



CONFERENCE

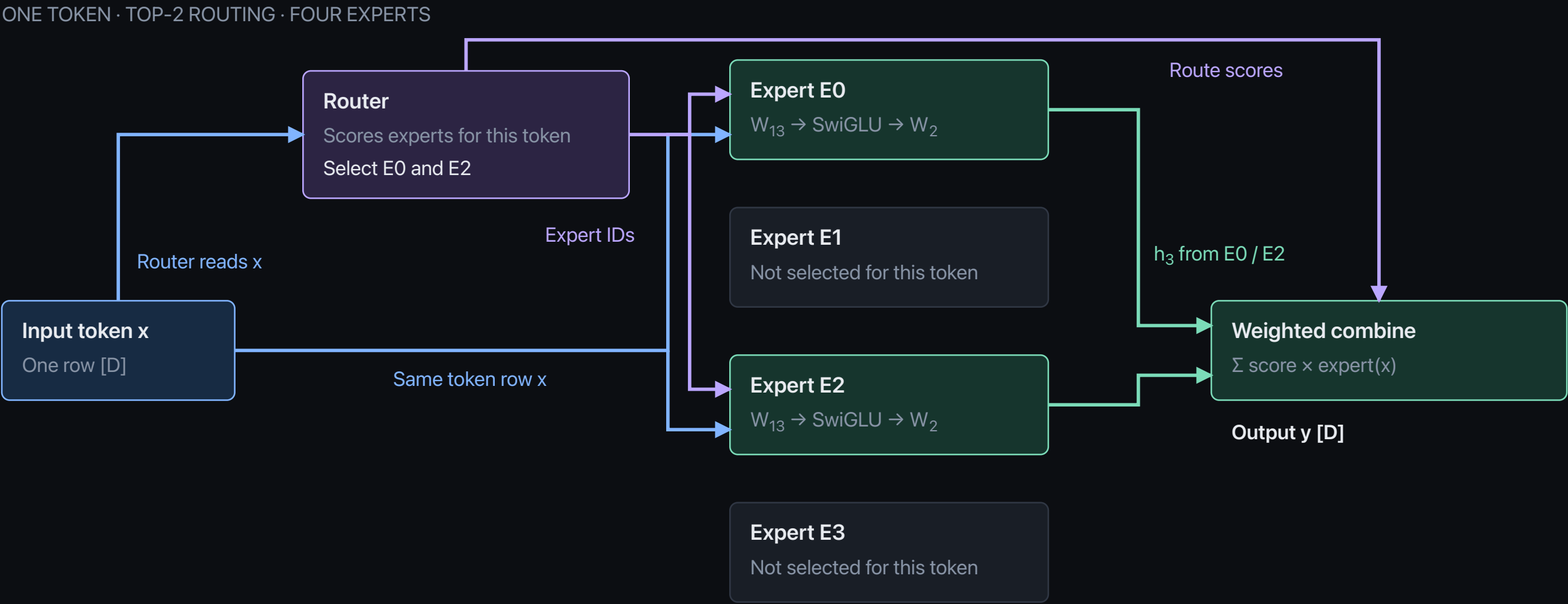
—NORTH AMERICA 2026—

Dist-MoE: Fused Comm-Compute Kernels & Activation Packing for Speed-of-Light MoEs

Shikai Li Sanket Purandare

(Meta)

MoE Overview



The caller supplies routing. Dist-MoE executes the selected experts and their distributed dispatch/combine.

EACH SELECTED EXPERT · COLUMN-VECTOR MATH

$$h_1 = W_{13}x = [g; u] \longrightarrow h_2 = \text{SiLU}(g) \odot u \longrightarrow h_3 = W_2h_2$$

g and u are the gate and up projections. SiLU(z) = z × sigmoid(z); ⊙ is elementwise multiplication.

A Mixture-of-Experts (MoE) layer contains several expert MLPs. A router selects a few for each token and assigns their scores. Only those experts run; their weighted outputs form one output row.

Open Source MoE Capability Comparison

CUDA Graphs and Activation Packing

Capability	Dist-MoE Meta	MoK Cursor	Comet / Flux ByteDance	Mega MoE DeepSeek
CUDA graph execution	Yes	Yes	No reviewed dynamic-routing path	Yes
Saved-state packing across layers into a bounded buffer	Yes	No builtin planner	No builtin planner	No training-state planner
Dynamic recompute	Yes layer-budget decisions	Yes macrobatch replay ¹	No builtin policy	No fused training backward
Fused comm-compute	Yes	Yes	Yes	Yes
Forward and backward kernels	Yes BF16 / MXFP8	Yes BF16 / MXFP8	Yes training primitives ²	Fused forward only
Runs without a shared-expert branch	Yes	No current API requires shared weights	Yes	Yes shared experts optional

¹ MoK replays macrobatches beyond retained capacity and supports manual checkpointing. Dist-MoE uses a shared layer budget.

² Comet provides training primitives. Framework integration coordinates backward.

Dist-MoE packs activations across layers during CUDA graph replay; this capability is absent from the reviewed alternatives.

Public-source review, October 14, 2026.

Dist-MoE Supported Modes: Droptless, Fully Cudagraphable

Training and Inference by Precision

Precision	Pipeline	Training	Inference
BF16	Standard	Yes	Yes
MXFP8 E4M3	Staged / Mega	Yes	Yes
NVFP4	Staged / Mega	No	Yes · prepared weights

BF16 and MXFP8 support training and inference. NVFP4 supports inference with prepared expert weights.

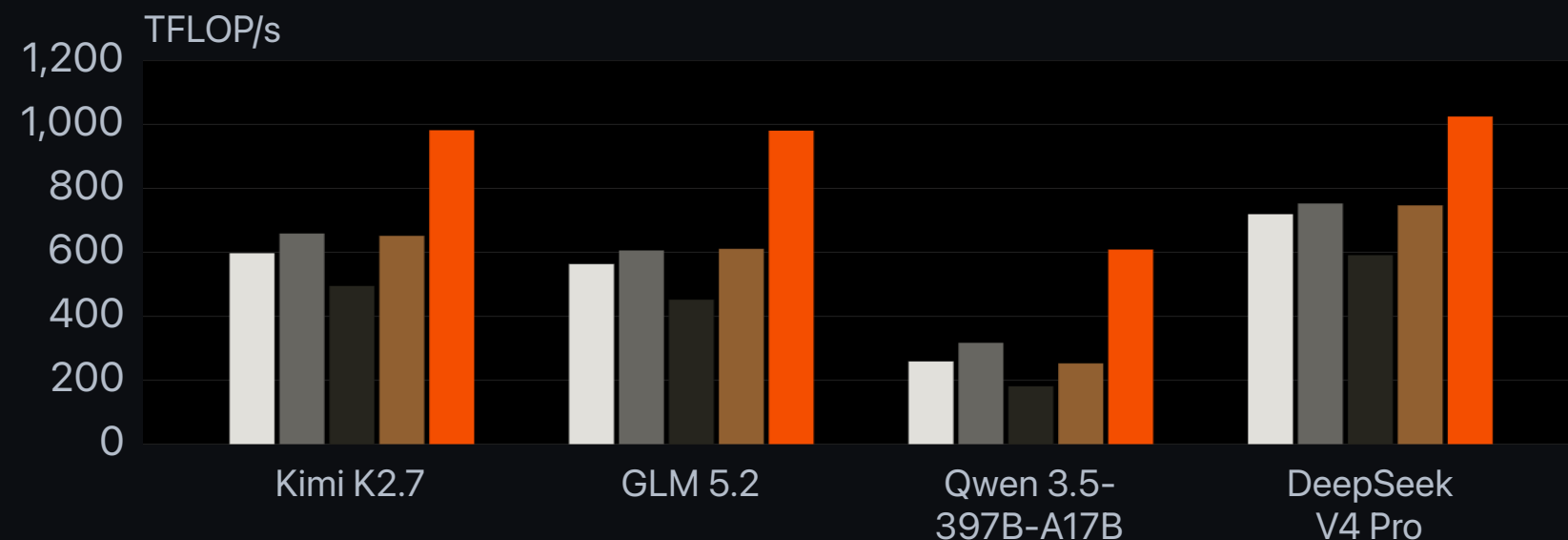
Supported modes in the public API · NVFP4 uses prepared expert weights.

MoK: SoTA OSS baseline

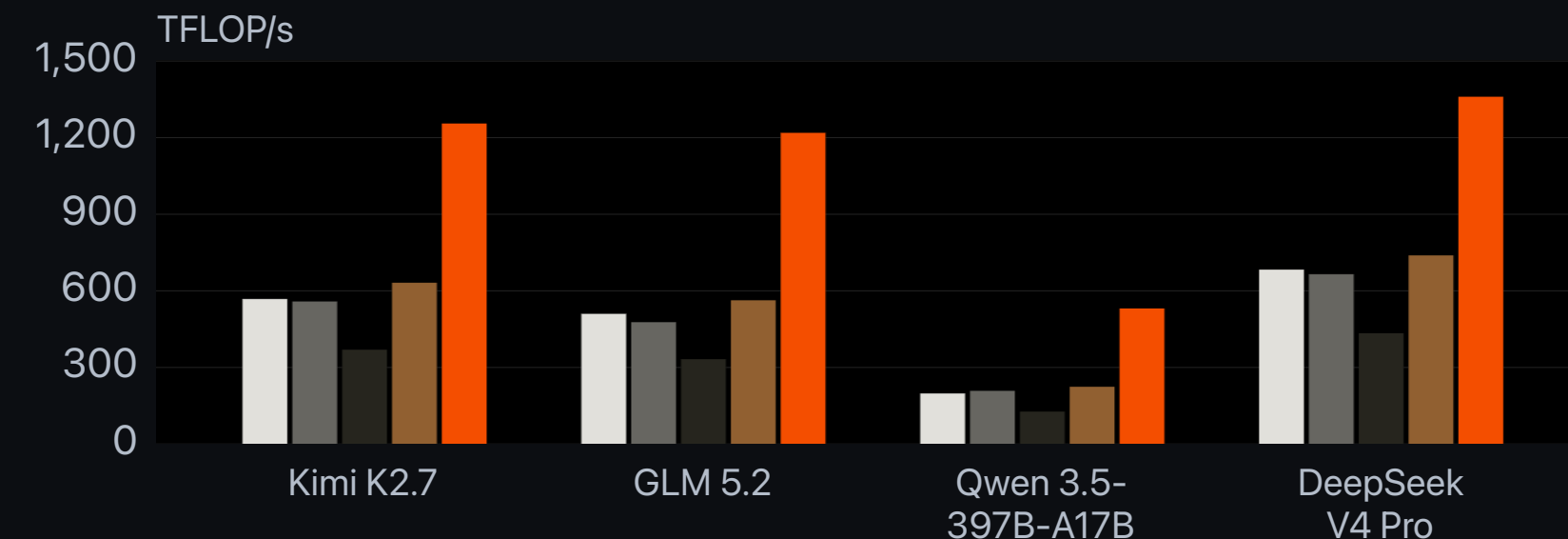
Strong published NVL72 training results motivate our baseline choice

GB300 NVL72 · EP64 · 2,048 tokens/GPU · single MoE layer

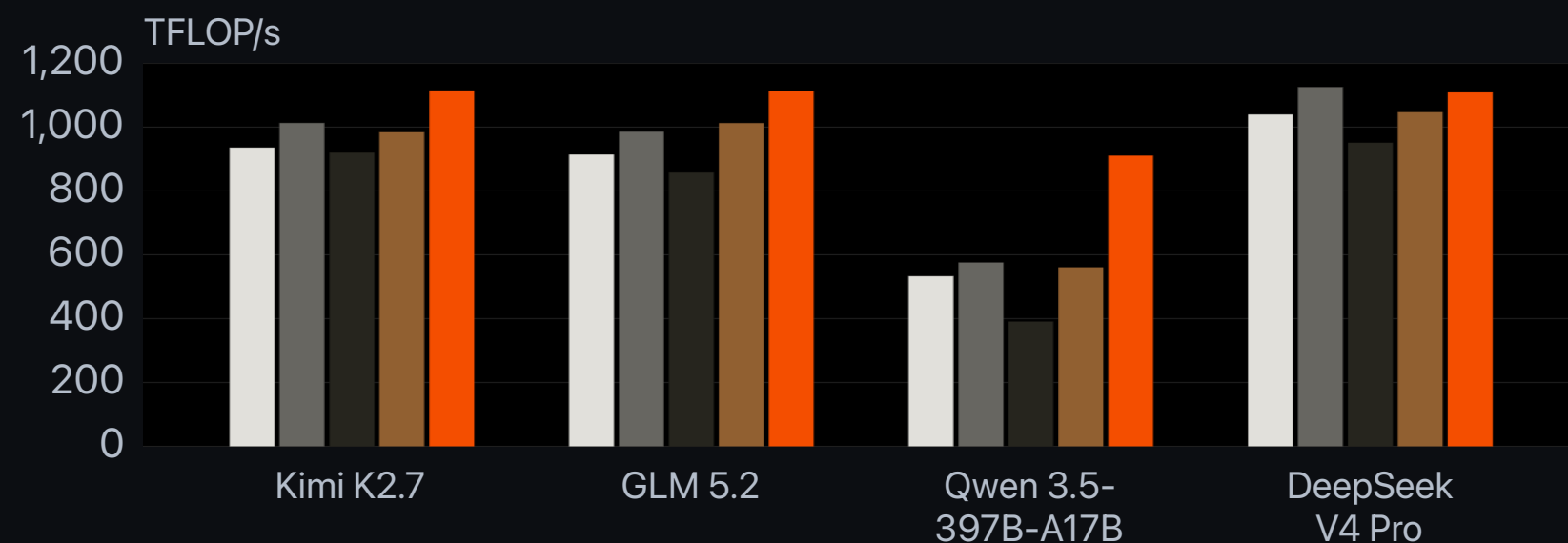
BF16 · Forward



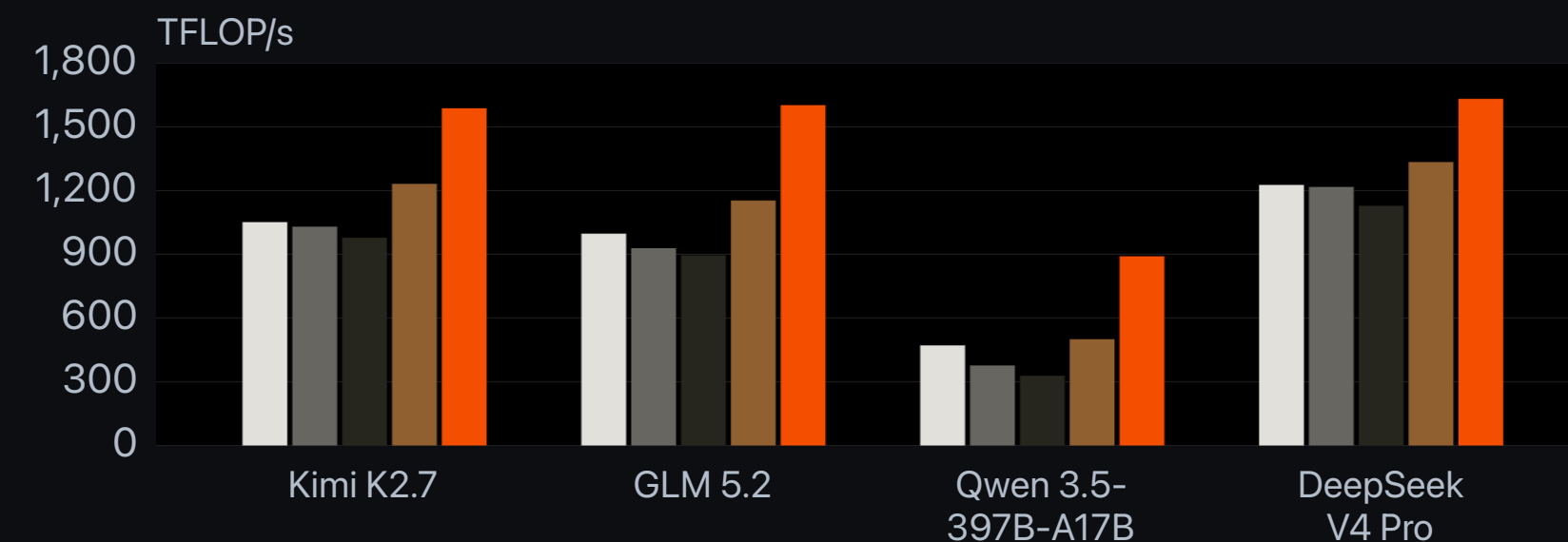
MXFP8 · Forward



BF16 · Backward



MXFP8 · Backward



■ NCCL + PyTorch ■ DeepEP + PyTorch ■ DeepEP + TransformerEngine ■ HybridEP + Megatron ■ MoK

Results picked from Cursor's [Mixture-of-Kittens blog](#) · [cursor.com/blog/mixture-of-kittens](#) ↗ · [MoK paper, §4.1 / Fig. 7](#) ↗.

MoK leads forward and MXFP8 backward; DeepEP + PyTorch is slightly ahead in BF16 backward for DeepSeek-V4-Pro. These are Cursor-reported results.

MoK vs Dist-MoE: Imbalanced Multi-Layer Workload

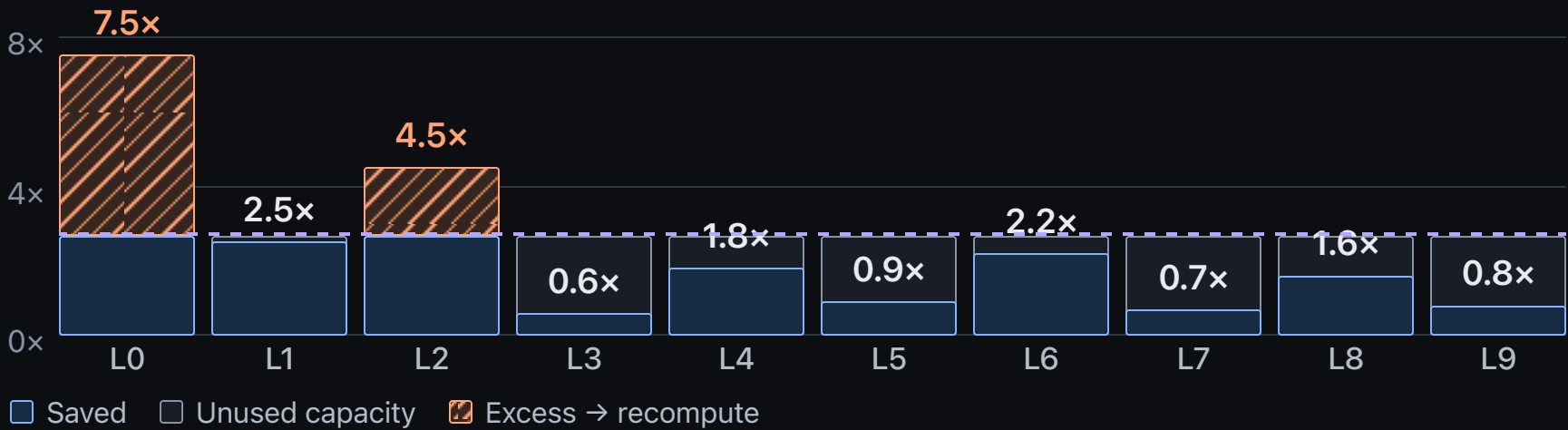
Synthetic ten-layer training · per-layer imbalance · BF16/MXFP8 · normalized units (1 = balanced layer)

Imbalance profile A

Average 2.31x · peak 7.5x

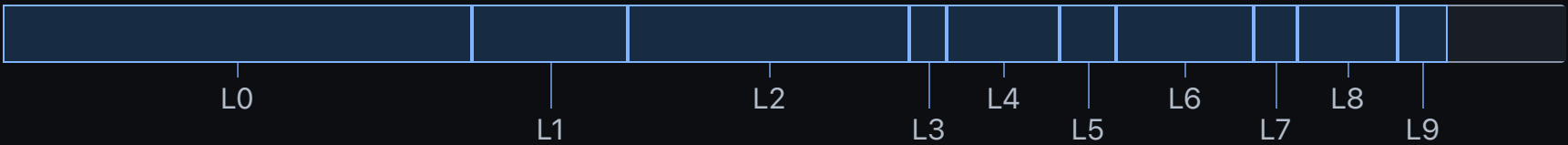
MoK · ten separate layer slots

--- Per-layer capacity ≈ 2.67 units



2 hot layers partly saved · 10.2 units stay unused

Dist-MoE · one 25-unit shared slot



23.1 / 25 units packed · 1.9 unused

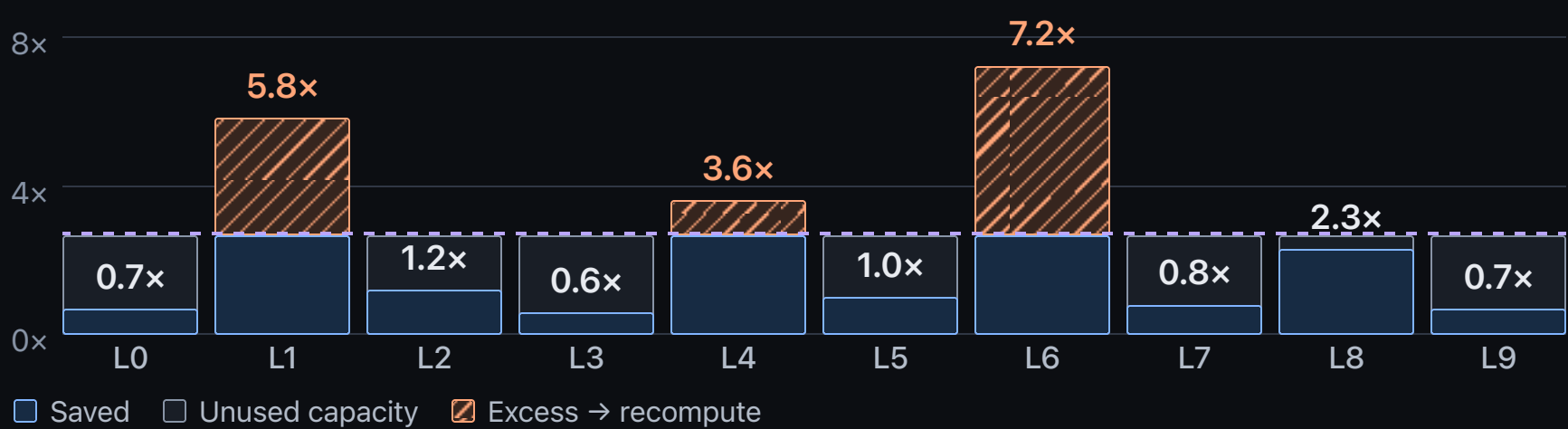
All ten layers saved · 0 capacity-induced recompute

Imbalance profile B

Average 2.39x · peak 7.2x

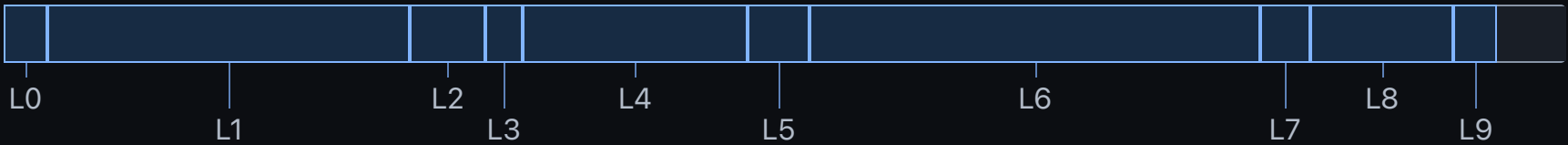
MoK · ten separate layer slots

--- Per-layer capacity ≈ 2.67 units



3 hot layers partly saved · 11.4 units stay unused

Dist-MoE · one 25-unit shared slot



23.9 / 25 units packed · 1.1 unused

All ten layers saved · 0 capacity-induced recompute

A GPU can be lightly loaded in one layer and heavily loaded in the next. Dist-MoE shares saved-state capacity across those layers; MoK reserves a separate budget for each.

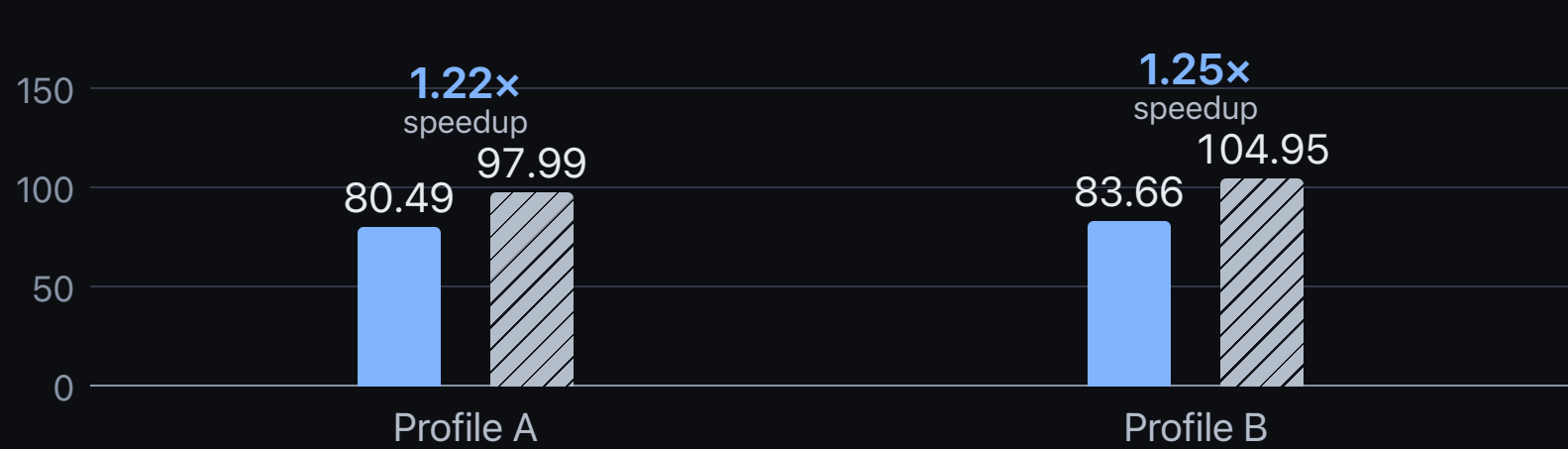
Widths illustrate normalized layer demand. Byte budgets depend on precision and padding; recorded Dist-MoE capacity flags are zero.

MoK vs Dist-MoE: Multi-Layer Training Performance

Ten-layer training · profiles A and B · F = 3,072

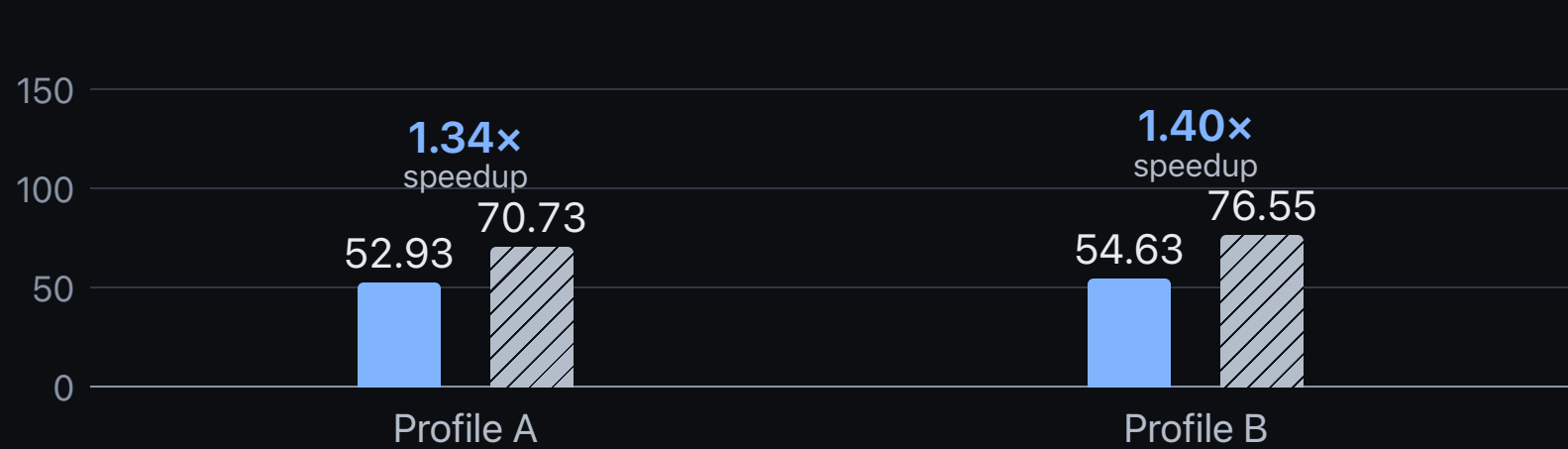
■ Dist-MoE ▨ MoK Columns: iteration ms ↓ · labels: speedup over MoK · table values: Dist / MoK

EP64 · Training · BF16



	Iteration time · ms	
Dist / MoK	Profile A	Profile B
Useful TFLOP/s / GPU	706 / 580	679 / 541
HBM model · TB/s	0.51 / 0.42	0.49 / 0.39
NVLink TX · GB/s	88.0 / 72.6	84.8 / 67.9
NVLink RX · GB/s	88.0 / 72.9	84.8 / 68.2
Peak allocated · GiB	46.99 / 49.74	46.99 / 49.74

EP64 · Training · MXFP8



	Iteration time · ms	
Dist / MoK	Profile A	Profile B
Useful TFLOP/s / GPU	1,074 / 803	1,040 / 742
HBM model · TB/s	0.56 / 0.42	0.54 / 0.38
NVLink TX · GB/s	134.2 / 100.6	129.9 / 93.0
NVLink RX · GB/s	134.2 / 101.0	129.9 / 93.4
Peak allocated · GiB	49.73 / 48.37	49.73 / 48.37

EP64 profile B: MXFP8 delivers a 1.40x speedup and 1,040 useful TFLOP/s per GPU versus 742 for MoK. Dist-MoE’s shared slot retains eligible state for all ten layers.

Matched shapes; one BF16 shared expert. TFLOP/s counts useful work; HBM is modeled; NVLink TX/RX is measured separately.

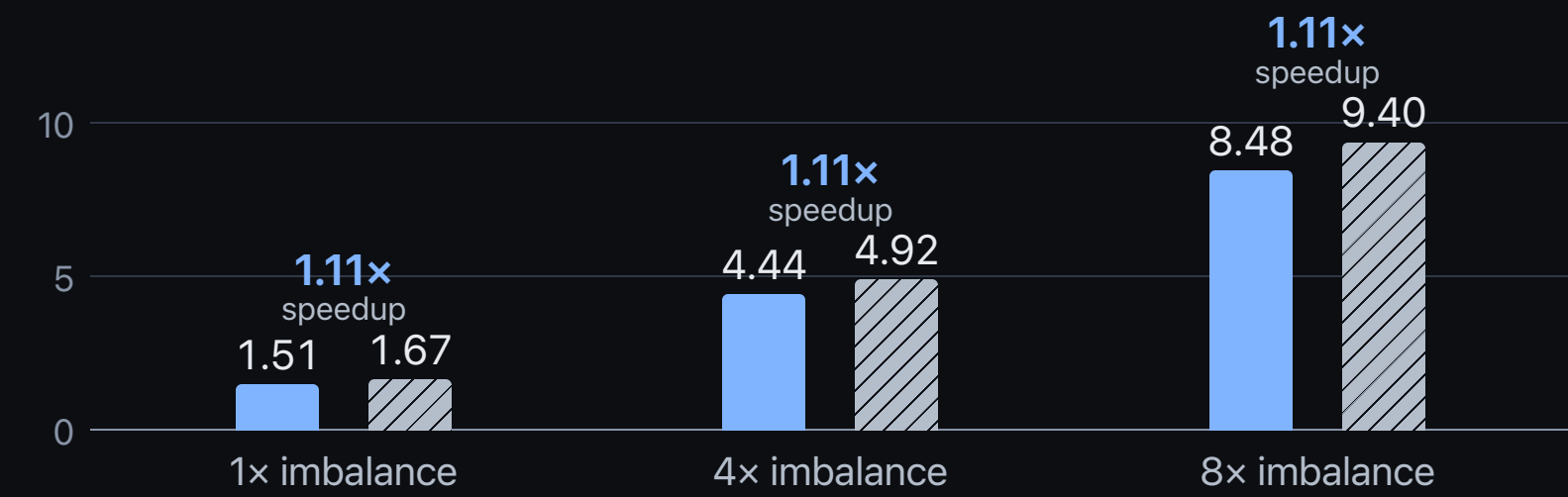
MoK vs Dist-MoE: Routing Imbalance

Inference · EP16 · up to 1.33× speedup over MoK

One layer · imbalance factors 1×, 4×, and 8× · F = 3,072

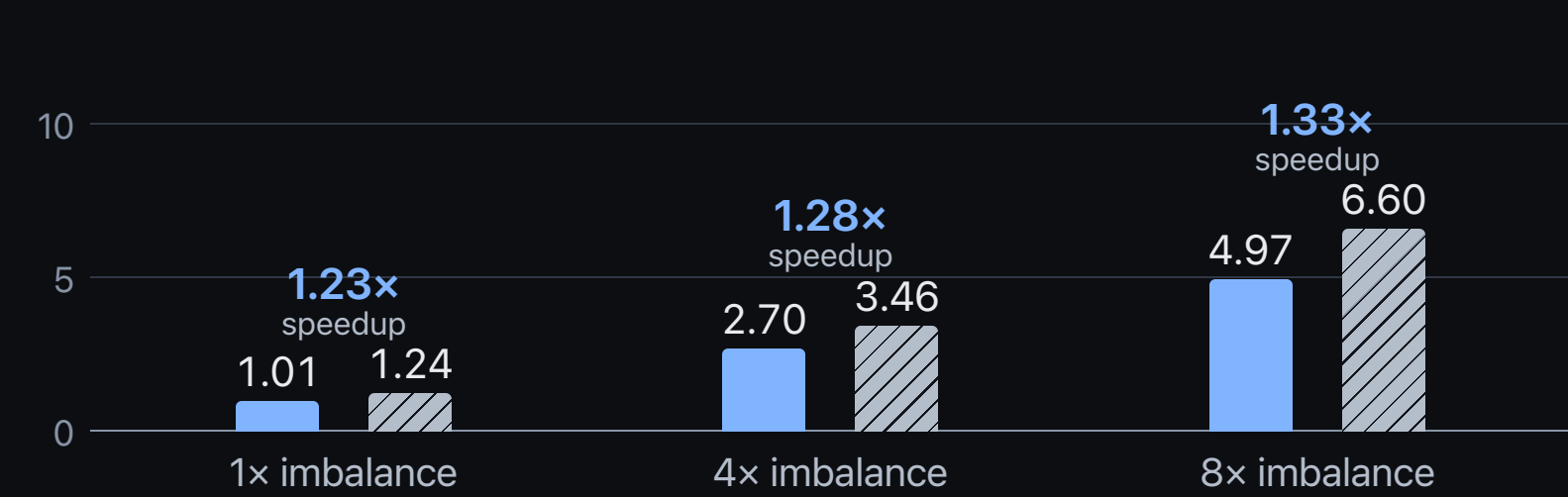
■ Dist-MoE ▨ MoK Columns: iteration ms ↓ · labels: speedup over MoK · table values: Dist / MoK

EP16 · Inference · BF16



	Iteration time · ms		
Dist / MoK	1× imbalance	4× imbalance	8× imbalance
Useful TFLOP/s / GPU	1,256 / 1,133	426 / 385	223 / 201
HBM model · TB/s	2.48 / 2.24	0.84 / 0.76	0.44 / 0.40
NVLink TX · GB/s	213.7 / 194.5	74.7 / 67.9	39.7 / 35.8
NVLink RX · GB/s	213.7 / 194.9	74.7 / 68.0	39.7 / 35.8

EP16 · Inference · MXFP8



	Iteration time · ms		
Dist / MoK	1× imbalance	4× imbalance	8× imbalance
Useful TFLOP/s / GPU	1,872 / 1,523	701 / 547	381 / 287
HBM model · TB/s	2.18 / 1.78	0.82 / 0.64	0.44 / 0.33
NVLink TX · GB/s	238.6 / 262.6	92.1 / 96.6	51.0 / 50.9
NVLink RX · GB/s	238.6 / 263.2	92.1 / 96.8	51.0 / 51.0

At 8× imbalance, BF16 and MXFP8 deliver 1.11× and 1.33× speedup over MoK. Both precisions compare the same routed workload.

Matched shapes; one BF16 shared expert. TFLOP/s counts useful work; HBM is modeled; NVLink TX/RX is measured separately.

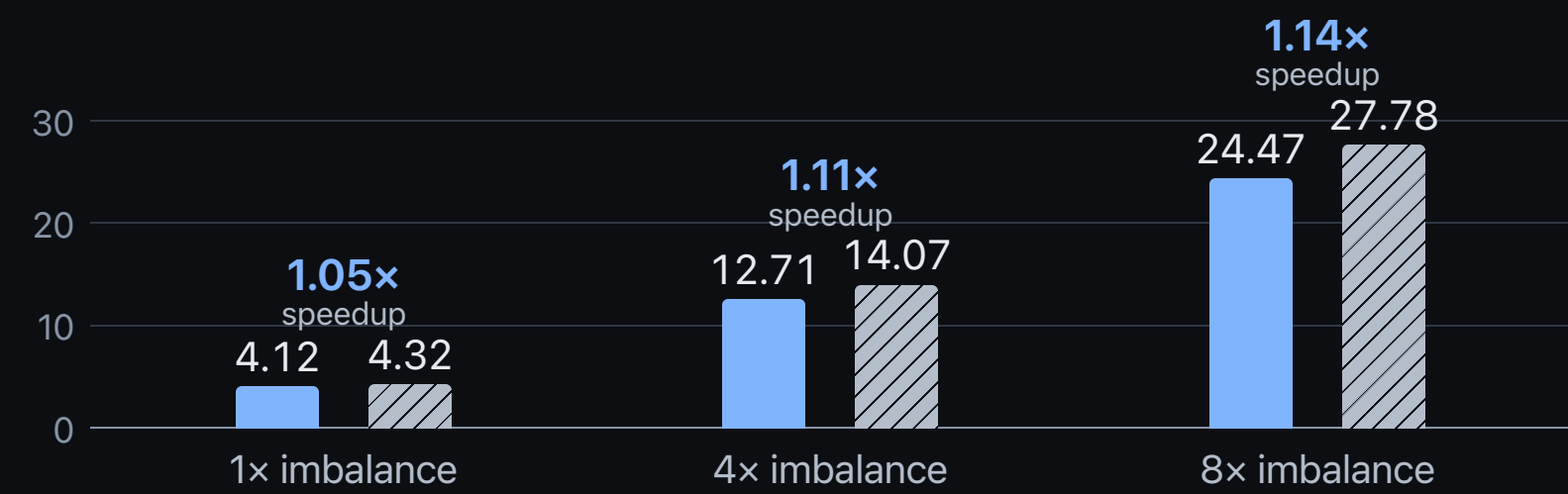
MoK vs Dist-MoE: Routing Imbalance

Training · EP64 · up to 1.16× speedup over MoK

One layer · imbalance factors 1×, 4×, and 8× · F = 3,072

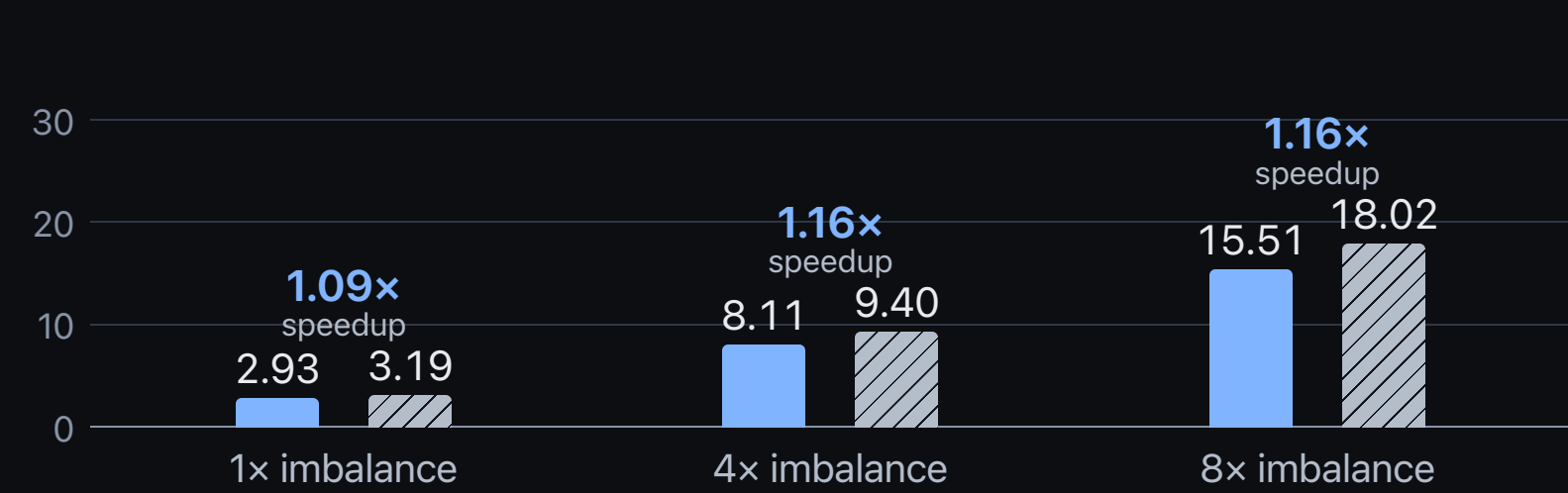
■ Dist-MoE ▨ MoK Columns: iteration ms ↓ · labels: speedup over MoK · table values: Dist / MoK

EP64 · Training · BF16



	Iteration time · ms		
Dist / MoK	1× imbalance	4× imbalance	8× imbalance
Useful TFLOP/s / GPU	1,378 / 1,315	447 / 404	232 / 205
HBM model · TB/s	0.99 / 0.95	0.32 / 0.29	0.17 / 0.15
NVLink TX · GB/s	169.4 / 161.2	55.6 / 50.1	28.9 / 25.5
NVLink RX · GB/s	169.4 / 161.9	55.6 / 50.3	28.9 / 25.6

EP64 · Training · MXFP8



	Iteration time · ms		
Dist / MoK	1× imbalance	4× imbalance	8× imbalance
Useful TFLOP/s / GPU	1,938 / 1,780	700 / 604	366 / 315
HBM model · TB/s	1.00 / 0.92	0.36 / 0.31	0.19 / 0.16
NVLink TX · GB/s	238.2 / 216.8	86.6 / 75.4	45.6 / 39.3
NVLink RX · GB/s	238.2 / 217.8	86.6 / 75.8	45.6 / 39.4

At 8× imbalance, BF16 and MXFP8 deliver 1.14× and 1.16× speedup over MoK. Both precisions compare the same routed workload.

Matched shapes; one BF16 shared expert. TFLOP/s counts useful work; HBM is modeled; NVLink TX/RX is measured separately.

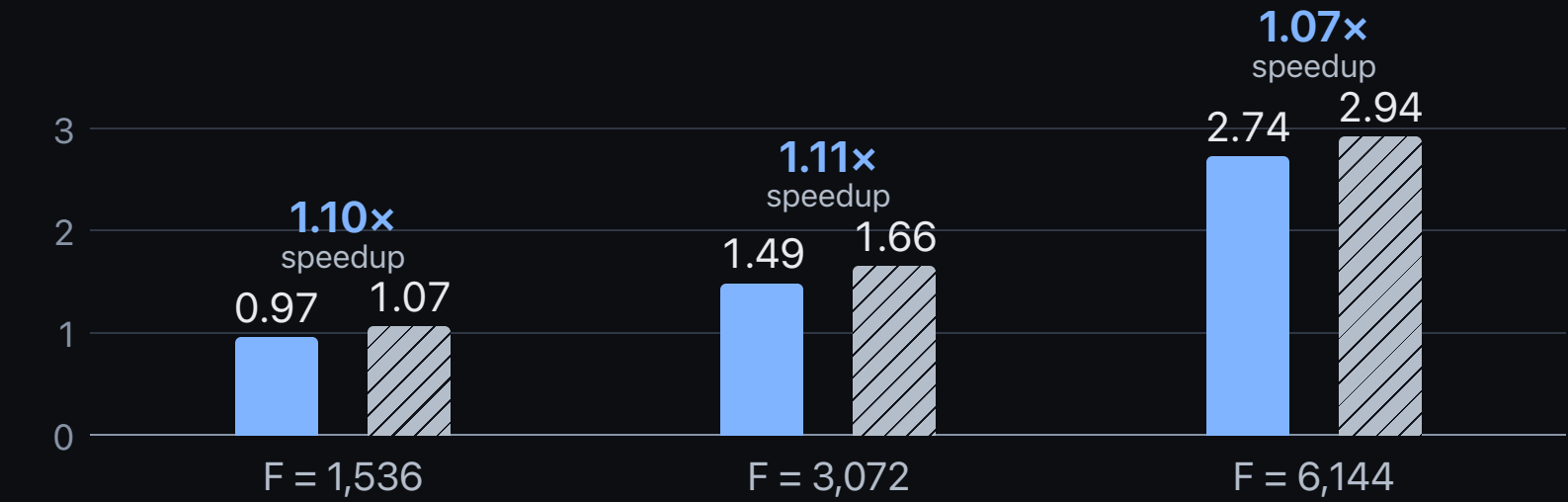
MoK vs Dist-MoE: Expert Width

Inference · EP16 · up to 1.34× speedup over MoK

One layer · balanced workload · F = 1,536 / 3,072 / 6,144

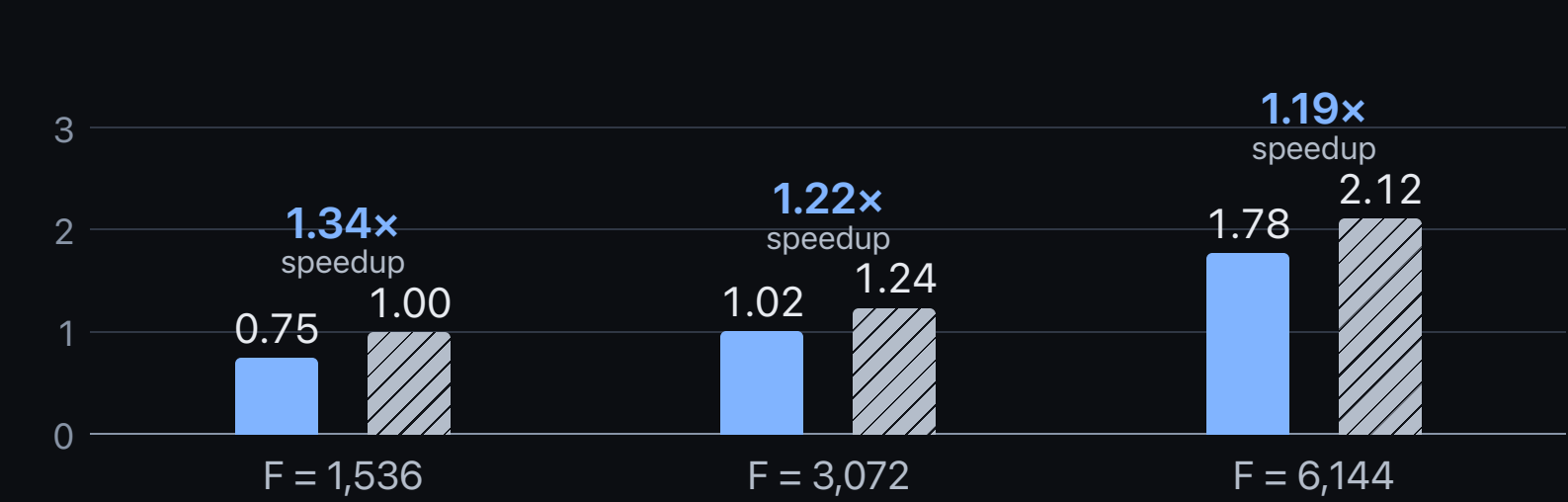
■ Dist-MoE ▨ MoK Columns: iteration ms ↓ · labels: speedup over MoK · table values: Dist / MoK

EP16 · Inference · BF16



	Iteration time · ms		
Dist / MoK	F = 1,536	F = 3,072	F = 6,144
Useful TFLOP/s / GPU	977 / 885	1,268 / 1,138	1,385 / 1,291
HBM model · TB/s	2.07 / 1.87	2.51 / 2.25	2.64 / 2.46
NVLink TX · GB/s	337.6 / 305.8	216.0 / 192.7	118.7 / 111.8
NVLink RX · GB/s	337.6 / 306.5	216.0 / 193.1	118.7 / 112.0

EP16 · Inference · MXFP8



	Iteration time · ms		
Dist / MoK	F = 1,536	F = 3,072	F = 6,144
Useful TFLOP/s / GPU	1,261 / 943	1,861 / 1,525	2,126 / 1,785
HBM model · TB/s	1.65 / 1.23	2.17 / 1.78	2.33 / 1.96
NVLink TX · GB/s	330.9 / 326.9	237.3 / 263.3	138.6 / 153.5
NVLink RX · GB/s	330.9 / 327.6	237.3 / 263.9	138.6 / 153.8

At F = 6,144, BF16 and MXFP8 deliver 1.07× and 1.19× speedup over MoK. Both precisions compare the same routed workload.

Matched shapes; one BF16 shared expert. TFLOP/s counts useful work; HBM is modeled; NVLink TX/RX is measured separately.

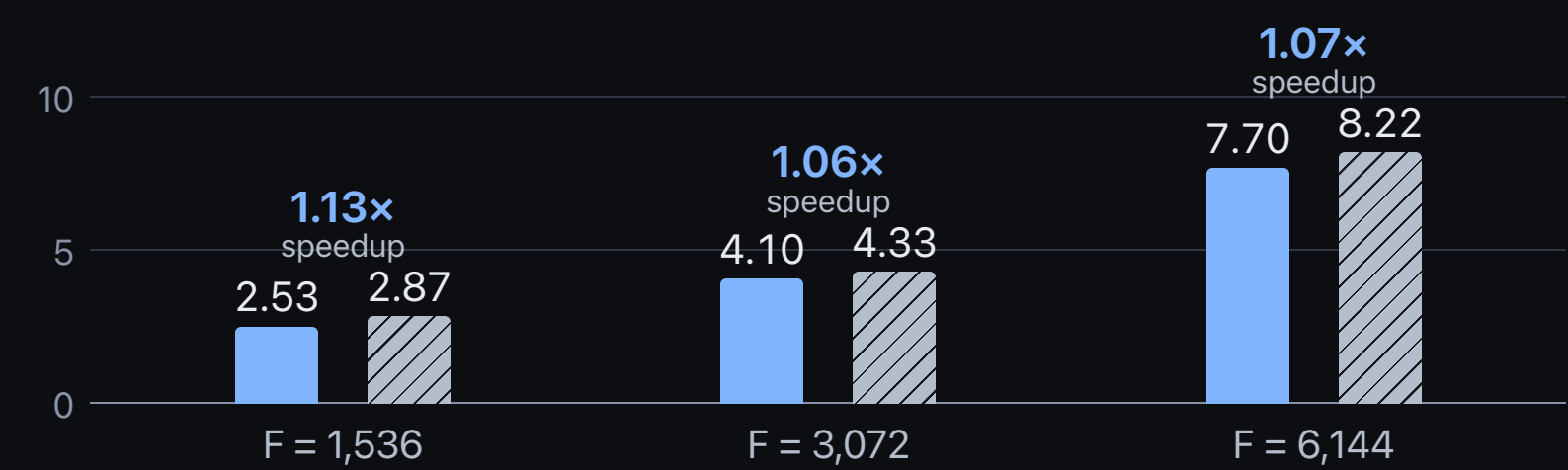
MoK vs Dist-MoE: Expert Width

Training · EP64 · up to 1.19× speedup over MoK

One layer · balanced workload · F = 1,536 / 3,072 / 6,144

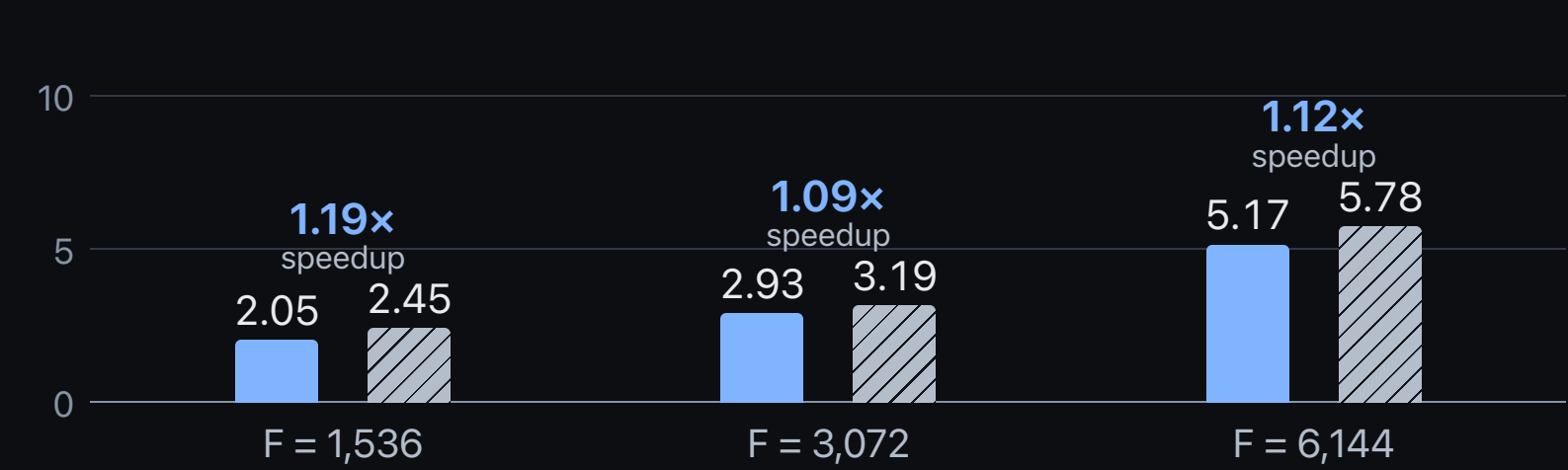
■ Dist-MoE ▨ MoK Columns: iteration ms ↓ · labels: speedup over MoK · table values: Dist / MoK

EP64 · Training · BF16



	Iteration time · ms		
Dist / MoK	F = 1,536	F = 3,072	F = 6,144
Useful TFLOP/s / GPU	1,122 / 990	1,387 / 1,311	1,477 / 1,382
HBM model · TB/s	0.97 / 0.85	1.00 / 0.95	0.96 / 0.90
NVLink TX · GB/s	274.4 / 244.2	168.7 / 161.2	91.8 / 85.7
NVLink RX · GB/s	274.4 / 245.3	168.7 / 162.0	91.8 / 86.1

EP64 · Training · MXFP8



	Iteration time · ms		
Dist / MoK	F = 1,536	F = 3,072	F = 6,144
Useful TFLOP/s / GPU	1,385 / 1,161	1,938 / 1,780	2,196 / 1,966
HBM model · TB/s	0.91 / 0.76	1.00 / 0.92	0.98 / 0.88
NVLink TX · GB/s	334.6 / 286.3	238.3 / 218.0	135.5 / 121.5
NVLink RX · GB/s	334.6 / 287.6	238.3 / 219.0	135.5 / 122.0

At F = 6,144, BF16 and MXFP8 deliver 1.07× and 1.12× speedup over MoK. Both precisions compare the same routed workload.

Matched shapes; one BF16 shared expert. TFLOP/s counts useful work; HBM is modeled; NVLink TX/RX is measured separately.

Public API

Configure the context

BF16 quickstart · configuration

```
T, D, F, K, E_local = 16, 256, 256, 2, 4
ep_size = dist.get_world_size(ep_group)
E = E_local * ep_size
```

```
config = dist_moe.Config(
    max_num_local_input_tokens=T,
    hidden_dim=D,
    intermediate_dim=F,
    top_k=K,
    num_experts=E,
    max_moe_layers_per_activation_slot=1,
    device_scratch_capacity_factor=1.0,
    activation_slot_capacity_factor=1.0,
    num_activation_slots=1,
)
```

One configuration sets routing dimensions, activation slots, and scratch budgets.

Public API

Inspect the memory plan

BF16 quickstart · inspect

```
plan = dist_moe.plan_memory(config, ep_size=ep_size)
print(plan.explain())
```

Printed output

```
Distributed MoE memory plan:
balanced/device/total rows: 32/32/32
device/total scratch capacity factors: 1/1
activation slots: 1 x 0.000 GiB (65,536 bytes) = 0.000 GiB (65,536 bytes) (balanced capacity factor 1)
device scratch: 0.000 GiB (98,304 bytes)
host overflow scratch: 0.000 GiB (0 bytes)
total device buffer: 0.000 GiB (163,840 bytes)
activation slot minimum/balanced full-save/maximum useful: 0.000 GiB (8,192 bytes) / 0.000 GiB (65,536 bytes) / 0.000 GiB (65,536 bytes)
virtual address range: 0.000 GiB (163,840 bytes); all executable routing fits without recompute
activation slots are a soft limit (recompute on exhaustion); total scratch capacity is a hard limit
```

`plan_memory` resolves the logical allocation. `plan.explain()` reports its bytes, row bounds, and recompute policy before the context allocates storage.

Public API

Create the context and execute

BF16 quickstart · execution

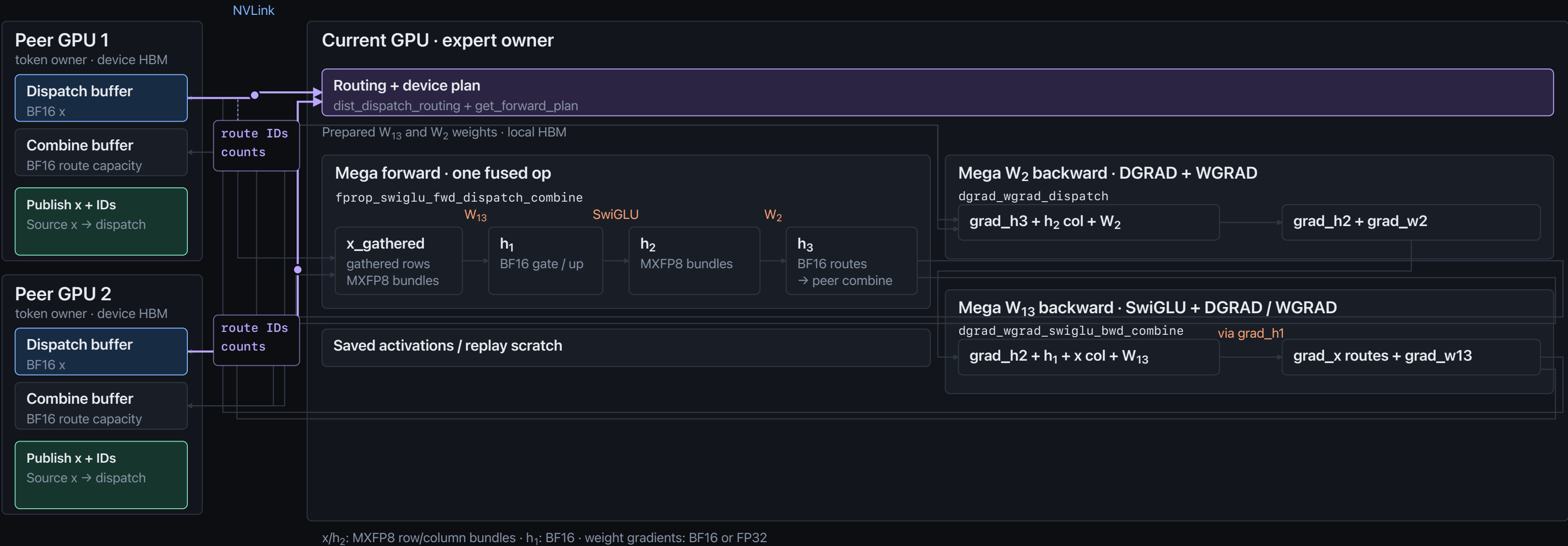
```
context = dist_moe.create_context(  
    group=ep_group,  
    config=config,  
    device=device,  
)  
  
try:  
    output_TD = dist_moe.routed_experts(  
        x_TD, topk_expert_ids_TK, topk_scores_TK,  
        w13_E2FD, w2_EDF, context,  
    )  
    output_TD.float().sum().backward()  
finally:  
    context.close()
```

The caller provides token rows, route IDs/scores, and expert weights. The matching backward runs before `context.close()`.

Dist-MoE: High-level End-to-End Execution

Publish inputs, route tokens, and plan storage

MXFP8 Mega training · forward + backward · eight op-level steps



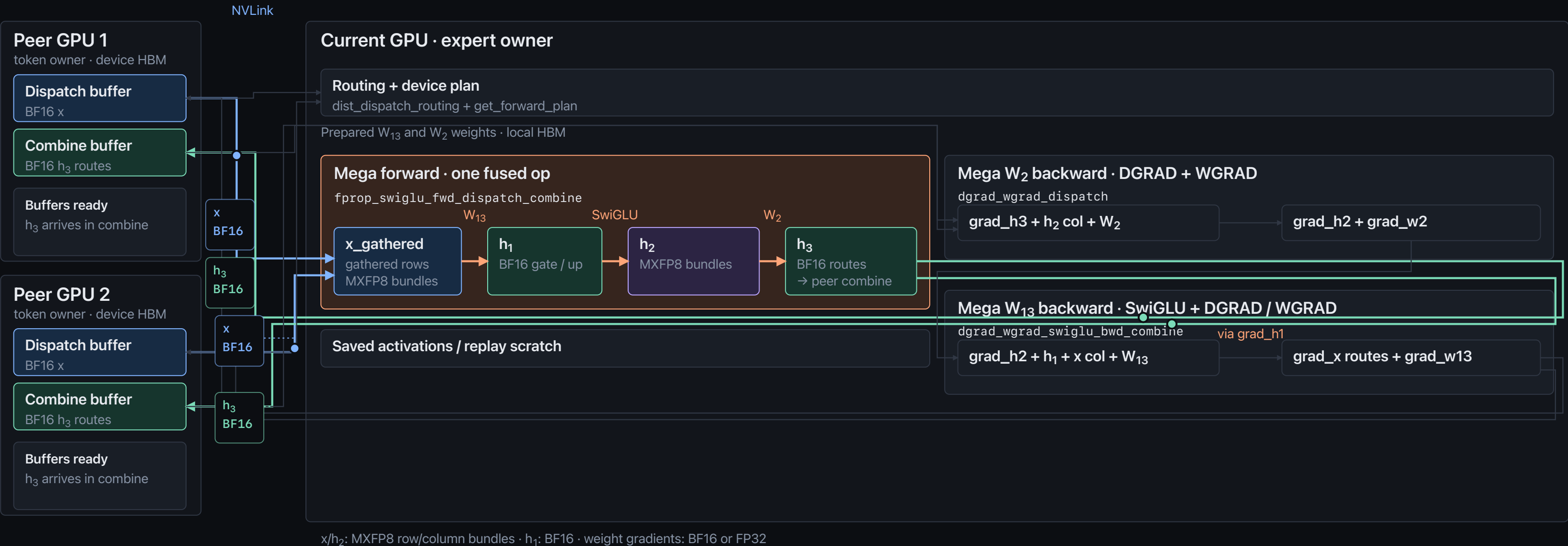
Each token owner publishes BF16 x and expert IDs in symmetric HBM. Routing builds gather/scatter pointers; get_forward_plan selects saved-state and scratch views.

MXFP8 Mega training · tensor boxes show op values, not separate allocations. The three fused compute labels omit the common dist_blockscaled_grouped_gemm_ prefix.

Dist-MoE: High-level End-to-End Execution

Run the fused MXFP8 Mega forward op

MXFP8 Mega training · forward + backward · eight op-level steps



fprop_swiglu_fwd_dispatch_combine gathers and quantizes x, computes W13, SwiGLU, and W2, then scatters BF16 h3 to peer combine buffers. Gathered x and h2 use MXFP8 bundles; h1 is BF16.

MXFP8 Mega training · tensor boxes show op values, not separate allocations. The three fused compute labels omit the common dist_blockscaled_grouped_gemm_ prefix.

Dist-MoE: High-level End-to-End Execution

Reduce expert outputs on each token owner

MXFP8 Mega training · forward + backward · eight op-level steps



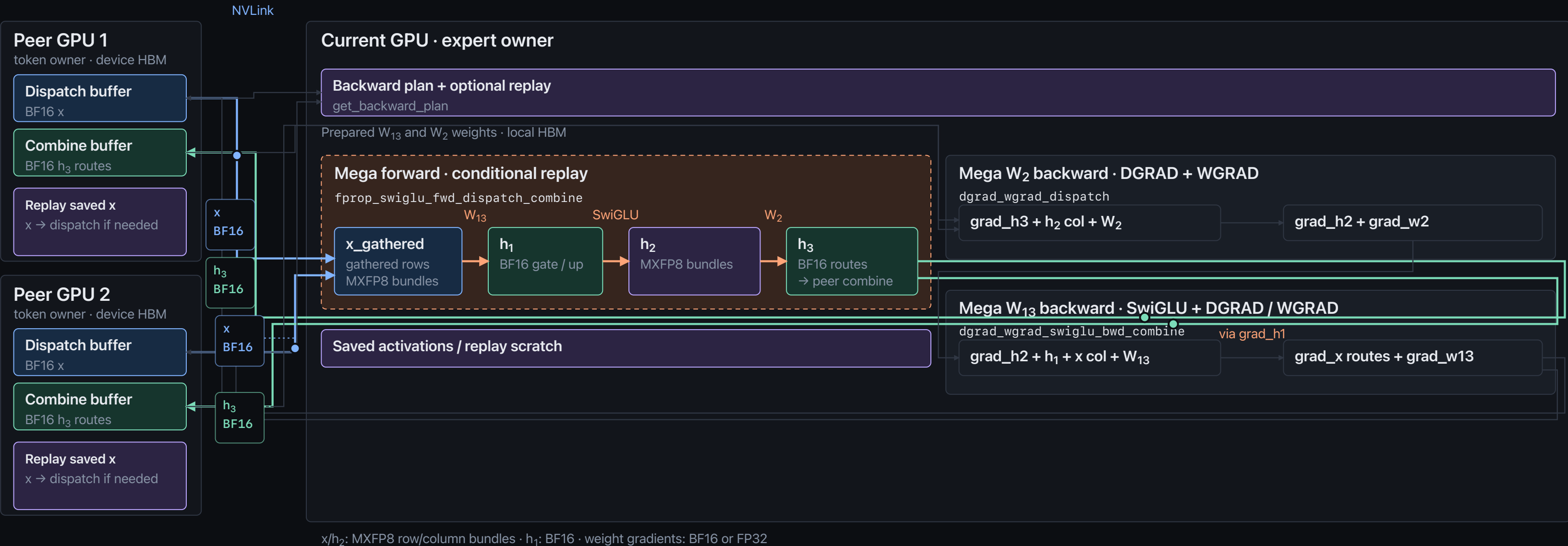
After the combine barrier, scale_and_sum weights the h3 routes by their router scores and produces local y.

MXFP8 Mega training · tensor boxes show op values, not separate allocations. The three fused compute labels omit the common `dist_blockscaled_grouped_gemm_` prefix.

Dist-MoE: High-level End-to-End Execution

Select saved state or replay into scratch

MXFP8 Mega training · forward + backward · eight op-level steps



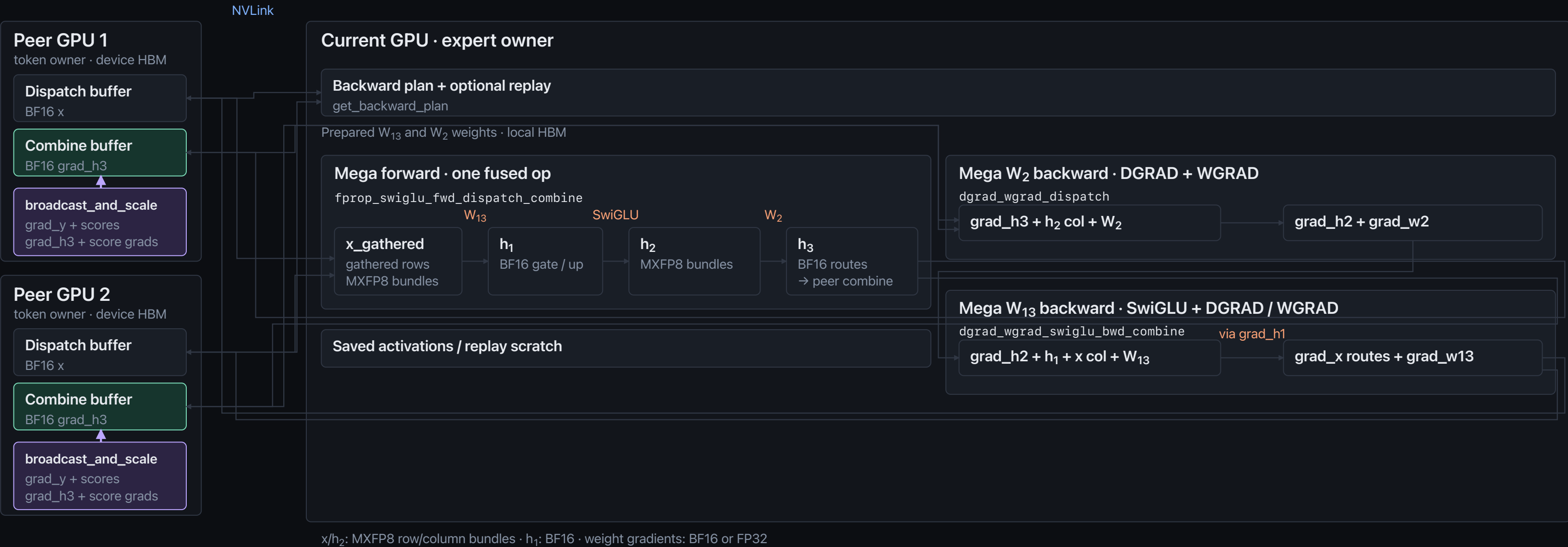
get_backward_plan selects retained intermediates or conditional replay. When replay is needed, saved source x is republished and the same Mega forward op regenerates the missing state into scratch.

MXFP8 Mega training · tensor boxes show op values, not separate allocations. The three fused compute labels omit the common dist_blockscaled_grouped_gemm_ prefix.

Dist-MoE: High-level End-to-End Execution

Publish route-output gradients

MXFP8 Mega training · forward + backward · eight op-level steps



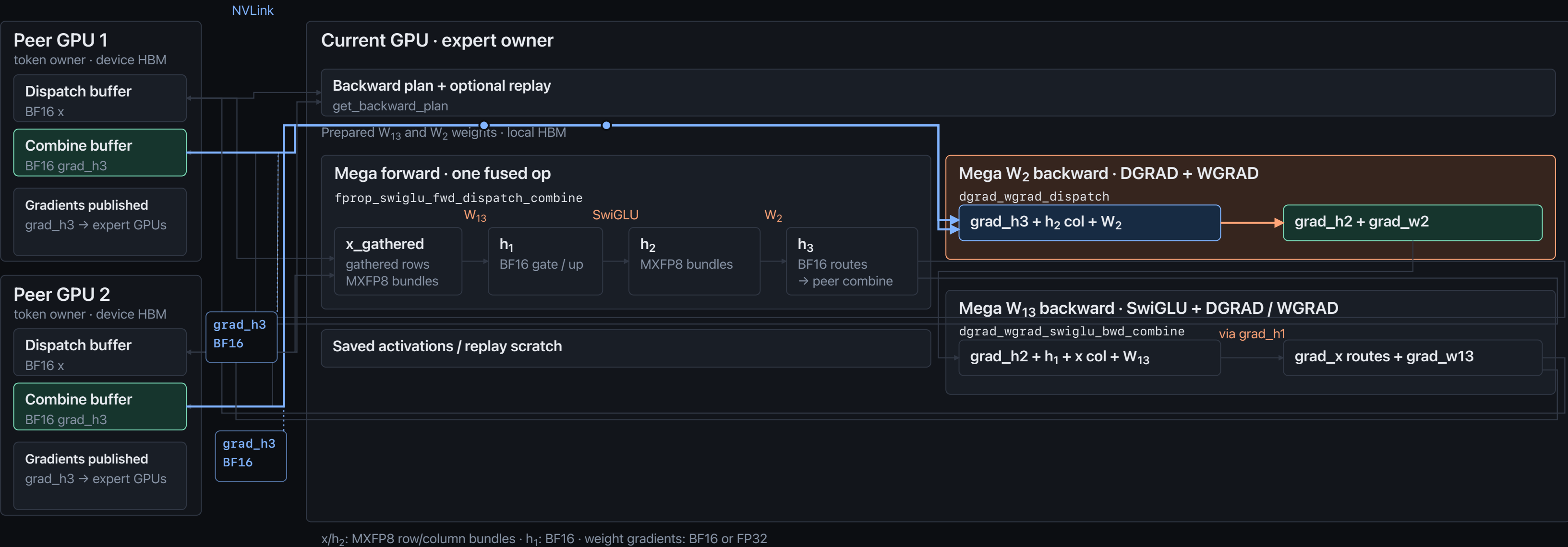
broadcast_and_scale uses grad_y, router scores, and h3 to form grad_h3 and grad_topk_scores. BF16 grad_h3 is published in peer combine HBM before the backward gather.

MXFP8 Mega training · tensor boxes show op values, not separate allocations. The three fused compute labels omit the common dist_blockscaled_grouped_gemm_ prefix.

Dist-MoE: High-level End-to-End Execution

Run the fused W2 DGRAD/WGRAD op

MXFP8 Mega training · forward + backward · eight op-level steps



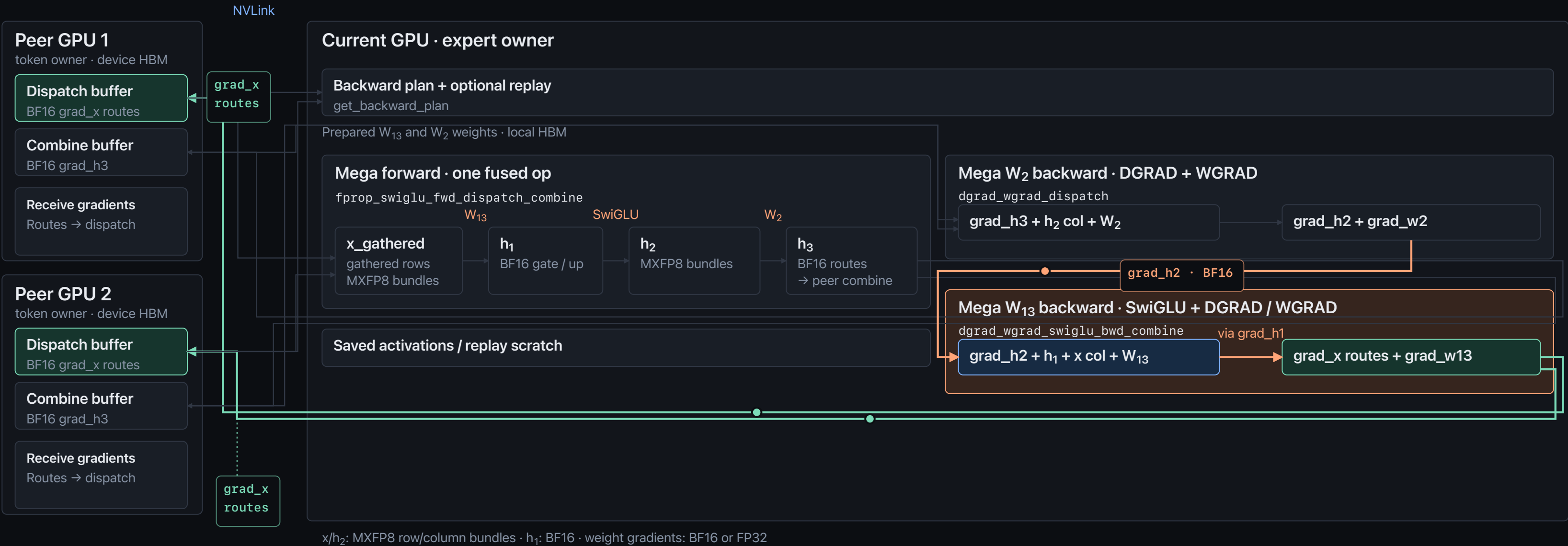
dgrad_wgrad_dispatch gathers grad_h3 and reads the saved or regenerated h2 column bundle and W2. It produces BF16 grad_h2 and local grad_w2.

MXFP8 Mega training · tensor boxes show op values, not separate allocations. The three fused compute labels omit the common dist_blockscaled_grouped_gemm_ prefix.

Dist-MoE: High-level End-to-End Execution

Run fused SwiGLU and W13 backward

MXFP8 Mega training · forward + backward · eight op-level steps



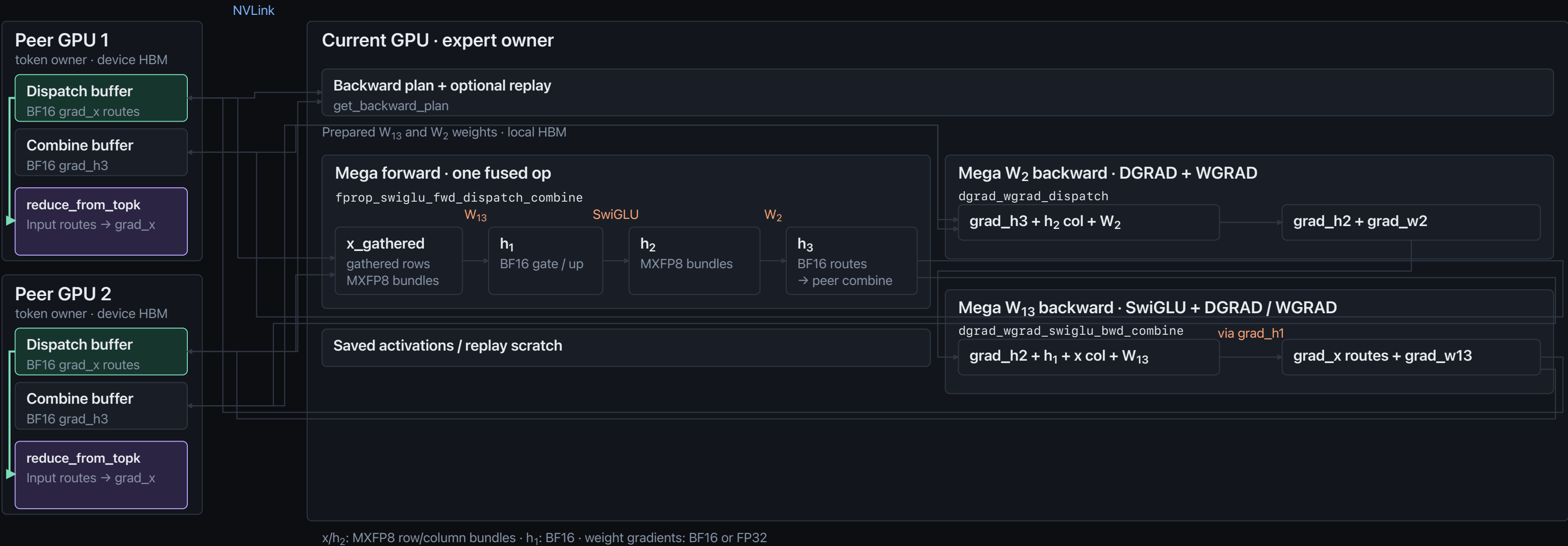
dgrad_wgrad_swiglu_bwd_combine reads grad_h2, h1, x column bundles, and W13. It computes SwiGLU backward and W13 DGRAD/WGRAD, stores local grad_w13, and scatters BF16 input-gradient routes to peer dispatch HBM.

MXFP8 Mega training · tensor boxes show op values, not separate allocations. The three fused compute labels omit the common dist_blockscaled_grouped_gemm_ prefix.

Dist-MoE: High-level End-to-End Execution

Reduce input-gradient routes on each owner

MXFP8 Mega training · forward + backward · eight op-level steps



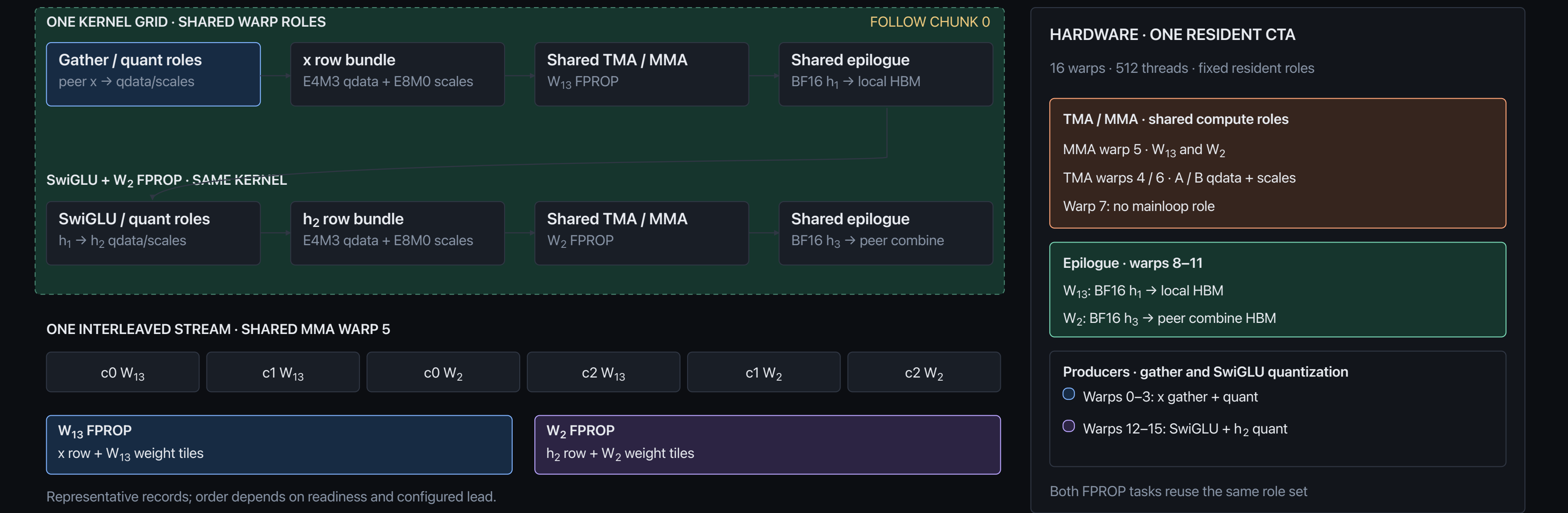
After the dispatch barrier, reduce_from_topk combines the peer-written routes into local BF16 grad_x. Weight gradients remain with the expert owner.

MXFP8 Mega training · tensor boxes show op values, not separate allocations. The three fused compute labels omit the common dist_blockscaled_grouped_gemm_ prefix.

MXFP8 Mega Forward

Gather and quantize x

MXFP8 · Mega forward · 3 × 512-row chunks · D = F = 256 · follow chunk 0



MXFP8 Mega Forward

W13: FPROP on the shared MMA warp

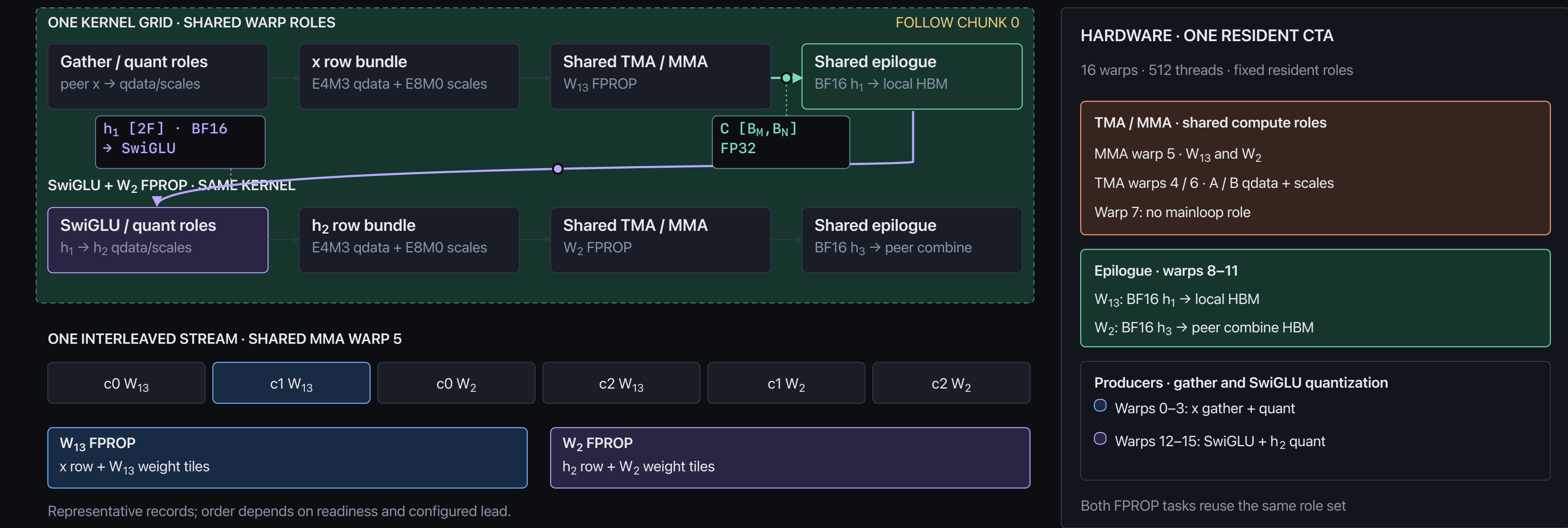
MXFP8 · Mega forward · 3 × 512-row chunks · D = F = 256 · follow chunk 0



MXFP8 Mega Forward

Publish h_1 and quantize SwiGLU output

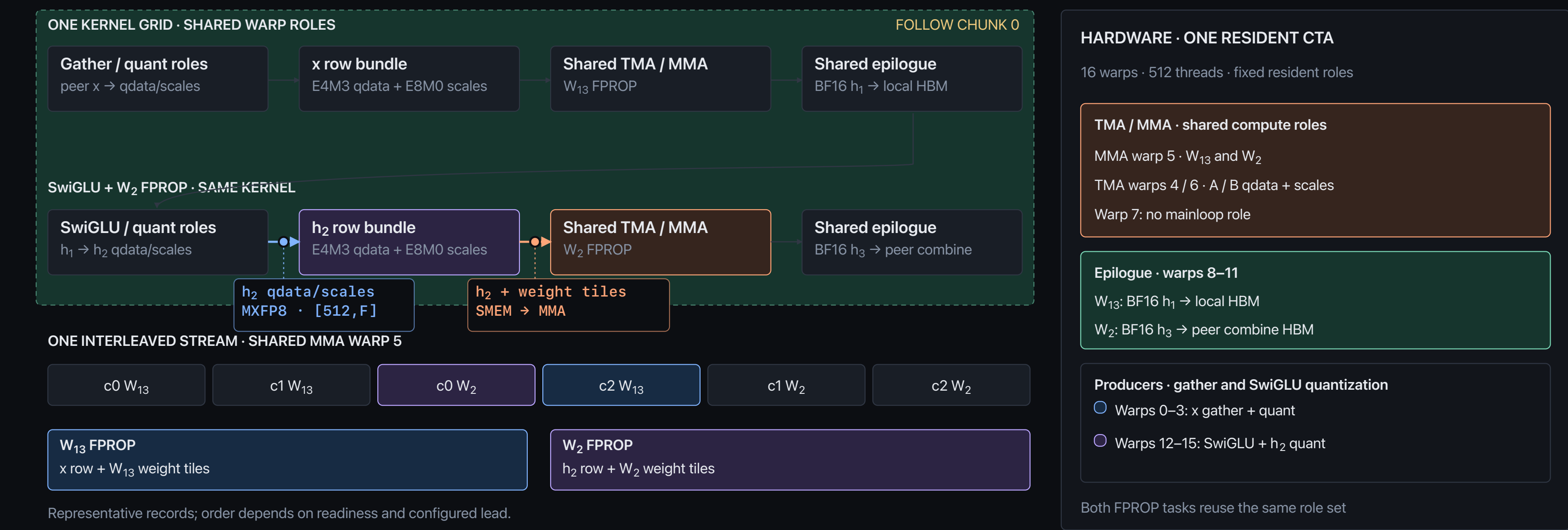
MXFP8 · Mega forward · 3×512 -row chunks · $D = F = 256$ · follow chunk 0



MXFP8 Mega Forward

W2: FPROP on that same MMA warp

MXFP8 · Mega forward · 3 × 512-row chunks · D = F = 256 · follow chunk 0



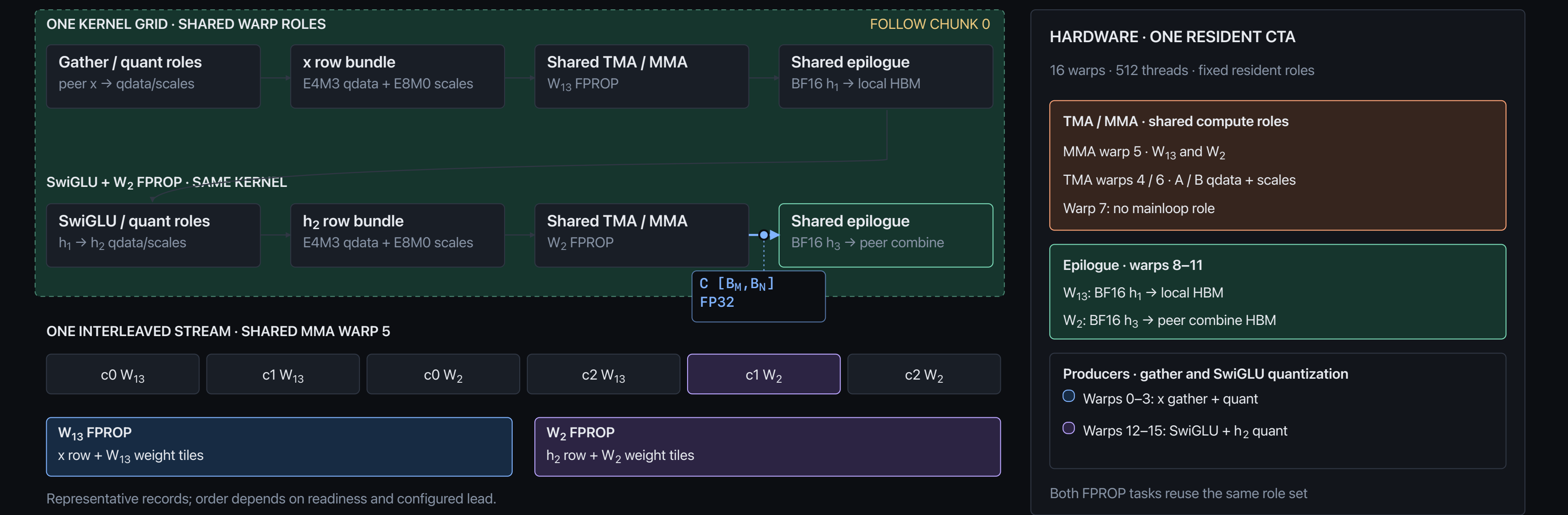
chunk 0: W2; chunk 1: SwiGLU/quant; chunk 2: W13. An h1 tile must be complete before its SwiGLU producer reads it; h2 qdata/scales must be ready before W2 consumes them.

W13/W2 work records share one TMA/MMA warp group and epilogue roles. Record order and exact occupancy depend on configuration.

MXFP8 Mega Forward

Scatter h3 to peer combine HBM

MXFP8 · Mega forward · 3 × 512-row chunks · D = F = 256 · follow chunk 0



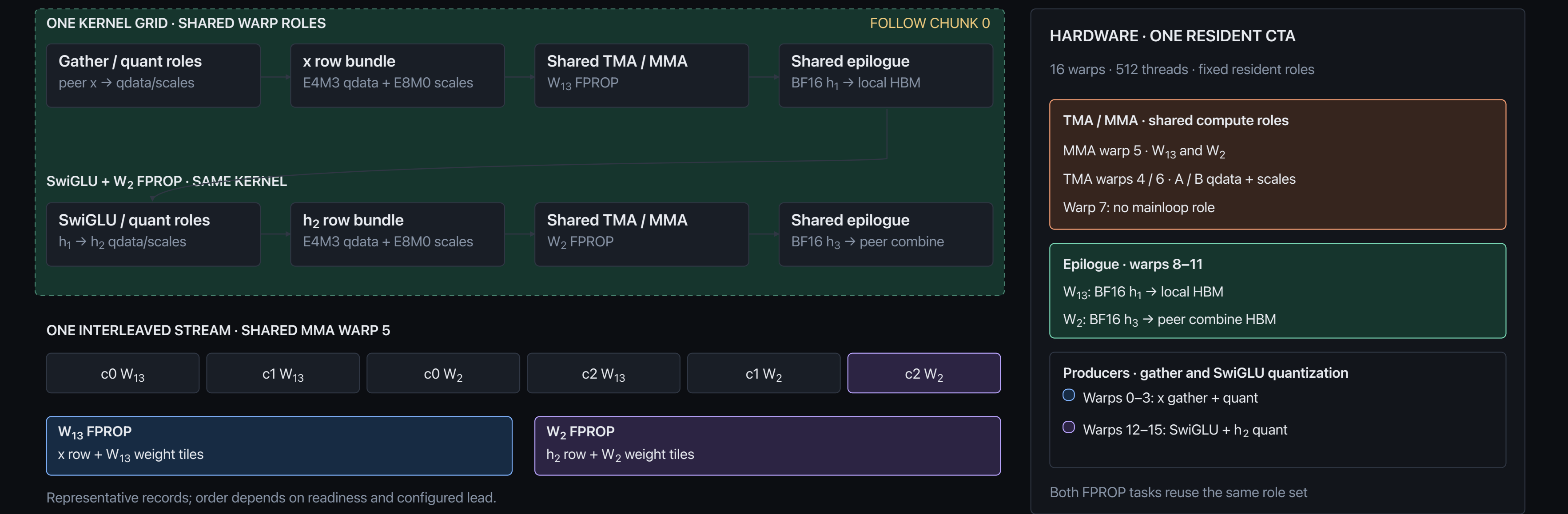
chunk 0: scatter; chunk 1: W_2 ; chunk 2: SwiGLU/quant. An h_1 tile must be complete before its SwiGLU producer reads it; h_2 qdata/scales must be ready before W_2 consumes them.

W_{13}/W_2 work records share one TMA/MMA warp group and epilogue roles. Record order and exact occupancy depend on configuration.

MXFP8 Mega Forward

Pipeline the remaining chunks

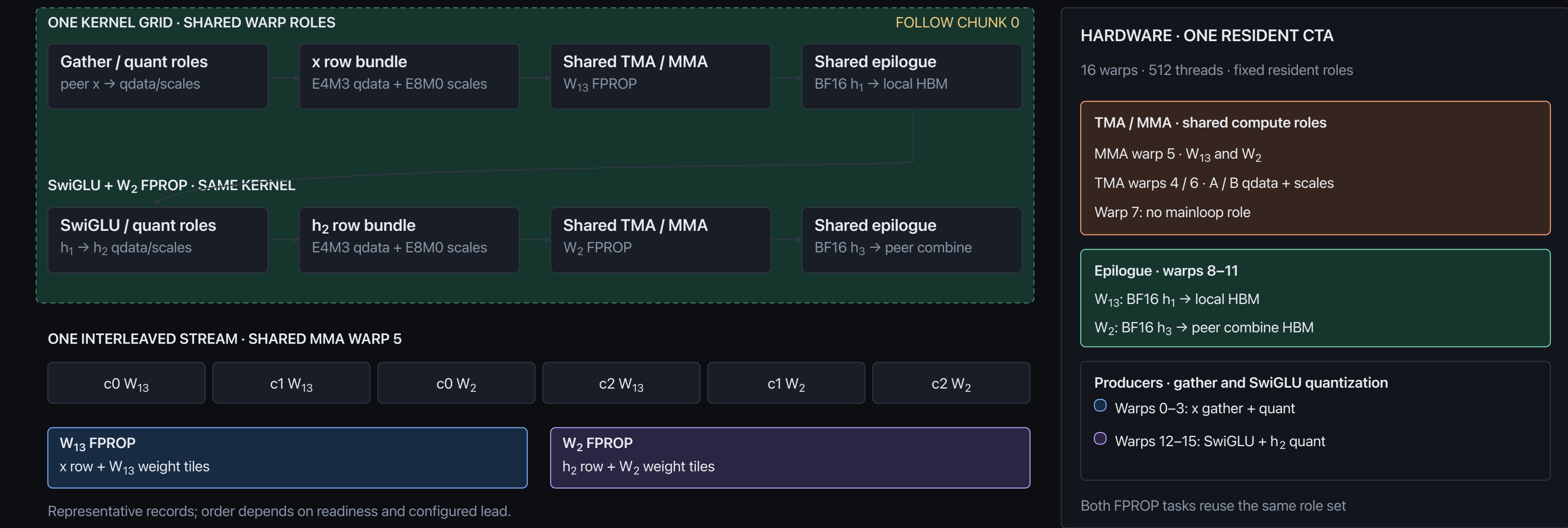
MXFP8 · Mega forward · 3 × 512-row chunks · $D = F = 256$ · follow chunk 0



MXFP8 Mega Forward

Complete the peer stores

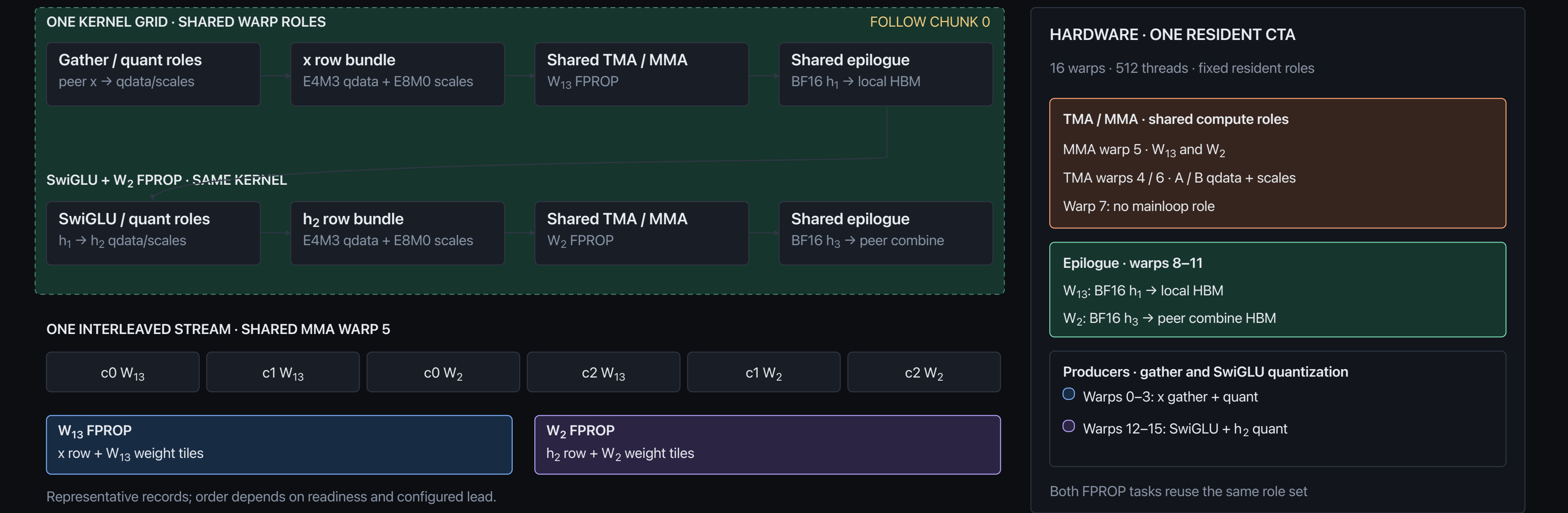
MXFP8 · Mega forward · 3 × 512-row chunks · D = F = 256 · follow chunk 0



MXFP8 Mega Forward

Finish the fused forward kernel

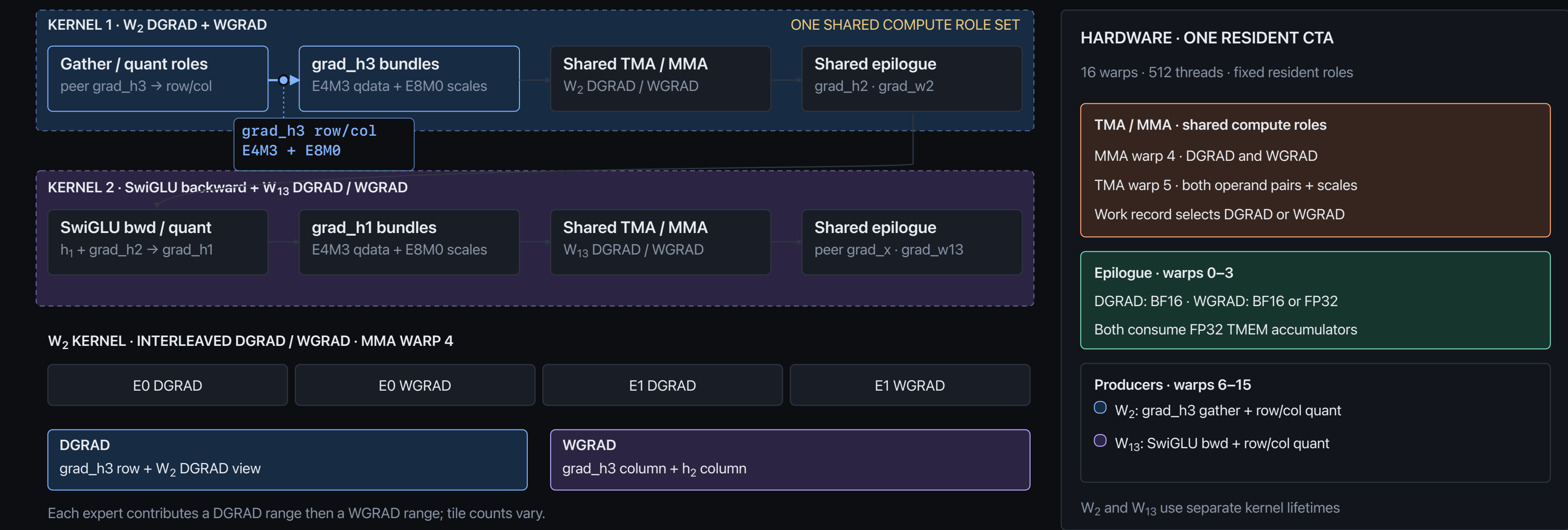
MXFP8 · Mega forward · 3 × 512-row chunks · D = F = 256 · follow chunk 0



MXFP8 Mega Backward

Gather and quantize grad_h3

MXFP8 · Mega backward · W₂ kernel then W₁₃ kernel · row/column operands



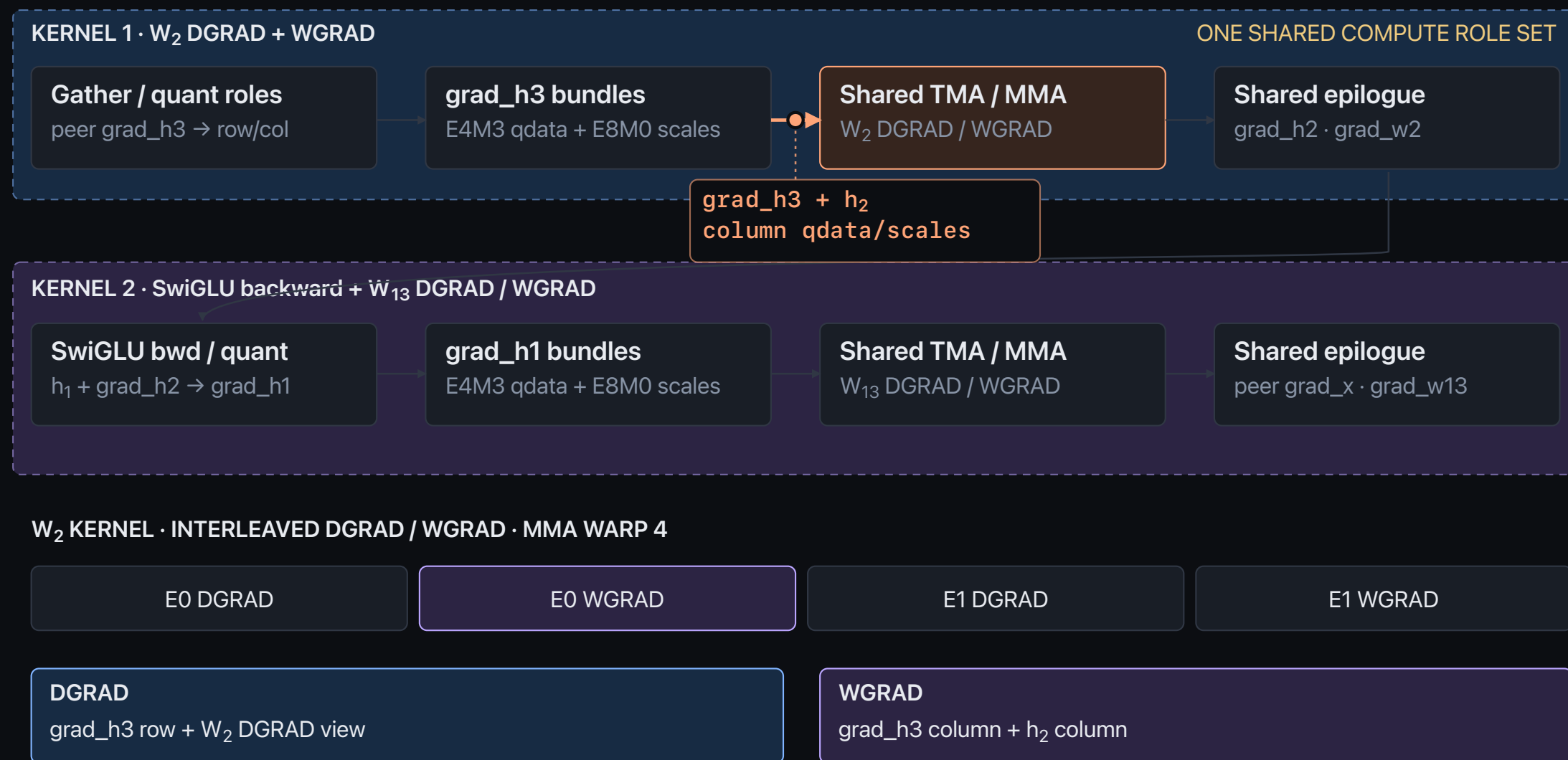
Producer warps gather BF16 route gradients from peer combine HBM and publish separate row/column MXFP8 qdata plus E8M0 scales.

W2 then W13: separate kernel lifetimes. Each kernel interleaves per-expert DGRAD/WGRAD ranges on shared roles; records are representative, not an instruction trace.

MXFP8 Mega Backward

W2: WGRAD on that same MMA warp

MXFP8 · Mega backward · W2 kernel then W13 kernel · row/column operands



Each expert contributes a DGRAD range then a WGRAD range; tile counts vary.

HARDWARE · ONE RESIDENT CTA

16 warps · 512 threads · fixed resident roles

TMA / MMA · shared compute roles

MMA warp 4 · DGRAD and WGRAD

TMA warp 5 · both operand pairs + scales

Work record selects DGRAD or WGRAD

Epilogue · warps 0–3

DGRAD: BF16 · WGRAD: BF16 or FP32

Both consume FP32 TMEM accumulators

Producers · warps 6–15

- W₂: grad_h3 gather + row/col quant
- W₁₃: SwiGLU bwd + row/col quant

W_2 and W_{13} use separate kernel lifetimes

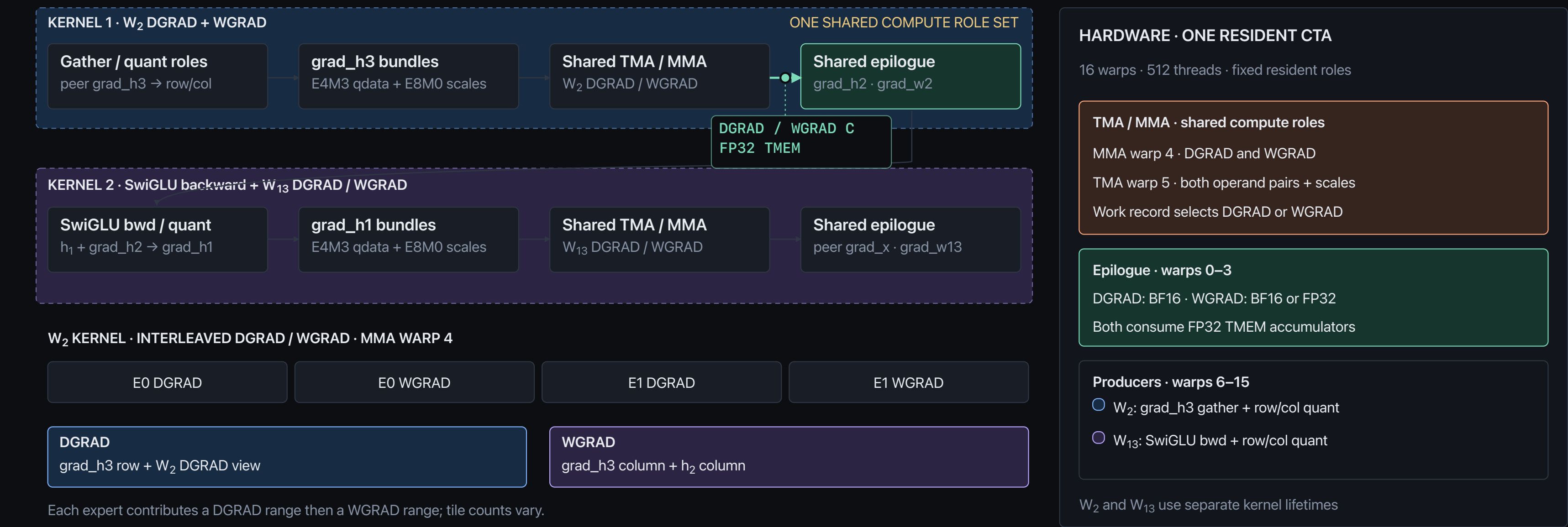
A WGRAD record selects column `grad_h3` and saved or regenerated column `h2`. The same TMA, MMA, and epilogue roles serve both kinds of record.

W2 then W13: separate kernel lifetimes. Each kernel interleaves per-expert DGRAD/WGRAD ranges on shared roles; records are representative, not an instruction trace.

MXFP8 Mega Backward

Finish the W2 DGRAD/WGRAD kernel

MXFP8 · Mega backward · W2 kernel then W13 kernel · row/column operands



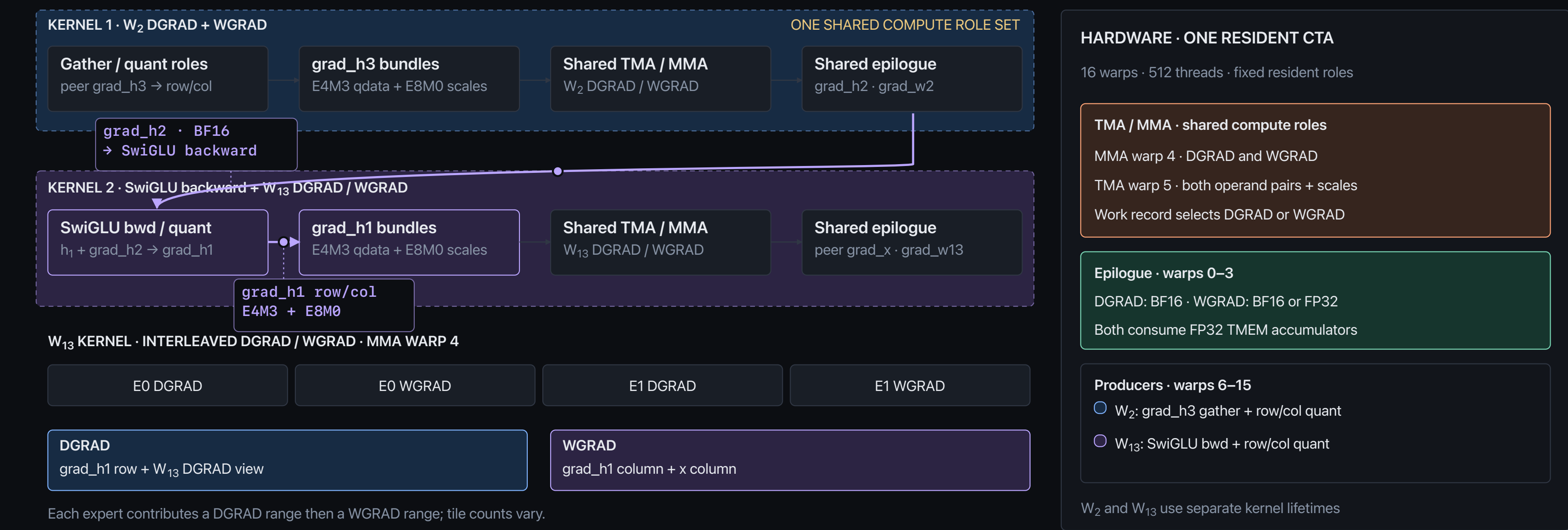
The epilogue writes BF16 grad_h2 and local BF16 or FP32 grad_w2. This fused launch completes before the W13 launch begins.

W2 then W13: separate kernel lifetimes. Each kernel interleaves per-expert DGRAD/WGRAD ranges on shared roles; records are representative, not an instruction trace.

MXFP8 Mega Backward

SwiGLU backward publishes grad_h1

MXFP8 · Mega backward · W2 kernel then W13 kernel · row/column operands



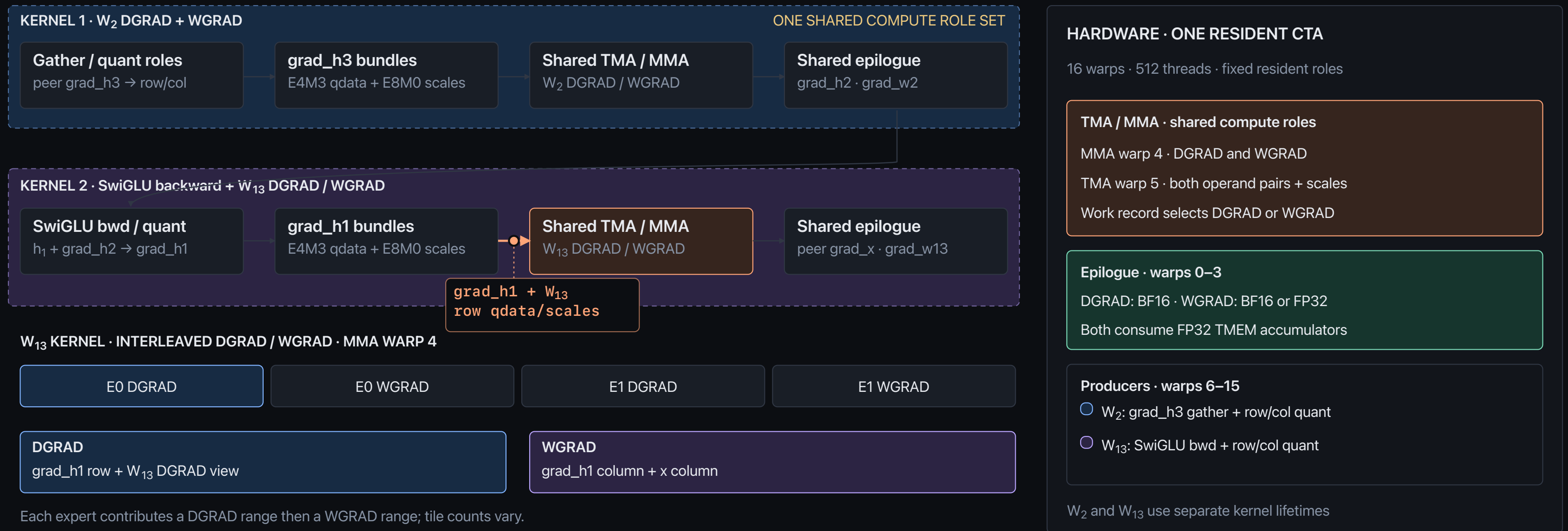
The next kernel reads BF16 h1 and grad_h2, evaluates the derivative in FP32 registers, then publishes row/column grad_h1 qdata and scales.

W2 then W13: separate kernel lifetimes. Each kernel interleaves per-expert DGRAD/WGRAD ranges on shared roles; records are representative, not an instruction trace.

MXFP8 Mega Backward

W13: DGRAD on the shared MMA warp

MXFP8 · Mega backward · W2 kernel then W13 kernel · row/column operands



A DGRAD record selects row grad_h1 and W13's DGRAD weight view. This kernel's warp 4 accumulates route input-gradient tiles in FP32 TMEM.

MXFP8 Mega Backward

W13: WGRAD on that same MMA warp

MXFP8 · Mega backward · W2 kernel then W13 kernel · row/column operands



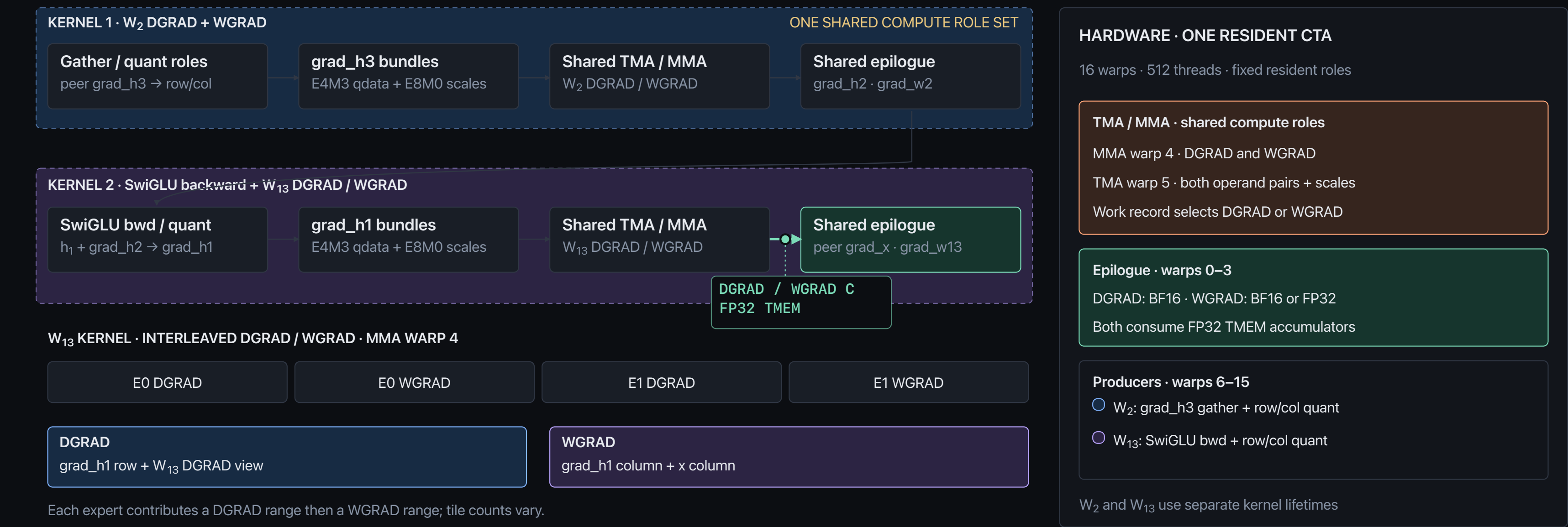
A WGRAD record selects column grad_h1 and saved or regenerated column x. The same warp 4 accumulates local expert-weight gradients.

W2 then W13: separate kernel lifetimes. Each kernel interleaves per-expert DGRAD/WGRAD ranges on shared roles; records are representative, not an instruction trace.

MXFP8 Mega Backward

Scatter route gradients and store grad_w13

MXFP8 · Mega backward · W2 kernel then W13 kernel · row/column operands

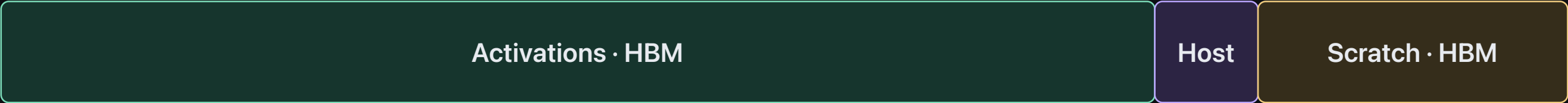


Activation, Scratch, and VMM

Saved-state capacity selects replay; scratch capacity bounds execution

ONE GPU VIRTUAL ADDRESS RANGE

Physical backing is fixed at setup



Saved state grows →

← scratch grows

○ Activation slots

Device HBM

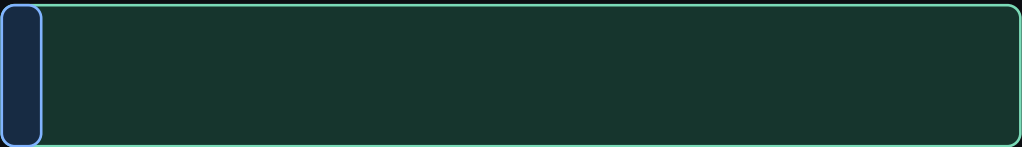
○ Overflow scratch

Pinned host · VMM

○ Shared scratch

Device HBM

ACTIVATIONS · SOFT BUDGET



Inputs needed for replay

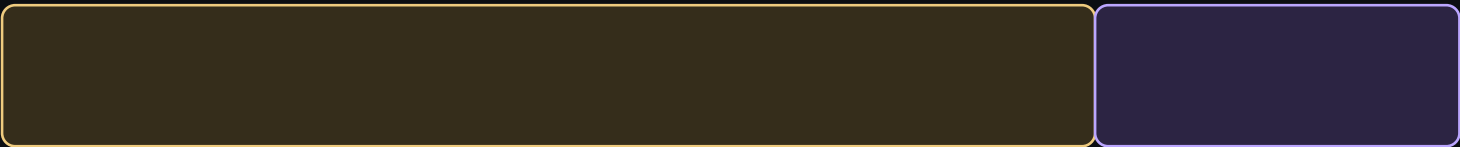
Optional-state capacity

Slot factor 0.35 scales optional saved bytes

Relative to balanced routing, above the input minimum

Less space selects recompute, without spilling to host

SCRATCH · HARD EXECUTION BOUND



Device factor 1.5x · hot HBM capacity

VMM total factor 2x · HBM + host capacity

Factors bound received rows relative to balanced routing

Past the device bound: slower host access with VMM

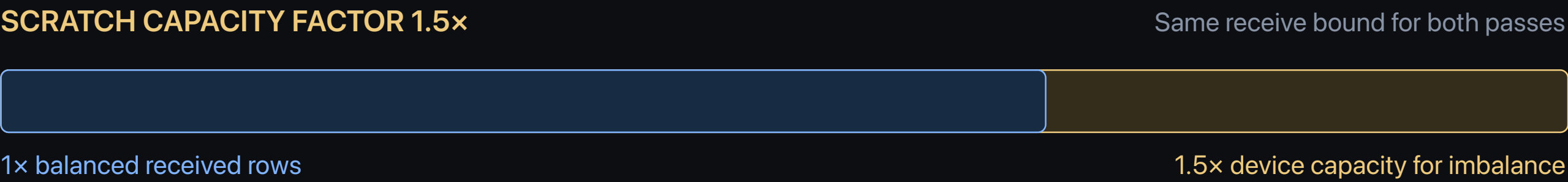
Past the total bound: execution error

The activation-slot factor controls optional saved bytes across layers. Scratch factors independently reserve capacity for per-call routing imbalance.

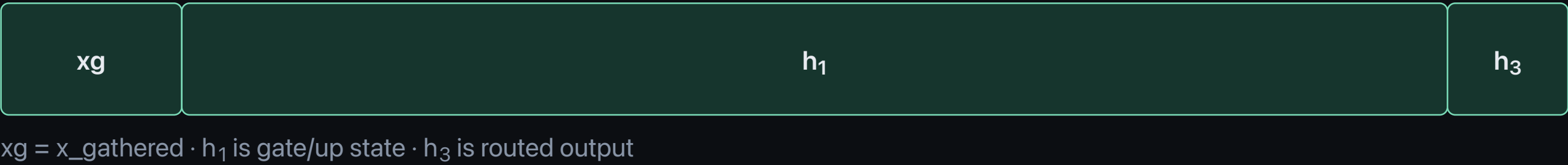
Logical layout · VMM fixes pinned-host backing at setup. Saved activations and separate symmetric communication buffers stay in HBM.

Forward and Backward Memory

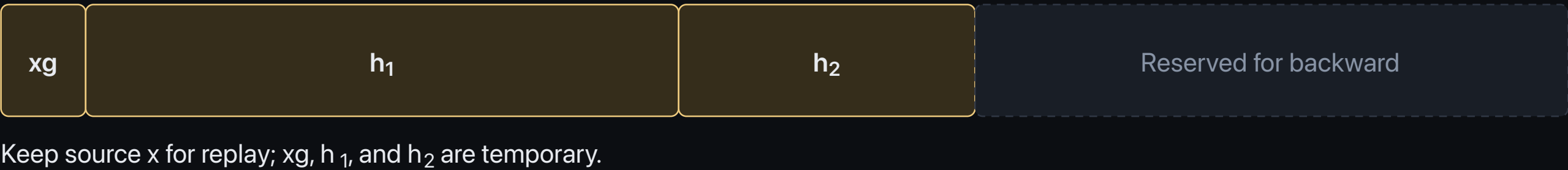
Forward: produce state and temporary operands



IF STATE FITS · SAVED UNTIL BACKWARD



FORWARD SCRATCH · WHEN ELIGIBLE STATE IS OMITTED



Saved state accumulates across layers

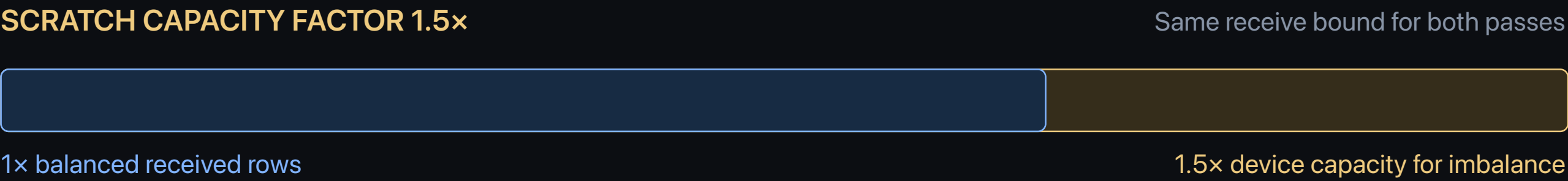
One scratch region covers the peak call

A layer can retain its eligible state or keep only its input for replay. Scratch covers the temporary working set.

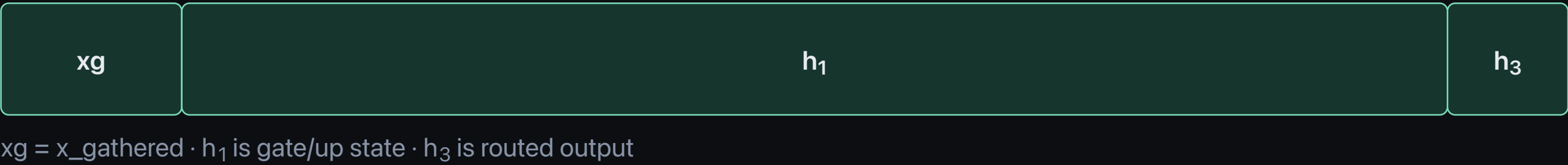
BF16 example · bar widths follow the shared byte model. Scratch is sized per call, not multiplied by the layer count.

Forward and Backward Memory

Backward: replay, compute gradients, release



IF STATE FITS · SAVED UNTIL BACKWARD



BACKWARD SCRATCH · REPLAY + GRADIENTS



Saved state accumulates across layers

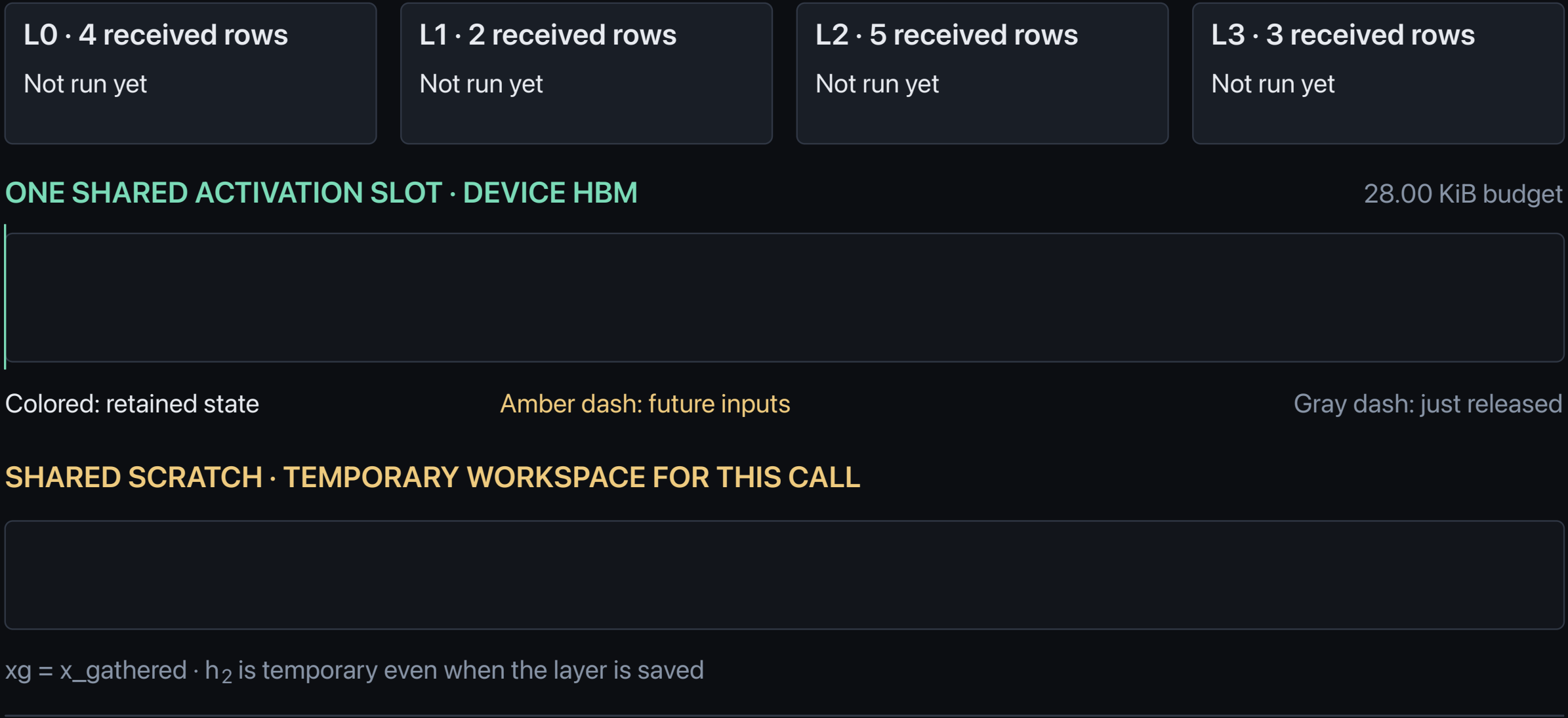
One scratch region covers the peak call

Reserve scratch for the backward recompute peak. Every sequential layer uses this same workspace.

BF16 example · bar widths follow the shared byte model. Scratch is sized per call, not multiplied by the layer count.

Activation Packing and Dynamic Recompute

Four layers share one activation slot



One budget covers all four layers.
Reserve enough room for every input needed to replay.

One slot packs variable-size layer states; one scratch region serves each call.

BF16 · one inspected GPU · both EP ranks agree on each save/replay decision · no host scratch in this example.

Activation Packing and Dynamic Recompute

Forward L0: keep eligible state



This layer’s state fits, including room for future inputs.
Keep its state until backward; scratch is reused by the next call.

L0 starts the packed slot. Space for the remaining replay inputs stays protected.

BF16 · one inspected GPU · both EP ranks agree on each save/replay decision · no host scratch in this example.

Activation Packing and Dynamic Recompute

Forward L1: keep eligible state



Activation Packing and Dynamic Recompute

Forward L2: keep eligible state



This layer's state fits, including room for future inputs.

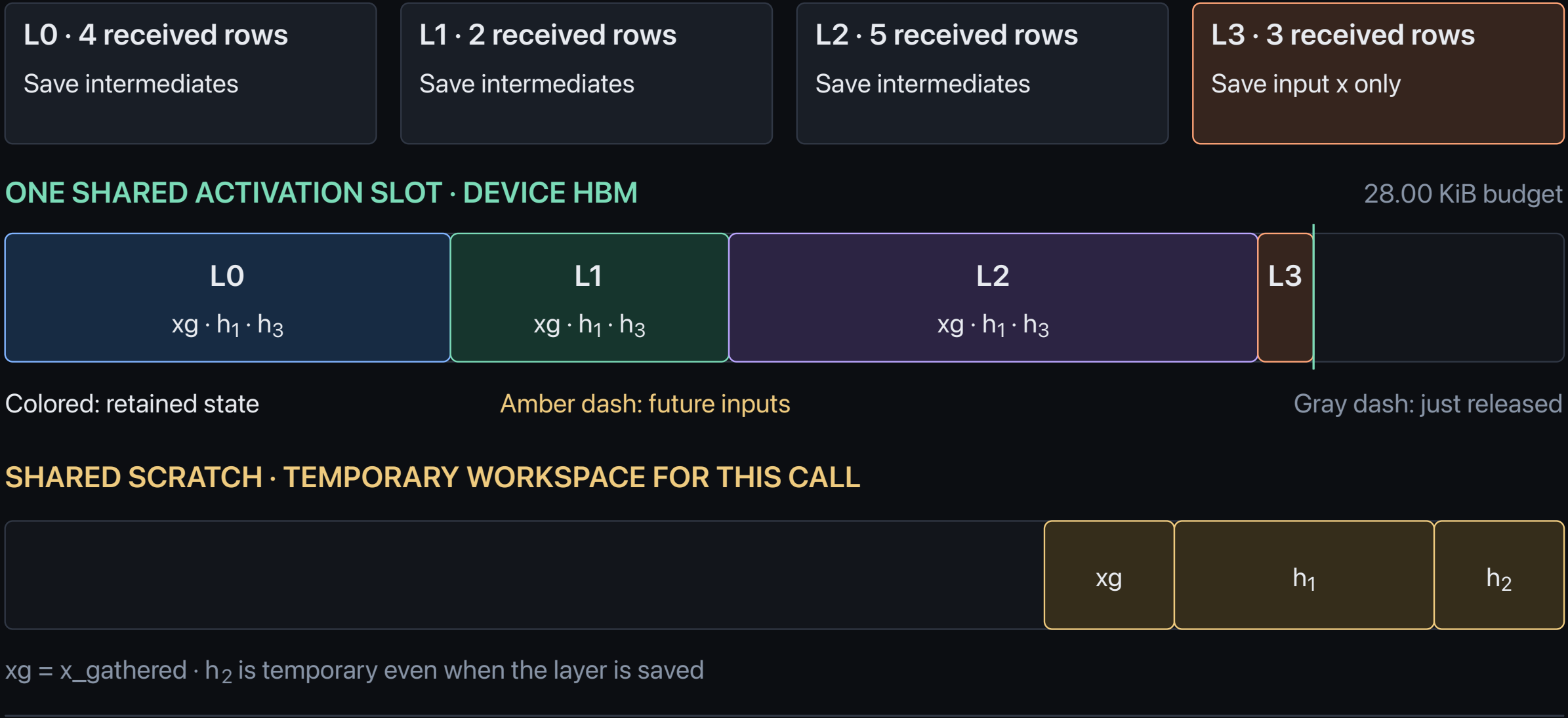
Keep its state until backward; scratch is reused by the next call.

L2 packs its state alongside 2 earlier layers. Space for remaining replay inputs stays protected.

BF16 · one inspected GPU · both EP ranks agree on each save/replay decision · no host scratch in this example.

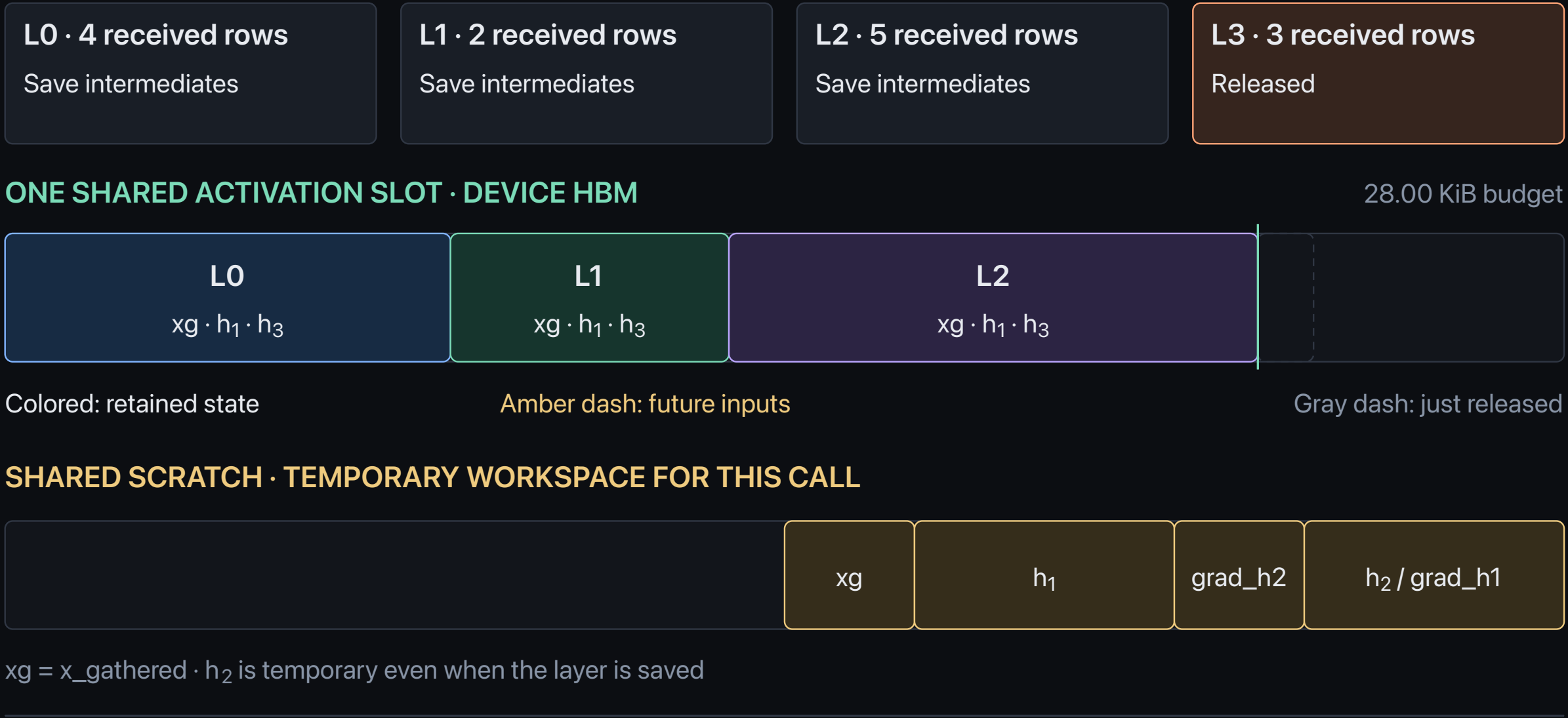
Activation Packing and Dynamic Recompute

Forward L3: save x for recompute



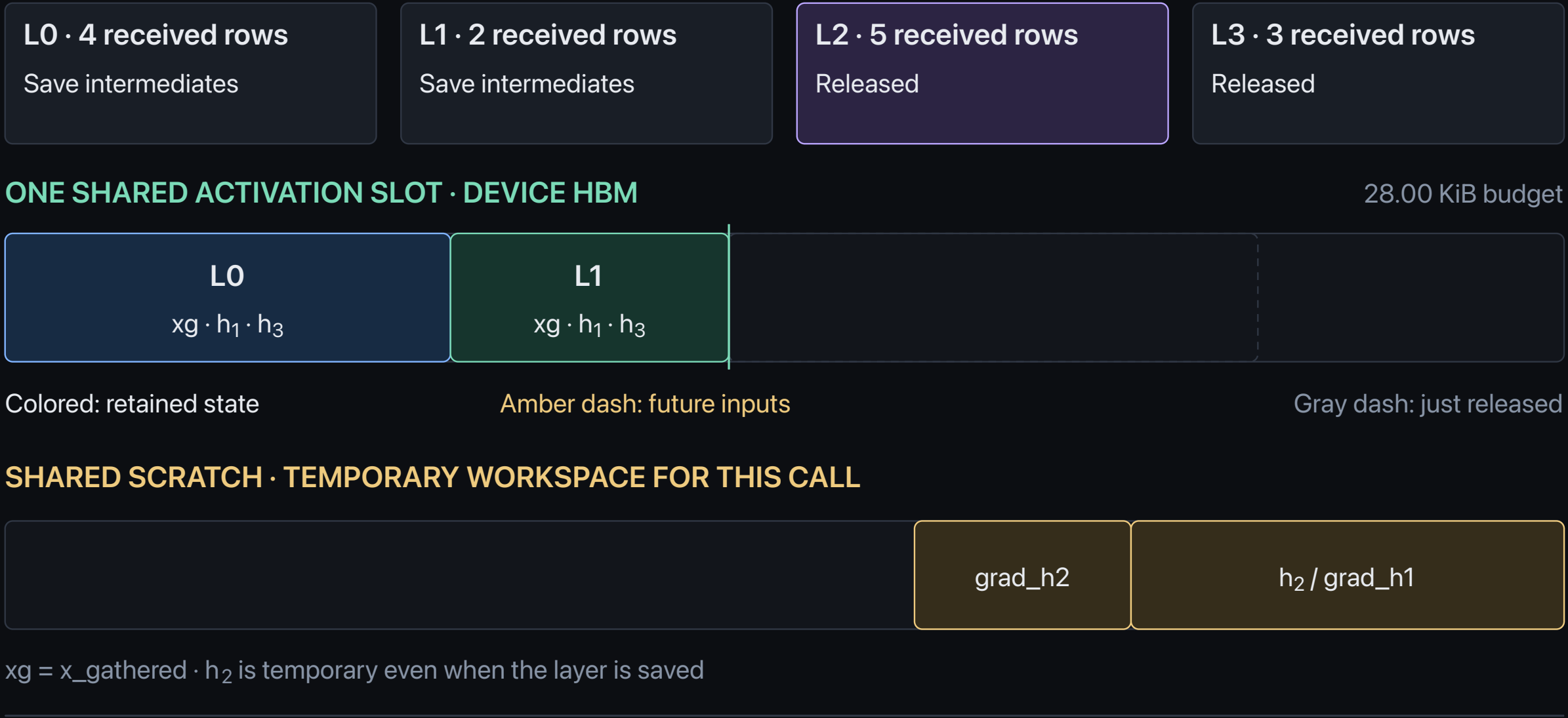
Activation Packing and Dynamic Recompute

Backward L3: replay, consume, release



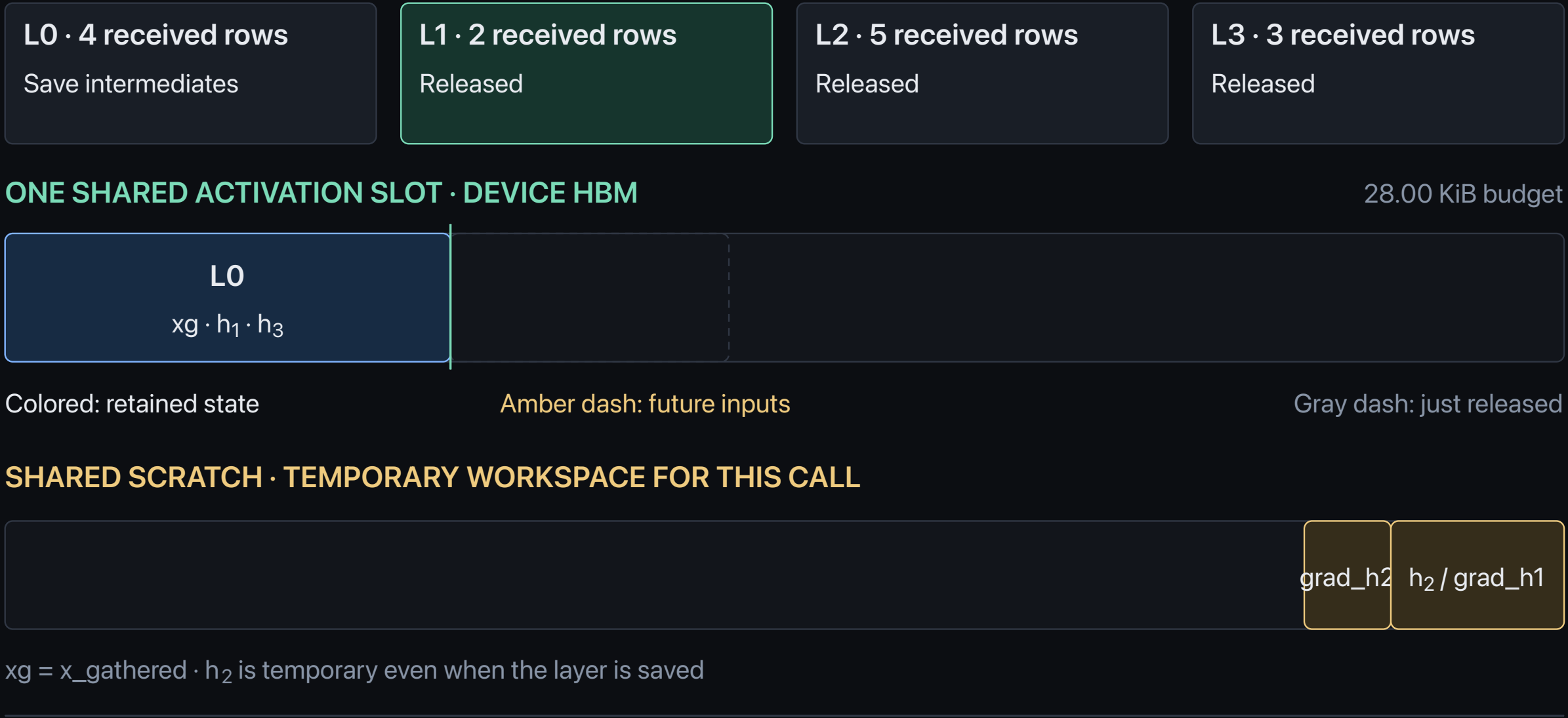
Activation Packing and Dynamic Recompute

Backward L2: consume and release



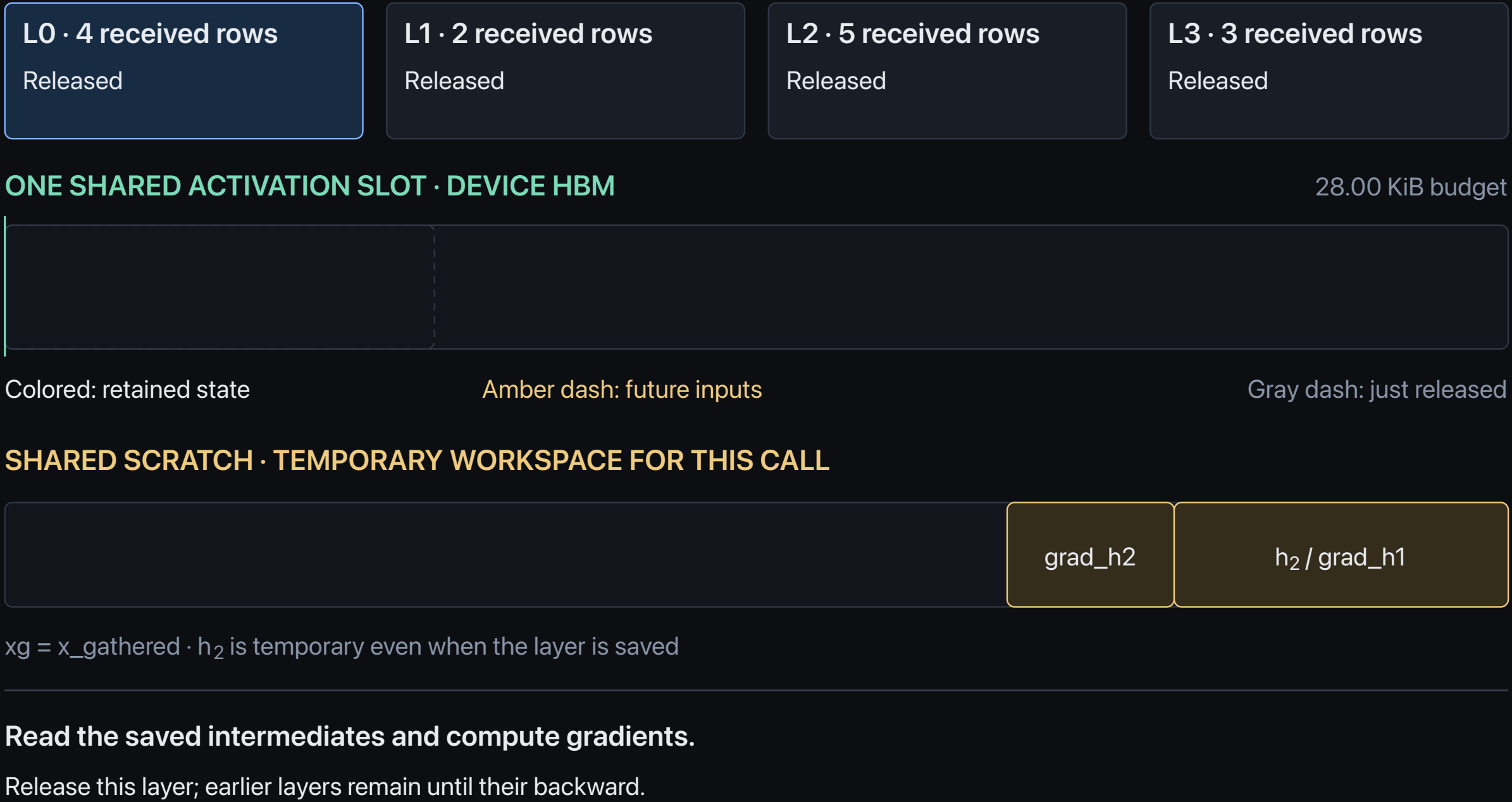
Activation Packing and Dynamic Recompute

Backward L1: consume and release



Activation Packing and Dynamic Recompute

Backward L0: consume and release



Backward L0 frees the slot for a later owner. Scratch has been reused throughout.

BF16 · one inspected GPU · both EP ranks agree on each save/replay decision · no host scratch in this example.

Performance Metrics and Theoretical Peaks

Useful GEMM work is counted · iteration time and NVLink are measured · HBM traffic is modeled

COUNTED

Useful GEMM Work

$$U_{\text{fwd}} = 6TKDF, \quad U_{\text{train}} = 18TKDF$$

Average per GPU, R = TK. Forward: gate/up/down GEMMs; backward: DGRAD + WGRAD. Multiply-add = 2 FLOPs.

Padding/replay FLOPs are not separately counted. Quantization, SwiGLU, and communication still contribute time.

MEASURED

Measured NVLink Bandwidth

NVML GPM samples TX/RX in a separate repeated-operation run of at least one second. Figures show rank mean and maximum.

$$\text{GB/s} = \text{MiB/s} \times 2^{20} / 10^9$$

Each direction's theoretical peak: 900 GB/s.

MEASURED

Routed-Layer Iteration Time

10 warmups → 5 windows × 30 iterations
CUDA events → rank maximum → window median

$$t = \text{median}_w(\max_r t_{r,w})$$

CUDA events time each 30-iteration CUDA-graph window. Divide by 30; take the maximum rank, then the median of five windows.

MODELED

Modeled HBM Byte Rate

$$\text{rate}_{\text{HBM,model}} = B_{\text{model}} / t$$

Logical inputs + activations + weights/scales + scores. Backward = 2× forward bytes; training = 3× as a coarse model.

No DRAM counters: cache effects and extra traffic are unmeasured. This is not HBM utilization.

Useful throughput divides counted GEMM work by the whole-layer iteration time. NVLink uses hardware counters; effective HBM uses a logical-byte model. Quantization, communication, padding, and replay remain in the timer.

Extra padding/replay FLOPs are not separately instrumented. All routed-operation costs remain in measured iteration time; effective HBM bandwidth is a model.

Standalone Inference Performance

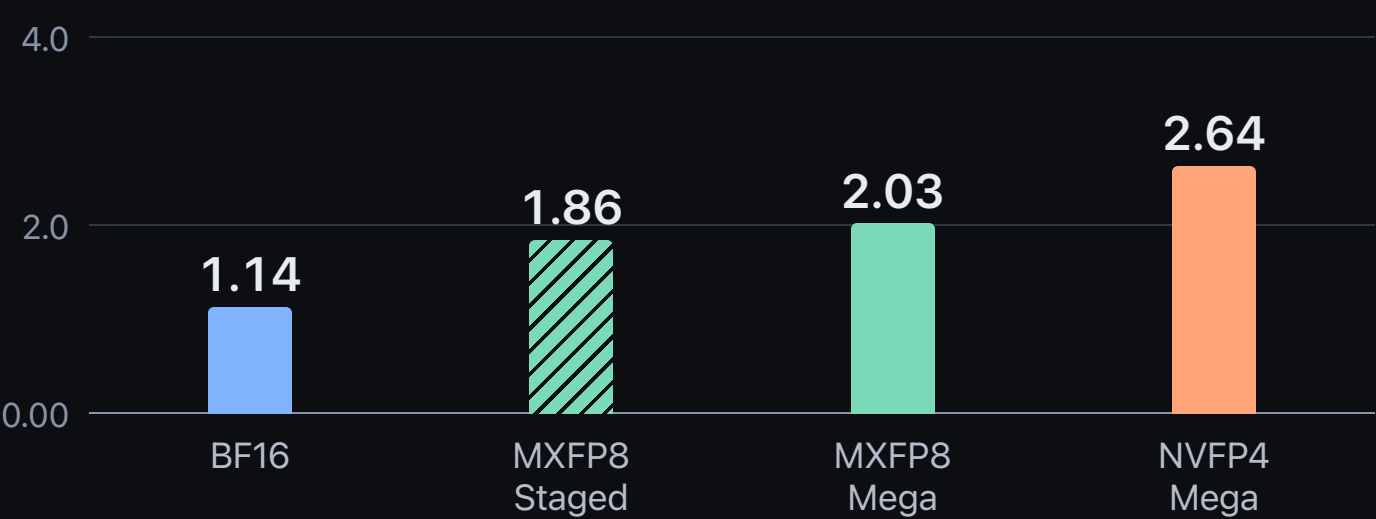
3.09 useful PFLOP/s per GPU · NVFP4 Mega · EP16

Inference · forward · GB300 · T = 2,048/GPU · D = 7,168 · E = 384 · top-6 · zero shared experts

EP8 · Inference · forward

Balanced routing · F = 3,072

Useful PFLOP/s / GPU ↑

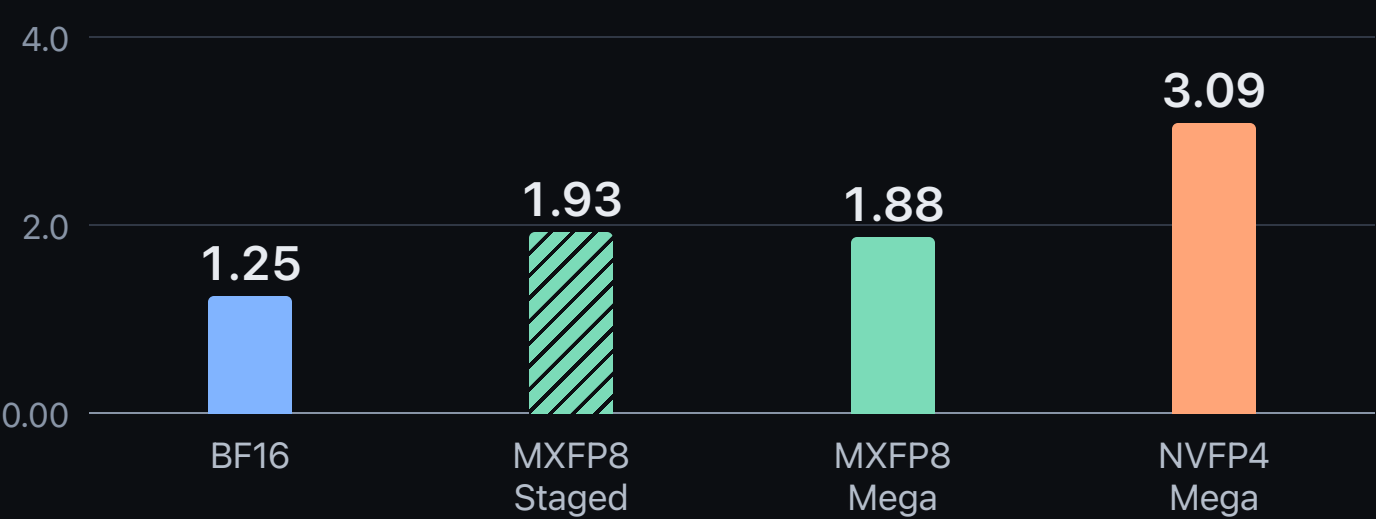


Precision / kernel	Useful PFLOP/s / GPU	Dense peak % of theory	Iteration ms	NVLink TX mean / max GB/s	HBM model TB/s
BF16	1.140	45.6	1.424	208 / 214	4.70
MXFP8 Staged	1.855	37.1	0.875	260 / 267	4.14
MXFP8 Mega	2.026	40.5	0.801	277 / 287	4.53
NVFP4 Mega	2.639	17.6	0.615	324 / 334	3.48

EP16 · Inference · forward

Balanced routing · F = 3,072

Useful PFLOP/s / GPU ↑



Precision / kernel	Useful PFLOP/s / GPU	Dense peak % of theory	Iteration ms	NVLink TX mean / max GB/s	HBM model TB/s
BF16	1.254	50.2	1.295	249 / 255	2.72
MXFP8 Staged	1.933	38.7	0.840	290 / 299	2.37
MXFP8 Mega	1.885	37.7	0.861	286 / 297	2.31
NVFP4 Mega	3.090	20.6	0.525	403 / 418	2.38

Useful rates include whole-layer time. Dense peaks are theoretical; NVLink TX is measured; HBM is modeled. TX theory: 900 GB/s per direction; HBM theory: 8 TB/s.

Standalone Training Performance

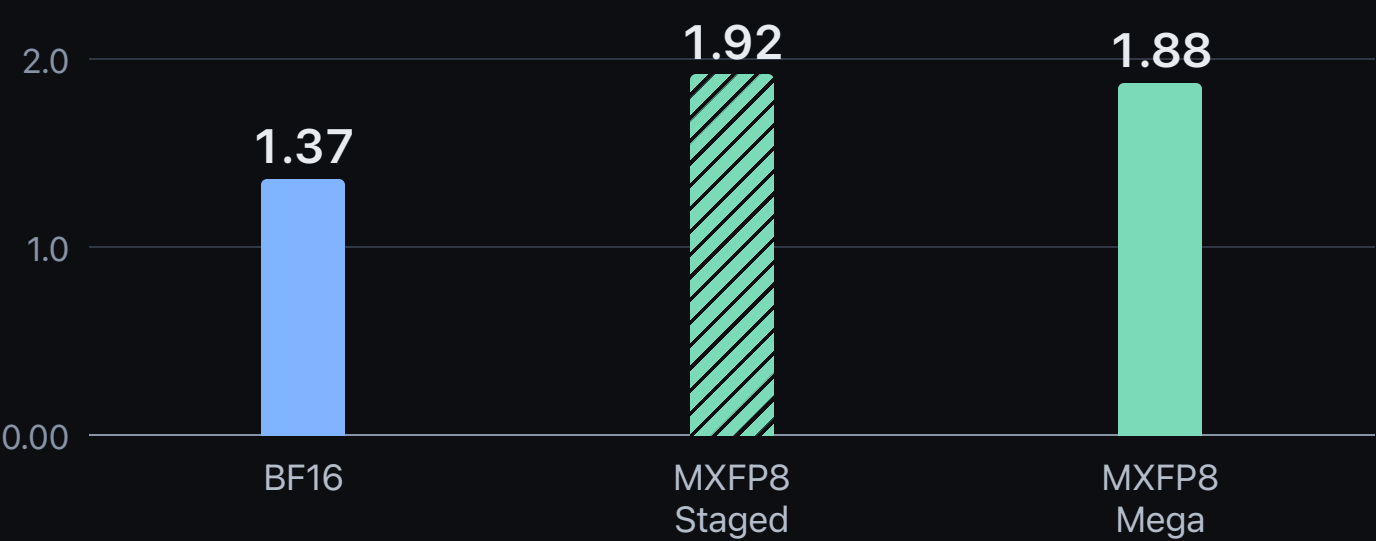
1.92 PFLOP/s at EP32 · 1.86 PFLOP/s at EP64 · MXFP8 Staged

Training · forward + backward · GB300 · T = 2,048/GPU · D = 7,168 · E = 384 · top-6 · zero shared experts

EP32 · Training · forward + backward

Balanced routing · F = 3,072

Useful PFLOP/s / GPU ↑

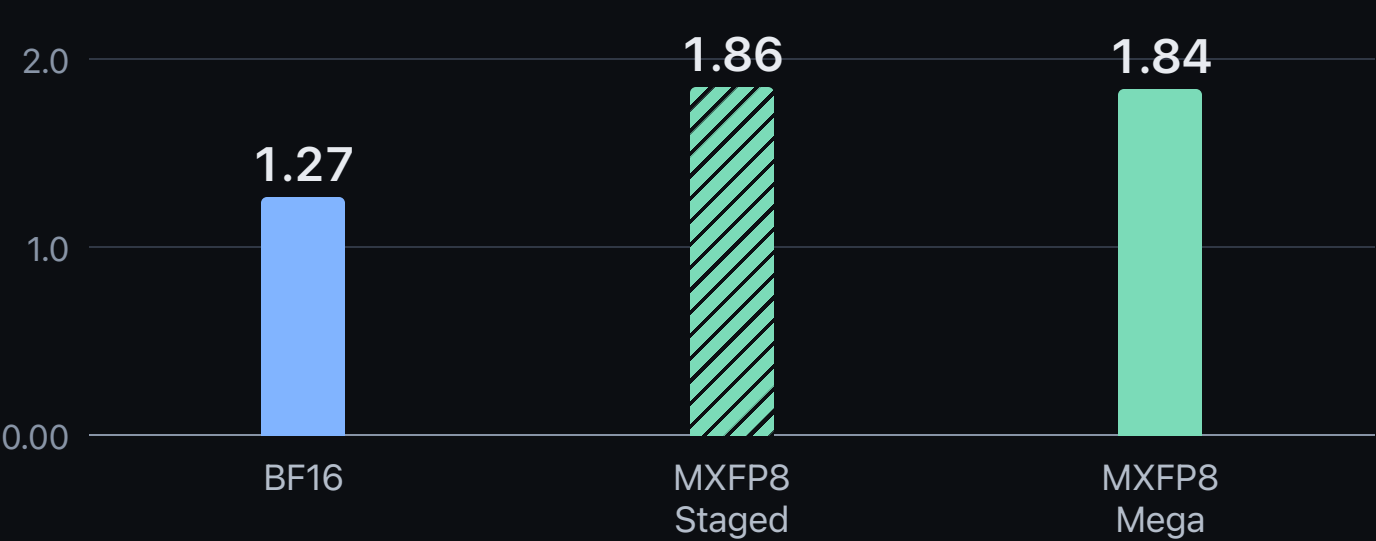


Precision / kernel	Useful PFLOP/s / GPU	Dense peak % of theory	Iteration ms	NVLink TX mean / max GB/s	HBM model TB/s
BF16	1.367	54.7	3.564	192 / 193	1.63
MXFP8 Staged	1.922	38.4	2.534	270 / 274	1.39
MXFP8 Mega	1.876	37.5	2.597	264 / 266	1.36

EP64 · Training · forward + backward

Balanced routing · F = 3,072

Useful PFLOP/s / GPU ↑



Precision / kernel	Useful PFLOP/s / GPU	Dense peak % of theory	Iteration ms	NVLink TX mean / max GB/s	HBM model TB/s
BF16	1.271	50.8	3.832	175 / 176	0.90
MXFP8 Staged	1.856	37.1	2.625	265 / 278	0.87
MXFP8 Mega	1.844	36.9	2.641	260 / 264	0.87

Useful rates include whole-layer time. Dense peaks are theoretical; NVLink TX is measured; HBM is modeled. EP32 and EP64 used different allocations.

Routing Imbalance: Work Growth and Execution Cost

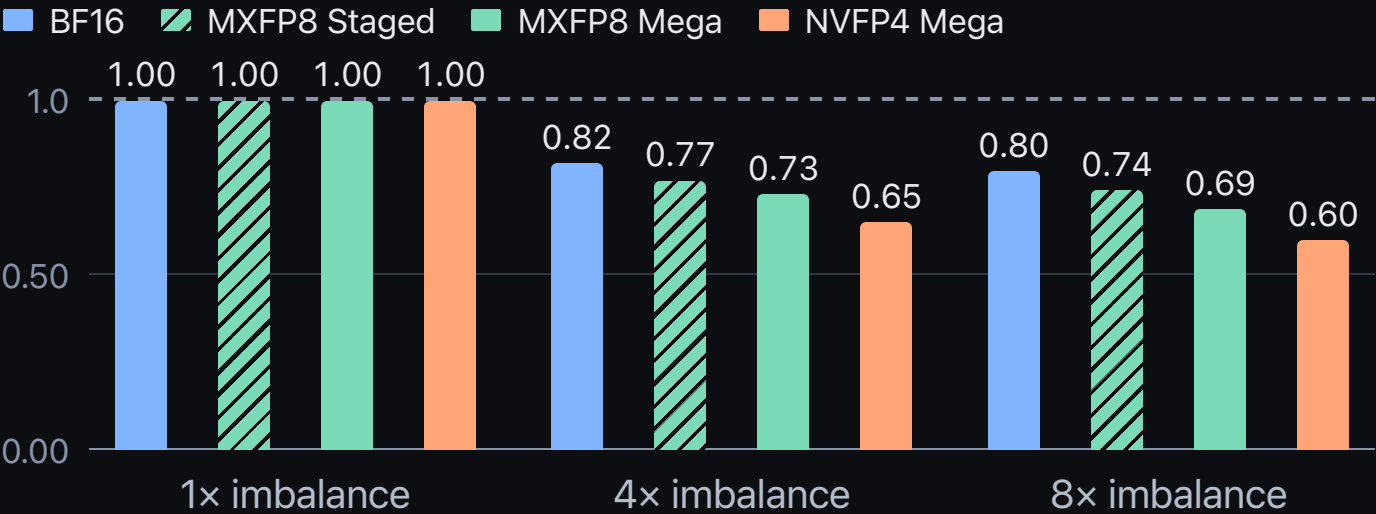
Sub-linear time scaling (EP16 NVFP4): 8× rows → 4.82× time; ≈40% lower cost/row

Inference EP16 · training EP64 · GB300 · T = 2,048/GPU · D = 7,168 · E = 384 · top-6 · zero shared experts

EP16 · Inference · forward

Hot GPU rows: 12,288 → 98,304 · same total routes

Time per hot-rank row / balanced ↓



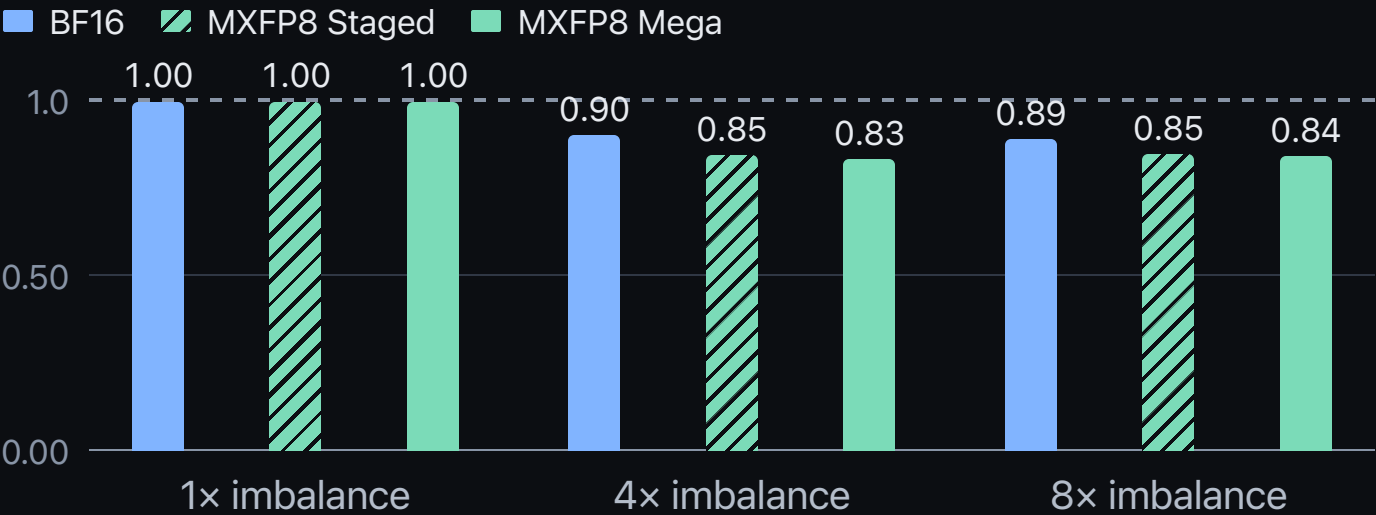
Reference: 8× received rows would take 8× time at unchanged cost per row.

Precision / kernel	Balanced ms	8× work ms	Time growth versus balanced	Cost / row versus balanced	8× busiest PFLOP/s, derived	8× average PFLOP/s / GPU
BF16	1.295	8.272	6.39×	0.80	1.57	0.20
MXFP8 Staged	0.840	4.989	5.94×	0.74	2.60	0.33
MXFP8 Mega	0.861	4.745	5.51×	0.69	2.74	0.34
NVFP4 Mega	0.525	2.530	4.82×	0.60	5.13	0.64

EP64 · Training · forward + backward

Hot GPU rows: 12,288 → 98,304 · same total routes

Time per hot-rank row / balanced ↓



Reference: 8× received rows would take 8× time at unchanged cost per row.

Precision / kernel	Balanced ms	8× work ms	Time growth versus balanced	Cost / row versus balanced	8× busiest PFLOP/s, derived	8× average PFLOP/s / GPU
BF16	3.832	27.383	7.15×	0.89	1.42	0.18
MXFP8 Staged	2.625	17.819	6.79×	0.85	2.19	0.27
MXFP8 Mega	2.641	17.838	6.75×	0.84	2.18	0.27

Whole-layer time divided by busiest-GPU received rows, normalized to balanced routing. Cluster-average throughput still falls under imbalance.

Expert Width: Useful Compute and Memory Traffic

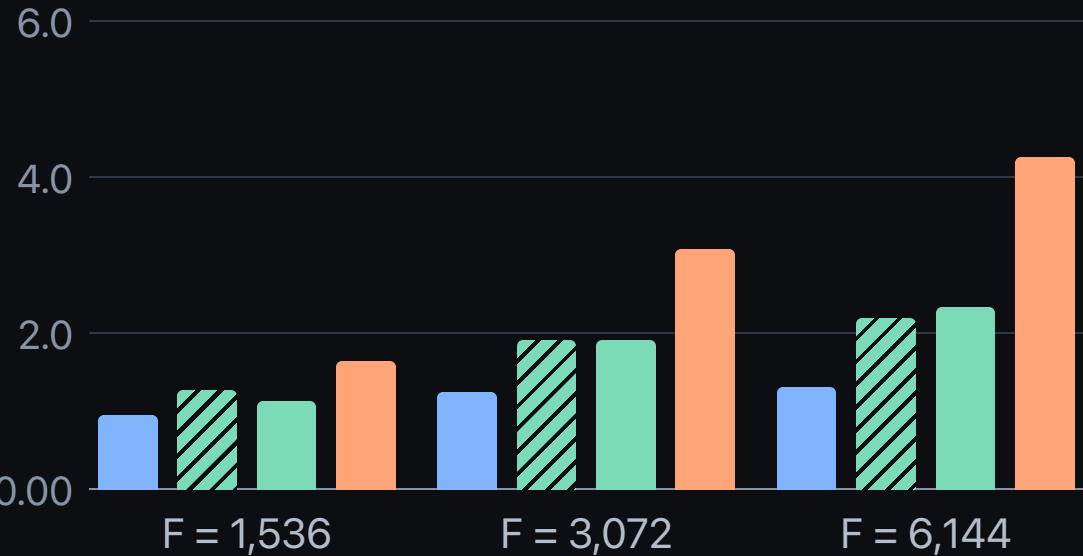
Sub-linear time scaling with F (EP16 NVFP4): 4× GEMM FLOPs → 1.54× time; 2.59× useful PFLOP/s

Inference EP16 · training EP64 · GB300 · T = 2,048/GPU · D = 7,168 · E = 384 · top-6 · zero shared experts

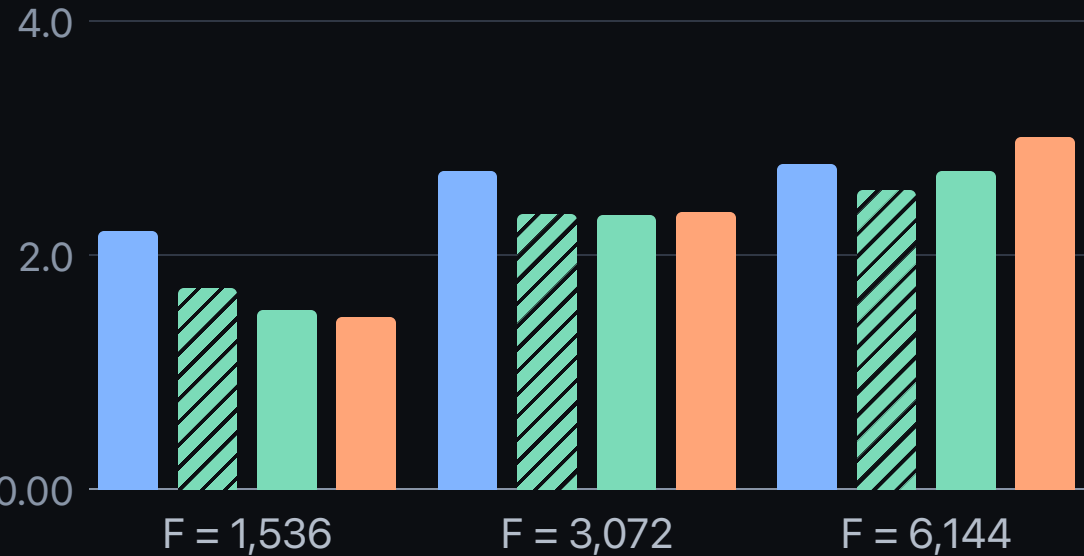
EP16 · Inference · forward

NVFP4 Mega: 4× work takes 1.54× time

Useful PFLOP/s / GPU ↑



Modeled HBM byte rate · TB/s



PFLOP/s per GPU / dense theoretical peak %

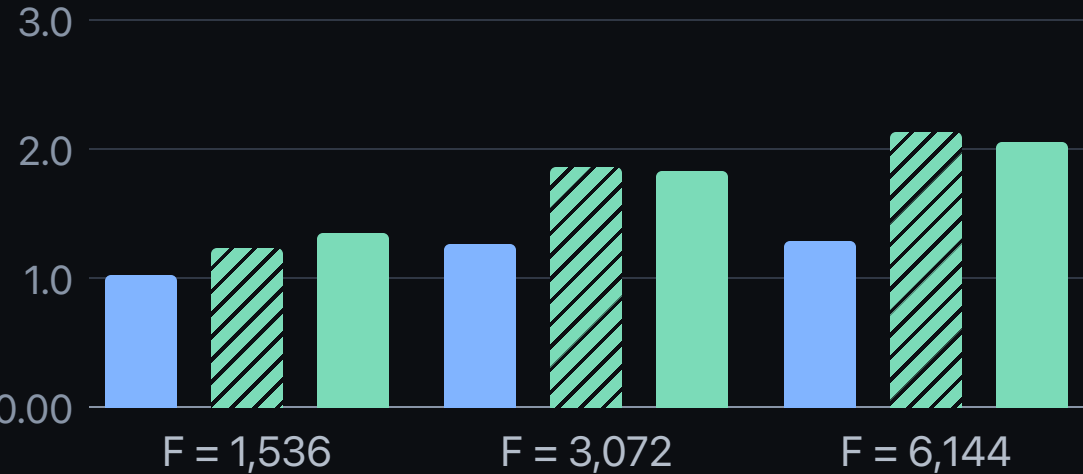
Second line: HBM model · TB/s

F →	1,536	3,072	6,144
BF16	0.96 / 38.5% 2.21 TB/s	1.25 / 50.1% 2.72 TB/s	1.32 / 52.8% 2.78 TB/s
MXFP8 Staged	1.28 / 25.5% 1.73 TB/s	1.92 / 38.4% 2.35 TB/s	2.20 / 44.0% 2.56 TB/s
MXFP8 Mega	1.13 / 22.7% 1.53 TB/s	1.91 / 38.3% 2.35 TB/s	2.34 / 46.8% 2.72 TB/s
NVFP4 Mega	1.65 / 11.0% 1.47 TB/s	3.08 / 20.6% 2.37 TB/s	4.27 / 28.4% 3.01 TB/s

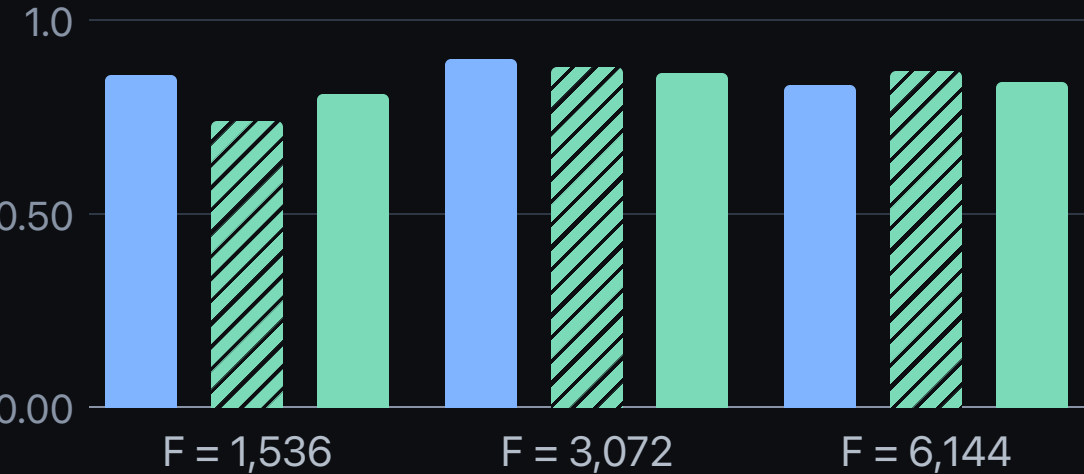
EP64 · Training · forward + backward

MXFP8 Staged: 4× work takes 2.32× time

Useful PFLOP/s / GPU ↑



Modeled HBM byte rate · TB/s



PFLOP/s per GPU / dense theoretical peak %

Second line: HBM model · TB/s

F →	1,536	3,072	6,144
BF16	1.03 / 41.2% 0.86 TB/s	1.27 / 51.0% 0.90 TB/s	1.30 / 51.8% 0.84 TB/s
MXFP8 Staged	1.24 / 24.8% 0.74 TB/s	1.87 / 37.4% 0.88 TB/s	2.14 / 42.7% 0.87 TB/s
MXFP8 Mega	1.36 / 27.1% 0.81 TB/s	1.84 / 36.7% 0.87 TB/s	2.06 / 41.2% 0.84 TB/s

Dense-peak % uses each format’s theoretical peak. HBM is a logical-byte model, not DRAM counters; EP64 training is roughly flat.

Resources

[Dist-MoE Repository ↗](#)

[BF16 Execution Guide ↗](#)

[Activation Memory Planner ↗](#)

[MXFP8 Kernel Design ↗](#)

[Pipeline Activation Slots ↗](#)