

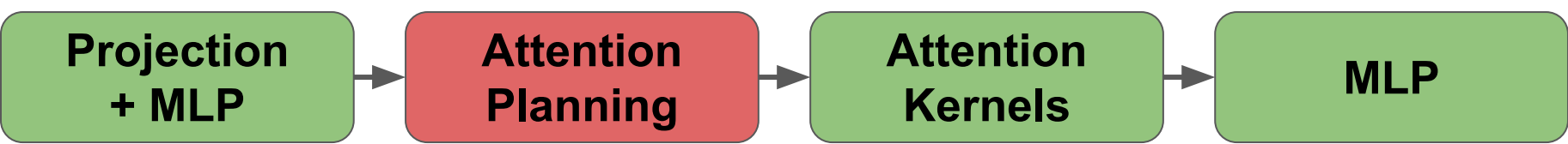
CUDA Graphs • Importance and Limitations

CUDA Graphs remove launch overhead by recording launches upfront for replay **BUT they come with constraints**

- All addresses must be static
- The same kernel/memory sequence must remain valid across executions
- CPU syncs, dynamic planning, and Python must not run inside of a graph replay

One incompatible operation makes full CUDA graph capture impossible. This makes some techniques hard to capture

- Attention planning based on live sequence lengths
- Host-side attention metadata preparation
- Runtime-dependent state updates



Four ways to execute the same mixed workload

- **Eager.** No CUDA graphs, full host-side overhead
- **Full CUDA graph capture.** Requires the entire workload to be capturable OR a lot of manual orchestration
- **torch.compile piecewise CUDA graphs.** Requires buying into the complexity of torch.compile
- **Breakable CUDA Graphs.** User-friendly, eager way to interleave CUDA graphs and plain eager executions

Breakable CUDA Graphs in PyTorch

The new API is simple & analogous to the existing `cuda.CUDAGraph`

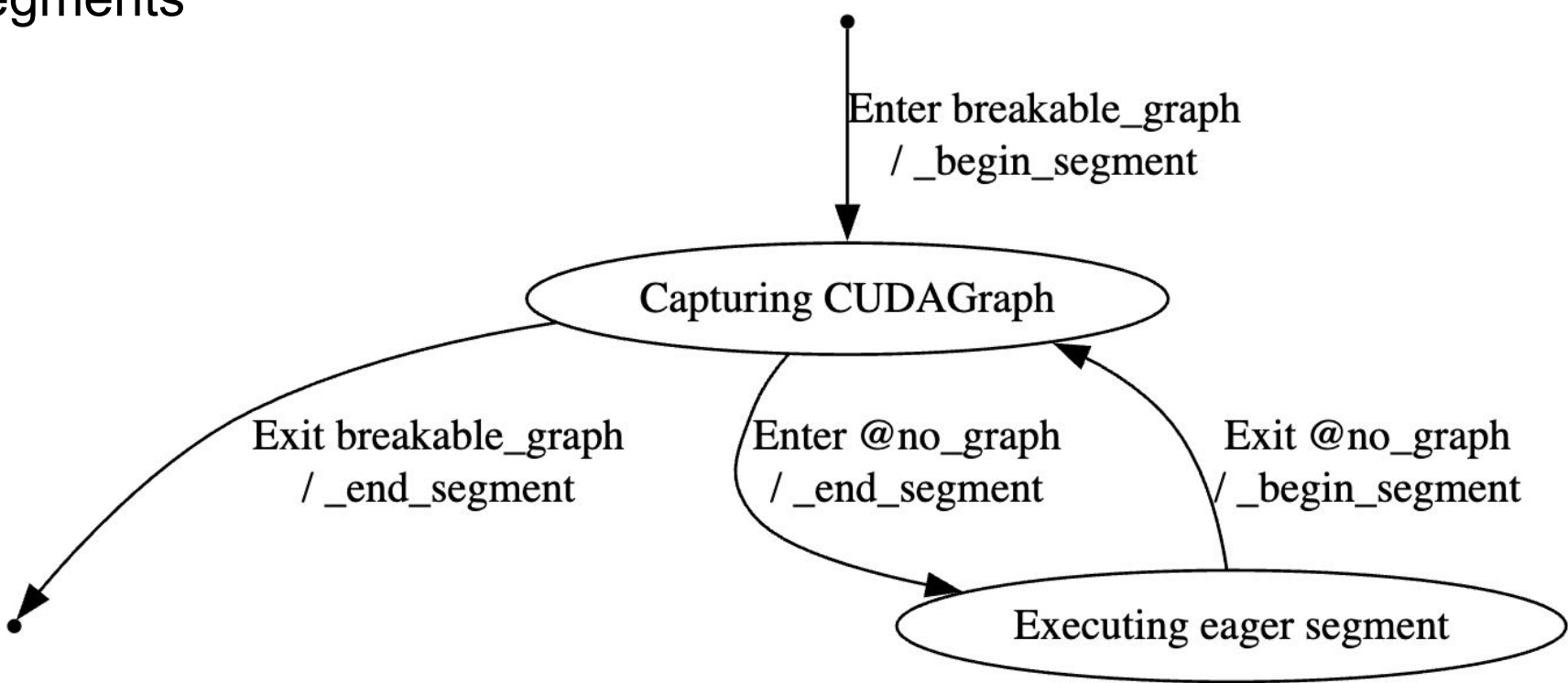
- Decorate CUDA graph-incompatible functions with `@no_graph`
- Use **BreakableCUDAGraph** instead of `CUDAGraph`

```
@no_graph
def dynamic_scale(x: torch.Tensor) -> None:
    """Reads a value to CPU - cannot be captured in a CUDA graph"""
    if x.sum().item() > 0:
        x.clamp_(min=0)

"""Capture: BCG automatically splits around @no_graph calls"""
seq = BreakableCUDAGraph()
with breakable_graph(seq):
    result.copy_(input * 2)
    dynamic_scale(result)
    result.add_(1.0)

"""Replay: same as regular CUDA graphs"""
seq.replay()
```

Breakable CUDA graphs enable a special CUDA graph context, which switches between two modes of recording for eager and CUDA graph segments



Details and OSS Lineage

CUDA graph segments share one memory pool

- (Breakable) CUDA graphs use their own memory pool to avoid conflicts with eager
- The memory pool is reused across graph segments, which allows for memory reuse across segments
- To reference these, eager replay uses non-owning pointers

Benefits that SGLang finds with breakable CUDA graphs¹

- Faster startup, no compile time overhead
- Broader compatibility w/o wrapping operators through `torch.library`
- Debuggable by construction - it's simple Python (compared to `torch.compile` piecewise CUDA graphs)

Open-Source Lineage

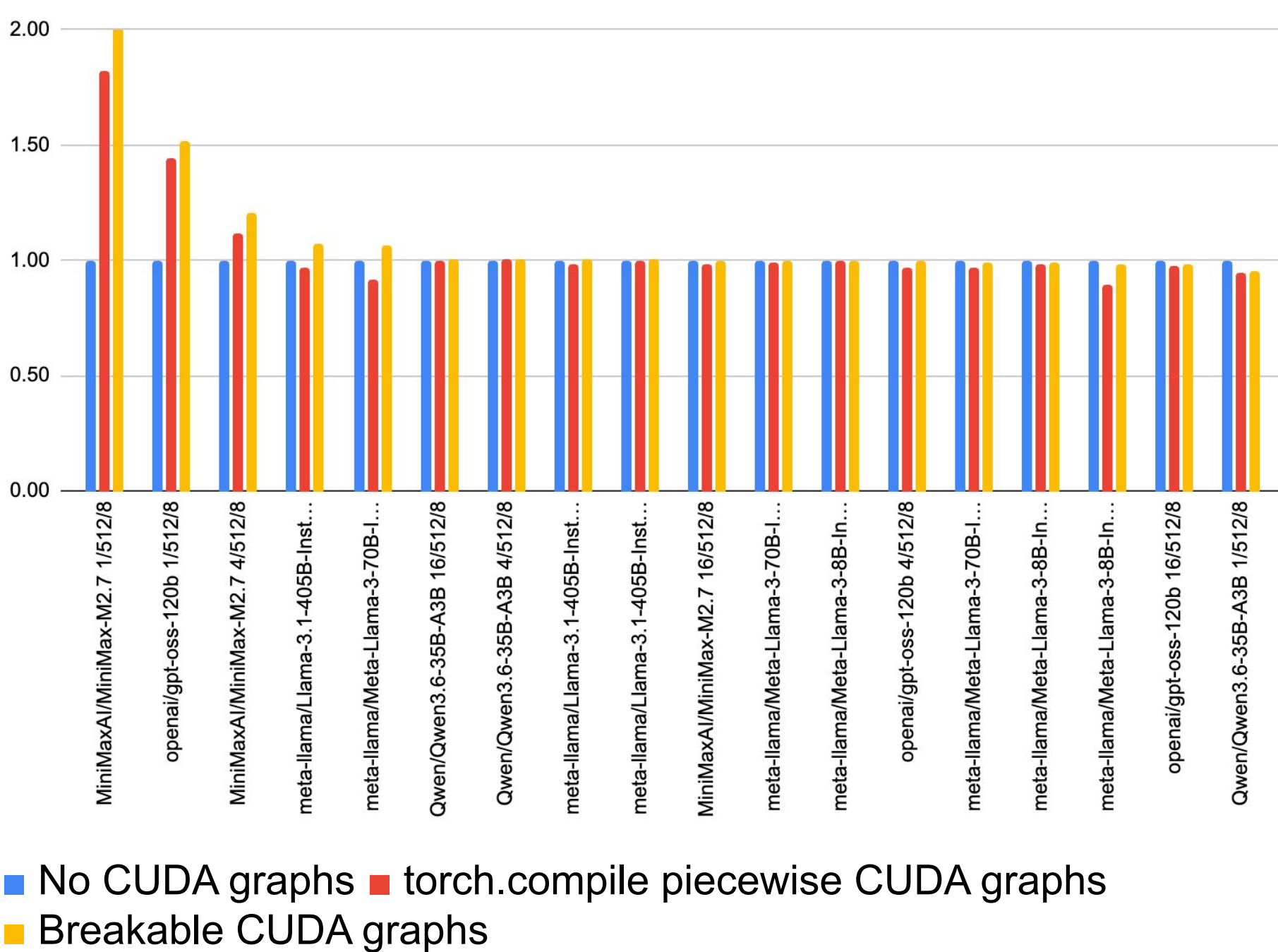
- SGLang introduced Breakable CUDA graphs in tree
- Standalone Python package makes them more robust & reusable outside of SGLang ([meta-pytorch/breakable-cuda-graphs](https://github.com/meta-pytorch/breakable-cuda-graphs))
 - No implicit copies
 - No implicit stream management
 - No monkey-patching
- This implementation moves to PyTorch providing a stable, backwards-compatible API as `torch.cuda.BreakableCUDAGraph` ([pytorch/pytorch](https://pytorch.org/docs/stable/torch_cuda.html))

Prefill Performance in SGLang

The test integration into SGLang achieved

- Performance parity with the in-tree implementation
- Surpassed `torch.compile` piecewise CUDA graphs in some cases, thanks to reduced host-side overhead

Prefill speedups over no CUDA graphs



■ No CUDA graphs ■ torch.compile piecewise CUDA graphs ■ Breakable CUDA graphs

Thanks 🙏

To Richard Zou, Edward Yang, Elias Ellison, and Yanan Cao (Meta) and Yuwei An (SGLang) for design feedback, code reviews.

¹ <https://www.lmsys.org/blog/2026-08-17-advanced-cuda-graph/>