SimpleFSDP optimization: Automatic Selective Unsharding

Intern: Shilong Lei

Mentor: Shuai Yang

Peers: Xuan Zhang, Yuhang Yang, Francisco Massa, Will Constable, Will Feng

Date: 08/21/2025

TL;DR

In this post, we present results from our initial implementation of the **automatic selective unsharding framework**. Built on top of SimpleFSDP, which is a new effort to implement Fully Sharded Data Parallelism (FSDP) in a PT2 (PyTorch 2.0) compile-friendly way, this intern project introduces an approach that **selectively** retains certain parameters after the forward all gather instead of releasing them, eliminating the need for additional all gathers during backward. By leveraging available free memory, the system reduces communication latency. The behavior can be flexibly tuned via APS front-end configurations, allowing users to exploit spare GPU capacity for more efficient training. Preliminary experiments on a pruned OmniFM_v2 model demonstrate a **QPS improvement of approximately 5.5%**. Overall, this work highlights a new perspective: trading idle memory for reduced communication and improved training efficiency.

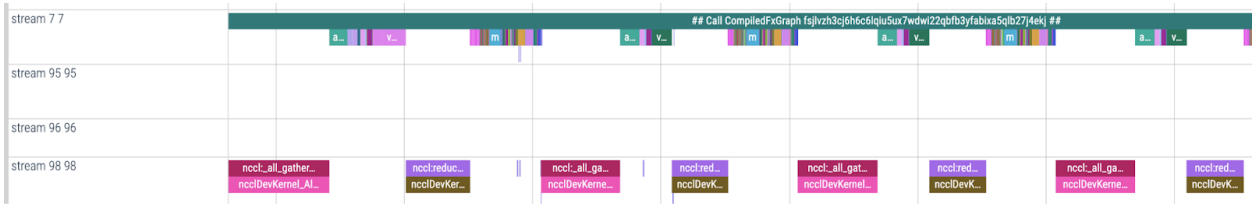
Motivation

Training large-scale models efficiently requires careful coordination of computation and communication. **PyTorch FSDP (Fully Sharded Data Parallelism)** is a widely used framework for distributed training of large models. Recently, **SimpleFSDP** has been introduced as a new effort to implement FSDP in a PT2 (PyTorch 2.0) compile-friendly way. Unlike FSDP, SimpleFSDP leverages PyTorch's native activation checkpointing mechanism to re-implement the way all-gather (AG) and reduce-scatter (RS) operations are launched. This design ensures that AG/RS operations appear explicitly as nodes in the compiled graph, enabling subsequent graph-level optimizations.

To reduce communication overhead, SimpleFSDP employs a **bucketing mechanism**: multiple AG/RS operations are grouped and launched together, since each launch carries a nontrivial

overhead. Grouping operations reduces the number of launches and thus improves efficiency. However, the current implementation uses a **fixed bucket size**, which often leads to suboptimal overlap between communication and computation. As a result, “communication bubbles” appear in the training timeline, degrading overall efficiency. A **tunable bucket size** would allow the system to better align communication with computation workloads and hide these bubbles more effectively.

Fixed AG/RS bucket size:



Another limitation arises from the way SimpleFSDP handles parameter all-gather and release. The current framework offers only two options:

1. **Free-all strategy** — Parameters are all-gathered in every forward pass and immediately freed, forcing them to be re-all-gathered during backward. This minimizes memory usage but incurs redundant communication, leading to inefficiency when additional GPU memory is available.
2. **Keep-all strategy** — Parameters, once all-gathered, are never freed. This eliminates redundant communication but leads to significant memory overhead, which is often impractical for large models.

Neither option provides a good balance between communication efficiency and memory consumption. In practice, when spare GPU memory exists, a more nuanced approach is desirable: selectively retaining certain parameters in memory after the forward pass. This would avoid repeated all-gathers during backward while still controlling memory usage.

This intuition motivates our **Selective Unsharding Framework**, which automatically adapts to the available GPU memory budget. Instead of rigidly applying free-all or keep-all policies, it identifies and retains the subset of parameters whose unsharding yields the highest benefit–cost trade-off. By doing so, the framework reduces redundant communication, better overlaps computation and communication, and ultimately improves overall training efficiency.

Design and Implementation

Configurable Bucket Size

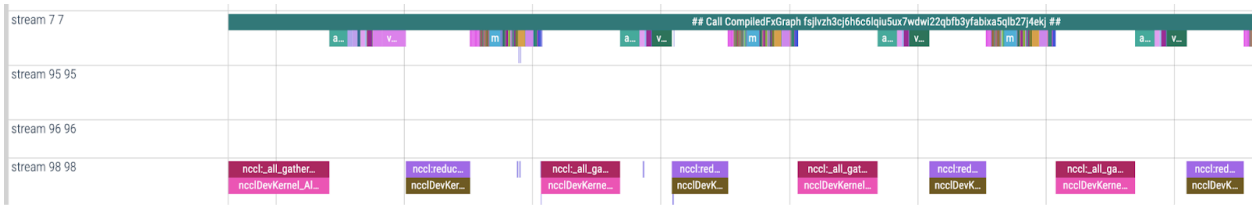
A key mechanism in **SimpleFSDP** is the use of **buckets** to group multiple all-gather (AG) and reduce-scatter (RS) operations together. Since each collective launch incurs a nontrivial overhead, grouping operations amortizes this cost and reduces overall communication overhead. However, using a **fixed bucket size** often results in poor overlap between communication and computation, leaving idle “communication bubbles.”

To address this limitation, we introduce **configurable bucket size**, which allows users to dynamically set the bucket capacity (in MB) for both AG and RS operations. This provides fine-grained control over how communication is scheduled during training.

Example configuration:

```
None
bucket_fsdp_all_gather_concat:
    all_gather_bucket_cap_mb: {"default": 2000, 0: 1000, 2: 1000, 3: 4000}
bucket_fsdp_reduce_scatter_concat:
    reduce_scatter_bucket_cap_mb: {"default": 2000, 1: 1000, 2: 1000, 3: 4000, 6:
1000}
```

Here, different layers (indexed by IDs) are assigned customized bucket capacities, while a default value applies to the rest.



Key benefits:

- **Flexible control:** Users can tailor bucket sizes per layer or keep a global default.
- **Improved efficiency:** By adapting bucket sizes to workload characteristics, communication can be better overlapped with computation.
- **General applicability:** This mechanism works seamlessly with existing SimpleFSDP workflows and integrates naturally into configuration-based front-ends. In the future, it could also be extended to support adaptive schemes that **automatically adjust bucket sizes** during training.

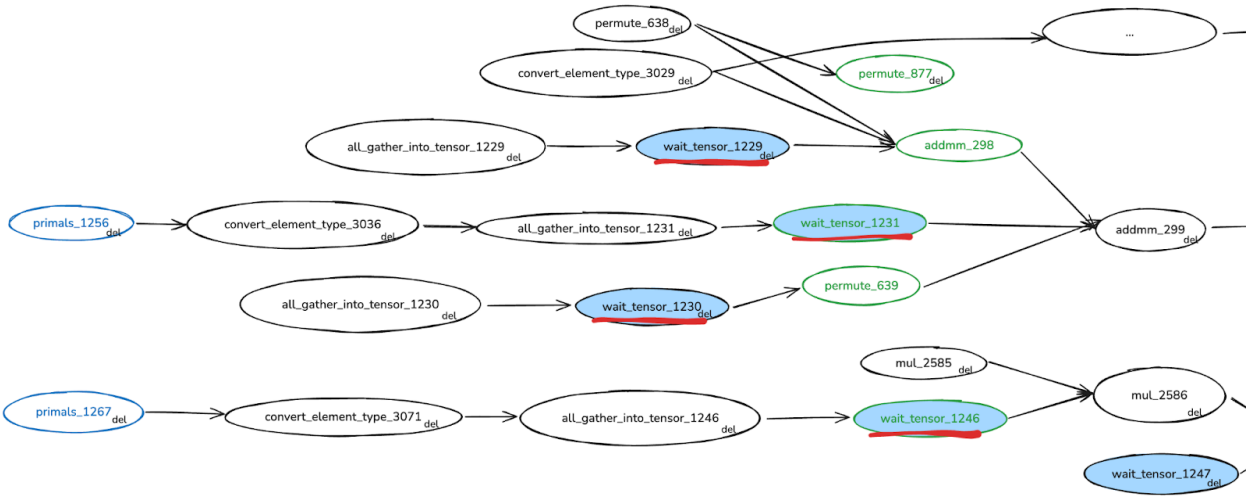
Hierarchical Module Memory Profiling

In SimpleFSDP, optimization passes such as bucketing and unsharding are applied at the compile stage, after the training graph has been generated. Since all-gather (AG) and reduce-scatter (RS) operations appear as **graph nodes**, it becomes difficult to systematically

decide which parameters to unshard: the mapping between graph nodes and their originating modules is not explicitly available (see the following example).

Example: tlparse_aot_joint_graph

```
None
...

# File:
/packages/aps.ads.icvr/icvr_launcher#link-tree/torch/distributed/fb/simple_fsdp/si
mple_fsdp.py:171 in forward, code: out = _AllGather.apply(
       convert_element_type_89: "bf16[56, 2546][2546, 1]cuda:0" =
torch.ops.prims.convert_element_type.default(view_3, torch.bfloat16);  view_3 =
None
       all_gather_into_tensor: "bf16[448, 2546][2546, 1]cuda:0" =
torch.ops._c10d_functional.all_gather_into_tensor.default(convert_element_type_89,
8, '0');  convert_element_type_89 = None
       wait_tensor: "bf16[448, 2546][2546, 1]cuda:0" =
torch.ops._c10d_functional.wait_tensor.default(all_gather_into_tensor);
all_gather_into_tensor = None

        # File:
/packages/aps.ads.icvr/icvr_launcher#link-tree/torch/distributed/fb/simple_fsdp/si
mple_fsdp.py:174 in forward, code: return out[: self.shape_0]
       slice_1: "bf16[448, 2546][2546, 1]cuda:0" =
torch.ops.aten.slice.Tensor(wait_tensor, 0, 0, 448);  wait_tensor = None

        # File:
/packages/aps.ads.icvr/icvr_launcher#link-tree/torch/distributed/fb/simple_fsdp/si
mple_fsdp.py:170 in forward, code: x = x.to_local()  # convert DTensor to local
tensor
       view_4: "f32[56][1]cuda:0" = torch.ops.aten.view.default(primals_138,
[56]);  primals_138 = None

        # File:
/packages/aps.ads.icvr/icvr_launcher#link-tree/torch/distributed/fb/simple_fsdp/si
mple_fsdp.py:171 in forward, code: out = _AllGather.apply(
       convert_element_type_90: "bf16[56][1]cuda:0" =
torch.ops.prims.convert_element_type.default(view_4, torch.bfloat16);  view_4 =
None
       all_gather_into_tensor_1: "bf16[448][1]cuda:0" =
torch.ops._c10d_functional.all_gather_into_tensor.default(convert_element_type_90,
8, '0');  convert_element_type_90 = None
       wait_tensor_1: "bf16[448][1]cuda:0" =
torch.ops._c10d_functional.wait_tensor.default(all_gather_into_tensor_1);
all_gather_into_tensor_1 = None

…
```



Challenges:
To effectively leverage surplus GPU memory for selective unsharding, we require detailed knowledge of:

- The correspondence between modules and their associated graph nodes.
- The memory footprint of these nodes.

Without this information, unsharding decisions cannot be made in an organized or memory-aware manner.

To address these challenges, we introduce **Hierarchical Module Memory Profiling**, implemented in *tlparse*. This mechanism records both the module-to-node mapping and the memory footprint of each node. It also supports an *"only_wait_tensors"* mode, which restricts profiling to `all_gather_into_tensor` nodes—those that can be unsharded to save memory. This mode provides valuable insights for guiding **selective unsharding**.

Here is an example of our tailored hierarchical module memory profile:

```
None
- [root] 11032.45 MB, 850 nodes
  - [uniarch_layers] 11032.45 MB, 850 nodes
    - [0] 4301.51 MB, 311 nodes
      - ...
    - [1] 3195.27 MB, 219 nodes
      - [seq_summarization_layer] 8.44 MB, 18 nodes
        - ...
      - [non_seq_interaction_module] 3186.82 MB, 201 nodes
        - ...
    - [2] 3535.67 MB, 320 nodes
      - [seq_interaction_layer] 1842.91 MB, 73 nodes
```

```
  - ...
- [seq_summarization_layer] 18.14 MB, 54 nodes
  - ...
- [non_seq_interaction_module] 1674.63 MB, 193 nodes
  - [layers] 1674.63 MB, 193 nodes
    - [0] 1593.41 MB, 100 nodes
    - [1] 81.21 MB, 93 nodes
      - [horizontal_dhen] 81.20 MB, 92 nodes
  - ...
```

This profiling infrastructure also lays the groundwork for **adaptive unsharding policies**, where memory profiling information could be used to dynamically adjust which modules are unsharded during training, achieving better cost–benefit trade-offs as runtime conditions change.

Selective Unsharding Framework

Overview:
A central limitation of existing SimpleFSDP is its binary choice between freeing all parameters after every forward pass (*free-all*) or keeping all parameters throughout training (*keep-all*). The former incurs redundant all-gathers during backward, while the latter leads to high memory overhead. To address this, we introduce the **Selective Unsharding Framework**, which provides a more flexible and memory-aware strategy. Here is a detailed document for the framework.

The framework supports two complementary modes:

- **Manual Mode.** Users explicitly specify which modules should be unsharded, offering full control over memory–communication trade-offs.
- **Automatic Mode.** The system automatically selects modules to unshard based on user-defined granularity and memory constraints, making better use of surplus GPU memory without requiring manual intervention.

Configuration format:
The framework is integrated into the APS configuration system, making it straightforward to adopt. An example configuration is shown below:

```
None
selective_unshard_all_gather:
  automatic: true
  granularity: 200
  hierarchical_module_memory_profile:
```

```
    only_wait_tensors: false
  unshard_fsdp_top_modules:
    domain_experts.shared.uniarch:   # FSDP top-level module1
      memory_limit: 2500
      target_modules:
        - uniarch_layers.1
        - uniarch_layers.2
    domain_experts2.shared.uniarch:  # FSDP top-level module2
      memory_limit: 5000
      target_modules:
        - uniarch_layers.1
```

Here, users can define **memory budgets** (memory_limit) for each FSDP top-level module and specify **target submodules** for selective unsharding. When automatic mode is enabled, the system leverages hierarchical module memory profiling to choose the optimal set of parameters to retain, while respecting user-specified memory limits and granularity.

Benefits:
This framework provides:

- **Fine-grained control** over parameter unsharding.
- **Automatic adaptation** to available GPU memory.
- **Improved training efficiency** by reducing redundant all-gathers while bounding memory overhead.

manual mode

In **manual mode**, users can explicitly specify which modules to unshard. This is configured by setting `automatic: false` under `selective_unshard_all_gather`. In this mode, the system bypasses any automatic selection logic and it will **directly unshard the modules listed by the user** without checking their sizes, memory footprints, or ordering. As a result, the behavior is entirely **user-defined** and deterministic.

Example configuration:

```
None
selective_unshard_all_gather:
  automatic: false
  unshard_fsdp_top_modules:
    domain_experts.shared.uniarch: # FSDP top-level module1
      target_modules:
        - uniarch_layers.2
```

```
    domain_experts2.shared.uniarch: # FSDP top-level module2
      target_modules:
        - uniarch_layers.3.seq_interaction_layer
        - uniarch_layers.3.seq_summarization_layer
```

This example demonstrates that `uniarch_layers.2` under the first FSDP top-level module and two submodules (`seq_interaction_layer`, `seq_summarization_layer`) under the second will be unsharded as specified.

automatic mode

In **automatic mode** (`automatic: true`), the system automatically determines which modules to unshard based on the given **memory limit** and **granularity**. Instead of relying on explicit user-specified modules, the algorithm traverses the target modules (default: all submodules) in reverse order. The selection proceeds greedily: modules are unsharded if doing so does not exceed the assigned memory budget.

Example configuration:

```
None
selective_unshard_all_gather:
  automatic: true
  granularity: 100
  unshard_fsdp_top_modules:
    domain_experts.shared.uniarch: # FSDP top-level module1
      memory_limit: 1500
      target_modules:
        - uniarch_layers.2
    domain_experts2.shared.uniarch: # FSDP top-level module2
      memory_limit: 1000
      target_modules:
        - uniarch_layers.3.seq_interaction_layer
        - uniarch_layers.3.seq_summarization_layer
```

Algorithm (recursive unsharding):

1. **Reverse iteration**: Traverse modules from last to first, prioritizing later stages of the forward pass.
2. **Fit check**: If a module fits within the current budget, unshard it and accumulate its size.

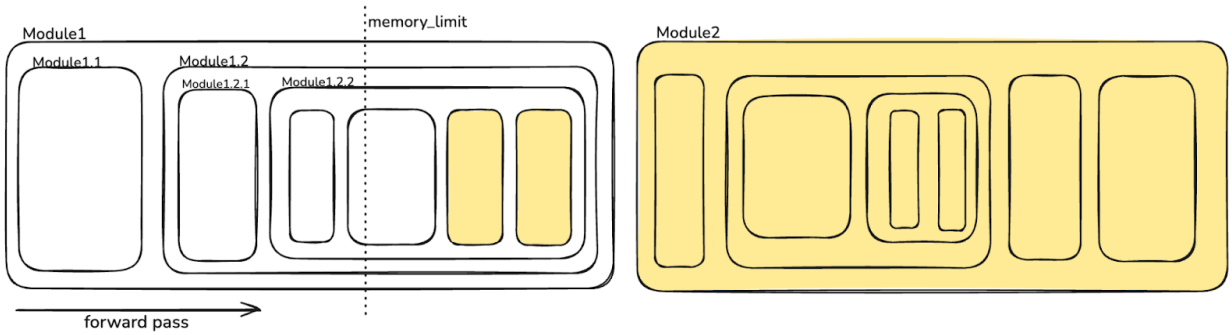
3. **Splitting**: If the module is too large but meets the `granularity` threshold and has submodules, recurse into its children.
4. **Early stop**: If the module is too small to split, or splitting still cannot fit within the budget, stop scanning.

Psuedo-code:

```
None
Algorithm RECURSIVE-UNSHARD(modules, memory_limit, granularity_size, total=0):
    for module in REVERSE(modules):
        if total + SIZE(module) ≤ memory_limit:
            UNSHARD(module)
            total ← total + SIZE(module)

        else if SIZE(module) ≥ granularity_size AND HAS_SUBMODULES(module):
            total ← RECURSIVE-UNSHARD(SUBMODULES(module),
                                      memory_limit, granularity_size, total)
            return total          // early exit after attempting a split

        else:
            return total          // too small to split; stop scanning

    return total
```

Detailed version of psuedo-code is P1874435908 for reference.

Illustrative example:
The figure below shows an example of automatic unsharding applied to a FSDP top-level module with 2 submodules named Module1 and Module2.



- The system first considers **Module2**. Since its size does not exceed the memory limit, it is unsharded entirely (highlighted in yellow).
- Next, **Module1** is examined. As its full size exceeds the budget, the algorithm checks whether it can be split.
    - **Module1.2** is explored recursively, starting from its last submodule.

    - The process continues until reaching **Module1.2.2**. At this point, unsharding further would exceed the memory limit, and no further splitting is possible. The algorithm stops here.

Another detailed example is EX511239 for reference.

This greedy, depth-first approach ensures that memory is utilized effectively while respecting user-specified constraints such as `memory_limit` and `granularity`.

Benefits:

- **No expertise required** – Manual unsharding needs deep knowledge of the model; auto mode makes optimal choices automatically, helping non-expert users.
- **Memory-aware** – Adapts to `memory_limit`, ensuring efficient use of available GPU memory.
- **Granularity support** – Splits large modules into submodules when needed, avoiding waste.
- **Efficient strategy** – Greedy reverse traversal prioritizes later stages, improving communication–memory tradeoffs.

Experiments

Experiment Settings

All experiments for both **manual mode** and **automatic mode** were conducted under the same setup:

- Hardware: 8 × GPUs (GRANDTETON)
- Model: OmniFM v2
- Batch size: 128
- Number of batches: 1000

Selective unsharding framework - manual mode

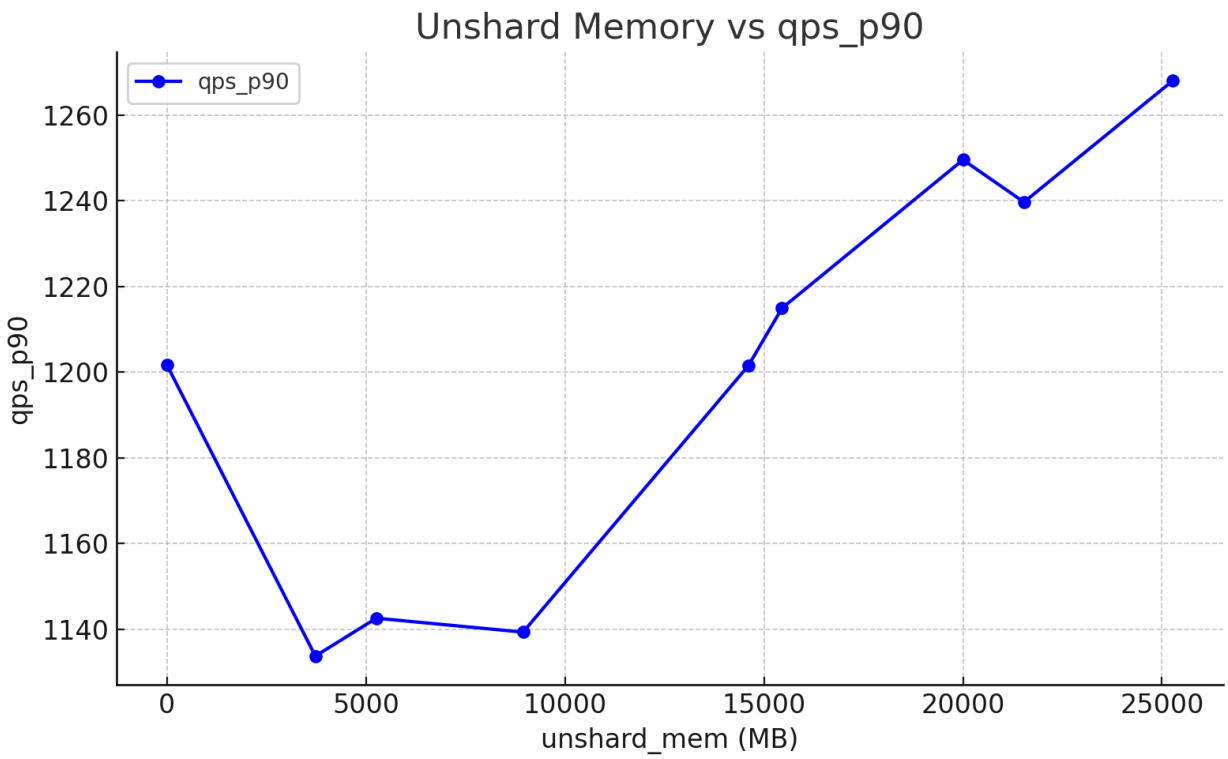
In this experiment, we **manually unshard progressively more modules from back to front**. The rationale for choosing the later modules first is that unsharding them only increases memory usage for a shorter duration of time, thus yielding better trade-offs.

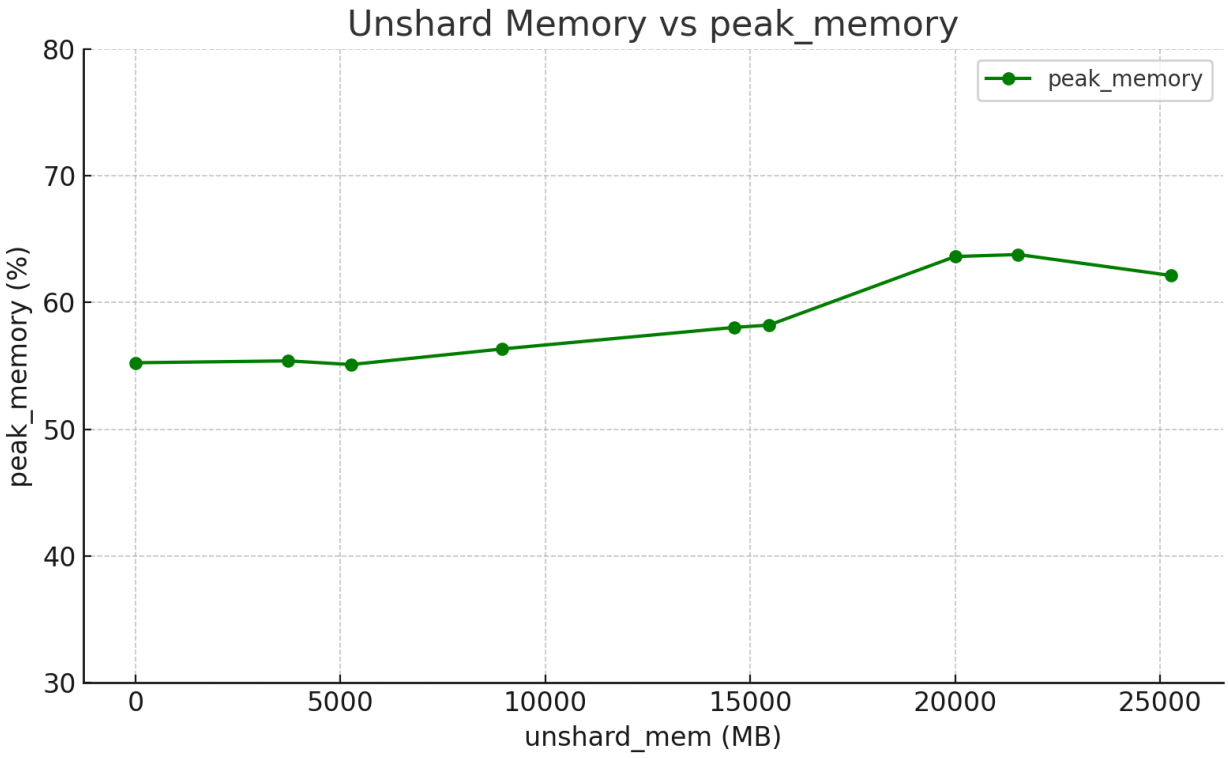
We measured **memory usage** and **QPS performance** under different unsharding configurations. The results are summarized below:

job	layers	unshard_	avg/peak_	qps_p90	qps_p90_r	qps_lifeti	qps_max	qps_p50

		mem	mem		el(%)	me		
7.80E+07	no	0	40.55% / 55.24%	1201.69	0	1171.14	1208.5	1193.38
ad17	last → 3.3	3728.7	38.85% / 55.39%	1133.78	-5.65	1115.17	1134.85	1125.7
f5d8	last → 3.2	5260.79	39.00% / 55.10%	1142.62	-4.92	1118.04	1146.53	1133.4
052b	last → 3.1	8947.8	39.39% / 56.33%	1139.36	-5.19	1123	1142.68	1128.26
b5fc	last → 2.2	14608.66	39.63% / 58.03%	1201.54	-0.01	1173.17	1201.83	1191.55
df03	last → 2.1	15453.2	40.25% / 58.21%	1214.96	1.1	1182.31	1216.45	1194.46
1.00E+43	last → 1.3	20000.88	42.00% / 63.62%	1249.58	3.98	1218.84	1256.76	1230.46
ba72	last → 1.2	21532.97	42.38% / 63.78%	1239.71	3.16	1198.92	1240.87	1227.61
b2dd	last → 1.1	25274.55	41.46% / 62.12%	1268.07	5.52	1224.48	1277.65	1250.7

We then plot the results as line charts to observe how **QPS changes** and **GPU memory usage** relative to **total unshard size**:


Unshard Memory vs qps_p90
qps_p90
1260
1240
1220
1200
1180
1160
1140
qps_p90
0
5000
10000
15000
20000
25000
unshard_mem (MB)


Unshard Memory vs peak_memory
peak_memory
80
70
60
50
40
30
peak_memory (%)
0
5000
10000
15000
20000
25000
unshard_mem (MB)

As the total unshard size on the x-axis increases, QPS shows an upward trend while GPU memory usage also grows. This highlights a clear trade-off, and suggests that fully utilizing the available memory could unlock even greater QPS improvements.

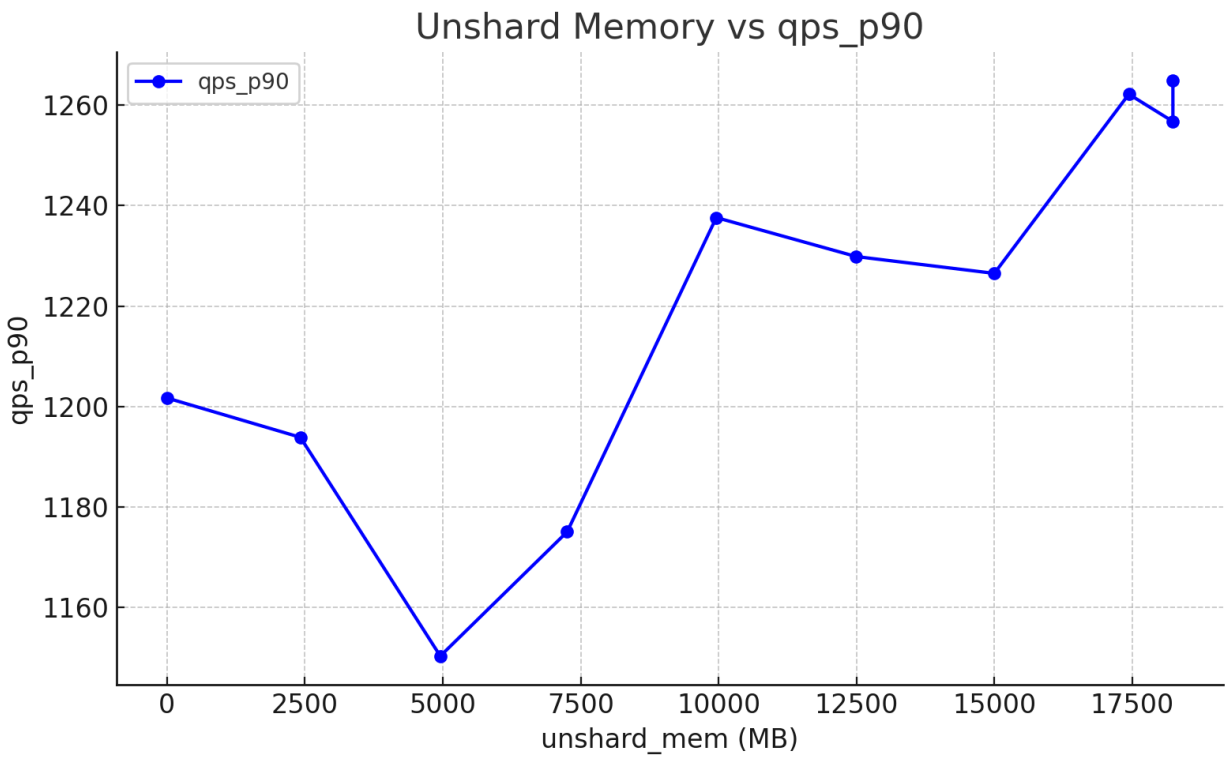
Selective unsharding framework - auto mode

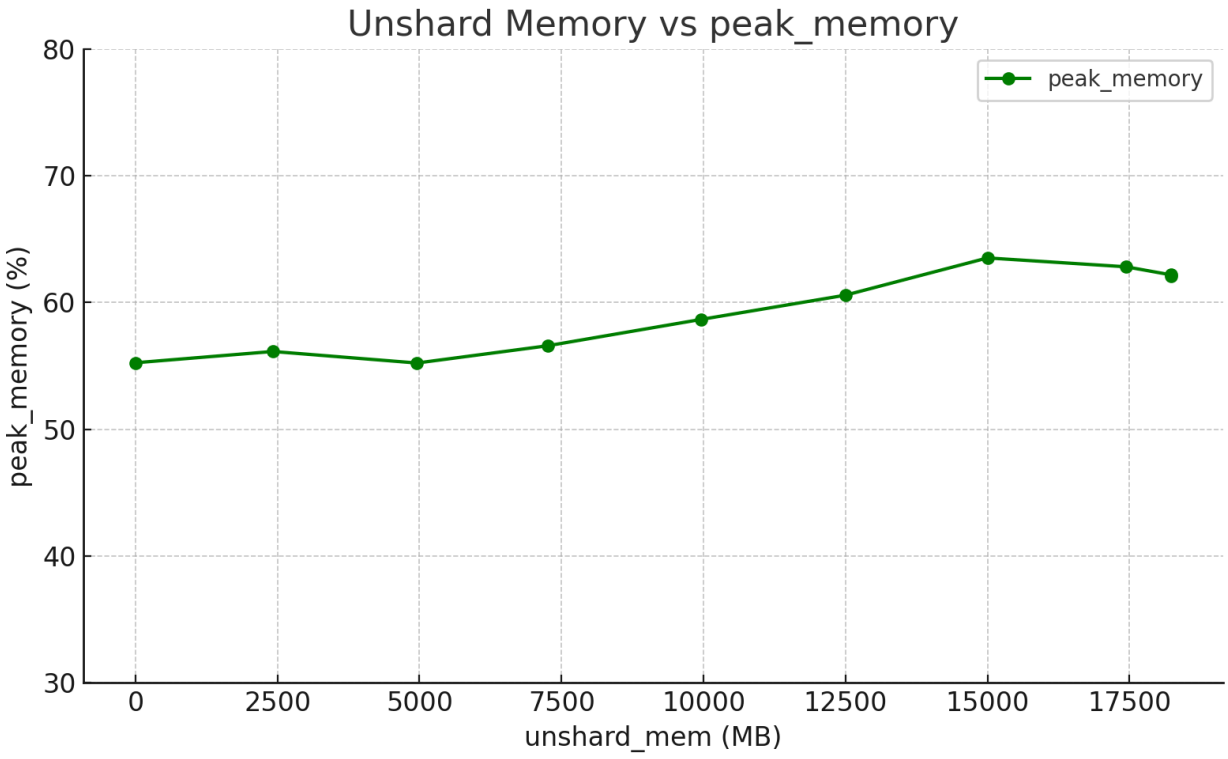
In this experiment, we enabled **automatic mode** and varied the `memory_limit` parameter from low to high. The system then greedily unsharded submodules in reverse order, up to the specified memory budget.

We measured **GPU memory usage** and **QPS performance** under different memory limits. The results are summarized below:

job	memory_limit	unshard_mem	avg/peak	qps_p90	qps_p90_rel(%)	qps_lifetime	qps_max	qps_p50
7.80E+07	0	0	40.55% / 55.24%	1201.69	0	1171.14	1208.5	1193.38
0f16	2500	2418.66	40.98% / 56.13%	1193.86	-0.65	1159.04	1198.39	1180.21
e511	5000	4955.78	40.91% / 55.22%	1150.29	-4.28	1127.88	1155.18	1134.84
93ba	7500	7255.63	40.81% / 56.58%	1175.04	-2.22	1148.79	1194.26	1160.36
6629	10000	9959.47	42.46% / 58.66%	1237.63	2.99	1181.41	1259.02	1214.5
a38e	12500	12496.09	41.86% / 60.57%	1229.86	2.34	1195.2	1239.68	1202.3
b2f4	15000	14999.88	43.14% / 63.51%	1226.51	2.06	1198.98	1228.88	1216.83
c8af	17500	17438.94	42.41% / 62.81%	1262.2	5.04	1216.87	1263.2	1255.11
3.00E+80	20000	18237.46	41.93% / 62.17%	1256.76	4.58	1216.89	1263.6	1243.71
7.30E+08	22500	18237.46	40.56% / 62.09%	1264.91	5.26	1233.69	1267.32	1256.04

We then plot the results as line charts to observe how **QPS changes** relative to **memory usage** under different memory limits.





As the memory limit increases, unshard size grows accordingly, leading to higher GPU memory usage. QPS generally improves with larger memory budgets, although the trend is not strictly

monotonic. This again highlights the trade-off, and shows that leveraging more free memory has the potential to unlock greater QPS gains. Moreover, while manually choosing the right modules to unshard requires expertise, automatic mode handles this process seamlessly, making it easier for non-expert users.

Discussion and Future Work

This work demonstrates the effectiveness of **selective unsharding** in improving training efficiency, but several directions remain for further exploration.

Efficiency Improvements

In our tailored model with 8 × GPUs, selective unsharding achieves approximately **5.5% QPS gain**. While encouraging, this result is limited to a single model. Future evaluations on a **wider spectrum of models** are necessary. We expect that larger and longer models will benefit more significantly, as they tend to expose greater communication overhead.

Dynamic Bucket Size

Currently, bucket sizes are fixed. More advanced strategies can further optimize performance:

- **Auto bucketing**: automatically determine appropriate bucket sizes before training.
- **Adaptive bucketing**: dynamically adjust bucket sizes during training to respond to runtime memory and communication conditions.

Smarter Unshard Policy

The current system employs a **greedy reverse traversal strategy**, unsharding parameters from the tail backward. This assumes that later modules are more critical to unshard. While simple and effective, future work could explore **smarter policies** that account for the **cost–benefit ratio** of each parameter—for example, prioritizing parameters whose all-gathers lie on the **critical path** of communication, i.e., those ‘exposed’ communications that directly impact performance.

Unexpected Regressions

In some cases above, partial unsharding can lead to **worse performance**. A likely reason is that certain all-gather operations are not actually exposed; thus, reducing them does not improve QPS. A more nuanced understanding of which all-gathers contribute to latency is needed to avoid such regressions.

Outlook

Looking ahead, we envision selective unsharding evolving beyond a standalone optimization. It could be integrated with compiler-level graph optimizations, scheduling strategies, or runtime memory managers, enabling tighter coordination between computation, communication, and memory usage. Such integration would open the door to **systematically leveraging idle memory** for communication reduction, ultimately pushing the efficiency frontier of large-scale training. More broadly, it provides **a perspective of trading idle memory for compute efficiency**, offering a new dimension to improve overall training efficiency.