

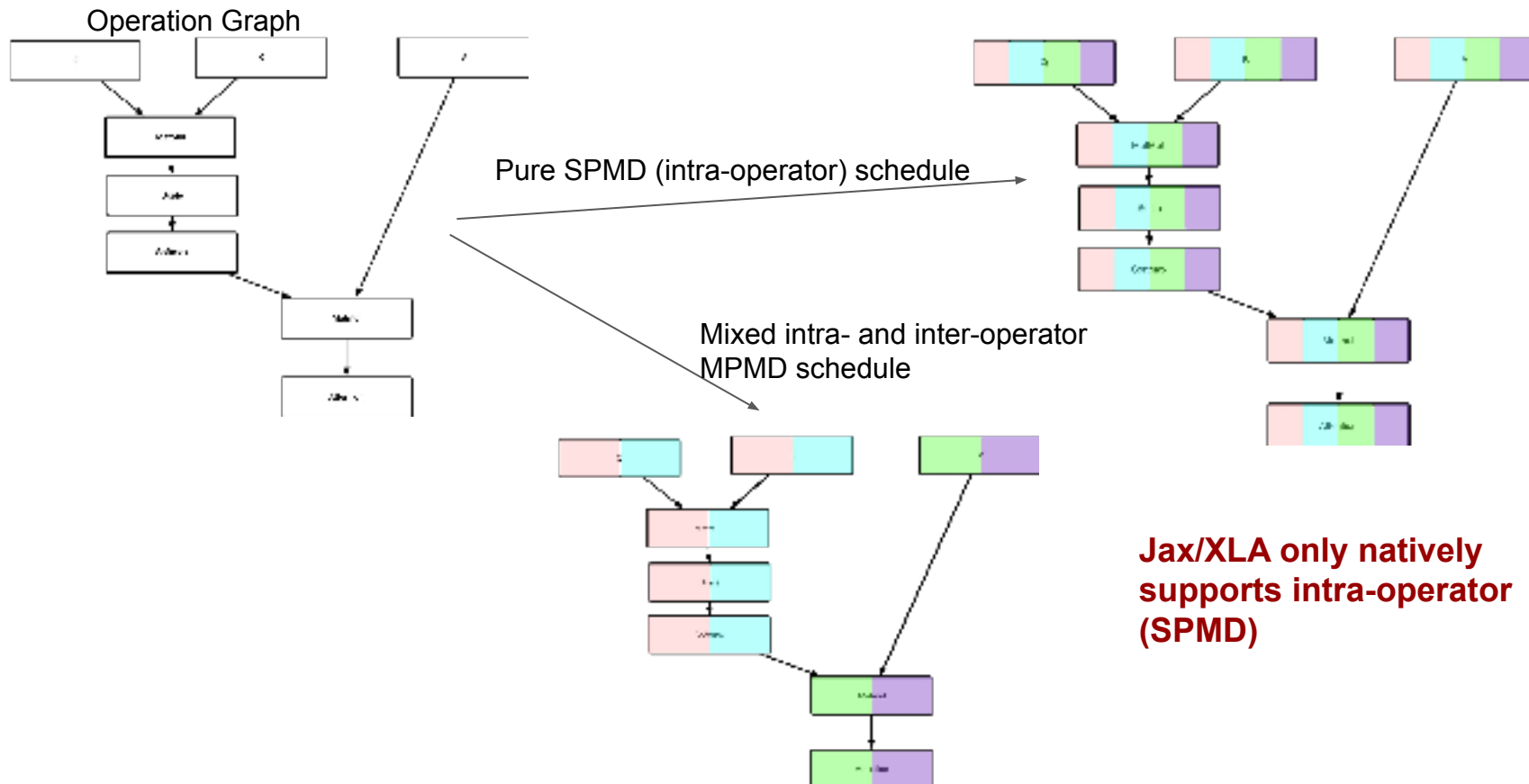


# The framework formerly known as Legate-Jax

Dec 4, 2024: Legion Retreat

Jeremiah Wilke | Wonchan Lee | Malte Förster

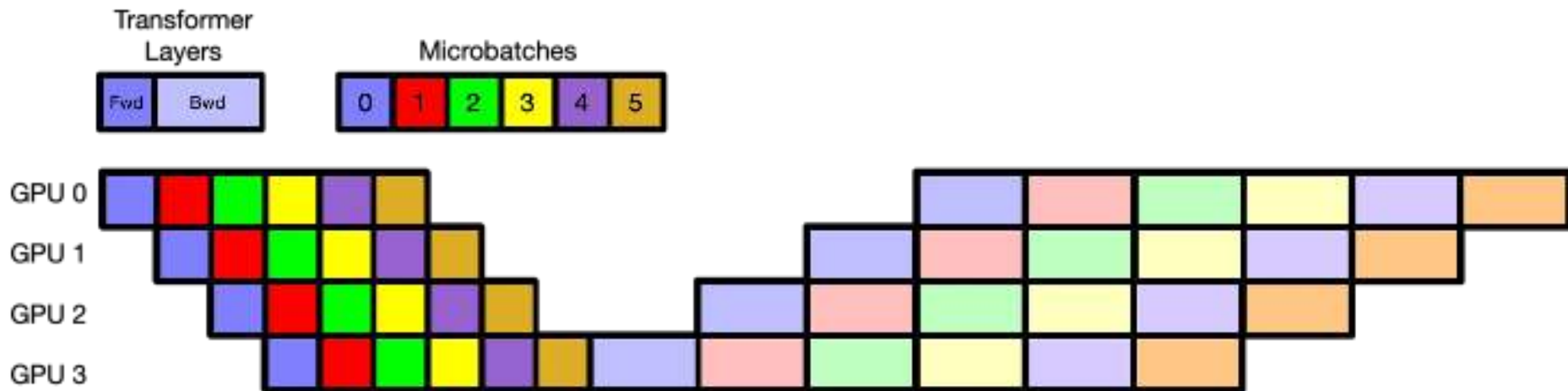
# Goal of Jax work is to enable arbitrary inter- and intra-operator parallelism via auto or user-guided parallelization



# Goal: improve performance for large-scale training (256-100K GPUs) via MPMD parallelism

- Realm *execution model* improves perf
  - Keep *device* executing critical path overlapped with *host-driven* off-critical path tasks
- Legate *programming model* makes complicated patterns easier to express
  - 1F1B, load balancing irregular stages, multiple overlapping communications naturally described with a sequential semantic + mapper
- Make MPMD execution easy to express in Jax *user-level libraries*
  - Simple decorators for defining arbitrary MPMD event graphs in sequential user code

Pipeline parallelism (with exceptions) is inter-operator parallelism (MPMD)



Jax + Legate programming model share  
a core philosophy

# Jax parallelization starts from sequential user code, finishes with asynchronous parallel execution

```
import jax.numpy as jnp

def layer(mlp, x)
    x = jnp.einsum("bh,hm->bm", x, mlp)

def model(params, batch):
    mlp0, mlp1 = params
    x, labels = batch
    x = layer(x)
    x = layer(x)
    diff = x - labels
    return (diff*diff).sum()

loss = jax.jit(model)(params, batch)
```

**Sequential code  
defining the model  
(no parallelism yet)**

JIT compilation of  
model to GPU device  
code

# Jax parallelization starts from sequential user code, finishes with asynchronous parallel execution

```
import jax.numpy as jnp
from jax import with_sharding_constraint as shard
from jax.sharding import PartitionSpec as P
```

```
def layer(mlp, x)
    mlp = shard(mlpx, P("model", None))
    x = jnp.einsum("bh,hm->bm", x, mlp)
    return shard(x, P("batch", "model"))
```

Assign names to the  
tensor axes to use for  
sharding later

```
def model(params, batch):
    mlp0, mlp1 = params
    x, labels = batch
    x = layer(x)
    x = layer(x)
    diff = x - labels
    return (diff*diff).sum()
```

```
params = init()
batch = load_batch()
```

```
loss = jax.jit(model)(params, batch)
```

# Jax parallelization starts from sequential user code, finishes with asynchronous parallel execution

```
import jax.numpy as jnp
from jax import with_sharding_constraint as shard
from jax.sharding import PartitionSpec as P, Mesh
```

```
def layer(mlp, x)
    mlp = shard(mlpx, P("model", None))
    x = jnp.einsum("bh,hm->bm", x, mlp)
    return shard(x, P("batch", "model"))
```

```
def model(params, batch):
    mlp0, mlp1 = params
    x, labels = batch
    x = layer(x)
    x = layer(x)
    diff = x - labels
    return (diff*diff).sum()
```

```
params = init()
batch = load_batch()
```

```
devices = np.array(jax.devices()).reshape(2,2)
with Mesh(devices, ('batch', 'model')):
    loss = jax.jit(model)(params, batch)
```

Define logical to  
physical mapping of  
axis names in  
device msh

Define device  
mesh as 2x2

jitted function  
executes with  
sharded data in  
parallel



# Jax and Legate/Legion share a core philosophy

- Write sequential code
- Define partitions (shardings) on tensors
- Let the compiler/runtime system automatically infer and schedule parallelism

# Jax requires a uniform, global mesh for all tensors which prevents MPMD and pipeline parallelism

```
import jax.numpy as jnp
from jax import with_sharding_constraint as shard
from jax.sharding import PartitionSpec as P, Mesh
```

```
def layer(mlp, x)
    mlp = shard(mlpx, P("model", None))
    x = jnp.einsum("bh,hm->bm", x, mlp)
    return shard(x, P("batch", "model"))
```

```
def model(params, batch):
    mlp0, mlp1 = params
    x, labels = batch
    x = layer(x)
    x = layer(x)
    diff = x - labels
    return (diff*diff).sum()
```

```
params = init()
batch = load_batch()
```

```
devices = np.array(jax.devices()).reshape(2,2)
with Mesh(devices, ('batch', 'model')):
    loss = jax.jit(model)(params, batch)
```

Mesh is uniform  
and global for all  
tensors!

Legate programming + execution model  
brings asynchronous MPMD parallelism to Jax

# Legate-Jax can define submeshes for different operations

```
import jax.numpy as jnp
from legate.jax import with_sharding_constraint as shard
from legate.jax import task, parallelize
from jax.sharding import PartitionSpec as P, Mesh

def layer(mlp, x):
    mlp = shard(mlp0, P("model", None))
    x = jnp.einsum('bh,hm->bm', x, mlp)
    return shard(x, P("batch", 'model'))

def model(params, batch):
    mlp0, mlp1 = params
    x = task(layer, mesh=...)(mlp0, x)
    x = task(layer, mesh=...)(mlp1, x)
    diff = x - labels
    return (diff*diff).sum()

params = init()
batch = load_batch()

loss_fxn = parallelize(model)(params, batch)
loss = loss_fxn(params, batch)
```

Two layers can be  
dispatched to  
different  
submeshes

Use MPMD  
parallelization  
instead of jitting  
on global mesh

# Legate/Legion brings flexible mappings to Jax

## Phase 1

Use Jax + MLIR  
+ XLA to create  
Legion tasks

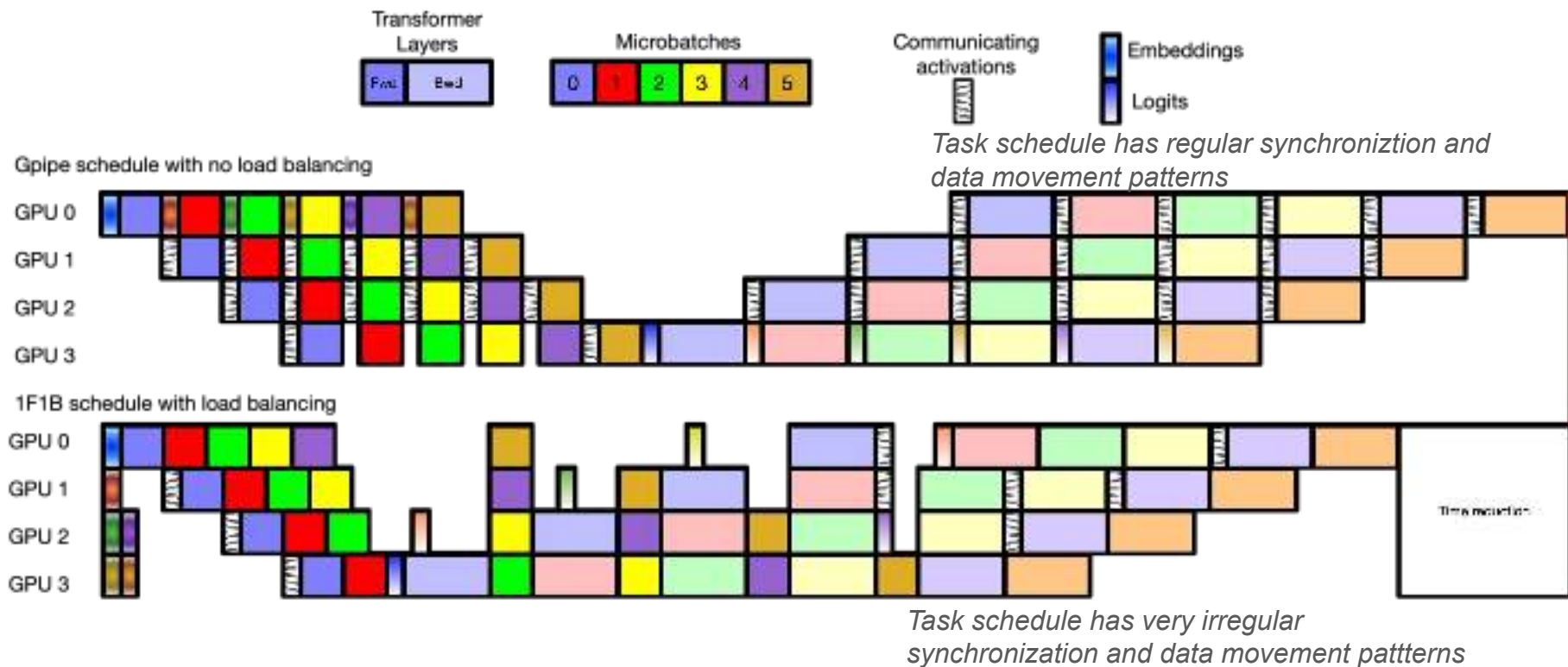
## Phase 2

Legion maps pipelines  
stages to different  
submeshes,  
automagically generating  
pipeline parallelism

## Phase 3

Profit

Programming model makes irregular schedules (1F1B with load balancing) easy to express with sequential semantic and mapper



Programming model expresses logical flow of application agnostic to communication, synchronization, and buffer allocation

```
struct Task {  
    vector<Store> inputs;  
    vector<Store> outputs;  
    int stage;  
    int microbatch;  
    bool forward;  
    DeviceList mesh;  
};
```

```
// Extracted from HLO pipeline passes
```

```
vector<Task> tasks = MpmdPartition(mlir_module);
```

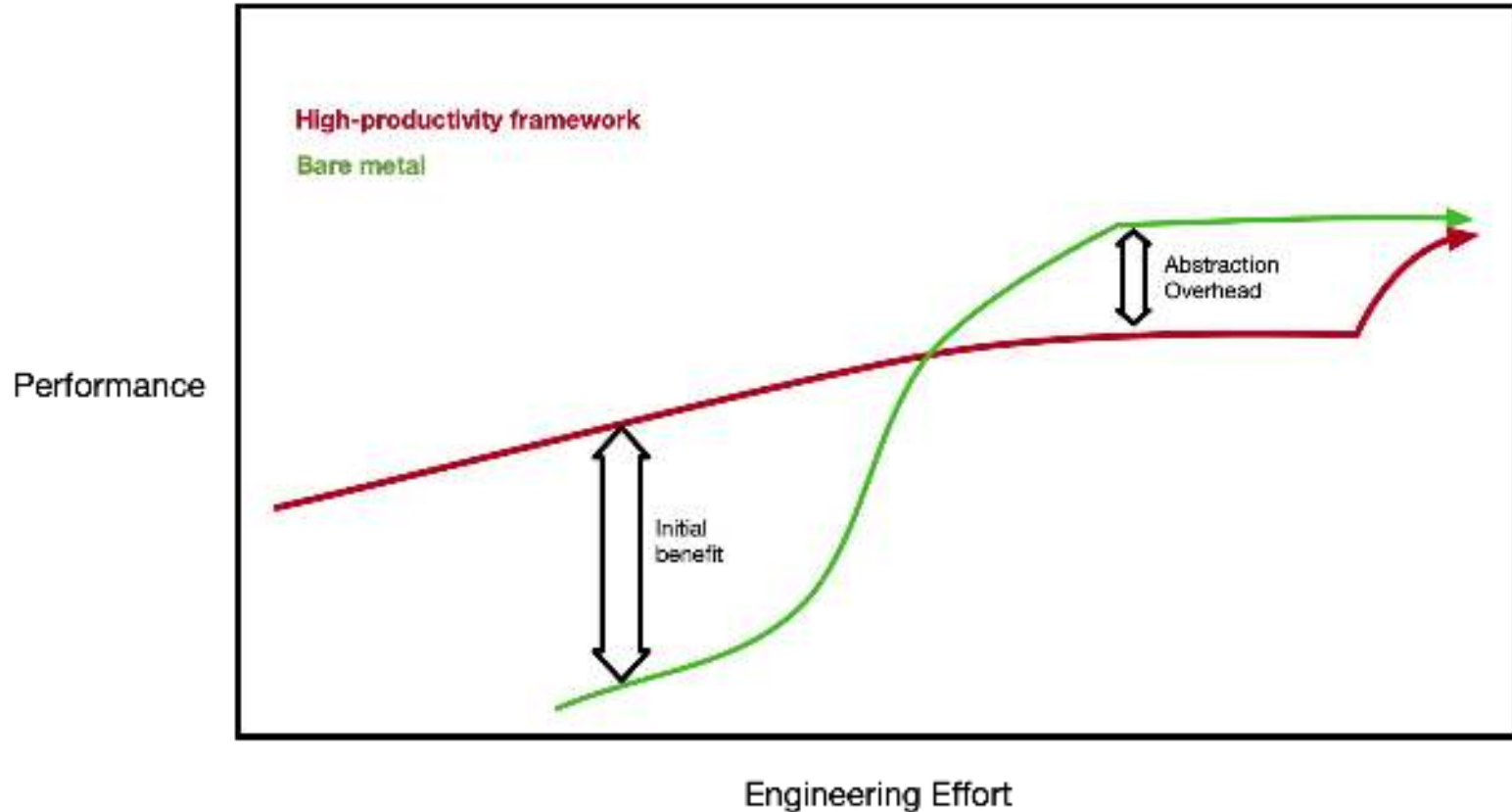
```
// Scheduler chooses ordering and mesh assignment
```

```
vector<Task> ordered = ScheduleTasks(std::move(tasks));
```

Where things break down...



# Productivity and performance are the same thing!



Different customers may have different a different calculus for cost/benefit on engineering effort

Customer	No. accelerators	Abstraction overhead	Accelerator TCO (hypothetical)	Optimization impact	Deadline
A	10K	10%	\$10K	\$ 10M	3 month delivery
B	1M	1%	\$10K	\$ 100M	None, ongoing

\*These numbers are totally made up and do not corresponding to any actual customers or products, real or perceived

# Jax (training LLMs) are not typical Legion workloads and expose missing features and pain points

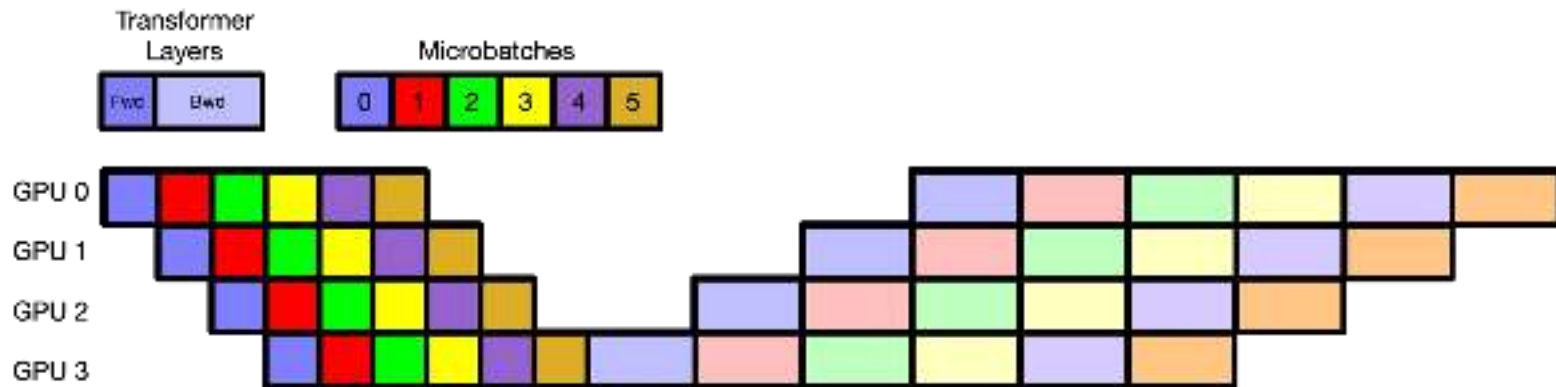
Dependency analysis and control replication overheads are problematic at 1000s of processes with hundreds of inputs/outputs

- Dozens of tensors can be inputs/outputs in LLM
  - Projection matrices, bias terms, MLP layers
  - Parameters, gradients, and optimizer state
- Dozens of scalar metrics (generated from NCCL all-reduce)
  - Future maps? Replicated writes?
- Inscrutable scaling bottlenecks due to thousands of small, control messages
  - Critical path analysis helpful, but still difficult to identify gaps in Legion prof

Jax (training LLMs) are not typical Legion workloads and expose missing features and pain points

## Pipeline parallelism needs a well-defined task ordering

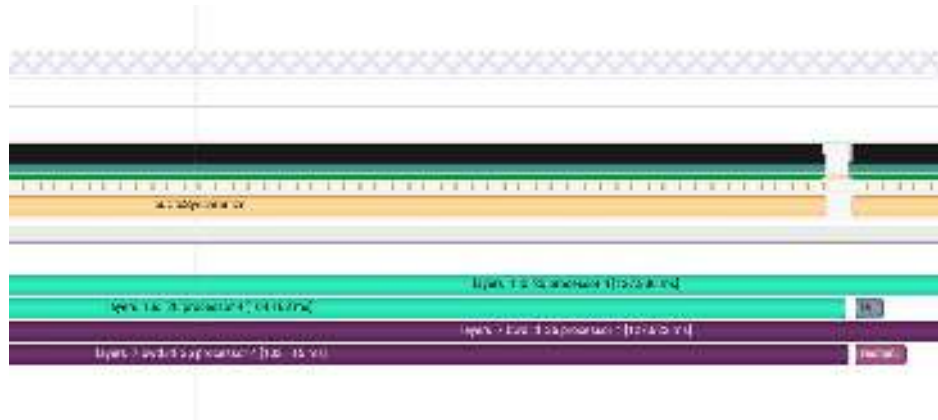
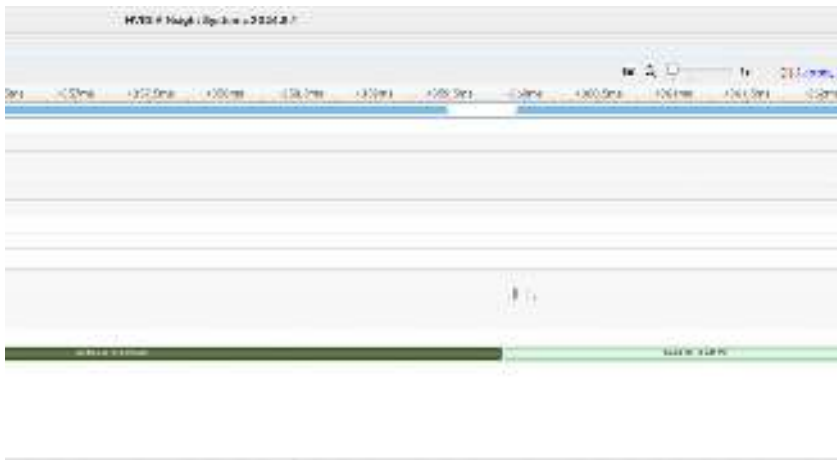
- The compiler has already statically derived an “optimal” schedule
- Dynamic reordering can delay the critical path and increase execution time
- Legate/Legion has no built-in mechanism for lightweight ordering of tasks



# Jax (training LLMs) are not typical Legion workloads and expose missing features and pain points

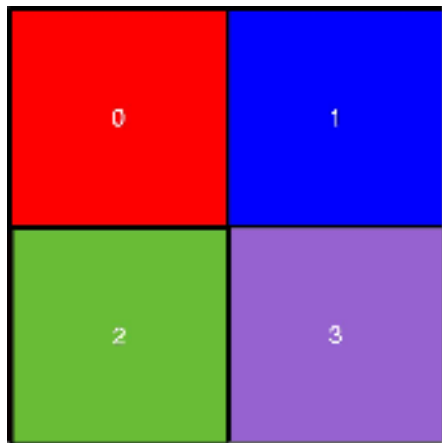
Task completion semantics prevent task lookahead with pipelining

- It is a pipeline! No task parallelism to hide task startup latency.
- Tasks are not marked done until all *data effects on the GPU* are visible
- Bubble in GPU utilization from end of task to start of next GPU kernel

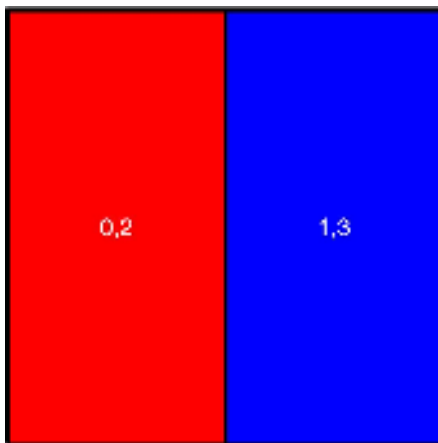


# Jax (training LLMs) are not typical Legion workloads and expose missing features and pain points

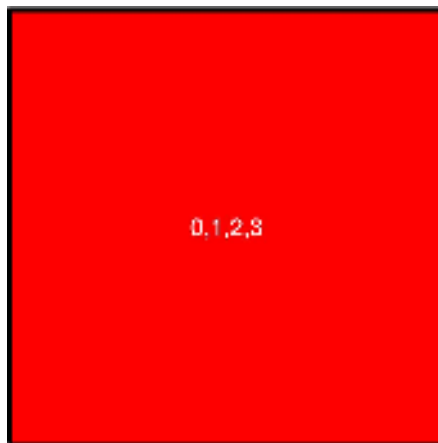
- Communication-avoiding sharding (i.e. data parallelism) writes replicated output
  - Not technically valid in a sequential semantic, but Mike heroically added support
  - Partially replicated/partially sharded data patterns not easily expressed in Legate
  - If done naively, replicated scalar outputs produce *huge* control overhead



Fully-sharded  
(disjoint partition)



Partially replicated  
(non-disjoint partition)

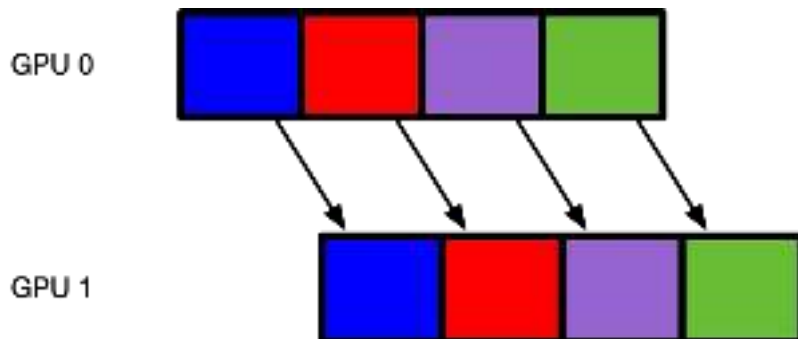


Fully replicated

# Jax (training LLMs) are not typical Legion workloads and expose missing features and pain points

Memory highly constrained, but difficult to optimize instance validation and reuse

- Large models can be 40GB of optimizer state, 30GB of activations
- Pipelined activations computed on node A read on node B are no longer needed on node A, but instance stays alive on node A
  - Can not use a different logical region for each logically distinct activation tensor
- *Abstraction inversion alert*: implemented a Legate store cache with reuse/invalidation that hopefully causes buffers to be allocated in desired way



A new C++ productivity layer  
built directly on Realm

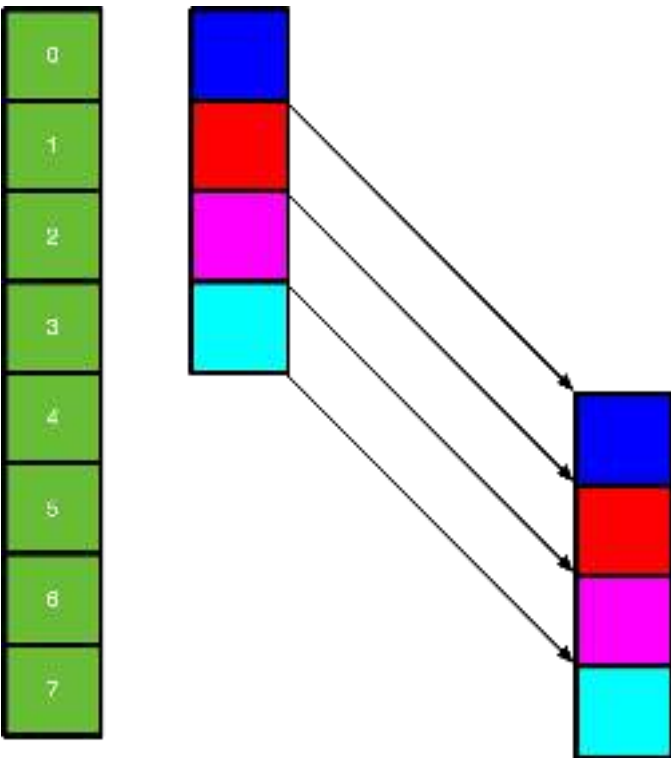


# Deep-learning requires only *some* of what Legate/Legion has to offer

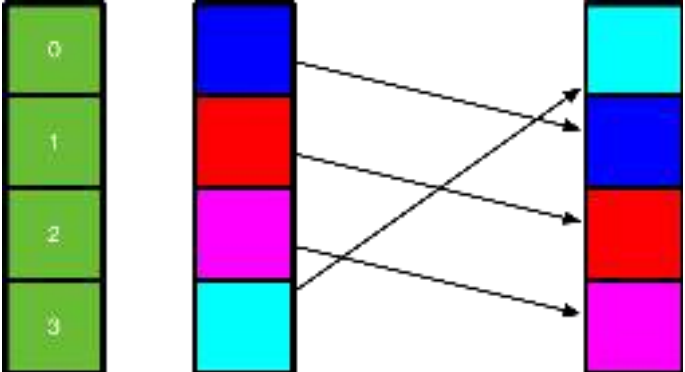
## Requirements

- Flexibly assign tasks and sharded tensors to different submeshes
- Efficient cross-mesh resharding communication primitives
- Execution dependencies for tuning task ordering
- (Simple) fine-grained control over instance reuse and invalidation

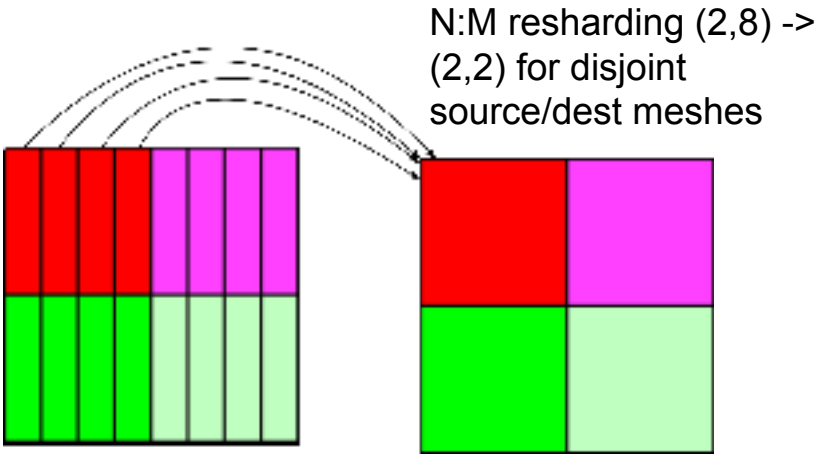
# Asynchronous cross-mesh resharding is most important library primitive



Point-to-point for disjoint source/dest meshes



Permute within a single submesh



N:M resharding (2,8) -> (2,2) for disjoint source/dest meshes

# New lightweight, data-effects C++ layer built directly on top of Realm with simplifications

## Simplifications

- Multi-controller execution with control replication allow dependency analysis to be restricted to *local shards* only
  - Dependency analysis happens in shared memory!
- Control replication forces all processes to agree on inter-process *reshard* operations to move data between submeshes
- No separation of logical and physical arrays
- No fancy slicing/aliasing/hierarchical data tree

# Realm event model made writing asynchronous task-based framework *very easy*

`defer(...)` takes a lambda and arguments and performs dependency analysis on arguments

`Reshard` is a library function that inspects the sharding of input/output and constructs resharding plan using Realm instance copies

```
DeviceList first_mesh = ...;
DeviceList second_mesh = ...;
auto waiter = on(ProcessorGroup::Local()).defer([=](Processor p) {
    Store<ShardedArray> source = ShardedArray::Create(
        GetShape(first_mesh, size), {.processor = p});
    Store<ShardedArray> dest = ShardedArray::Create(
        GetShape(second_mesh, size), {.processor = p});
    across(first_mesh).on(p).defer([](ShardedArray& array){
        int* data = array.tile().ptr<int>();
        // write some values
    }, source);
    Reshard(p, source, dest);
    across(second_mesh).on(p).defer([](const ShardedArray& array){
        const int* data = array.tile().ptr<int>();
        // read some values
    }, dest);
})
```

`const-ref` input automatically tells dependency analysis to use read-only privileges

```
auto event = across(devices)
    .if_on(p)
    .after(precondition)
    .stream_ordered()
    .defer(
```

Lightweight ordering  
mechanism on execution  
preconditions

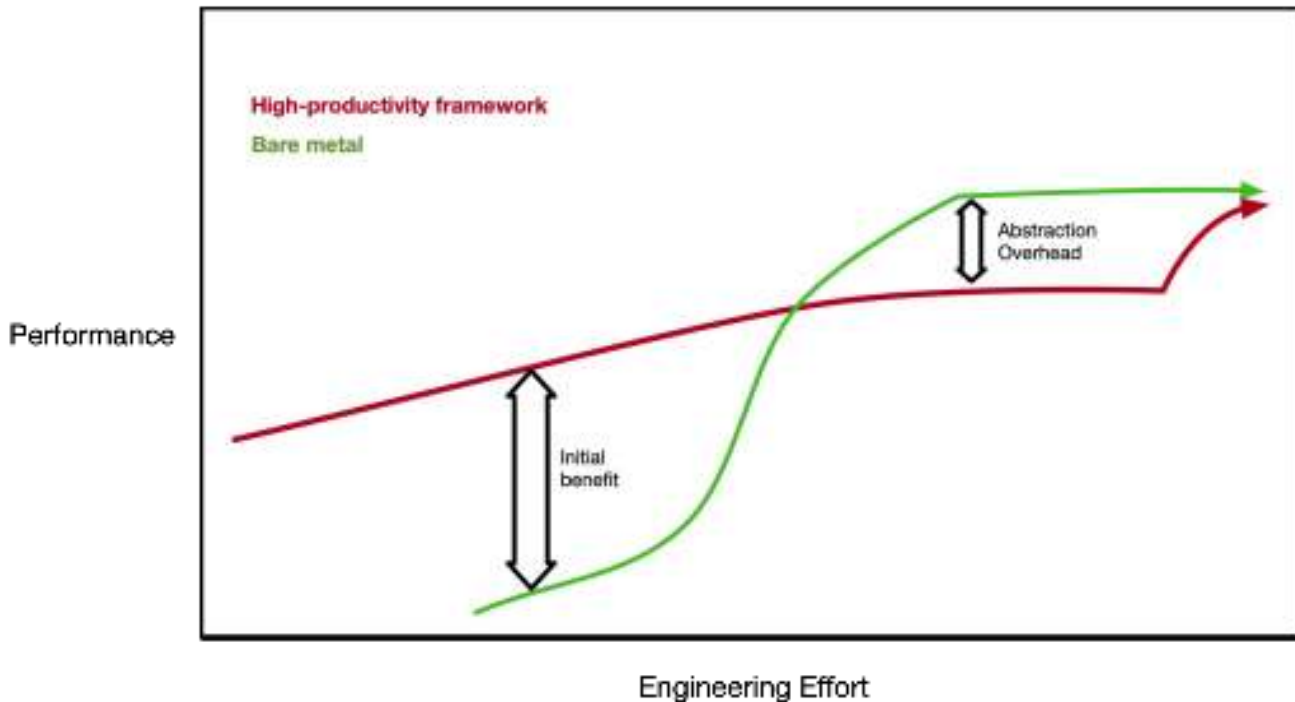
```
    [](Stream* s,
        std::shared_ptr<LegateCompiler> compiler,
        ro_vector<ShardedArray> inputs,
        rw_vector<ShardedArray> outputs,
        const ArrayTile &temp) {
```

Declares data effects on  
the input/output tensors

```
    ...
}, std::move(devices), compiler, std::move(inputs),
    std::move(outputs), std::move(temp));
```

# Conclusions

- Legate/Legion was **great** for 80% solution at modest (128 GPU) scale
- 99% solution with Realm required less engineering effort than Legion
- Good or bad is entirely a matter of customer and use-case requirements

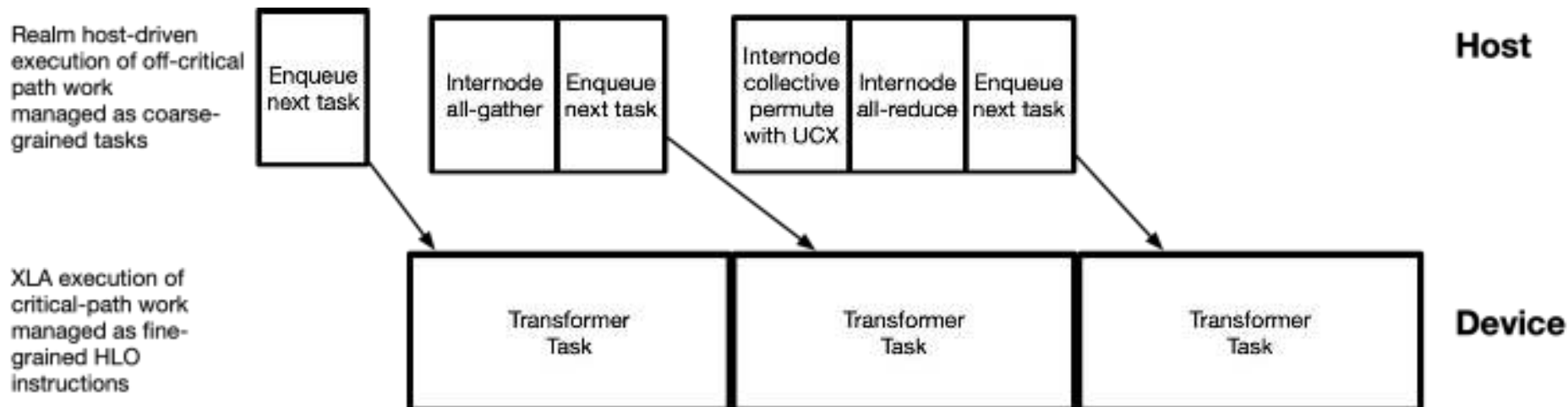




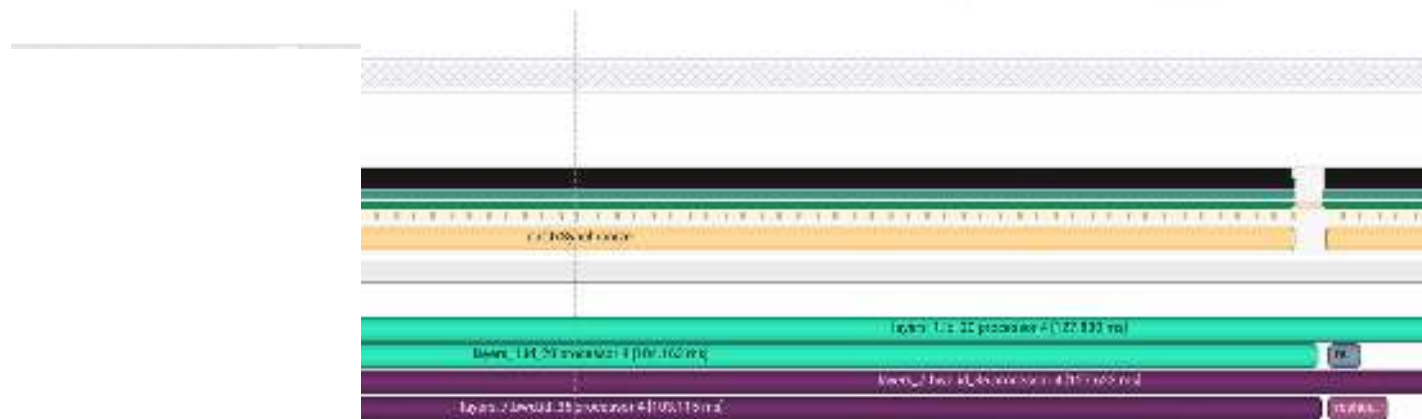
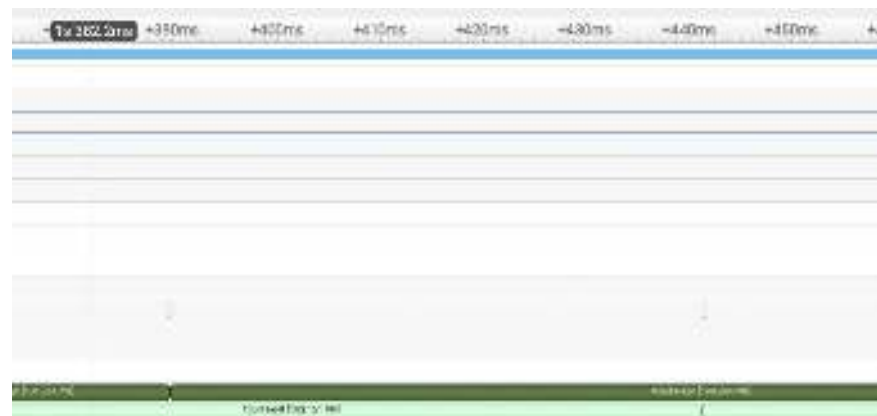
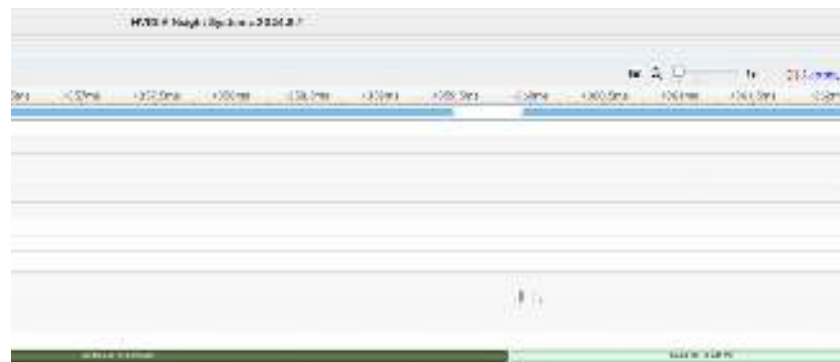
Extras



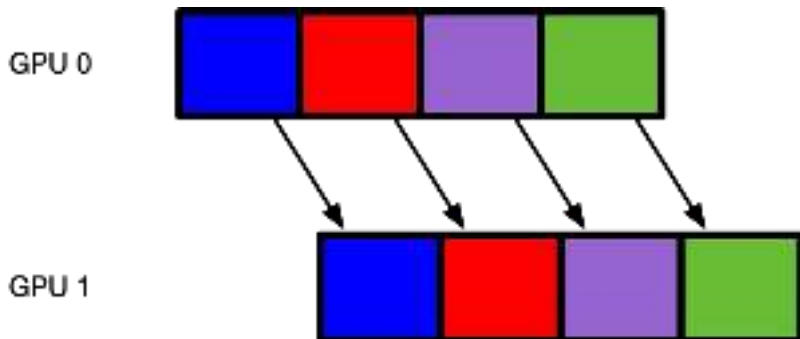
Realm execution model schedules coarse-grained work units, overlaps data movement/scheduling with critical path



# Delays at end of task



# Problem with instances



## Requirements

- Flexibly assign tasks and sharded tensors to different submeshes
- Efficient cross-mesh resharding communication primitives
  - Legion doesn't always see higher-level structure of required data movement
- Execution dependencies for tuning task ordering
  - Not exposed
- Fine-grained control over instance reuse and invalidation
  - Not exposed

