Recap: Computations of DL

EECE695D: Efficient ML Systems

Spring 2025

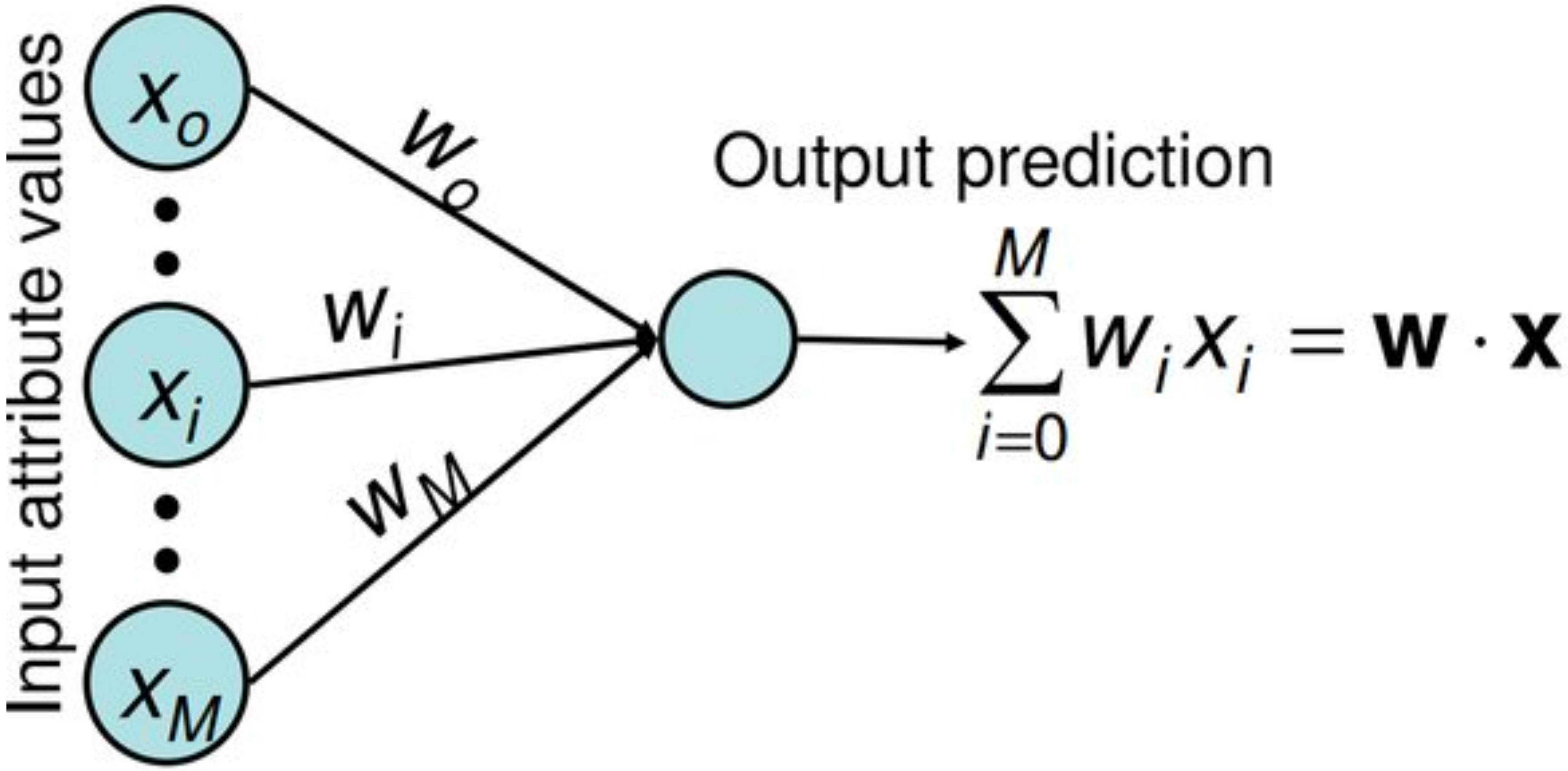
Agenda

- **Last Class**
    - Motivations
    - What to expect
    - Administrivia
- **Today**
    - A brief recap of ML/DL
        - Inference cost?
        - Training cost?

Linear Models

Model

- Linear Model = Extremely simple neural net
  - e.g., linear regression, perceptrons, …
  - for an easy discussion, suppose that $y \in \mathbb{R}^1$ (the value to be predicted)



Inference

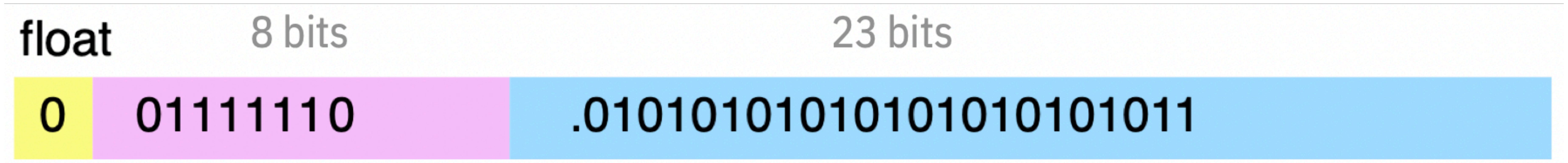
- **Question.** How many computations do we need for an inference?

$$\hat{y} = \mathbf{w}^\top \mathbf{x}, \qquad \mathbf{x}, \mathbf{w} \in \mathbb{R}^d$$

  - Which unit will you use?

Recap: FLOPs

- One option is to use FLOPs (Floating Point Operations)
  - 1 addition of floats = 1 FLOP
  - 1 multiplication of floats = 1 FLOP



Sign Exponent Fraction (a.k.a. Mantissa)

$$\text{value} = (\text{sign}) \quad \times \quad 2^{(\text{exponent})} \quad \times \quad 1.(\text{fraction})$$

https://docs.nvidia.com/cuda/floating-point/index.html

Inference (again)

$$\hat{y} = \mathbf{w}^\top \mathbf{x}, \qquad \mathbf{x}, \mathbf{w} \in \mathbb{R}^d$$

- **Answer.** We need $2d$ FLOPs
  - We are performing an elementary operation d times:

$$s \leftarrow s + (w_i \times x_i)$$

MAC / MAD

- Multiply and add (or accumulate) abstracted into a single operation

$$a \leftarrow a + (b \times c)$$

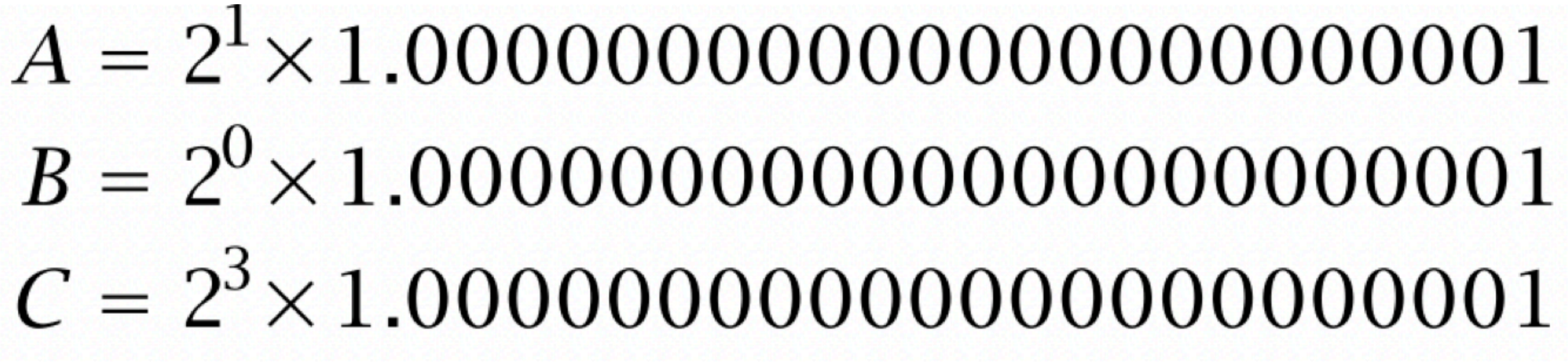
  - Rounding only done once; better precision

$$a \leftarrow \mathrm{rn}\big(a + (b \times c)\big)$$

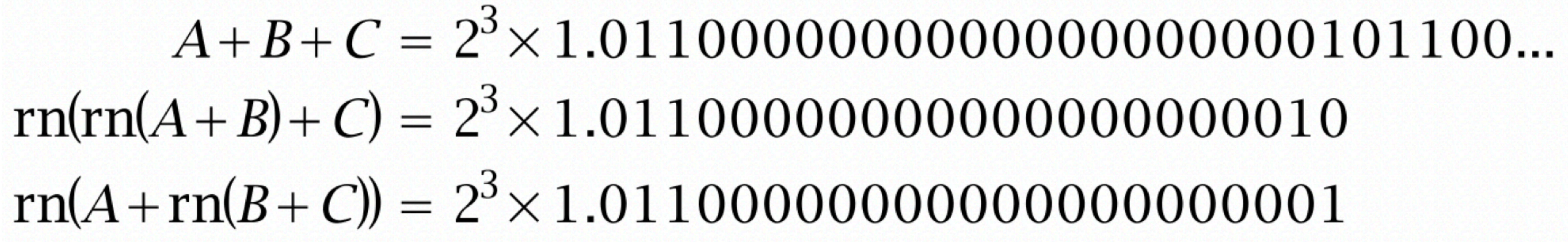
$$a \leftarrow \mathrm{rn}\big(a + \mathrm{rn}(b \times c)\big)$$

Example

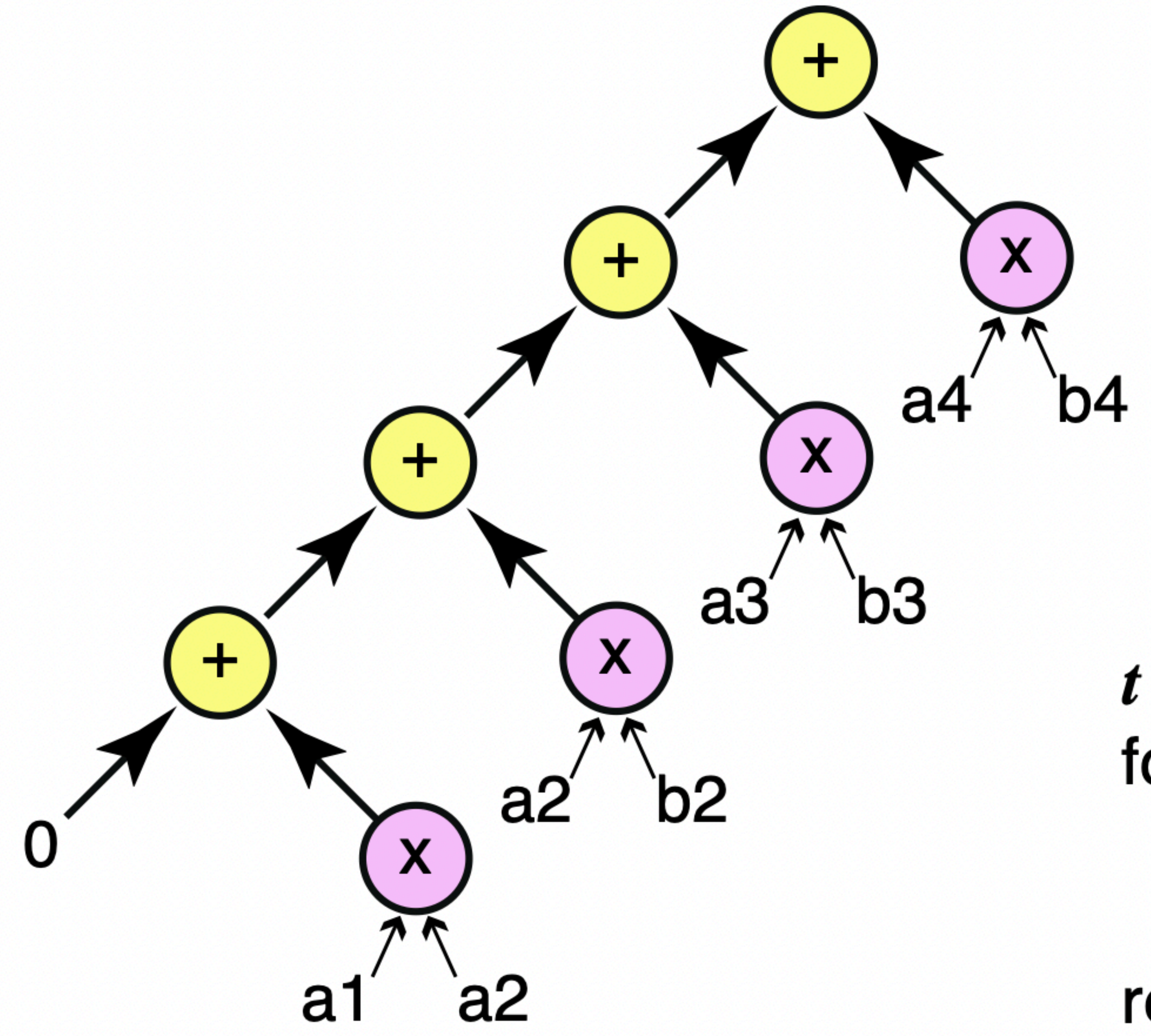
- To see the importance of rounding, consider the following example:



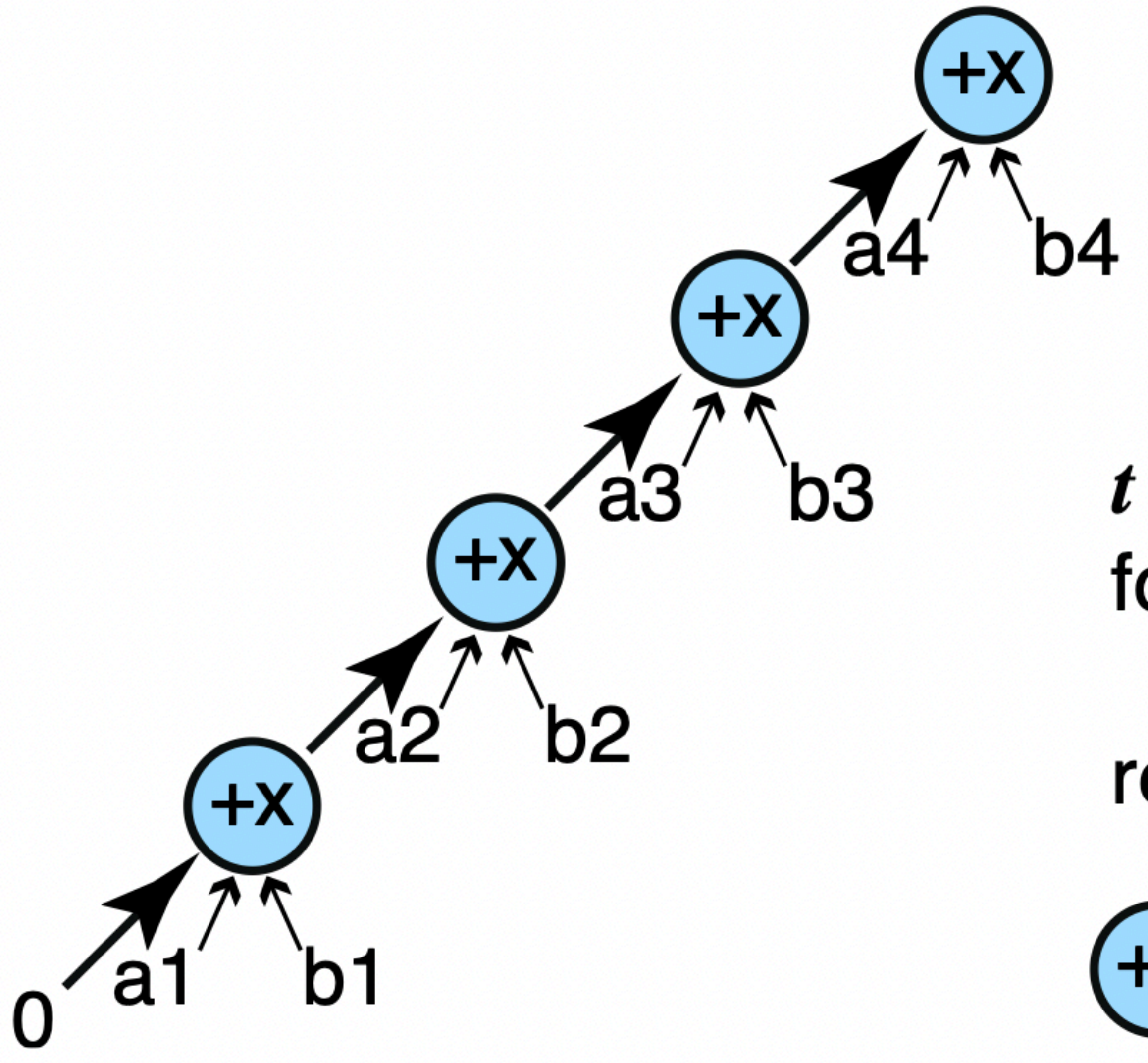
$$
\begin{aligned}
A &= 2^1 \times 1.00000000000000000000001 \\
B &= 2^0 \times 1.00000000000000000000001 \\
C &= 2^3 \times 1.00000000000000000000001
\end{aligned}
$$



$$
\begin{aligned}
A+B+C &= 2^3 \times 1.0110000000000000000000101100\ldots \\
\mathrm{rn}(\mathrm{rn}(A+B)+C) &= 2^3 \times 1.01100000000000000000010 \\
\mathrm{rn}(A+\mathrm{rn}(B+C)) &= 2^3 \times 1.01100000000000000000001
\end{aligned}
$$

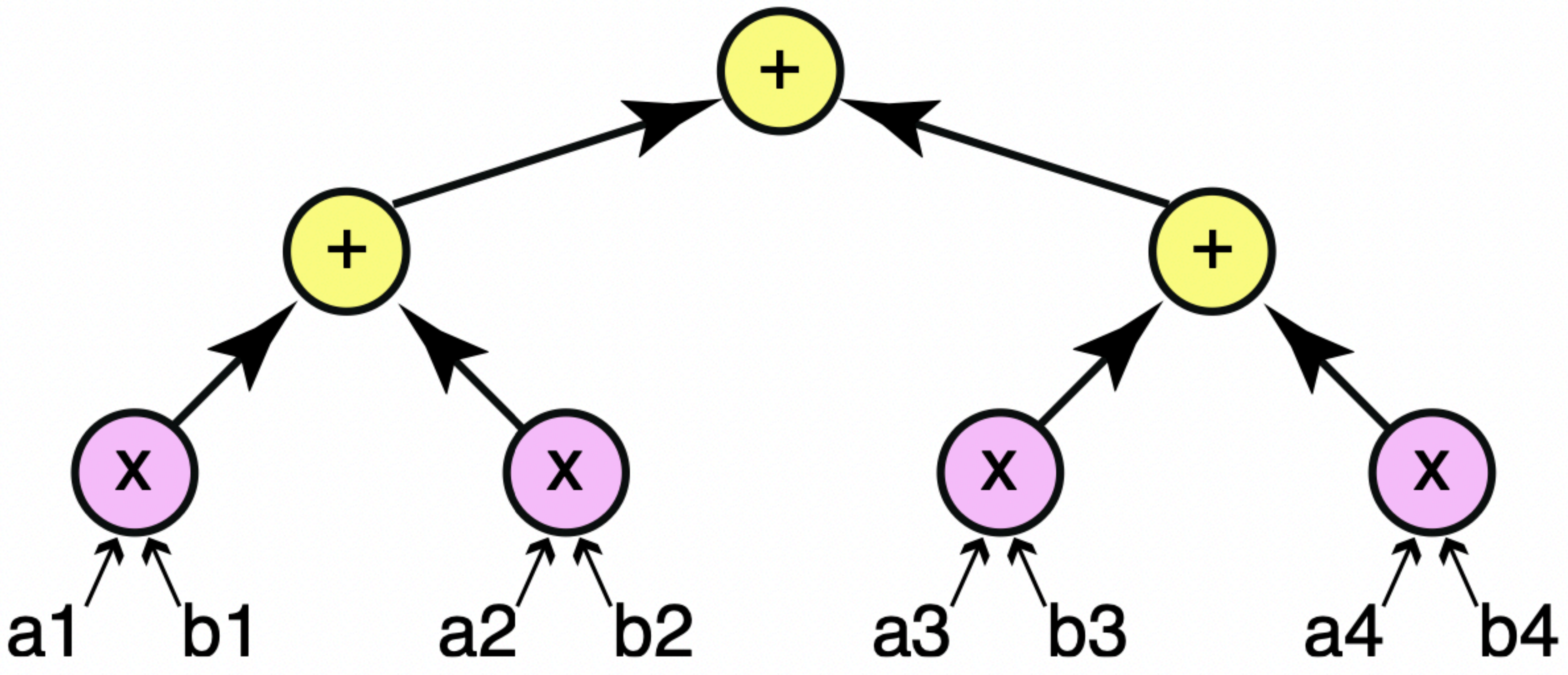


```
$t = 0$
for $i$ from 1 to 4
        $p$ = rn ($a_i$ x $b_i$)
        $t$ = rn ($t + p$)
return $t$
```



```
$t = 0$
for $i$ from 1 to 4
        $t$ = rn ($a_i$ x $b_i + t$)
return $t$
```

+x Fused multiply-add



```
$p1$ = rn ($a_1$ x $b_1$)
$p2$ = rn ($a_2$ x $b_2$)
$p3$ = rn ($a_3$ x $b_3$)
$p4$ = rn ($a_4$ x $b_4$)
$s_{left}$ = rn ($p_1 + p_2$)
$s_{right}$ = rn ($p_3 + p_4$)
$t$ = rn ($s_{left} + s_{right}$)
return $t$
```

TOPs

- Sometimes, you would see “TOPs”
  - Usually an umbrella term for (a trillion) INT8 / INT4 operations (here, “op” could mean a single fused multiply-adds)

	Jetson AGX Orin series				Jetson Orin NX series		Jetson Orin Nano series		
	Jetson AGX Orin Developer Kit	Jetson AGX Orin 64GB	Jetson AGX Orin Industrial	Jetson AGX Orin 32GB	Jetson Orin NX 16GB	Jetson Orin NX 8GB	Jetson Orin Nano Super Developer Kit	Jetson Orin Nano 8GB	Jetson Orin Nano 4GB
AI Performance	275 TOPS		248 TOPs	200 TOPS	157 TOPS	117 TOPS	67 TOPS	67 TOPS	34 TOPS
GPU	2048-core NVIDIA Ampere architecture GPU with 64 Tensor Cores			1792-core NVIDIA Ampere c GPU with 56 Tensor Cores	1024-core NVIDIA Ampere architecture GPU with 32 Tensor Cores		1024-core NVIDIA Ampere architecture GPU with 32 Tensor Cores		512-core NVIDIA Ampere architecture GPU with 16 Tensor Cores
GPU Max Frequency	1.3 GHz		1.2 GHz	930 MHz	1173MHz	1173MHz	1020MHz	1020MHz	1020MHz

Training

- **Question.** How many computations do we need for training?

$$\hat{y} = \mathbf{w}^\top \mathbf{x}, \qquad \mathbf{x}, \mathbf{w} \in \mathbb{R}^d$$

Training

- This is ill-posed, as we require more setup:
  - We have a **dataset** $D = \{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_N, y_N)\}$
  - We use the **squared loss** $\ell(y, \hat{y}) = (y - \hat{y})^2$
  - We solve the **empirical risk minimization**

$$\min_{\mathbf{w} \in \mathbb{R}^d} \sum_{i=1}^{N} (y_i - \mathbf{w}^\top \mathbf{x}_i)^2 \quad \Leftrightarrow \quad \min_{\mathbf{w} \in \mathbb{R}^d} \|\mathbf{y} - \mathbf{w}^\top \mathbf{X}\|_2^2$$

- **Question.** Can we answer the question now?

Training

- Not really!
    - It depends on the optimization method to solve:

$$\min_{\mathbf{w}\in\mathbb{R}^d} \|\mathbf{y} - \mathbf{w}^\top \mathbf{X}\|_2^2$$

        - Exact solution
        - Gradient descent

Exact solution

- Exact solution can be found as

$$\mathbf{w}^* = (\mathbf{X}^\top \mathbf{X})^\dagger \mathbf{X}^\top \mathbf{y}$$

  - Here, dagger means Moore-Penrose pseudoinverse

- Typically, computing the matrix inverse is quite costly

  - $O(n^3)$ FLOPs, where constants depend on "how"
    (tradeoff of numerical stability & computation)

Gradient descent

- We'll use gradient descent (does this make our life easier?)

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \cdot \nabla L(\mathbf{w})$$

  - For linear regression, this is:

$$\mathbf{w} \leftarrow (\mathbf{I} - 2\eta\mathbf{X}^\top\mathbf{X})\mathbf{w} + 2\eta \cdot \mathbf{X}^\top\mathbf{y}$$

    - Luckily, blue terms can be pre-computed, and thus negligible.
      - **Q.** What if we use SGD?
      - **Q.** Any sacrifice in memory?

Per-iteration compute

- After pre-computing, we are doing simply

$$\mathbf{w} \leftarrow \mathbf{A}\mathbf{w} + \mathbf{b}, \qquad \mathbf{A} \in \mathbb{R}^{d\times d}, \mathbf{b} \in \mathbb{R}^{d}$$

  - **MV mults.** Equivalent to computing dot products d times = $2d^2$ FLOPs
  - **VV adds.** d FLOPs

- **Brainteaser.** Can you find a better way, if $d \gg N$?

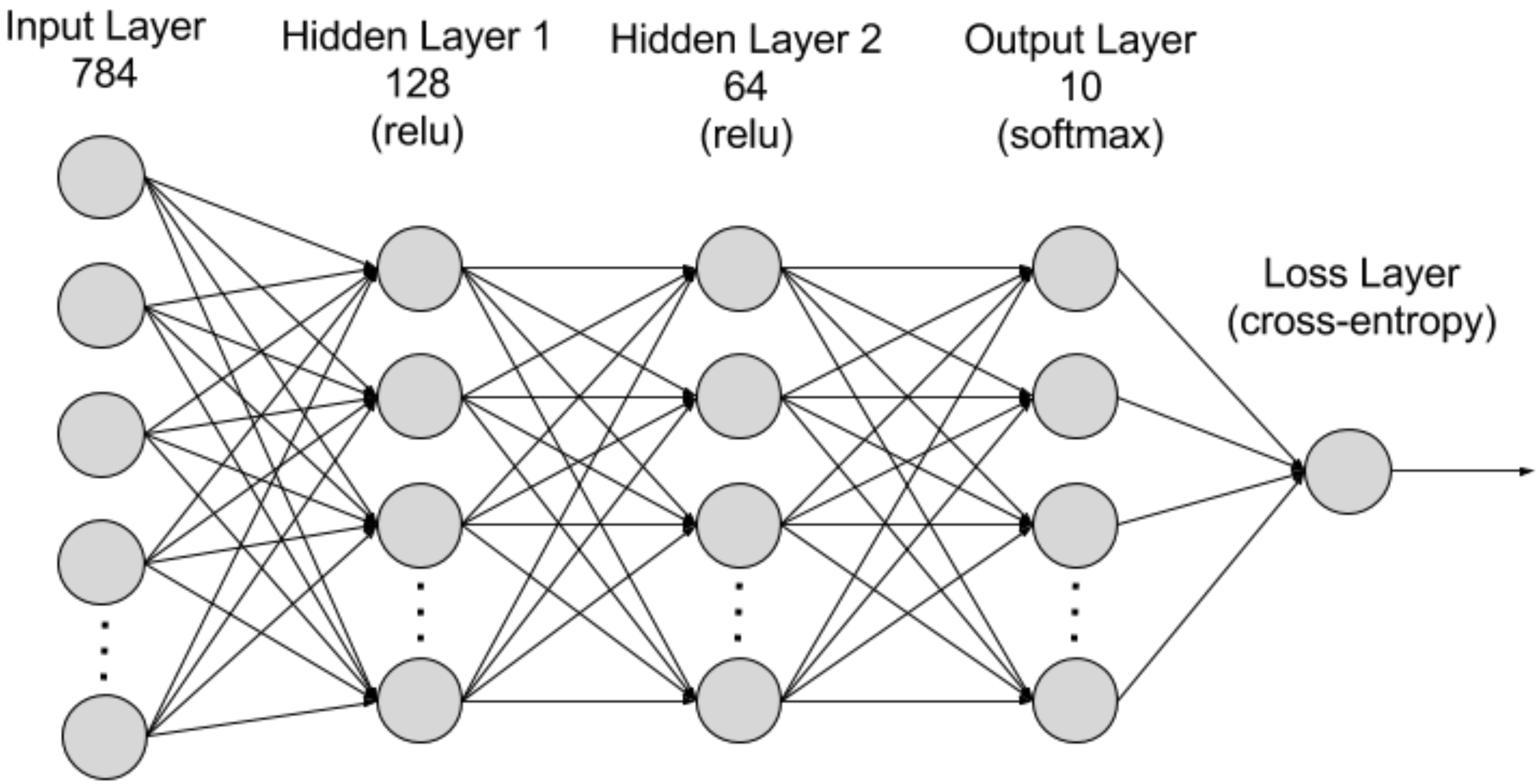
Takeaways

- Many things affect the computational cost
  - Dataset size
  - Model size
  - Optimization algorithm
  - Number of iterations
- Even worse, the optimal way to compute can vary
  - d vs N
- Luckily, measuring the inference cost is relatively simple 😅

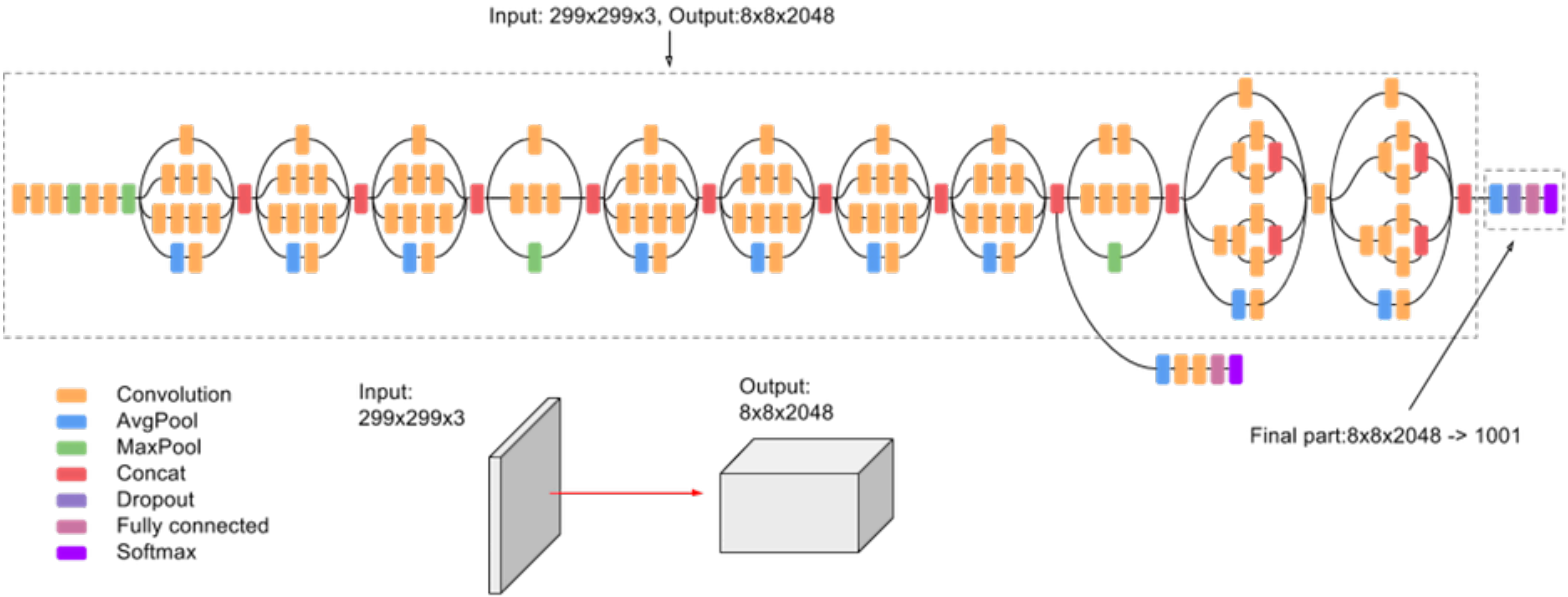
Neural nets

Neural nets

- A graph of **layers**
  - Each layer performs an elementary operation
  - **Example**. MLP



Example: Inceptionv3



Input: 299x299x3, Output:8x8x2048

- 🟧 Convolution
- 🟦 AvgPool
- 🟩 MaxPool
- 🟥 Concat
- 🟪 Dropout
- Fully connected
- 🟪 Softmax

Input:
299x299x3

Output:
8x8x2048

Final part:8x8x2048 -> 1001

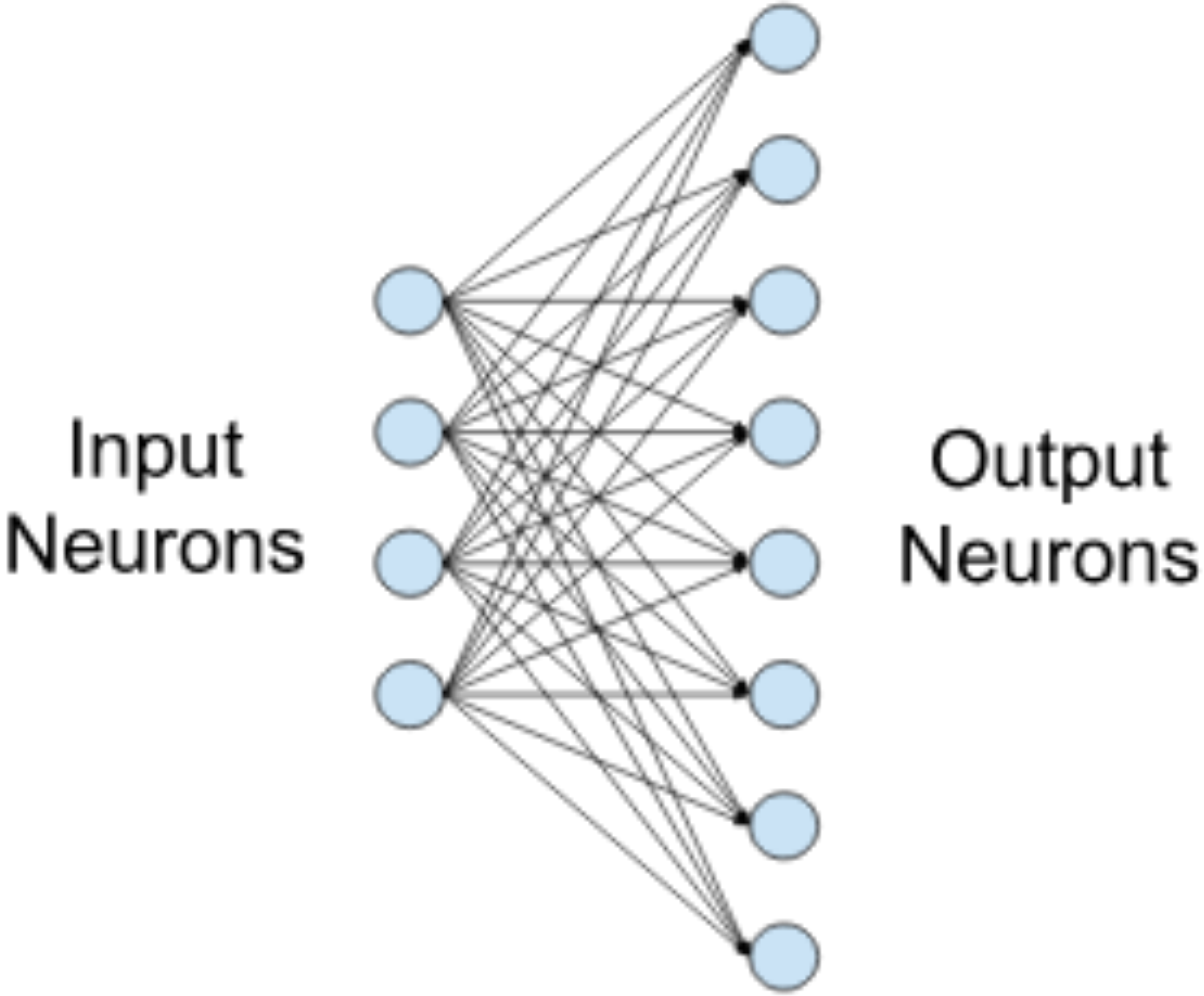
https://cloud.google.com/tpu/docs/inception-v3-advanced?hl=en

Linear layer

- The simplest building block

$$\mathbf{Y} = \sigma(\mathbf{W}\mathbf{X} + \mathbf{b}\mathbf{1}^\top)$$

  - Input $\mathbf{X} \in \mathbb{R}^{d_i \times n}$
    - d_i: Input dimension
    - n: batch size
  - Weight $\mathbf{W} \in \mathbb{R}^{d_o \times d_i}$
    - d_o: output dimension
  - Activation function $\sigma(\cdot)$



Linear layer

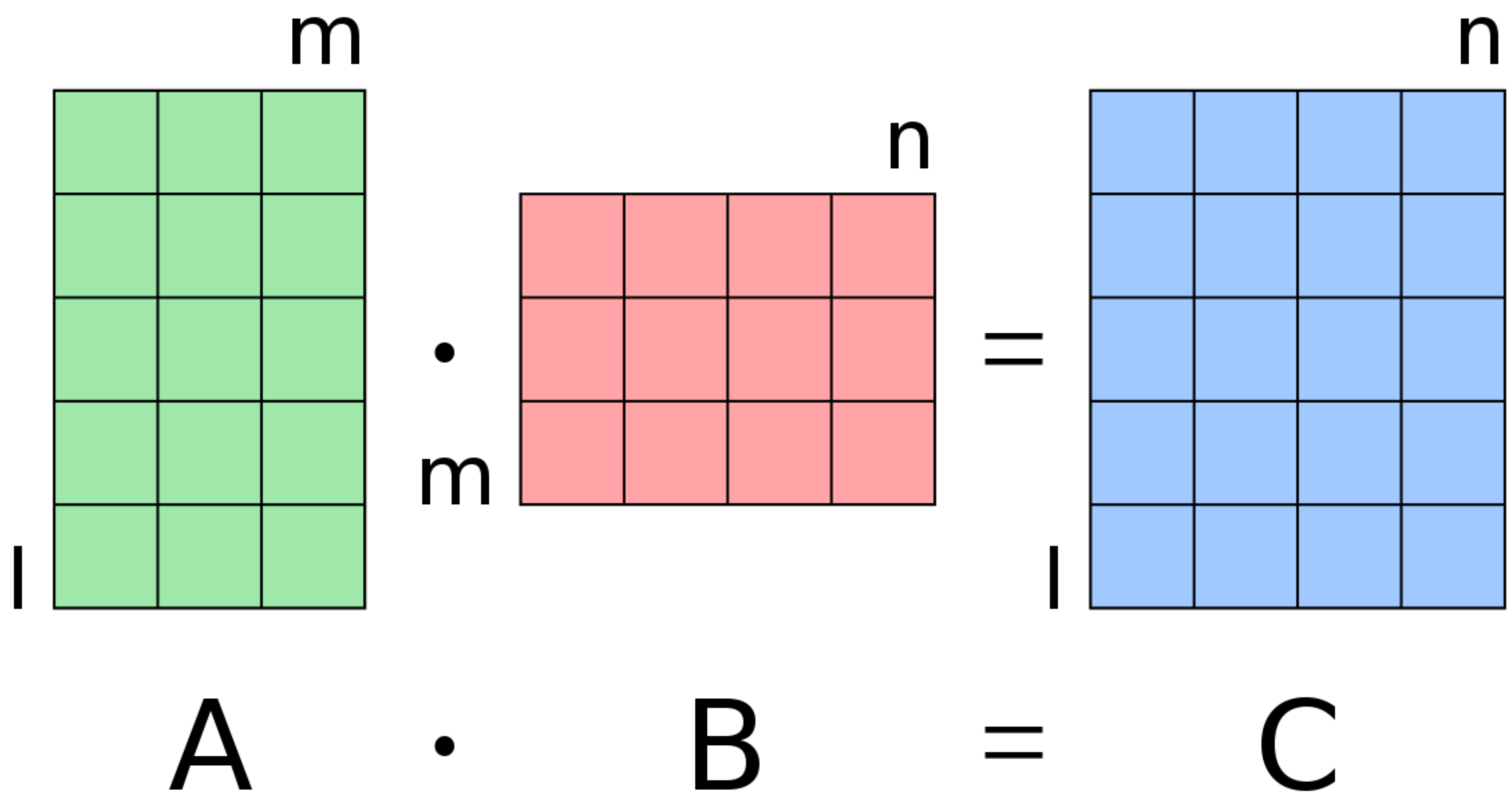
$$\mathbf{Y} = \sigma(\mathbf{W}\mathbf{X} + \mathbf{b}\mathbf{1}^\top)$$

- Sequentially perform three operations
  - MM multiply
  - MM addition
  - Activation

- **Question.** What is the heaviest?

Matrix multiplications

- Of course, the **matmul**
    - How many FLOPs for multiplying two matrices?



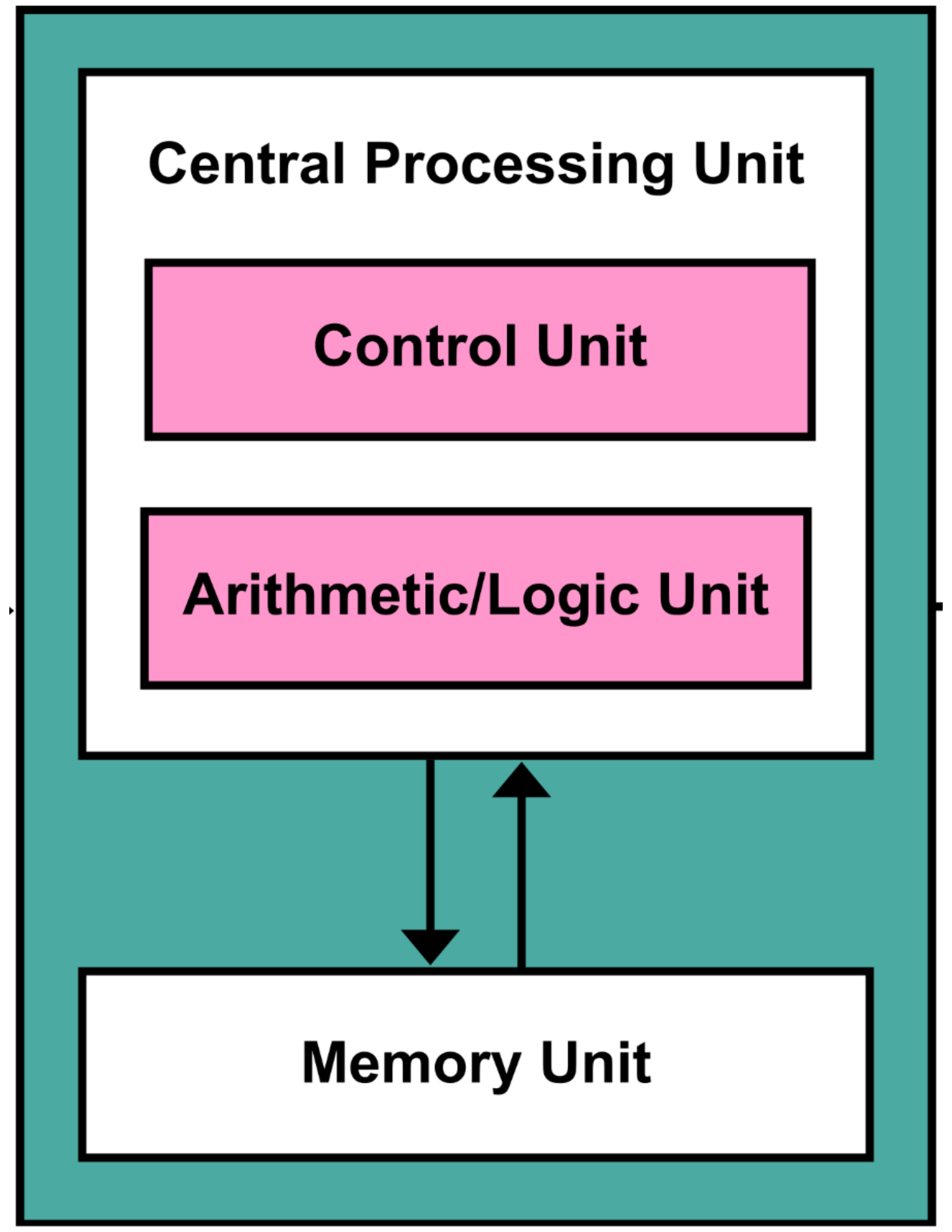
Matrix multiplications

- More broadly, GEMM (generalized matmuls) is at the center of deep learning comptuations
    - e.g., convolution, self-attention, gradient computation, …
    - A good reference: this NVIDIA doc

- **Question.** Other than FLOPs, what should we care about?

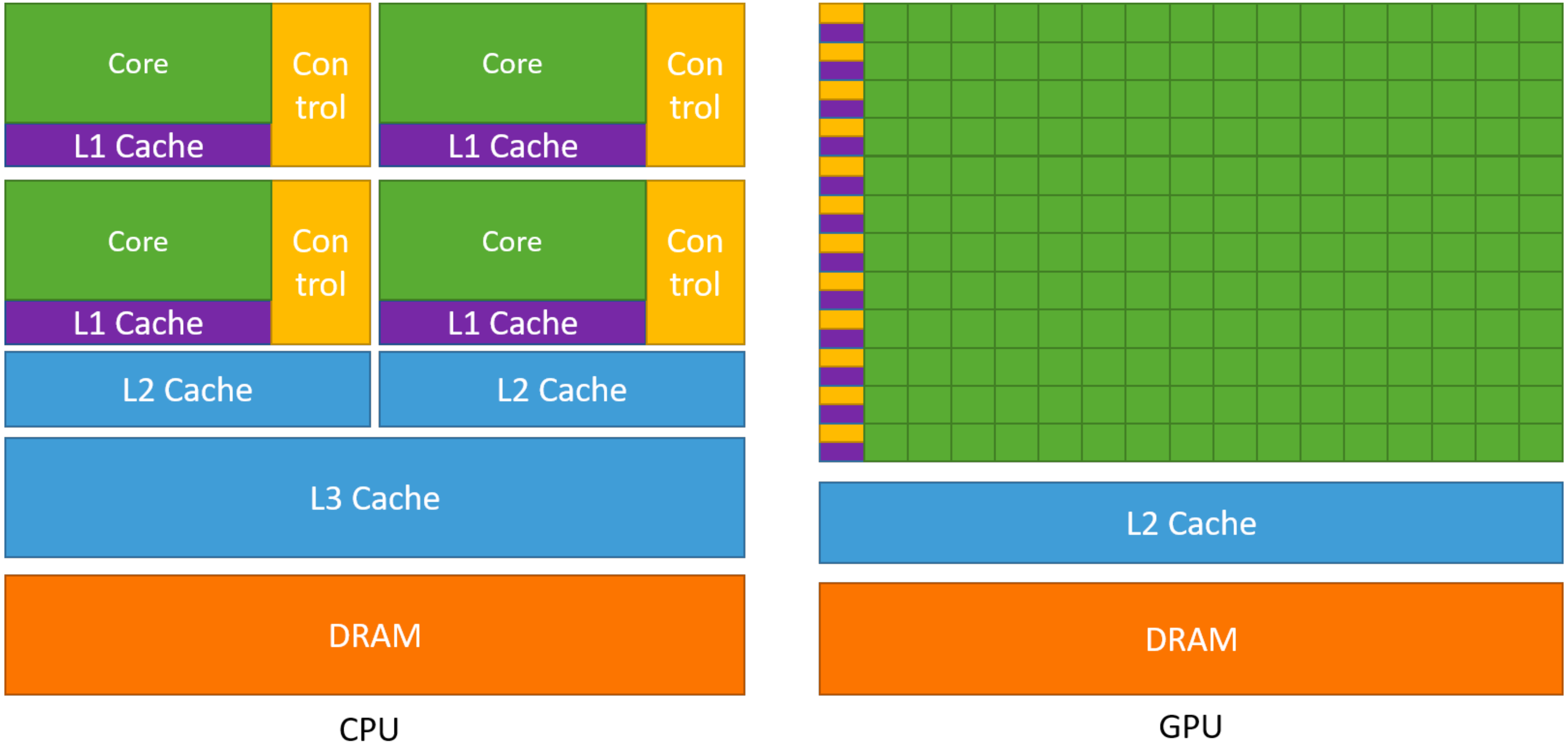
Memory issues

- Consider multiplying very large matrices
  - Suppose that we multiply two 4096x4096 matrices.
  - **Question.** If they are in FP32, how much memory do you need?
    - Can your L1 cache hold it?
    - Can your L2 cache hold it?
    - Need to care about data movement!



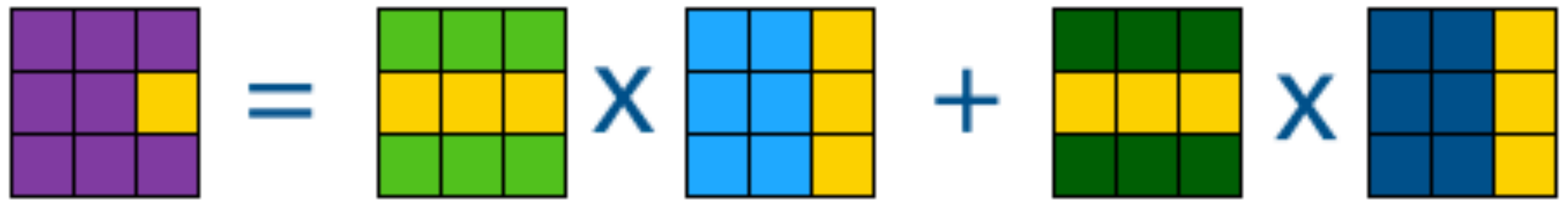
Memory issues

- DL HWs tend to have spacious memory & high memory bandwidths
- **Example**. NVIDIA H100 has ~60MB L2 cache & 80GB GPU memory, connected with several TB/s bandwidths

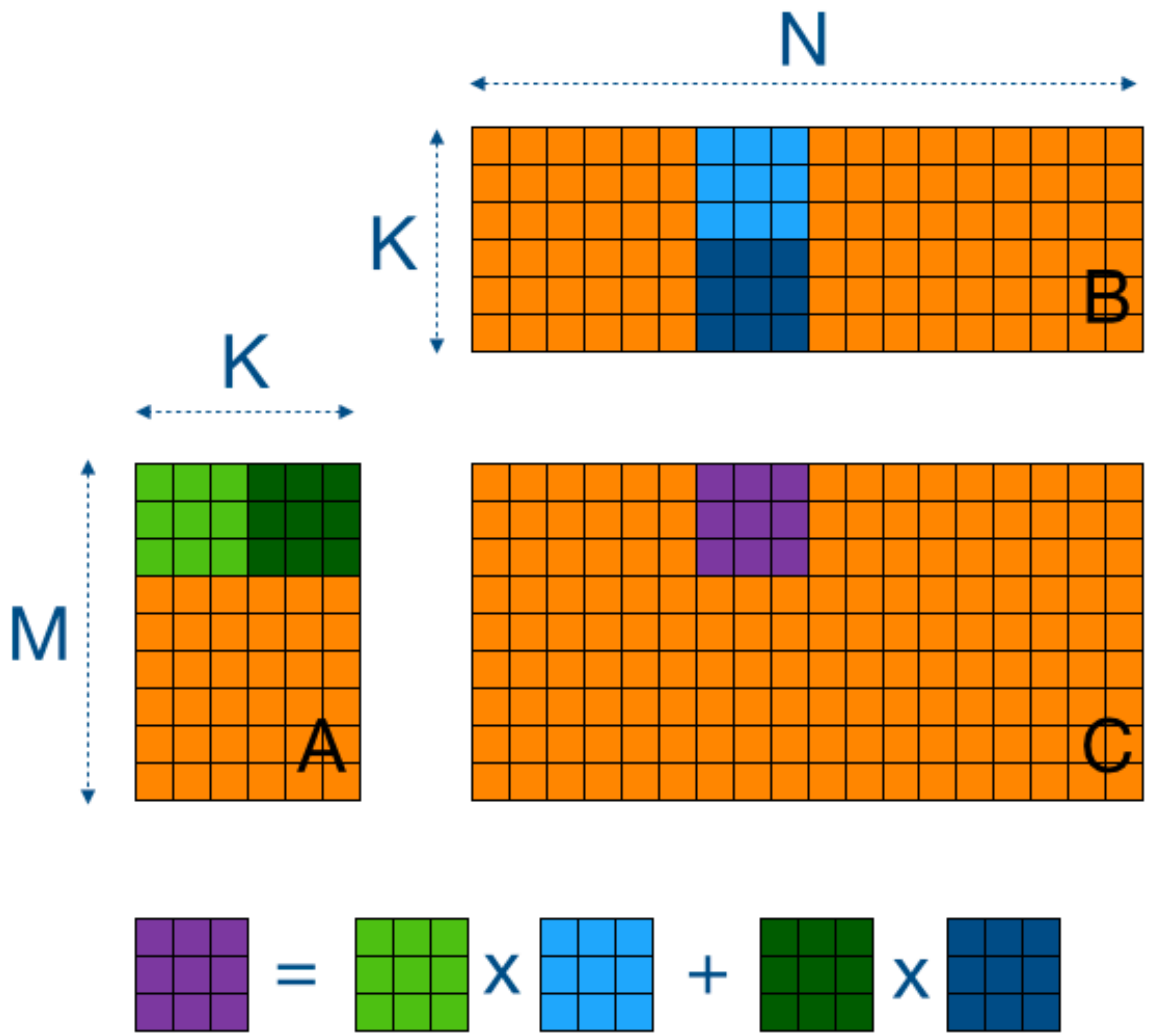


Matmuls on GPUs

- On GPUs, matmuls are grouped into **tiles**.
  - Better utilization of cores and memory
  - Allows us to re-use loaded elements:

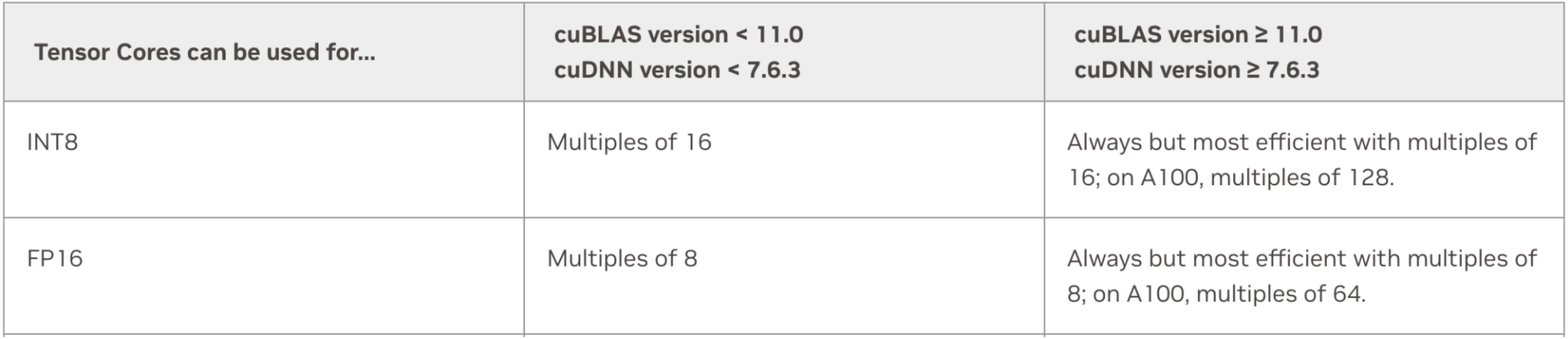


  - See also: link



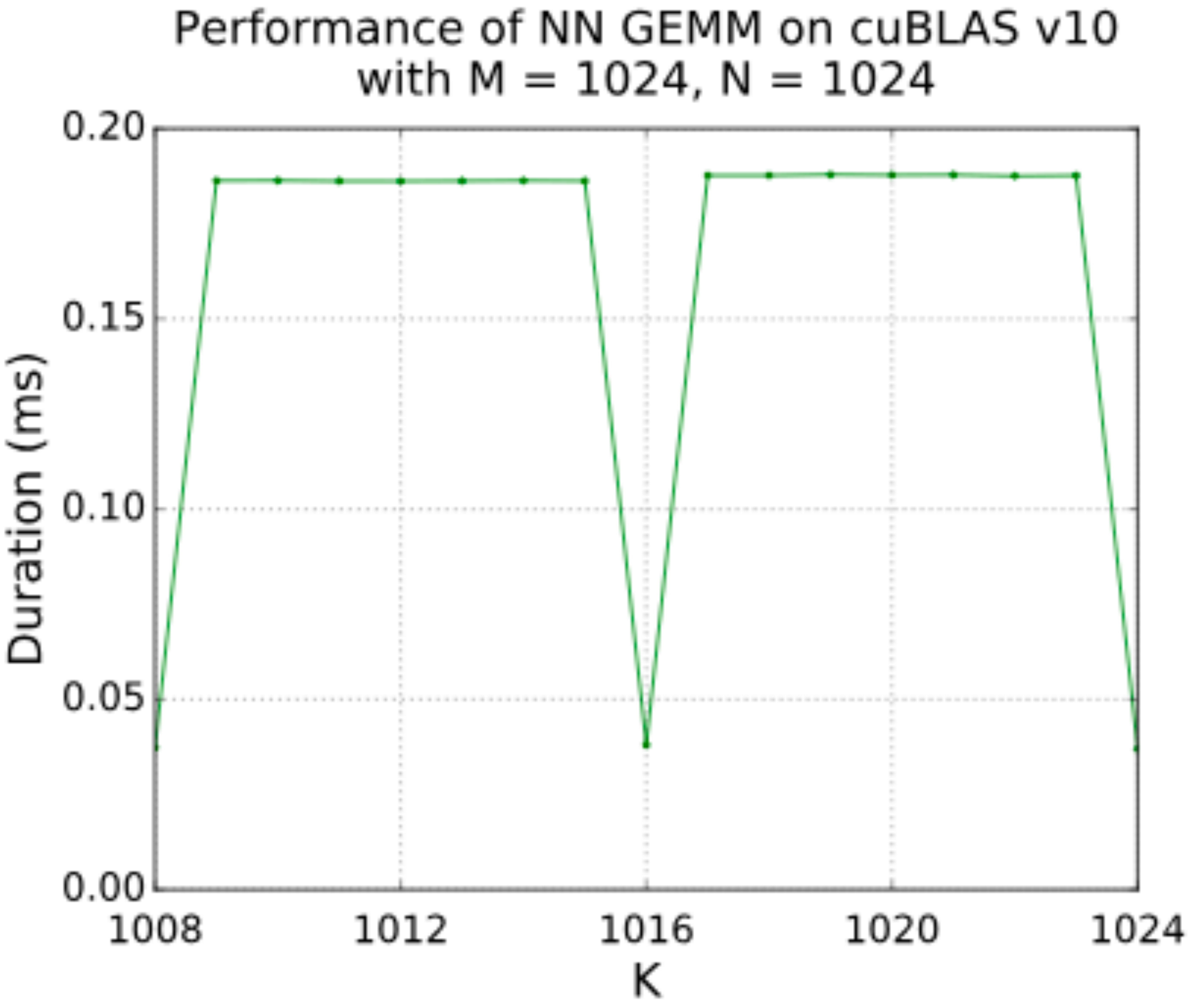
Side note: Tensor Cores

- To fully utilize tensor cores, pay attention to dimensions
  - Cannot get full benefits if dimensions are not regular.



Tensor Cores can be used for...	cuBLAS version < 11.0 cuDNN version < 7.6.3	cuBLAS version ≥ 11.0 cuDNN version ≥ 7.6.3
INT8	Multiples of 16	Always but most efficient with multiples of 16; on A100, multiples of 128.
FP16	Multiples of 8	Always but most efficient with multiples of 8; on A100, multiples of 64.

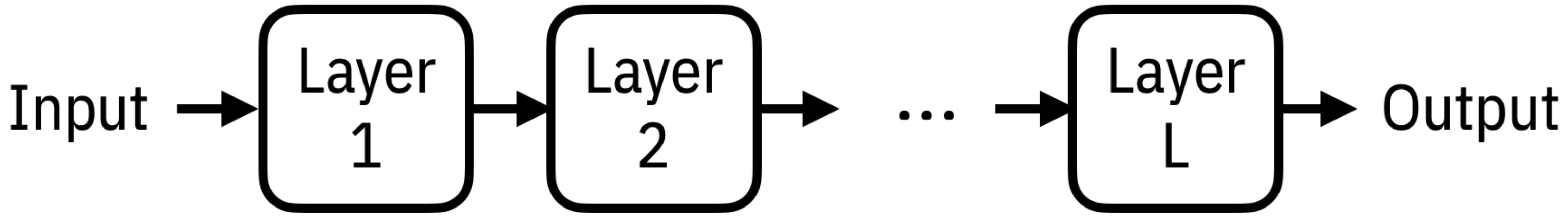
https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html



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

Memory vs. Compute

- Suppose that we have an MLP without biases

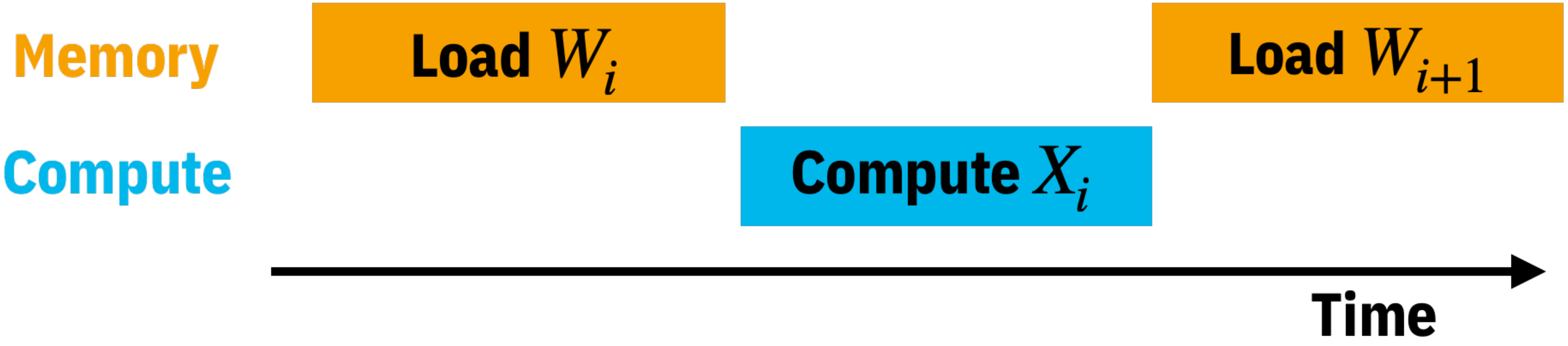


- An overly simplified sketch for computing matmuls:

```
# Currently $X_{i-1}$ is on cache
1. Load $W_i$ on cache                                  ← Memory bandwidth
2. Compute $X_i = \sigma(W_i X_{i-1})$                  ← Compute
3. Store $X_i$ on cache #free $X_{i-1}$
4. Repeat 1–3.
```

Memory vs. Compute

- Given limited on-chip memory, we can do a stupid thing:



- A nightmare in terms of **runtime**
  - More money (cloud GPU)
  - More electricity (idle energy)

Memory vs. Compute

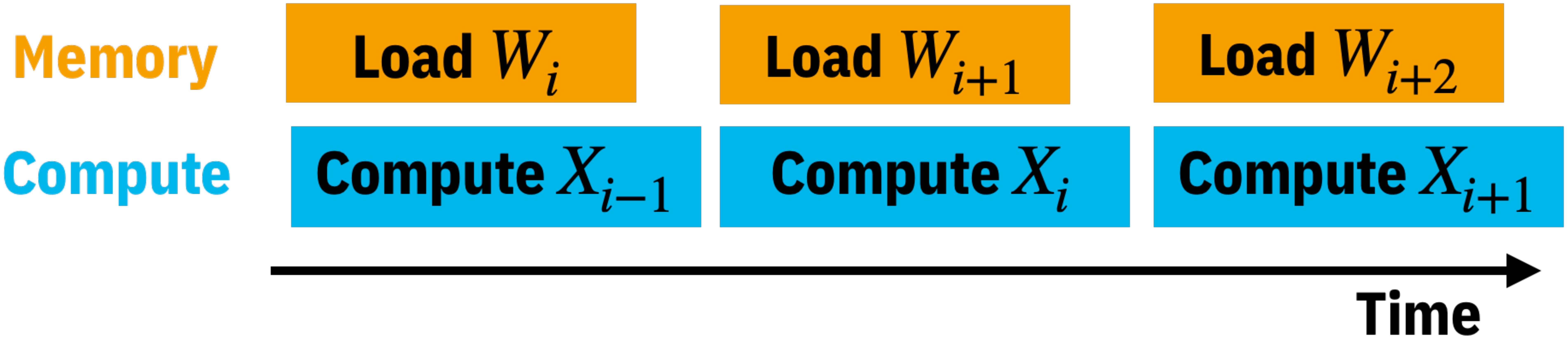
- With more spacious on-chip memory, we can be smarter (double buffering)

Currently X_{i-1} and W_i on cache

1. Compute $X_i = \sigma(W_i X_{i-1})$ + Load W_{i+1} on other cache

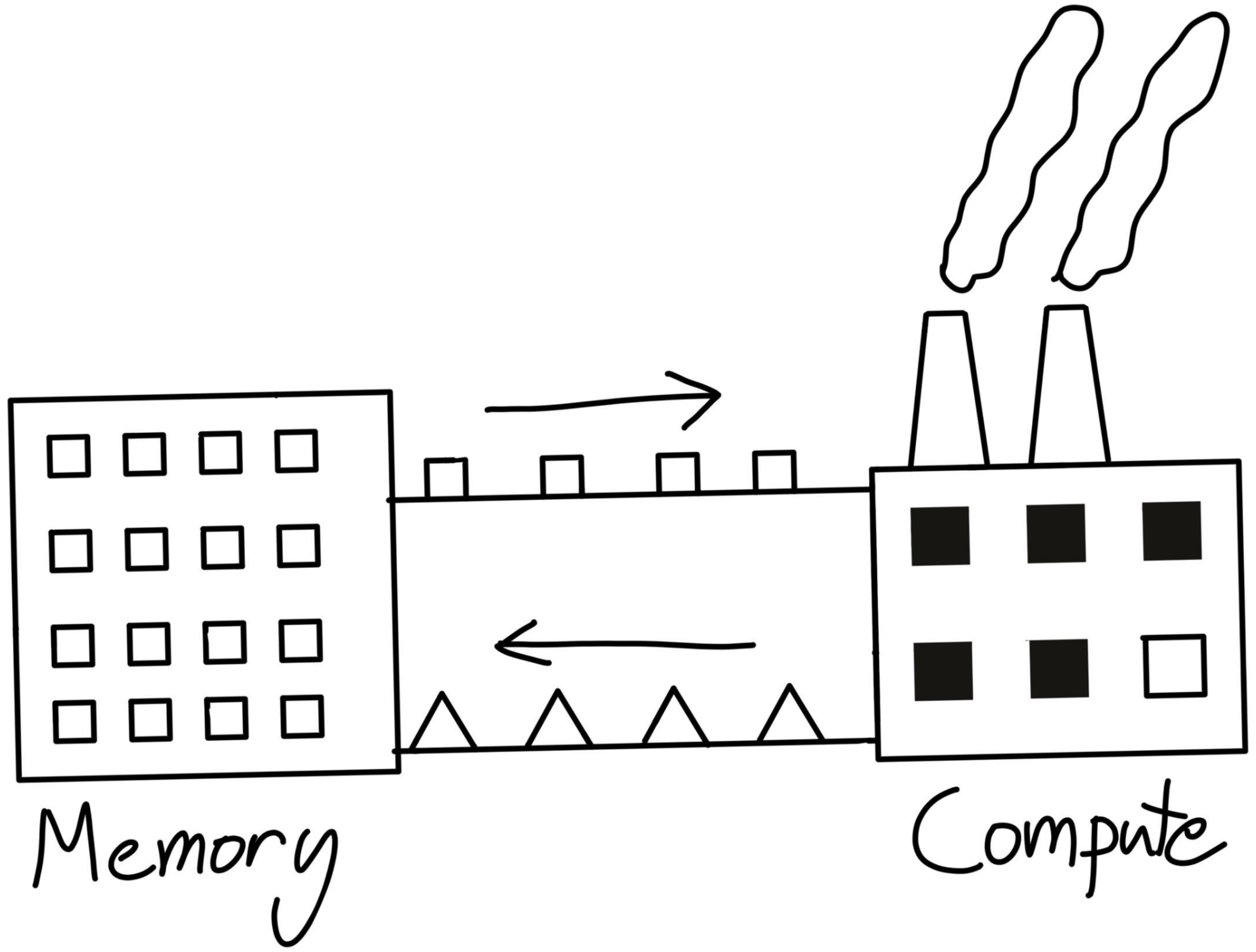
2. Store X_i on cache

3. Repeat 1—2.



Bottleneck?

- Either can be the bottleneck
    - Compute-bound
    - Memory-bound
- Depends on the hardware, model architecture, inference vs. training
    - Nice blog: link



Hardware

- Even with the same model, the runtime distribution differs a lot depending on which HW we use
  - Typically, GPUs spend much less time for compute

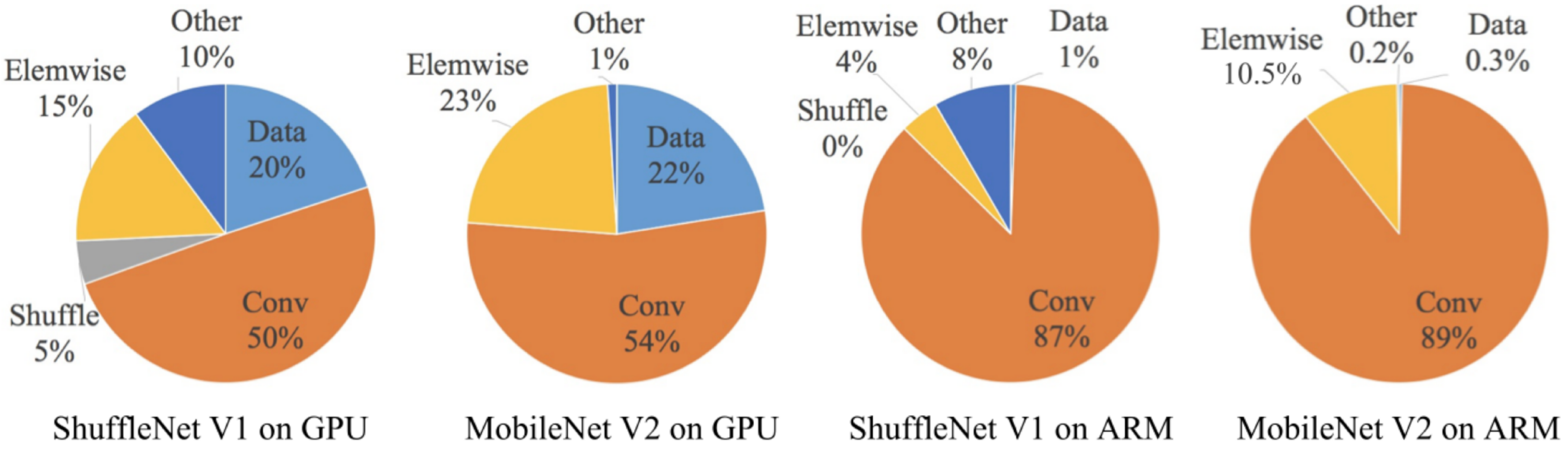
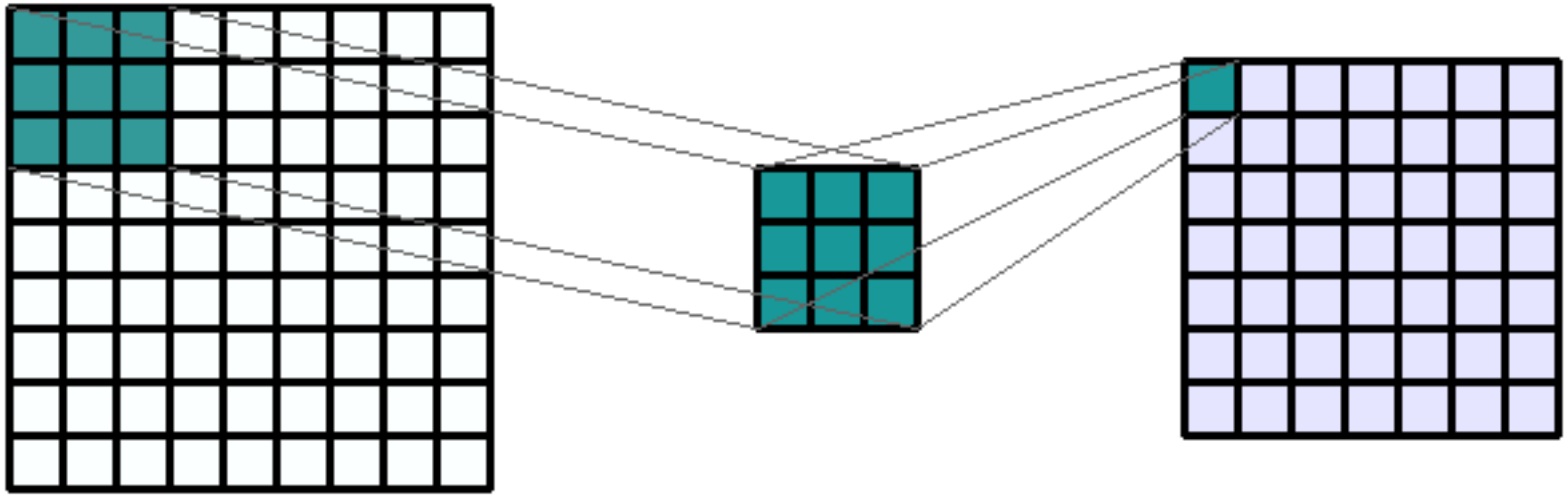


Fig. 2: Run time decomposition on two representative state-of-the-art network architectures, *ShuffeNet v1* [15] (1×, $g = 3$) and *MobileNet v2* [14] (1×).

Ma et al., "ShuffleNet V2: Practical Guidelines for Efficient CNN Architecture Design," ECCV 2018

Model architecture

- Consider the example of convolution
  - Parameter sharing, thus reduces the memory burden



Model architecture

- To map a $d \times d \times c$ input to an output of the same size:

Option 1) Fully-connected Layer

Params. Requires c^2d^4 parameters.

Compute. Requires $2c^2d^4$ FLOPs

Option 2) Convolutional layer (3×3)

Params. Requires $9c^2$ parameters

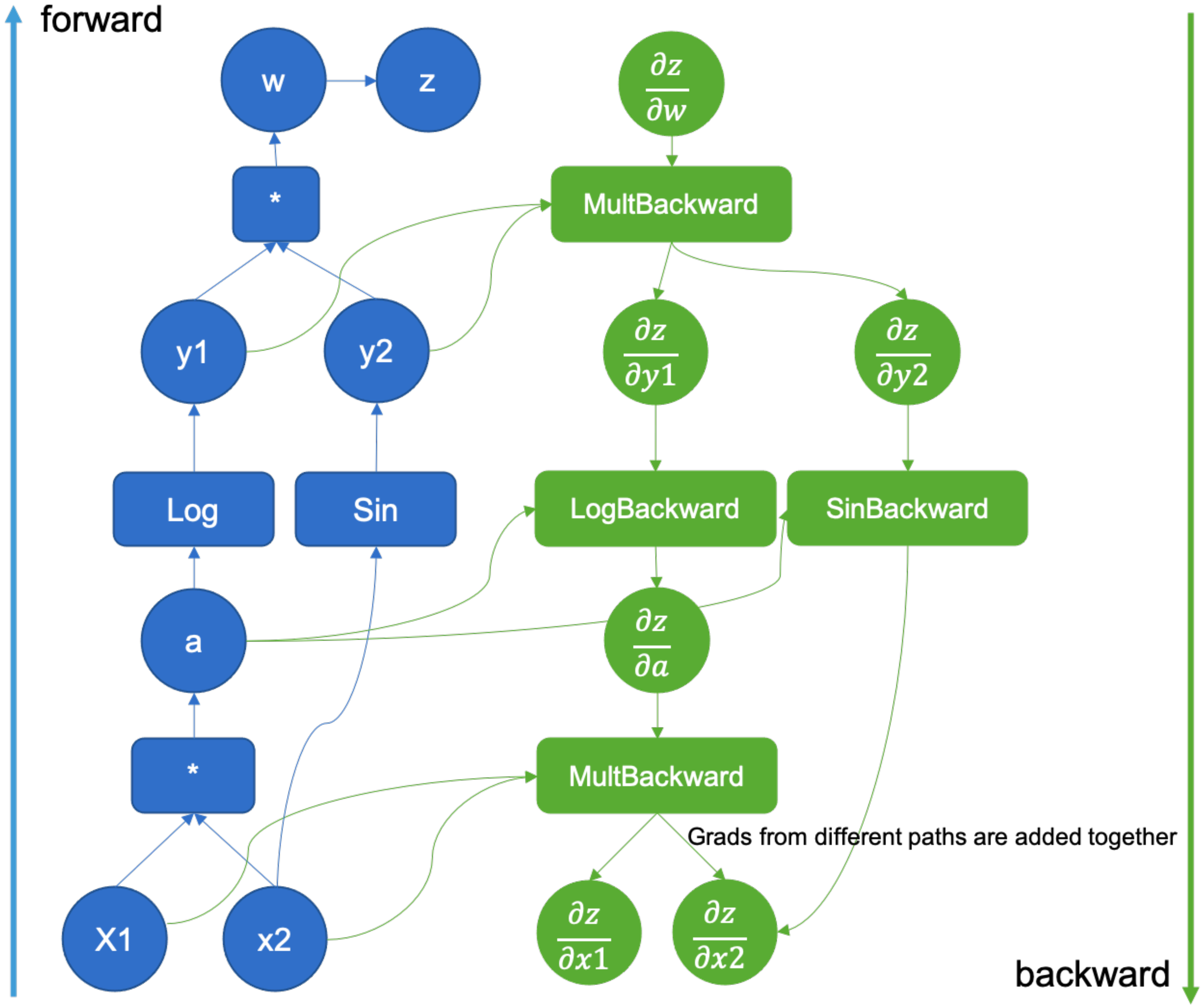
Compute. Requires $18c^2d^2$ FLOPs

- Saves d^4 in parameters, and d^2 in computations

Neural nets: Training

Training

- Much more complicated
- **Recap: Backprop**
  - Re-uses the activations computed during the forward
    - Less computation
    - More memory


forward
w
z
*
y1
y2
Log
Sin
a
*
X1
x2
$\frac{\partial z}{\partial w}$
MultBackward
$\frac{\partial z}{\partial y1}$
$\frac{\partial z}{\partial y2}$
LogBackward
SinBackward
$\frac{\partial z}{\partial a}$
MultBackward
Grads from different paths are added together
$\frac{\partial z}{\partial x1}$
$\frac{\partial z}{\partial x2}$
backward

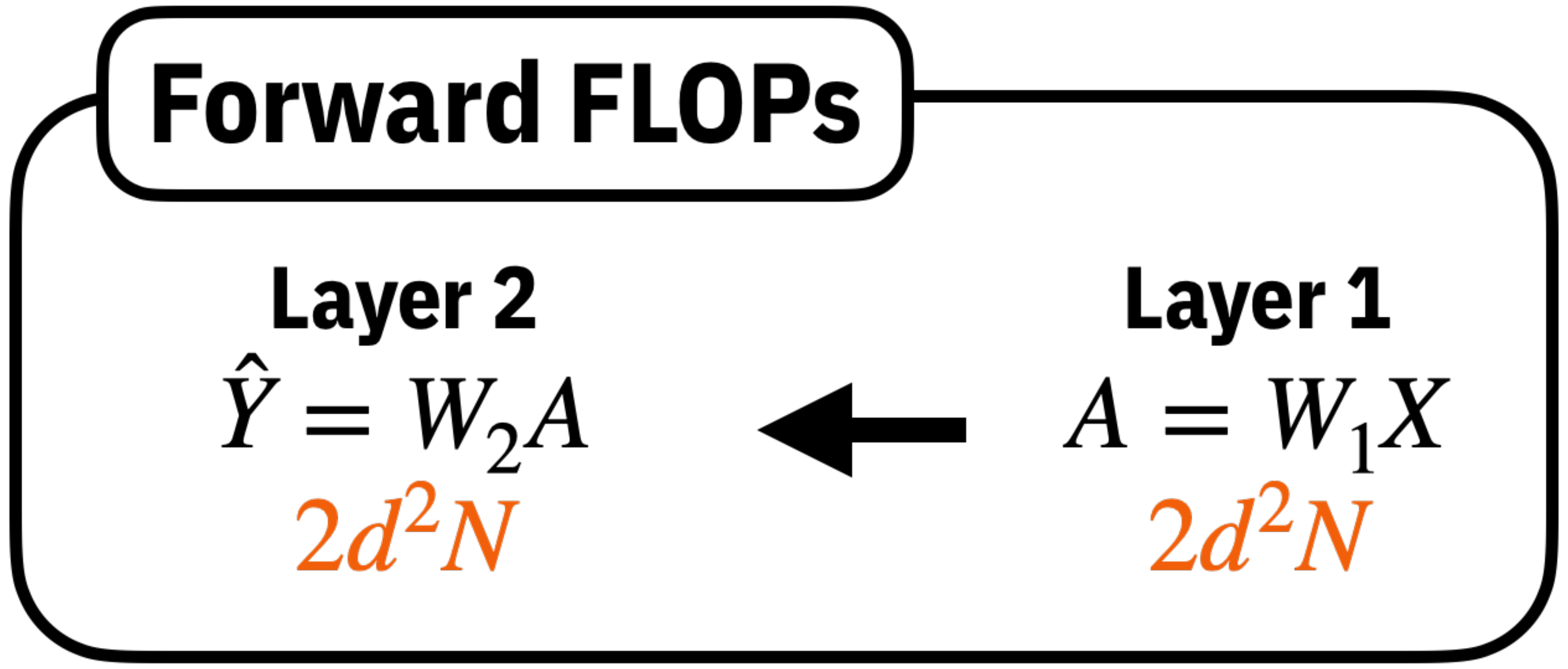
https://pytorch.org/blog/computational-graphs-constructed-in-pytorch/

```
RuntimeError: CUDA out of memory. Tried to allocate 200.00 MiB (GPU 0; 15.78 GiB total
capacity; 14.56 GiB already allocated; 38.44 MiB free; 14.80 GiB reserved in total by
PyTorch) If reserved memory is >> allocated memory try setting max_split_size_mb to avoid
fragmentation. See documentation for Memory Management and PYTORCH_CUDA_ALLOC_CONF
```

Example

- Consider a two-layer MLP

$$\hat{Y} = W_2 \sigma(W_1 X)$$



Forward FLOPs

Layer 2

$\hat{Y} = W_2 A$

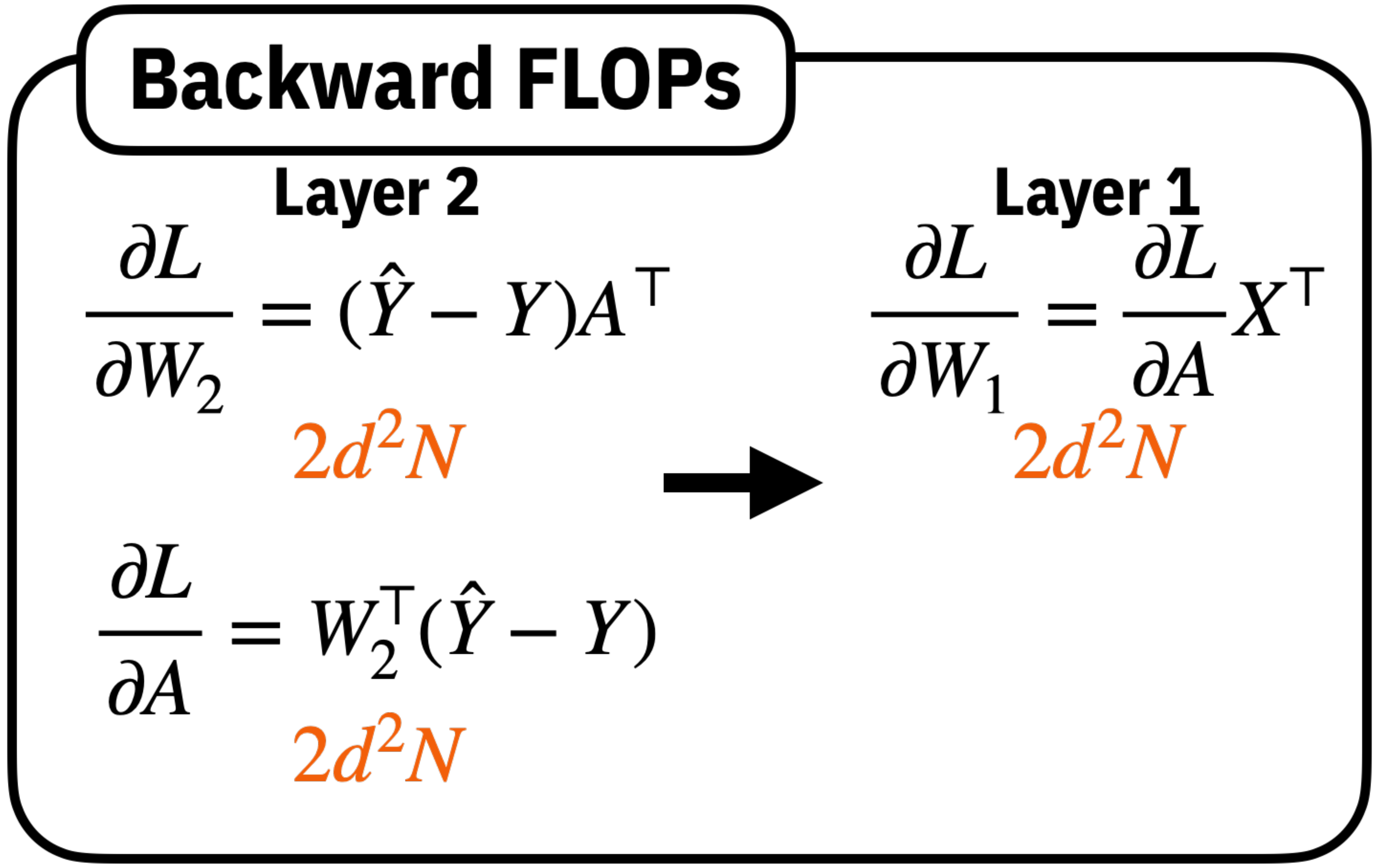
$2d^2N$

←

Layer 1

$A = W_1 X$

$2d^2N$



Backward FLOPs

Layer 2

$$\frac{\partial L}{\partial W_2} = (\hat{Y} - Y)A^\top$$

$2d^2N$

$$\frac{\partial L}{\partial A} = W_2^\top(\hat{Y} - Y)$$

$2d^2N$

→

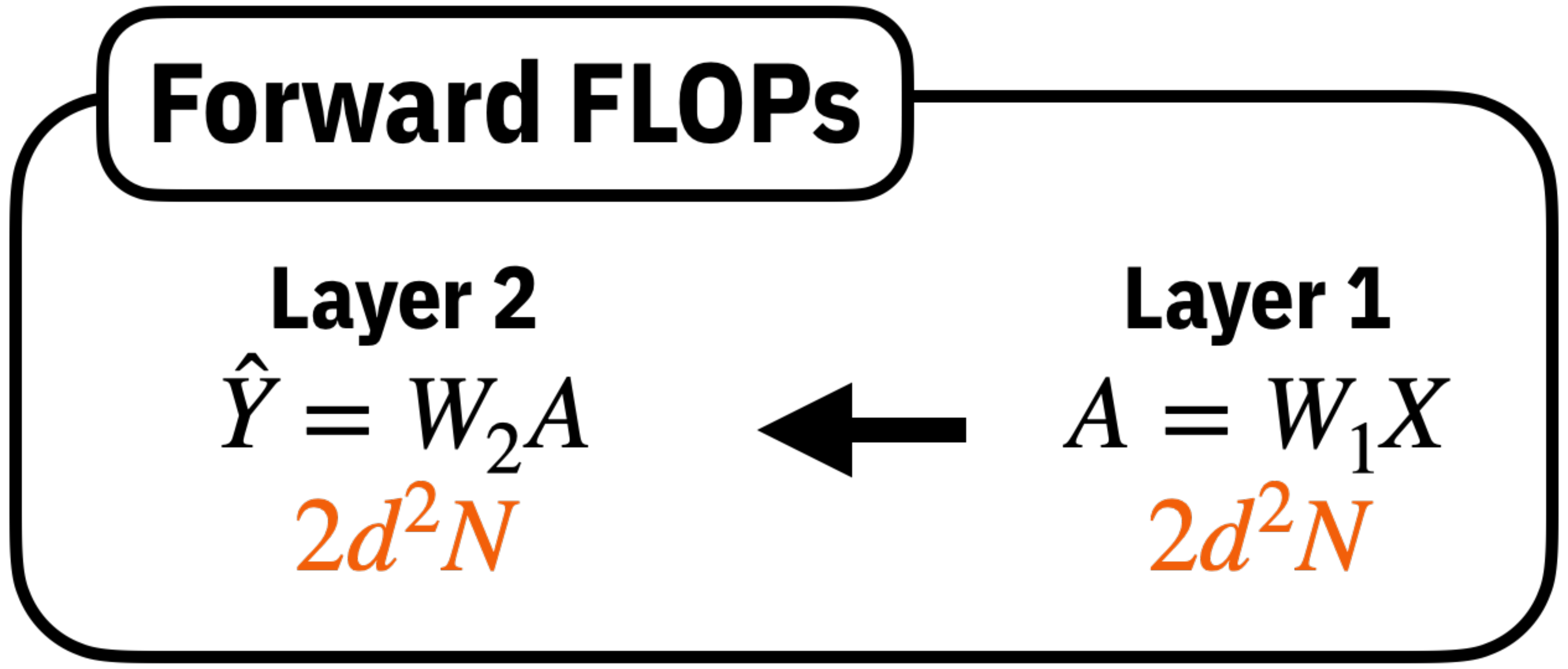
Layer 1

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial A} X^\top$$

$2d^2N$

Example

- Holding intermediate results in memory helps computing backward.
  - Sacrifice in memory
  - Advantages in computation



Forward FLOPs

Layer 2

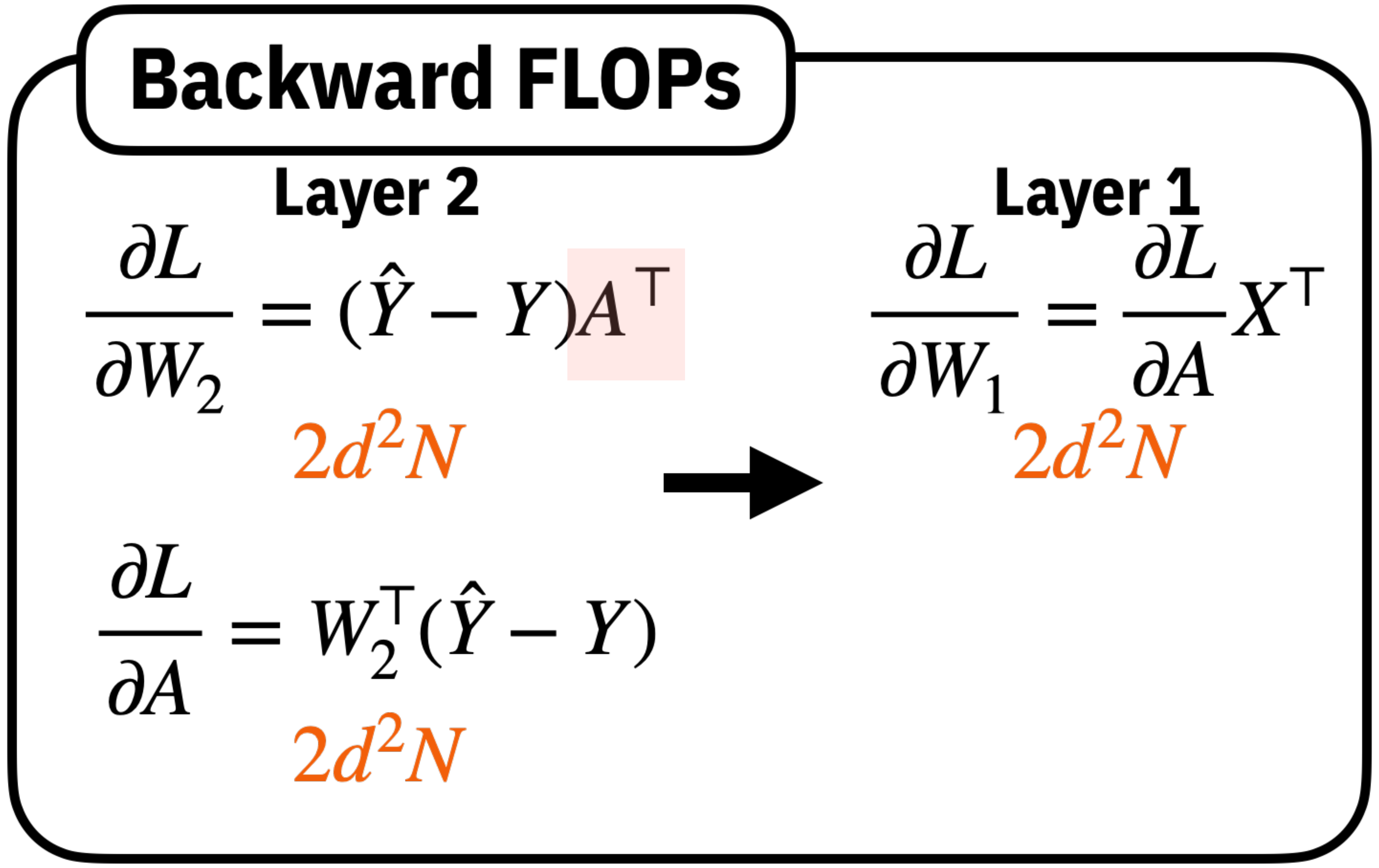
$\hat{Y} = W_2 A$

$2d^2N$

Layer 1

$A = W_1 X$

$2d^2N$



Backward FLOPs

Layer 2

$$\frac{\partial L}{\partial W_2} = (\hat{Y} - Y)A^\top$$

$2d^2N$

$$\frac{\partial L}{\partial A} = W_2^\top(\hat{Y} - Y)$$

$2d^2N$

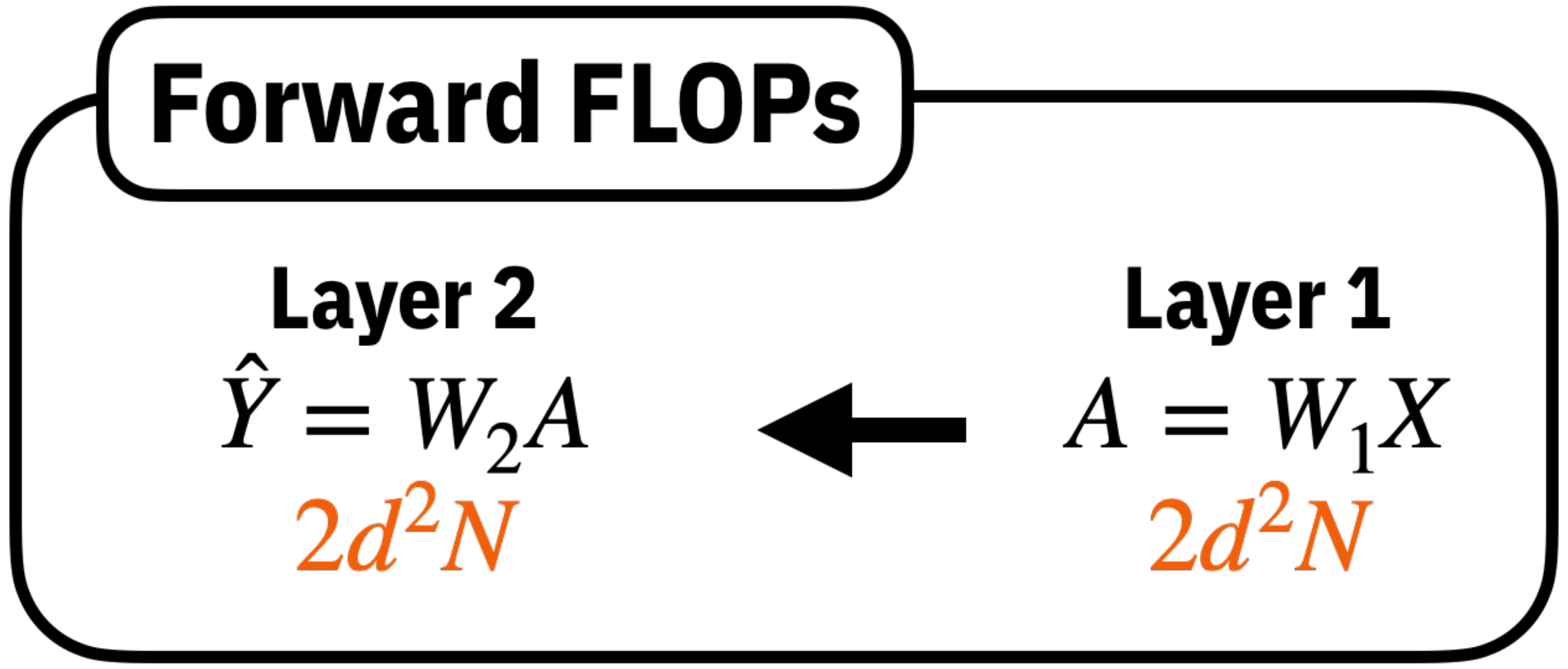
Layer 1

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial A} X^\top$$

$2d^2N$

Example

- **Note.** We need 2x more computation for BW than FW
    - useful rule of thumb



Forward FLOPs

Layer 2

$\hat{Y} = W_2 A$

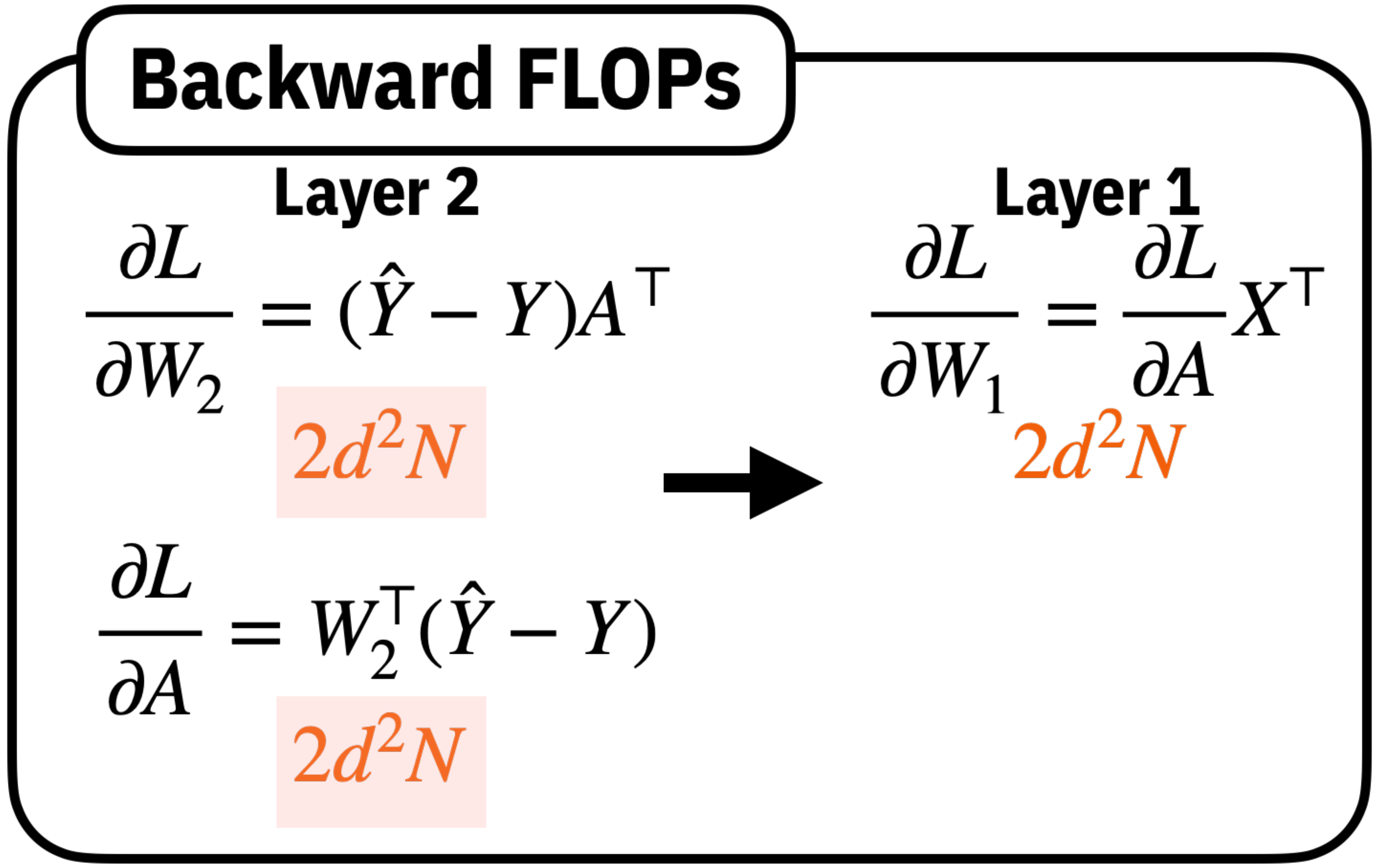
$2d^2N$

←

Layer 1

$A = W_1 X$

$2d^2N$



Backward FLOPs

Layer 2

$\frac{\partial L}{\partial W_2} = (\hat{Y} - Y)A^\top$

$2d^2N$

$\frac{\partial L}{\partial A} = W_2^\top(\hat{Y} - Y)$

$2d^2N$

→

Layer 1

$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial A} X^\top$

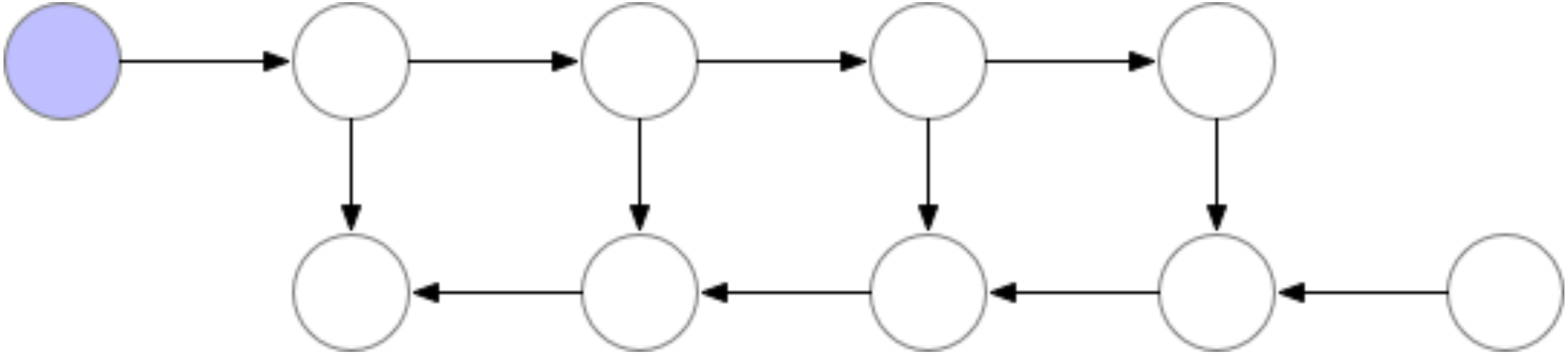
$2d^2N$

FLOP backward-forward ratios
backward-forward ratio
0.0
0.5
1.0
1.5
2.0
2.5
1-ratio
2.0-ratio
2.5-ratio
resnet18_1
resnet18_64
resnet18_128
resnet18_256
resnet34_1
resnet34_64
resnet34_128
resnet34_256
resnet50_1
resnet50_64
resnet50_128
resnet101_1
resnet101_64
resnet152_1
resnet152_64
vgg11_1
vgg11_64
vgg11_128
vgg13_1
vgg13_64
vgg13_128
vgg16_1
vgg16_64
vgg19_1
vgg19_64

Tradeoff

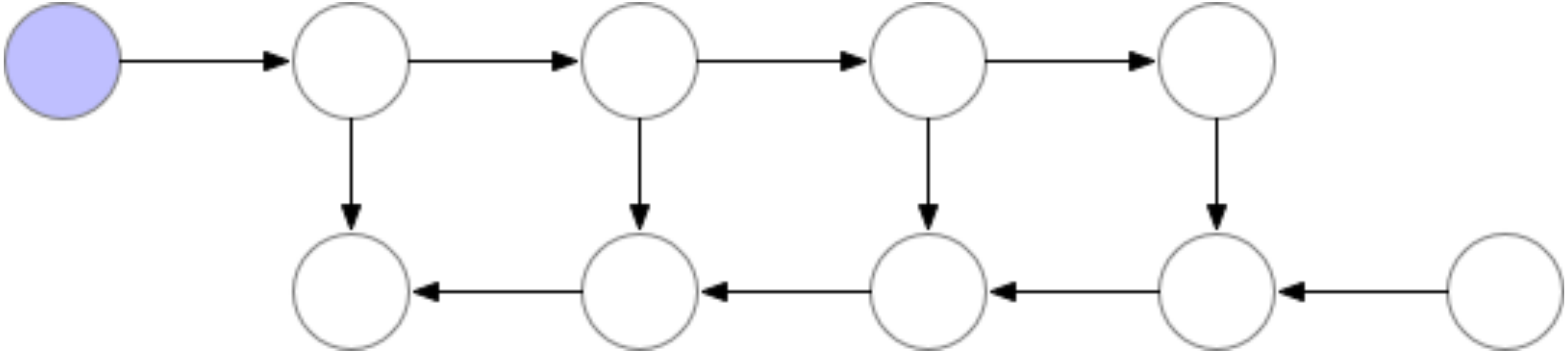
- Backprop trades memory for computation
- **Gradient Checkpointing.** Trades less memory for less computational benefit.
    - Discard “some activations,” and re-materialize whenever needed
        - Visual explanation here: https://github.com/cybertronai/gradient-checkpointing?tab=readme-ov-file

Vanilla Backprop



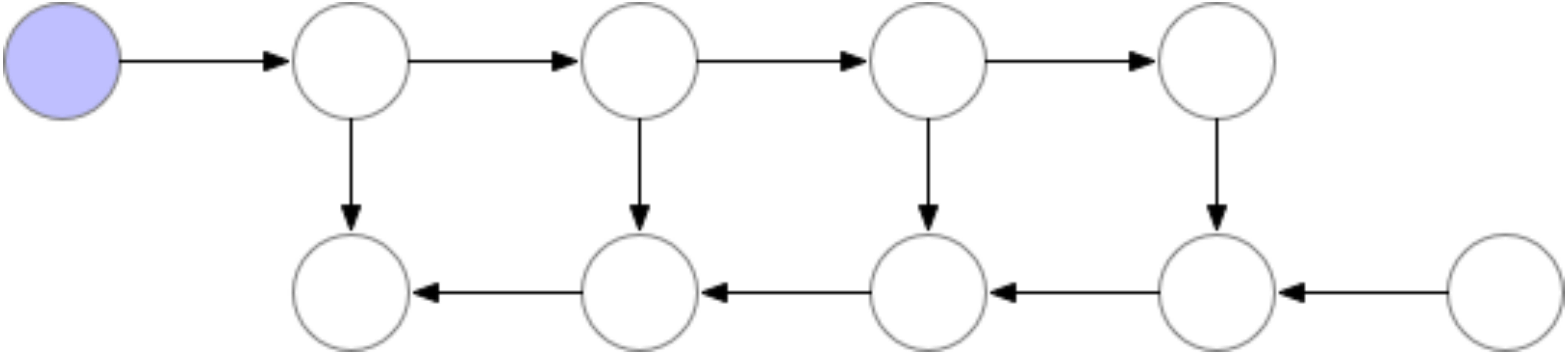
Memory-Poor

- If we can hold only three circles



With Gradient Checkpoints

- If we can hold slightly more circles



Remarks

- Discussion so far focuses compute vs. memory, in terms of runtime
  - Making AI faster
- This is not necessarily aligned with other notions of “efficiency”
  - Making AI smaller (#params, #bits)
  - Making AI greener (energy usage)
- Also, duration $\neq$ latency $\neq$ throughput
  - See excellent treatise on “the efficiency misnomer”

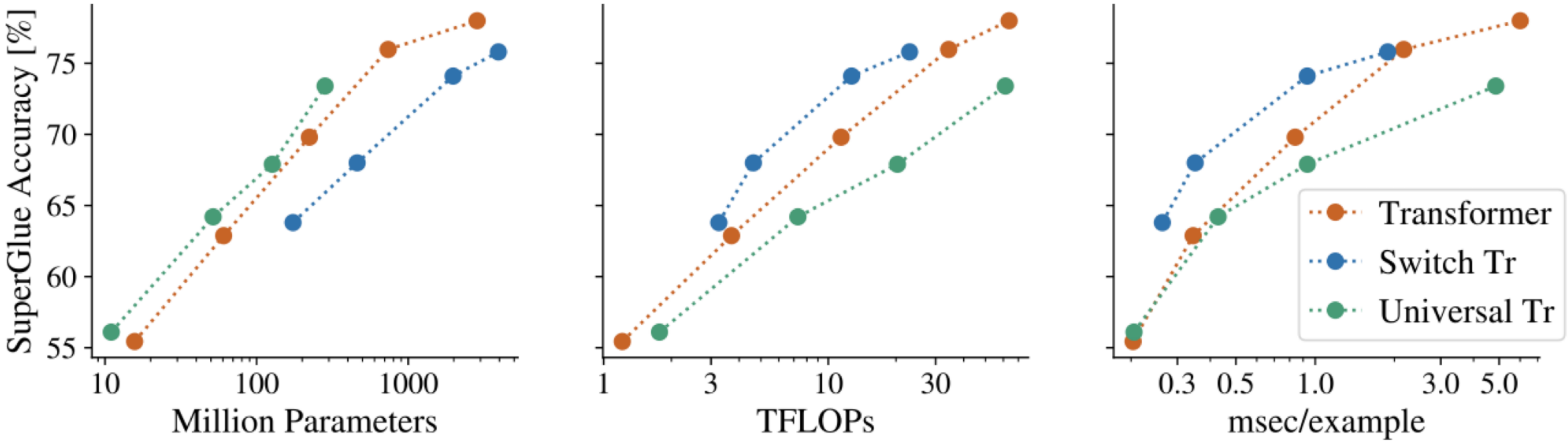


Figure 1: Comparison of standard Transformers, Universal Transformers and Switch Transformers in terms of three common cost metrics: number of parameters, FLOPs, and throughput. Relative ranking between models is reversed between the two cost indicators. Experiments and the computation of cost metrics were done with Mesh Tensorflow (Shazeer et al., 2018), using 64 TPU-V3.

Wrapping up

- **Today.** Recaps on basic ideas
- **Next.** Sparsity
- Ask yourself:
  - How does forward-mode autodiff compare with backward-mode, in terms of compute & memory?
  - How does depthwise convolution compare with vanilla?
  - How does Adam compare with SGD, in terms of memory?
  - How does GeLU compare with ReLU, in terms of memory during backprop?

That’s it for today 🙌