Compiling Tensor Graphs and Their Adjoints for Heterogeneous Clusters

A November 2026 Survey and Higher Categorical Synthesis

Anonymous submission draft

3 November 2026

Contents

Abstract **4**

1 Scope, evidence, and the central question **5**
1.1 What is being compiled? . . . 5
1.2 Evidence cutoff and inclusion policy . . . 5
1.3 Four meanings of heterogeneity . . . 6
1.4 The current answer . . . 6

2 A compiler stack for distributed differentiable programs **6**
2.1 The seven boundaries . . . 6
2.2 Capture is a semantic decision . . . 7
2.3 Differentiation is a program transformation with storage . . . 7
2.4 Distribution and schedule are different objects . . . 8
2.5 Local compilation and distributed compilation interact . . . 8
2.6 Recovery is part of the executable contract . . . 8

3 Global-array compilation and automatic parallelization **9**
3.1 GSPMD and the global-array contract . . . 9
3.2 PartIR and Shardy . . . 9
3.3 JAX: three levels of control . . . 9
3.4 JAX's explicit cotangent distribution . . . 10
3.5 Alpa and hierarchical search . . . 10
3.6 Unity and joint graph rewriting . . . 10
3.7 PyTorch capture and AOTAutograd . . . 11
3.8 DTensor and explicit SPMD types . . . 11
3.9 PyTorch AutoParallel . . . 11
3.10 nnScaler and user-defined search spaces . . . 12

4 Scheduling, asymmetric resources, and composed training **12**
4.1 Tessel, GraphPipe, and the shape of the schedule . . . 12
4.2 Metis and device-aware planning . . . 12
4.3 HexiScale, formerly FlashFlex, and workload allocation . . . 13
4.4 Hetu v2 and HSPMD . . . 13
4.5 JaxPP and MPMD pipelines . . . 13
4.6 Tessera and overlap-aware stage costs . . . 14
4.7 Piper and a programmable training DAG . . . 14

4.8 Established training systems remain useful boundaries . . . 14

5 Communication-aware kernel compilation 15
5.1 Why operator boundaries are insufficient . . . 15
5.2 Triton-distributed . . . 15
5.3 DITRON . . . 15
5.4 Syncopate . . . 16
5.5 Mirage and persistent execution . . . 16
5.6 Helion, CompileIQ, and local search . . . 16
5.7 Virtual tensors and transient materialization . . . 16

6 Differentiation across compiler and runtime boundaries 17
6.1 Enzyme and parallel source programs . . . 17
6.2 DaCe AD and data-centric representations . . . 17
6.3 Higher-order differentiation and language semantics . . . 18
6.4 The derivative contract for a foreign operator . . . 18
6.5 Differentiating the optimizer itself . . . 18

7 Task runtimes, languages, and execution substrates 19
7.1 DISTAL and SpDISTAL . . . 19
7.2 Legion, Realm, and logical regions . . . 19
7.3 StarPU and explicit task implementations . . . 19
7.4 Chapel and distributed language support . . . 20
7.5 IREE as a deployable compiler/runtime boundary . . . 20
7.6 Apache TVM, Relax, and Disco . . . 20
7.7 Ray Compiled Graph . . . 20
7.8 Monarch and the programmable cluster . . . 21

8 Rust, Julia, and smaller distributed stacks 21
8.1 Burn and CubeCL in the 2026 snapshot . . . 21
8.2 Rust ownership and GPU kernels . . . 22
8.3 Reactant and Enzyme in Julia . . . 22
8.4 Dagger and distributed task composition . . . 22
8.5 MLX and a direct TCP example . . . 22
8.6 llama.cpp RPC and inference boundaries . . . 23
8.7 A language does not determine the whole stack . . . 23

9 Additional compiler lineages and capability boundaries 23
9.1 TensorFlow DTensor, OneFlow, and MindSpore . . . 23
9.2 Galvatron and SuperScaler . . . 24
9.3 Futhark and Dex . . . 24
9.4 TileLang and the kernel-language layer . . . 24
9.5 A capability map . . . 25

10 TCP/IP, RDMA, and remote-memory models 26
10.1 Control traffic and tensor traffic . . . 26
10.2 Host staging is a real part of the graph . . . 26
10.3 UCX and UCC . . . 26
10.4 NVSHMEM and PGAS . . . 27
10.5 PyTorch communication interfaces . . . 27
10.6 MPI and progress . . . 27
10.7 A simple communication lower bound . . . 28
10.8 Ordinary networks favor coarse regions . . . 28

11 Weakly connected islands and changed training algorithms 28
11.1 DiLoCo and the optimizer boundary . . . 28
11.2 Decoupled DiLoCo . . . 29
11.3 A taxonomy of equivalence claims . . . 29

12 Comparative evidence and reported performance 29
12.1 Reading the results without creating a false leaderboard . . . 29
12.2 What the literature does not yet establish . . . 30

13 A representation semantics for distributed adjoints 31
13.1 The mathematical setting and its limits . . . 31
13.2 Logical tensors and physical representations . . . 31
13.3 Representation-preserving differentiation . . . 31
13.4 Why replication causes reductions . . . 32
13.5 The transposes of collectives . . . 32
13.6 Legal deferral of reductions . . . 33
13.7 Residuals compose as reverse programs . . . 33
13.8 Higher derivatives and linearization boundaries . . . 34
13.9 Approximate forward equality is insufficient . . . 34

14 Execution plans as a higher categorical structure 34
14.1 Event structures and interfaces . . . 34
14.2 A schedule-equivalence theorem . . . 35
14.3 Costs are annotations, not equalities . . . 35
14.4 Layout change as a double-category intuition . . . 35
14.5 What the higher structure contributes . . . 36

15 Local execution records, sheaves, and descent 36
15.1 A site of dependency-closed contexts . . . 36
15.2 A precise gluing result . . . 36
15.3 Why the cover should not simply be the device partition . . . 37
15.4 From sets to spaces . . . 37
15.5 Smooth semantics is additional structure . . . 37

16 Effects, indexed monads, and metaprogramming 38
16.1 Why pure tensor syntax is not enough . . . 38
16.2 An indexed computation interface . . . 38
16.3 Differentiation and effects need a compatibility law . . . 38
16.4 A proposed differentiable core language . . . 39
16.5 What an AI-assisted metaprogram should generate . . . 39

17 A proposed compiler/runtime contract 39
17.1 Separate the model, the strategy, and the deployment . . . 39
17.2 Capability discovery . . . 40
17.3 A pass pipeline with explicit obligations . . . 40
17.4 A schedule certificate . . . 40
17.5 Exact accumulation and update commits . . . 41
17.6 A nontrivial training region . . . 41
17.7 Rust as an execution shell . . . 42

18 Evaluation requirements for the next generation 42
18.1 Workload coverage . . . 42
18.2 Hardware and network coverage . . . 42

18.3 Semantic and numerical checks 42
18.4 Performance measures 43
18.5 Failure and elasticity 43
18.6 Metaprogramming effort 43

19 Research agenda and final assessment **44**
19.1 The missing shared intermediate contract 44
19.2 Proof-carrying strategies with empirical costs 44
19.3 The answer for heterogeneous AI clusters 44

Authorship, evidence, and availability **45**

References **45**

Abstract

Compiling a tensor program to a cluster means more than selecting GPU kernels. The compiler must choose representations of distributed arrays, generate communication, preserve reverse-mode derivatives, schedule asynchronous events, manage transient storage, and define what happens when workers or networks fail. Heterogeneous machines make each obligation more visible: device capabilities, memory capacities, link speeds, and available differentiation rules need not coincide. We survey the research and publicly documented systems available by **3 November 2026**, emphasizing developments in 2025 and 2026 and retaining earlier work where it establishes the architecture of current approaches. The survey covers global-array compilers, automatic parallelization, programmable distributed training, communication-aware kernel compilers, scientific-computing runtimes, Rust and Julia systems, and training across weakly connected resource islands.

The main finding is architectural. The evidence supports substantial progress on individual layers and several useful integrated stacks, but does not establish a universally optimizing compiler for arbitrary effectful graphs on arbitrary mixtures of CPUs, GPUs, and network fabrics. Recent advances include asymmetric SPMD annotations, explicit coupling of primal and cotangent distribution, distributed tiling, chunk-based computation/communication overlap, programmable global training DAGs, and cluster-wide actor control. These advances solve different problems and should not be ranked by unrelated speedup numbers.

We give a precise semantic synthesis. Layouts are representations equipped with decoding maps; communication has linear transposes; training plans carry residuals and optimizer versions; and schedule transformations are directed refinements between event structures. We prove conditional statements about representation-preserving differentiation, collective transposition, schedule equivalence, and compatible local execution records. Higher categorical and topos-theoretic structures organize these obligations without substituting for backend validation. Finally, we propose a compiler/runtime contract and an evaluation agenda for simplifying AI-cluster metaprogramming. No new cluster benchmarks, executable compiler implementation, machine-checked proof, or experimental performance result is claimed.

Keywords: distributed compilation; automatic differentiation; tensor graphs; heterogeneous computing; SPMD; MPMD; GPU communication; higher categorical semantics; effect systems; metaprogramming.

1 Scope, evidence, and the central question

1.1 What is being compiled?

The question is whether we can write a mathematical tensor computation, differentiate it, and execute the resulting training computation across a collection of machines without repeatedly writing placement logic, collective protocols, remote-call adapters, and GPU-specific kernels. This is a question about the composition of several compilers and runtimes. A local tensor compiler may have excellent fusion and code generation while providing no distributed placement. A distributed runtime may schedule arbitrary remote tasks while knowing nothing about derivatives. A language may support reverse-mode differentiation and several GPU vendors while requiring substantial additional work for multi-node training.

Throughout, a *tensor graph* is a typed program whose primitive operations act on finite-dimensional arrays or explicitly represented sparse arrays. Its edges carry values together with shape, element type, layout, and effect information. A graph can contain structured control flow, but such control flow must have defined capture and lowering semantics. The phrase *arbitrary graph* is always relative to a supported language, primitive library, shape discipline, and effect interface. It does not mean that an unannotated foreign function, a Python callback with hidden mutation, or an unspecified communication protocol automatically acquires a correct distributed derivative.

An *adjoint* means the reverse-mode pullback of a linearization, usually called a vector-Jacobian product. For a differentiable map $f : X \to Y$, at a primal point x we write

$$f_x^* : T^*_{f(x)}Y \longrightarrow T^*_x X, \qquad \bar{y} \longmapsto Df(x)^*\bar{y}.$$

This is not an inverse of f, and it is not in general the right adjoint of a functor. In Euclidean tensor coordinates, the star is a transpose after the chosen pairings have identified tangent and cotangent spaces. Complex differentiation needs its own convention. Nondifferentiable primitives require stated subgradient or surrogate rules. These distinctions become operational when replicated parameters, partial sums, quantization, and distributed normalization interact.

1.2 Evidence cutoff and inclusion policy

The evidence cutoff is 3 November 2026, using the requester's local date. The search combined recent-date queries with searches of official proceedings, authors' preprints, project repositories, and current documentation. The bibliography identifies papers separately from software documentation and design proposals. The newest systems were checked against their primary landing pages or maintained project material; selected documentation was read for the exact limitations relevant to distributed execution. Benchmark numbers are attributed to their reporting authors and are not independently reproduced here.

This is a broad, source-audited narrative survey. It is not a registered systematic review, and it does not claim that every workshop paper, vendor branch, private deployment, or subsequently edited documentation page has been discovered. Search-engine crawl dates are not publication dates. A preprint appearing in 2025 and a proceedings version appearing in 2026 are one research lineage, with the proceedings version preferred for venue and numerical claims. A conference talk announcement is evidence that a feature was presented, not evidence that it has a universal correctness proof or stable production API.

The principal inclusion criterion is a concrete reduction in the work needed to express, transform, differentiate, distribute, or execute tensor computations. Local-only systems are included where they supply an essential component or clarify a common category mistake. Inference-only systems are included with that limitation visible. Distributed optimizer algorithms are

included where they address the weak-network part of the question, with a separate warning that changing the optimizer is a change of semantics rather than a transparent lowering.

1.3 Four meanings of heterogeneity

We distinguish hardware heterogeneity, memory heterogeneity, communication heterogeneity, and program heterogeneity. Hardware heterogeneity includes different GPU generations, different vendors, CPUs alongside accelerators, and devices with different supported data types. Memory heterogeneity includes host RAM, device memory, registered network buffers, pinned staging allocations, and persistent checkpoint storage. Communication heterogeneity includes direct GPU links, PCIe, shared memory, Ethernet sockets, InfiniBand, RoCE, and links crossing administrative boundaries. Program heterogeneity includes different pipeline stages, different expert workloads, dynamic sequence lengths, host-side control, and distinct executables on different subsets of the cluster.

These dimensions do not imply one another. A framework can run the same source code on NVIDIA and AMD in separate jobs without allowing one collective communicator to span both. A compiler can generate CPU and GPU binaries without assigning one differentiable graph across them. A runtime can use TCP/IP for rendezvous while moving tensor payloads over a different transport. A system can support multiple programs on one vendor's GPUs without supporting arbitrary device mixtures. Consequently, a portability claim should be read as a statement about a specific boundary, not about the entire cluster.

The most useful formulation is therefore conditional: given a supported graph, a differentiated primitive set, a discovered resource topology, and transport implementations meeting specified ordering contracts, produce a legal distributed training plan. Performance optimization takes place inside this feasible set. Correctness and optimization should be independently inspectable.

1.4 The current answer

The current research does not point to one universal language. It points to convergence around explicit intermediate representations and composable boundaries. Global-array systems reduce manual sharding work; strategy languages reduce manual schedule implementation; communication-aware kernels reduce hand-written overlap; actor and task runtimes reduce process orchestration; and typed distribution contracts reduce errors in the relationship between forward and backward execution. Each can simplify metaprogramming while leaving the others necessary.

We organize the survey around this separation. Sections on existing systems report observed capabilities. The semantic sections supply our own conditional derivations and proposed abstractions. The final architecture is a research synthesis, not a description of an already implemented universal compiler. This separation is essential for a publication: it prevents a useful common mathematical language from being mistaken for experimental validation.

2 A compiler stack for distributed differentiable programs

2.1 The seven boundaries

A practical stack crosses seven boundaries: source capture, differentiation, distribution, scheduling, local code generation, transport execution, and durable recovery. Source capture turns language execution into an inspectable program. Differentiation constructs primal/residual/pullback computations. Distribution chooses logical ownership and partial results. Scheduling assigns events to resources and establishes dependencies. Local code generation produces device-specific executables. Transport execution realizes remote data

movement and completion. Recovery maintains the relationship among model state, optimizer state, random state, and execution epochs.

The order need not be a single straight pipeline. A compiler may differentiate before partitioning, differentiate a distributed IR, or alternate transformations. It may revisit partitioning after a memory estimate changes. It may retain high-level operations while specializing low-level kernels. The important requirement is that every reordering of passes comes with a reason why its observable behavior and derivative interpretation are preserved.

MLIR provides a framework for representing and transforming computations at several abstraction levels; its dialect mechanism is a useful implementation substrate, not a distributed correctness theorem [R81]. StableHLO provides a specified tensor language and collective operations, supporting interchange across frameworks and compilers [R82]. PJRT defines a compiler/runtime integration boundary through plugins [R83]. These three solve related but distinct interoperability problems. None, by its existence alone, determines a global optimizer policy, a cross-vendor collective implementation, or a recovery protocol.

2.2 Capture is a semantic decision

Tracing specializes a program relative to observed structure and metadata. It can be extremely effective for numerical code whose control flow is represented by explicit loop and branch operators. It can also miss behavior that remains in the host language. A Python conditional evaluated while tracing is not automatically a data-dependent conditional in the resulting graph. A shape inferred from one sample may become a recompilation guard. Foreign calls need abstract evaluation and lowering definitions. Mutation must be represented, functionalized, or excluded.

These choices determine what *arbitrary* can mean. For a graph compiler, arbitrary supported DAG topology is already a significant capability: residual connections, branching, multiple losses, shared weights, and expert dispatch are more general than a linear chain of Transformer blocks. Arbitrary host-language programs add recursion, unbounded data structures, unknown effects, and partiality. A paper about general tensor graphs should specify the first boundary before claiming the second.

The distinction also affects distributed optimization. Static global structure enables cost modeling and advance allocation. Dynamic graphs require online decisions, a family of specialized plans, or a runtime with safe fallbacks. The compiler must not silently substitute one sample's structure for all executions. Conversely, rejecting all dynamic behavior would unnecessarily exclude useful workloads. Structured dynamism is the practical middle ground: shapes, control-flow regions, and effect dependencies remain explicit enough to analyze.

2.3 Differentiation is a program transformation with storage

Reverse mode needs information from the forward evaluation. A derivative rule for multiplication needs the multiplicands; a nonlinear activation needs its input or a sufficient residual; an attention backward pass needs selected intermediate information or a recomputation recipe. Thus the transformed program is better written

$$f^{\mathrm{aug}} : X \to Y \times R_f, \qquad f^{\mathrm{rev}} : R_f \times T^*Y \to T^*X.$$

The residual object R_f may itself be sharded, offloaded, rematerialized, compressed, or checkpointed. Its lifetime often extends past the lifetime of a forward activation as seen by the original source program. Any cluster compiler that treats AD as merely appending a few mathematical operators will underestimate storage and communication.

It is equally important to keep parameter versions explicit. If the reverse pass reads a parameter value different from the one used by the forward pass, the resulting quantity need not equal

the derivative of that forward computation. Weight stashing, recomputation, pipelining, and asynchronous updates are therefore linked. A legal plan identifies the parameter epoch associated with each residual and each pullback. This is a semantic condition that can be checked separately from the schedule's throughput.

2.4 Distribution and schedule are different objects

Distribution says where the pieces of a logical value reside and how partial pieces represent the whole. Scheduling says when and on which execution resources the operations run. Two schedules can implement one distribution. Two distributions can implement one mathematical function. Combining these into a single opaque collection of runtime callbacks makes both optimization and debugging difficult.

A representation of distributed arrays should distinguish disjoint shards, identical replicas, and unreduced contributions. An unreduced tensor is not simply a replica with stale data: it represents an additive contribution to a logical result. Its legal consumers differ. Likewise, a pending asynchronous transfer is not a fully available tensor on the destination. The IR should distinguish a future handle from a readable buffer.

Scheduling is constrained by more than data dependencies. A collective imposes a participation protocol. Streams impose ordering. A bounded communication queue imposes capacity constraints. A device allocation imposes lifetime constraints. A remote buffer may require a registration lease. A correct scheduler must satisfy all of them. The graph of tensor values is therefore only one projection of the graph that actually executes.

2.5 Local compilation and distributed compilation interact

Kernel fusion changes launch overhead, memory traffic, and the points at which remote data becomes available. A partitioner that treats each operator as an indivisible latency can miss overlap exposed by tiled execution. A kernel optimizer that ignores network readiness can fuse away a useful communication boundary. This explains why the newest work crosses the old separation between local kernels and distributed communication.

Nevertheless, integration should not require one monolithic optimizer. A local compiler can expose several legal kernel variants, including their tile readiness and resource footprints. A global scheduler can choose among them using network and memory constraints. This division permits backend-specific expertise while retaining a common representation of dependencies. It also makes negative results explainable: a plan may fail because no legal layout exists, because storage is insufficient, or because no available kernel exposes the necessary overlap.

2.6 Recovery is part of the executable contract

An all-reduce that completes twice after a retry can double a gradient if the retry protocol is not idempotent. A checkpoint that saves weights but not optimizer moments can resume a different training trajectory. A worker group that changes membership mid-collective can block or produce an inconsistent update. These are not errors that a correct matrix-multiplication kernel can prevent.

We therefore treat recovery as part of the training program's specification. The contract may require fail-stop execution and restart from a committed checkpoint; it may permit elastic reconfiguration only between optimizer steps; or it may define an asynchronous optimizer whose semantics explicitly tolerates staleness. Each is reasonable in some setting. What is unreasonable is leaving the choice implicit while claiming that every execution preserves the same sequential training algorithm.

3 Global-array compilation and automatic parallelization

3.1 GSPMD and the global-array contract

GSPMD established a particularly influential approach: describe a computation on logical global tensors, supply selected sharding annotations, propagate layouts, and generate a program that runs on each partition [R01]. It makes data parallelism and operator partitioning instances of one representation problem. Its importance is the abstraction boundary, not a promise that all placement decisions become automatic. The programmer still needs a suitable logical mesh and useful constraints, and the backend still needs implementations for the resulting collectives.

The global-array view reduces the amount of source code that names individual workers. It also makes extensional reasoning possible: the distributed result should decode to the result of the global tensor program. That contract is stronger than merely agreeing on local tensor shapes. It requires every partial sum, replicated value, and resharding operation to be interpreted consistently. A compiler may choose different physical plans while preserving this logical meaning.

3.2 PartIR and Shardy

PartIR studies composition of SPMD partitioning strategies, addressing the difficulty of combining data, model, and optimizer partitioning through controlled transformations [R04]. Shardy brings work from the GSPMD and PartIR teams into an MLIR-based partitioning infrastructure. Its documented components include an axis-based representation, constraints, manual-computation regions, propagation, and a partitioning boundary; the documentation distinguishes the current use of the GSPMD partitioner from plans for further MLIR-based partitioning [R05].

The research direction is useful because fully implicit propagation can be hard to debug, while fully manual collectives can be hard to compose. A representation that exposes priorities and region boundaries allows expert knowledge to enter without discarding the global program. However, a partitioning IR does not by itself solve heterogeneous scheduling or remote memory semantics. Those are separate analyses and lowering contracts.

Shardy's MPMD design work is also relevant. Its public RFC introduces explicit computational fragments and transfers between meshes [R11]. It should be read as a design and implementation lineage rather than as evidence that every MPMD feature is released in every consumer. The significant idea is to preserve a global representation while permitting different programs on different device groups. This gives a natural place to describe pipeline boundaries and device-specific specialization.

3.3 JAX: three levels of control

Current JAX documentation distinguishes compiler-chosen automatic sharding, explicit sharding with automatic partitioning, and manual per-device programming. Explicit sharding makes distribution inspectable in types, while manual mode exposes collectives directly [R06]. This is a useful spectrum rather than a competition among mutually exclusive styles. A large application can keep ordinary tensor algebra global, make expensive layout choices explicit, and use a manual region for a specialized communication pattern.

JAX's multi-controller model still imposes coordination obligations across processes; the multi-process documentation explains initialization and distributed execution conventions [R09]. A global array abstraction does not eliminate process membership, deployment configuration, or the need for compatible calls across participants. In particular, an interconnect requirement cannot be inferred merely from the fact that a CPU backend and a GPU backend both exist.

For metaprogramming, JAX is attractive where the program is already expressible through its transformations and staged operations. Its transformations can be composed at a high level, and the resulting graph remains visible to the compiler. The remaining engineering question is how much of a heterogeneous application belongs inside a compiled region and how much belongs in an outer coordinator.

3.4 JAX's explicit cotangent distribution

The current autodiff-and-sharding guide makes a central distinction between physical replication and the intended distribution of cotangents. It documents `reduced` and `unreduced` states, whose cotangent roles are paired, and shows how delaying reductions can combine gradient contributions before communication [R07]. JEP 17111 separately analyzes efficient transposition of replication-inducing collectives [R08].

This is more than an ergonomic refinement. Distributed AD needs to know whether a value is one logical quantity repeated in memory, several independent quantities, or several contributions to a quantity. Those meanings can have identical local shapes. Treating them as different types exposes a reason why a reduction is required and where it can legally move. The semantic derivations later in this paper use their own notation, because the precise names and pairings of real APIs must not be conflated.

The practical limitation is equally clear. A good type discipline specifies the meaning of communication; it does not make a slow network fast or prove that a runtime has implemented every communication rule correctly. It provides a tractable boundary at which such claims can be tested.

3.5 Alpa and hierarchical search

Alpa combines inter-operator and intra-operator parallelism in a hierarchical planning architecture. Its compiler partitions a graph into stages and device meshes, searches sharding within stages, and performs orchestration and cross-mesh resharding [R02]. This made automatic model-parallel compilation substantially more concrete than a collection of manually selected parallelism flags.

The main lesson is that restricting a search problem can make it useful. A hierarchical search may be optimal inside a stated subproblem while not being globally optimal over every possible distributed program. Candidate stage boundaries, cost models, mesh choices, memory estimates, and schedule families determine the scope of the result. Therefore, the word *optimal* in an automatic parallelizer should always be followed by its search space and objective.

Alpa remains an important architectural reference. Whether a particular deployment should adopt a historical research implementation is a separate maintenance and compatibility question. This survey does not infer present-day support for current accelerator generations from the original paper's results.

3.6 Unity and joint graph rewriting

Unity jointly optimizes algebraic transformations and parallelization through a common parallel computation graph [R03]. The significance is that the graph being partitioned need not be fixed. An algebraically equivalent formulation can expose a more favorable parallel decomposition, alter communication volume, or change which operations can fuse.

This challenges a common compiler architecture in which local algebraic optimization finishes before distributed planning begins. The two can interact: an apparently more expensive local formulation may reduce remote transfers, and a locally optimal fusion may obstruct distributed overlap. A common representation lets the search consider these choices together.

The correctness obligation is not merely that each rewrite preserves tensor shapes. In a training compiler it must preserve the function and the required derivative interpretation, subject to the chosen numerical policy. Reassociation in floating-point arithmetic needs a different contract from exact algebraic equality. Our later refinement model keeps this distinction explicit.

3.7 PyTorch capture and AOTAutograd

PyTorch 2's compiler architecture combines dynamic Python capture with graph compilation, retaining a close relationship to the eager programming model [R12]. Its compiler documentation explains that AOTAutograd captures forward and backward graphs and that this can interact with the hooks traditionally used to overlap distributed communication [R13]. Thus successful local compilation is not sufficient evidence that a complete distributed training program has the desired overlap.

This architecture is attractive when the existing model ecosystem matters. The user can progressively compile regions while retaining ordinary host-language orchestration. The tradeoff is that graph boundaries, guards, custom operators, and collective visibility become part of performance engineering. A compiler that sees the backward graph can optimize it, but it must also retain enough structure to schedule communication at the intended time.

The newer AOT compilation interface documents autograd-aware compiled functions, including distributed examples [R87]. It reduces repeated compilation and deployment friction where supported. Artifact reuse still requires compatible shapes, devices, code, compiler settings, and runtime contracts. A reusable executable is not automatically a portable executable for every worker type.

3.8 DTensor and explicit SPMD types

DTensor represents distributed tensors using a device mesh and placements such as sharding, replication, and partial values [R14]. The 2026 TorchTitan account explains its move toward `spmd_types` in compute regions: communication is explicit, forward/backward distribution is coupled, and development-time checking can be erased for production execution [R15]. The package also distinguishes local and global SPMD reasoning [R16].

This change should not be summarized as the abandonment of distributed tensor abstractions. The documented architecture still uses DTensor for persistent storage in FSDP2. The boundary between storage representation and compute representation is being made more deliberate. The reported throughput improvements concern the authors' stated debug-model benchmarks, not every model or cluster.

There is an especially useful general principle here: one can automate validation without automating every placement decision. Explicit collectives can be checked against distribution types, allowing a programmer or generated schedule to remain in control. Conversely, an automatic planner can emit such a checked program as its output. The two approaches can reinforce one another.

3.9 PyTorch AutoParallel

The public AutoParallel project describes compiler-driven automatic sharding using a model, a mesh, and a linear-programming formulation; it is explicitly experimental [R17]. The October 2026 conference material reports a Llama-3 8B example matching a strong manual baseline with lower memory, but that is an announcement-level result and not a universal performance guarantee [R88].

Its relevance is direct: automatic discovery is entering a widely used imperative model ecosystem. The important evaluation questions are operator coverage, shape support, cost-model calibration, compilation time, memory feasibility, and interaction with backward execution. A bounded

optimizer can be exact for its encoding while the encoding omits useful schedules or inaccurately predicts execution time.

For a new metaprogramming layer, this suggests an interface rather than a dependency on a single solver. The planner should accept constraints, expose its candidate space, and return a schedule that another component can validate and explain. Solver choice can then change without changing model semantics.

3.10 nnScaler and user-defined search spaces

nnScaler's central contribution is to let domain experts construct search spaces through transformation, assignment, and ordering primitives rather than only choose among fixed parallelism recipes [R18]. Its documentation describes generation of a parallel PyTorch module and states restrictions for pipeline use, including an end-to-end loss-returning module form and static graph capture [R19]. These restrictions matter when interpreting generality claims.

This is a productive compromise between a universal optimizer and hand-written distributed code. Experts provide structure; the system handles repetitive plan construction and execution. A metaprogram can generate constraints or search-space definitions instead of emitting low-level send/receive code directly.

The limitation is that extending the primitive set transfers work to the extension author. A new operator needs semantics, partitioning behavior, cost information, and derivative support. Making these extension obligations explicit is a strength. Concealing them behind the phrase *arbitrary model* would be misleading.

4 Scheduling, asymmetric resources, and composed training

4.1 Tessel, GraphPipe, and the shape of the schedule

Tessel searches pipeline schedules for operator placement strategies and is implemented on a lineage related to nnScaler [R20]. GraphPipe generalizes the sequential chain of pipeline stages to a stage DAG, allowing independent branches to overlap [R21]. These address a different decision from choosing tensor shards: once work has a location, its execution order still has a large effect on memory and throughput.

The two ideas expose why a layer-sequence abstraction can be too restrictive. Neural networks have shared subcomputations, parallel branches, multiple outputs, and irregular expert paths. A schedule language should represent these without forcing them into one arbitrary total order. At the same time, the language must retain enough information to enforce resource and collective constraints.

The graph topology alone does not decide which schedule is best. Microbatch count, residual storage, backward dependencies, transfer staging, and device contention all matter. A useful system therefore exposes both the structural graph and a tunable scheduling layer. It should also report whether a schedule requires extra copies or additional materialized parameter versions.

4.2 Metis and device-aware planning

Metis explicitly targets automatic training plans for heterogeneous GPU combinations. It incorporates profiling, estimation, and search methods that account for different compute and memory capacities; its reported speedups vary widely across the evaluated models and hardware combinations [R22]. The exact point is heterogeneity-aware planning rather than merely selecting a different backend.

This addresses an important failure of uniform meshes. Equal tensor partitions need not produce equal execution time, and equal work assignments need not fit equal memory budgets. A planner

should permit unequal partitions or heterogeneous stage assignments. It also needs to distinguish resource capacity from throughput: a slow device with large memory may be useful for one role and harmful for another.

The paper does not justify arbitrary cross-vendor collectives or an unrestricted CPU/GPU mixture. Hardware combinations, network topology, and implementation support still bound the evidence. For an implementer, Metis is a model of how to formulate the heterogeneous search problem, not a license to ignore those boundaries.

4.3 HexiScale, formerly FlashFlex, and workload allocation

The work originally circulated as FlashFlex is titled HexiScale in its May 2026 revision. It studies asymmetric data, pipeline, and tensor partitioning over heterogeneous GPUs [R23]. Its relevance is the interaction between workload allocation and composed parallelism. Heterogeneity changes not just stage placement but the amount of data and model work assigned to different resources.

A heterogeneous cluster should not necessarily be treated as one flat collective group. Grouping similar devices into locally efficient islands can reduce synchronization penalties, while a higher-level schedule balances work between islands. This is a design inference from the problem structure, not a claim that every system implements the same hierarchy.

The objective also matters. Maximum tokens per second, minimum cost per token, fastest completion, and use of otherwise idle hardware are different optimization problems. A plan that improves utilization may worsen time to a fixed validation target. A publication should report which objective is being optimized and whether communication or convergence costs have been included.

4.4 Hetu v2 and HSPMD

Hetu v2, published at OSDI 2026, introduces hierarchical and heterogeneous SPMD annotations. Its account includes asymmetric sharding, progressive graph specialization for spatial heterogeneity, and graph switching for temporal heterogeneity such as changing workloads [R24]. This is one of the most directly relevant recent results for the question posed here.

The conceptual advance is that a declarative distributed tensor model need not imply perfectly symmetric local execution. Different subgroups can receive different specializations while preserving an interpretable global contract. This extends the usefulness of annotations beyond uniform accelerator arrays.

However, asymmetric expressiveness and unrestricted deployability are different claims. One must still inspect supported operators, device types, network paths, and graph-switching conditions. The system's demonstrated scenarios are evidence for those scenarios. A generic heterogeneous compiler architecture should expose the same distinctions in its capability schema rather than compress them into one Boolean *heterogeneous* flag.

4.5 JaxPP and MPMD pipelines

JaxPP combines JAX with multiple-program, multiple-data pipeline execution. Its MLSys 2025 proceedings version reports improved hardware utilization relative to the compared SPMD configuration; the project exposes annotations for stage boundaries and pipeline transformation [R10]. Its role is to make different pipeline-stage programs accessible within an existing transformation-oriented ecosystem.

The distinction between MPMD and SPMD is about executable structure, not about a mandatory transport. An MPMD plan may still use tightly coupled GPU collectives inside a stage and point-to-point transfers between stages. This permits different local compilation decisions without requiring the user to implement every stage process manually.

It is especially important to retain gradient accumulation semantics. Microbatches can be scheduled in different orders while their contributions are combined before one update, provided parameter versions and normalization are controlled. Changing the update frequency or permitting stale versions is a different algorithm. A schedule annotation should make that boundary inspectable.

4.6 Tessera and overlap-aware stage costs

Tessera's OSDI 2026 operational-systems paper reports production MoE pipeline training on thousands of GPUs. It co-optimizes partitioning and overlap, and adds runtime handling of routing-induced idle periods. Reported throughput improvements are 20-33% over its production baseline on the stated workloads, with additional controlled comparisons [R25].

The broadly useful lesson is that serial layer cost is not necessarily the right stage cost. Two layer combinations with similar isolated execution time can hide different amounts of communication. Consequently, stage balancing should reason about the schedule after overlap, not merely add independent profiling measurements.

This does not mean that production MoE results transfer unchanged to small Ethernet clusters. The available overlap depends on compute duration, link capacity, local concurrency, and routing behavior. The result is valuable because it changes the model used by the planner, not because its numerical improvement can be treated as a constant multiplier in another environment.

4.7 Piper and a programmable training DAG

The 2026 Piper preprint separates distributed strategy from runtime implementation. Model annotations and scheduling directives transform a unified global training DAG containing computation and communication; the system then emits per-device plans [R26]. This is particularly close to the metaprogramming problem: the user expresses a strategy without rebuilding the execution engine for every new composition.

The abstraction is useful even when no fully automatic planner is available. A human expert, a constraint solver, or an AI-assisted tool can generate the same schedule language. The runtime does not need to know which produced it. The next requirement is a checker that rejects schedules violating dependency, memory, collective, or versioning constraints.

Piper should be cited as a preprint in this evidence snapshot. Its claims about supported strategies and performance are not evidence for unrestricted dynamic graphs or arbitrary devices. Its architectural contribution remains important independently of those extensions.

4.8 Established training systems remain useful boundaries

Megatron Core documents combinations of tensor, pipeline, context, expert, and data parallelism, while DeepSpeed's current AutoEP documentation gives concrete compatibility restrictions [R84] [R85]. These are substantial systems, but their automation is relative to supported model patterns and parallelism mechanisms. They should not be mistaken for compilers from arbitrary tensor languages.

Zero-bubble pipeline research demonstrates that distinguishing different backward components can improve scheduling opportunities [R86]. This reinforces a general point: a backward pass need not be one indivisible task per layer. Exposing its structure can improve overlap and memory use.

A new metaprogramming layer should therefore interoperate with established libraries. It can generate supported configurations, preserve their state formats, and introduce custom regions only where necessary. Reimplementing every optimized kernel and optimizer is usually unnecessary. The scientific challenge is to specify the boundaries so that such composition has a clear meaning.

5 Communication-aware kernel compilation

5.1 Why operator boundaries are insufficient

The conventional distributed graph treats a matrix multiplication and a collective as separate operations. This is convenient for composition but can hide opportunities. A matrix product may produce tiles incrementally; a network operation may consume them incrementally. If communication waits until the whole producer completes, and the next compute kernel waits until the whole transfer completes, the system serializes work that could overlap.

Overlapping entire streams is one approach. A finer approach exposes readiness at the tile or chunk level. This requires more than moving calls onto different streams: the receiving kernel must not read a chunk before it arrives, and the sender must not reuse storage before the relevant completion. Resource use also matters. A communication kernel can consume execution capacity needed by computation, so overlap is not automatically free.

The newest compiler work is notable because it represents these dependencies explicitly enough to transform them. This is a distinct axis from automatic sharding. One can have excellent sharding and poor overlap, or a hand-chosen sharding with an excellent generated overlapping kernel.

5.2 Triton-distributed

Triton-distributed extends a Python kernel-programming model with communication primitives based on OpenSHMEM-style operations and studies computation/communication overlap in single-node and multi-node settings [R29]. The maintained repository documents NVIDIA and AMD targets and recent backend work [R98]. That is evidence for implementation paths on those platforms, not evidence that one arbitrary mixed-vendor communicator is interchangeable with another.

For metaprogramming, the gain is that communication becomes part of the kernel program rather than a separately hand-maintained CUDA extension. An optimization can coordinate tile production, memory access, and remote transfer inside one compilation framework. The cost is a lower-level programming responsibility than a global tensor expression: the author still needs to understand synchronization and communication structure.

An end-to-end training compiler can use such kernels as specialized implementations of typed distributed operators. It should retain a declarative operator contract above the kernel, so that derivative generation and global planning need not reason directly about every device instruction.

5.3 DITRON

DITRON's ICML 2026 paper introduces a distributed multi-level tiling abstraction with Core, Device, and Task levels. It reports 6-30% improvements on evaluated isolated kernels and 5-30% on evaluated vLLM inference configurations, together with NVIDIA and AMD portability results [R30]. These are scoped reported results; they do not establish training performance or arbitrary-graph adjoint compilation.

Its contribution to the present question is the extension of tiling across a cluster's hierarchy. A local tile is no longer considered independently of its device and remote placement. The programmer can express distributed structure at a higher level than raw communication calls while retaining the optimization opportunities of a kernel DSL.

The natural next step is compositional integration: a graph-level planner chooses a distributed operator implementation, and the tiling compiler supplies code plus readiness and storage contracts. That integration should preserve the distinction between logical tensor partitions and physical transfer chunks, which need not have identical boundaries.

5.4 Syncopate

Syncopate, published at OSDI 2026, introduces a communication-chunk abstraction and transforms local Triton kernels to align compute with chunk availability. It reports an average end-to-end speedup of 1.3 times and a maximum of 4.7 times on its evaluated multi-GPU workloads [R31]. The result concerns its workload and baseline set rather than all collectives or network fabrics.

The useful separation is between a chunk schedule and a kernel's internal structure. A schedule can be generated, reused from a template, or supplied by an expert. The compiler then performs the alignment work that would otherwise require bespoke kernel engineering.

This is directly relevant to simplifying metaprogramming. A higher-level strategy generator should be able to request a legal communication schedule without emitting the complete low-level fused kernel. Conversely, the kernel compiler should expose when a requested schedule is infeasible because of alignment, buffer, or resource constraints.

5.5 Mirage and persistent execution

Mirage's OSDI 2025 work searches across several levels of tensor-program optimization, including transformations that cross conventional kernel boundaries [R27]. The 2026 MPK work moves further into execution: its compiler lowers multi-GPU inference to SM-level task graphs, and an in-kernel runtime schedules them within persistent execution. The paper reports up to 1.7 times lower end-to-end inference latency in its evaluation [R28].

Persistent scheduling changes the relationship between compiler and runtime. More scheduling takes place on the GPU, reducing dependence on repeated host launches. However, the absence of host launches does not remove dependencies, memory lifetimes, or synchronization protocols. It moves their implementation boundary.

The training extension should not be presumed. Reverse-mode residuals, parameter versions, and gradient accumulation introduce obligations beyond inference. An inference mega-kernel is therefore an important component result, not a complete answer to distributed adjoint compilation. Its task representation may still offer a useful target for a future differentiated graph compiler.

5.6 Helion, CompileIQ, and local search

Helion provides a higher-level tile-oriented DSL using PyTorch-style operations and lowering through compiler infrastructure to Triton [R32]. The October 2026 account of its CompileIQ integration expands the autotuning search to include compiler controls alongside kernel configuration [R33]. This addresses local performance engineering and portability of kernel authoring.

The relevance to clusters is indirect but substantial. Automatic distributed planning becomes more valuable when local implementations can also be specialized without manual rewriting. A planner with an inaccurate local kernel model can make poor global choices; exposing a set of tuned local variants can improve that model.

Local autotuning should nevertheless remain distinguishable from cluster strategy search. An optimal tile shape for one isolated kernel may not be optimal under concurrent communication. A useful interface reports resource use and timing distributions rather than only one best isolated latency. That is a proposed integration requirement, not a claim that existing autotuners already supply the complete interface.

5.7 Virtual tensors and transient materialization

VTC's OSDI 2026 work represents data movement through virtual tensors and index mappings to eliminate unnecessary physical movement within its compilation scope [R34]. The separate

2026 Chorus work studies global scheduling of transient parameter materialization in sharded training, including the memory tradeoff between earlier all-gathers and longer buffer lifetimes [R35].

Both make a hidden cost explicit: a tensor's mathematical existence does not imply that every representation must be physically materialized at every boundary. However, eliminating local layout traffic is not the same as eliminating information that must cross a network cut. Remote consumers still need the required information unless the algorithm changes or recomputation supplies it.

An integrated compiler should combine these ideas carefully. Virtual views can avoid copies, while materialization scheduling chooses when actual storage becomes necessary. The checker must distinguish a view, a future allocation, and a readable materialized tensor. Otherwise, an apparent optimization can become an early read or an invalid alias.

6 Differentiation across compiler and runtime boundaries

6.1 Enzyme and parallel source programs

Enzyme differentiates compiler IR rather than requiring all numerical work to originate in one tensor framework. Its SC 2022 work addresses multiple parallel paradigms, including DAG-based parallelism, and demonstrates differentiation across supported parallel programming structures [R36]. This is especially relevant to applications mixing simulation, numerical kernels, and learned components.

The CUDA guide states a concrete limitation of cross-device differentiation: CPU code calling GPU kernels requires exported augmented-forward and reverse kernel functions connected through custom derivative support [R37]. This is a useful counterexample to the idea that LLVM lowering automatically solves heterogeneous AD. Target boundaries and launch semantics still need explicit treatment.

Enzyme can therefore serve as a derivative compiler inside a larger distributed stack. It does not by itself choose every graph partition, build every network communicator, or define checkpoint consistency. Its strength is a broad differentiation boundary that can include existing code, subject to supported semantics and custom rules.

6.2 DaCe AD and data-centric representations

DaCe AD's 2025 preprint studies automatic differentiation in a data-centric representation for machine learning and scientific computing [R38]. The earlier DaCeML project exposes symbolic AD and data-centric model optimization; its repository is archived in the inspected snapshot [R39]. These should be distinguished as related work rather than treated as one uniformly current software package.

A data-centric IR is attractive for reverse mode because storage movement, intermediate lifetimes, and recomputation are first-class concerns. That makes it possible to optimize the augmented forward computation and the reverse computation jointly. It also gives the compiler a place to express offload and distributed execution.

Still, support for data-centric differentiation and support for a particular distributed lowering are separate capabilities. A publication should identify whether AD is performed before or after distribution, which communication operations have derivative rules, and which stateful behaviors are covered. A general representation does not eliminate the need for those rules.

6.3 Higher-order differentiation and language semantics

CHAD gives a compositional, type-respecting account of forward and reverse AD for an expressive functional language [R75]. Efficient CHAD studies how to obtain the expected computational complexity through appropriate representations and transformations [R76]. Work on diffeological semantics addresses higher-order functions and higher derivatives without assuming that ordinary finite-dimensional vector spaces form a cartesian closed category [R77].

These results answer a semantic question that kernel and cluster papers often leave implicit: why should a source transformation compute the derivative of the program's meaning? They do not directly solve heterogeneous scheduling. Conversely, a sophisticated schedule does not solve the higher-order semantic problem.

For an AI-cluster language, the productive division is to use a small, well-specified differentiable core and make foreign or distributed operations explicit extensions. Each extension supplies a denotation, a differentiation rule, and an execution contract. Higher-order metaprogramming can then compose verified core transformations while acknowledging where backend obligations enter.

6.4 The derivative contract for a foreign operator

A foreign operation should provide five things: a primal specification, a shape and type rule, an augmented-forward/residual specification, a pullback specification, and a resource/effect signature. If the operator is distributed, it additionally supplies participation and completion rules. A cost model is useful but is not part of mathematical correctness.

The pullback condition is local and precise. If the primal implementation denotes f, its reverse implementation must satisfy, for every admissible x, direction v, and output cotangent $\bar{y}$,

$$\langle f_x^*(\bar{y}), v\rangle = \langle \bar{y}, Df(x)v\rangle.$$

This statement assumes that $Df(x)$ exists and that the pairings and numerical policy are specified. For custom gradients, the contract may deliberately specify a different operator, such as a surrogate derivative; it must then avoid claiming equality with the mathematical derivative of the primal.

A compiler can trust a registered rule, validate it empirically, or require a proof artifact. Those are different assurance levels. A finite-difference check can catch many mistakes but is not a proof for all inputs. Conversely, a proof of the abstract rule does not verify the compiled kernel's memory safety or its communication ordering. The registration interface should record both levels rather than collapse them.

6.5 Differentiating the optimizer itself

The phrase *adjoints of arbitrary graphs* can include differentiation through training. Then an optimizer step is part of the primal graph, and its state must be modeled. Momentum, adaptive moments, clipping, random minibatch selection, and parameter updates all become operations with explicit semantics. Some are smooth only on restricted domains or away from branch boundaries.

Distributed execution makes this harder because the optimizer state may be sharded differently from the parameters, and because a training step may contain reductions with numerical effects. A hypergradient through several steps also extends the residual horizon. The compiler may need rematerialization or checkpointing at the step level rather than only inside one model evaluation.

The survey's core results concern reverse-mode compilation of supported tensor programs. They should not be interpreted as evidence that every surveyed training system supports higher-order

differentiation through its distributed optimizer and recovery machinery. That is a separate capability to request and test.

7 Task runtimes, languages, and execution substrates

7.1 DISTAL and SpDISTAL

DISTAL compiles dense tensor algebra to distributed heterogeneous systems with separate descriptions of data distribution and computation scheduling. Its paper includes multicore CPU and multi-GPU nodes and evaluates generated programs on a large supercomputer [R40]. SpDISTAL extends the approach to distributed sparse tensor computations, separating algebra, sparse format, data distribution, and computation distribution [R41].

These are particularly relevant because the original question is broader than neural-network frameworks. Tensor algebra appears in scientific simulation, sparse methods, and numerical optimization. A compiler can simplify distributed metaprogramming by making data and computation mappings independently expressible.

However, a tensor-algebra compiler is not automatically a complete neural-network training compiler. Nonlinear primitives, differentiation, dynamic shapes, random operations, optimizer state, and recovery need additional semantics. DISTAL is a strong example of heterogeneous compilation at a defined language boundary, not evidence that every arbitrary neural graph and its adjoint are already covered.

7.2 Legion, Realm, and logical regions

Legion's logical regions describe data through index and field spaces, making access relationships visible to the runtime [R42]. Realm is the lower-level distributed, event-based runtime for clusters of CPUs, GPUs, and other accelerators; it is also documented as a standalone substrate [R43]. Their relevance is resource-aware execution with explicit dependencies and data placement.

This approach provides a natural target for a compiler that already knows its task graph and access privileges. The runtime can manage execution while retaining information about which tasks read or write which regions. A differentiated compiler could use the same structure for residuals and cotangent accumulation.

Such a target does not supply derivative rules automatically. The compiler must still generate the reverse tasks, access modes, and accumulation semantics. In particular, a gradient contribution is often a reduction into a logical region, not an ordinary exclusive overwrite. A correct lowering has to preserve that distinction.

7.3 StarPU and explicit task implementations

StarPU is a heterogeneous task runtime in which tasks can have CPU and GPU implementations, with data movement and scheduling managed through registered data and dependencies [R44]. It represents a mature alternative to placing every concern inside a tensor compiler. The compiler emits tasks and access information; the runtime selects and executes available implementations.

The flexibility is valuable for mixed numerical applications. Some operations may be efficient on GPUs while others are limited by host-side control or have only a CPU implementation. An implementation portfolio allows the runtime to make choices without changing the mathematical program.

The cost is that useful variants, performance models, and derivative tasks still need to exist. A task runtime does not turn an arbitrary opaque callback into an optimizable tensor operator. For cluster metaprogramming, the correct abstraction is a task contract with visible effects and variants, rather than an unconstrained remote function.

7.4 Chapel and distributed language support

Chapel integrates parallelism within nodes, across distributed locales, and on GPUs; its GPU documentation discusses GPU-resident arrays and interactions with remote declarations [R45]. This addresses a language-level part of the problem: distributed ownership and accelerator execution can be expressed without making every remote access an ad hoc RPC.

A language with distributed arrays can reduce orchestration code for scientific applications. It can also serve as a generated target or as the host language around compiled kernels. But language support for remote data and GPUs is not evidence for automatic differentiation or automatic neural-network strategy search.

The broader lesson is that an AI-cluster language need not invent every distributed-programming concept anew. Locales, access privileges, futures, structured concurrency, and region ownership are reusable ideas. The differentiable layer adds a need to model residuals, transposed communication, and accumulation, while the underlying distributed semantics remains necessary.

7.5 IREE as a deployable compiler/runtime boundary

IREE represents both execution logic and scheduling dependencies in its compilation approach, with a hardware abstraction layer and multiple local target backends [R46]. Its C API exposes runtime, device, and synchronization integration points [R47]. These make it relevant as a deployable target for generated numerical regions and as a way to separate model compilation from a Python host process.

It should not be described simply as a universal cluster training system. Available drivers, collective paths, deployment configuration, and frontend differentiation determine what a particular build can do. Support for CPU and GPU backends is a necessary ingredient for heterogeneous execution, not sufficient evidence for arbitrary mixed-device multi-node graphs.

For a compiler architecture, the useful role is a stable executable-region boundary. An outer planner can manage distributed work while a local compiler specializes regions. To make this efficient, buffer ownership, synchronization, and external-call semantics must be shared explicitly across the boundary.

7.6 Apache TVM, Relax, and Disco

TVM's current architecture documentation describes Disco as a distributed runtime using sessions and distributed references, with compiled modules dispatched to workers and collective communication coordinating shards [R48]. Relax's distributed API includes distributed tensor and mesh representations for inference [R49]. The inference scope should remain visible when comparing this stack to training systems.

The design is relevant because a single controller can manipulate distributed objects without directly materializing them on the controller. This can reduce repetitive metaprogramming around worker handles. A distributed reference nevertheless carries placement and lifetime obligations; it is not simply a local tensor whose data happens to be far away.

Training would require the corresponding differentiation, residual, and update contracts. It is possible to use a distributed inference runtime as a component in a training or reinforcement-learning application, but that does not establish that the runtime automatically differentiates every dispatched program. That distinction should guide both implementation and evaluation.

7.7 Ray Compiled Graph

Ray Compiled Graph specializes repeated actor DAG execution and documents GPU-to-GPU tensor transport through NCCL. Its overlap support is identified as experimental in the inspected

documentation [R50]. This reduces runtime overhead and coordination work for repeated distributed workflows.

The important boundary is an actor DAG rather than an arbitrary differentiable tensor language. The actor's internal computation can be separately compiled, and the graph can connect training, inference, and data-processing components. But reverse-mode differentiation across arbitrary actor methods is not supplied merely by compiling their invocation graph.

This suggests a useful two-level architecture: a coarse workflow graph manages services and resource groups, while a finer differentiable graph manages mathematical operations inside a training region. Collapsing the two may create unnecessary restrictions. Keeping them separate requires explicit contracts for tensor ownership, model versions, and transfer completion at the interface.

7.8 Monarch and the programmable cluster

Monarch's April 2026 account describes a Python-controlled model of hosts, processes, and actors, with cluster management, RDMA facilities, and distributed SQL telemetry [R51]. Its maintained release notes document ongoing changes to transport, observability, and lifecycle behavior [R52]. A separate 2026 report describes ROCm support and integration with training and fault-tolerance components on AMD GPUs [R99]. The introductory material also distinguishes its distributed tensor engine from a fully optimized general training interface [R53].

This is highly relevant to simplifying metaprogramming even though it is not itself a universal tensor compiler. Much complexity in AI clusters lies outside the model graph: starting workers, moving code, inspecting state, coordinating training and rollout, and recovering failed components. A coherent cluster API can make these tasks reusable libraries.

The compiler/runtime boundary remains important. A remote actor may host a compiled training step whose internal collectives obey a separate protocol. An outer actor call should not accidentally overlap incompatible communicator operations or release a parameter snapshot before its consumers finish. The orchestration layer needs typed version and lifetime information from the compiled layer.

8 Rust, Julia, and smaller distributed stacks

8.1 Burn and CubeCL in the 2026 snapshot

Burn's inspected 0.22-era documentation describes runtime device selection, several GPU and CPU backends, AD-enabled devices, and remote execution through Iroh or WebSocket transports [R61]. The 0.21 release account introduced redesigned differentiable collectives and improved distributed training foundations [R62]. The 0.22 migration guide records substantial API changes, including removal of backend parameters from model and tensor types [R63].

This is a material development for Rust-oriented work. Older descriptions that reduce Burn to a local backend-generic framework miss the current remote execution and collective direction. Conversely, remote execution and available AD do not prove automatic partitioning of every tensor graph across heterogeneous servers.

The distributed-computing guide explicitly limits current collective support by runtime and describes CUDA all-reduce as the current CubeCL implementation path [R64]. This is the kind of limitation a survey must retain. A client may reach remote devices through a portable tensor API while distributed training remains constrained by the server's collective capabilities. The transport used to submit operations and the transport implementing a collective should be checked separately.

8.2 Rust ownership and GPU kernels

The 2026 cuTile Rust research presents ownership-aware tile-based GPU programming and states a data-race-freedom result for its safe surface under the Tile IR memory model, while retaining bounds-checking obligations [R65]. This is relevant to the safety of generated local kernels and asynchronous host execution.

It does not imply cluster-wide ownership safety or distributed AD. A Rust borrow checker cannot infer the lifetime of a remote memory registration unless the API models it. Nor can it prove that an external collective uses the intended parameter epoch. A cluster runtime written in Rust still needs explicit leases, completion tokens, and cancellation rules.

The opportunity is compositional. The local safe kernel contract can become one premise of a distributed correctness argument. The outer compiler adds network ownership, residual lifetime, and update atomicity. Keeping the premises visible is more credible than claiming that a safe implementation language makes an arbitrary heterogeneous training system safe by construction.

8.3 Reactant and Enzyme in Julia

Reactant compiles Julia functions through MLIR and XLA and integrates Enzyme-powered differentiation [R66]. Its multi-host documentation describes distributed initialization and sharding across nodes [R67]. The 2026 Julia GPU ecosystem account places this approach alongside the array and kernel programming ecosystem and discusses tracing limitations [R69].

The combination is promising for programs that mix numerical algorithms and learned components. Julia can remain the source language while compiler transformations expose larger optimization regions. Multi-host sharding makes the distributed path more direct than manually wrapping every operation in a remote call.

However, the same capability boundaries remain. A function must be representable in the compilation path, each differentiated operation needs support, and a mesh must correspond to a valid backend configuration. The existence of CPU, GPU, and TPU execution paths does not establish one unrestricted heterogeneous mesh across them. Deployment should use the actual documented configuration rather than infer it from a backend list.

8.4 Dagger and distributed task composition

Dagger is a Julia task runtime with GPU support, data movement, distributed scheduling, and documented composition with an MPI backend [R68]. It complements a whole-program compiler: some applications naturally contain independently compiled tasks, dynamic control, or a mix of packages that cannot all be captured into one numerical IR.

Its role should be understood as scheduling and data management rather than automatic differentiation of every task graph. A task can encapsulate a differentiated computation, but crossing a task boundary with a cotangent requires an explicit derivative protocol. Similarly, a transfer that converts data representation needs a defined pullback if it lies in the differentiable program.

The integration opportunity is to let a compiler generate tasks with residual and cotangent access metadata. This can preserve the runtime's flexible scheduling while making the AD dependencies visible. It is a proposed architecture, not a claim of complete out-of-the-box end-to-end integration.

8.5 MLX and a direct TCP example

MLX documents distributed communication for training and inference, including a ring backend using TCP sockets and additional backend choices [R70]. This is a concrete reminder that useful distributed differentiation does not require every installation to have a specialized RDMA fabric.

The hardware scope matters. An Apple-oriented stack is relevant to a cluster of supported Apple systems; it is not evidence for arbitrary cross-vendor GPU mixtures. Likewise, the performance of small collective messages differs from that of large transfers, and latency can dominate even when aggregate bandwidth appears adequate.

For metaprogramming research, such a stack is valuable as a deployment class with an explicit ordinary-network path. It motivates evaluating different communication granularities and batching policies. A compiler should not hide the fact that a sequence of many tiny remote operations can be much slower than one fused local region with a larger boundary transfer.

8.6 llama.cpp RPC and inference boundaries

The llama.cpp RPC backend documents remote devices over TCP and an optional RDMA path [R71]. It demonstrates a practical way to distribute inference across remote resources without treating them as one shared address space.

Its presence in this survey serves a specific purpose: distributed inference feasibility is often confused with distributed training feasibility. Serving a forward graph requires parameter access and activation movement. Training additionally requires residuals, reverse communication, gradient accumulation, optimizer state, and consistent updates. A remote inference backend therefore answers only part of the original question.

The same caution applies to many serving systems. Their batching, cache management, and model-placement methods are useful, but a training compiler cannot simply reverse the serving scheduler. The reverse program has different memory lifetimes and may require communication patterns absent from the forward-only workload.

8.7 A language does not determine the whole stack

Python currently offers the broadest integration with mainstream training systems; Rust offers compelling systems-level and deployment properties; Julia offers a productive numerical language with compiler-based AD and task runtimes; C++ and compiler IR remain essential for low-level integration. These are architectural assessments based on the surveyed capabilities, not a ranking by one benchmark.

The decisive question is which boundaries a project already solves. A Rust service can orchestrate PJRT or C-ABI executables. A Python model can call generated communication-aware kernels. A Julia task can host an MLIR-compiled region. The useful unit of interoperability is the typed executable contract, not the source-language name.

Consequently, the best metaprogramming interface should be multilingual at its boundaries even if its first frontend is not. It should serialize tensor types, shapes, layouts, effects, derivative rules, and executable capabilities in a way that does not require the controller to share the worker's source runtime. That reduces coupling while preserving the information required for correctness.

9 Additional compiler lineages and capability boundaries

9.1 TensorFlow DTensor, OneFlow, and MindSpore

TensorFlow DTensor presents a synchronous global programming model, with sharding directives and multi-client execution; its guide discusses CPU, GPU, and TPU device classes [R90]. This is an important antecedent and parallel lineage to the global-array systems already surveyed. Device-class support should still not be read as an unrestricted single heterogeneous communicator.

OneFlow's global-view model uses placement and SBP signatures, distinguishing split, broadcast, and partial-sum representations and inserting conversions between them [R91]. It provides another concrete instance of the same semantic requirement: equal local shapes do not imply equal logical roles. The documentation is evidence of its programming model, not of a newly demonstrated 2026 benchmark.

MindSpore's current automatic-parallel documentation describes sharding-strategy propagation and recursive strategy search at the distributed-operator level [R92]. It reinforces that automatic parallelization is an established family of approaches rather than one new feature in a single framework. Backend support and operator coverage remain specific to the framework version and deployment.

These lineages matter to a metaprogramming survey because they expose recurring representations: meshes, placements, partial contributions, and redistribution. A common compiler contract should describe these concepts without assuming that their names or precise AD conventions are identical across systems. An adapter must preserve meaning rather than merely translate enum labels.

9.2 Galvatron and SuperScaler

Galvatron studies automatic hybrid parallelism for Transformer training, combining several parallelism dimensions in a model-aware search [R97]. SuperScaler is an experimental parallel-training platform with compiler and runtime components and documented multi-node execution paths [R96]. They illustrate additional points in the space between fixed recipes and unrestricted graph search.

Such projects should be evaluated for their specific purpose and software lineage. A research implementation can establish an algorithmic result while depending on historical framework versions. A maintained production framework can have narrower search expressiveness while supporting a broader deployment ecosystem. Neither axis should be inferred from the other.

For a new compiler, the useful lesson is to preserve search-space definitions and cost-model assumptions as artifacts. That allows later implementations to reuse the research insight without assuming that the original dependency stack remains the right deployment choice.

9.3 Futhark and Dex

Futhark exposes compiler-implemented AD through a functional array-programming interface, including custom reverse rules and documented control-flow limitations [R93]. Dex is a research language for typed array processing, mathematical transformations, and user-directed parallel compilation [R94]. Both are relevant to simplifying numerical metaprogramming at the language level.

They should not be presented here as demonstrated turnkey arbitrary heterogeneous-cluster training systems. Their value is the treatment of source structure, types, transformations, and local parallelism. A cluster layer would need to add the distributed representation and protocol contracts discussed in this paper.

The comparison is useful because a high-level functional language can make AD and fusion easier to reason about, while a general distributed runtime can make remote execution easier to express. Combining the strengths requires an explicit boundary. It does not follow automatically from embedding one language inside the other.

9.4 TileLang and the kernel-language layer

TileLang's ICLR 2026 paper describes explicit tile-level primitives for memory placement, data movement, and parallel scheduling, together with inference and recommendation of tile configura-

tions [R95]. It belongs beside Triton, Helion, and related local kernel systems in the architecture, even when a distributed planner supplies its surrounding communication.

The principal distinction is between local kernel programmability and automatic global distribution. A kernel DSL can dramatically reduce implementation effort for a specialized operator without determining where every operator in a model should run. Conversely, a global planner can use a library of such kernels without exposing the DSL to the model author.

This division supports a layered language design. Most users write tensor programs and strategy constraints. Kernel authors provide specialized implementations with derivative and resource contracts. Runtime authors implement transport and lifecycle semantics. The metaprogramming system composes these contributions instead of requiring every user to become an expert in all three.

9.5 A capability map

The table summarizes the primary abstraction each family contributes. It is intentionally not a Boolean support matrix: exact coverage is version-, operator-, and backend-dependent.

Family	Primary user-visible object	Main reduction in manual work	Boundary still requiring evidence
JAX/OpenXLA, TensorFlow DTensor	Global arrays and layouts	Partitioning and collective insertion	Effects, heterogeneous backend composition, recovery
PyTorch distributed compilation	Captured graphs and distributed types	Integration with existing model code	Capture coverage and communication semantics
Alpa, nnScaler, AutoParallel, Galvatron	Strategy/search representation	Parallelism selection and plan generation	Search-space scope and cost accuracy
Hetu v2, Metis, HexiScale	Asymmetric resource strategies	Heterogeneity-aware allocation	Exact device and network combinations
Piper, JaxPP, Tessel	Training DAG or pipeline schedule	Schedule implementation and composition	Dynamic behavior, storage, epochs
Triton-distributed, DITRON, Syncopate	Distributed tiles and chunks	Communication-aware kernel engineering	Global AD and optimizer semantics
Mirage/MPK, Helion, TileLang	Fused tensor/kernel programs	Local optimization and launch reduction	Training coverage and cluster integration
Legion/Realm, StarPU, Dagger	Tasks, regions, and dependencies	Placement and execution management	Derivative-task generation
Monarch, Ray Compiled Graph	Actors and repeated workflows	Cluster orchestration and coordination	Differentiation across workflow boundaries
Burn, Reactant, Futhark, Dex	Language-level tensor programs	Numerical transformations and deployment paths	End-to-end multi-node capability composition

The map should be read as a composition guide. A project may use one family for source capture, another for local kernels, and another for cluster control. The hard part is not selecting labels from the table; it is preserving types, effects, and derivative meaning across the boundaries.

10 TCP/IP, RDMA, and remote-memory models

10.1 Control traffic and tensor traffic

The statement that a cluster is connected over TCP/IP is incomplete. A system may use TCP for process discovery and a GPU-aware transport for tensor payloads. It may use TCP for both. It may send tensors through host staging buffers even when the source and destination are GPU allocations. It may use a remote-memory model in which operations identify a remote process and an offset rather than exchange serialized tensor messages.

These choices change the executable contract. A graph compiler should identify the path for each tensor edge: direct device-to-device, host-staged, shared-memory, or remote registered-memory. The path determines which buffers must exist, which completion events are required, and which resources can overlap. A launch API that hides all this behind a generic future can be convenient, but the planner still needs the information internally.

NCCL's documentation explicitly distinguishes an internal Socket transport using TCP/IP from its IB path and exposes transport-related settings [R56]. Thus the presence of NCCL does not itself imply one particular network fabric. Performance and compatibility depend on the selected path and deployment. The survey makes no claim that an ordinary Ethernet configuration reproduces results obtained on a high-bandwidth accelerator fabric.

10.2 Host staging is a real part of the graph

Suppose a remote transfer of n bytes uses a device-to-host copy, a network send, and a host-to-device copy. A simple serial estimate is

$$T_{\text{staged}}(n) \approx \sum_{j \in \{\text{D2H},\text{net},\text{H2D}\}} \left(\alpha_j + \frac{n}{\beta_j} \right).$$

This is an explanatory cost model, not a measured result. Pipelining may overlap the terms, and contention or registration overhead may increase them. Its purpose is to prevent the planner from charging only the network transfer while ignoring staging.

Pinned buffers also consume host memory and can have reuse constraints. An asynchronous send cannot reuse its source until the transport has finished reading it. A destination tensor cannot be read until the receiving path has completed and the device sees the data. Treating these events as one unqualified *done* loses information needed for correct buffer reuse.

For heterogeneous CPU/GPU graphs, staging may be unavoidable for some edges. That need not make the design useless. It makes graph granularity important: a large locally compiled region can amortize boundary costs better than a stream of fine-grained remote tensor calls.

10.3 UCX and UCC

UCX provides a communication substrate spanning transports and memory types, including TCP, RDMA, shared memory, CUDA, and ROCm paths. Its documentation describes conditions for GPU-memory RDMA and the need to enable relevant memory-type support [R58]. UCC supplies a collective-communication abstraction with defined memory-type categories [R59].

These are useful substrate interfaces for a multilingual runtime. They reduce the amount of transport-specific code a compiler-generated executable must contain. They do not imply that

every combination of devices and memory types supports every operation, or that all paths have identical ordering and performance.

The compiler should query capabilities and select a legal implementation. If an operation requires a fallback through host memory, that fallback belongs in the cost and lifetime model. A capability failure should be reported at planning time where possible, rather than as an obscure remote runtime error after a large job has started.

10.4 NVSHMEM and PGAS

NVSHMEM exposes an OpenSHMEM-derived partitioned global address-space model for NVIDIA GPU clusters. Its documentation explains symmetric allocations and the ordering roles of fence, quiet, and synchronization operations [R57]. A remote address is meaningful within that model; it is not a promise of ordinary cache-coherent shared memory across the cluster.

PGAS can simplify communication-aware kernels because device code can initiate remote operations. It also makes ordering and buffer ownership more visible. Completion, visibility, and program order must be specified at the appropriate scope. A compiler that moves a consumer across a completion boundary can produce incorrect code even if the corresponding mathematical tensor graph is unchanged.

This is why remote memory should appear as an effect in the IR. A remote write produces an obligation that is discharged by the relevant completion or synchronization operation. Differentiation can transpose the mathematical transfer, but it must also generate a valid execution protocol for the reverse transfer.

10.5 PyTorch communication interfaces

TorchComms presents a modern communication API with attention to heterogeneous systems, observability, and different backend implementations [R54]. PyTorch's Symmetric Memory documentation describes low-level facilities for fused communication/computation and NVSHMEM-backed cross-node capabilities; it also identifies the interface as under development [R55].

These developments reduce the distance between tensor frameworks and communication-aware kernels. They allow implementations to expose more than an opaque all-reduce call. However, a backend list is not a proof of a mixed-vendor execution path. A deployment must verify the exact communicator and memory configuration.

The broader direction is an explicit separation between logical communication semantics and transport implementation. That is the right boundary for a compiler: it can transform a typed reduction or transfer while a backend supplies the supported execution mechanism and completion rules.

10.6 MPI and progress

MPI's nonblocking collective specification makes buffer-use obligations explicit: associated send buffers must not be modified and receive buffers must not be accessed before completion. It also distinguishes completion from assumptions about synchronization among processes [R60]. These are directly relevant to generated asynchronous tensor programs.

Nonblocking syntax alone does not guarantee useful overlap. A runtime may need a progress mechanism, and the device and network must have resources available. Conversely, a computation can be deadlock-free at the tensor-DAG level and still deadlock because a task holds a buffer or execution resource while waiting for communication that needs that resource.

A compiler should therefore model progress assumptions. It can require an independently progressing communication engine, reserve communication resources, or emit a schedule whose waits

occur only when counterpart operations can run. The chosen assumption must be part of the executable's capability contract.

10.7 A simple communication lower bound

Let C be a cut separating resource groups, and let b_C be the number of bytes that a particular execution plan must send through that cut per iteration. If the usable aggregate capacity in the counted direction is at most B_C, then

$$T_{\text{iteration}} \geq \frac{b_C}{B_C}.$$

This is a capacity bound for that plan and information movement, not a universal lower bound for every equivalent algorithm. A rewrite may reduce b_C, recomputation may replace a transfer, or a different optimizer may change what must move. A planner should explain which of these it is doing.

For an idealized homogeneous ring all-reduce of an n-byte tensor on p workers, a common latency-bandwidth model is

$$T_{\text{ring}} \approx 2(p-1)\alpha + 2\frac{p-1}{p}\frac{n}{\beta}.$$

The formula assumes a suitable ring implementation, balanced links, and no extra contention. It is included as an analytical model, not a benchmark. It explains why increasing worker count can make small messages expensive even when aggregate compute grows. Heterogeneous links and host staging require a richer model.

10.8 Ordinary networks favor coarse regions

When network latency is large relative to a local operator, the compiler should generally seek fewer, larger boundaries, subject to memory and concurrency constraints. This is an optimization hypothesis that follows from latency-bandwidth models. It is not a recommendation to fuse all operations: fusion can increase residual storage, delay useful outputs, or reduce parallelism.

A good plan exposes the tradeoff. It can place tightly coupled tensor-parallel work inside a fast local island, move activations between coarser stages, and keep control-oriented CPU work outside the critical tensor path. If a CPU operation lies on the differentiable path, its primal and reverse transfers still need explicit contracts.

The same reasoning explains why remote memory does not eliminate network topology. A convenient address-space abstraction changes how the programmer names communication. It does not remove bandwidth, latency, registration, or visibility costs. The compiler's cost model should retain the physical topology even when the source language presents a uniform view.

11 Weakly connected islands and changed training algorithms

11.1 DiLoCo and the optimizer boundary

DiLoCo reduces communication between relatively weakly connected groups by performing local optimization and less frequent inter-group synchronization [R72]. This addresses a resource regime that fine-grained tensor partitioning may serve poorly. It is relevant to geographically distributed or opportunistically connected compute.

It is not a semantics-preserving lowering of every synchronous training step. Local steps and outer synchronization define a different optimization procedure. A compiler can implement

that procedure correctly, but it should not claim that the resulting parameter trajectory equals ordinary synchronous SGD or an arbitrary original optimizer.

This distinction is central to the user's question. Sometimes the best way to simplify an AI cluster is to choose a training algorithm suited to the network. That is a valid design decision, but the metaprogramming interface should make it a named algorithm choice. Exact graph compilation and optimizer redesign should remain separately reviewable.

11.2 Decoupled DiLoCo

The April 2026 Decoupled DiLoCo preprint describes independent learners communicating parameter fragments to a synchronizer, with quorum, grace-window, and token-weighted merging mechanisms. Its resilience claims include simulated large-scale failure scenarios as well as model evaluations [R73]. Simulated chip counts must not be reported as a physical deployment at that scale.

The contribution is a move beyond lock-step synchronization at the outer training level. Such a system can continue useful work when learners are slow or unavailable, but the semantics includes asynchronous aggregation and its weighting policy. Correctness means implementing that policy and preserving its stated invariants, not reproducing a single fixed synchronous execution.

For a compiler/runtime interface, this suggests two distinct modules. One compiles the local differentiated learner. The other implements the distributed optimizer protocol with explicit versions, update identities, and quorum rules. This separation permits each to evolve without pretending that network failure is a differentiable tensor primitive.

11.3 A taxonomy of equivalence claims

We recommend distinguishing exact real-arithmetic equivalence, permitted floating-point variation, bounded numerical approximation, optimizer-level equivalence, and empirical training-quality similarity. These are different relations. Only the first is straightforward equality of the denoted smooth maps. The others need their own observations, bounds, or evaluation criteria.

For example, changing the reduction tree can change floating-point results while preserving an exact real-arithmetic expression. Quantizing communicated gradients introduces an approximation. Taking several local optimizer steps changes the update map. Reporting similar validation loss is an empirical result about a training process. No one of these statements implies the others.

This taxonomy makes metaprogramming safer and easier to review. A generator can state which relation it intends to preserve, and a checker can enforce the corresponding structural obligations. Empirical claims remain empirical. A categorical presentation should preserve these distinctions rather than force all transformations into one equivalence relation.

12 Comparative evidence and reported performance

12.1 Reading the results without creating a false leaderboard

The following table records selected recent results already discussed. Every number is reported by the cited source. Hardware, model, baseline, and metric differ, so the rows must not be multiplied, averaged, or ranked as if they were measurements of one common task.

Work	Evidence snapshot	Reported result	Scope that limits interpretation
Hetu v2 [R24]	OSDI 2026 paper	Asymmetric and hierarchical SPMD evaluation	Supported heterogeneous, unstable-device, and mixed-length scenarios
Tessera [R25]	OSDI 2026 operational paper	20-33% throughput improvement over production baseline	Five reported production workloads; large MoE clusters
DITRON [R30]	ICML 2026 paper	6-30% kernel and 5-30% inference improvements	Evaluated kernels and vLLM configurations
Syncopate [R31]	OSDI 2026 paper	1.3 times average; up to 4.7 times end-to-end speedup	Evaluated multi-GPU workloads and baselines
MPK [R28]	OSDI 2026 paper	Up to 1.7 times lower inference latency	Inference; not a reported general training compiler
`spmd_types` [R15]	August 2026 engineering report	Up to 46% eager and 5-9% Inductor throughput gains	Stated debug-model benchmarks versus `partial_dtensor`
AutoParallel [R17] [R88]	Experimental code and conference account	Example matching a manual baseline with 12% less memory	Llama-3 8B example; announcement-level benchmark evidence
Piper [R26]	June 2026 preprint	Strategy/runtime separation with evaluated composed schedules	Preprint; no universal device or graph claim

The strongest general conclusion is qualitative: the optimization boundary is expanding. Current systems increasingly expose backward distribution, overlap, materialization, and schedule structure. The table does not establish which system wins on a particular user's cluster. That requires a common workload, pinned versions, a measured topology, and a common objective.

12.2 What the literature does not yet establish

Within the inspected evidence, no single demonstrated stack establishes all of the following simultaneously: arbitrary supported higher-order tensor programs, arbitrary effectful foreign calls, automatic exact adjoints, unrestricted CPU/GPU and cross-vendor mixing, automatic network-aware placement, optimal scheduling, transparent recovery, and performance portability on every TCP or RDMA network.

This is an evidence-limited statement, not a mathematical impossibility theorem. Several components are already practical. The gap is their composition with precise contracts and a shared representation. It would be a mistake either to deny the progress or to erase the remaining boundaries.

The following semantic development is intended to describe those boundaries precisely enough that future systems can compose them. It does not claim to formalize the internals of every surveyed implementation. It supplies a small common vocabulary in which such implementations can state and check their obligations.

13 A representation semantics for distributed adjoints

13.1 The mathematical setting and its limits

We now give our own semantic synthesis. Unless stated otherwise, tensor values inhabit finite-dimensional real vector spaces, primitive maps are continuously differentiable on the domains in question, and arithmetic is exact. Shapes and mesh sizes are fixed within a compiled specialization. Random keys, parameter versions, and discrete routing choices are explicit inputs or are held fixed while differentiating. Floating-point execution is discussed separately below.

These assumptions define a useful core; they do not cover every operation in a practical training framework. Sparse structure changes, top-k routing, clipping boundaries, quantization, foreign state, and dynamic allocation require additional rules. The purpose of the core is to isolate what can be proved compositionally and to make the extension obligations visible.

Reverse derivative categories provide an established categorical axiomatization of reverse differentiation [R74]. We use ordinary finite-dimensional smooth maps for the concrete derivations, avoiding a claim that every topos or higher category automatically carries such an operation. The categorical terminology organizes the result after the required differential structure has been supplied.

13.2 Logical tensors and physical representations

Let X be a logical tensor space and D_ℓ the product of local storage spaces for layout ℓ. A representation consists of linear maps

$$E_\ell : X \to D_\ell, \qquad Q_\ell : D_\ell \to X, \qquad Q_\ell E_\ell = 1_X.$$

E_ℓ is a chosen encoding and Q_ℓ is decoding. The image of E_ℓ is a canonical representation, but the decoder can admit additional physical states with the same logical meaning. Disjoint sharding, replication, and partial contributions fit different choices of these maps. The choice is part of the semantics, not something determined by local shape alone.

For disjoint shards, decoding concatenates pieces. For replication, encoding repeats a value, while decoding can select a replica or average compatible replicas. For partial contributions, decoding sums contributions; a chosen encoding may place the whole value in one component or split it by fixed weights. A physical partial-contribution state need not equal that chosen encoding. This distinction prevents us from identifying every valid representation with one preferred storage pattern.

If replica consistency is required, it can be stated as an invariant on the valid states. If arbitrary contributions are allowed, the valid states may instead be a whole fiber of Q_ℓ. A compiler checker should know which interpretation applies. Replacing one representation by another can preserve the logical value while changing the valid-state invariant and its derivative interpretation.

13.3 Representation-preserving differentiation

Proposition 1 (pullback through representation). Let $P : D_\ell \to D_m$ be a continuously differentiable physical program, and suppose that, on an open domain of logical inputs,

$$Q_m \circ P \circ E_\ell = f.$$

Then, at every input x in that domain,

$$Df(x)^* = E_\ell^* \circ DP(E_\ell x)^* \circ Q_m^*.$$

Proof. Differentiate the equality and use linearity of encoding and decoding:

$$Df(x) = Q_m \circ DP(E_\ell x) \circ E_\ell.$$

Taking transposes reverses composition, giving the stated formula. Equivalently, pair both sides with an arbitrary input direction and output cotangent. Nondegeneracy of the pairings identifies the resulting maps. □

The proposition is elementary but operationally important. A logical output cotangent is lifted with Q_m^*, and the physical input cotangent is decoded with E_ℓ^*. Reusing the primal encoder and decoder in the same direction is generally wrong. The statement also requires equality on a domain, not equality on a finite test set.

The formula does not claim that all physical cotangent buffers have one canonical representation. Components normal to the encoded logical subspace can occur. What matters for the logical derivative is their image under E_ℓ^*. A concrete type system may choose canonical cotangent representations and insert collectives to maintain them. Another may retain partial contributions longer. Both need to satisfy an equivalent logical contract.

13.4 Why replication causes reductions

For a real vector space X with its usual pairing, define replication and summation by

$$\Delta_p(x) = (x, \dots, x), \qquad \Sigma_p(x_1, \dots, x_p) = \sum_i x_i.$$

With the product pairing on X^p,

$$\langle \Delta_p x, (\bar{x}_i)_i \rangle = \sum_i \langle x, \bar{x}_i \rangle = \left\langle x, \sum_i \bar{x}_i \right\rangle.$$

Therefore $\Delta_p^* = \Sigma_p$ and $\Sigma_p^* = \Delta_p$. If independent computations consume copies of one logical parameter, their cotangent contributions must be summed. If the forward operation is a mean, its transpose includes the same normalization factor. No convention about averaging gradients should be implicit.

This derivation explains a source of distributed bugs. Several identical physical copies can denote one logical parameter or several independent parameters constrained to happen to agree. Their derivatives are different problems. The layout type must identify the intended logical object, not merely record that the bytes currently match.

13.5 The transposes of collectives

Let local spaces X_i form a direct sum, and let concatenation identify that sum with a global tensor space Y. An all-gather maps local shards to a copy of the concatenated tensor on each participant. Its transpose takes every output cotangent, selects the block corresponding to each input shard, and sums those blocks across participants. This is a sum reduce-scatter under the stated representation.

For equal local spaces, a sum all-reduce is

$$A = \Delta_p \Sigma_p : X^p \to X^p.$$

It is self-transpose because $A^* = \Sigma_p^* \Delta_p^* = \Delta_p \Sigma_p = A$. A mean all-reduce is A/p and is also self-transpose with that scaling. These statements concern independent physical inputs and

cotangents under the product pairing. A framework's logical replicated type can induce a different-looking API rule because its encoding and decoding maps have already accounted for replication.

Permutation communication transposes to the inverse permutation. A gather with repeated indices transposes to scatter-add, not to an overwrite. An all-to-all transpose reverses the corresponding block permutation. A point-to-point copy has a reverse information path, but the implementation must also preserve who owns the buffers and when they become readable.

Proposition 2 (collective transposition). Any finite composition of the preceding linear collective maps has a pullback obtained by reversing composition and transposing each primitive, including its scaling and index map.

Proof. For linear maps B and C, the pairing identity gives $(CB)^* = B^*C^*$. Induction on the expression proves the claim. The tensor-product case follows from the product pairing. □

This proposition supplies mathematical derivative rules, not a transport implementation. A generated reverse collective must still use compatible participant groups, message identities, and synchronization. A transpose of an index map cannot repair a mismatched communicator.

13.6 Legal deferral of reductions

Suppose u_i are contributions to one logical value and L is linear. Then

$$L\left(\sum_i u_i\right) = \sum_i L(u_i).$$

Thus summation can move across L, provided the physical program preserves the same inputs and normalization. This justifies many deferred-reduction optimizations. It does not justify moving a reduction across an arbitrary nonlinear primitive: in general $\sigma(\sum_i u_i) \neq \sum_i \sigma(u_i)$.

The compiler should therefore distinguish linear consumers of pending contributions. Adding contributions, multiplying by a fixed linear map, and rearranging indices have different status from nonlinear activation, data-dependent routing, or multiplication of two pending sums. A type system can enforce this distinction, and a planner can exploit it to batch communication.

Floating-point reassociation changes the exact numerical statement. The real-arithmetic proof remains a specification-level argument, while a numerical policy decides whether the altered reduction order is permitted. The transformation record should say which policy justified it.

13.7 Residuals compose as reverse programs

For $f : X \to Y$ and $g : Y \to Z$, an augmented forward execution can retain r_f and r_g. Its reverse computation applies the pullback of g and then the pullback of f:

$$(g \circ f)^*_x(\bar{z}) = f^*_x(g^*_{f(x)}(\bar{z})).$$

The residual representation may be a product, a closure, a tape, or a rematerialization recipe. The logical equation does not require one particular implementation. It does require that the residual correspond to the same primal execution.

This gives a useful lens-like presentation: a forward map produces an output and a residual; a backward map consumes that residual and an output cotangent. Composition combines residuals and reverses the backward direction. Calling this a lens-like structure does not imply every law of an unrelated lens formalism. The intended laws are exactly the chain rule, pairing compatibility, and the stated residual invariants.

In a distributed compiler, residual placement becomes part of the morphism's implementation. A residual can move across the network or be reconstructed, but its logical meaning and parameter epoch must remain fixed. This is where differentiation, memory planning, and scheduling meet.

13.8 Higher derivatives and linearization boundaries

Differentiating a program that computes a pullback introduces higher derivatives and additional residual dependencies. It is not enough to transpose the original forward communication once more without tracking which values the first pullback depends on. Custom reverse rules also need higher-order support if they are to participate in nested AD.

The higher-order reverse chain rule has a categorical treatment in recent work on reverse Faà di Bruno formulas [R89]. That work supplies a mathematical account of iterated reverse differentiation; it does not establish distributed implementations for arbitrary runtime effects.

For a compiler specification, each primitive should advertise its differentiation depth and supported modes. A first-order custom VJP may be perfectly adequate for ordinary training while unsuitable for hypergradient computation. Such a limitation is a capability boundary, not a defect. Making it explicit prevents a metaprogram from silently composing unsupported transformations.

13.9 Approximate forward equality is insufficient

Proposition 3 (no derivative bound from value error alone). A uniform bound on function-value error does not imply a uniform bound on derivative error, even for smooth scalar functions.

Proof. For $\varepsilon > 0$, let $f(x) = 0$ and $g_\varepsilon(x) = \varepsilon \sin(x/\varepsilon^2)$. Then $|g_\varepsilon(x) - f(x)| \le \varepsilon$ for every real x, while $g'_\varepsilon(0) = 1/\varepsilon$ and $f'(0) = 0$. The derivative error is unbounded as $\varepsilon \to 0$. □

Consequently, an approximate primal compiler needs a derivative-specific contract if it claims gradient accuracy. Quantization, approximate kernels, communication compression, and surrogate differentiation cannot be justified solely by small forward error. Suitable assumptions might bound derivatives of the approximation, establish a backward-error interpretation, or evaluate a specific training-quality criterion. Those are additional premises.

This is an important limit on overbroad compiler correctness claims. Exact semantic preservation can transport a derivative theorem through the chain rule. Approximate preservation requires a richer analysis. A higher categorical notation does not remove that distinction.

14 Execution plans as a higher categorical structure

14.1 Event structures and interfaces

Fix a finite compiled specialization. An execution plan consists of typed compute and communication events, a precedence relation, buffer-access information, resource assignments, and externally visible input/output ports. Parameter and residual ports carry epochs. Communication events carry participant groups and completion obligations. We assume that message payload semantics and primitive kernels are specified independently.

Plans compose by connecting compatible ports and introducing the necessary dependency edges. Independent plans can be placed side by side when their resources and effects are separated or a resource-sharing contract is supplied. Renaming internal event and buffer identifiers gives the usual coherence isomorphisms. This yields a monoidal bicategorical presentation once the chosen class of rewrites is closed under composition and context.

The objects are typed interfaces, the 1-morphisms are plans, and the 2-morphisms are justified transformations between plans. Transformations may preserve exact denotation, a numerical

policy, or a directed refinement relation; these classes should be labeled separately. A directed optimization should not be turned into an equivalence merely because it improves one cost estimate.

14.2 A schedule-equivalence theorem

Proposition 4 (independent-event reordering). Consider a finite acyclic event graph with deterministic events. Assume that incomparable events have disjoint writes, no conflicting read/write accesses, no hidden effects, and no shared ordering-sensitive reduction. Assume communication is represented by explicit events and immutable delivered values. Then every topological ordering yields the same visible output.

Proof. Any two linear extensions of a finite partial order are connected by swaps of adjacent incomparable events. Under the access and effect assumptions, each such pair commutes: neither changes an input or visible effect used by the other. Therefore each swap preserves the resulting state on all visible ports. Composing the swaps proves equality of final outputs. □

The theorem justifies a class of schedule transformations. It does not prove liveness under bounded queues, GPU resource exhaustion, or a transport without progress. It also deliberately excludes reordering conflicting floating-point reductions. Those require additional policies and arguments.

This illustrates the role of 2-morphisms. A swap is accompanied by a local independence witness. Larger transformations can compose such witnesses. A planner need not rediscover a global proof from scratch for every schedule, but the primitive side conditions must be checkable.

14.3 Costs are annotations, not equalities

A plan has several costs: work, makespan, communication volume, peak memory, energy, compilation time, and recovery overhead. These do not all compose additively. Sequential execution may add time; concurrent execution may take a maximum only when resources do not contend; peak memory depends on overlapping lifetimes. A scalar cost functor that ignores these distinctions would be misleading.

We therefore attach resource-sensitive estimates to plans rather than identify semantic equivalence with equal cost. A semantics-preserving rewrite can improve one cost and worsen another. An optimizer chooses among equivalent or policy-related plans under an explicit objective and constraints.

The distinction also matters for higher coherence. Two transformation sequences may end at equivalent programs while producing different profiling metadata or search histories. Their computational meaning can agree without their optimization procedures being identical. A higher cell can record agreement at the chosen observational level without asserting equality of every artifact.

14.4 Layout change as a double-category intuition

There are two natural directions of transformation. A horizontal arrow is a computation. A vertical arrow changes a representation of its input or output. A square states that the implementation and representation changes commute after decoding. This is a useful double-category viewpoint on distributed compilation.

For chosen encodings and decoders, a canonical re-encoding is $R_{\ell m} = E_m Q_\ell$. Because $Q_m E_m = 1$, these maps satisfy

$$R_{mn} R_{\ell m} = R_{\ell n}.$$

However, $R_{\ell\ell} = E_\ell Q_\ell$ is a projection on the ambient physical space, not necessarily its identity. It becomes the identity on canonical encoded states. This is precisely why we do not casually declare all physical layouts to form a groupoid of isomorphisms.

Real resharding implementations may preserve a larger valid-state relation rather than use the canonical section. They may also expose different asynchronous behavior. Their squares should state the relation actually preserved. The abstract equation gives a specification against which implementation paths can be checked.

14.5 What the higher structure contributes

Higher categorical language is useful when transformations themselves have transformations: different sequences of reshards, fusions, AD passes, and schedule rewrites can be compared. Coherence asks whether those comparisons agree on larger compositions. It provides a disciplined way to state pass-order independence rather than assume it.

For a fixed plan, invertible independent-event swaps generate a groupoid of schedule presentations. Its nerve is a space recording those presentations and their identifications. Directed optimizations should remain outside that groupoid unless they have actual inverses at the chosen observational level. General higher-category references provide the surrounding language [R80].

No performance theorem follows from calling this space an infinity-groupoid. The benefit is organizational: transformation paths, witnesses, and coherences can be represented explicitly. A practical compiler may implement only the 1- and 2-dimensional part. That can already be useful and should not be presented as a failure to realize an unnecessary infinite hierarchy.

15 Local execution records, sheaves, and descent

15.1 A site of dependency-closed contexts

Fix a finite event DAG E, extended with explicit input events. Let $\mathcal{J}(E)$ be the poset of downward-closed subsets of events. An element U is an execution context containing every predecessor of each of its events. A family $\{U_i \subseteq U\}$ covers U when its union is U. Intersections are downward closed, so these covers define the usual open-cover site of the associated finite Alexandrov space.

For each context U, define $\mathcal{H}(U)$ to be assignments of values to the ports in U that satisfy every primitive equation whose event lies in U. Inputs are explicit ports with no defining primitive equation. All state needed to make an event deterministic, including random keys and parameter epochs, is included among its predecessors. Restriction forgets the values outside a smaller context.

This construction concerns logical execution records. It does not assume that the records have already been produced by a live distributed runtime. It can be used to describe a specification, a trace checker, or a partial execution. That distinction is necessary before making a local-to-global claim.

15.2 A precise gluing result

Proposition 5 (sheaf of compatible histories). With the preceding definition, $\mathcal{H}$ is a sheaf of sets on $\mathcal{J}(E)$.

Proof. Let $\{U_i\}$ cover U, and let $h_i \in \mathcal{H}(U_i)$ agree after restriction to every intersection. Each port in U occurs in at least one U_i. Assign it the value supplied by such an h_i. Agreement on intersections makes this independent of the choice. For an event $e \in U$, choose U_i containing e. Downward closure places all its predecessors in that same U_i, so its primitive equation holds in

the glued assignment. Thus the assignment lies in $\mathcal{H}(U)$. Any other gluing has the same value at every port and is equal to it. □

The assumptions carry the content. If local records omit a required dependency, use different epochs, or disagree on overlapping data, the proposition does not apply. A family of independently plausible worker logs is not automatically a matching family. Likewise, overlapping storage addresses are not the same thing as overlapping logical contexts.

This gives a limited but useful interpretation of distributed consistency. A global logical history can be reconstructed from compatible local histories when all constraints are local to dependency-closed contexts. It does not establish termination, availability, or physical memory coherence.

15.3 Why the cover should not simply be the device partition

A device's events need not form a downward-closed context. A GPU consumer may depend on a CPU-produced value or a remote shard. Covering a computation by device ownership alone can therefore omit the very equations that couple the devices.

One can instead enrich a local context with the relevant boundary ports and transfer certificates, or use dependency-closed contexts as above. Either choice makes cross-device information explicit. The site should be selected to reflect the intended locality of constraints, not merely the physical arrangement of machines.

This is an instructive use of topos theory. The mathematical machinery does not declare that a distributed system is consistent; it clarifies which local data and overlap conditions would make a consistency statement true. A proposed formalization should present those choices before invoking descent.

15.4 From sets to spaces

The category $\mathrm{Sh}(\mathcal{J}(E), \mathbf{Set})$ is a Grothendieck topos. If local data is replaced by spaces together with coherent equivalences, the corresponding category of space-valued sheaves gives an infinity-topos. Higher topos theory provides the general framework for such descent constructions [R79].

For schedules, a space-valued record could retain multiple equivalent execution presentations and transformations between them. Gluing then requires coherent agreement on overlaps, not merely pairwise equality. This is a proposed extension of the simple history construction. We do not claim here that an arbitrary distributed runtime's trace space satisfies the required descent condition.

The higher formulation is useful when proof witnesses matter, but it should not erase operational distinctions. A refinement that drops information or changes an approximation policy is not automatically an equivalence in a local space. Such directed behavior may need a category-valued or higher-categorical object rather than a sheaf of groupoids.

15.5 Smooth semantics is additional structure

An arbitrary topos does not automatically provide the intended calculus of real tensor derivatives. One must choose suitable objects, maps, and differential structure. The concrete proofs above use external finite-dimensional smooth maps. A richer semantic project could combine smooth parameter contexts with execution contexts, but it would need to specify that site and its differential interpretation.

The presence of a natural number object supports internal arithmetic and iteration specifications; it does not imply unbounded physical storage or the termination of every compiled loop. Similarly, an infinity-topos provides coherent gluing language, not a GPU scheduling algorithm. These are structural tools whose operational meaning must be supplied by an interpretation.

For the present survey, the useful conclusion is modest and precise: layout, derivative, and execution correctness can be organized as local contracts with compatibility conditions. A topos-theoretic account makes those compatibility questions explicit. It should complement, rather than replace, the finite-dimensional derivative proofs and runtime checks.

16 Effects, indexed monads, and metaprogramming

16.1 Why pure tensor syntax is not enough

A complete cluster program allocates buffers, launches kernels, sends data, waits for completion, publishes model versions, and commits checkpoints. These operations have effects. Treating them all as ordinary pure tensor functions hides resource ownership and sequencing obligations.

A proposed effect signature can include allocation, device launch, message initiation, receive completion, lease release, actor spawn, and checkpoint commit. Each operation has a typed result and a transition on resource state. For example, initiating a send changes the status of a source buffer from reusable to in-flight; completing it can release that restriction. Reading a received tensor requires evidence that the destination is ready.

Algebraic effects provide an established way to separate operations from their handlers [R78]. Here the important handlers are concrete execution mechanisms: a TCP handler, a registered-memory handler, a simulator, or a trace validator. The same logical operation can have different implementations if each handler satisfies the required equations and ordering contract.

16.2 An indexed computation interface

Let s describe available resources, leases, and epochs. Write

$$T_{s,t}(A)$$

for a computation returning an A while changing state from s to t. Sequencing has the form

$$T_{s,t}(A) \times (A \to T_{t,u}(B)) \longrightarrow T_{s,u}(B),$$

with a unit at $T_{s,s}(A)$. This is an indexed-monadic specification pattern. It is more informative than an unindexed monad that forgets whether buffers are ready or leases are outstanding. A practical language may approximate it with affine types, effect rows, typestate, or runtime checks.

The equations for sequencing describe associativity and units of legal composition. They do not license reordering effects. Independent operations may commute only when their resource and protocol conditions permit it. A remote write followed by a read has a different contract from two independent local launches.

This formulation also clarifies cancellation. Cancellation is a transition with cleanup obligations, not simply disappearance of a future. It must say whether remote operations can still complete and when their buffers may be reclaimed. A compiler-generated destructor cannot safely assume that a network peer stopped using a buffer merely because the local task was cancelled.

16.3 Differentiation and effects need a compatibility law

One might wish to combine a differentiation transformation D with an effect interpretation T through a distributive law or a related compatibility structure. Such a law cannot simply be asserted for arbitrary effects. A stateful foreign call may have no derivative; a message retry may duplicate an accumulation; a clock read may not belong to a smooth semantics.

The safe approach is to define a differentiable subset and prove compatibility for its primitives. A send of an immutable mathematical value can be represented as transport plus explicit dependency. Its reverse rule transports a cotangent in the opposite direction. An optimizer commit is modeled separately as a state transition. A retry is checked against an idempotency policy rather than differentiated as if it were arithmetic.

This is the main reason to distinguish the mathematical graph from the execution protocol. Their interaction can be specified compositionally, but they are not the same object. A higher monad, understood as a coherently associative monad object in a specified higher-categorical ambient structure, would organize such compatibility data. It would not create that data for unsupported effects.

16.4 A proposed differentiable core language

A suitable core language has tensor types, products, functions, structured loops, explicit random keys, and typed foreign operators. Distribution annotations refine tensor types; resource effects refine computations. A schematic judgment is

$$\Gamma; \mathcal{R} \vdash e : \mathrm{Tensor}(\tau, \mathbf{n}, \ell, \rho, \nu) \,!\, \epsilon,$$

where τ is an element type, $\mathbf{n}$ a shape, ℓ a layout, ρ a contribution role, ν an epoch, and ϵ an effect description. $\mathcal{R}$ records resource capabilities. This is a proposed notation, not an existing parser or executable implementation.

The AD transformation returns an augmented forward computation and a linear pullback closure. The closure's type records which residuals and parameter epochs it captures. A distribution pass may move those captures or replace them with rematerialization recipes, provided it preserves their meaning. A scheduling pass may then reorder independent events with explicit witnesses.

Higher-order functions are useful here because strategies can be functions from a model region and a resource description to a checked plan. But higher-order source syntax should not force the runtime to execute arbitrary closures remotely. Closure conversion can produce a finite code artifact plus typed captured data. The resulting serialization contract is part of compilation.

16.5 What an AI-assisted metaprogram should generate

An AI-assisted tool should generate a proposed graph transformation, a schedule, or a strategy specification together with declared assumptions. It should not be the sole authority on whether the generated distributed program is legal. A deterministic checker can validate types, supported primitives, buffer lifetimes, collective participation, and epoch consistency.

This separation improves iteration. The generator can explore novel schedules while the checker rejects invalid ones before expensive cluster execution. A profiler evaluates feasible candidates, and the search retains evidence about cost estimates and observed behavior. The controller can explain why a candidate was rejected or why a slower-looking local kernel won globally.

The same architecture supports human-written strategies and conventional solvers. Nothing in the semantic contract requires the strategy to originate from an AI model. That is desirable: the reusable object is the checked plan, not the identity of its authoring tool.

17 A proposed compiler/runtime contract

17.1 Separate the model, the strategy, and the deployment

We propose three independently versioned artifacts. The model artifact contains mathematical operations, shapes, differentiation rules, and effect declarations. The strategy artifact contains

layouts, partitions, collective structure, residual policy, and schedules. The deployment artifact maps abstract resources to actual devices, transports, runtime versions, and storage services.

This separation reduces repeated metaprogramming. A model can be retargeted to another cluster by changing deployment and perhaps strategy. A strategy can be tuned without rewriting model code. A runtime can improve a transport implementation while retaining the same logical communication contract. The separation also makes review more concrete: a change to the optimizer is visibly different from a change to a network route.

The artifacts must remain linked. A strategy names the exact model and derivative schema it was checked against. A deployment names the capabilities that justify the strategy. A compiled executable records those identities. Reusing an artifact after an incompatible change should fail a compatibility check rather than silently run a different program.

17.2 Capability discovery

A resource description should record device architecture, supported element types, memory capacity, available kernels, AD support, peer-access paths, transport backends, progress behavior, and failure/reconfiguration capabilities. Link descriptions should include directionality, bandwidth estimates, latency distributions, and sharing relationships. A flat count of GPUs is insufficient.

Capabilities need a distinction between availability and evidence. A kernel may exist but have no measured cost for a particular shape. A transport may support host memory while lacking a direct path for one device type. A custom operator may have a first-order pullback but no higher-order rule. The planner should either use a supported fallback or reject the plan with a clear reason.

This is a practical place for a machine-readable schema. It allows a metaprogram to ask whether a proposed layout is legal before compiling expensive kernels. It also prevents a source-language frontend from hard-coding assumptions about all future hardware. The schema is a proposal; this paper does not claim to have implemented or standardized it.

17.3 A pass pipeline with explicit obligations

The frontend first captures a typed graph and identifies unsupported effects. The AD pass constructs augmented-forward and reverse programs with explicit residual dependencies. A distribution pass selects logical layouts and contribution roles. A planner chooses device groups, operator variants, microbatch structure, and memory policies. A schedule checker validates the event plan. Backend compilers generate local code, and a deployment linker binds transports and resource identities.

Feedback can travel backward. A kernel compiler may report a larger storage requirement than estimated. A profiler may reveal a slow link. A collective backend may reject a particular device mixture. The planner can revise its strategy, but each revision is rechecked against the same semantic contract.

The critical requirement is that unsupported cases remain visible. A fallback to host execution may be acceptable, but it changes cost and transfer structure. A fallback to a surrogate gradient changes the derivative contract. A fallback to asynchronous local training changes the optimizer. These should not be treated as equivalent implementation details.

17.4 A schedule certificate

A schedule certificate should identify the graph being implemented, the types and epochs of its ports, the dependency order, buffer lifetimes, collective participant sets, and the intended numerical policy. It can contain local rewrite witnesses rather than a full formal proof of every kernel instruction. The checker validates what it can and records the trusted backend assumptions.

For a collective, the certificate includes compatible operation order, group membership, shapes, element types, and normalization. For an asynchronous transfer, it includes source stability, destination readiness, and release conditions. For a rematerialized residual, it includes the inputs and parameter epoch required to recompute it. For an optimizer step, it includes the set of gradient contributions expected before commit.

A certificate need not be enormous. Many facts can be derived from typed SSA values and a compact event graph. The point is to preserve the facts across compiler boundaries. A collection of unstructured callbacks makes those facts hard to recover after a failure.

17.5 Exact accumulation and update commits

Each gradient contribution should have an identity tied to the model version, step, microbatch, parameter partition, and producing region. Exact-once accumulation means that the logical commit consumes each required contribution once. The runtime may achieve this through fail-stop execution, deduplication, or replay from a checkpoint, but the policy must be defined.

An optimizer commit should publish a new parameter epoch only after its required state transition is complete. The corresponding optimizer state, random-state position, and data-order state must be included in the recovery contract when reproducing the original training process matters. A checkpoint need not preserve every transient buffer, but it must preserve or reconstruct the logical state it claims to resume.

This does not require a distributed transaction for every tensor operation. Coarse step boundaries can be enough. The correct granularity depends on the failure model and algorithm. A tightly coupled job may restart a whole step; an asynchronous learner protocol may commit independent fragments. Those choices should be explicit modules of the system.

17.6 A nontrivial training region

Consider a differentiable region with a residual block, a learned gate, and a shared projection:

$$
\begin{aligned}
h &= \sigma(XW_1 + b_1), \qquad r = XW_r, \\
g &= \operatorname{sigmoid}(XW_g + b_g), \qquad Y = (hW_2) \odot g + r, \\
\mathcal{L} &= \operatorname{loss}(Y, T) + \lambda \Omega(W_1, W_2, W_r, W_g).
\end{aligned}
$$

The purpose is a specification-level example, not an executable benchmark. It includes nonlinear operations, branching, elementwise interaction, shared input dependence, and parameter regularization. An implementation may shard the batch, partition hidden channels, and use a different layout for the residual branch.

The pullback for X receives contributions through the nonlinear branch, the gate branch, and the residual branch. Those contributions must be combined according to the logical input representation. If the input or parameters are replicated, local gradients may be pending sums. The regularizer's normalization must be specified so it is not counted once per worker unintentionally.

Residual planning must retain or reconstruct the activation and gate information needed by the reverse pass. If a branch runs on a CPU and the other on a GPU, the forward merge and its reverse split introduce transfers. A graph compiler can automate those transfers only if the branch interfaces and derivative rules are visible. A remote actor call hiding the whole branch would need an explicit derivative endpoint.

This region illustrates why arbitrary DAG support matters. A chain-only pipeline representation may serialize independent branches or obscure the accumulation point. A graph-based schedule can expose overlap while preserving the sum of reverse contributions. The representation and epoch conditions from the preceding sections state what that preservation means.

17.7 Rust as an execution shell

A Rust execution service can own compiled modules, transport endpoints, buffer leases, and step-commit state. It can expose C-ABI, PJRT, or another explicitly specified executable interface. The frontend and local compiler need not both be written in Rust. This architecture uses Rust where ownership and lifecycle control are useful without requiring a rewrite of every numerical compiler.

The service should distinguish host buffers, device buffers, and remote handles at the type/API level. A remote handle contains an owner, allocation generation, shape, type, and access capability. Completion produces a usable state transition rather than merely a Boolean flag. Destruction of a handle must respect outstanding remote access.

This is a proposed implementation path. Burn provides a more integrated Rust tensor and AD route within its supported capabilities; a service around existing compiler artifacts provides another route. The choice depends on operator coverage, required transports, interoperability, and maintenance, rather than on a claim that one language solves the whole problem.

18 Evaluation requirements for the next generation

18.1 Workload coverage

A credible evaluation should include more than one dense Transformer. It should cover a branched residual network, a sparse or irregular tensor workload, a mixture-of-experts region, structured dynamic control flow, a CPU/GPU mixed numerical application, and a program with custom differentiated operators. If higher-order AD is claimed, include a hypergradient or Hessian-vector workload through the supported training region.

The point is not to maximize benchmark count. Each workload should test a specific claimed capability. A branch tests graph scheduling; sparse structure tests representation; custom operators test extension contracts; mixed host/device code tests the differentiation boundary; dynamic shapes test specialization policy; failure injection tests the recovery contract.

Inference-only optimizations should be evaluated separately from training. A fast forward path can be valuable while requiring an inefficient or unsupported backward path. Reporting both without separation obscures the central question about adjoints.

18.2 Hardware and network coverage

At minimum, a heterogeneous claim should state whether it means different generations of one vendor, multiple vendors in separate jobs, multiple vendors in one job, CPU and GPU computation in one graph, or merely portable source code. These are distinct experiments.

The network evaluation should identify payload paths and include ordinary TCP/IP if that deployment is claimed. It should measure small-message latency, sustained bandwidth, staging costs, and interference under overlap. A topology diagram or machine-readable resource graph should accompany the results. The study should not infer tensor-payload behavior from the control-plane transport.

For memory-over-network models, report registration and allocation overhead, completion semantics, and buffer-lifetime pressure. For weakly connected islands, report the actual optimizer and synchronization policy. That makes it possible to compare systems without pretending that all use the same algorithm.

18.3 Semantic and numerical checks

Validation should compare the logical primal output and the pullback under the declared layout interpretation. Pairing checks of the form

$$\langle \bar{y}, Jv \rangle \approx \langle J^* \bar{y}, v \rangle$$

are useful across several shapes, layouts, and participant counts. They test the relationship between forward and reverse linearizations, including collective normalization. They remain numerical tests rather than proofs.

The test set should deliberately exercise fan-out, shared parameters, repeated indices, partial contributions, uneven shards, and delayed reductions. These are cases where local shapes can look correct while the logical derivative is wrong. Parameter epochs and microbatch normalization should also be checked.

An exact real-arithmetic proof and a numerical execution comparison answer different questions. Both can be valuable. A publication should state which is supplied for each layer and which backend components remain trusted.

18.4 Performance measures

Useful measures include end-to-end time to a fixed amount of work, time to a stated validation target, peak device and host memory, network traffic, compile/search time, and recovery-adjusted goodput. Kernel-only speedups should remain kernel-only claims. Search time matters when shapes or resource availability change frequently.

A plan should be compared with a strong manually configured baseline and with simpler strategies, including a single-device baseline when feasible. Adding more devices can lower utilization or increase total cost. Reporting only aggregate FLOP utilization can conceal a worse application objective.

For communication-aware kernels, report whether improvements come from less traffic, lower launch overhead, better overlap, or a changed numerical representation. These mechanisms have different portability. A result obtained by using an approximate low-precision path should not be presented as a pure scheduling improvement.

18.5 Failure and elasticity

A failure evaluation should specify where failures occur: before send completion, during a collective, after gradient accumulation, during checkpoint writing, or after an optimizer commit. The invariant to check is that recovery produces an allowed logical state under the stated policy.

For synchronous semantics, verify that a restarted step does not duplicate contributions and that all consumers use the intended parameter epoch. For asynchronous semantics, verify quorum, staleness, and weighting rules. A system that keeps processes alive but silently changes its optimizer state has not demonstrated the intended recovery property.

Elasticity also needs a boundary. Changing a data-parallel group between steps is different from changing a tensor-parallel group inside a live collective. The compiler should either support a valid state migration or require a restart/recompile point. This is a capability to document, not an assumption to leave to the cluster operator.

18.6 Metaprogramming effort

The original motivation is productivity, so the evaluation should measure it. Report how much strategy code is needed, how a new operator is registered, how a new backend is added, how a failed plan is diagnosed, and how much manual tuning is required. Generated line count alone is a weak measure if the generator embeds extensive model-specific knowledge.

A useful extension experiment is to add an operator with a custom pullback and a new partitioning rule, then show which parts of the compiler remain unchanged. Another is to retarget a model from a uniform GPU cluster to an asymmetric CPU/GPU cluster using a documented

capability description. The goal is to expose reusable mechanisms rather than hide bespoke work.

No such extension experiment was executed for this manuscript. These are proposed evaluation criteria derived from the survey and semantic analysis.

19 Research agenda and final assessment

19.1 The missing shared intermediate contract

The largest remaining opportunity is a shared contract connecting global tensor meaning, cotangent roles, event dependencies, and resource effects. Existing systems often have excellent representations for some of these and ad hoc adapters for the rest. A useful common layer need not standardize every kernel language or scheduler. It should standardize the information required to compose them.

The most important fields are logical shape and type, representation and contribution role, derivative interface, residual lifetime, parameter epoch, communication participants, and completion semantics. A common schema for these could support multiple frontends, planners, and execution backends. The schema should have versioned capability negotiation and explicit unsupported cases.

This is a narrower and more achievable goal than a universally optimal compiler. It makes the search problem modular and gives each subsystem a testable obligation. Higher categorical semantics is useful here because it distinguishes programs, representation changes, and transformations between transformations without identifying them prematurely.

19.2 Proof-carrying strategies with empirical costs

A promising system combines generated strategies with a small deterministic checker and empirical cost models. The checker establishes structural legality relative to trusted primitive contracts. Profiling establishes performance evidence for the current deployment. A strategy can be legal and slow, or fast in a model and infeasible on real hardware; the architecture should make both outcomes easy to understand.

The proof burden should be proportionate. It is unnecessary to machine-check every exploratory plan before any experiment, but the IR should retain enough information that stronger checking can be added. Formal verification of a core transformation is useful even when local kernels remain trusted. Conversely, an extensive benchmark suite does not substitute for specifying what the transformation intends to preserve.

The most productive automation target is therefore a checked, explainable strategy artifact. It can be generated by a solver, an expert, or an AI tool and retuned as hardware changes. Its semantics remains stable while its cost estimates are updated.

19.3 The answer for heterogeneous AI clusters

The latest evidence shows that the necessary pieces are becoming substantially more expressive. Asymmetric SPMD handles nonuniform resources; explicit cotangent distribution makes backward communication easier to reason about; programmable training DAGs decouple strategies from runtimes; distributed tiling and chunk scheduling reduce bespoke overlap code; and actor/task systems make cluster orchestration more coherent.

There is still no evidence in this survey for one tool that automatically and optimally compiles every arbitrary tensor program and adjoint onto every CPU/GPU/network mixture. The practical answer is a layered compiler with explicit capabilities and contracts. It can automate a large supported class while rejecting or isolating unsupported effects.

For TCP/IP clusters, coarse compiled regions, topology-aware placement, communication batching, and explicit staging are central. For registered-memory fabrics, visibility and lifetime effects remain central. For weakly connected islands, a different distributed optimizer may be appropriate, but it must be identified as a different algorithm. Across all three regimes, correctness depends on the relationship between logical values, cotangent contributions, and committed state.

The higher categorical account contributes a disciplined language for that relationship. Its value is in stating assumptions, composing local obligations, and identifying where a claimed equivalence fails. The resulting research program is concrete: build interoperable checked plans, connect them to existing high-performance backends, and evaluate both the computation and the engineering effort they save.

Authorship, evidence, and availability

This manuscript was prepared with AI assistance at the request of the submitting researcher. The byline is left anonymous for submission preparation; no affiliation, author credential, peer-review outcome, or acceptance is asserted. The responsible human submitter should review the manuscript and journal requirements before publication.

The work is a literature survey and an original semantic synthesis. Its propositions are written mathematical arguments, not machine-checked artifacts. All performance numbers are attributed secondary reports of primary sources; no cluster benchmark was run for this paper. The proposed language, schema, checker, and runtime architecture are specifications and research directions, not delivered executable software.

The accompanying package contains the manuscript source, rendered PDF, bibliography data, source-evidence register, build instructions, and publication-preparation notes for both named journals. Third-party papers, repositories, and book PDFs are linked rather than redistributed. The same manuscript is intended for the two destinations; any actual simultaneous-submission disclosure and journal-specific terms remain matters for the submission workflow.

References

Links identify primary papers, author-hosted texts, official project material, or official documentation. Living documentation was inspected for the 3 November 2026 evidence snapshot. Publication dates and inspection dates are distinguished; future edits at a living URL may differ. A linked source is not a claim that its complete implementation or benchmark has been independently validated.

[R01] Yuanzhong Xu and colleagues. *GSPMD: General and Scalable Parallelization for ML Computation Graphs*. 2021 preprint. Primary source.

[R02] Lianmin Zheng and colleagues. *Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning*. OSDI 2022. Primary source.

[R03] Colin Unger and colleagues. *Unity: Accelerating DNN Training Through Joint Optimization of Algebraic Transformations and Parallelization*. OSDI 2022. Primary source.

[R04] Sami Alabed and colleagues. *PartIR: Composing SPMD Partitioning Strategies for Machine Learning*. 2024 preprint; ASPLOS 2025 lineage. Primary source.

[R05] OpenXLA contributors. *Shardy overview*. Living project documentation; inspected November 2026. Primary source.

[R06] JAX contributors. *Distributed arrays and automatic parallelization*. Living documentation; inspected November 2026. Primary source.

[R07] JAX contributors. *Autodiff and sharding.* Living documentation; inspected November 2026. Primary source.

[R08] JAX contributors. *JEP 17111: Efficient transposition of replication-inducing collectives.* Design documentation; inspected November 2026. Primary source.

[R09] JAX contributors. *Multi-process JAX documentation.* Living project documentation; inspected November 2026. Primary source.

[R10] Anxhelo Xhebraj and colleagues. *Scaling Deep Learning Training with MPMD Pipeline Parallelism.* MLSys 2025. Primary source.

[R11] OpenXLA Shardy contributors. *Shardy:MPMD RFC.* Design proposal dated 18 June 2025; inspected November 2026. Primary source.

[R12] Jason Ansel and colleagues. *PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation.* ASPLOS 2024. Primary source.

[R13] PyTorch contributors. *Torch compiler frequently asked questions.* Living documentation; inspected November 2026. Primary source.

[R14] PyTorch contributors. *Distributed tensor documentation.* Versioned documentation; inspected November 2026. Primary source.

[R15] Pian Pawakapan. *SPMD types in TorchTitan.* Engineering report; 26 August 2026. Primary source.

[R16] Meta PyTorch contributors. *spmd_types README.* Project documentation; inspected November 2026. Primary source.

[R17] Meta PyTorch contributors. *AutoParallel.* Experimental project documentation; inspected November 2026. Primary source.

[R18] Zhiqi Lin and colleagues. *nnScaler: Constraint-Guided Parallelization Plan Generation for Deep Learning Training.* OSDI 2024. Primary source.

[R19] nnScaler contributors. *Paralleling a Module.* Project documentation; inspected November 2026. Primary source.

[R20] Tessel contributors. *Tessel: Boosting Distributed Execution of Large DNN Models via Flexible Schedule Search.* HPCA/TPDS 2024 project lineage. Primary source.

[R21] Sunghyun Kim. *GraphPipe: Improving the Performance and Scalability of DNN Training with Graph Pipeline Parallelism.* MIT SM thesis, May 2024. Primary source.

[R22] Taegeon Um and colleagues. *Metis: Fast Automatic Distributed Training on Heterogeneous GPUs.* USENIX ATC 2024. Primary source.

[R23] Ran Yan and colleagues. *HexiScale: Facilitating Large Language Model Training over Heterogeneous Hardware.* 2024 preprint; revised 12 May 2026; formerly FlashFlex. Primary source.

[R24] Haoyang Li and colleagues. *Hetu v2: A General and Scalable Deep Learning System with Hierarchical and Heterogeneous Single Program Multiple Data Annotations.* OSDI 2026. Primary source.

[R25] Weifang Hu and colleagues. *Tessera: A Holistic Pipeline Parallelism Framework for Trillion-Parameter Heterogeneous MoE Training (Operational Systems).* OSDI 2026. Primary source.

[R26] Megan Frisella and colleagues. *Piper: A Programmable Distributed Training System.* 2026 preprint; submitted 9 June 2026. Primary source.

[**R27**] Mengdi Wu and colleagues. *Mirage: A Multi-Level Superoptimizer for Tensor Programs.* OSDI 2025. Primary source.

[**R28**] Xinhao Cheng and colleagues. *MPK: A Compiler and Runtime for Mega-Kernelizing Tensor Programs.* OSDI 2026. Primary source.

[**R29**] Size Zheng and colleagues. *Triton-distributed: Programming Overlapping Kernels on Distributed AI Systems with the Triton Compiler.* 2025 preprint. Primary source.

[**R30**] Size Zheng and colleagues. *DITRON: Distributed Multi-level Tiling Compiler for Parallel Tensor Programs.* ICML 2026; PMLR 306. Primary source.

[**R31**] Xinwei Qiang and colleagues. *Syncopate: Efficient Multi-GPU AI Kernels via Automatic Chunk-Centric Compute-Communication Overlap.* OSDI 2026. Primary source.

[**R32**] PyTorch Team at Meta. *Helion: A High-Level DSL for Performant and Portable ML Kernels.* Engineering announcement; October 2025. Primary source.

[**R33**] Aditya Srikanth and colleagues. *One Search, Two Layers: Compiler Controls Inside PyTorch Helion using NVIDIA CompileIQ.* Engineering report; 19 October 2026. Primary source.

[**R34**] Muyan Hu and colleagues. *VTC: DNN Compilation with Virtual Tensors for Data Movement Elimination.* OSDI 2026. Primary source.

[**R35**] Zuoyan Zhang, Jinyi Wang, Zhuo Tang, and Jie Zhao. *Global Scheduling of Transient Parameter Materialization for Sharded Training.* ACM TACO; 5 September 2026; primary metadata/abstract only. Primary source.

[**R36**] William S. Moses and colleagues. *Scalable Automatic Differentiation of Multiple Parallel Paradigms through Compiler Augmentation.* SC 2022. Primary source.

[**R37**] Enzyme contributors. *CUDA Guide: Heterogeneous AD.* Living documentation; inspected November 2026. Primary source.

[**R38**] Afif Boudaoud, Alexandru Calotoiu, Marcin Copik, and Torsten Hoefler. *DaCe AD: Unifying High-Performance Automatic Differentiation for Machine Learning and Scientific Computing.* 2025 preprint. Primary source.

[**R39**] SPCL contributors. *DaCeML: A Data-Centric Compiler for Machine Learning.* Archived project repository in inspected snapshot. Primary source.

[**R40**] Rohan Yadav, Alex Aiken, and Fredrik Kjolstad. *DISTAL: The Distributed Tensor Algebra Compiler.* PLDI 2022. Primary source.

[**R41**] Rohan Yadav, Alex Aiken, and Fredrik Kjolstad. *SpDISTAL: Compiling Distributed Sparse Tensor Computations.* SC 2022. Primary source.

[**R42**] Legion contributors. *Logical Regions.* Project tutorial; inspected November 2026. Primary source.

[**R43**] Realm contributors. *Realm overview.* Project documentation; inspected November 2026. Primary source.

[**R44**] StarPU contributors. *StarPU: A Unified Runtime System for Heterogeneous Multicore Architectures.* Project documentation; inspected November 2026. Primary source.

[**R45**] Chapel contributors. *GPU Programming.* Chapel 2.10 documentation; inspected November 2026. Primary source.

[**R46**] IREE contributors. *IREE overview.* Project documentation; inspected November 2026. Primary source.

[**R47**] IREE contributors. *C API*. Project documentation; inspected November 2026. Primary source.

[**R48**] Apache TVM contributors. *Design and Architecture: Disco Distributed Runtime*. Project documentation; inspected November 2026. Primary source.

[**R49**] Apache TVM contributors. *tvm.relax.distributed*. Project API documentation; inspected November 2026. Primary source.

[**R50**] Ray contributors. *Ray Compiled Graph and communication/computation overlap*. Project documentation; inspected November 2026. Primary source.

[**R51**] PyTorch Team at Meta. *Monarch: an API to your supercomputer*. Engineering report; 8 April 2026. Primary source.

[**R52**] Meta PyTorch contributors. *Monarch releases*. Release records; inspected November 2026. Primary source.

[**R53**] Monarch contributors. *Getting Started*. Monarch v0.7.0 documentation; inspected November 2026. Primary source.

[**R54**] PyTorch contributors. *torchcomms: a modern PyTorch communications API*. Engineering report; inspected November 2026. Primary source.

[**R55**] PyTorch contributors. *PyTorch Symmetric Memory*. Versioned documentation; inspected November 2026. Primary source.

[**R56**] NVIDIA. *NCCL Environment Variables*. NCCL documentation; inspected November 2026. Primary source.

[**R57**] NVIDIA. *Using NVSHMEM*. NVSHMEM API documentation; inspected November 2026. Primary source.

[**R58**] OpenUCX contributors. *Frequently Asked Questions*. UCX documentation; inspected November 2026. Primary source.

[**R59**] OpenUCX contributors. *Unified Collective Communications Specification*. UCC API specification v1.1; inspected November 2026. Primary source.

[**R60**] MPI Forum. *Nonblocking Collective Operations*. MPI 4.1 standard. Primary source.

[**R61**] Tracel contributors. *Burn crate documentation*. Burn 0.22.1 snapshot; inspected November 2026. Primary source.

[**R62**] Nathaniel Simard and Guillaume Lagrange. *Burn 0.21.0 Release: Up to 8x Lower Framework Overhead, Differentiable Collectives and Improved Kernels*. Release report; 7 May 2026. Primary source.

[**R63**] Burn contributors. *Migrating to Burn 0.22*. Migration documentation; inspected November 2026. Primary source.

[**R64**] Burn contributors. *Distributed Computing*. Burn Book; inspected November 2026. Primary source.

[**R65**] Melih Elibol, Jared Roesch, Isaac Gelado, Eric Buehler, and Michael Garland. *Fearless Concurrency on the GPU*. 2026 preprint; NVIDIA primary research record; 16 June 2026. Primary source.

[**R66**] Reactant contributors. *Reactant.jl: Optimizing Julia Functions with MLIR*. Project documentation; inspected November 2026. Primary source.

[**R67**] Reactant contributors. *Multi-Host Environments*. Project tutorial; inspected November 2026. Primary source.

[R68] JuliaParallel contributors. *Dagger GPU support.* Project documentation; inspected November 2026. Primary source.

[R69] JuliaGPU contributors. *State of Julia's GPU ecosystem in 2026.* Ecosystem engineering report; 3 August 2026. Primary source.

[R70] MLX contributors. *Distributed Communication.* MLX 0.32.4 documentation; inspected November 2026. Primary source.

[R71] ggml-org contributors. *llama.cpp RPC backend README.* Project documentation; inspected November 2026. Primary source.

[R72] Arthur Douillard and colleagues. *DiLoCo: Distributed Low-Communication Training of Language Models.* 2023 preprint. Primary source.

[R73] Arthur Douillard and colleagues. *Decoupled DiLoCo for Resilient Distributed Pre-training.* 2026 preprint; submitted 23 April 2026. Primary source.

[R74] Robin Cockett and colleagues. *Reverse Derivative Categories.* CSL 2020. Primary source.

[R75] Matthijs Vakar and Tom Smeding. *CHAD: Combinatory Homomorphic Automatic Differentiation.* 2021 preprint. Primary source.

[R76] Tom Smeding and Matthijs Vakar. *Efficient CHAD.* 2023 preprint. Primary source.

[R77] Mathieu Huot, Sam Staton, and Matthijs Vakar. *Higher Order Automatic Differentiation of Higher Order Functions.* 2021 preprint; version 9 revised 6 May 2026. Primary source.

[R78] Gordon Plotkin and John Power. *Algebraic Operations and Generic Effects.* Author-hosted research paper. Primary source.

[R79] Jacob Lurie. *Higher Topos Theory.* Princeton University Press, 2009; author-hosted text. Primary source.

[R80] Emily Riehl and Dominic Verity. *Elements of Infinity-Category Theory.* Cambridge University Press, 2022; author-hosted prepublication text. Primary source.

[R81] MLIR contributors. *MLIR Rationale.* Compiler documentation; inspected November 2026. Primary source.

[R82] OpenXLA contributors. *StableHLO Specification.* Language specification; inspected November 2026. Primary source.

[R83] Google Open Source contributors. *PJRT: Simplifying ML Hardware and Framework Integration.* Official technical announcement; May 2023. Primary source.

[R84] NVIDIA Megatron Core contributors. *Parallelism Strategies Guide.* Project documentation; inspected November 2026. Primary source.

[R85] DeepSpeed contributors. *AutoEP: Automatic Expert Parallelism.* Project documentation; inspected November 2026. Primary source.

[R86] Penghui Qi, Xinyi Wan, Guangxing Huang, and Min Lin. *Zero Bubble (Almost) Pipeline Parallelism.* ICLR 2024. Primary source.

[R87] PyTorch contributors. *Ahead-of-Time Compilation with torch.compile.* Main-branch documentation; inspected November 2026. Primary source.

[R88] PyTorch contributors. *Open Research, Tooling and Optimization at PyTorch Conference North America 2026.* Conference preview/report; October 2026; announcement-level evidence. Primary source.

[R89] Aaron Biggin and Jean-Simon Pacaud Lemay. *Reverse Faa di Bruno's Formula for Cartesian Reverse Differential Categories.* 2025 proceedings preprint; ACT 2024 lineage. Primary source.

[**R90**] TensorFlow contributors. *DTensor concepts*. Official guide; inspected November 2026. Primary source.

[**R91**] OneFlow contributors. *Global View*. Official guide; inspected November 2026. Primary source.

[**R92**] MindSpore contributors. *Automatic Parallel Strategy Search*. MindSpore 2.9.0 documentation; inspected November 2026. Primary source.

[**R93**] Futhark contributors. *Automatic differentiation prelude documentation*. Official language documentation; inspected November 2026. Primary source.

[**R94**] Dex contributors. *Dex research language*. Official project repository; inspected November 2026. Primary source.

[**R95**] Lei Wang and colleagues. *TileLang: Bridge Programmability and Performance in Modern Neural Kernels*. ICLR 2026. Primary source.

[**R96**] Microsoft contributors. *SuperScaler*. Experimental project repository; inspected November 2026. Primary source.

[**R97**] Xupeng Miao and colleagues. *Galvatron: Efficient Transformer Training over Multiple GPUs Using Automatic Parallelism*. 2022 preprint. Primary source.

[**R98**] ByteDance Seed contributors. *Triton-distributed repository*. Maintained project documentation; inspected November 2026. Primary source.

[**R99**] PyTorch contributors. *Bringing PyTorch Monarch to AMD GPUs: Single-Controller Distributed Training on ROCm*. Engineering report; 2026. Primary source.