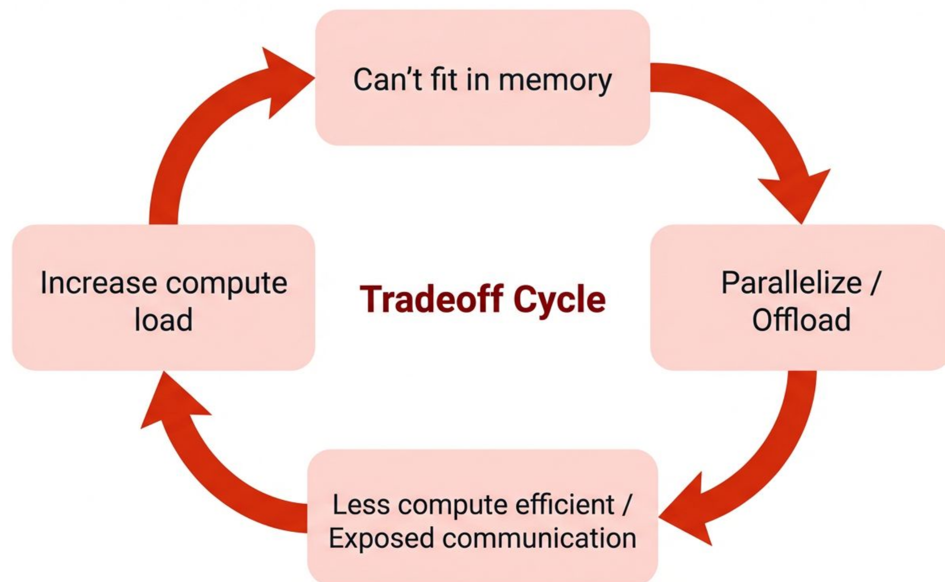
Graph based Pipeline Parallelism

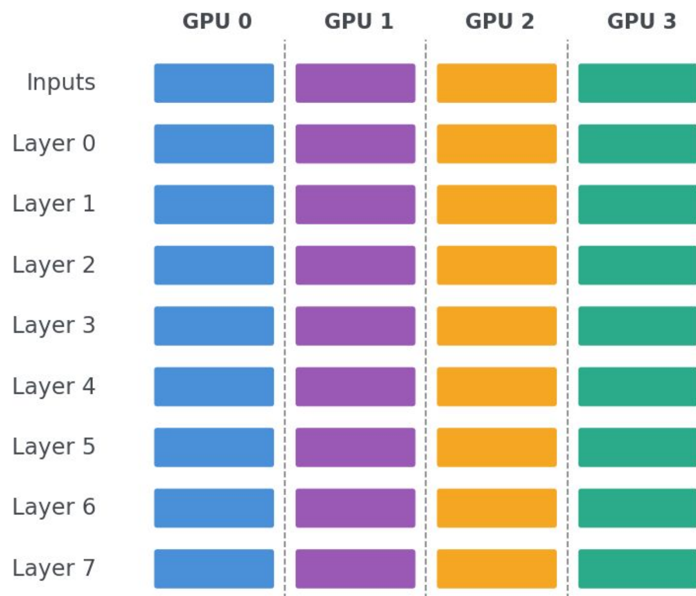
Sanket Purandare, Simon Fan
Meta Superintelligence Labs

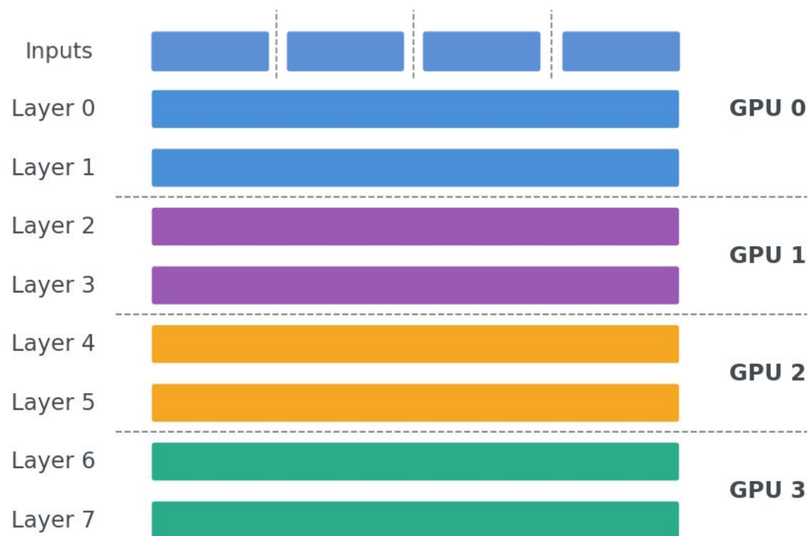
Models have become increasingly large

Parallelisms introduces a loop of challenges for efficient training.

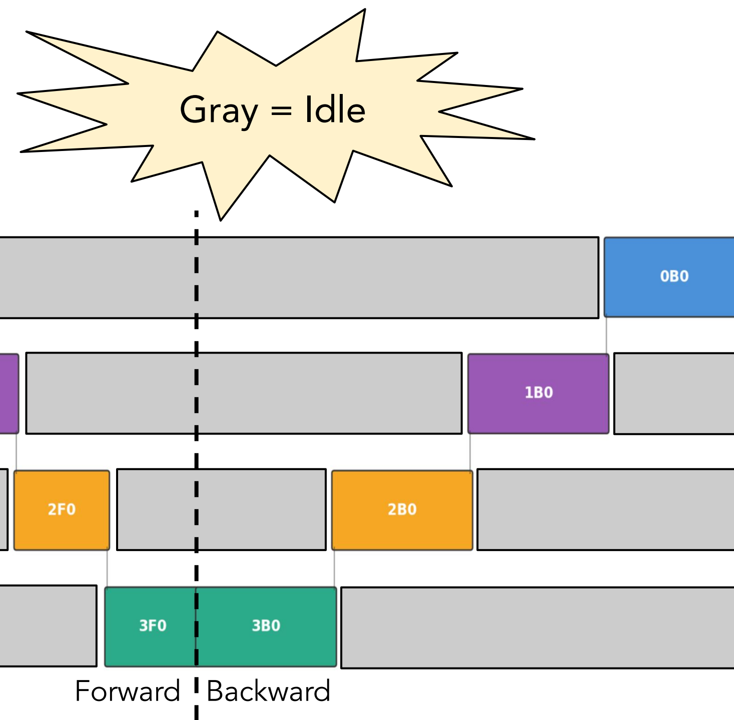


The promise of Pipeline Parallelism





New inefficiency: Pipeline Bubbles



Schedule: ZB2P (Zero-bubble PP w/ 2x pending backward)

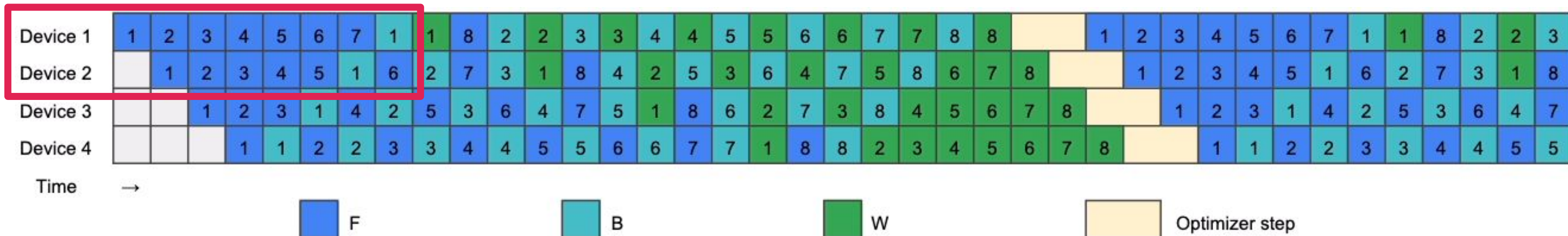
Device 1
Device 2
Device 3
Device 4
Time →

F
B
W
Optimizer step

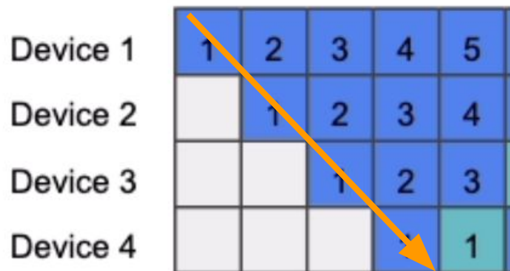
The smaller the microbatch
the faster later devices can run

Device 1
Device 2
Device 3
Device 4

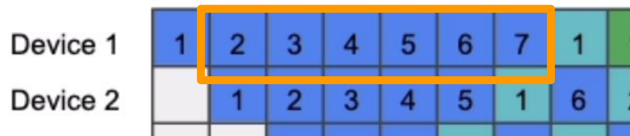
Schedule: ZB2P (Zero-bubble PP w/ 2x pending backward)



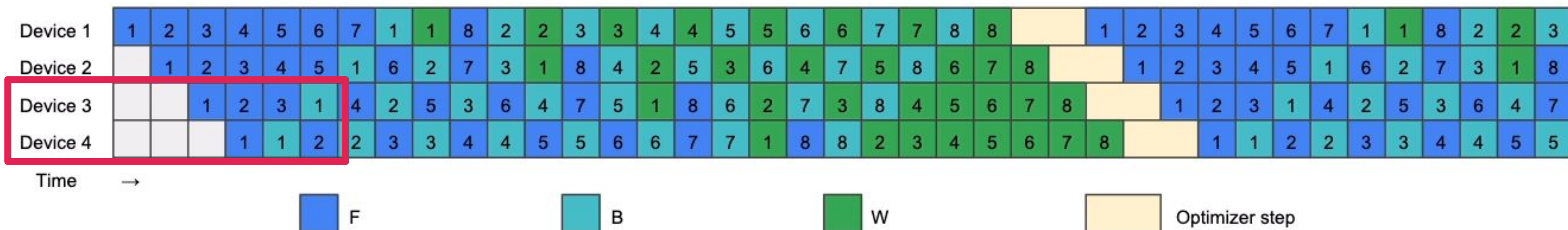
The smaller the microbatch
the faster later devices can run



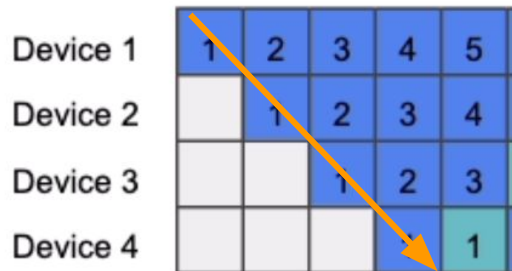
Fill gaps with
more microbatches



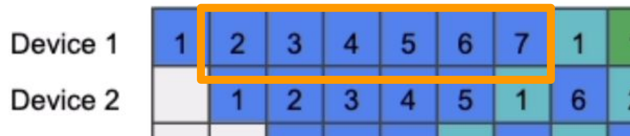
Schedule: ZB2P (Zero-bubble PP w/ 2x pending backward)



The smaller the microbatch
the faster later devices can run



Fill gaps with
more microbatches



Device 4 prioritizes
unblocking Device 3
by delaying F2

Schedule: ZB2P (Zero-bubble PP w/ 2x pending backward)

Device 1: 1 2 3 4 5 6 7 1 1 8 2 2 3 3 4 4 5 5 6 6 7 7 8 8 1 2 3 4 5 6 7 1 1 8 2 2 3
Device 2: 1 2 3 4 5 1 6 2 7 3 1 8 4 2 5 3 6 4 7 5 8 6 7 8 1 2 3 4 5 1 6 2 7 3 1 8
Device 3: 1 2 3 1 4 2 5 3 6 4 7 5 1 8 6 2 7 3 8 4 5 6 7 8 1 2 3 1 4 2 5 3 6 4 7
Device 4: 1 1 2 2 3 3 4 4 5 5 6 6 7 7 1 8 8 2 3 4 5 6 7 8 1 1 2 2 3 3 4 4 5 5

Time →

F B W Optimizer step

Forward

Backward

x

W

Wx

$N\times$

z

$\sigma(z)$

y

$\nabla_x L$

$W^T \nabla_z L$

$\nabla_z L$

$\frac{d\sigma(z)}{dz} \nabla_y L$

$\nabla_y L$

$\nabla_z L$

$\nabla_z L x^T$

$\nabla_W L$

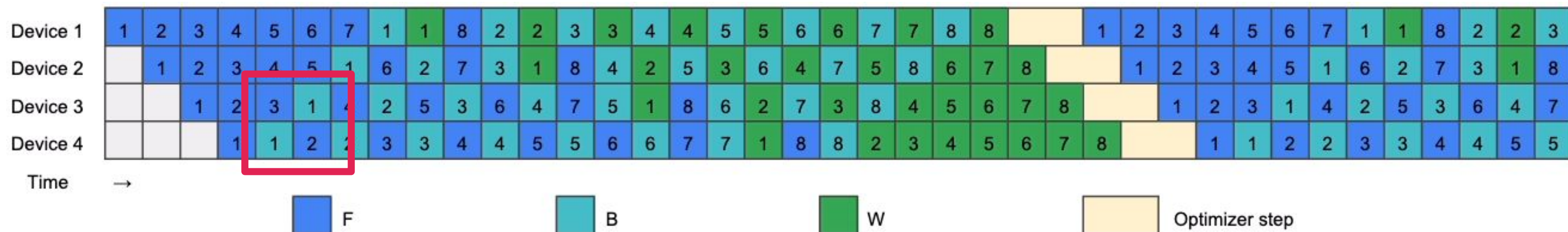
$\times N$

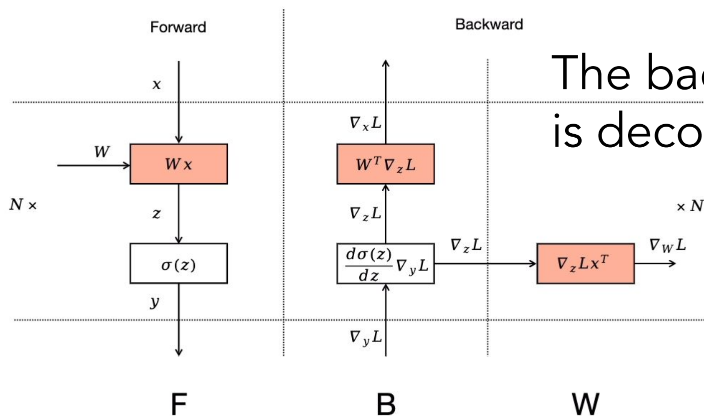
F B W

The backward pass is decomposed

PyTorch CONFERENCE EUROPE 2026

Schedule: ZB2P (Zero-bubble PP w/ 2x pending backward)





The backward pass is decomposed

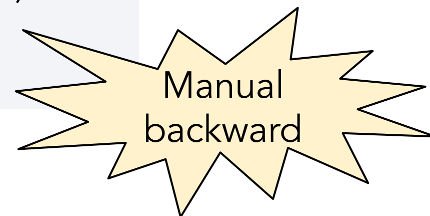
Device 3's B1 only depends on Device 4's B1, not its W1

PyTorch CONFERENCE EUROPE 2026

Authoring backward decomposition

sail-sg/zero-bubble-pipeline-parallelism (forked from Megatron)

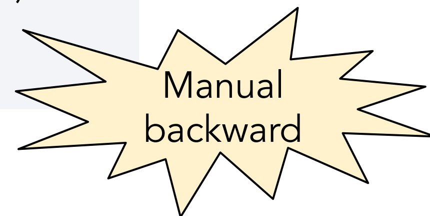
```
class LinearWithGradAccumulationAndAsyncCommunication(torch.autograd.Function):
    def backward(ctx, grad_output):
        <delay W's work and return B>
```



Authoring backward decomposition

sail-sg/zero-bubble-pipeline-parallelism (forked from Megatron)

```
class LinearWithGradAccumulationAndAsyncCommunication(torch.autograd.Function):
        def backward(ctx, grad_output):
                <delay W's work and return B>
```



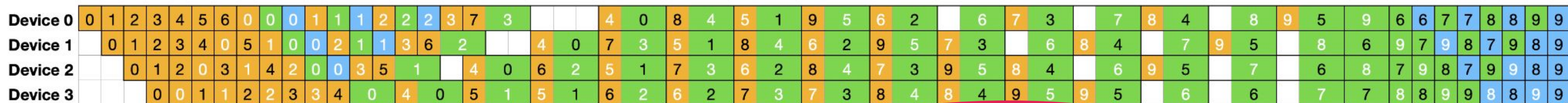
torch.distributed.pipelining

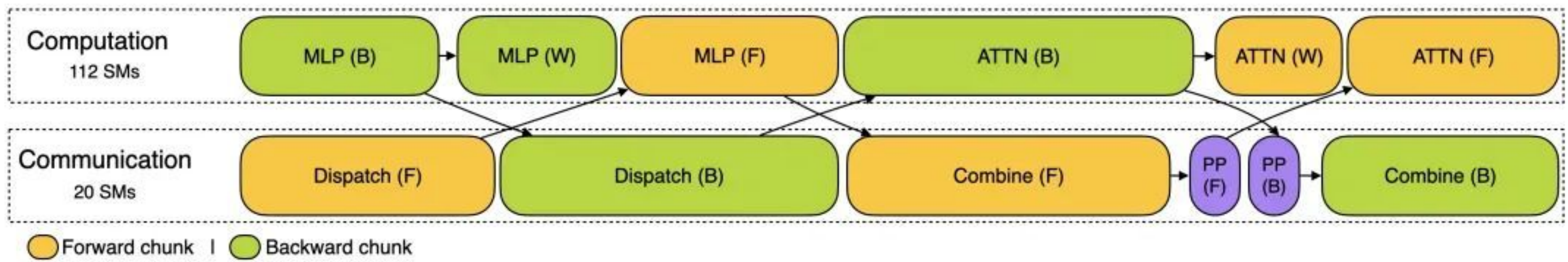
```
dinputs = torch.autograd.grad(
        stage_outputs_or_loss,
        inputs=input_values,
        grad_outputs=output_grad,
        retain_graph=True,
)
```

```
dweights = torch.autograd.grad(
        valid_edges,
        inputs=weights_edges,
        grad_outputs=valid_grad_outputs,
        retain_graph=retain_graph,
)
```

Difficulties composing with:
- torch.compile
- checkpointing

Schedule: DualPipeV (Specifically for MoEs)





Authoring overlapped forward backward

Python thread 1 (Autograd)

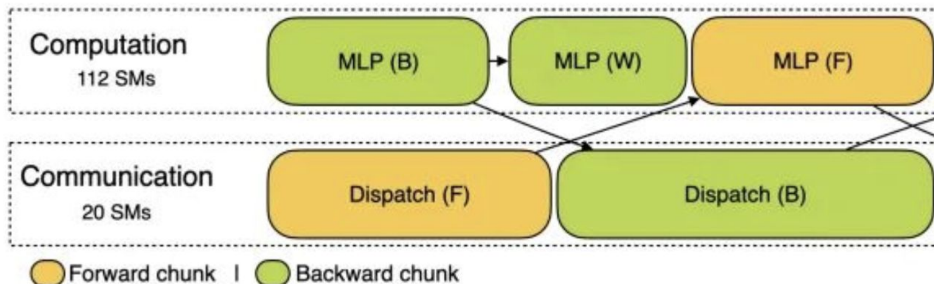
1. Compute: MLP (B)
2. Comm: Dispatch (B)
3. Compute: MLP (W)
4. …

Dispatch (F) then Dispatch (B)

MLP (W) then MLP (F)

Python thread 2 (Forward)

1. Comm: Dispatch (F)
2. Compute: MLP (F)
3. …



Coordinating fine-grained overlap across nn.Module boundaries is difficult.

deepseek-ai/Dualpipe requires model author to define their manual backward.

PyTorch CONFERENCE EUROPE 2026

Pain points of authoring advanced schedules

Limited control over backward

For full flexibility, you need to write manual backward.

Thread/streams management

In order to achieve Dualpipe-like fine-grained overlap.

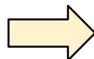
Composability challenges

High complexity model code with lots of branching.
Incompatibility with activation checkpointing APIs, torch.compile.

Graph based Pipeline Parallelism

Limited control over backward

For full flexibility, you need to write manual backward.

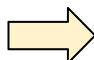


Full control over backward

Provided by functional graph capture API and AOTAutograd.

Thread/streams management

In order to achieve Dualpipe-like fine-grained overlap.

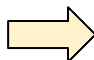


No threads management

The overlap schedule can run off the main thread.
Optional streams management.

Composability challenges

High complexity model code with lots of branching.
Edge case scenarios with activation checkpointing, torch.compile.



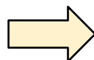
Compiler-first

No model code changes required once traceable.
Fine-grained control over activations.
Provides out of the box performance with Inductor.

Graph based Pipeline Parallelism

Limited control over backward

For full flexibility, you need to write manual backward.

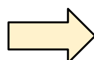


Full control over backward

Provided by functional graph capture API and AOTAutograd.

Thread/streams management

In order to achieve Dualpipe-like fine-grained overlap.

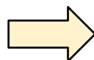


No threads management

The overlap schedule can run off the main thread.
Optional streams management.

Composability challenges

High complexity model code with lots of branching.
Edge case scenarios with activation checkpointing, torch.compile.



Compiler-first

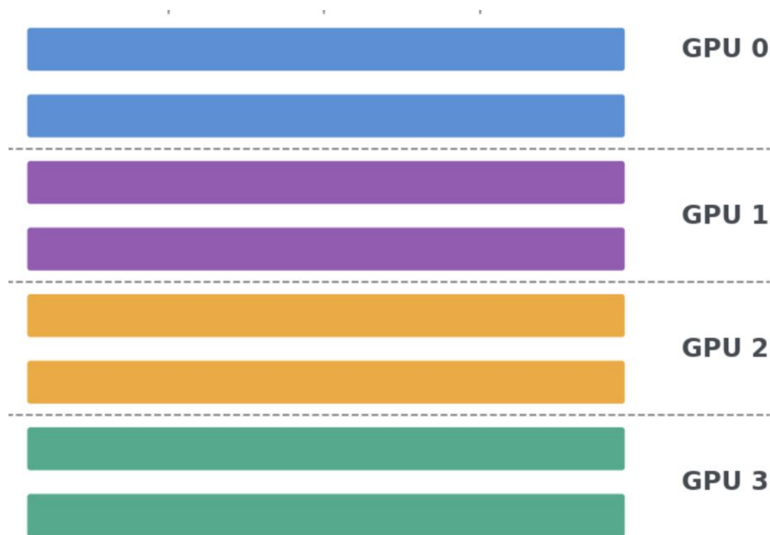
No model code changes required once traceable.
Fine-grained control over activations.
Provides out of the box performance with Inductor.

Highly customizable

All optimizations can be written as graph passes with full user control using annotations
Can adjust schedules to fit your specific workload needs

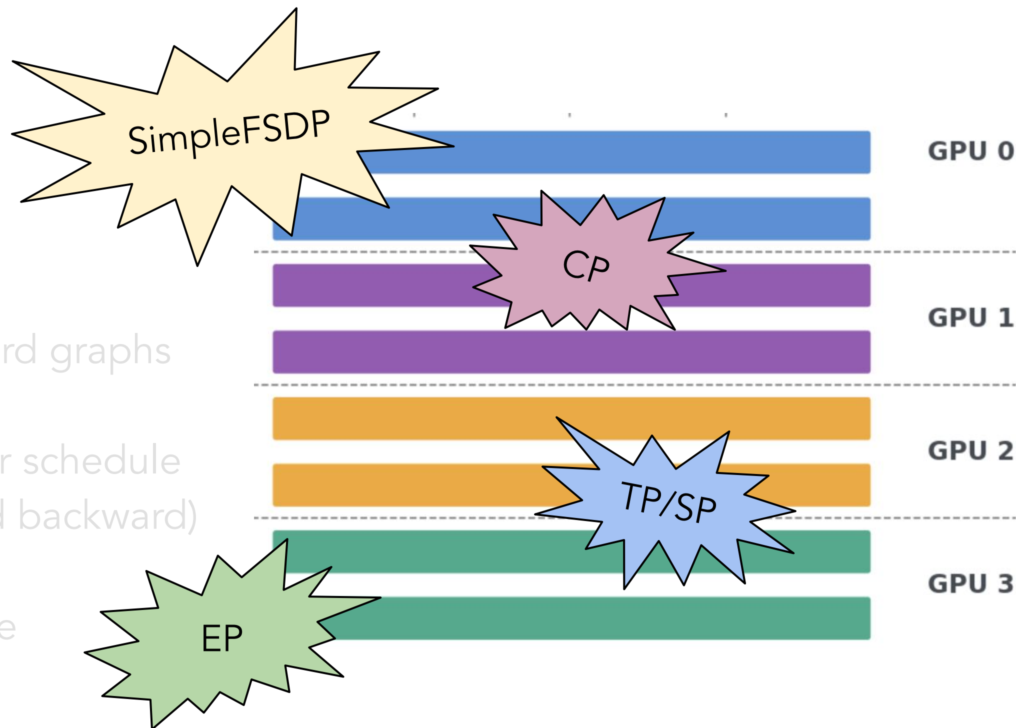
Authoring via Graph based PP

1. Split model into stages
2. Parallelize each stage
3. Compile to obtain Forward/Backward graphs
4. Apply graph transforms required for schedule (e.g. B/W split, Overlapped forward backward)
5. Integrate into your trainer or use the torch.pipelining runtime



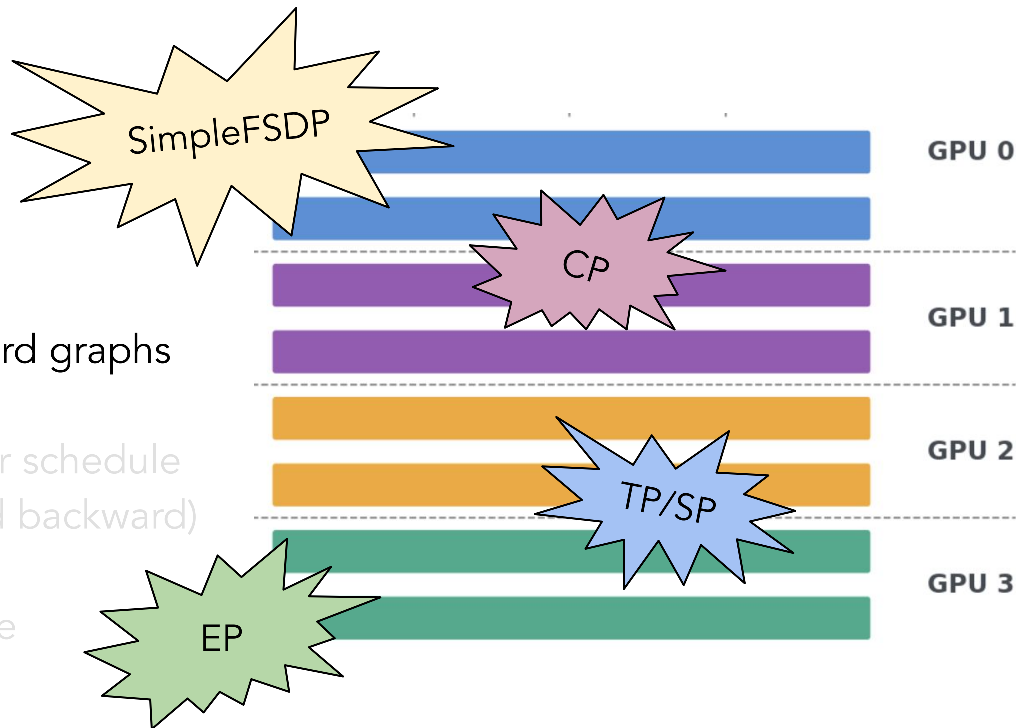
Authoring via Graph based PP

1. Split model into stages
2. Parallelize each stage
3. Compile to obtain Forward/Backward graphs
4. Apply graph transforms required for schedule (e.g. B/W split, Overlapped forward backward)
5. Integrate into your trainer or use the torch.pipelining runtime



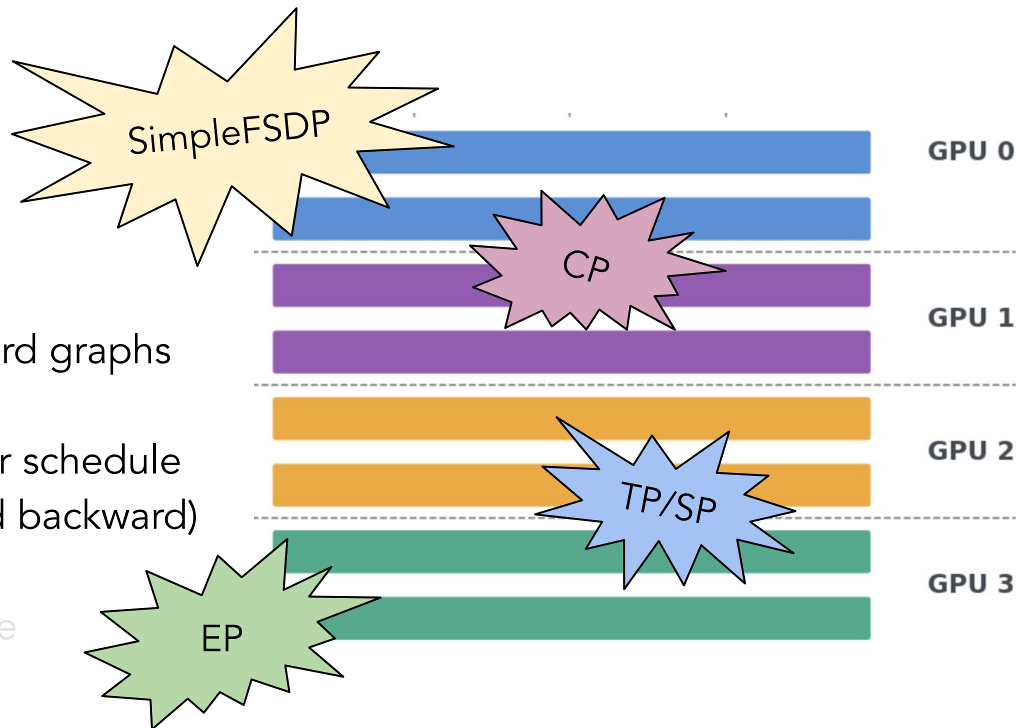
Authoring via Graph based PP

1. Split model into stages
2. Parallelize each stage
3. Compile to obtain Forward/Backward graphs
4. Apply graph transforms required for schedule (e.g. B/W split, Overlapped forward backward)
5. Integrate into your trainer or use the torch.pipelining runtime



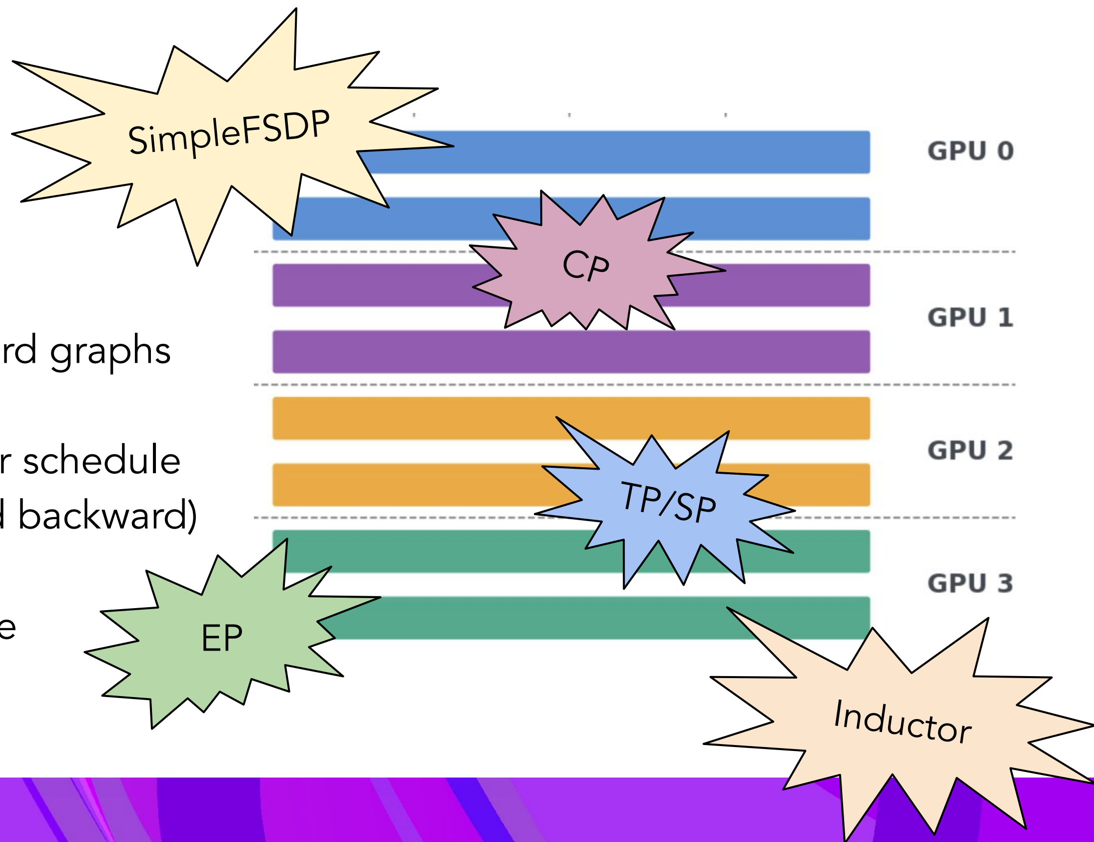
Authoring via Graph based PP

1. Split model into stages
2. Parallelize each stage
3. Compile to obtain Forward/Backward graphs
4. Apply graph transforms required for schedule (e.g. B/W split, Overlapped forward backward)
5. Integrate into your trainer or use the torch.pipelining runtime



Authoring via Graph based PP

1. Split model into stages
2. Parallelize each stage
3. Compile to obtain Forward/Backward graphs
4. Apply graph transforms required for schedule (e.g. B/W split, Overlapped forward backward)
5. Integrate into your trainer or use the torch.pipelining runtime



Graph transforms: Backward decomposition

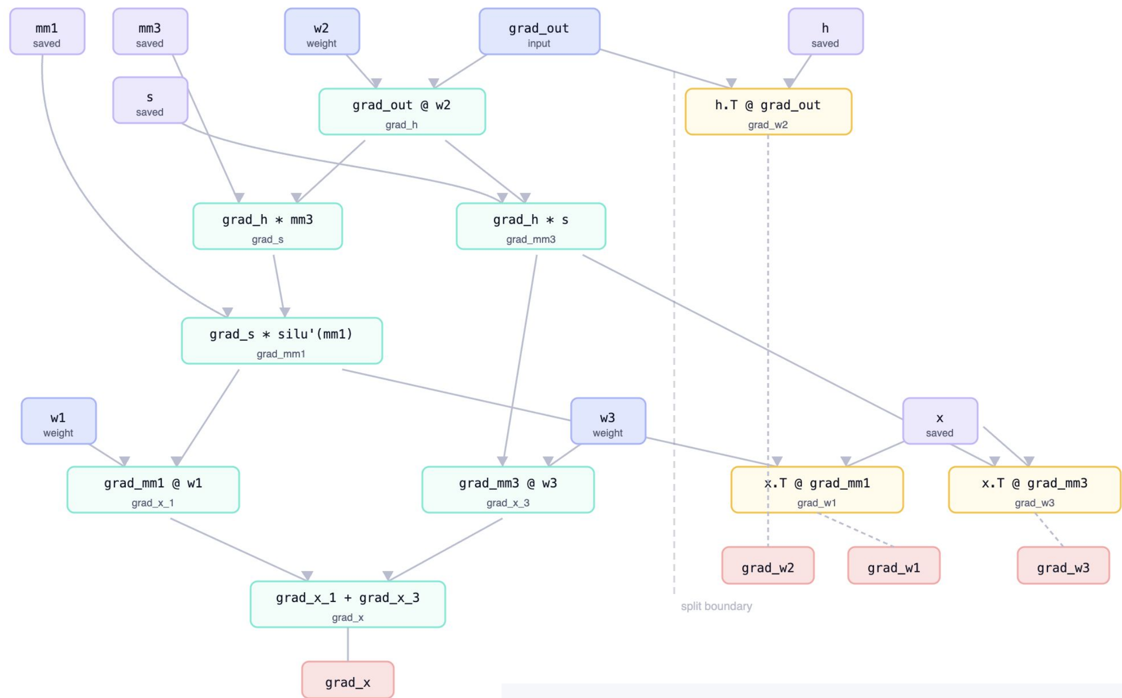
```
# MLP Forward
mm1 = x @ w1.T   // gate projection
mm3 = x @ w3.T   // up projection
s   = silu(mm1) // activation
h   = s * mm3   // gated product
out = h @ w2.T   // down projection
```

dI node (input gradients) dW node (weight gradients) Weight (parameter) Saved activation Graph output

Graph transforms: Backward decomposition

```
# MLP Forward
mm1 = x @ w1.T   // gate projection
mm3 = x @ w3.T   // up projection
s   = silu(mm1) // activation
h   = s * mm3   // gated product
out = h @ w2.T   // down projection
```

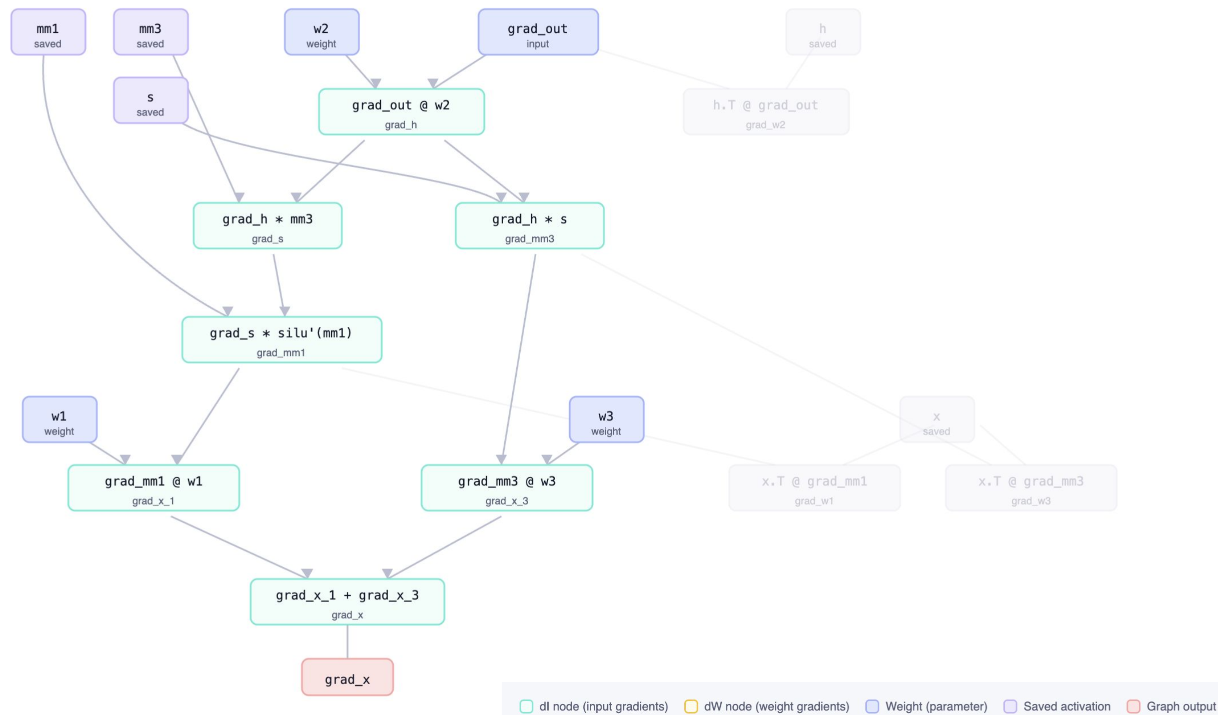
We obtain its backward via tracing (AOTAutograd)



Graph transforms: Backward decomposition

```
# MLP Forward
mm1 = x @ w1.T   // gate projection
mm3 = x @ w3.T   // up projection
s   = silu(mm1) // activation
h   = s * mm3   // gated product
out = h @ w2.T   // down projection
```

Keep only the nodes that lead to grad_x for B



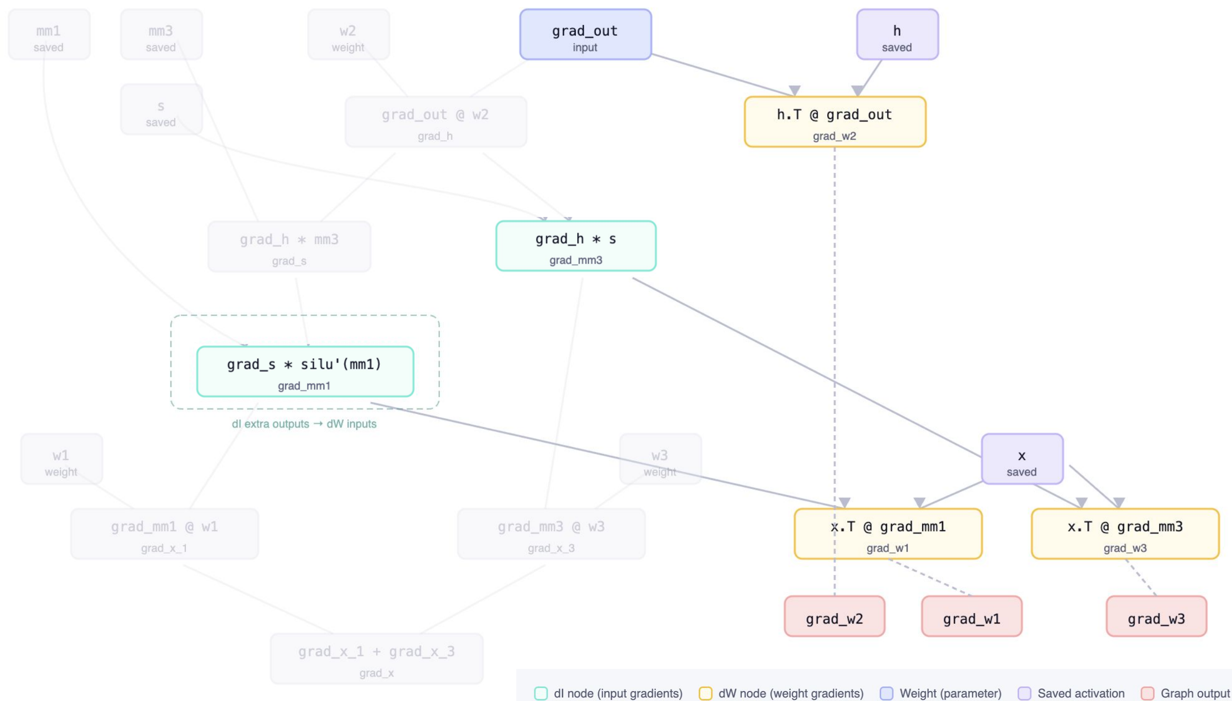
PyTorch CONFERENCE EUROPE 2026

Graph transforms: Backward decomposition

```
# MLP Forward
mm1 = x @ w1.T   // gate projection
mm3 = x @ w3.T   // up projection
s   = silu(mm1) // activation
h   = s * mm3   // gated product
out = h @ w2.T   // down projection
```

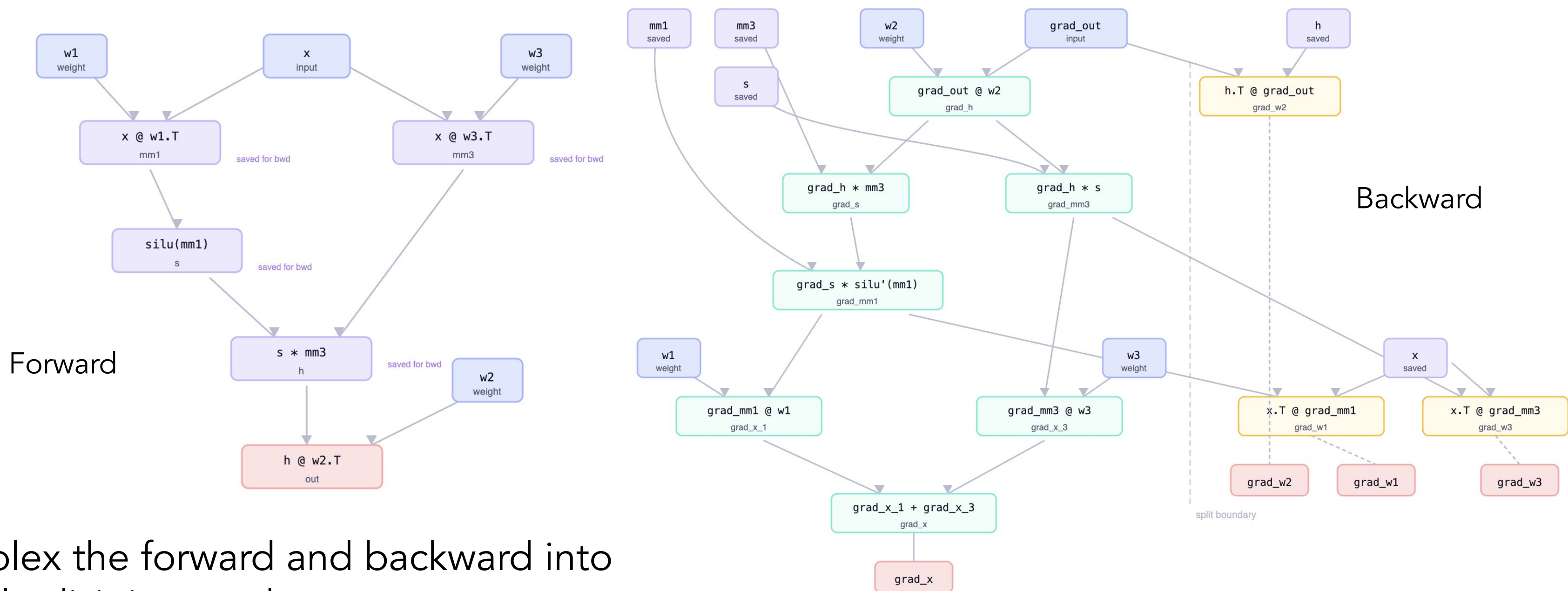
Keep only the nodes that lead to grad_x for B

Or to grad_w1/2/3 for W



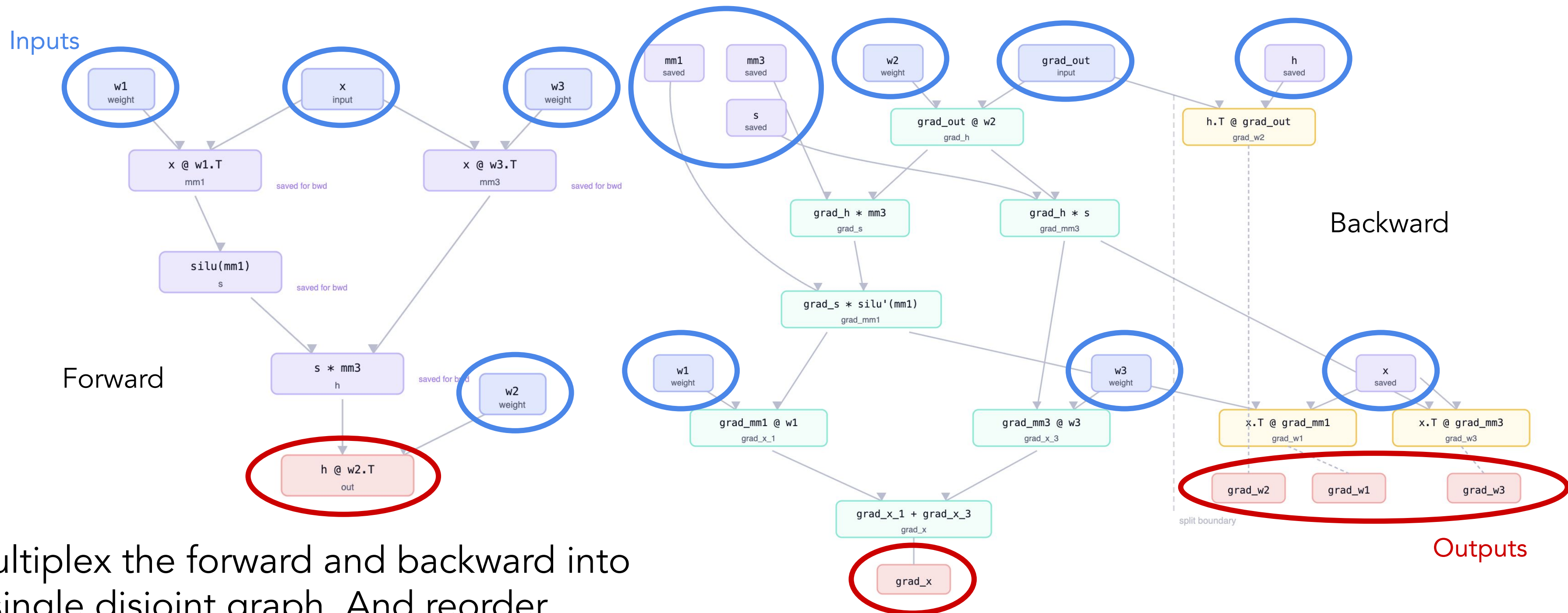
PyTorch CONFERENCE EUROPE 2026

Graph transforms: Overlapped forward backward



Multiplex the forward and backward into a single disjoint graph.
(Only MLP is shown)

Graph transforms: Overlapped forward backward

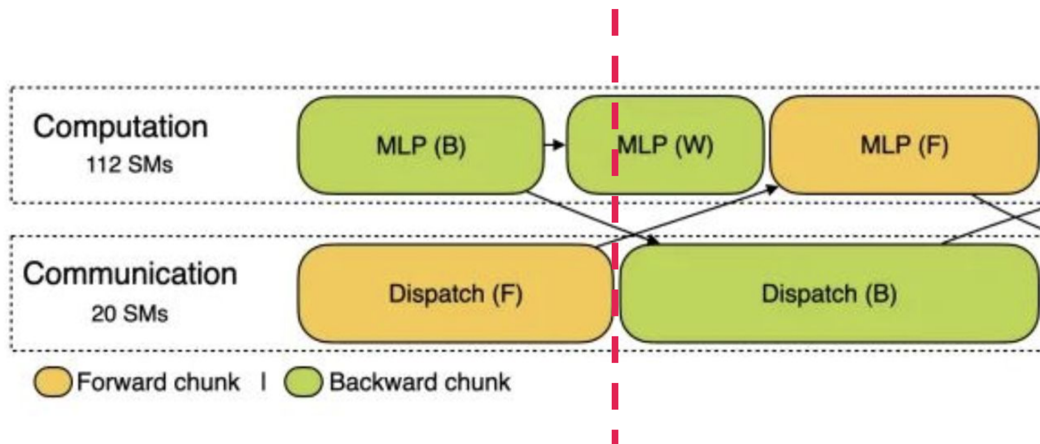


Multiplex the forward and backward into a single disjoint graph. And reorder.

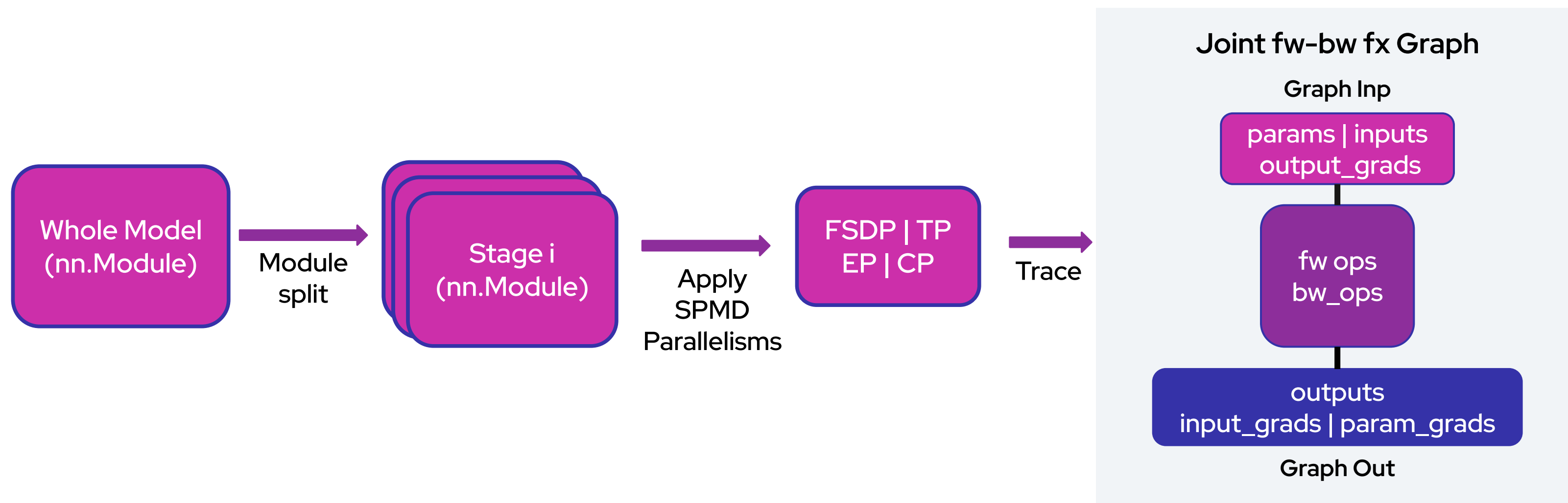
(Only MLP is shown)

Graph transforms: Overlapped forward backward

```
# Multiplexed graph
comm stream:
  Dispatch(F) nodes
compute stream:
  MLP(B) nodes
  MLP(W) nodes part 1
comm stream:
  Dispatch(B) nodes
compute stream:
  MLP(W) nodes part 2
  MLP(F) nodes
```



Graph Trainer Workflow: Graph Capture



Graph Based PP Schedule IR

Device 0
Device 1
Device 2
Device 3
Time →

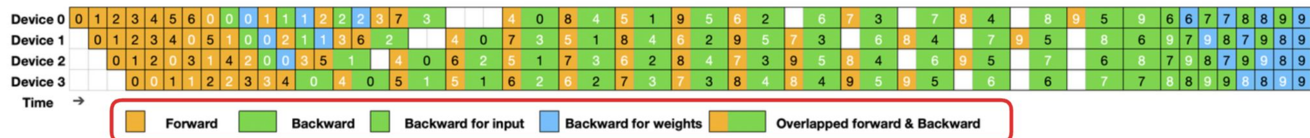
Forward | Backward | Backward for input | Backward for weights | Overlapped forward & Backward

pipeline operations

The IR has specifications of the following types:

iFj - Run Stage i's Forward with microbatch j
iIj - Run Stage i's Backward Input Grad with microbatch j
iWj - Run Stage i's Weight Grad with microbatch j
iBj - Run Stage i's Backward with microbatch j
(iFj; mBk) - Run Stage i's forward and Stage m's Backward with microbatches j and k respectively
Unshard i - Unshard Stage i's params
Reshard i - Reshard Stage i's params
ReduceGrad i - Reduce Stage i's param grads
iSEND_F/Bj - Send output/input_grad of F/B Stage i and microbatch j to next PP rank
iRECV_F/Bj - Receive input/output_grad of F/B Stage i and microbatch j from prev PP rank

Graph Based PP Schedule IR



pipeline operations

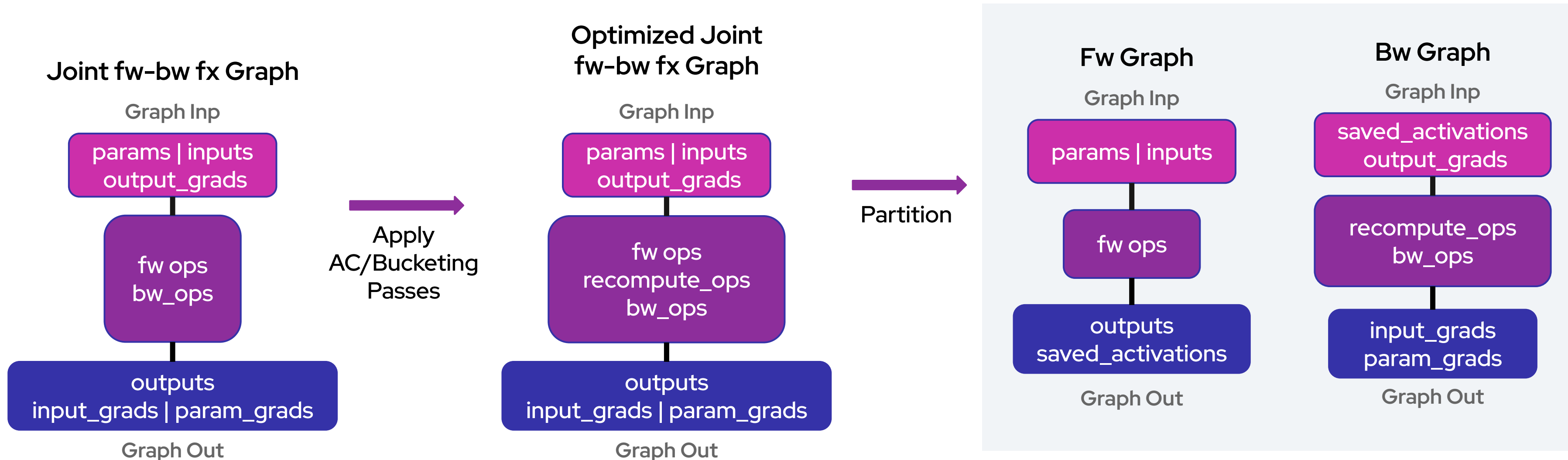
Compute only PP IR

```
          Rank 0    Rank 1
Step 00:  0F0       1F0
Step 01:  3F0       2F0
Step 02:  3I0       2I0
Step 03:  3W0       2W0
Step 04:  0I0       1I0
Step 05:  0W0       1W0
```

Compute PP IR augmented with PP comms

```
          Rank 0     Rank 1
Step 00:  0F0        1RECV_F0
Step 01:  0SEND_F0   1F0
Step 02:  3RECV_F0   2F0
Step 03:  3F0        2SEND_F0
Step 04:  3I0        2RECV_B0
Step 05:  3SEND_B0   2I0
Step 06:  3W0        2W0
Step 07:  0RECV_B0   1I0
Step 08:  0I0        1SEND_B0
Step 09:  0W0        1W0
```

Graph Passes: Partitioning fw-bw graphs

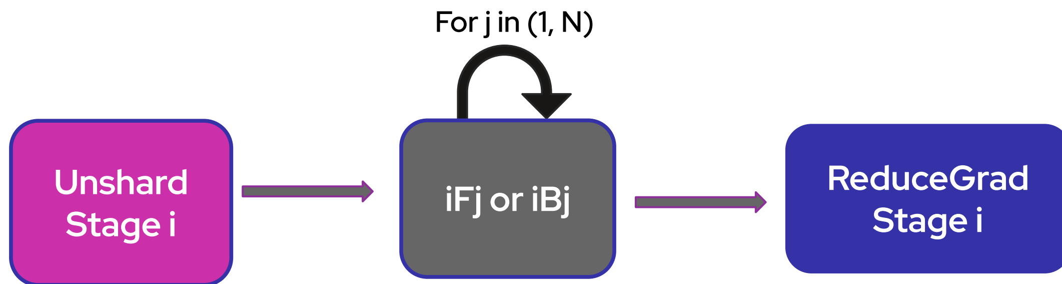


iFj - Run Stage i's Forward with microbatch j

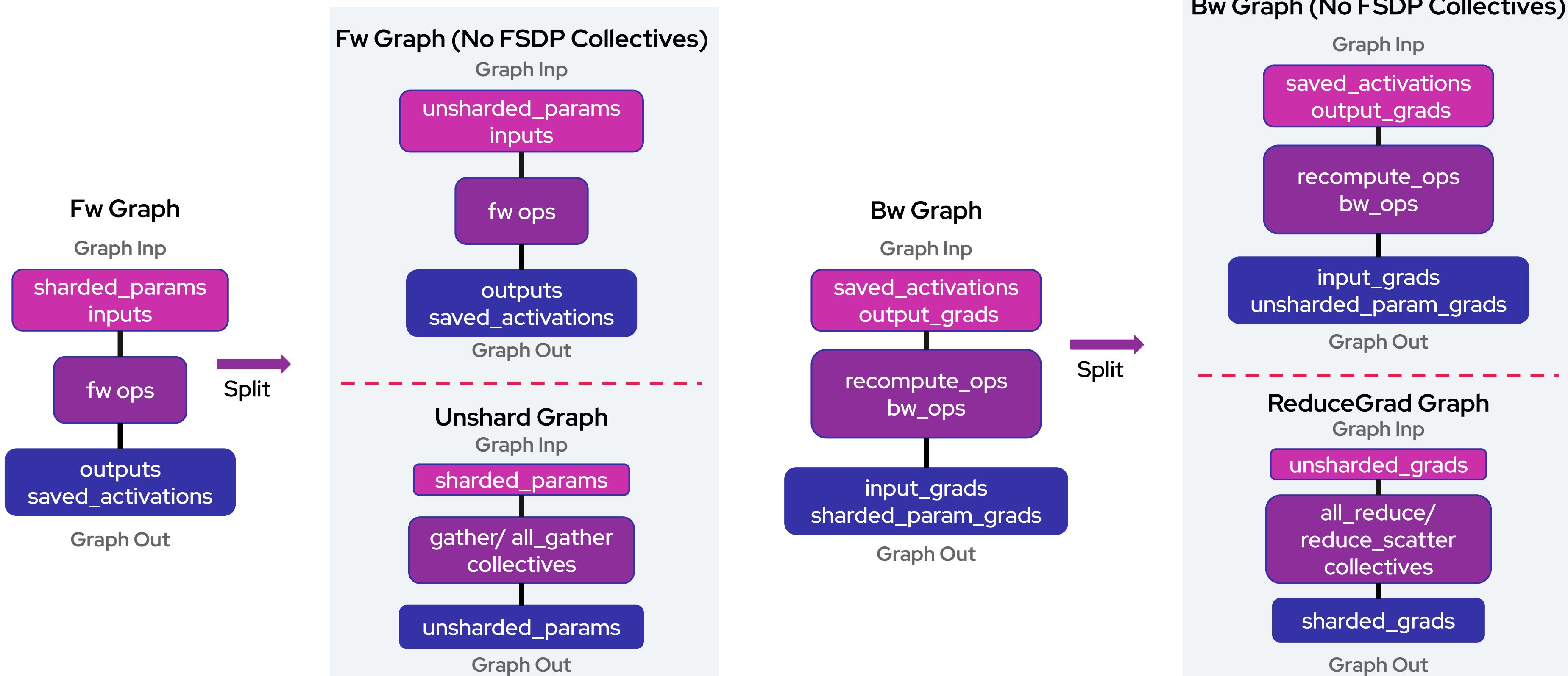
iBj - Run Stage i's Backward with microbatch j

Graph Passes: Need to Unshard/Reshard only once

iFj - Run Stage i's Forward with microbatch j
iBj - Run Stage i's Backward with microbatch j
Unshard i - Unshard Stage i’s params
ReduceGrad i - Reduce Stage i’s param grads



Graph Passes: Splitting FSDP Collectives



Graph Passes: Splitting Bw into dI-dW

Forward:

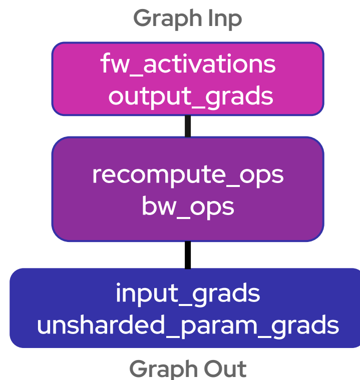
$g_{[A,C]} = x_{[A,B]} @ w^T_{[B,C]}$

Backward:

dI: $x'_{[A,B]} = g'_{[A,C]} @ w_{[C,B]}$

dW: $w'^T_{[B,C]} = (g'^T_{[C,A]} @ x_{[A,B]})^T$

Bw Graph (No FSDP Collectives)



Split

dI Graph

Graph Inp

fw_activations
output_grads

recompute_ops
dI_ops

input_grads
bw_activations

Graph Out

dW Graph

Graph Inp

bw_activations

dW_ops

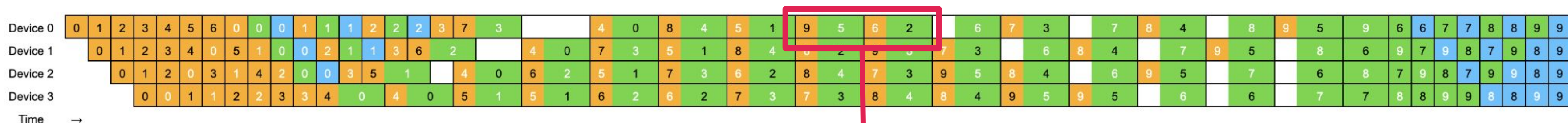
unsharded_param_grads

Graph Out

iIj - Run Stage i's Backward Input Grad with microbatch j

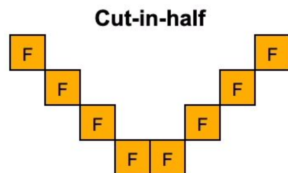
iWj - Run Stage i's Weight Grad with microbatch j

Graph Passes: Overlapping Fw-Bw Graphs



	Model Layers	
Device 0	0	7
Device 1	1	6
Device 2	2	5
Device 3	3	4

Cut-in-half

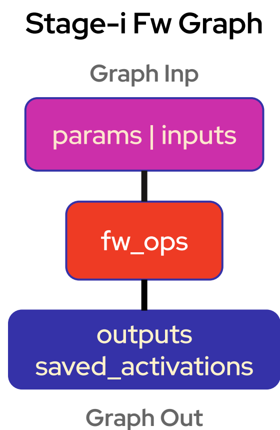


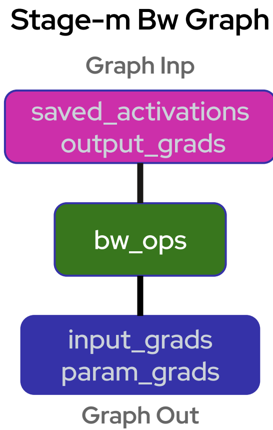
(iFj; mBk) - Run Stage i's forward and Stage m's Backward with microbatches j and k respectively

(mFk; iBj) - Run Stage m's forward and Stage i's Backward with microbatches k and j respectively

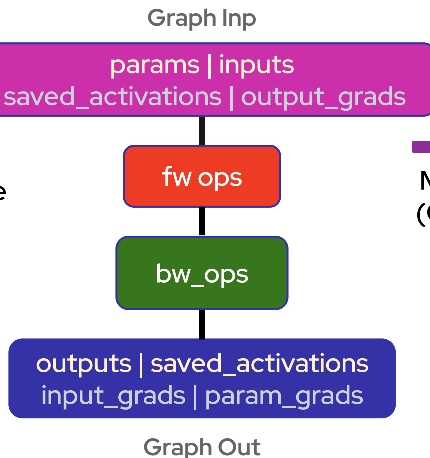
PyTorch CONFERENCE EUROPE 2026

Graph Passes: Overlapping Fw-Bw Graphs

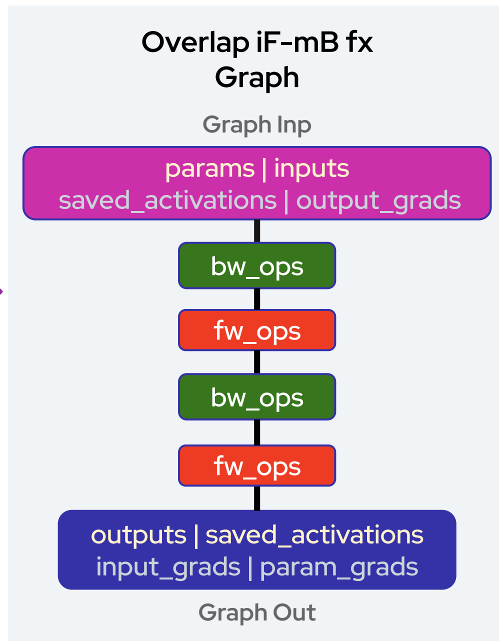




Concatenate



Multiplex
(Overlap)



(iFj; mBk) - Run Stage i's forward and Stage m's Backward with microbatches j and k respectively

Similarly obtain Overlap mF-iB fx Graph

Graph Based PP Runtime

Per Stage(i)
Graph Callables

unshard
fw
full_bw
bw_dI
bw_dW
reduce_grad

Cross Stage(i, m)
Multiplexed
Graph Callables

overlap_fw_bw

Schedule IR Actions

iFj
iIj
iWj
iBj
(iFj; mBk)
Unshard i
Reshard i
ReduceGrad i
iSEND_F/Bj
iRECV_F/Bj

Per Stage(i)
Params | Grads

unsharded_params
sharded_params
unsharded_grads
sharded_grads

Per Microbatch(j)
Recv Buffers

inputs
output_grads

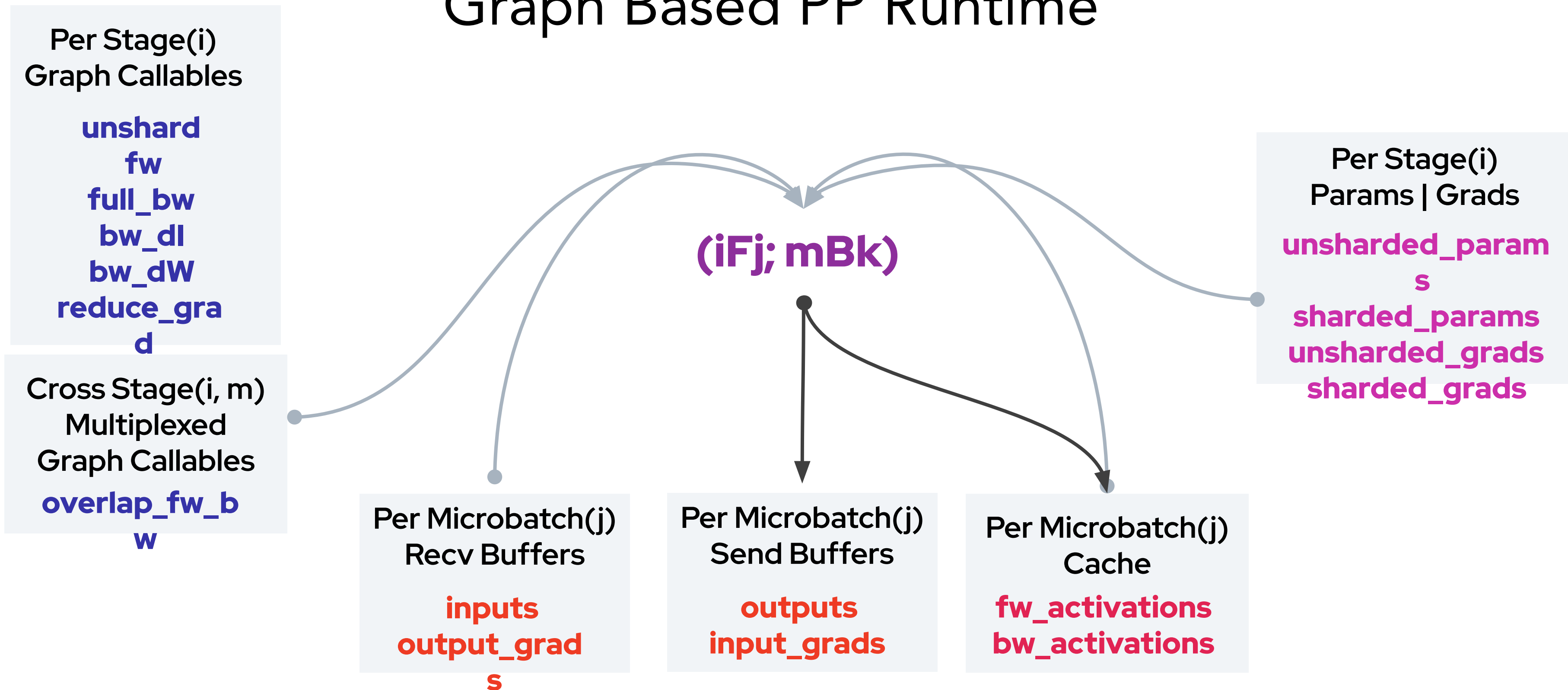
Per Microbatch(j)
Send Buffers

outputs
input_grads

Per Microbatch(j)
Cache

fw_activations
bw_activations

Graph Based PP Runtime



Graph Passes APIs

```python
def split_fsdp_prefetch(
    gm: torch.fx.GraphModule,
    num_params: int,
) -> tuple[torch.fx.GraphModule, torch.fx.GraphModule]:
```

```python
def split_fsdp_reduce_scatters_epilogue(
    gm: torch.fx.GraphModule,
    num_grads: int,
) -> tuple[torch.fx.GraphModule, torch.fx.GraphModule]:
```

```python
def split_di_dw_graph(
    bw_gm_old: fx.GraphModule, *, num_weight_gradients: int
) -> tuple[fx.GraphModule, fx.GraphModule, int]:
```

```python
def multiplex_fw_bw_graph(
    fw_gm: fx.GraphModule, bw_gm: fx.GraphModule, overlap_with_annotations: bool = True
) -> fx.GraphModule:
```

Graph PP APIs

```python
@dataclass
class GraphCallables:
    fw: fx.GraphModule
    full_bw: fx.GraphModule
    bw_dI: Optional[fx.GraphModule] = None
    bw_dW: Optional[fx.GraphModule] = None
    unshard: Optional[fx.GraphModule] = None
    reduce_grad: Optional[fx.GraphModule] = None


@dataclass
class GraphMeta:
    num_mutate_inputs: int
    num_user_outputs: int
    num_symints_saved_for_bw: int
    num_params: int
    num_buffers: int
    num_input_grads: int
```

```python
class GraphPPRunner:
    def __init__(
        self,
        schedule: _PipelineScheduleRuntime,
        inductor: bool = False,
    ):
```

```python
class GraphPipelineStage(PipelineStage):
    def __init__(
        self,
        submodule: torch.nn.Module,
        graph_callables: GraphCallables,
        graph_meta: GraphMeta,
        stage_index: int,
        num_stages: int,
        device: torch.device,
        input_args: Optional[Union[torch.Tensor, tuple[torch.Tensor, ...]]] = None,
        output_args: Optional[Union[torch.Tensor, tuple[torch.Tensor, ...]]] = None,
        group: Optional[torch.distributed.ProcessGroup] = None,
        dw_builder: Optional[Callable[[], Callable[..., None]]] = None,
        numerics_logger: Optional[NumericsLogger] = None,
        should_log_fw_outs: bool = False,
    ):
```

Experimental Set-Up

Parameter	Value
Model	DeepSeek-V3 16B (27 layers, dim=2048, 16 heads, MLA attention)
MoE	64 routed + 2 shared experts, top_k=6, hidden_dim=1408
Sequence Length	4,096
Precision	bf16 mixed precision
Hardware	64× H100 96GB (8 nodes × 8 GPUs)
Interconnect	Intra-node: NVLink/NVSwitch (~900 GB/s), Inter-node: InfiniBand (~400 GB/s)
Parallelism	PP=4, EP=16 (cross-node), FSDP=16
Batch Config	local_batch_size=16, microbatch_size=1 (16 microbatches)

GraphPP (w/ compile) vs EagerPP

Schedule	GraphPP Speedup	GraphPP Passes	EagerPP Infra
DualPipeV	**+70.8%**	split_fsdp_collectives, split_dI_dW, multiplex_fw_bw	unshard/reshard APIs, split_dI_dW (autograd), overlap_fw_bw (coroutines)
ZeroBubbleV	**+63.3%**	split_fsdp_collectives, split_dI_dW	unshard/reshard APIs, split_dI_dW (autograd)
Interleaved1F1B	**+49.2%**	split_fsdp_collectives	unshard/reshard APIs

GraphPP Speed-up Breakdown

Comparison	Speed-up	Graph Passes
DualPipeV vs 1F1B	17.3%	+multiplex_fw_bw, +split_dI_dW
DualPipeV vs ZBV	+9.6%	+multiplex_fw_bw
ZBV vs 1F1B	+7.0%	+split_dI_dW

Current Status

GraphPP is part of the GraphTrainer experiment in TorchTitan.

Next Steps

Extending CUDA Graph support
Enabling Compile on one rank
Upstreaming to torch.pipelining