Towards ML System Extensibility

Weixin Deng*, Andy Ruan*, Megan Frisella*, Kai-Hsun Chen†, SangBin Cho†,
Jack Tigar Humphries†, Rui Qiao†, Stephanie Wang*†
*University of Washington †while at Anyscale

ABSTRACT

With the rise of large language models, distributed execution across multiple accelerators has become commonplace. Current ML systems must adopt complex distributed execution strategies for efficiency, but do so at the cost of extensibility. We believe that it is time to introduce a general-purpose distributed runtime for programming clusters of accelerators that enables: (1) placement flexibility, and (2) interoperability, without sacrificing (3) codesign. We propose using the DAFT API: *distributed actors, futures, and tasks.* To enable a smooth tradeoff between flexibility vs. performance, we introduce two execution modes: interpreted vs. compiled. We show how current applications in LLM inference and training can be executed as interpreted and compiled DAFT programs and discuss open questions and challenges.

ACM Reference Format:

Weixin Deng, Andy Ruan, Megan Frisella, Kai-Hsun Chen, SangBin Cho, Jack Tigar Humphries, Rui Qiao, Stephanie Wang. 2025. Towards ML System Extensibility. In *Workshop in Hot Topics in Operating Systems (HOTOS 25), May 14–16, 2025, Banff, AB, Canada.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3713082.3730392

1 INTRODUCTION

Large language models (LLMs) have revolutionized AI applications. They are also highly resource-intensive, inspiring a renaissance in efficient ML systems. A key change driven by LLMs is that distributed execution across multiple accelerators is now commonplace. This is now true for pre-training, post-training, offline inference, online inference, and combinations thereof. Consequently, ML systems now require increasingly complex distributed execution strategies, including 4D parallelism for pre-training [13, 18, 22, 36], prefill-decode disaggregation for online inference [28, 45], and offloading of compute and/or memory to the CPU [15, 31, 35].

HOTOS 25, May 14–16, 2025, Banff, AB, Canada

ACM ISBN 979-8-4007-1475-7/25/05
https://doi.org/10.1145/3713082.3730392

Recent open-source ML systems have adopted such strategies for better performance and scale. However, *as ML frameworks support more complex distributed execution strategies, extensibility suffers.* For example, the vLLM [14] framework's control plane was designed to minimize data movement between the CPU and GPU for tensor-parallel (TP) inference, but the same optimization turned out to introduce bottlenecks for pipeline-parallel (PP) inference (Section 2).

Given these trends, a key question arises: *how do we build performant and extensible ML systems in the face of constantly evolving applications and hardware?* In this paper, we study the distributed problem. Note that single-node or even single-GPU systems can be viewed as "distributed", since at minimum GPUs require coordination with the CPU. We consider extensibility along three dimensions:

(1) **Placement flexibility**: Can the developer control when and where computations should execute?
(2) **Interoperability**: Can the system be easily and efficiently composed with other systems?
(3) **Codesign**: Can the developer introduce performance optimizations specialized to the workload?

(2) implies that we need an *intermediate abstraction* to unify different ML systems. (1) implies that the abstraction should *decouple the data plane from the control plane.* The challenge is how to achieve these *without sacrificing performance (3).*

Most distributed ML systems today optimize entirely for (3), sacrificing (1) and (2). They use the single program multiple data (SPMD) model, which runs one copy of the same program on each accelerator (Figure 1a). SPMD makes it simple to express distributed programs that execute in lockstep, such as in data-parallel (DP) training. It is also efficient because all computation and communication operations are specified ahead of time, eliminating control plane overhead.

However, SPMD programs also tightly couple the user code to a *static* execution strategy and device placement. Supporting *dynamic* control flow and arbitrary parallelism strategies is difficult. This is because modularity and composition is challenging; orchestrating two SPMD programs requires direct modification to schedule the two programs.

While SPMD's limitations have been noted before [7, 19, 34], most open-source ML systems today are still built with SPMD. We believe this is because the community lacks a fully extensible and performant alternative. We propose such a system and in particular the *distributed actors, futures, and tasks* (DAFT) API for programming clusters of accelerators (example in Figure 1b). DAFT supports distributed dataflow graphs with arbitrary user-defined functions. The system uses a single controller [7], i.e. a centralized scheduler to coordinate cluster resources and communication operations.

Previous works such as Ray [25] have used the DAFT API to improve extensibility of CPU-centric dataflow systems [23, 25, 26, 38, 39]. However, extending current DAFT systems to support ML workloads is challenging due to the differences between CPU and accelerator execution. As an analogy, single GPU performance depends on the CPU running ahead of the GPU, dispatching as many kernels for execution as possible before synchronizing the device. The same principle applies to the controller of a distributed GPU pool. However, to support dynamic scheduling, the controller must synchronize with the workers.

Our key idea is to smoothly trade off between dynamic vs. static execution by introducing two DAFT execution modes: *interpreted* vs. *compiled*. An interpreted DAFT program executes eagerly, one task at a time. A compiled DAFT program freezes the dataflow graph, allowing the system to execute control plane decisions at compile time. We propose supporting both modes in the same system and allowing the two modes to interoperate in one application. Thus, we hope to provide the best of both worlds: flexibility and debuggability along with high performance.

We have built a prototype for compiled DAFT based on the system Ray [25, 39]. Ray is an interpreted DAFT system with a custom CPU-based object store. It does not support GPU objects nor compiled DAFT. We show that a compiled mode can reduce Ray's control plane overhead to provide performance on par with the SPMD-based vLLM system [14] for LLM inference and the SPMD-based PyTorch library for data-parallel training [18]. We conclude with a discussion of open systems challenges and opportunities.

```
model = modelA if self.rank == 0 else modelB
while True:
  if self.rank == 0:
    inp = torch.rand(N, device="cuda")
    tensor: torch.Tensor = model.execute(inp)
    send(tensor, peer=1)
  else:
    tensor = torch.zeros(N)
    recv(tensor, peer=0)
    output = model.execute(tensor)
```

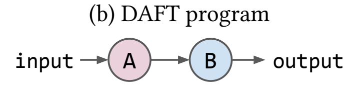
(a) SPMD program

```
@ActorClass(num_gpus=1)
class ModelA:
  @TypeHint(type=torch.Tensor, comm="nccl", shape="auto")
  def execute(self, input: torch.Tensor) -> torch.Tensor:
    return self.model.execute(input)

@ActorClass(num_gpus=1)
class ModelB:
  def execute(self, tensor: torch.Tensor) -> torch.Tensor:
    return self.model.execute(tensor)

A, B = ModelA.remote(), ModelB.remote()
def schedule(A: Actor[ModelA], B: Actor[ModelB]):
    inp = torch.rand(N, device="cuda")
    tensor_ref: Ref = A.execute.remote(inp)
    output_ref: Ref = B.execute.remote(tensor_ref)
    output: torch.Tensor = daft.get(output_ref)
```

(b) DAFT program



(c) Dataflow graph for (b)

Figure 1: Code samples for a simple chain of two models. In (a), the SPMD-style program is simple but does not support dynamic inputs, dynamic tensor shapes, dynamic scheduling, communication-computation overlap, failure detection, and sending CPU metadata alongside the GPU tensor. (b) only requires modifying the execute definition to support these.

2 MOTIVATION

We overview recent LLM inference systems to show today's difficulties with extensibility. Similar trends can be seen in training systems [18, 30, 36, 42], but we focus on inference as there is greater need for dynamicity.

As background, LLM inference is split into two phases: prefill, which processes an input prompt, and decode, which generates output tokens one token at a time. Typically, the system caches the state per prompt in the *key-value cache* (KV cache), whose size is proportionate to the context length.

(1) Placement flexibility. Large models are typically sharded across multiple GPUs. Tensor parallelism (TP) shards each layer, which reduces latency but requires more communication. Pipeline parallelism (PP) shards by layer and pipelines across devices, which improves throughput but not latency.

Even a single model replica's control plane is nontrivial to implement. The scheduler decides how to slice requests into batches of prefill and/or decode tokens. The scheduler must transfer metadata and data, e.g., request ID and prompt, to and from the workers without becoming a bottleneck.

For example, vLLM initially used a single controller design built on Ray, hosting the scheduler and each GPU worker in separate processes. However, Ray's dynamic RPC-based task dispatch caused high overheads from copying metadata first from the scheduler to each worker CPU, then from each worker CPU to GPU. Thus, *vLLM+Ray interpreted* has the worst throughput in the inference benchmark in Figure 2.

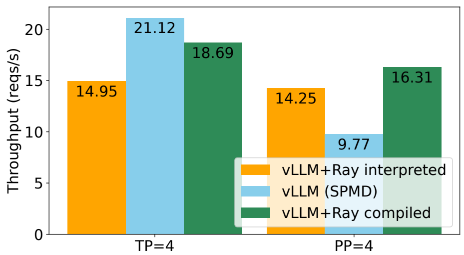


Figure 2: vLLM throughput with different control planes. *vllm+Ray* uses separate processes for the scheduler and workers, *vLLM (SPMD)* colocates the scheduler with a worker. *Ray interpreted* uses one RPC per task, while *Ray compiled* executes an entire task graph in one round-trip. Server: NVIDIA 4xA100 SXM 80GB; model: Llama 3.1 8B; dataset: ShareGPT.

To avoid these overheads, a later iteration of vLLM rewrote the system using SPMD, co-locating the vLLM scheduler with the rank 0 GPU worker. TP throughput improved, as the scheduler could copy just to its local GPU, then broadcast over inter-GPU links (*vLLM SPMD*, Figure 2). Unfortunately, this also worsened PP throughput: PP relies on the scheduler to keep the pipeline full, but the scheduler could now be delayed by the colocated worker's GPU execution.

TP vs. PP is the most basic placement decision. Strategies such as speculative decoding [17], mixture-of-experts [33], and offloading [15, 35] multiply the possible placement choices.

(2) Interoperability. Composition of ML systems can unlock new efficiency gains. Unfortunately, current frameworks are built monolithically and do not interoperate. This problem occurs even across deployments of the same framework. Prefill-decode (P-D) disaggregation, for example, places prefill and decode requests on different inference deployments for performance [28, 29, 45]. This requires transferring KV-cache between GPUs. Doing so efficiently requires coordination across different workers' local compute tasks.

Compound AI systems can also unlock new applications. Many post-training techniques apply reinforcement learning (RL) to LLMs, including RL from human feedback (RLHF) [27] and RL for reasoning [11]. These applications compose inference and training, which require different optimizations and frameworks. RL for LLMs also often requires multiple models that share resources for efficiency. For example, models may share model weights to reduce the GPU memory needed.

Thus, RL for LLMs must synchronize model weights and transfer the dynamically generated training data between workers and frameworks. Today, data ingest and egress requires patching these frameworks [34], and there is little support for more complex orchestration strategies such as for asynchronous RL algorithms.

(3) Codesign. The potential risk of introducing an intermediate framework for LLM inference is that we limit developers' ability to specialize to the workload. LLM inference is a fast-evolving field, and it is critical to support custom communication [1, 4] and compute [10, 41] kernels. Similarly, instead of restricting developers to one compiler stack, our goal is to interoperate with existing technologies such as torch.compile [6] and CUDA graphs [2].

The developer must also have control over system overheads. Higher control plane overheads can be acceptable during development, if the application is inherently dynamic, and/or if tasks and data transfers can be sufficiently pipelined. Ideally, the system should provide a smooth performance tradeoff between dynamic vs. static applications. For example, while LLM inference has static subroutines, such as one TP forward pass, developers must have the flexibility to dynamically invoke, modify, and compose these static subroutines, e.g., migrating KV-cache in response to load.

3 DAFT FOR DISTRIBUTED GPU PROGRAMMING

We address the problem of extending the DAFT API with first-class support for accelerators. Thus, we first briefly explain the challenges of accelerator-centric distributed execution using NVIDIA GPUs as an example.

3.1 Challenges of distributed GPUs

GPUs lack OS abstractions such as network sockets. Thus, many of the features that an OS would typically provide must be managed by the CPU. This produces the central tension: *extensibility calls for more CPU coordination across GPUs, but performance requires the CPU to run ahead of the GPU.*

We list some desirable features that an OS would normally help support. First, GPUs lack efficient CPU communication primitives such as variable-size messages, which require sending both metadata, i.e. the message size, and the data. This is because all data movement in and out of the GPU must be coordinated by the CPU. For example, the CPU must pre-allocate buffers of the correct size (line 8 in Figure 1a). Requiring metadata to be exchanged first may require expensive synchronization between the CPU and GPU.

Second, asynchronous GPU communication is challenging. While CUDA streams enable some concurrency, operations must still be serially executed if resources are oversubscribed. Kernels are also not preemptible. Thus, communication operations must be executed by each device in the same order to avoid deadlock and ideally simultaneously to avoid stalling [3]. The CPU must also coordinate overlapping GPU compute and communication [18, 42]. While one-sided GPU communication primitives backed by GPU RDMA can reduce some of this complexity, typically the CPU is still needed to coordinate the distributed GPU buffers, e.g., to prevent writing to a local buffer that a remote GPU is reading.

Third, GPU failure handling is complex and software support is less mature. Many distributed ML applications require collective GPU execution. Failures manifest as hung collective operations, likely blocking other operations too. Thus, at large scale, failures are difficult to root cause. Also, collective communication libraries such as NCCL [3] are expensive to (re)initialize and only support static membership.

Thus, while the SPMD program in Figure 1a is simple, it is also insufficient for scenarios involving variable-size tensors, asynchronous communication, and/or failure handling. For example, to extend Figure 1a to send tensors with different sizes and message handlers, the rank 1 process would need to know: (a) the order of tensor `sends`, (b) the handlers to execute, (c) the tensor shapes, and (d) when rank 0 begins `send`, to prevent idle waiting. While none of these are fundamentally impossible to implement, today's systems eventually implement some subset and do so differently, which duplicates effort and complicates interoperability.

3.2 DAFT

We propose using DAFT for distributed GPU programming. *Actors* are stateful workers that can be dynamically created and destroyed and are associated with one or more devices (line 1 in Figure 1b). Anyone with a handle to an actor can create an actor *task* by calling an RPC-like function. This returns a *distributed future* (`Ref` in Figure 1b), i.e. a reference to the eventual return value, which may reside in remote memory. `daft.get` blocks and fetches a copy of the `Ref`'s value to the caller. Values are immutable to ensure consistency.

A caller can asynchronously create multiple tasks before `get`-ting the results. `Refs` can be passed as arguments to other tasks to specify data dependencies. For example, to extend Figure 1b to transfer tensors with different sizes and message handlers, one could: (a) repeat the `A.execute.remote` call with different inputs, (b) specify different handlers by calling different tasks on `B`, (c) return tensors of any shape from `ModelA.execute`, while allowing the system to handle buffer allocation and (d) process synchronization.

GPU dataplane. The controller does not know at task submission time whether it returns CPU data, GPU data, or both. Thus, when a dependent task is submitted, the controller may not yet know how to coordinate the transfer. To solve this, we propose *type hints* to annotate function return values. For example, line 3 in Figure 1b hints that `ModelA.execute` returns a torch.Tensor to transfer via NCCL. The system can then schedule the tensor transfer from A to B at line 16, even before `A.execute` has finished. Ideally, the more hints the developer can provide, the lower the system overheads.

Table 1 further extends DAFT for collective communication. The caller can initialize a communicator for a group of actors, then pass the communicator and a list of `Refs` to collective operations such as `allreduce`. The application may specify a custom `Comm` implementation and can use custom communication kernels within a task definition.

init(List[Actor]) → Comm	Initialize the communicator.
allreduce(List[Ref], Opt[Comm]) → List[Ref]	Asynchronously allreduce a list of distributed futures, using the optional `Comm`.
TypeHint(Type, Comm, Shape[Type])	Annotation for actor methods that return GPU data with the given Type.
destroy(Comm)	Tear down the communicator.

Table 1: Example DAFT APIs for GPU communication.

Execution model. Under eager execution, the system will try to execute each task as soon as it is invoked. Eager execution is easier to program and debug but can lead to higher system overheads. For example, in Figure 1b, the system does not know how `tensor_ref` will be used when it is created at line 15. Thus, the tensor transfer from actor A to B cannot start until the controller invokes `B.execute` at line 16.

Some dataflow graphs may be provided entirely ahead of execution, instead of one task at a time. Then, lazy execution can effectively hide the above overheads, as discussed in [7].

When the dataflow graph is also executed many times on different inputs, *compilation* achieves further benefits because the system can perform operations such as scheduling and resource allocations at compile time. For example, the system could expose this functionality through a `compile` call that traces all DAFT calls (`.remote` and `.get`) in a user-defined function (e.g., `schedule` in Figure 1b). We discuss applications suited for compilation in Section 4.

Fault tolerance. Failure handling in current ML systems is relatively limited: training uses global checkpoint and restart [32] while many inference systems do not consider failures [20, 29, 45]. DAFT enables advanced fault tolerance schemes. The system is aware of data dependencies and could propagate failures such as actor crashes to downstream tasks and `get` calls. Upon detection, the controller could dynamically reconfigure using interpreted execution, then compile a new DAFT program for normal execution.

4 APPLICATIONS

We overview different applications and show how they can be executed as interpreted and/or compiled DAFT programs.

LLM inference. In TP inference (`tp_schedule`, Figure 3a), the scheduler sends the same input batch to all actors, and gathers the first result (all results are identical). In PP inference (`pp_schedule`), the scheduler sends the input batch to the first actor, and each actor sends its intermediate activations to the next actor. Pipelining is achieved by scheduling multiple batches concurrently.

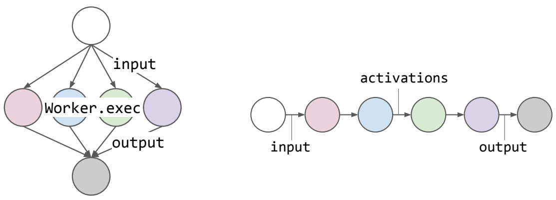
Both schedules produce a static dataflow graph (Figures 3b and 3c). The duration of each model forward pass may be

```python
workers = [Worker.remote(i) for i in range(W)]
def tp_schedule(input):
  results = [worker.exec.remote(input) for worker in workers]
  return daft.get(results)[0]

def pp_schedule(input):
  for worker in workers:
    input = worker.exec.remote(input)
  return daft.get(result)

def transfer(request, prefill_group: List[Worker],
             decode_group: List[Worker]):
  kv_cache_ref = prefill_group.get_kv.remote(request)
  return decode_group.enqueue.remote(request, kv_cache_ref)
```

(a) DAFT programs for LLM inference



(b) TP dataflow (c) PP dataflow

Figure 3: (a) DAFT programs for TP, PP, and KV-cache transfer in LLM inference. (b-c) The TP and PP dataflow graphs produced by (a). White nodes are inputs, gray nodes are outputs, each other color is one GPU task.

only 10s of ms. Thus, compiling the `tp/pp_schedule` functions is both suitable and critical for performance. In Figure 2, our compiled DAFT prototype shows that compiled execution improves TP throughput to within 12% of the SPMD-based vLLM, while improving PP throughput by 67%.

DAFT programs can also be composed. For example, Figure 3a (`transfer`) shows how to orchestrate a KV-cache transfer between actors. Eager execution enables dynamic selection of the prefill and decode `Worker` groups.

Distributed training. Training large models requires parallelism along many dimensions: TP, PP, data parallelism (DP), and more [22, 33, 36]. For example, in DP [18], each worker executes a model replica over a different data batch, then allreduces the gradients during the backwards pass (Figure 4). This can be expressed with DAFT in 10s of LoC.

Distributed training is mostly static, so compiled DAFT can eliminate the overheads of specifying and scheduling the dataflow graph. Compilation can also apply important optimizations such as compute-communication overlap [18]. To demonstrate this, Table 2 compares DP training throughput for ResNet152 [12], using PyTorch DDP [18] vs. Ray. As expected, *Ray Interpreted* has high overheads from unnecessarily copying data between GPU and CPU. *Ray Compiled* eliminates these overheads and overlaps compute and communication to achieve parity with *PyTorch DDP*.

RL for LLMs. RLHF can be expressed as a composition of the inference and training dataflow graphs in Figures 3 and 4. Composition can be done either dynamically, useful for elastic scaling, or statically, if control plane overheads become a bottleneck. By decoupling the data plane from the control plane, DAFT also allows the developer to declaratively choose how to transfer data, e.g., through collective communication (Table 1) or p2p transfers.

5 RELATED WORK

ML control planes. TensorFlow v1 [5] and Pathways [7] are ML systems that use a single-controller dataflow model. TensorFlow v1 can be used to distribute tensor dataflow graphs but not arbitrary code containing a mix of tensor and other application logic. Pathways is closed-source as of this paper's writing, but we believe it may have similar limitations. Our interpreted execution mode is inspired by Pathways' system design; we extend upon it with the DAFT API to support arbitrary application code.

Compared to other ML algorithms, RL requires dynamic task parallelism, which is difficult to support with SPMD [25]. Thus, systems such as RLlib [19] and HybridFlow [34] use Ray to orchestrate and compose tasks. Ray is a widely used interpreted DAFT system, but it lacks support for GPU-based distributed futures and compiled execution, and thus has limited applicability to the ML applications in Section 2.

Most other ML systems [9, 18, 36, 42, 44] are written in SPMD. Some of these systems support MPMD patterns such as pipeline parallelism (PP) but extensibility, e.g., modifying the PP schedule, is limited. Legion [8] supports the ML framework FlexFlow [24, 37] and provides an alternative API to SPMD and DAFT: a logically partitioned global address space with dynamic task parallelism.

ML compilers. Compilers [6, 16, 37, 40, 43] are widely used for distributing model training. Most of these systems compile to an SPMD program. Application of compilers to distributed inference is less common due to the need for custom communication kernels and dynamic execution. We hope that our work can augment compilers by providing a common distributed runtime for both training and inference.

6 DISCUSSION

Distributed memory management. The DAFT programs shown so far orchestrate *between* actors, leaving the user to manage each actor's local state. However, this usually includes GPU state that must be managed in concert with inter-GPU communication, e.g., to avoid OOM. For example, in fully sharded data parallelism (FSDP) [42], each actor's local state is one weights shard, which is allgather-ed during execution. The maximum amount to allgather at a time is limited by each actor's local memory usage.

We envision building an automatically managed distributed tensor store using DAFT. The user would write high-level

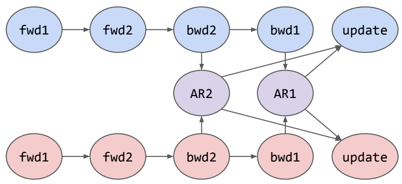


Figure 4: A dataflow graph for data-parallel training of a 2-layer model with two replicas. AR=allreduce.

	PyTorch DDP [18]	Ray Interpreted	Ray Compiled	Ray Compiled +overlap
Throughput (samples/s)	106	19	93	107

Table 2: Throughput for data-parallel training, averaged over 40 steps. Server: 4xNVIDIA L4 24GB GPUs; Model: ResNet-152. "+overlap" transparently overlaps communication and computation, a key optimization in PyTorch DDP [18].

code using distributed tensor references, while the system manages distributed memory operations such as rematerialization, replication, migration, and fault tolerance.

Compiler codesign. A primary challenge in building a distributed tensor frontend is performance; eagerly executing each distributed tensor operation with interpreted DAFT is simple but slow, requiring many controller round trips. A promising approach is to generate compiled DAFT fragments using a just-in-time (JIT) compiler frontend such as TorchDynamo [6]. Operations that can be compiled ahead of time are batched, while those that can't are executed eagerly.

Broadly, we see the codesign of compilers and distributed systems for ML workloads as a critical opportunity and challenge. Thus far, we have compilers that can reason about distributed operations [37, 40, 43]. But a pure compiler approach is limited: it is difficult to adapt to changes such as failures and to incorporate run-time information. Conversely, most distributed ML systems interpose at the orchestration layer and treat GPU compilation as a black box.

As an example, consider the problem of GPU multiplexing. Time slicing GPU kernels is difficult, and the mapping to compute units is opaque. Thus, interference between overlapped kernels, such as for compute and communication, can be difficult for a compiler to predict [46]. Meanwhile, reducing interference requires modifying application logic, which could be made transparent with compiler support.

Message passing. Message passing is an alternative API for composing ML systems. For example, a developer could encapsulate each system in an actor and orchestrate by sending messages in between. An advantage of this architecture is that it avoids any centralized bottleneck.

However, message passing is best suited to loosely coupled systems that can safely execute messages in any order. It is cumbersome when resources must be shared between systems, such as in RL for LLMs (Section 2). Users must also be careful to avoid deadlocks, which can be challenging when using collective GPU execution. For example, consider an LLM inference system using P-D disaggregation and TP within each replica. Such a system can deadlock if the GPUs in a decode replica receive KV cache for different requests in different orders, as the replica cannot execute a decode step until an entire request's KV cache is received.

DAFT is better suited to such cases because centralized scheduling allows for tight control over resources and simplifies deadlock prevention. However, while compilation can help mitigate run-time overheads, centralized scheduling can become the bottleneck, e.g., in large-scale and highly dynamic workloads. In this case, one could leverage a collective-aware message passing API to loosely couple distinct DAFT systems. Exploring the combination of the two approaches is a promising direction towards further scale.

Improving GPU primitives. To improve the efficiency of high-level primitives such as message passing or DAFT, we must also address limitations at the lower layers. For example, while one-sided GPU communication promises better performance, it remains constrained: developers must either build upon low-level Infiniband verbs, which may be complex and inefficient, or they can use higher-level libraries such as NVSHMEM or NCCL that restrict programmability. Others have proposed improvements on the former, e.g., offloading point-to-point GPU communication to the hardware to reserve valuable SMs for overlapped compute [21]. We also identify potential improvements for collective libraries such as NCCL, including: (1) dynamic process membership without expensive reinitialization, (2) tagged asynchronous communication operations to reduce coordination overhead, (3) kernel preemption to mitigate deadlocks, and (4) improved error propagation to support failure detection and handling. These features can ease the goal of achieving flexible orchestration without sacrificing performance.

Conclusion. As ML systems have grown in scale and complexity, it is no longer feasible to codesign entire systems with narrowly defined workloads. Meanwhile, the unique properties of distributed accelerator execution make it challenging to efficiently apply known techniques for system extensibility: modularity, composition, and layering. Solving this challenge will require a combination of techniques across distributed systems, compilers, and more, and it is critical to the evolution of future applications.

ACKNOWLEDGEMENTS

We thank our anonymous reviewers for their insightful feedback. We also thank Tom Anderson, Arvind Krishnamurthy, and Zihao Ye for their helpful discussion and comments. This work was supported in part by UW FOCI and its partners (Alibaba, Amazon, Cisco, Google, Microsoft, and VMware).

REFERENCES

[1] 3x faster allreduce with nvswitch and tensorrt-llm multishot. https://developer.nvidia.com/blog/3x-faster-allreduce-with-nvswitch-and-tensorrt-llm-multishot/. Accessed: 2025-01-15.

[2] Getting started with cuda graphs. https://developer.nvidia.com/blog/cuda-graphs/. Accessed: 2025-01-15.

[3] NVIDIA Collective Communications Library. https://developer.nvidia.com/NCCL.

[4] vllm github: Custom all reduce kernels pr#2192. https://github.com/vllm-project/vllm/pull/2192. Accessed: 2025-01-15.

[5] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Savannah, Georgia, USA*, 2016.

[6] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024.

[7] Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Dan Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, Brennan Saeta, Parker Schuh, Ryan Sepassi, Laurent El Shafey, Chandramohan A. Thekkath, and Yonghui Wu. Pathways: Asynchronous distributed dataflow for ml. In *Proceedings of the Fourth Annual Conference on Machine Learning and Systems*, 2022.

[8] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2012.

[9] Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Zhe Fu, et al. Deepseek llm: Scaling open-source language models with longtermism. *arXiv preprint arXiv:2401.02954*, 2024.

[10] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.

[11] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[13] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. Gpipe: efficient training of giant neural networks using pipeline parallelism. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, 2019.

[14] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023.

[15] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. {InfiniGen}: Efficient generative inference of large language models with dynamic {KV} cache management. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 155–172, 2024.

[16] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020.

[17] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *Proceedings of the 40th International Conference on Machine Learning*, 2023.

[18] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. Pytorch distributed: experiences on accelerating data parallel training. *Proceedings of the VLDB Endowment*, 13(12), 2020.

[19] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph E. Gonzalez, Michael I. Jordan, and Ion Stoica. RLlib: Abstractions for distributed reinforcement learning. In *International Conference on Machine Learning (ICML)*, 2018.

[20] Bin Lin, Chen Zhang, Tao Peng, Hanyu Zhao, Wencong Xiao, Minmin Sun, Anmin Liu, Zhipeng Zhang, Lanbo Li, Xiafei Qiu, et al. Infinite-llm: Efficient llm service for long context with distattention and distributed kvcache. *arXiv preprint arXiv:2401.02669*, 2024.

[21] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[22] Hao Liu, Matei Zaharia, and Pieter Abbeel. Ring attention with blockwise transformers for near-infinite context. *arXiv preprint arXiv:2310.01889*, 2023.

[23] Frank Sifei Luan, Stephanie Wang, Samyukta Yagati, Sean Kim, Kenneth Lien, Isaac Ong, Tony Hong, Sangbin Cho, Eric Liang, and Ion Stoica. Exoshuffle: An extensible shuffle architecture. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 564–577, 2023.

[24] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, et al. Specinfer: Accelerating generative large language model serving with tree-based speculative inference and verification. *arXiv preprint arXiv:2305.09781*, 2023.

[25] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, Carlsbad, CA, 2018. USENIX Association.

[26] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. CIEL: A universal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 113–126, Berkeley, CA, USA, 2011. USENIX Association.

[27] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

[28] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024.

[29] Ruoyu Qin, Zheming Li, Weiran He, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: A kvcache-centric disaggregated architecture for llm serving. *arXiv preprint arXiv:2407.00079*, 2024.

[30] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimization towards training A trillion parameter models. *CoRR*, abs/1910.02054, 2019.

[31] 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)*, 2021.

[32] Elvis Rojas, Albert Njoroge Kahira, Esteban Meneses, Leonardo Bautista Gomez, and Rosa M Badia. A study of checkpointing in large scale training of deep neural networks. *arXiv preprint arXiv:2012.00825*, 2020.

[33] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc V. Le, Geoffrey E. Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. 2017.

[34] Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. In *Proceedings of the 20th ACM European Conference on Computer Systems*, 2025. Accepted. arXiv:2409.19256.

[35] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*, pages 31094–31116. PMLR, 2023.

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

[37] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, Xi Luo, Dheevatsa Mudigere, Jongsoo Park, Misha Smelyanskiy, and Alex Aiken. Unity: Accelerating DNN training through joint optimization of algebraic transformations and parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022.

[38] Stephanie Wang, Benjamin Hindman, and Ion Stoica. In reference to rpc: it's time to add distributed memory. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2021.

[39] Stephanie Wang, Eric Liang, Edward Oakes, Ben Hindman, Frank Sifei Luan, Audrey Cheng, and Ion Stoica. Ownership: A distributed futures system for fine-grained tasks. 2021.

[40] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, et al. Gspmd: general and scalable parallelization for ml computation graphs. *arXiv preprint arXiv:2105.04663*, 2021.

[41] Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, et al. Flashinfer: Efficient and customizable attention engine for llm inference serving. *arXiv preprint arXiv:2501.01005*, 2025.

[42] 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. *Proceedings of the VLDB Endowment*, 16(12), 2023.

[43] 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)*, 2022.

[44] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody_Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Efficiently programming large language models using sglang. 2023.

[45] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. Distserve: disaggregating prefill and decoding for goodput-optimized large language model serving. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation*, 2025.

[46] Kan Zhu, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Yufei Gao, Qinyu Xu, Tian Tang, Zihao Ye, Keisuke Kamahori, Chien-Yu Lin, Ziren Wang, Stephanie Wang, Arvind Krishnamurthy, and Baris Kasikci. Nanoflow: Exploiting intra-device parallelism for high throughput large language model serving. 2025.