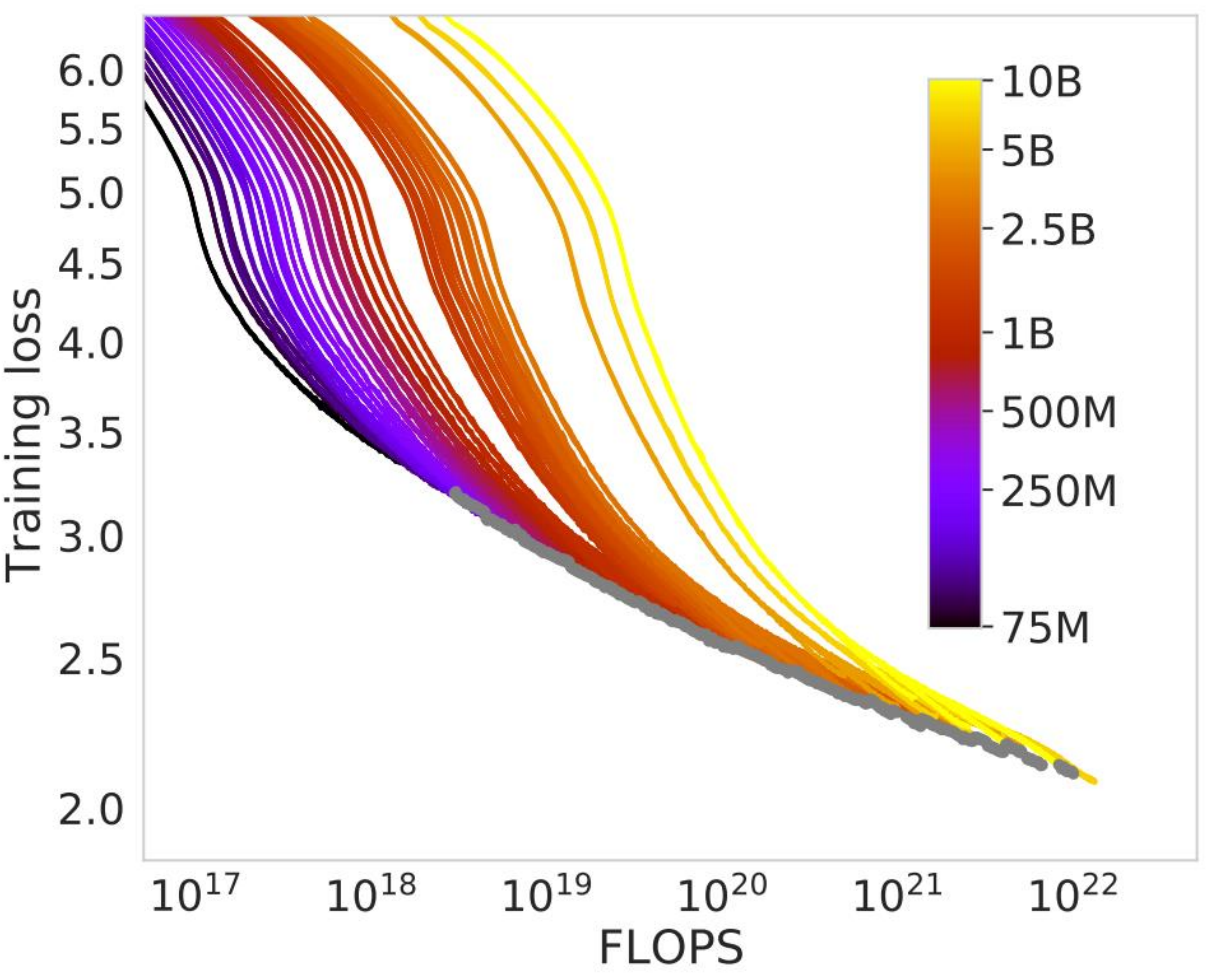
DSL for GPU Kernels
& Automatic Kernel Authoring with LLMs

Tri Dao
https://tridao.me

Scaling Laws: More Compute + Better Algorithms + More Data = More Capabilities

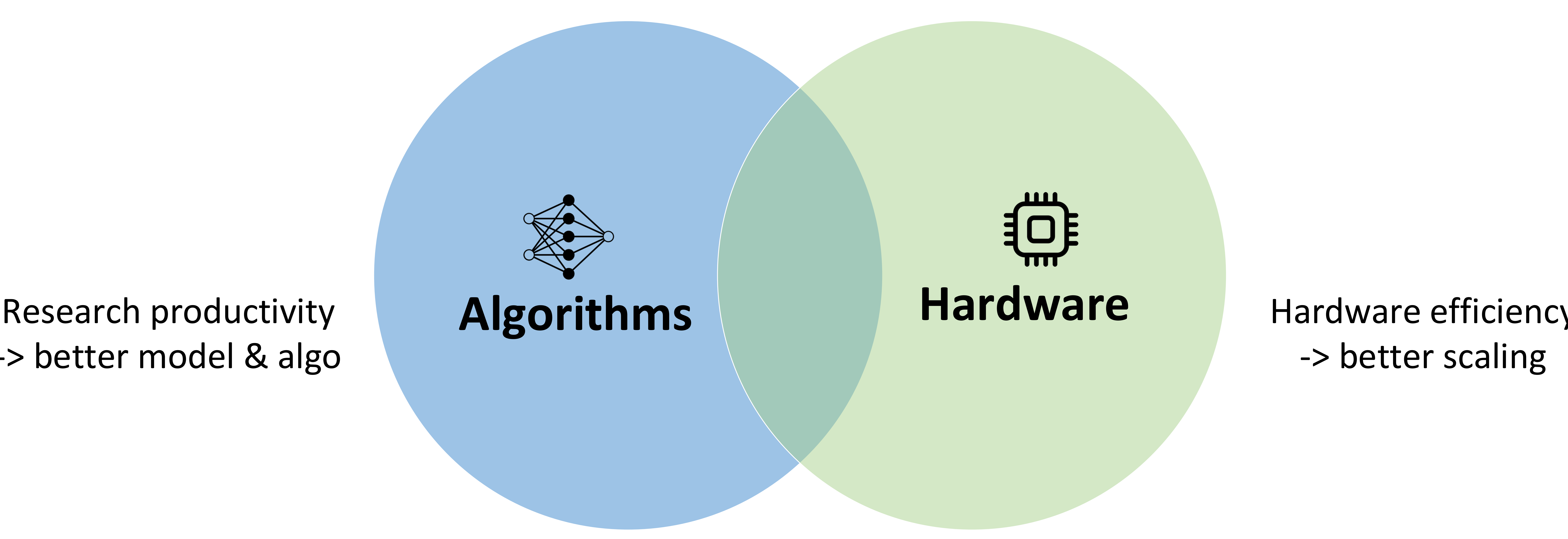


$$\frac{\text{Intelligence}}{\text{Dollar}} = \frac{\text{Intelligence}}{\text{FLOPS}} \times \frac{\text{FLOPS}}{\text{Dollar}}$$

Algorithm & data efficiency

Hardware efficiency

Why DSL: Research productivity + hardware efficiency



Bonus: DSL + good abstractions -> easy for LLMs to generate kernels

Outlines

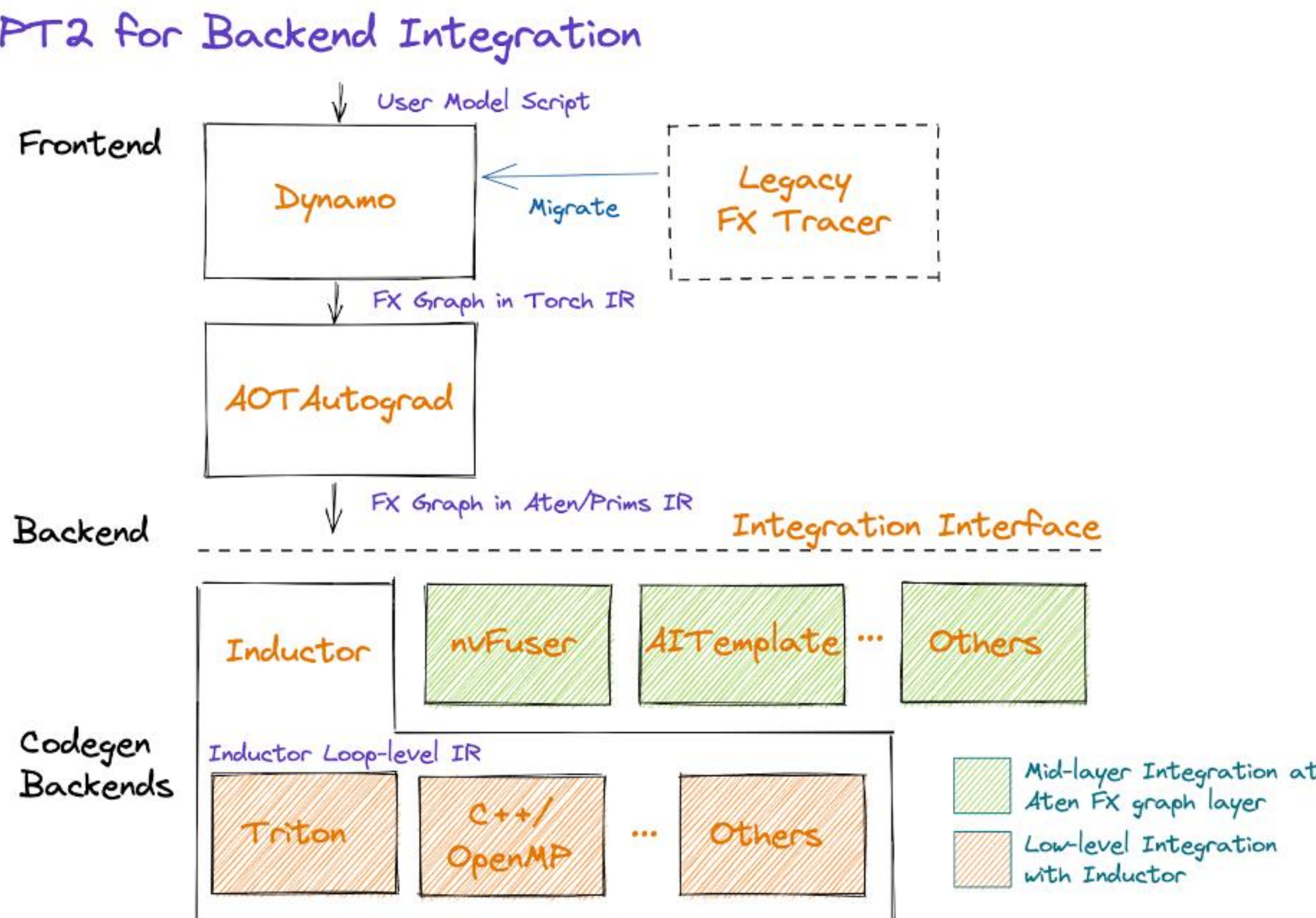
Kernels for AI

DSLs
Softmax example
Gemm & Attn perf

AI for Kernels

Data
Eval
Algorithms

PyTorch



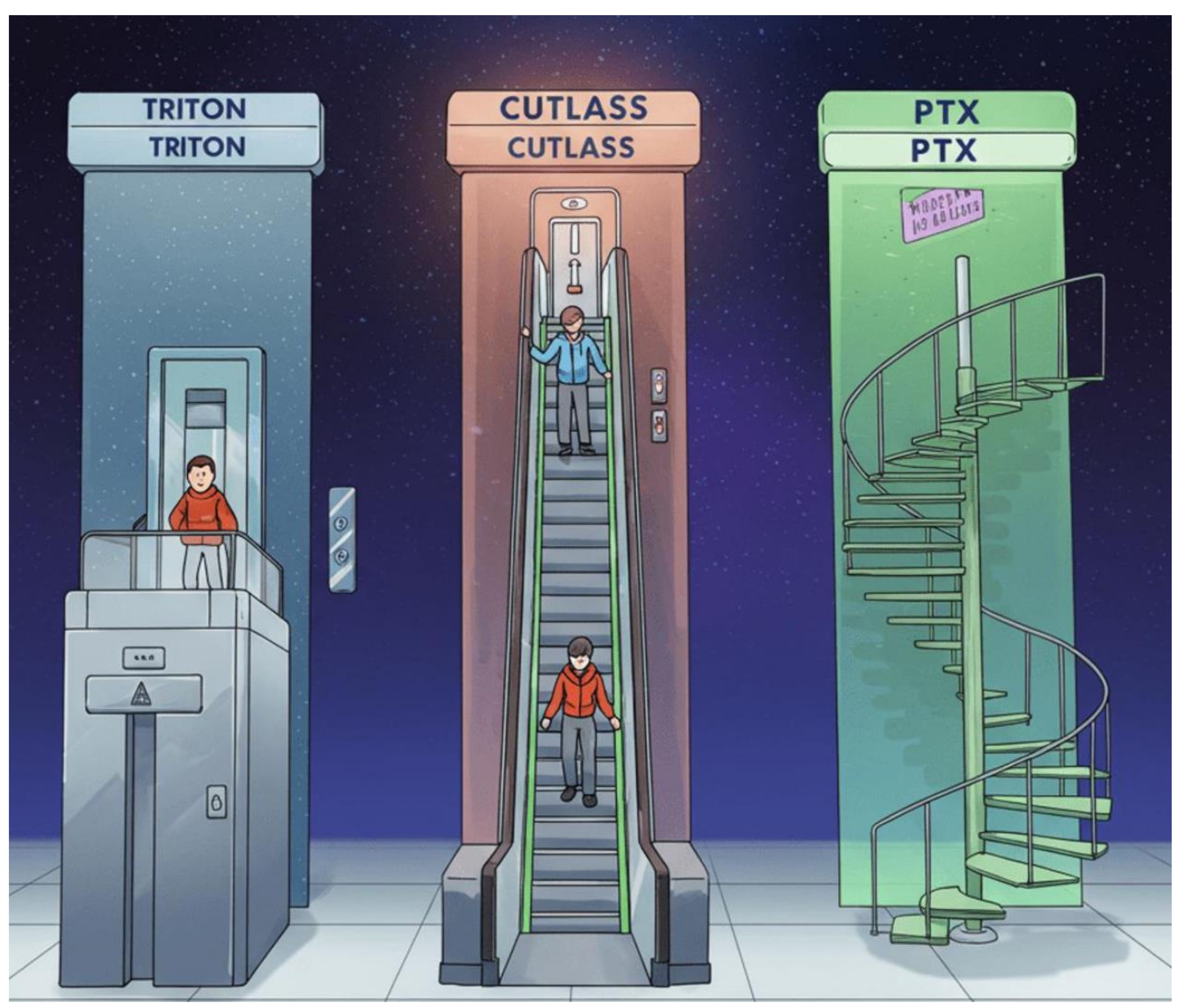
https://pytorch.org/get-started/pytorch-2-x/

Triton: Tile-based GPU kernel programming

	CUDA	TRITON
Memory Coalescing	Manual	Automatic
Shared Memory Management	Manual	Automatic
Scheduling (Within SMs)	Manual	Automatic
Scheduling (Across SMs)	Manual	Manual

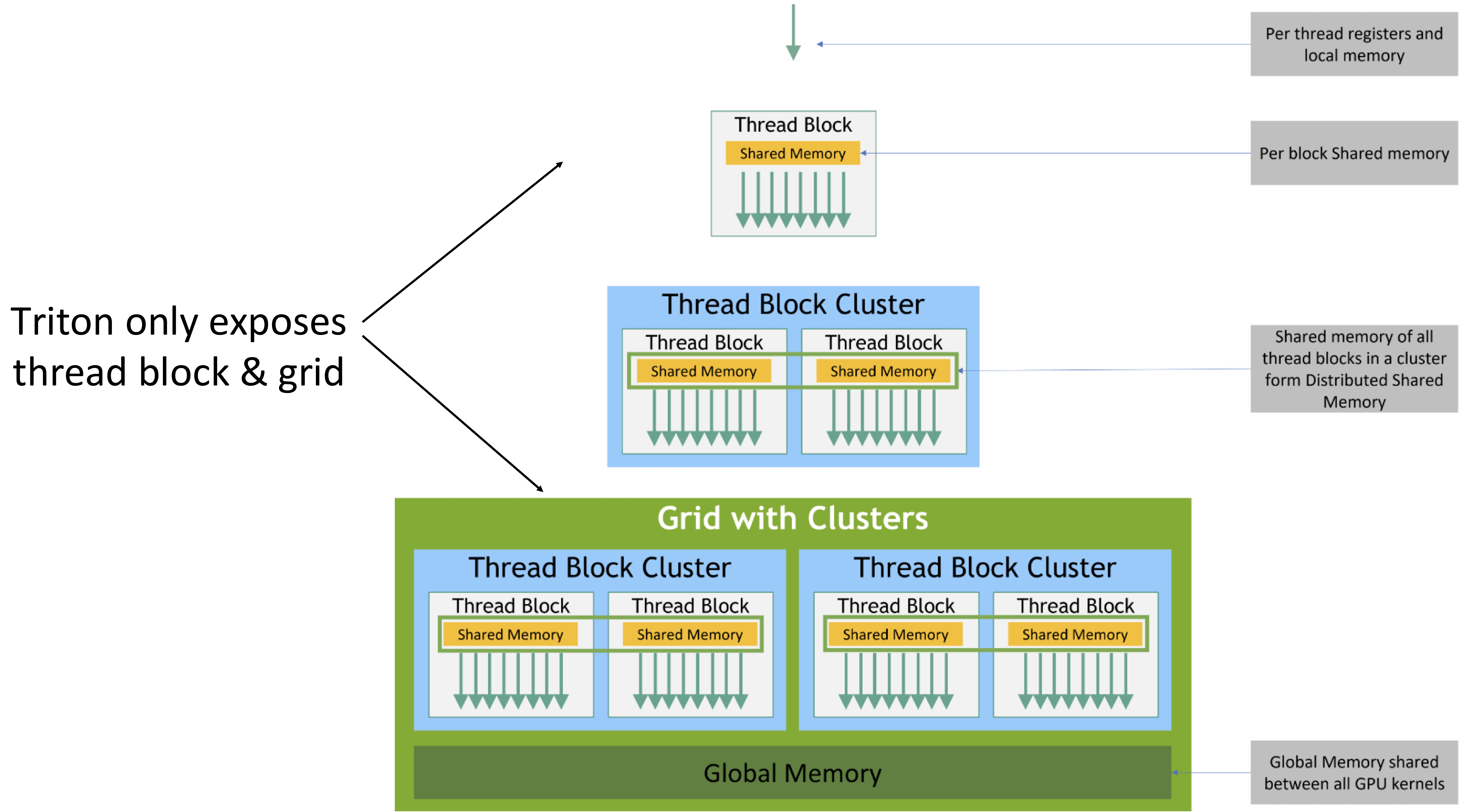
https://openai.com/index/triton/

Cute-DSL: Cutlass C++ embedded in Python



https://www.nvidia.com/en-us/on-demand/session/gtc25-s74639/

Cute-DSL: Expose all 4 levels of thread / memory hierarchy



Other DSLs

ThunderKittens: Simple, Fast, and *Adorable* AI Kernels

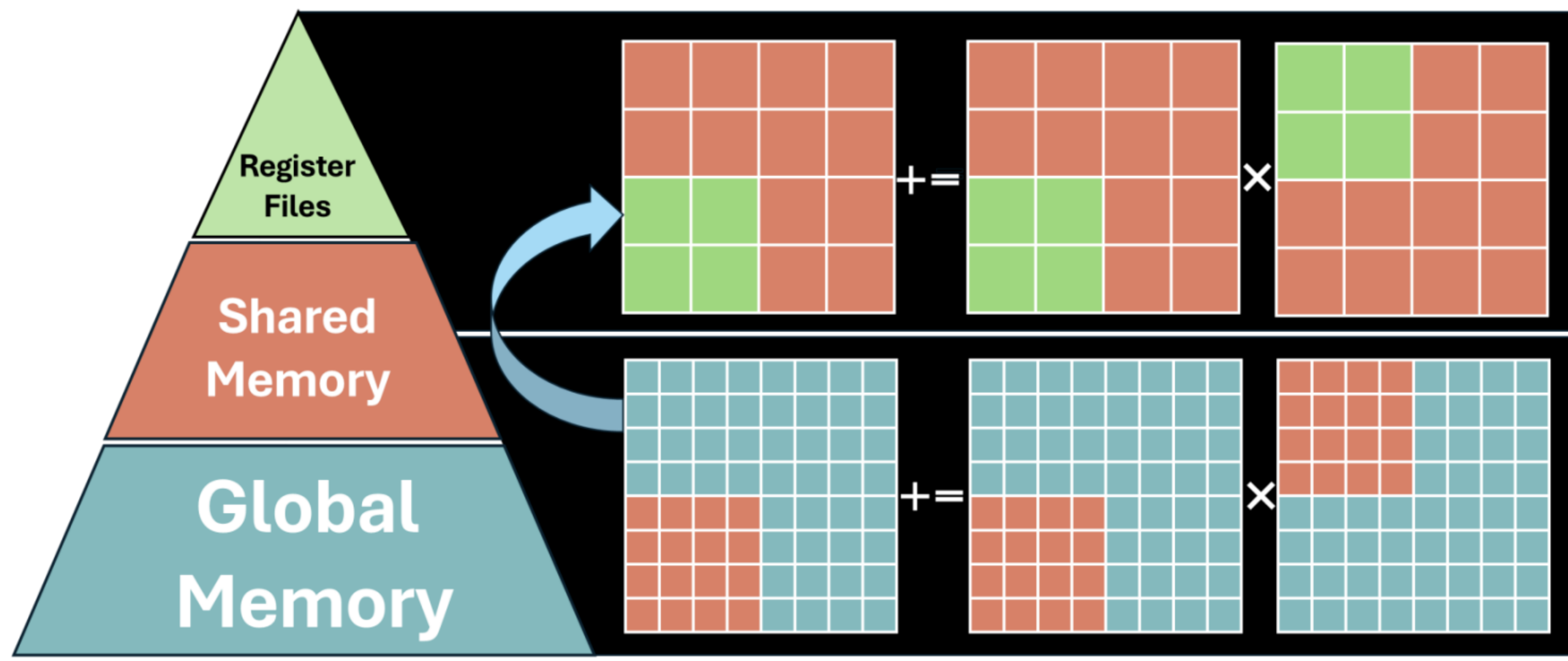
Benjamin F. Spector, Simran Arora, Aaryan Singhal, Daniel Y. Fu, and Christopher Ré

Stanford University

Global Memory

Shared Memory

Register Files



```
@parameter
for n_mma in range(num_n_mmas):
    alias mma_id = n_mma * num_m_mmas + m_mma

    var mask_frag_row = mask_warp_row + m_mma *
MMA_M
    var mask_frag_col = mask_warp_col + n_mma *
MMA_N

    @parameter
    if is_nvidia_gpu():
        mask_frag_row += lane // (MMA_N //
p_frag_simdwidth)
        mask_frag_col += lane * p_frag_simdwidth %
MMA_N
    elif is_amd_gpu():
        mask_frag_row += (lane // MMA_N) *
```

Mosaic GPU

```
buffers = 3  # In reality you might want even more
assert a_smem.shape == (buffers, m, k)
assert b_smem.shape == (buffers, k, n)
assert acc_ref.shape == (m, n)

def fetch_a_b(ki, slot):
  a_slice = ... # Replace with the right M/K slice
  b_slice = ... # Replace with the right K/N slice
  plgpu.copy_gmem_to_smem(a_gmem.at[a_slice], a_smem.at[slot], a_loaded.at[slot])
  plgpu.copy_gmem_to_smem(b_gmem.at[b_slice], b_smem.at[slot], b_loaded.at[slot])

def loop_body(i, _):
  slot = jax.lax.rem(i, buffers)
  plgpu.barrier_wait(a_loaded.at[slot])
  plgpu.barrier_wait(b_loaded.at[slot])
  plgpu.wgmma(acc_ref, a_smem.at[slot], b_smem.at[slot])
  # We know that only the last issued WGMMA is running, so we can issue a async load in
  # into the other buffer
  load_i = i + buffers - 1
  load_slot = jax.lax.rem(load_i, buffers)
  @pl.when(jnp.logical_and(load_i >= buffers, load_i < num_steps))
  def _do_fetch():
    fetch_a_b(load_i, slot)
for slot in range(buffers):
  fetch_a_b(slot, slot)
jax.lax.fori_loop(0, num_steps, loop_body, None)
```

Softmax: Torch compile

```python
import torch
import torch.nn.functional as F

def softmax(x):
   return F.softmax(x, dim=-1)
```

```python
import torch
import torch.nn.functional as F

@torch.compile
def softmax(x):
   return F.softmax(x, dim=-1)
```

Softmax: Triton (1)

```
@triton.jit
def _softmax_single_block_forward_kernel(
    Y_ptr,
    Y_row_stride,
    X_ptr,
    X_row_stride,
    n_cols,
    BLOCK_SIZE: tl.constexpr,
):
    row_id = tl.program_id(0)
    offs = tl.arange(0, BLOCK_SIZE)
    mask = offs < n_cols

    x = tl.load(X_ptr + row_id * X_row_stride + offs, mask=mask, other=-float("inf"), cache_modifier=".ca")
    m = tl.max(x, axis=0)
    e = tl.exp(x - m)
    d = tl.sum(e, axis=0)
    y = e / d
    tl.store(Y_ptr + row_id * Y_row_stride + offs, y, mask=mask, cache_modifier=".cs")
```

https://github.com/linkedin/Liger-Kernel/blob/main/src/liger_kernel/ops/softmax.py

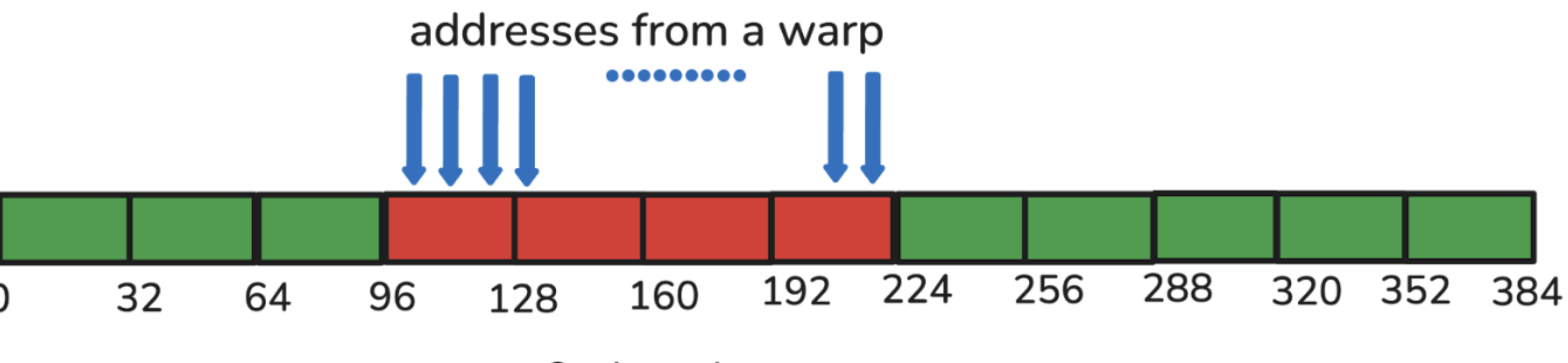
Softmax: Triton (2)

```
@triton.jit
def _softmax_multi_block_forward_kernel(
    Y_ptr,
    Y_row_stride,
    X_ptr,
    X_row_stride,
    n_cols,
    BLOCK_SIZE: tl.constexpr,
):
    row_id = tl.program_id(0)
    offs = tl.arange(0, BLOCK_SIZE)

    m = tl.float32(-float("inf"))
    d = tl.float32(0.0)
    for start in tl.range(0, n_cols, BLOCK_SIZE):
        idx = start + offs
        mask = idx < n_cols
        xblk = tl.load(X_ptr + row_id * X_row_stride + idx, mask=mask, other=-float("inf"), cache_modifier=".ca")
        blk_max = tl.max(xblk, axis=0)
        new_m = tl.max(m, blk_max)
        d = d * tl.exp(m - new_m) + tl.sum(tl.exp(xblk - new_m), axis=0)
        m = new_m

    for start in tl.range(0, n_cols, BLOCK_SIZE):
        idx = start + offs
        mask = idx < n_cols
        xblk = tl.load(X_ptr + row_id * X_row_stride + idx, mask=mask, other=-float("inf"), cache_modifier=".ca")
        yblk = tl.exp(xblk - m) / d
        tl.store(Y_ptr + row_id * Y_row_stride + idx, yblk, mask=mask, cache_modifier=".cs")
```

https://github.com/linkedin/Liger-Kernel/blob/main/src/liger_kernel/ops/softmax.py

Softmax: Cute-DSL async copy

```
# blkX: logical id -> address
blkX = ...
# allocate shared memory for the input vectors
smem = cutlass.utils.SmemAllocator()
sX   = smem.allocate_tensor(gX.element_type, ...)
# declare the copy atoms which will be used later for memory copy
copy_atom_load_X_async = cute.make_copy_atom(
                                cute.nvgpu.cpasync.CopyG2SOp(),
                                gX.element_type, num_bits_per_copy=128)
# create a tiled type given a TV partitioner and tiler
thr_copy_X_async = cute.make_tiled_copy(copy_atom_load_X_async,
                                        tv_layout,
                                        tiler_mn).get_slice(tidx)
# partition the inputs in gmem and smem with TV layout and tiler
tXgX = thr_copy_X_async.partition_S(blkX)
tXsX = thr_copy_X_async.partition_S(sX)
# allocate registers
tXrX = cute.make_fragment_like(tXgX)
# asynchronously copy inputs from gmem to smem
cute.copy(copy_atom_load_X_async, tXgX, tXsX)
# commit and wait until current async copy op finishes
cute.arch.cp_async_commit_group()
cute.arch.cp_async_wait_group(0)
# copy from smem to registers
cute.autovec_copy(tXsX, tXrX)
x = tXrX.load()
```



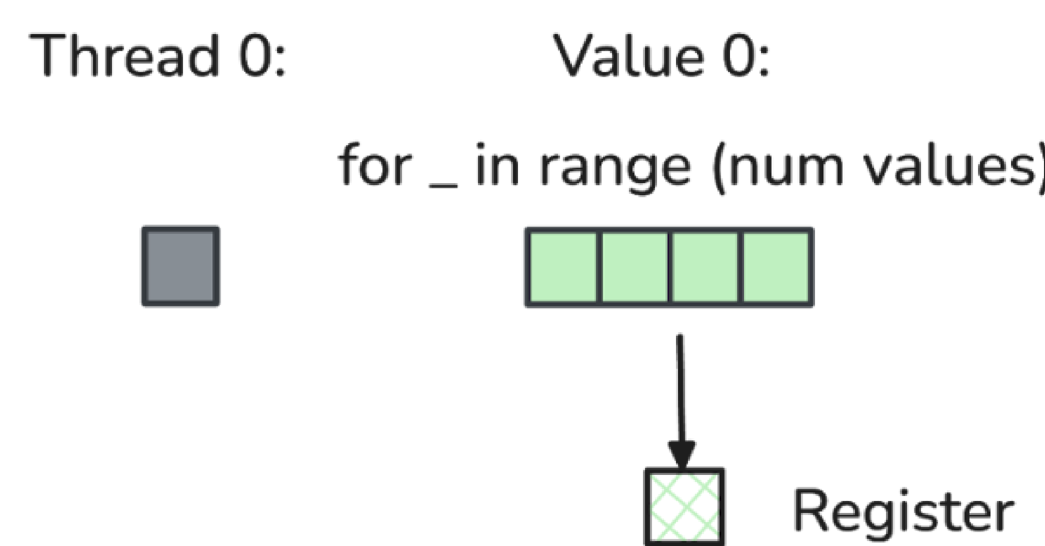
Softmax: Cute-DSL thread reduction

CuTe member function:

```
TensorSSA.reduce(op, init_val, reduction_profile)
```

Our example usage:

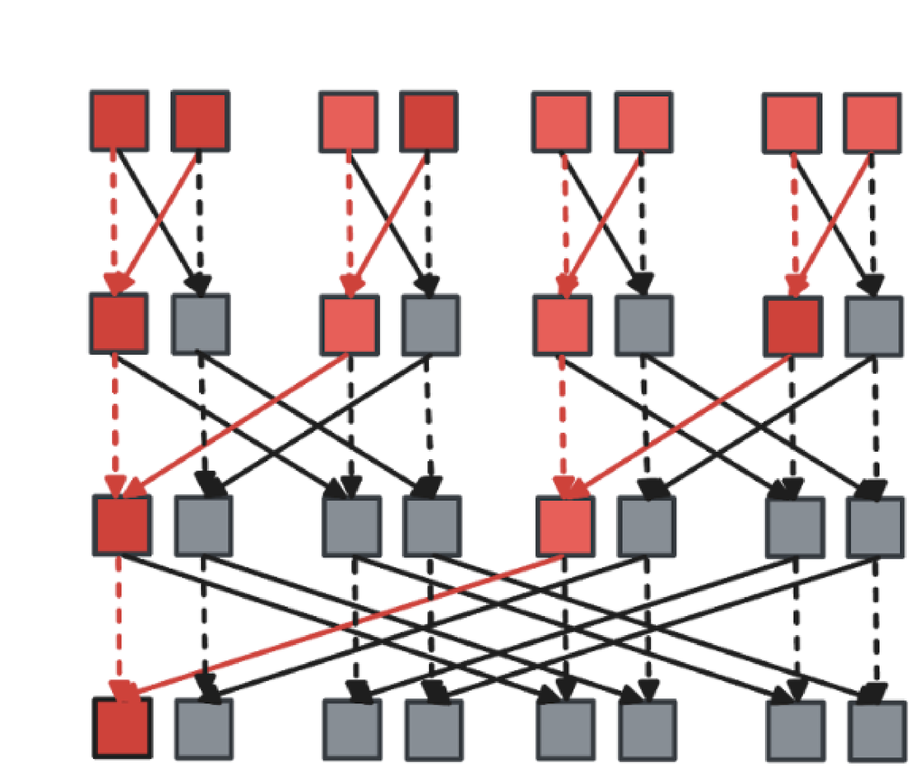
```
max_x = x.reduce(cute.ReductionOp.MAX, init_val=float('-inf'),
                 reduction_profile=0)
```



Softmax: Cute-DSL warp reduction

Our helper function:

```
@cute.jit
def warp_reduce(val: cute.Numeric,
                op: Callable,
                width: cutlass.Constexpr = 32) -> cute.Numeric:
    for i in range(int(math.log2(width))):
        # cute.arch.shuffle_sync_bfly will read from another thread's registers
        val = op(val, cute.arch.shuffle_sync_bfly(val, offset=1 << i))
    return val
```



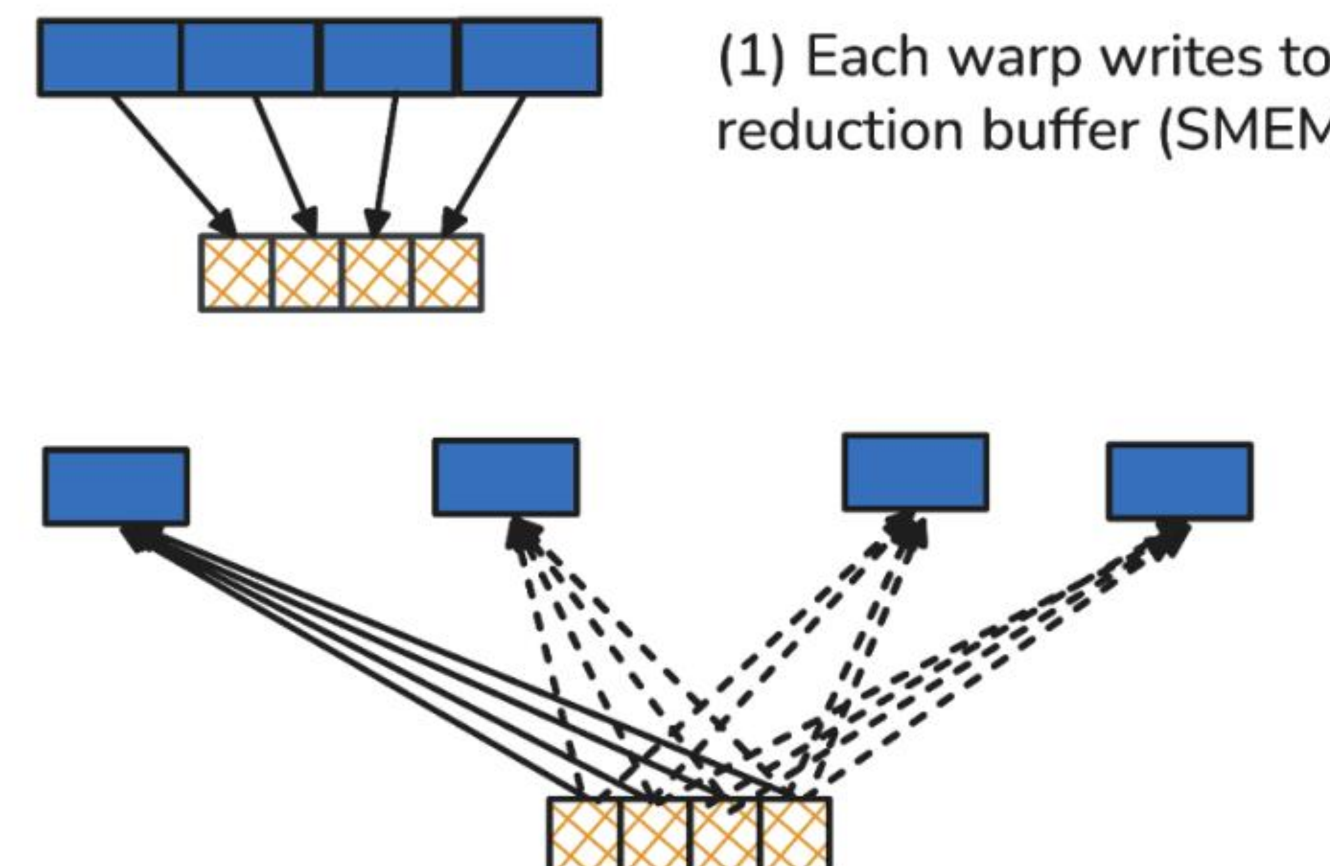
Our example usage:

```
    max_x  = x.reduce(cute.ReductionOp.MAX, init_val=float('-inf'),
                      reduction_profile=0)
    max_x  = warp_reduce(
        max_x, cute.arch.fmax,
        width=32, # every thread in a warp will participate in the warp reduction
    )
```

Softmax: Cute-DSL thread block reduction

Our helper function:

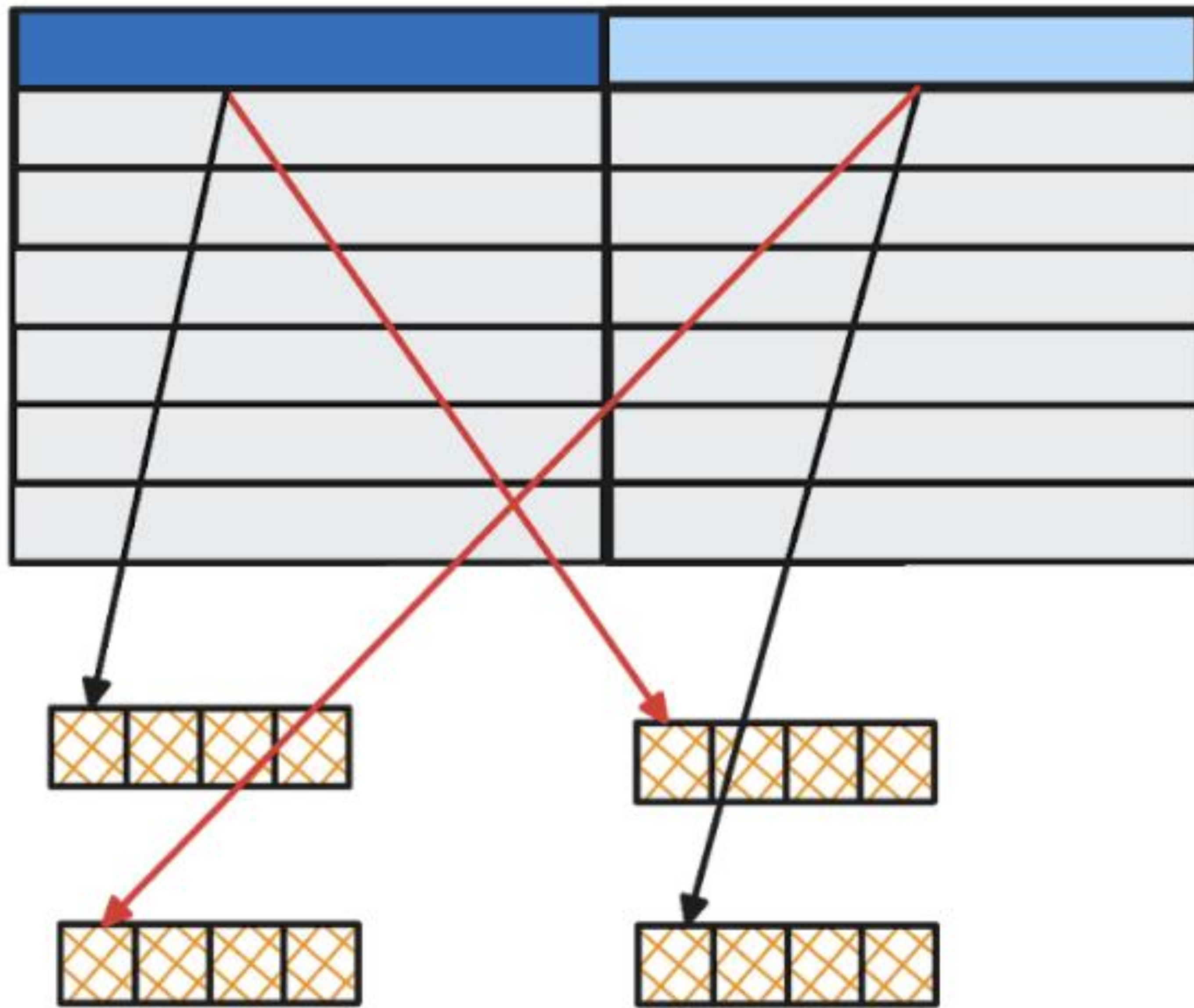
```
@cute.jit
def block_reduce(val: cute.Numeric,
                 op: Callable,
                 reduction_buffer: cute.Tensor,
                 init_val: cute.Numeric = 0.0) -> cute.Numeric:
    lane_idx, warp_idx = cute.arch.lane_idx(), cute.arch.warp_idx()
    warps_per_row      = reduction_buffer.shape[1]
    row_idx, col_idx   = warp_idx // warps_per_row, warp_idx % warps_per_row
    if lane_idx == 0:
        # thread in lane 0 of each warp will write the warp-reduced value to the
reduction buffer
        reduction_buffer[row_idx, col_idx] = val
    # synchronize the write results
    cute.arch.barrier()
    block_reduce_val = init_val
    if lane_idx < warps_per_row:
        # top-laned threads of each warp will read from the buffer
        block_reduce_val = reduction_buffer[row_idx, lane_idx]
    # then warp-reduce to get the block-reduced result
    return warp_reduce(block_reduce_val, op)
```



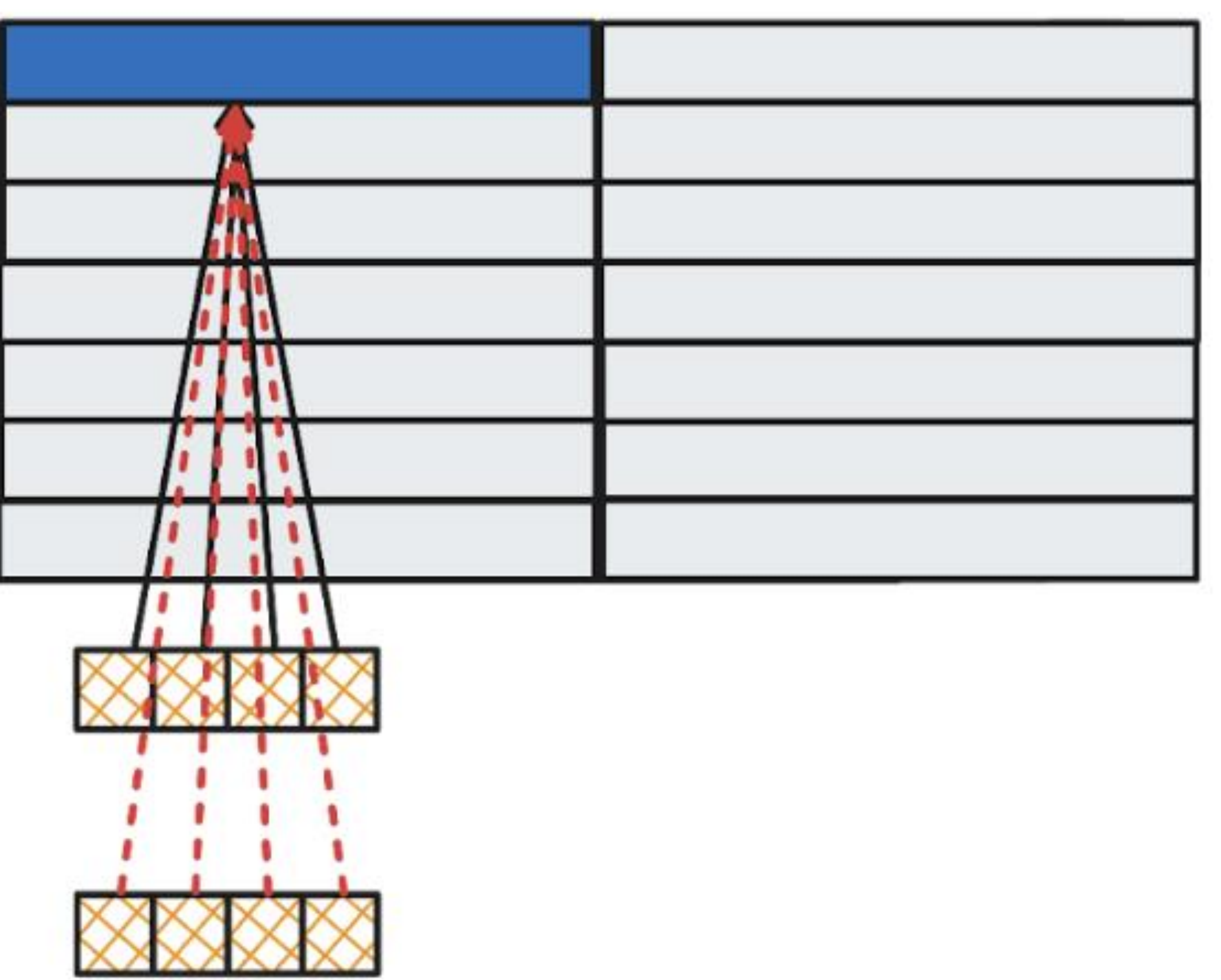
Our example usage:

```
max_x = block_reduce(max_x, cute.arch.fmax, max_val_reduction_buffer,
                     init_val=max_x)
```

Softmax: Cute-DSL cluster reduction



(1) Each warp writes to the reduction buffer (SMEM) of its thread block & other thread blocks within its cluster (due to the distributed shared memory feature of H100)

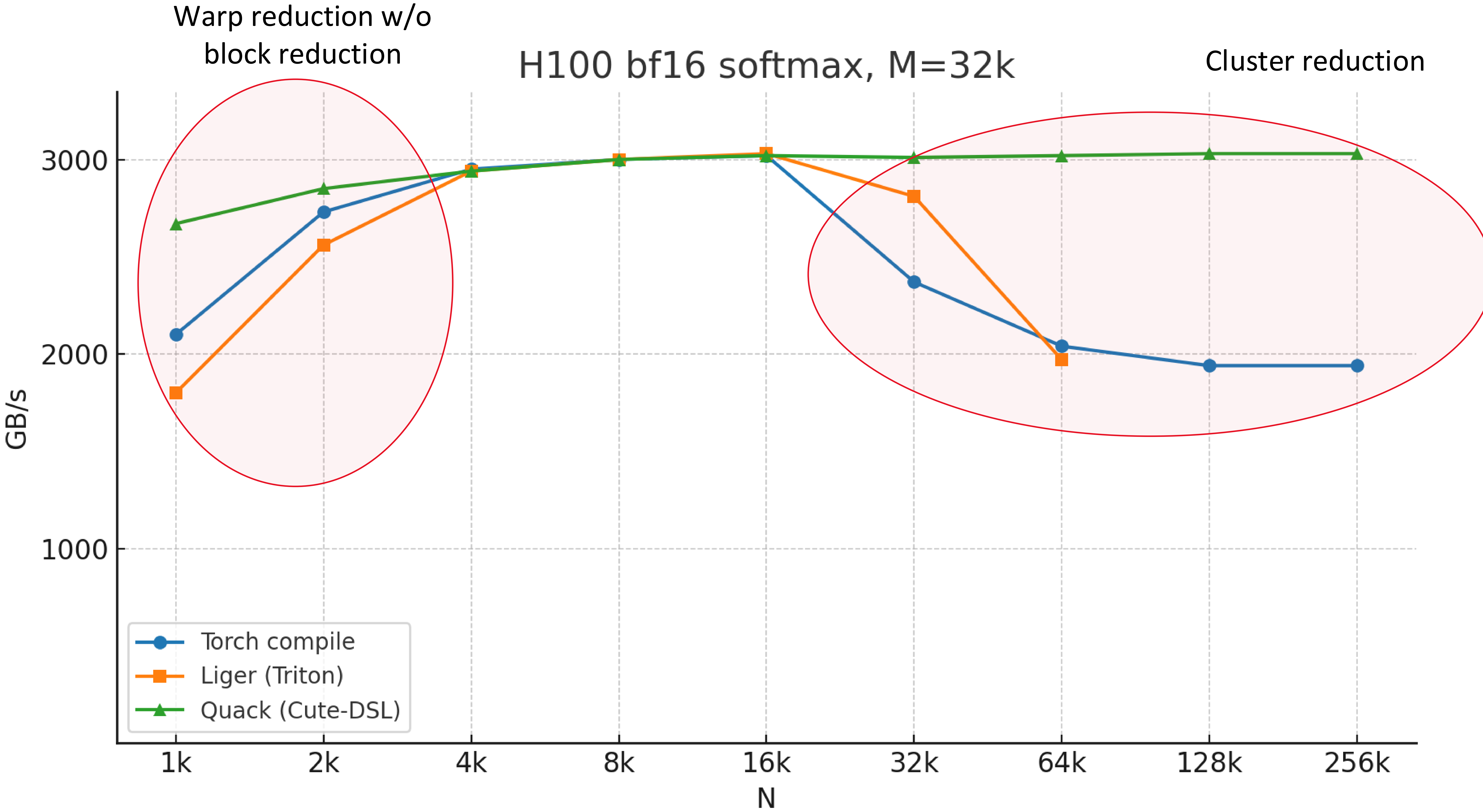


(2) Each warp reads from the reduction buffer (SMEM) of its thread block.

Softmax: reduction from top to bottom

```
# load from GMEM to the thread-owned registers
x = tXrX.load().to(cute.Float32)
# thread reduction
max_x = x.reduce(cute.ReductionOp.MAX,
                 init_val=float('-inf'),
                 reduction_profile=0)
# warp reduction
max_x = warp_reduce(max_x, cute.arch.fmax, width=32)

if cutlass.const_expr(warps_per_row * cluster_n) > 1:
    # more than 1 warp per row, block or cluster reduction is needed!
    if cutlass.const_expr(cluster_n) == 1:
        # block reduction
        max_x = block_reduce(max_x, cute.arch.fmax,
                             max_val_reduction_buffer,
                             init_val=max_x)
    else:
        # cluster reduction
        max_x = cluster_reduce(max_x, cute.arch.fmax,
                               max_val_reduction_buffer,
                               max_val_mbar_ptr,
                               init_val=max_x)
```

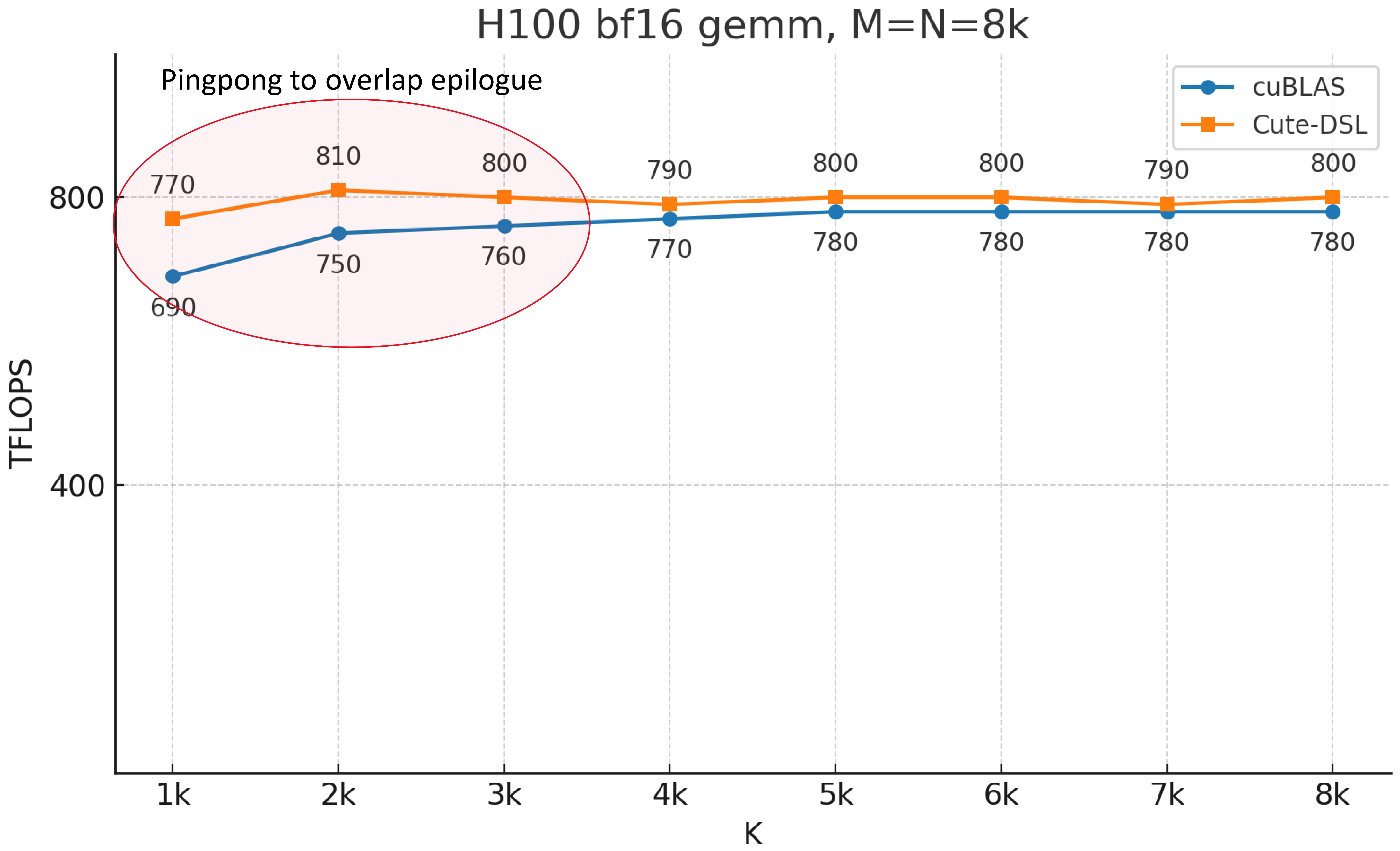
Softmax: Perf



https://github.com/Dao-AILab/quack/blob/main/quack/softmax.py

How much perf do we give up by using a DSL vs CUDA C++ for more complicated kernels?

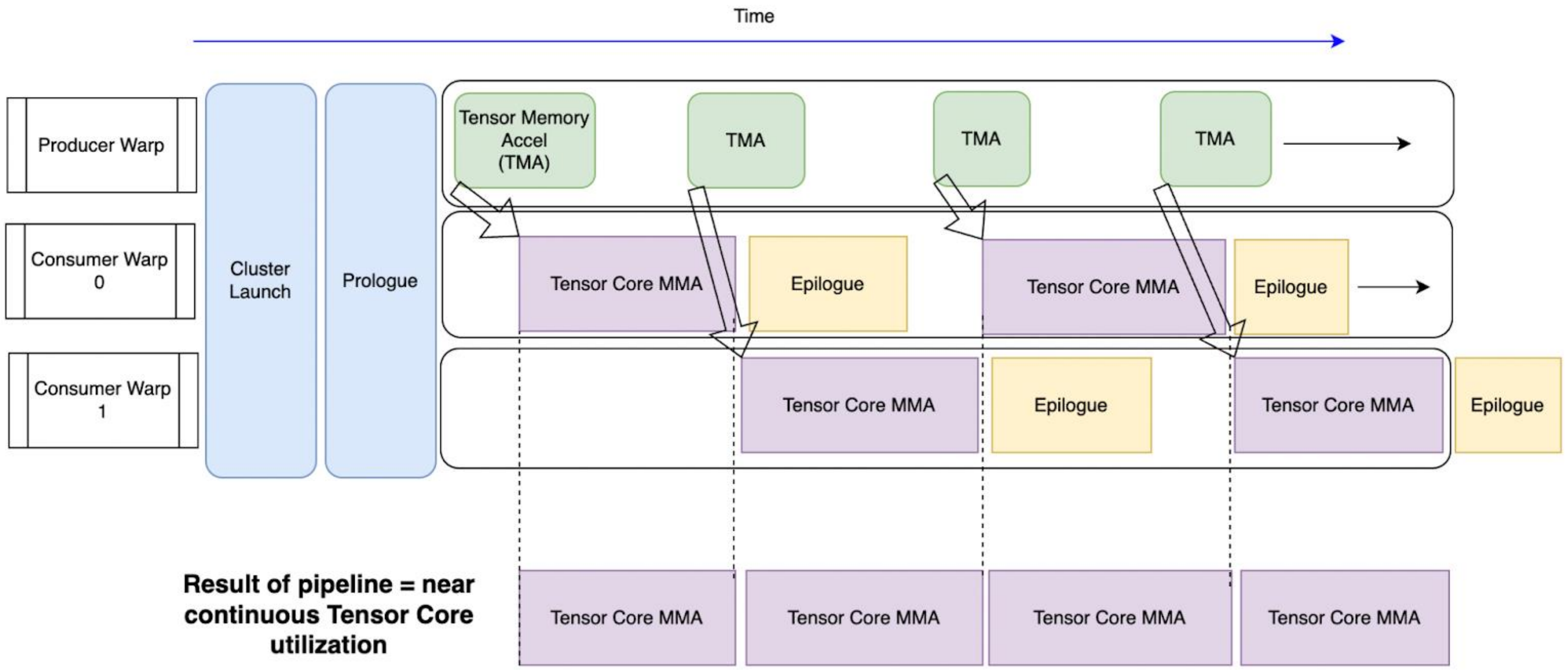
Gemm on Hopper



cuBLAS 13.0

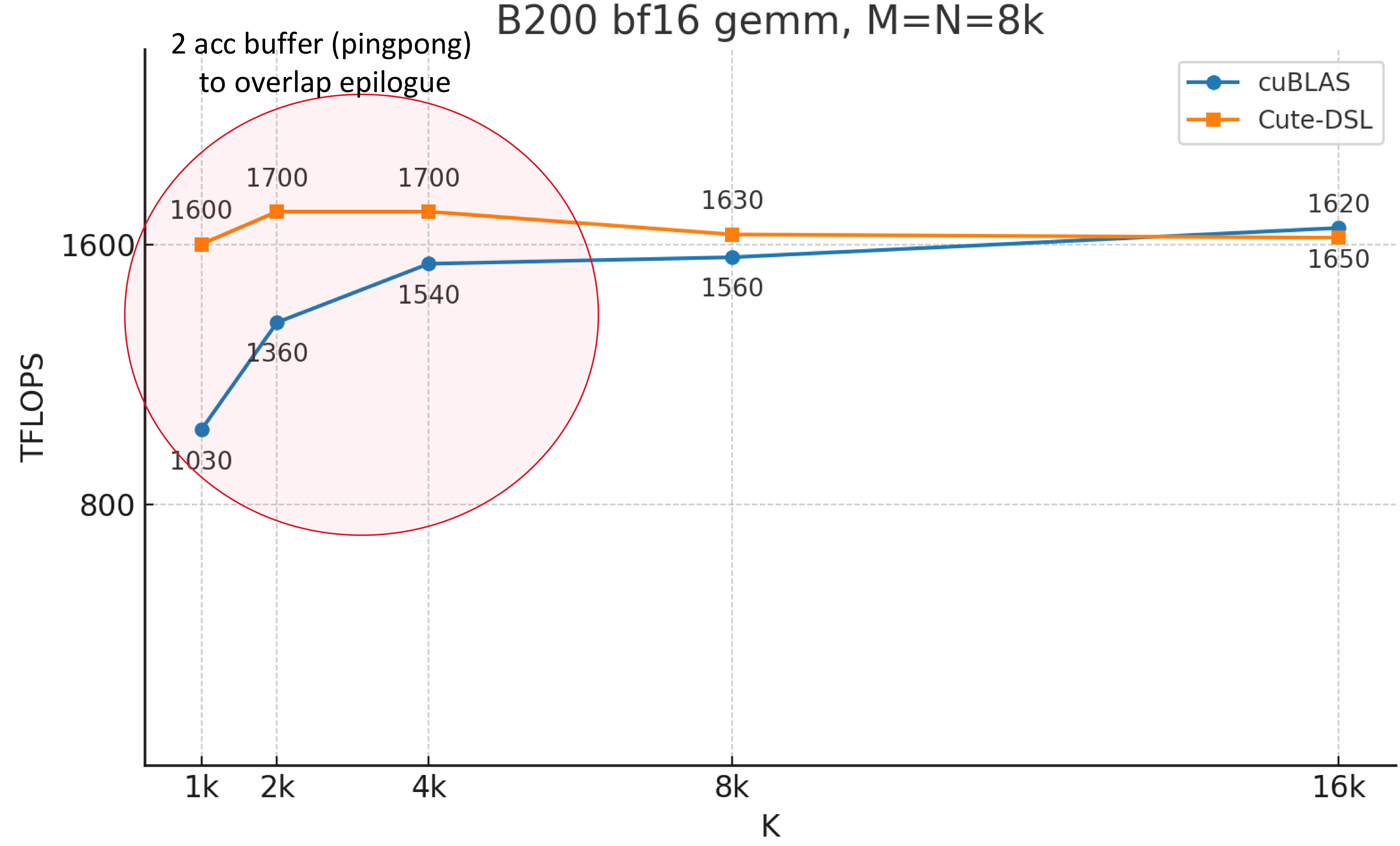
https://github.com/NVIDIA/cutlass/blob/main/examples/python/CuTeDSL/hopper/dense_gemm.py
https://github.com/Dao-AILab/quack/blob/main/quack/dense_gemm_sm90.py

Gemm on Hopper



https://pytorch.org/blog/cutlass-ping-pong-gemm-kernel/

Gemm on Blackwell

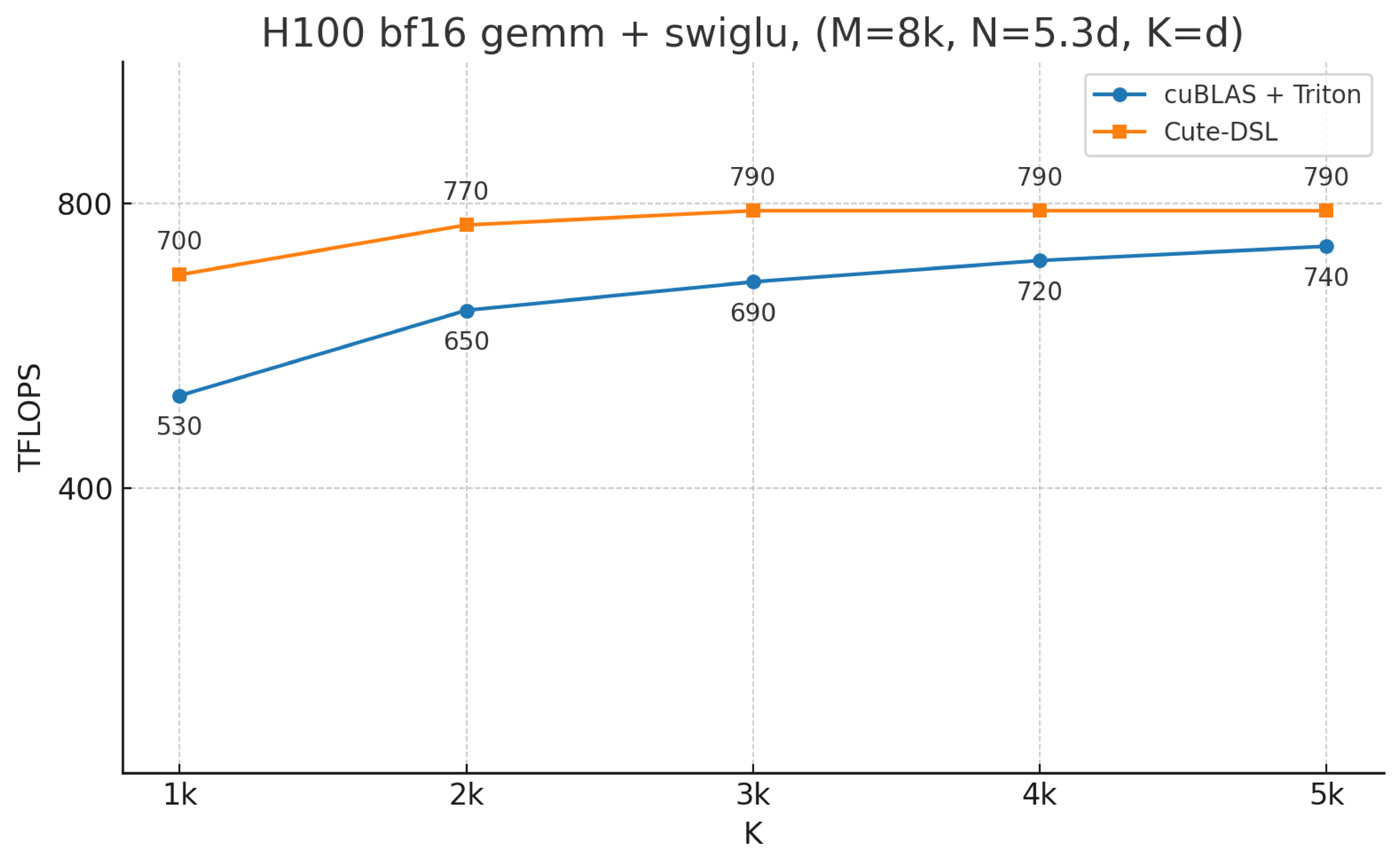


cuBLAS 13.0

https://github.com/NVIDIA/cutlass/blob/main/examples/python/CuTeDSL/blackwell/dense_gemm_persistent.py
https://github.com/Dao-AILab/quack/blob/main/quack/dense_gemm_sm100.py

GEMMs are getting easier (hardware make it so).
How about GEMM + X (epilogue, attention)?

Gemm + Swiglu on Hopper

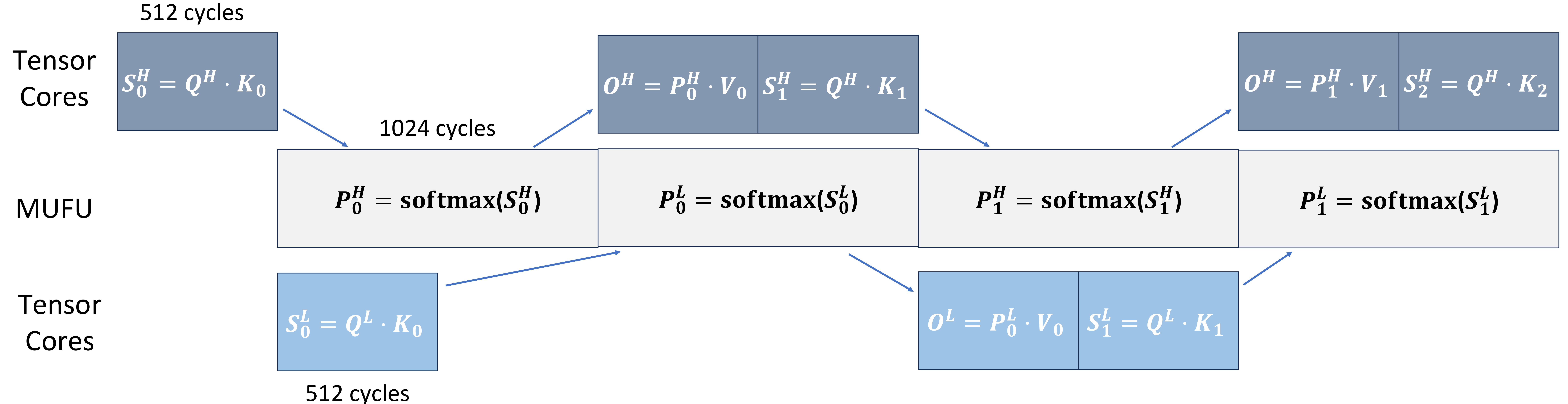


cuBLAS 13.0

7-15% speedup from epilogue fusion for the most commonly used gemm

FlashAttention4 on Blackwell: Pipelining

Example: headdim 128, block size 128 x 128, B200
BF16 UMMA: 8192 FLOPS/cycle -> **512 cycles per GEMM**
MUFU.EX2: 16 OPS/cycle -> **1024 cycles**

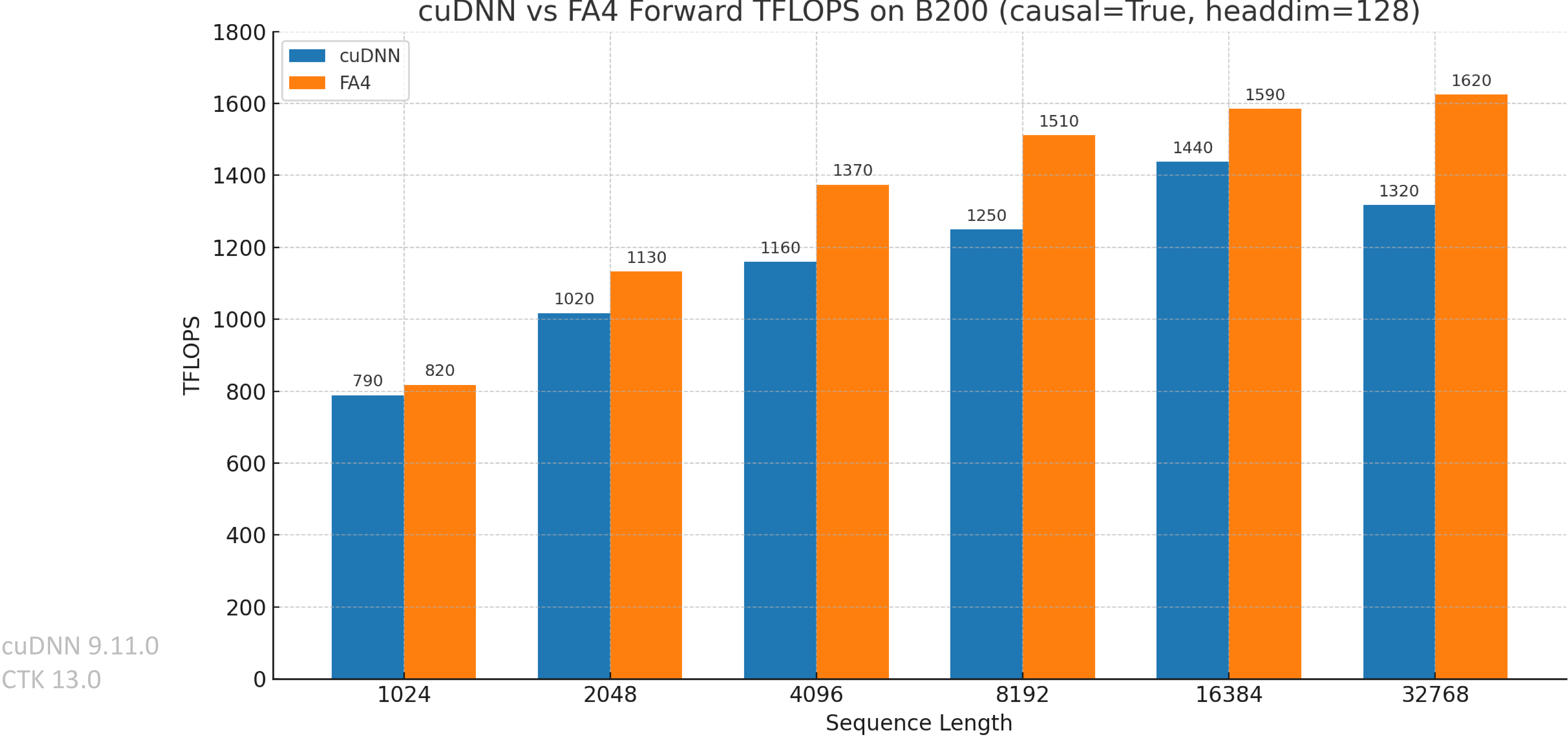


https://github.com/NVIDIA/cutlass/tree/main/examples/77_blackwell_fmha
https://github.com/Dao-AILab/flash-attention/blob/main/flash_attn/cute/flash_fwd_sm100.py

FlashAttention 4 on Blackwell: Key Algorithmic Changes

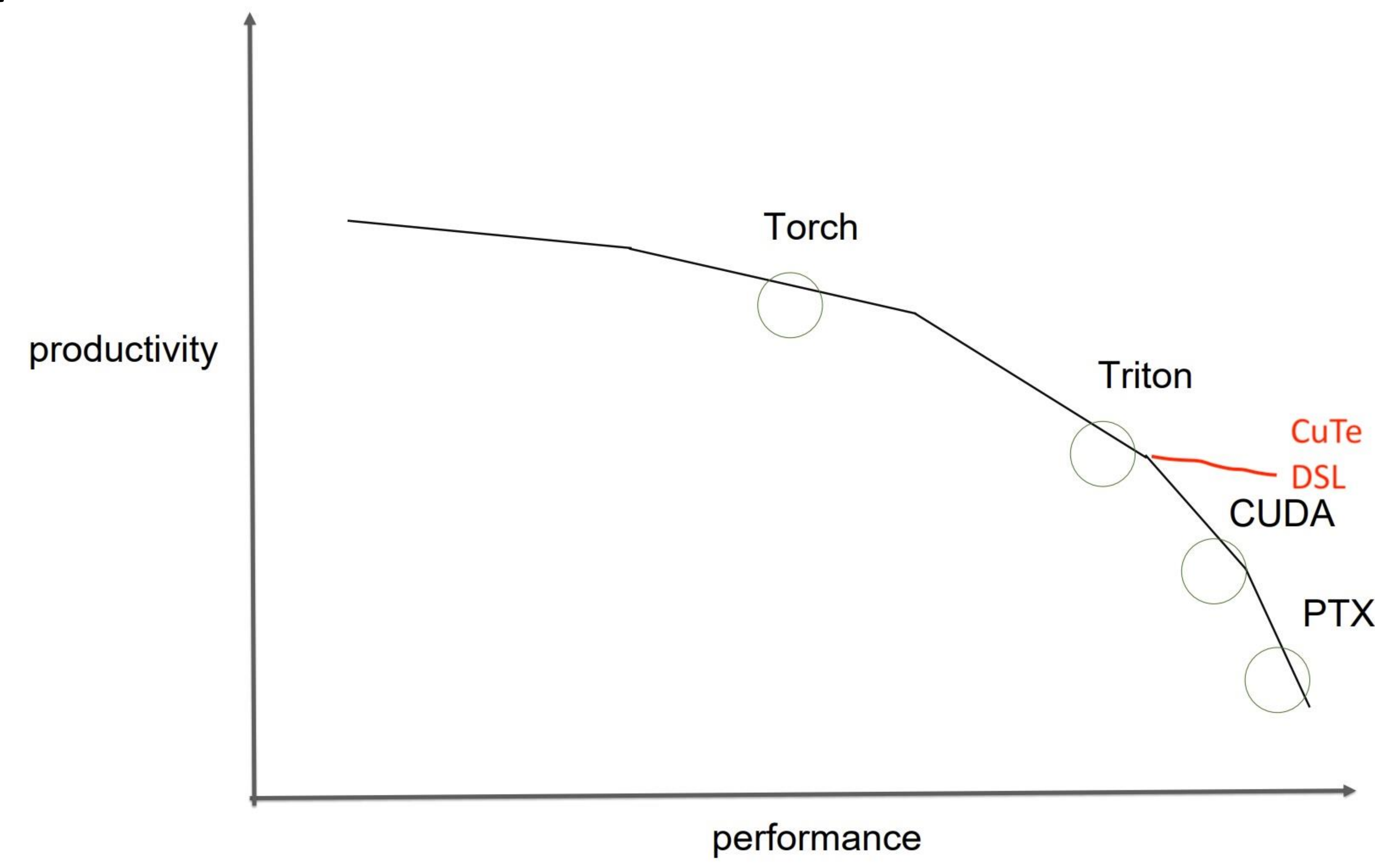
1. **Forward pass:** New online softmax algo to skip 90% of output rescaling
2. **Forward pass:** Software emulation of exponential (MUFU.EX2) to boost throughput
3. **Backward pass:** New design with 2-CTA MMA to reduce atomic add L2 traffic

FlashAttention 4 on Blackwell



https://github.com/Dao-AILab/flash-attention/blob/main/flash_attn/cute/flash_fwd_sm100.py

Trading off effort & perf



	Perf (mem-bound)	Perf (compute bound)	Onboarding time
Torch compile	~90%	~70-80%	hours-days
Triton	~90%	~80-90%	days-weeks
Cute-DSL	100%	100%	weeks-months

https://semianalysis.com/wp-content/uploads/2025/03/Blackwell-Programming-for-the-Masses-With-OpenAI-Triton-Phil-Tillet.pdf

Recommendation

1. **Try torch compile first:** 1 line. Generally great for elementwise / reduction fusion
2. **Triton:** Good for reduction / mem-bound kernels that require more tricks
3. **Cute-DSL:** Compute-bound kernels that need Tensor Cores

Outlines

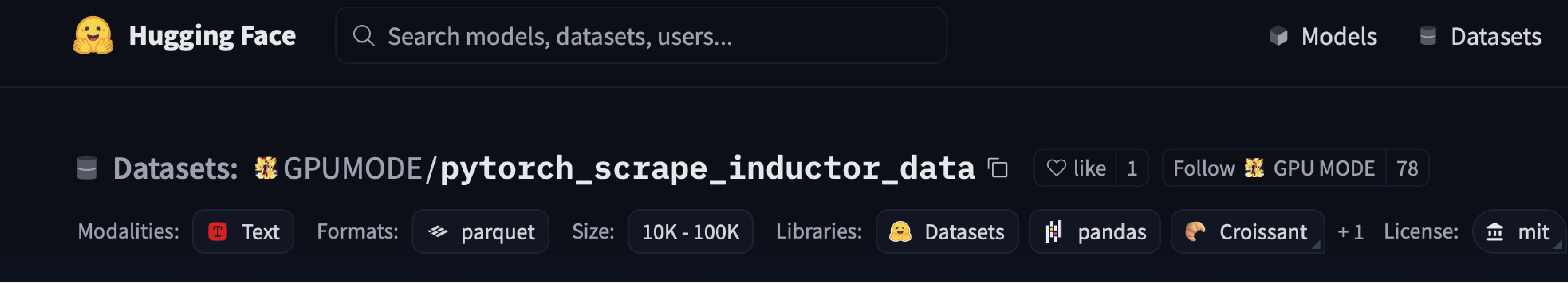
Attention Optimization

Prefill
Decode

AI for Kernels

Inference
Long context

Dataset: pairs of (PyTorch, Triton) programs



1. 18k pairs of (Pytorch, Triton) kernels

2. 2k of Triton kernels from permissive repos

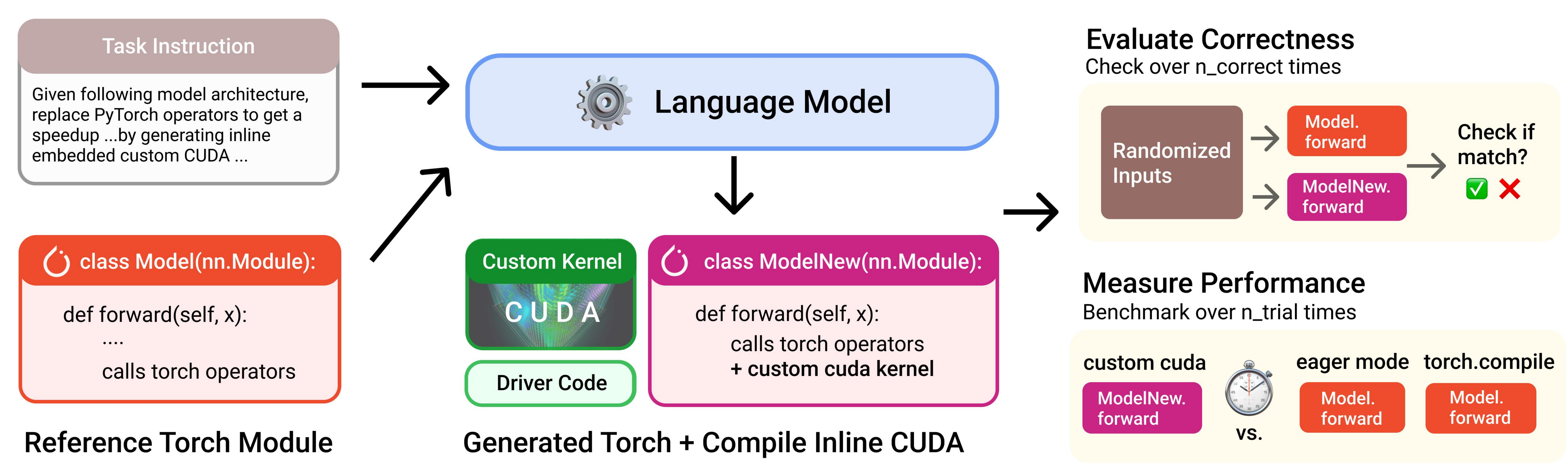
Main challenge: how to get **high quality** data

https://huggingface.co/datasets/GPUMODE/pytorch_scrape_inductor_data

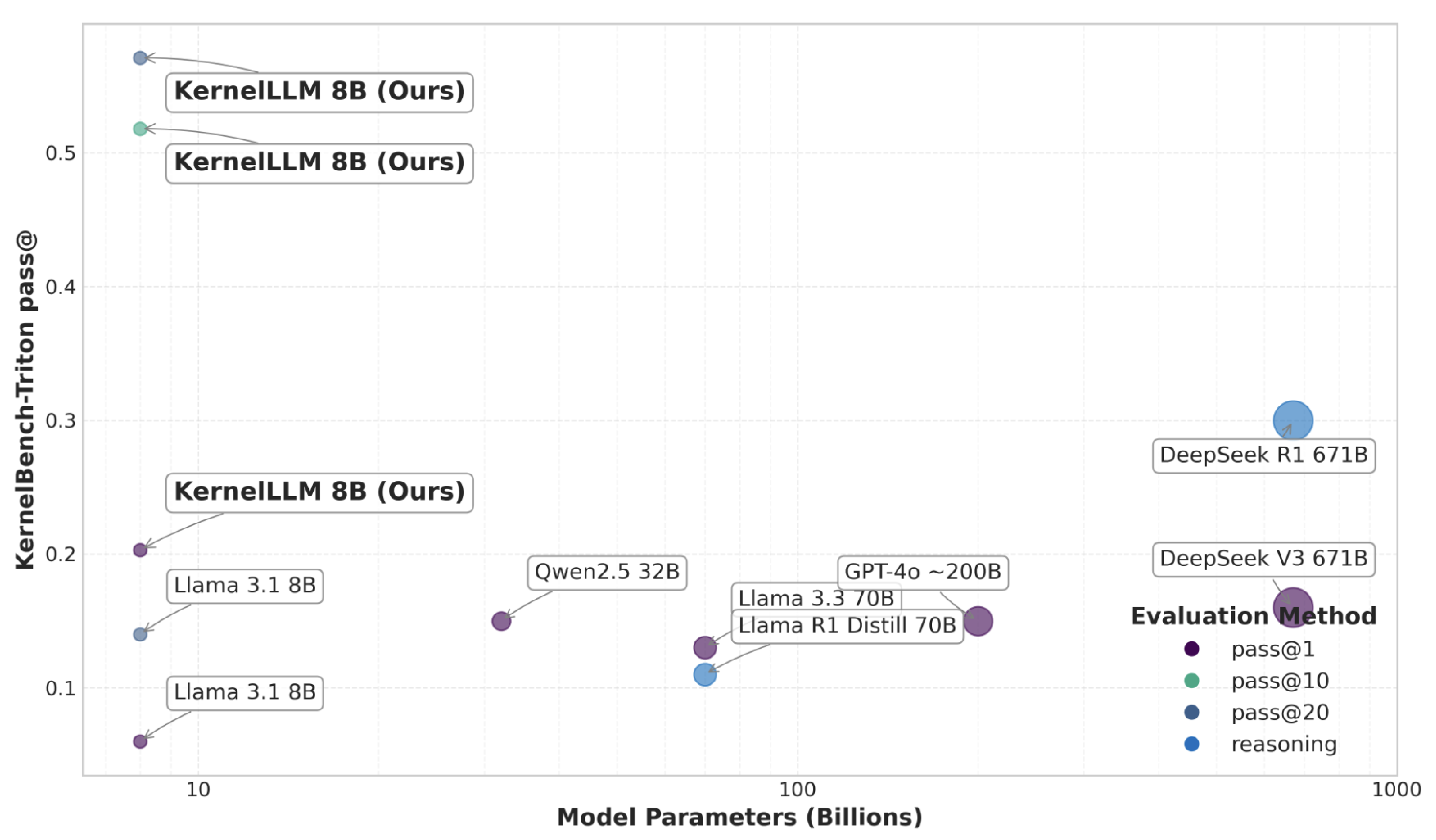
Eval: Kernel bench

KernelBench: Can LLMs Write Efficient GPU Kernels?

Anne Ouyang[1,*], Simon Guo[1,*], Simran Arora[1], Alex L. Zhang[2], William Hu[1], Christopher Ré[1], and Azalia Mirhoseini[1]



Algo: Supervised fine-tuning



https://huggingface.co/facebook/KernelLLM

Algo: Reinforcement Learning

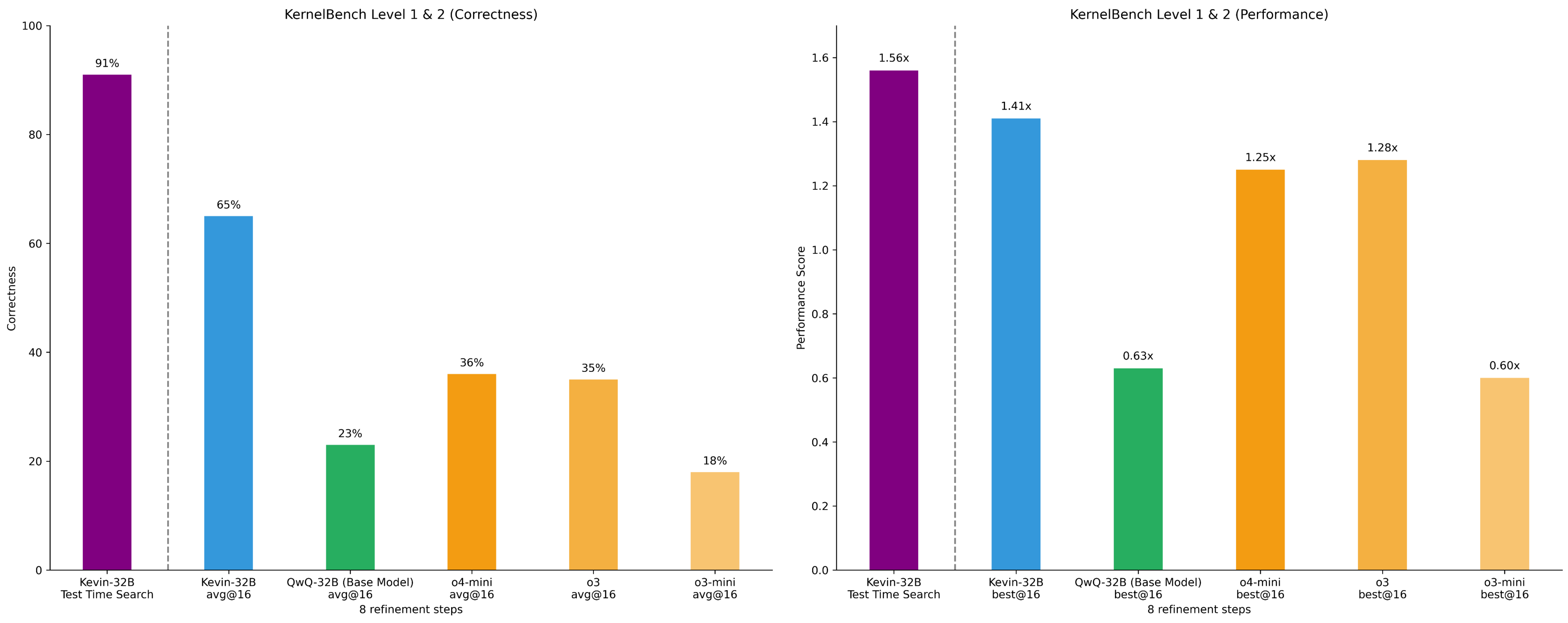
Kevin-32B: Multi-Turn RL for Writing CUDA Kernels

We explore reinforcement learning in a multi-turn setting, using intermediate feedback from the environment, and masking model thoughts to avoid exploding context over multiple turns.

2025-05-06 CARLO BARONIO, PIETRO MARSELLA, BEN PAN, SILAS ALBERTI

Carlo Baronio*, Pietro Marsella*, Ben Pan*, Silas Alberti

Stanford University, Cognition AI



State of automatic LLM-generated kernels

1. **Simple elementwise/reduction kernels:** ✅ correct & ✅ fast
2. **Medium-difficulty mem-bound kernels:** ❌ incorrect
3. **Advanced mem-bound kernels:** ❌ incorrect (e.g. cp.async, TMA, pipelining)
4. **Compute-bound kernels:** ❌ incorrect (tensor cores)

LLMs are (surprisingly) good at algorithmic reasoning

This is a softmax kernel in cute-dsl. How can we improve it to reduce number of synchronization?

```
@cute.kernel
def kernel(
    self,
    mX: cute.Tensor,
    mO: cute.Tensor,
    tv_layout: cute.Layout,
    tiler_mn: cute.Shape,
):
```

1) Make each row tile warp-local (avoid CTA-wide reductions)

If you can arrange `tv_layout` so that **one warp owns one row-stripe** (or a power-of-two chunk of a row), both reductions become **pure warp shuffles → no SMEM, no CTA sync**, no `mbarrier` within the CTA.

- Map `threads_per_row = 32` (or 16/8) and keep `N` tiled so each warp handles a contiguous K-chunk per row.
- Replace `utils.row_reduce(...)` with a warp-local reduce:

https://chatgpt.com/share/68ab90b9-fd28-8012-aa20-5990499898a7

2) Use multistage cp.async ping-pong to hide the only mandatory wait

Right now you do:

```python
cute.arch.cp_async_commit_group()
cute.arch.cp_async_wait_group(0)
```

...once, before compute. Switch to a **looped, double-buffered** pipeline over `N` with two SMEM tiles (`sX[0]` , `sX[1]`) and `kStages = 2` :

3) Drop the OOB fill sync with predicated zfill

You call `utils.fill_oob(...)` after the wait. If your CUTE/CUTLASS build exposes **zero-fill cp.async** (or predicated copies that write zero/–inf on miss), switch to that and remove the extra pass + implicit barrier.

4) Collapse cluster syncs (or disable cluster mode for softmax)

If `cluster_n > 1` you pass `mbar_ptr` into `row_reduce`. For softmax, inter-CTA reduction is rarely worth it unless a single **logical row** is explicitly partitioned across CTAs.

5) One-pass streaming softmax across tiles (fewer global phases)

Classic softmax is 2 passes (max, then sum(exp(x–max))). To avoid an extra global "phase" barrier, compute **(m, s)** online while you stream tiles (per warp):

- Maintain `(m, s)` per row (in registers).
- For each fragment `x` :

```ini
m_new = max(m, max(x))
s     = s * exp2((m - m_new) * log2e) + sum(exp2((x - m_new) * log2e))
m     = m_new
```

New abstractions needed

LLM …

is good at:

Algorithmic reasoning

is poor at:

Low-level implementation details

Compiler …

is good at:

Low-level implementation details

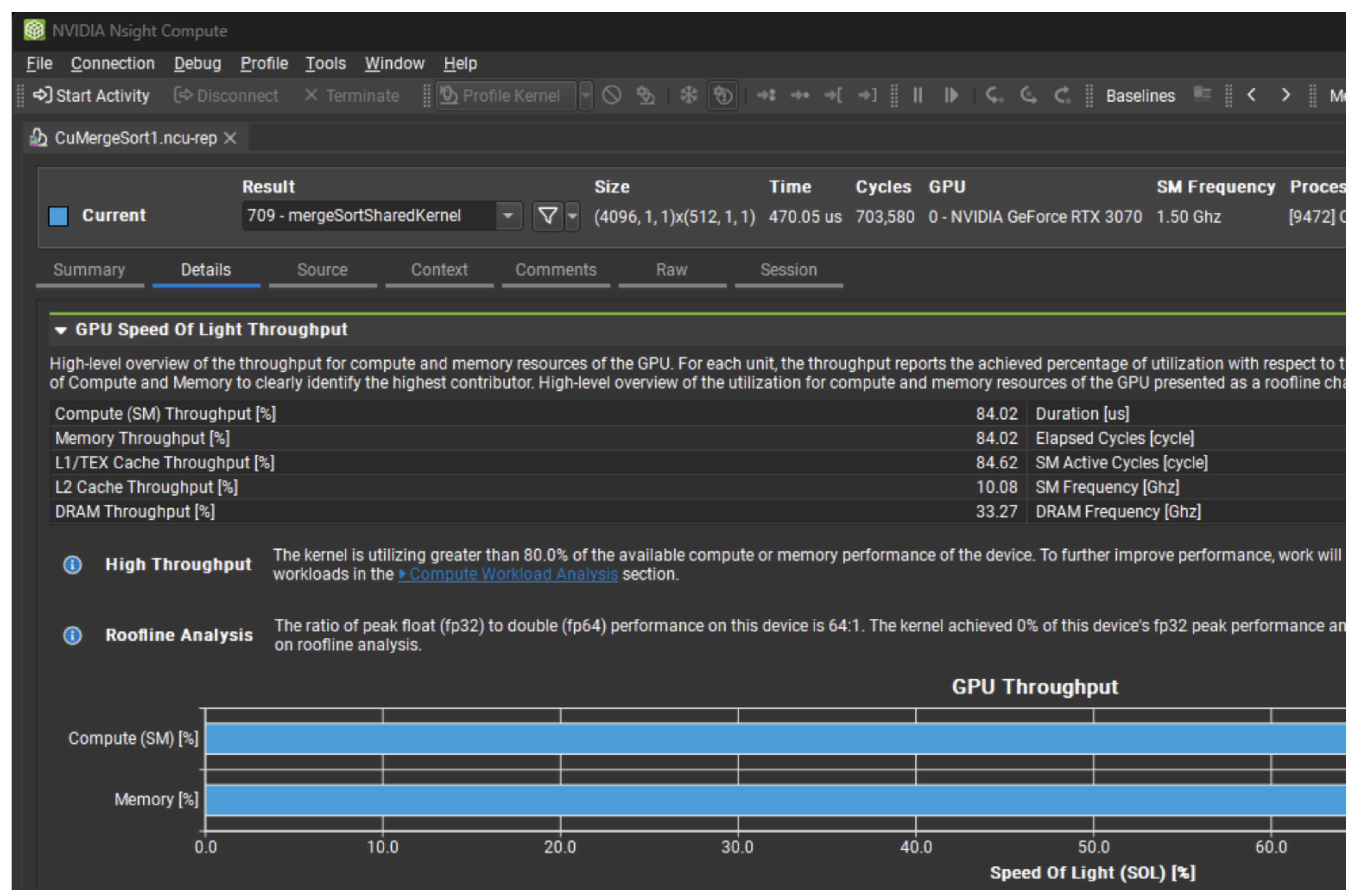
is poor at:

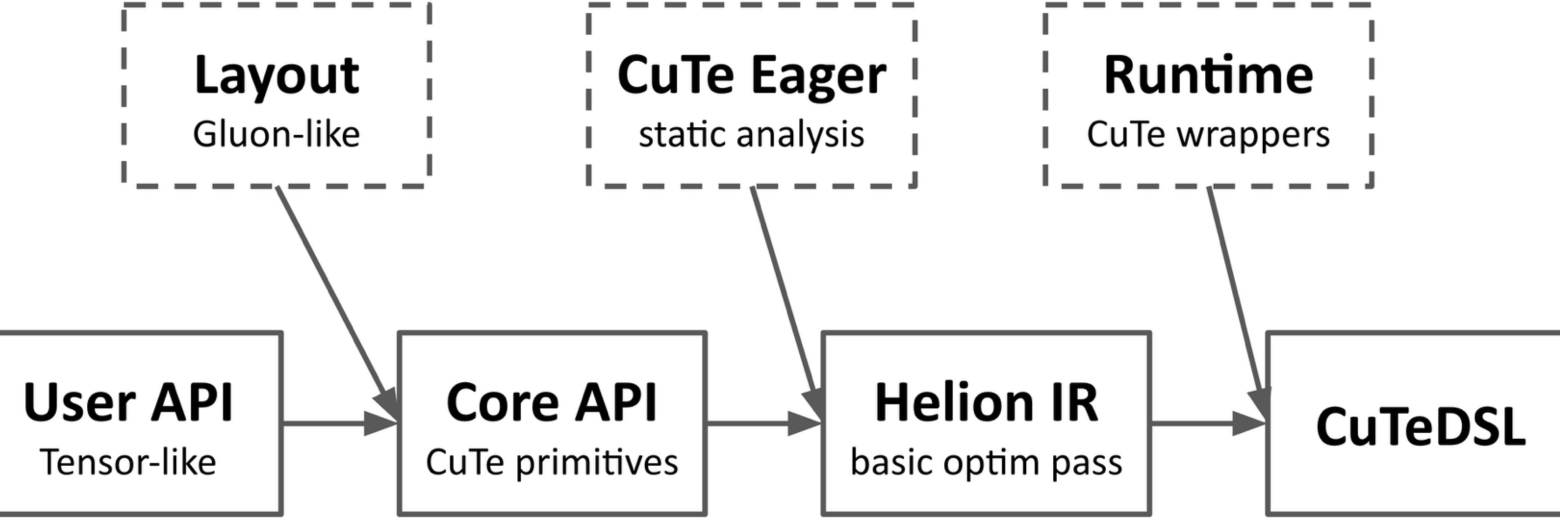
Algorithmic reasoning

Open problem: what's a good abstraction for LLMs to write kernels?

Ingredients we need to train LLMs to write kernels

```
#
# Mma mainloop
#
for k_block in range(k_block_cnt):
    if is_leader_cta:
        # Conditionally wait for AB buffer full
        ab_pipeline.consumer_wait(
            ab_consumer_state, peek_ab_full_status
        )

        # tCtAcc += tCrA * tCrB
        num_kphases = cute.size(tCrA, mode=[2])
        for kphase_idx in cutlass.range(num_kphases, unroll_full=True):
            kphase_coord = (
                None,
                None,
                kphase_idx,
                ab_consumer_state.index,
            )

            cute.gemm(
                tiled_mma,
                tCtAcc,
                tCrA[kphase_coord],
                tCrB[kphase_coord],
                tCtAcc,
            )
            # Enable accumulate on tCtAcc after first kphase
            tiled_mma.set(tcgen05.Field.ACCUMULATE, True)

        # Async arrive AB buffer empty
        ab_pipeline.consumer_release(ab_consumer_state)
```





Expert-level data

Tool use

Abstractions