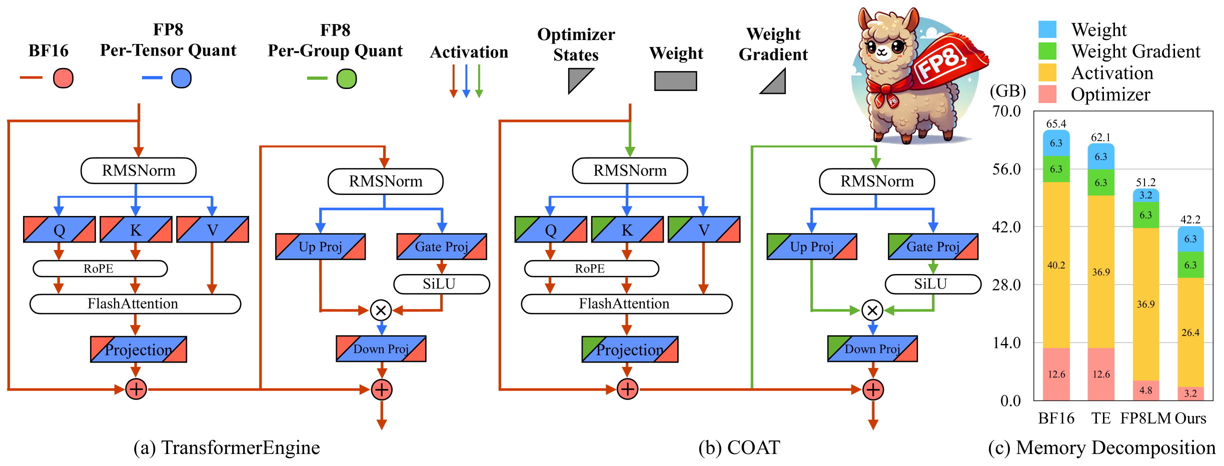
□COAT: Compressing Optimizer states and Activation for Memory-Efficient FP8 Training

[paper] | [website-TODO]



□ Introduction

We introduce COAT (Compressing Optimizer states and Activation for Memory-Efficient FP8 Training), a novel method designed to optimize the memory efficiency of training large models by compressing optimizer states and activations using FP8 quantization.

This technique allows:

- Reduced end-to-end memory footprint by 1.54×
- Speedup training by 1.43× while maintaining model accuracy
- Double the batch size and utilize GPU better
- Scalable performance for large-scale AI models

By leveraging FP8 precision, COAT significantly decreases memory usage by 1.54×, which is critical for training large models on limited hardware resources.

□ News

- [2024/10] [paper-TODO] is on arxiv!

□ Abstract

FP8 training has emerged as a promising method for improving training efficiency. Existing frameworks accelerate training by applying FP8 computation to linear layers while leaving optimizer states and activations in higher precision, which fails to fully optimize memory usage. This paper introduces COAT (**C**ompressing **O**ptimizer States and **A**ctivations for FP8 **T**raining), a novel FP8 training framework designed to significantly reduce memory footprint when training large models.

COAT addresses current limitations through two key innovations: (1) **Dynamic Range Expansion**, which aligns optimizer state distributions more closely with the FP8 representation range, thereby reducing quantization error, and (2) **Mixed-Granularity Activation Quantization**, which optimizes activation memory using a combination of per-tensor and per-group quantization strategies.

Experiments demonstrate that COAT effectively reduces end-to-end training memory footprint by **1.54×** compared to BF16 while achieving nearly lossless performance across various tasks, such as Large Language Model pretraining and fine-tuning and Vision Language Model training. COAT also achieves a **1.43×** end-to-end training speedup compared to BF16, performing on par with or surpassing TransformerEngine's speedup. COAT enables efficient full-parameter training of large models on fewer GPUs, and facilitates doubling the batch size in distributed training settings, providing a practical solution for scaling large-scale model training.

⚙ Installation

```
git clone --recurse-submodules TODO
cd TODO

# Create the conda environment and install coat
chmod +x environment_setup.sh
./environment_setup.sh coat

conda activate coat

# Install OLMo
cd examples/OLMo/
pip install -e .[all]
```

Performance

In all multi-GPU training setting, COAT can double the micro-batch size and therefore lead to even higher speedup. For example, our method can achieve $2.25\times$ speedup when training Llama-2-13B on 4-GPUs since we can effectively increase the batch size to 2.

Overall, COAT significantly reduces end-to-end memory usage by up to $1.55\times$ and speeds up the end-to-end training by nearly $1.44\times$. This facilitates full-parameter training on fewer GPUs, which is particularly beneficial for larger language models.

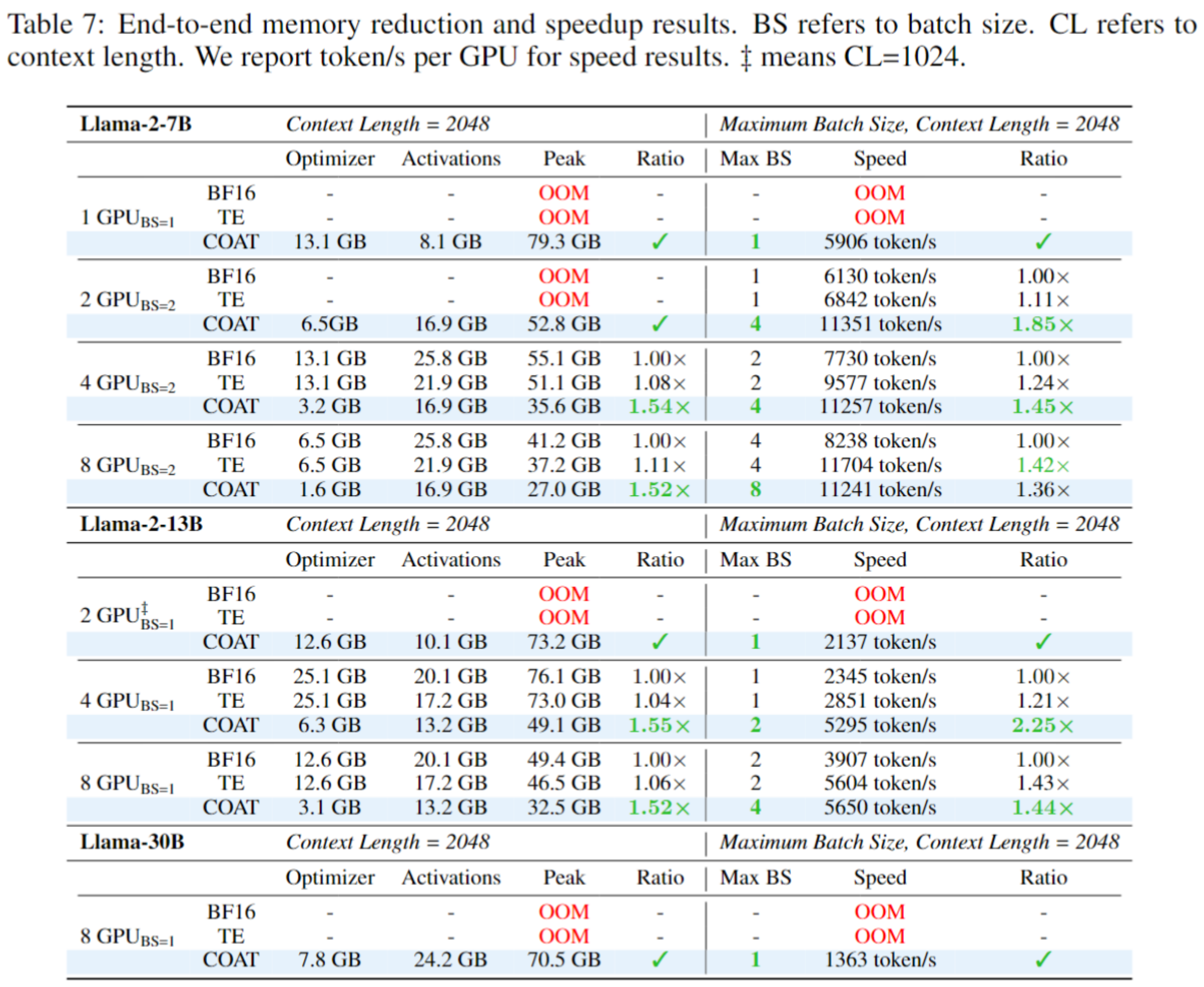
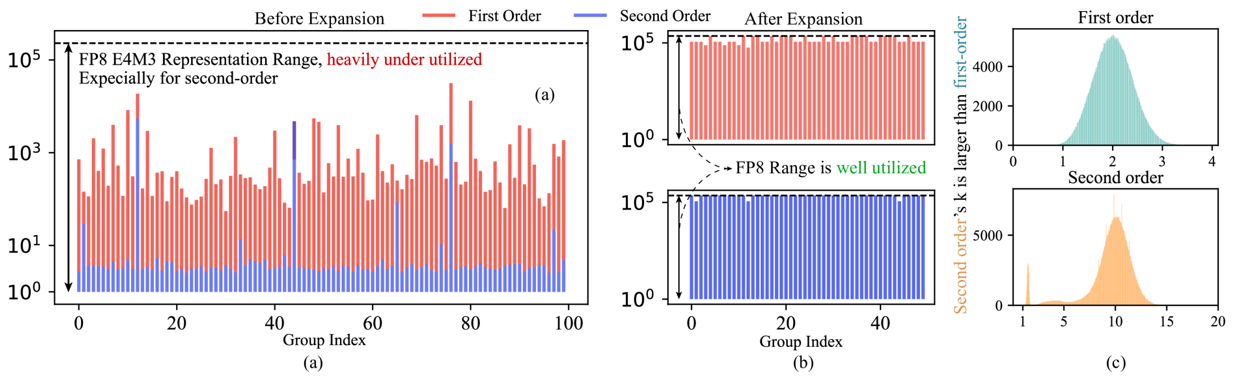


Table 7: End-to-end memory reduction and speedup results. BS refers to batch size. CL refers to context length. We report token/s per GPU for speed results. ‡ means CL=1024.

Llama-2-7B		*Context Length = 2048*				*Maximum Batch Size, Context Length = 2048*		
		Optimizer	Activations	Peak	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	21.9 GB	51.1 GB	1.08×	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	21.9 GB	37.2 GB	1.11×	4	11704 token/s	1.42×
	COAT	1.6 GB	16.9 GB	27.0 GB	1.52×	8	11241 token/s	1.36×
Llama-2-13B		*Context Length = 2048*				*Maximum Batch Size, Context Length = 2048*		
		Optimizer	Activations	Peak	Ratio	Max BS	Speed	Ratio
2 GPU$^{‡}_{BS=1}$	BF16	-	-	OOM	-	-	OOM	-
	TE	-	-	OOM	-	-	OOM	-
	COAT	12.6 GB	10.1 GB	73.2 GB	✓	1	2137 token/s	✓
4 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	17.2 GB	73.0 GB	1.04×	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 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	17.2 GB	46.5 GB	1.06×	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	Peak	Ratio	Max BS	Speed	Ratio
8 GPU$_{BS=1}$	BF16	-	-	OOM	-	-	OOM	-
	TE	-	-	OOM	-	-	OOM	-
	COAT	7.8 GB	24.2 GB	70.5 GB	✓	1	1363 token/s	✓

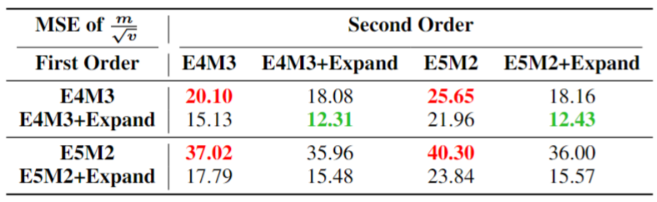
Key Observations

- FP8 E4M3's representation range is **under-utilized** for optimizer states quantization. The second-order momentum's dynamic range is also smaller than first order momentum's.



- **Dynamic Range Expansion** can fully utilize the FP8's representation range, therefore greatly reduce the quantization error. This finding is also compatible with other data formats than FP8.

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

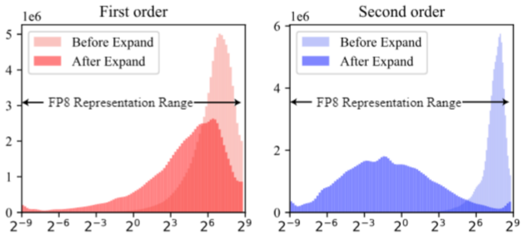
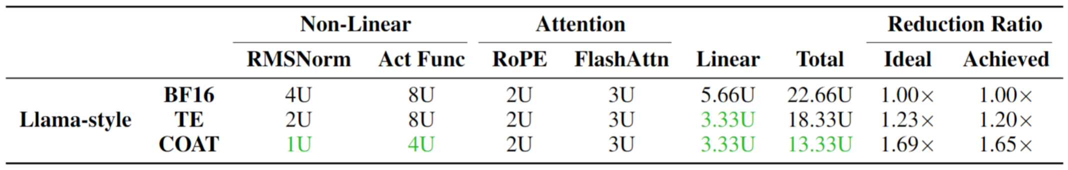


Figure 3: Dynamic Range Expansion can better utilize E4M3 representation range.

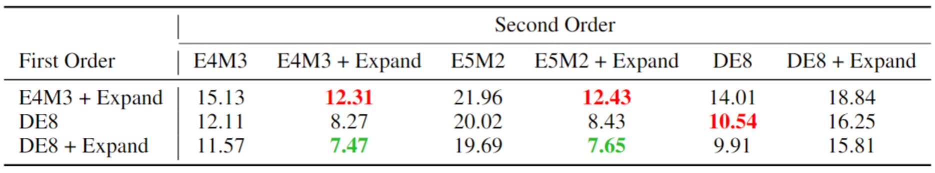
- Quantizing non-linear layers is important to reduce memory consumption. It usually consumes much more memory than linear layers.

Table 2: Activation memory footprint of different operators. U is a unit to measure memory usage, where 1U = Batch Size × Sequence Length × Hidden Size × 2 bytes (for BF16). For Llama-style model, Act Func refers to SiLU & Multiply, and Linear refers to the summation of QKV/Attn/Up/Gate/Down projection. RMSNorm is upcast to float32 in `transformers` implementation, so the memory usage of LayerNorm in BF16 is 4U. Our method reduces activation memory by quantizing them to FP8. More details about FlashAttention in Appendix D.



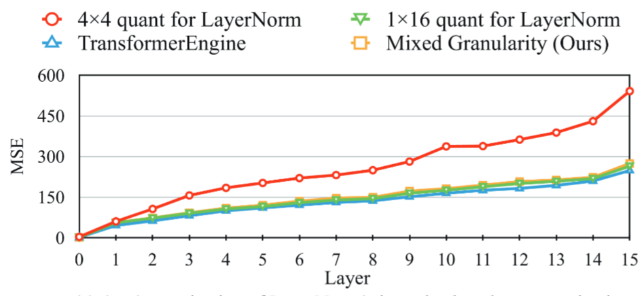
		Non-Linear		Attention				Reduction Ratio	
		RMSNorm	Act Func	RoPE	FlashAttn	Linear	Total	Ideal	Achieved
Llama-style	BF16	4U	8U	2U	3U	5.66U	22.66U	1.00×	1.00×
	TE	2U	8U	2U	3U	3.33U	18.33U	1.23×	1.20×
	COAT	1U	4U	2U	3U	3.33U	13.33U	1.69×	1.65×

Table 8: Dynamic Range Expansion is compatible with DE8 (8-bit dynamic quantization).

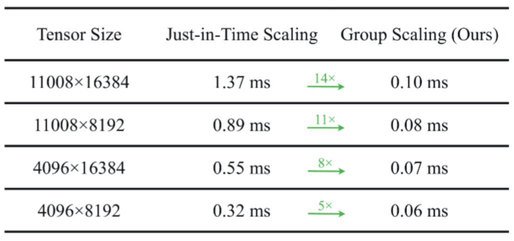


	Second Order					
First Order	E4M3	E4M3 + Expand	E5M2	E5M2 + Expand	DE8	DE8 + Expand
E4M3 + Expand	15.13	12.31	21.96	12.43	14.01	18.84
DE8	12.11	8.27	20.02	8.43	10.54	16.25
DE8 + Expand	11.57	7.47	19.69	7.65	9.91	15.81

- Mixed granularity FP8 Precision Flow optimizes memory and accuracy for activation quantization. **FP8 precision flow** means the input and the output of every linear and non-linear operator is in FP8 precision and reduces the memory footprint by 2×. **Mixing the granularity** of linear and non-linear layers can improve the quantization accuracy. **Group Scaling** is used to further reduce the overhead for just-in-time scaling.



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



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

(b) Time usage to calculate per-tensor max value. Our Group Scaling greatly reduce the reduction overhead, while it doesn't suffer from Delayed Scaling's instability issue.

Figure 4: (a) Quantization Error in forward pass. (b) Time comparison of various scaling methods.

Reproduce the results

Reproduce 1: End-to-end memory reduction and speedup on 7B model

```
# Reproduce the memory reduction in Table 7 of our paper.
cd examples/OLMo/

# BF16
MEMORY_BENCH=1 torchrun --nproc_per_node=4 scripts/train.py
configs/reproduce/OLMo-7B-reproduce-MemBench.yaml

# COAT
MEMORY_BENCH=1 torchrun --nproc_per_node=4 scripts/train.py
configs/coat/OLMo-7B-COAT-Both-MemBench.yaml
```

```
# Reproduce the speedup in Table 7 of our paper.
cd examples/OLMo/

# BF16
SPEED_BENCH=1 torchrun --nproc_per_node=4 scripts/train.py
configs/reproduce/OLMo-7B-reproduce-SpeedBench.yaml

# COAT
SPEED_BENCH=1 torchrun --nproc_per_node=4 scripts/train.py
configs/coat/OLMo-7B-COAT-Both-SpeedBench.yaml
```

Reproduce 2: Per TransformerLayer memory reduction and speedup

```
# Automatically compare BF16 and COAT
python benchmark/benchmark_olmolayer.py
```

Reproduce 3: OLMo-1B and OLMo-7B Pretraining

Dataset

First Prepare the training data and validation data following the instructions in OLMo's readme.

```
# An example to download the Dolma dataset
wget https://olmo-data.org/preprocessed/olmo-mix/v1_5/gpt-neox-20b-pii-
special/part-000-00000.npy
```

```
# Reproduce pretraining of OLMo-1B
cd examples/OLMo/

# BF16
```

```
torchrun --nproc_per_node=8 scripts/train.py configs/reproduce/OLMo-1B-
reproduce.yaml

# COAT
torchrun --nproc_per_node=8 scripts/train.py configs/coat/OLMo-1B-COAT-
BOTH.yaml

# Only linear layer in FP8
torchrun --nproc_per_node=8 scripts/train.py configs/fp8linear/OLMo-1B-
COAT-FP8Linear.yaml
```

```
# Reproduce pretraining of OLMo-7B
cd examples/OLMo/

# BF16
torchrun --nproc_per_node=8 scripts/train.py configs/reproduce/OLMo-7B-
reproduce.yaml

# COAT
torchrun --nproc_per_node=8 scripts/train.py configs/coat/OLMo-7B-COAT-
BOTH.yaml
```

Examples

Use COAT's FP8 Optimizer

```
from dataclasses import asdict, dataclass, field
from coat.optimizer.fp8_adamw import CoatAdamW

@dataclass
class QuantOptimizerConfig:
    """
    Quantization for model configuration.
    """

    # Whether to use QOLMo to train
    use_quantize_optimizer: Optional[str] = True

    # The quantization group size of optimizer states.
    qgroup_size: Optional[int] = 128

    # The granularity of the k when doing expansion.
    expand_min: Optional[int] = 16

    # Precision of the first order momentum
    first_order_bit: Optional[str] = "E4M3"

    # Whether to use Dynamic Range Expansion
    first_order_expansion: Optional[str] = "expansion"
    # Precision of the second order momentum
    second_order_bit: Optional[str] = "E4M3"

    # Whether to use Dynamic Range Expansion
    second_order_expansion: Optional[str] = "expansion"

    # Quantization epsilon to prevent all zero cases.
    epsilon: Optional[float] = 1e-15

qargs = QuantOptimizerConfig()

optimizer = CoatAdamW(qargs, param_groups, lr=1e-3)
```

Use COAT's FP8 Activation

We currently only support OLMo model. Llama 2/3 will be supported soon!

📅To-Do List

We will try our best to release

- [ ] COAT on TorchTitan
- [ ] COAT on FSDP2
- [ ] COAT on VILA

FAQ

If you have some problem when installing `qoptim_cuda`, you can try to install cudatoolkit following this link

Citation

```
@misc{xi2024coatcompressingoptimizerstates,
      title={COAT: Compressing Optimizer states and Activation for Memory-Efficient FP8 Training},
      author={Haocheng Xi and Han Cai and Ligeng Zhu and Yao Lu and Kurt Keutzer and Jianfei Chen and Song Han},
      year={2024},
      eprint={2410.19313},
      archivePrefix={arXiv},
      primaryClass={cs.LG},
      url={https://arxiv.org/abs/2410.19313},
}
```