Towards ML System Extensibility

Weixin Deng, Andy Ruan, Megan Frisella, Kai-Hsun Chen, SangBin Cho,
Jack Tigar Humphries, Rui Qiao, **Stephanie Wang**

PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

RAY
Anyscale

The rise of large models

Distributed optimizations for ML systems:

Trend: Current distributed ML systems focus on **performance** at the cost of **extensibility**.

But can current systems meet future application demands?

The extensibility problem

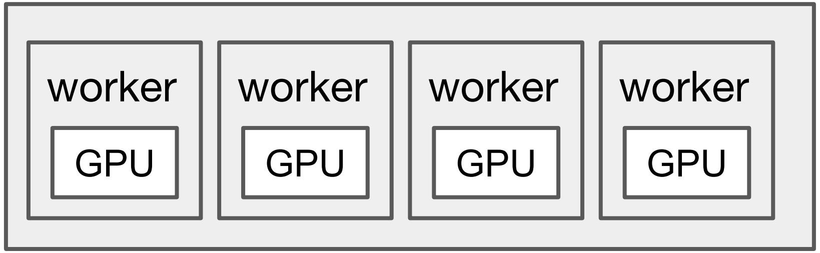
1. **Codesign:** Can the developer introduce performance optimizations specialized to the workload?
2. **Placement flexibility:** Can the developer control when and where computations should execute?
3. **Interoperability:** Can the system be easily and efficiently composed with other systems?

The extensibility problem

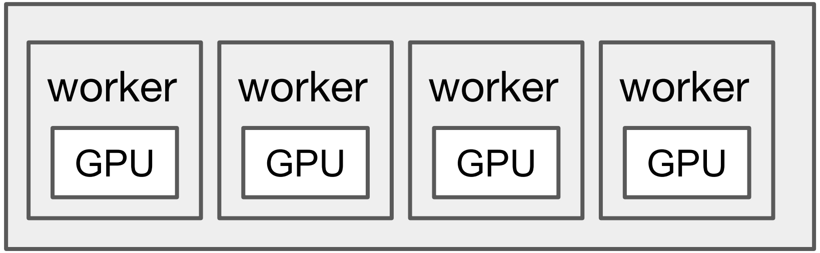
1. **Codesign:** Can the developer introduce performance optimizations specialized to the workload?
2. **Placement flexibility:** Can the developer control when and where computations should execute?
3. **Interoperability:** Can the system be easily and efficiently composed with other systems?

Distributed ML frameworks today

Training
DeepSpeed, Megatron-LM,
PyTorch FSDP, …



Inference
vLLM, SGLang,
TensorRT-LLM, …



Distributed execution strategies interact in complex ways.
→ **Uneven support across frameworks.**

Placement flexibility in distributed training

	Megatron-LM	DeepSpeed	PyTorch
2019	Data parallelism (DP) **Tensor parallelism (TP)**	DP	DP
2020	Pipeline parallelism (PP)	**ZeRO / FSDP** PP	
2021		EP	
2022	Sequence parallelism (SP) Expert parallelism (EP)		**ZeRO / FSDP**
2023	Context parallelism (CP)	**TP** CP	
2024	**ZeRO / FSDP**		

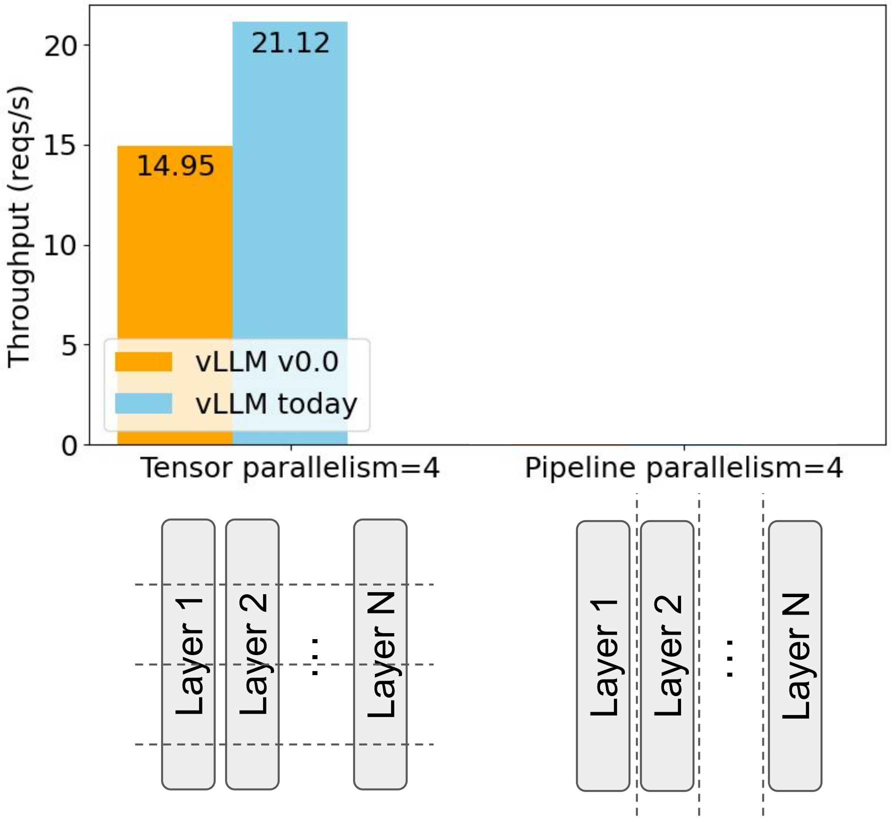
Placement flexibility in distributed training

	Megatron-LM	DeepSpeed	PyTorch
2019	Data parallelism (DP) **Tensor parallelism (TP)**	DP	DP
2020	Pipeline parallelism (PP)	**ZeRO / FSDP** PP	
2021		EP	
2022	Sequence parallelism (SP) Expert parallelism (EP)		**ZeRO / FSDP**
2023	Context parallelism (CP)	**TP** CP	
2024	**ZeRO / FSDP**		

Not including time to:

- Optimize performance
- Support variants of each strategy
- Ensure feature compatibility
- Support more models
- Fix bugs
- …

Placement flexibility in LLM inference



- Tensor parallelism (TP): Shard within a layer
- Pipeline parallelism (PP): Shard by layer

→ Improving support for one placement worsened performance for the other.

The extensibility problem

1. **Codesign:** Can the developer introduce performance optimizations specialized to the workload?

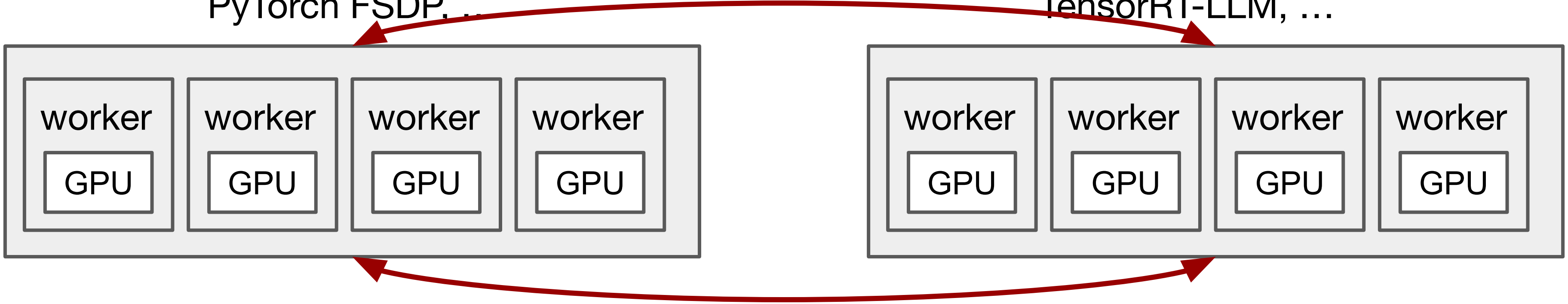
2. **Placement flexibility:** Can the developer control when and where computations should execute?

3. **Interoperability:** Can the system be easily and efficiently composed with other systems?

Distributed ML frameworks today

Training
DeepSpeed, Megatron-LM,
PyTorch FSDP, …

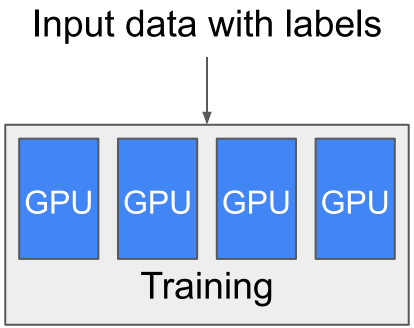
Inference
vLLM, SGLang,
TensorRT-LLM, …



Frameworks are built monolithically.
→ **Difficult to efficiently compose models and frameworks.**

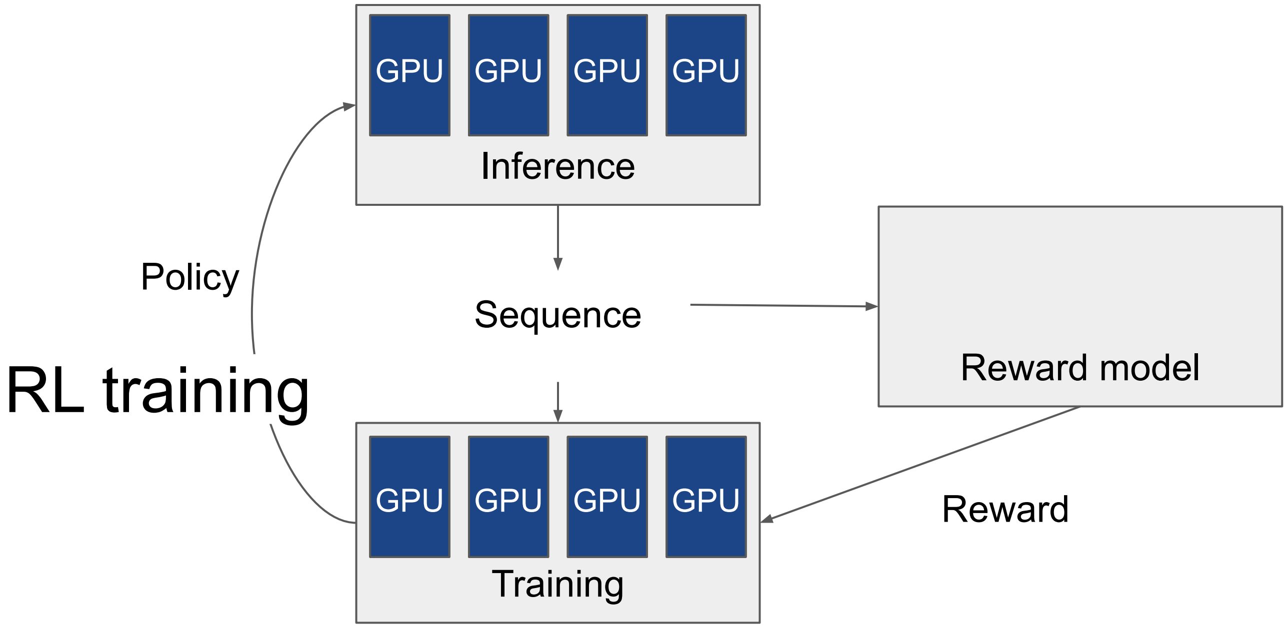
Interoperability in RL for LLMs

Supervised learning



Interoperability in RL for LLMs

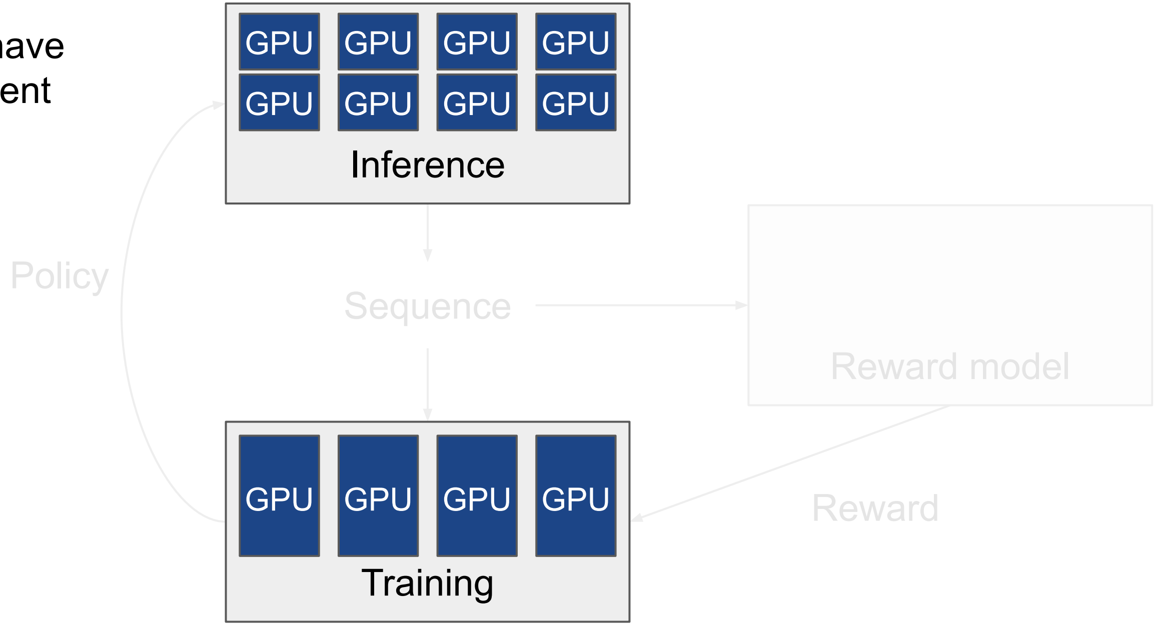
Reinforcement learning (RL) for LLMs requires **composition.**



Interoperability in RL for LLMs

Reinforcement learning (RL) for LLMs requires **composition.**

1. Training and inference have different optimal placement strategies

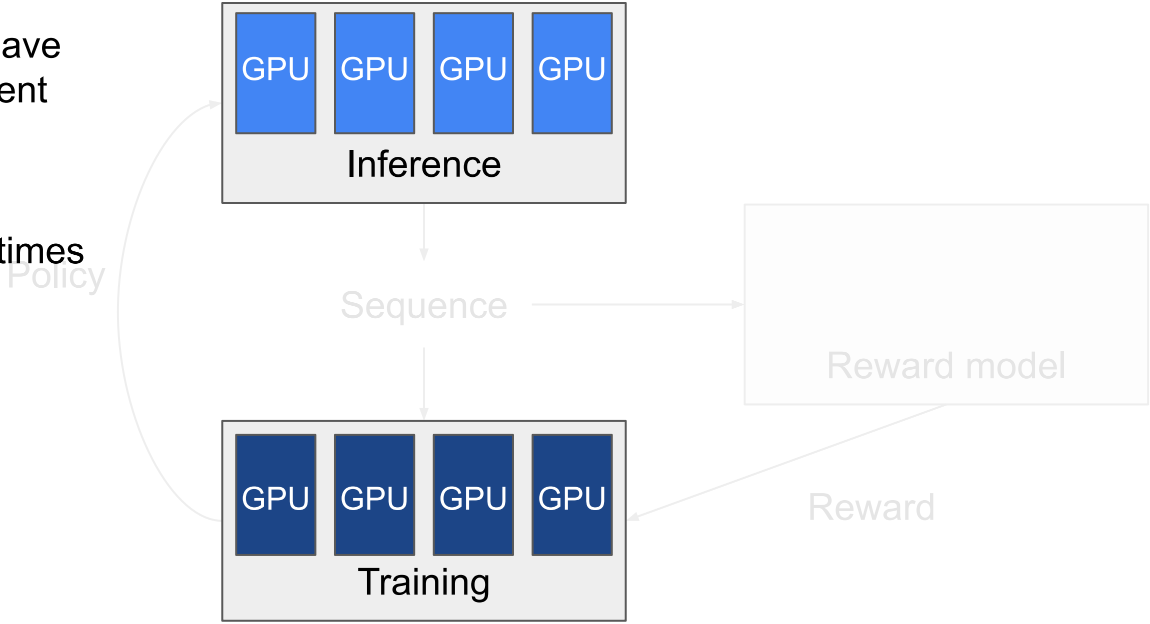


Interoperability in RL for LLMs

Reinforcement learning (RL) for LLMs requires **composition.**

1. Training and inference have different optimal placement strategies

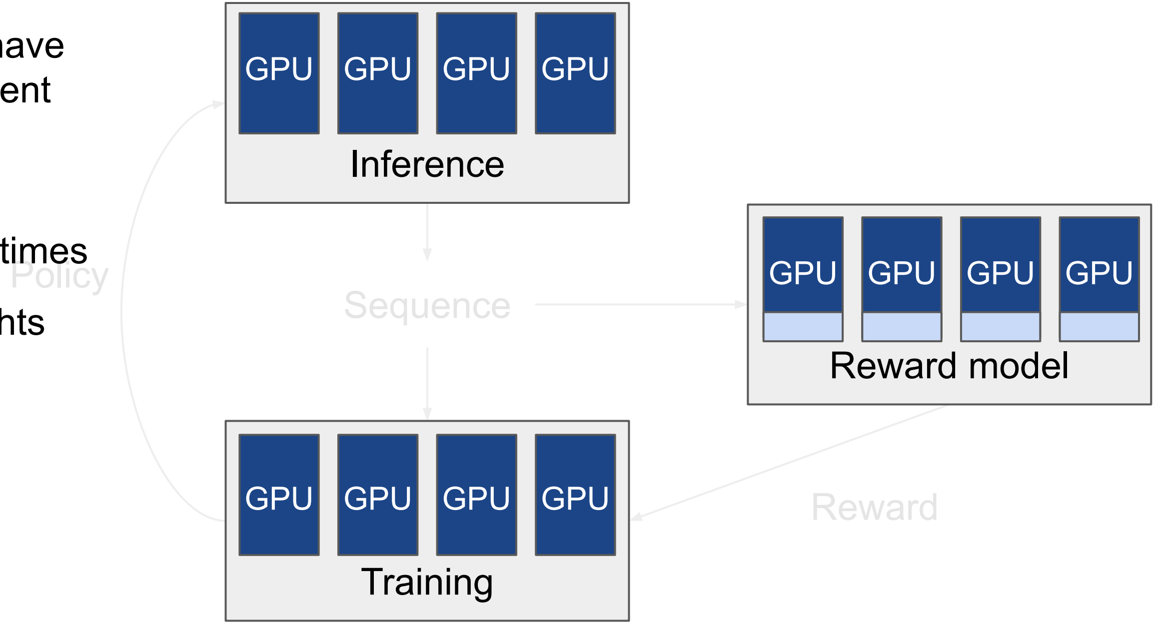
2. Different algorithms synchronize at different times



Interoperability in RL for LLMs

Reinforcement learning (RL) for LLMs requires **composition.**

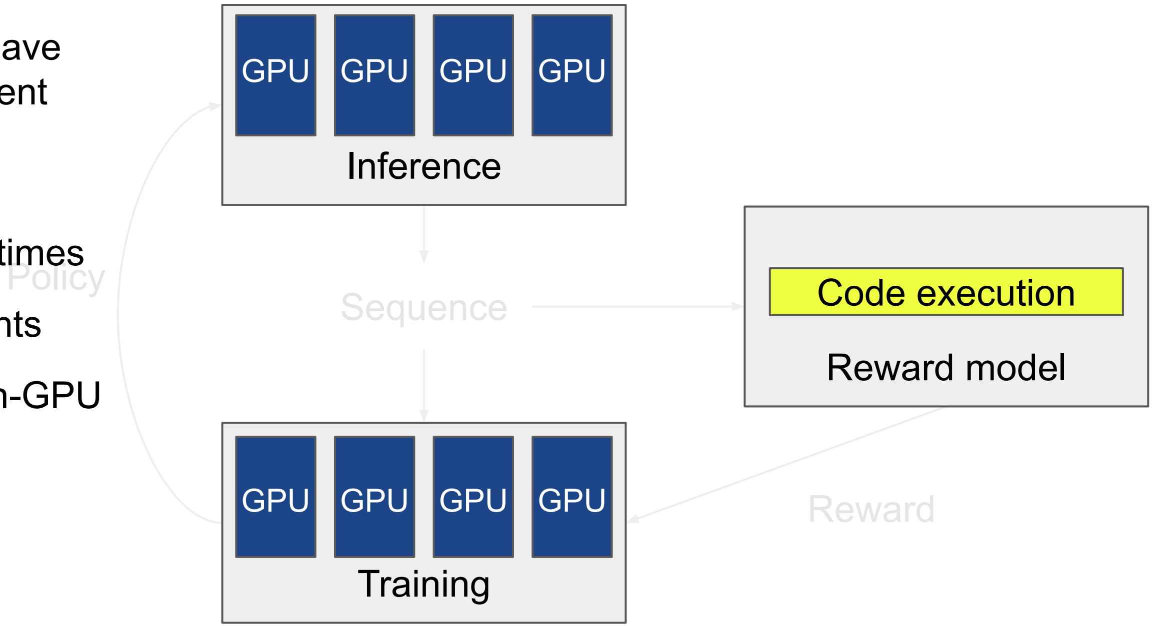
1. Training and inference have different optimal placement strategies
2. Different algorithms synchronize at different times
3. Models may share weights



Interoperability in RL for LLMs

Reinforcement learning (RL) for LLMs requires **composition.**

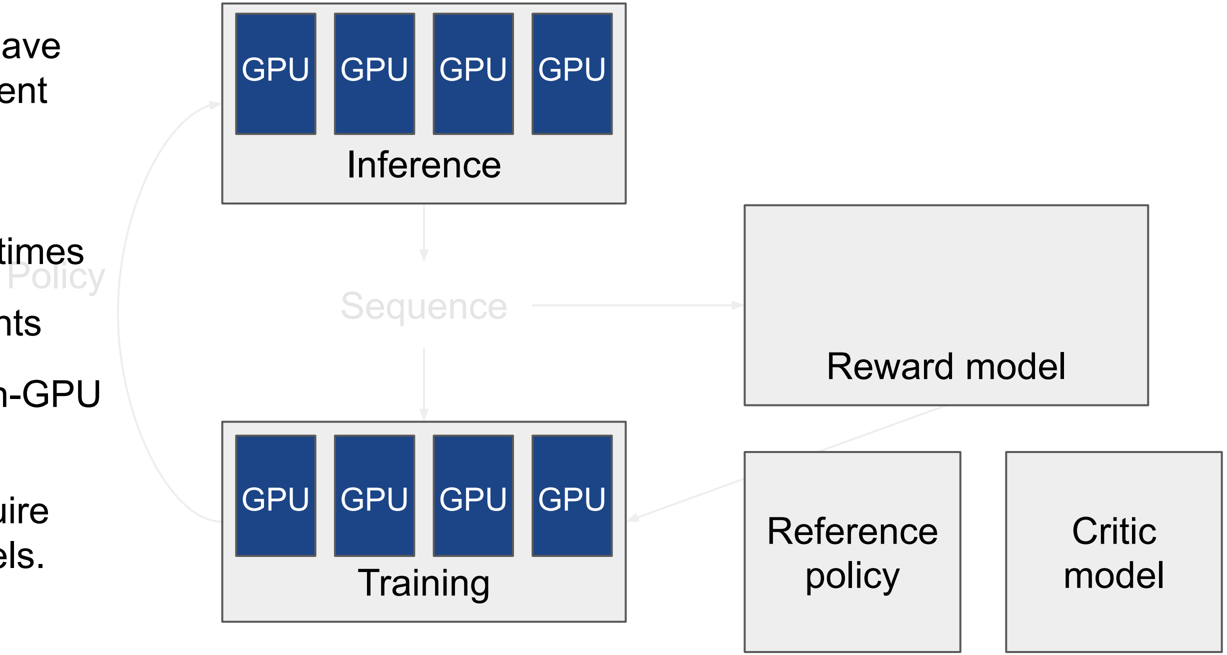
1. Training and inference have different optimal placement strategies
2. Different algorithms synchronize at different times
3. Models may share weights
4. Reward may require non-GPU resources



Interoperability in RL for LLMs

Reinforcement learning (RL) for LLMs requires **composition.**

1. Training and inference have different optimal placement strategies
2. Different algorithms synchronize at different times
3. Models may share weights
4. Reward may require non-GPU resources
5. Different algorithms require different additional models.



Interoperability in RL for LLMs

Reinforcement learning (RL) for LLMs requires **composition.**

1. Training and inference have different optimal placement strategies
2. Different algorithms synchronize at different times
3. Models may share weights
4. Reward may require non-GPU resources
5. Different algorithms require different additional models.
6. …

Diverse composition strategies

→ **Diverse data movement and scheduling strategies**

The extensibility problem

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

Current distributed ML systems use **single program multiple data**.
→ **Codesign** at the cost of **placement flexibility** and **interoperability**.

SPMD: Single program, multiple data

SPMD: Each accelerator runs a copy of the same program.

```
# Compose two models A and B
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)                        A
sync
  else:
    tensor = torch.zeros(N)
    recv(tensor, peer=0)
    output = model.execute(tensor)              B
```

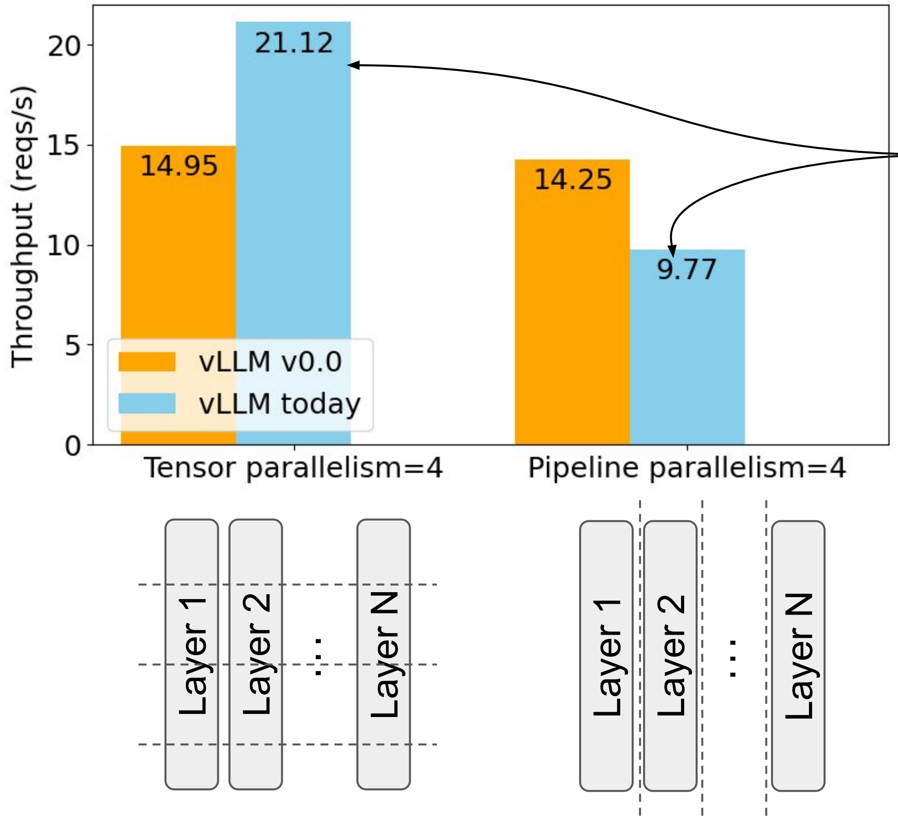
Pros:

- Simple
- Efficient

Cons:

- Couples user code to a *specific placement*
- Couples user code to a *static* execution strategy
- *Composition* is difficult

Placement flexibility in LLM inference

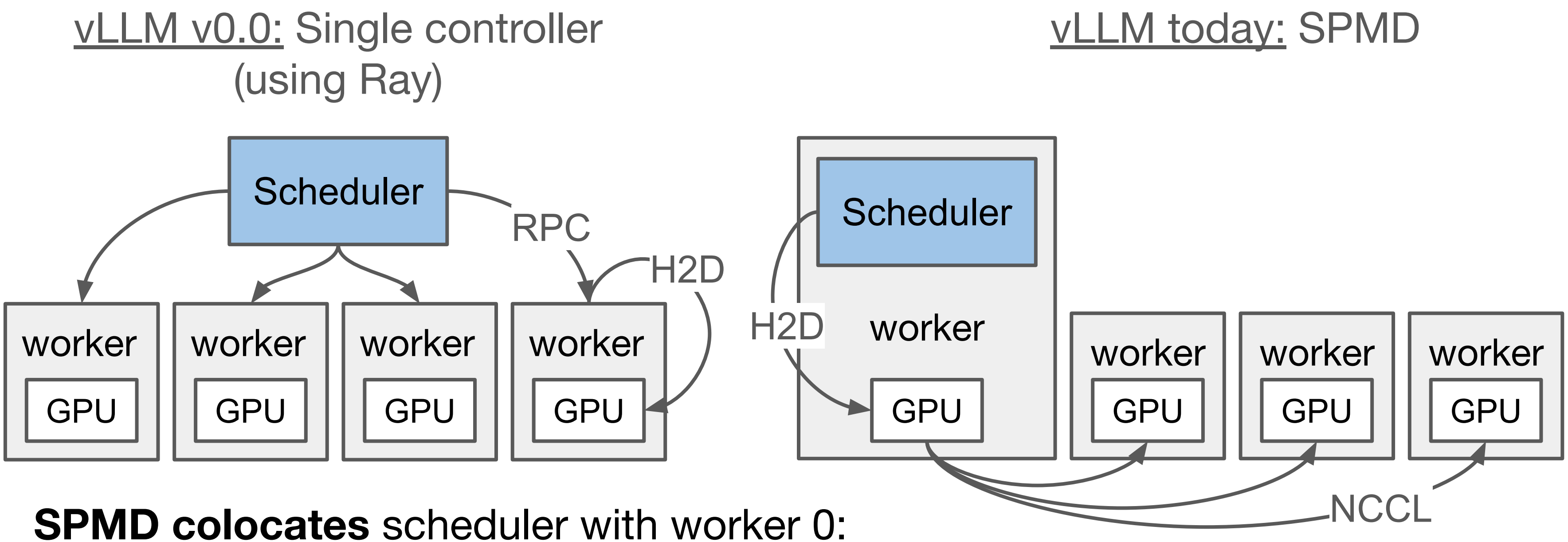


Using SPMD

TP vs PP is a **placement** decision.

Improving support for one placement worsened performance for the other.

Placement flexibility: Single controller vs. SPMD

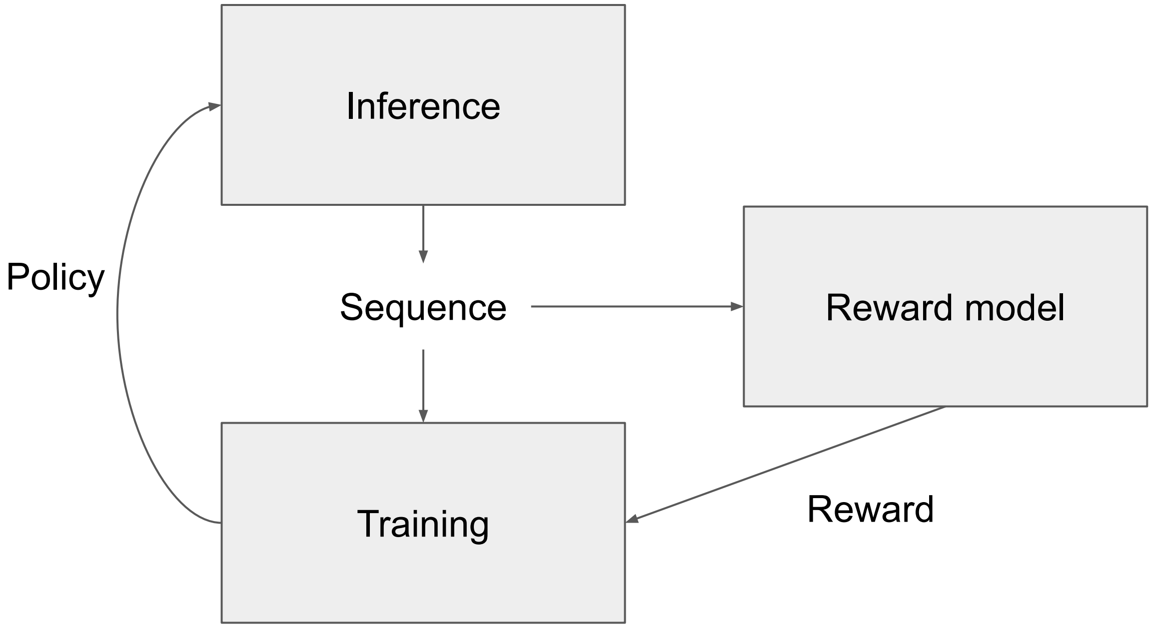


SPMD colocates scheduler with worker 0:

- **Reduces overhead** of metadata transfer
- But **couples** the scheduler and worker

Interoperability: SPMD for composition?

Reinforcement learning (RL) for LLMs requires **composition.**



Interoperability: SPMD for composition?

```
def train(self, …):
  def vllm_generate(...):
    r = vllm_engine.generate.remote(...)
    queue.put(ray.get(r))
  def broadcast_to_vllm(...):
    for param in self.model.weights:
      dist.broadcast(param, 0, group=...)

  threading.Thread(vllm_generate).start()
  for _ in range(training_steps):
    broadcast_to_vllm(...)
    g_vllm_responses = queue.get()
    dist.broadcast(g_vllm_responses, 0)
    ...
```

Inference

Weight syncing

Data transfer

Training

Interoperability: SPMD for composition?

```
def train(self, …):
  def vllm_generate(...):
    r = vllm_engine.generate.remote(...)
    queue.put(ray.get(r))
  def broadcast_to_vllm(...):
    for param in self.model.weights:
      dist.broadcast(param, 0, group=...)

  threading.Thread(vllm_generate).start()
  for _ in range(training_steps):
    broadcast_to_vllm(...)
    g_vllm_responses = queue.get()
    dist.broadcast(g_vllm_responses, 0)
    ...
```

- **Tightly coupled** to particular algorithm (e.g., on-policy vs. off-policy)
- Collective ops and groups need to be **manually scheduled**
- **Additional optimizations** are challenging: reducing data movement, elastic scaling, …

SPMD: Single program, multiple data

SPMD: Each accelerator runs a copy of the same program.

```python
# Compose two models A and B
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)
```

Does not support:

- Variable-size tensors
- Asynchronous communication
- Failure handling
- CPU metadata + GPU data
- …

SPMD: Single program, multiple data

SPMD: Each accelerator runs a copy of the same program.

```
# Compose two models A and B
```

Improving **extensibility** requires more CPU coordination.
But **performance** requires GPUs to run ahead of CPUs.

Static, inflexible and fast vs.
Dynamic, flexible and slow

```
tensor = torch.zeros(N)
recv(tensor, peer=0)
output = model.execute(tensor)
```

data

- …

The extensibility problem

SPMD:

→ works well for **codesign** with **one to few static** strategies

→ is difficult to adapt to different **placement** choices

→ discourages **interoperability**, because composition requires changes to each program to implement the global schedule

DAFT: An intermediate representation for distributed GPU programming

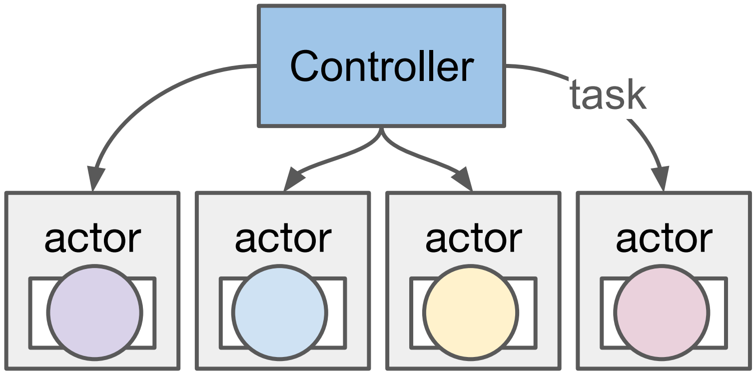
DAFT

Distributed: Program distributed GPUs with a "single controller" program

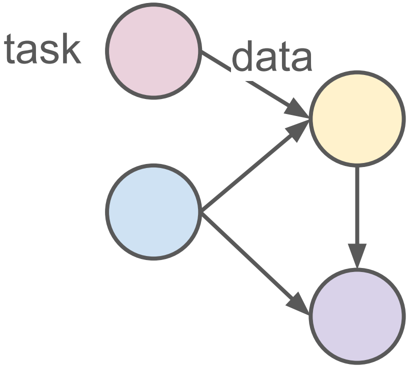
Actors: Stateful and remote workers, wrap any framework

Futures: Async execution, dataflow programming

Tasks: RPC-like interface



Dataflow graph



Related work

- Most other distributed ML control planes based on SPMD or limited MPMD
- Single controller frameworks:
  - Pathways, TensorFlow v1: Tied to XLA compiler
  - RLlib, Hybridflow (veRL): RL-specific
- Previous DAFT systems are CPU-centric: Ray, Ciel, Dask

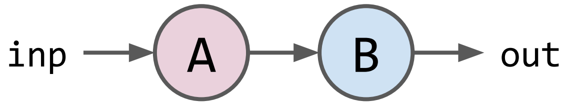
→ **OS** coordinates **CPU** execution and communication

→ **CPU** coordinates **GPU** execution and communication

A DAFT example (using Ray)

```
@ray.remote(num_gpus=1)
class ModelA:
  @ray.method(tensor_transport="nccl")
  def execute(self, input: torch.Tensor) -> torch.Tensor:
    return self.model.execute(input)
```

Dataflow graph



Distributed future: Reference to remote, eventual value

System triggers p2p (or collective communication) operation

A DAFT example (using Ray)

```python
@ray.remote(num_gpus=1)
class ModelA:
  @ray.method(tensor_transport="nccl")
  def execute(self, input: torch.Tensor) -> torch.Tensor:
    return self.model.execute(input)

@ray.remote(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.Tensor = torch.rand(N, device="cuda")
  tensor_ref: Ref = A.execute.remote(inp)
  output_ref: Ref = B.execute.remote(tensor_ref)
  out: torch.Tensor = ray.get(output_ref)
```

Already supports:

- Variable-size tensors
- Asynchronous communication
- Failure handling
- CPU metadata + GPU data
- …

DAFT for composition in RL for LLMs

```
# Multi-GPU inference replica.
vllm_workers: WorkerGroup[vllm.LLMEngine]
# Multi-GPU training replica.
train_workers: WorkerGroup[Trainer]

# Single controller program.
def train():
```

+ Algorithm **decoupled** from worker code
+ **Transparent scheduling** for collectives
+ Reduce data movement by creating different dataflows

Elastic scaling of WorkerGroups

DAFT as an IR for distributed GPUs

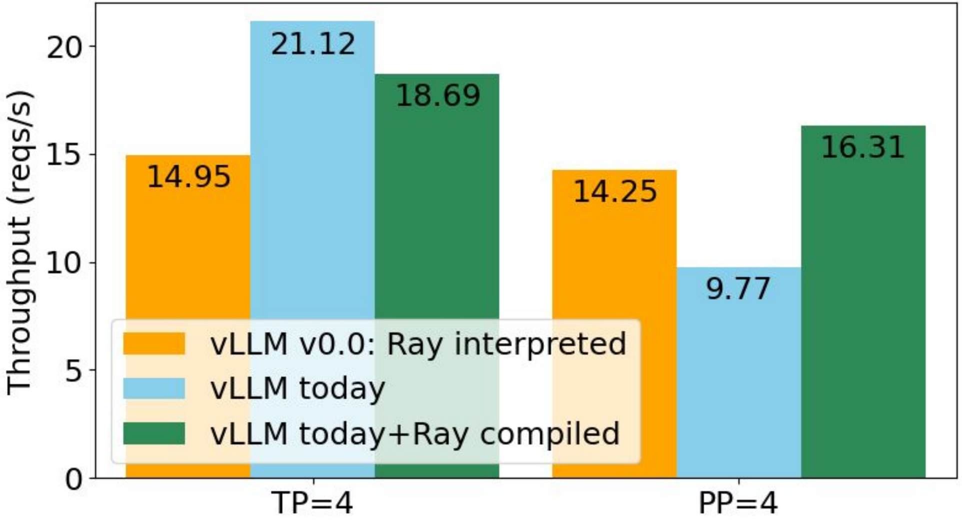
Problem: Dynamic dispatch can add high overheads compared to SPMD.

Solution: Interpreted vs. compiled execution.

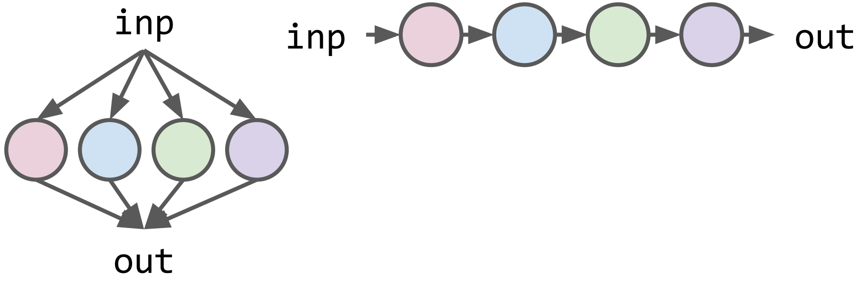
- **Interpreted**: Program executes eagerly, one task at a time
  → For coarse-grained composition, debugging, dynamic failover, …
- **Compiled**: Freeze a dataflow (sub)graph, schedule all tasks in one round-trip
  → For fine-grained GPU orchestration, static control flow

Allow user to **control** tradeoff between **dynamicity vs. system overheads**!

Interpreted vs. Compiled DAFT



DAFT dataflows



Control plane optimizations:

- Reduce communication overhead by using shared memory
- Reduce scheduling overhead by executing multiple tasks in one round trip
- …

Open systems challenges

A distributed tensor runtime

Distributed tensor API	User-controllable distributed strategy
Compiler frontend	Extract non-distributed static tensor subgraphs
Distributed tensor runtime	Apply distributed strategy, plan distributed ops
Interpreted and compiled DAFT	Intermediate representation for distributed GPU programming

Opportunity: Codesigning compilers and distributed systems?

Message passing APIs

An alternative API to DAFT for application composition:

- **Scales** better than single controller
- Best for loosely coupled systems/applications

But there is no good message-passing APIs for distributed GPUs:

- Messages can be executed in any order → **collective operations** may deadlock

And current collective communication libraries are limited :

- Statically declared collective groups
- User must manually schedule calls to the library

Opportunity: A message-passing API for distributed GPUs?

The rise of large models

Models are getting larger.

But also more *complicated*: RL, multimodal models, multi-model routing, heterogeneous resources, …

Performance is important.
But so is **extensibility**!