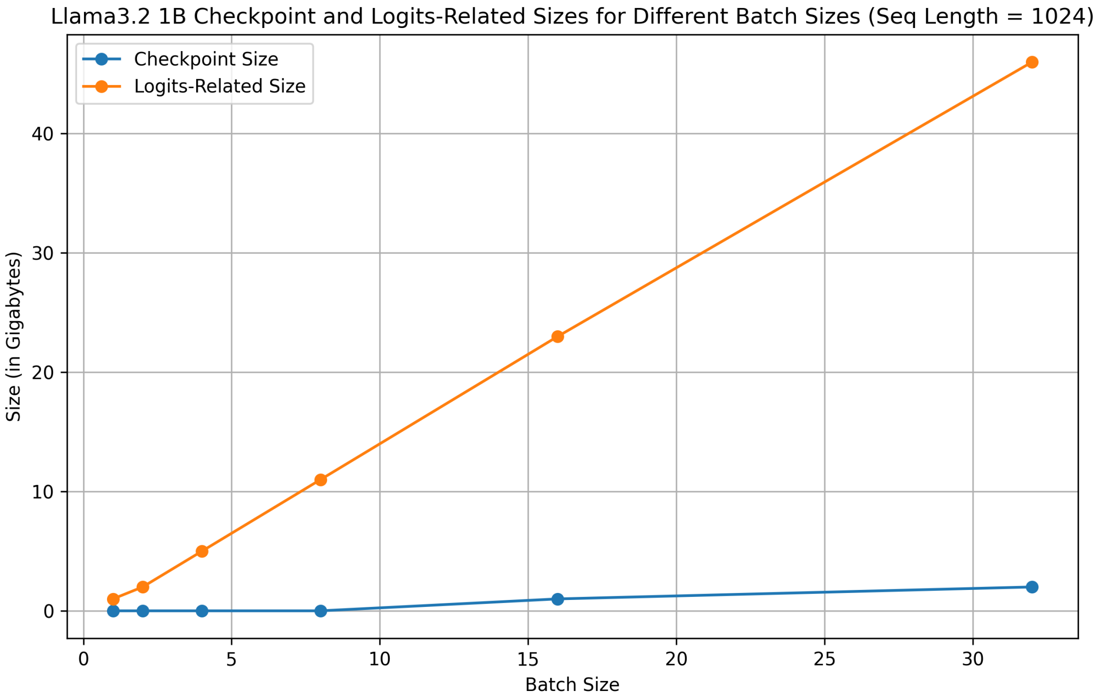
- **Figure 5: Llama3.2 Logits related Implementation of HuggingFace Transformers Library.**

- This is significant because the `vocab_size` is often much larger than the `hidden_size`. For example, in LLaMA 3.2 1B, `vocab_size = 128,256`, while `hidden_size = 2048`.
- Thus, a single logits value can occupy:
  - `4 * 1024 * 128,256 * 4 (cast to float for loss calculation) = 2,101,346,304 bytes ≈ 2GB`
- Moreover, there are multiple instances of these logits-related values. In our experiments, we observed that there are three logits-related values active simultaneously (logits, shift_logits, cross-entropy temp value). This results in: `2GB * 3 = 6GB` VRAM usage.



- **Figure 6: VRAM Snapshot for training Llama3.2 1B while enable gradient checkpoint + zero infinity cpu offloading with batch=4, seq=1024.**

- The factor affecting logits memory usage is primarily `batch_size` and `seq_size`. Therefore, increasing the batch size or sequence length leads to a rapid increase in peak memory usage.



- **Figure 7: Comparison of Estimated Scaling Memory Usage between Checkpoint and Logits-Related Size.**

- This peak memory usage limits the batch size that can be increased, even if there is space available to use.

Kernel Operation Level Optimization

In Current PyTorch, we can easily develop the model by executing the model calculation in eager execution model. However this is not the best way to fully utilize the GPU power. Because there are a lot of computational overheads, including function call stack, dispatching, and CUDA kernel launch latencies [https://pytorch.org/blog/accelerating-pytorch-with-cuda-graphs/, https://pytorch.org/blog/optimizing-production-pytorch-performance-with-graph-transformations/, https://developer.nvidia.com/blog/understanding-the-visualization-of-overhead-and-latency-in-nsight-systems/].

Dispatching Overhead

PyTorch uses a dynamic dispatching mechanism to determine which implementation of an operation to use based on input types and devices. This process introduces overhead in several ways:

1. Type checking: The system must check the types of input tensors for each operation.
2. Device selection: The appropriate implementation (CPU, CUDA, etc.) must be selected for each operation.
3. Operator lookup: The correct operator implementation must be found and called for each operation.

CUDA Kernel Launch Latencies

When using GPUs, eager mode can lead to significant overhead due to CUDA kernel launch latencies:

1. Frequent kernel launches: Each operation typically results in a separate CUDA kernel launch.
2. Launch overhead: There's a fixed overhead associated with each kernel launch, which can be significant for small operations.

Memory Transfer Overhead

Eager mode can result in suboptimal memory usage patterns:

1. Frequent host-to-device transfers: Input data may need to be transferred from CPU to GPU memory more frequently than necessary6.
2. Intermediate results: Each operation may write its results back to memory, increasing memory bandwidth usage4.

Kernel Fusion

Due to the above overhead, kernel fusion is a technique that combines multiple operations into a single kernel to reduce overhead and improve performance. By fusing operations together and optimizing the kernel implementation, we can reduce the number of kernel launches, memory transfers, and dispatching overheads. A popular to do kernel fusion is writing GPU kernel operation in Triton [https://openai.com/index/triton/].

Triton is a language and compiler for parallel programming. It aims to provide a Python-based programming environment for productively writing custom DNN compute kernels capable of running at maximal throughput on modern GPU hardware.

```
BLOCK = 512
@jit
def add(X, Y, Z, N):
   # In Triton, each kernel instance
   # executes block operations on a
   # single thread: there is no construct
   # analogous to threadIdx
   pid = program_id(0)
   # block of indices
   idx = pid * BLOCK + arange(BLOCK)
   mask = idx < N
   # Triton uses pointer arithmetics
   # rather than indexing operators
   x = load(X + idx, mask=mask)
   y = load(Y + idx, mask=mask)
   store(Z + idx, x + y, mask=mask)
```

The Liger Kernel is using Triton to implement the fused operation in the kernel level. The reason to use Triton is as follows:

1. Easier programming: Compare to write the kernel in c++ CUDA, Triton can write the kernel in python, which is easier to develop and debug.
2. Kernel Level Optimization: Compare to the eager execution model, Triton can optimize at the kernel operation level, which can do more detailed optimization.
3. Python native: No need to maintain different file type of code, like c++ and python.
4. Clean dependency: Triton is a standalone library, which can be easily integrated into the existing codebase.

5. There already a lot of success kernel operation level project done by Triton, like flashattention, unsloth [flashattention, unsloth].

Method

Fused Linear Cross Entropy (Main Method)

1. explain the cross entropy

2. explain method

3. disadvantage (if chunk is too small, calculate chunk by chunk will decrease gpu utilization)

4. advantage

    - reduce a lot of memory usage
    - reduce some computation overhead
    - implement in kernel level, so it can reduce a lot of overhead

Other Method (Only fused some calculation)

RMS Norm

1. explain the original forward formula
2. show the original implementation in pytorch
3. explain the fused part
4. which part can increase the performance (reduce overhead)
5. which part can reduce memory usage

RoPE

1. explain the original forward formula
2. show the original implementation in pytorch
3. explain the fused part
4. which part can increase the performance (reduce overhead)
5. which part can reduce memory usage

SwiGLU

1. explain the original forward formula
2. show the original implementation in pytorch
3. explain the fused part
4. which part can increase the performance (reduce overhead)
5. which part can reduce memory usage

Liger kernel framework Use Case

Experimental Results

Execution Time

Peak Memory

Real Use Case Benchmark

DEMO

1. mine experiments environment

    1. llama3.2 1B
    2. 24 GB gpu (4090)
    3. no lora -> full parameter training
    4. fp16 mixed precision training
    5. deepspeed cpu offloading for weights and optimizer states
    6. gradient checkpointing

2. show the memory usage by batch size

3. show the throught scale by batch size

4. show the ablation study on different method

    1. compare throughput at different batch size
    2. compare memory usage at different batch size

Conclusion and Personal Reflection

The main problem of the current LLM training

The main contribution of the Liger kernel -> fused kernel + chunking

How to train the large language model efficiently in 2024

Reference

1. Liger kernel paper
2. liger kernel talk: https://www.youtube.com/watch?v=gWble4FreV4
3. pytorch profiler and tensorboard plugin
4. deepspeed zero, fsdp
5. deepspeed offloading
6. gradient checkpointing
7. lora, qlora
8. 8bit optimizer
9. triton
10. sympy for forward and automatically backward
11. flashattention
12. unsloth.ai
13. nvidia mixed precision training