COAT: Compressing Optimizer states and Activation for Memory-Efficient FP8 Training

Haocheng Xi, Han Cai, Ligeng Zhu, Yao Lu, Kurt Keutzer, Jianfei Chen, Song Han

Motivation: High Memory Usage

- Model Memory (per-parameter)
  - Optimizer States: FP32 & FP32
    - 8 bits per parameter
    - First-order momentum & Second-order momentum
  - Master Copy Weight: FP32
    - 4 bits per parameter
  - Gradients: FP32
    - 4 bits per parameter
  - Total: 16 bit per parameter
- Activations: BF16
  - Proportional to Batch Size (BS) and Sequence Length (SL)
  - Dominate when optimizer/gradient/parameters are sharded across GPUs

Motivation: High Memory Usage

- Train Llama-2-7B on 4 * H100, using PyTorch FSDP
  - Batch Size=2, Sequence Length=2048

Optimizer States	Weights	Gradients	Activations	Total	Peak
13.1 GB	6.5 GB	6.5 GB	25.8 GB	52.0 GB	55.1 GB

Memory Usage Per GPU

- Train Llama-2-13B on 8 * H100, using PyTorch FSDP
  - Batch Size=1, Sequence Length=2048

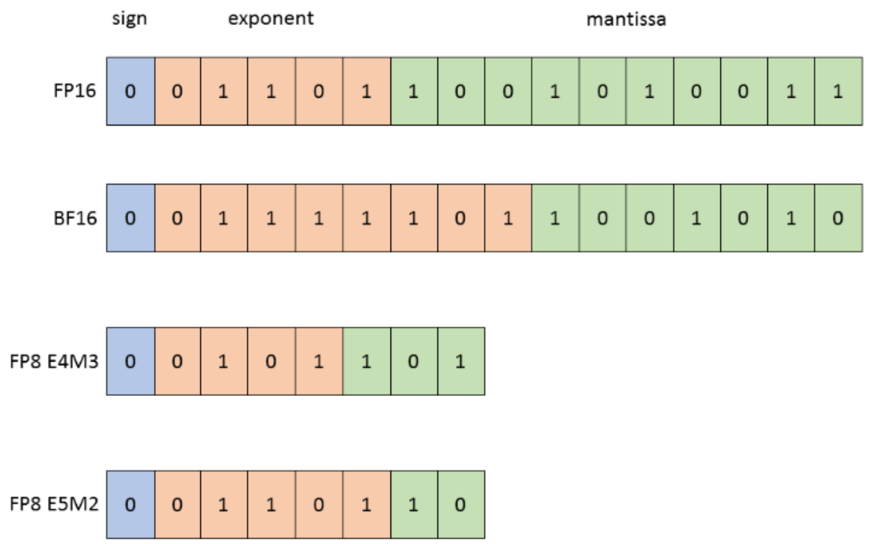
Optimizer States	Weights	Gradients	Activations	Total	Peak
12.6 GB	6.3 GB	6.3 GB	20.1 GB	45.3 GB	49.4 GB

Memory Usage Per GPU

Optimizer states and **Activations** lead to the largest memory usage
Limit the scale-up of sequence length & model size

Solution: Quantize Optimizer states and Activations to FP8

- FP8 format – E4M3 and E5M2
  - TransformerEngine uses FP8 to accelerate, but not being memory-efficient
- Optimizer states: 4x memory efficient compared with FP32
- Activations: 2x memory efficient compared with BF16



Ideally, we can achieve roughly 1.75x memory reduction ratio when optimizer states and activations are all quantized to FP8

Optimizer	Activation	Total	Ratio
12.6 GB	20.1 GB	45.3 GB	1.00x
3.2 GB	10.0 GB	25.8 GB	1.75x

https://github.com/NVIDIA/TransformerEngine

Part 1: Preliminaries of Optimizer States Quantization

- FP8 Quantization

$$X_{\text{FP8}}, S_X = Q(X_{\text{FP32}}), \text{ where } X_{\text{FP8}} = \left\lceil \frac{X_{\text{FP32}}}{S_X} \right\rfloor, \ S_X = \frac{\max\left(|X_{\text{FP32}}|\right)}{\Delta_{\max}^{\text{E4M3}}}$$

$\Delta_{\max}^{\text{E4M3}} = 448$ is the largest representable value

- Optimizer Step
    - AdamW optimizer has first-order momentum m and second-order momentum v
    - At time step t:
        - Update the optimizer states

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_{t-1} \qquad v_t = \beta_2 v_{t-1} + (1-\beta_2) g_{t-1}^2$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \qquad \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

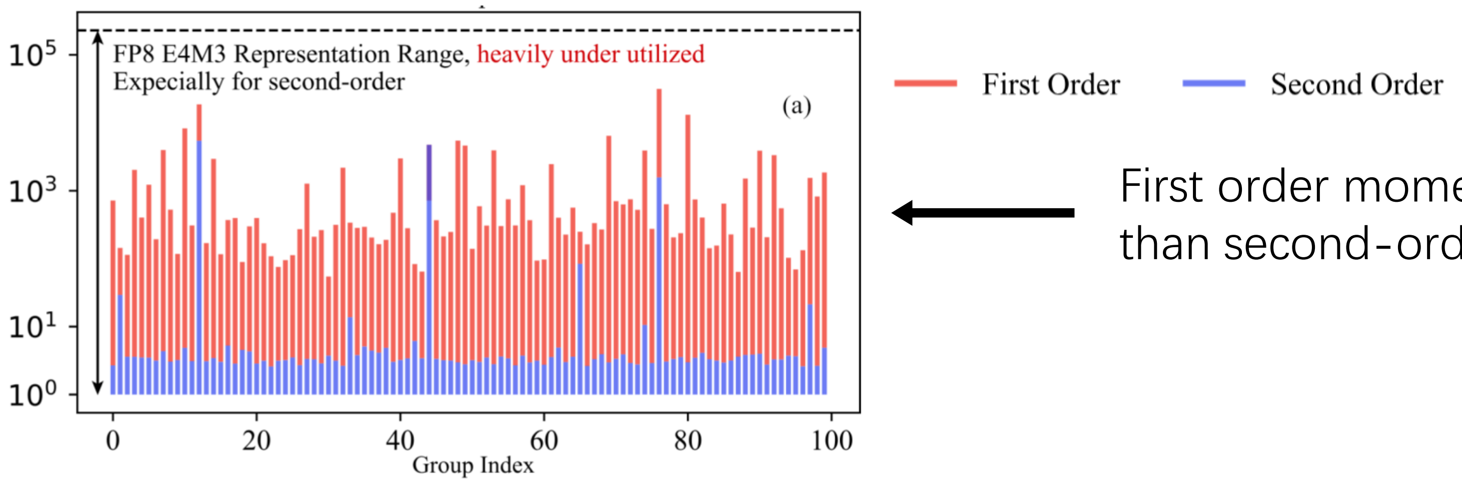
        - Update the weights

$$w_{t+1} = w_t - \eta \left(\frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} + \lambda w_t \right)$$

η is learning rate, λ is weight decay

Part 1: Difficulty of Optimizer States Quantization

- We apply **per-group quantization**
  - Every G elements are quantized independently (G=128)
- We Observe that FP8's representation range is **under-utilized**
  - E4M3's minimum value = 2^{-9}, maximum value = 448 ⟶ Dynamic range $\approx 2 \times 10^5$
  - However, most groups' dynamic range is small
    - **Dynamic Range** $\mathcal{R}_X$ is defined as $\frac{\max\_value}{\min\_value}$ in a quantization group
    - Can not utilize this range well, therefore leads to large quantization error



First order momentum's dynamic range is larger than second-order momentum's dynamic range

Part 1: Methodology —— Dynamic Range Expansion

- We hope to fully utilize the FP8 representation range

- Introduce an **expand function** to expand the dynamic range before quantization
  - The function is $f(x) = \mathrm{sign}(x)|x|^k$.
  - We do expansion before quantization: $X_{\mathrm{FP8}}, S_X = Q(f(X_{\mathrm{FP32}}))$

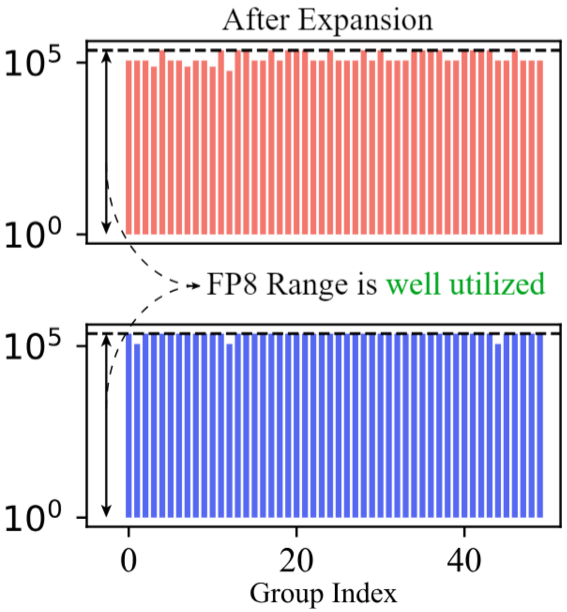
- Dynamic range will be expanded if $k > 1$

$$\mathcal{R}_{f(X)} = \frac{\max(|f(X)|)}{\min(|f(X)|)} = \frac{\max(|\mathrm{sign}(X)X^k|)}{\min(|\mathrm{sign}(X)X^k|)} = \left(\frac{\max(|X|)}{\min(|X|)}\right)^k = (\mathcal{R}_X)^k.$$

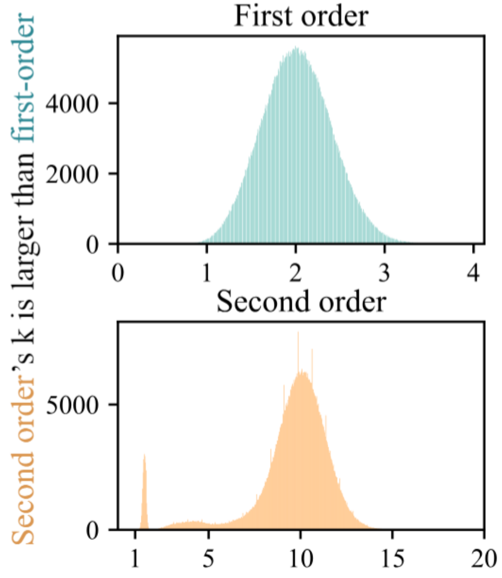
- Optimal k should fully utilize the range, and satisfy that $(\mathcal{R}_X)^k = \mathcal{R}_{\mathrm{E4M3}}$
  - Can be directly calculated as $k = \log_{\mathcal{R}_X}(\mathcal{R}_{\mathrm{E4M3}})$
  - Computed *on-the-fly* in every optimizer.step() for *every quantization group*

Part 1: Result of Dynamic Range Expansion

- After Expansion, every quantization group can **fully utilized** the FP8' representation range



- Second Order momentum has a larger k than First Order momentum



Part 1: Result of Dynamic Range Expansion

- Quantization error can be greatly reduced
- The actual effective term when updating weight is $\frac{m}{\sqrt{v}+\varepsilon}$, so we report the quantization error of this term

$$w_{t+1} = w_t - \eta\left(\frac{\hat{m}_t}{\sqrt{\hat{v}_t}+\epsilon} + \lambda w_t\right)$$

Table 1: Quantization error of $\frac{m}{\sqrt{v}}$ under different quantization settings. +Expand means applying our Dynamic Range Expansion method.

MSE of $\frac{m}{\sqrt{v}}$	Second Order			
First Order	**E4M3**	**E4M3+Expand**	**E5M2**	**E5M2+Expand**
E4M3	**20.10**	18.08	**25.65**	18.16
E4M3+Expand	15.13	**12.31**	21.96	**12.43**
E5M2	**37.02**	35.96	**40.30**	36.00
E5M2+Expand	17.79	15.48	23.84	15.57

- The distribution after Dynamic Range Expansion can **better utilize** FP8 E4M3's representation range

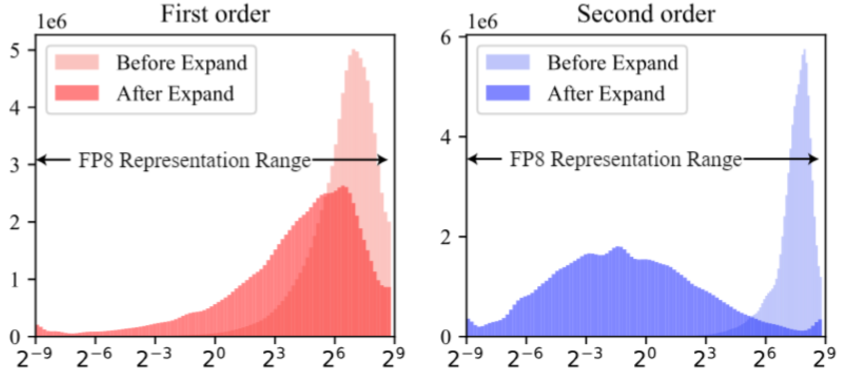


Figure 3: Dynamic Range Expansion can better utilize E4M3 representation range.

Part 2: Activation Memory Decomposition

- Need to save activation to calculate the gradient in backward pass
  - Consumes much memory
- Non-linear layer contributes to >50% memory usage

		Non-Linear		Attention				Reduction Ratio	
		RMSNorm	**Act Func**	**RoPE**	**FlashAttn**	**Linear**	**Total**	**Ideal**	**Achieved**
Llama-style	**BF16**	4U	8U	2U	3U	5.66U	22.66U	-	
	TE	4U	8U	2U	3U	3.33U	20.33U	1.11×	1.09×
	COAT	1U	4U	2U	3U	3.33U	13.33U	1.69×	1.65×

1U = Batch Size × Sequence Length × Hidden Size × 2 bytes (for BF16)

- Important to quantize **both linear and non-linear layers**' input to save memory

Part 2: Mixed Granularity Activation Quantization

- Uses **FP8 precision flow** to reduce activation memory
  - Input and Output of every linear and non-linear layers are in FP8 precision
    - The output of this layer is the input of the next layer
  - Save the FP8 activation instead of BF16 to reduce activation memory usage
  - Fuse quantize and dequantize operation into kernels to reduce overhead

- Propose **Mixed Granularity** Activation Quantization
  - **Vary** the quantization **granularity** across different layers
  - balance precision and efficiency

- Linear layers
  - Uses per-tensor quantization, since it is more hardware Friendly
    - Also compatible with other quantization methods ☺

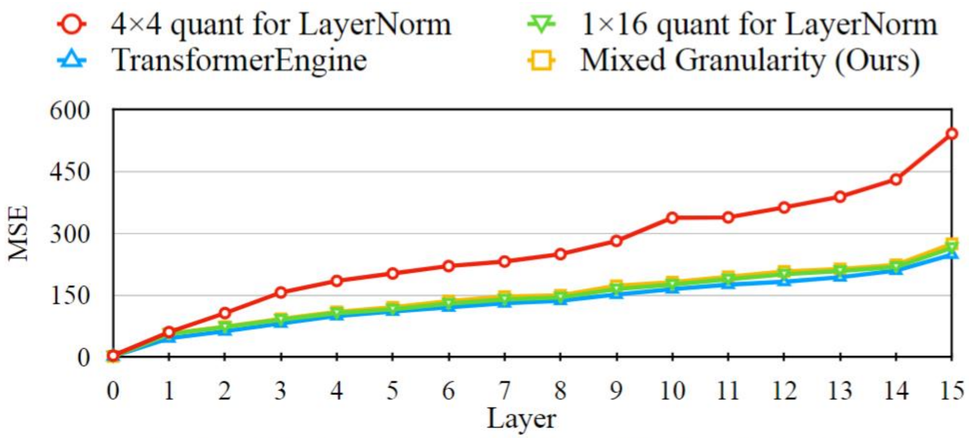
- Non-linear layers
  - Uses fine-grained quantization, since it offers better precision

Part 2: Mixed Granularity Activation Quantization

- Non-linear layers
  - Uses **Per-group quantization**, since it offers better precision
  - Possible choices
    - **Per-group quantization** – quantize every 1 x G elements independently
    - Per-block quantization – quantize every B x B elements independently

Per-group quantization is better than per-block quantization

LayerNorm's quantization error is large if quantization is performed across different token-axis



(a) 4 × 4 quantization of LayerNorm's input leads to large quantization error. The error becomes larger in deeper layers.

Part 2: Mixed Granularity Activation Quantization

- Linear Layers
  - Propose **Group Scaling** – Efficient just-in-time scaling for per-tensor quantization

  - Per-tensor quantization requires to first calculate the maximum value
    - Needs per-tensor reduction, and just-in-time scaling will introduce overhead

  - Delayed Scaling uses history to estimate the max value
      - Add implementation complexity and instability

Part 2: Mixed Granularity Activation Quantization

- Split the max reduction into two stages
  - (1) Perform max reduction on each $1 \times G$ element and storing the results as intermediate values
    - Can be seamlessly fused into previous kernels, without adding too much overhead
  - (2) Apply max reduction on the intermediate tensor to obtain the per-tensor max value
    - The intermediate tensor is smaller than original tensor, therefore the latency is reduced

The latency caused by reduction is greatly reduced

Tensor Size	Delayed Scaling	Just-in-Time Scaling		Group Scaling (Ours)
11008×16384	0.00 ms	1.37 ms	$\xrightarrow{14\times}$	0.10 ms
11008×8192	0.00 ms	0.89 ms	$\xrightarrow{11\times}$	0.08 ms
4096×16384	0.00 ms	0.55 ms	$\xrightarrow{8\times}$	0.07 ms
4096×8192	0.00 ms	0.32 ms	$\xrightarrow{5\times}$	0.06 ms

Experiments – Accuracy

- OLMo-7B pretraining from scratch
  - Train for approximately 250B tokens
  - Batch Size = 4M tokens
  - Nearly lossless performance!

Table 4: OLMo-7B pretraining performance on downstream tasks. We report the performance after training for 250B tokens.

	Train Loss	WikiText	C4	Pile	Avg ppl
BF16	2.366	12.053	12.874	8.596	11.174
COAT	2.379	12.166	12.988	8.684	11.279
	COPA	**ARC(Easy)**	**SciQ**	**HellaSwag**	**Avg Acc**
BF16	83.0%	65.7%	87.5%	56.9%	73.2 %
COAT	81.0%	61.9%	87.2%	60.6%	72.7 %

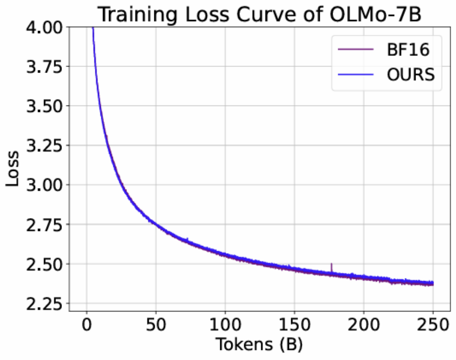


Figure 6: OLMo-7B training loss curve.

Experiments – Accuracy

- OLMo-1B pretraining from scratch
  - Train for 300B tokens
  - Batch Size = 4M tokens
  - Nearly lossless performance!

Table 3: OLMo-1B pretraining performance on downstream tasks. We report the performance after training for 300B tokens.

	Train Loss	WikiText	C4	Pile	Avg ppl
BF16	2.551	15.234	15.538	10.563	10.083
COAT	2.568	15.384	15.695	10.672	10.176
	COPA	**ARC(Easy)**	**SciQ**	**HellaSwag**	**Avg Acc**
BF16	77.0 %	57.3 %	84.0 %	54.5 %	68.2 %
COAT	75.0 %	58.1 %	83.9 %	54.3 %	67.8 %

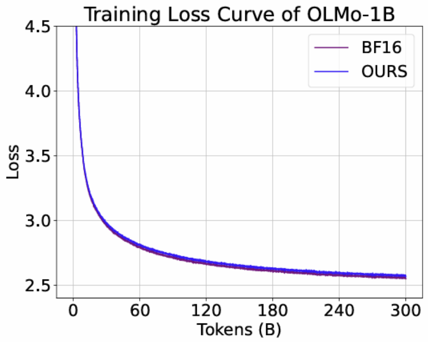


Figure 5: OLMo-1B training loss curve.

Experiments – Accuracy

- LLM Fine-tuning
  - On math corpus, using MAmmoTH dataset
  - Nearly lossless performance!

Table 4: Evaluation result of fine-tuning Llama-2-7B on math corpus. `Llama-2-7B` refers to the evaluation metric before fine-tuning. TE refers to TransformerEngine.

	Mathmeticas	**SVAMP**	**NumGLUE**	**GSM8k**	**Avg**
`Llama-2-7B`	6.0	14.6	34.5	29.9	21.3
BF16	46.3	64.2	54.8	57.7	55.7
TE	45.3	66.1	53.5	57.7	55.6
COAT	47.8	64.4	53.3	56.6	55.5

Experiments – Accuracy

- Vision Language Model Training
  - Perform Stage-3 SFT on VILA 1.5-7B
  - Nearly lossless performance!
  - Loss curve matches with baselines

Table 5: VILA1.5-7B Stage-3 SFT performance on downstream tasks. * means it has seen the training data.

Stage 3	VideoMME	POPE	VizWiz	GQA*	VQAv2*
BF16	42.96	86.90	61.42	64.55	81.47
TE	43.19	87.64	57.61	64.53	81.34
COAT	44.56	87.43	61.36	64.44	81.20

		SEED			
Stage 3	TextVQA	Image	Video	MMMU Val	Average
BF16	65.60	73.40	45.65	38.56	62.80
TE	64.70	73.51	43.12	35.89	61.88
COAT	64.65	73.36	43.76	37.22	62.51

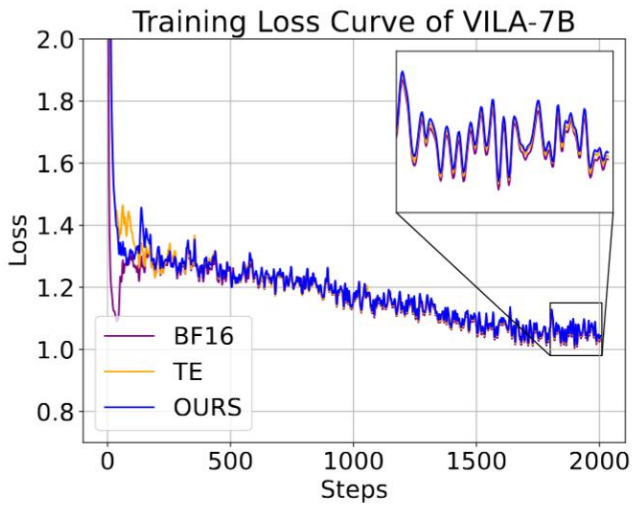


Figure 7: VILA1.5-7B Stage-3 SFT loss curve.

Experiments – Efficiency

Table 6: Memory Saving and Speedup for a single Transformer Layer. Memory refers to Activation Memory. Our method achieves better speedup than TransformerEngine and significantly reduces the activation memory footprint by 1.65×.

Hidden Size = 2048, Batch Size = 4												
	Sequence Length = 2048						Sequence Length = 4096					
	Forward	Backward	Total	Ratio	Memory	Ratio	Forward	Backward	Total	Ratio	Memory	Ratio
BF16	3.36	8.47	11.83	1.00×	1457 MB	1.00×	6.88	17.24	24.12	1.00×	2914 MB	1.00×
TE	2.96	5.32	8.28	1.42×	1338 MB	1.09×	5.94	11.29	17.23	1.39×	2677 MB	1.09×
COAT	2.88	5.16	8.04	**1.47×**	883 MB	**1.65×**	5.89	10.82	16.71	**1.44×**	1766 MB	**1.65×**
Hidden Size = 4096, Batch Size = 4												
	Sequence Length = 2048						Sequence Length = 4096					
	Forward	Backward	Total	Ratio	Memory	Ratio	Forward	Backward	Total	Ratio	Memory	Ratio
BF16	7.77	18.78	26.55	1.00×	2914 MB	1.00×	16.37	38.43	54.80	1.00×	5828 MB	1.00×
TE	6.19	11.79	17.98	1.47×	2677 MB	1.09×	12.66	24.58	37.24	1.47×	5355 MB	1.09×
COAT	5.89	10.96	16.85	**1.57×**	1766 MB	**1.65×**	12.16	23.44	35.6	**1.53×**	3533 MB	**1.65×**

Experiments – Efficiency

Our method achieves 1.54x memory reduction ratio compared with BF16
Comparable or even higher speedup compared with TE
Double the maximum batch size and therefore increase speedup

Table 7: End-to-end memory reduction and speedup results. BS refers to batch size. CL refers to context length. We report token/s per GPU for speed results. ‡ means CL=1024.

Llama-2-7B		*Context Length = 2048*				*Maximum Batch Size, Context Length = 2048*		
		Optimizer	Activations	Total	Ratio	Max BS	Speed	Ratio
$1\ GPU_{BS=1}$	BF16	-	-	OOM	-	-	OOM	-
	TE	-	-	OOM	-	-	OOM	-
	COAT	13.1 GB	8.1 GB	79.3 GB	✓	1	5906 token/s	✓
$2\ GPU_{BS=2}$	BF16	-	-	OOM	-	1	6130 token/s	1.00×
	TE	-	-	OOM	-	1	6842 token/s	1.11×
	COAT	6.5GB	16.9 GB	52.8 GB	✓	4	11351 token/s	1.85×
$4\ GPU_{BS=2}$	BF16	13.1 GB	25.8 GB	55.1 GB	1.00×	2	7730 token/s	1.00×
	TE	13.1 GB	23.9 GB	53.1 GB	1.04×	2	9577 token/s	1.24×
	COAT	3.2 GB	16.9 GB	35.6 GB	1.54×	4	11257 token/s	1.45×
$8\ GPU_{BS=2}$	BF16	6.5 GB	25.8 GB	41.2 GB	1.00×	4	8238 token/s	1.00×
	TE	6.5 GB	23.9 GB	39.3 GB	1.05×	4	11704 token/s	1.42×
	COAT	1.6 GB	16.9 GB	27.0 GB	1.52×	8	11241 token/s	1.36×

Experiments – Efficiency

- Our method enables us to double the batch size, and train Llama-2-13B with 2 GPU, and Llama-30B with 8 GPU

Llama-2-13B		*Context Length = 2048*				*Maximum Batch Size, Context Length = 2048*		
		Optimizer	Activations	Total	Ratio	Max BS	Speed	Ratio
$2\ \text{GPU}^{\ddagger}_{BS=1}$	BF16	-	-	OOM	-	-	OOM	-
	TE	-	-	OOM	-	-	OOM	-
	COAT	12.6 GB	10.1 GB	73.2 GB	✓	1	2137 token/s	✓
$4\ \text{GPU}_{BS=1}$	BF16	25.1 GB	20.1 GB	76.1 GB	1.00×	1	2345 token/s	1.00×
	TE	25.1 GB	18.6 GB	74.5 GB	1.02×	1	2851 token/s	1.21×
	COAT	6.3 GB	13.2 GB	49.1 GB	1.55×	2	5295 token/s	2.25×
$8\ \text{GPU}_{BS=1}$	BF16	12.6 GB	20.1 GB	49.4 GB	1.00×	2	3907 token/s	1.00×
	TE	12.6 GB	18.6 GB	47.9 GB	1.03×	2	5604 token/s	1.43×
	COAT	3.1 GB	13.2 GB	32.5 GB	1.52×	4	5650 token/s	1.44×
Llama-30B		*Context Length = 2048*				*Maximum Batch Size, Context Length = 2048*		
		Optimizer	Activations	Total	Ratio	Max BS	Speed	Ratio
$8\ \text{GPU}_{BS=1}$	BF16	-	-	OOM	-	-	OOM	-
	TE	-	-	OOM	-	-	OOM	-
	COAT	7.8 GB	24.2 GB	70.5 GB	✓	1	1363 token/s	✓

- Thanks!