DS-GA 1011: Fundamentals of Natural Language Processing

Lecture 10: GPUs & Efficiency



NYU

Greg Durrett

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

Administrative details and recap

Efficiency

GPUs

GPU Principles

Profiling

Parallelism



Administrative details and recap

Efficiency

GPUs

GPU Principles

Profiling

Parallelism

Administrative details and recap

Administrivia

- Assignment 4 due today, Quiz 3 today
- Assignment 3, proposals back
- Final projects due December 14
- Lab: primer on tools for model inference training (slime, SGLang, etc.)
- No class next week

Today

- Crash course on GPUs and some principles of making things fast

Administrative details and recap

Efficiency

GPUs

GPU Principles

Profiling

Parallelism

Efficiency

Resource Accounting

Question: How long would it take to train a 70B parameter model on 15T tokens on 1024 H100s?

where does 6 come from?

```
total_flops = 6 * 70e9 * 15e12  # @inspect total_flops
h100_flop_per_sec = 1979e12 / 2
mfu = 0.5
flops_per_day = h100_flop_per_sec * mfu * 1024 * 60 * 60 * 24  # @inspect flops_per_day
days = total_flops / flops_per_day
```

what is MFU?

Answer: Around 143

Question: What's the largest model trainable on 8 H100s with AdamW (naively)?

```
h100_bytes = 80e9  # @inspect h100_bytes
bytes_per_parameter = 4 + 4 + (4 + 4)  # parameters, gradients, optimizer state  @
num_parameters = (h100_bytes * 8) / bytes_per_parameter  # @inspect num_parameters
print(num_parameters)
```

Answer: 40B

Resource Accounting

What has to be stored? (let's brainstorm)

https://erees.dev/transformer-memory/

Resource Accounting

What has to be stored?

At the start of the forward pass:

- FP32 copies of the weights of your model, $M_{\text{model}} = 4N_{\text{param}} + 4N_{\text{buf}}$ (fp32 implies 4 bytes per element)
- FP32 copies of optimizer states, 2 for adam, $M_{\text{optimizer}} = 8N_{\text{param}}$
- Copies of your data and targets, assuming int64 inputs (as in nanoGPT), $M_{\text{data}} = 2 \times \text{Bsz} \times T \times 8$ (int64 implies 8 bytes per element)

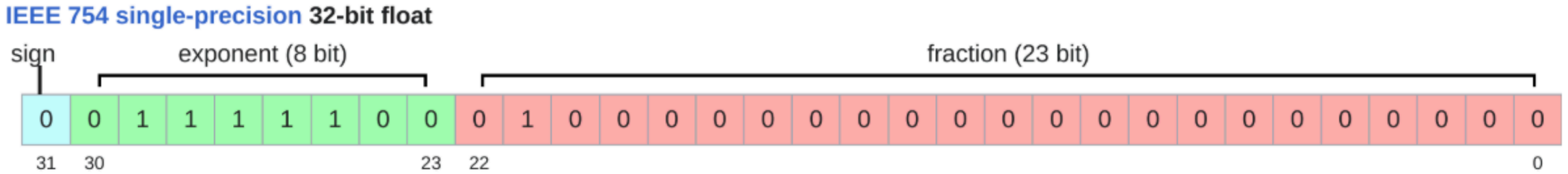
After the backwards pass (and possibly persisting):

- FP32 copies of the gradients size, $M_{\text{gradients}} = 4N_{\text{param}}$

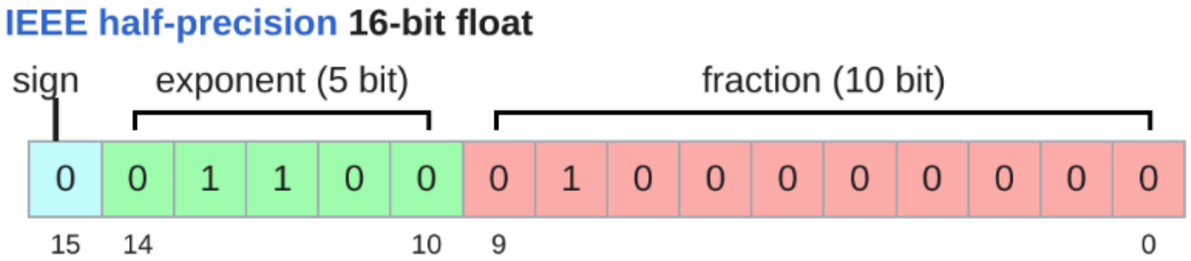
What is fp32, int64, …?

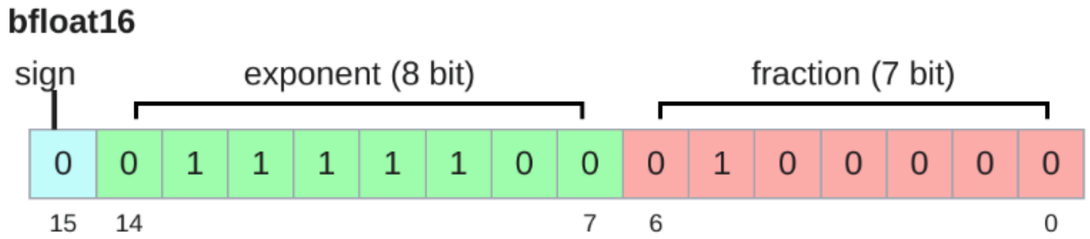
https://erees.dev/transformer-memory/

Data Types



"brain float" from Google





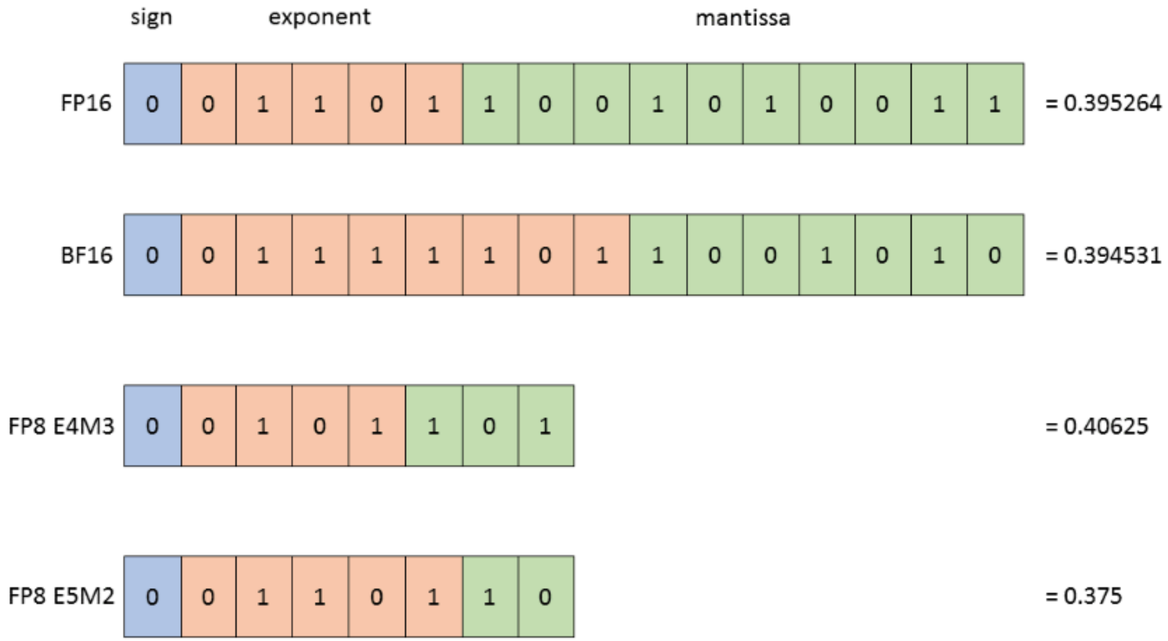
1e-8 is an underflow

Goes down to 1e-38 but lower resolution

How much memory to store a linear layer from d_{model} to 4 d_{model} if dmodel is 12k?

Data Types

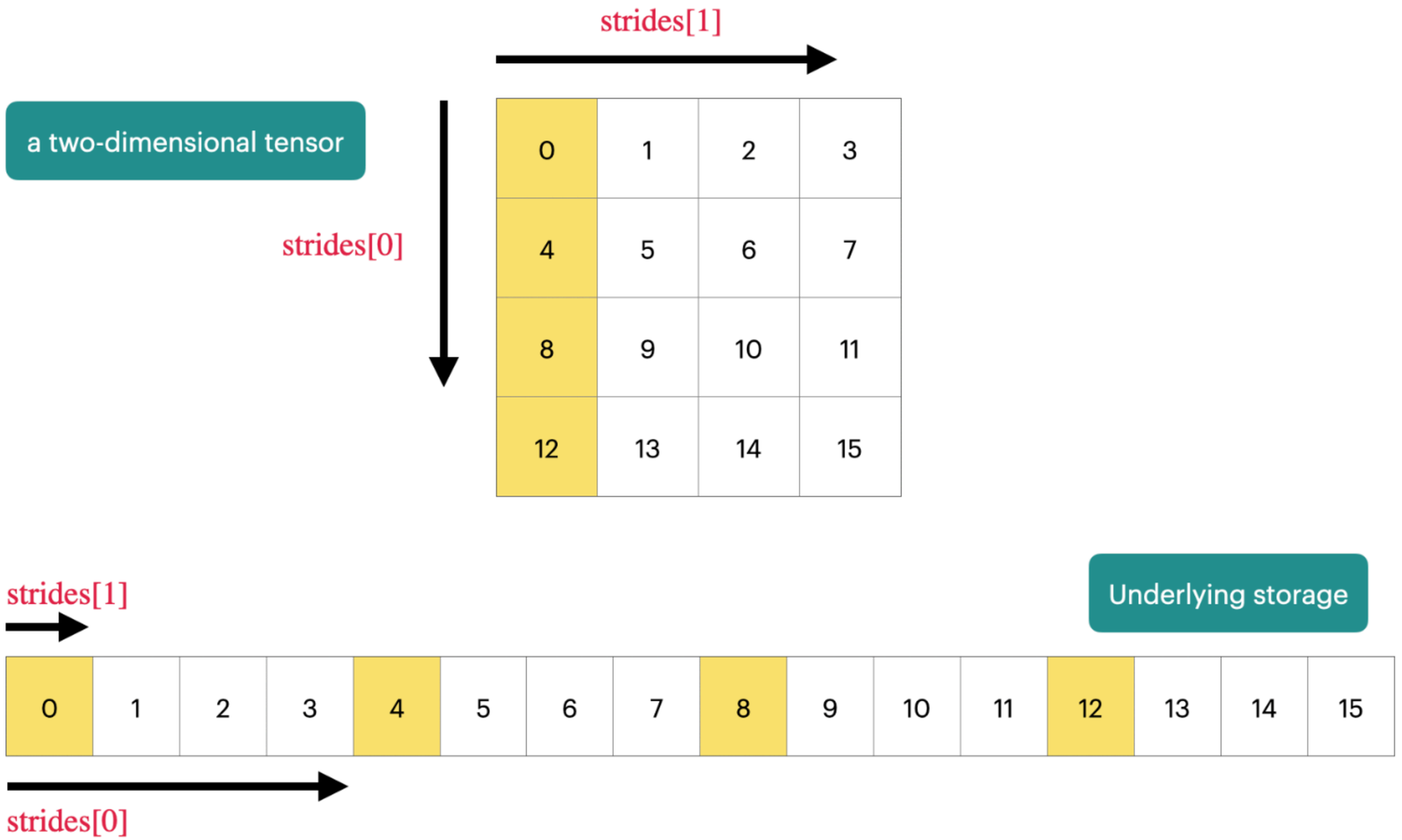
https://docs.nvidia.com/deeplearning/transformer-engine/user-guide/examples/fp8_primer.html



Both E4M3 and E5M2 supported on H100

Mixed precision training gets the best of both worlds: fast without losing accuracy

Tensors



Slicing, transpose, etc. do not reallocate memory, but just change the view

Resource Accounting

FLOPs: floating-point operations (measure of computation done)

- How expensive is adding two M x N matrices A and B? M * N
- How expensive is A^TB? 2 * N^2 * M

Why 2? multiply the elements, then accumulate

FLOP/s: floating-point operations per second (also written as FLOPS), which is used to measure the speed of hardware.

A100 has a peak performance of 312 teraFLOP/s [spec]

```
assert a100_flop_per_sec == 312e12
```

H100 has a peak performance of 1979 teraFLOP/s with sparsity, 50% without [spec]

```
assert h100_flop_per_sec == 1979e12 / 2
```

Resource Accounting: Forward

Case study: linear layer. How many FLOPs for the forward pass on this layer?

Batch = 1024

Dim = 256

Hidden dim size = 64

2 * 1024 * 256 * 64

2 * data * model params

Resource Accounting: Backward

Case study: two linear layers. How many FLOPs for forward+backward passes?

Batch = 1024

Dim = 256

Hidden dim size = 64

Output size = 64

Model: x --w1--> h1 --w2--> h2 -> loss

Forward pass for w2: 2 * 1024 * 64 * 64

Compute grad for w2: 2 * 1024 * 64 * 64

Do backward pass to w1: 2 * 1024 * 64 * 64

Total F+B:

6 * data points * params

Resource Accounting

Further reading:

https://erees.dev/transformer-memory/

https://www.adamcasson.com/posts/transformer-flops

Administrative details and recap

Efficiency

GPUs

GPU Principles

Profiling

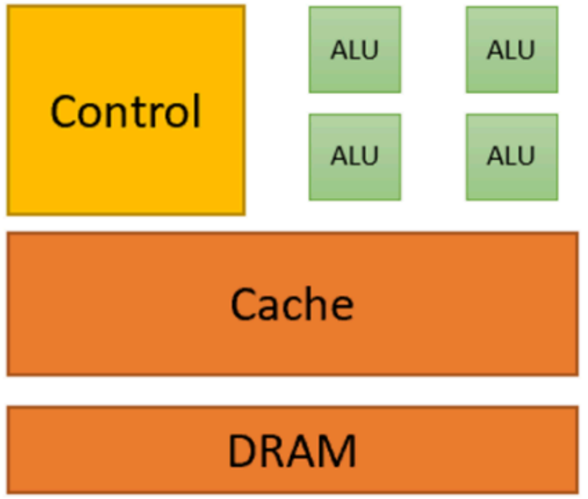
Parallelism

GPUs

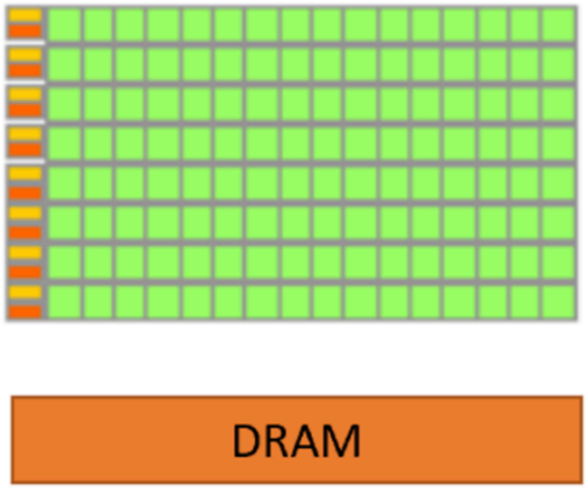
Slide credits for this section:
Tatsu Hashimoto / CS 336
(his credits: Horace He, CUDA Mode)

GPUs

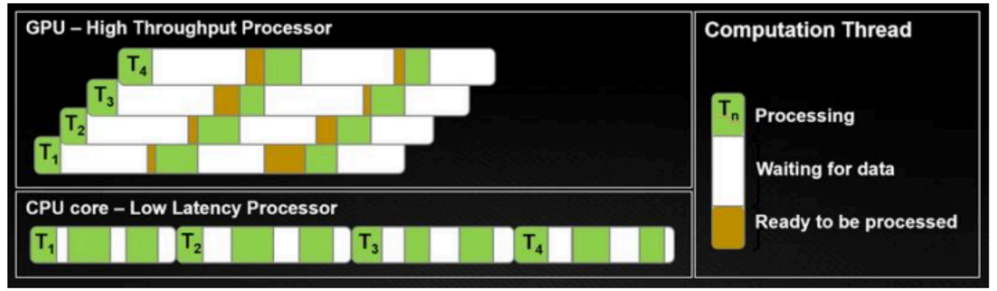
CPUs optimize for a few, fast threads while GPUs optimize for many many threads



CPU



GPU



Many tiny compute units (ALUs).
Much less support for branching (control, cache)

CPUs optimize for latency (each thread finishes quickly)
GPUs optimize for throughput (total processed data)

https://developer.nvidia.com/blog/cuda-refresher-reviewing-the-origins-of-gpu-computing/

Cores



SM



GA100 Full GPU with 128 SMs

Each SM further contains many **SPs (streaming processor)** that can execute 'threads' in parallel

SP is a "CUDA core", operate in lockstep

GPUs have *many* **SM (streaming multiprocessors)** that independently execute 'blocks' (jobs).

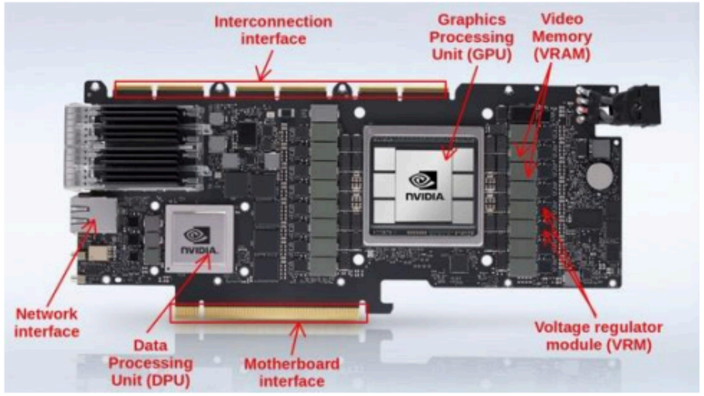
SM is a "core". Independent of others

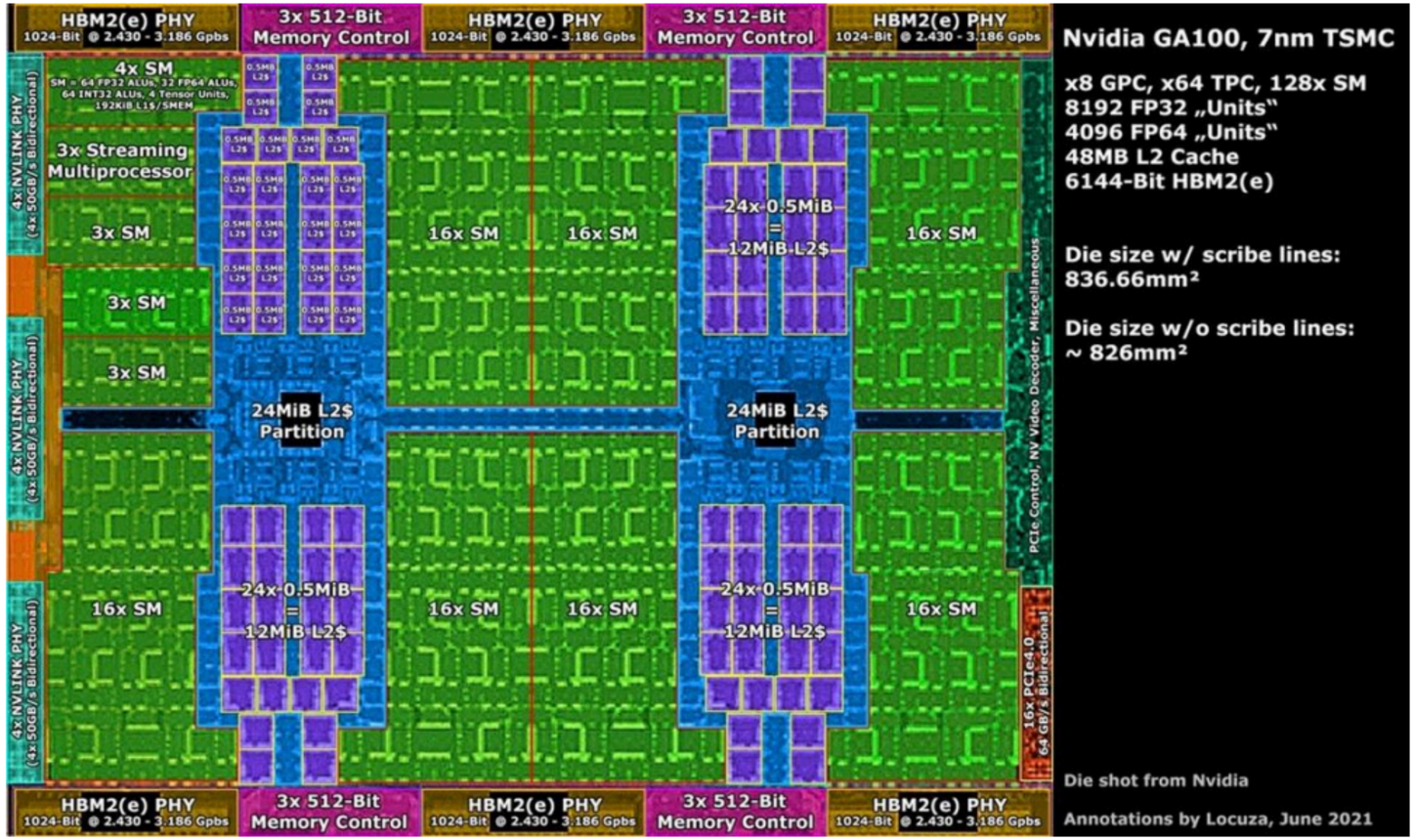
GPU Memory

The closer the memory to the SM, the faster it is – L1 and shared memory is *inside* the SM. L2 cache is on die, and global memory are the memory chips next to the GPU

TABLE IV
THE MEMORY ACCESSES LATENCIES

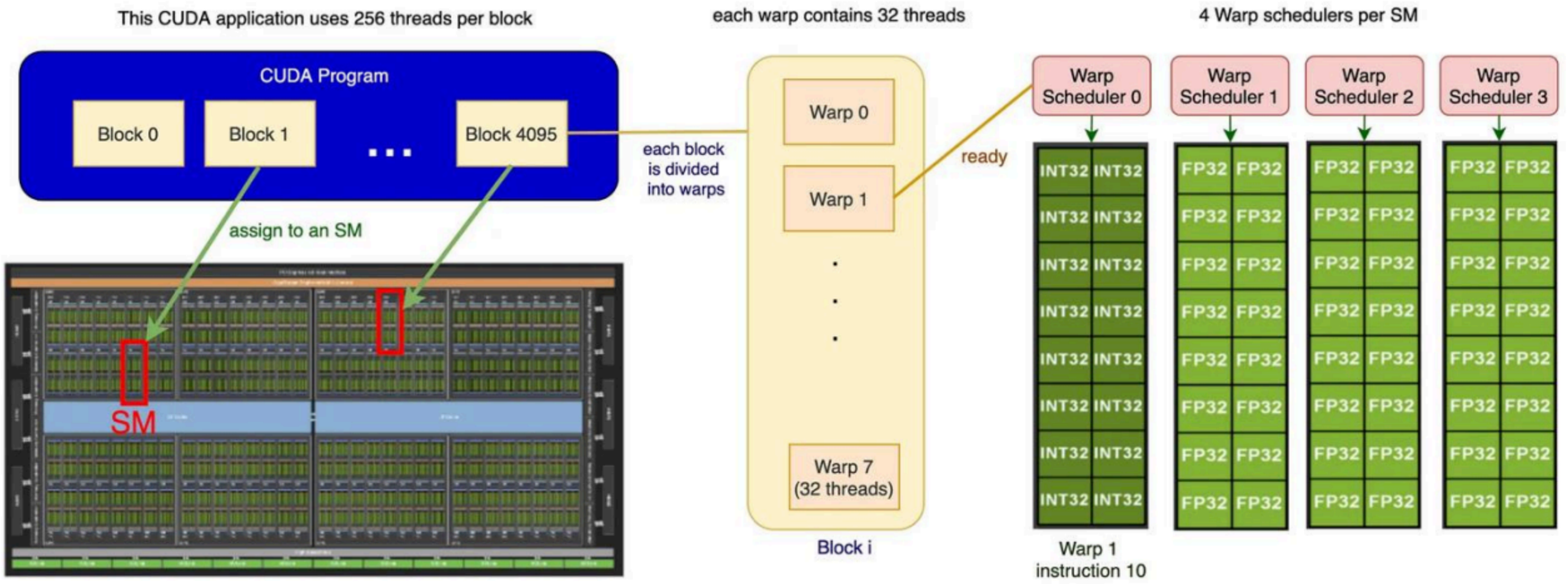
Memory type	*CPI (cycles)*
Global memory	290
L2 cache	200
L1 cache	33
Shared Memory (ld/st)	(23/19)





SRAM (shared/cache memory) is much more expensive (100x) but ~ 8x faster than DRAM (Global memory)

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.

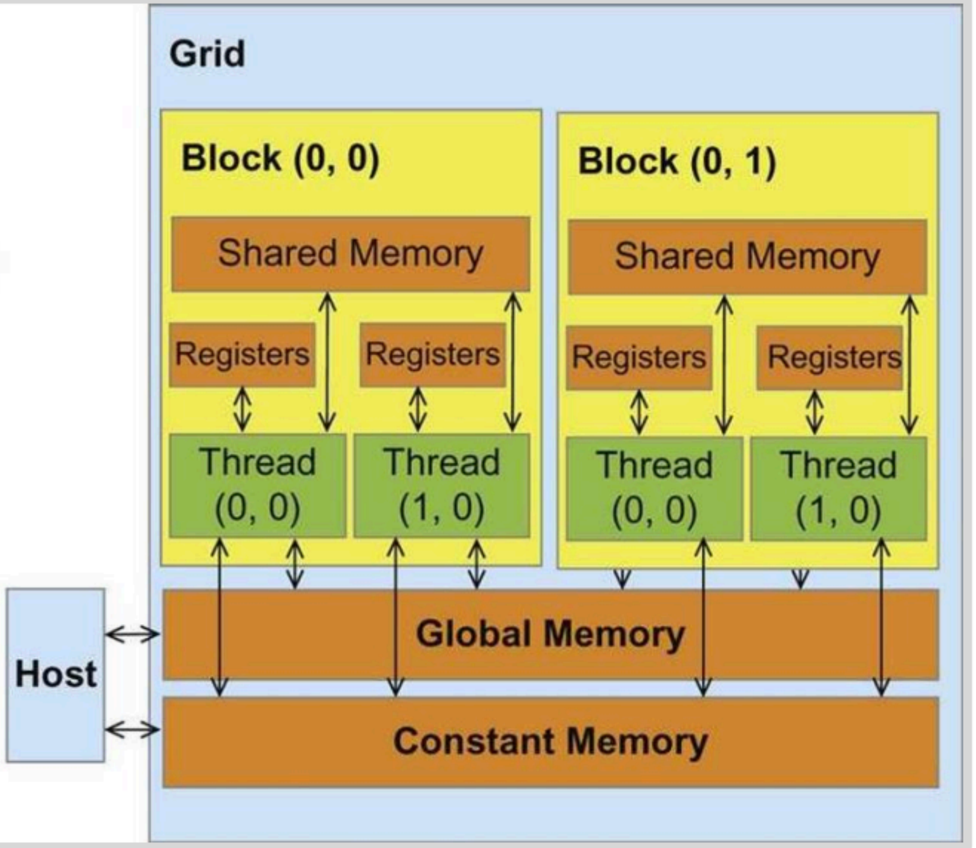
Memory Model

Device code can:

- R/W per-thread registers
- R/W per-thread local memory
- R/W per-block shared memory
- R/W per-grid global memory
- Read only per-grid constant memory

Host code can

- Transfer data to/from per grid global and constant memories

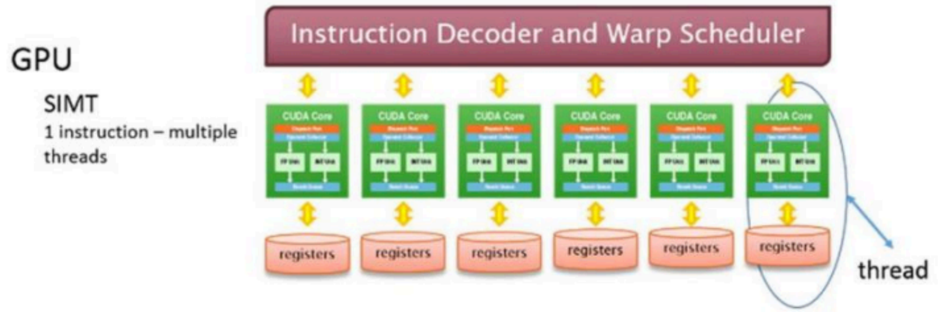


Each thread can access its own register, and shared memory within the block.

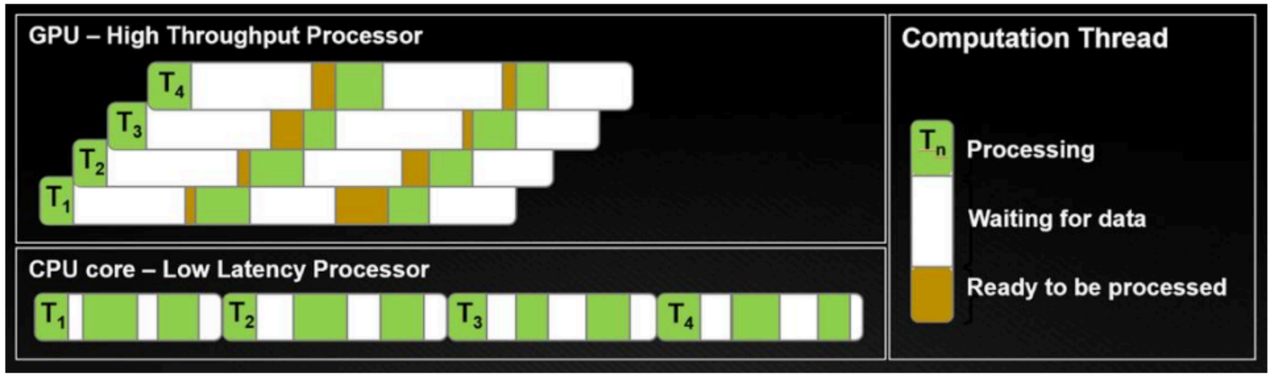
Information that goes across blocks need to be read/written to global memory (slow)

Strengths

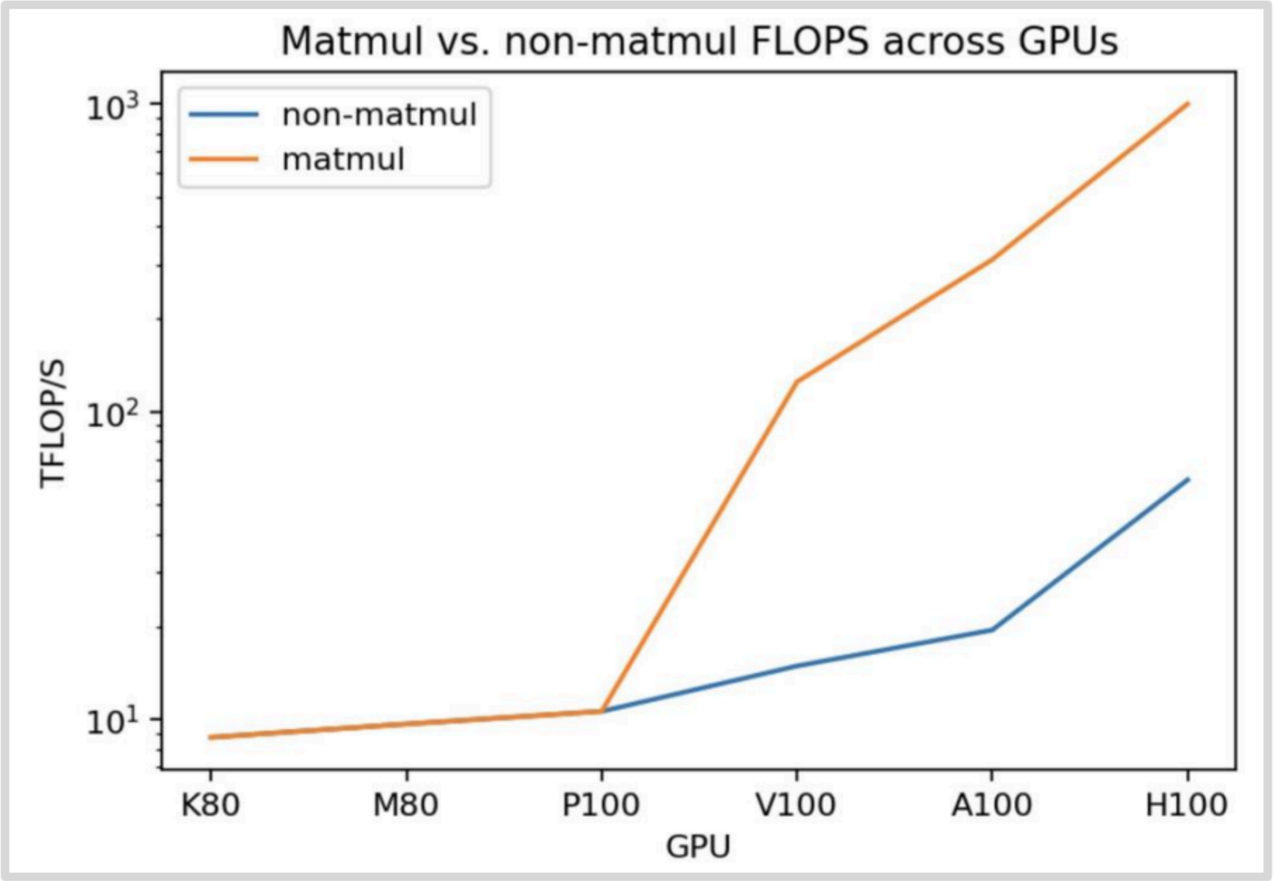
- Easily scales up hard workloads (by adding more SMs)
- Easy (?) to program due to the SIMT model



- Threads are ‘lightweight’ and can be stopped and started



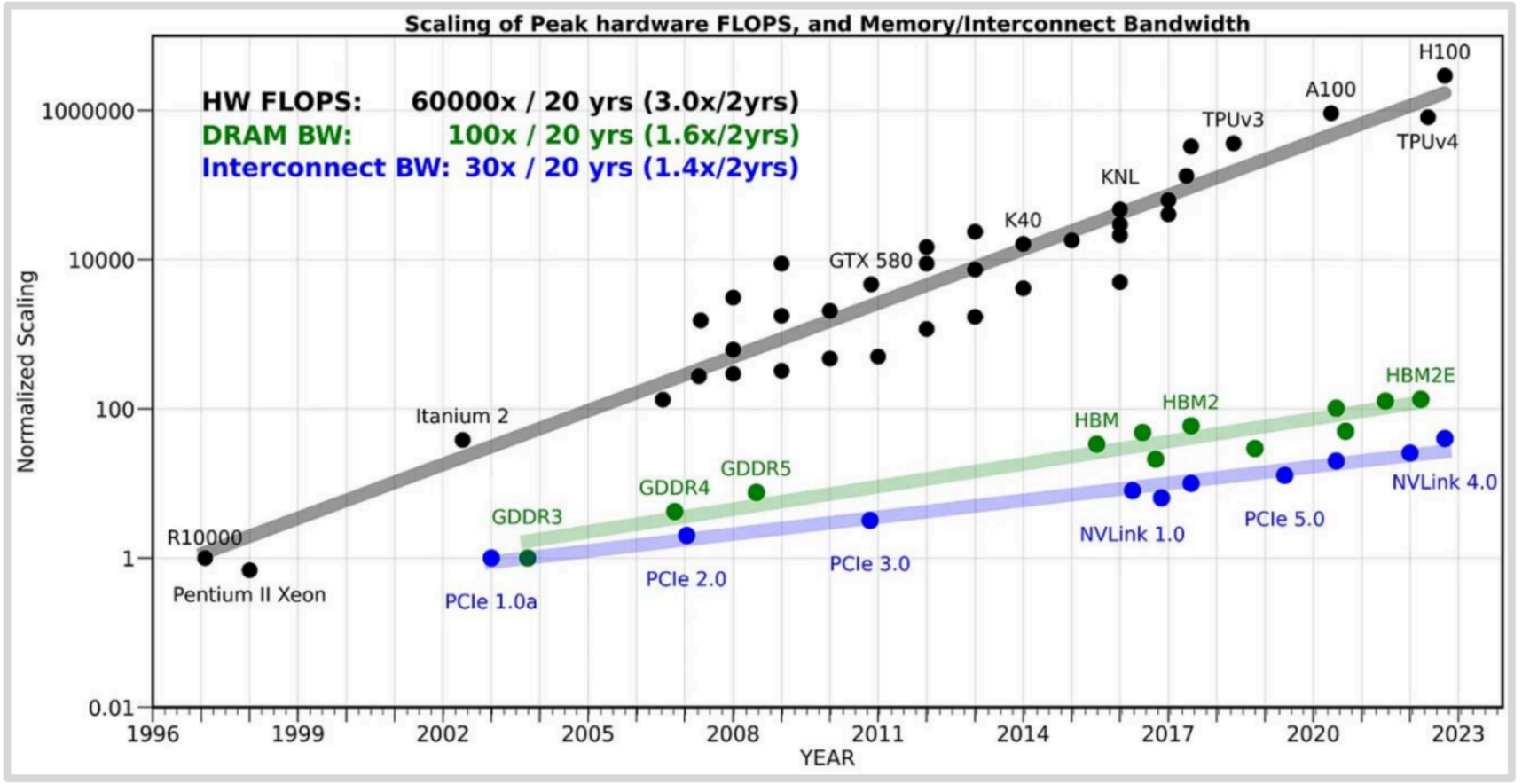
Matmuls



Tensor cores (introduced in V, T series) are specialized matrix multiplication circuits.

Matmuls are >10x faster than other floating point ops!

Scaling: FLOPs vs. Memory

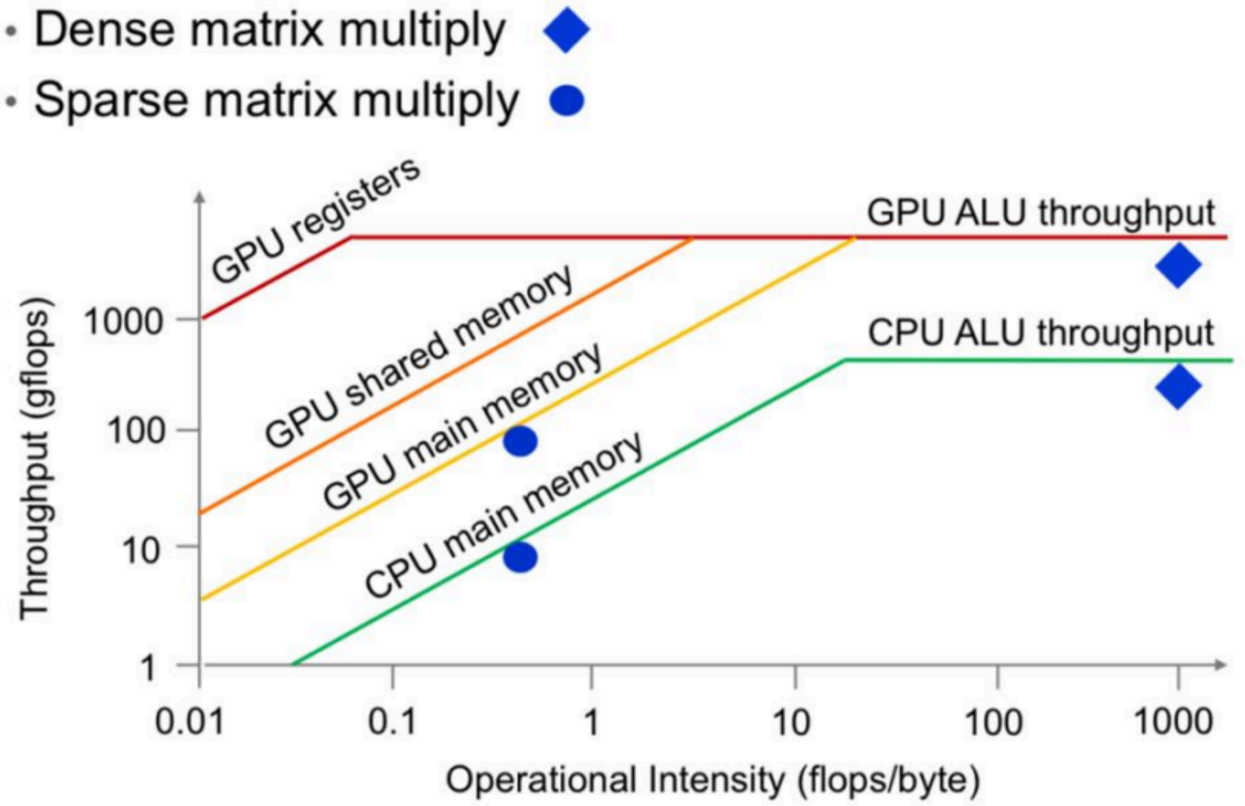


https://medium.com/riselab/ai-and-memory-wall-2cb4265cb0b8

FLOPs scale faster than memory – it's hard to keep our compute units fed with data!

FLOPs vs. Memory

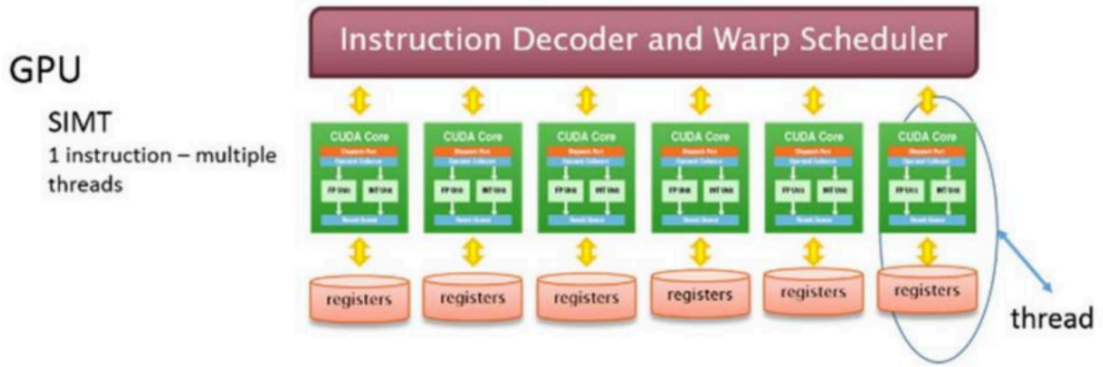
The roofline model



Key to this section: **how do we avoid being memory bound?**

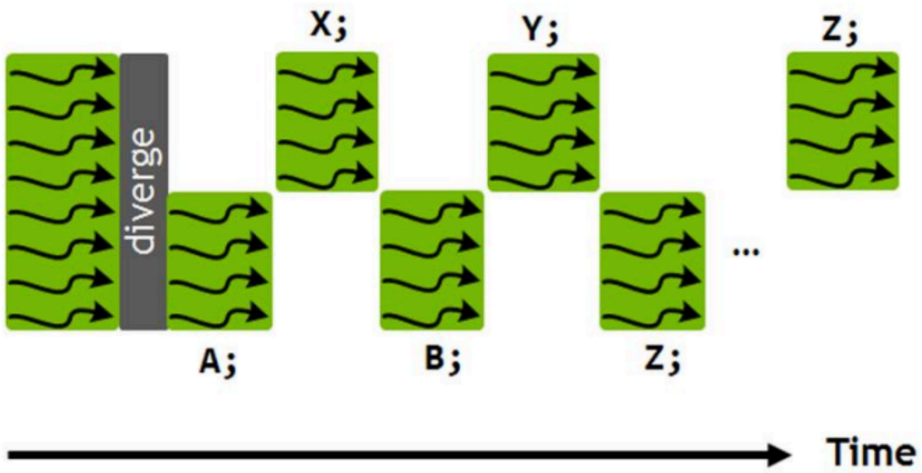
Execution

GPUs operate in a SIMT model – every thread in a warp is executing the same instruction



Conditionals are fine, but lead to significant overhead from the execution model

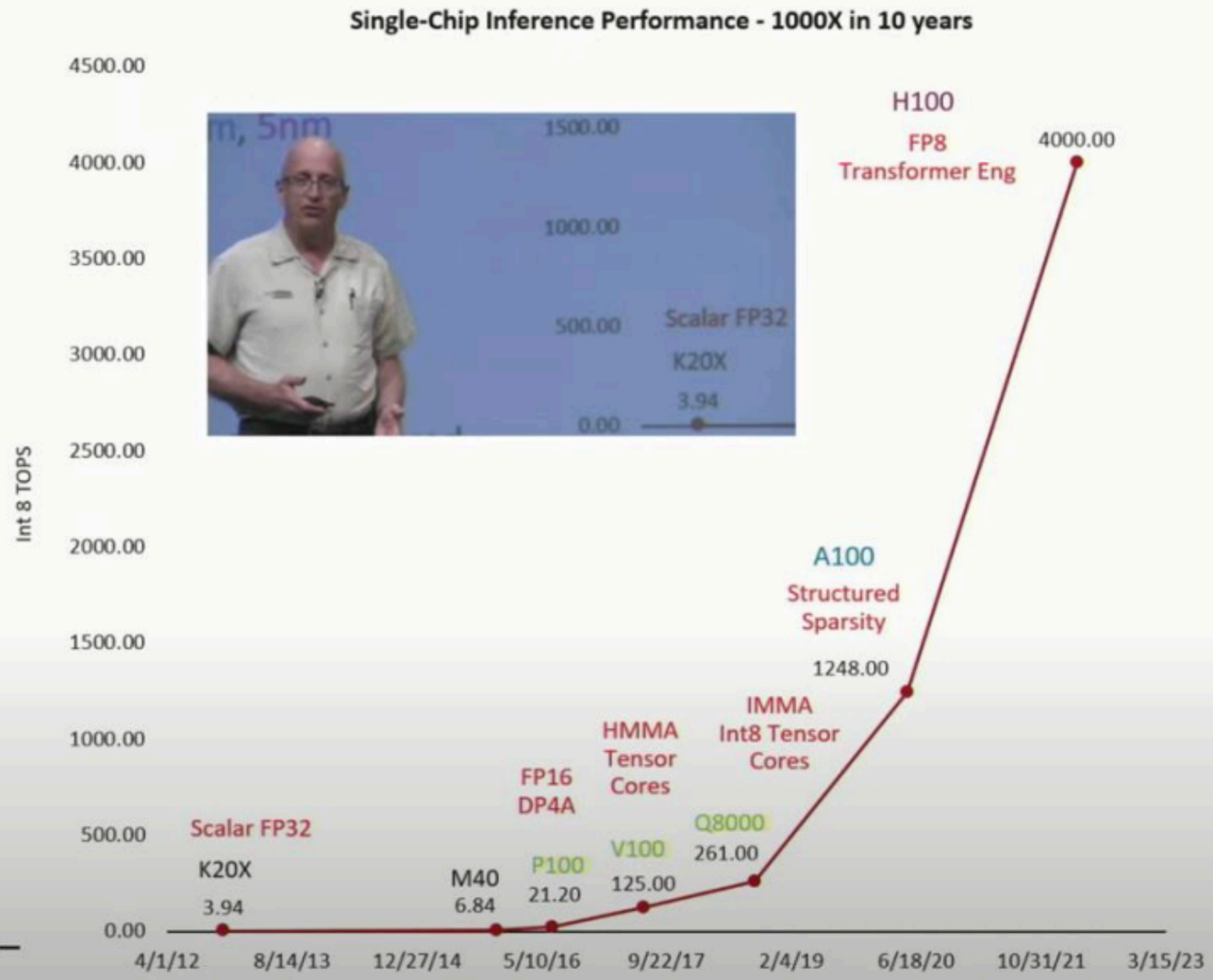
```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```



Speedups

Gains from

- Number Representation
  - FP32, FP16, Int8
  - (TF32, BF16)
  - ~16x
- Complex Instructions
  - DP4, HMMA, IMMA
  - ~12.5x
- Process
  - 28nm, 16nm, 7nm, 5nm
  - ~2.5x
- Sparsity
  - ~2x
- Model efficiency has also improved – overall gain > 1000x



Administrative details and recap

Efficiency

GPUs

GPU Principles

Profiling

Parallelism

GPU Principles

Slide credits for this section:
Tatsu Hashimoto / CS 336
(his credits: Horace He, CUDA Mode)

Five Tricks for GPUs

Trick 1: Low/mixed precision

Trick 2: Operator fusion

Trick 3: Recomputation

Trick 4: Memory coalescence

Trick 5: Tiling

Trick 1: Low/mixed precision

Low precision improves arithmetic intensity

Example: elementwise ReLU ($x = \max(0, x)$) on a vector of size n.

(Float 32 case)

Memory access: 1 read (x), 1 write (if x < 0), float 32 = 8 bytes

Operations: 1 comparison op, 1 FLOP.

Intensity: 8 bytes / FLOP

(Float 16)

Memory access: 1 read (x), 1 write (if x < 0), float 16 = 4 bytes

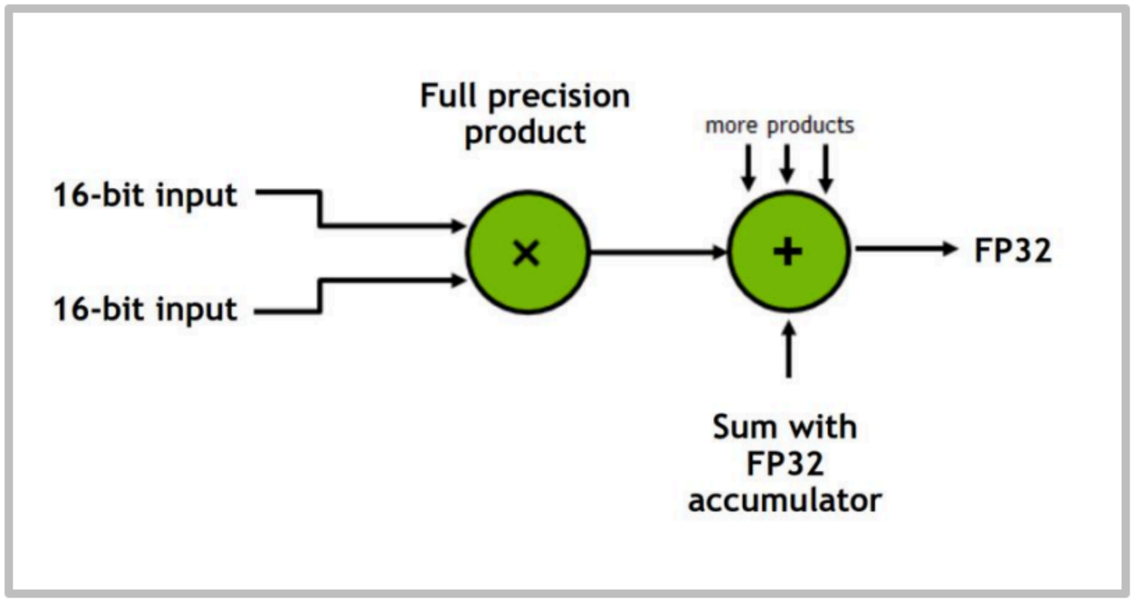
Operations: 1 comparison op, 1 FLOP.

Intensity: 4 bytes / FLOP

Trick 1: Low/mixed precision

Lots of operations in modern GPUs are accelerated via low / mixed precision operations

Tensor cores



Operations that can use 16-bit storage (FP16/BF16)

- Matrix multiplications
- Most pointwise operations (e.g. relu, tanh, add, sub, mul)

Operations that need more precision (FP32/FP16)

- Adding small values to large sums can lead to rounding errors
- Reduction operations (e.g. sum, softmax, normalization)

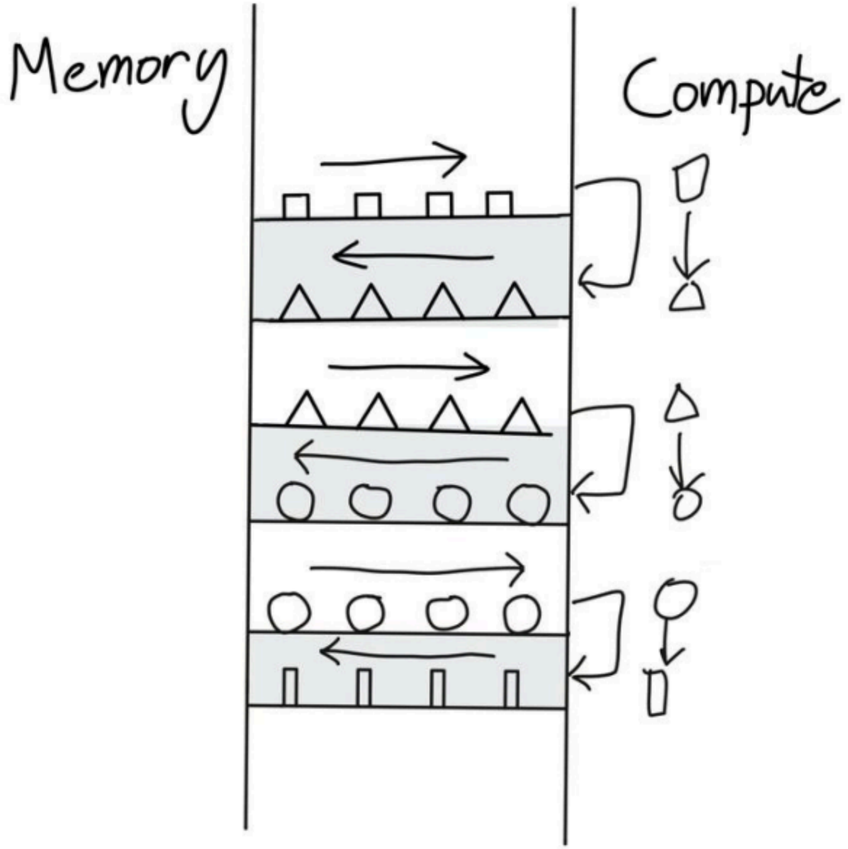
Operations that need more range (FP32/BF16)

- Pointwise operations where $|f(x)| \gg |x|$ (e.g. exp, log, pow)
- Loss functions

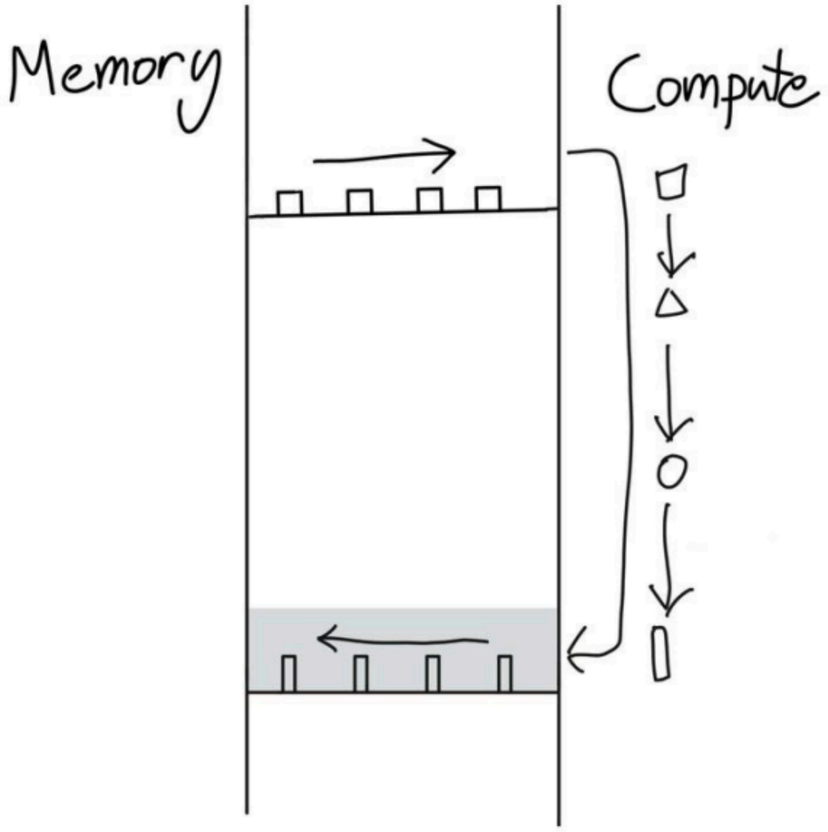
https://nvlabs.github.io/eccv2020-mixed-precision-tutorial/files/dusan_stosic-training-neural-networks-with-tensor-cores.pdf

Trick 2: Operator Fusion

What if we have to do many operations? Shipping back and forth is somewhat silly



Naïve (non-fused)



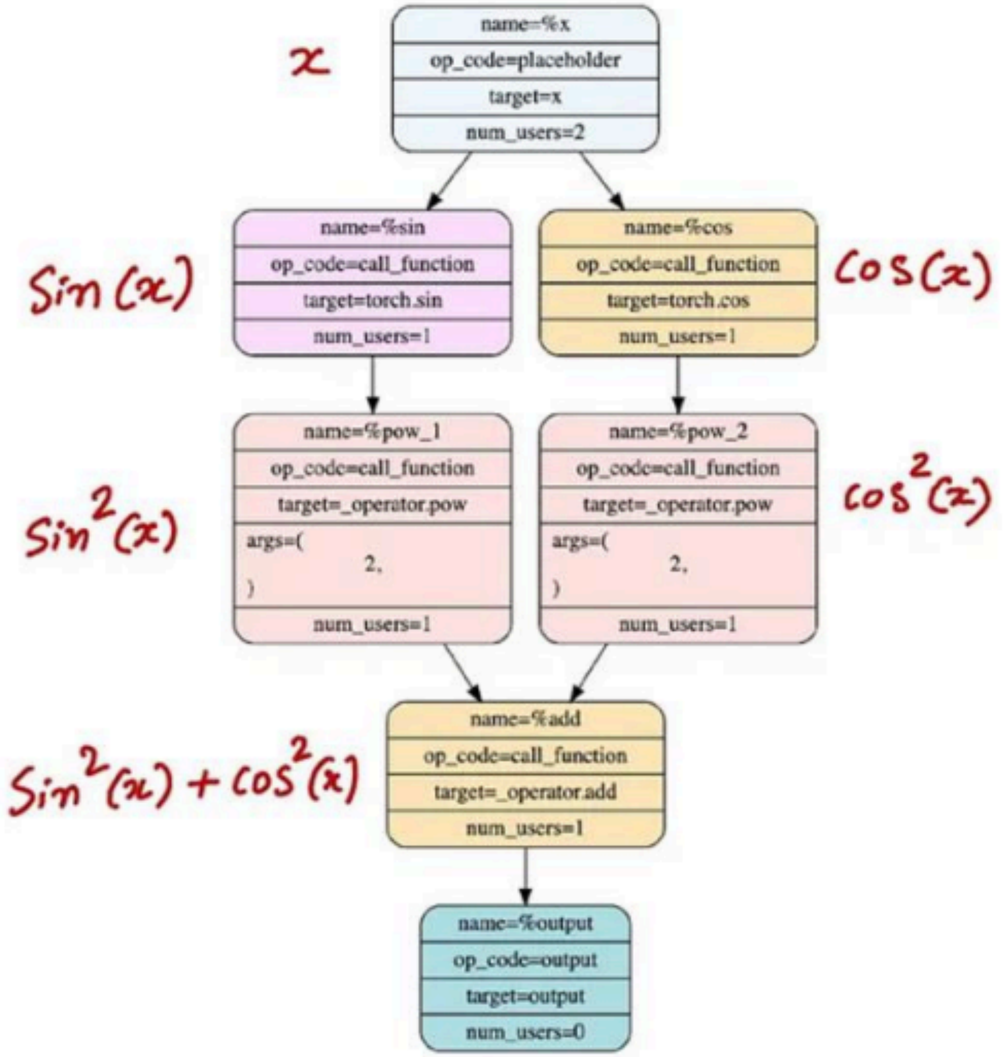
Fused kernel

Trick 2: Operator Fusion: Sines and Cosines

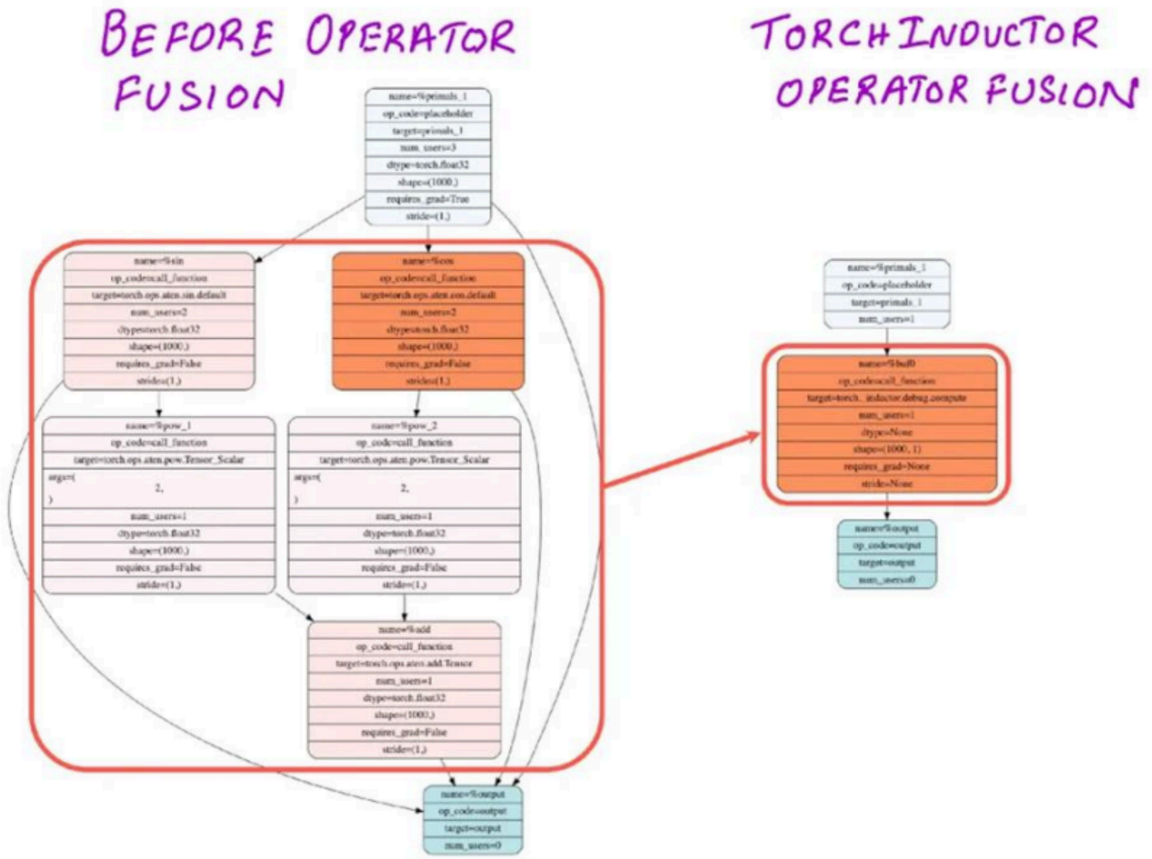
FX GRAPH IR

```
ass GraphModule(torch.nn.Module):
  def forward(self, x : torch.Tensor):
      # File: /tmp/ipykernel_2583/1502985755.py:2, code:
      sin = torch.sin(x)
      pow_1 = sin ** 2;  sin = None
      cos = torch.cos(x);  x = None
      pow_2 = cos ** 2;  cos = None
      add = pow_1 + pow_2;  pow_1 = pow_2 = None
      return (add,)
```

GRAPH VIZ



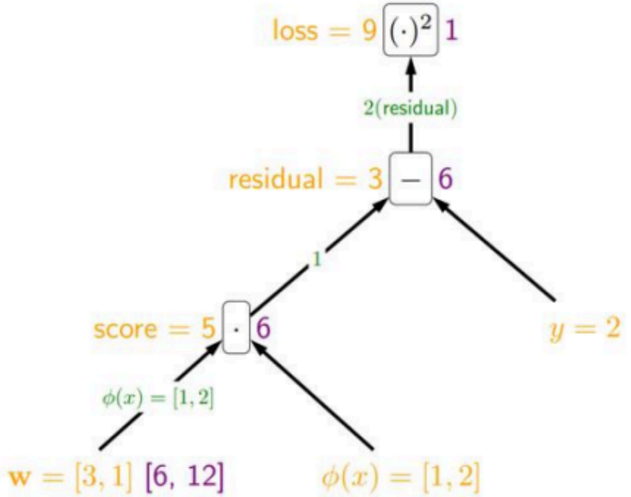
Trick 2: Operator Fusion: After



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 3: Recomputation



$$\text{Loss}(x, y, \mathbf{w}) = (\mathbf{w} \cdot \phi(x) - y)^2$$

$\mathbf{w} = [3, 1], \phi(x) = [1, 2], y = 2$

backpropagation

$\nabla_{\mathbf{w}} \text{Loss}(x, y, \mathbf{w}) = [6, 12]$

Definition: Forward/backward values

Forward: f_i is value for subexpression rooted at i

Backward: $g_i = \frac{\partial \text{loss}}{\partial f_i}$ is how f_i influences loss

Algorithm: backpropagation algorithm

Forward pass: compute each f_i (from leaves to root)

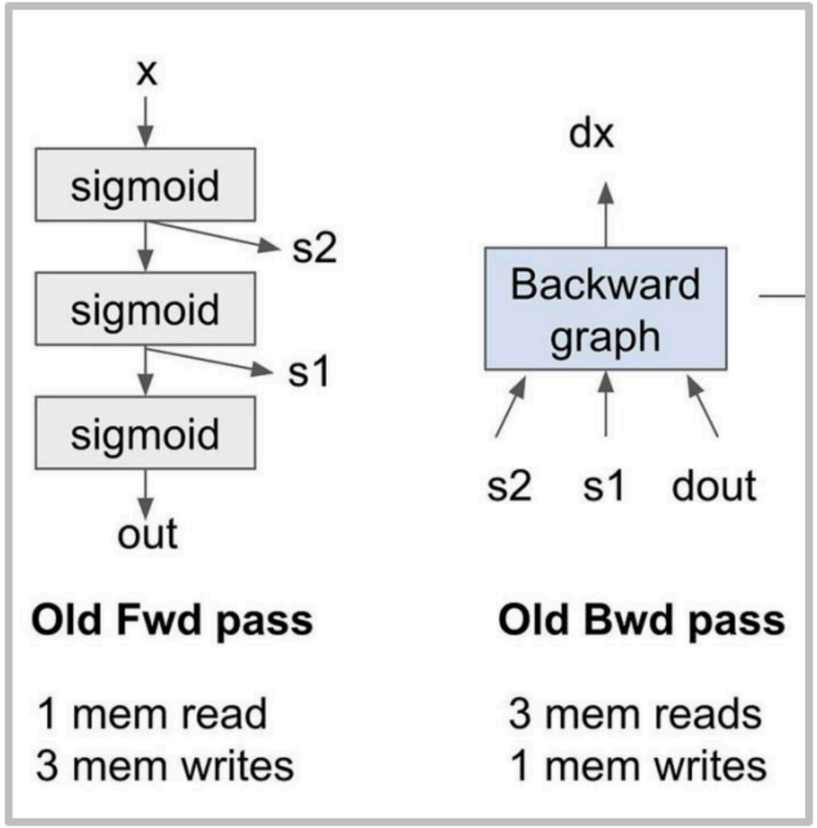
Backward pass: compute each g_i (from root to leaves)

[From cs221]

In backpropagation, we store the activations (yellow) and compute Jacobians (green)

Trick 3: Recomputation

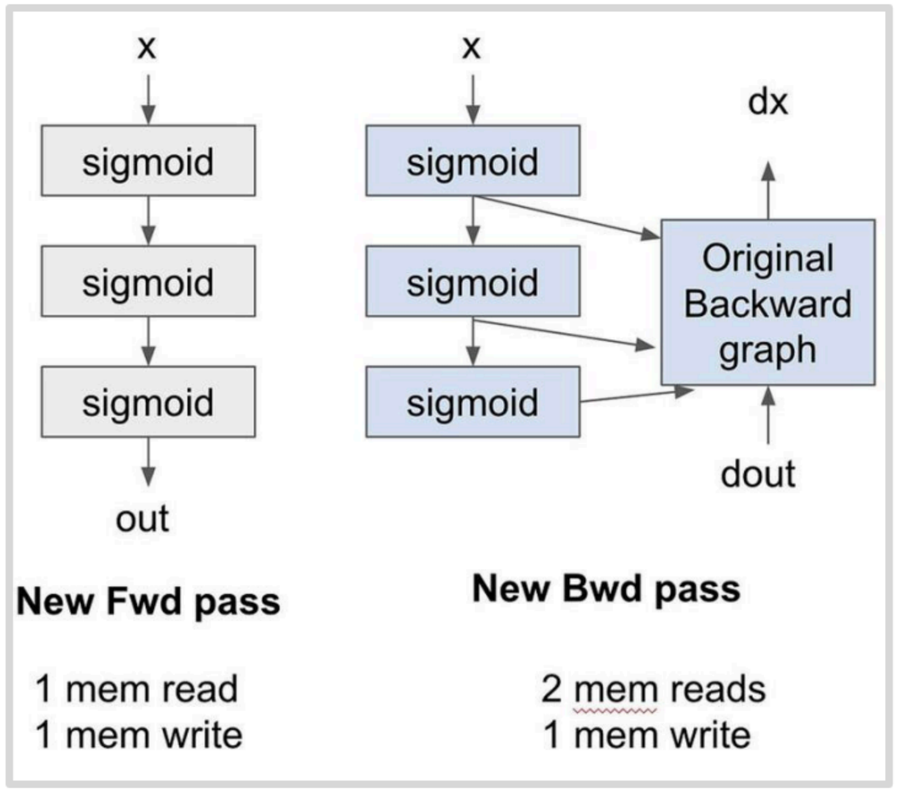
Let's say we stack 3 sigmoids on top of each other.



This is really terrible for perf – 8 mem read/writes, very low arithmetic intensity.

https://dev-discuss.pytorch.org/t/min-cut-optimal-recomputation-i-e-activation-checkpointing-with-aotautograd/467

Trick 3: Recomputation

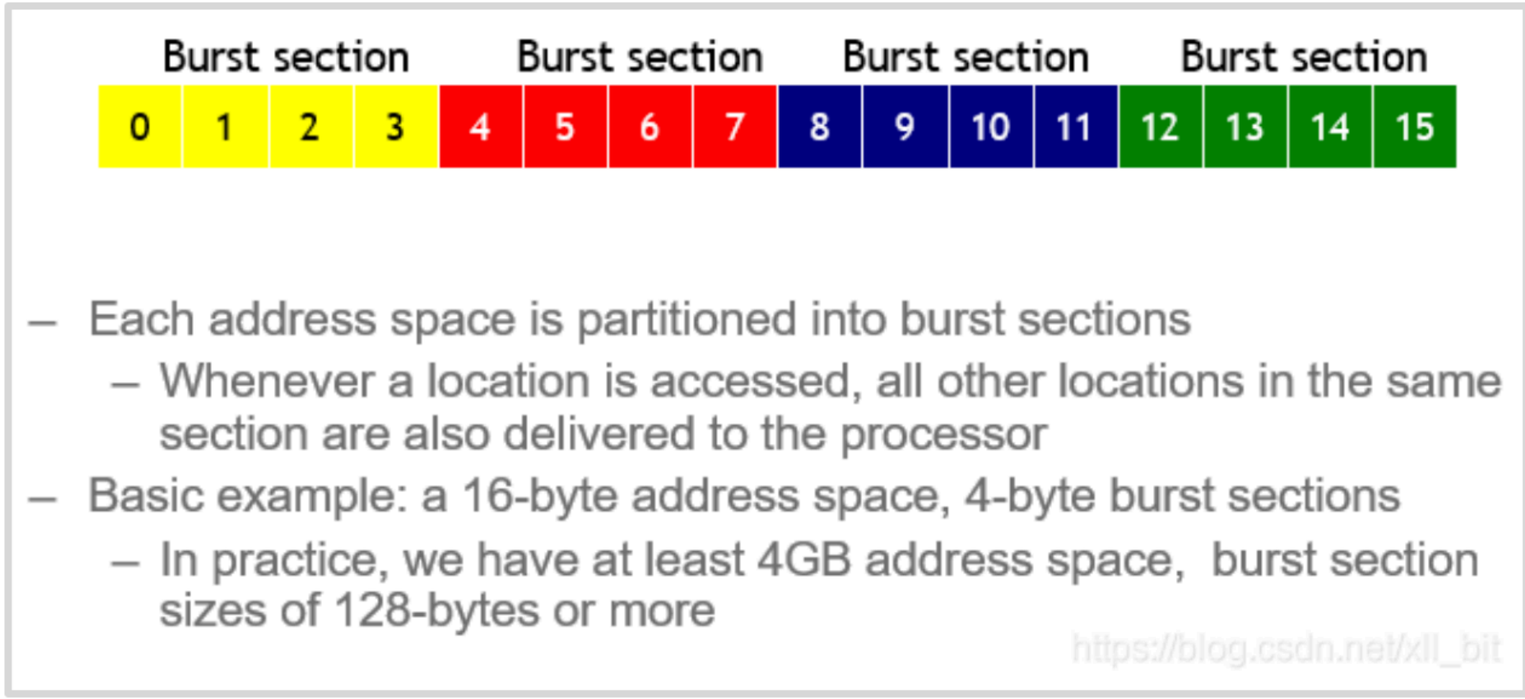


Throwing away computation can actually be optimal, w/ 5/8th the memory accesses!

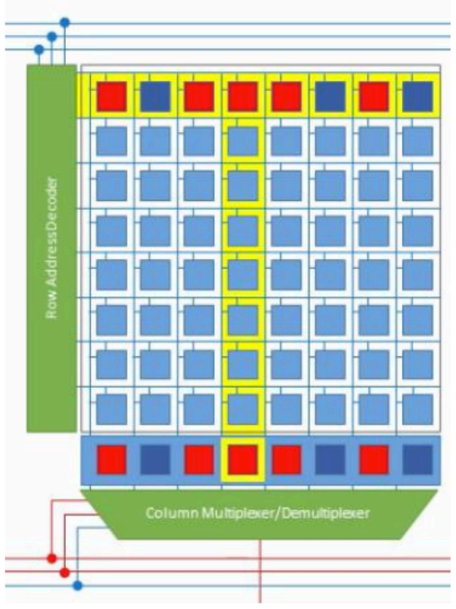
https://dev-discuss.pytorch.org/t/min-cut-optimal-recomputation-i-e-activation-checkpointing-with-aotautograd/467

Trick 4: Memory Coalescence

DRAM (global memory) is read in 'burst mode' – each read gives you many bytes!



[https://blog.csdn.net/xll_bit/article/details/117702476]

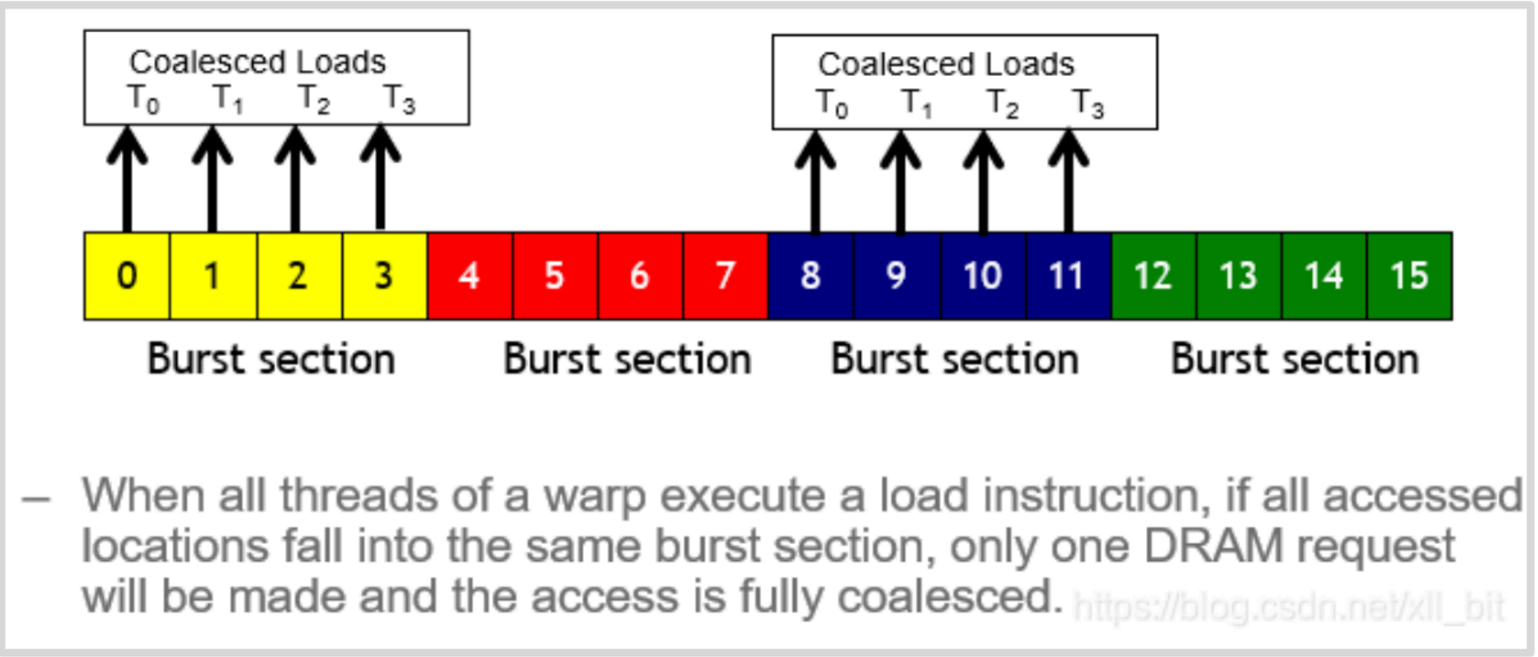


← Burst mode comes from the slow per-row copy to the sense amplifier

[https://www.youtube.com/watch?v=9BjVUmaXaCQ]

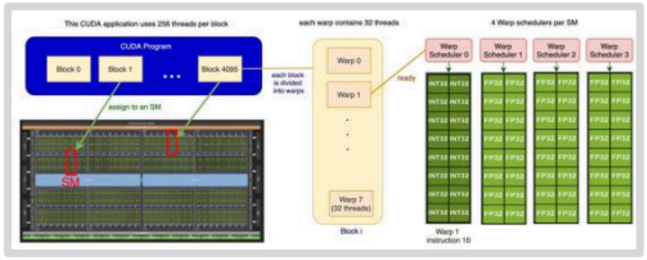
Trick 4: Memory Coalescence

Memory accesses are *coalesced* if all the threads (in a warp) fall within the same burst



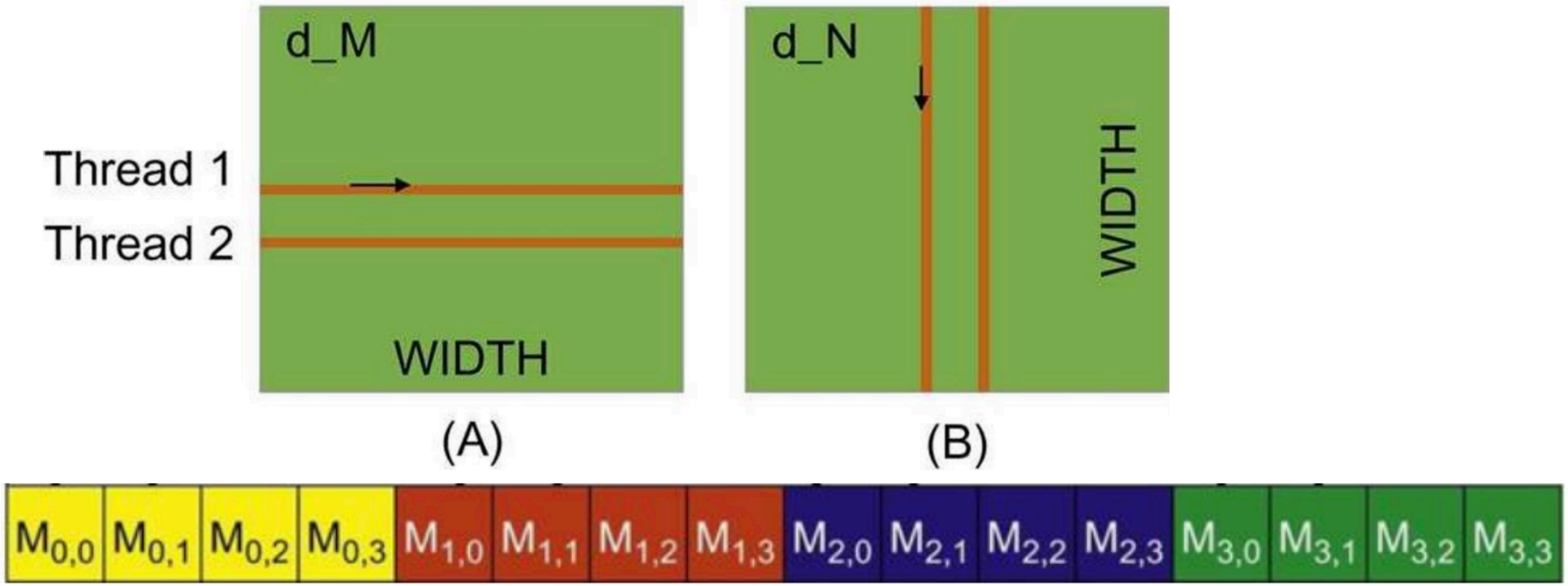
– When all threads of a warp execute a load instruction, if all accessed locations fall into the same burst section, only one DRAM request will be made and the access is fully coalesced.

https://blog.csdn.net/xll_bit



Reminder: a warp is a set of 32 consecutively numbered threads that execute together in a block. Memory accesses happen together

Trick 4: Memory Coalescence

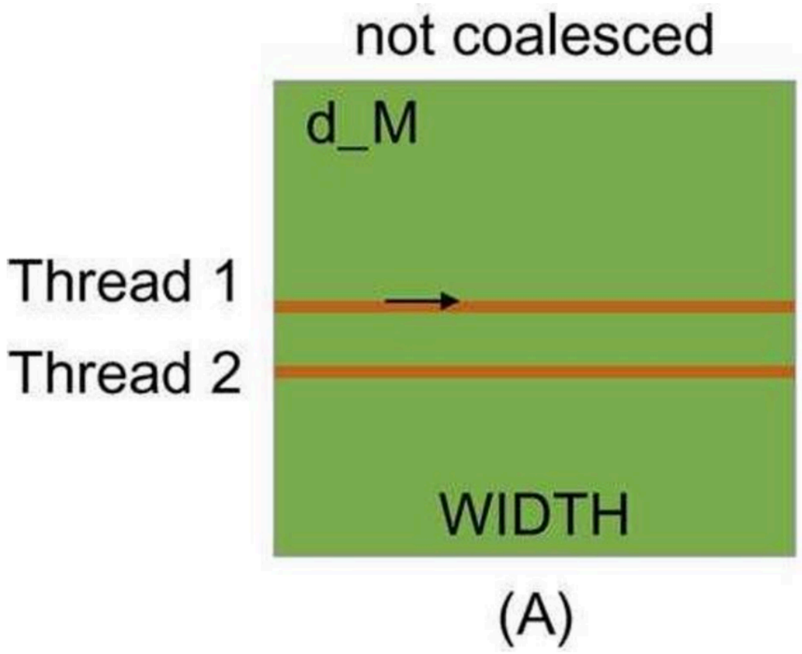


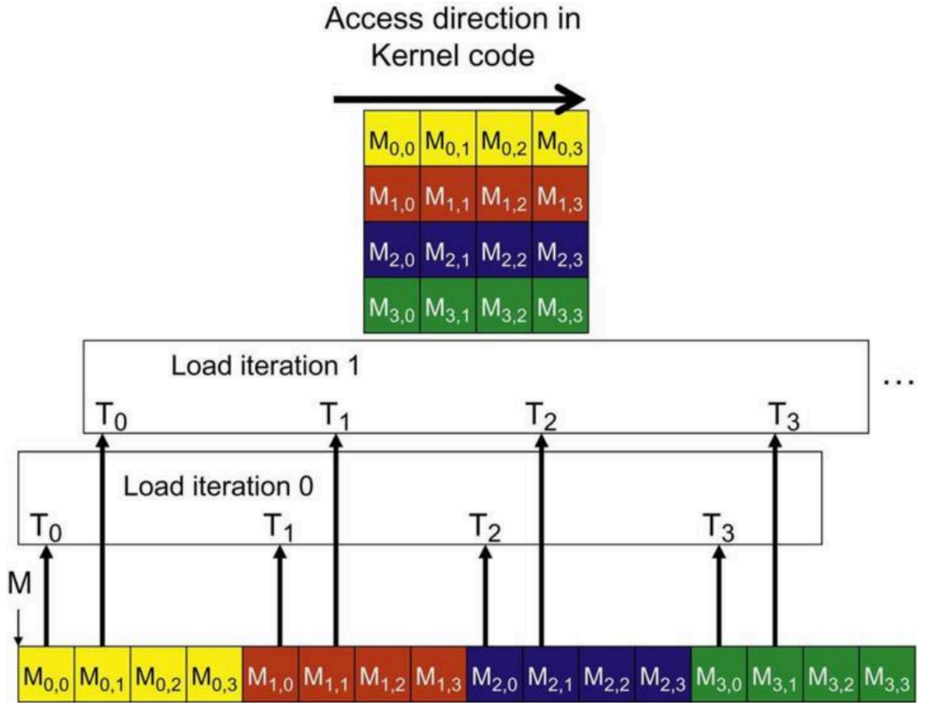
Suppose we have a row-major matrix: rows are stored contiguously in matrix

For an elementwise operation (e.g., sigmoid), should threads move over rows over columns?

Trick 4: Memory Coalescence

Row-major matrix in memory:

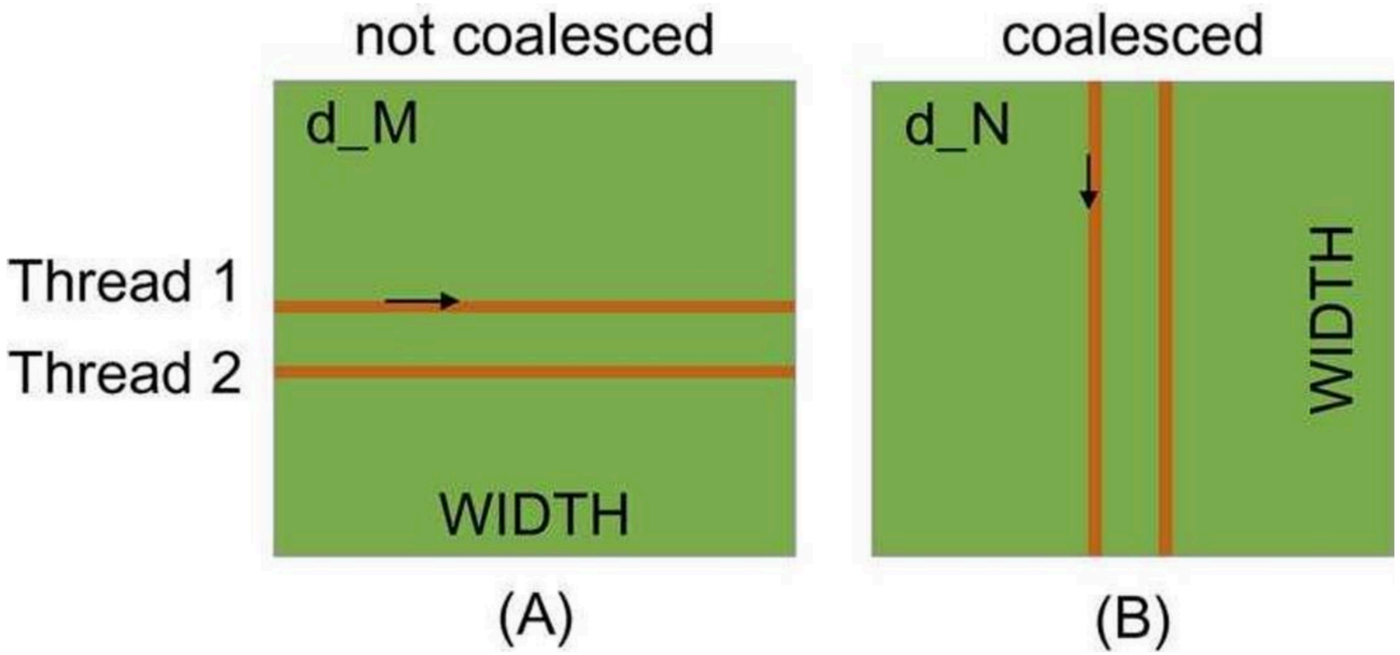


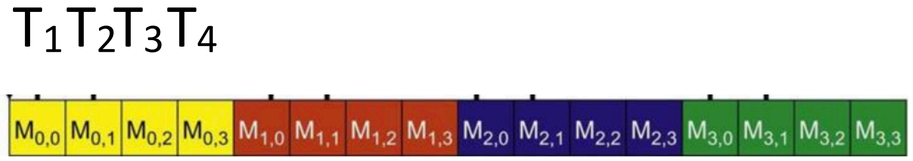


Reads entire vector every step

Trick 4: Memory Coalescence

Row-major matrix in memory:



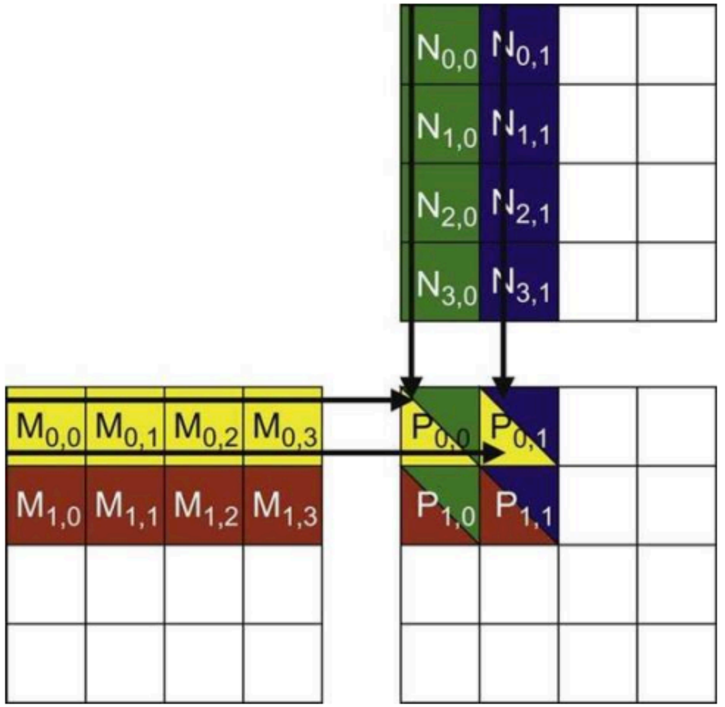


Moving along columns: coalesced. Moving along rows: not coalesced.

Trick 5: Tiling

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

Let's go back to matrix multiplication..



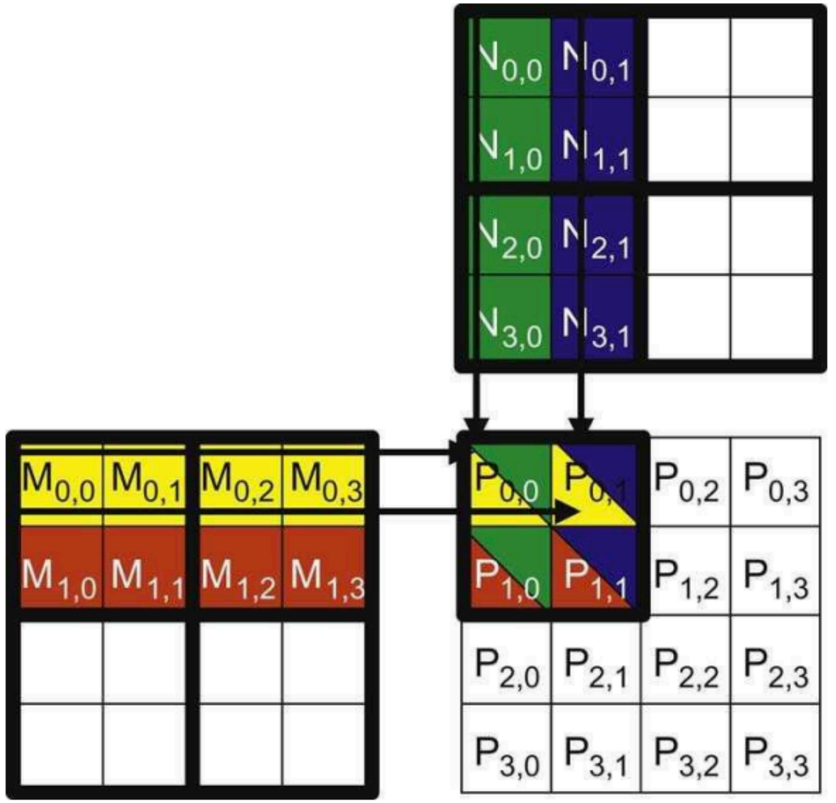
Access order

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

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



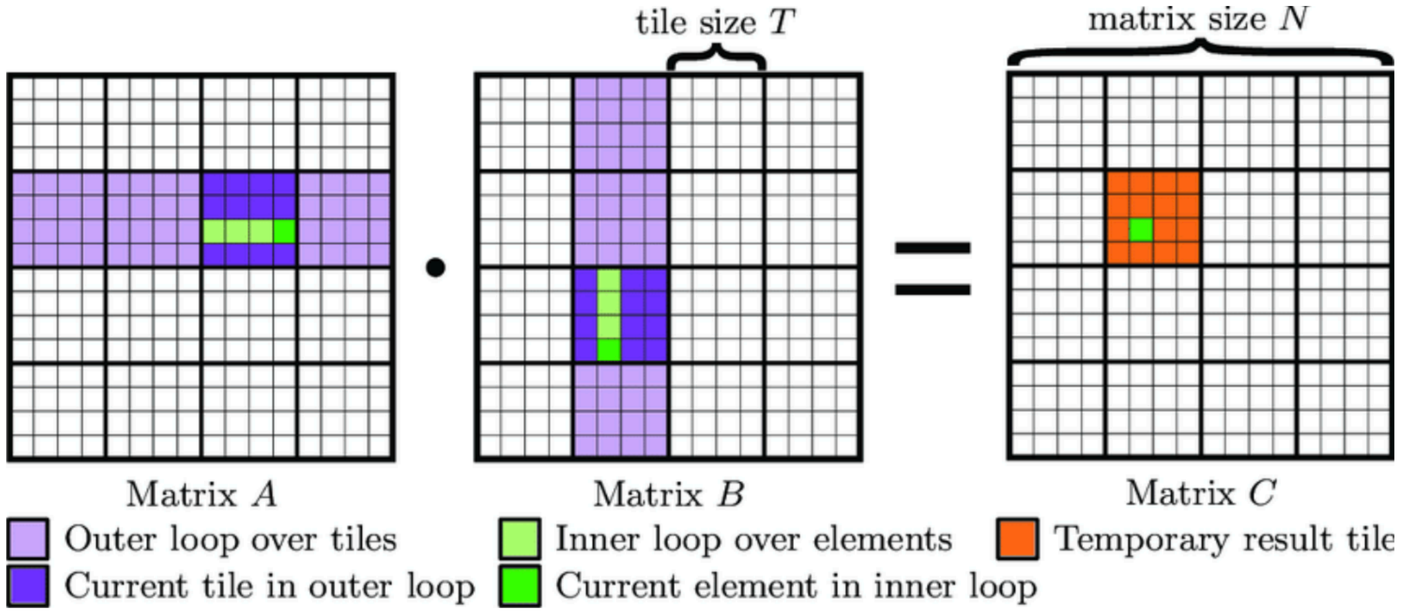
Compute the matrix multiply in ‘phases’

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

Trick 5: Tiling



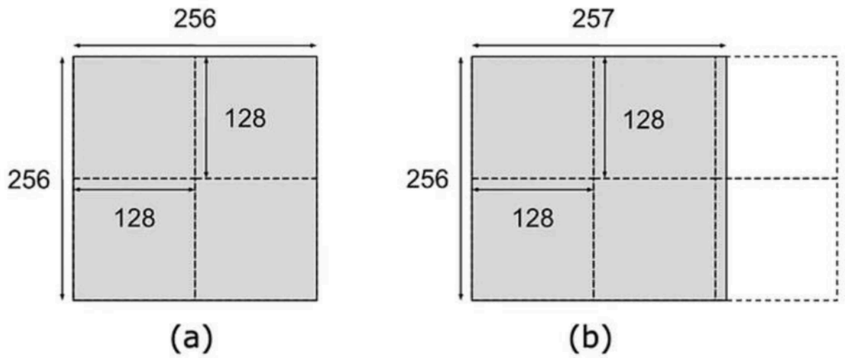
Non-tiled matrix multiply: each input is read N times from global memory

Tiled matrix multiply: each input is read $\frac{N}{T}$ times from global memory, and T times within each tile. This is a factor of T reduction in global memory access

Trick 5: Tiling

Tile sizes may not divide the matrix size and lead to low utilization

Figure 6. Example of tiling with 128x128 thread block tiles. (a) Best case - matrix dimensions are divisible by tile dimensions (b) Worse case - tile quantization results in six thread blocks being launched, two of which waste most of their work.



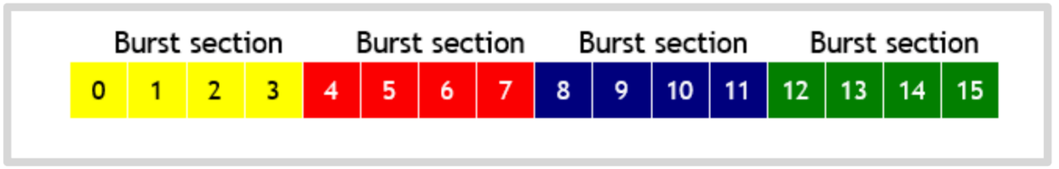
Factors affecting tile sizes

- Coalesced memory access
- Shared memory size
- Divisibility of the matrix dim

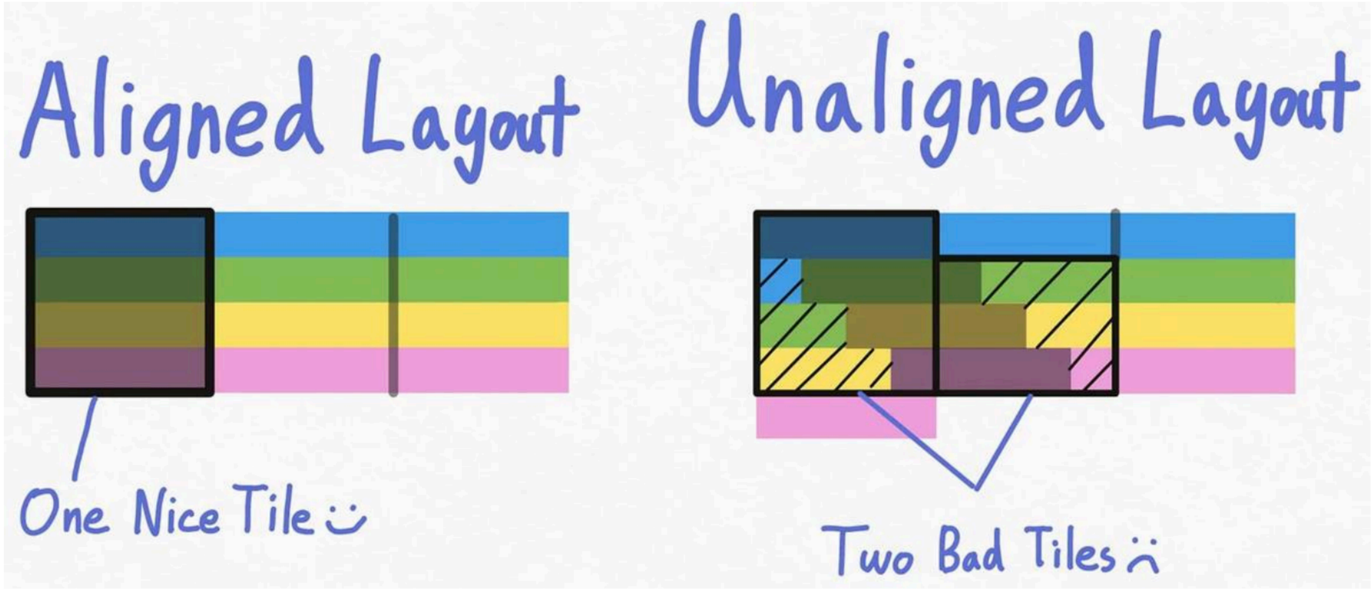
https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html#tile-quant

Trick 5: Tiling

Memory comes in bursts



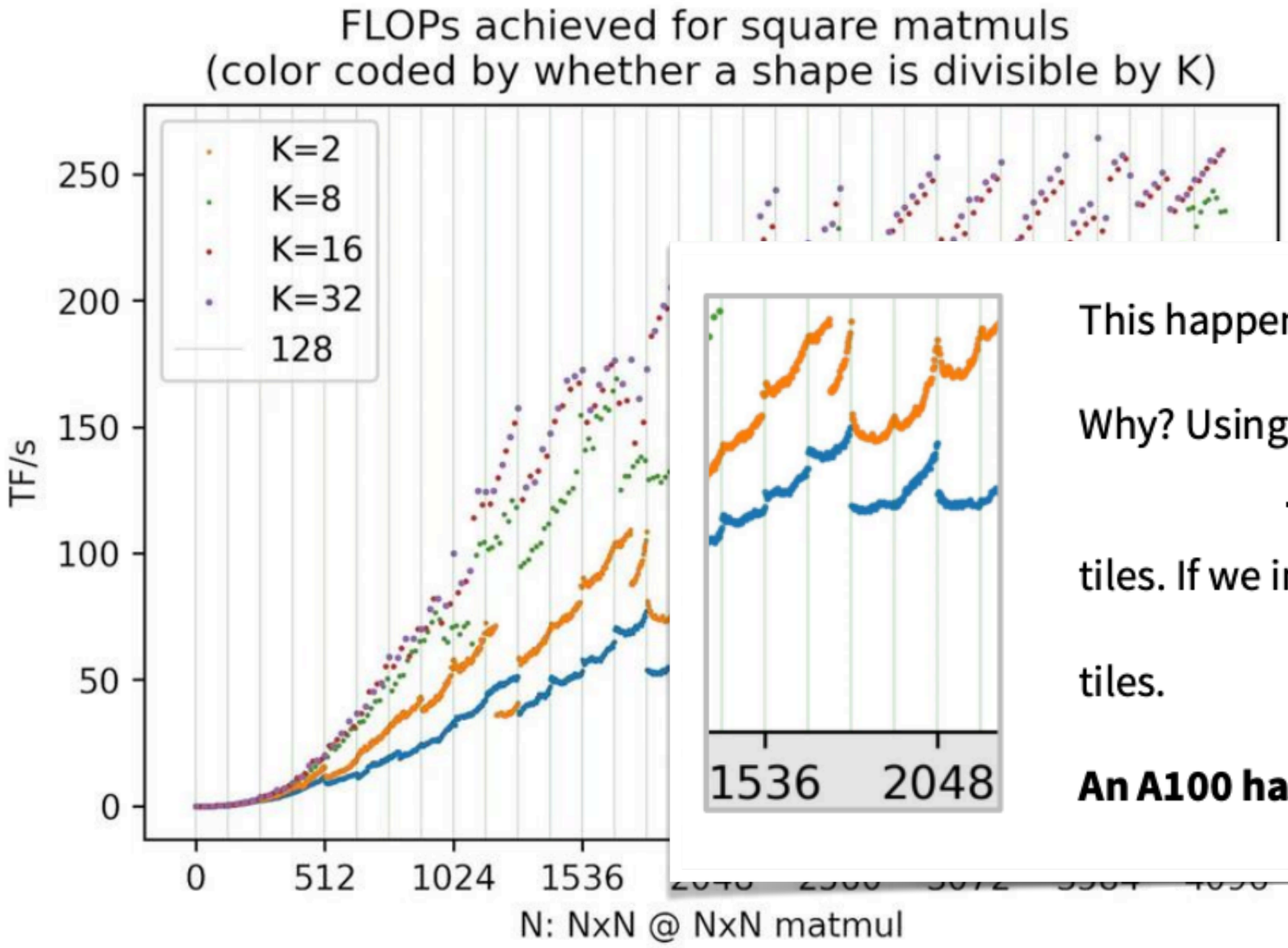
Loading tiles are fast if bursts align with the matrix



https://www.thonking.ai/p/what-shapes-do-matrix-multiplications

Coalesced accesses may be impossible depending on the dimension of the matrix..
(have to do padding)

Impact



This happens at 1792 to 1793 size.

Why? Using a tile size of 256×128, there are

$$\frac{1792}{256} \times \frac{1792}{128} = 7 \times 14 = 98$$

tiles. If we increase this to 1793, we have

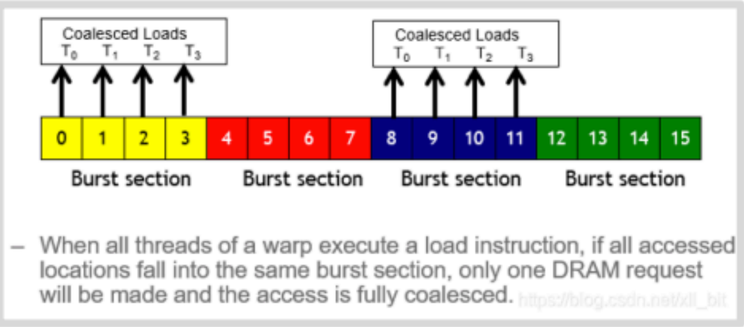
$$8 \times 15 = 120$$

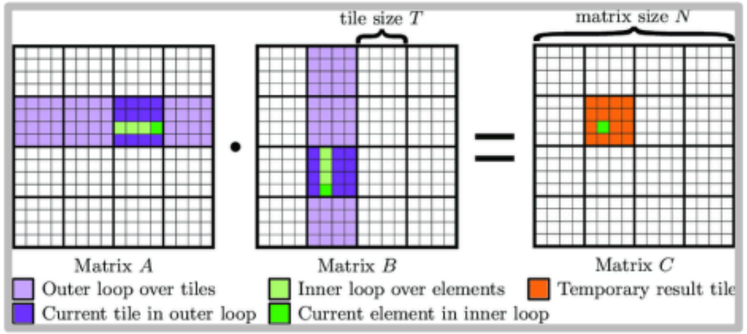
tiles.

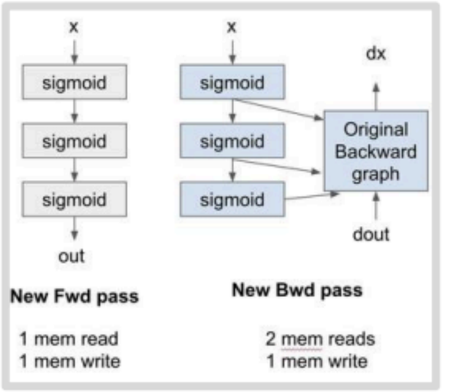
An A100 has 108 SMs, so it cannot execute all 120

GPU Principles

- Reduce memory accesses
  - Coalescing
  - Fusion
- Move memory to shared memory
  - Tiling
- Trade memory for compute/accuracy
  - Quantization
  - Recomputation







Administrative details and recap

Efficiency

GPUs

GPU Principles

Profiling

Parallelism

Profiling

Code examples: credit to Percy Liang

Code available with Assignment 2

Benchmarking and Profiling

- Many of these “mysterious” GPU behaviors can be understood better with profiling
- Benchmarking: figuring out how long it takes
- Profiling: figuring out where the time is being spent

Warmup: Benchmarking Matmul

```python
def benchmark(description: str, run: Callable, num_warmups: int = 1, num_trials: int = 3) -> float:
    """Benchmark `run` by running it `num_trials` times, return mean time in ms."""
    # Warmup
    for _ in range(num_warmups):
        run()
    if torch.cuda.is_available():
        torch.cuda.synchronize()

    # Time it
    times: list[float] = []
    for trial in range(num_trials):
        start_time = time.time()
        run()
        if torch.cuda.is_available():
            torch.cuda.synchronize()
        end_time = time.time()
        times.append((end_time - start_time) * 1000)

    mean_time = mean(times)
    print(f"{description}: {mean_time:.2f} ms (over {num_trials} trials)")
    return mean_time
```

Warmup: Benchmarking Matmul

- Define two random matrices and then multiply them
- Dims: 1024, 2048, 4096, 8192, 16384
- What trend do you expect?



1024: 0.22 ms, 9.80 TFLOPS

2048: 1.28 ms, 13.40 TFLOPS

4096: 9.32 ms, 14.74 TFLOPS

8192: 59.71 ms, 18.41 TFLOPS

16384: 462.52 ms, 19.02 TFLOPS

Device: NVIDIA A100-SXM4-40GB

Multiprocessors: 108

Warmup: Benchmarking MLP

```python
class MLP(nn.Module):  1 usage
    """Simple MLP: linear -> GeLU -> linear -> GeLU -> ... -> linear -> GeLU"""
    def __init__(self, dim: int, num_layers: int):
        super().__init__()
        self.layers = nn.ModuleList([nn.Linear(dim, dim) for _ in range(num_layers)])

    def forward(self, x: torch.Tensor):
        for layer in self.layers:
            x = layer(x)
            x = torch.nn.functional.gelu(x)
        return x
```

Warmup: Benchmarking MLP

- Base config: dim=256, layers=4, batch=256, steps=2
- Scaling steps: what do we expect?

```
--- Scaling number of steps ---
run_mlp(2x num_steps): 3.81 ms (over 3 trials)
  Expected ~2x, actual ~1.81x
run_mlp(3x num_steps): 5.71 ms (over 3 trials)
  Expected ~3x, actual ~2.71x
run_mlp(4x num_steps): 7.56 ms (over 3 trials)
  Expected ~4x, actual ~3.59x
run_mlp(5x num_steps): 9.43 ms (over 3 trials)
  Expected ~5x, actual ~4.47x
```

Warmup: Benchmarking MLP

- Base config: dim=256, layers=4, batch=256, steps=2
- Scaling steps: what do we expect? Linear increase.
- Scaling layers: what do we expect?

```
--- Scaling number of layers ---
run_mlp(2x num_layers): 3.40 ms (over 3 trials)
  Expected ~2x, actual ~1.61x
run_mlp(3x num_layers): 4.79 ms (over 3 trials)
  Expected ~3x, actual ~2.27x
run_mlp(4x num_layers): 6.20 ms (over 3 trials)
  Expected ~4x, actual ~2.94x
run_mlp(5x num_layers): 7.70 ms (over 3 trials)
  Expected ~5x, actual ~3.65x
```

Warmup: Benchmarking MLP

- Base config: dim=256, layers=4, batch=256, steps=2
- Scaling steps: what do we expect? Linear increase.
- Scaling layers: what do we expect? Linear, but lower than expected.
- Scaling batch size: what do we expect?

```
--- Scaling batch size ---
run_mlp(2x batch_size): 1.94 ms (over 3 trials)
  Expected ~2x, actual ~0.92x
run_mlp(3x batch_size): 1.97 ms (over 3 trials)
  Expected ~3x, actual ~0.93x
run_mlp(4x batch_size): 2.02 ms (over 3 trials)
  Expected ~4x, actual ~0.96x
run_mlp(5x batch_size): 2.03 ms (over 3 trials)
  Expected ~5x, actual ~0.96x
```

Warmup: Benchmarking MLP

- Base config: dim=256, layers=4, batch=256, steps=2
- Scaling steps: what do we expect? Linear increase.
- Scaling layers: what do we expect? Linear, but lower than expected.
- Scaling batch size: what do we expect? Constant due to parallelism
- Scaling dimension: what do we expect?

```
--- Scaling dimension ---
run_mlp(2x dim): 1.92 ms (over 3 trials)
  Expected ~4x (quadratic), actual ~0.91x
run_mlp(3x dim): 1.99 ms (over 3 trials)
  Expected ~9x (quadratic), actual ~0.94x
run_mlp(4x dim): 2.04 ms (over 3 trials)
  Expected ~16x (quadratic), actual ~0.97x
run_mlp(5x dim): 1.97 ms (over 3 trials)
  Expected ~25x (quadratic), actual ~0.93x
```

Warmup: Benchmarking MLP

- Base config: dim=256, layers=4, batch=256, steps=2
- Scaling steps: what do we expect? Linear increase.
- Scaling layers: what do we expect? Linear, but lower than expected.
- Scaling batch size: what do we expect? Constant due to parallelism
- Scaling dimension: what do we expect? Constant due to parallelism

- How can we tell more specifically where the time is going?

Profiling

```python
def profile(description: str, run: Callable, num_warmups: int = 1, with_stack: bool = False) -> str:
    """Profile a function and return the profiler table."""
    # Warmup
    for _ in range(num_warmups):
        run()
    if torch.cuda.is_available():
        torch.cuda.synchronize()

    # Run with profiler
    with torch.profiler.profile(
            activities=[ProfilerActivity.CPU, ProfilerActivity.CUDA],
            with_stack=with_stack,
            experimental_config=torch._C._profiler._ExperimentalConfig(verbose=True)) as prof:
        run()
        if torch.cuda.is_available():
            torch.cuda.synchronize()

    # Get table
    table = prof.key_averages().table(
        sort_by="cuda_time_total",
        max_name_column_width=80,
        row_limit=10
    )
```

Profiling: element-wise addition 2048

```
-------------------------------------------------------------------------------
                                                                         Name
-------------------------------------------------------------------------------
                                                                    aten::add
void at::native::vectorized_elementwise_kernel<4, at::native::CUDAFunctor_add...
                                                                cudaLaunchKernel
                                                            cudaDeviceSynchronize
-------------------------------------------------------------------------------
```

Self CPU %	Self CPU	CPU total %	CPU total	CPU time avg	Self CUDA	Self CUDA %	CUDA total	CUDA time avg	# of Calls
96.15%	955.874us	98.70%	981.218us	981.218us	43.489us	100.00%	43.489us	43.489us	1
0.00%	0.000us	0.00%	0.000us	0.000us	43.489us	100.00%	43.489us	43.489us	1
2.55%	25.344us	2.55%	25.344us	25.344us	0.000us	0.00%	0.000us	0.000us	1
1.30%	12.904us	1.30%	12.904us	6.452us	0.000us	0.00%	0.000us	0.000us	2

- aten::add: PyTorch method that calls a CUDA kernel
- What do we see about the overhead in this case?

Profiling: matmul 2048

Name	Self CPU %	Self CPU	CPU total %	CPU total	CPU time avg
aten::matmul	0.64%	20.118us	62.75%	1.958ms	1.958ms
aten::mm	60.91%	1.900ms	62.11%	1.937ms	1.937ms
ampere_sgemm_128x64_nn	0.00%	0.000us	0.00%	0.000us	0.000us
cudaOccupancyMaxActiveBlocksPerMultiprocessor	0.15%	4.713us	0.15%	4.713us	4.713us
cudaLaunchKernel	1.05%	32.690us	1.05%	32.690us	32.690us
cudaDeviceSynchronize	37.25%	1.162ms	37.25%	1.162ms	580.981us

Name	Self CUDA	Self CUDA %	CUDA total	CUDA time avg	# of Calls
aten::matmul	0.000us	0.00%	1.244ms	1.244ms	1
aten::mm	1.244ms	100.00%	1.244ms	1.244ms	1
ampere_sgemm_128x64_nn	1.244ms	100.00%	1.244ms	1.244ms	1
cudaOccupancyMaxActiveBlocksPerMultiprocessor	0.000us	0.00%	0.000us	0.000us	1
cudaLaunchKernel	0.000us	0.00%	0.000us	0.000us	1
cudaDeviceSynchronize	0.000us	0.00%	0.000us	0.000us	2

- ampere_sgemm_128x64_nn is expensive, but CPU overhead still dominates

Profiling

matmul (dim=2048)

```
-------------------------------------------------
                                           Name
-------------------------------------------------
                                    aten::matmul
                                        aten::mm
                          ampere_sgemm_128x64_nn
    cudaOccupancyMaxActiveBlocksPerMultiprocessor
                                  cudaLaunchKernel
                               cudaDeviceSynchronize
-------------------------------------------------
```

matmul (dim=128)

```
-----------------------------------
                             Name
-----------------------------------
                      aten::matmul
                          aten::mm
      ampere_sgemm_32x32_sliced1x4_nn
                    cudaLaunchKernel
                 cudaDeviceSynchronize
-----------------------------------
```

- Note that the kernels are different!

Profiling

Name	Self CPU %	Self CPU	Self CUDA	Self CUDA %
autograd::engine::evaluate_function: AddmmBackward0	1.21%	2.323ms	0.000us	0.00%
AddmmBackward0	0.89%	1.715ms	0.000us	0.00%
aten::mm	3.21%	6.150ms	121.023ms	60.46%
aten::linear	0.27%	509.344us	0.000us	0.00%
aten::addmm	3.23%	6.187ms	68.843ms	34.39%
ampere_sgemm_128x32_tn	0.00%	0.000us	68.843ms	34.39%
ampere_sgemm_64x32_sliced1x4_nt	0.00%	0.000us	60.974ms	30.46%
ampere_sgemm_128x32_nn	0.00%	0.000us	59.798ms	29.87%
ne::evaluate_function: torch::autograd::AccumulateGrad	0.69%	1.333ms	0.000us	0.00%
torch::autograd::AccumulateGrad	0.62%	1.191ms	0.000us	0.00%

- What operations take time in the MLP?

Administrative details and recap

Efficiency

GPUs

GPU Principles

Profiling

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

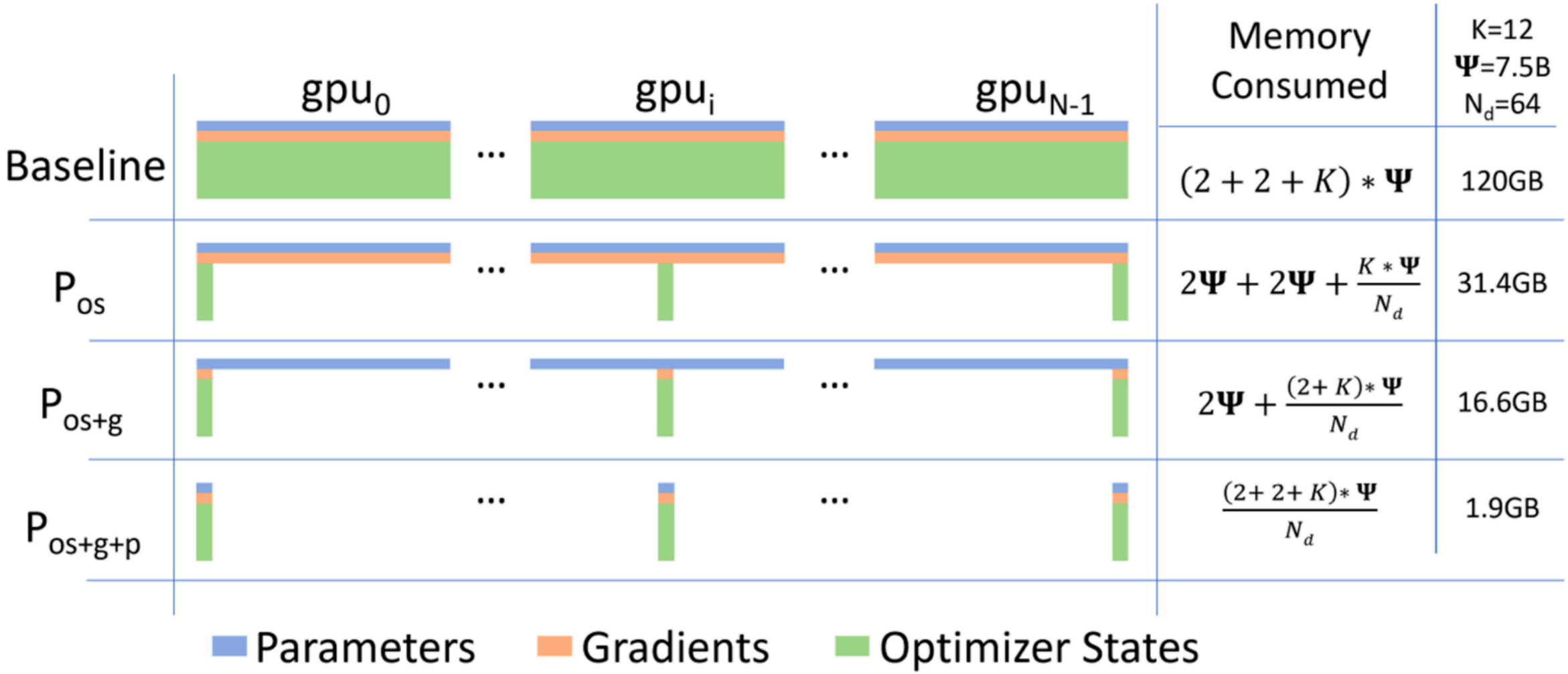
- Suppose we have M nodes (or cards) and batch size B. Run B/M examples on each node/card
- Does this work well for inference?
- Does this work well for training?
  - Every card/node has to store the entire weights & optimizer state. When doing mixed-precision, you're storing a lot:

- 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

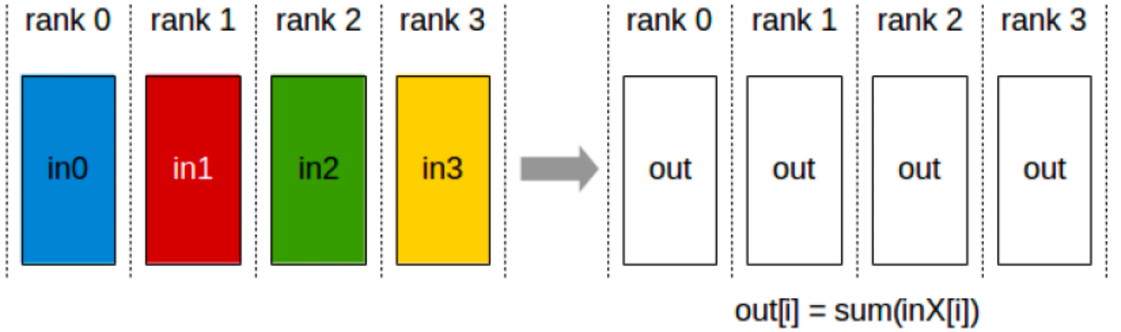
"Optimizer state"

ZeRO

- Split things across GPUs

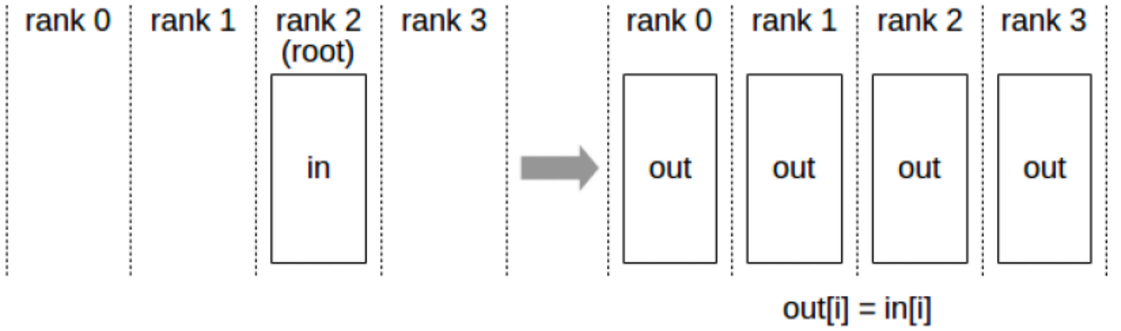


All reduce



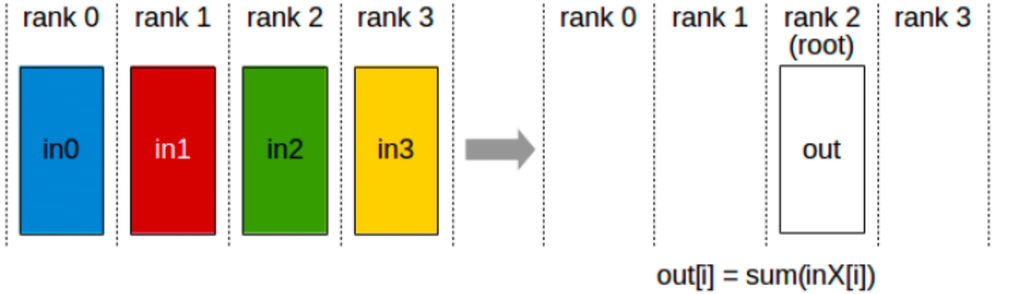
out[i] = sum(inX[i])

Broadcast



out[i] = in[i]

Reduce



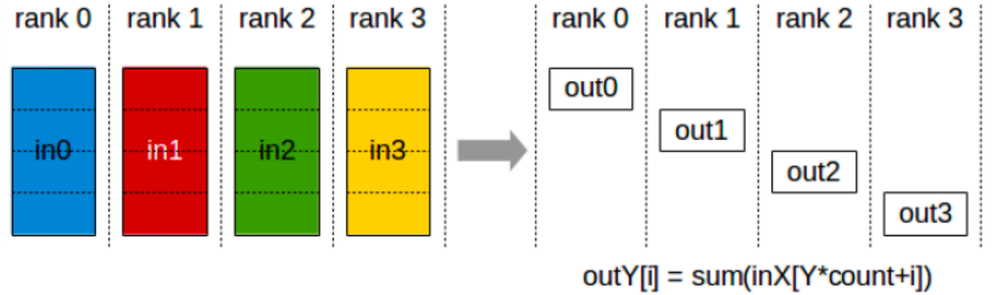
out[i] = sum(inX[i])

All Gather



out[Y*count+i] = inY[i]

Reduce Scatter

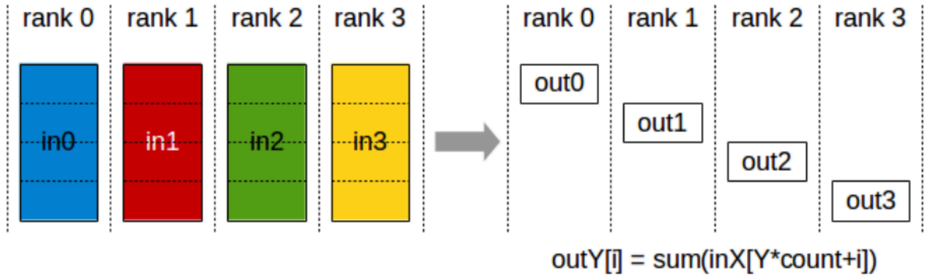


outY[i] = sum(inX[Y*count+i])

ZeRO Stage 1

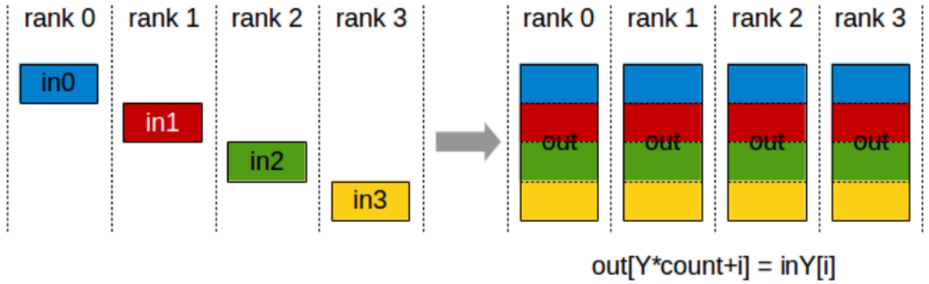
Step 1. Everyone computes a full gradient on their subset of the batch

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

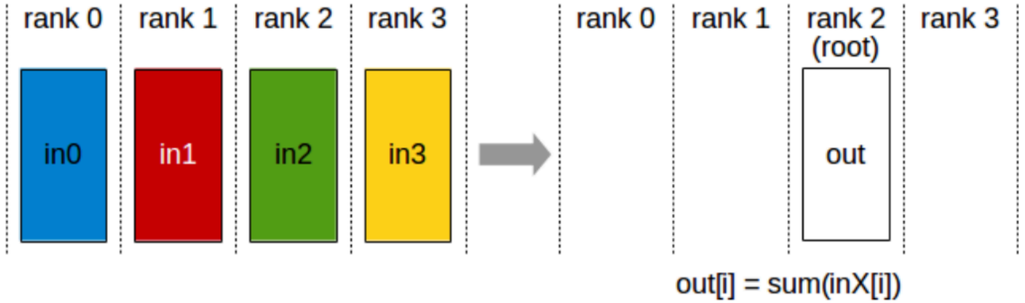


ZeRO Stage 2



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

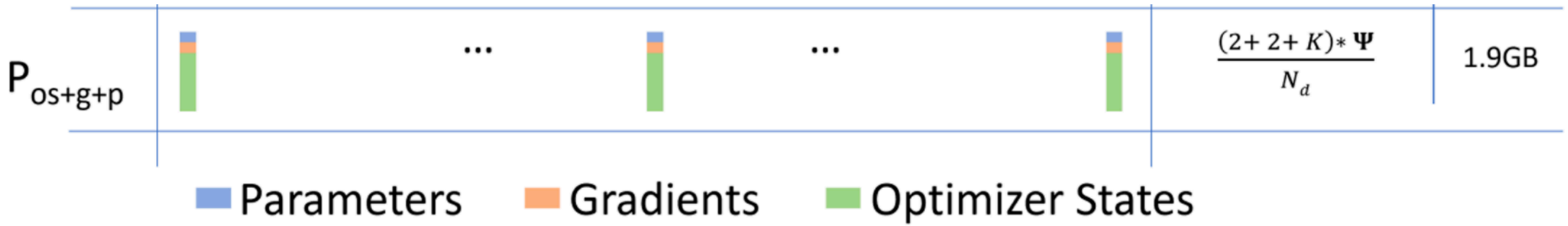


Step 1b. Once gradients are not needed in the backward graph, immediately free it.

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

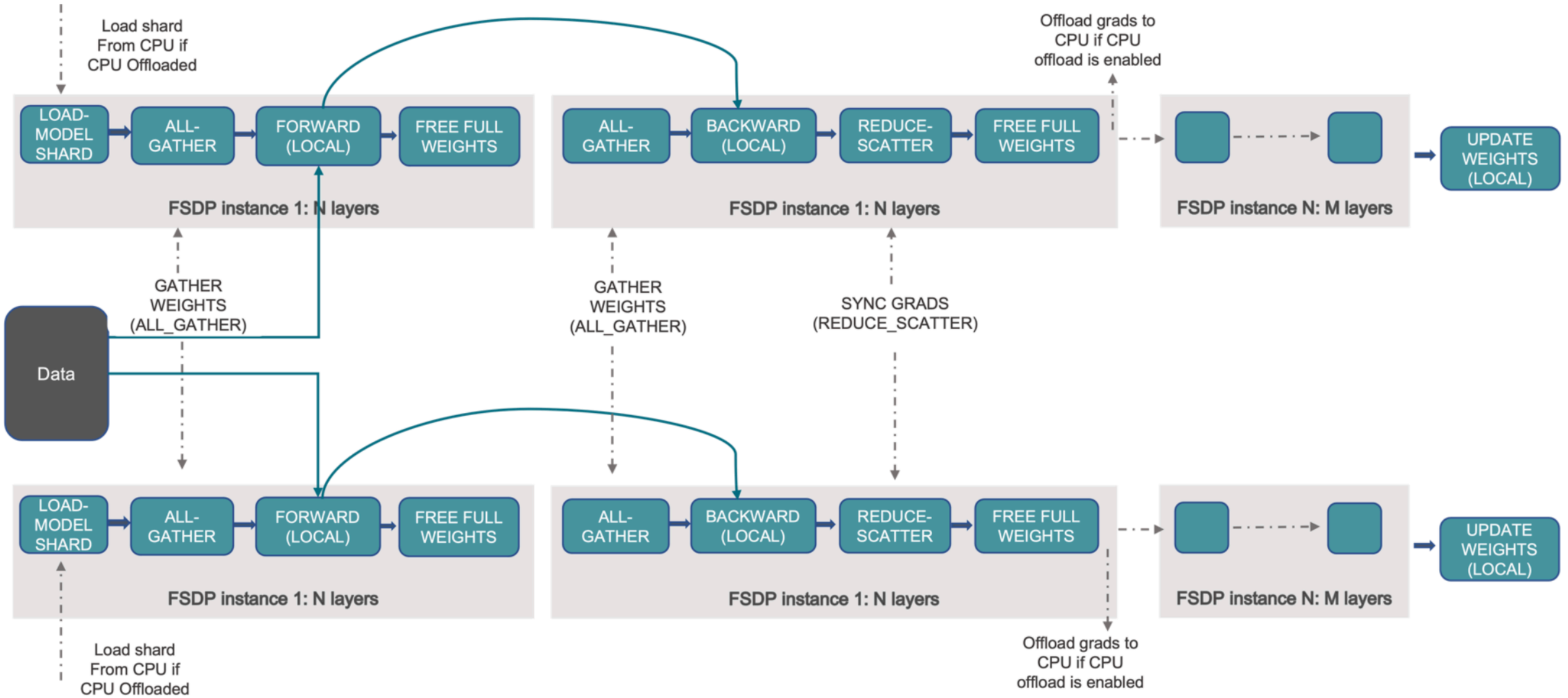
Step 3. All Gather the parameters.

ZeRO Stage 3 (FSDP)



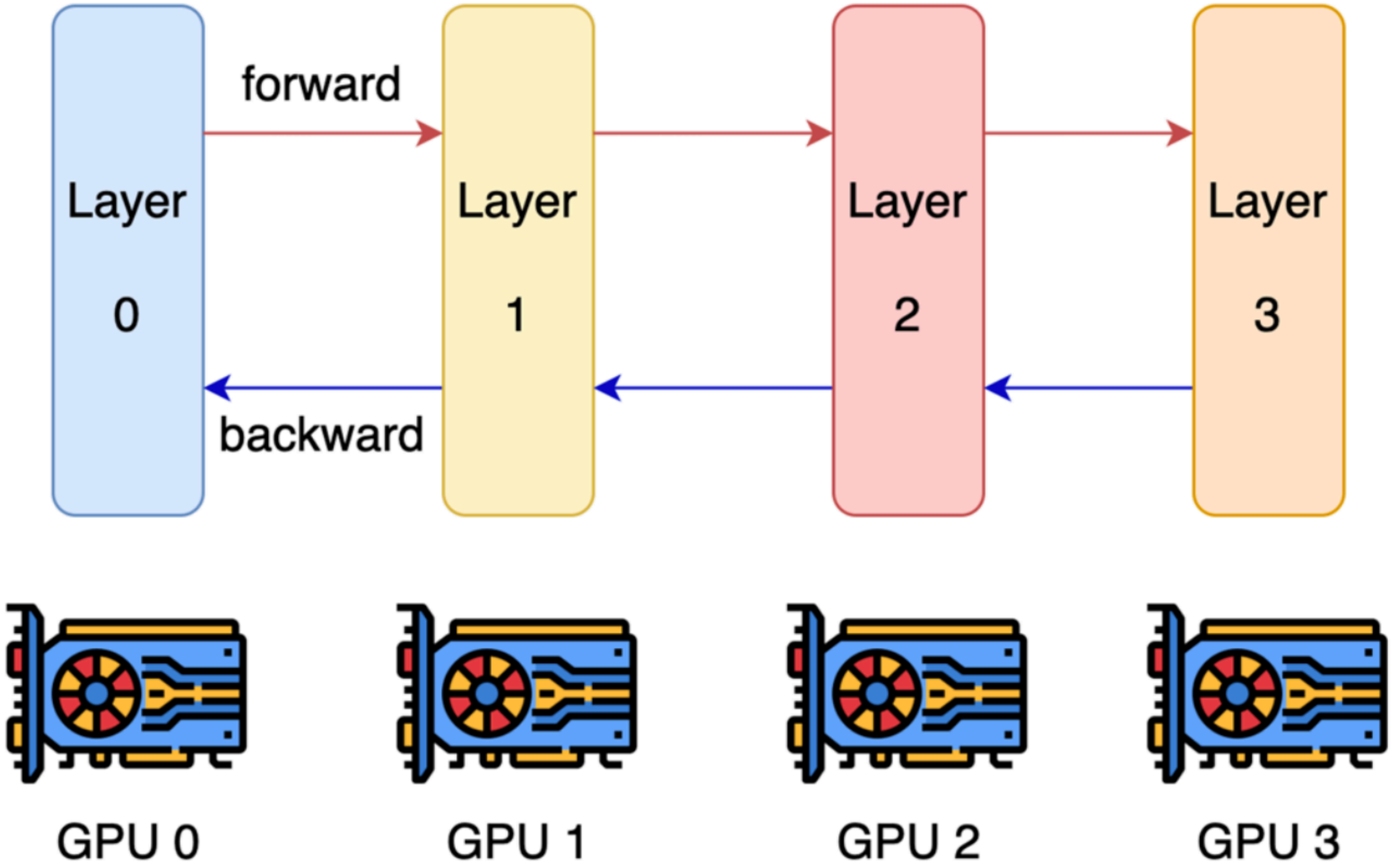
- Now everything is sharded. Each node only has a fraction of the params
- As we move through the computation, each node will have to gather the shards of parameters from all other nodes

ZeRO Stage 3 (FSDP)



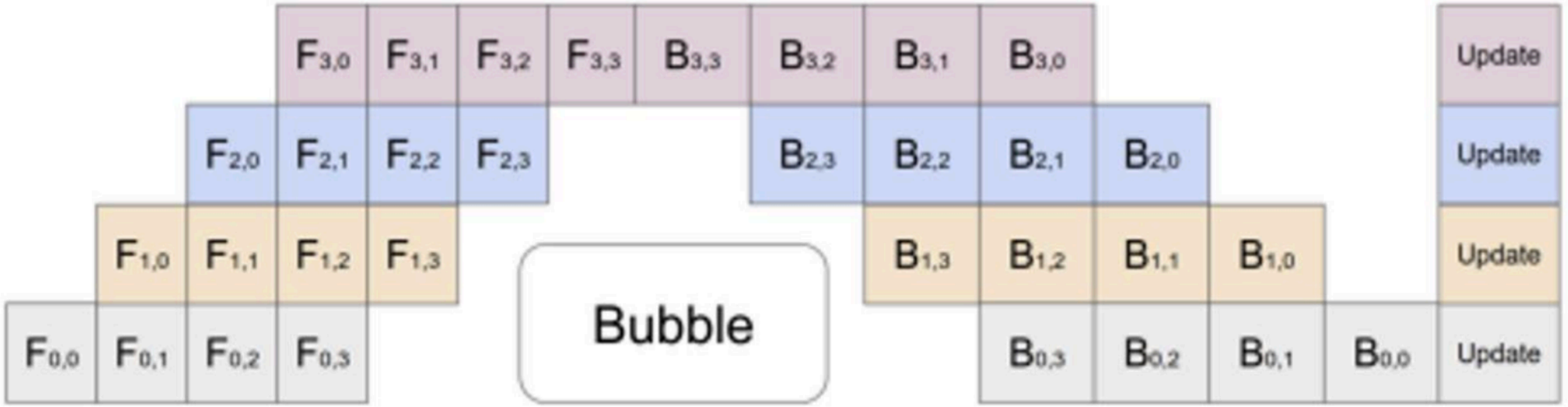
Pipeline Parallelism

- 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 and recap

Efficiency

GPUs

GPU Principles

Profiling

Parallelism