Introduction to Distributed Shampoo: Algorithm and Systems

Hao-Jun Michael Shi and Shintaro Iwasaki

AI and Systems Co-Design, Meta Platforms, Inc.

March 25, 2025

Why Should We Care About Optimizers?

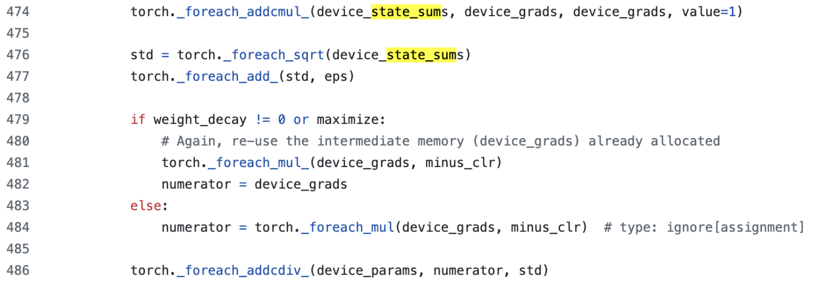
- Training algorithm improvements lead to:
  - Faster convergence to the same model quality, or
  - Higher quality models (in single epoch training) with the same amount of data / iterations.
- Unlike model scaling, only changes training costs, with fixed inference and serving costs.
- Unlike previous element-wise optimizers (SGD, Adam, AdaGrad), Shampoo requires:
  - Higher memory utilization and more compute during training.
  - More complex operators (root inverse computation).
  - Is tensor-shape dependent.
- Shampoo-like algorithms are being used at major companies, including Google and Meta!
- Already new developments such as eigenvalue-corrected Shampoo / SOAP, Muon, etc.

Recalling AdaGrad

Let $\epsilon > 0$. Initialize $v_0 = 0$. Then:

$$v_k = v_{k-1} + g_k^2$$
$$w_{k+1} = w_k - \alpha_k \frac{g_k}{\sqrt{v_k} + \epsilon}$$

Due to only leveraging element-wise operators, we implement AdaGrad by constructing optimizer states for each parameter with the same shape, and apply a series of element-wise operations, i.e.,



```
torch._foreach_addcmul_(device_state_sums, device_grads, device_grads, value=1)

std = torch._foreach_sqrt(device_state_sums)
torch._foreach_add_(std, eps)

if weight_decay != 0 or maximize:
    # Again, re-use the intermediate memory (device_grads) already allocated
    torch._foreach_mul_(device_grads, minus_clr)
    numerator = device_grads
else:
    numerator = torch._foreach_mul(device_grads, minus_clr)  # type: ignore[assignment]

torch._foreach_addcdiv_(device_params, numerator, std)
```

Recalling AdaGrad

Let $\epsilon > 0$. Initialize $v_0 = 0$. Then:

$$\begin{aligned} v_k &= v_{k-1} + g_k^2 \\ w_{k+1} &= w_k - \alpha_k \frac{g_k}{\sqrt{v_k} + \epsilon} \end{aligned}$$

Recalling AdaGrad

Let $\epsilon > 0$. Initialize $v_0 = 0$. Then:

$$v_k = v_{k-1} + g_k^2$$
$$w_{k+1} = w_k - \alpha_k \frac{g_k}{\sqrt{v_k} + \epsilon}$$

Ignoring ϵ, this is equivalent to using a diagonal scaling:

$$\underbrace{\begin{bmatrix} w_{k+1,1} \\ w_{k+1,2} \\ \vdots \\ w_{k+1,n} \end{bmatrix}}_{w_{k+1}} = \underbrace{\begin{bmatrix} w_{k,1} \\ w_{k,2} \\ \vdots \\ w_{k,n} \end{bmatrix}}_{w_k} - \alpha_k \underbrace{\begin{bmatrix} v_{k,1} & 0 & \dots & 0 \\ 0 & v_{k,2} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & v_{k,n} \end{bmatrix}^{-1/2}}_{A_k^{-1/2}} \underbrace{\begin{bmatrix} g_{k,1} \\ g_{k,2} \\ \vdots \\ g_{k,n} \end{bmatrix}}_{g_k}$$

This gives us a more general mathematical formulation that is more commonly used for understanding optimization algorithms.

Adaptive Gradient Methods [Duchi, et al., 2011]

If g_k is the (mini-batch) stochastic gradient, we can write AdaGrad as:

$$w_{k+1} = w_k - \alpha_k A_k^{-1/2} g_k$$

where $\alpha_k > 0$ is the learning rate/steplength and

$$A_k = \begin{cases} \sum_{t=1}^{k} \mathrm{diag}(g_t^2) & \text{if diagonal Adagrad,} \\ \sum_{t=1}^{k} g_t g_t^T & \text{if full-matrix Adagrad.} \end{cases}$$

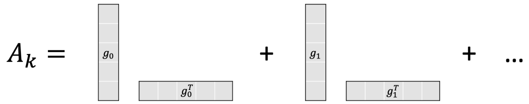
Adaptive Gradient Methods [Duchi, et al., 2011]

If g_k is the (mini-batch) stochastic gradient, we can write AdaGrad as:

$$w_{k+1} = w_k - \alpha_k A_k^{-1/2} g_k$$

where $\alpha_k > 0$ is the learning rate/steplength and

$$A_k = \begin{cases} \sum_{t=1}^{k} \mathrm{diag}(g_t^2) & \text{if diagonal Adagrad,} \\ \sum_{t=1}^{k} g_t g_t^T & \text{if full-matrix Adagrad.} \end{cases}$$



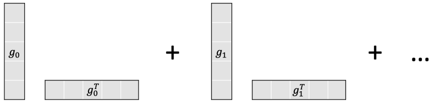
Adaptive Gradient Methods [Duchi, et al., 2011]

If g_k is the (mini-batch) stochastic gradient, we can write AdaGrad as:

$$w_{k+1} = w_k - \alpha_k A_k^{-1/2} g_k$$

where $\alpha_k > 0$ is the learning rate/steplength and

$$A_k = \begin{cases} \sum_{t=1}^{k} \operatorname{diag}(g_t^2) & \text{if diagonal Adagrad,} \\ \sum_{t=1}^{k} g_t g_t^T & \text{if full-matrix Adagrad.} \end{cases}$$

$A_k =$ 

- Diagonal approximations do not capture any pairwise correlations!
- Diagonal AdaGrad is cheap: $O(d)$ memory, $O(d)$ FLOPs/step.
- Full-matrix AdaGrad is expensive: $O(d^2)$ memory, $O(d^3)$ FLOPs/step.
- Note that $A^{1/2}$ refers to the **matrix** square-root ($A^{1/2}A^{1/2} = A$), which is not equivalent to **element-wise** square-root $B_{ij} = \sqrt{A_{ij}}$, $B \odot B = A$.

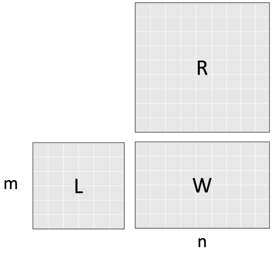
Shampoo (for Matrices)

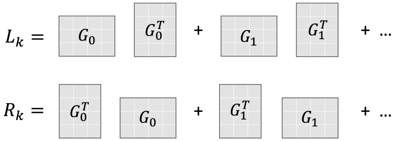
Let us focus on a single fully-connected layer (without bias) for now, with parameters and gradients $W, G \in \mathbb{R}^{m \times n}$.

Initialize: $L_0 = 0 \in \mathbb{R}^{m \times m}, R_0 = 0 \in \mathbb{R}^{n \times n}$.
Then for each step k:

$$\begin{aligned} L_k &= L_{k-1} + G_k G_k^T \\ R_k &= R_{k-1} + G_k^T G_k \\ W_{k+1} &= W_k - \alpha_k L_k^{-1/4} G_k R_k^{-1/4} \end{aligned}$$

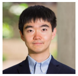
L_k, R_k are symmetric positive semi-definite!




$L_k = G_0 \; G_0^T + G_1 \; G_1^T + \dots$
$R_k = G_0^T \; G_0 + G_1^T \; G_1 + \dots$

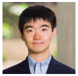
This can be generalized to tensors of arbitrary order.

Matrix Root Inverse Computation



How to compute $A \mapsto A^{-1/4}$ (or $A^{-1/p}$ for $p \in \mathbb{Z}$) for A symmetric positive semi-definite?

Matrix Root Inverse Computation

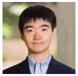


How to compute $A \mapsto A^{-1/4}$ (or $A^{-1/p}$ for $p \in \mathbb{Z}$) for A symmetric positive semi-definite?

Main Approaches:

1. Direct Methods: **Symmetric Eigendecomposition (Focus)**
2. Iterative Methods: Coupled Newton (or Higher-Order) Inverse Iteration [Higham, 2008, Lakic, 1998]
3. Warm-Started QR Algorithm (Orthogonal Iteration)

Matrix Root Inverse Computation



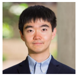
How to compute $A \mapsto A^{-1/4}$ (or $A^{-1/p}$ for $p \in \mathbb{Z}$) for A symmetric positive semi-definite?

Main Approaches:

1. Direct Methods: **Symmetric Eigendecomposition (Focus)**
2. Iterative Methods: Coupled Newton (or Higher-Order) Inverse Iteration [Higham, 2008, Lakic, 1998]
3. Warm-Started QR Algorithm (Orthogonal Iteration)

Key Idea: Compute eigendecomposition of $A = Q\Lambda Q^T$, then construct matrix root inverse by $A^{-1/4} = Q\Lambda^{-1/4}Q^T$, with as small modification to A as possible.

Matrix Root Inverse Computation



How to compute $A \mapsto A^{-1/4}$ (or $A^{-1/p}$ for $p \in \mathbb{Z}$) for A symmetric positive semi-definite?

Main Approaches:

1. Direct Methods: **Symmetric Eigendecomposition (Focus)**
2. Iterative Methods: Coupled Newton (or Higher-Order) Inverse Iteration [Higham, 2008, Lakic, 1998]
3. Warm-Started QR Algorithm (Orthogonal Iteration)

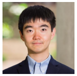
Key Idea: Compute eigendecomposition of $A = Q\Lambda Q^T$, then construct matrix root inverse by $A^{-1/4} = Q\Lambda^{-1/4}Q^T$, with as small modification to A as possible.

Practical challenge of supporting this computation on different hardware platforms:

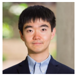
- `torch.linalg.eigh` or `torch.linalg.qr` requires `cuSOLVER` for NVIDIA, `rocSOLVER` for AMD.
- CPU offloading or coupled inverse iterations for MTIA.

Pseudocode

Performance Optimizations

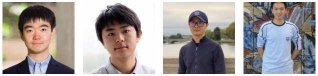


Performance Optimizations

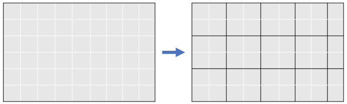


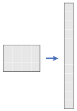
1. **Periodic Root Inverse Computation**:
   - Periodically compute the matrix root inverses every `precondition_frequency` iterations.
   - Introduces staleness in the matrix root inverses.

Performance Optimizations

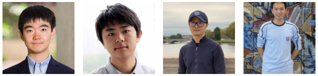


1. **Periodic Root Inverse Computation**:
   - Periodically compute the matrix root inverses every `precondition_frequency` iterations.
   - Introduces staleness in the matrix root inverses.
2. **Blocking and Merging**:
   - Block large tensors and apply Shampoo to each block.
   - Merge away small consecutive dimensions.

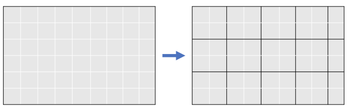


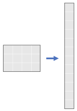


Performance Optimizations



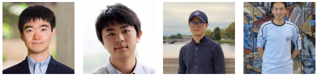
1. **Periodic Root Inverse Computation**:
   - Periodically compute the matrix root inverses every `precondition_frequency` iterations.
   - Introduces staleness in the matrix root inverses.
2. **Blocking and Merging**:
   - Block large tensors and apply Shampoo to each block.
   - Merge away small consecutive dimensions.



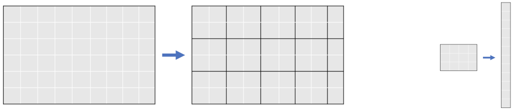


3. **Distributed Computation and Memory via `DTensor`**:
   - Distribute computation and optimizer states of different parameter blocks in distributed data-parallel training to reduce computational and memory requirements.
   - `AllGather` updates at every iteration.

Performance Optimizations

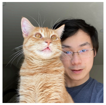


1. **Periodic Root Inverse Computation**:
   - Periodically compute the matrix root inverses every `precondition_frequency` iterations.
   - Introduces staleness in the matrix root inverses.
2. **Blocking and Merging**:
   - Block large tensors and apply Shampoo to each block.
   - Merge away small consecutive dimensions.



3. **Distributed Computation and Memory via `DTensor`**:
   - Distribute computation and optimizer states of different parameter blocks in distributed data-parallel training to reduce computational and memory requirements.
   - `AllGather` updates at every iteration.
4. `foreach` **Operators (Horizontal Fusion)**
5. **PyTorch 2.0 Compiler (Vertical Fusion)**

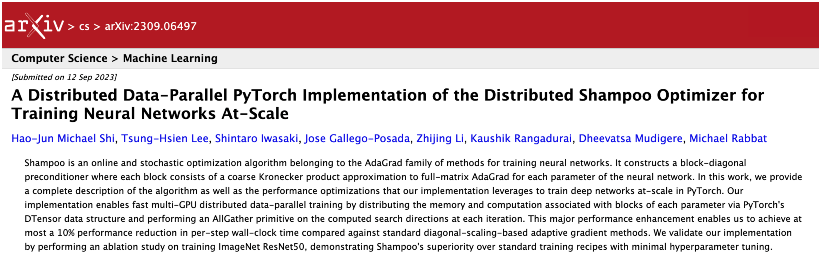
Acknowledgements - PyTorch Shampoo Core Team



Amongst many others for internal infra support and model onboarding!

Major credit to **Rohan Anil** *and* **Vineet Gupta (Google)** *for development of the original Shampoo algorithm, including grafting!*

For More Details...



arXiv > cs > arXiv:2309.06497

Computer Science > Machine Learning

[Submitted on 12 Sep 2023]

A Distributed Data-Parallel PyTorch Implementation of the Distributed Shampoo Optimizer for Training Neural Networks At-Scale

Hao-Jun Michael Shi, Tsung-Hsien Lee, Shintaro Iwasaki, Jose Gallego-Posada, Zhijing Li, Kaushik Rangadurai, Dheevatsa Mudigere, Michael Rabbat

Shampoo is an online and stochastic optimization algorithm belonging to the AdaGrad family of methods for training neural networks. It constructs a block-diagonal preconditioner where each block consists of a coarse Kronecker product approximation to full-matrix AdaGrad for each parameter of the neural network. In this work, we provide a complete description of the algorithm as well as the performance optimizations that our implementation leverages to train deep networks at-scale in PyTorch. Our implementation enables fast multi-GPU distributed data-parallel training by distributing the memory and computation associated with blocks of each parameter via PyTorch's DTensor data structure and performing an AllGather primitive on the computed search directions at each iteration. This major performance enhancement enables us to achieve at most a 10% performance reduction in per-step wall-clock time compared against standard diagonal-scaling-based adaptive gradient methods. We validate our implementation by performing an ablation study on training ImageNet ResNet50, demonstrating Shampoo's superiority over standard training recipes with minimal hyperparameter tuning.

Our open-source implementation is available at:
github.com/facebookresearch/optimizers

Questions?

References

1. Agarwal, Naman, et al. "Disentangling adaptive gradient methods from learning rates." arXiv preprint arXiv:2002.11803 (2020).
2. Anil, Rohan, et al. "On the Factory Floor: ML Engineering for Industrial-Scale Ads Recommendation Models." arXiv preprint arXiv:2209.05310 (2022).
3. Anil, Rohan, et al. "Scalable second order optimization for deep learning." arXiv preprint arXiv:2002.09018 (2020).
4. Dahl, George E., et al. "Benchmarking neural network training algorithms." arXiv preprint arXiv:2306.07179 (2023).
5. Defazio, Aaron. "Momentum via primal averaging: theoretical insights and learning rate schedules for non-convex optimization." arXiv preprint arXiv:2010.00406 (2020).
6. Duchi, John, Elad Hazan, and Yoram Singer. "Adaptive subgradient methods for online learning and stochastic optimization." Journal of machine learning research 12.7 (2011).
7. Gupta, Vineet, Tomer Koren, and Yoram Singer. "Shampoo: Preconditioned stochastic tensor optimization." International Conference on Machine Learning. PMLR, 2018.
8. Higham, Nicholas J. "Functions of matrices: theory and computation." Society for Industrial and Applied Mathematics, 2008.

References

9. Kunstner, Frederik, et al. "Limitations of the empirical Fisher approximation for natural gradient descent." Advances in neural information processing systems 32 (2019).
10. Lakić, Slobodan. "On the computation of the matrix k-th root." ZAMM-Journal of Applied Mathematics and Mechanics/Zeitschrift für Angewandte Mathematik und Mechanik: Applied Mathematics and Mechanics 78.3 (1998): 167-172.
11. Morwani, Depen, et al. "A New Perspective on Shampoo's Preconditioner." arXiv preprint arXiv:2406.17748 (2024).
12. Rajbhandari, Samyam, et al. "Zero: Memory optimizations toward training trillion parameter models." SC20: International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, 2020.
13. Reid, Machel, et al. "Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context." arXiv preprint arXiv:2403.05530 (2024).
14. Shallue, Christopher J., et al. "Measuring the effects of data parallelism on neural network training." Journal of machine learning research 20.112 (2019): 1-49.
15. Singh, Siddharth, Zachary Sating, and Abhinav Bhatele. "Jorge: Approximate Preconditioning for GPU-efficient Second-order Optimization." arXiv preprint arXiv:2310.12298 (2023).

References

16. Van Loan, Charles F., and Nikos Pitsianis. “Approximation with Kronecker products.” Springer Netherlands, 1993.
17. Vyas, Nikhil, et al. ”SOAP: Improving and Stabilizing Shampoo using Adam.” arXiv preprint arXiv:2409.11321 (2024).
18. Zhao, Yanli, et al. ”Pytorch FSDP: experiences on scaling fully sharded data parallel.” arXiv preprint arXiv:2304.11277 (2023).

DDP Distributed Shampoo

Shintaro Iwasaki, Michael Shi

Technical Challenges in Shampoo

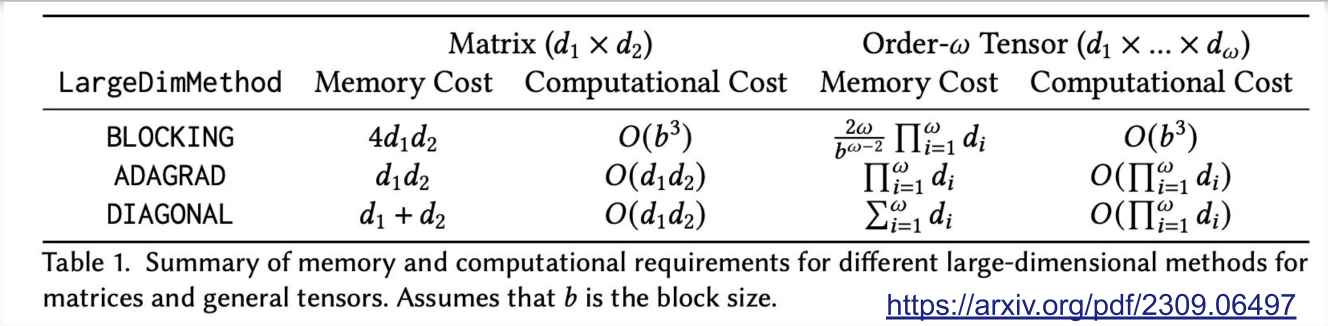
In training:

1. Shampoo requires more computation
2. Shampoo requires more memory
3. Shampoo uses complex operations
4. Complex checkpointing
5. More hyper-parameters
6. ...

The main focus of this presentation.

https://arxiv.org/pdf/2309.06497

More Memory/Computation Needed



	Matrix ($d_1 \times d_2$)		Order-ω Tensor ($d_1 \times \ldots \times d_\omega$)	
LargeDimMethod	Memory Cost	Computational Cost	Memory Cost	Computational Cost
BLOCKING	$4d_1d_2$	$O(b^3)$	$\frac{2\omega}{b^{\omega-2}} \prod_{i=1}^{\omega} d_i$	$O(b^3)$
ADAGRAD	d_1d_2	$O(d_1d_2)$	$\prod_{i=1}^{\omega} d_i$	$O(\prod_{i=1}^{\omega} d_i)$
DIAGONAL	$d_1 + d_2$	$O(d_1d_2)$	$\sum_{i=1}^{\omega} d_i$	$O(\prod_{i=1}^{\omega} d_i)$

Table 1. Summary of memory and computational requirements for different large-dimensional methods for matrices and general tensors. Assumes that b is the block size.

https://arxiv.org/pdf/2309.06497

Memory (*):

- For AdaGrad, we need P where the number of parameters is P.
- For Shampoo, we need at least 4P.

Computation (**)

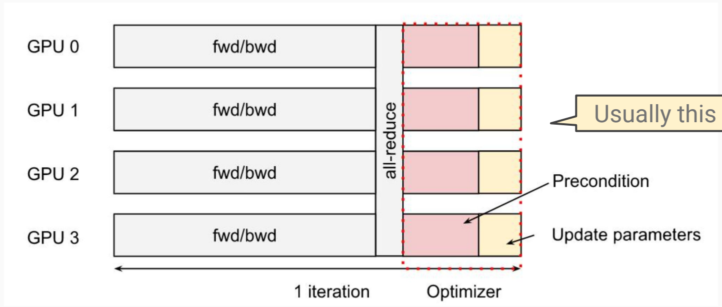
- For AdaGrad, computational cost is O(P)
- For Shampoo, it is O(P * b)

(*): Assuming 2D tensors.
(**): Assuming 2D tensors. GEMM cost for each block is **O(b^3)** where the number of blocks is d1/b * d2/b.

DDP: Distributed Data Parallel

Idea: DDP Naive Shampoo

AdaGrad

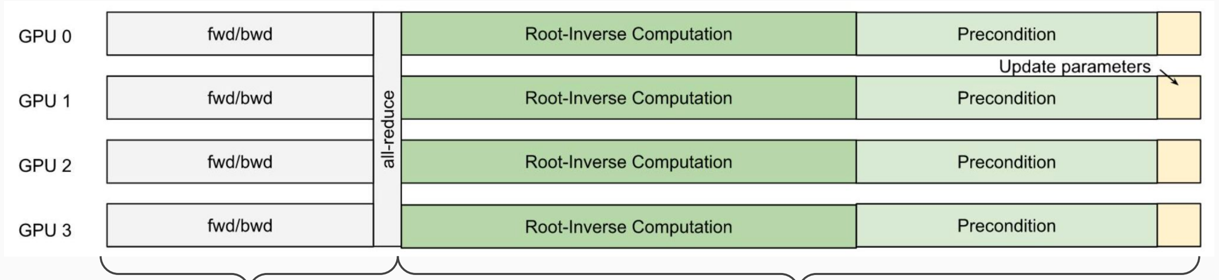


Usually this optimizer is very lightweight.

It highly depends on the model, but if fwd/bwd exec time is 100%:

- AdaGrad precond + param update: 2%
- Shampoo root-inverse: 10000%
- Shampoo precond + param update: 100%

Shampoo (naive)

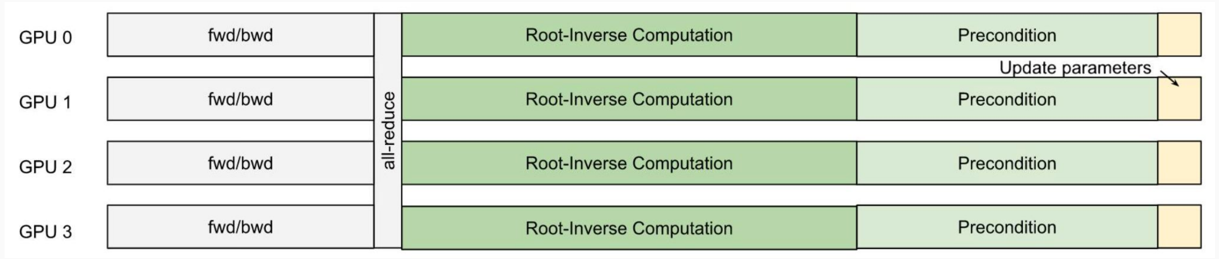


PyTorch’s DDP provides this.

PyTorch’s DDP optimizer interface

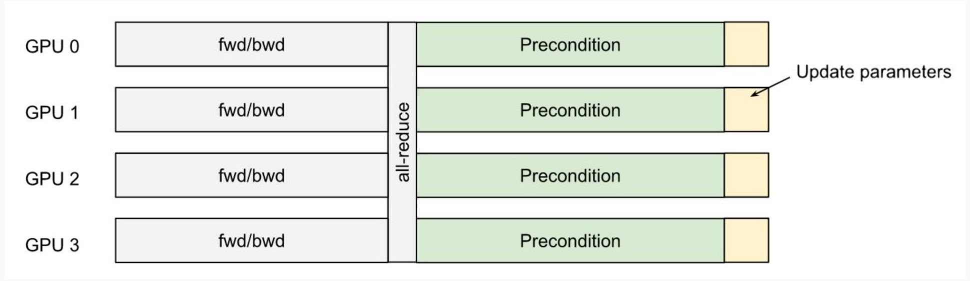
1. Precondition Frequency

Shampoo (naive)



Execute root-inverse computation every N iterations (where N is, for example, 8000)

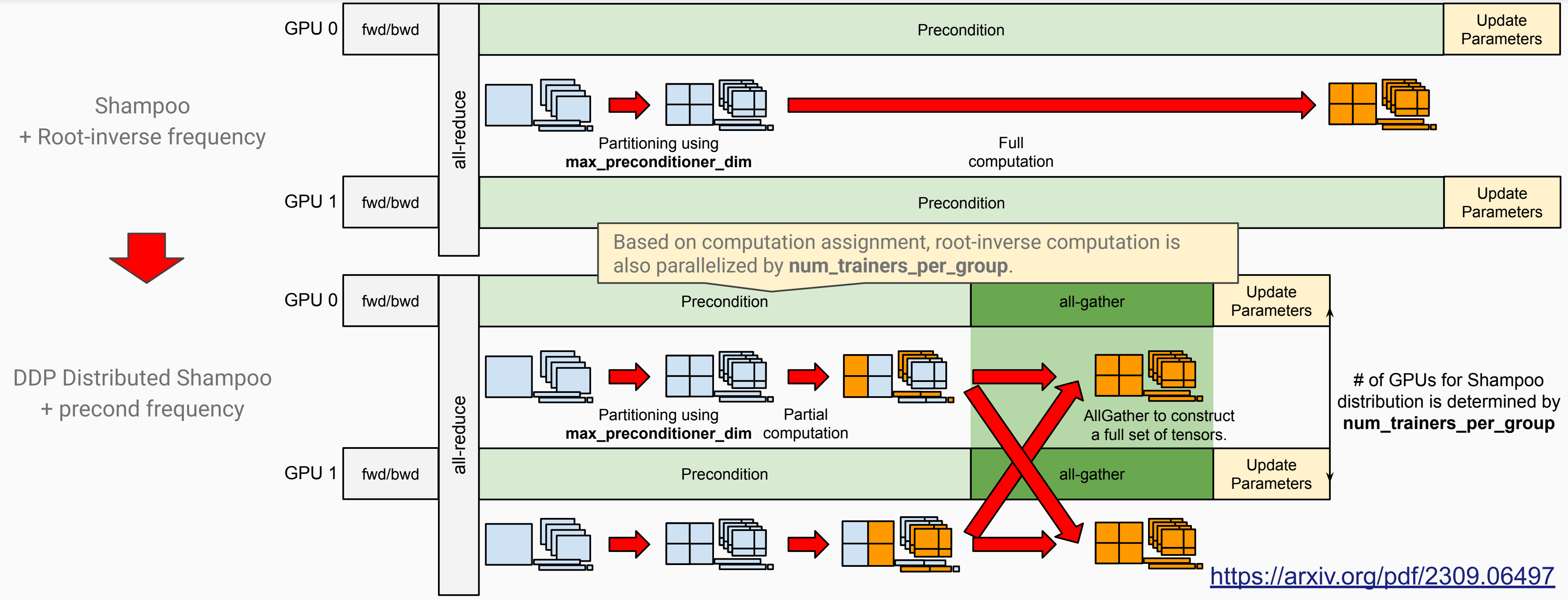
Shampoo
+ Root-inverse frequency



2. DDP Distributed Shampoo

Shampoo
+ Root-inverse frequency

DDP Distributed Shampoo
+ precond frequency


GPU 0
fwd/bwd
all-reduce
Precondition
Update Parameters
Partitioning using **max_preconditioner_dim**
Full computation
GPU 1
fwd/bwd
Precondition
Update Parameters
Based on computation assignment, root-inverse computation is also parallelized by **num_trainers_per_group**.
GPU 0
fwd/bwd
all-reduce
Precondition
all-gather
Update Parameters
Partitioning using **max_preconditioner_dim**
Partial computation
AllGather to construct a full set of tensors.
# of GPUs for Shampoo distribution is determined by **num_trainers_per_group**
GPU 1
fwd/bwd
Precondition
all-gather
Update Parameters

https://arxiv.org/pdf/2309.06497

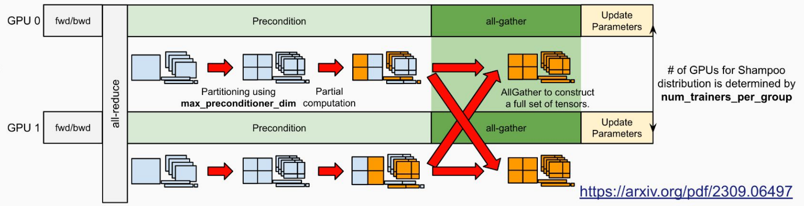
Memory/Computation/Convergence Trade-Off

Sometimes N != all the number of GPUs used for training.

Compared with the original Shampoo, where N is **number_trainers_per_group**

- Memory (*): 4P
  -> (4P / N + P) * C(N)
- Computation (**): O(P * b)
  -> $O(P * b / N) * C(N) + O_{comm}(P) * C(N) + O_{update}(P)$

DDP Distributed Shampoo
+ precond frequency


GPU 0
fwd/bwd
Precondition
all-gather
Update Parameters
all-reduce
Partitioning using **max_preconditioner_dim**
Partial computation
AllGather to construct a full set of tensors.
GPU 1
fwd/bwd
Precondition
all-gather
Update Parameters
# of GPUs for Shampoo distribution is determined by **num_trainers_per_group**

https://arxiv.org/pdf/2309.06497

In practice, C(N) is an important imbalance factor as we cannot distribute blocked parameters evenly across all the trainers. When N is large, C(N) gets larger.

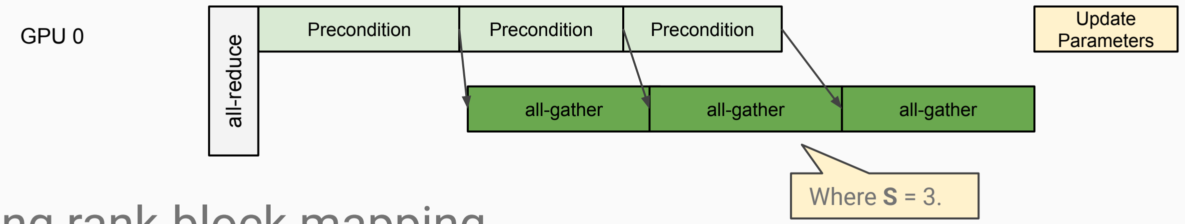
	Per-Iteration Performance	**Convergence**	**Memory Usage**
Large **precondition_frequency**	Increase	Worse	No effect
Large **max_preconditioner_dim**	Sweet spot around 2K - 8K	Better	Increase
Large **num_trainers_per_group**	Sweet spot around 16 - 64	No effect	(Depends)

(*): Assuming 2D tensors. 4P/ N for local Shampoo states. "+ P" accounts for the communication buffer before parameter updates. C(N) is a coefficient for imbalanced distribution across trainers (>= 1.0).
(**): Assuming 2D tensors. O(P * b / N) for precondition cost. $O_{update}(P)$ to update parameters in the end. $O_{comm}(P)$ is an overhead of AllGather.

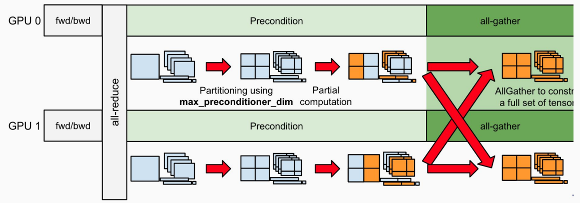
More Optimization Ideas (1/2)

In addition to this, you can come up with more ideas:

- Computation-communication overlapping (by having **S** stages)



- Optimizing rank-block mapping

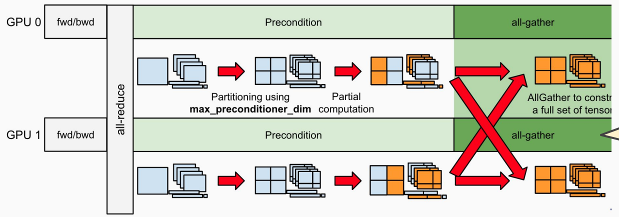


How to map rank and blocks for a large **N**?
Note that for the block size (b1 * b2):

- Memory cost: b1 * b2
- Computation cost: $b1^2 + b2^2$
- Communication cost: $\max_{rank}$(sum(block_size))

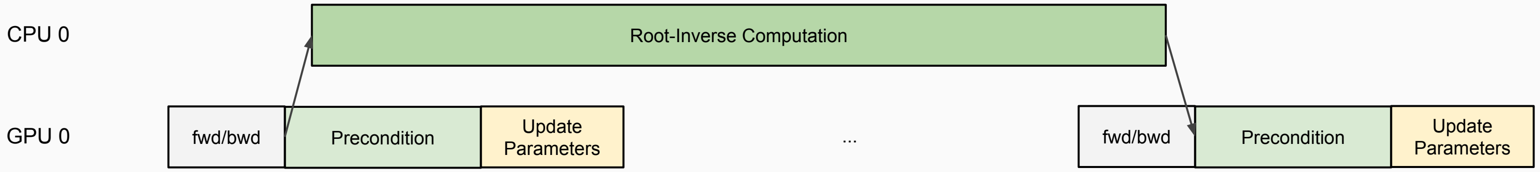
More Optimization Ideas (2/2)

- Quantized computation/communication



e.g., use quantized comm to reduce communication cost.

- Root-inverse computation overlapping



Practicality Aspects

Most optimizations introduce additional complexity, making Shampoo harder to use.

- Stability of complex operations (matrix root-inverse computation)
  - Prefer CPU? New algorithm?
- Distributed checkpointing
  - How to check-point optimizer states?
  - How to reshard it if we want to change the number of trainers?
- Simple hyper-parameter tuning
  - Shampoo exposes too many hyperparameters that affect memory/computation/convergence/ …
  - Autotuning? What is an objective function?
- Support for FSDP
- Simpler code for maintenance

Conclusions

Distributed Shampoo:

- Alleviates performance and memory consumption issues
- Introduces further optimization opportunities and complexity

Understanding the entire trade-off is crucial for building a good system.

- Distributed Shampoo is a good example. Don’t make it too complex.