



CONFERENCE

—NORTH AMERICA 2026—

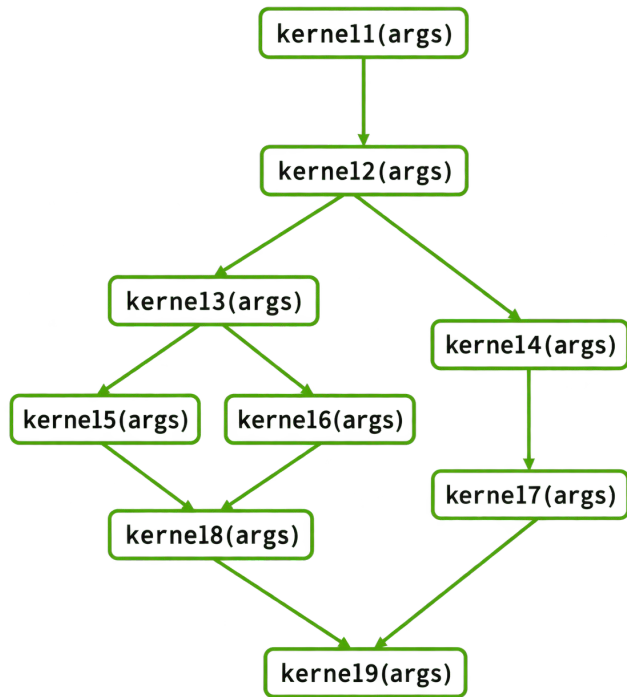
Dynamic Shape CUDA Graphs

One CUDA graph, many shapes

Elias Ellison, Daniel Galvez

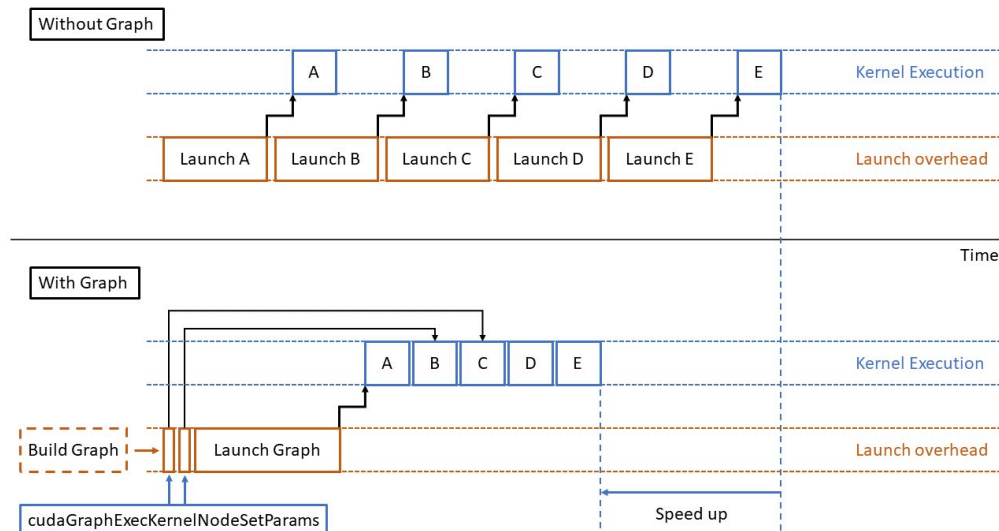
CUDA Graphs Background

- A CUDA Graph is a static record of GPU work
- Created Once
- Replayable with a single CUDA API call, as many times as desired
- Each replay runs the same kernels, (normally) with the same arguments.



CUDA Graphs Always Improve Performance for Static Shape Workloads

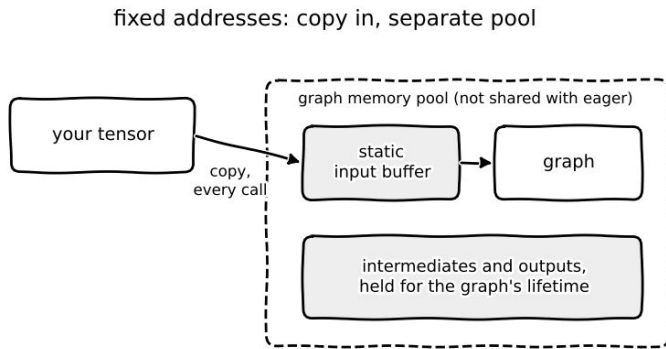
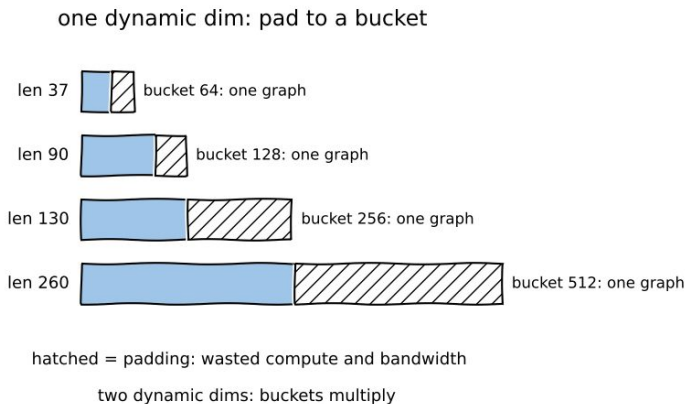
- Faster host: one launch per graph, not one per op
- 1-2 microseconds of GPU-side overhead saved per kernel



CUDA Graph Usability Limitations

Dynamic shapes: wasted padding, increased capture time

Memory: cost of copying in tensors, can't reuse memory allocated to private pool



Of the CUDA Graphs captured for different shapes in vLLM, fewer than 5% of them vary by anything other than parameters.

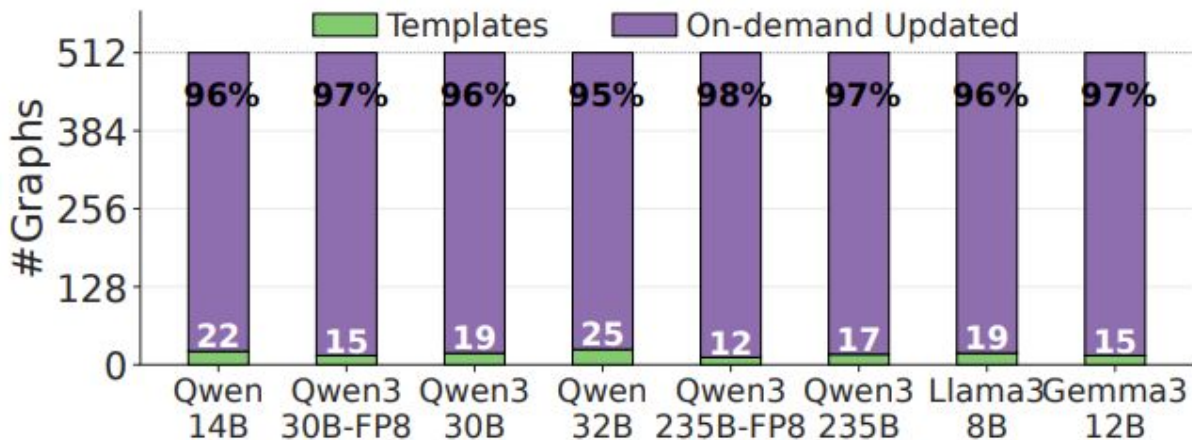


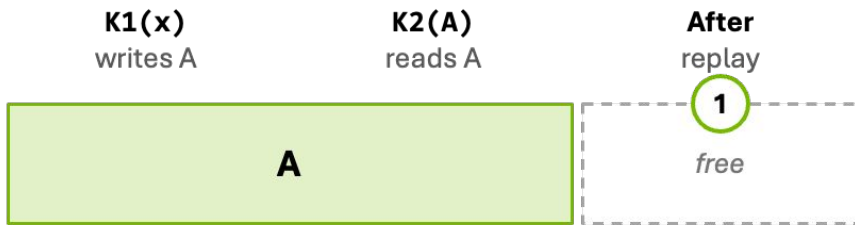
Figure from "Foundry: Template-Based CUDA Graph Context Materialization for Fast LLM Serving Cold Start"

Why CUDA Graph Private Pool Dead Memory Cannot be Reused by non-CUDA Graph code

```
g = torch.cuda.CUDAGraph()
with torch.cuda.graph(g):
    A = K1(x)  # A allocated in Block 1
    y = K2(A)  # last use of A
    del A      # Block 1 free again

g.replay()    # writes A to Block 1
```

Block 1 (0x1000) during one g.replay() →



① **Free is not unused:** every replay writes A to Block 1 again.

② Suppose non-cudagraph code allocated memory in Block 1:

E = torch.ones(x)
lands in Block 1



g.replay() runs K1(x)
and writes A into Block 1



E is silently overwritten

Symbolic Tracing of CPU Code as the Solution

Problems:

1. Wasted Compute due to Padding
2. Excess Number of Graphs Captured
3. Excess Memory due to Private Pool

Solution:

- Guards: Know whether a given cuda graph applies to a set of user provided inputs.
- Update rules: Given user provided inputs, know which kernel parameters to update to work on these new inputs. Given fresh allocations, know which kernel parameters to update to work on these outputs and temporaries.

Dynamic CUDA Graph Interface: HostTraceReplay

```
class DynamicGraphTraceReplay:
    def __init__(self, fn):
        self.fn = fn
        self.cache = {} # maps guards -> captured CUDA graph

    def __call__(self, *args):
        # 1. Check if inputs match guards of any existing CUDA graph
        for graph, guards in self.cache.items():
            if guards.check(*args):
                outputs, temporaries = allocate_outputs_and_temporaries(graph)
                update_cuda_graph_params(graph, *args, *outputs, *temporaries) # Hit: Update parameters
                graph.replay() # Hit: replay cached graph
                return outputs

        # 2. Guard mismatch / cache miss: capture a new CUDA graph
        new_graph = capture_cuda_graph_with_host_trace(self.fn, *args)
        self.cache[new_graph] = new_graph.guards
        new_graph.replay()
        return new_graph.outputs
```

Background: pt2 symint tracing

Dynamo runs the user's Python with sizes as SymInts; each has a hint.
A branch on a size becomes a guard. The graph is reused wherever the guards hold.

```
user code
def f(x):
    if x.shape[0] > 4:
        return x * 2
    return x + 1
```

```
dynamo FX graph (dynamic=True, called with x = randn(8, 64))
def forward(self, s77: SymInt, s27: SymInt, L_x_: Tensor):
    mul = L_x_ * 2
    return (mul,)
```

```
guards
5 <= L['x'].size()[0]          <- the branch
2 <= L['x'].size()[1]          <- 0/1 specialization
L['x'].stride()[0] == L['x'].size()[1] <- row stride == width
```

`x = randn(3, 64)` fails `5 <= size[0]`: recompile, and trace the `x + 1` branch.

Tracing through Triton

```
@triton.jit
def add_kernel(x_ptr, # *Pointer* to first input vector.
              y_ptr, # *Pointer* to second input vector.
              output_ptr, # *Pointer* to output vector.
              n_elements, # Size of the vector.
              BLOCK_SIZE: tl.constexpr, # Number of elements each program should process.
              ):
    pid = tl.program_id(axis=0)
    block_start = pid * BLOCK_SIZE
    offsets = block_start + tl.arange(0, BLOCK_SIZE)
    mask = offsets < n_elements
    x = tl.load(x_ptr + offsets, mask=mask)
    y = tl.load(y_ptr + offsets, mask=mask)
    output = x + y
    tl.store(output_ptr + offsets, output, mask=mask)

def add(x: torch.Tensor, y: torch.Tensor):
    output = torch.empty_like(x)
    n_elements = output.numel()
    grid = lambda meta: (triton.cdiv(n_elements, meta['BLOCK_SIZE']), )
    add_kernel[grid](x, y, output, n_elements, BLOCK_SIZE=1024)
    return output
```

Triton tutorial 01: vector addition

Tracing through Triton

Triton's launcher, desugared into a PT2 custom op. Input pointers are passed in as SymInts.

```
@torch.library.custom_op("demo::add_kernel", mutates_args=("out",))
def add_kernel(x, y, out, x_ptr: int, y_ptr: int, out_ptr: int,
               n_elements: int, grid: int, BLOCK_SIZE: int,
               ptr_div16: bool, n_div16: bool, n_i32: bool) -> None:
    ... # the launch, with exactly these arguments

def add(x, y, x_ptr, y_ptr): # what Triton's launcher does, as user code
    out = torch.empty_like(x)
    n = out.numel()
    out_ptr = data_ptr(out)
    BLOCK_SIZE = 256 if n < 1000 else 1024 # config dispatch
    grid = (n + BLOCK_SIZE - 1) // BLOCK_SIZE
    ptr_div16 = x_ptr % 16 == 0 and y_ptr % 16 == 0 and out_ptr % 16 == 0
    n_div16 = n % 16 == 0 # Triton's specialization
    n_i32 = n <= 2**31 - 1
    add_kernel(x, y, out, x_ptr, y_ptr, out_ptr, n, grid,
               BLOCK_SIZE, ptr_div16, n_div16, n_i32)
    return out

add(x, y, x.data_ptr(), y.data_ptr()) # input pointers: SymInts
# data_ptr(out): 256 * u, u unbacked (caching allocator: 256-aligned)
```

post-grad graph (dynamic=True, n = 98432)

```
def forward(self, s77, x: f32[s77], s37, s11,
            y: f32[s77]):
    out = aten.empty([s77])
    u0 = demo.data_ptr(out).item()
    grid = (s77 + 1023) // 1024
    demo.add_kernel(x, y, out,
                    s37, s11, 256*u0,
                    s77, grid, 1024,
                    True, True, True)
```

guards

```
1000 <= L['x'].size()[0] <= 2147483647 # dispatch, i32
(L['x'].size()[0] % 16) == 0 # n % 16
(L['x_ptr'] % 16) == 0 # alignment
(L['y_ptr'] % 16) == 0
L['y'].size()[0] == L['x'].size()[0] # duck sizing
out_ptr = 256*u0: aligned, no guard
```

Tracing through Triton

We get this automatically, from the tutorial code unchanged:
we intercept the launcher that kernel[grid] returns and run Triton's own binder on SymInts.
Each specialization check it makes becomes a guard.

```
add_kernel[grid](x, y, output, n_elements, BLOCK_SIZE=1024)
```

	traced	guard
grid	(ceil(n/1024), 1, 1)	
x_ptr	x.data_ptr()	x_ptr % 16 == 0 # tt.divisibility 16
y_ptr	y.data_ptr()	y_ptr % 16 == 0 # tt.divisibility 16
output_ptr	out.data_ptr()	none: fresh allocation, 256-aligned
n_elements	n	n != 1, n % 16 == 0, n < 2^31

$n = 1000$ fails $n \% 16 == 0$: a different compilation, so a separate variant.

Updating Kernel Invocation Parameters

A captured kernel node is a function, a grid, and a buffer of parameter bytes.
For a new call we rewrite them: per parameter, where it lives and what goes there.

param	where	what (traced)	n = 4096	n = 16000	
grid		(s61 + 1023)//1024	4	16	
x_ptr	@0, 8 B	s8 + 4*s88	0xffffbbdc00000	0xffffbbdc1c200	new input
y_ptr	@8, 8 B	4*s34 + s38	0xffffbbdc04000	0xffffbbdc2bc00	new input
output_ptr	@16, 8 B	256*s69	0x333000000	0xffffbbdc3b600	new allocation
n_elements	@24, 4 B	s61	4096	16000	

Per call: evaluate the expressions, write the bytes, `cuGraphExecKernelNodeSetParams`.
The function and block stay the same.

[Triton binder: create function from signature, triton/runtime/jit.py](#)

Tracing through CUTE DSL

```
@cute.jit
def __call__(self, x_ptr, w_ptr, o_ptr, M, eps, stream):
    N = self.cfg.N # 4096: compile-time
    mX = cute.make_tensor(x_ptr,
                          cute.make_layout((M, N), stride=(N, 1)))
    mO = cute.make_tensor(o_ptr,
                          cute.make_layout((M, N), stride=(N, 1)))
    mW = cute.make_tensor(w_ptr, cute.make_layout((N,)))
    self.kernel(mX, mW, mO, eps, ...).launch(
        grid=[cute.ceil_div(M, 2), 1, 1], # depends on M
        block=[128, 1, 1], smem=16400, stream=stream)
```

user code (@cute.jit)

```
func.func @__call__(%arg0: !cute.ptr<bf16, gmem, align<16>>,
                  %arg1, %arg2, %arg3: i32, %arg4: f32, %arg5: !cuda.stream) {
    %shape = cute.make_shape(%arg3) // "(?, 4096)"
    %lay = cute.make_layout(%shape, %stride)
    %view = cute.make_view(%arg0, %lay) // mX
    ... // mO, mW likewise
    %shp = cute.ceil_div(%int_tuple, %tile)
    %0 = cute.get_scalars(%e0) // ceil_div(M, 2)
    %6 = cuda.launch_cfg.create(blockDim = (128, 1, 1),
                                dynamicSmemBytes = 16400, gridDim = (%0, 1, 1))
    %7 = cuda.launch_ex @kernels::@kernel_...<%6>(<
        %view, %view_8, %view_3, %arg4, %lay_12)
    }
```

emitted host MLIR

The DSL emits this host MLIR on every compile; we read it and interpret it with SymInts.

As with Triton, we work out how each launch parameter is computed and update it per call.

C++ CUDA kernels

```
void LayerNormKernelImplInternal(const Tensor& X, ...,
-   int64_t M, int64_t N, T_ACC eps,
+   c10::SymInt M, c10::SymInt N, c10::SymFloat eps,
    Tensor* Y, Tensor* mean, Tensor* rstd) {
-   const T* X_data = X.const_data_ptr<T>();
-   T* Y_data = Y->data_ptr<T>();
+   const c10::SymInt X_data = graph_trace::sym_const_data_ptr<T>(X);
+   const c10::SymInt Y_data = graph_trace::sym_mutable_data_ptr<T>(*Y);
    ...
    if (... && can_vec_X && can_vec_Y && ...) { // each term: a guard
        launch_vectorized_layer_norm_kernel<T, T_ACC, rms_norm>(N, M, eps, X_data, ...);
    } else {
-       RowwiseMomentsCUDAKernel<T, T_ACC, rms_norm><<<M, kThreads, 0, stream>>>(M, N, eps, X_data, ...);
+       graph_trace::launch(RowwiseMomentsCUDAKernel<T, T_ACC, rms_norm>, graph_trace::Grid(M), kThreads, 0, stream,
+           M, N, eps, X_data, ...);
-       LayerNormForwardCUDAKernel<T, T_ACC, rms_norm><<<M, kThreads, 0, stream>>>(M, N, X_data, ...);
+       graph_trace::launch(LayerNormForwardCUDAKernel<T, T_ACC, rms_norm>, graph_trace::Grid(M), kThreads, 0, stream,
+           M, N, X_data, ...);
    }
}
```

Layer norm forward: sizes and addresses become SymInts, <<<>>> becomes graph_trace::launch.
The kernels are untouched. A raw data_ptr() on a traced tensor raises: safe by default.

Nested Per Kernel Shape Env Guards

One CUDA graph means every kernel's guards are checked together.
One misaligned layer norm weight would miss the whole graph.

```
native_layer_norm(x, w, b):  
    out = empty_like(x)  
    mean = empty(M); rstd = empty(M)  
    if aligned(w, 8) and aligned(b, 8) and N % vec == 0:  
        launch(vectorized_layer_norm, ...)  
    else:  
        launch(rowwise_moments, ...)  
        launch(layer_norm_fwd, ...)
```

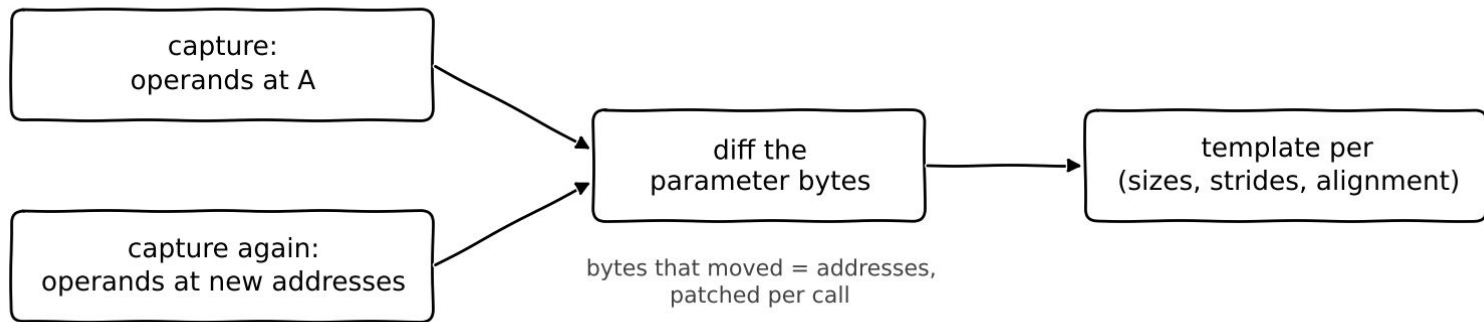
meta: output shapes
-> graph-level guards

kernel dispatch
-> per-kernel guards
(outputs identical either way)

While tracing, dispatch runs in a nested context, so its guards are tagged per kernel.
k layer norms: up to 2^k graphs as graph guards, or k independent per-kernel choices.

Handling Closed Libraries (cuBLAS)

cuBLAS is closed: we specialize completely on sizes, strides and alignment. Each combination is captured twice and diffed; only addresses are patched per call.



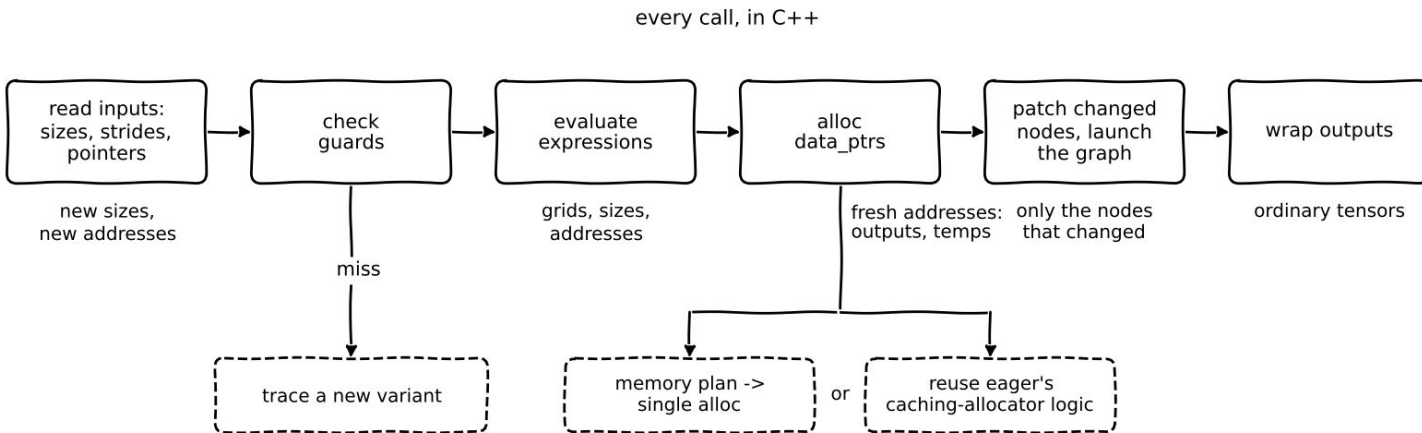
Opt-in per op; first serve checked against eager

What It Takes to Support a Kernel

	Triton	CuTe DSL	CUDA C++	Closed ops (cuBLAS)
User changes	none	none	retype sizes and pointers; launch through graph_trace	opt-in per op
New shape	no recapture	no recapture	no recapture	captured & diffed once per shape, then cached

Ops we can't trace or capture run as labelled eager steps, never silently wrong.

The Runtime, Per Call



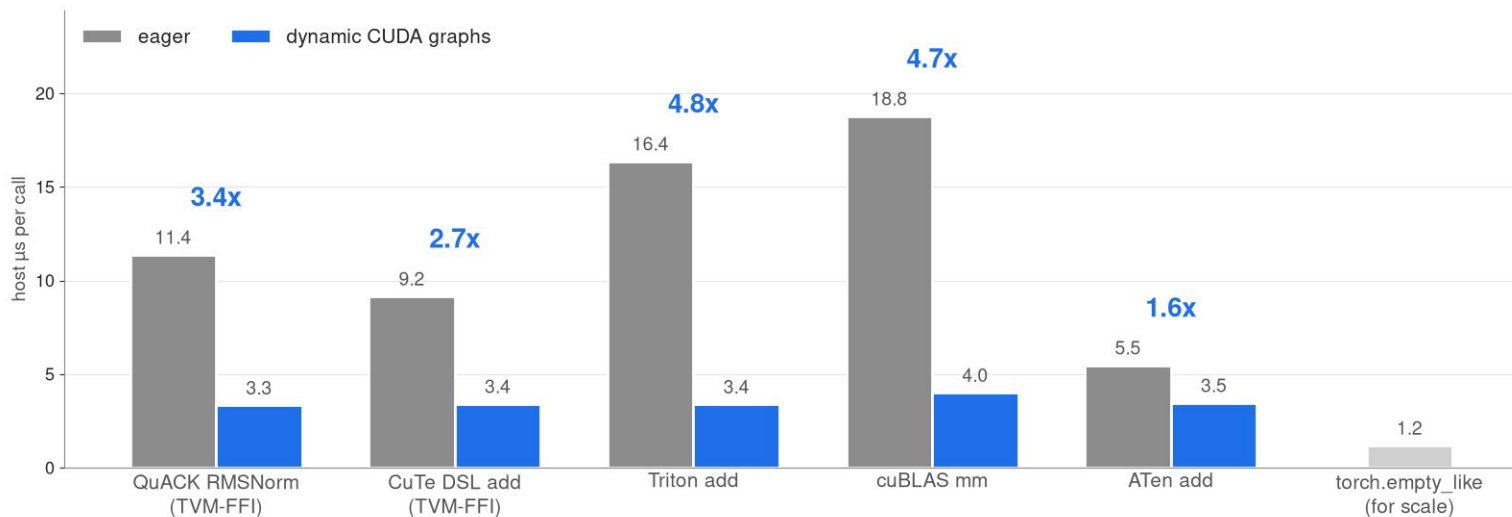
Every call gets fresh data pointers for its outputs and intermediates.

Intermediates get memory but no Tensor object

Optional: split the CUDA graph to hide CPU overhead (~2 us per split)

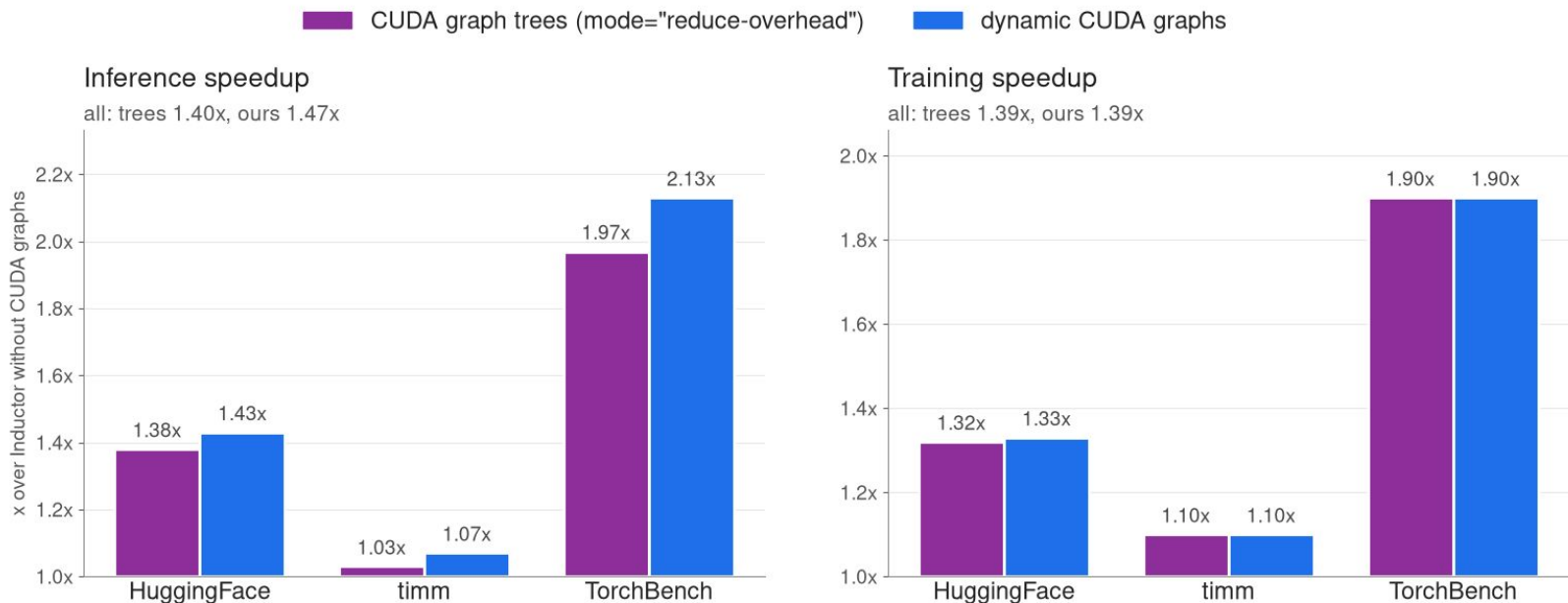
Easy Gradual Adoption

No memory cost, no copy-in cost: use it even on a single op.



Measured on GH200: host μs per call

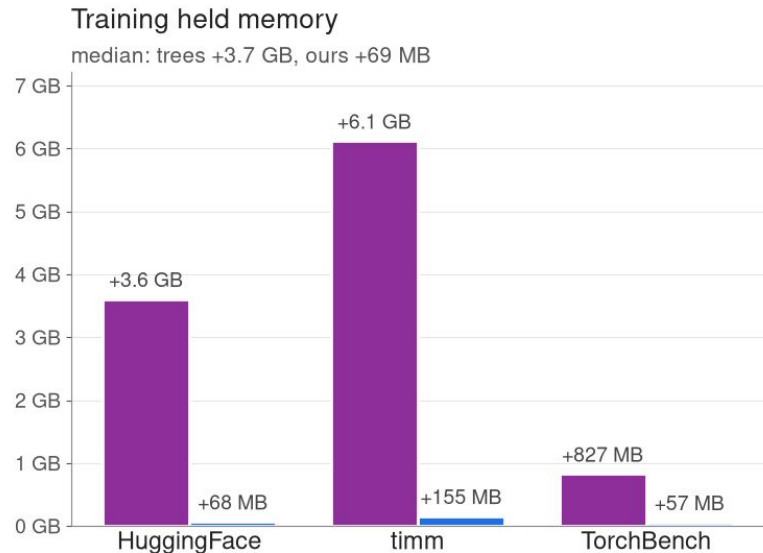
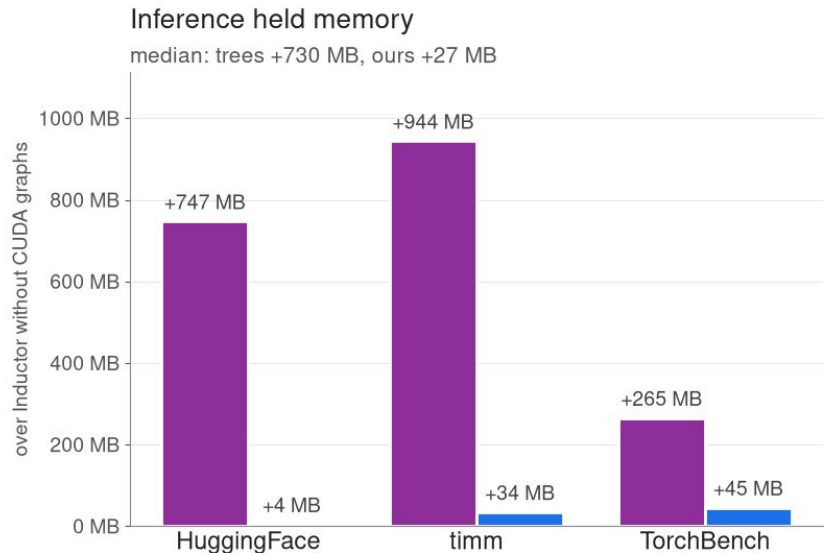
HF / timm / TorchBench: Speedup



Measured on GH200. Baseline: torch.compile without CUDA graphs.

HF / timm / TorchBench: Memory

■ CUDA graph trees (mode="reduce-overhead") ■ dynamic CUDA graphs



Measured on GH200. Baseline: torch.compile without CUDA graphs.

How vLLM Captures Today

Full CUDA Graph

capture the whole forward as one graph

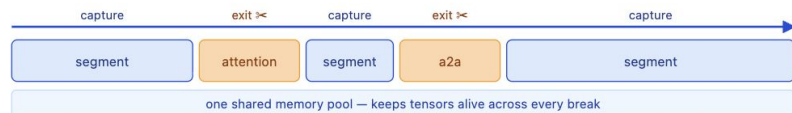
one capture



Breakable CUDA Graph

cut in time, while capturing — no whole-graph pass

one pass



TC Piecewise CUDA Graph

walk the whole graph first, then cut

① trace entire forward

Torch Dynamo traces every op

② split at registered split points

③ compile + capture each piece



vLLM CUDA graphs

decode: full graph, padded

prefill / mixed: eager

vLLM breakable (piecewise)

decode: full graph, padded

prefill / mixed ≤ 1024 : pieces, attention eager

> 1024 tokens: eager

vLLM default (torch.compile piecewise)

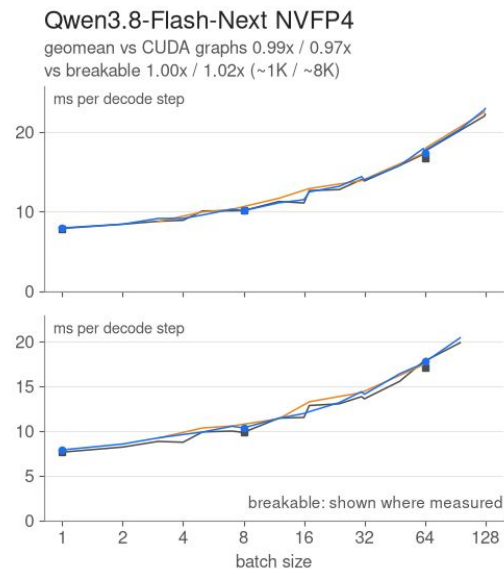
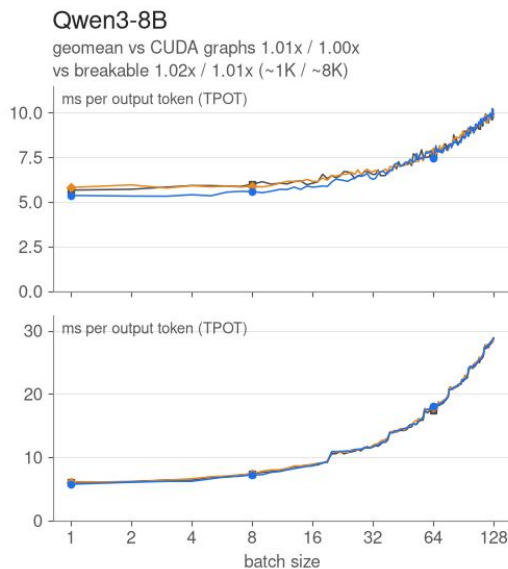
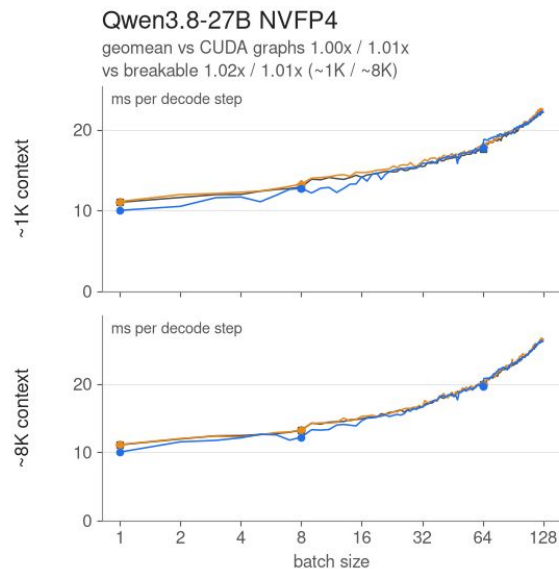
same pieces and 1024-token limit as breakable

Ours: one graph at every size and step type,
attention included, exact size (no padding).

Figure: LMSYS, "Advanced CUDA Graph Techniques in SGLang", 2026-08-17.

vLLM Decode

—■— vLLM CUDA graphs —●— vLLM breakable —●— ours



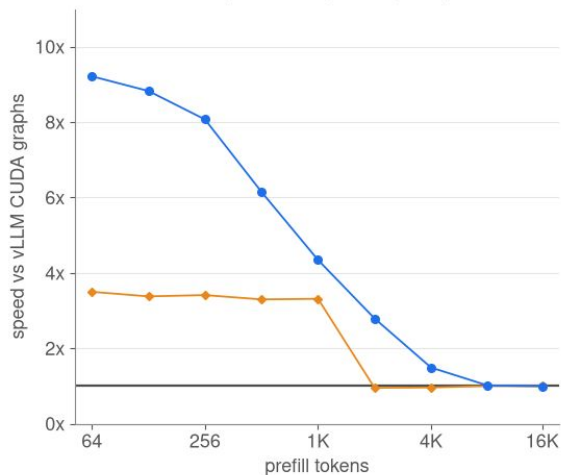
vLLM 0.29, measured on GB300

vLLM Prefill

— vLLM CUDA graphs = 1.0 — vLLM breakable — ours

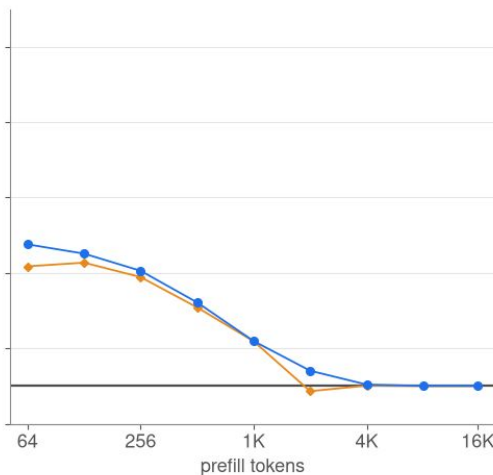
Qwen3.8-27B NVFP4

geomean vs CUDA graphs 3.47x
vs breakable 2.09x (<=1K tokens) / 1.46x (above)



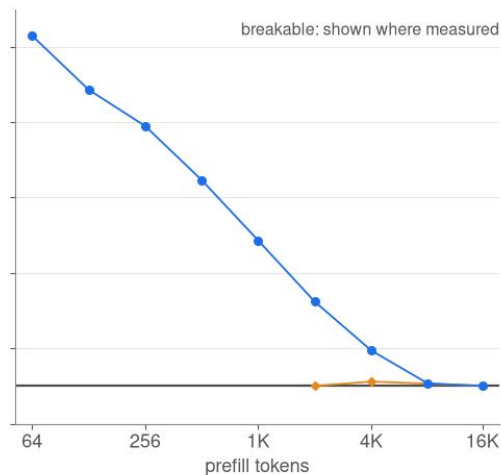
Qwen3-8B

geomean vs CUDA graphs 2.13x
vs breakable 1.09x



Qwen3.8-Flash-Next NVFP4

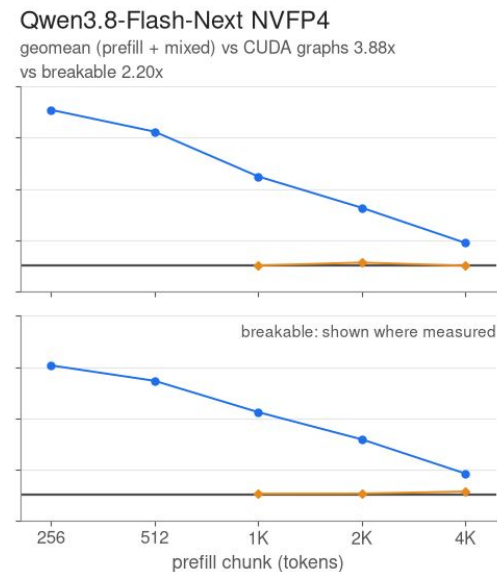
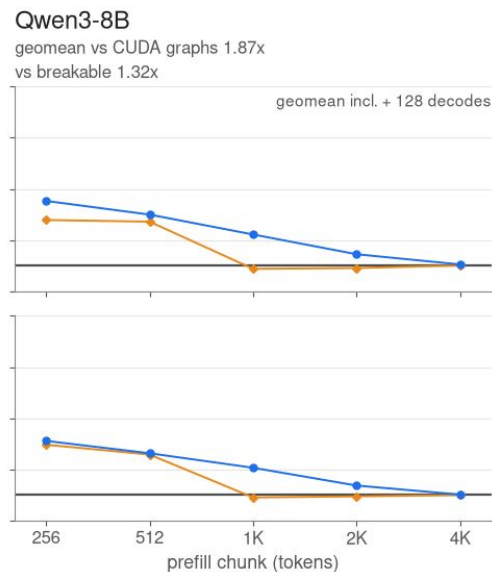
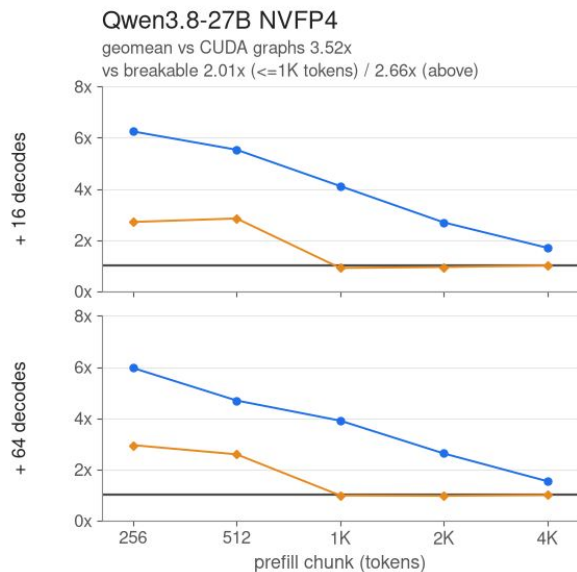
geomean (prefill + mixed) vs CUDA graphs 3.88x
vs breakable 2.20x



vLLM 0.29, measured on GB300

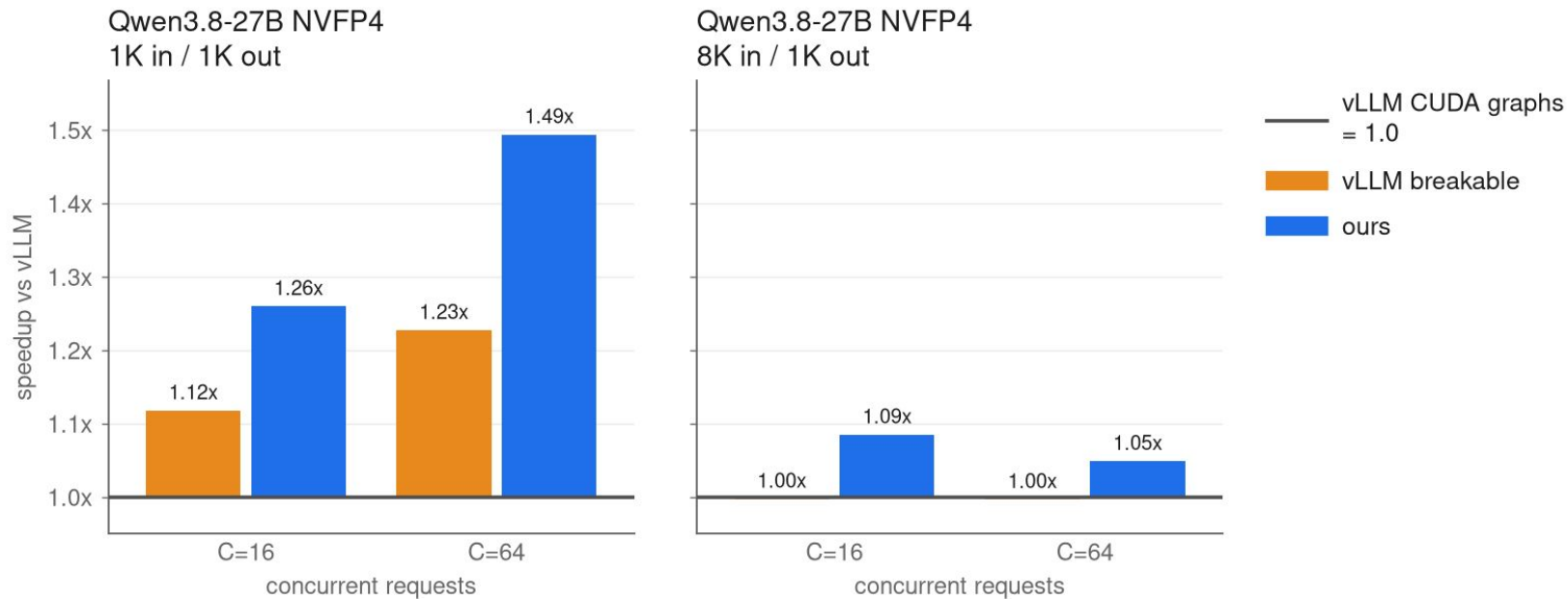
vLLM Mixed Prefill + Decode

— vLLM CUDA graphs = 1.0 — vLLM breakable — ours



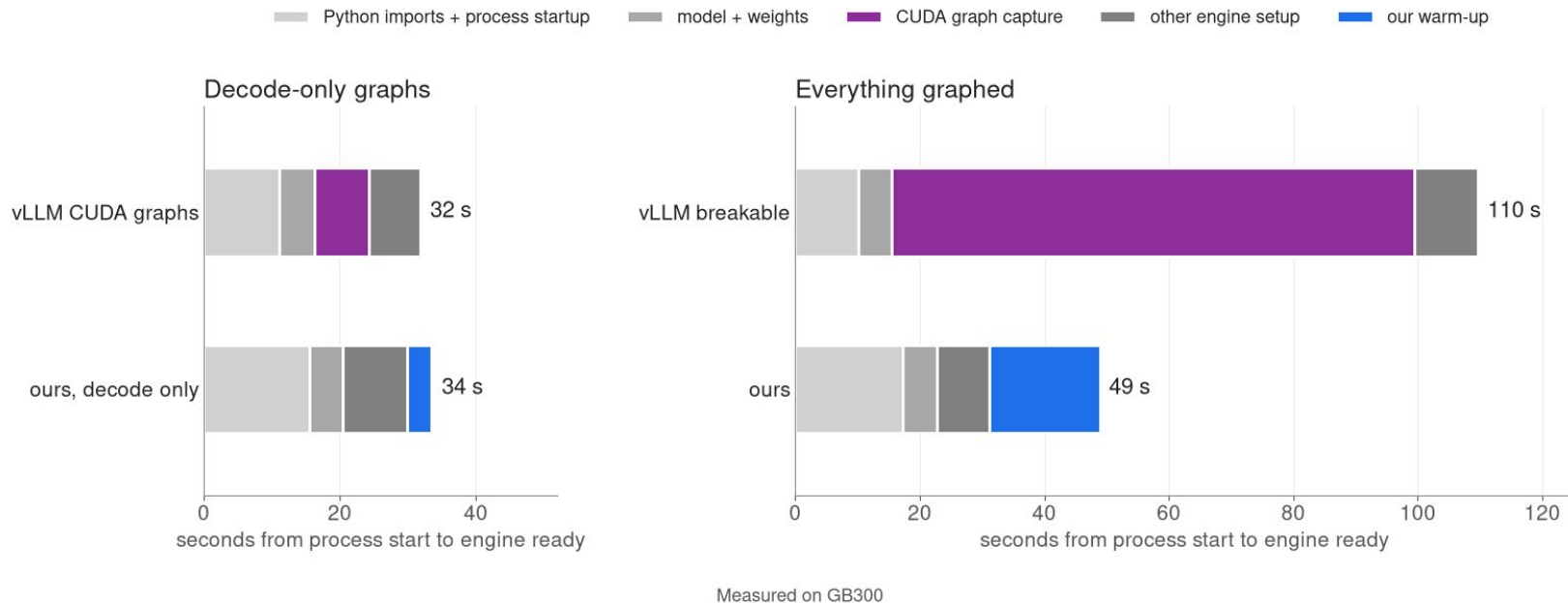
vLLM 0.29, measured on GB300

vLLM End-to-End Serving



InferenceX workload settings (1K/1K, 8K/1K), run in-process on vLLM's offline engine, GB300

Cold Start: Qwen3-8B, Ideal Case



Ideal case: our learned templates for closed-source kernels (cuBLAS) are persisted from a one-time offline run
Not implemented yet: overlapping our warm-up with model and weight loading. vLLM 0.29, GB300.

Next Steps and Conclusion

- Work with the CUTE DSL Team for less invasive symbolic tracing
- Upstream private branch to pytorch
- Work with flash infer, other libraries for support
- Integrate with vLLM and SGLang
- Default runtime for torch.compile
- Breakable CUDAGraphs-compatible API