Building LLM Reasoners

Lecture 4: Making LLMs Fast II

Greg Durrett

Slide credit: Tatsu Hashimoto &
Percy Liang, Stanford CS 336

Administrative details

Operator Fusion

Triton: Weighted Sum

Flash Attention

Flash Attention: Backward Pass

Parallelism

Administrative details

Operator Fusion

Triton: Weighted Sum

Flash Attention

Flash Attention: Backward Pass

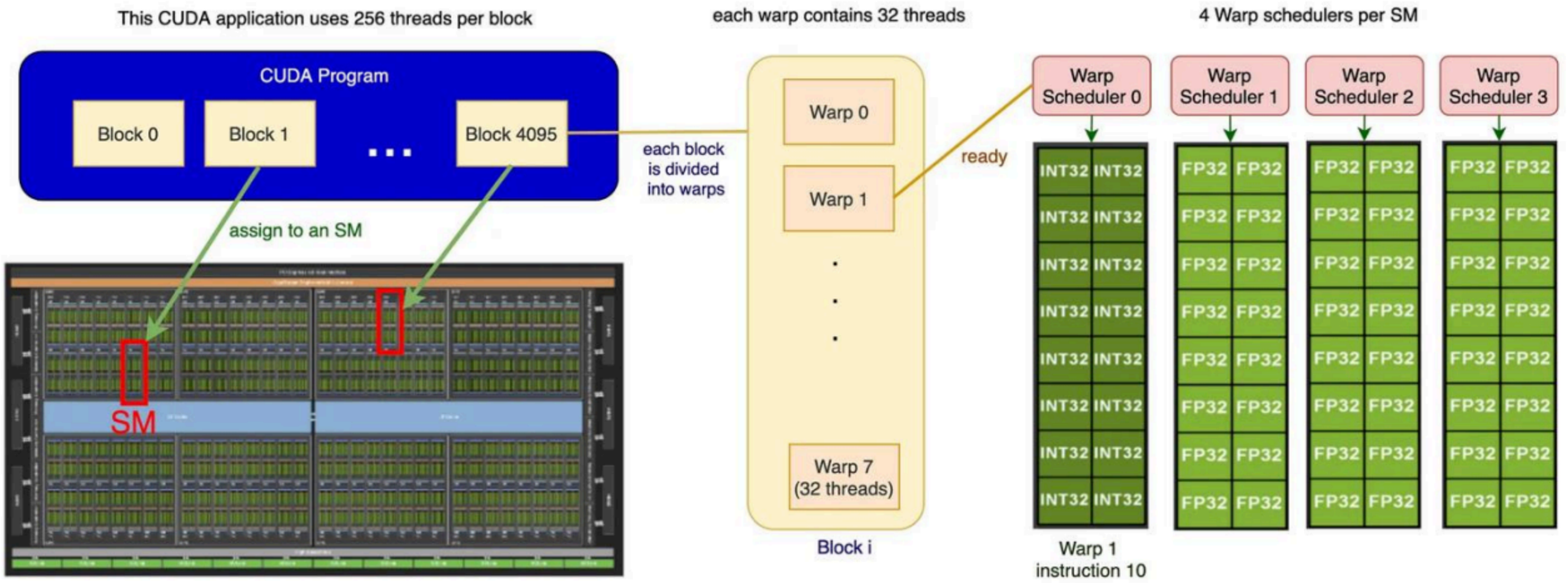
Parallelism

Administrative details

Administrivia

- Assignment 1 due today
- Homework quiz
- Assignment 2 out Monday, due 3 weeks from today
- If you're interested in a paid summer RA position, please email me a CV

Execution Model



There are 3 important players in the execution model

Threads: Threads 'do the work' in parallel – all threads execute the same instructions but with different inputs (SIMT).

Blocks: Blocks are groups of threads. Each block runs on a SM w/ its own shared memory.

Warp: Threads always execute in a 'warp' of 32 consecutively numbered threads each.

Five Tricks for GPUs

Trick 1: Low/mixed precision

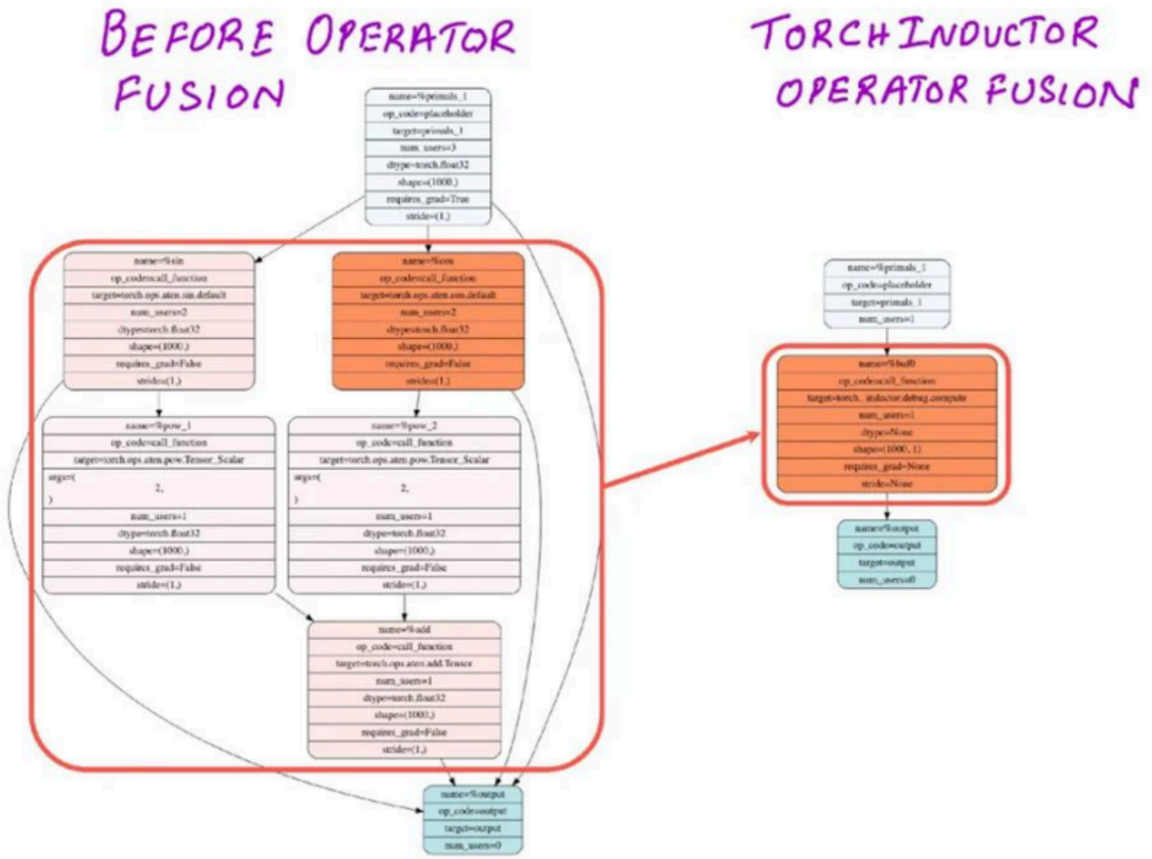
Trick 2: Operator fusion

Trick 3: Recomputation

Trick 4: Memory coalescence

Trick 5: Tiling

Trick 2: Operator Fusion: After



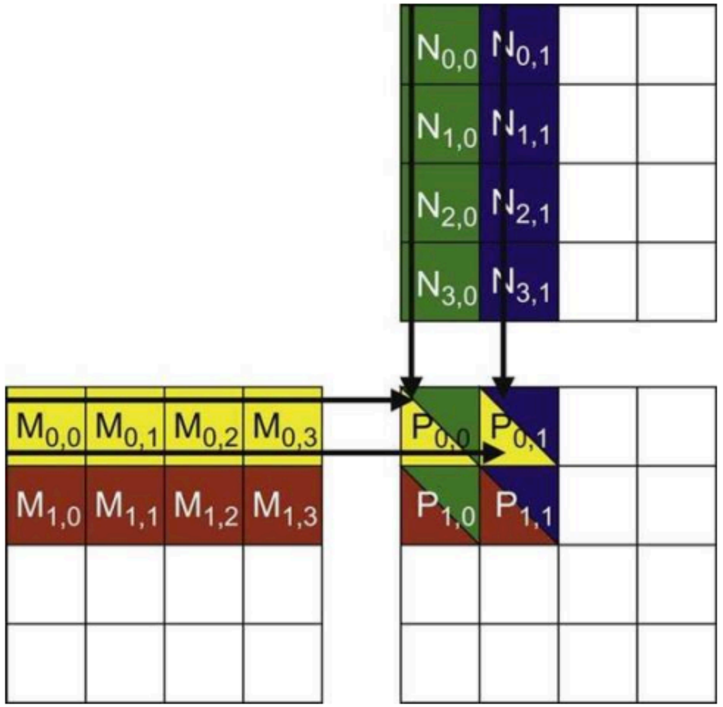
All 5 pointwise operations can be fused into a single CUDA kernel call.

‘Easy’ fusions like this can be done automatically by compilers (torch.compile)

Trick 5: Tiling

Tiling is the idea of grouping and ordering threads to minimize global memory access.

Let's go back to matrix multiplication..



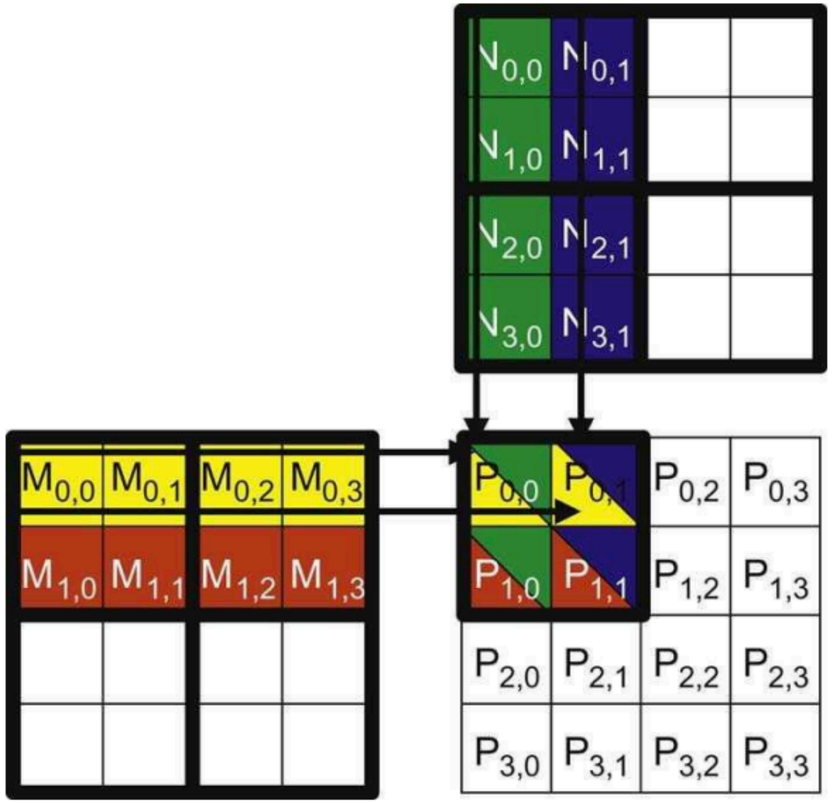
Access order

$thread_{0,0}$	$M_{0,0} * N_{0,0}$	$M_{0,1} * N_{1,0}$	$M_{0,2} * N_{2,0}$	$M_{0,3} * N_{3,0}$
$thread_{0,1}$	$M_{0,0} * N_{0,1}$	$M_{0,1} * N_{1,1}$	$M_{0,2} * N_{2,1}$	$M_{0,3} * N_{3,1}$
$thread_{1,0}$	$M_{1,0} * N_{0,0}$	$M_{1,1} * N_{1,0}$	$M_{1,2} * N_{2,0}$	$M_{1,3} * N_{3,0}$
$thread_{1,1}$	$M_{1,0} * N_{0,1}$	$M_{1,1} * N_{1,1}$	$M_{1,2} * N_{2,1}$	$M_{1,3} * N_{3,1}$

Note that memory access is not coalesced, and repeated (M0,0 and N1,0)

Trick 5: Tiling

Cut up the matrix into smaller ‘tiles’, and load this into shared memory



Compute the matrix multiply in ‘phases’

1. Load $M_{0,0}$ and $N_{0,0}$ tiles into SHM
2. Compute partial sums for P

   (Done with one tile)
3. Load the $M_{0,0}$ and $N_{2,0}$ tile into SHM
4. …

Advantages: repeated reads now access shared, not global memory and memory access can be coalesced

Administrative details

Operator Fusion

Triton: Weighted Sum

Flash Attention

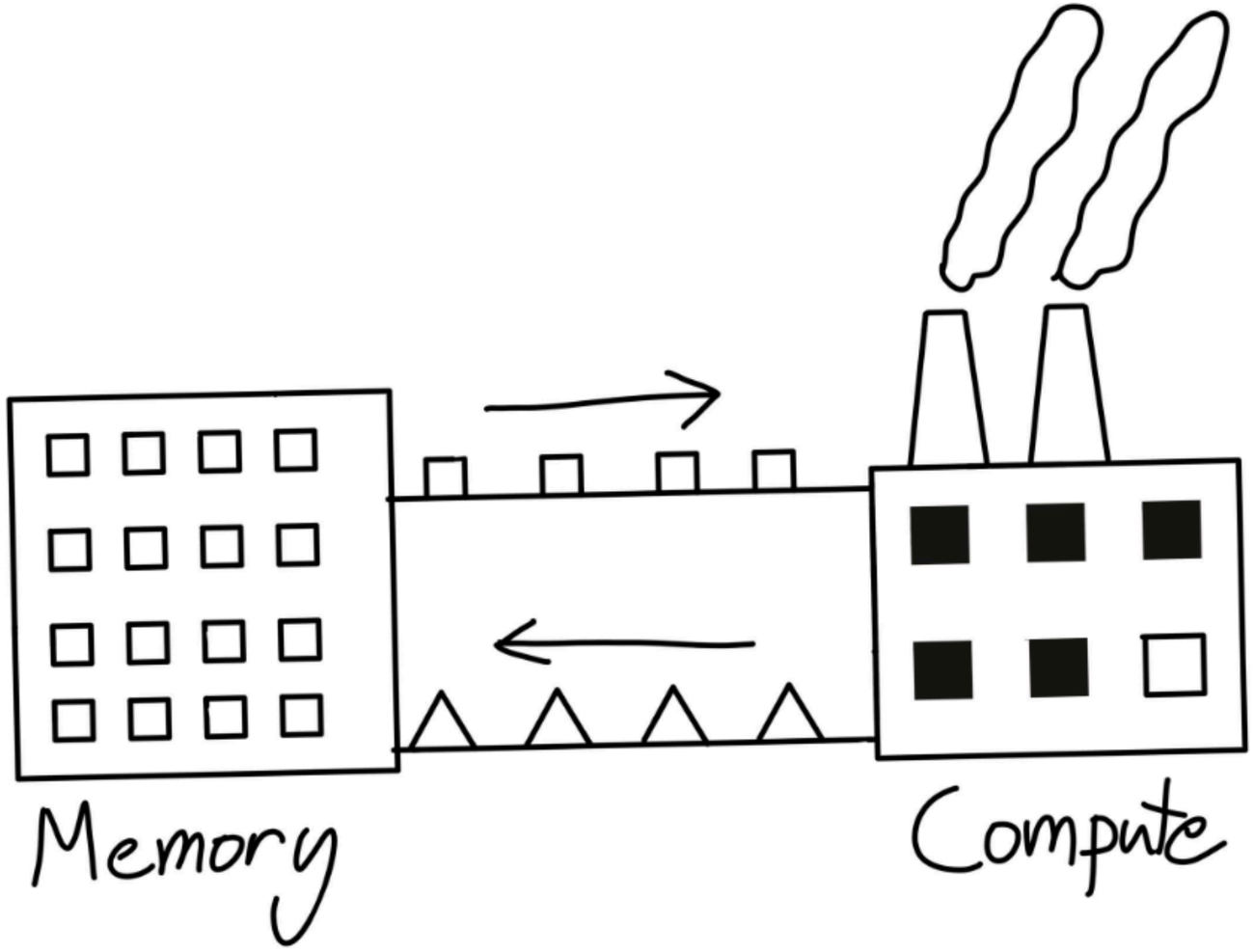
Flash Attention: Backward Pass

Parallelism

Operator Fusion

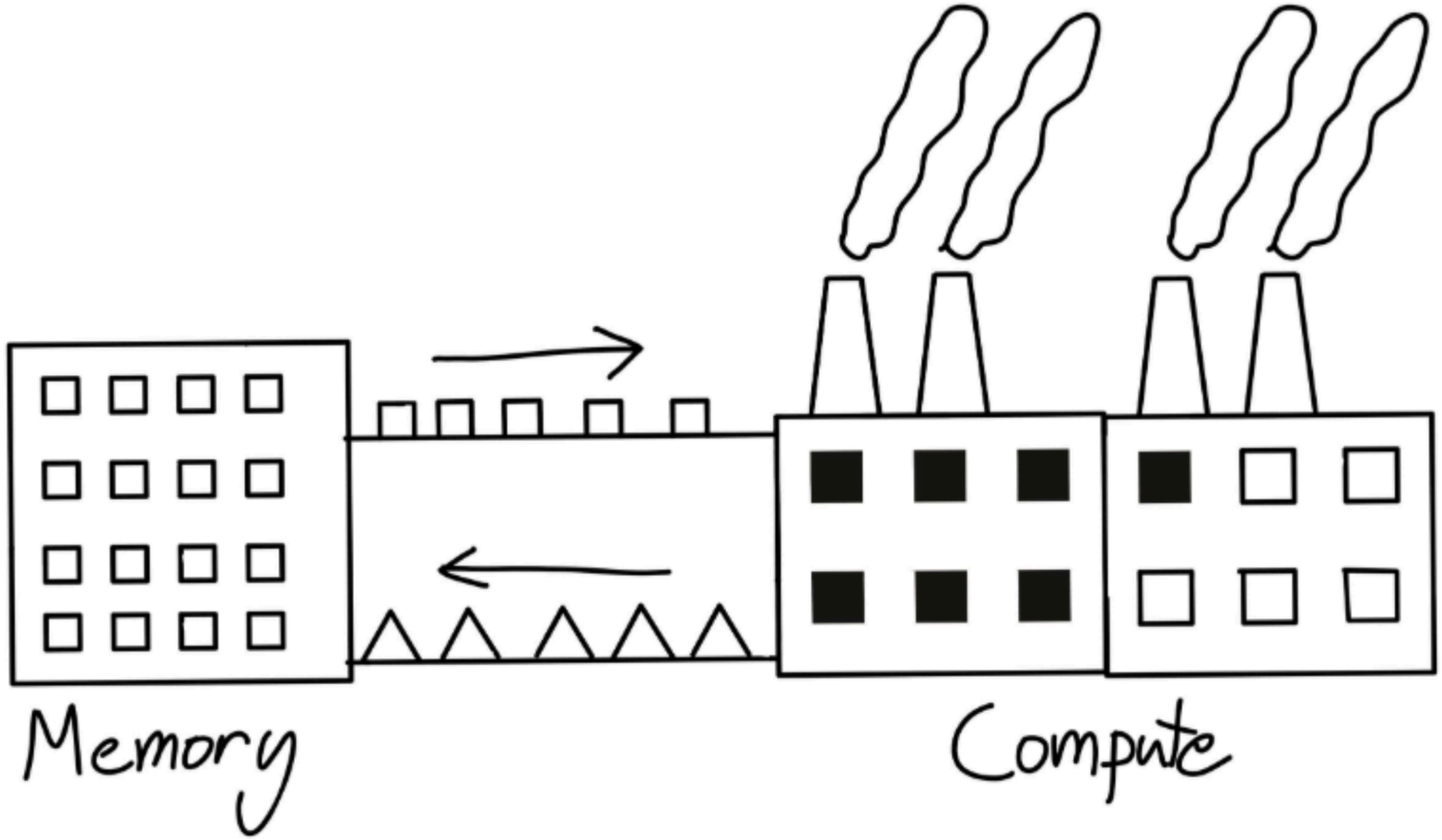
Code available with Assignment 2
Code credit: Stanford CS336

Operator Fusion



https://horace.io/brrr_intro.html

Operator Fusion



- Reduce bandwidth —> better compute utilization

https://horace.io/brrr_intro.html

Operator Fusion: GeLU

```python
def pytorch_gelu(x: torch.Tensor) -> torch.Tensor:  4 usages  Greg Durrett
    """PyTorch's fused GeLU implementation (tanh approximation)."""
    return torch.nn.functional.gelu(x, approximate="tanh")


def manual_gelu(x: torch.Tensor) -> torch.Tensor:  4 usages  Greg Durrett
    """Manual GeLU implementation (NOT fused - each operation is a separate kernel)."""
    # Each of these operations triggers a separate CUDA kernel:
    # 1. x * x * x (2 multiplications)
    # 2. 0.044715 * x^3
    # 3. x + (0.044715 * x^3)
    # 4. 0.79788456 * (...)
    # 5. torch.tanh(...)
    # 6. 1 + tanh(...)
    # 7. x * (1 + tanh(...))
    # 8. 0.5 * (...)
    # Each operation reads from DRAM and writes back to DRAM!
    return 0.5 * x * (1 + torch.tanh(0.79788456 * (x + 0.044715 * x * x * x)))
```

Operator Fusion: GeLU

Unfused (manual):

Name	Self CPU %	Self CPU	CPU total %	CPU total	CPU time avg	Self CUDA
aten::mul	9.37%	1.758ms	9.75%	1.829ms	304.808us	11.686ms
at::native::BinaryFunctor<f...	0.00%	0.000us	0.00%	0.000us	0.000us	7.039ms
at::native::AUnaryFunctor<f...	0.00%	0.000us	0.00%	0.000us	0.000us	4.647ms
aten::add	0.16%	29.589us	0.25%	46.151us	23.076us	3.896ms
at::native::CUDAFunctor_add...	0.00%	0.000us	0.00%	0.000us	0.000us	2.343ms
at::native::CUDAFunctorOnSe...	0.00%	0.000us	0.00%	0.000us	0.000us	1.553ms
aten::tanh	0.06%	11.307us	0.10%	18.908us	18.908us	1.545ms
at::native::tanh_kernel_cud...	0.00%	0.000us	0.00%	0.000us	0.000us	1.545ms
cudaLaunchKernel	0.50%	94.646us	0.50%	94.646us	10.516us	0.000us
cudaDeviceSynchronize	89.91%	16.872ms	89.91%	16.872ms	8.436ms	0.000us

Fused

Name	Self CPU %	Self CPU	CPU total %	CPU total	CPU time avg	Self CUDA	Self CUDA %
aten::gelu	52.77%	1.707ms	53.44%	1.729ms	1.729ms	1.545ms	100.00%
at::native::GeluCUDAKernelI...	0.00%	0.000us	0.00%	0.000us	0.000us	1.545ms	100.00%
cudaLaunchKernel	0.67%	21.711us	0.67%	21.711us	21.711us	0.000us	0.00%
cudaDeviceSynchronize	46.56%	1.507ms	46.56%	1.507ms	753.361us	0.000us	0.00%

(Slow) Fused GeLU Kernel

```
__global__ void gelu_kernel(const float* __restrict__ x,
                            float* __restrict__ y,
                            int num_elements) {
    // Compute global thread index
    // blockIdx.x = which block this thread is in
    // blockDim.x = number of threads per block
    // threadIdx.x = index of this thread within the block
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    // Bounds check (some threads may be out of range)
    if (i < num_elements) {
        // Read input value
        float xi = x[i];

        // Compute GeLU using tanh approximation:
        // gelu(x) = 0.5 * x * (1 + tanh(sqrt(2/pi) * (x + 0.044715 * x^3)))
        // sqrt(2/pi) ≈ 0.79788456
        float a = 0.79788456f * (xi + 0.044715f * xi * xi * xi);
        float tanh_a = tanhf(a);
        float yi = 0.5f * xi * (1.0f + tanh_a);
```

Triton

- “Between” Python and CUDA
- Write low-level code, but memory management is more automatic
- Write in Python, but compiled into GPU code
- Launched via a “launch grid”, which is usually defined in your Pytorch function

Triton

```
def triton_gelu(x: torch.Tensor):
    assert x.is_cuda
    assert x.is_contiguous()

    # Allocate output tensor
    y = torch.empty_like(x)

    # Determine grid (elements divided into blocks)
    num_elements = x.numel()
    block_size = 1024  # Number of threads
    num_blocks = triton.cdiv(num_elements, block_size)

    triton_gelu_kernel[(num_blocks,)](x, y, num_elements, BLOCK_SIZE=block_size)

    return y
```

- 1D launch grid

Triton

```
@triton.jit
def triton_gelu_kernel(x_ptr, y_ptr, num_elements, BLOCK_SIZE: tl.constexpr):
    # Input is at `x_ptr` and output is at `y_ptr`
    #     |        Block 0            |          Block 1          |      ...      |
    #                              BLOCK_SIZE                                  num_elements

    pid = tl.program_id(axis=0)
    block_start = pid * BLOCK_SIZE

    # Indices where this thread block should operate
    offsets = block_start + tl.arange(0, BLOCK_SIZE)

    # Handle boundary
    mask = offsets < num_elements

    # Read
    x = tl.load(x_ptr + offsets, mask=mask)
```

- 1D launch grid, figure out where this Triton program is
- Identify where this operates and mask anything it shouldn't read/write to
- Then compute GeLU and store (not shown)

Administrative details

Operator Fusion

Triton: Weighted Sum

Flash Attention

Flash Attention: Backward Pass

Parallelism

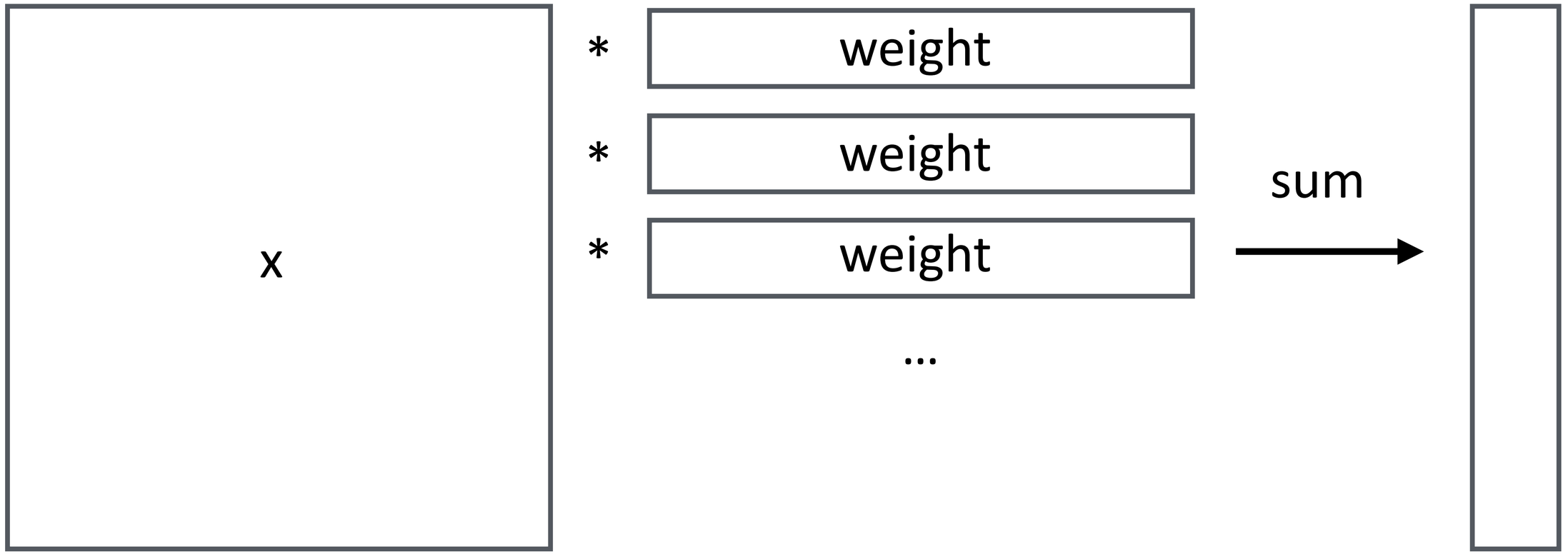
Triton: Weighted Sum

Code available with Assignment 2
Code credit: Stanford CS336

Triton

Example: weighted sum

```
(weight * x).sum(axis=-1)
```



Triton: Weighted Sum

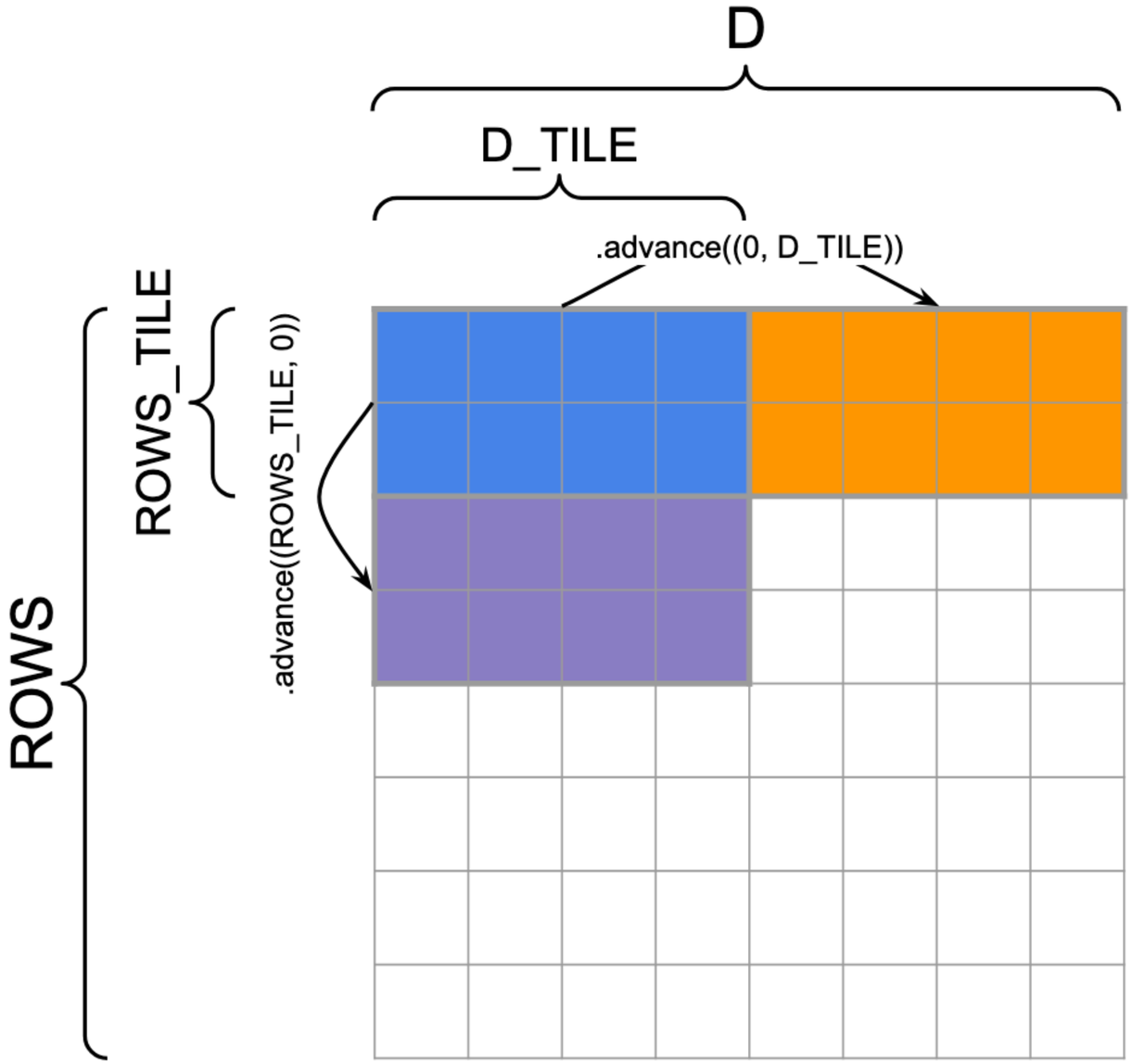
Example: weighted sum

```
(weight * x).sum(axis=-1)
```

We are going to process the matrix in tiles

Warmup: assume this is row-major. What is the

- row stride?
- dim (column) stride?



Block Pointers

```
@triton.jit  1 usage
def weighted_sum_fwd(
    x_ptr, weight_ptr,  # Input pointers
    output_ptr,  # Output pointer
    x_stride_row, x_stride_dim,  # Strides tell us how to move one element in each axis of a tensor
    weight_stride_dim,  # Likely 1
    output_stride_row,  # Likely 1
    ROWS, D,
    ROWS_TILE_SIZE: tl.constexpr, D_TILE_SIZE: tl.constexpr,  # Tile shapes must be known at compile time
):
    # Each instance will compute the weighted sum of a tile of rows of x.
    # `tl.program_id` gives us a way to check which thread block we're running in
    row_tile_idx = tl.program_id(0)
```

```
# Each instance will compute the weighted sum of a tile of rows of x.
# `tl.program_id` gives us a way to check which thread block we're running in
row_tile_idx = tl.program_id(0)

# Block pointers give us a way to select from an ND region of memory
#  and move our selection around.
# The block pointer must know:
# - The pointer to the first element of the tensor
# - The overall shape of the tensor to handle out-of-bounds access
# - The strides of each dimension to use the memory layout properly
# - The ND coordinates of the starting block, i.e., "offsets"
# - The block shape to use load/store at a time
# - The order of the dimensions in memory from major to minor
#    axes (= np.argsort(strides)) for optimizations, especially useful on H100

x_block_ptr = tl.make_block_ptr(
    x_ptr,
    shape=(ROWS, D,),
    strides=(x_stride_row, x_stride_dim),
    offsets=(row_tile_idx * ROWS_TILE_SIZE, 0),
    block_shape=(ROWS_TILE_SIZE, D_TILE_SIZE),
    order=(1, 0),
)
```

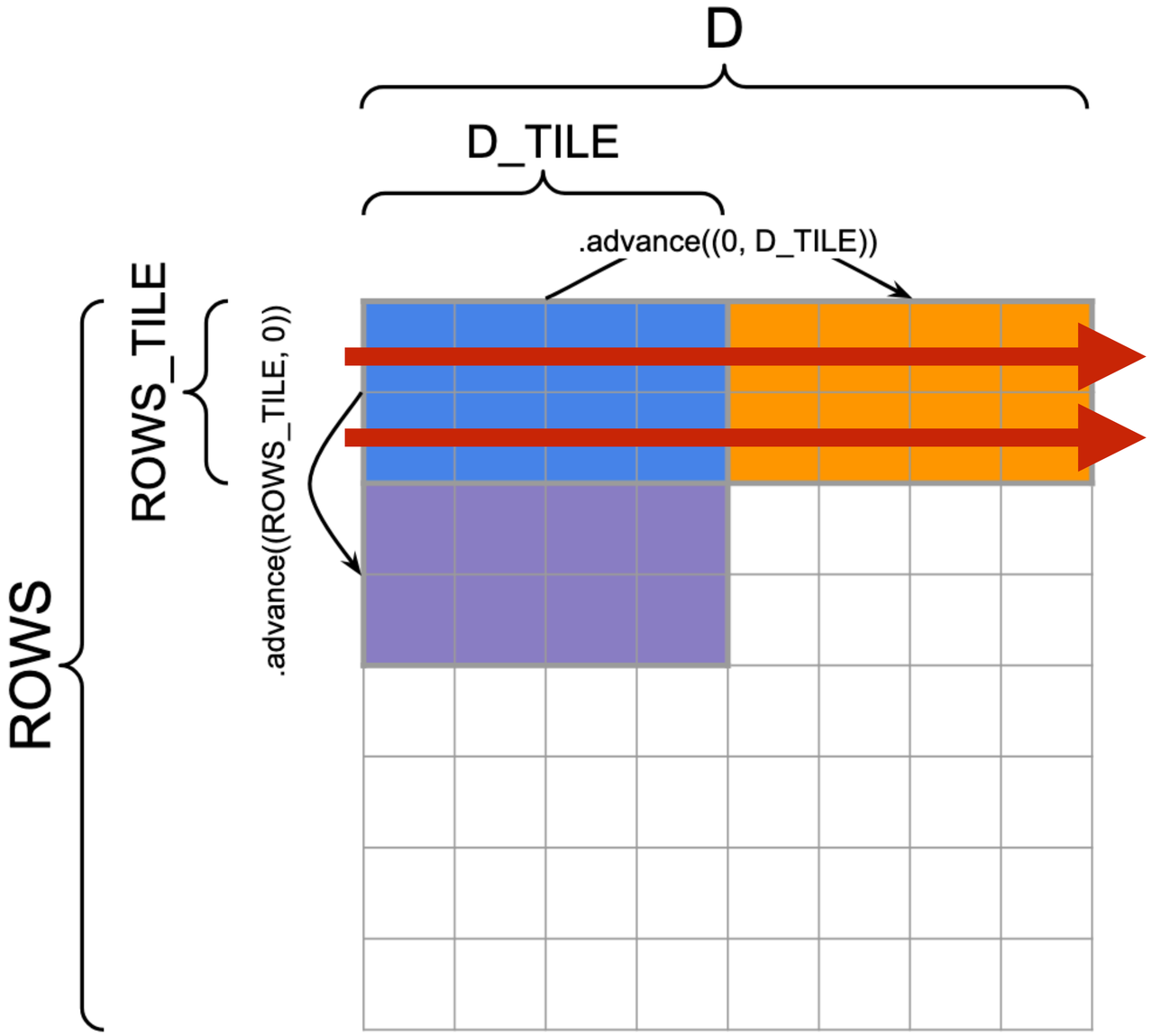
order tells us dim is the “fast” axis

```
    weight_block_ptr = tl.make_block_ptr(
        weight_ptr,
        shape=(D,),
        strides=(weight_stride_dim,),
        offsets=(0,),
        block_shape=(D_TILE_SIZE,),
        order=(0,),
    )

    output_block_ptr = tl.make_block_ptr(
        output_ptr,
        shape=(ROWS,),
        strides=(output_stride_row,),
        offsets=(row_tile_idx * ROWS_TILE_SIZE,),
        block_shape=(ROWS_TILE_SIZE,),
        order=(0,),
    )
```

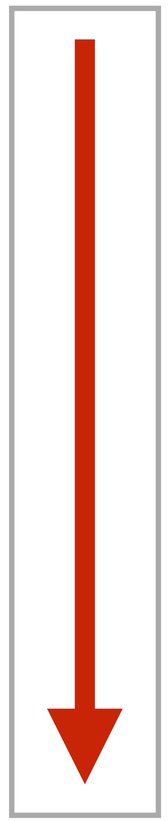
```python
# Initialize a buffer to write to
output = tl.zeros((ROWS_TILE_SIZE,), dtype=tl.float32)

for i in range(tl.cdiv(D, D_TILE_SIZE)):
    # Load the current block pointer
    # Since ROWS_TILE_SIZE might not divide ROWS, and D_TILE_SIZE might not divide D,
    # we need boundary checks for both dimensions
    row = tl.load(x_block_ptr, boundary_check=(0, 1), padding_option="zero")  # (ROWS_TILE_SIZE, D_TILE_SIZE)
    weight = tl.load(weight_block_ptr, boundary_check=(0,), padding_option="zero")  # (D_TILE_SIZE,)

    # Compute the weighted sum of the row.
    output += tl.sum(row * weight[None, :], axis=1)

    # Move the pointers to the next tile.
    # These are (rows, columns) coordinate deltas
    x_block_ptr = x_block_ptr.advance((0, D_TILE_SIZE))  # Move by D_TILE_SIZE in the last dimension
    weight_block_ptr = weight_block_ptr.advance((D_TILE_SIZE,))  # Move by D_TILE_SIZE

# Write output to the output block pointer (a single scalar per row).
# Since ROWS_TILE_SIZE might not divide ROWS, we need boundary checks
tl.store(output_block_ptr, output, boundary_check=(0,))
```

Block pointers are advanced

Triton: Weighted Sum



One execution of the kernel



weight

output

Memory coalescence depends on sizes

Administrative details

Operator Fusion

Triton: Weighted Sum

Flash Attention

Flash Attention: Backward Pass

Parallelism

Flash Attention

Recall: Attention

Forward pass of attention:

$$\mathbf{S} = \mathbf{Q}\mathbf{K}^\top \in \mathbb{R}^{N \times N}, \quad \mathbf{P} = \mathrm{softmax}(\mathbf{S}) \in \mathbb{R}^{N \times N}, \quad \mathbf{O} = \mathbf{P}\mathbf{V} \in \mathbb{R}^{N \times d}$$

(they omit the normalization)

Why might this be slow on a GPU?

Which of our five tricks is useful?

Trick 1: Low/mixed precision

Trick 2: Operator fusion

Trick 3: Recomputation

Trick 4: Memory coalescence

Trick 5: Tiling

Attention

Forward pass of attention:

$$\mathbf{S} = \mathbf{Q}\mathbf{K}^\top \in \mathbb{R}^{N\times N}, \quad \mathbf{P} = \mathrm{softmax}(\mathbf{S}) \in \mathbb{R}^{N\times N}, \quad \mathbf{O} = \mathbf{P}\mathbf{V} \in \mathbb{R}^{N\times d}$$

Backward:

$$\mathbf{dV} = \mathbf{P}^\top \mathbf{dO} \in \mathbb{R}^{N\times d}$$

$$\mathbf{dP} = \mathbf{dO}\mathbf{V}^\top \in \mathbb{R}^{N\times N}$$

$$\mathbf{dS} = \mathrm{dsoftmax}(\mathbf{dP}) \in \mathbb{R}^{N\times N}$$

$$\mathbf{dQ} = \mathbf{dS}\mathbf{K} \in \mathbb{R}^{N\times d}$$

$$\mathbf{dK} = \mathbf{Q}\mathbf{dS}^\top \in \mathbb{R}^{N\times d},$$

Let's ignore backward for now…

The challenge

Forward pass of attention:

$$\mathbf{S} = \mathbf{Q}\mathbf{K}^\top \in \mathbb{R}^{N\times N}, \quad \mathbf{P} = \mathrm{softmax}(\mathbf{S}) \in \mathbb{R}^{N\times N}, \quad \mathbf{O} = \mathbf{P}\mathbf{V} \in \mathbb{R}^{N\times d}$$

How to do this with as few reads and writes as possible?

Idea: construct **S** in tiles, use those to compute **P**, and build **O** directly

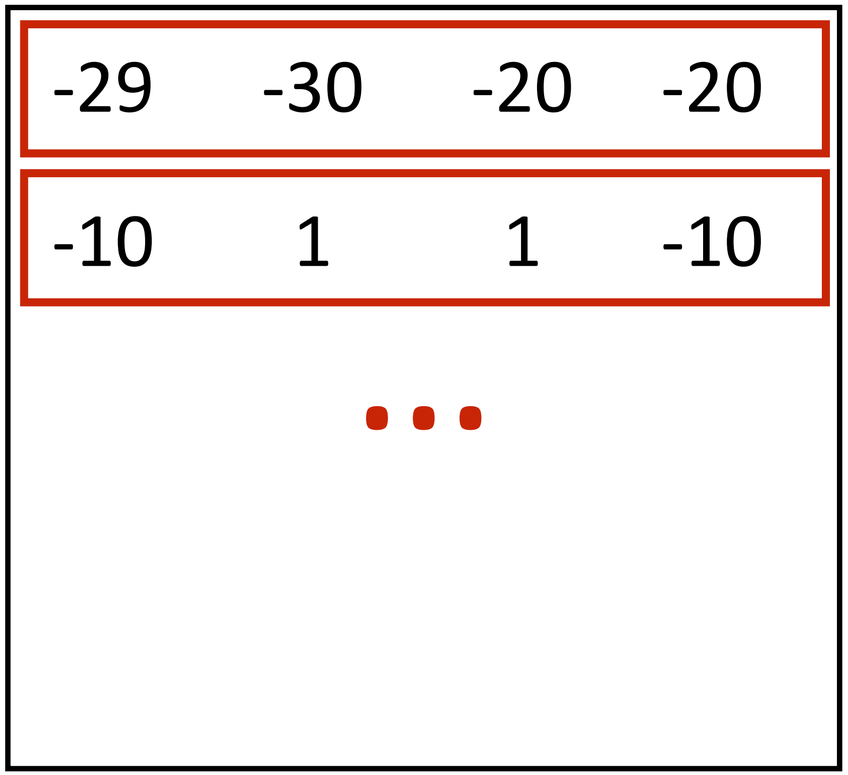
$$\text{Split } \mathbf{Q} \text{ into } T_q = \left\lceil \frac{N_q}{B_q} \right\rceil \text{ tiles } \mathbf{Q}_1, \ldots, \mathbf{Q}_{T_q} \text{ of size } B_q \times d$$
$$\text{Split } \mathbf{K}, \mathbf{V} \text{ into } T_k = \left\lceil \frac{N_k}{B_k} \right\rceil \text{ tiles } \mathbf{K}^{(1)}, \ldots, \mathbf{K}^{(T_k)} \text{ and } \mathbf{V}^{(1)}, \ldots, \mathbf{V}^{(T_k)} \text{ of size } B_k \times d$$

Each query becomes a row of attention. Handle in blocks of Bq rows

Why is this not so simple?

Softmax

S



Subtract off max (-20)

-9 -10 0 0

Exponentiate

0.0001 0.000045 1 1

Sum 2.000145

Normalize

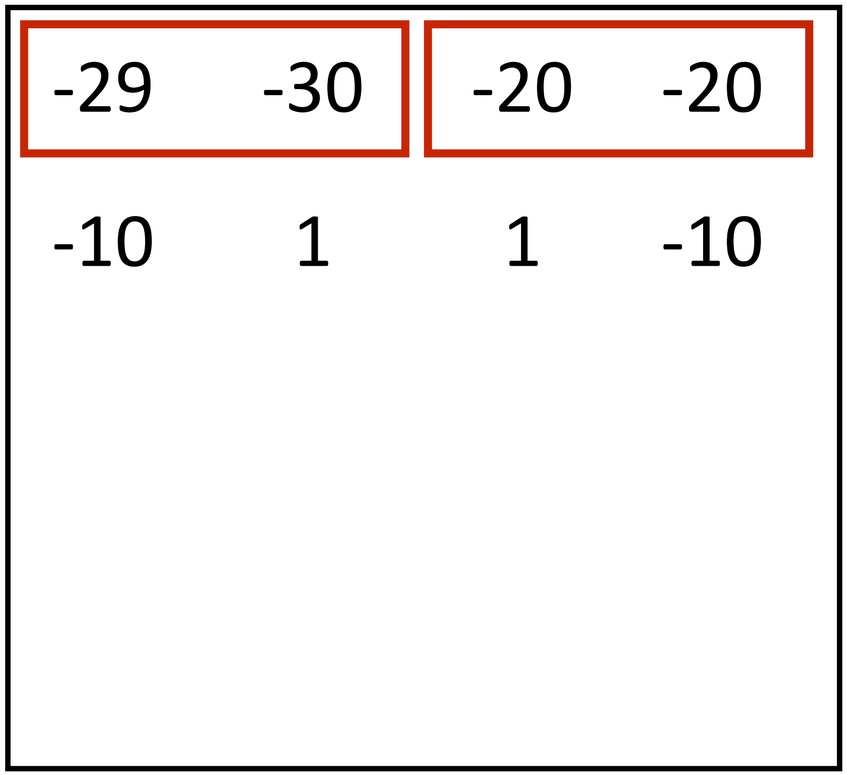
Then multiply by V to get O

$$\mathbf{P} = \mathrm{softmax}(\mathbf{S}) \in \mathbb{R}^{N \times N}$$

$$\mathbf{O} = \mathbf{PV} \in \mathbb{R}^{N \times d}$$

Softmax

S



Subtract off max (-20), but that's not in the first tile

Exponentiate

Sum (only partial)

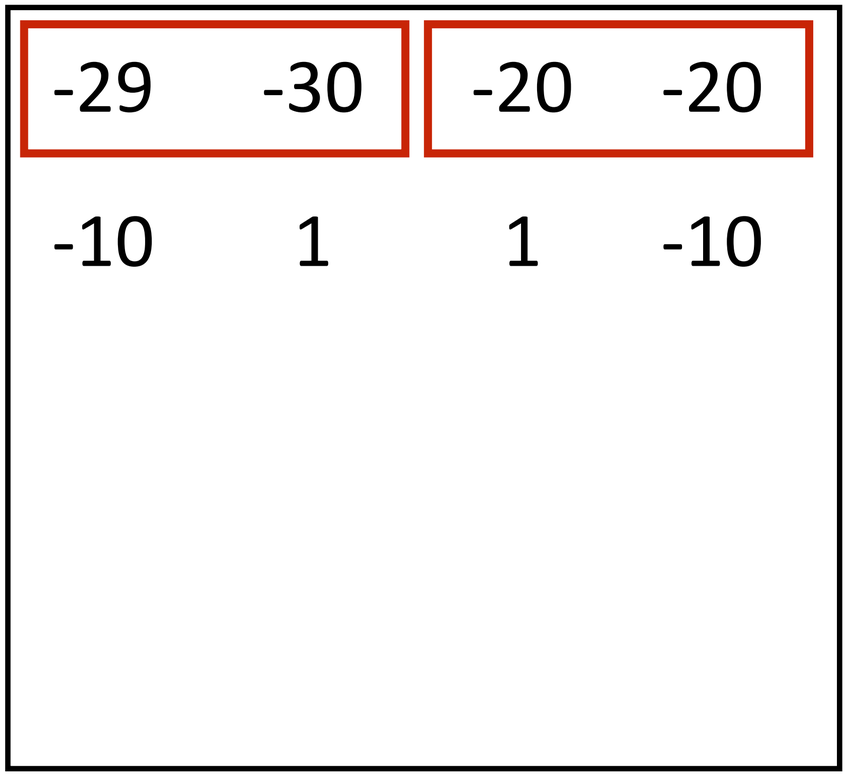
Normalize

Then multiply by V to get O

$$\mathbf{P} = \mathrm{softmax}(\mathbf{S}) \in \mathbb{R}^{N \times N}$$

$$\mathbf{O} = \mathbf{PV} \in \mathbb{R}^{N \times d}$$

Online Softmax



$$m^{(1)} = \text{rowmax}(\mathbf{S}^{(1)}) \in \mathbb{R}^{B_r}$$

$$\ell^{(1)} = \text{rowsum}(e^{\mathbf{S}^{(1)} - m^{(1)}}) \in \mathbb{R}^{B_r}$$

-29

partial sum with m1 “held out”

$$\tilde{\mathbf{P}}^{(1)} = \text{diag}(\ell^{(1)})^{-1} e^{\mathbf{S}^{(1)} - m^{(1)}} \in \mathbb{R}^{B_r \times B_c}$$

partially normalized

$$\mathbf{O}^{(1)} = \tilde{\mathbf{P}}^{(1)} \mathbf{V}^{(1)} = \text{diag}(\ell^{(1)})^{-1} e^{\mathbf{S}^{(1)} - m^{(1)}} \mathbf{V}^{(1)} \in \mathbb{R}^{B_r \times d}$$

$$m^{(2)} = \max(m^{(1)}, \text{rowmax}(\mathbf{S}^{(2)})) = m$$

-20

$$\ell^{(2)} = e^{m^{(1)} - m^{(2)}} \ell^{(1)} + \text{rowsum}(e^{\mathbf{S}^{(2)} - m^{(2)}})$$

new partial sum with m2 “held out”

Require: $\mathbf{Q} \in \mathbb{R}^{N_q \times d}$, $\mathbf{K}, \mathbf{V} \in \mathbb{R}^{N_k \times d}$, tile sizes B_q, B_k

Split $\mathbf{Q}$ into $T_q = \left\lceil \frac{N_q}{B_q} \right\rceil$ tiles $\mathbf{Q}_1, \ldots, \mathbf{Q}_{T_q}$ of size $B_q \times d$

Split $\mathbf{K}, \mathbf{V}$ into $T_k = \left\lceil \frac{N_k}{B_k} \right\rceil$ tiles $\mathbf{K}^{(1)}, \ldots, \mathbf{K}^{(T_k)}$ and $\mathbf{V}^{(1)}, \ldots, \mathbf{V}^{(T_k)}$ of size $B_k \times d$

for $i = 1, \ldots, T_q$ **do**

  Load $\mathbf{Q}_i$ from global memory

  Initialize $\mathbf{O}_i^{(0)} = \mathbf{0} \in \mathbb{R}^{B_q \times d}, l_i^{(0)} = 0 \in \mathbb{R}^{B_q}, m_i^{(0)} = -\infty \in \mathbb{R}^{B_q}$

  for $j = 1, \ldots, T_k$ **do**

    Load $\mathbf{K}^{(j)}, \mathbf{V}^{(j)}$ from global memory

    Compute tile of pre-softmax attention scores $\mathbf{S}_i^{(j)} = \frac{\mathbf{Q}_i (\mathbf{K}^{(j)})^\top}{\sqrt{d}} \in \mathbb{R}^{B_q \times B_k}$

    Compute $m_i^{(j)} = \max\left(m_i^{(j-1)}, \text{rowmax}\left(\mathbf{S}_i^{(j)}\right)\right) \in \mathbb{R}^{B_q}$

    Compute $\tilde{\mathbf{P}}_i^{(j)} = \exp\left(\mathbf{S}_i^{(j)} - m_i^{(j)}\right) \in \mathbb{R}^{B_q \times B_k}$

    Compute $l_i^{(j)} = \exp\left(m_i^{(j-1)} - m_i^{(j)}\right) l_i^{(j-1)} + \text{rowsum}\left(\tilde{\mathbf{P}}_i^{(j)}\right) \in \mathbb{R}^{B_q}$

    Compute $\mathbf{O}_i^{(j)} = \text{diag}\left(\exp\left(m_i^{(j-1)} - m_i^{(j)}\right)\right) \mathbf{O}_i^{(j-1)} + \tilde{\mathbf{P}}_i^{(j)} \mathbf{V}^{(j)}$

  end for

  Compute $\mathbf{O}_i = \text{diag}\left(l_i^{(T_k)}\right)^{-1} \mathbf{O}_i^{(T_k)}$

  Compute $L_i = m_i^{(T_k)} + \log\left(l_i^{(T_k)}\right)$

  Write $\mathbf{O}_i$ to global memory as the i-th tile of $\mathbf{O}$.

  Write L_i to global memory as the i-th tile of L.

end for

Return the output $\mathbf{O}$ and the logsumexp L.

why is *L* needed at all? backward pass!

for $i = 1, \ldots, T_q$ **do**

Load $\mathbf{Q}_i$ from global memory

Initialize $\mathbf{O}_i^{(0)} = \mathbf{0} \in \mathbb{R}^{B_q \times d}, l_i^{(0)} = 0 \in \mathbb{R}^{B_q}, m_i^{(0)} = -\infty \in \mathbb{R}^{B_q}$

for $j = 1, \ldots, T_k$ **do**

Load $\mathbf{K}^{(j)}, \mathbf{V}^{(j)}$ from global memory

Compute tile of pre-softmax attention scores $\mathbf{S}_i^{(j)} = \frac{\mathbf{Q}_i (\mathbf{K}^{(j)})^\top}{\sqrt{d}} \in \mathbb{R}^{B_q \times B_k}$

Compute $m_i^{(j)} = \max\left(m_i^{(j-1)}, \text{rowmax}\left(\mathbf{S}_i^{(j)}\right)\right) \in \mathbb{R}^{B_q}$

Compute $\tilde{\mathbf{P}}_i^{(j)} = \exp\left(\mathbf{S}_i^{(j)} - m_i^{(j)}\right) \in \mathbb{R}^{B_q \times B_k}$

Compute $l_i^{(j)} = \exp\left(m_i^{(j-1)} - m_i^{(j)}\right) l_i^{(j-1)} + \text{rowsum}\left(\tilde{\mathbf{P}}_i^{(j)}\right) \in \mathbb{R}^{B_q}$

Compute $\mathbf{O}_i^{(j)} = \text{diag}\left(\exp\left(m_i^{(j-1)} - m_i^{(j)}\right)\right) \mathbf{O}_i^{(j-1)} + \tilde{\mathbf{P}}_i^{(j)} \mathbf{V}^{(j)}$

end for

Compute $\mathbf{O}_i = \text{diag}\left(l_i^{(T_k)}\right)^{-1} \mathbf{O}_i^{(T_k)}$

Compute $L_i = m_i^{(T_k)} + \log\left(l_i^{(T_k)}\right)$

m, *l* are easy to store

finally normalize by *l*

sanity-check: why is *m* not needed here?

Administrative details

Operator Fusion

Triton: Weighted Sum

Flash Attention

Flash Attention: Backward Pass

Parallelism

Flash Attention Backward Pass

Backward Pass

$$\mathbf{S} = \mathbf{Q}\mathbf{K}^\top \in \mathbb{R}^{N\times N}, \quad \mathbf{P} = \mathrm{softmax}(\mathbf{S}) \in \mathbb{R}^{N\times N}, \quad \mathbf{O} = \mathbf{P}\mathbf{V} \in \mathbb{R}^{N\times d}$$

$$\mathbf{dV} = \mathbf{P}^\top \mathbf{dO} \in \mathbb{R}^{N\times d}$$

$$\mathbf{dP} = \mathbf{dO}\mathbf{V}^\top \in \mathbb{R}^{N\times N}$$

$$\mathbf{dS} = \mathrm{dsoftmax}(\mathbf{dP}) \in \mathbb{R}^{N\times N}$$

$$\mathbf{dQ} = \mathbf{dS}\mathbf{K} \in \mathbb{R}^{N\times d}$$

$$\mathbf{dK} = \mathbf{Q}\mathbf{dS}^\top \in \mathbb{R}^{N\times d},$$

Gradients

$$\mathbf{O} = \mathbf{PV} \in \mathbb{R}^{N \times d}$$

$$\mathbf{dV} = \mathbf{P}^\top \mathbf{dO} \in \mathbb{R}^{N \times d}$$

$$\mathbf{dP} = \mathbf{dOV}^\top \in \mathbb{R}^{N \times N}$$

$$dO = \frac{\partial L}{\partial O}$$

These rules follow from standard rules of differentiation applied to matrices (you can break them down into individual elements)

Gradient of Softmax

$\mathbf{dS} = \mathrm{dsoftmax}(\mathbf{dP}) \in \mathbb{R}^{N \times N}$, consider each row $p_i = \dfrac{e^{s_i}}{\sum_{k=1}^{n} e^{s_k}}$

From ChatGPT:

$$\frac{\partial p_i}{\partial s_j} = \frac{\partial}{\partial s_j}\left(\frac{e^{s_i}}{Z}\right) = \frac{\delta_{ij} e^{s_i}\, Z - e^{s_i}\, \frac{\partial Z}{\partial s_j}}{Z^2}.$$

But $\frac{\partial Z}{\partial s_j} = e^{s_j}$. Substitute and simplify using $p_i = e^{s_i}/Z$ and $p_j = e^{s_j}/Z$:

$$\frac{\partial p_i}{\partial s_j} = \frac{\delta_{ij} e^{s_i}}{Z} - \frac{e^{s_i}}{Z}\frac{e^{s_j}}{Z} = \delta_{ij} p_i - p_i p_j = p_i(\delta_{ij} - p_j).$$

So the Jacobian $J \in \mathbb{R}^{n \times n}$ with $J_{ij} = \partial p_i / \partial s_j$ is

$$J = \mathrm{diag}(p) - pp^{\top}.$$

$\mathbf{dS} = \mathrm{dsoftmax}(\mathbf{dP}) \in \mathbb{R}^{N \times N}$ means applying this operator row-wise to dP

n-dim gradient vector, n x n operator ->

More intuition

Think of P as an (n x n)-length vector of probabilities and S as an (n x n)-length vector of scores

$$\frac{\partial \mathrm{vec}(P)}{\partial \mathrm{vec}(S)} = \begin{bmatrix} J(p_1) & 0 & \dots & 0 \\ 0 & J(p_2) & \dots & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \dots & J(p_L) \end{bmatrix}$$

since the first row of P just depends on the first row of S, etc.

Essentially we are collapsing this operation down

Backward Pass

$$\mathbf{S} = \mathbf{Q}\mathbf{K}^\top \in \mathbb{R}^{N\times N}, \quad \mathbf{P} = \mathrm{softmax}(\mathbf{S}) \in \mathbb{R}^{N\times N}, \quad \mathbf{O} = \mathbf{P}\mathbf{V} \in \mathbb{R}^{N\times d}$$

$$\mathbf{dV} = \mathbf{P}^\top \mathbf{dO} \in \mathbb{R}^{N\times d}$$

$$\mathbf{dP} = \mathbf{dO}\mathbf{V}^\top \in \mathbb{R}^{N\times N}$$

$$\mathbf{dS} = \mathrm{dsoftmax}(\mathbf{dP}) \in \mathbb{R}^{N\times N}$$

$$\mathbf{dQ} = \mathbf{dS}\mathbf{K} \in \mathbb{R}^{N\times d}$$

$$\mathbf{dK} = \mathbf{Q}\mathbf{dS}^\top \in \mathbb{R}^{N\times d},$$

We haven't stored S or P, but we do have the sums L. So (1) gradients are gnarly; (2) we have to do some recomputation, but it's actually a bit more straightforward

Algorithm 2 Tiled FlashAttention-2 backward pass

Require: $\mathbf{Q}, \mathbf{O}, \mathbf{dO} \in \mathbb{R}^{N_q \times d}$, $\mathbf{K}, \mathbf{V} \in \mathbb{R}^{N_k \times d}$, $L \in \mathbb{R}^{N_q}$, tile sizes B_q, B_k

Compute $D = \text{rowsum}(\mathbf{dO} \circ \mathbf{O}) \in \mathbb{R}^{N_q}$

Split $\mathbf{Q}, \mathbf{O}, \mathbf{dO}$ into $T_q = \left\lceil \frac{N_q}{B_q} \right\rceil$ tiles $\mathbf{Q}_1, \dots, \mathbf{Q}_{T_q}$, $\mathbf{O}_1, \dots, \mathbf{O}_{T_q}$, $\mathbf{dO}_1, \dots, \mathbf{dO}_{T_q}$, each of size $B_q \times d$

Split $\mathbf{K}, \mathbf{V}$ into $T_k = \left\lceil \frac{N_k}{B_k} \right\rceil$ tiles $\mathbf{K}^{(1)}, \dots, \mathbf{K}^{(T_k)}$ and $\mathbf{V}^{(1)}, \dots, \mathbf{V}^{(T_k)}$, each of size $B_k \times d$

Split L, D into T_q tiles $L_1, \dots, L_{T_q}$ and $D_1, \dots, D_{T_q}$, each of size B_q

for $j = 1, \dots, T_k$ **do**

  Load $\mathbf{K}^{(j)}, \mathbf{V}^{(j)}$ from global memory

  Initialize $\mathbf{dK}^{(j)} = \mathbf{dV}^{(j)} = \mathbf{0} \in \mathbb{R}^{B_k \times d}$

  for $i = 1, \dots, T_q$ **do**

    Load $\mathbf{Q}_i, \mathbf{O}_i, \mathbf{dO}_i, \mathbf{dQ}_i$ from global memory

    Compute tile of attention scores $\mathbf{S}_i^{(j)} = \frac{\mathbf{Q}_i (\mathbf{K}^{(j)})^\top}{\sqrt{d}} \in R^{B_q \times B_k}$

    Compute attention probabilities $\mathbf{P}_i^{(j)} = \exp\left(\mathbf{S}_i^{(j)} - L_i\right) \in \mathbb{R}^{B_q \times B_k}$

    Compute $\mathbf{dV}^{(j)} \mathrel{+}= (\mathbf{P}_i^{(j)})^\top \mathbf{dO}_i \in \mathbb{R}^{B_k \times d}$

    Compute $\mathbf{dP}_i^{(j)} = \mathbf{dO}_i \mathbf{V}_j^\top \in \mathbb{R}^{B_q \times B_k}$

    Compute $\mathbf{dS}_i^{(j)} = \mathbf{P}_i^{(j)} \circ \left(\mathbf{dP}_i^{(j)} - D_i\right) / \sqrt{d} \in \mathbb{R}^{B_q \times B_k}$

    Load $\mathbf{dQ}_i$ from global memory, then update $\mathbf{dQ}_i \mathrel{+}= \mathbf{dS}_i^{(j)} \mathbf{K}^{(j)} \in \mathbb{R}^{B_q \times d}$, and write back to global memory. Must be atomic for correctness!

    Compute $\mathbf{dK}^{(j)} \mathrel{+}= (\mathbf{dS}_i^{(j)})^\top \mathbf{Q}_i \in \mathbb{R}^{B_k \times d}$.

  end for

  Write $\mathbf{dK}^{(j)}$ and $\mathbf{dV}^{(j)}$ to global memory as the j-th tiles of $\mathbf{dK}$ and $\mathbf{dV}$.

end for

Return $\mathbf{dQ}, \mathbf{dK}, \mathbf{dV}$.

for $j = 1, \ldots, T_k$ **do**

Load $\mathbf{K}^{(j)}, \mathbf{V}^{(j)}$ from global memory

Initialize $\mathbf{dK}^{(j)} = \mathbf{dV}^{(j)} = \mathbf{0} \in \mathbb{R}^{B_k \times d}$

for $i = 1, \ldots, T_q$ **do**

Load $\mathbf{Q}_i, \mathbf{O}_i, \mathbf{dO}_i, \mathbf{dQ}_i$ from global memory

Compute tile of attention scores $\mathbf{S}_i^{(j)} = \frac{\mathbf{Q}_i (\mathbf{K}^{(j)})^\top}{\sqrt{d}} \in R^{B_q \times B_k}$

Compute attention probabilities $\mathbf{P}_i^{(j)} = \exp\left(\mathbf{S}_i^{(j)} - L_i\right) \in \mathbb{R}^{B_q \times B_k}$

Compute $\mathbf{dV}^{(j)} \mathrel{+}= (\mathbf{P}_i^{(j)})^\top \mathbf{dO}_i \in \mathbb{R}^{B_k \times d}$

Compute $\mathbf{dP}_i^{(j)} = \mathbf{dO}_i \mathbf{V}_j^\top \in \mathbb{R}^{B_q \times B_k}$

Compute $\mathbf{dS}_i^{(j)} = \mathbf{P}_i^{(j)} \circ \left(\mathbf{dP}_i^{(j)} - D_i\right) / \sqrt{d} \in \mathbb{R}^{B_q \times B_k}$

Load $\mathbf{dQ}_i$ from global memory, then update $\mathbf{dQ}_i \mathrel{+}= \mathbf{dS}_i^{(j)} \mathbf{K}^{(j)} \in \mathbb{R}^{B_q \times d}$, and write back to memory. Must be atomic for correctness!

Compute $\mathbf{dK}^{(j)} \mathrel{+}= (\mathbf{dS}_i^{(j)})^\top \mathbf{Q}_i \in \mathbb{R}^{B_k \times d}$.

end for

Write $\mathbf{dK}^{(j)}$ and $\mathbf{dV}^{(j)}$ to global memory as the j-th tiles of $\mathbf{dK}$ and $\mathbf{dV}$.

Administrative details

Operator Fusion

Triton: Weighted Sum

Flash Attention

Flash Attention: Backward Pass

Parallelism

Parallelism

Parallelism

- Easier to get more GPUs than to get faster GPUs
- How to leverage *multiple* GPUs?
- Types of parallelism we will discuss:
  - Data parallelism
  - Fully-sharded data parallelism (FSDP)
  - Tensor parallelism/pipeline parallelism

Data Parallelism

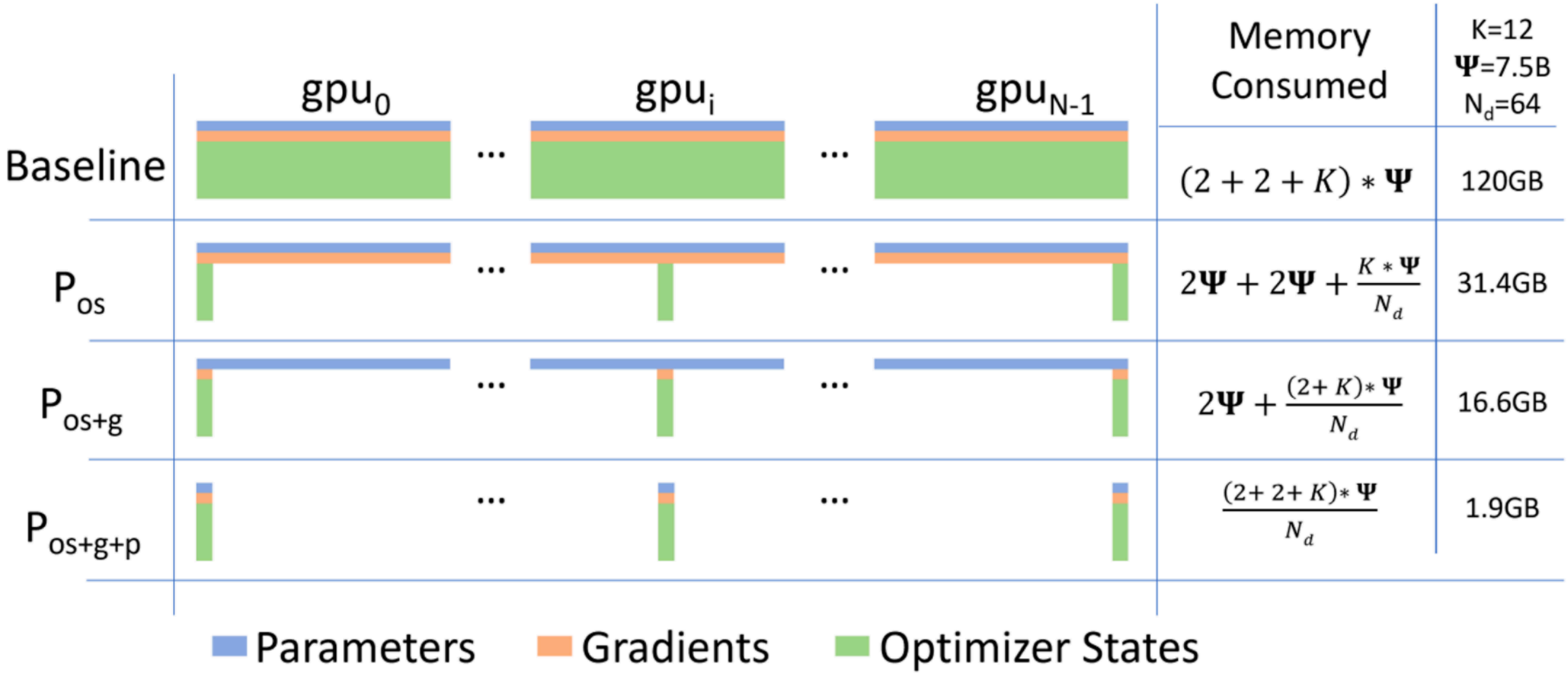
- Suppose we have M nodes (or cards) and batch size B. Run B/M examples on each node/card
- Does this work well for inference?
- Does this work well for training?
  - Every card/node has to store the entire weights & optimizer state. When doing mixed-precision, you're storing a lot:

- 2 bytes for FP/BF 16 model parameters
- 2 bytes for FP/BF 16 gradients
- 4 bytes for FP32 master weights (the thing you accumulate into in SGD)
- 4 (or 2) bytes for FP32/BF16 Adam first moment estimates
- 4 (or 2) bytes for FP32/BF16 Adam second moment estimates

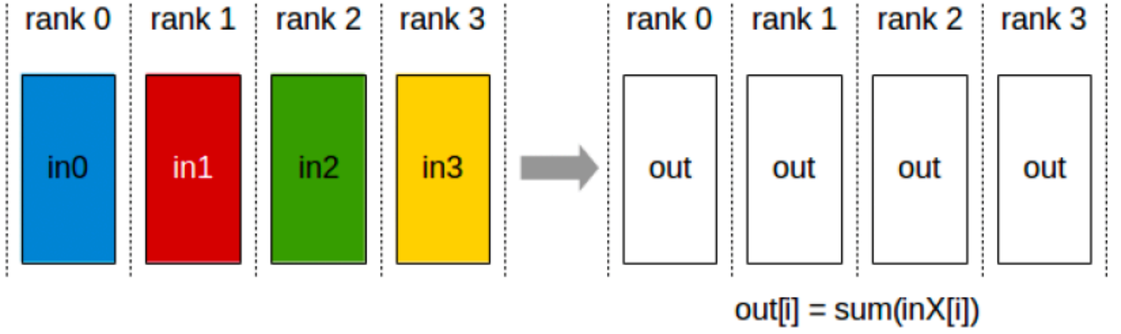
"Optimizer state"

ZeRO

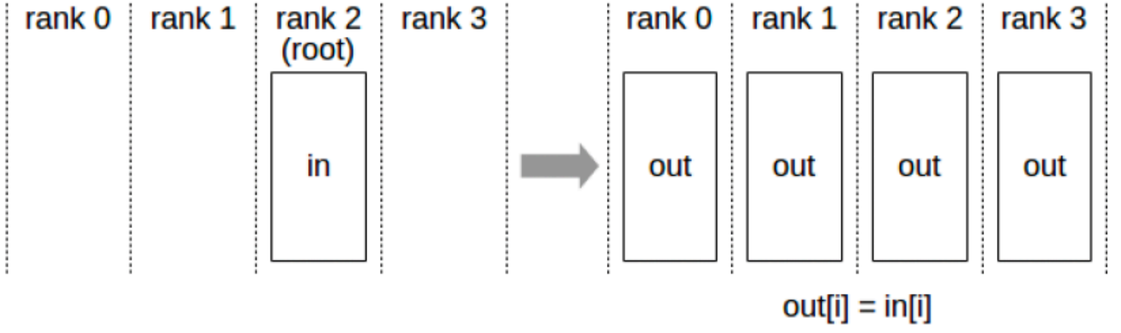
- Split things across GPUs



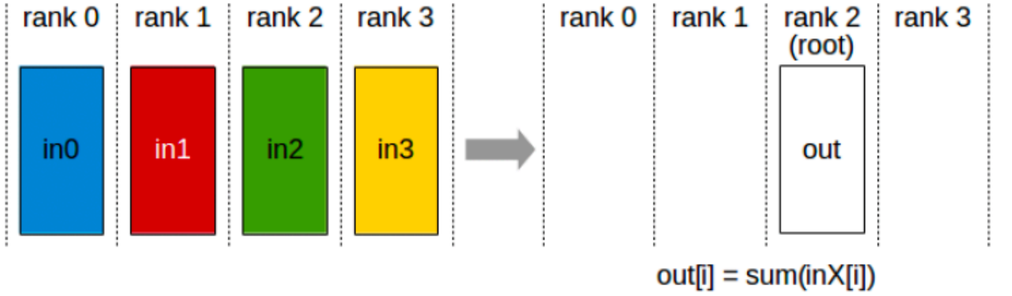
All reduce



Broadcast

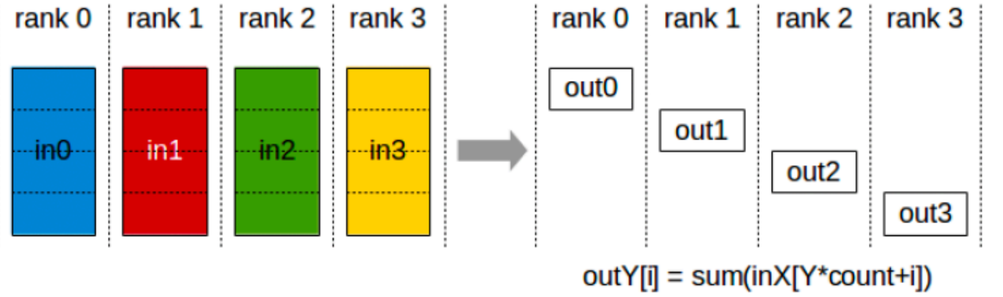


Reduce



All Gather

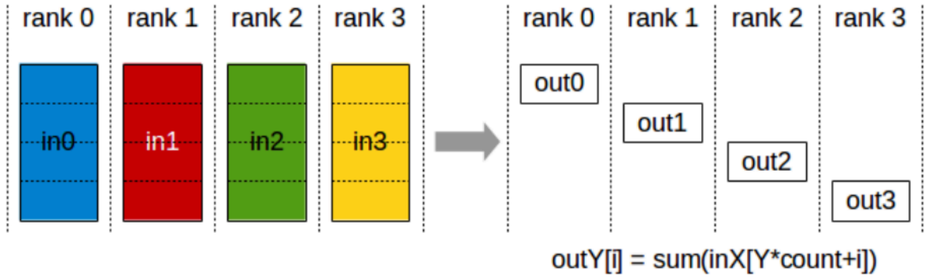
Reduce Scatter



ZeRO Stage 1

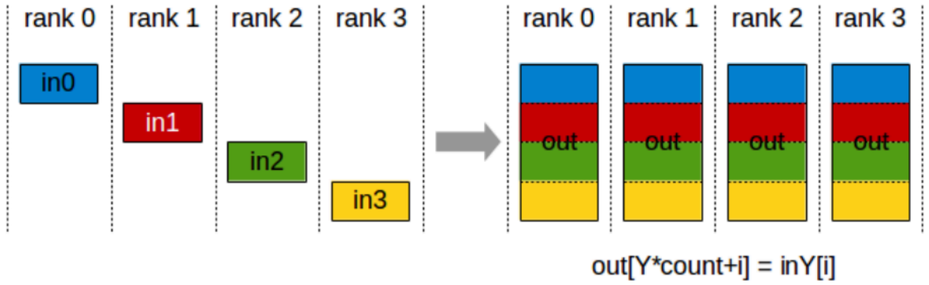
Step 1. Everyone computes a full gradient on their subset of the batch

Step 2. ReduceScatter the gradients – incur #params communication cost



Step 3. Each machine updates their param using their gradient + state.

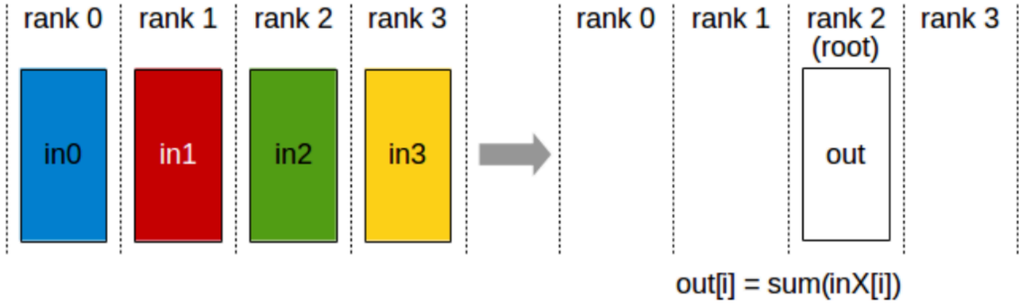
Step 4. All Gather the parameters – incur #params communication cost



ZeRO Stage 2

Step 1. Everyone incrementally goes backward on the computation graph

Step 1a. After computing a layer's gradients, immediately reduce to send this to the right worker

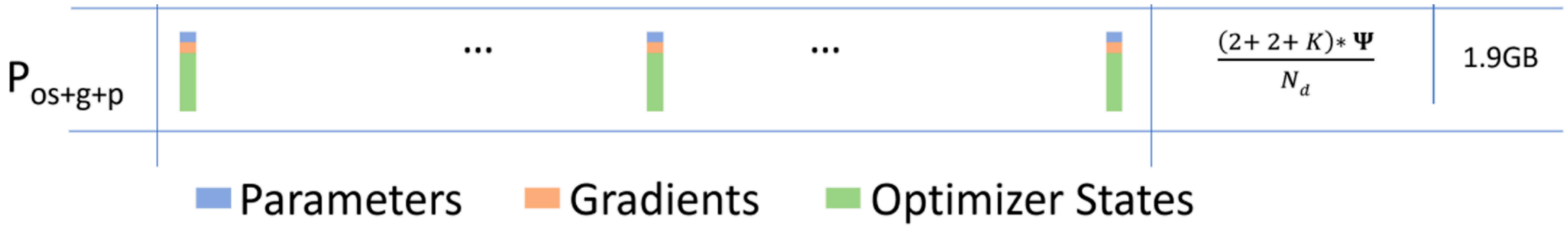


Step 1b. Once gradients are not needed in the backward graph, immediately free it.

Step 2. Each machine updates their param using their gradient + state.

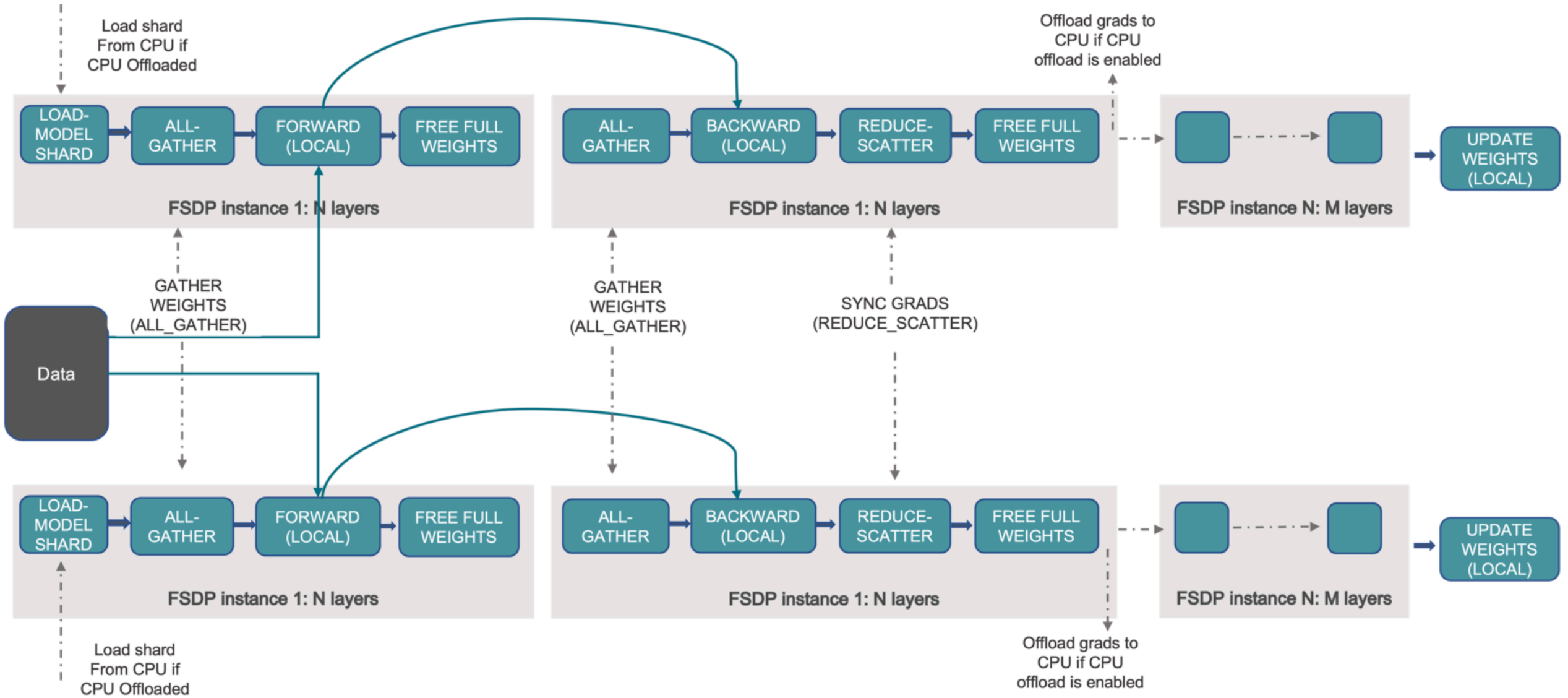
Step 3. All Gather the parameters.

ZeRO Stage 3 (FSDP)



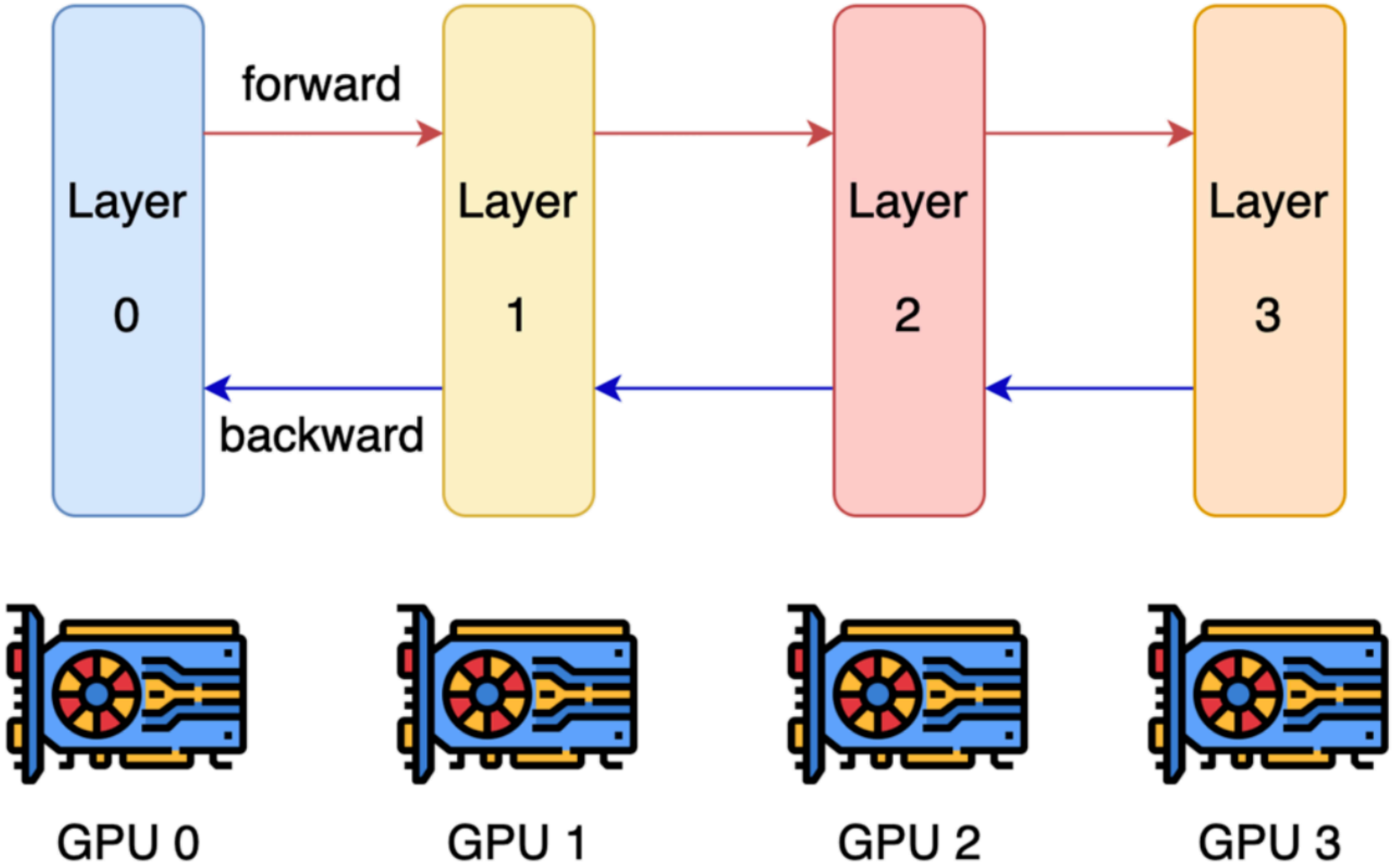
- Now everything is sharded. Each node only has a fraction of the params
- As we move through the computation, each node will have to gather the shards of parameters from all other nodes

ZeRO Stage 3 (FSDP)



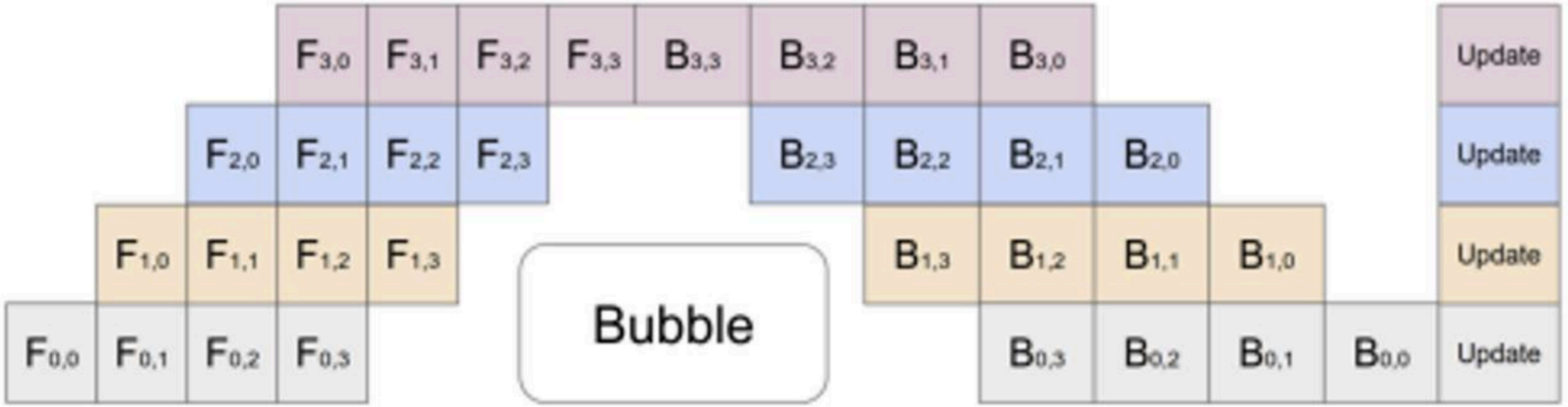
Pipeline Parallelism

- FSDP is only for training. What about making inference fast?



Pipeline Parallelism

- Your GPUs will naturally be idle sometimes under this scheme



- Good memory properties and less communication (only communicate activations), but only a good idea with large batches and when the network is slow.

Other Types of Parallelism

- Tensor Parallelism: split individual tensors across GPUs, each GPU processes a different slice of activations
- Stacks with other forms of parallelism

Administrative details

Operator Fusion

Triton: Weighted Sum

Flash Attention

Flash Attention: Backward Pass

Parallelism

Next time

- This concludes our discussion of GPUs/how to make things fast
- Next time: principles of scaling laws and how they impact LLM training
- Then: moving on from pre-training into post-training (SFT, RL)