ECE7115 ~~Multimodal VLM~~ LLM

1. Resource Accounting

Spring 2026

Namhyuk Ahn, Inha University

Motivating Questions

- Q: How long would it take to train a 70B parameter model on 15T tokens (# dataset) on 1024 H100s?
  - # FLOPs = 6 x 70e9 x 15e12
  - H100 FLOPs / sec = 1979e12 / 2
  - MFU (Model FLOPs Utilization) = 0.5

  - FLOPs / day = (H100 FLOPs / sec) x MFU x 1024 x 3600 x 24
  - Days = # FLOPs / (FLOPs / day)

Motivating Questions

- Q: What's the largest model that can you can train on 8 H100s using AdamW (naively)?
  - H100 memory: 80e9 (bytes)
  - Bytes / params = 4 + 4 + (4 + 4) # params, gradients, states
  - # Params = (H100 memory x 8) / (Bytes / params)

Lecture Overview

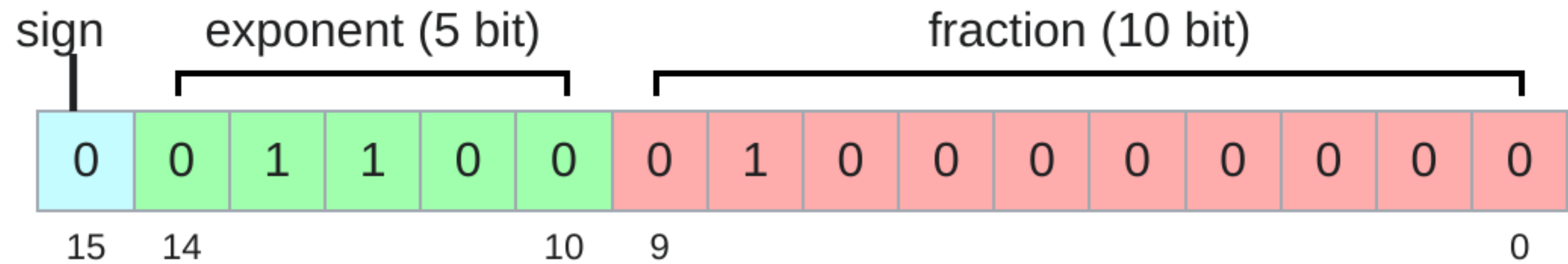
- We will account for two types of resources
  - Memory (GB)
  - Compute (FLOPs)

Floating Point

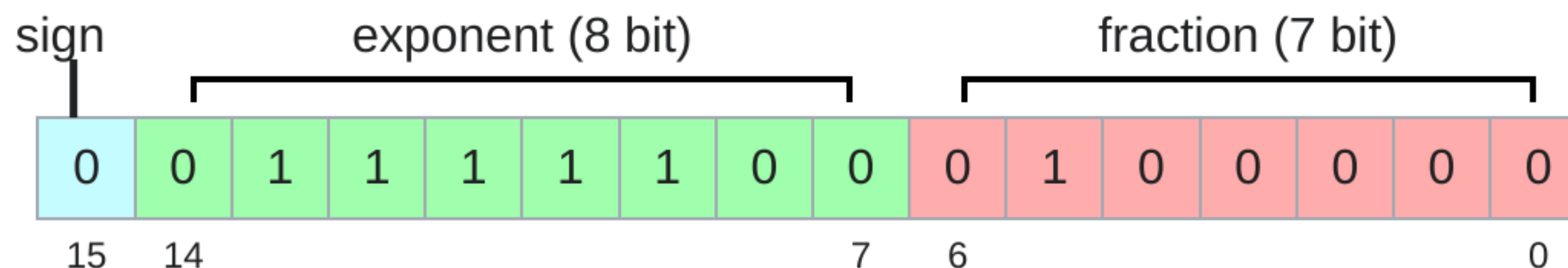
- Almost everything is stored as floating point numbers

- float32 (fp32)
  - The float32 data type (also known as single precision) is the default
  - Traditionally, in scientific computing, float32 is the baseline
  - And you could use double precision (float64) in some cases

Floating Point

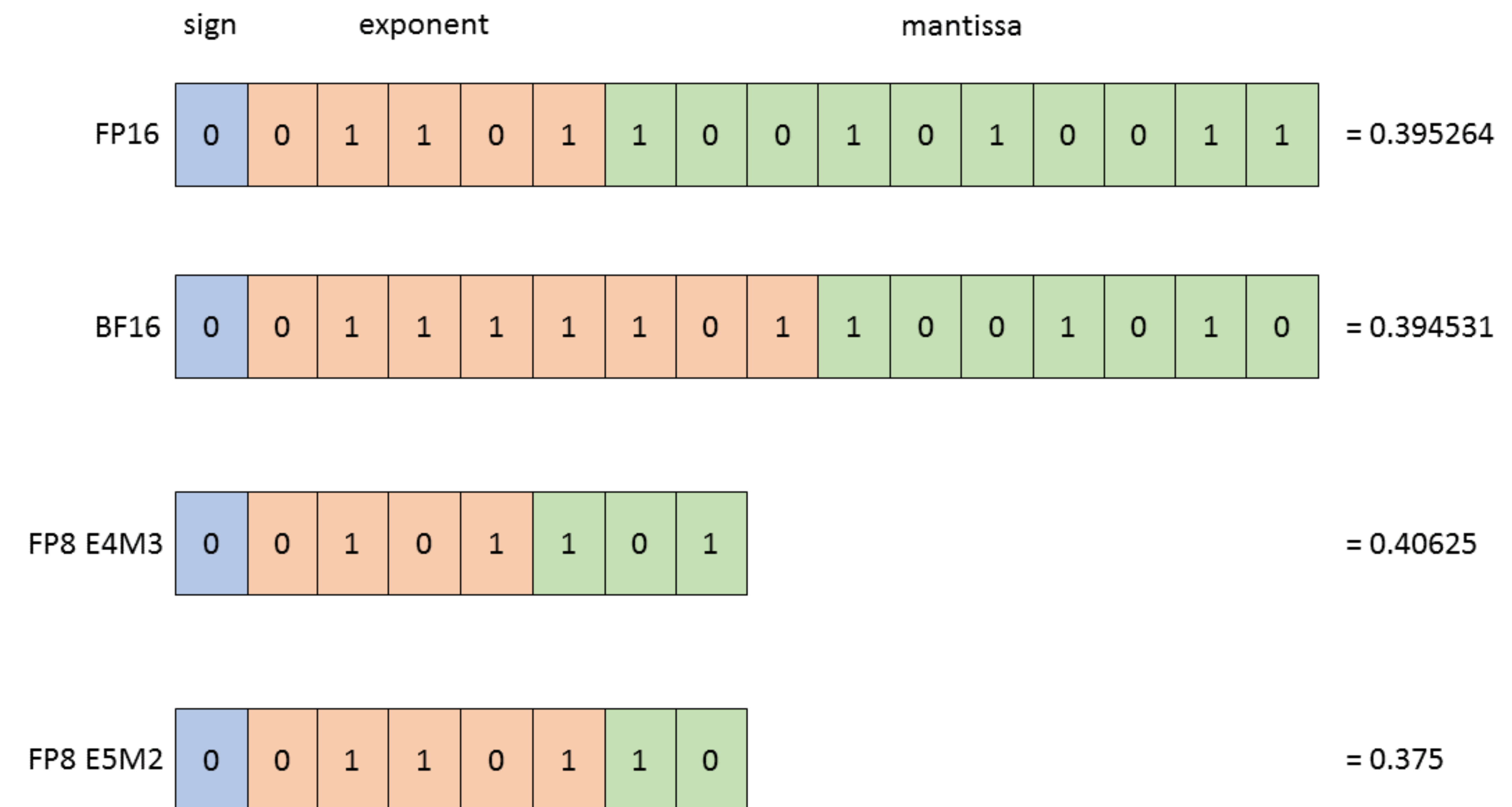


- float16 (fp16)
  - The float16 data type (aka half precision) cuts down the memory
  - The dynamic range isn't great (gradient overflow, underflow)
- bfloat16 (bf16)
  - Google Brain developed bfloat (brain floating point)
  - Uses the same memory as fp16 but has the same range as fp32
  - The resolution is worse, but this matters less for deep learning



Floating Point

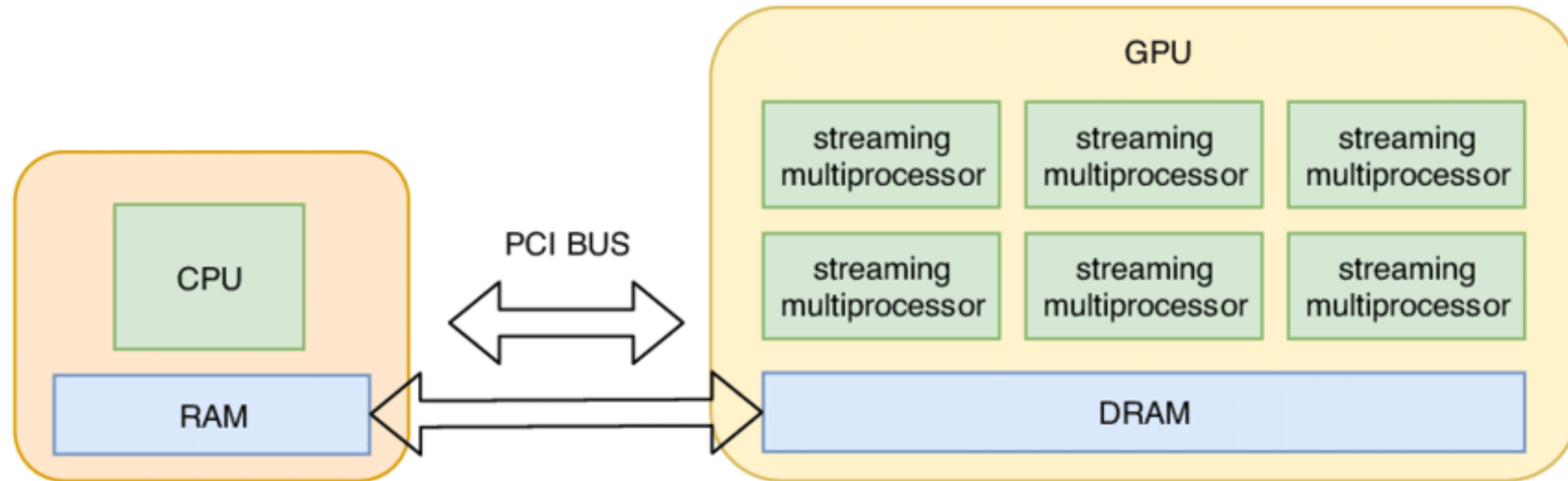
- fp8
  - H100s support two variants of FP8
  - E4M3 ([-448, 448])
  - E5M2 ([-57344, 57344])



- Implications on training
  - Training with fp32 works, but requires lots of memory
  - Training with fp8, fp16 and even bf16 is risky; you can get instability
  - Solution: use mixed precision training (we will see later)

Tensors in GPU

- When we do `x = torch.zeros(32, 32, device="cuda:0")`
- Tensor `x` moves to GPU



- We will see the details of the GPU later

Floating-Point Operation (FLOP)

- FLOP is a basic operation like addition (x + y) or multiplication (x * y)
  - FLOPs: floating-point operations (measure of computation done)
  - FLOP/s: FLOP per sec (also written as FLOPS), measure the speed

- Intuitions
  - Training GPT-3 (2020) takes 3.14e23 FLOPs
  - Training GPT-4 (2023) takes 2e25 FLOPs (speculation)

  - H100 has a peak performance of 1979 teraFLOP/s with sparsity, 50% performance without sparsity (1979e12 / 2)
  - 8 H100s for 2 weeks = 8 x (3600 x 24 x 7) x (1979e12 / 2) FLOPs

Floating-Point Operation (FLOP)

- Linear layer: Suppose we have N = 16384, D_in = 32768, D_out = 8192
  - `x = torch.ones(N, D_in, device=device)`
  - `w = torch.randn(D_in, D_out, device=device)`
  - `y = x @ w`

  - We have one mul (x[i, j] * w[j, k]) and one addition per (i, j, k) triple

  - FLOPs = 2 x N x D_in x D_out (2 for multiplication and addition)

- In general, FLOPs for forward pass is 2 x (# data or tokens) x (# params)
  - This can be generalized to Transformers

Floating-Point Operation (FLOP)

- Elementwise operation on a m x n matrix = O(mn) FLOPs
  - O(.): depends on the operations
- Addition of two m x n matrices = mn FLOPs

- In general, matmul is the most expensive operation in deep models
  - And most of the modules are made with matmul

Model FLOPs Utilization (MFU)

- Actual FLOP/s = FLOPS (of module) / wall clock time
- Promised FLOP/s =

	H100 SXM
FP64	34테라플롭스
FP64 Tensor 코어	67테라플롭스
FP32	67테라플롭스
TF32 Tensor 코어*	989테라플롭스
BFLOAT16 Tensor 코어*	1,979테라플롭스
FP16 Tensor 코어*	1,979테라플롭스
FP8 Tensor 코어*	3,958테라플롭스

- MFU = (actual FLOP/s) / (promised FLOP/s)
  - Usually, MFU of >= 0.5 is quite good
  - And will be higher if matmuls dominate

Recap

- Matrix multiplications dominate: (2mnp) FLOPs
- FLOPs for forward pass is 2 x (# data or tokens) x (# params)
- FLOP/s depends on hardware (H100 >> A100) and dtype(bf16 >> fp32)
- Model FLOPs utilization (MFU): (actual FLOP/s) / (promised FLOP/s)

FLOPs of Backward Pass

- Suppose we have N = 16384, D_in = 32768, D_out = 8192
- `X = [N, D_in], W = [D_in, D_out] # X --W--> loss`

- FLOPS of backward = 4 x N x D_in x D_out
  - 2 for calculating grads of W (w.grad[j,k] = sum_i x[i,j] * h.grad[i,k])
  - 2 for calculating grads of X (x.grad[i,j] = sum_k w[j,k] * h.grad[i,k])

- Putting it together
  - Forward pass: 2 x (# data or tokens) x (# params)
  - Backward pass: 4 x (# data or tokens) x (# params)
  - Total: 6 x (# data or tokens) x (# params)

Memory

- Linear layer [D_in x D_out] (no bias)

- # params = D_in x D_out
- # activations = N x D_out
- # grads = # params
- # optimizer states = 2 x # params (assume we use Adam family)

- Total memory (bytes) = 4 x (4 x # params + # activations)
  - 4 for fp32
  - But in reality, we use mixed precision; more complicated

Recall: Adam

```
moment1 = 0
moment2 = 0
for t in range(1, num_steps + 1):  # Start at t = 1
    dw = compute_gradient(w)
    moment1 = beta1 * moment1 + (1 - beta1) * dw
    moment2 = beta2 * moment2 + (1 - beta2) * dw * dw
    moment1_unbias = moment1 / (1 - beta1 ** t)
    moment2_unbias = moment2 / (1 - beta2 ** t)
    w -= learning_rate * moment1_unbias / (moment2_unbias.sqrt() + 1e-7)
```

this is state!

Momentum
RMSProp
Bias correction

- Momentum: 1st moment (average), adding momentum to gradient
- RMSProp: 2nd moment (variance), scaling learning rate

Recall: AdamW

- Adam is really good, but sometimes not good at generalizations
  - Because standard Adam + weight decay is not effective

Algorithm 2 Adam with L_2 regularization and Adam with decoupled weight decay (AdamW)

1: **given** $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}, \lambda \in \mathbb{R}$
2: **initialize** time step $t \leftarrow 0$, parameter vector $\boldsymbol{\theta}_{t=0} \in \mathbb{R}^n$, first moment vector $\boldsymbol{m}_{t=0} \leftarrow \boldsymbol{0}$, second moment vector $\boldsymbol{v}_{t=0} \leftarrow \boldsymbol{0}$, schedule multiplier $\eta_{t=0} \in \mathbb{R}$
3: **repeat**
4: $\quad t \leftarrow t + 1$
5: $\quad \nabla f_t(\boldsymbol{\theta}_{t-1}) \leftarrow \text{SelectBatch}(\boldsymbol{\theta}_{t-1})$ ▷ select batch and return the corresponding gradient
6: $\quad \boldsymbol{g}_t \leftarrow \nabla f_t(\boldsymbol{\theta}_{t-1}) + \lambda\boldsymbol{\theta}_{t-1}$
7: $\quad \boldsymbol{m}_t \leftarrow \beta_1 \boldsymbol{m}_{t-1} + (1 - \beta_1)\boldsymbol{g}_t$ ▷ here and below all operations are element-wise
8: $\quad \boldsymbol{v}_t \leftarrow \beta_2 \boldsymbol{v}_{t-1} + (1 - \beta_2)\boldsymbol{g}_t^2$
9: $\quad \hat{\boldsymbol{m}}_t \leftarrow \boldsymbol{m}_t/(1 - \beta_1^t)$ ▷ β_1 is taken to the power of t
10: $\quad \hat{\boldsymbol{v}}_t \leftarrow \boldsymbol{v}_t/(1 - \beta_2^t)$ ▷ β_2 is taken to the power of t
11: $\quad \eta_t \leftarrow \text{SetScheduleMultiplier}(t)$ ▷ can be fixed, decay, or also be used for warm restarts
12: $\quad \boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \eta_t \left(\alpha \hat{\boldsymbol{m}}_t / (\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon) + \lambda\boldsymbol{\theta}_{t-1} \right)$
13: **until** *stopping criterion is met*
14: **return** optimized parameters $\boldsymbol{\theta}_t$

Motivating Questions... Again

- Q: How long would it take to train a 70B parameter model on 15T tokens (# dataset) on 1024 H100s?
  - # FLOPs = 6 x 70e9 x 15e12
  - H100 FLOPs / sec = 1979e12 / 2
  - MFU (Model FLOPs Utilization) = 0.5

  - FLOPs / day = (H100 FLOPs / sec) x MFU x 1024 x 3600 x 24
  - Days = # FLOPs / (FLOPs / day)

Motivating Questions… Again

- Q: What's the largest model that can you can train on 8 H100s using AdamW (naively)?
  - H100 memory: 80e9 (bytes)
  - Bytes / params = 4 + 4 + (4 + 4) # params, gradients, states
  - # Params = (H100 memory x 8) / (Bytes / params)

  - For simplicity, we omit # activations