Lecture 6: Hardware and Hardware Algorithms I

Agenda:

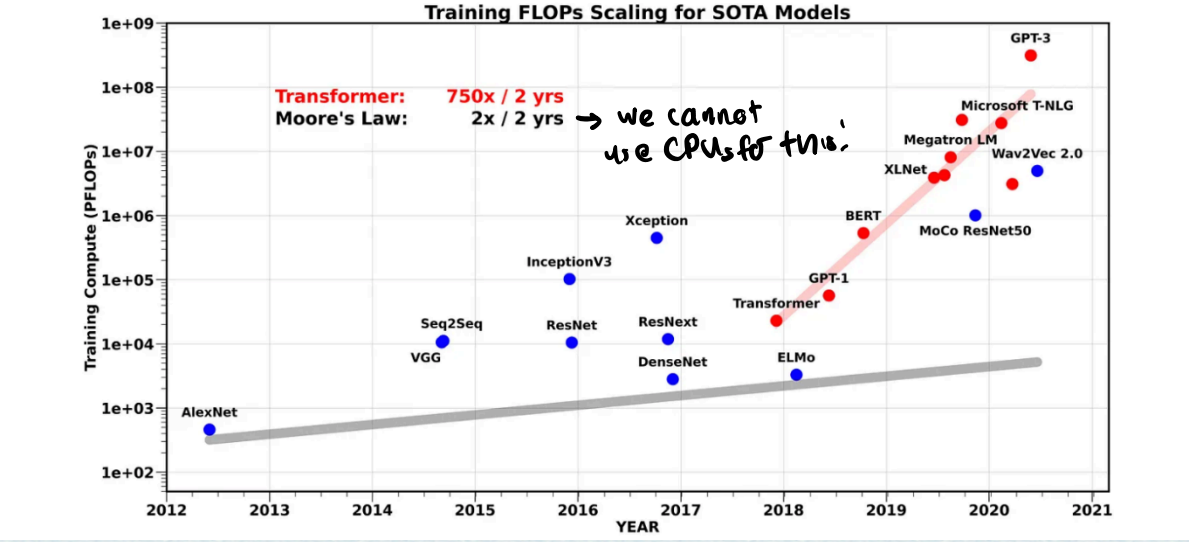
- CPU vs. GPU overview
- understanding perf.

* This lecture was entirely taken from Stanford's CS229s

.https://docs.google.com/presentation/d/14hK7SmkUNfSEIRGyptFD2bGO7K9sJOTnwjAVg3vgg6g/edit#slide=id.g286de50af37_0_237

* Rising Demand for compute that moore's law is not meeting

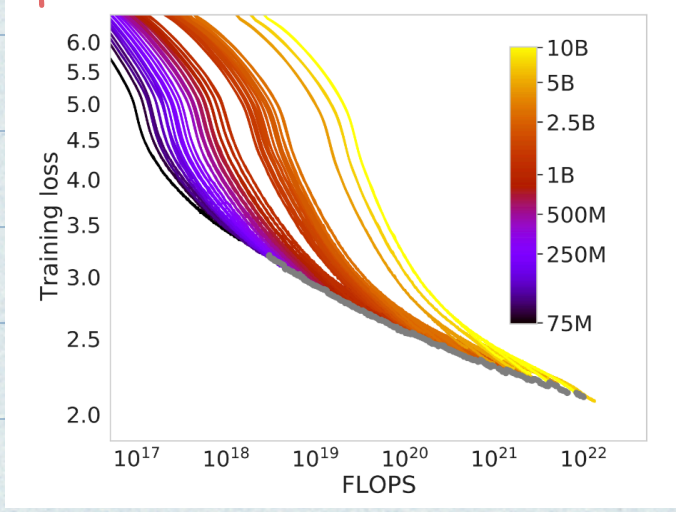
FLOP= floating pt ops



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

* The more FLOPs the better!

→ given a max # of FLOPs (x-axis) - what's the best loss we can get? (for arange of model sizes)

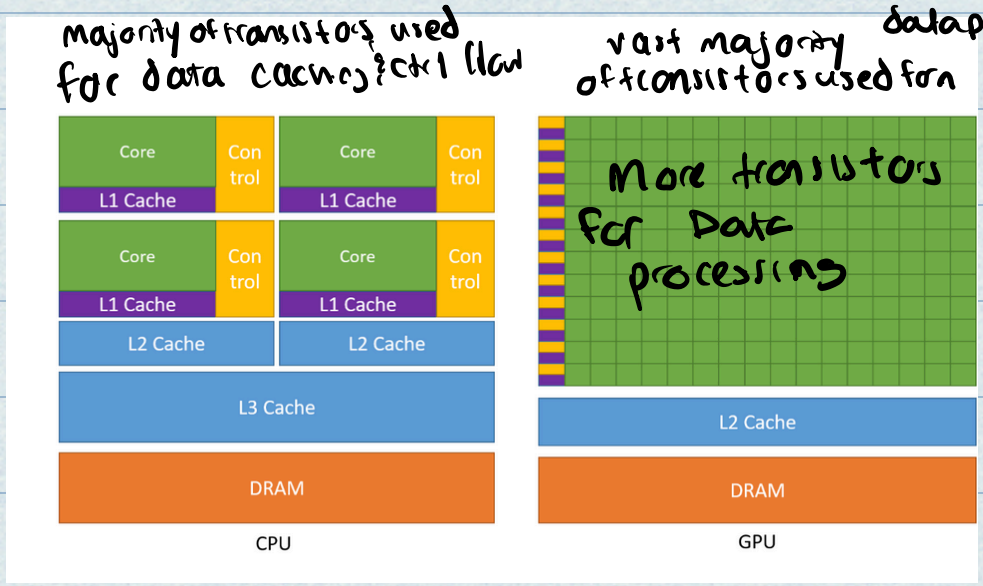


- also: high model size leads to lower loss

Chinchilla scaling laws - https://arxiv.org/pdf/2203.15556.pdf

* CPU vs. GPU

↳ note there are many dif. new AI accelerators, which we will briefly touch over later



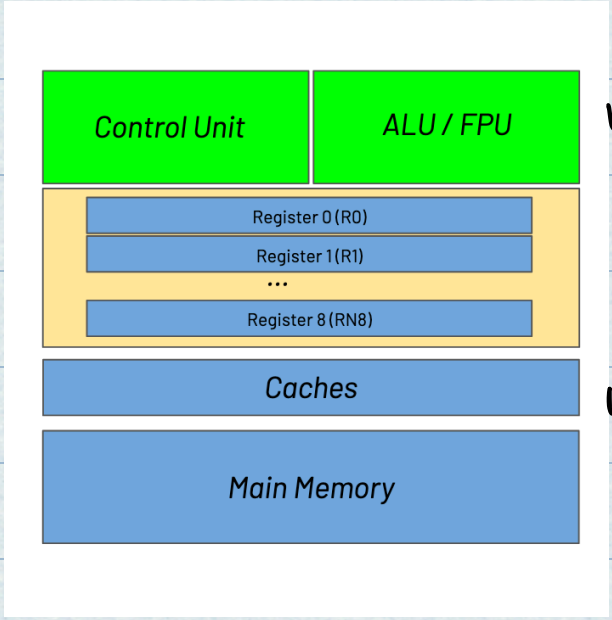
→ core building block of DL = vector and tensor multiplications and additions

→ GPU can do many of these in parallel

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#

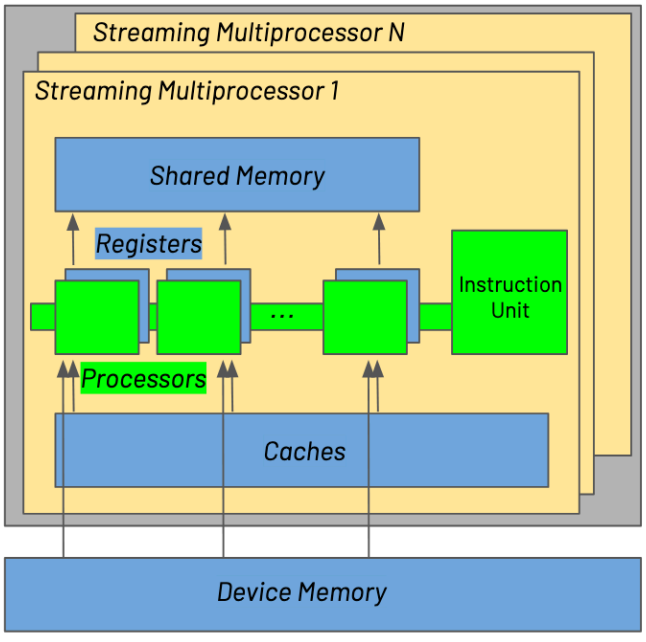
- CPUs are designed to execute a seq. of operations, called a thread, w/ minimum latency
- GPUs, in contrast, are designed to excel at executing thousands of threads in parallel
  - → generally has slower single-thread performance
  - → but achieves significantly high tput
- CPUs try to use lose data caches and complex flow control to hide / avoid long memory access latencies
- GPUs in contrast hide memory access latencies w/ computation.

- CPU (Central Processing Unit) - Designed for general-purpose computation



* Control Unit: fetches next instruction from memory (or cache) and directs arithmetic and floating point units
* ALU / FPU: performs bitwise instructions on integer and floating point #s
* Registers: for next instruction, we can read input units in and write intermediate values back out
* Caches / main memory: much larger capacity than registers, used to store instructions and data

- Graphics Processing Unit (GPU) - for parallelizable programs

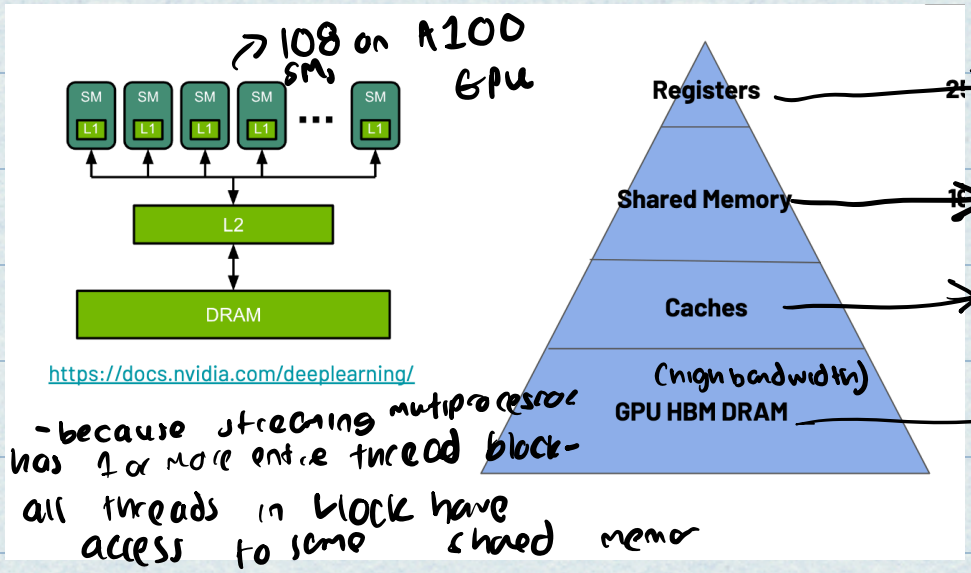


* streaming multiprocessors: each has one instruction unit that controls many processors that run the instruction in parallel
* registers, shared memory, caches, device memory: for reading and writing intermediate data

(more on execution later) - threads are divided into blocks, each is given 1 or more threads to execute blocks

* GPU memory hierarchy:

40 GB A100



- because streaming multiprocessor has 1 or more entire thread block- all threads in block have access to same shared memor

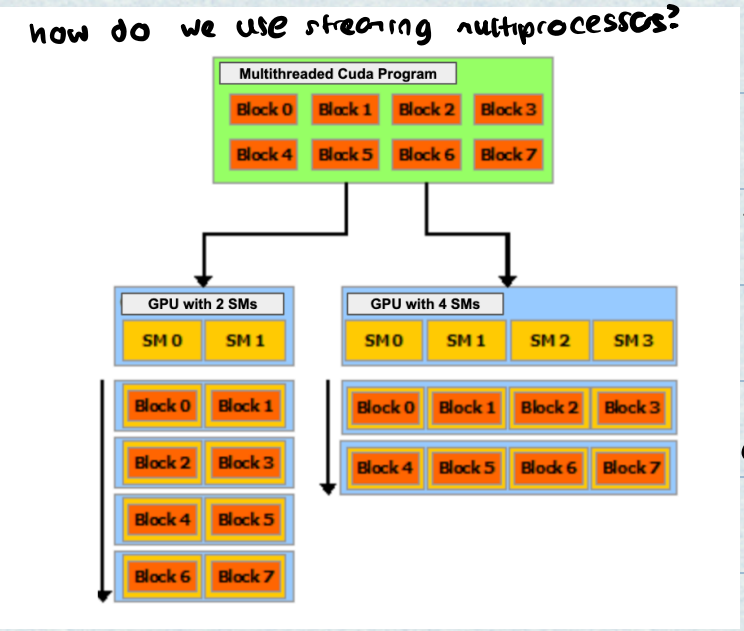
* on chip SRAM (shared mem) is ~9x faster than HBM DRAM but ORDERS of magnitude smaller

* It is key to use on-chip shared memory effectively when designing hardware-aware algorithms

- big difference between CPU and GPU:
  - in GPU programmer has more control of what ends up in each streaming multiprocessor's shared memory
  - in contrast CPU uses caching hierarchy transparently → programmer can only control data layout and order of instructions in program

* Programming Nvidia GPUs:

how do we use streaming multiprocessors?



→ organize threads into blocks

→ blocks organized into a GRID

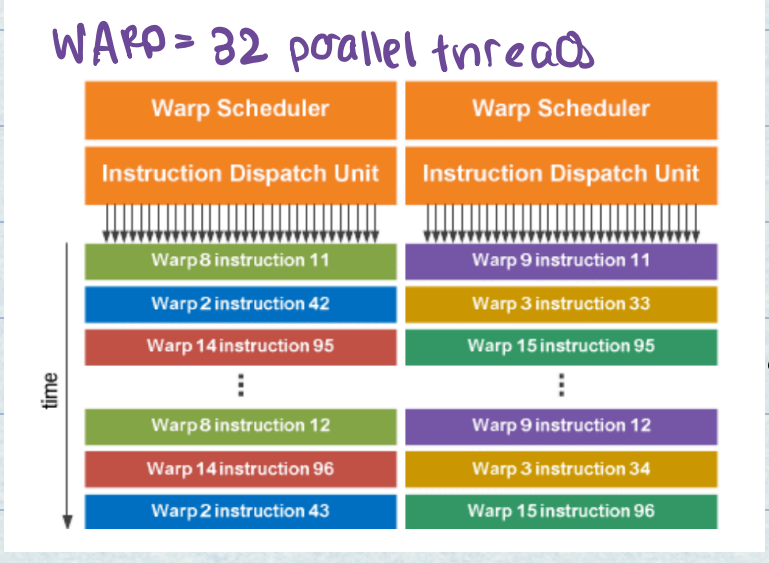
→ blocks of grid are distributed to SMs w/ available execution capacity

→ multiple thread blocks can execute concurrently on a single SM

https://docs.nvidia.com/cuda/cuda-c-programming-guide/

→ threads in a block can ALSO execute concurrently on a single SM

* SIMT (Single-instruction, multiple threads)



- multiprocessor designed to execute hundreds of thread CONCURRENTLY

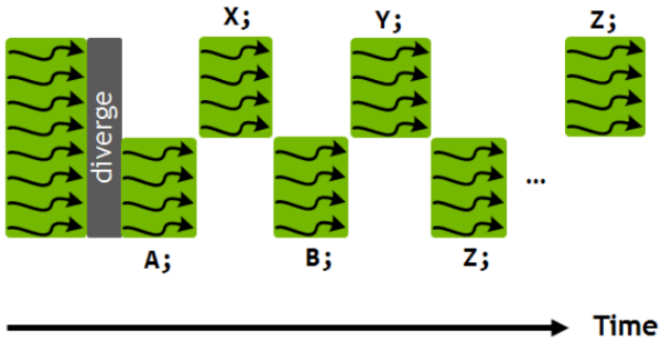
- each multiprocessor groups 32 threads within a thread block into a WARP

- within a warp: individual threads START at same address, but each thread has its own instruction address counter and register state - to can branch independently

↳ but at each time-unit, SM can only execute one instruction at a time, so

optimal for there to be no branching:

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



How an if-else is executed in the Volta implementation of SIMT.

https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

(Figure 22)

→ within an SM, all execution context (program counters, registers, etc) are maintained on chip until warp is done executing - so "switching" btwn dif. warps on same SM has no cost

↳ at each timestep warp scheduler will try to select a warp that has next thread ready to run

Perf

* Part 2 of Lecture: Thinking About

- in alg. class, we think about algorithms in terms of asymptotics:

↳ Strassen's algorithm for matrix multiply takes $O(N^{2.8074})$

↳ naive matrix multipy $O(N^3)$ for NxN matrices

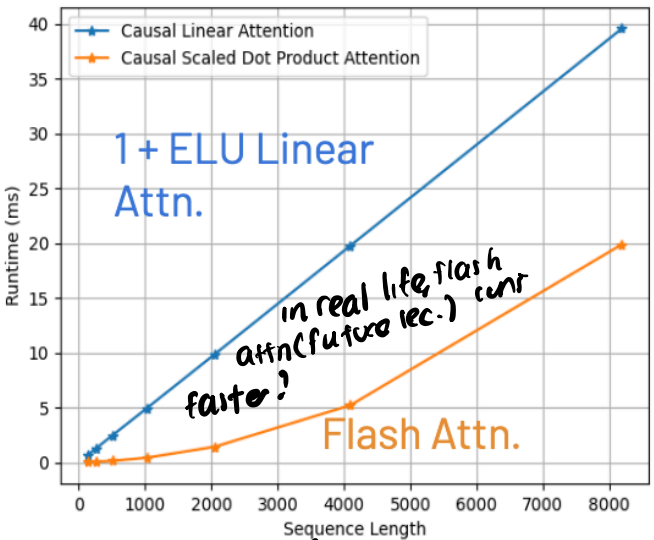
- but big O notation ignores:
  - constants that can make a lose dif. for dif. problem sizes
  - implementation specific details! (these matter A LOT in real life)
- an algorithm could be ASYMPTOTICALLY BETTER but have worse perf. if not suited for specific hardware

*concrete example: Algorithmic scaling ! = real world perf.

→ 1+ELU* is a fancy new attn algorithm w/ better asymptotic scaling than normal (softmax) attn.

Method:	complexity (input seq. length)
Softmax	$O(N^2 d)$
1+ELU	$O(Nd^2)$

↓ dimension of Q/K/V matrices



(N)

Taken from cs299s lecture, original references:

* Transformers are RNNs: Fast Autoregressive Transformers with Linear Attention. Katharopoulos et al., ICML 2020. https://linear-transformers.com/
**Scaling without linear attention cuda kernel. Custom cuda kernel makes linear attention go faster.

Credit: Michael Zhang https://michaelzhang.

* How do we THINK about performance ???

- alg. could be good asymptotically but not suited for specific hardware

*HARDWARE UTILIZATION (consider):

↳ - theoretical peak perf.

- memory bandwidth of HW

- try to find FRACTION of peak theoretical perf. attained by an alg

- use this info to decide HOW to improve perf.

* Theoretical Peak Performance

- In any HW (CPU or GPU), we:

- move items from memory to a processing unit

- perform some computation

- move items back to memory

- to get peak perf, we need the following HW properties:

* memory bandwidth → max # of items that can be moved to/from memory to/from processor in each second

* compute bandwidth: max # of ops that can be done in processor per second

* Example peak perf. for Nvidia A100s:

	A100 80GB PCIe	A100 80GB SXM
FP64	9.7 TFLOPS	
FP64 Tensor Core	19.5 TFLOPS	
FP32	19.5 TFLOPS	
Tensor Float 32 (TF32)	156 TFLOPS | 312 TFLOPS*	
BFLOAT16 Tensor Core	312 TFLOPS | 624 TFLOPS*	
FP16 Tensor Core	312 TFLOPS | 624 TFLOPS*	
INT8 Tensor Core	624 TOPS | 1248 TOPS*	
GPU Memory	80GB HBM2e	80GB HBM2e
GPU Memory Bandwidth	1,935 GB/s	2,039 GB/s

→ many logs

flops vs memory BW

https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/

* compute vs. memory bound:

- T_{mem} = time spent accessing memory
- T_{math} = time spent doing math (on compute)
- when memory / IO is overlapped, total execution time is:

$$\max(T_{mem}, T_{math})$$

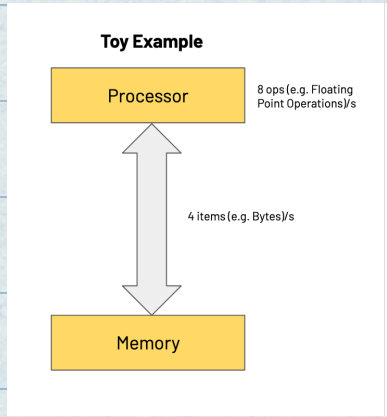
* compute bound: when $T_{math} > T_{mem}$
* memory bound: when $T_{mem} > T_{math}$

* Peak Bandwidth vs. Peak Compute mismatch:

Here:

compute BW = 8 ops/s

mem BW = 4 items/s



- suppose we implement the

following ads:

1. Load 8 items from memory into processor.
2. run 1 op on each item
3. write all 8 outputs back

- if compute overlaps w/ I/O, how long does this take to run?

overlap

Total time = 4: 2s (load) + ~~2s (compute)~~ stall + 2s (write)

→ second 0-1: first 4 item arrive

→ 1-2: 4 items computed on, 2nd 4 arrive

→ 2-3: first 4 written back, 2nd 4 computed

→ 3-4: 2nd 4 written back

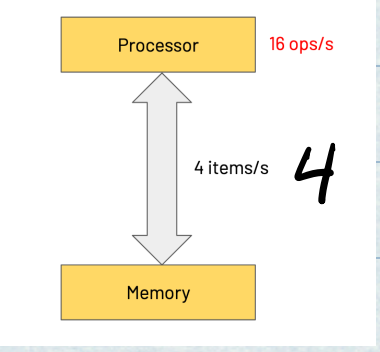
- for sufficiently large (# of) items:

- 4 items arrive per s to processor
- processor can do 8 ops/s, but does 4

↳ 50 % compute utilization ∴ but 100 % mem BW utilization

* Example 2: Increase compute speed to 16 ops/s

→ total time = unchanged!



overlap

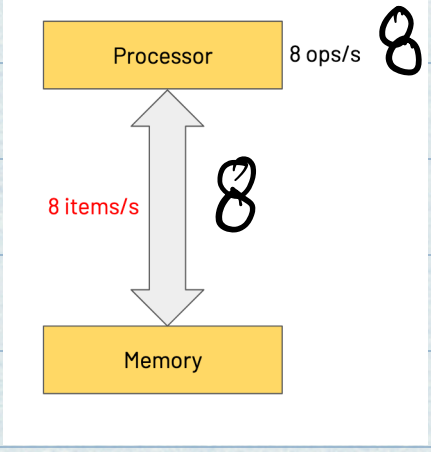
2s (load) + ~~2s (compute)~~ stall + 2s (write)

↳ 25 % compute utilization

and 100 % mem BW utilization

* increasing compute speed does NOT improve perf

* Example 2: Increase mem BW to 8



Total time = 3 (overlap)

1s (load) + 1s (compute) + 1s (load)

3 seconds!

→ 0-1: 8 items arrive

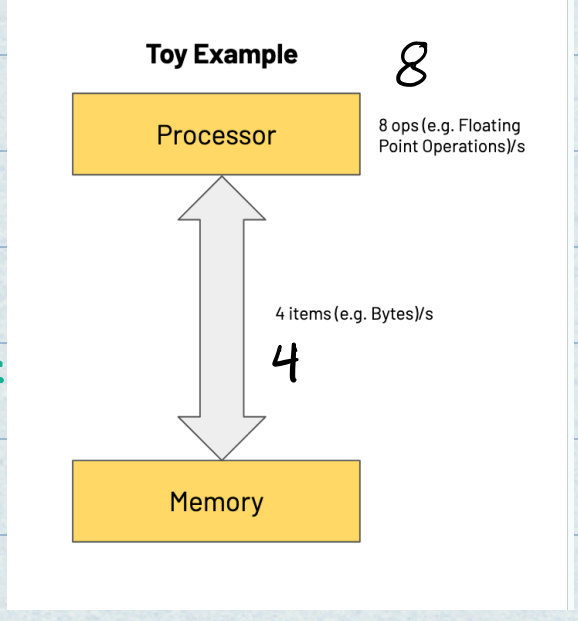
→ 1-2: 8 items computed

→ 2-3: 8 items written

↳ 100 % compute and BW utilization

* example 4: original HW, but new algs

1. Load 4 items from mem
2. compute 4 ops on EACH item
3. write 4 outputs back



total time = 4

1s (load) + 2s (compute) + 1s (write):

- 4 items arrive at proc. in steady state PER second
- processor needs to do 16 ops per second, but can only do 8

→ 100 % compute AND BW utilization

* example 5: a lg 2. but increase compute speed to 16

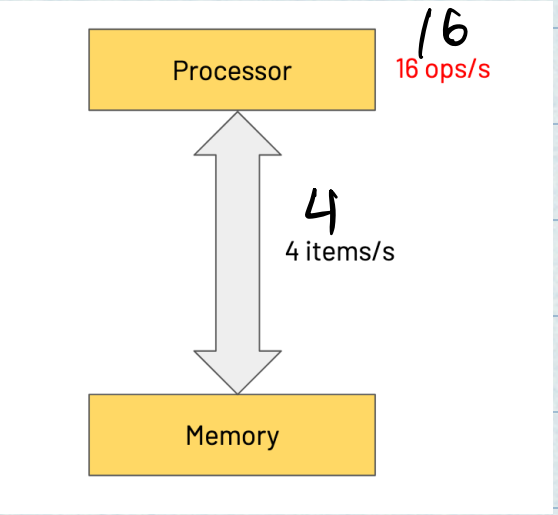
total time = 3

1s (load) + 1s (compute) + 1s (write)

→ 4 items arrive per second

→ processor needs to do 4 opr/item /second, it can!

→ 100% compute and BW utilization



* Compute Bound vs. Bandwidth Bound

- recall: compute bound when $T_{math} > T_{mem}$
- mem. bound when $T_{mem} > T_{math}$

$$T_{math} = \frac{\#\ ops}{compute\ BW} \qquad T_{mem} = \frac{\#\ bytes\ accessed}{mem\ BW}$$

(alternately - think abt ops/byte):

- compute bound:

$$\frac{\#\ ops}{bytes\ accessed} > \frac{compute\ BW}{mem\ BW}$$

- mem bound:

$$\frac{\#\ ops}{bytes\ accessed} < \frac{compute\ BW}{mem\ BW}$$

* Arithmetic (operational) intensity:

→ how many ops to we perform for each byte moved?

→ Arithmetic Intensity = $\frac{\text{Total \# ops}}{\text{\# Bytes written/read to/from mem.}}$

* Lets re-consider A100:

- 1935 GB/s mem. bw
- 312 TFLOPS for FP16

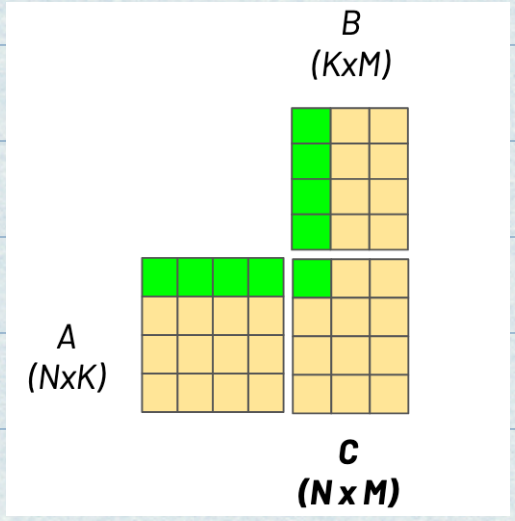
	A100 80GB PCIe	A100 80GB SXM
FP64	9.7 TFLOPS	
FP64 Tensor Core	19.5 TFLOPS	
FP32	19.5 TFLOPS	
Tensor Float 32 (TF32)	156 TFLOPS | 312 TFLOPS*	
BFLOAT16 Tensor Core	312 TFLOPS | 624 TFLOPS*	
FP16 Tensor Core	312 TFLOPS | 624 TFLOPS*	
INT8 Tensor Core	624 TOPS | 1248 TOPS*	
GPU Memory	80GB HBM2e	80GB HBM2e
GPU Memory Bandwidth	1,935 GB/s	2,039 GB/s

- peak flops (or ratio of compute to mem. bw): $\frac{312 \text{ FLOPS}}{1935 \text{ GB/s}} = 161 \frac{\text{FLOPs}}{\text{byte}}$

* we can also compute arithmetic intensity of common algorithms:

* Lets consider matrix multiply!

→ recall: output C has N×M entries, each entry comes from dot product of vectors (A_i, B_j) of dimension K:



→ each dot product is K multiplies and K adds, totaling 2K flops:

$$C_{ij} = A_{i1} \cdot B_{1j} + A_{i2} \cdot B_{2j} \dots A_{ik} \cdot B_{kj}$$

total flops = $2MNK$

*what abt memory?

→ I/O components:

1) read A: NxK
2) read B: KxM
3) write C: NxM

→ for fp16, each val = 2 bytes

→ total bytes accessed:

$$2(NK + KM + NM)$$

*arithmetic intensity of matmul is:

$$\frac{2MNK}{2(NK + KM + NM)}$$

* consider 2 examples:

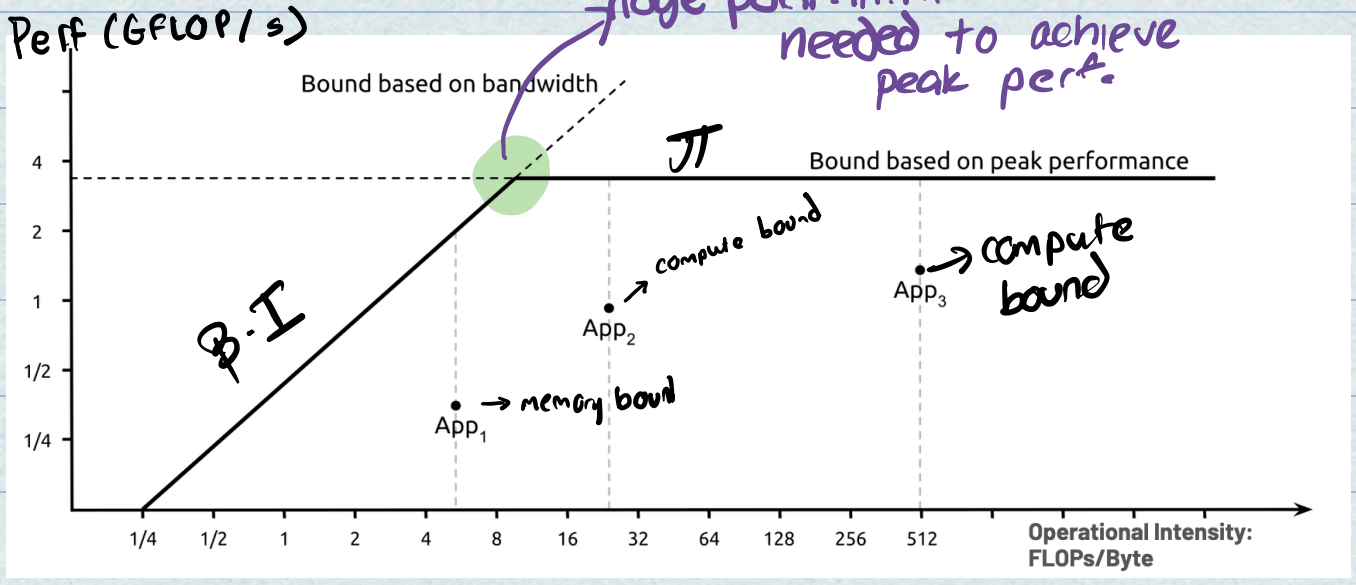
1) $M = K = 8192$, $N = 128$ → $\frac{2(8192 \cdot 128 \cdot 8192)}{2(8192 \cdot 128 + 8192^2 + 128 \cdot 8192)} = 124.1$

for A100: $124.1 < 161$ → memory bound

2) $M = K = N = 8192$ → arithmetic intensity = 2730.6

2730 . 6 > 161 → compute bound!

* ROOFLINE MODEL



https://en.wikipedia.org/wiki/Roofline_model

- Idea: plot perf. (GFLOP/s) as function of arithmetic /operational intensity (FLOPs/Byte)

$P = \min \begin{cases} \pi = \text{peak perf. of this HW} \left(\frac{\text{peak compute BW}}{\text{peak mem BW}}\right) \\ \beta \cdot I = \text{product of mem. BW and algorithm's arithmetic intensity} \end{cases}$

→ next time:

- some simple steps for improving perf.
- analyzing efficiency of transformers