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

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)

Group 10
智能所 312581029 廖永誠

Outline

1. Background and Motivation
2. Method
3. Experimental Results
4. Conclusion
5. Personal Reflection
6. Reference

Background and Motivation

1. Recent LLM Model Architecture
2. The GPU VRAM Usage During Training
3. Kernel Operation Level Optimization

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 [1]. However, after the OpenAI GPT series, the state-of-the-art architecture became a decoder-only model [2,3,4].
- 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.

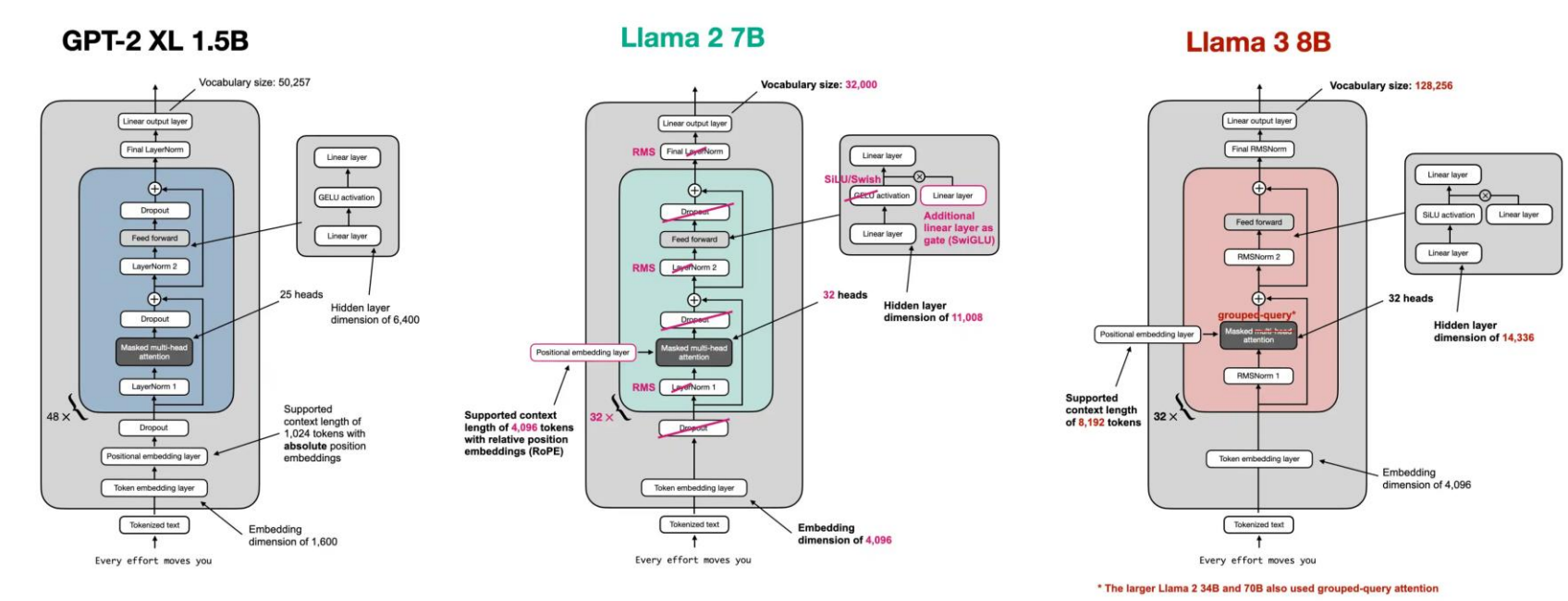

Figure1: From GPT Architecture to Llama3 Architecture [5].

Background and Motivation

1. Recent LLM Model Architecture
2. The GPU VRAM Usage During Training
3. Kernel Operation Level Optimization

The GPU VRAM Usage During Training

1. Static Memory
2. Activation Memory

Static Memory

1. **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).
2. **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.
3. **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 [11] or AdamW [12] 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:
    - fp32 copy of the parameters (4 bytes per parameter)
    - Momentum (4 bytes per parameter)
    - Variance (4 bytes per parameter)

- **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:
    1. Static Memory = (2P + 2P + 12P) bytes = 16P bytes
- For example, a 1.5 billion parameter model would require at least 24GB of static memory during training.

Static Memory Optimization Techniques

1. **ZeRO:** This technique distributes optimizer states across multiple GPUs, reducing the memory required on each device. The popular implementation of ZeRO [13] is available in DeepSpeed [14] or PyTorch FSDP [15].
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 [16,17].
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 [18].
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 [19].
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 [20].
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 [21,22].

The GPU VRAM Usage During Training

1. Static Memory
2. Activation Memory

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.

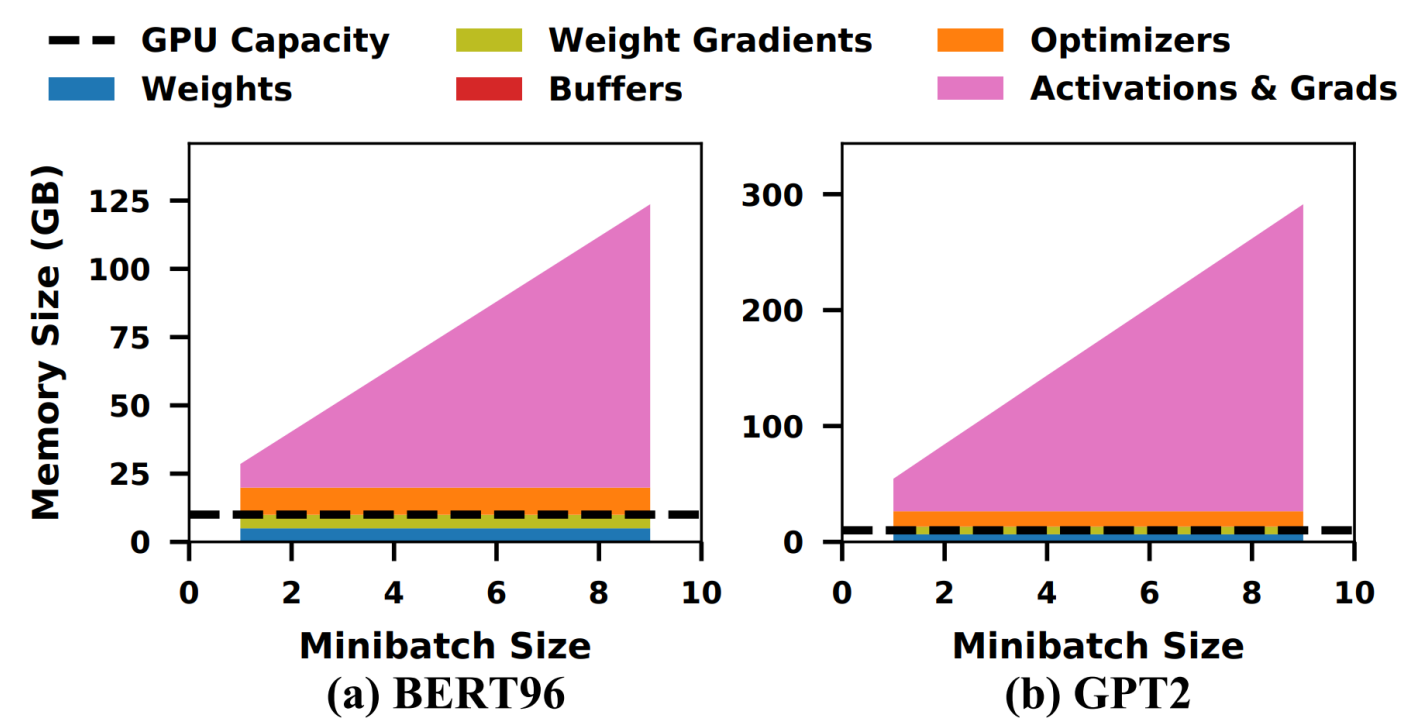

Figure2: Memory footprint statistics for training massive models [10].

Gradient (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 [23]

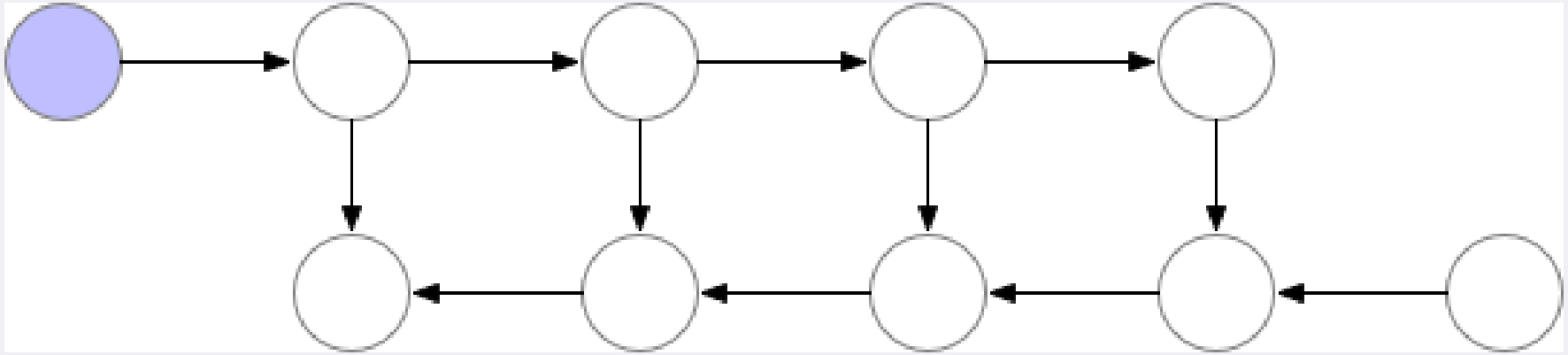

Figure3: Example of Gradient Checkpointing [23].

GPU VRAM Bottleneck

- 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:
  1. **The actual checkpointing value (which is the stored checkpoint).**
  2. **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 3.2Bwith 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

GPU VRAM Bottleneck (cont'd)

- 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.
- 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

```
1200            hidden_states = outputs[0]
1201            if self.config.pretraining_tp > 1:
1202                lm_head_slices = self.lm_head.weight.split(self.vocab_size // self.config.pretraining_tp, dim=0)
1203                logits = [F.linear(hidden_states, lm_head_slices[i]) for i in range(self.config.pretraining_tp)]
1204                logits = torch.cat(logits, dim=-1)
1205            else:
1206                # Only compute necessary logits, and do not upcast them to float if we are not computing the loss
1207                logits = self.lm_head(hidden_states[:, -num_logits_to_keep:, :])
1208                                Shape: [batch * seq_length * vocab_size]
1209            loss = None
1210            if labels is not None:
1211                # Upcast to float if we need to compute the loss to avoid potential precision issues
1212                logits = logits.float()
1213                # Shift so that tokens < n predict n  Shape: [batch * (seq_length-1) * vocab_size]
1214                shift_logits = logits[..., :-1, :].contiguous()
1215                shift_labels = labels[..., 1:].contiguous()
1216                # Flatten the tokens
1217                loss_fct = CrossEntropyLoss()
1218                shift_logits = shift_logits.view(-1, self.config.vocab_size)
1219                shift_labels = shift_labels.view(-1)
1220                # Enable model parallelism
1221                shift_labels = shift_labels.to(shift_logits.device)
1222                loss = loss_fct(shift_logits, shift_labels)
1223
1224            if not return_dict:
1225                output = (logits,) + outputs[1:]
1226                return (loss,) + output if loss is not None else output
1227
1228            return CausalLMOutputWithPast(
1229                loss=loss,
1230                logits=logits,
1231                past_key_values=outputs.past_key_values,
1232                hidden_states=outputs.hidden_states,
1233                attentions=outputs.attentions,
1234            )
```

Figure4: Llama3.2 Logits related Implementation of HuggingFace Transformers Library.

GPU VRAM Bottleneck (cont'd)

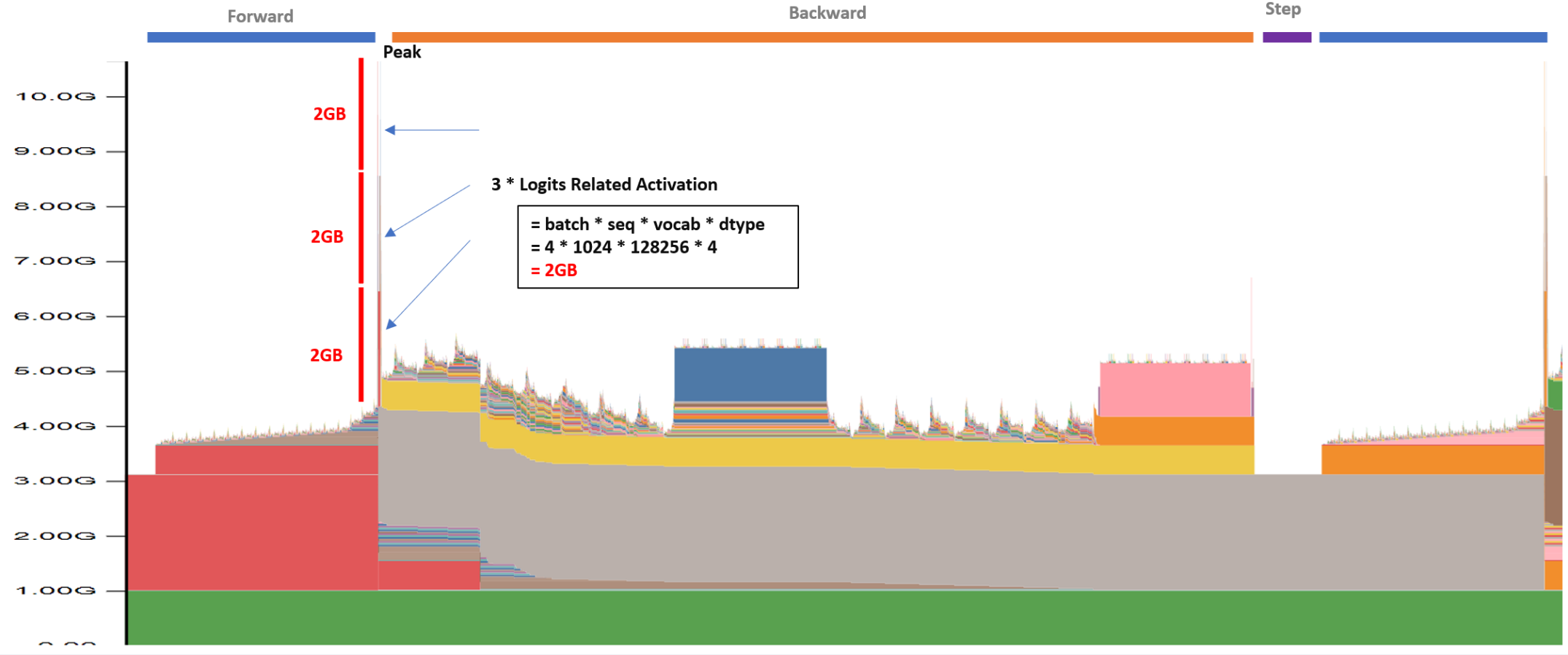

Figure5: VRAM Snapshot for training Llama3.2 1B while enable gradient checkpoint + zero infinity cpu offloading with batch=4, seq=1024.

GPU VRAM Bottleneck (cont'd)

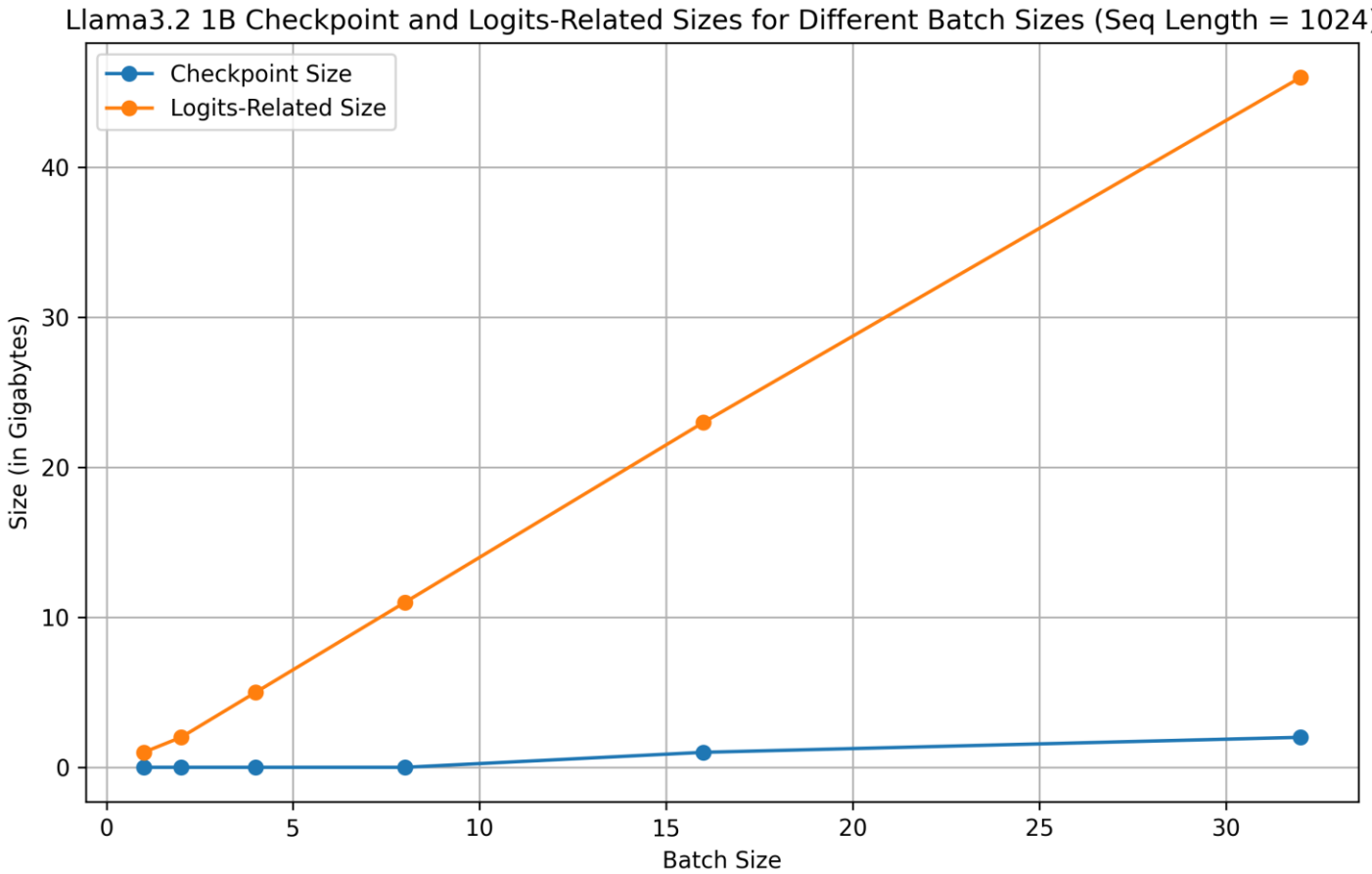

Figure6: Comparison of Estimated Scaling Memory Usage between Checkpoint and Logits-Related Size.

Background and Motivation

1. Recent LLM Model Architecture
2. The GPU VRAM Usage During Training
3. Kernel Operation Level Optimization

Kernel Operation Level Optimization

- In current PyTorch, we can easily develop models by executing calculations in the eager execution model. However, this is not the best way to fully utilize GPU power because there are many computational overheads, including function call stack, dispatching, and CUDA kernel launch latencies [24,25,26].

1. **Dispatching Overhead**
    - Type checking: The system must check the types of input tensors for each operation.
    - Device selection: The appropriate implementation (CPU, CUDA, etc.) must be selected for each operation.
    - Operator lookup: The correct operator implementation must be found and called for each operation.
2. **CUDA Kernel Launch Latencies**
    - Frequent kernel launches: Each operation typically results in a separate CUDA kernel launch.
    - Launch overhead: There's a fixed overhead associated with each kernel launch, which can be significant for small operations.
3. **Memory Transfer Overhead**
    - Frequent host-to-device transfers: Input data may need to be transferred from CPU to GPU memory more frequently than necessary.
    - Intermediate results: Each operation may write its results back to memory, increasing memory bandwidth usage.

Kernel Fusion

- Due to the above overheads, 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 approach to perform kernel fusion is writing GPU kernel operations in Triton [27].
- 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 arithmetic
   # rather than indexing operators
   x = load(X + idx, mask=mask)
   y = load(Y + idx, mask=mask)
   store(Z + idx, x + y, mask=mask)
```

Why Triton ?

- The Liger Kernel uses Triton to implement fused operations at the kernel level. The reasons for using Triton are as follows:

1. **Easier programming:** Compared to writing the kernel in C++ CUDA, Triton allows writing kernels in Python, making development and debugging easier.
2. **Kernel Level Optimization:** Compared to the eager execution model, Triton can optimize at the kernel operation level, enabling more detailed optimizations.
3. **Python-native:** There's no need to maintain different types of code files, like C++ and Python.
4. **Clean dependency:** Triton is a standalone library that can be easily integrated into existing codebases.
5. **Real Production Ready Usecase:** There are already several successful kernel operation-level projects done using Triton, such as FlashAttention and Unsloth [28,29].

Method

1. Fused Linear Cross Entropy
2. Other Fused Kernels

Fused Linear Cross Entropy

1. **Split the cross-entropy calculation into chunks to reduce peak memory usage.**

$$\mathcal{L}_{\text{cross-entropy}} = -\frac{1}{N}\frac{1}{T-1}\sum_{i=1}^{N}\sum_{t=1}^{T-1}\log p(y_t|x_{1:t})$$

- ***N*** is the batch size
- ***T*** is the sequence length
- ***y_t*** is the ground truth token at time step t+1
- ***p(y_t | x_{1:t})*** is the predicted probability of token ***y_t*** given the tokens ***x_{1:t}***

Fused Linear Cross Entropy (cont'd)

2. **Calculate the backward gradient when calculating the forward pass, utilizing in-place calculation to reduce memory usage.**

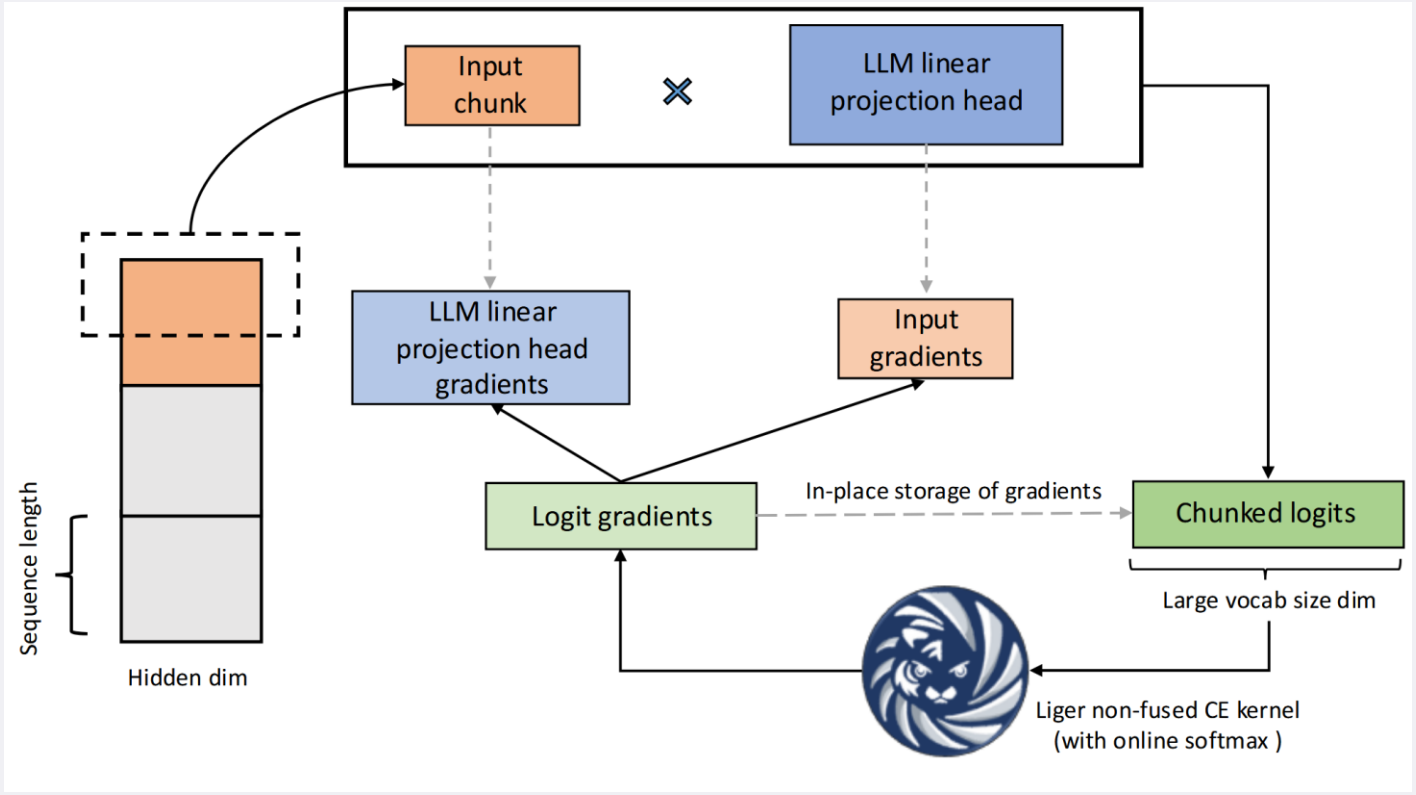

Figure7: Fused Linear Cross Entropy.

The Advantage of Fused Linear Cross Entropy

- **Advantage:** memory usage is -> batch_size * seq_size * vocab_size to chunk_size * vocab_size.
- **Disadvantage:** If the chunk size is too small, the calculation will be done chunk by chunk, which will decrease GPU utilization.
- Therefore, setting the correct chunk size is crucial for the performance of the Fused Linear Cross Entropy.
- Try to set the chunk size can make temporary activation memory is the same as the hidden_size activation.

$$2^{\lceil \log_2 \lceil \frac{BT}{\lceil \frac{V}{H} \rceil} \rceil \rceil}$$

- since hidden activation is BT * H, and chunk activation is chunk_size * V
- chunk_size = $\frac{BT}{V/H}$

Method

1. Fused Linear Cross Entropy
2. Other Fused Kernels

Other Fused Kernels

- Following the same principle as the Fused Linear Cross Entropy, the Liger Kernel also provides other fused kernels to optimize the training process. These fused kernels include:
    1. **RMSNorm (Root Mean Square Normalization) [7]**
    2. **RoPE (Rotary Positional Embedding) [6]**
    3. **SwiGLU [8]**
    4. **GeGLU [8]**
    5. **Layer Normalization [30]**
- These fused kernels aim to fuse small operations together rather than executing them one by one.
- For example, in the case of RMSNorm, the formula is:

$$x_{\text{norm}} = \frac{x}{\text{RMS}}, \text{RMS} = \sqrt{\frac{1}{N}\sum_{i=1}^{N} x_i^2}$$

- The calculation includes both the normalization and scaling parts, which can be fused together. Additionally, because it is a fused operation, some calculations can be done in place to reduce memory usage, and values like the RMS can be cached to reduce computation overhead.

Experiments Results

- The Liger Kernel runs all benchmarks on a single NVIDIA A100 GPU (80 GB).
- The CrossEntropy kernel is benchmarked on vocab sizes in the set
  - {40960, 81920, 122880, 163840}.
- The GeGLU and SwiGLU kernels are benchmarked on varying sequence lengths.
- The RMSNorm, LayerNorm, and RoPE kernels are benchmarked on varying hidden dimensions.
- The sequence lengths and hidden dimension sizes are chosen from
  - {4096, 8192, 12288, 16384}.

Speed Benchmark

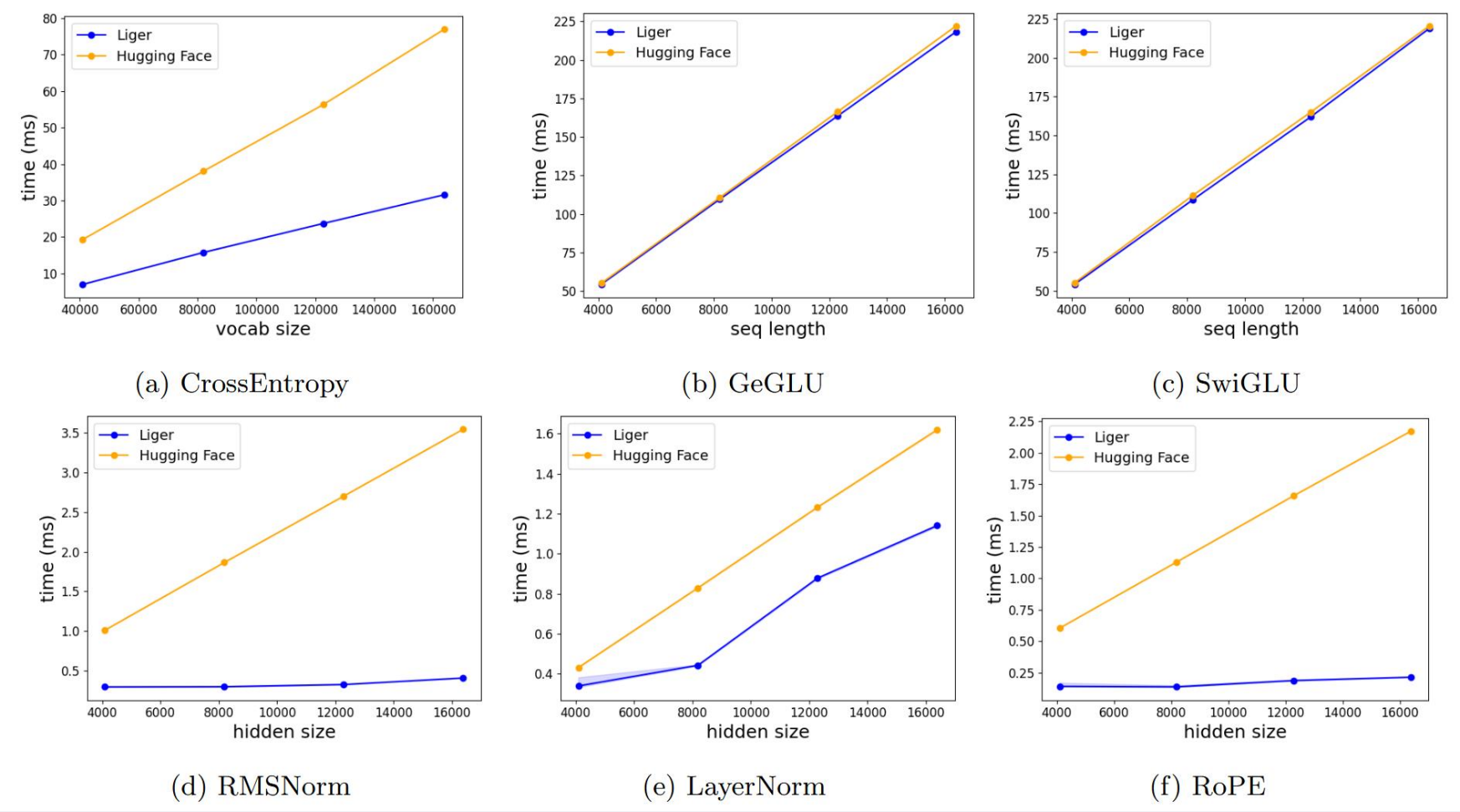

Figure8: Kernel execution speed benchmarks

Peak Memory Benchmark

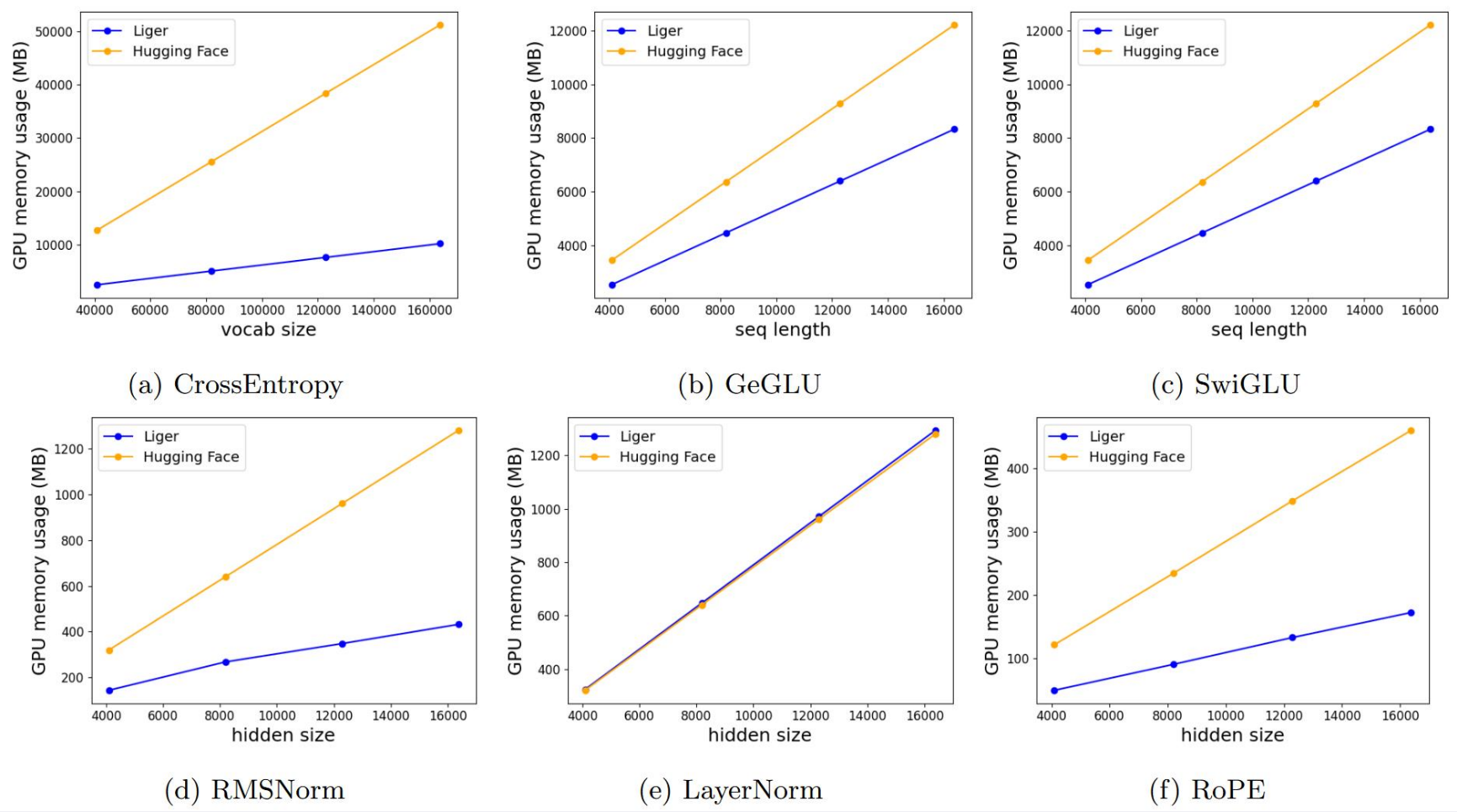

Figure9: Kernel peak allocated memory benchmarks.

Llama3 8B Training with Fixed seq_length 512

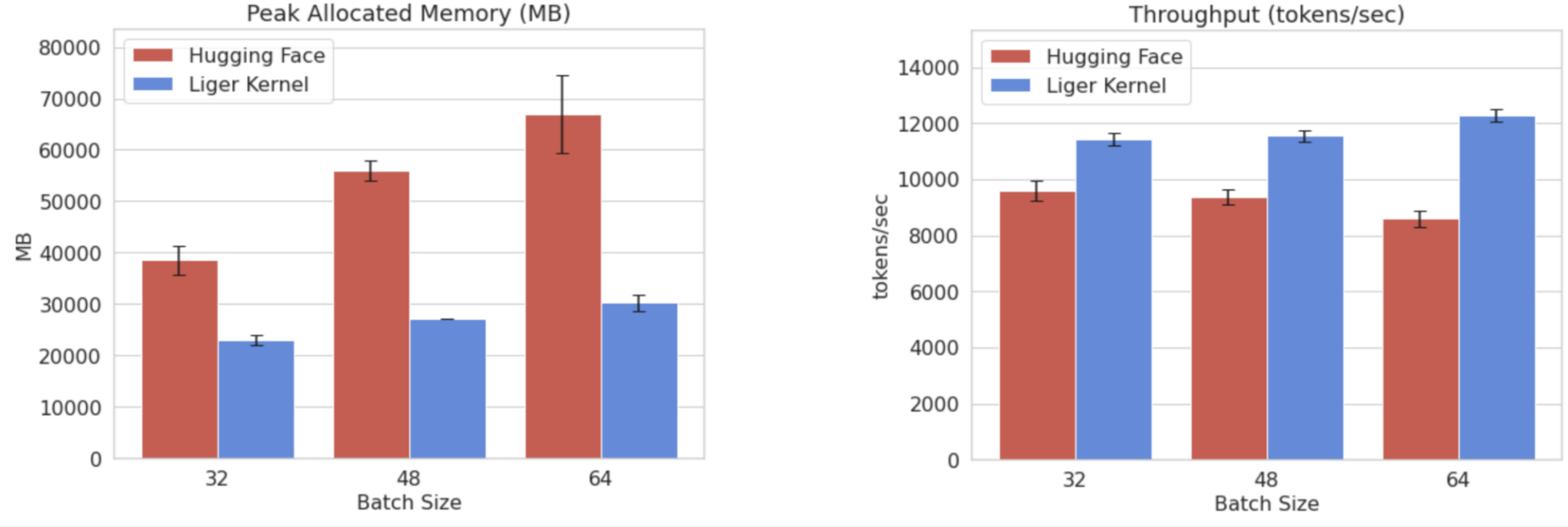

Figure10: Comparison of peak allocated memory and throughput for Llama3 8B.

Bonus: DEMO

- Reproduce the results and profile the actual memory usage reduction and throughput improvement by the Liger Kernel
- All demo code is available on the following GitHub Repository **113-1-Artificial-Intelligence-Midterm**.
- **Experiment Environment:**
    1. Operating System: Ubuntu 24.04 Desktop
    2. CPU: i9-13900K
    3. DRAM: 128GB
    4. GPU: NVIDIA 4090 24GB
- **Experiment Setup:**
    1. Batch Size: 1, 4, 8, 16, 32, 64, 96, 112
    2. Sequence Length: 1024
    3. Model: Llama3.2 1B (Huggingface Transformers)
    4. Full Parameter Training: No LoRA
    5. Enable mixed precision training (fp16)
    6. Enable Gradient Checkpointing
    7. Enable DeepSpeed CPU Weights and Optimizer States Offloading

Memory Usage Profiling

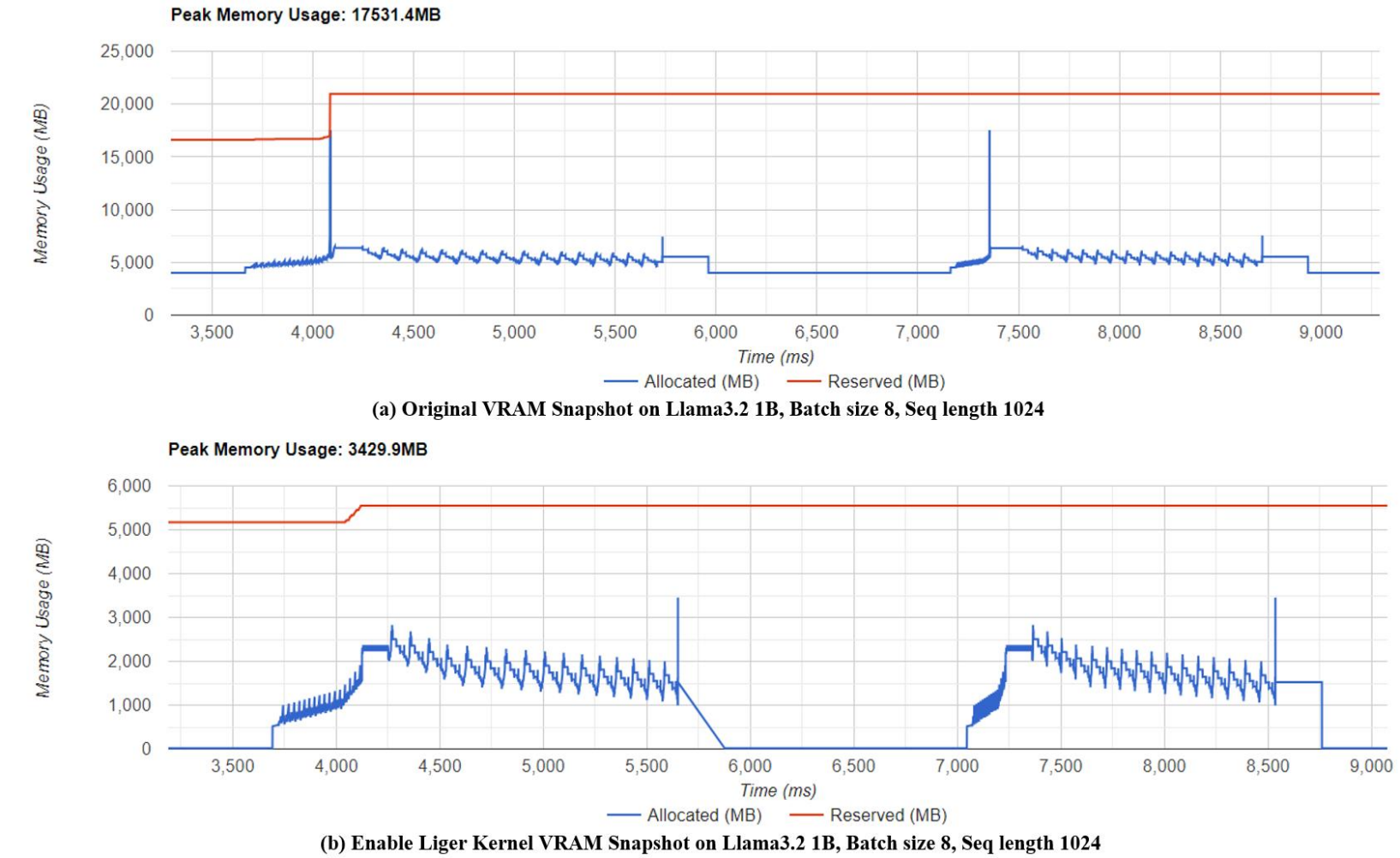

Figure11: VRAM Snapshot of Llama3.2 1B, Batch Size 8, Seq Length 1024 with (a) HuggingFace Transformers (b) Liger Kernel enabled.

Comparison of Throughput and Peak Memory Usage

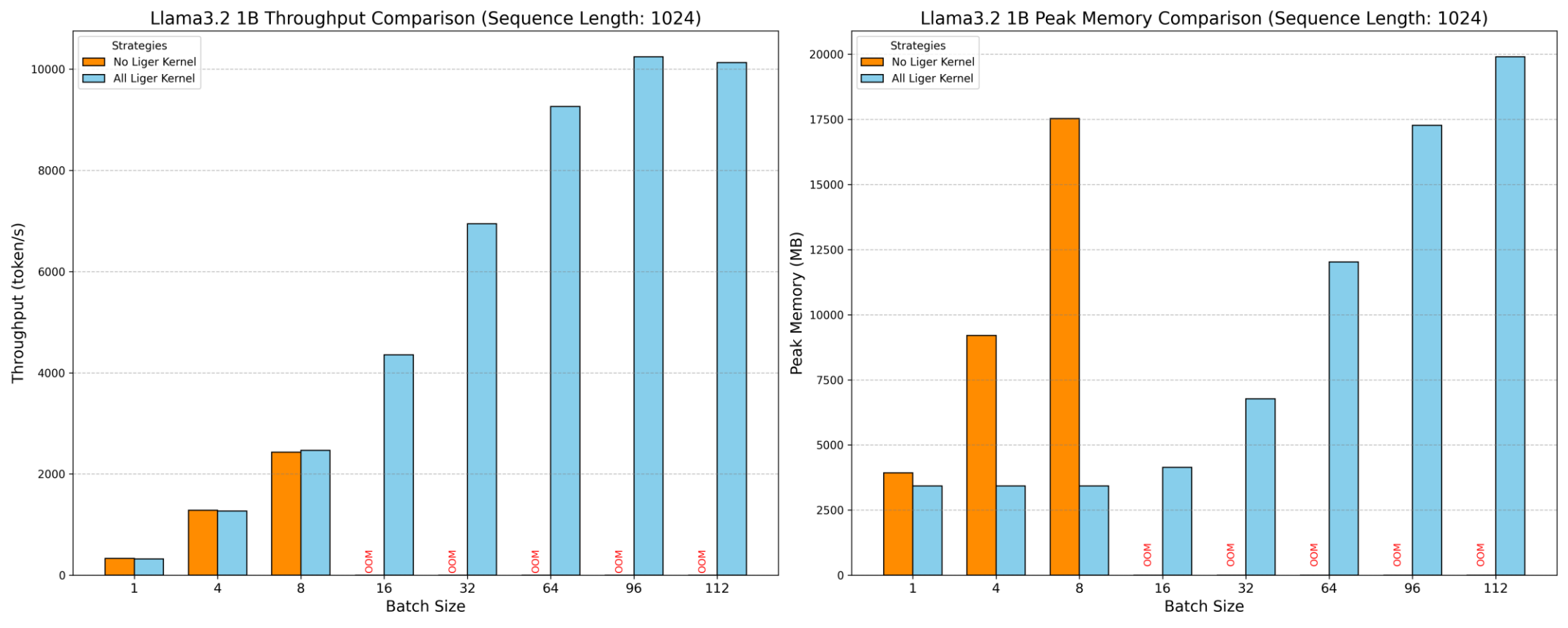

Figure12: Comparison of Throughput and Peak Memory Usage between Liger Kernel and Huggingface Transformers.

Ablation Study on Fused Linear Cross Entropy

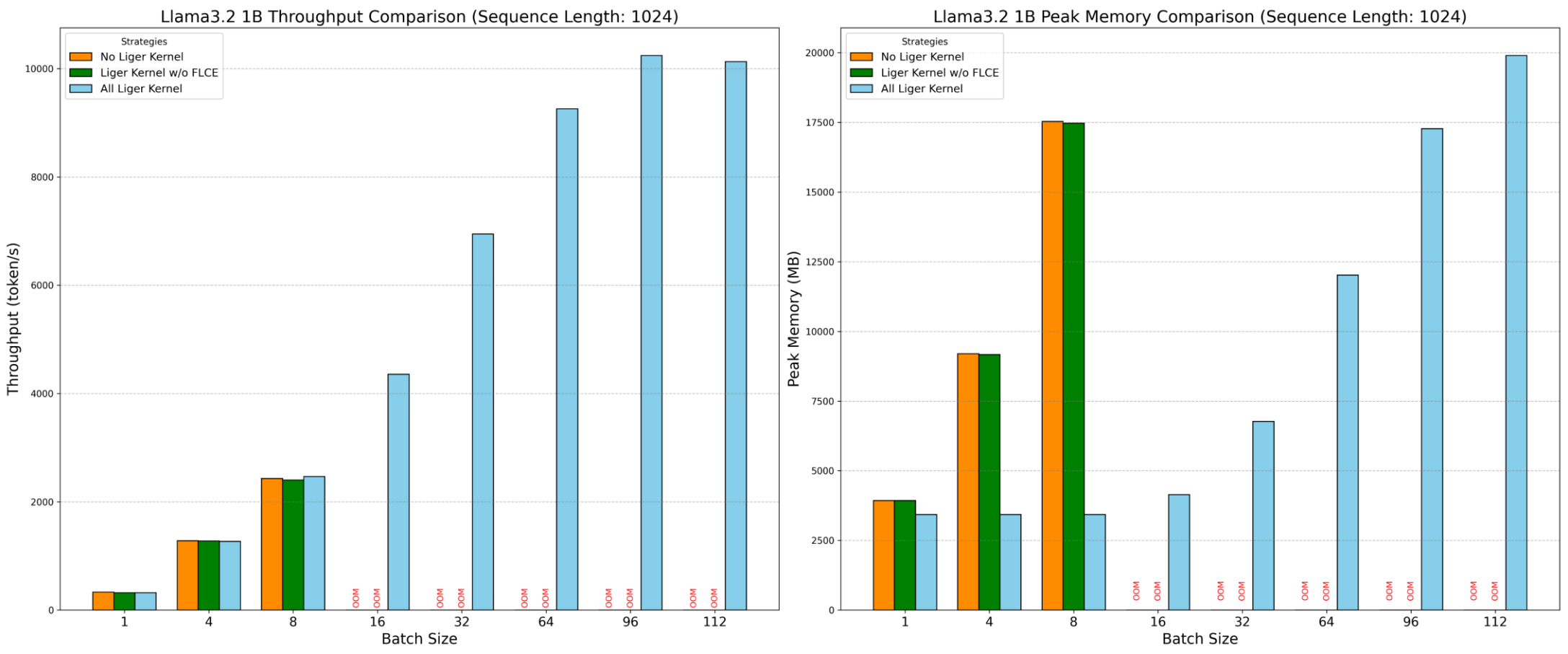

Figure13: Ablation Study on Fused Linear Cross Entropy.

Conclusion

- The main problem of the current LLM training
  - significant peak memory consumption associated with the cross-entropy loss computation
  - This memory bottleneck restricts the scalability of training by limiting the permissible batch size and sequence length
- The main contribution of the Liger kernel
  1. **Fused Kernel Operations:** By combining multiple operations into a single kernel, the Liger Kernel reduces the overhead associated with kernel launches, memory transfers, and dispatching. This fusion not only optimizes the execution speed but also minimizes the peak memory usage by eliminating intermediate storage of activation tensors.
  2. **Chunking Strategy:** The Liger Kernel employs a chunking mechanism to divide the cross-entropy calculation into smaller, manageable segments. This approach ensures that the peak memory usage is proportional to the chunk size rather than the entire batch size or sequence length. By dynamically adjusting the chunk size based on the model's hidden size and vocabulary size, the Liger Kernel maintains a stable memory footprint even as the batch size and sequence length scale up. This chunking strategy effectively decouples memory usage from model scaling, enabling more flexible and efficient training of large-scale language models.

Personal Reflect

- How to train the large language model efficiently in 2024 ?
- There are many memory-efficient training techniques for training large language models nowadays, thanks to the Huggingface Transformers library developers and the open-source community. These techniques (including the Liger kernel) are all integrated into the Huggingface Transformers library, making them easy to use and deploy. The following is my recommended approach to efficiently train a large language model with full parameter training (not considering LoRA) in 2024:
- **Essential settings to enable no matter what:**
    1. Enable Liger kernel
    2. (If multiple GPUs) Enable Stage 2 ZeRO
    3. Enable gradient checkpointing (slightly increases computation overhead)
- **If still facing Out of Memory issues:**
    4. Enable mixed precision training (slightly reduces precision)
    5. (If multiple GPUs) Enable Stage 3 ZeRO (with additional communication overhead)
    6. Enable DeepSpeed CPU offloading (increases PCIe transfer overhead)
    7. Enable 8-bit optimizer or GaLore (reduces precision)
    8. Enable DeepSpeed NVMe offloading (significantly increases PCIe transfer overhead)

Reference

1. Attention Is All You Need, https://doi.org/10.48550/arXiv.1706.03762.
2. Improving Language Understanding by Generative Pre-Training, https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf
3. Language Models are Unsupervised Multitask Learners, https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf
4. Language Models are Few-Shot Learners, https://arxiv.org/abs/2005.14165
5. Converting GPT to Llama2, https://github.com/rasbt/LLMs-from-scratch/blob/main/ch05/07_gpt_to_llama/converting-gpt-to-llama2.ipynb
6. RoFormer: Enhanced Transformer with Rotary Position Embedding, https://arxiv.org/abs/2104.09864
7. Root Mean Square Layer Normalization, https://arxiv.org/abs/1910.07467
8. GLU Variants Improve Transformer, https://arxiv.org/abs/2002.05202
9. GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints, https://arxiv.org/abs/2305.13245
10. Harmony: Overcoming the Hurdles of GPU Memory Capacity to Train Massive DNN Models on Commodity Servers, https://arxiv.org/abs/2202.01306
11. Adam: A Method for Stochastic Optimization, https://arxiv.org/abs/1412.6980
12. Decoupled Weight Decay Regularization, https://arxiv.org/abs/1711.05101
13. ZeRO: Memory Optimizations Toward Training Trillion Parameter Models, https://arxiv.org/abs/1910.02054
14. DeepSpeed, https://github.com/microsoft/DeepSpeed
15. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel, https://arxiv.org/abs/2304.11277
16. ZeRO-Offload: Democratizing Billion-Scale Model Training, https://arxiv.org/abs/2101.06840
17. ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning, https://arxiv.org/abs/2104.07857
18. Mixed-Precision Training of Deep Neural Networks, https://developer.nvidia.com/blog/mixed-precision-training-deep-neural-networks/

Reference (cont'd)

19. 8-bit Optimizers via Block-wise Quantization, https://arxiv.org/abs/2110.02861

20. GaLore: Memory-Efficient LLM Training by Gradient Low-Rank Projection, https://arxiv.org/abs/2403.03507

21. LoRA: Low-Rank Adaptation of Large Language Models, https://arxiv.org/abs/2106.09685

22. QLoRA: Efficient Finetuning of Quantized LLMs, https://arxiv.org/abs/2305.14314

23. gradient-checkpointing, https://github.com/cybertronai/gradient-checkpointing

24. Accelerating PyTorch with CUDA Graphs, https://pytorch.org/blog/accelerating-pytorch-with-cuda-graphs/

25. Optimizing Production PyTorch Models' Performance with Graph Transformations, https://pytorch.org/blog/optimizing-production-pytorch-performance-with-graph-transformations/

26. Understanding the Visualization of Overhead and Latency in NVIDIA Nsight Systems, https://developer.nvidia.com/blog/understanding-the-visualization-of-overhead-and-latency-in-nsight-systems/

27. Introducing Triton: Open-source GPU programming for neural networks, https://openai.com/index/triton/

28. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness, https://arxiv.org/abs/2205.14135

29. Unsloth, https://unsloth.ai/

30. Layer Normalization, https://arxiv.org/abs/1607.06450

31. Liger Kernel: Efficient Triton Kernels for LLM Training, https://arxiv.org/abs/2410.10989

32. GPU MODE Lecture 28: Liger Kernel - Efficient Triton Kernels for LLM Training, https://www.youtube.com/watch?v=gWble4FreV4

33. PyTorch Profiler With TensorBoard, https://pytorch.org/tutorials/intermediate/tensorboard_profiler_tutorial.html

Q & A

END

Thanks for listening