CATFlow: Data-Monistic Parallelism Plan Discovery for Distributed Training

Abstract

As neural networks grow in size and complexity, distributed training across multiple devices becomes essential. An expressive abstraction is crucial for describing diverse parallelism plans to maximize hardware resource utilization. However, prevailing operator-centric abstractions fall short, as they fail to capture diverse data movement optimizations and data lifecycle management, and lack mechanisms for the automatic discovery of new parallelism plans.

In this paper, we introduce CATFlow, a framework with a novel abstraction that presents a key insight: defining parallelism spaces from a pure data-monistic perspective offers greater expressiveness and completeness than operator-centric approaches. CATFlow is built upon three fundamental concepts: tensors, space-time coordinates, and their relationships—formalized as constraints. It also provides a set of primitives for constructing and manipulating these elements, enabling the description of diverse schedules and the co-optimization of computation and data movement. A key advantage of CATFlow is its capacity to explicitly encode constraints on spatial dataflow trajectories and precise tensor lifecycles through these primitives, forming a space that can contain unforeseen strategies. This facilitates systematic searches within this constrained optimization space to discover new, efficient parallelism plans. Using a constraint-based potential-descent algorithm, CATFlow has discovered novel parallel training solutions such as ZeRO-3.5 and ZeRO-4, which achieve up to 39.1% performance improvement over ZeRO-3 and 36.6% memory reduction compared with ZeRO-2 across hardware configurations, as well as new offloading strategies that boost throughput by up to 89.4%. With these capabilities, CATFlow contributes to democratizing LLM training in enhancing resource utilization, particularly in resource-constrained scenarios.

1 Introduction

Deep Neural Networks (DNNs), particularly large language models (LLMs), have achieved advanced performance across a wide range of domains [2, 4, 32, 41]. However, the efficient training of large-scale models necessitates substantial resources, frequently surpassing the capabilities of one device [30]. Consequently, distributing training across multiple devices is crucial [1, 5]. The core challenge lies in identifying an efficient parallelism plan for distributed training. An NN model can be formally modeled as a computational graph [24], where nodes represent either data storage (tensors) or computations (operators), and directed edges represent dependencies between nodes. A parallelism plan specifies the distribution and scheduling of storage and computation in both space (device placement) and time (execution order) [9, 26, 36, 50]. The space of feasible parallelism plans is vast, and the problem of determining an optimal plan is NP-hard [27].

Due to the immense search space and complex constraints, distributed training optimizations often rely on meticulously designed parallelism paradigms that incorporate domain expertise. Examples include Megatron-LM [30, 40], which incorporates parameterized parallelism paradigms known as data, tensor, and pipeline parallelism (DP, TP, PP, collectively termed 3D parallelism) to support Transformer-based models. Alternatively, systems like Piper [45] and Alpa [53] adopt a hierarchical search, first determining a plan for pipeline (inter-operator) parallelism, then optimizing tensor (intra-operator) parallelism within each stage. A common pattern across these methods is to first define a specific parallelism paradigm (e.g., 3D or SPMD parallelism) and then perform an efficient hyper-parameter search within that paradigm—for instance, tuning the degrees of data, tensor, and pipeline parallelism.

While many studies [7, 10, 42, 43, 45, 46, 49, 53, 54] aim to find optimal hyperparameter configurations within well-established parallelism paradigms, recent work such as nnScaler [26] advocates for a more flexible approach. It provides domain-specific primitives that enable experts to construct customized parallelism plans, thus creating an abstraction framework for expressing more flexible solutions and unlocking further optimizations for large model training, e.g., 3F1B for AlphaFold2 [15]. Specifically, nnScaler's abstraction framework is predominantly operator-centric, introducing

Table 1: High-level comparison of different methodologies.

Feature	Alpa [53]	nnScaler [26]	CATFlow
Fundamental view	**Strategy-specific:** fixed combination of named strategies	**Operator-centric:** define strategy by operator behaviors	**Data-monistic:** define strategy by data space-time trajectories
Search space def. & Prim. semantics	**Fixed templates:** combination of existing paradigms	**Constructive:** default-deny, explicitly permit; build from scratch.	**Subtractive:** default-allow, explicitly forbid; carve from universal set
Completeness	Combination of built-ins	Complete for computation scheduling, no data-centric optimizations	Computation/memory/communication co-optimization
Searchability	**Param. search in fixed space**	**Param. search in user-defined space**	**Automatic strategy discovery**
Support examples	DP, TP, PP and their hybrid (3D parallelism), 1F1B	+ Existing: SP [20], ZeRO [37], GPipe [8], Chimera [21], etc. + New: Coshard, 3F1B, Interlaced pipeline, etc.	+ Existing: BPipe [16], WeiPipe [23], TeraPipe [22], Async PP [29], Cross-mesh resharding [55], etc. + New: ZeRO-2.5/3.5/4, PTD-offload, etc.

three primitives—`op-trans`, `op-assign`, and `op-order`—to specify operator placement, transformation, and execution order. Focusing primarily on operator scheduling, nnScaler enables the description of parallelism plans that extend far beyond the scope of traditional 3D or SPMD approaches.

Although nnScaler [26] represents a significant advance, capable of describing most existing training plans and new expert-crafted parallelism paradigms, existing approaches remain limited in expressing optimizations that require coordinated data movement and lifecycle management, such as BPipe [16], WeiPipe [23], complex cross-mesh resharding [55], and ZeRO-3.5 & 4 (new strategies discovered in this work). This limitation hinders the ability to fully exploit hardware resources. The root cause is that prior operator-centric abstractions [25,26,53] typically build on predefined dataflow assumptions and cannot effectively describe flexible spatial dataflow trajectories and fine-grained tensor lifecycle. Consequently, many potential optimization opportunities cannot be expressed or discovered due to this incompleteness.

We therefore present a core contrary insight in this work: a **data-monistic** perspective offers a more expressive and complete abstraction for parallelism plans description and strategy space construction. In this view, plans are defined by describing the trajectory of data and imposing constraints on its movement. From this vantage point, computation is implicitly represented as a **constraint** on dataflow, specifically, that all required data must meet at the same device. Consequently, the scheduling of computation follows naturally from the constraints governing data movement.

Building on this insight, we present CATFlow—a new framework for describing, generating, and searching for more advanced distributed-training plans. Its core is an abstraction centered on a single entity: the constraint-aware tensor (CAT). The constraints associated with each tensor specify its space-time coordinates—that is, when and where the tensor resides. On top of this representation, CATFlow provides a set of primitives for manipulating CATs to constrain parallelism plans that coordinate data movement and computation, including tensor distribution, dependency management, and strategy-space delineation primitives.

CATFlow's abstraction paves the way for addressing another important problem: prior work limits automated search to tuning hyperparameters (e.g., partition sizes, device assignments, execution order) within a fixed, manually crafted parallelism space, preventing the automatic discovery of **unforeseen parallelism plans**. In contrast to predefined search spaces, CATFlow employs declarative, **subtractive** primitives to **carve** a *universal* design space through constraints. While traditional search frameworks depend on constructive primitives, mandating that users define what a parallelism plan should be, CATFlow inverts this logic: designers simply state what a parallelism plan should not be. This reductive methodology generates a search space spanning known and unknown designs, fostering the discovery of advanced parallelism plans. Coupled with a data-monistic perspective, it delivers a unified, high-resolution strategy landscape that increases optimization tractability and supports seamless parallelism plan transitions during search. Building on this foundation, we introduce a constraint-based potential-descent algorithm capable of discovering novel parallelism plans. Table 1 summarizes the key distinctions between CATFlow and prior approaches.

CATFlow's core contributions are summarized as follows:

1. **Data-Monistic Abstraction and Parallelism Space:** We introduce CATFlow, a framework featuring novel abstractions and primitives for constructing spaces of parallelism plans in distributed NN training. From a data-monistic perspective, CATFlow enables the explicit and complete specification of plans governing both computation and data movement. In contrast to existing operator-centric methods, it unlocks a strictly larger parallelism-plan space.

2. **Automatic Discovery of Novel Training Plans:** By using CATFlow's unified abstractions to reductively carve a training-plan search space, we establish a foundation for the automated discovery of unforeseen training plans, in conjunction with an efficient potential-descent search algorithm.

3. **Invention of New Plans for Large Model Training:**

Based on CATFlow, we present new, efficient distributed training plans for resource-constrained scenarios that surpass state-of-the-art, manually designed solutions. These include new ZeRO strategies, ZeRO-3.5 & 4 that achieve the memory efficiency of ZeRO-3 and performance approaching that of ZeRO-2 (up to 39.1% improvement over ZeRO-3), as well as new offloading strategies that boost throughput by 89.4%.

2 Background & Motivation

2.1 Parallelism Paradigms & Plan Search

A *parallelism plan* specifies how a model and the associated data are partitioned and scheduled across multiple devices, including the distribution of tensors (data), operators (computations), and their temporal order. The challenge lies in the vast combinatorial space of possible plans, and finding an optimal solution is NP-hard [27]. Existing search methods commonly construct a search space from handcrafted, parameterized *parallelism paradigms* and then tune parameters within that space to efficiently find good plans. Typical *parallelism paradigms* include *data parallelism* (DP), *tensor parallelism* (TP), *pipeline parallelism* (PP), and *sequence parallelism* [20] (SP), and *3D(4D) parallelism* integrates DP, TP, PP (and SP) together. Megatron-LM [30, 40] is a representative framework that combines DP, TP, and PP for Transformer training. Subsequent systems like Piper [45] and Alpa [53] extend 3D parallelism with staged SPMD (single program multiple data). While effective, these systems often rely on fixed pipeline schedules [35], such as 1F1B [6], and overlook the potential of fine-grained micro-batch scheduling. Tessel [25] addresses this limitation by optimizing micro-batch scheduling for a given operator placement.

To enable automatic parallelization, existing training systems typically construct parameterized search spaces from predefined *parallelism paradigms*. Consequently, the quality of discovered *parallelism plans* depends strongly on the quality of these predefined spaces. As shown by nnScaler [26], such fixed spaces can miss important optimization opportunities. For example, AlphaFold2 [14] requires three forward passes before each backward pass, a pattern that cannot be efficiently handled by 1F1B pipeline schedules. In multilingual LLMs [31], applying standard 3D parallelism/staged SPMD can force large embedding tables to occupy excessive devices, leading to poor computational utilization. These cases demonstrate how hand-crafted, fixed search spaces fail to capture novel optimization patterns in evolving workloads.

2.2 Construction of New Parallelism Plans

Recent research has introduced a flexible abstraction framework for constructing diverse *parallelism plans*. Specifically, nnScaler [26] defines three core primitives for operator partitioning, spatial placement, and temporal ordering. By composing these primitives with user-specified constraints, nnScaler express a broader spectrum of *parallelism plans* than prior approaches, such as specialized pipeline schedules (e.g., 3F1B for AlphaFold2) and staged SPMD with customized operator mappings. Nevertheless, existing operator-centric abstractions introduce inherent limitations: they typically rely on predefined dataflow assumptions and cannot effectively handle flexible spatial dataflow and tensor lifecycle management.

In modern distributed training, performance depends not only on how computation is parallelized but also on how data is orchestrated and moved [9, 16, 23, 36, 50, 54]. Many state-of-the-art optimizations are explicitly data-driven, e.g., ZeRO [37, 52], activation checkpointing [3], and offloading [39], which tightly coordinate parameter sharding, prefetching, asynchronous communication, and conditional transfers with computation scheduling. However, operator-centric frameworks [25, 26] typically treat data movement as an implicit byproduct of operator placement and ordering, lacking mechanisms to explicitly control or co-optimize dataflow; consequently, they cannot express *parallelism plans* that jointly orchestrate computation and data movement [55] or require complex tensor lifecycle management [16, 23]. This limitation prevents the discovery and representation of a critical class of high-performance *parallelism plans*, motivating a more expressive, data-centric abstraction.

3 Abstraction with Constraint-Aware Tensor

To address the limitations of operator-centric abstractions, CATFlow introduces a new representation: the **constraint-aware tensor (CAT)**, associated with three tightly coupled concepts: *Tensor*, *Space-Time Coordinate*, and *Constraint*.

Tensor: Represents a data block within the computation graph, including both persistent model parameters and transient intermediate results, like activation values.

Space-Time Coordinate: Denotes the location (device) and temporal position. Rather than using physical time, the temporal coordinate is defined by Lamport logical time [18], which reflects the execution order (see § 4.2).

Constraint: Specifies the relationships between tensors and their space-time coordinates, determining where (device/location) and when (temporal order) a tensor exists or is operated upon. This enables explicit modeling of both data placement and execution timing for each tensor instance.

- **Intra-tensor constraint**: Restricts the trajectory of a single tensor across different space-time coordinates, governing its movement and lifecycle.
- **Inter-tensor constraint**: Ensures that multiple tensors coincide at the same space-time coordinate, thereby enabling computation or enforcing temporal relationships.

A CAT is more than a static data block—it encapsulates both the data themselves and the dynamic trajectory

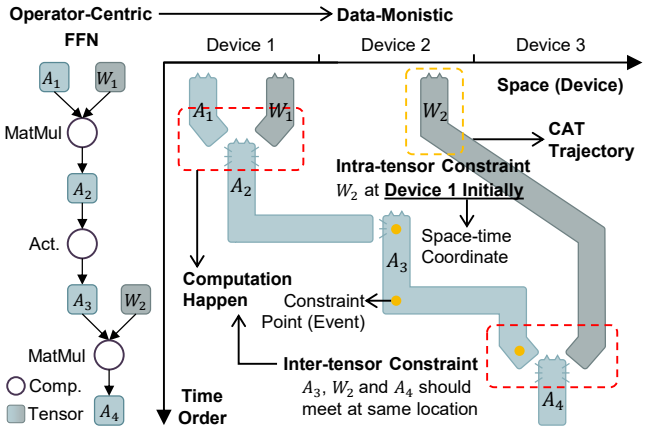


Figure 1: Illustration of the data-monistic abstraction: a baisc FFN layer in Transformer. Each tensor is annotated with constraints that explicitly specify its space-time coordinates.

of their generation, movement, and release. We use a series of **constraint points**, which form a **constraint vector**, to describe the space-time trajectory of a CAT. During implementation, these constraint points are realized as **events** that operate on the tensor. For instance, the first and last points in the constraint vector represent the CAT's generation and release events, respectively, and intermediate points typically signify movement events between devices. Fig. 1 demonstrates how a FFN layer is represented in the CAT abstraction. In this case, the trajectory of tensor A_3 is represented by the following constraint vector: $\mathbf{C}_{A_3} = [(\text{Device}_2, t_1), (\text{Device}_2, t_2), (\text{Device}_3, t_3)]$. $\mathbf{C}_{A_3}$ also signifies its lifecycle: generated on Device 2 by consuming A_2, and sent to Device 3, where it is consumed to produce A_4.

This data-monistic abstraction is strictly more expressive than operator-centric approaches. Instead of treating data movement as an implicit side effect of operator scheduling, CATFlow defines computation as a consequence of data colocation: an operator executes when its input tensors meet at the same space-time coordinate to generate output tensors in place. By describing the trajectory of data, we uniformly define the scheduling of computation, storage, and communication. This reversal allows CATFlow to represent any operator-centric plan, plus a wider range of parallelism plans that co-optimize computation and dataflow and manage complex data dependencies, lifecycle management, and movement.

4 Primitives for Abstraction Manipulation

To simplify the construction of data-monistic abstractions, CATFlow provides six high-level primitives and corresponding mechanisms. These primitives serve as tools for building parallelism spaces, connecting tensors and space-time coordinates through constraints to generate valid representations within CATFlow. They can be categorized into three groups:

Tensor distribution primitives include `split`, `assign`, `copy`, and `reduce`, which manage the spatial distribution of tensors. Together they define how data are partitioned, placed, and moved across devices.

Dependency management primitives include `order` and `redirect`, which govern the logical temporal ordering of events. `order` enforces specific ordering between events, whereas `redirect` transfers dependencies across tensors.

Strategy-space delineation mechanisms declaratively define and constrain the search space. For instance, a `domain` specifies the permissible range of values for a primitive's configuration (e.g., space-time coordinates), and `not` can explicitly exclude certain parallelism plans.

4.1 Glance over Primitives with ZeRO

ZeRO [37, 52] is a parallelism plan that builds upon DP to reduce memory usage. It achieves this by sharding the model parameters, gradients, and optimizer states across all available devices. The sharding strategy consists of tiered levels of ZeRO (Zero Redundancy Optimizer) [37]: ZeRO-1 shards optimizer states, ZeRO-2 adds gradient sharding, and ZeRO-3 (FSDP [52]) also shards parameters for maximum memory savings. From an operator-centric perspective, these variants appear identical if dataflow management is treated as a side effect of operator scheduling. Algorithm 1 shows how CATFlow's primitives can represent the core of ZeRO variants. Specifically, the following primitives are used:

`dst_tensor=copy(src_tensor,copy_num,path)` copies the `src_tensor` into `copy_num` replicas. The `path` parameter can specify the copy trajectory, e.g., W_1^2 is copied from W_1, W_1^3 is copied from W_1^2, etc. (Fig. 2(a)).

`split_tensors=split(tensor,split_vector)` partitions a `tensor` into multiple sub-tensors (`split_tensors`) according to `split_vector` (Fig. 2(b)).

`assign(tensor,devices)` assigns `tensor` to a sequence of `devices`, specifying the spatial trajectory of the tensor across multiple locations (Fig. 2(c)).

During each layer's computation, it may be necessary to gather sharded weights or gradients from other devices; for example, in ZeRO-3, each device must collect all shards of the weight to compute the current layer's forward and backward passes. This process does not need to be explicitly represented in CATFlow, as the backend compiler automatically infers and inserts such data-gathering based on data dependencies and constraints (see § 6). However, if required, the `reduce` primitive can be used to explicitly specify data gathering or reduction, as shown in Fig. 2(d).

4.2 Dependency and Partial Temporal Order

In § 4.1, we demonstrated how CATFlow expresses parallelism paradigms (e.g., ZeRO-1 to 3) that are cumbersome to capture with operator-centric primitives. Here, we further

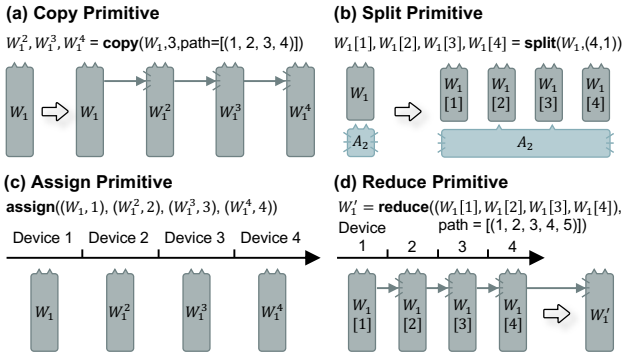


Figure 2: Illustration of data-centric primitives. (a) The `copy` primitive replicates tensors, enabling data replication across devices. (b) The `split` primitive partitions tensors, implicitly defining operator partitioning. (c) The `assign` primitive assigns tensors to devices, implicitly defining operator placement. (d) The `reduce` primitive manages data movement.

Algorithm 1 Primitive program for vanilla DP and ZeRO-1 to 3. Assume the number of devices is 4.

```
1:  # Vanilla DP space: Replicate weights
2:  activation[1 to 4] = split(activation, 4, dim=BATCH)
3:  assign(activation[i], device_i)
4:  weight^2, weight^3, weight^4 = copy(weight, 3)
5:  assign(weight^i, device_i) # weight^1 is weight
6:  # ZeRO-1 space: Add optimizer states sharding
7:  opt_state[1 to 4] = split(opt_state, 4)
8:  assign(opt_state[i], device_i)
9:  # ZeRO-2 space: Add weight gradients sharding
10: gradient[1 to 4] = split(gradient, 4)
11: assign(gradient[i], device_i)
12: # ZeRO-3 space: Add weights sharding, no weight copy
13: weight[1 to 4] = split(weight, 4)
14: assign(weight[i], device_i)
15: # Explicit all-gather
16: weight_buf[i] = reduce(W[j for j ≠ i])
```

illustrate how the data-monistic view articulates new parallelism paradigms. As an example, we can further augment ZeRO-3 with additional constraints: during the backward pass, activations are consumed progressively. The memory released by discarded activations can be immediately reused to host the incoming weight shards of the current layer. By the end of each gradient accumulation step, the model weights are already gathered, thereby eliminating all-gather communication during forward. In the subsequent forward pass, weight shards are discarded layer-by-layer, and the freed space is reclaimed by newly generated activations (ZeRO-3.5 in § 5.2).

To implement this, we introduce a new primitive, `order`. As the event vector of a CAT defines its space-time trajectory (§ 3), the ordering of events governs their temporal execution. A typical use of ordering is to enforce inter-CAT dependencies (Fig. 1) or temporal relationships. Consider two CATs, T_1 and T_2, where T_2 can be created only when T_1 exists. This translates to the constraint that the start event of T_2 ($T_2 \rightarrow$ start) occurs before the last event of T_1 ($T_1 \rightarrow$ complete). We use $T\{i\}$ to reference the i-th event in the constraint vector of T ($T\{0\}$ for $T \rightarrow$ start, $T\{-1\}$ for $T \rightarrow$ complete). While the system already preserves dependencies inherited from the original computational graph, the primitive `order(event1,event2)` explicitly injects additional ordering, ensuring that `event1` occurs before `event2`. For ZeRO-3.5, we use `order(`$A[i]^{ms+1}\{-1\}$`,`$weight\_buf[i]^{ms}\{-1\}$`)` (*ms* for micro-batch) to guarantee that weights gathered during backward are retained until the corresponding activations of the next forward are completed.

`redirect(tensor1, tensor2, tensor3)` is a syntactic-sugar primitive provided by CATFlow for complex dependency operations. For example, if the original dependencies are $T_3 \rightarrow T_2 \rightarrow T_1$, applying `redirect(`$T_2, T_4, T_3$`)` creates a new tensor T_4 from T_2 (T_4 also depends on T_1) and changes the dependency of T_3 from T_2 to T_4. The redirect primitive can be used to achieve advanced strategies such as recomputation.

4.3 Parallelism Space Delineation

Based on the preceding analysis, CATFlow's primitives are, at their core, a set of declarative constraints. By default, CATFlow posits that the initial parallelism space (before any primitives are written) is the universal set of all valid parallelism plans for a given computation graph and hardware system, constrained only by inherent data dependencies. In contrast to existing frameworks, which imperatively construct a plan from a null state, applying a primitive in CATFlow is a subtractive process. Each primitive declaratively refines the parallelism search space by imposing an additional constraint, thereby narrowing the universe of valid plans. For example, if Algorithm 1 is specified only at line 7, the behavior of weight gradients remains unspecified. The strategy space of ZeRO-1 is actually includes ZeRO-2; similarly, the strategy space of ZeRO-3 includes ZeRO-3.5. This approach can yield unforeseen parallelism plans that are not explicitly encoded by the user. The resulting space is then passed to a search algorithm (§ 5.1) and a compilation process (§ 6) to materialize a concrete plan. This design philosophy necessitates that CATFlow provide users with mechanisms to guide the delineation of the parallelism plan space and to express design preferences.

A primary mechanism is the **`domain`**. A domain represents a discrete, iterable set of valid candidates for a given parameter, such as a device location or a time step. For example, a user can define `d=Domain(Device1,Device2,Device3)` and then apply the constraint `assign(tensor/event,d)`. This declaration constrains the search space by asserting that the assignment must be one of the devices within domain `d`. As illustrated in Fig. 3, imposing event `move2` in the spatial

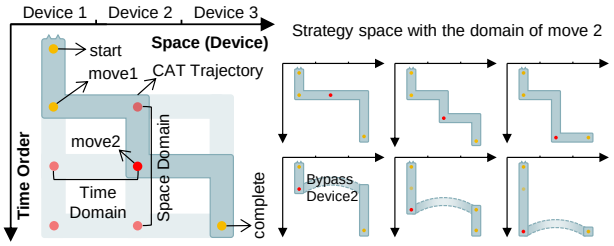


Figure 3: The spatial and temporal domain of `move2` yields a search space encompassing six feasible CAT trajectories.

and temporal domain can yield six possible CAT trajectories in this space. The **`not`** declaration allows users to explicitly forbid certain configurations by combining it with the primitives and `domain`. For instance, `not(assign(tensor,d))` means the tensor should not be mapped to domain `d`. The **`symmetry`** binds a set of CATs to perform uniform actions during search, reducing search dimensionality and avoiding irregular outcomes. During search, grouped CATs are treated as a single object for move selection and evaluation.

5 Parallelism Plan Discovery

5.1 Constraint-Based Potential Descent Search

The fine-grained control over tensor space-time trajectories and the subtractive method of carving the strategy space can result in a potentially larger search space. While this is advantageous for including emergent strategies and creating a more connected, higher-resolution landscape for heuristic search, it also inherently increases the search complexity. Therefore, a biased reduction of the space at the primitive declaration stage is not only necessary but also plays a crucial role in guiding the search. We do not claim that our method can automatically find an optimal strategy from the universal set of all possibilities. A rigorous mathematical analysis of the properties of this data-monistic search space, for instance, whether this unified abstraction enhances the topology smoothness and continuity, is also beyond the scope of this paper. Instead, this section presents a feasible search algorithm from the CAT perspective. The efficacy of this algorithm in discovering novel strategies will then be demonstrated empirically in § 7.

The search algorithm is inspired by potential-field methods [17]. It reframes the combinatorial-optimization problem as the search for a minimum-energy state in a potential field. At its core, we assigns a potential function to every CAT; each CAT then greedily migrates or transforms to a state that lowers its own potential. A simple illustrative example: suppose CAT1 currently resides on device 1. If we define the potential of CAT1 as the memory already occupied on that device, CAT1 will tend to migrate to a device with lower occupancy (more free memory). Repeating this greedy minimization for every CAT yields a reasonably balanced memory distribution.

Algorithm 2 Two-Level Potential-Descent Search Algorithm

$S \leftarrow S_{initial}$
for $i = 1$ to $max\_iterations$ **do**
  $best\_cat \leftarrow$ None; $best\_move \leftarrow$ None; $best\_\Delta \leftarrow 0$
  for each CAT T in S **do**
    $P_T \leftarrow \text{CalculateCatPotential}(T, S)$
    $C_{event} \leftarrow \text{GenerateEventCandidates}(T, S)$
    $C_{vector} \leftarrow \text{GenerateVectorCandidates}(T, S)$
    $C_{macro} \leftarrow \text{GenerateMacroCandidates}(T, S)$
    $M_T \leftarrow C_{event} \cup C_{vector} \cup C_{macro}$
    $m_T^\star \leftarrow \underset{m \in M_T}{\text{argmax}} (P_T - \text{CalculateCatPotential}(T,$
      $\text{ApplyCatMove}(S, T, m))$
    $\Delta_T \leftarrow P_T - \text{CalculateCatPotential}(T,$
      $\text{ApplyCatMove}(S, T, m_T^\star))$
    if $\Delta_T > best\_\Delta$ **then**
      $best\_cat \leftarrow T$; $best\_move \leftarrow m_T^\star$; $best\_\Delta \leftarrow \Delta_T$
    end if
  end for
  if $best\_\Delta \leq 0$ **then**
    break
  end if
  $S \leftarrow \text{ApplyCatMove}(S, best\_cat, best\_move)$
end for
return S

However, real scenarios are far more complicated. A strategy comprises many CATs, and each CAT contains a constraint vector composed of multiple events. Although each event can be scheduled independently, intricate dependencies may exist among events. Our search algorithm therefore employs a two-level potential definition, with each level paired with its own set of searchable actions (the next search move).

Level-1 potential is attached to individual events. In a concrete plan, an event's space-time coordinate is fixed; its potential is therefore quantified by the resource pressure of the underlying hardware at that coordinate. For event E:

$$P(E) = \alpha \frac{d_c(E)}{C - U_c} + \beta \frac{d_m(E)}{M - U_m} + \gamma \frac{d_b(E)}{B - U_b}, \qquad (1)$$

where $d_c(E)$, $d_m(E)$, and $d_b(E)$ denote the event's demand on compute, memory, and bandwidth respectively; C, M, and B are the device capacities; U_c, U_m, and U_b are the instantaneous usages by other events on the same device; and α, β, γ are scalar coefficients. Users can customize the potential function to reflect the optimization objective; for example, when high storage utilization is not the target, the memory-pressure can be set to zero whenever no overflow occurs.

Level-2 potential is associated with an individual CAT and comprises three components: intrinsic, corrective, and attractive. The intrinsic term sums the event-level potentials in the CAT's constraint vector $C = (E_1, E_2, \ldots, E_n)$. The corrective term adds the aggregate potential of events belonging to other

Algorithm 3 GenerateEventCandidates(T, S)

$C \leftarrow \emptyset$
for each event e in the constraint vector of T **do**
  $\mathcal{D}_e \leftarrow$ AdjacentDevices(e, S)
  $\mathcal{T}_e \leftarrow$ AdjacentTimes(e, S)
  $m_e^{st} \leftarrow \underset{(d,t)\in\mathcal{D}_e\times\mathcal{T}_e}{\text{argmin}}$ (EventPotential(e, d, t, S))
  if DecreasePotential(e, m_e^{st}, S) **and** Feasible(e, m_e^{st}, S) **then**
    $C \leftarrow C \cup \{\text{MakeMove}(e, m_e^{st})\}$
  end if
end for
return C

CATs coupled via inter-CAT constraints. The attractive term favors retaining a CAT on a device when future CATs on the same device depend on data derived from it. This design provides global awareness without simulating end-to-end performance at every action step. For CAT T, the accumulated potential is defined as:

$$P(T) = \sum_{i=1}^{n} P(E_i) + \sum_{T' \in \mathcal{A}(T)} \sum_{j=1}^{m} P(E'_j) + \delta A_{\text{attr}}(T), \quad (2)$$

where $\sum P(E_i)$ is the T's intrinsic potential, $\mathcal{A}(T)$ is the set of all CATs associated with T via inter-CAT constraints, and $A_{\text{attr}}(T)$ is computed by summing, over future CATs on the same device that depend on same data with T, the positive gap between each dependent's start time and this T's end time.

Search actions for a CAT fall into three categories: (1) modify a single event in the constraint vector—moves are atomic to handle inter-event constraints, and the best change follows the steepest descent of the event's Level-1 potential (`GenerateEventCandidates`); (2) change the vector's dimensionality by deleting an event or inserting a event in the tensor trajectory (`GenerateVectorCandidates`); (3) apply macro scheduling to the whole CAT (split, copy, reduce) (`GenerateMacroCandidates`). Available transformations are enumerated from permitted high-level policies. The union of these categories forms the candidate set for the CAT's next move. Algorithm 2 instantiates the search, which is greedy and iterative. Starting from an initial strategy sampled from the space, each iteration computes every CAT's potential, finds its best next move (maximum potential drop), then globally selects the CAT with the largest drop and commits its move. The process repeats until the stop condition.

The details of `GenerateEventCandidates` are shown in Algorithm 3. Device candidates are enumerated from the spatial adjacents; time candidates are the immediately preceding or succeeding order positions. For each event, the routine enumerates the Cartesian product of adjacent devices and times, evaluates the event potential on each pair, and selects the steepest-descent pair while preserving all constraints.

5.2 Representative Newly Discovered Plans

In this section, we introduce two representative strategies discovered automatically by CATFlow, ZeRO-3.5 & 4 (detailed search analysis in § 7.3). Compared to ZeRO-3 that shards weights throughout training and therefore requires all-gather in both forward and backward passes to assemble full weights, ZeRO-3.5 does not reshard the gathered weights during the backward pass. Instead, these gathered weights are retained and reused in the next accumulation step's forward pass, eliminating the forward all-gather. As shown in Fig. 4(c), during the backward pass, ZeRO-3.5 reclaims memory from progressively released activations and uses it to hold the gathered weight shards; when the total activation memory exceeds the total weight memory, ZeRO-3.5 achieves the same peak memory as ZeRO-3 while removing forward all-gather communication (1 block/step in Fig. 4(b)).

Building on ZeRO-3.5, ZeRO-4 further selectively delays reduce-scatter operations on gradients (realized by `order`$(A[j]^{\text{ms}+1}\{-1\}, G[j]^{\text{ms}}\{0\})$ in CATFlow). This allows part of gradients to remain on the local device during backward instead of being immediately communicated; the reduce-scatter is postponed to the next forward pass. As shown in Fig. 4(d), during backward, activations are progressively released and the freed memory accommodates the gathered weights and the deferred gradients. In the subsequent forward pass, gradients are reduced and weights are discarded to free memory for newly generated activations. In Fig. 4(d), we retain two layer's gradients for every three layers (choose j as multiples of three); in practice, j can be tuned according to actual resource usage. ZeRO-4 achieves a more balanced backward-forward communication pattern (0.67 block/step in Fig. 4(d)) than ZeRO-3.5, which can benefit from better communication-computation overlap.

§ 7 presents further plans discovered by CATFlow.

6 Implementation

The data-monistic abstraction of CATFlow represents a paradigm shift from existing operator-centric distributed-training frameworks [38], whose core architectures (IR, schedulers, runtimes) are tightly coupled to operator scheduling. Retrofitting CATFlow into these frameworks is highly challenging. Moreover, the runtime of existing systems is designed to execute a pre-compiled sequence of computational kernels and communication collectives. It remains an open research topic how to dynamically interpret a data-centric schedule to generate efficient low-level CUDA kernels on the fly [19]. Therefore, CATFlow is not implemented as an end-to-end system but rather serves as a strategy-exploration framework. It discovers parallelization plans that are then manually implemented in CUDA kernels or existing frameworks for high-performance execution.

Beyond these difficulties, our implementation focuses on

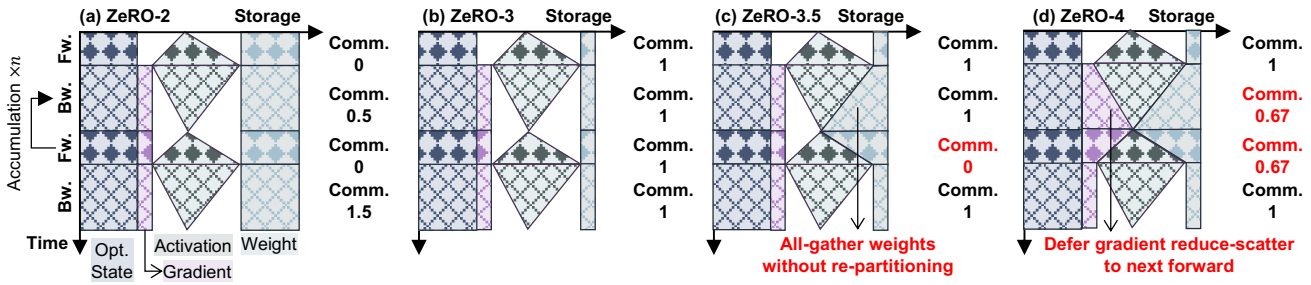


Figure 4: Memory and averaged communication over time for ZeRO-2, 3, 3.5, and 4. Assume each layer's forward computation takes 1 time step and each device transmits 1 block of data per weight or gradient transfer. The backward takes 2 time steps. Details for calculating averaged communication over time are omitted for simplicity.

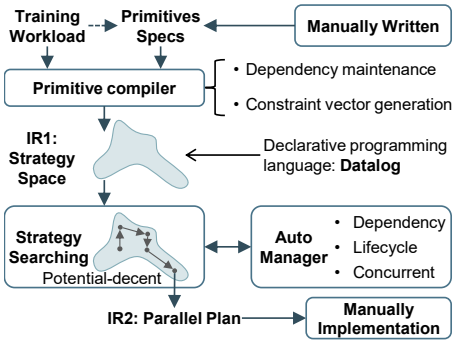


Figure 5: Implementation stack of CATFlow.

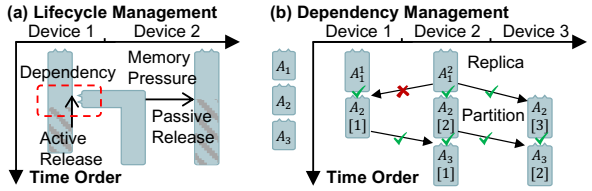


Figure 6: Example of lifecycle and dependency management.

two aspects: (1) Primitive Compilation & Strategy-Space IR (IR1) — the design of the primitive interface in high-level languages and its compilation into a constraint-declarative strategy-space IR; (2) Search & Plan Synthesis (IR2) — the search procedure over IR1, augmented with automatic management rules, that yields a concrete parallelization plan. The implementation stack of CATFlow is shown in Fig. 5.

We implement our primitives in Python, with the original training workload coming from PyTorch. Each primitive invocation is translated into a set of declarative constraints in Datalog; the strategy-space IR1 itself is represented in Datalog to enable constraint normalization. Datalog is a declarative logic-programming language [28] which is universal for constraint specification and integrates cleanly with Python. IR1 captures, for every CAT, its constraint vector (events), the spatial and temporal domains of each event, inter-CAT couplings, and resource constraints. The primitive compiler is responsible for maintaining IR semantic invariants while applying the transformations specified by the primitives. The most critical invariant is dependency preservation: if a CAT B depends on CAT A, and a primitive transforms A (e.g., `split`, `copy`) into derived CATs, the compiler rewrites B's dependencies to the derived CATs that equivalently replace A.

To ensure correct dependency rewriting, every original Tensor in the training workload is assigned a globally unique identifier. CATs generated by CATFlow carry provenance metadata that record the original Tensor identifier (origin_id) and the positional interval (range). When primitives generate CATs, the compiler compares the positional intervals of producer and consumer CATs: overlapping intervals introduce a dependency, while disjoint intervals permit pruning of the dependency. Following nnScaler's pTensor-vTensor mechanism [26], we use intersection-based matching to decide dependency creation or removal. For `copy` primitives that create redundant dependencies, the compiler preserves all candidate edges and defers selecting the most cost-effective dependency to the plan-synthesis stage.

Temporal constraints are encoded through a Lamport logical time system. Each event is assigned a unique timestamp. However, when an event possesses a temporal domain, its timestamp is superseded by a partial-order domain induced by other events. For instance, if event e must occur after both e_1 and e_2 and before e_3, and e_1 and e_2 are unordered, the constraint is represented as $(\{e_1, e_2\}, e_3)$.

6.1 Plan Synthesis

The strategy search is performed via the potential-descent algorithm described in § 5.1. At every search iteration, a concrete strategy (IR2) within the space is synthesized. To ensure unambiguous interpretation, we adopt a set of default management rules that complement the explicitly constraints.

Lifecycle Management: The lifecycle management ensures that each CAT is properly released after its usage. It

implements two rules: (1) *active release*: a CAT is freed immediately once no subsequent event depends on it; (2) *passive release*: a CAT may remain in memory even when unreferenced, but is evicted when memory pressure exceeds a predefined threshold (Fig. 6(a)). These rules may steer the search algorithm toward different strategies due to their different influences on potential evaluation (§ 5.1).

Dependency Management: A CAT may depend on other CATs whose data are scattered across multiple devices. The `copy` primitive can also render the dependencies recorded in IR1 redundant. The dependency manager therefore materializes a CAT's complete dependencies on demand. It first enumerates every required predecessor, then fetches them in ascending order of estimated acquisition cost until all dependencies are satisfied. For instance, if CAT A on device 1 needs CAT B and B exists on both device 1 and device 2, A prefers to consume the local replica, eliminating inter-device transfer. Fig. 6(b) illustrates examples involving `split` and `copy`.

Concurrency Management: We model the most common case for the concurrent execution of events: overlapping computation with communication. Other concurrent events are temporarily excluded because it is hard to find universal rules for them. For example, multiple concurrent *move* events on a same device will still be executed in a default order.

7 Evaluation

7.1 Experiment Setup

We structure our evaluation in three phases to assess the efficiency of CATFlow's generated strategies and its search procedure. First, we compare ZeRO-2, 3, 3.5, & 4 across different LLMs and hardware configurations, focusing on training throughput and memory usage. Second, we demonstrate CATFlow's search capability: starting from ZeRO-3, we trace how the potential distribution evolves under potential descent toward ZeRO-3.5 & 4. Finally, through comprehensive analysis, we identify 7 optimization directions suggested by potential descent under various hardware resources. By combining several of these directions, CATFlow proposes and implements a novel offloading strategy. CATFlow's expressiveness and search capability enable coordinated cross-resource trade-offs, including leveraging redundancy in one resource to alleviate bottlenecks in another. Accordingly, we evaluate CATFlow in improving utilization with tight resource budgets on small clusters to facilitate the democratization of LLM training for small companies and laboratories [44], rather than scaling under ever-increasing resources.

Models and Hardware. Our evaluations use 8B, 14B, and 32B models (Qwen3 family [51]). Hardware configurations are shown in Tab. 2. CPU in Offload experiments is Intel Xeon Platinum 8558.

Implementation and Configuration. For potential evaluation during search, we use a behavioral simulator that estimates compute, storage, and bandwidth usage, validated using measurements from real clusters. On real systems, we extend DeepSpeed's ZeRO-3 implementation with ZeRO-3.5/4 and offload-related strategies, and we unify other settings (e.g., reduce/prefetch bucket sizes, maximum live parameters) to ensure fair comparisons. We use FP16-FP32 mixed-precision training. Detailed configurations are reported with the results.

Table 2: Hardware configurations used in evaluation

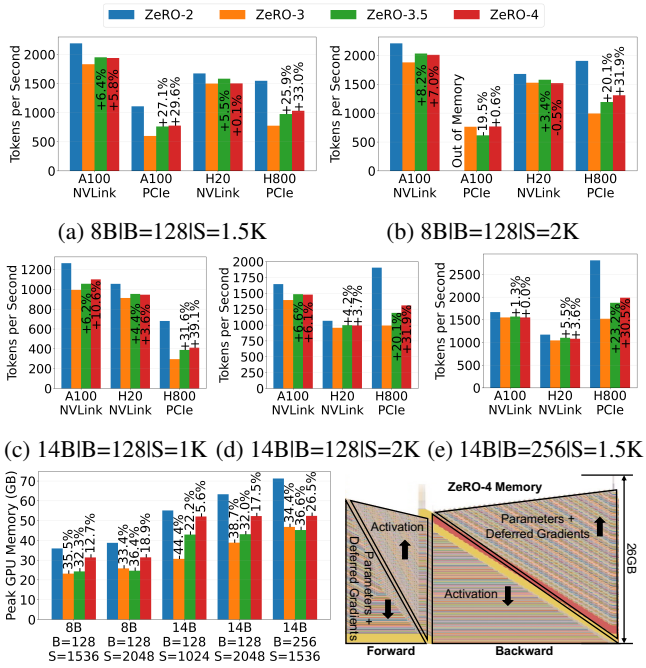
Platform	BW (GB/s)	Mem. (GB)	FP16 (TFLOPS)
H20-NVLink	900	96	148
A100-NVLink	400	80	312
H800-PCIe	128	80	756.5
A100-PCIe	64	40	312

7.2 Evaluation on New ZeRO Strategies

Fig. 7 compares the performance and memory footprint of ZeRO-2, ZeRO-3, ZeRO-3.5, and ZeRO-4 across different hardware and model settings. These experiments use 8 GPUs, 16 accumulation steps, and micro-batch sizes per device of 1 and 2 (total batch sizes 128 and 256). Results for Qwen3-14B on A100-PCIe (40 GB) are omitted due to OOM in most configurations. In ZeRO-3.5 (Fig. 4(c)), the backward pass reuses activation memory that is progressively released to hold gathered weight shards, which are discarded in the subsequent forward step to free space for activations. This preserves ZeRO-3's peak memory (Fig. 7(f)) while eliminating forward all-gather communication, resulting in up to a 31.6% throughput improvement over ZeRO-3 (Fig. 7(c)). ZeRO-4 further retains a portion of unreduced gradient buckets in the freed activation memory and defers their reduce-scatter to the next forward pass (Fig. 4(d)). By shifting part of the backward communication into the forward pass and overlapping it with forward computation, ZeRO-4 achieves more balanced communication and up to a 39.1% throughput improvement.

Fig. 7(g) shows a memory snapshot of ZeRO-4 for Qwen3-8B (batch size = 256; seq_len = 1.5K; memory for optimizer states, 32-bit parameters, and gradients is omitted since these are identical across strategies): peak activation memory = 26 GB; total 16-bit parameter memory = 16 GB; and 1/4 of gradient buckets deferred to the next forward pass, so parameters plus deferred gradients occupy 21 GB. Let P denote parameter memory, α the fraction of parameters retained during the backward pass, and β the fraction of gradients deferred to the next forward pass. If the activation peak A satisfies $A \geq (\alpha + \beta) \cdot P$, then the peak memory usage of ZeRO-3, ZeRO-3.5, and ZeRO-4 is comparable (Fig. 7(f)).

From a hardware perspective (Fig. 7(a-d)), the performance gains of ZeRO-3.5/4 over ZeRO-3 are more pronounced on systems with lower communication bandwidth (e.g., PCIe vs. NVLink) and under lighter compute loads (since com-

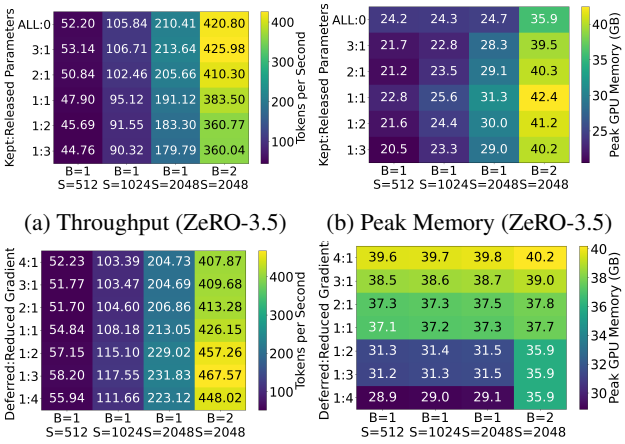


(a) 8B|B=128|S=1.5K (b) 8B|B=128|S=2K

(c) 14B|B=128|S=1K (d) 14B|B=128|S=2K (e) 14B|B=256|S=1.5K

(f) Peak Memory (vs ZeRO-2) (g) Memory Snapshot

Figure 7: Comparison of ZeRO-2, ZeRO-3, ZeRO-3.5, and ZeRO-4 in terms of performance (baseline: ZeRO-3) and memory footprint (baseline: ZeRO-2).



(a) Throughput (ZeRO-3.5) (b) Peak Memory (ZeRO-3.5)

(c) Throughput (ZeRO-4) (d) Peak Memory (ZeRO-4)

Figure 8: Configuration exploration of ZeRO-3.5 & ZeRO-4.

munication overhead is harder to overlap). ZeRO-3.5 eliminates the forward all-gather, and ZeRO-4 further shifts part of backward communication into the forward pass to improve communication-computation overlap, thereby yielding larger performance gains on communication-constrained systems.

We further analyze how hyperparameters affect the performance and memory usage of ZeRO-3.5 and ZeRO-4. We vary batch size/sequence length (to vary activation memory and computation loads) and strategy-specific hyperparameters: the ratio between retained gathered parameters and released parameters during backward (ZeRO-3.5), and the ratio between reduced gradient buckets and buckets whose gradient reduce-scatter is deferred to the subsequent forward pass (ZeRO-4). For ZeRO-4 exploration, we fix the retained-parameter ratio to `ALL:0` (all gathered parameters are kept during backward).

As shown in Fig. 8(a), ZeRO-3.5's performance generally increases as the fraction of retained parameters grows, with larger gains under lighter computation loads. However, increasing the retained-parameter ratio can also raise peak memory usage. When activation is small, the gathered parameters dominate peak memory, therefore, peak memory increases with the retained-parameter ratio (Fig. 8(b)). When activation is sufficiently large, the retained parameters can occupy the freed activation memory (Fig. 4(c)), and retaining more parameters no longer increases peak memory.

Figure 8(c) shows that ZeRO-4 attains the best performance at a moderate deferred-gradient ratio (1:3). A low deferred ratio degenerates to ZeRO-3.5, while a high deferred ratio introduces excessive forward communication, which can overload the forward pass and reduce communication-computation overlap. When activation is small, peak memory grows with the deferred-gradient ratio due to more retained gradient buckets during backward (Fig. 8(d)). When activation is large enough, peak memory becomes less sensitive to the deferred-gradient ratio where both deferred gradient buckets and retained parameters can be accommodated in the released activation memory (Fig. 4(d) & 7(g)).

7.3 Search Analysis

CATFlow's search process constructs a sequence of intermediate strategies that connect the initial strategy to the terminal one. As illustrated in Fig. 9, under the data-monistic abstraction, each all-gather operation in ZeRO-3 creates a Weight Buffer CAT that collects weight shards from peer devices. The transition from ZeRO-3 to ZeRO-3.5 is a progressively delaying the release event of the Weight Buffer CAT during the backward pass until the corresponding forward layer is reached. At that point, the automatic management selects the Weight Buffer resident on the local device rather than all-gathering weights from other devices.

In ZeRO search experiments, we set $\alpha = \beta = \gamma = 1$ in Equation (1) and $\delta = 0.1$ in Equation (2). Fig. 10 shows the distribution of event potential across CATs for ZeRO-3 and ZeRO-3.5. The changes of the Weight Buffer CAT are highlighted. It can be observed that two backward Weight Buffer CATs in the first accumulation step elongate their lifespans and replace the Buffer CATs of the subsequent forward results with a smaller accumulated potential.

Fig. 11(a) further plots the accumulated potential of each Weight Buffer CAT along the trajectory with every interme-

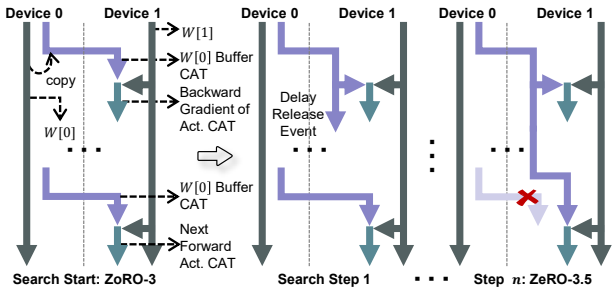


Figure 9: The intermediate plans during search from ZeRO-3 to ZeRO-3.5. A two-device example.

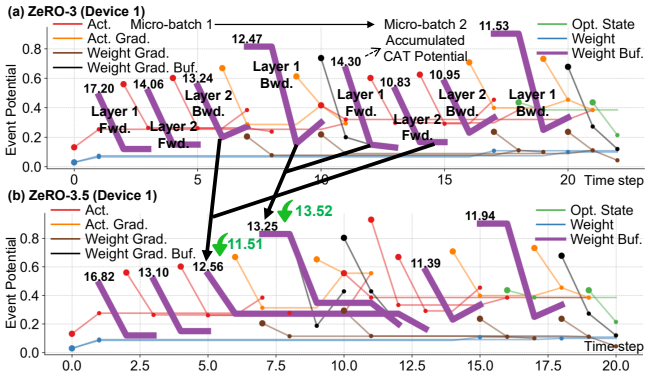


Figure 10: Potential distributions for (a) ZeRO-3 and (b) ZeRO-3.5. Both strategies are illustrated using a 2-layer, 2-accumulation-step, 4-device configuration.

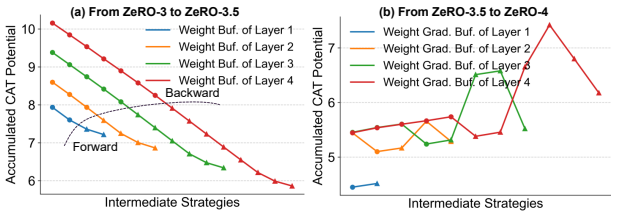


Figure 11: Search trajectory of weight buffer CAT (a) from ZeRO-3 to ZeRO-3.5. (b) from ZeRO-3.5 to ZeRO-4.

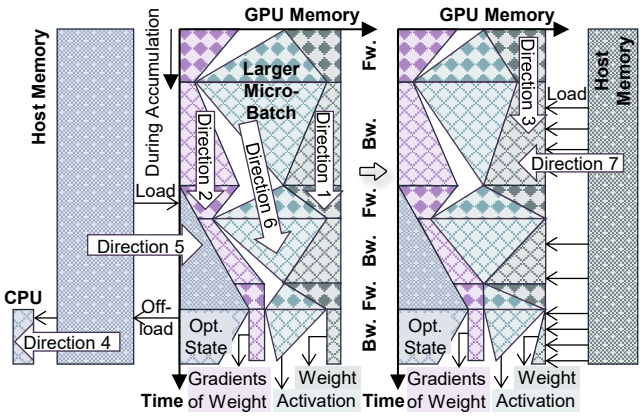


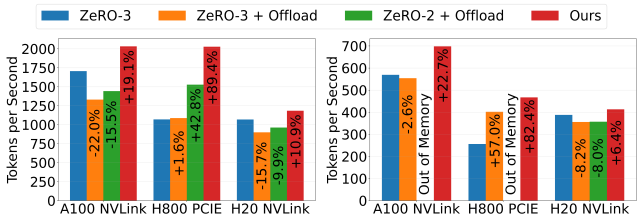
Figure 12: Comprehensive potential-descent directions discovered by CATFlow starting from ZeRO-3. The seven directions span ZeRO improvement (1–3) and offload (4–7) optimizations, each targeting different computation-memory–communication trade-offs.

diate plans from ZeRO-3 to ZeRO-3.5. With the defined potential function, this transition constitutes a natural potential-descent process, causing the high reachability of ZeRO-3.5. In contrast, attaining ZeRO-4 is more challenging. Deferring the backward Weight Gradient Buffer to the subsequent forward pass does not yield a monotonically decreasing communication load, while the memory usage grows with deferral (Fig. 11(b)) and the attraction part in Equation (2) cannot be applied. Consequently, the initial potential-descent of some Weight Gradients primarily serves to reveal promising directions rather than directly producing the final ZeRO-4 strategy.

7.4 Comprehensive Exploration

Starting from ZeRO-3, data-centric potential-decent identifies seven meaningful scheduling optimization directions across various model and hardware configurations. These seven directions are comprehensively illustrated in Figure 12. Directions 1 through 3 involve the GPU performing all storage and computation. **Direction 1** represents the ZeRO-3.5 strategy, i.e., delaying the release of backward weights. **Direction 2** represents the ZeRO-4 strategy, i.e., delaying the gradient reduce-scatter. **Direction 3** entails retaining a portion of forward weights for the backward pass, thereby increasing memory usage to reduce backward communication, which serves as an intermediate strategy between ZeRO-2 and ZeRO-3 (ZeRO-2.5). Directions 4 through 7 involve offload strategies: **Direction 4:** Conventionally storing optimizer states in host memory (or SSD), with the CPU performing optimizer state updates. **Direction 5:** Preloading optimizer states stored in host memory onto the GPU for updates. **Direction 6:** Employing a larger micro-batch size before optimizer state preloading, and a smaller size after loading. **Direction 7:** Off-loading weights to host memory. To the best of our knowledge, only Direction 4 and 7 are existing plans. Guided by these, we propose a new offloading strategy, referred to as PTD-offload, that combines Directions 3 to 6. We deploy this design on real hardware and evaluate it against classical offload baselines across model sizes and hardware configurations, reporting throughput and memory usage to characterize its behavior.

Baseline Offload (ZeRO-style). Conventional offload techniques store optimizer states in host memory (DRAM) or SSD. During backpropagation, weight gradients on the GPUs are transferred to the host, the CPU applies the optimizer update (Direction 4), and the updated weights are sent back to the GPUs. End-to-end iteration time is frequently limited by host-device bandwidth and CPU throughput.



(a) Qwen3-14B (b) Qwen3-32B

Figure 13: Offload strategy comparison (relative to: ZeRO-3).

Preload-then-discard Offload Strategy (PTD-offload). The strategy proposed by CATFlow exploits available GPU memory to preload optimizer states onto the GPUs ahead of the final backward within a batch, performs the optimizer updates on the GPUs, and directly discards the optimizer states thereafter (Direction 5). In parallel, the CPU executes a redundant optimizer-state update (Direction 4), but the GPUs do not wait for the CPU results; they proceed to the next batch's gradient accumulation and later load the CPU-updated states when available. In addition, when optimizer states are not resident on the GPUs, the freed memory can be used to (1) increase the micro-batch size to better amortize communication and improve utilization (Direction 6), and (2) preserve forward weights for reuse in the backward pass (Direction 3).

Fig. 13 compares our strategy with ZeRO-style offload strategies across two model sizes and three hardware configurations on 8 GPUs. We fix the per-GPU batch size and sequence length to 64 and 2048. For each configuration, we use the largest micro-batch size that does not cause OOM to maximize GPU utilization. Due to memory constraints, we enable gradient checkpointing for the 32B model, allowing a minimum batch size of 1 under ZeRO-3. We do not include ZeRO-2, as it triggers OOM across all configurations.

In low-bandwidth configurations (e.g., H800-PCIe), ZeRO-3 throughput is limited by communication overhead. Offloading with ZeRO-3 enables larger batch sizes and can improve throughput. However, for the 14B model, the CPU weight update adds compute cost, yielding a small net improvement. In contrast, for the 32B model with gradient checkpointing, offloading enables a large increase in batch size (e.g., from 1 to 16), thereby improving throughput. PTD-offload combines the benefits of ZeRO and offloading: it begins with larger micro-batches and retained weights during early accumulation, then transitions to fully weight-sharded after preloading optimizer states. This segmented approach improves compute–communication overlap and avoids waiting on CPU-side optimizer updates, yielding significant performance gains across most settings. The exception is the H20 configuration, which offers higher bandwidth and larger memory, implying larger batch sizes and higher compute utilization.

8 Related Works

Existing Training Optimizations. The growth of DNNs has led to diverse parallelism for distributed training [5, 47]. Common paradigms include DP, TP, PP, and SP [11], and modern systems often adopt hybrid parallelism [12, 33]. Megatron-LM [30] unifies DP, TP, and PP into 3D parallelism; TeraPipe [22] combines SP with PP for finer pipelining. As models scale, research focus has shifted to co-optimizing computation, memory, and communication to improve utilization [54, 55]. ZeRO [37] shards model states across DP devices to reduce redundancy, while offloading [39] and recomputation [3] further address memory limits. Mist [54] automates the co-optimization of memory-reduction techniques with parallelism, and many PP studies explore memory–performance trade-offs [16, 34, 48]. Overall, synergistic optimization of computation, memory, and communication has become a cornerstone of today's large-scale training.

Parallelism Plan Search. Finding an optimal plan is NP-hard [27]. Existing frameworks typically build a parameterized search space from combinations of parallelism paradigms and then tune parameters within that space [13, 45, 49, 53]. FlexFlow [13] searches SOAP strategies. Alpa [53] and Piper [45] use staged SPMD: first partition pipelines and then optimize intra-stage parallelism. Tessel [25] targets micro-batch pipeline scheduling under fixed placement. These methods operate in fixed, template-defined spaces, limiting discovery of new plans beyond those predefined boundaries.

Abstractions for Distributed Training. To support more flexible plans, recent frameworks introduce new abstractions. GSPMD [50] uses tensor-sharding annotations for compiler-driven parallelization. nnScaler [26] provides operator-centric primitives for placement, transformation, and ordering. Operator-centric views often treat data movement as a side effect of computation, whereas data-centric optimizations such as BPipe [16] and WeiPipe [23] emphasize explicit dataflow management. CATFlow adopts a data-monistic perspective, treating data as first-class citizens to enable co-optimization of computation, memory, and communication.

9 Conclusion

We present CATFlow, a data-monistic framework that abstracts distributed training by unifying representations of data and computation. Building on CATs and data-centric primitives, CATFlow enables the declarative specification of versatile parallelism plan space. We propose a constraint-based potential descent search algorithm that explores this vast strategy space, discovering parallelism plans that outperform existing solutions. CATFlow focuses on high-level resource scheduling and does not extend to optimizations involved with low-level hardware architecture, such as virtual memory management. In addition, more efficient runtime search algorithms for inference services could be further explored.

References

[1] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. Varuna: scalable, low-cost training of massive deep learning models. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 472–487, 2022.

[2] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020.

[3] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *CoRR*, abs/1604.06174, 2016.

[4] DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. 2025.

[5] Jiangfei Duan, Shuo Zhang, Zerui Wang, Lijuan Jiang, Wenwen Qu, Qinghao Hu, Guoteng Wang, Qizhen Weng, Hang Yan, Xingcheng Zhang, Xipeng Qiu, Dahua Lin, Yonggang Wen, Xin Jin, Tianwei Zhang, and Peng Sun. Efficient training of large language models on distributed infrastructures: A survey. 2024.

[6] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, et al. Dapple: A pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 431–445, 2021.

[7] Hao Ge, Fangcheng Fu, Haoyang Li, Xuanyu Wang, Sheng Lin, Yujie Wang, Xiaonan Nie, Hailin Zhang, Xupeng Miao, and Bin Cui. Enabling parallelism hot switching for efficient training of large language models. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 178–194, 2024.

[8] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

[9] Yuzhen Huang, Yingjie Shi, Zheng Zhong, Yihui Feng, James Cheng, Jiwei Li, Haochuan Fan, Chao Li, Tao Guan, and Jingren Zhou. Yugong: Geo-distributed data and job placement at scale. *Proceedings of the VLDB Endowment*, 12(12):2155–2169, 2019.

[10] Mikhail Isaev, Nic McDonald, Larry Dennison, and Richard Vuduc. Calculon: a methodology and tool for

high-level co-design of systems and large language models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2023.

[11] Sam Ade Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Shuaiwen Leon Song, Samyam Rajbhandari, and Yuxiong He. Deepspeed ulysses: System optimizations for enabling training of extreme long sequence transformer models. *arXiv preprint arXiv:2309.14509*, 2023.

[12] Arpan Jain, Ammar Ahmad Awan, Asmaa M Aljuhani, Jahanzeb Maqbool Hashmi, Quentin G Anthony, Hari Subramoni, Dhableswar K Panda, Raghu Machiraju, and Anil Parwani. Gems: Gpu-enabled memory-aware model-parallelism system for distributed dnn training. In *SC20: international conference for high performance computing, networking, storage and analysis*, pages 1–15. IEEE, 2020.

[13] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *Proceedings of Machine Learning and Systems*, 1:1–13, 2019.

[14] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, et al. Highly accurate protein structure prediction with alphafold. *nature*, 596(7873):583–589, 2021.

[15] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A. A. Kohl, Andrew J. Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W. Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. Highly accurate protein structure prediction with alphafold. *Nature*, 596(7873):583–589, July 2021.

[16] Taebum Kim, Hyoungjoo Kim, Gyeong-In Yu, and Byung-Gon Chun. Bpipe: Memory-balanced pipeline parallelism for training large language models. In *International Conference on Machine Learning*, pages 16639–16653. PMLR, 2023.

[17] Yoram Koren, Johann Borenstein, et al. Potential field methods and their inherent limitations for mobile robot navigation. In *Icra*, volume 2, pages 1398–1404, 1991.

[18] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*, pages 179–196. 2019.

[19] Robert Tjarko Lange, Aaditya Prasad, Qi Sun, Maxence Faldor, Yujin Tang, and David Ha. The ai cuda engineer: Agentic cuda kernel discovery, optimization and composition. Technical report, Technical report, Sakana AI, 02 2025, 2025.

[20] Shenggui Li, Fuzhao Xue, Chaitanya Baranwal, Yongbin Li, and Yang You. Sequence parallelism: Long sequence training from system perspective. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2391–2404, Toronto, Canada, July 2023. Association for Computational Linguistics.

[21] Shigang Li and Torsten Hoefler. Chimera: efficiently training large-scale neural networks with bidirectional pipelines. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.

[22] Zhuohan Li, Siyuan Zhuang, Shiyuan Guo, Danyang Zhuo, Hao Zhang, Dawn Song, and Ion Stoica. Terapipe: Token-level pipeline parallelism for training large-scale language models. In *International Conference on Machine Learning*, pages 6543–6552. PMLR, 2021.

[23] Junfeng Lin, Ziming Liu, Yang You, Jun Wang, Weihao Zhang, and Rong Zhao. Weipipe: Weight pipeline parallelism for communication-effective long-context large model training. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 225–238, 2025.

[24] Junfeng Lin, Huanyu Qu, Songchen Ma, Xinglong Ji, Hongyi Li, Xiaochuan Li, Chenhang Song, and Weihao Zhang. Songc: A compiler for hybrid near-memory and in-memory many-core architecture. *IEEE Transactions on Computers*, 2023.

[25] Zhiqi Lin, Youshan Miao, Guanbin Xu, Cheng Li, Olli Saarikivi, Saeed Maleki, and Fan Yang. Tessel: Boosting distributed execution of large dnn models via flexible schedule search. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 803–816. IEEE, 2024.

[26] Zhiqi Lin, Youshan Miao, Quanlu Zhang, Fan Yang, Yi Zhu, Cheng Li, Saeed Maleki, Xu Cao, Ning Shang, Yilei Yang, Weijiang Xu, Mao Yang, Lintao Zhang, and Lidong Zhou. nnScaler: Constraint-Guided parallelization plan generation for deep learning training. In *18th USENIX Symposium on Operating Systems Design and*

Implementation (OSDI 24), pages 347–363, Santa Clara, CA, July 2024. USENIX Association.

[27] Guodong Liu, Youshan Miao, Zhiqi Lin, Xiaoxiang Shi, Saeed Maleki, Fan Yang, Yungang Bao, and Sa Wang. Aceso: Efficient parallel dnn training through iterative bottleneck alleviation. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 163–181, 2024.

[28] David Maier, K Tuncay Tekle, Michael Kifer, and David S Warren. Datalog: concepts, history, and outlook. In *Declarative Logic Programming: Theory, Systems, and Applications*, pages 3–100. 2018.

[29] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM symposium on operating systems principles*, pages 1–15, 2019.

[30] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*, pages 1–15, 2021.

[31] Jianmo Ni, Gustavo Hernandez Abrego, Noah Constant, Ji Ma, Keith Hall, Daniel Cer, and Yinfei Yang. Sentence-t5: Scalable sentence encoders from pretrained text-to-text models. In *Findings of the association for computational linguistics: ACL 2022*, pages 1864–1874, 2022.

[32] OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Jan Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr H. Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine B. Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang,

Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. Gpt-4 technical report. 2024.

[33] Jay H Park, Gyeongchan Yun, M Yi Chang, Nguyen T Nguyen, Seungmin Lee, Jaesik Choi, Sam H Noh, and Young-ri Choi. Hetpipe: Enabling large dnn training on (whimpy) heterogeneous gpu clusters through integration of pipelined model parallelism and data parallelism. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 307–321, 2020.

[34] Penghui Qi, Xinyi Wan, Nyamdavaa Amar, and Min Lin. Pipeline parallelism with controllable memory. *Advances in Neural Information Processing Systems*, 37:46539–46566, 2024.

[35] Penghui Qi, Xinyi Wan, Guangxing Huang, and Min Lin. Zero bubble (almost) pipeline parallelism. In *The Twelfth International Conference on Learning Representations*, 2024.

[36] Ziyue Qiu, Hojin Park, Jing Zhao, Yukai Wang, Arnav Balyan, Gurmeet Singh, Yangjun Zhang, Suqiang Jack Song, Gregory R Ganger, and George Amvrosiadis. Moirai: Optimizing placement of data and compute in hybrid clouds. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, pages 875–891, 2025.

[37] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020.

[38] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 3505–3506, 2020.

[39] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. Zero-offload: Democratizing billion-scale model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 551–564, 2021.

[40] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[41] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, October 2017.

[42] Foteini Strati, Zhendong Zhang, George Manos, Ixeia Sánchez Périz, Qinghao Hu, Tiancheng Chen, Berk Buzcu, Song Han, Pamela Delgado, and Ana Klimovic. Sailor: Automating distributed training over dynamic, heterogeneous, and geo-distributed clusters. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, SOSP '25, page 204–220, New York, NY, USA, 2025. Association for Computing Machinery.

[43] Zhenbo Sun, Huanqi Cao, Yuanwei Wang, Guanyu Feng, Shengqi Chen, Haojie Wang, and Wenguang Chen. Adapipe: Optimizing pipeline parallelism with adaptive recomputation and partitioning. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, page 86–100, New York, NY, USA, 2024. Association for Computing Machinery.

[44] Zhenbo Sun, Shengqi Chen, Yuanwei Wang, Jian Sha, Guanyu Feng, and Wenguang Chen. Mepipe: Democratizing llm training with memory-efficient slice-level pipeline scheduling on cost-effective accelerators. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, page 1263–1278, New York, NY, USA, 2025. Association for Computing Machinery.

[45] Jakub M Tarnawski, Deepak Narayanan, and Amar Phanishayee. Piper: Multidimensional planner for dnn parallelization. *Advances in Neural Information Processing Systems*, 34:24829–24840, 2021.

[46] Taegeon Um, Byungsoo Oh, Minyoung Kang, Woo-Yeon Lee, Goeun Kim, Dongseob Kim, Youngtaek Kim, Mohd Muzzammil, and Myeongjae Jeon. Metis: Fast automatic distributed training on heterogeneous gpus. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 563–578, 2024.

[47] Joost Verbraeken, Matthijs Wolting, Jonathan Katzy, Jeroen Kloppenburg, Tim Verbelen, and Jan S. Reller-

meyer. A survey on distributed machine learning. *ACM Comput. Surv.*, 53(2), March 2020.

[48] Xinyi Wan, Penghui Qi, Guangxing Huang, Min Lin, and Jialin Li. Pipeoffload: Improving scalability of pipeline parallelism with memory optimization. *arXiv preprint arXiv:2503.01328*, 2025.

[49] Minjie Wang, Chien-chin Huang, and Jinyang Li. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–17, 2019.

[50] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, Ruoming Pang, Noam Shazeer, Shibo Wang, Tao Wang, Yonghui Wu, and Zhifeng Chen. Gspmd: General and scalable parallelization for ml computation graphs, 2021.

[51] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report. 2025.

[52] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. Pytorch fsdp: Experiences on scaling fully sharded data parallel. 2023.

[53] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, Carlsbad, CA, July 2022. USENIX Association.

[54] Zhanda Zhu, Christina Giannoula, Muralidhar Andoorveedu, Qidong Su, Karttikeya Mangalam, Bojian Zheng, and Gennady Pekhimenko. Mist: Efficient distributed training of large language models via memory-parallelism co-optimization. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, page 1298–1316, New York, NY, USA, 2025. Association for Computing Machinery.

[55] Yonghao Zhuang, Lianmin Zheng, Zhuohan Li, Eric Xing, Qirong Ho, Joseph Gonzalez, Ion Stoica, Hao Zhang, and Hexu Zhao. On optimizing the communication of model parallelism. *Proceedings of Machine Learning and Systems*, 5:526–540, 2023.