Parallelize Muon with FSDP2

Motivation

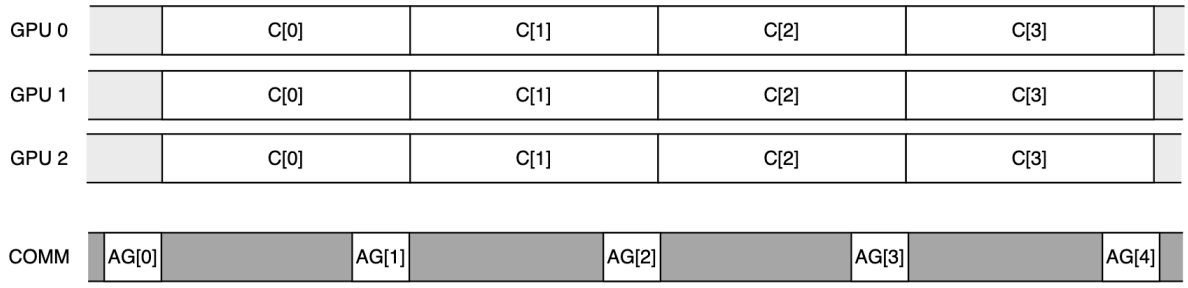
Algorithm 1 Distributed Muon

Require: Full Gradients $\mathbf{G}$, DP partitioned Momentum $\mathbf{m}$, DP partitioned parameters $\mathbf{p}$, momentum μ.

// Reduce-scatter G on DP for correct gradients
$\mathbf{g} =$ reduce_scatter($\mathbf{G}$, dp_group)
// Apply momentum to $\mathbf{g}$ using local partitioned momentum $\mathbf{m}$
$\mathbf{g}' =$ update_with_momentum($\mathbf{g}, \mathbf{m}, \mu$)
// DP Gather: gathering $\mathbf{g}'$ across DP into a full matrix $\mathbf{G}$
$\mathbf{G} =$ gather($\mathbf{g}'$, dp_group)
// Calculate Muon update
$\mathbf{U} =$ Newton-Schulz($\mathbf{G}$)
// Discard the rest of $\mathbf{U}$ and only keep the local partition $\mathbf{u}$, then apply the update rule
$\mathbf{p}' =$ apply_update($\mathbf{p}, \mathbf{u}$)
// All-gather updated $\mathbf{p}'$ into $\mathbf{P}$
$\mathbf{P} =$ all_gather($\mathbf{p}'$, dp_group)
// Return the update RMS for logging
return $\sqrt{\mathbf{u}^2.\texttt{mean}()}$

Distributed Muon by Moonlight

While a distributed version of Muon is available, it has the drawback of redundant computations across GPUs.



Execution timeline of Distributed Muon

- `C[i]` : Compute Newton-Schulz(G) for i-th gradient
- `AG[i]` : AllGather i-th gradient
- `G[i]` : Gather i-th gradient
- `SC[i]` : Scatter i-th gradient

Algorithm

Parallel Muon

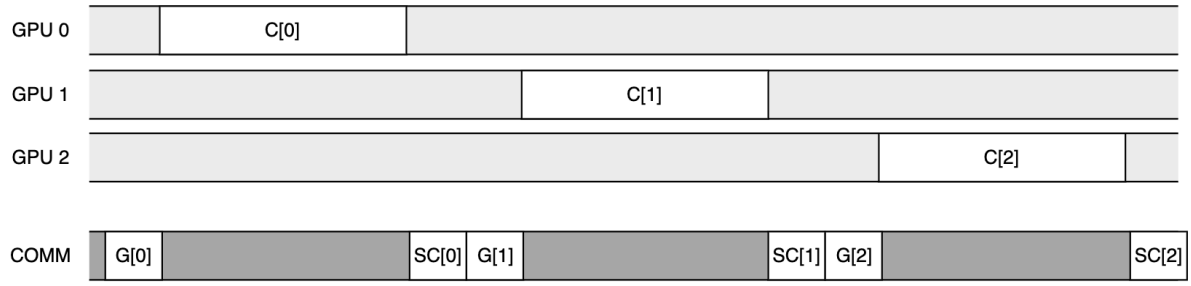
Algorithm 1 Parallel Muon

Require: DP partitioned gradient $\mathbf{g}$, DP partitioned Momentum $\mathbf{m}$, DP partitioned parameter $\mathbf{p}$, momentum μ, local rank $\mathbf{r}$

// Apply momentum to $\mathbf{g}$ using local partitioned momentum $\mathbf{m}$
$\mathbf{g}' \leftarrow$ update_with_momentum$(\mathbf{g}, \mathbf{m}, \mu)$
// Schedule $\mathbf{g}'$ to rank $\mathbf{R}$
$\mathbf{R} \leftarrow$ schedule$(\mathbf{g}',$ dp_group$)$
// Gather $\mathbf{g}'$ across DP into a full matrix $\mathbf{G}$ to rank $\mathbf{R}$
$\mathbf{G} \leftarrow$ gather$(\mathbf{g}',$ dp_group, dst=$\mathbf{R})$
// Calculate Newton-Schulz only in $\mathbf{R}$
if $\mathbf{r} == \mathbf{R}$ **then**
    $\mathbf{u} \leftarrow$ Newton-Schulz$(\mathbf{G})$
else
    $\mathbf{u} \leftarrow None$
end if
// Scatter a full matrix $\mathbf{u}$ across DP
$\mathbf{u}' \leftarrow$ scatter$(\mathbf{u},$ dp_group, src=$\mathbf{R})$
// Apply DP partitioned $\mathbf{u}'$ to $\mathbf{p}$
$\mathbf{p}' \leftarrow$ apply_update$(\mathbf{p}, \mathbf{u}')$
return $\mathbf{p}'$

We eliminate redundant computation by assigning each parameter to a specific GPU.

However, without proper scheduling, this optimization can lead to poor GPU utilization. In particular, although redundant computation is avoided by assigning each parameter to a specific rank, it causes idle time—since all other ranks must wait for the scatter communication to complete before proceeding.

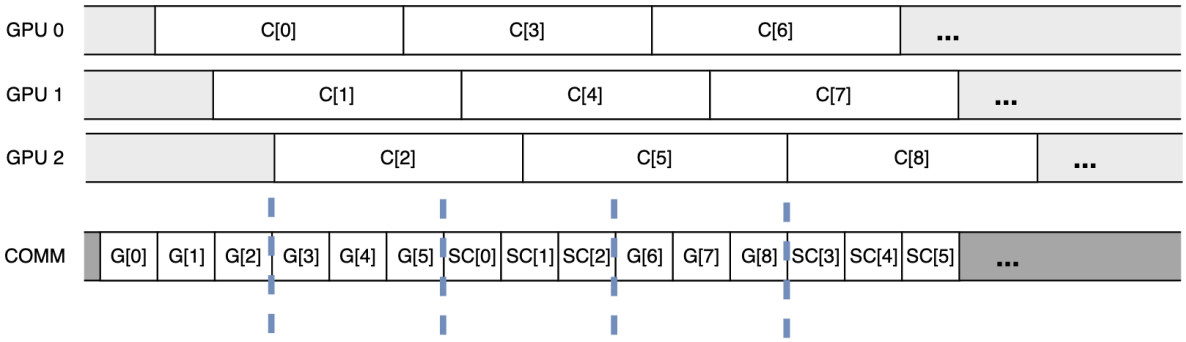


Execution timeline of Parallel Muon

Scheduling Sub-Operations

We can schedule the whole sub-operations as follows, due to the following reasons:

- There are no dependencies between parameters.
- GPUs can execute computation and communication concurrently.



Execution timeline of re-scheduled Parallel Muon

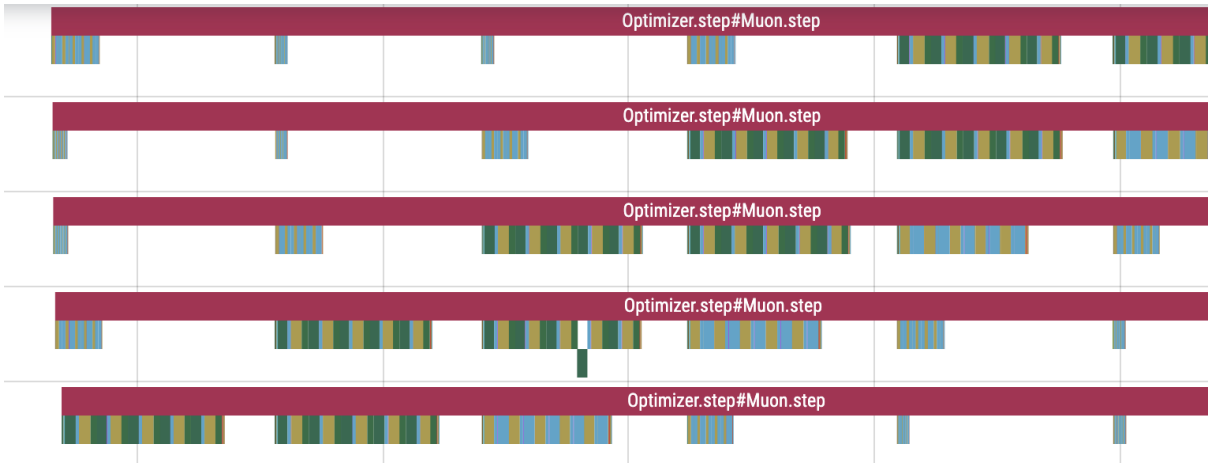
We define the chunk size C as the number of GPUs and schedule each sub-operation in batches of size C. This scheduling allows each GPU to continue computation even while waiting for collective communication to complete.

[Algorithm] (To be written)

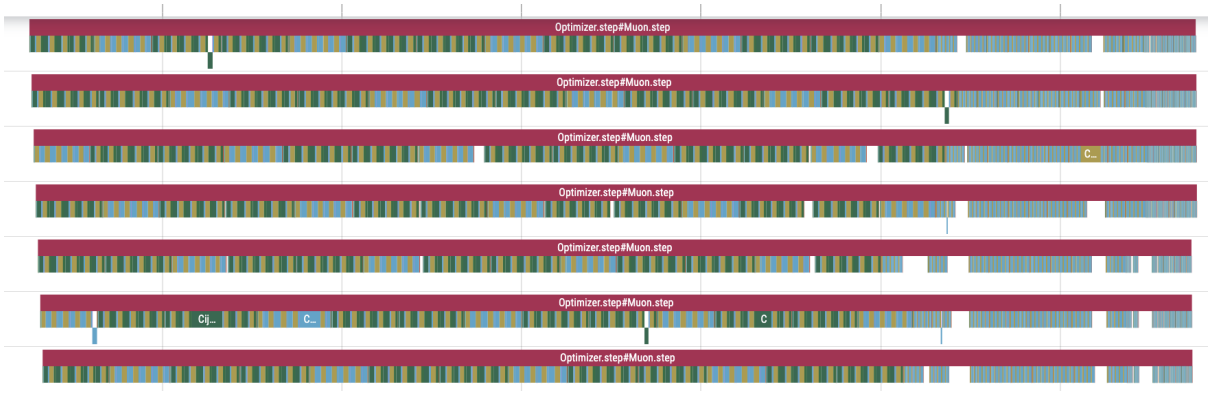
Load Balancing

If parameters in a chunk have imbalanced computation loads, idle bubbles may occur.
To mitigate this, we apply load balancing based on per-parameter FLOPs.

Imbalanced (Round Robin)



After Load Balancing



Implementation

The full implementation is available in `optimizer/torch-ext/optimizer/muon.py`. To enable concurrent computation and communication, we use separate compute and communication streams (`torch.cuda.Stream`) and use `torch.cuda.Event` to synchronize between sub-operations.

Thanks to the simplicity of `torch.DTensor` and `torch.distributed`, the implementation remains straightforward and low in complexity.

Evaluation

We evaluated the performance using 10B model currently in development, achieving 151 TFLOPS per GPU during the optimizer step.

Model Size	TFLOPs for Muon	GPUs	Elapsed time	TFLOPS/GPU
10B	847.45	4xMI250 (8 devices)	1.4 s	151

Based on the breakdown, 7% of the time is attributed to updating sharded gradients and parameters, 78% to GEMM operations, and the remaining 15% to non-overlapped communication overhead.