# Enabling State-of-the-art Asynchronous Execution in Torch.compile With CUDA Streams

Michael Lazos, Meta

PyTorch CONFERENCE EUROPE 2026

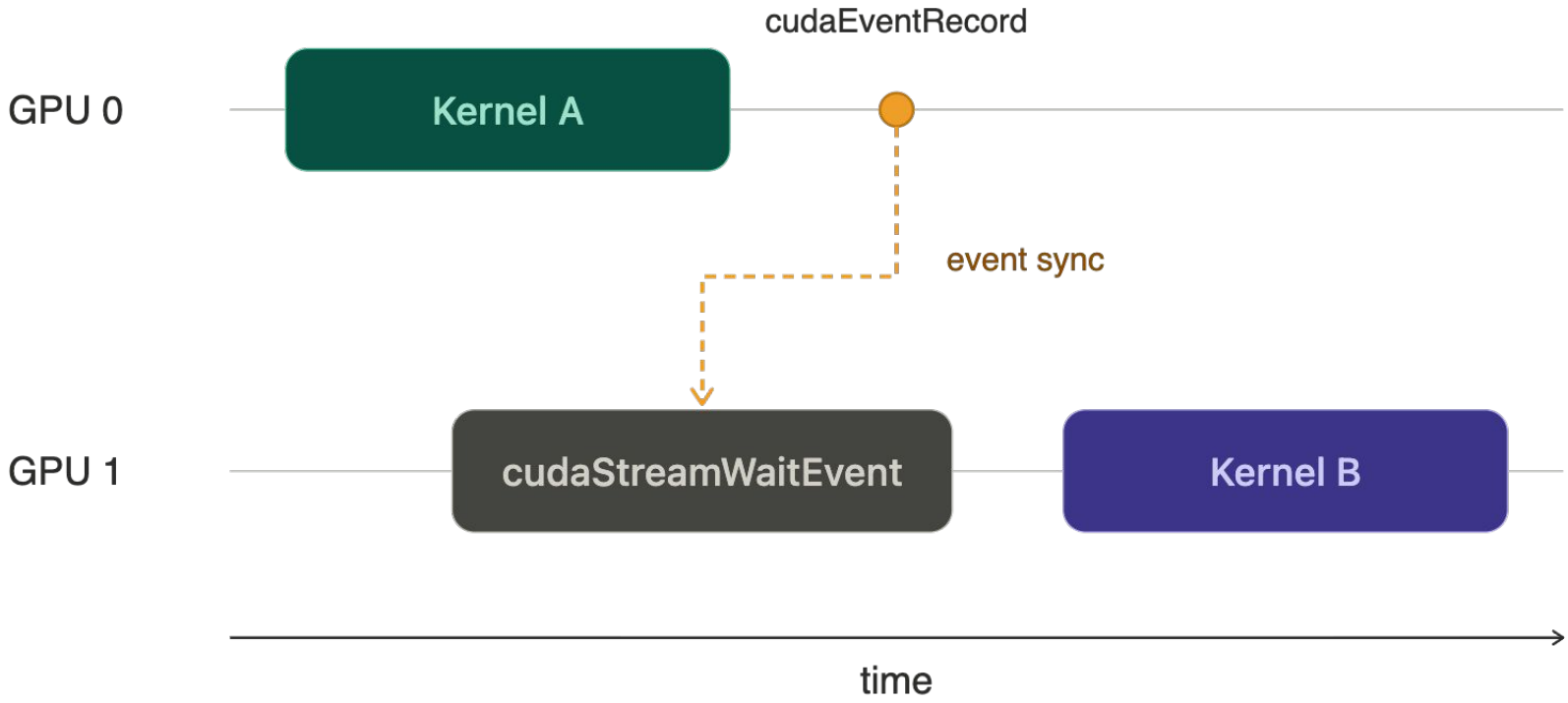
# Agenda

- What Are Streams?
- Overall Architecture
- Implementation Challenges
- Use Case: Microbatch Overlapping
- Use Case: Activation Offloading
- Results
- Q&A

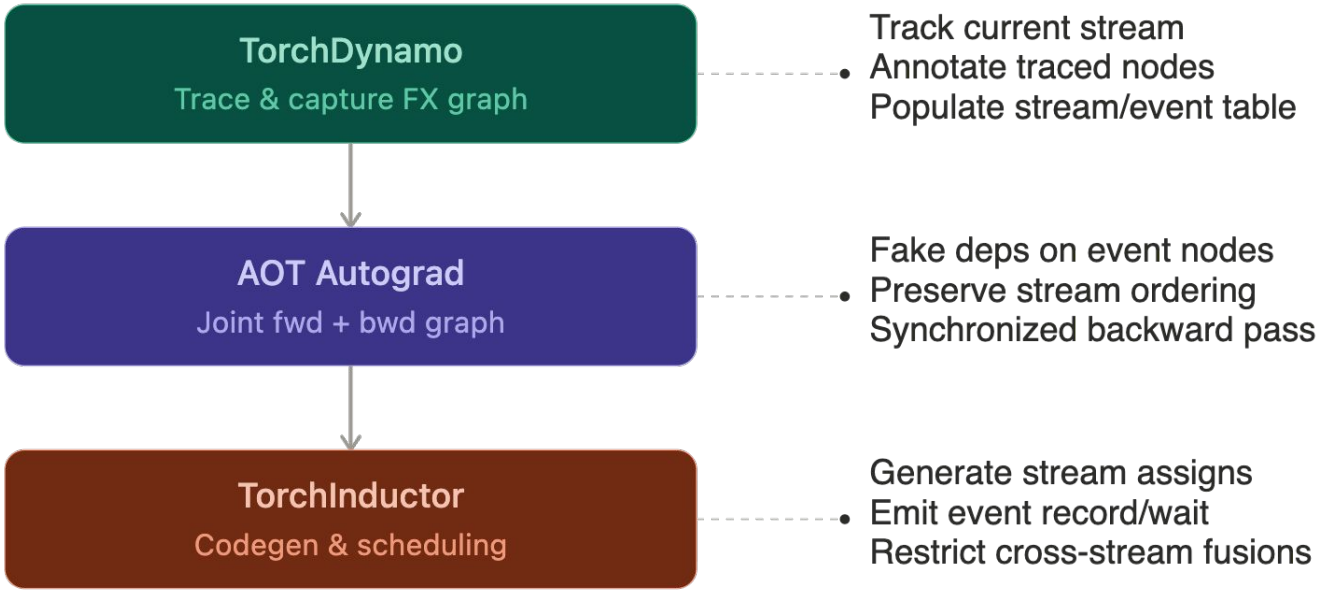
# Background: What are streams?

- Enable concurrent execution of independent kernels - e.g. comm-compute overlap, memory data transfer hiding.
- Events are used to synchronize between different streams
- Streams can also synchronize across devices

# Background: Streams Example Usage



# Stream Handling Architecture in torch.compile

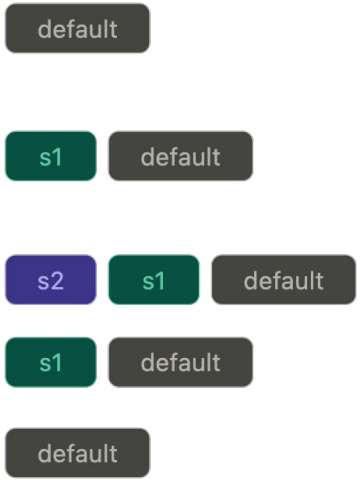


# Tracking Streams In TorchDynamo

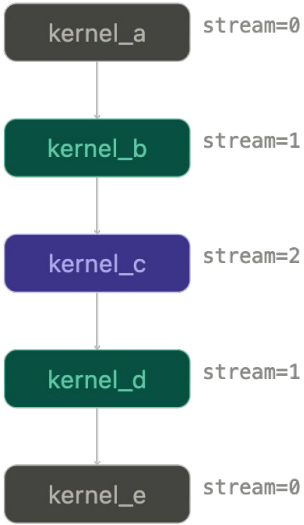
User code

```
s1 = cuda.Stream()
s2 = cuda.Stream()

# default stream
kernel_a(x)

with cuda.stream(s1):
    kernel_b(x)

    with cuda.stream(s2):
        kernel_c(x)

    kernel_d(x)

kernel_e(x)
```

Stream stack




FX graph



# Supporting Streams and Events as Graph Inputs

## User code

```
s1 = cuda.Stream()
s2 = cuda.Stream()

with cuda.stream(s1):
    kernel_a(x)

with cuda.stream(s2):
    kernel_b(x)
```

Streams are opaque user objects — can't be graph constants.

## Prefix bytecode

Runs before compiled graph:

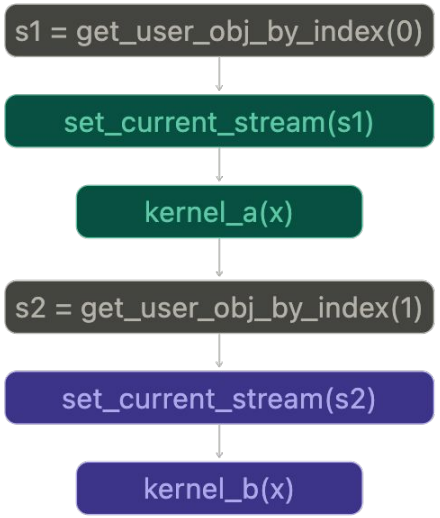
```
user_obj_table[0] = s1
user_obj_table[1] = s2
```

Index → Stream table

`0 → Stream(id=1) 1 → Stream(id=2)`

Prefix is regular Python bytecode. Captures live stream objects into an indexed table the graph reads.

## FX graph




get_user_obj_by_index reads from the table at runtime — no stream objects baked as constants

# Graph IR Representation

## Stream/event ops in FX graph

torch.ops.streams.*

```
torch.ops.streams.record_event(event_idx, stream_idx)
torch.ops.streams.wait_event(event_idx, stream_idx)
torch.ops.streams.wait_stream(wait_idx, stream_idx)
torch.ops.streams.get_user_obj_by_index(idx)
```

## Node metadata

Single custom field on every node:

```
node.meta["stream_idx"]
= 0, 1, 2, ...
```

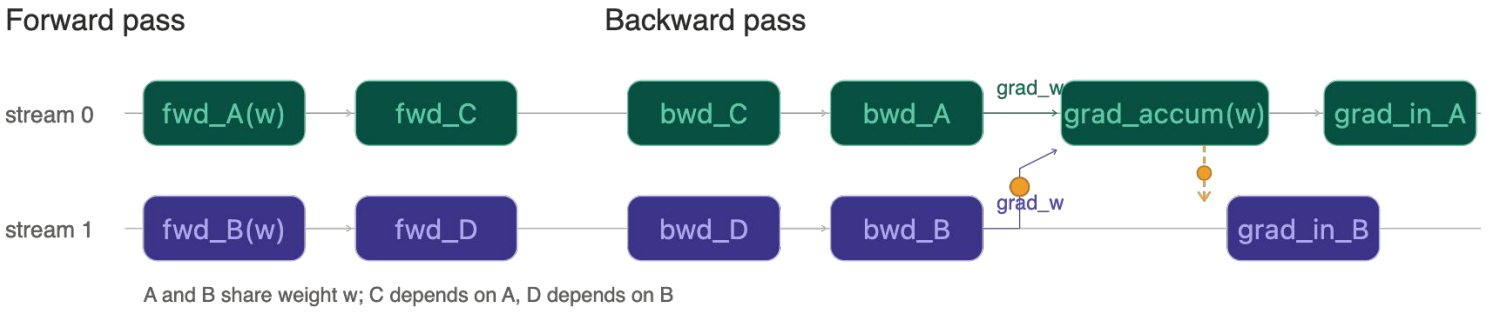
Tracks which stream each op executes on. Used by AOT Autograd and Inductor for scheduling, fusion decisions, and codegen.

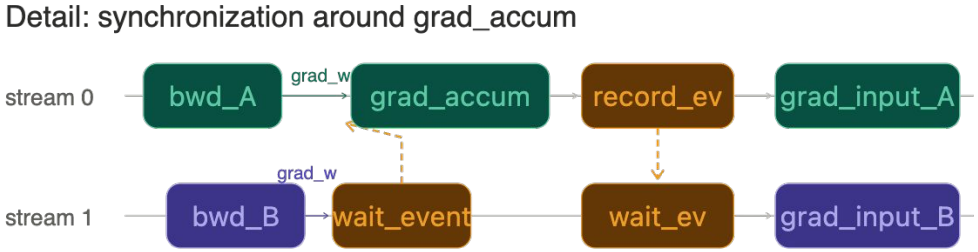
## Example FX graph


record_event(0, 1): record event 0 on stream 1 after kernel_b
wait_event(0, 0): wait for event 0 on stream 0 before kernel_c

stream_idx on each node is set by Dynamo during tracing — no set/get_current_stream ops in the graph.

# Generating the Synchronized Backward Pass




Rules

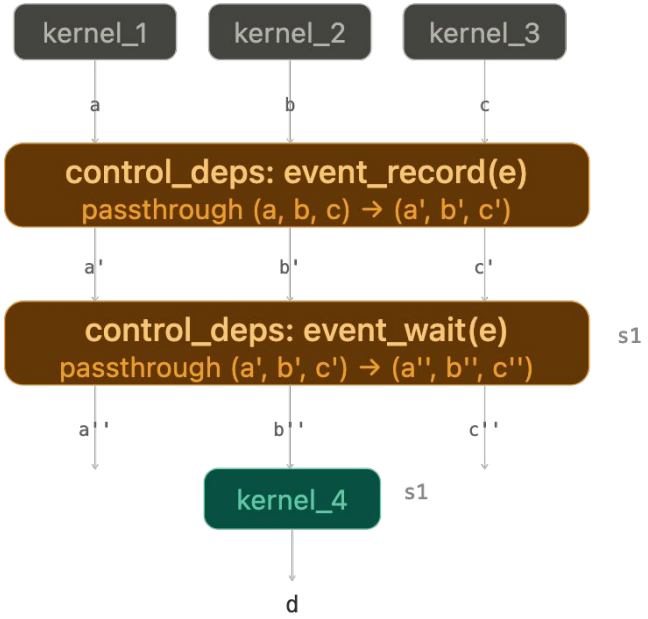
1. bwd node → same stream as its forward node
2. grad_accum → stream of its first user
3. Cross-stream grad inputs → wait_event before accum
4. After accum → record + wait to broadcast result

# Composition with Compiler Pass Reordering

User code

```
a = kernel_1(x)
b = kernel_2(x)
c = kernel_3(x)

e = torch.cuda.Event()
e.record()

with cuda.stream(s1):
    e.wait()
    d = kernel_4(a, b, c)
```

FX graph




Each HOP passes deps through as identity, preventing reordering across events

# Background: Functionalization with Input Mutation

## User code (in-place)

```
# x is a graph input
x.mul_(2)
y = x.add(1)
return y
```

x.mul_(2) mutates x in-place.
The caller sees the mutation
after the compiled region returns.

## After functionalization

Graph body (functional):

```
x_updated = x * 2
y = x_updated + 1
```

Graph epilogue:

```
x.copy_(x_updated)
```

Write-back deferred to end!

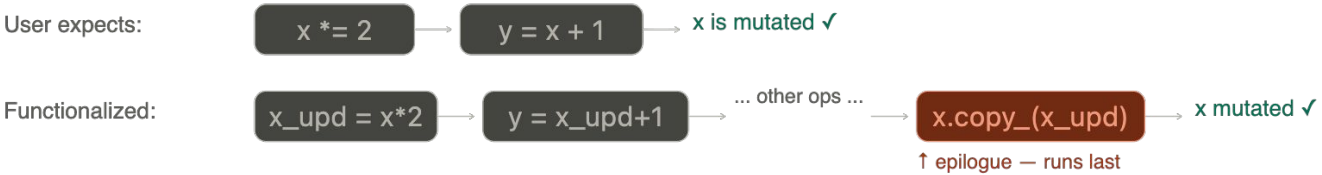
## Why?

Functional graphs are
easier to optimize:

- No aliasing concerns
- Safe to reorder ops
- Enables fusion
- Clean backward pass

But the caller still needs
to see the mutation, so
copy_ writes it back at the end.

## Execution order

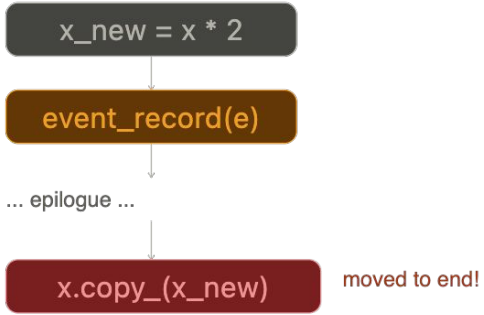


# Incorrectly Recording a Functionalized Input Mutation

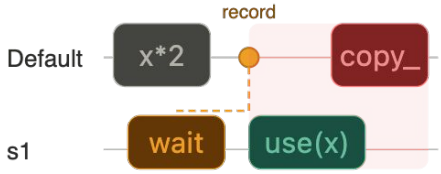
### User code

```
# x is a graph input
x.mul_(2)

e = cuda.Event()
e.record()

# outside compiled region
with cuda.stream(s1):
    e.wait()
    use(x)
```

### After functionalization




### Problem




s1 sees stale x — copy_ hasn't written back yet.

### Detection

Dynamo detects when event_record depends on a mutated graph input whose copy_ would be deferred past the record by functionalization. This raises a compile-time error.

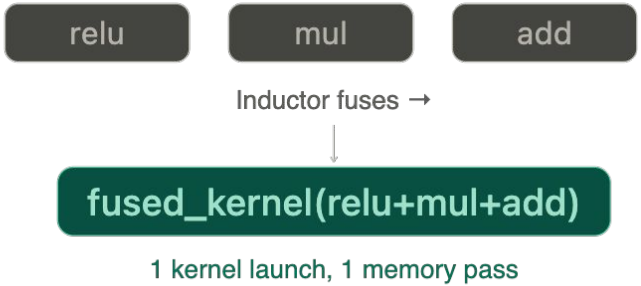
### Allowed

Input mutation without synchronization → OK

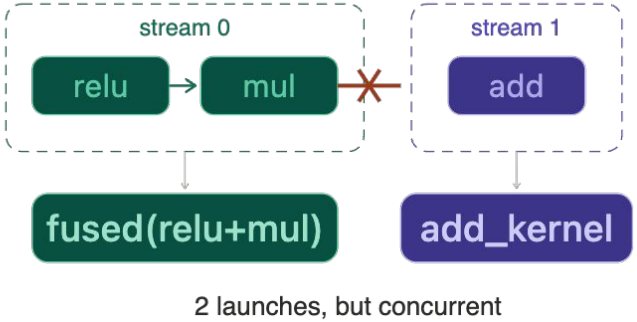
Epilogue copy_ is safe when no event captures the order.

# TorchInductor Streams Codegen

### Without streams (fuses freely)




### With streams (fusion blocked)




### Inductor fusion rules with streams

- Same stream → fuse normally (pointwise, reduction, etc.)
- Different streams → never fuse (need separate launches for concurrency)
- Event nodes → act as fusion barriers (cannot fuse across record/wait)

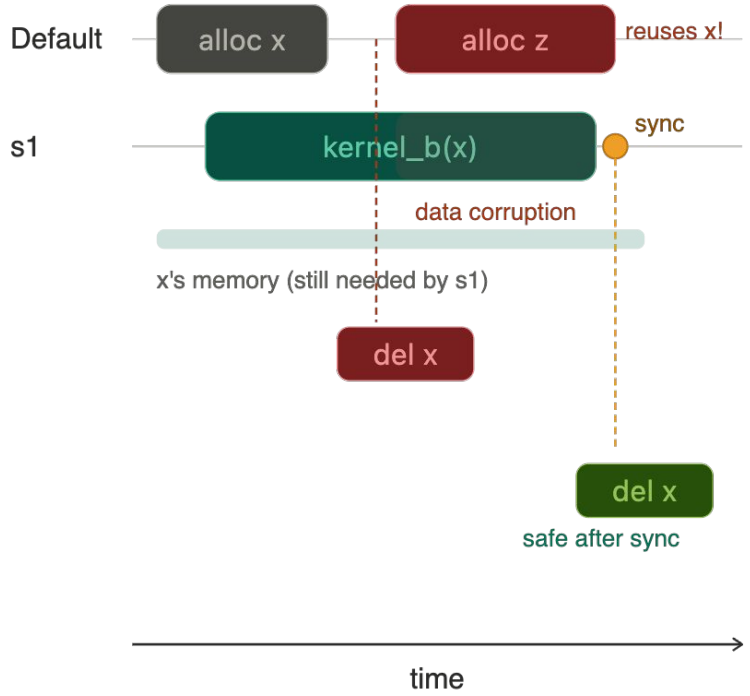
Trade-off: fewer fusions = more kernel launches, but concurrency from multiple streams can outweigh launch overhead when kernels are small or memory-bound.

# Cross-stream Memory Safety

User code

```
x = torch.randn(1024, device="cuda")

with cuda.stream(s1):
    y = kernel_b(x)

# x ref dropped on default stream
# allocator may reuse x's memory
# while s1 still reads from it

del x
z = torch.randn(1024) # reuses x

# correct: sync before freeing
s1.synchronize()
del x
```

Timeline



# Handling User Deallocations

## User code

```
# x allocated on default
x = torch.randn(1024)

with cuda.stream(s1):
    y = kernel_b(x)

# back on default
s1.synchronize()
del x
z = torch.randn(1024)
```

Dynamo respects the user's synchronize and del.

## Dynamo inserts

Dynamo tracks:

x.origin_stream = default

x crosses to s1:

e = record_event(s1)

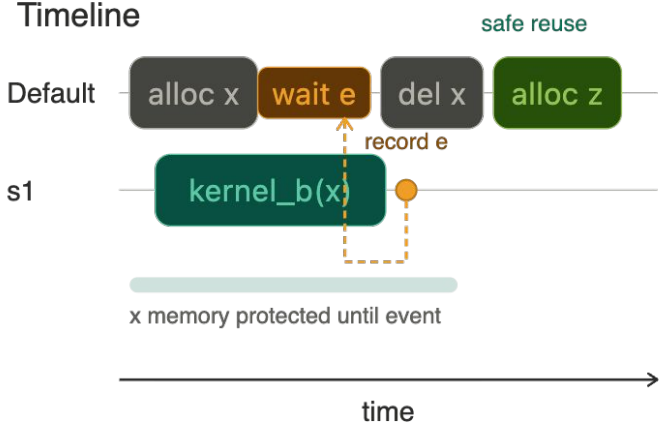
Before alloc z on default:

wait_event(default, e)

Delays the alloc, not the del.

Non-blocking on the host.

## Timeline

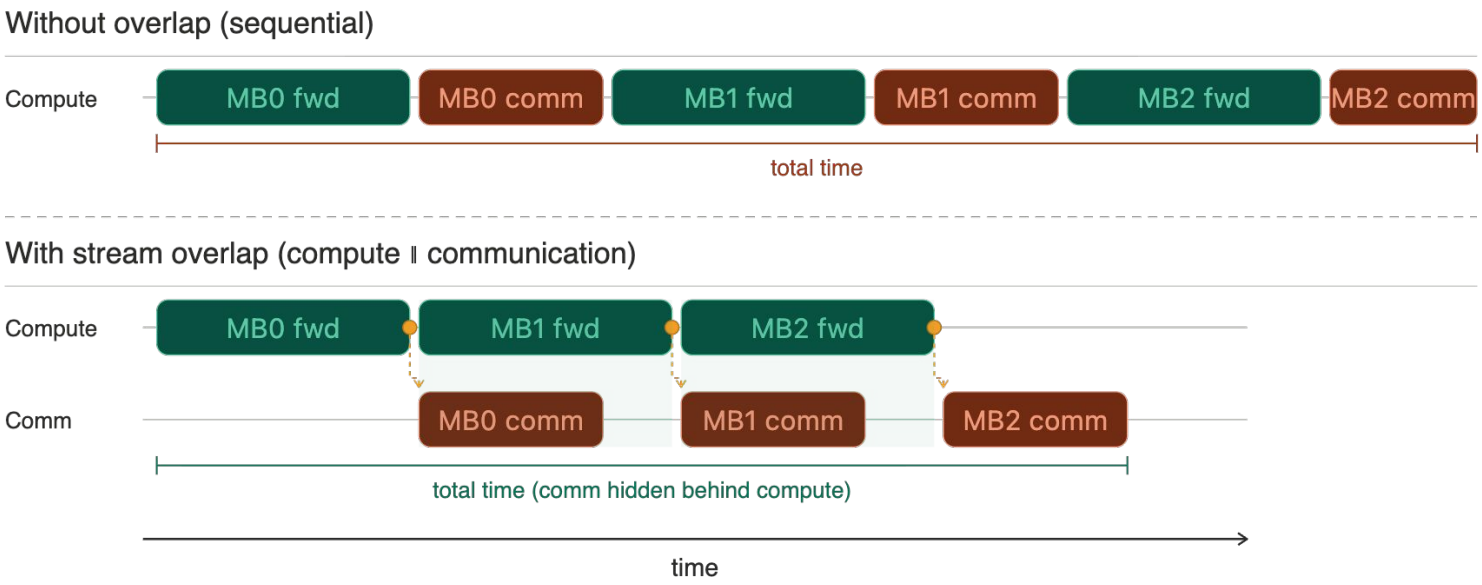



wait e → del x → alloc z

Default stream waits for s1,
then frees x, then alloc z
can safely reuse the memory.

Allocator respects the event —
correct without host-side blocking.

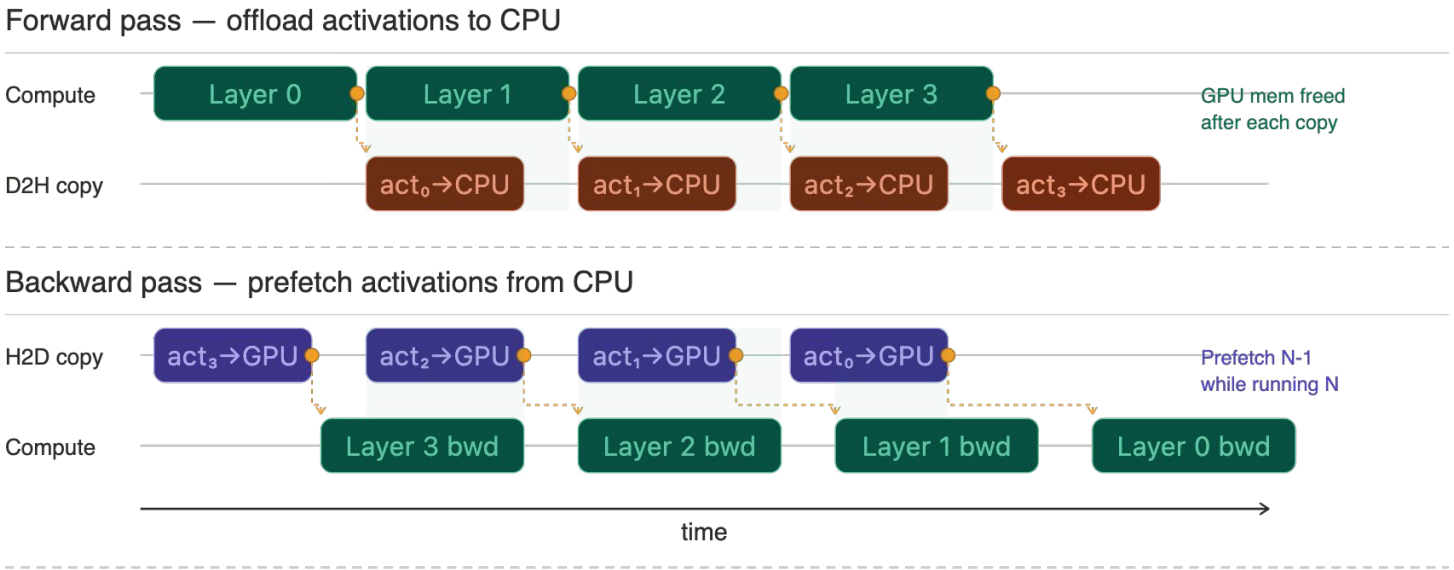
# Applications: Microbatch Overlap




Key

Each microbatch's communication (allreduce, allgather) runs on a separate stream, overlapping with the next microbatch's forward pass. Events ensure comm only starts after compute finishes.

# Applications: Activation Offloading




## How it works

Forward: after each layer, activations are copied to CPU on a separate stream, freeing GPU memory.
Backward: activations are prefetched one layer ahead on the H2D stream, overlapping with compute.

Trades PCIe bandwidth for GPU memory — enables training larger models that wouldn't fit otherwise.

# Results: Activation Offloading

## Memory savings benchmark

Step time · peak allocation · runtime overhead

| Config | Step time | Peak alloc | Memory savings | Runtime overhead |
|---|---|---|---|---|
| Small (1024, 8L, b64, s512) | 470 ms | 9,664 MB | 1,024 MB (9.6%) | 9% |
| Med (2048, 16L, b64, s1024, fsdp=2) | 1,250 ms | 36,906 MB | 7,424 MB (16.7%) | 4.2% |
| XL (2048, 16L, b64, s2048, fsdp=2) | 1,525 ms | 64,611 MB | 15,232 MB (19.1%) | 0.1% |

# How to try out streams today

- Torch.compile handles your stream code as it exists today, it is available in the nightly
- Beta release in PyTorch 2.12
- See the eager streams documentation at https://docs.pytorch.org/docs/stable/generated/torch.Stream.html for writing stream-oriented code

# Q & A