26. DeepEP

Chapter Abstract. An MoE router can choose the right experts and still leave the system with a harder problem. Each token row must move to its expert-owning rank. The result must then return to its original position. A four-rank example follows that round trip through dispatch, expert work, and combine. The mechanism map then explains how MoE-aware records and low-precision expert-major layouts connect to topology-specific GPU protocols. Analytical SM/QP selection, JIT specialization, event overlap, and valid layout reuse adapt that path. A separate generation boundary prevents the V1 zero-SM IBGDA hook from being mistaken for the V2 NCCL Gin path.

The ownership and completion vocabulary from Chapter 19 joins the expert-placement axes from Chapter 25. Together they distinguish a logical row, its current owner, and the completion that makes it safe to consume. The four-rank trace instantiates that framework. A DeepEP call selects both a communication protocol and a generated kernel artifact. Later sections show how transport topology and JIT identity determine how the row moves, which artifact runs, and when completion may be observed.

26.1. Why MoE Becomes Communication

Sparse routing selects experts. Expert parallelism turns those selections into destinations. DeepEP moves each token row to the rank that owns its selected expert. It then returns expert outputs to the original token order. The row's origin, route slot, expert id, and weight must survive both directions [19, 81].

The running example uses four ranks and eight routed experts:

$$\text{owner}(e) = \lfloor e/2 \rfloor .$$

Rank 0 therefore handles expert 1 locally. It nonetheless sends a route for expert 6 to rank 3. With maximum local token count $M_{\max} = 4$, token i on origin rank s has global origin id $g = sM_{\max} + i$.

Definition 26.1.1 (Expert-parallel domain) *An expert-parallel domain is an ordered rank group, an expert set, and an ownership map from experts to ranks. Dispatch and combine are defined relative to that versioned domain.*

Definition 26.1.1 fixes the versioned rank and expert ownership against which the local/remote and buffer contracts are read.

Definition 26.1.2 (Local and remote expert) *Expert e is local to rank r when* $\text{owner}(e) = r$*, and remote otherwise.*

Symbol	Meaning
$r, e, \text{owner}(e)$	Rank, logical expert, and current owner
$H, M_{\max}$	Hidden width and maximum local token count
$i, s, g = sM_{\max} + i$	Local index, origin rank, and global origin id
R_g	Expert/weight records for origin token g
k_{top}	Number of route slots
$y_{g,j}, z_g$	Route-slot output and restored combined output

Table 26.1.: Local notation for DeepEP dispatch and combine.

Definition 26.1.3 (DeepEP dispatch/combine buffer contract) *The DeepSeekMoE chapter in Volume I fixes the logical dispatch and combine maps. DeepEP realizes them across an expert-parallel domain: dispatch maps origin rows and route records $R_g = ((e_{g,j}, w_{g,j}))_j$ into destination buffers, and combine restores expert outputs to origin-token order and applies*

$$z_g = \sum_{j=0}^{k_{\text{top}}-1} w_{g,j} y_{g,j}.$$

Under Definition 26.1.3, token $g = 1$ is routed to experts 2 and 7. Dispatch therefore sends route slot 0 to rank 1 and slot 1 to rank 3. Combine must return both contributions to token 1 on the origin rank with their original weights. Swapped slots or a lost origin id can produce plausible values in the wrong row.

Table 26.1 fixes the symbols reused by the dispatch, handle, and combine traces.

The observed lane, `REQUEST-TRACE/OBSERVED`, records DeepEP normal mode on two H200 ranks. It retains 156 calls each to layout, dispatch, and combine, all ordered after the matching router record; every dispatch carries an FP8 payload with FP32 scales. Those counts establish one request-scoped handoff path, not physical traffic, bandwidth, retry behavior, or EPLB placement.

The scaled systems lane of the Volume II request trace, `REQUEST-TRACE/SCALED`, applies the same contract to row ρ_7: origin rank 0, route slot 0, logical expert 6, weight 0.65, chosen owner rank 2, and a fixed placement generation. That fixture does not model retry or supply a timing result. Instead, it makes the ownership and completion obligations testable.

With that round trip fixed, the implementation choices can be read as ways to preserve it while reducing movement and exposed waiting.

Table 26.2 gives the chapter's protocol map before those choices are developed separately. The rows are layers of one call, not competing names for the same mechanism: token semantics constrain the layout, the layout constrains movement, and completion constrains when any consumer may use the result.

Why DeepEP is fast

1. **Exploit MoE structure.** Dispatch sends one token record per distinct destination rank, with every valid local route slot attached, instead of blindly sending one activation copy per selected expert.
2. **Keep an expert-parallel protocol on the GPU.** Generated prologue, movement, and epilogue kernels carry counts, payload, origin identity, route

Table 26.2.: DeepEP protocol and lifecycle stack. The semantic, layout, transport, completion, artifact, generation, and recovery layers coexist in one expert round trip but carry different state and failure obligations.

Layer	Contract or state carried	Where developed
Semantic round trip	Dispatch preserves origin id, destination owner, expert, route slot, weight, and mask; combine restores weighted contributions to the origin row.	Sections 26.2 and 26.4
Layout and handle	Variable split counts, expert-major segments, semantic row counts, aligned offsets, scales, and the inverse slot map remain attached to a compatible `EPHandle`.	Section 26.3
Generated execution	Notify-and-move dispatch, its receive-layout copy epilogue, return movement, the final reduction epilogue, and the explicit buffer fence form a generated multi-launch chain.	Section 26.5
Topology transport	Direct mode targets the final owner; hybrid mode crosses a scale-out rail and then forwards within the destination scale-up domain.	Sections 26.8 and 26.9
Publication and completion	Entry barriers protect reusable slots; TMA completion, QP flushes, release–acquire publication, final barriers, and `EventOverlap` establish consumer readiness.	Sections 26.9 and 26.7
Reuse and lifetime	Cached dispatch reuses accepted layout only under a compatible handle; every consuming stream installs its event dependency before the event and tensors are released.	Section 26.7
JIT artifact lifecycle	Generated source, transitive headers, compiler state, flags, and target architecture identify a cached CUBIN that is compiled, atomically published, loaded, and launched.	Section 26.5
Generation and recovery boundary	V2 NCCL Gin state remains separate from legacy V1 NVSHMEM/IBGDA state; a failed channel epoch requires caller-side buffer or communicator recovery and fresh handles before replay.	Sections 26.12 and 26.15

metadata, and completion state through GPU workspaces. This is protocol specialization; a call can still comprise several kernel launches.

3. **Match the physical topology.** Direct mode targets final ranks; hybrid mode crosses a scale-out rail and then forwards inside the destination scale-up domain. NVLink-like and RDMA paths retain different completion state.
4. **Move a useful representation.** FP8 dispatch reduces activation bytes only when its scale pack travels with the row. The destination places valid rows in aligned expert-major segments ready for grouped expert compute.
5. **Budget communication resources.** V2 models traffic and fabric bandwidth to choose an even SM count, then derives direct or hybrid QP counts. The model has declared routing and mode limits; manual overrides remain part of the run record.

6. **Specialize at runtime.** Topology, hidden width, top-k, expert count, payload layout, and SM/QP choices select generated CUDA and a cache identity rather than one generic precompiled communication kernel.
7. **Expose dependencies instead of serializing streams.** `EventOverlap` lets independent compute proceed until received expert rows are actually needed.
8. **Reuse layout, not routing wishes.** A cached `EPHandle` can skip layout reconstruction and CPU synchronization only while destination ownership and every layout-relevant assumption remain compatible.

Generation boundary. The archived V1 low-latency path uses NVSHMEM/IBGDA and a receive hook that can progress RDMA without communication SMs. V2 uses NCCL Gin and does not preserve that zero-SM low-latency EP surface [19].

26.2. Dispatch Semantics and Invariants

Dispatch begins with a routing table and ends with receive buffers on the expert-owner ranks. Calling this merely "an all-to-all" hides the important part: each transmitted row carries an origin token identity, selected expert slots, route weights, and enough layout metadata for combine to reverse the movement. DeepEP's reference dispatch implementation makes this explicit. It builds per-destination send buffers, exchanges variable counts, masks top-k entries owned by other ranks, and returns origin token indices with the received rows [19].

Definition 26.2.1 (Origin token global index) *The origin token global index is*

$$g = sM_{\max} + i,$$

where s is the origin rank, $M_{\max}$ is the maximum token count per rank used by this origin-index encoding, and i is the origin-local token index. Equivalent metadata may be used, but combine needs enough information to recover the origin rank and origin-local output row.

The numerical example is small. With $M_{\max} = 4$, rank r has global ids $4r, 4r + 1, 4r + 2, 4r + 3$. If rank 3 later receives a row tagged with $g = 1$, it knows the row originated as local token 1 on rank 0. If it receives $g = 6$, it knows the row originated as local token 2 on rank 1. This identity indexes the row that combine must restore.

For rank 0, use the following top-2 routes:

g	$(e_{g,0}, e_{g,1})$	$(\text{owner}(e_{g,0}), \text{owner}(e_{g,1}))$
0	$(1, 6)$	$(0, 3)$
1	$(2, 7)$	$(1, 3)$
2	$(4, 5)$	$(2, 2)$
3	$(0, 3)$	$(0, 1)$

If dispatch sends one token row per destination rank, deduplicating the case where two route slots land on the same destination, then rank 0's per-destination

send counts are

$$s^{(0)} = (2, 2, 1, 2).$$

The two rows to destination rank 0 are tokens 0 and 3. The two rows to rank 1 are tokens 1 and 3. The single row to rank 2 is token 2, even though token 2 has two selected experts on that rank. The two rows to rank 3 are tokens 0 and 1. Deduplication produces seven token rows with route-slot masks instead of $4 \cdot 2 = 8$ route-slot copies.

Example 26.2.1 (Rank-0 dispatch counts) For token 0, the two selected experts are 1 and 6, so the destination ranks are 0 and 3. For token 1, experts 2 and 7 give destination ranks 1 and 3. For token 2, both selected experts 4 and 5 live on rank 2, so the deduplicated dispatch emits one token row to rank 2 with two valid route slots. For token 3, experts 0 and 3 give destination ranks 0 and 1. Adding one row for each unique destination per token gives

$$s^{(0)} = (2, 2, 1, 2),$$

where the entries count rows sent from rank 0 to ranks 0, 1, 2, and 3. Token 2 uses one copy because both of its experts share an owner rank.

How one warp realizes destination grouping

For one token, lane $j < k_{\text{top}}$ holds route slot j's destination rank, while the remaining lanes hold an invalid marker. The pinned V2 helper `match` returns the warp mask of lanes with the same rank. `get_master_lane_idx` applies `bfind` to elect the highest participating lane, so `deduplicate` is true for exactly one lane per unique valid destination value:

```
__device__ __forceinline__ unsigned match(const int& value) {
    return __match_any_sync(0xffffffff, value);
}

// ... book elision: unrelated fns and ffs helpers ...
__device__ __forceinline__ int get_master_lane_idx(
    const unsigned& mask) {
    // Equivalent to '31 - __clz(mask)'
    int highest_idx;
    asm volatile("bfind.u32 %0, %1;"
                 : "=r"(
                     highest_idx) : "r"(mask));
    return highest_idx;
}

__device__ __forceinline__ bool deduplicate(
    const int& value, const int& lane_idx) {
    return get_master_lane_idx(match(value)) == lane_idx;
}
```

Only that representative increments the destination's atomic sender counter and receives a physical slot. The payload row still carries every top-k expert id, weight, and validity field; lane election removes duplicate payload copies while preserving route semantics.

For token $g = 2$, the lanes holding experts 4 and 5 both compute destination rank 2. Their match masks are equal, lane 1 is the helper-selected representative, and the rank-2 sender counter advances once. The representative choice is an implementation detail: combine relies on the assigned slot and route fields, while lane 1 is semantically arbitrary. Notify warps apply the same per-rank rule when forming split counts, then use warp shuffles and inclusive scans to produce the count prefixes consumed by the later receive-layout epilogue [19, 66].
The excerpt is verbatim apart from line wrapping and the declared elision, from DeepEP revision `60d44037a702`, under the repository's MIT License. The public companion lists the exact upstream source coordinates in `chapter_deepep/README.md`. Its dispatch call site computes the predicate for every lane value, then admits the elected lane only when the destination rank is nonnegative. The call site, rather than the helper alone, allocates slots and publishes payload bytes; end-to-end checks establish that combine received every route.

The executable checkpoint applies the same semantic rule to the four-token fixture without importing the larger companion implementation. It groups route slots by token and destination rank, emits one payload row per group, and keeps both expert routes inside token 2's rank-2 row.

Executable checkpoint: One payload copy, two retained route slots

```python
# token_id, route_slot, expert_id, destination_rank
routes = (
    (0, 0, 1, 0),
    (0, 1, 6, 3),
    (1, 0, 2, 1),
    (1, 1, 7, 3),
    (2, 0, 4, 2),
    (2, 1, 5, 2),
    (3, 0, 0, 0),
    (3, 1, 3, 1),
)

grouped: dict[tuple[int, int], list[tuple[int, int]]] = {}
for token_id, route_slot, expert_id, destination_rank in routes:
    key = (token_id, destination_rank)
    grouped.setdefault(key, []).append((route_slot, expert_id))

records = tuple(
    (token_id, rank, f"x{token_id}", tuple(route_slots))
    for (token_id, rank), route_slots in sorted(grouped.items())
)
token_2_rank_2 = next(record for record in records if record[:2] ==
    (2, 2))

assert len(records) == 7
assert token_2_rank_2 == (2, 2, "x2", ((0, 4), (1, 5)))

if __name__ == "__main__":
    print(f"token 2 -> rank 2: payload={token_2_rank_2[2]},
    routes={token_2_rank_2[3]}")
```

Run: `python3 chapter_deepep/dispatch_combine_checkpoint.py`

The printed assertions expose seven payload rows for eight route slots and show token 2's single rank-2 payload with route-slot/expert pairs $(0, 4)$ and $(1, 5)$. The full CPU-only reference at `chapter_deepep/` also applies route weights during combine and rejects missing, duplicated, or unexpected route contributions. Its scope is dispatch and combine identity; CUDA transport, workspace layout, synchronization cost, and throughput require their own checks.

The scan is hierarchical because a shared-memory count belongs to one SM, not to the whole dispatch grid. Let S be the number of participating SMs and let $c_i^{(s)}$ be SM s's count for rank or expert entry i. In the pinned implementation, every SM atomically contributes the packed word

$$2^{32} + c_i^{(s)}$$

to one 64-bit workspace cell. The high half therefore counts arrivals, while the low half accumulates $\sum_s c_i^{(s)}$. SM 0 may consume the low half only after the high half equals S. This fuses a grid rendezvous and a sum without pretending that a partially accumulated count is ready [19].

Notify phase	Owned state	Readiness gate	State made safe
Count locally	Per-SM shared rank and expert counters	Notify-thread named barrier	One complete local contribution $c_i^{(s)}$ per entry.
Rendezvous and sum	64-bit workspace word: high half arrivals, low half count sum	High half equals S	Complete grid-wide rank and expert counts; SM 0 then clears the workspace cell for reuse.
Exchange counts	Positive-ready encoded rank and expert mailboxes	Receiver decodes a ready value from every required origin	Counts from all relevant ranks, rather than only the local GPU's contribution.
Publish layout	Unaligned semantic expert counts, aligned expert counts, and rank/expert prefix arrays	Peer counts have arrived and notify warps have rejoined	Inclusive rank boundaries and exclusive aligned expert offsets for the receive-layout epilogue.

Table 26.3.: The pinned V2 notify path is a count–rendezvous–exchange–scan pipeline. Each gate prevents a downstream phase from treating partial counts as a complete receive layout.

Local completion, grid completion, peer arrival, and layout publication are successive gates; each authorizes the next phase.

For a small instance with $S = 2$, suppose rank 0's two SMs contribute 1 and 0 to its destination-rank-2 count. After both packed additions, the workspace word represents arrival count 2 and token-row count 1. A low-half value of 1 after the first arrival remains provisional because the second SM can change it. Once the arrival half reaches 2, SM 0 can publish count 1 to rank 2. There it occupies sender rank 0's entry, and rank 2's inclusive sender-rank scan places the endpoint of rank 0's receive interval. The scheme assumes the configured token bounds keep the accumulated count within the low 32-bit field.

Definition 26.2.2 (All-to-all-v token exchange) *An all-to-all-v token exchange is a variable-size exchange in which each rank sends a possibly different number of token rows to each destination rank and receives a possibly different number from each origin rank. The split vectors are part of the exchange layout because they determine receive buffer boundaries.*

The route-slot mask explains why a receive row can carry fewer active slots than the original top-k row. When rank 3 receives token 0 from rank 0, the original selected experts are $(1, 6)$. Only expert 6 is local to rank 3, so the local view masks the expert-1 slot. When rank 2 receives token 2, both selected experts $(4, 5)$ are local, so both route slots remain valid. This matches the reference implementation's habit of replacing top-k entries owned by other ranks with an invalid marker before local expert work [19].

Table 26.4 is also a correctness test in prose. If an implementation sends token 2 twice to rank 2 and the combine path expects deduplicated token rows, buffer

Table 26.4.: Rank 0 dispatch rows in the running example. A token may carry one or two valid route slots after masking at the destination rank.

Dest.	Rows from rank 0	Valid local slots	Metadata
0	tokens 0 and 3	expert 1 for token 0; expert 0 for token 3	$g = 0, g = 3$
1	tokens 1 and 3	expert 2 for token 1; expert 3 for token 3	$g = 1, g = 3$
2	token 2	experts 4 and 5 for the same token	$g = 2$
3	tokens 0 and 1	expert 6 for token 0; expert 7 for token 1	$g = 0, g = 1$

Algorithm 13: DeepEP-style dispatch fanout procedure. The routine states the metadata-preservation rule before expert compute and combine are discussed.

Input: Rank-local token rows; top-k expert ids; expert-owner map
Output: Receive-side token rows with origin metadata and local-slot masks

```
foreach origin token row do
    compute the global origin token id and the destination owner for each selected expert;
    group route slots by destination rank for that token;
    append one token row per destination with all route slots and weights attached;
exchange per-destination split counts and payload buffers across ranks;
foreach received token row do
    mask route slots whose expert owner differs from the receiving rank;
    retain origin id, route weight, and local expert ids for combine;
return receive buffers whose metadata can reconstruct original token order;
```

counts and route-slot positions will disagree. If an implementation sends token 2 once but forgets that two local experts must run, one expert output will be missing. If an implementation preserves both expert outputs but loses $g = 2$, combine restores the row to the wrong origin. Dispatch is correct when the payload and metadata agree.

Dispatch Identity Checkpoint Follow one token through three names. The origin id g names where the token came from. The destination rank names where its selected expert work must run. The local route-slot mask names which of the token's selected experts are valid on that destination. Combine needs all three. A correct byte transfer with the wrong origin id or route-slot mask is still a wrong MoE layer.

Figure 26.1 shows the same split vector as a fanout diagram. The figure shows the destination count alongside the route-slot mask: rank 2 receives one token row carrying two valid local slots.

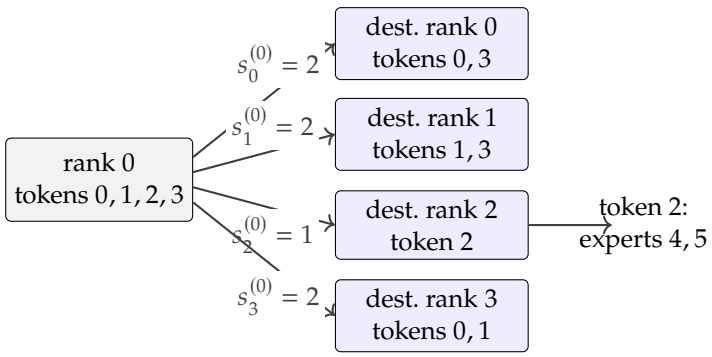

Figure 26.1.: Rank 0's deduplicated dispatch fanout has split vector $s^{(0)} = (2, 2, 1, 2)$. A destination count measures token rows, while route metadata identifies the local expert slots that remain valid.

Algorithm 13 is the procedural version of the rank-0 example. First, it tags token rows with origin identity and destination ranks. Then it exchanges variable split buffers. Finally, it masks remote slots while preserving the metadata combine needs to restore token order.

26.3. The Dispatch Handle and Buffer Layout

Dispatch produces tensors, but it also produces a state object. In DeepEP V2, that state object is an `EPHandle`. The handle is the bridge between dispatch and combine. It records the top-k routing view, received-token prefix sums, per-expert counts, origin metadata, destination buffer slots, and optional hybrid-mode metadata used by later calls [19]. Without such a handle, combine would have to rediscover the layout that dispatch already computed.

Definition 26.3.1 (Dispatch handle) *A dispatch handle is the metadata returned by dispatch and consumed by combine to interpret receive buffers. At minimum, it must identify the origin-token rows, valid route slots, destination buffer locations, and per-expert receive layout needed to run local experts and restore outputs.*

In the chapter-local four-rank buffer-layout fixture, rank 2 receives one row from rank 0 whose local-fixture global origin id is $g = 2$. That record has two valid local expert slots, experts 4 and 5. If rank 2 also receives a row from rank 1 with local-fixture origin id $g = 7$ for expert 5, then its local expert row counts are $M_4 = 1$ and $M_5 = 2$. A simple unaligned expert layout could store expert 4's row first and expert 5's two rows next, with prefix sums $(1, 3)$. The prefix sum counts rows and defines the segment boundary that the grouped GEMM or expert kernel will read.

Definition 26.3.2 (Low-precision dispatch-to-GEMM contract) *An FP8 dispatch record consists of the quantized hidden row, the scale pack needed to reconstruct that row, and the origin and route metadata needed to interpret it. The destination places valid records in expert-major segments whose semantic row counts may differ from their physically aligned lengths. Expert compute consumes the valid rows and their scales; padding is storage, not an additional token.*

The contract explains why "FP8 dispatch" is not shorthand for sending only a one-byte array. For $H = 256$, a BF16 activation row occupies 512 data bytes. An E4M3 FP8 row occupies 256 data bytes. With one four-byte scale per 128 coordinates, its two scales raise the low-precision representation to 264 bytes before route ids, weights, origin metadata, or alignment. That is 51.6% of the BF16 activation bytes, a 48.4% reduction. Even so, only the data-plus-scales representation is reconstructible. The byte saving and the metadata obligation are therefore one contract rather than competing descriptions [19].

Table 26.5 follows the same record across the network and into expert compute. The physical alignment rule now specifies how the middle stage reserves its expert-major segments.

Table 26.5.: The low-precision dispatch-to-GEMM handoff. FP8 saves activation bytes, while scale and layout metadata keep the received representation usable.

Stage	Representation	Consumer obligation
Wire record	FP8 hidden bytes, scale pack, origin id, route slots, and weights	Reconstruct the same hidden row and routing identity selected at the origin.
Expert-major placement	Semantic count M_e, aligned physical segment, and per-record scale location	Keep valid rows separate from padding and expose the segment boundary to expert compute.
Grouped expert compute	Valid expert rows and the matching weight/scaling interpretation	Multiply only semantic rows; preserve route identity for combine.

Definition 26.3.3 (Expert alignment) *Expert alignment rounds each per-expert receive segment up to an implementation-required multiple. If an expert has M_e semantic rows and alignment A, the physical segment may reserve*

$$\operatorname{align}(M_e, A) = A\lceil M_e/A \rceil$$

slots. Padding slots are layout, not model tokens.

In this chapter, the expert prefix sum is an offset into the receive-side expert buffer. A semantic count says how many real token rows an expert owns. An aligned count says how much physical buffer space is reserved for that expert. The prefix sum marks where the next expert segment begins.

With $M_4 = 1$, $M_5 = 2$, and alignment $A = 4$, rank 2 reserves four slots for expert 4 and four slots for expert 5. The physical prefix sums are $(4, 8)$. The semantic row count is nonetheless 3. This distinction is small in the toy example and large in a production batch. If padding is mistaken for valid rows, the expert computation can create extra outputs. If valid rows are packed without respecting the expected alignment, a kernel may read the wrong segment or reject the layout.

Table 26.6.: Aligned expert-buffer layout for rank 2 with alignment $A = 4$. The handle must preserve both semantic counts and physical offsets.

Local expert	Semantic rows	Start offset	Physical end offset	Padding slots
$e = 4$	$M_4 = 1$	0	4	slots 1–3
$e = 5$	$M_5 = 2$	4	8	slots 6–7

Table 26.6 makes the padding rule concrete. Expert 5 begins at physical offset 4 even though expert 4 has only one semantic row. Combine and grouped expert compute therefore need the handle's layout metadata in addition to the dense token payload. Figure 26.2 turns the same prefix-sum rule into a buffer picture.

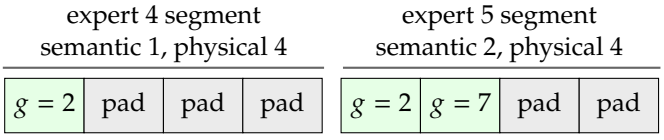

Figure 26.2.: The handle has to distinguish semantic expert rows from physical aligned slots. Padding occupies buffer space, but it is not model data and must not become a combine contribution.

handle records physical prefix sums $(4, 8)$ and semantic row counts

A dispatch handle's conceptual roles may span several repository-level fields. The role-to-field map expands the dispatch handle in Definition 26.3.1, and its

Handle role	Running value	Mathematical role	Failure if wrong
Top-k view	route slots for $g = 2, 7$	local expert selection	missing or extra expert row
Origin metadata	$g = 2$, $g = 7$	restore origin	output written to wrong token
Per-expert count	$M_4 = 1, M_5 = 2$	expert segment lengths	bad GEMM row count
Aligned prefix sum	$(4, 8)$	physical layout	padding treated as semantic data
Buffer slot index	receive row locations	tensor addressing	stale or crossed rows

Table 26.7.: Conceptual roles of dispatch-handle fields. The concrete DeepEP names are repository addresses for these roles, not the roles themselves.

aligned prefix-sum row is the physical layout consequence of Definition 26.3.3. Exact names identify DeepEP's repository fields. The mathematical roles instead explain why combine can trust the received layout. Chapter-local worked trace: dispatch receives rows $g = 2$ and $g = 7$, writes their local expert rows into the aligned segments in Table 26.6, returns the roles summarized in Table 26.7, and combine uses the origin metadata to put the expert outputs back on $g = 2$ and $g = 7$. This makes the first table the layout instance and the second table the handle-field map that explains it.

Figure 26.3 expands that trace into the ordering problem that the handle solves. Origin-token order, dispatch-record order, expert-major buffer order, and final output order are four different views of the same rows. Correctness comes from carrying the route weights, expert-owner rank, buffer slot, and inverse map together.

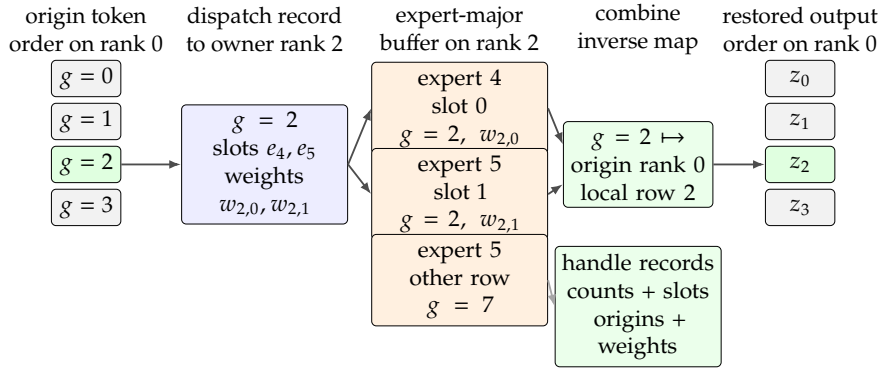

Figure 26.3.: The handle ties dispatch and combine across changing buffer orders. Token $g = 2$ starts in origin order, travels once to rank 2, becomes two expert-major rows with route weights, and returns through the inverse map to origin row 2.

Concept-to-code map

Reusable state lives in `https://github.com/deepseek-ai/DeepEP/blob/60d44037a702f651a6e18bd4aea65ed8409051c2/deep_ep/buffers/elastic.py`; reference semantics live in `https://github.com/deepseek-ai/DeepEP/blob/60d44037a702f651a6e18bd4aea65ed8409051c2/deep_ep/utils/refs.py`, with executable checks in `https://github.com/deepseek-ai/DeepEP/blob/60d44037a702f651a6e18bd4aea65ed8409051c2/tests/elastic/test_ep.py` [19].

- **Handle and buffer layout.** The handle roles map to `EPHandle`, `ElasticBuffer.dispatch`, and `ElasticBuffer.combine` in the recorded DeepEP source.
- **Reference semantics.** Textbook dispatch and combine behavior maps to `dispatch`, `generate_pre_combine_data`, and `combine` in the recorded DeepEP source.
- **Executable assertions.** Cached handles, expanded dispatch, expert alignment, and combine checks map to the DeepEP test surface.

DeepEP also exposes an expanded dispatch variant and a cached-handle pattern [19]. Expanded dispatch represents route slots more directly. That form can be useful when the next compute surface wants one row per valid slot. A cached handle reuses layout metadata in a later dispatch call. The cached case is attractive for decode because the batch may repeatedly use the same routing layout. Even so, it has a strict validity condition: the layout-relevant top-k indices, maximum token count, expert ownership, rank group, and mode assumptions must still match. If token 2 changes from experts $(4, 5)$ to $(6, 7)$, the old handle describes the wrong destinations.

A cached handle is a layout promise; reuse is safe only while routing layout and ownership assumptions stay compatible.

Definition 26.3.4 (Cached handle validity) *A cached dispatch handle is valid for a later dispatch only when the later call has the same routing-layout identity as the call that produced the handle. Payload values may change, but the rank group, owner map, maximum token convention, selected expert layout, and mode assumptions must remain compatible.*

The handle is therefore a compact expression of the main invariant: communication layout is state. The row tensor by itself is insufficient. A correct expert-parallel path must carry the state that says where each row came from and which expert slots are valid locally. That state also says where each expert segment starts and how combine should reverse the movement.

26.4. Combine, Reductions, and Original Token Order

Combine correctness depends on origin-token order after weighted expert fragments return.

Combine does more than play dispatch backward. Dispatch decides which ranks need each origin token row. Local expert computation then changes the payload: the row that returns is an expert output, not the original activation row. The combine path must gather those expert outputs, apply the route-slot weights, and write one final row for each original token. DeepEP's reference utilities model this by generating per-route outputs from origin token ids and then accumulating them in route-slot order [19].

Definition 26.4.1 (Combine restoration invariant) *For every origin token g, combine must write the final combined vector z_g into the output row corresponding to g's origin rank and local token position. Receive-buffer order, expert order, and network arrival order must not change the final token order.*

Algorithm 14: DeepEP-style combine restoration. The procedure keeps the dispatch handle, optional staged reductions, route-slot order, weights, masks, and final origin row attached to the same contribution.

Input: Expert-output rows; compatible dispatch handle; route weights and masks; declared local/scale-up reduction mode; optional bias
Output: One combined row for every origin token, in origin-token order

require the handle generation, rank group, route layout, and expert-buffer segments to match the expert outputs;
attach each valid expert output $y_{g,r}$ to its handle record $(g, r, e, w_{g,r}$, buffer slot), define its logical contribution as $c_{g,r} \leftarrow w_{g,r} y_{g,r}$, and give every invalid slot zero contribution;
if *local reduction is enabled* **then**
    stably group contributions by destination rank and reduce each group in route-slot order;
if *scale-up reduction is enabled* **then**
    stably group the surviving contributions by scale-up domain and reduce each group in route-slot order;
move every surviving contribution to the origin rank named by g;
foreach *origin token* g **do**
    initialize z_g from the declared bias or zero;
    accumulate valid contributions in route-slot order and write z_g to the local row encoded by g;
return *combined tensor and the reduction record needed by the correctness oracle*

Take token 1 from the running example. It is routed to expert 2 on rank 1 and expert 7 on rank 3. Suppose the two expert outputs are

$$y_{1,0} = (2, 0, 1, 1), \qquad y_{1,1} = (0, 4, 1, 3),$$

and the route weights are $w_{1,0} = 0.65$ and $w_{1,1} = 0.35$. The combined row is

$$z_1 = 0.65(2, 0, 1, 1) + 0.35(0, 4, 1, 3) = (1.3, 1.4, 1.0, 1.7).$$

If the two route slots are swapped while the same weights are retained, the first coordinate becomes 0.7 instead of 1.3. If the row is written into token 0's output slot, every coordinate of z_1 can be numerically right and still be semantically wrong.

Definition 26.4.2 (Ordered route-slot accumulation) *Ordered route-slot accumulation reduces the route outputs for a token in the same slot order used by the routing record. For top-k routing, the abstract operation is*

$$z_g = b_g + \sum_{j=0}^{k_{\text{top}}-1} w_{g,j} y_{g,j},$$

where b_g is an optional initial value such as a bias contribution or zero.

Algorithm 14 begins by treating handle compatibility as a gate rather than rediscovering dispatch layout from tensor shape. It then forms each logical contribution by applying its attached route weight exactly once before any optional transport reduction; an implementation may fuse that multiplication into a reduction kernel, but the record must preserve the same $c_{g,r} = w_{g,r} y_{g,r}$ boundary. Next, it performs only the reduction stages selected by the declared mode, using stable route-slot order so different hierarchy choices remain auditable. Finally, it decodes each global origin id and writes exactly one combined row at that origin, which is why correct network arrival without the saved inverse state is insufficient.

The word "ordered" in Definition 26.4.2 matters because reductions are not always exact in finite precision. A BF16 accumulation over two vectors is usually stable enough for this toy arithmetic, while a large production hidden dimension and multiple reductions can expose different rounding paths. DeepEP's tests distinguish configurations that reduce locally, reduce within scale-up groups, or avoid multiple reduction, and they compare the final tensors with reference results [19]. The chosen reduction structure must be part of the correctness and measurement record, even when the kernel details stay outside the local discussion.

Warp-level sparse gather and reduction

1. Each top-k lane contributes a source-slot index or an invalid marker. Depending on the selected layout, lanes first deduplicate equal destination ranks. A ballot forms the valid mask.
2. `compute_topk_slots` repeatedly takes the lowest set bit and uses an unconditional warp exchange to compact valid source slots to the front. This preserves increasing route-slot order while avoiding a divergent load branch for each missing contribution.
3. `ldg_with_gez_pred` initializes a vector to zero and executes the global load only when its slot index is nonnegative. Invalid lanes therefore contribute a defined zero rather than stale buffer contents.
4. With no bias and at most two valid compacted slots, the fast path adds packed BF16 pairs directly. If a third slot is valid or either bias is present, the general path converts packed BF16 values into FP32 pairs, accumulates all valid contributions, and rounds once when writing the BF16 shared row.
5. After the declared local or scale-up reduction, a proxy fence makes the shared row visible to TMA. The movement path can then store or forward that row; the final combine epilogue still applies the saved origin and route policy.

This sequence implements the zero-contribution rule from Definition 26.4.2; the predicate is a memory-safety and control-flow mechanism, not permission to reorder semantic route slots [19, 66].

Table 26.8.: The pinned combine helper selects its numerical path from the compacted valid-slot list and optional bias state.

Compacted reduction case	Kernel path	Consequence for the oracle
Invalid source slot	Predicated load remains at its zero initializer	A masked route cannot import stale memory into the sum.
One or two valid slots, no bias	Packed BF16 pairwise addition	The local fast path has a short rounding path, but its result must still be checked under the recorded BF16 tolerance.
Third valid slot or optional bias	FP32-pair accumulation followed by BF16 conversion	A different accumulation tree and rounding point are part of the declared reduction mode.

The reader-visible reduction record in Table 26.8 includes the precision branch. In the running token-2 case, experts 4 and 5 produce two valid local contributions. When the selected mode permits local reduction and supplies no bias, those two slots take the packed-BF16 fast path. Adding a third valid fragment or a bias switches the same conceptual sum to FP32-pair accumulation. The

mathematical destination z_2 is unchanged, but its finite-precision execution path and tolerance record are not.

The same operators form the training backward path. DeepEP V2's released training example makes the autograd duality explicit: the backward of dispatch is a combine, while the backward of combine is a dispatch [19]. If forward dispatch maps origin rows x_g to expert-owner rows, its input-gradient path must apply the inverse communication and reduction represented by combine. Conversely, if forward combine maps expert contributions back to z_g, its input-gradient path must fan $\partial \mathcal{L} / \partial z_g$ back to the expert-owner rows through the saved handle.

Route weights add a second gradient attachment. The released `dispatch_backward` example calls `ElasticBuffer.combine` with gradients for received route weights. It obtains both the combined activation gradient and the gradients in the original top-k weight layout. The resulting oracle is therefore not just "the tensors came back". Activation gradients must return to origin-token order, and route-weight gradients must return to the same origin-token and route-slot coordinates used in forward dispatch.

Autograd edge	Released communication operator	Backward invariant
Dispatch backward	Combine gradient rows with the forward handle and optional received route-weight gradients	Restore activation and route-weight gradients to original token and route-slot order.
Combine backward	Dispatch the combined-output gradient with the forward handle	Recreate the expert-owner row layout consumed by forward combine.

Table 26.9.: Forward/backward duality in the released DeepEP V2 example. The forward handle is saved communication state, not optional optimization metadata.

The two released autograd edges are explicit in Table 26.9: each backward communication operator is the forward operator's semantic dual, using the saved handle to preserve row and route-slot identity.

Token	Expert ranks	Expert outputs	Final destination
$g = 0$	0 and 3	$y_{0,0}, y_{0,1}$	rank 0, local row 0
$g = 1$	1 and 3	$y_{1,0}, y_{1,1}$	rank 0, local row 1
$g = 2$	2 and 2	$y_{2,0}, y_{2,1}$	rank 0, local row 2
$g = 3$	0 and 1	$y_{3,0}, y_{3,1}$	rank 0, local row 3

Table 26.10.: Combine restoration for the four origin tokens on rank 0. The final row order is the origin order rather than the receive order at any expert rank.

The rank-0 slice in Table 26.10 instantiates the restoration rule. Each expert output can travel through a different owner rank, while all four final rows return to rank 0 in token order. Figure 26.4 focuses on token 1 and makes the row-restoration part explicit.

Masked route slots introduce another boundary. If an expert id is invalid or masked out, the corresponding route slot must not contribute to z_g. DeepEP's reference code uses invalid markers for route slots that are not local to a destination rank and zeroes masked selections in the generated pre-combine data [19]. The mathematical rule is simple: the sum ranges over valid route slots only, or invalid slots have zero contribution. An implementation that lets a masked slot carry stale memory into combine violates the masked-slot invariant even if all valid slots were routed correctly.

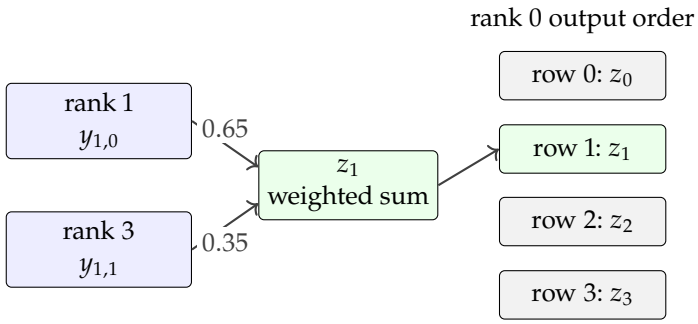

Figure 26.4.: Combine reduces route-slot outputs and writes the result into the origin-token row. Receive order and expert-owner rank do not determine the final row index.

The dispatch handle from Section 26.3 is what prevents this section from becoming guesswork. It binds received rows to origin ids, route-slot positions, expert segment counts, and destination buffer slots. Combine is correct when that binding is used to return the weighted expert outputs to the original token order under the stated reduction policy.

26.5. DeepEP V2 Surface and Generated Kernel Chain

The V2 surface realizes the row-identity, handle, and ordered-combine contracts established by the round trip.

Once dispatch identity, handle state, and combine restoration are fixed, repository code shows how one release realizes them. DeepEP V2 organizes its expert-parallel operations around `ElasticBuffer` [19]. It provides both high-throughput and low-latency paths. The interface described here follows the DeepEP snapshot in Appendix A. The same recorded release calls itself fully JIT: it compiles every kernel at runtime instead of compiling CUDA during package installation. In this chapter, that repository-wide claim is traced through the V2 elastic dispatch, combine, barrier, and epilogue kernel families. Experimental PP, CP, and Engram kernels remain outside the worked communication path.

The data path has four objects. A router produces token-expert assignments. Dispatch moves selected token rows to the ranks that own the experts. Expert kernels transform those rows. Combine returns weighted expert outputs to the original token order. DeepEP owns the communication object between routing and expert compute: buffers, handles, events, route metadata, and reduction order.

Concept-to-code map

Generated launch state enters through `https://github.com/deepseek-ai/DeepEP/blob/60d44037a702f651a6e18bd4aea65ed8409051c2/csrc/jit/launch_runtime.hpp`; compilation and loading split across `https://github.com/deepseek-ai/DeepEP/blob/60d44037a702f651a6e18bd4aea65ed8409051c2/csrc/jit/compiler.hpp` and `https://github.com/deepseek-ai/DeepEP/blob/60d44037a702f651a6e18bd4aea65ed8409051c2/csrc/jit/kernel_runtime.hpp` [19].

- **Extension boundary.** The build declares `deep_ep._C`, and the pybind initializer exposes the callable C++ surface to Python.
- **Buffer and handle state.** `ElasticBuffer` owns the V2 communication buffers; `EPHandle` represents reusable dispatch state; `EventOverlap` carries stream dependencies.

- **Dispatch and combine.** `ElasticBuffer.dispatch` and `ElasticBuffer.combine` move rows and reduce returned expert outputs while preserving route identity.
- **Generate and identify.** Runtime classes such as `DispatchRuntime` and `CombineRuntime` render one template specialization through `generate_impl`; `LaunchRuntime::generate` binds the generated source to the transitive DeepEP-header digest.
- **Cache, load, and launch.** `Compiler::build` derives a digest-named cache identity from the kernel name, compiler signature, flags, architecture, and source. `KernelRuntime` loads the selected CUBIN and symbol, and `LaunchRuntime::launch` supplies launch and call state.

That last kernel-path bullet can otherwise sound like one opaque launch. The recorded V2 implementation instead exposes a short generated launch chain. Those boundaries matter when reading a profiler trace, deciding when expert GEMM may consume received rows, and identifying which phase must preserve the dispatch handle.

Definition 26.5.1 (V2 kernel chain) *At the recorded DeepEP revision, a V2 dispatch call has two generated phases: dispatch movement, whose notify warps derive counts and prefix sums while its dispatch warps take the direct or hybrid path, and a receive-layout copy epilogue. After expert computation, combine has two generated phases: direct or hybrid return movement with optional local reduction, then final origin-row materialization. Table 26.11 gives the exact runtime and kernel symbols. Expert GEMM is outside DeepEP. Movement kernels contain their own communication barriers; the separate explicit buffer-barrier runtime is control state rather than an additional payload-transform stage in every dispatch or combine call [19].*

Table 26.11 expands the V2 kernel chain in Definition 26.5.1 into its dispatch and combine phases.

The receive-layout epilogue solves a different problem from transport. The movement kernel has delivered rank-major token records, but expanded grouped expert compute needs one row per valid local route in expert-major segments. The epilogue must perform that ragged transpose while preserving the inverse address that combine will later follow.

Algorithm 15 has four chronological phases. First, its launch-dependency gate prevents the epilogue from reading a record before dispatch publication. Second, the rank prefix P maps the linear receive index back to the correct scale-up sender buffer interval. Third, each valid local route either reserves one unique expert row from the cursor seeded at that expert's aligned start offset or reuses the exact row saved by a compatible cached handle. Finally, fresh mode records origin, transport-slot, and per-route destination metadata; optional padding initialization uses the semantic counts rather than treating cursor endpoints as model rows. The correctness invariant is that every valid local route owns exactly one row inside its expert segment, every inverse destination names that same row, and no alignment slot becomes a semantic contribution [19].

Table 26.11.: The recorded DeepEP V2 generated kernel chain. "Notify" is a warp role inside the generated dispatch movement family, not a separate `DispatchPrologueRuntime` launch.

Round-trip phase	Recorded source surface	State produced or consumed
Notify and move	`DispatchRuntime`; `dispatch_impl` or `hybrid_dispatch_impl`	Deduplicated rank counts, expert counts, prefix sums, payload, scales, route fields, and origin metadata. Cached mode may reuse accepted slot state instead of assigning notify warps.
Materialize receive layout	`DispatchCopyEpilogueRuntime`; `dispatch_copy_epilogue_impl`	Expert-major received rows, aligned segment boundaries, optional scale packs, top-k fields, and source metadata after dispatch visibility is established.
Transform rows	Grouped expert GEMM, outside DeepEP	Expert outputs indexed by the received expert-major layout and saved inverse map.
Move contributions	`CombineRuntime`; `combine_impl` or `hybrid_combine_impl`	Returned expert fragments, with topology-dependent movement and optional local reductions.
Restore origin rows	`CombineReduceEpilogueRuntime`; `combine_reduce_epilogue_impl`	Route-weighted accumulation in original token order, including the final result-buffer reduction when required.
Fence communication state	`BarrierRuntime`; `barrier_impl`, plus internal `comm::gpu_barrier` calls	The explicit buffer fence is a control operation; embedded barriers and the returned `EventOverlap` expose per-call readiness.

Algorithm 15: Expanded receive-layout materialization in the pinned `dispatch_copy_epilogue_impl`. The stable obligation is a reversible expert-major placement; atomic cursors, cached indices, TMA, and warp scans are the recorded implementation mechanisms.

Input: Visible rank-major receive records; inclusive scale-up sender-rank boundaries P, with $P_{-1} = 0$; mutable expert cursors C_e initialized from exclusive aligned offsets; semantic counts M_e; optional compatible cached inverse map

Output: Expert-major rows with matching scales and weights; source metadata and inverse destinations; zeroed alignment gaps when requested

wait until the dispatch movement stores are visible to the dependent launch;
foreach *received record index i assigned to this warp* **do**
    locate the sender-rank interval $P_{r-1} \le i < P_r$ and stage that record;
    validate its route slots and retain only experts local to this rank;
    foreach *valid local route to expert e* **do**
        choose destination d from the cached inverse map, or atomically fetch and increment C_e in fresh mode;
        copy the hidden row and its matching scale pack and route weight to d;
        record d as this route's reduction source when building a fresh handle;
    record origin and sender/slot metadata when building a fresh handle;
finish any fresh hybrid linked-list state needed by combine;
if *expanded padding must be initialized* **then**
    compute $p_e = \mathrm{align}(M_e, A) - M_e$ for every local expert;
    warp-scan the p_e values to map each padding ordinal to its physical gap;
    zero the hidden row, scale pack, and weight at every mapped padding slot;
return *consumer-ready expert segments and their reusable inverse map*;

Example 26.5.1 (One token record through the generated chain) In the four-rank running example, let token $g = 0$ route to experts 1 and 6 and token $g = 2$ route to experts 4 and 5. The notify warps inside `DispatchRuntime` count one destination for each distinct owner, so the two experts owned by the same rank for $g = 2$ share one transported activation record. Dispatch warps move each record with its scale and route metadata. `DispatchCopyEpilogueRuntime` then places valid rows in the aligned expert-major buffers that grouped GEMM consumes. After GEMM, `CombineRuntime` returns the expert fragments, and `CombineReduceEpilogueRuntime` uses the saved origin and route-slot map to write, for example, $z_0 = w_{0,0}y_{0,0} + w_{0,1}y_{0,1}$ into origin row 0. A profiler may show multiple launches, but every launch belongs to one reversible row-identity protocol.
For the rank-2 layout fixture from Section 26.3, let $M_4 = 1$, $M_5 = 2$, and $A = 4$. Fresh expert cursors begin at physical offsets 0 and 4. Token $g = 2$'s two local routes reserve row 0 for expert 4 and one of rows 4–5 for expert 5; token $g = 7$'s expert-5 route reserves the other. The order of the two expert-5 atomic reservations is not a semantic promise, so the handle records the actual destinations. The padding pass maps the five remaining pad ordinals to rows 1–3 and 6–7 and zeros their payload, scale, and weight storage. A compatible cached dispatch skips fresh allocation and reads those saved destinations; it is safe only under Definition 26.3.4.

Figure 26.5 makes the load-bearing claim visual: V2 is a generated multi-launch round trip whose handle connects receive layout to combine restoration. The explicit buffer barrier is adjacent control state, not another expert-row transformation.

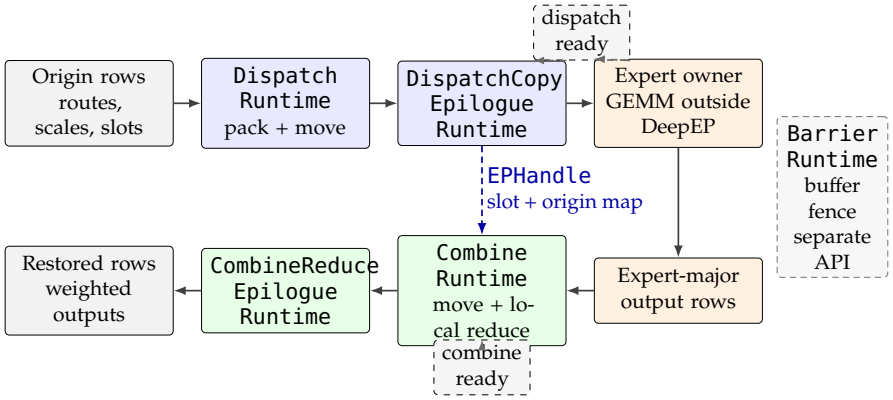

Figure 26.5.: The pinned V2 dispatch–expert–combine chain. Solid arrows carry token records or expert results; dashed edges carry the inverse-map metadata and completion dependencies. The ready markers expose the publication gates around expert compute. `BarrierRuntime` remains a separate buffer fence rather than another token-row transformation.

Definition 26.5.2 (DeepEP JIT identity) *A DeepEP JIT identity is the generated kernel family and name together with the exact generated source and transitive header digest, compiler signature and flags, and target GPU architecture. Payload addresses and values are call data, not identity. A topology branch, hidden width, top-k, expert count, or selected SM/QP configuration that changes generated code creates a different artifact. Equal Python API names or output shapes do not establish equal JIT identity.*

Three state classes must remain separate. *Generated identity* contains the specialization fields compiled into source. *Launch state* contains pointers, current valid-row counts, rank indices, streams, and other arguments supplied to one invocation.

Measurement state records the repository revision, toolchain, cache directory, initial cache state, and timing boundary. Changing launch state need not produce a new CUBIN; changing measurement state need not change dispatch semantics, but it changes which performance claim the observation supports.

Figure 26.6 follows one generated identity to an executable kernel. The lookup has two reuse levels: a process may already hold the loaded runtime, or it may load a valid published CUBIN from the digest directory. Only a miss enters the compiler path.

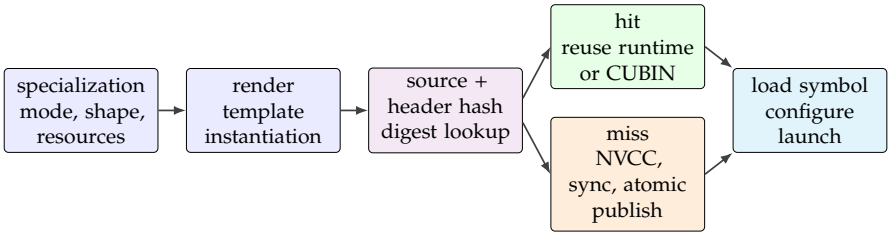

Figure 26.6.: DeepEP's source-dependent JIT cache lifecycle. A hit bypasses compilation. A miss builds in temporary storage and atomically publishes one complete digest directory before both paths load and launch the kernel.

On a miss, the pinned compiler writes the generated CUDA source, invokes NVCC for the target architecture, and writes `kernel.cubin`. It synchronizes the temporary directory to storage and atomically renames that directory to its digest path. If another rank publishes the same identity first, the losing rank loads the completed winner. A disk entry is valid only when both generated source and CUBIN are present. `KernelRuntime` then discovers the eligible symbol, loads it, constructs the launch configuration, and submits the ordinary arguments on the selected CUDA stream [19].

Example 26.5.2 (One dispatch identity across three calls) Fix the four-rank running example's hidden width, direct topology, payload layout, and SM/QP choices. Call A begins with empty process and disk caches, so it generates source, compiles and publishes a CUBIN, loads the symbol, and only then moves rows. Call B changes payload values and addresses but preserves every generated field, so it may reuse Call A's identity. Call C selects hybrid topology or a different hidden width. Its generated source changes, so it needs a different digest and artifact even though the public operation is still dispatch. This is a source-derived lifecycle trace, not an observed cold-start timing.

The timing boundary must say whether generation, lookup, compilation, publication, module and symbol loading, launch, execution, and synchronization are inside the interval. An empty-cache first call, a disk-cache process start, and a process-warm repeated launch are three different observations. The JIT failure table in Section 26.15 later turns the same lifecycle into compiler, cache-integrity, publication-race, identity-drift, and timing checks.

The Python package checks CUDA and NVIDIA Collective Communication Library (NCCL) state before importing the extension. NCCL is GPU communication software; NVLink names a local interconnect, while RDMA describes remote-memory access over an eligible cross-node path. They therefore occupy different layers [64, 19]. Initialization establishes the buffer configuration and communication context; dispatch returns received rows plus metadata and an optional overlap event; combine consumes expert outputs, route weights, and compatible

handle state. Cached dispatch can avoid rebuilding a known layout, but only while the handle still matches the participating ranks, capacities, and route shape.

V2 is not a rename of the legacy interface. It changes the backend, the resource-selection policy, the exposed buffer object, and one important latency tradeoff. Tables 26.12 and 26.13 keep those changes beside the properties that remain constant.

Surface	Legacy V1	Released V2
Communication backend	NVSHMEM-based normal and low-latency paths	NCCL Gin, able to reuse an existing NCCL communicator
Python surface	`Buffer`, with separate normal and low-latency methods	One `ElasticBuffer` dispatch/combine surface for training, prefill, and decode
Scale and resources	Configuration helpers and tuned choices in the legacy examples	Advertised EP domains up to 2048 and analytical SM/QP selection, with manual call-site overrides

Table 26.12.: Released V1-to-V2 backend, surface, and resource changes.

The generation must therefore be part of every call-site and benchmark record. A V1 integration does not prove that a runtime exercises V2, and an analytical V2 choice does not remove the need to record effective SM and QP counts.

Property	Legacy V1	Released V2
Buffer and latency tradeoff	Low-latency mode includes a zero-SM RDMA path	Buffer consumption is larger, and zero-SM RDMA low-latency EP is removed
Semantic invariant	Dispatch carries rows and routing metadata; combine returns contributions to origin order	The same identity, route-slot, and weighted restoration invariant

Table 26.13.: Released V1-to-V2 latency change and semantic continuity. The two tables describe public interfaces and repository claims, not a hosted DeepSeek deployment.

V2's lower SM use for reported V3-like training does not preserve every V1 latency mechanism. The mathematical correctness oracle does survive the generation change.

26.6. Analytical SM and QP Selection

Communication SMs compete with attention and expert kernels for execution capacity. Too few communication SMs may fail to drive the available NVLink or RDMA bandwidth; too many can occupy resources that independent compute could use. QPs have a related tradeoff: channels need enough network work queues to make progress, while unnecessary QPs add doorbell and connection state. V2 therefore makes the resource choice a traffic model rather than an unexplained constant or a mandatory autotuning sweep [19].

For a balanced gate over E experts, top-k routing, and G equal owner groups, the pinned selector first estimates the number of distinct groups touched by one token:

$$D(G, E, k) = G\left(1 - \frac{\binom{E-E/G}{k}}{\binom{E}{k}}\right).$$

This is an occupancy calculation. The fraction subtracted from one is the probability that none of the k selected experts belongs to a particular group. Multiplying by G gives the expected number of distinct destination groups. DeepEP uses the corresponding expected scale-up and scale-out fanout to derive normalized SM-read work m_r, SM-write work m_w, and NVLink/RDMA traffic fractions.

Definition 26.6.1 (DeepEP analytical SM/QP selector) *Let fabric b be the modeled bottleneck, with traffic fraction τ_b and bandwidth B_b. Let b_r and b_w be the assumed per-SM HBM read and write bandwidths. The pinned balanced-gate selector forms*

$$S_{\text{raw}} = \max\left\{\frac{B_b}{\tau_b}\frac{m_r}{b_r}, \frac{B_b}{\tau_b}\frac{m_w}{b_w}\right\}.$$

When overlap with compute is preferred, it adds 25% headroom, enforces a four-SM floor, rounds up to an even count, and caps the result at the physical SM count:

$$S = \min\left\{S_{\text{device}}, \text{align}_2\left(\max\{4, \lceil 1.25 S_{\text{raw}}\rceil\}\right)\right\}.$$

Without the overlap preference, the implementation raises the pre-cap choice to at least 64 SMs. Given S, direct mode requests

$$Q_{\text{direct}} = \min\{Q_{\text{alloc}}, S, 9\},$$

whereas hybrid mode requests

$$Q_{\text{hybrid}} = \min\{Q_{\text{alloc}}, 16S + 1\}.$$

The extra hybrid QP is the notify path; the factor 16 assigns independent channel capacity before the allocation cap.

S is the communication-SM budget and Q is the QP budget; both come from the traffic model, not rank count alone.

Example 26.6.1 (One source-derived selector calculation) Consider a teaching configuration with one eight-rank scale-up domain, $E = 64$, $k = 8$, modeled NVLink bandwidth $B_{\text{NVL}} = 800$ GB/s, per-SM read/write assumptions $b_r = 200$ and $b_w = 50$ GB/s, and a device with at least 24 SMs. The occupancy expression gives $D(8, 64, 8) \approx 5.43$. In the pinned one-domain branch, $m_r = 1/D \approx 0.184$, $m_w = 1$, and $\tau_{\text{NVL}} = 7/8$. The write term dominates:

$$S_{\text{raw}} = \max\left\{\frac{800}{7/8}\frac{0.184}{200}, \frac{800}{7/8}\frac{1}{50}\right\} \approx 18.29.$$

Headroom gives $\lceil 1.25(18.29)\rceil = 23$, and even alignment selects $S = 24$. With sufficient allocation, direct mode therefore selects

$$Q_{\text{direct}} = \min(24, 9) = 9.$$

This is a transparent execution of DeepEP V2's selector model under declared hypothetical bandwidths, not a measured optimum.

Table 26.14 summarizes the selector from routing assumptions through its validity boundary.

Selector step	Source-visible rule	Why it exists
Expected fanout	Estimate distinct owner groups with $D(G, E, k)$.	Convert top-k routing into expected scale-up and scale-out work.
Bottleneck fabric	Compare modeled RDMA and NVLink traffic time.	Size SM work against the slower active transport rather than raw rank count.
SM budget	Balance modeled HBM read/write demand, add 25%, floor at four, round even, and cap at device SMs.	Drive communication while preserving an explicit overlap-versus-peak-throughput policy.
QP budget	Direct: at most nine and no more than S; hybrid: $16S + 1$; cap both by allocated QPs.	Limit direct-mode doorbell overhead but give hybrid channels and notification independent capacity.
Validity boundary	Balanced gate only; the pinned code marks expanded dispatch, multiple reduction, and V3 group-limited routing as unsupported by this model.	Require measurement or a declared manual override outside the model's domain.

Table 26.14.: How the pinned DeepEP V2 selector turns routing and fabric assumptions into SM and QP budgets.

The selector's final row marks its validity boundary. That boundary matters when a release-level performance result uses a routing regime outside the model.

The README separately reports that a V3-like legacy-training configuration reduced communication use from 24 SMs in V1 to 4–6 in V2 while maintaining equivalent or better reported performance. It also reports up to 1.3× peak performance and up to a 4× SM-count saving. Those are repository performance claims, not outputs of Example 26.6.1. In particular, the pinned selector warns not to apply its balanced-gate formula directly to V3's group-limited gate. A defensible run must record the effective SM/QP counts, the overlap preference, detected or overridden fabric bandwidths, routing regime, and any manual override [19]. The exact release rows and their logical-bandwidth boundary appear in Table 26.29 in the benchmarking section.

Resource-selection concept-to-code map

- **SM model.** `ElasticBuffer.get_theoretical_num_sms` implements expected fanout, traffic selection, HBM read/write balancing, headroom, floor, alignment, overlap preference, and device cap.
- **QP model.** `ElasticBuffer.get_theoretical_num_qps` implements the direct and hybrid formulas and applies `num_allocated_qps`.
- **Automatic call path.** Passing zero for `num_sms` or `num_qps` invokes the corresponding selector; nonzero values are explicit overrides that belong in the run record.

Table 26.15 separates the API, reference, test, and kernel responsibilities.

The reference utilities construct top-k assignments, compare dispatch with a reference all-to-all transformation, and compare combine with ordered weighted accumulation. These checks make route identity and reduction order explicit before bandwidth enters the discussion. FP8 dispatch additionally carries scale factors with the payload; expanded dispatch retains route-slot identity when one token contributes several rows.

Table 26.15.: DeepEP V2 layers and their responsibilities.

Layer	Main objects	Technical responsibility
Python API	`ElasticBuffer`, `EPHandle`, `EventOverlap`	Buffer ownership, callable dispatch/combine variants, cached state, and stream dependencies
Reference utilities	Top-k indices, rank counts, origin ids, masks, and ordered accumulation	Simple dispatch and combine semantics for comparison
Tests	Generated routing cases, assertions, and profiling variants	Check dispatch, cached dispatch, expanded dispatch, combine, and reduced combine
Kernel and JIT layer	Communication kernels, PTX helpers, generated launch state, and network parameters	Execute the selected high-throughput or low-latency path

The repository also retains legacy V1 material and experimental Engram, pipeline-parallel, context-parallel, AGRS, and remote-memory-adjacent paths [19]. They illustrate increasing integration pressure: dispatch and combine may share buffers, events, and routing metadata with other parallel dimensions. The V2 surface used in this chapter remains the `ElasticBuffer` dispatch/combine path, with those adjacent paths treated only where they change its state or dependencies.

26.7. Throughput Path, Decode Path, and Event Overlap

The same dispatch/combine semantics can serve different latency regimes. In training and inference prefilling, many tokens may be routed at once, so the communication path is usually judged by throughput and overlap with expert compute. In inference decoding, each step may involve a much smaller token set, so per-step latency and layout reuse become more visible. DeepEP's README uses the same V2 `ElasticBuffer` interface for both the training or prefill example and the decode example, while showing a cached-handle pattern for decoding [19].

Definition 26.7.1 (Event overlap) *Communication event overlap is the use of a stream event or equivalent dependency so that communication can run concurrently with independent compute, and the consuming stream waits only before it reads communication outputs. The overlap changes the timeline; it does not change dispatch or combine semantics.*

Applying event overlap from Definition 26.7.1, rank 0 dispatches four token rows in a prefill-like use of the running example. While the communication stream moves those records for tokens 0 through 3, the compute stream might perform independent work that does not need those received expert rows. Before local expert kernels consume the received rows, the compute stream waits on

the event. A compact timeline is:

stream	t_0	t_1	t_2
communication	dispatch launch	token transfer	event recorded
compute	independent work	wait if needed	expert GEMM

If the independent work lasts longer than the transfer, dispatch latency can be largely hidden. If the transfer lasts longer, the compute stream waits and the remaining time becomes visible as a bubble. The mathematical output is the same in either case. Figure 26.7 shows this as a stream dependency rather than a change to dispatch semantics.

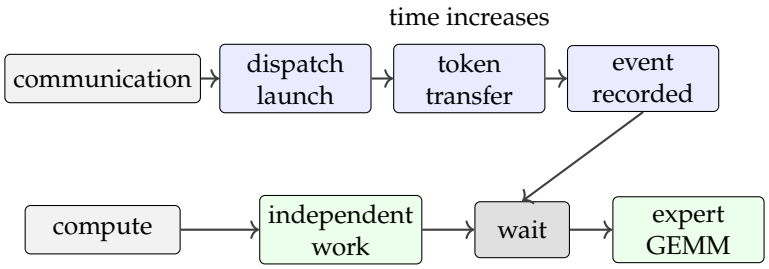

Figure 26.7.: Event overlap is a timing relation. The dispatch and combine semantics stay fixed; only the point where compute must wait becomes visible in the record.

Decode adds a different optimization pressure. Suppose the current decode step uses the same route layout as the previous step: token rows have new values, but the top-k expert destinations and buffer layout are unchanged. Then a cached handle can avoid recomputing the layout and avoid a synchronization point that would otherwise expose CPU-side work. If the route layout changes, reuse is unsafe. The invariant from Definition 26.3.4 is therefore not a formality; it is the correctness condition that separates a useful latency optimization from a stale-layout bug.

Path	Main pressure	Record required before a claim
Training or prefill	Throughput over many token rows and overlap with expert compute	token count, hidden width, top-k, dtype, rank topology, warmup, bytes, correctness check
Decode with fresh layout	Small-batch latency plus layout construction	same as above, plus layout construction and synchronization costs
Decode with cached handle	Reuse of layout metadata across compatible steps	cached-handle validity condition and stale-handle test

Table 26.16.: The same dispatch/-combine semantics apply under different timing pressures.

Table 26.16 shows why a single latency or bandwidth number cannot describe every use of the same communication semantics. The routing and restoration invariants stay fixed, while the timing record changes with the path.

Compare throughput, fresh-layout decode, and cached decode only under explicit timing and reuse boundaries.

Event and graph lifetime. `EventOverlap` is also an ownership object. The released V2 wrapper retains the underlying event handle and the tensors whose stream lifetime it protects. A consumer may call `current_stream_wait()` or use the context-manager form, but it must keep the wrapper alive until

Algorithm 16: Safe asynchronous DeepEP consumption with optional cached layout. The compatibility comparison is an integration-side guard because cached dispatch reuses accepted layout instead of discovering new routes.

Input: Payload; current routing-layout identity; optional cached handle; consumer streams; graph-replay assumptions
Output: Received rows with safe consumer dependencies, a fresh-layout requirement, or a failed poisoned-call record

if *a cached handle is proposed* **then**
  compare its rank group, owner map, token bound, selected-expert layout, mode, and alignment with the current routing-layout identity;
  reject reuse and request fresh dispatch when any layout field differs;
launch fresh or cached dispatch and retain its received tensors, handle, and `EventOverlap` while independent work executes;
foreach *consumer stream that may read the received rows* **do**
  install a wait on the communication event before the first read;
  run any registered deterministic post-wait hook on that consumer path;
if *the call participates in graph replay* **then**
  require the captured buffers, handle identity, and every replay dependency to remain valid;
if *communication or replay fails* **then**
  retain the failure evidence, poison the call state, and require caller synchronization plus affected buffer or handle rebuild before later reuse;
  return *failed poisoned-call record; no rows are consumer-ready*;
release the event handle only after the last required wait has been installed;
return *consumer-ready rows and the retained layout identity*

every consuming stream has installed its wait. Releasing the handle at the first wait is valid only when no second stream or captured replay needs it. This is particularly important around CUDA graphs: V2 stores graph-relevant tensor lifetime in the event handle rather than relying on an ordinary one-stream `record_stream` call [19].

Table 26.17.: EventOverlap and CUDA-graph lifetime obligations.

Lifetime point	Required action	Failure if omitted
Asynchronous return	Retain `EventOverlap` and received tensors while independent work runs.	Communication-owned storage can be reused or freed before its consumer is ordered.
First consumer	Wait on the event before reading dispatch/combine outputs.	Expert compute or reduction observes incomplete data.
Multiple consumers or graph replay	Install every required stream dependency before releasing the event handle; preserve captured buffer and handle assumptions across replay.	The first wait succeeds while another stream or replay has no valid lifetime edge.
Final consumer	Release the handle only after the last required wait.	Early release turns a timing optimization into a use-after-lifetime risk.

Table 26.17 makes waiting for readiness distinct from retaining the event and tensor lifetimes needed by every consumer and replay.

Algorithm 16 first decides whether reuse is legal; cached dispatch is work only after the caller has established layout identity. It then separates independent compute from the readiness gate and installs that gate on every consumer, rather than releasing the event after the first wait. A communication or replay failure is

terminal for the current call: it returns no consumer-ready rows and rebuilds state only for later calls. Finally, the success path retains graph and buffer ownership until all consumers are ordered, preventing a shape-compatible stale handle or early release from becoming a silent routing or lifetime error.

The running example gives a full stale-handle test. Cache a handle for rank 0's route table $(1, 6), (2, 7), (4, 5), (0, 3)$. Token 2 originally used experts $(4, 5)$, both owned by rank 2. Now change only token 2's route to $(6, 7)$, both owned by rank 3. The input payload still has four rows, the top-k table still has shape 4×2, and the total number of deduplicated dispatch records still sums to seven. The ownership split is what changed.

Table 26.18.: A shape-preserving stale-handle bug. The tensor dimensions and total record count remain plausible, but token $g = 2$'s rank ownership changes from rank 2 to rank 3.

Trace point	Correct later call	Stale-handle bug	Why shape checks miss it
Route table	Token 2 uses $(6, 7)$, so both valid route slots are owned by rank 3.	The cached handle still describes token 2 as $(4, 5)$ on rank 2.	Both route tables are 4×2.
Split vector	Rank 0's deduplicated counts become $(2, 2, 0, 3)$.	The old handle reuses $(2, 2, 1, 2)$.	Both vectors sum to 7 token records.
Dispatch buffer	Token 2's record carries $g = 2$, weights $w_{2,0}, w_{2,1}$, and valid slots for owner rank 3.	Token 2's record is placed in the rank-2 slot described by the old handle.	The hidden row width and total packed payload size can remain compatible.
Expert-major buffer	Rank 3 runs expert 6 and expert 7 rows for $g = 2$.	Rank 2 either runs stale expert rows or interprets the buffer under the wrong owner map.	Expert outputs still have hidden width H.
Combine inverse map	The two weighted fragments return to origin rank 0, local row 2, as z_2.	Combine can write a tensor row of the right shape using fragments from the wrong owner rank.	The output is still a $4 \times H$ tensor.

Table 26.18 is the bug that a cached-handle test has to catch. The old handle preserves a plausible tensor shape while breaking the route record's ownership map. A correct replay therefore checks the origin id, destination owner rank, route-slot weights, buffer slot, and combine inverse map. Shape equality alone is not evidence that the MoE layer computed the requested experts.

The V2 call contract makes the limitation observable: when a cached `EPHandle` is supplied, fresh `topk_idx` and `topk_weights` must be absent. CPU synchronization for exact new receive counts is also disabled. Cached dispatch is therefore payload replay under an already accepted layout, not a mechanism for discovering that the gate chose different experts. The released kernel also disallows the cumulative expert receive-statistics tensor in cached mode, so load monitoring needs a fresh-layout sample or another declared counter source.

26.8. Network Topology and Transport Domains

DeepEP performance depends jointly on the software stack and rank topology. Its requirements include Hopper-class GPUs or compatible SM90 PTX support, CUDA and PyTorch versions, and NCCL as communication software [19, 64]. At the data-path layer, the README names NVLink for intranode movement and an RDMA network for internode movement. It reports full testing with InfiniBand. It describes RoCE compatibility as theoretical [19]. Software, local interconnect, and cross-node fabric are independent parts of the run configuration. Expert-parallel performance therefore depends jointly on the software stack and rank topology.

Table 26.19 distinguishes software eligibility from fabric settings that must be verified for each deployment.

Reproducibility therefore requires both categories: pinned software versions identify the software stack, while the fabric configuration identifies the effective network path and switch policy.

Definition 26.8.1 (Scale-up and scale-out domains) *A scale-up domain is the local high-bandwidth group of ranks within a node or closely connected device domain. A scale-out domain spans such groups across the network. In DeepEP, the elastic buffer API exposes logical and physical domain sizes so a run can distinguish scale-up ranks from scale-out ranks [19].*

Scale-up names the local device domain; scale-out names the cross-node domain. Route identity stays constant across both.

Map the running example onto two nodes with two GPUs each. The scale-up and scale-out terms in Definition 26.8.1 become concrete under this placement. Ranks 0 and 1 are on node A, and ranks 2 and 3 are on node B. For token 0, the route slot for expert 6 sends a row from origin rank 0 to owner rank 3, so it is scale-out traffic. For token 3, the route slot for expert 3 sends a row from rank 0 to owner rank 1, so it is scale-up traffic. Both destination ranks are remote relative to the origin token row. Their network costs and failure modes nonetheless differ. A reproducible bandwidth result reports the traffic split between NVLink-like scale-up paths and RDMA-capable scale-out paths.

Figure 26.8 follows one origin row through both paths. The topology changes the transport cost, but the hidden payload, origin identity, route slots, and route weights still have to reach expert compute and return through combine together [19].

The plate shows the direct teaching path to final expert owners. Hybrid mode adds intermediate forwarding and completion state while preserving the origin-id and combine-restoration semantics.

Definition 26.8.2 (Hybrid mode) *Hybrid mode is a multi-node communication mode in which communication is structured hierarchically across scale-up and scale-out domains. In DeepEP's elastic buffer documentation, hybrid mode changes communication accounting and can change how combine reductions are staged [19].*

The same token table can therefore generate different timing stories under different placement. If ranks 0, 1, 2, and 3 all sit inside one NVLink domain, rank 0's seven token rows use scale-up links. If ranks 0 and 1 are local and ranks 2 and 3 are remote, three of rank 0's seven deduplicated token rows cross

Requirement	Documented V2 behavior	Deployment consequence
V2 compute/-software	Hopper or compatible SM90 PTX support; CUDA 12.3 or newer, PyTorch 2.10 or newer, and NCCL 2.30.4 or newer	Treat these as the documented V2 floor for the pinned revision; capture exact versions and GPU architecture.
V2 links	NVLink for intranode movement and RDMA for internode movement	A multi-node result must identify both scale-up and scale-out domains and the effective path.
Fabric validation	DeepEP V2 has been tested on InfiniBand; RoCE compatibility is described as theoretically compatible, while tested support covers InfiniBand.	Preserve that scope and state the NIC, fabric, and routing configuration.
Traffic isolation	V2 recommends separating expert-parallel and other workloads with InfiniBand virtual lanes, selected by the `sl_idx` argument or `EP_OVERRIDE_RDMA_SL`.	State the effective service level and competing traffic rather than assuming isolation.
Adaptive routing	V2 recommends enabling adaptive routing under all network load conditions despite its additional latency.	Treat this as the repository's tested-environment recommendation, scoped to that environment; state whether it was enabled.
Congestion control	V2 reports congestion control disabled because it reduced maximum bandwidth and recommends low-priority virtual lanes for unavoidable congestion.	State the actual switch policy independently of DeepEP defaults.
PCI atomics	When hardware supports it, V2 recommends NIC `PCI_ATOMIC_MODE=4` to improve RDMA atomic performance.	This is privileged platform tuning: verify hardware support, retain the value, and retain an administrator-approved rollback.
Configuration lifetime	V2 builds persistent defaults for `EP_JIT_CACHE_DIR`, `EP_JIT_PRINT_COMPILER_COMMAND`, `EP_NUM_TOPK_IDX_BITS`, and `EP_NCCL_ROOT_DIR`; current environment variables may override them at import.	Preserve both build-time defaults and run-time overrides in a reproducibility manifest.

Table 26.19.: DeepEP V2 software eligibility and fabric settings. The target system must verify network recommendations before deployment.

nodes. Those rows carry four remote expert route slots. The dispatch semantics are identical in both cases; the topology description changes the interpretation of bandwidth, latency, and overlap. Table 26.20 keeps that distinction explicit when interpreting timing results.

Figure 26.9 connects that topology description to the JIT state that a measurement must also name.

Topology fixes which rows remain inside a scale-up domain and which cross the scale-out fabric. A run therefore states topology and transport alongside its JIT identity: source revision, generated artifact, cache state, and cold or warm timing interval.

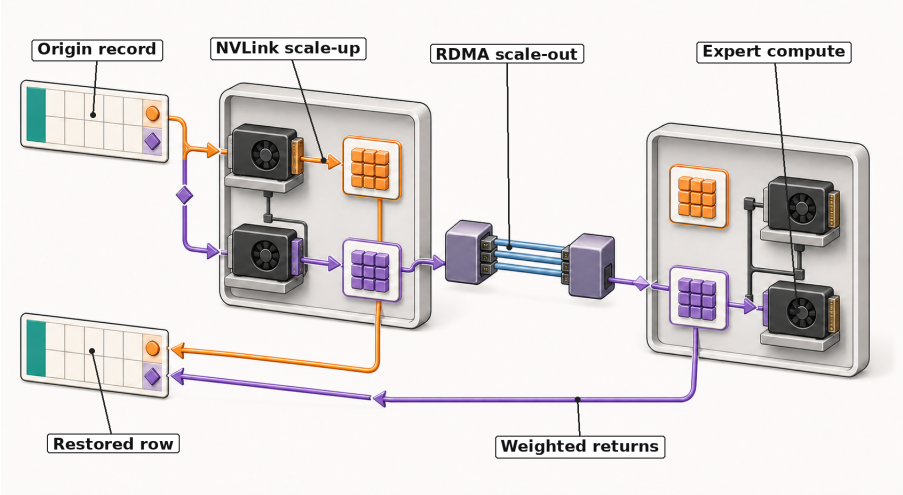

Figure 26.8.: One routed token across a two-node DeepEP topology. Dispatch preserves the hidden payload, origin identity, route slots, and weights while one selected expert is reached through the local NVLink scale-up path and another through the RDMA scale-out path; combine returns the weighted fragments to the original row. The plate abstracts switch wiring and kernel scheduling.

Table 26.20.: Single-node and multi-node DeepEP runs share the same dispatch/combine semantics, but they support different performance and deployment conclusions.

Question	Single node, scale-up only	Multi-node, scale-up plus scale-out
Rank placement	All expert-owner ranks in the running example fit inside one local device domain.	Ranks are partitioned across nodes; for the running example, ranks 0 and 1 can be local while ranks 2 and 3 are remote.
Dispatch ownership	The same route table, origin ids, route weights, handle, and inverse map define correctness.	The same identity and restoration invariants still apply; topology changes the path, not the token-ownership rule.
Traffic accounting	Rank 0's seven deduplicated rows use local scale-up links.	Rank 0's rows split across local scale-up and network scale-out paths; byte accounting reports each path independently.
Likely bottleneck	Local peer bandwidth, kernel scheduling, expert imbalance, and event overlap dominate the interpretation.	RDMA-capable path setup, hybrid staging, cross-node reduction, and timeout behavior shape the result.
Interpretation limit	A bandwidth row supports only the local scale-up domain it measured.	Report completed small-shape scale-out rows and blocked large-shape rows in their own categories; serving throughput requires a serving test.
Failure signal	A wrong result usually points back to handle validity, local slot layout, route weights, or reduction order.	The same correctness checks remain necessary, but launch, transport, and hybrid-mode failures can block a performance measurement before correctness is reached.

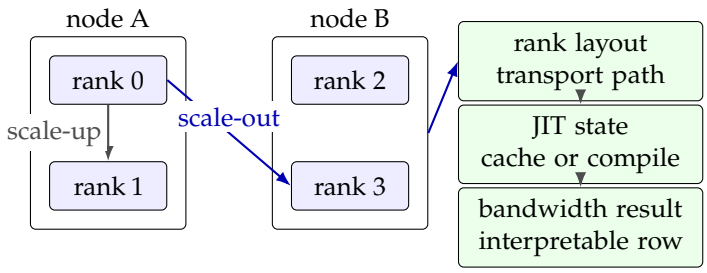

Figure 26.9.: The same dispatch rows have different timing meaning under different rank placement. A bandwidth row is interpretable only after the topology path and the JIT state are stated.

26.9. Direct and Hybrid Transport Paths

The same token record can cross either a one-stage direct path or a two-stage hybrid path. Those modes preserve token semantics, but they cannot share an implicit state machine because channel reuse, progress publication, and completion occur at different owners.

The direct path sends each record to its final rank. The hybrid path first moves a record across a scale-out rail and then forwards it inside the destination scale-up domain. Treating both as "an RDMA put" erases the intermediate owner, the progress record, and the point at which a channel can be reused. At the pinned revision, the two modes use separate generated kernels because the one-stage and two-stage protocols maintain different ownership and completion state [19].

Earlier kernel examples centered on global-to-shared TMA loads. The direct publication path below also uses the reverse shared-to-global TMA store direction. Dispatch workers assemble a record in shared memory, and the TMA store moves it toward the global or send buffer. The path proves those stores complete before issuing the dependent RDMA operation. This store completion is not the load transaction-barrier event from the earlier teaching trace [19].

One DeepEP record through TMA and remote publication

1. An elected lane initializes the dispatch warp's shared `mbarrier` and issues a global-to-shared TMA load for the hidden row. Optional FP8 scale-factor packs arrive through `cp.async` and join the same shared-buffer completion protocol.
2. While those bytes move, warp lanes attach the top-k fields, weights, and origin id. A shared-to-async-proxy fence orders those ordinary shared-memory writes before a later TMA store can read the assembled record.
3. The elected lane registers the promised hidden-row byte count and waits for the current barrier parity. A successful wait proves that the asynchronous load obligations for this use of the shared buffer are complete; it does not yet prove a remote write or peer rendezvous.
4. A peer-accessible destination can receive the record through a shared-to-global TMA store. Otherwise, TMA first stages the record in a global send buffer; the warp waits for that store before issuing the dependent NCCL Gin put.
5. The final movement-kernel barrier waits outstanding TMA stores, flushes any payload-carrying QPs on Gin paths, and only then signals peers. That later barrier, rather than the load-side transaction wait, establishes consumer-ready arrival.

The parity and promised-byte facts are exactly the ones defined by Definition 19.8.1. DeepEP binds that generic copy-completion object to one packed token record and then adds a separate transport-completion proof.

Direct and hybrid kernel map

- **Direct dispatch.** `dispatch_impl` implements final-destination movement.
- **Direct combine.** `combine_impl` implements the one-stage return path.
- **Hybrid dispatch.** `hybrid_dispatch_impl` implements scale-out receipt followed by scale-up forwarding.
- **Hybrid combine.** `hybrid_combine_impl` implements the reverse two-stage reduction and return path.

Table 26.21.: Concept-to-code concordance for the pinned DeepEP V2 kernels. The PTX helpers implement role-local state transitions; none alone supplies the whole dispatch/combine contract.

Kernel role	State and representative helpers	Algorithmic obligation
Notify/count	Destination ranks and expert counts; `match`, `deduplicate`, `warp_inclusive_sum`, `red_add`	Elect one lane per distinct destination, reduce counts across SMs, and publish prefix sums before the receive-layout epilogue consumes them.
Dispatch producer	Shared token record and arrival parity; `elect_one_sync`, `tma_load_1d`, `mbarrier_wait_and_flip_phase`, `tma_store_1d`	Assemble one route-bearing record, prove its loads complete, and prove any send-buffer store complete before a dependent remote put.
Hybrid forwarder	Per-source old tail, packed finish/tail, and destination slot state; `ld_acquire_sys`, `gather`, `ffs`, `deduplicate`	Select ready or unfinished sources in round-robin order, consume only the chosen source's newly published interval, and emit at most one scale-up copy per distinct destination rank.
Combine reducer	Valid-route mask, source slots, partial sums, and pending stores; `ldg_with_gez_pred`, `accumulate`, `tma_store_fence`	Treat invalid slots as zero, preserve the declared reduction order and precision path, and make the reduced shared row visible to TMA.
Completion worker	TMA groups, QP state, signal phase, and timeout; `tma_store_wait`, `red_add_rel_sys`, `ld_acquire_sys`	Publish readiness only after the covered payload operations and fail-stop rather than recycle an unproved epoch.

Role-local mechanisms do not by themselves supply the end-to-end contract. The hybrid-combine kernel adds resource specialization to these semantic roles. For selected non-expanded layouts with four or eight channels per SM, scale-up warps lower their per-thread register ceiling to 40 through `warpgroup_reg_dealloc`, while forwarding/reduction warps raise theirs to $216 = 256 - 40$ through `warpgroup_reg_alloc`. The underlying `setmaxnreg` operations redistribute register capacity among the CTA's resident warpgroups so the state-heavy reduction role can retain more live indices, pointers, masks, and partial sums. This is a conditional resource policy, not a dispatch/combine semantic: changing it may change occupancy or speed, but it must not change route identity, reduction order, or completion state [19, 66].

26.10. Channel-Epoch State Machines

Direct and hybrid paths differ in intermediate ownership, but both need an explicit rule for when channel storage belongs to a new call. A channel epoch turns that reuse question into entry, progress, completion, and failure states that can be compared across the two modes.

Definition 26.10.1 (DeepEP channel epoch) *A DeepEP channel epoch is one use of a channel's payload slots, counters, tails, and finish state between the entry synchronization that makes those records reusable and the exit synchronization that proves their data have arrived. A timeout, trapped kernel, or communicator error leaves the epoch failed; the caller must not infer that its slots or cached handle are reusable.*

Protocol phase	Direct path	Hybrid path
Domain and team	One logical rank domain; local peer access can bypass the network, otherwise NCCL Gin uses a full-connection path.	A scale-out rank and a scale-up rank identify each process; rail-team puts cross nodes and LSA/NVLink stores forward within a node.
Entry ownership	A start barrier prevents dispatch or combine from overwriting remote buffers still owned by the previous call.	Parallel scale-out and scale-up barriers protect the receive buffers, channel tails, linked lists, and forwarding metadata of the new epoch.
Progress	Each channel writes or stages a final-destination record; there is no intermediate tail consumer.	A scale-out warp publishes a packed (finish, tail) value; a forwarding warp consumes only the new interval and advances per-scale-up linked-list state.
Publication	TMA stores complete before RDMA issue; the final barrier flushes the QPs that carried payloads before signaling peers.	`red_add_rel` release-publishes tails under Definition 19.7.1, and `ld_acquire_sys` observes them at system scope; a final scale-up barrier proves that forwarded records arrived.
Cleanup	Sender counters are reset only after the final barrier.	The forwarder writes an end marker, cleans consumed scale-out tails, and resets scale-up counters only after the final scale-up handoff.

Table 26.22.: Direct and hybrid DeepEP protocols preserve the same token semantics through different ownership and completion state.

Table 26.22 contrasts direct completion with the hybrid forwarder's intermediate publication.

Algorithm 17 first acquires a clean epoch; the entry barrier is a reuse gate, not token movement. It then follows one of two work paths. Direct mode publishes to the final owner, whereas hybrid mode publishes an intermediate interval that a separate scale-up owner must acquire and forward. Finally, only the mode-specific completion barrier permits cleanup. A timeout skips that proof, so the old slots and handle remain poisoned instead of being silently recycled.

Algorithm 17: Protocol-level DeepEP V2 channel epoch for direct and hybrid modes. The sketch exposes ownership, publication, completion, cleanup, and fail-stop state without reproducing the generated kernels.

Input: Mode $m \in \{\text{direct}, \text{hybrid}\}$; channel-epoch state; packed token records; participating teams; timeout policy
Output: Consumer-ready records and a reusable epoch, or a failed poisoned epoch

```
reject entry when a previous failed epoch has not been destroyed and rebuilt;
complete the mode-specific entry barrier before overwriting payload slots, counters, tails, or
  linked-list state;
pack payload, scales, origin ids, route slots, and weights, then prove dependent TMA stores
  complete;
if m = direct then
    move each record toward its final owner, flush every payload-carrying QP, and signal the
      final-rank team;
else
    move each remote record across its rail to the destination-node ingress;
    release-publish the new tail interval and finish state;
    have the scale-up forwarder acquire the interval, forward only its new records, and publish
      the end marker;
wait for the mode-specific final barrier that proves consumer-visible arrival;
if a poll, transport step, or barrier times out then
    return a failed epoch and require caller-side stream synchronization plus buffer or
      communicator rebuild before another call;
reset counters, consumed tails, and forwarding state only after final completion;
return ready records and reusable channel epoch
```

26.11. Publication Barriers, Reuse, and Reversible Recovery

The pinned communication implementation assigns each warp channel a QP with `get_qp_mode`. When there are enough QPs, an SM receives whole QPs with CTA sharing; when QPs are fewer than SMs, channels share them at grid or GPU scope. A notify warp can reserve QP 0. In the NCCL backend, the device communicator requests exclusive Gin contexts, a queue depth, traffic class, and signal slots. Direct mode requests `NCCL_GIN_CONNECTION_FULL`; hybrid mode requests `NCCL_GIN_CONNECTION_RAIL`. These choices explain why SM count, QP count, service level, physical scale-up size, and hybrid-mode flag are part of the executable signature rather than tuning trivia.

Flush, signal, and timeout order

The helper `gin_barrier_wo_local_sync` first flushes every QP that may contain payload stores when release semantics are required. It then uses QP 0 to increment signals on either `ncclTeamWorld` or `ncclTeamRail`. Each receiver increments a local shadow target and polls the corresponding signal with `ld_acquire_sys`; observing that target therefore follows the QP flush. For NVLink scale-up, `nvlink_barrier_wo_local_sync` uses a phased workspace signal and the same acquire-side rule. `gpu_barrier` additionally waits for TMA stores and synchronizes the cooperative grid before splitting a hybrid barrier across scale-up and scale-out SMs.

Every poll runs through `timeout_while`. On expiry the last check prints the tag, rank, channel, signal, and target state; the helper then waits briefly so participating threads can report and executes `ptx::trap`. This is fail-stop

diagnostics, not reconnect or retry. No later cleanup barrier has been proved after the trap. Operationally, the failed call must poison its channel epoch. Synchronize the stream, surface the CUDA error, destroy and rebuild the affected buffer or communicator, and recompute routing handles before accepting traffic. Replaying into the same workspace without that reset risks consuming stale tails, counters, or slot indices.

QP flush precedes signal, acquire observation gates reuse, and timeout poisons the channel epoch instead of recovering it.

The phased NVLink barrier above is a concrete reusable counter algorithm rather than an unspecified flag. Let q be the low two bits of the local barrier counter at entry and let R be the number of ranks. The phase chooses one of two symmetric signal banks; the upper bit chooses whether every rank adds or subtracts one from that bank.

Entry state q	Signal bank $q \,\&\, 1$	Direction	Each rank issues	Acquire target
0	0	fill	+1	R
1	1	fill	+1	R
2	0	drain	−1	0
3	1	drain	−1	0

Table 26.23.: Four-state reuse cycle of `nvlink_barrier_wo_local_sync`. Two banks separate adjacent barriers; alternating signed updates return each bank to zero without a separate clear racing the next use.

Table 26.23 exposes the state hidden by the phrase *phased workspace signal*. At each entry, a rank reads q, release-updates the chosen bank on every peer, advances its local counter, and acquire-polls its local bank for the displayed target. Successful prior use keeps every participant at the same entry state. That is a precondition, not something the signed counter repairs: if ranks enter with different phases, opposite-signed updates can invalidate the rendezvous.

Under the common-phase precondition, no rank exits that barrier use until all R updates have arrived. When bank 0 is selected again at $q = 2$, it still holds R from state 0, so the R negative updates return it to zero; bank 1 follows the same cycle one step later. The scheme is therefore a signed, two-bank analogue of a sense-reversing barrier. Its rendezvous fact covers only the declared team. Payload visibility still depends on the preceding TMA waits or QP flush and the release–acquire ordering already stated in the protocol table.

Example 26.11.1 (Four consecutive uses of the signed two-bank barrier) Let $R = 4$, let every rank enter the first use with the same low counter bits $q = 0$, and view the two signal banks at one receiver as $(B_0, B_1) = (0, 0)$. After all four ranks' updates arrive, the receiver observes four transitions:

1. At $q = 0$, four $+1$ updates to B_0 produce $(B_0, B_1) = (4, 0)$, then advance the low bits to 1.
2. At $q = 1$, four $+1$ updates to B_1 produce $(4, 4)$, then advance the low bits to 2.
3. At $q = 2$, four -1 updates to B_0 produce $(0, 4)$, then advance the low bits to 3.
4. At $q = 3$, four -1 updates to B_1 restore $(0, 0)$, then advance the low bits to 0.

The first two uses fill different banks, so adjacent barriers cannot satisfy one another. The next two uses drain those same banks to zero. No rank performs a separate clear, and after the fourth use the low counter bits and both signal banks are ready for the same cycle again. The full local counter remains monotone; only its low two bits repeat.

The hybrid dispatch trace makes the intermediate ownership concrete. A scale-out warp writes records into its channel's receive buffer and periodically publishes a tail; its final publication sets the finish bit even when the last interval is short. A forwarding warp round-robins over source ranks and waits for either tail > old tail or finish. It forwards only that interval to scale-up peers and records `token_metadata_at_forward` and `dst_buffer_slot_idx`. It then terminates the linked list with a negative source id and clears the consumed tail. Hybrid combine walks those records in reverse, reducing locally where permitted before issuing rail-team puts. The finish flag means "this producer will publish no more slots in this epoch"; it does not mean that every forwarded store is globally complete. That stronger fact comes from the final scale-up barrier.

Example 26.11.2 (Reversible hybrid channel epoch) Instantiate the pinned V2 NCCL Gin hybrid path as a scaled-down trace over the chapter's existing four-rank domain. Put ranks 0 and 1 on node A, ranks 2 and 3 on node B, and retain $\text{owner}(e) = \lfloor e/2 \rfloor$. For two rank-0 tokens, fix the top-2 records

$$R_0 = ((e_1, 0.35), (e_6, 0.65)), \qquad R_2 = ((e_4, 0.55), (e_5, 0.45)).$$

The weights do not change this two-token subtrace's deduplicated split vector; write it as $s^{(0)}_{\{0,2\}} = (1, 0, 1, 1)$. Token $g = 0$ produces a local record for expert 1 on rank 0 and a remote record for expert 6 on rank 3. Under the displayed two-rank-per-node ordering, hybrid dispatch moves the remote record from rank 0 to the same-rail ingress rank 2, then forwards it inside node B to rank 3. Token $g = 2$ is different: experts 4 and 5 share owner rank 2, so dispatch moves one token record with two valid route slots, not two activation copies.
Let $H = 256$, use FP8 dispatch with one four-byte scale per 128 coordinates, and write B_{meta} for origin, route, weight, and layout metadata. The data-plus-scale wire record has

$$B_{\text{dispatch}} = H + 4\lceil H/128 \rceil + B_{\text{meta}} = 264 + B_{\text{meta}} \text{ bytes}.$$

The hybrid path moves that record once across the scale-out domain and once across the destination scale-up domain. The two local experts selected by $g = 2$ still share one wire payload. At rank 2, combine the already introduced $g = 7$ row for expert 5 with $g = 2$. With alignment $A = 4$, the expert-major layout is

$$e_4 : [g_2, \bot, \bot, \bot], \qquad e_5 : [g_2, g_7, \bot, \bot],$$

so the handle preserves semantic counts $(1, 2)$, physical ends $(4, 8)$, and the route-slot-to-buffer inverse map. Padding reserves storage but creates neither a routed row nor an expert contribution.
For the remote $g = 0$ interval, a scale-out producer writes a channel slot and release-publishes a new tail. The rank-2 forwarder acquire-observes that tail and forwards only the newly published interval to rank 3. A final publication may set finish $= 1$, but expert compute must still wait for the final scale-up barrier, or for the returned `EventOverlap` dependency that represents it. After expert work, combine may receive route fragments in either physical order; the saved handle restores

$$z_0 = 0.35y_{0,0} + 0.65y_{0,1}, \qquad z_2 = 0.55y_{2,0} + 0.45y_{2,1}$$

to rank-0 rows 0 and 2.

Figure 26.10 places the semantic, physical, layout, and temporal views on one page. The route table determines which records exist. The topology determines where they move. The handle determines how expert rows and combine slots are interpreted. The channel epoch determines when any consumer may safely read them.

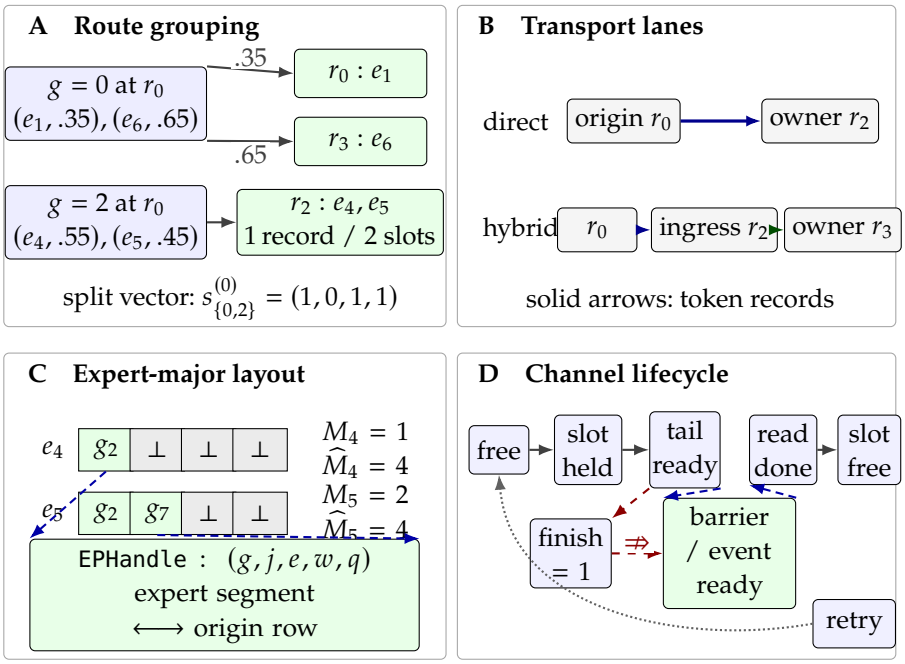

Figure 26.10.: Reversible V2 channel epoch. Route identity survives direct or hybrid transport, aligned expert placement, and the dashed inverse-map metadata. A slot advances from free to reusable only through publication and consumer completion; finish = 1 alone does not satisfy the final barrier/event dependency, and dotted recovery returns a timed-out epoch to a safe retry.

The figure supplies four independent failure checks. Sending $g = 2$ twice breaks the deduplicated split layout. Counting $\perp$ slots as tokens breaks the semantic expert counts. Starting expert GEMM from the finish bit rather than the final dependency permits a stale read. Losing g, j, or the inverse-map slot can return numerically plausible expert fragments to the wrong origin row or apply the wrong route weight.

These V2 state machines are the primary path at the pinned revision. Section 26.12 isolates the older NVSHMEM/IBGDA implementation so that legacy evolution does not blur the current transport contract.

26.12. Legacy NVSHMEM and IBGDA Transport

The legacy V1 path remains useful source-history and compatibility evidence. Even so, it uses different transport components, cache assumptions, and aggressive PTX choices. Keeping it separate prevents a V1 fallback or Hopper-specific observation from being mistaken for the V2 NCCL-Gin contract.

Legacy-track waypoint. Readers following the V2 route may continue at Section 26.13.

The archived V1 path remains operationally relevant. That is because integrations may still instantiate `deep_ep.Buffer`. It is not the V2 `ElasticBuffer` protocol. Table 26.24 keeps the two generations separate.

Transport	Mechanism	Compatibility and recovery constraint
V2 elastic EP	NCCL Gin, analytic SM/QP selection, direct or hybrid generated kernels, and trap-on-timeout barriers.	A timeout requires communicator and workspace recovery outside the kernel; V2 does not preserve the legacy zero-SM low-latency RDMA surface.
V1 normal path	NVSHMEM-backed asymmetric-domain forwarding with configurable SM use; receive counts can require a CPU wait on a GPU signal.	The CPU wait is not CUDA-graph compatible, and queue-based buffers add ordering and deadlock-sensitive state.
V1 low-latency path	GPU-side IBGDA helpers reserve WQE slots, translate symmetric addresses to rkeys, write RDMA or atomic WQEs, update the doorbell record, and ring the NIC doorbell. A receive hook can let network progress run without communication SMs.	This depends on NVSHMEM/IBGDA installation, fixed capacity and QP assumptions, explicit hook invocation, buffer restoration for graph replay, and legacy timeout/error handling. These V1 dependencies and failure semantics do not describe V2 Gin.

Table 26.24.: The released V1 and V2 transports solve related EP movement problems with different dependencies and recovery boundaries.

Example 26.12.1 (Identify V1 and V2) Runtime A constructs `deep_ep.Buffer`, invokes a legacy low-latency dispatch with `return_recv_hook=True`, and later calls the returned hook while an expert-compute batch runs. The hook drives the V1 NVSHMEM/IBGDA receive path; observing network progress without communication SMs is evidence about that legacy mechanism.
Runtime B constructs `ElasticBuffer`, calls `dispatch` with `async_with_compute_stream=True`, and receives `EventOverlap`. Its generated kernels use the V2 NCCL Gin path and a selected nonzero SM budget. Runtime B may use fewer SMs than V1, but Runtime A's zero-SM hook observation cannot be transferred to it. The class, backend, completion object, and effective SM/QP counts identify which claim is valid [19].

The legacy V1 mechanism separates queue-entry construction, low-latency network operations, and deployment constraints. These components define the transport behavior. They do not, however, show that a downstream framework selected the V1 path [19].

Legacy V1 transport map

- **WQE and doorbell construction.** `ibgda_reserve_wqe_slots`, `ibgda_submit_requests`, and `ibgda_post_send` connect queue reservation to WQE publication and doorbell progress.
- **Low-latency network operations.** `internode_ll::dispatch` and `internode_ll::combine` issue puts and atomics and mask timed-out receive ranks.
- **Deployment constraints.** `deep_ep.Buffer`, `return_recv_hook`, and `DISABLE_AGGRESSIVE_PTX_INSTRS` expose the legacy buffer, explicit receive progress, and conservative cache-operation fallback.

A V1 run record must name the buffer class, backend, NVSHMEM mode, QP count, hook timing, graph-replay reset, and conservative-PTX fallback.

The notify, copy, and reduction obligations below are generation-neutral semantic roles. Table 26.11 binds them to the recorded V2 runtime classes; here they are restated to identify the corresponding V1 work. The V1 NVSHMEM/IBGDA kernels and the V2 NCCL Gin generated families are distinct implementations.

The notify phase is a correctness phase before it is a performance phase. The kernel counts destination ranks after deduplicating route slots that go to the same rank. It also counts destination experts, reduces those counts across SMs, and writes prefix sums for later buffer slots. If those counts are wrong, the copy phase can be perfectly fast and still write a token record into the wrong receive slot. Cached dispatch skips or reuses parts of this work only when the slot-index record is already valid for the current routing layout.

The copy phase binds bytes to identity. A dispatch warp stages the hidden payload in shared memory and stages scale-factor packs when FP8-style dispatch uses them. It writes top-k indices and weights, and it writes the origin global token id before remote movement. The send operation can be a peer-accessible NVLink write or a send-buffer path followed by a remote put. The semantic obligation is nonetheless identical: payload, scale metadata, route slots, and origin id must land together.

Table 26.25 follows the records created by the DeepEP implementation, the buffers they select, the topology branch they enter, and the later check that proves the movement was semantically correct.

Table 26.25.: Inside one DeepEP dispatch/combine call. Each phase is an implementation-level record behind the higher-level communication protocol.

Phase	Concrete internal record	Correctness or timing obligation
Count and split	Per-destination token counts, expert counts, and prefix sums derived from top-k routes	Receive-buffer boundaries and expert segments are data-dependent, not fixed by tensor shape.
Pack send buffers	Hidden rows, optional scale packs, top-k masks, route weights, and origin ids are placed in dispatch slots	Combine must later see the same payload identity that expert compute used.
Choose transport branch	Rank layout selects local peer writes, staged send buffers, or hybrid scale-up/scale-out movement	A bandwidth claim must state whether the row used NVLink-like or RDMA-like paths.
Record overlap event	Dispatch or combine returns an event or handle consumed by the compute stream	Overlap hides time only until expert compute needs the received rows.
Reuse or reject handle	Cached slot indices are reused only when route layout, shape, dtype, and topology assumptions still match	A stale handle is a correctness bug even when payload dimensions are unchanged.
Combine reduction	Expert outputs are gathered by origin id, route slot, and weight, with optional local reductions before remote writes	The final tensor must restore original token order and route-weighted accumulation.

The elastic-buffer API exposes the handle and event surfaces. The reference utilities define the origin-id and combine semantics. The CUDA dispatch and combine paths make counts, slot indices, workspace records, and topology

branches part of the protocol state [19]. Aggregate bandwidth is meaningful only after all six phases preserve that state.

Combine reverses the direction but not the logic. It reads origin metadata to find the origin token and route slot. It then chooses direct peer writes or staged remote writes by topology, and it optionally reduces multiple local fragments before sending. Expanded-layout and multiple-reduction modes change how many fragments are locally merged before the final token buffer is written. That is why a combine correctness test needs both byte-arrival checks and weighted route reconstruction checks.

The topology and JIT record remains necessary when inspecting the legacy path: record participating ranks, scale-up and scale-out links, source revision, generated artifact, cache state, and timing boundary. Its compiler/cache lifecycle and the V2 NCCL Gin transport must not be conflated. The hybrid-mode fields refer to Definition 26.8.2. The reason is that a hierarchical run changes the interpretation of traffic and reductions. The pinned CUDA-kernel protocol defines QP allocation, team selection, flush and signal order, direct and hybrid ownership, timeout trapping, and the legacy IBGDA boundary. Its public interfaces do not expose switch implementation details, internal NCCL Gin behavior beyond the device API used by DeepEP, or framework-specific recovery orchestration. An interpretable deployment result therefore requires both the observed kernel state and the external reset or failover policy.

26.13. Load Skew, Hot Experts, and Communication Bubbles

A hot expert becomes a communication problem only after route counts map to owner ranks and wait time.

Sparse routing can make the average compute budget look attractive while creating uneven communication. If many tokens choose experts on the same rank, that rank receives more rows, runs more expert work, and may delay combine for other ranks. DeepEP's test utilities include generators for unbalanced score distributions, which makes skew a first-class test condition rather than an afterthought [19]. The vocabulary should separate two effects. Hot experts create extra expert work, and hot ranks create extra communication pressure.

The released V2 dispatch surface also accepts `cumulative_local_expert_recv_stats`, a device integer tensor with one counter per local expert. Dispatch atomically adds the semantic received-route count for each local expert. The supplied tensor is cumulative. The caller owns initialization, reset, window boundaries, and aggregation. Padding introduced by expert alignment is not load. The V2 test checks the counter against reference route counts rather than aligned buffer lengths [19].

Definition 26.13.1 (Receive-statistics window) *A receive-statistics window is the interval between an explicit counter reset and snapshot. Its record names the layer, physical-to-logical expert map, participating ranks, number of dispatches, and whether each dispatch computed a fresh layout or reused a cached handle.*

The cached-handle distinction prevents a subtle monitoring error. Supplying a handle requires fresh top-k ids and weights to be absent, so the dispatch reuses the

handle's expert layout. More strongly, the released V2 kernel rejects a cumulative expert-statistics tensor in cached mode. The public counter is therefore available only on fresh-layout dispatch. A runtime that mostly uses cached dispatch must schedule separate fresh-layout sampling windows or collect an equivalent counter elsewhere. It cannot silently treat the cached path as observed load.

Table 26.26.: From public DeepEP receive counters to an EPLB-ready load matrix.

Counter step	State preserved	Downstream use
Reset	zeroed local-expert counters and a placement-map version	Establish a new, auditable observation window.
Fresh dispatch	current top-k ids, local expert counts, and dispatch event	Add actual received route assignments for this call.
Cached dispatch	handle identity and the fact that no new top-k layout was accepted	Cumulative expert counting is unsupported on this path; exclude it from the public-counter window or use another declared observation source.
Snapshot and aggregate	per-rank counters mapped from physical copies back to logical experts	Build a per-layer logical-expert load matrix for a declared time window.
Smooth and plan	aggregation rule, moving-average state, and profile revision	Supply an explicit load matrix to EPLB; EPLB itself does not collect or smooth these counters.

Table 26.26 is the complete public-counter handoff: fresh-layout dispatch supplies the observation, while aggregation, smoothing, and placement planning remain explicit downstream steps.

Definition 26.13.2 (Hot expert) *For a fixed batch and routing policy, a hot expert is an expert whose routed row count is substantially larger than the mean expert row count. A simple hotness ratio is*

$$\rho_{\text{expert}} = \frac{\max_e M_e}{\frac{1}{E}\sum_e M_e}.$$

Use the first eight tokens in the running example. Their selected experts are $(1,6)$, $(2,7)$, $(4,5)$, $(0,3)$, $(6,7)$, $(1,2)$, $(4,0)$, and $(3,5)$. Counting route slots by expert gives

$$(M_0, \ldots, M_7) = (2,2,2,2,2,2,2,2).$$

The mean is 2 and the maximum is 2, so the hot-expert ratio from Definition 26.13.2 is $\rho_{\text{expert}} = 1$. This balanced toy table is useful because it shows that cross-rank communication can exist even when expert counts are perfectly balanced.

Now change the routes for four additional tokens so that each selects experts 6 and 7. The added route slots increase M_6 and M_7 by four each. Across twelve tokens, the total route slots are 24, the mean over eight experts is 3, and the maximum expert count is 6. The hotness ratio becomes $6/3 = 2$. Rank 3 owns experts 6 and 7. It therefore receives a disproportionate share of the rows and must send back a disproportionate share of the combine outputs.

Definition 26.13.3 (Communication bubble) *A communication bubble is time in which a rank, stream, or downstream compute kernel is ready to proceed but waits for required communication or reduction data. It is distinct from pure compute imbalance, although routing skew can cause both effects at once.*

The distinction matters in diagnosis. If rank 3 receives twice as many rows and its expert kernels run twice as long, the bottleneck is at least partly compute imbalance. If rank 0's expert kernels are short but its combine path waits for rank 3's remote outputs, the visible idle interval on rank 0 is a communication or synchronization bubble. A token histogram can suggest the risk, but it cannot prove the bubble. A bubble claim needs a timeline, trace, or timing record that shows waiting.

Scenario	Counts	Hotness ratio
Balanced first eight tokens	max 2, mean 2	1.0
Four extra tokens to experts 6 and 7	max 6, mean 3	2.0

Table 26.27.: A small skew calculation. The second row predicts pressure on rank 3, but a timeline is still needed before claiming a measured bubble.

Table 26.27 gives only the histogram side of the diagnosis. A measured bubble still requires a trace or timing record showing that some rank waited for communication or reduction data. Figure 26.11 keeps the histogram warning separate from the wait evidence.

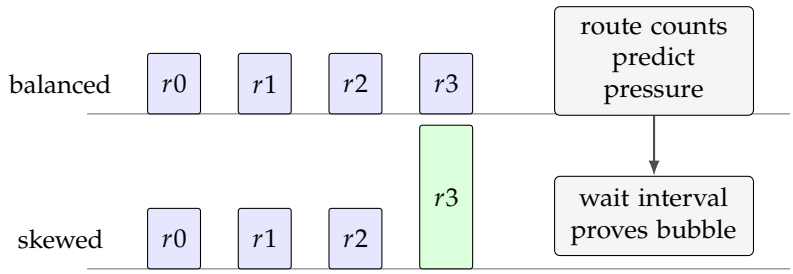

Figure 26.11.: A hot-rank histogram is a warning signal, not a bubble proof. The bubble claim requires a later wait interval showing that another stream or rank was ready but blocked on communication or reduction data.

Trace field	What the trace must show	Claim it supports
Route histogram	rank 3 owns the hot experts and receives extra rows	skew pressure, not yet a bubble
Communication interval	transfer or reduction for rank 3 extends past local work on rank 0	possible wait source
Wait interval	rank 0's combine stream is ready but blocked on rank 3 output	communication bubble

Table 26.28.: Minimal trace structure for upgrading a skew suspicion into a communication-bubble claim.

Table 26.28 prevents a common overclaim. A hotness ratio can identify the likely owner rank, but only a wait interval in the trace turns that suspicion into a bubble diagnosis.

Chapter 28 will later discuss placement and load balancing. The local requirement here is to derive the communication consequence of the route distribution. For each batch, the diagnostic questions are: which experts are hot, which ranks own them, how many token records cross each scale-up or scale-out boundary, and where would a combine call wait if the slowest owner rank lags? DeepEP

provides the communication mechanism; the route histogram and topology explain where that mechanism is likely to be stressed.

26.14. Dispatch Performance by Route Shape

DeepEP timing is meaningful only after the communication workload is clear. Dispatch sends token activations together with routing metadata; combine returns expert outputs to their original token positions and restores route weights. Token count T, hidden width H, top-k, expert count E, and dtype therefore change both the traffic and the useful work. DeepEP's test harness uses different byte estimates for dispatch, cached or expanded dispatch, combine, and reduced combine [19]. A bandwidth comparison is valid only when its numerator describes the operation whose latency supplies the denominator.

The dtype rule is especially important for FP8. Definition 26.3.2 requires a dispatch payload to include the low-precision token data and the scale metadata needed to reconstruct it. For example, a token with $H = 256$ has 256 FP8 data bytes. With one four-byte scale per 128 coordinates, two scales raise that to 264 bytes before route ids, weights, origin metadata, or alignment. Counting only the data array would make FP8 traffic appear smaller than the representation that the receiver actually needs.

Table 26.29 gives the release-level calibration from the pinned V2 README. All rows use a V3-shaped configuration with 8K tokens per batch, hidden width 7168, top-$k = 8$, FP8 dispatch, and BF16 combine. The reported bandwidth is *logical*: for a hybrid topology such as EP 8×2, it includes local-rank traffic; this effective-bandwidth metric differs from RDMA link rate. These rows therefore calibrate the released transport families without serving as baselines for the book's different H200 route shapes and byte rules.

Table 26.29.: REPORTED DeepEP V2 logical bottleneck bandwidth from the README at commit `60d44037`. The EP 8 rows are its peak-performance and minimum-SM settings. The topology, transport, SM budget, workload, and logical-byte convention are part of each result; the rows were not remeasured by the book.

Arch.	NIC	Topology	Dispatch	Combine	SM budget
SM90	CX7	EP 8×2	90 GB/s RDMA	81 GB/s RDMA	12
SM90	CX7	EP 8×4	61 GB/s RDMA	61 GB/s RDMA	6
SM100	CX7	EP 8×2	90 GB/s RDMA	91 GB/s RDMA	12
SM100	—	EP 8	726 GB/s NVLink	740 GB/s NVLink	64 (peak)
SM100	—	EP 8	643 GB/s NVLink	675 GB/s NVLink	24 (minimum)

Correctness and throughput answer different questions. Dispatch must preserve destination and origin metadata; combine must restore weighted results to the right token; cached dispatch must match the current routing generation. Timing is admitted only after those gates pass. Here host channel adapter (HCA) names the physical RDMA device, not V4 Heavily Compressed Attention.

The successful physical-HCA rows form a route-shape sweep rather than one portable bandwidth number. Figure 26.12 exposes the observed rank ranges for dispatch and combine while keeping latency and scale-out bandwidth in separate facets. Failed bonded-HCA targets remain only in Table 26.30; the chart omits them from the numerical series.

Route and topology	Retained observation	Interpretation
Single-node small shape	Two H200 captures each pass all 144 mode combinations for $T = 256, H = 256, k = 2, E = 4$; dispatch, cached dispatch, combine, and reduced-combine ranges repeat closely	Scale-up correctness and harness bandwidth for the tested layouts, not serving performance
Single-node V3-shaped and skewed routes	Balanced first-mode dispatch/combine is about 572–609 GB/s; increasing skew preserves a fast heavy rank while the light-rank rate falls with its byte share	Per-rank bytes and waiting time are needed; one aggregate bandwidth hides imbalance
Two-node small shape	Three captures pass the 144-mode matrix, with first-mode dispatch-family latency roughly 152–250 μs and combine-family latency roughly 196–228 μs	Replicated small-shape scale-out path under its NIC boundary, not a scaling curve
Bonded-HCA targets	Small controls pass, while six larger synchronized targets time out or fail before an admissible performance row	Reproduced transport failure, not zero throughput
One physical HCA	All bracketed and both DeepSeek-shaped targets pass on two nodes and four H200s; the precision-preserving rerun retains per-rank ranges	Exact-allocation correctness and rank-range evidence; the fastest rank describes one participant rather than aggregate fabric throughput
Balanced versus 4:1 trace	Skew raises dispatch and combine barrier residence while hybrid kernels remain observable	Operation-internal wait evidence, not a compute-aligned communication-bubble proof
DeepEP-enabled serving row	A pinned TP2/EP2 compatibility path completes 16/16 measured requests at 1.35054 accepted requests/s and 39.1656 output tokens/s	No matched baseline, current-head, low-latency, quality, capacity, or production claim

Table 26.30.: Consolidated local DeepEP findings. Every measured row retains its route shape, byte rule, topology, NIC selection, and correctness gate; the NVIDIA execution-evidence chapter carries exact campaign provenance.

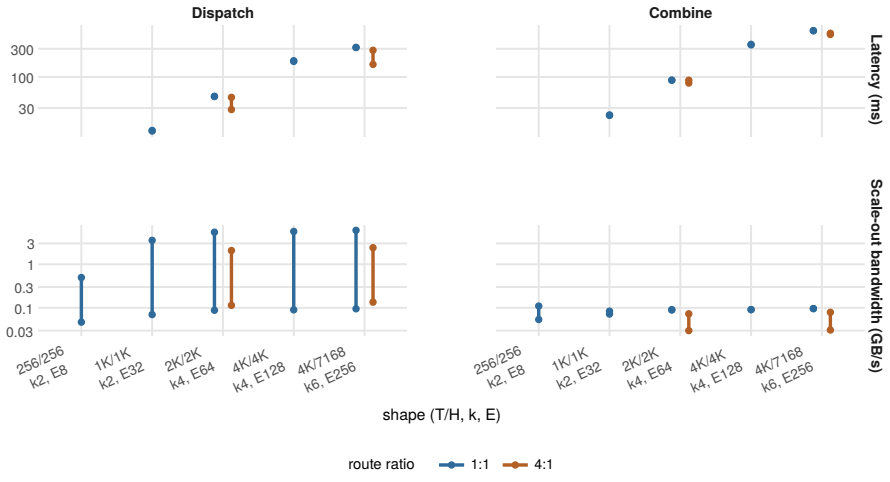

Figure 26.12.: Reviewed two-node ordinary-operation ranges on one physical HCA. The upper facets show latency from the repaired physical-HCA packet; each range spans four rank records in one successful scenario, and each scenario produced 20 metric records. The lower facets show per-rank scale-out bandwidth from the precision-preserving packet; scale-out excludes local traffic. Vertical scales are logarithmic and differ by row. These are test-harness ranges, not aggregate fabric or serving throughput.

The skew result supplies the mechanism-level lesson. A heavy rank can continue moving a large payload efficiently while a light rank reports a much smaller rate and waits at a barrier. Interpretable performance therefore needs per-rank bytes, operation latency, and waiting time in addition to aggregate throughput.

The closure packet and the adjacent EPLB and DualPipe observations are compared in Chapter 35. It preserves the causal boundary: independent work may intersect dispatch, but expert work still waits for dispatch and combine still waits for the expert result. Selected overlap consequently does not imply speedup.

Topology also remains part of the result. Isolated rail subnets limit connectivity even when every physical HCA is selected, and a successful one-HCA allocation covers only that HCA configuration. Likewise, an application route-identity guard can reject aliased cached-handle mutation without proving that native DeepEP detects every stale handle. DeepEP performance is a property of the routed workload, transport path, and topology rather than a portable bandwidth constant.

26.15. Handoffs and Failures

DeepEP is the communication layer inside a larger MoE system. Before DeepEP, the router produces top-k expert ids and route weights. After dispatch, local expert computation consumes the received rows, often through grouped GEMM or a fused MoE path. After combine, the runtime continues with the rest of the transformer layer, scheduler, and request state. DeepGEMM explains the grouped compute interface, while EPLB will later explain expert placement and load balancing as an independent deployment problem [81, 25]. The local job is to make the handoff between routing, communication, expert compute, and combine precise.

That modular boundary is also an oracle boundary. The Chapter 27 compares the complete DeepEP-style dispatch–expert-compute–combine composition with a fused SM100 endpoint. Mega MoE does not call DeepEP internally; the two paths must instead preserve the same route, source-token, weight, and output obligations.

Definition 26.15.1 (Expert-parallel handoff record) *An expert-parallel handoff record is the conceptual state passed between the router, communication layer, expert-compute layer, and combine layer. It contains origin token identity, selected experts, route weights, owner ranks, destination slots, expert segment counts, dtype and scale metadata, and the reduction policy.*

In the running example, token 2's handoff state fixes origin id $g = 2$, selected experts $(4, 5)$, owner rank 2 for both route slots, and route weights. It also states that rank 2 receives one token row with two valid local slots. The later combine rule returns one weighted row to rank 0 local row 2. The grouped GEMM layer may only need the rows for experts 4 and 5, while the combine layer still needs the origin and slot information. A clean integration preserves information needed by later stages even when the next kernel does not use it. Figure 26.13 marks the transitions where that state can be lost.

Table 26.31 distinguishes fields that make a runtime call correct from the additional provenance needed to interpret a benchmark result.

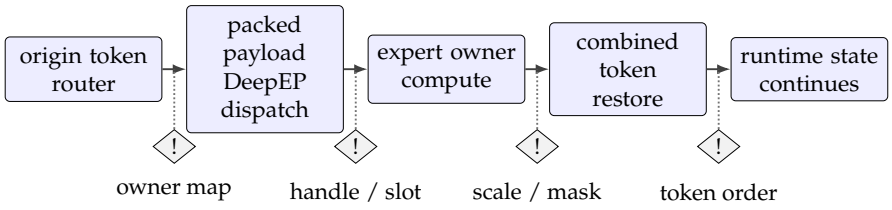

Figure 26.13.: Expert-parallel failures sit beneath the exact handoff they corrupt. Solid arrows carry token payloads through the same origin–payload–owner–restore positions used by the dispatch/combine figures; dotted markers identify metadata failures that can preserve tensor shape while changing meaning.

Table 26.31.: Runtime handoff fields support correct execution; benchmark fields support reproducible interpretation.

Obligation	Runtime handoff state	Run description
Owner and topology	owner ranks and rank group used by this dispatch	physical rank layout, scale-up/scale-out path, hybrid flag
Counts and byte rule	destination slots, expert counts, dtype, and scale metadata needed by kernels	activation bytes, metadata bytes, padding, local-traffic policy, FP8 scales
Cached handle	layout assumptions under which reuse is valid	valid-reuse case and stale-handle rejection or failure test
Correctness gates	origin ids, route slots, masks, and reduction policy	reference dispatch, reference combine, FP8 reconstruction, masked-slot checks
Revision	source revision belongs to the run description	DeepEP commit, build/JIT state, CUDA, NCCL, driver, and run date

The runtime handoff state lets the next layer run correctly. The run description makes the result interpretable and reproducible.

Table 26.32 turns the handoff state into concrete checks. The checks are small because the dangerous mistakes are often metadata mistakes that preserve tensor shape.

JIT lifecycle failures. Fully JIT execution adds failures before the communication protocol begins. Table 26.33 maps behavior implemented by the pinned DeepEP cache/compiler path to checks that the enclosing distributed runtime must impose.

These failure modes are deliberately mundane. Expert-parallel communication may return successfully while rows are assigned to the wrong origin token. A cached handle can save time while silently preserving obsolete destinations. A benchmark can include local traffic in its byte numerator while its interpretation assumes only cross-node RDMA traffic. A correct implementation review must test these conditions as carefully as it checks throughput.

Pinned integration check. SGLang provides a snapshot-pinned source example of an open serving runtime carrying the DeepEP handoff state across expert compute. At the pinned SGLang source snapshot, its GPU path imports `Buffer` and `Config` from the `deep_ep` package rather than V2 `ElasticBuffer`. The normal

Table 26.32.: Failure modes that cross the communication interface. Each one threatens correctness as well as performance.

Failure mode	What goes wrong	Minimal check
Wrong owner map	rows sent to ranks that do not own the selected experts	compare destination ranks with owner(e)
Stale cached handle	payload uses old route layout after routing changes	mutate one top-k row and require recomputation or rejection
Wrong $M_{\max}$	origin token ids decode to wrong origin rows	round-trip $g = sM_{\max} + i$ for every origin rank
Missing scale metadata	FP8 bytes reconstruct the wrong represented values	include data and scales in byte and correctness checks
Padding as data	aligned rows produce semantic outputs	assert padded rows do not contribute to combine
Masked slot mishandling	invalid route slots carry stale values	compare masked reference with zero contribution
Topology mismatch	benchmark interpreted under wrong network assumptions	state rank layout, NVLink/RDMA path, and hybrid flag

Table 26.33.: DeepEP JIT failures. Native cache construction protects artifact publication; the distributed runtime must still enforce participant identity and honest timing scope.

Failure mode	Native behavior	Required integration check
Unavailable or unsupported compiler	JIT initialization and NVCC version or compile checks fail before a communication kernel launches.	Retain NVCC path and version, target architecture, effective command, and failure output.
Incomplete cache directory	`KernelRuntime` rejects a digest directory missing either the `generatedCUDAsource` or `kernel.cubin`.	Identify and quarantine only the failing digest; rerun from an empty run-scoped cache.
Same-identity rank race	Each contender compiles in temporary storage; atomic rename publishes one complete directory, and a losing rank loads the winner.	Stress concurrent first use and require one valid published identity plus successful loads on every participant.
Cross-rank identity drift	Different source, compiler signatures, or flags produce different digest paths; hashing prevents accidental cache aliasing but requires the runtime to enforce job-wide agreement.	Exchange revision and JIT identities before dispatch and fail the integration when they differ.
Cold work inside a kernel timer	The call can include generation, compilation, publication, symbol discovery, and loading before communication executes.	Report empty-cache first-call and warm repeated-call timings under the same identity.

path retains received hidden states, optional scale metadata, top-k ids and weights, per-expert receive counts, and the handle produced by dispatch for the matching combine. The low-latency path additionally retains `masked_m` and `expected_m`; its combine likewise consumes the matching handle [72]. This source inspection corroborates the handoff obligations. Its scope is the legacy `Buffer` integration; V2 interface use and executed correctness or performance remain open [19].

For token 2 in the running example, a useful integration replay can focus on the communication handoff. Retain $g = 2$, route $(4, 5)$, the rank-2 destination slots, route weights, required scales or masks, execution mode, and rank topology. Keep the dispatch-produced handle paired with the combine that must return the weighted result to rank 0 local row 2. A run that selects another communication backend tests that backend rather than DeepEP. If DeepEP is selected but the `deep_ep` package is unavailable, the SGLang dispatcher raises `ImportError`; the inspected path supplies no equivalent fallback.

Broader methodology belongs in Chapter 34. The DeepEP-local tests remain: compare dispatch with a simple all-to-all-v reference; compare combine with ordered route-slot accumulation; reject or recompute a V2 cached handle after its route layout changes; and account for FP8 scale metadata. Those checks are the executable form of this chapter's definitions.

The interface with expert placement is similar. If one rank owns too many hot experts, DeepEP can still execute the requested communication correctly while the layer remains slow or bubble-prone. Placement belongs to Chapter 28; this chapter supplies the token movement, metadata, and measurement obligations at that interface.

26.16. Summary

DeepEP turns sparse expert choices into a reversible communication protocol. In the four-rank example, dispatch carried origin row, route slot, expert, weight, and scale metadata to the owning rank. Combine used that identity to restore weighted contributions to the original token order. Counts, padding, placement generation, handle identity, and completion state are therefore part of correctness rather than optional transport detail.

Performance comes from composition. Destination deduplication exploits MoE structure, FP8 carries scales with each row, and expert-major layouts feed grouped compute. Generated kernels use warp votes for sender election, transaction barriers for shared-record reuse, phased signals for completion, and conditional register redistribution for state-heavy warps. Topology selects direct or hybrid movement, while cached handles remain valid only while their routing and placement assumptions match.

V2 uses NCCL Gin, `ElasticBuffer`, and a selected communication-SM budget. Direct mode targets final ranks; hybrid mode crosses a scale-out rail and forwards inside the destination scale-up domain. Its launch chain is `DispatchRuntime`→ `DispatchCopyEpilogueRuntime`, then, after expert compute, `CombineRuntime`→ `CombineReduceEpilogueRuntime`. Movement kernels expose readiness through `EventOverlap`; `BarrierRuntime` is a separate buffer fence.

The archived V1 NVSHMEM/IBGDA hook is a legacy mechanism. A timeout poisons the channel epoch, so recovery replaces the communicator, workspace, and stale handles before replay.

The production context in Chapter 20 explains why the library separates high-throughput prefilling from low-latency decoding: DeepSeek reports separate prefill/decode units and overlap schedules. The overview does not identify the public DeepEP revision as the hosted binary, so the H200 records remain component evidence [23].

The H200 records establish selected transport paths, correctness gates, and small-shape behavior under their topology, not portable throughput, production capacity, low-latency serving, or current-head behavior. DeepGEMM begins after rows are local. Chapter 27 instead fuses movement, both expert linears, activation, and combine under a separate symmetric-memory implementation; it does not turn DeepEP into an internal kernel stage. EPLB changes placement rather than either transport or fused execution.

26.17. Exercises

1. Core: Let owner$(e) = \lfloor e/2 \rfloor$ and $M_{\max} = 4$. A token at local index 3 on origin rank 2 routes to experts 0 and 7, but combine returns global token id 7. Compute the correct id, identify the faulty field, and write a shape-insensitive assertion that catches the error.
2. Core: Define dispatch and combine formally for a top-2 MoE layer. State one invariant that dispatch and combine must each preserve. For four routed tokens, tabulate origin id, expert, destination rank, route slot and mask, local row, combine destination, and weight. Show which row fails after a one-position mask shift.
3. Core: A rank receives expert row counts $(3, 5)$ with alignment 4. The output changes when the four padding rows contain sentinels instead of zeros. Compute both aligned segment boundaries, identify the violated invariant, and specify the failing mask check.
4. Applied: Token g routes to experts 2 and 7 with weights 0.65 and 0.35. For expert outputs $a = (2, 0)$ and $b = (0, 4)$, a trace reports combined output $(0.7, 2.6)$. Compute the correct output, diagnose the route-slot fault that produces the trace, and state a minimal reference assertion that distinguishes the two orders.
5. Core: Contrast fresh-layout and cached-handle dispatch. State the handle-validity condition, why cached calls omit fresh top-k ids and cannot accept the cumulative receive-statistics tensor, and how you would obtain an EPLB load profile.
6. Applied: Explain why an FP8 dispatch payload must include scale metadata. Give a small hidden-width example showing how byte accounting changes when scale factors are included.
7. Applied: Construct a skewed routing example over four ranks where one rank receives twice the mean number of token records. Identify whether the resulting bottleneck is compute imbalance, communication imbalance, or both.
8. Core: Order and name the recorded V2 runtime families from route records through restored origin rows, placing expert GEMM at the correct boundary. Distinguish the embedded movement-kernel barriers and returned overlap event from the explicit `BarrierRuntime` API.
9. Applied: Calls A–C use the same DeepEP revision and public kernel name. A starts with empty caches under fixed direct topology, hidden width, top-k, expert count, and SM/QP choices. B changes payload values and addresses; C switches B to hybrid topology. Classify identity reuse, name the identity fields, separate cold and warm timing, and state the atomic-publication condition for concurrent first use.
10. Research: Design a benchmark record for comparing dispatch and combine bandwidth on two topologies. List the model settings, topology fields, software versions, timing protocol, local-traffic policy, SM/QP counts, generated kernel parameters, and JIT identity and correctness checks that must be recorded. Also compare direct/hybrid QP, team, publication, barrier, and timeout-recovery records; keep V1 IBGDA separate.
11. Applied: Compare two dispatch/combine bandwidth traces with different route skew. Name which fields distinguish compute imbalance from communication imbalance.
12. Applied: A dispatch returns `EventOverlap` to two compute streams inside a CUDA-graph-aware runtime. Write the wait and release order, then explain the failure caused by releasing the event handle after only the first stream waits. Include one assertion for dispatch-backward/combine duality or route-weight-gradient order.