INDEPENDENT TECHNICAL REPORT

System Substrate Graph

A Cross-Scale Representation of AI Computing

Xiakai Pan, Chaohan Pei

November 2026 - Author’s web edition

This edition adds an author-identification cover to the historical September 2026 technical manuscript, which follows unchanged.

The work is an unreviewed technical report; no conference acceptance or independent reproduction is claimed.

No institutional affiliation is stated for either author in this edition.

System Substrate Graph: One Cross-Scale Representation of Workload, System, and Schedule for AI Computing

Anonymous Author(s)

Abstract

Modern AI systems make execution decisions across multiple layers: workload transformations determine what is computed, hardware and runtime configurations determine what resources and paths are available, and scheduling determines how the workload is mapped onto those resources. Existing compilers, communication libraries, and distributed runtimes expose these decisions through separate abstractions, making it difficult to preserve their relationships across execution scales or to distinguish a legal representation from an executable and numerically correct realization.

We present the *System Substrate Graph* (SSG), a cross-scale execution representation that models Workload and System independently over a shared Compute, Memory, and Communication (CMC) vocabulary and binds them through an explicit Schedule. SSG separates graph transformation from strategy mapping, represents spatial and temporal mapping as Parallel and Pipeline schedules, and materializes the resulting execution as Compute, Remap, and Sync (CRS). A Build–Traverse–Translate prototype carries the same representation from operator-level kernel generation to framework-level scheduling and selected model-path integration, while retaining the evidence state and provenance of each realization.

We evaluate SSG on H20 and an eight-GPU A800 node across operator, framework, and model-path scopes. Operator enumeration exposes structured gaps between what is represented, what a backend realizes, and what is numerically correct: one frozen GEMM shape admits 2,892 candidates, of which 288 are correct. Framework experiments show that the preferred parallelization changes with offered load, model size, and interconnect; model-path integration shows both successful substitution and substantial integration cost. These results establish neither an automatic optimizer nor an end-to-end speedup. They show that one Workload–System–Schedule object can preserve execution decisions across scopes and make legality, realization, correctness, and measured outcomes explicit.

CCS Concepts: • **Software and its engineering** → **Compilers**; • **Computer systems organization** → *Parallel architectures*; • **Computing methodologies** → Machine learning.

Keywords: system substrate graph, execution representation, workload–system–schedule, model-path integration, communication scheduling

1 Introduction

1.1 The MNK Engineering Problem

An AI-computing system must combine three constituents: **System configurations**—hardware resources, memory, topology, Scale-Up/Scale-Out organization, and the low-level substrate software that exposes and constrains their execution capabilities; **Workload forms**—DNN and language-model graphs, dense or sparse, and their training, inference, and RL phases; and **Schedule choices**—spatial partitioning and placement, temporal staging and overlap, and the movement and synchronization they induce.

This authoring surface is written $\mathcal{M} \times \mathcal{N} \times \mathcal{K}$: $\mathcal{M}$ is System configurations, $\mathcal{N}$ is Workload forms, and $\mathcal{K}(m, n)$ the admissible Schedule choices under a given pair. The calligraphic symbols denote these sets throughout and are unrelated to the italic M, N, and K used for GEMM dimensions in Section 5. Figure 1 contrasts the two ways this surface can be authored.

Existing stacks divide ownership of this surface. Operator compilers optimize local regions under a fixed distributed context; communication libraries optimize movement after a payload and interface are fixed; frameworks expose parallel degrees and phase controls around workload blocks already chosen. Each layer is effective within its own scope, but the same layout, movement, placement, or synchronization decision is encoded repeatedly in objects that cannot be checked against one another. The joint Workload–System–Schedule decision is therefore not itself a persistent execution object, and no interface exposes a shared object for checking its legality, realization, and provenance consistently. It is consequently difficult to determine whether a candidate is representable, legal, realized, correct, or attainable, and where a failure occurs.

Scale is not a fourth axis beside Workload, System, and Schedule; it is a built-in property of the first two. Crossing a scale boundary usually changes the engineering rather than only the magnitude: one logical decision is realized with registers and shared memory inside a core, with global memory and device-wide barriers inside a device, and with a transport and a collective library across devices. The intra-device

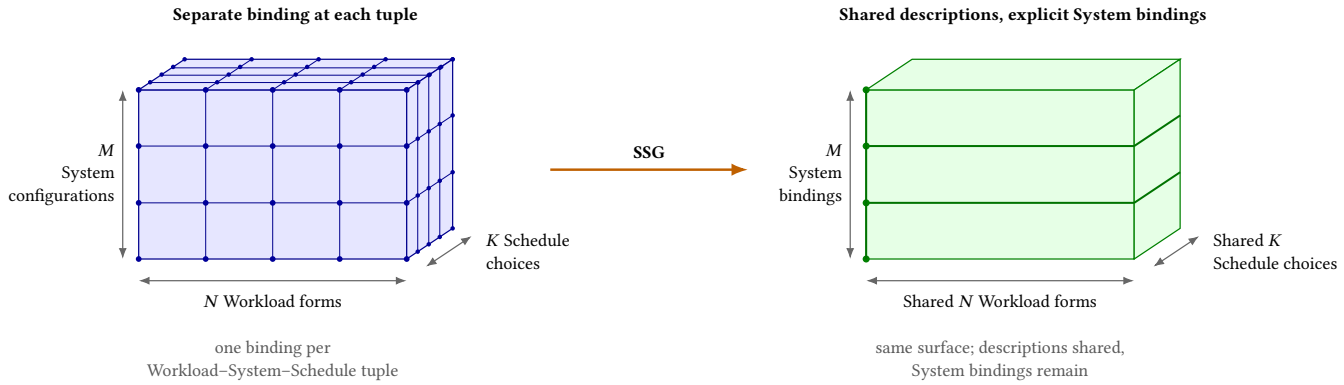


Figure 1. The Workload–System–Schedule authoring surface. A mark is a point at which a binding must be authored. Left: every Workload–System–Schedule tuple carries its own binding. Right: Workload and Schedule descriptions are shared, and marks remain only along M (System-specific bindings). The volume is unchanged; only where bindings are written differs. Schematic only; no quantitative claim.

and inter-device forms are consequently owned by different tools. Keeping Scale inside the Workload and System descriptions enables one Schedule vocabulary to cross that boundary. The evaluation spans placement inside a device up to one eight-GPU Scale-Up node; multi-node Scale-Out is outside that span.

1.2 SSG as a Shared Execution Representation

SSG separates the three constituents and binds them in one relation:

$$\text{Execution} = \text{Schedule}(\text{Workload}, \text{System}).$$

- AbstractSSG represents Workload structure—its CMC graph and Op Hierarchy.
- CalibratedSSG represents System structure—resources, paths, and substrate-software capabilities—through a System Hierarchy.
- **Schedule** maps one onto the other through **Parallel** (spatial) and **Pipeline** (temporal) organization. Tiling is a construction within Parallel; DP, TP, EP, PP, and related projections such as FSDP, are named views of this space rather than separate primitives.
- InstantiatedSSG records the selected mapping and expresses the induced execution as CRS—Compute, Remap, Sync—together with its realization evidence.

Graph transformation chooses an equivalent or partitioned Workload form; *strategy mapping* chooses how one such form executes on the System. Schedule determines requirements and CRS represents their execution; a named collective is therefore a Translate artifact rather than a Schedule primitive.

Existing intermediate representations capture substantial subsets of this relation without the same object boundary. Operator IRs primarily represent computation and local transformations; distributed parallelization systems expose selected mapping decisions; communication libraries encode realized movement; and framework runtimes encode higher-level plans. These representations may interoperate, but the Workload form, the System capability description, and the binding Schedule are not maintained as independent inputs to one shared representation with explicit realization evidence. SSG is not proposed as a replacement for these IRs or runtimes. Its distinction is the representation boundary: graph transformation produces a Workload form, Schedule maps that form onto a calibrated System, and the resulting InstantiatedSSG preserves the selected mapping together with the CRS execution structure and its realization state. This separation is what permits the same execution decision to be inspected before lowering, after realization, and at different execution scales.

1.3 Positioning and Contributions

SSG sits at the architecture–compiler–runtime boundary. This paper makes three contributions. **First**, a paired Op and System Hierarchy over shared CMC, bound by Parallel×Pipeline Schedule and materialized as CRS. **Second**, a Build/Traverse/Translate architecture that separates graph transformation from strategy mapping while preserving the selected mapping, provenance, and unequal realization depth in an InstantiatedSSG. **Third**, a three-group evaluation—operator (G1), framework (G2), and model-path (G3)—that uses explicit evidence states to distinguish representation, admission, realization, correctness, and measured performance, including negative cost on live Qwen3 replacement paths. The prototype uses bounded enumeration, explicit selection, and controlled overrides. It does not claim a global optimizer or uniform backend depth; it tests whether the representation remains meaningful when realization mechanisms differ across scopes.

2 Problem Context and Positioning

2.1 Necessary Specialization and Accidental Repetition

Specialization is necessary: code generation depends on device capabilities, communication on topology and transport, and integration on framework interfaces. The difficulty is maintaining their relationship. A tensor partition changes local operator shapes, buffer ownership, and communication together, so a change can require coordinated edits across several interfaces. SSG makes that joint relationship an explicit object whose realization stays associated with the selected graph form and resources.

2.2 Existing Local Owners

Operator compilers, communication libraries, and distributed frameworks provide complementary abstractions at different execution scales. SSG relates these scales through paired Workload and System hierarchies. Each body of work below can serve as a realization target for a selected region rather than an approach SSG displaces.

Operator compilers and kernels. Halide [29], TVM [2], MLIR [16], Triton [35], and CUTLASS [22] provide local transformation and device code generation. XLA [1], GSPMD [37], StableHLO [27], and IREE [15] improve portability, while FlashAttention [5, 6, 32] and MLA implementations [7] show the value of specialized fused kernels. SSG adopts their separation of computation from execution choices and places operator refinements within an explicit hierarchy of System Domains.

Distributed placement and MoE training. Sparse MoE systems, including GShard [17], Switch Transformers [8], Tutel [14], MegaBlocks [10], and DeepSeekMoE [4], establish routing and execution contracts. Placement and parallelism controls appear in MegatronLM [34], DeepSpeed [30], Alpa [42], ColossalAI [18], and recent expert-folding mappings [19, 38]. These systems motivate the framework-scale mapping choices considered here [43]; SSG connects their parallelism controls to the vocabulary used for operator-scale partitioning and temporal staging.

MoE communication and EP. NCCL [26], DeepEP [41], and UniEP [44] optimize dispatch, combine, and collective paths. HybridEP [21], NCCL EP [11], UBEP [20], incast-aware scheduling [3], and topology-aware replica placement [40] further specialize communication and placement. SSG represents movement together with adjacent computation and storage, so that their resource assignments and ordering can be described within the same selected mapping.

RL and post-training runtimes. HybridFlow/verl [33], OpenRLHF [12], and AReaL [9] orchestrate phases, resharding, and rollout–training overlap. MoE legality planners [39], dispatch correctness oracles [31], DAMoE [13], and serving systems such as ExpertFlow [36] address planning, correctness, and execution at related boundaries. This framework-scale study uses their phase organization and parallelism as projections of a Schedule. MOEFS [39] searches for deployment-realizable plans over parallelism, schedules, and kernels; SSG is complementary: it defines the cross-scale representation in which Workload form, System capability, Schedule binding, and realization evidence remain explicit before and after lowering, rather than introducing a stronger optimizer.

2.3 Scope

This paper focuses on DNN and language-model Workloads, including dense and sparse model graphs and their training, inference, and RL phases. The evaluation is accelerator-centric; host CPUs provide control and runtime support.

System comprises the hardware substrate and the low-level substrate software that exposes and constrains its execution capabilities: device drivers and runtimes, kernel and collective libraries, communication transports, and memory services. System does not include DNN/LM model semantics, high-level Workload phases, or upper framework policy that is a Translate target (e.g., Megatron, SGLang, or veRL execution plans).

3 SSG Concept and Architecture

3.1 Three Graph Forms

- **AbstractSSG** represents a DNN/LM Workload's CMC graph and Op Hierarchy.
- **CalibratedSSG** represents the System side—resources, paths, and the low-level substrate-software capabilities that expose and constrain execution—organized as a System Hierarchy.
- **InstantiatedSSG** records the selected Workload-to-System mapping; its execution product is expressed as CRS.

The first two are independent inputs: a hardware or substrate-software change rewrites only CalibratedSSG, and a Workload change rewrites only AbstractSSG.

3.2 Shared Structure: CMC and Paired Hierarchies

CMC is the common static vocabulary shared by Workload and System sides of SSG:

- **Compute:** value-producing and state-transforming operations;
- **Memory:** data, layout, residency, ownership, and lifetime;

- **Communication:** dependencies and movement between locations.

The same CMC terms apply from local operators to larger model paths. On the Workload side, the graph contains Compute and Memory nodes connected by Communication edges; on the System side, hardware Nodes and Memory nodes are organized by Domain layers and connected by *physical edges* where movement crosses Domains outside the parent–child hierarchy. Communication is therefore part of the shared structure, not a side effect of compute. Workload and System project that shared vocabulary through paired hierarchies:

- the Workload-side **Op Hierarchy** organizes CMC structure by grain—MacroOp (framework-scale block), DeviceOp (one device launch), and MicroOp (intra-launch unit)—together with Basic Ops and fusibility/launch boundaries called Groups;
- the System-side **System Hierarchy** organizes compute and storage resources, communication paths, Device/Fabric scopes, and substrate-software capabilities into legal realization scopes. Its layers are Domains, and the concrete layer instance a Schedule names is a Domain instance.

On both sides, homogeneous and heterogeneous organization distinguishes repeated units from differently formed or differently capable ones.

Figure 2 summarizes this pairing; body examples use the levels shown there.

3.3 Schedule

Schedule is the binding relation in

$$\text{Execution} = \text{Schedule}(\text{Workload}, \text{System}).$$

Given one Workload form and one calibrated System, it maps Workload axes and their implicit homogeneous or heterogeneous organization onto System Domain layers and their explicit resources. Its two families are **Parallel**, which places computation, data, or state across resources and has Tiling as its main construction, and **Pipeline**, which stages and overlaps them in time. The selected mapping determines placement, ownership, temporal staging, and resource assignment; the movement and synchronization needed to realize it are materialized as Remap and Sync in the resulting InstantiatedSSG.

The terminal representation of this binding is a strategy-matrix row, materialized at each relevant System Hierarchy boundary for one Workload node. A row carries the Domain instance, the spatial or temporal choice, the Workload-axis binding, and the hardware and edge decisions taken at that boundary; every field stays explicit, including those a given backend will not consume. The same schema describes a GEMM tile, a reduction fold, a communication edge, or a MacroOp-level framework plan at different Op and System extents, and the cases in Sections 5–7 share it.

MoE structure is a Workload fact rather than a Schedule family; its execution still combines with Parallel and Pipeline. How operator-scale communication enters this vocabulary is measured in Group 1 (Section 5.4); how training/inference/RL frameworks expose Schedule projections is measured in Group 2 (Section 6).

3.4 CRS: Execution Representation

After Schedule selection, InstantiatedSSG records the chosen Workload-to-System mapping. SSG represents the resulting dynamic execution as CRS—the abstract execution product of that Instantiated object, before target-specific realization. Here Compute is the *execution* of selected work, distinct from the static Compute nodes of the shared CMC vocabulary:

- **Compute:** perform selected computation or transformation;
- **Remap:** change data layout, residency, ownership, path, or endpoint;
- **Sync:** enforce ordering, visibility, dependency, and resource coordination.

3.5 Build, Traverse, and Translate

The prototype organizes this processing into three stages (Figure 3). **Build** constructs the two independent inputs: Workload CMC structure and Op Hierarchy become AbstractSSG, and System resources, paths, capabilities, and System Hierarchy become CalibratedSSG. **Traverse** applies one AbstractSSG to one CalibratedSSG in the order Section 3.6 sets out—graph transformation first, then strategy mapping over the paired hierarchies—walking the Op Hierarchy against the System Hierarchy and mapping communication operators alongside compute ones. Its output is one selected InstantiatedSSG with explicit selection provenance. **Translate** emits that selection at the layer the target exposes: an operator or communication artifact carrying launch-level correctness obligations, or a framework-native plan and process topology. It preserves the selected graph form and strategy mapping, and records the target layer and the source of the selection with the resulting artifact or plan, which is what lets one description admit unequal realization depth. The three stages are a conceptual dataflow; an implementation may realize them with compiler passes, runtime interfaces, or backend ABIs.

3.6 Two Separated Decisions

The representation separates two decisions that implementations frequently couple. *Graph transformation* takes a Workload form to an equivalent or explicitly partitioned Workload form—Sibling Fusion, Producer–Consumer Fusion, or decomposition into explicit CMC structure. *Strategy*

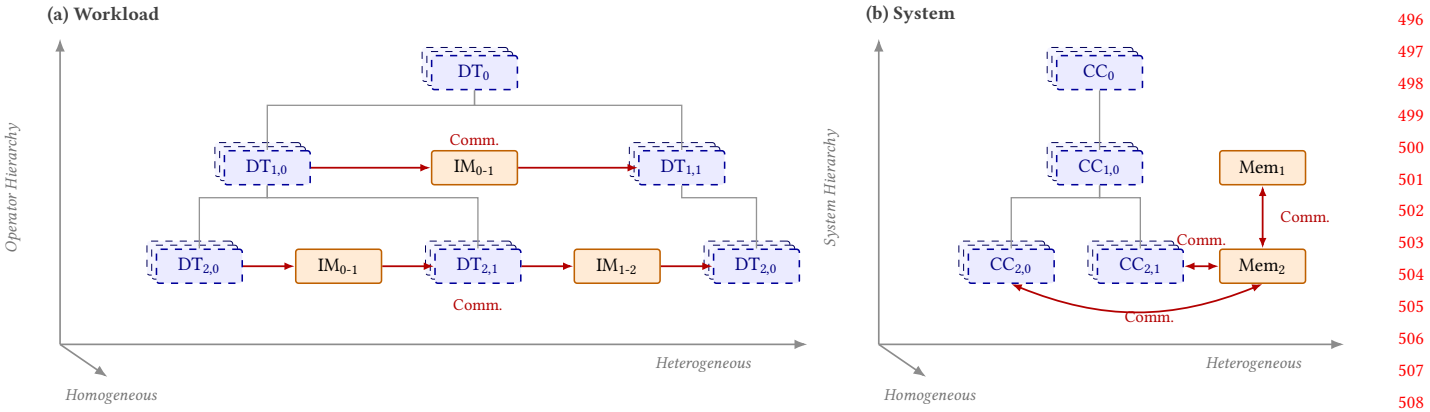


Figure 2. SSG concept framing. Workload and System sides expose corresponding hierarchical and homogeneous/heterogeneous organization; shared CMC structure is the Schedule substrate. Axes: Heterogeneous = pipeline-oriented scale; Homogeneous = parallel/pipeline replication (stack depth). Notation: $DT_{\ell[,i]}$ = layer-ℓ data tiles (MacroOp i); $IM_{i\text{-}j}$ = intermediate memory between MacroOps i and j; $CC_{\ell[,t]}$ = layer-ℓ compute cores (type t); Mem_ℓ = layer-ℓ memory; Comm. = Communication. Memory nodes attach only via Comm. (no hierarchy edges); System Comm. links CC↔Mem and Mem↔Mem only.

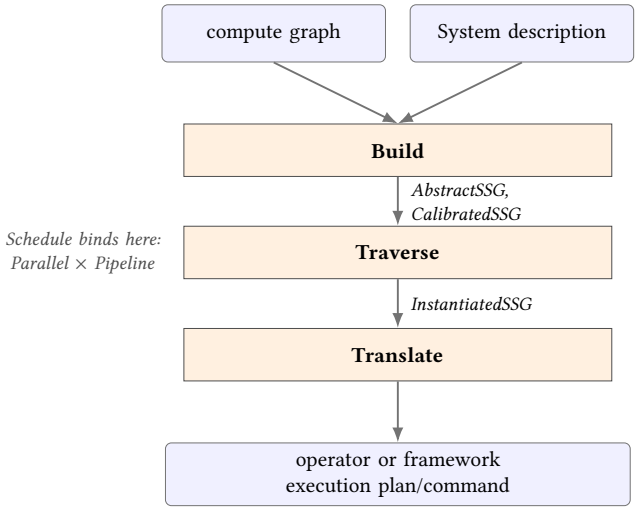


Figure 3. Process view of SSG (vertical dataflow). Sharp boxes are stages; rounded boxes and arrow labels are data. Build consumes a compute graph and System description and emits AbstractSSG and CalibratedSSG; Traverse emits InstantiatedSSG; Translate emits an operator or framework execution plan/command. Schedule is the binding performed during Traverse.

mapping then applies Schedule to one such form against one CalibratedSSG, yielding the selected execution. A transformation is not itself a Schedule, and backend translation must not silently change the selected mapping.

4 Evaluation Methodology

4.1 Prototype and Platforms

The evaluation uses a prototype covering the Architecture stages to different depths. On H20, operator Cases exercise Build, Traverse, and Translate into generated artifacts where admitted. On one eight-A800 Scale-Up machine, training/inference/RL Cases Build and Traverse to a selected mapping, then Translate primarily into framework-native plans. Bounded model-path Cases insert selected decisions into live Qwen3 GQA/MoE paths while leaving unmatched framework regions explicit.

4.2 Evidence States

Five states of an execution candidate are kept apart throughout, each with its own predicate. A candidate is *represented* when its Workload, System, and Schedule choices can be encoded in the SSG schema; *admitted* when the prototype's static capability and legality checks accept the selected mapping; *realized* when that mapping is lowered to a target artifact or framework-native plan and observed to execute; *correct* when its output satisfies the declared numerical or functional criterion against the matched reference; and *attainable* only for performance actually measured for a realized and correct candidate under the declared protocol, which is not a global optimum. The predicates are independent: a candidate may be representable but inadmissible, admissible but unrealizable, realizable but incorrect, or correct and substantially slower than its reference.

4.3 Baselines and Validity Conditions

A leaf GEMM is compared with a leaf GEMM; a content-mode all-to-all is not compared with an equal-chunk

collective; a communication-only result is not a whole-MoE result. Every quantitative claim is tied to its Workload, System, Schedule, baseline, repetition, correctness test, and selection source. Model, shape, dtype, hardware generation, transport, prompt regime, placement, graph state, clocks, repetitions, dispersion, run ID, and artifact lineage are recorded per admitted row.

The G2 experiments use two explicitly separated inference protocols. Figure 5a uses a distinct prompt per repetition and measures the offered-load response on one eight-A800 budget. Figure 5b uses a repeated shared-prefix prompt with radix prefix caching enabled and therefore characterizes a cache-hit, decode-dominant regime; nominal input length then differs from measured prefill work. We do not combine the two protocols into a single aggregate result; each panel is interpreted only under its stated prompt, caching, and measurement conditions.

5 Evaluation: Operator Realization

5.1 Four Operator Treatments

Operator scope is where the representation is exercised most fully: Build, Traverse, and Translate all reach generated artifacts here. The evidence is organized into four treatments, ordered so that each adds one mechanism to the one before it.

A **basic single operator** is one functional block, and the question is whether its axes can be enumerated, mapped onto System Hierarchy boundaries, lowered, and checked. **Fusion** adds no operator; it changes which functional blocks share a launch or a Schedule, so it is examined only once single operators are representable. A **communication operator** is also a functional block—it occupies control, compute, and memory resources to move data—so it enters the same Schedule instead of being delegated to a fixed library interface after the compute boundary is fixed. **Compute+communication fusion** is then the fusion question applied to a Compute–Communication pair rather than a new mechanism. Table 1 lists the four treatments with the evidence attached to each.

5.2 Basic Single Operators

Decomposability is a property of an operator's axes, not of its name, and three classes cover the operators evaluated here. On an *Independent* axis a tile produces a final output region and needs no merge: elementwise operators on every axis, GEMM on M and N. On a *Reduce* axis a tile produces a partial value that a finalization step combines: GEMM on K, and a reduction on the axis it reduces. On a *Prefix* axis a tile produces a partial value *and* an ordering obligation, because a correction must be applied in axis order. SSG records per axis whether inputs may be partitioned, whether a tile's output is final, and whether an order is imposed, so that decomposition legality is decided before a System mapping is chosen. One operator can therefore have axes from two classes, which is why GEMM appears in both the Independent and the Reduce discussions below.

Table 1. The four operator treatments of Group 1. Each row adds one mechanism to the row above it. All evidence listed was taken on H20; the A800 sm90 path of the last row is not supported on that platform.

Treatment	Representative forms	Measured evidence
Basic single operator	Independent (elementwise, GEMM M/N); Reduce (GEMM K, row reduction); Prefix (scan)	candidates, funnel, correctness, matched leaves
Fusion	Producer–Consumer and Sibling Fusion; GQA and MoE boundaries	selected Launch Fusion compile and correctness
Communication operator	dispatch/combine, gather/scatter, collectives	selected correctness and transport gates
Compute+ Comm. Fusion	dispatch+GEMM1, expert+combine, GEMM+ReduceScatter	selected realization and correctness

Candidate construction. Candidate construction over four frozen GEMM shapes yields between 2,892 and 2,988 candidates. The counts are nearly equal because the space is enumerated over System Hierarchy boundaries and axis contracts rather than over shape values: a shape changes which candidates are legal at the margin, not how many degrees of freedom exist. Those degrees of freedom are unevenly distributed, and Table 2 shows how: 22 distinct per-layer combinations per shape, `Single` only at some boundaries, and a family mix that changes with depth—Parallel alone at the outermost Device–Physical Core boundary and at warp(-group)–CUDA unit, both families present at the CTA–warp(-group) boundaries and in equal numbers at the group level, and Pipeline with more than twice the records of Parallel at warp(-group)–tensor unit. Which family exists at a boundary is a property of the hardware rather than of the taxonomy, and the representation records it instead of assuming a uniform space.

From admission to correctness. We first enumerate 2,892 structurally admitted candidates for the frozen H20 FP16 GEMM shape (conventional GEMM M=1024, N=256, K=1024). After execution, 482 distinct measured candidate configurations are observed, of which 288 are numerically correct; the remaining 194 fail either the numerical or launch/execution correctness checks. We do not merge candidates according to an assumed performance threshold.

Table 2. Where the two Schedule families are available in the H20 GEMM candidate pool. Rows are System Hierarchy boundaries; counts are layer-wise records from the same admitted pool, so a column sums over boundaries rather than over unique candidates. Single and MultiParallel are aggregated under Parallel.

System Hierarchy boundary	Parallel	Pipeline
Device–Physical Core	2,892	–
Physical Core–CTA	2,784	108
CTA–warp(-group)	1,266	702
CTA–warp(-group) group	462	462
warp(-group)–tensor unit	264	660
warp(-group)–WMMA unit	282	282
warp(-group)–CUDA unit	1,404	–

The purpose of the funnel is therefore to separate structural admission, execution, and correctness rather than to claim statistical equivalence among candidates.

The failures are structured rather than uniformly distributed. All 77 warp-group matrix-multiply-accumulate (WGMMA) configurations with `cta__warp_group=Pipeline` fail, while the corresponding 77 spatially parallel (`MultiParallel`) configurations pass. This identifies `cta__warp_group` as a boundary at which the observed failure population separates; we do not interpret the correlation alone as a causal explanation of the failure mechanism. The value of the SSG representation is that the differing execution choices remain explicit at that boundary. Among the labels that do pass, throughput spans 1.9%–57.4% of matched cuBLASLt [24], a 29.5× range: clearing the correctness gate does not narrow performance. The frozen default is absent from the 482-configuration space and therefore has no percentile here.

Figure 4 presents both readings, on separate populations and separate axes. Its right panel counts the successive populations of this one shape's enumerated pool. Its left panel reports an independent remeasurement over nine shapes—18 FP16/BF16 points agreeing elementwise with cuBLAS—attaining 49.60%–110.08% of the reference as pool-best measured attainments per shape rather than automatic selections. Together, the panels show that representation admits a large space, that realization and correctness reduce it, and that what survives is competitive on some shapes and not others.

Reduce and Prefix. Two further H20 sweeps extend the same construction to the axis classes whose tiles produce non-final output: 66 last-dimension FP32 row-reduction shapes, and 48 CumSum cases split into six one-dimensional scans and 42 column scans over two-dimensional inputs. Every generated implementation in both sweeps is numerically correct, so the two classes differ only in what they attain.

SSG outperforms the matched CUB reference [23] on all 66 declared row-reduction cases, with a CUB/SSG throughput ratio of 1.087–2.791×. This comparison is intentionally a contract-level comparison rather than an inner-loop comparison: under the declared interface, SSG emits one fused launch, whereas the reference implementation requires two library launches. We therefore attribute most of the observed margin to launch structure and do not interpret the result as a general reduction-kernel speedup. The complementary CumSum sweep illustrates the same boundary: one-dimensional scans benefit from the launch amortization (CUB/SSG 0.827–1.671 in four of six cases), whereas wide two-dimensional column scans generally do not (0.625–1.122, faster in only two of 42, both at the largest grid measured). The current lowering emits one CTA per scan—enough to outperform two library launches on a single long row, but not enough to fill the device when the work should spread across a wide grid. The result is thus evidence that the representation can express and realize fused execution under the stated contract, rather than evidence of uniformly superior primitive performance.

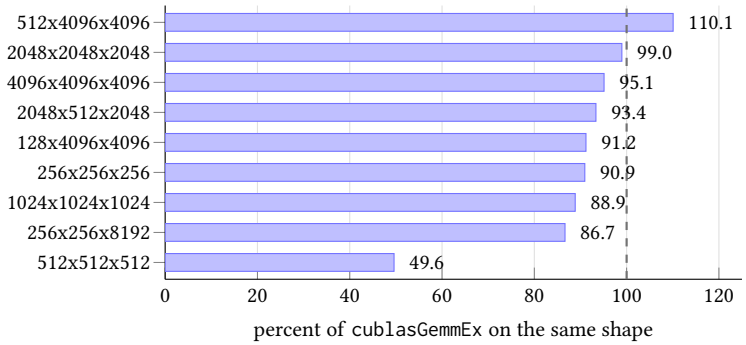
5.3 Fusion: Launch and Schedule Levels

Fusion in SSG is not a new operator but a change of boundary, and it occurs at two levels of the Op Hierarchy.

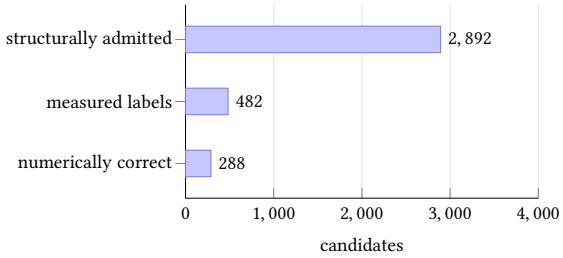
Launch Fusion places several Groups (fusibility units in the Op Hierarchy) inside one DeviceOp: the fused blocks share one device launch, while each Group keeps its own Schedule. Two forms occur. *Producer–Consumer Fusion* joins blocks connected by a data dependency, so the consumer reads its input where the producer left it instead of through a round trip to a shared level of the memory hierarchy. *Sibling Fusion* joins blocks with no direct data dependency between them, and yields launch and occupancy amortization rather than data locality. Both are graph transformations: they produce another legal Workload form, and the Schedule is chosen afterward for that form, which is the separation stated in Section 3.6.

Schedule Fusion operates inside one Group: a series of operators that can genuinely cooperate is given a joint Parallel or Pipeline Schedule over the same resources, so their executions interleave rather than merely share a launch. It is representable in the current prototype but not implemented.

The two levels are what separate a functional boundary from a fusion boundary. Which blocks exist is a Workload fact; which blocks share a launch is a Launch Fusion decision, and the two need not agree. In Section 7, the framework's fused MoE forward operation is one launch containing routing, expert computation, and combine, and the region SSG selects is a single functional block inside it. In the attention path of the same section, the attention core and the output projection are separately replaceable functional blocks, while the prefill block-pair variants place different numbers of blocks in one launch. Current evidence covers



(a) Realized GEMM throughput, fp16, H20. Every point is *numerically identical* to the reference, so this compares two bitwise-agreeing implementations. bf16 tracks fp16 to within 0.64 percentage points on all nine shapes. The dashed line is the reference.



139 numerical + 55 launch failures among the 482 measured

(b) Admission to correctness for one frozen H20 fp16 GEMM shape. 194 of 482 measured labels, 40.2%, do not reach correctness, and the failures are structured: 77 pipelined variants fail while their 77 spatially parallel pairs pass.

Figure 4. Attained performance and admitted population for H20 GEMM (Section 5.2). (a) Throughput of the generated implementation as a percentage of the matched cuBLAS reference on nine shapes, all at elementwise agreement; the reference is 100%. (b) Successive populations of one frozen shape's enumerated pool: structurally admitted, distinct measured, and numerically correct. The panels report different quantities and are read separately.

selected Launch Fusion compile and correctness points together with those model-path integrations.

5.4 Communication as Scheduled Work

A communication operator is a functional block like any other: it occupies control and compute resources, it has input and output layouts, and it completes. What is specific to it is that the selected Schedule and System facts determine source and destination ownership, payload and layout, physical path, peer count, memory transition, synchronization, and the CTA, warp, engine, or queue resources used to issue, progress, and complete the movement. Each of those is already a field of the terminal strategy-matrix row (Section 3.3), so movement is traversed and mapped in the same pass as compute instead of being delegated to a fixed library interface once the compute boundary is fixed.

The consequence is a change of status rather than of implementation. A collective is a Translate artifact of a selected CRS subgraph, so two peer counts or two issue-resource choices are two rows of one matrix rather than two library calls, and Communication Execution Resources are selected execution resources on the same footing as compute units. The evidence available covers correctness and transport gates on selected H20 communication cases; the CTA/warp/engine traversal is enumerated but measured only at those selected points.

5.5 Compute + Communication Fusion

Because a communication operator is a scheduled block, fusing it with adjacent compute needs no additional mechanism: it is Launch Fusion across a Compute–Communication pair, with the same legality question of whether the two blocks' axis contracts agree on the shared axis.

An 8-rank CUDA-IPC dispatch+GEMM1 measurement closes a correctness point and records device-side evidence of overlap, and NVSHMEM dispatch/combine passes its stated transport criterion. Matched peer comparison is mixed: a single-expert small case matches the Torch baseline, and multi-expert configurations are slower in the current realization. Recording graph lineage, selected CRS, correctness, and matched performance as four separate states of one row is what lets that shortfall be attributed to a realization at a named boundary rather than to the representation. These are operator-level measurements on synthetic MoE structure, at the scope of one dispatch or one expert stage.

The A800 single-kernel communication+compute path depends on sm90 tensor-memory and barrier primitives that this platform does not expose: a capability fact about the substrate, recorded as such in the evidence state.

Across the four treatments, one terminal row expresses a GEMM tile, a reduction fold, a scan, a dispatch, and a fused dispatch+GEMM. In none of these cases do represented, admitted, realized, correct, and attainable coincide, and the distances between those populations are what Group 1 measures.

6 Evaluation: Framework Scheduling

6.1 Framework Controls as Schedule Projections

Training, inference, and reinforcement-learning (RL) runtimes expose data, tensor, pipeline, and expert parallelism (DP, TP, PP, EP) and phase placement as separate controls. In SSG they are named projections of one Parallel×Pipeline

Table 3. Equal-budget DP×TP throughput (tok/s) under two offered loads on eight A800 GPUs.

Offered load	DP1×TP8	DP2×TP4	DP4×TP2
64	3,881.79	4,639.34	4,540.40
512	4,941.50	6,723.58	7,781.20

space, distinguished by the Remap and Sync that each binding induces: a TP binding tiles the weight axes of adjacent layers and therefore requires a cross-device reduction at every such layer, whereas a PP binding stages layers in time and transfers one activation per stage boundary. The System description determines the cost of that induced movement through its paths. Under Execution = Schedule(Workload, System), no projection is therefore preferred independently of its two inputs: the preferred projection changes when either input changes. The two series that carry throughput data vary one input each.

MoE structure is a Workload fact rather than a Schedule family; EP is an expert-axis Tiling choice applied to that Workload, so model structure and its execution mapping vary independently within the representation.

All measurements use one eight-A800 budget and vary transport explicitly. Where a choice is supplied by the caller rather than by automatic selection, the record retains that override so each measurement is attributed to the configuration that produced it; the remaining conditions, including the shared-prefix cache-hit decode-dominant protocol, are as stated in Section 4.3.

6.2 Varying the Workload: Offered Load Moves the Parallel Binding

DP1×TP8, DP2×TP4, and DP4×TP2 are one Parallel binding over the same eight devices, differing in which Workload axis is bound to the device Domain: DP consumes extent on the batch axis, TP on the within-layer weight axes. The batch-axis extent available for consumption is set by offered load, so a batch-heavy binding becomes profitable only once that extent exists. Table 3 measures Qwen3-8B decode throughput at two loads on one budget.

The measured response follows that extent. At load 64 the batch axis is narrow: DP2×TP4 and DP4×TP2 differ by 2.1%, which we do not read as an ordering under a three-repetition protocol, and both exceed pure TP8 by 17–20%. At load 512 the batch axis is wide and the ordering separates, with DP4×TP2 reaching 7,781.20 tok/s and exceeding TP8 by 57%. The same eight devices running the same model therefore prefer a different Parallel binding at two points on one Workload axis, which is why offered load is recorded as a condition of the comparison rather than as a tuning detail.

6.3 Varying the System: Transport Inverts the PP/TP Ranking

Holding the binding vocabulary fixed and varying the System's paths tests the other input. Because a TP binding induces a reduction at every tensor-parallel layer whereas a PP binding induces one activation transfer per stage boundary, degrading the path between devices should penalize TP more heavily, and the measured ranking should move toward PP.

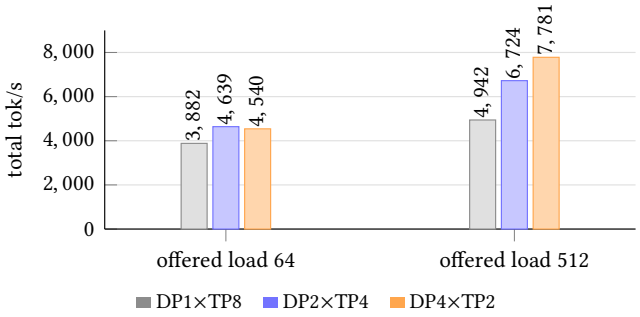
For Qwen3-8B the PP2×TP4-to-TP8 throughput ratio spans 0.588–1.031 across the sampled NVLink conditions and 0.856–3.401 with P2P disabled. For Qwen3-32B, PP2×TP4 stays below TP8 on NVLink (0.664–0.862) and crosses parity under host staging (0.791–1.134). Both shifts are in the predicted direction: the projection that induces less movement improves when the path carrying that movement degrades, and improves further with model size, which increases the volume each per-layer reduction must transfer. Figure 5 plots both series. The largest host-staged ratio coincides with a sharp throughput degradation in the TP8 baseline, so that point records a configuration-specific reversal rather than a proportional PP gain.

6.4 Phase- and State-Level Bindings

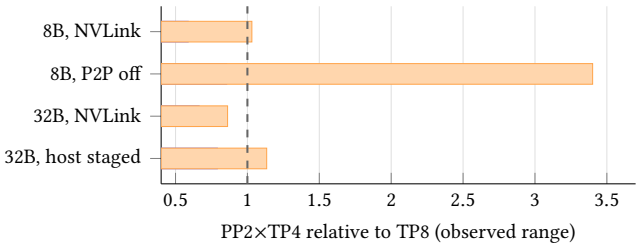
The remaining paths exercise the same vocabulary at coarser grain, each reported at the depth its interface exposes. Prefill/decode disaggregation is a Pipeline binding over phases, and its interface exposes one depth control per phase adjacency rather than an arbitrary Stage+Buffer composition; the PD runs establish functional execution, with interference-affected measurements excluded and the remainder exploratory. Training and RL bind resources rather than layers: supervised fine-tuning records 1.316× scaling from four to eight devices at a fixed batch size (65.8% efficiency), and two RL allocations specified as named resource partitions yield step times of 13.052 s and 22.323 s. Device-clock and machine-isolation records are absent for both, so these are descriptive observations of executable resource bindings, and a further RL DP comparison mixes incompatible timing regimes and is excluded.

Increasing the exposed PP buffer depth reduces throughput in three tested conditions, and with P2P disabled a nonzero depth causes a reproducible illegal memory access at high concurrency, so that control is non-monotonic in performance. Only one such depth is reachable through the current interface, which bounds the Pipeline coordinate this group measures.

Across the four paths, one Parallel×Pipeline vocabulary describes framework-native plans as well as the operator-scale bindings of Section 5, and the two series carrying throughput data show the preferred projection moving with the Workload input and with the System input in



(a) Equal-budget response, Qwen3-8B on eight A800 GPUs, NVLink, a distinct prompt per repetition, medians of three repetitions. At load 64 the two batch-parallel configurations differ by 2.1%; at load 512 DP4×TP2 exceeds both.



(b) Observed range of the same ratio across the sampled model-size and transport configurations, shared-prefix cache-hit decode-dominant protocol. The endpoints are the minimum and maximum observed ratios, not confidence intervals: light bars mark the minimum and dark bars the maximum, and the dashed line is parity. The ranking inverts with transport as well as with model size.

Figure 5. Framework-reachable schedule response on one eight-A800 budget (Sections 6.2 and 6.3). The two panels come from different prompt regimes and are read separately.

turn. The empirical scope is the evaluated A800 hardware, frameworks, workloads, and transport settings.

7 Evaluation: Model-Path Integration

7.1 Question and Experimental Scope

G3 examines whether an execution region described by SSG can be embedded in an existing model runtime, and what cost that integration introduces. We study selected attention regions in Qwen3-0.6B [28] and a *router region* in Qwen3-30B-A3B [28], using SGLang on H20. The surrounding model, memory management, and unreplaced operators remain framework-native. Separately, we evaluate an expert down-projection at single-rank operator scope as an operator-path check. These experiments therefore distinguish model-path substitution of a routed region from isolated operator realization. Output agreement, observed execution of the replacement, and timing are recorded separately because, under CUDA graph [25] capture, a replacement that executes in eager mode need not execute during graph replay.

Table 4. Qwen3-0.6B region ablation on H20 with CUDA graphs enabled. Decode throughput is in tokens/s; total request latency is in seconds.

Replacement	Decode	Total
None (framework)	541.4164	0.126172
Attention core only	58.8802	1.083824
Output projection only	427.9485	0.158000
Attention core and output projection	57.2896	1.115633

7.2 Attention-Region Cost Attribution

For Qwen3-0.6B, we vary two regions independently: the *attention core* computes attention scores, masking and softmax, and the product with the values; the *output-projection region* concatenates heads and applies the output projection. QKV projection and the preceding normalization and rotary-embedding region are unchanged. All four configurations use batch size one, 32 input tokens, 64 output tokens, and CUDA graphs with a maximum captured batch size of one. Table 4 reports the recorded median per-token decode throughput and total request latency. Each configuration has one recorded benchmark run; these medians are not estimates over independent repeated runs.

Replacing the attention core increases total latency by 0.957652 s, compared with 0.031828 s for the output projection alone. The attention-core increment accounts for 96.8% of the increment observed when both regions are replaced. The attention-core replacement is therefore not presented as an end-to-end performance improvement. Its role in G3 is diagnostic: because the replacement boundary is explicit, the cost of the generated implementation can be separated from the unchanged framework regions. In this case, the boundary localizes the dominant slowdown to the attention-core realization, while the output-projection replacement incurs a much smaller incremental cost. This illustrates the intended use of SSG evidence states at model-path scope: successful integration and poor performance remain simultaneously observable rather than being collapsed into a single end-to-end result. The ablation characterizes this implementation at the recorded sequence length, batch size, and attention kernel.

7.3 Local Fusion and End-to-End Behavior

Separate attention experiments exercise several realizations of the same prefill block pair, differing in how many operations share a launch. Two serving comparisons then move decode throughput by under one percent—57.3925 to 57.4278 tokens/s for a fusion of the output-projection region, and 58.8802 to 58.5662 tokens/s for an optimization at the attention-core boundary—so end-to-end performance here is not attributable to launch count. The replacement boundary makes regional cost measurable; the quality of the implementation inside that boundary remains decisive.

Table 5. Qwen3-30B-A3B latency diagnostic on H20. Both configurations enable CUDA graphs; the replacement-enabled configuration falls back during capture. Throughput is in tokens/s and total latency in seconds.

Configuration	Prefill	Decode	Total
Framework	728.7357	214.3947	0.337985
Replacement enabled	32.7010	205.0093	1.286168

7.4 Expert Computation in a Routed Model

For Qwen3-30B-A3B, SSG replaces the router region inside the fused MoE forward operation on eight ranks (TP8×EP8). The selected region is bound on every rank and records 13,056 generated calls (1,632 per rank), zero fall-throughs to the framework path, and identical greedy output for the tested prompt: the region executes on every rank and its output agrees. A separate single-rank expert down-projection experiment is reported as an operator-path check.

A separate latency experiment enables CUDA graphs and generates 64 output tokens from 32 input tokens at batch size one (Table 5). The replacement-enabled configuration executes 5,536 generated-kernel dispatches, but also records 48 fallbacks during graph capture, so its decode replay can use the framework implementation. The observed 205.0093 tokens/s, 95.6% of the framework decode throughput, therefore characterizes this mixed execution configuration, as do the substantially slower prefill and total latency.

The correctness and timing experiments differ in graph mode, output length, and implementation revision, and they support separate conclusions: the graph-disabled experiment demonstrates kernel integration with output agreement, whereas the graph-enabled experiment characterizes an integration configuration with capture-time fallback.

7.5 Interpretation

G3 shows selected-region integration with explicit boundaries: attention substitutions localize cost, and a generated expert GEMM executes inside a routed serving operator. The attention path is substantially slower and graph capture can change the executed MoE path, so the supported claim is diagnostic integration rather than end-to-end speedup.

8 Scope and Limitations

Representation, selection, and coverage. Candidate enumeration, manual selection, and controlled overrides expose choices; neither an automatic optimizer nor a predictive performance model is evaluated, and the fastest measured candidate is not a global optimum. Realization depth is uneven: local operators may reach generated kernels, whereas some distributed and phase-level choices bind to framework plans, so a common description does not imply a uniform compilation path.

Model-level validity. Model-path studies replace selected regions only, so greedy-output agreement holds for the recorded inputs and modes. No combined attention-plus-expert replacement or matched multi-rank serving speedup is claimed.

External validity. Results cover the declared H20 and A800 single-node configurations and do not establish behavior on other accelerators, inter-node fabrics, or arbitrary models.

Engineering cost. The evaluation does not measure authoring effort, backend implementation cost, or framework-integration maintenance burden.

9 Conclusion

SSG makes the relationship between a Workload and the System on which it executes explicit. Paired Workload and System hierarchies share a common Compute, Memory, and Communication vocabulary; Schedule binds them through Parallel and Pipeline organization; and the resulting InstantiatedSSG exposes the induced Compute, Remap, and Sync structure together with its realization evidence.

The three evaluation groups test distinct consequences of that representation. At operator scope, enumeration and realization separate admitted candidates from those that execute and produce correct results. At framework scope, the preferred Schedule projection changes with the Workload input and with the System input in turn. At model-path scope, explicit replacement boundaries make both successful execution and integration cost observable. These experiments do not establish an automatic optimizer, uniform backend coverage, or end-to-end performance gains; they show that the same Workload–System–Schedule object can preserve decisions across scales while exposing where realization succeeds, fails, or incurs cost. Making Schedule search systematic and extending realization depth across heterogeneous backends, under the same representation and evidence contract, are the natural next steps.

Generative AI Disclosure

Large-language-model assistants (Claude, ChatGPT, and Cursor) were used to draft, revise, and copy-edit portions of this manuscript, including text and table wording; all technical claims, numbers, and figure contents were checked and finalized by the authors.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga,

Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 265–283.

[2] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, 578–594.

[3] Evyatar Cohen, Jose Yallouz, Alexander Shpiner, Mark Silberstein, Sylvia Ratnasamy, and Isaac Keslassy. 2026. Incast-Free MoE Rate-Based Scheduling. arXiv:2607.26340 [cs.DC] https://arxiv.org/abs/2607.26340

[4] Damai Dai, Chengqi Deng, Chenggang Zhao, R. X. Xu, Huazuo Gao, Deli Chen, Jiashi Li, Wangding Zeng, Xingkai Yu, Y. Wu, Zhenda Xie, Y. K. Li, Panpan Huang, Fuli Luo, Chong Ruan, Zhifang Sui, and Wenfeng Liang. 2024. DeepSeekMoE: Towards Ultimate Expert Specialization in Mixture-of-Experts Language Models. arXiv:2401.06066 [cs.CL] https://arxiv.org/abs/2401.06066

[5] Tri Dao. 2023. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. arXiv:2307.08691 [cs.LG] https://arxiv.org/abs/2307.08691

[6] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *Advances in Neural Information Processing Systems (NeurIPS 2022)*.

[7] DeepSeek-AI. 2024. DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model. Introduces Multi-head Latent Attention (MLA) and the DeepSeekMoE architecture. arXiv:2405.04434 [cs.CL] https://arxiv.org/abs/2405.04434

[8] William Fedus, Barret Zoph, and Noam Shazeer. 2022. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. *Journal of Machine Learning Research* 23, 120 (2022), 1–39. arXiv:2101.03961 [cs.LG] https://arxiv.org/abs/2101.03961

[9] Wei Fu, Jiaxuan Gao, Xujie Shen, Chen Zhu, Zhiyu Mei, Chuyi He, Shusheng Xu, Guo Wei, Jun Mei, Jiashu Wang, Tongkai Yang, Binhang Yuan, and Yi Wu. 2025. AReaL: A Large-Scale Asynchronous Reinforcement Learning System for Language Reasoning. NeurIPS 2025. arXiv:2505.24298 [cs.LG] https://arxiv.org/abs/2505.24298

[10] Trevor Gale, Deepak Narayanan, Cliff Young, and Matei Zaharia. 2023. MegaBlocks: Efficient Sparse Training with Mixture-of-Experts. MLSys 2023. arXiv:2211.15841 [cs.LG] https://arxiv.org/abs/2211.15841

[11] Amos Goldman, Nimrod Boker, Maayan Sheraizin, Nimrod Admoni, Artem Polyakov, Subhadeep Bhattacharya, Fan Yu, Kai Sun, Georgios Theodorakis, Hsin-Chun Yin, Peter-Jan Gootzen, Aamir Shafi, Assaf Ravid, Salvatore Di Girolamo, James Dinan, Xiaofan Li, Manjunath Gorentla Venkata, and Gil Bloch. 2026. NCCL EP: Towards a Unified Expert Parallel Communication API for NCCL. arXiv:2603.13606 [cs.DC] https://arxiv.org/abs/2603.13606

[12] Jian Hu, Xibin Wu, Wei Shen, Jason Klein Liu, Zilin Zhu, Weixun Wang, Songlin Jiang, Haoran Wang, Hao Chen, Bin Chen, Weikai Fang, Xianyu, Yu Cao, Haotian Xu, and Yiming Liu. 2024. OpenRLHF: An Easy-to-use, Scalable and High-performance RLHF Framework. arXiv:2405.11143 [cs.AI] https://arxiv.org/abs/2405.11143

[13] En-Ming Huang, An-Cheng Chang, Bai-Cheng Jeng, Shih-Hao Hung, and H. T. Kung. 2026. Decoding the Skew: Distribution-Aware MoE Inference with Adaptive Kernel Dispatch. arXiv:2607.23099 [cs.DC] https://arxiv.org/abs/2607.23099

[14] Changho Hwang, Wei Cui, Yifan Xiong, Ziyue Yang, Ze Liu, Han Hu, Zilong Wang, Rafael Salas, Jithin Jose, Prabhat Ram, Joe Chau, Peng Cheng, Fan Yang, Mao Yang, and Yongqiang Xiong. 2023. Tutel: Adaptive Mixture-of-Experts at Scale. arXiv:2206.03382 [cs.DC] https://arxiv.org/abs/2206.03382

[15] IREE Authors. 2024. IREE: Intermediate Representation Execution Environment. https://github.com/iree-org/iree. Accessed 2026-08-03.

[16] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2021)*. 2–14. doi:10.1109/CGO51591.2021.9370308

[17] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. 2020. GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding. arXiv:2006.16668 [cs.CL] https://arxiv.org/abs/2006.16668

[18] Shenggui Li, Hongxin Liu, Zhengda Bian, Jiarui Fang, Haichen Huang, Yuliang Liu, Boxiang Wang, and Yang You. 2023. Colossal-AI: A Unified Deep Learning System For Large-Scale Parallel Training. In *Proceedings of the 52nd International Conference on Parallel Processing (ICPP 2023)*. ACM, 766–775. doi:10.1145/3605573.3605613

[19] Dennis Liu, Zijie Yan, Xin Yao, Tong Liu, Vijay Korthikanti, Evan Wu, Shiqing Fan, Gao Deng, Hongxiao Bai, Jianbin Chang, Ashwath Aithal, Michael Andersch, Mohammad Shoeybi, Jiajie Yao, Chandler Zhou, David Wu, Xipeng Li, and June Yang. 2025. MoE Parallel Folding: Heterogeneous Parallelism Mappings for Efficient Large-Scale MoE Model Training with Megatron Core. arXiv:2504.14960 [cs.DC] https://arxiv.org/abs/2504.14960

[20] Yipeng Liu, Chang Liu, Si Shen, Jiaqi Zheng, Mingfan Li, Yuyang Yang, Guanhua Li, Yuquan Zhang, Yimeng Xu, Zhongzhe Hu, Zhiyuan Huang, Qihang Duan, Junsong Wang, Wenkai Ling, Baochuan Yang, Xianzhi Yu, Han Bao, Yijie Chen, and Guihai Chen. 2026. UBEP: Re-architecting Expert Parallelism Communication Library for Production Superpods. In *Proceedings of the ACM SIGCOMM 2026 Conference (SIGCOMM 2026)*. ACM. arXiv:2607.06202 [cs.DC] doi:10.1145/3789240.3829183

[21] NVIDIA. 2025. Optimizing Communication for Mixture-of-Experts Training with Hybrid Expert Parallel. https://developer.nvidia.com/blog/optimizing-communication-for-mixture-of-experts-training-with-hybrid-expert-parallel/. NVIDIA Technical Blog; Hybrid-EP.

[22] NVIDIA Corporation. 2023. CUTLASS: CUDA Templates for Linear Algebra Subroutines. https://github.com/NVIDIA/cutlass. Accessed 2026-08-03.

[23] NVIDIA Corporation. 2024. CUB: Cooperative Primitives for CUDA C++. https://nvidia.github.io/cccl/cub/. CUDA C++ Core Compute Libraries documentation.

[24] NVIDIA Corporation. 2024. cuBLAS Documentation: cuBLASLt and Device API. https://docs.nvidia.com/cuda/cublas/. CUDA Toolkit documentation.

[25] NVIDIA Corporation. 2024. CUDA Graphs. https://docs.nvidia.com/cuda/cuda-c-programming-guide/. CUDA Programming Guide.

[26] NVIDIA Corporation. 2024. NCCL: NVIDIA Collective Communications Library. https://github.com/NVIDIA/nccl. Accessed 2026-08-03.

[27] OpenXLA. 2023. StableHLO: A Portability Layer Between ML Frameworks and Compilers. https://github.com/openxla/stablehlo. Specification and dialect; accessed 2026-08-03.

[28] Qwen Team. 2025. Qwen3 Technical Report. arXiv:2505.09388 https://arxiv.org/abs/2505.09388

[29] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2013)*. ACM, 519–530. doi:10.1145/2491956.2462176

[30] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2020)*. ACM, 3505–3506. doi:10.1145/3394486.3406703

[31] Dipankar Sarkar. 2026. The Correctness Illusion in LLM-Generated GPU Kernels. arXiv:2606.20128 [cs.SE] https://arxiv.org/abs/2606.20128

[32] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. 2024. FlashAttention-3: Fast and Accurate Attention with Asynchrony and Low-precision. arXiv:2407.08608 [cs.LG] https://arxiv.org/abs/2407.08608

[33] Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. 2024. HybridFlow: A Flexible and Efficient RLHF Framework. Open-source implementation: verl. arXiv:2409.19256 [cs.DC] https://arxiv.org/abs/2409.19256

[34] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. arXiv:1909.08053 [cs.CL] https://arxiv.org/abs/1909.08053

[35] Philippe Tillet, H. T. Kung, and David Cox. 2019. Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL 2019)*. ACM, 10–19. doi:10.1145/3315508.3329973

[36] Bingyang Wu, Chao Jin, Zili Zhang, Xinming Wei, Yinmin Zhong, Ruidong Zhu, Chengxu Yang, Xin Jin, and Yuliang Liu. 2026. ExpertPlex: A High-Goodput Disaggregated Serving System for MoE LLMs with Adaptive Persistent Kernels. arXiv:2607.18002 [cs.DC] https://arxiv.org/abs/2607.18002

[37] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, Ruoming Pang, Noam Shazeer, Shibo Wang, Tao Wang, Yonghui Wu, and Zhifeng Chen. 2021. GSPMD: General and Scalable Parallelization for ML Computation Graphs. arXiv:2105.04663 [cs.DC] https://arxiv.org/abs/2105.04663

[38] Zijie Yan, Hongxiao Bai, Xin Yao, Dennis Liu, Tong Liu, Hongbin Liu, Pingtian Li, Evan Wu, Shiqing Fan, Li Tao, Robin Zhang, Yuzhong Wang, Shifang Xu, Jack Chang, Xuwen Chen, Kunlun Li, Yan Bai, Gao Deng, Nan Zheng, Vijay Anand Korthikanti, Abhinav Khattar, Ethan He, Soham Govande, Sangkug Lym, Zhongbo Zhu, Qi Zhang, Haochen Yuan, Xiaowei Ren, Deyu Fu, Tailai Ma, Shunkang Zhang, Jiang Shao, Ray Wang, Vasudevan Rengasamy, Rachit Garg, Santosh Bhavani, Xipeng Li, Chandler Zhou, David Wu, Yingcan Wei, Ashwath Aithal, Michael Andersch, Mohammad Shoeybi, Jiajie Yao, and June Yang. 2026. Scalable Training of Mixture-of-Experts Models with Megatron Core. arXiv:2603.07685 [cs.LG] https://arxiv.org/abs/2603.07685

[39] Quan Yuan and Jie Zhao. 2026. Searching for Plans You Can Actually Build: A Realizability-Aware Full-Space Optimizer for MoE Training and Serving. arXiv:2607.18631 [cs.DC] https://arxiv.org/abs/2607.18631

[40] Lingyun Zhang, Henghua Zhang, Shilei Gu, Kai Mo, Shuai Han, Shiyong Li, Yanpeng Wang, and Dou Shen. 2026. TAOT: Topology-Aware Optimal Transport for Dynamic Expert Replica Placement in MoE Training. arXiv:2608.03676 [cs.DC] https://arxiv.org/abs/2608.03676

[41] Chenggang Zhao, Shangyan Zhou, Liyue Zhang, Chengqi Deng, Zhean Xu, Yuxuan Liu, Kuai Yu, Jiashi Li, and Liang Zhao. 2025. DeepEP: An Efficient Expert-Parallel Communication Library. https://github.com/deepseek-ai/DeepEP. Accessed 2026-08-03.

[42] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. 2022. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, 559–578.

[43] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2024. SGLang: Efficient Execution of Structured Language Model Programs. In *Advances in Neural Information Processing Systems*. arXiv:2312.07104 https://arxiv.org/abs/2312.07104

[44] Size Zheng, Xuegui Zheng, Li-Wen Chang, and Jidong Zhai. 2026. UniEP: Unified Expert-Parallel MoE MegaKernel for LLM Training. In *Proceedings of the 35th International Symposium on High-Performance Parallel and Distributed Computing (HPDC 2026)*. ACM, 387–401. arXiv:2604.19241 [cs.DC] https://arxiv.org/abs/2604.19241