Agenda:

(1) - as we didn't get to it last time: FLOPs analysis in transformer training

(2) Revisit scaling in normal, softmax attention

(3) How is Attention Executed on GPUs today?

(4) Revisit of Fusion, Tiling, caching vs. recomputation, Hardware-specific optimizations

(5) Flash Attention: → we didn't get to this today!

→ credits:

- Lec 4 of CS229s 23': https://docs.google.com/presentation/d/1PV0cKnzcHRCAbJy3OtER1UdJME7T7WBEqLLYmKWJR4g/edit#slide=id.g24c4df0ad9e_0_52

- Lec 5 of CS 229s 23': https://docs.google.com/presentation/d/1kY5MOOJfWny7SvV4D7YeSAulOTMnnVJTKKG5DQYLid0/edit#slide=id.g24c4f25c1b8_0_6

- Lec 13 of CMU 15-442: https://mlsyscourse.org

(1) FLOPs Analysis in ML Training for Transformers

- reminder of storage vs recomputation and how it is present in backprop

→ consider:

$u = w_1 w_2 \quad L = v w_3$

$v = 2u$

$$L(w_1, w_2, w_3) = 2(w_1 \cdot w_2) \cdot w_3$$

$$\frac{dL}{dv} = w_3 \qquad \frac{dL}{dw_3} = v \qquad \frac{dv}{du} = 2 \qquad \frac{du}{dw_1} = w_2$$

du ideally

$\frac{\partial}{\partial w_2} = w_1$ we reuse this!

→ so:

1) $\frac{\partial L}{\partial w_1} = \left(\frac{\partial L}{\partial v}\right)\left(\frac{\partial v}{\partial u}\right)\left(\frac{\partial u}{\partial w_1}\right)$

2) $\frac{\partial L}{\partial w_2} = \left(\frac{\partial L}{\partial v}\right)\left(\frac{\partial v}{\partial u}\right)\left(\frac{\partial u}{\partial w_2}\right)$

) $\frac{\partial L}{\partial w_3} = v$

↳ reusing v ($v = 2u$) requires storing v from FW pass

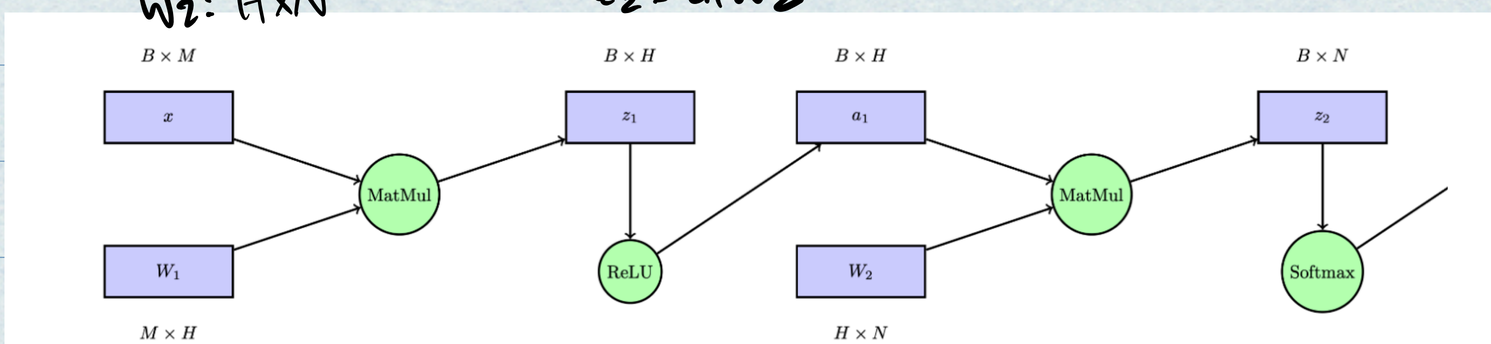
*tension btwn storing activations to make backprop more efficient, and minimizing memory w/

* FLOPs analysis for MLP (multi layer perceptron)

$X : B \times M$ $\quad z_1 = XW_1$

$W_1 : M \times H$ $\quad a_1 = \mathrm{Relu}(z_1)$

$W_2 : H \times N$ $\quad z_2 = a_1 W_2$



2BHN

- Lets consider $a_1 \cdot W_2$: 2BHN flops

→ what about backprop:

$$\frac{\partial L}{\partial W_2} = a_1^T * \frac{\partial L}{\partial z_2} \qquad \frac{\partial L}{\partial a_1} = \frac{\partial L}{\partial z_2} \cdot W_2^T$$

↓ $H \times B \cdot B \times N =$ $2HBN$ flop

↓ $B \times N \times N \times H$ $= 2BNH$ flops

→ and we would need to cache a_1

* interesting pt:

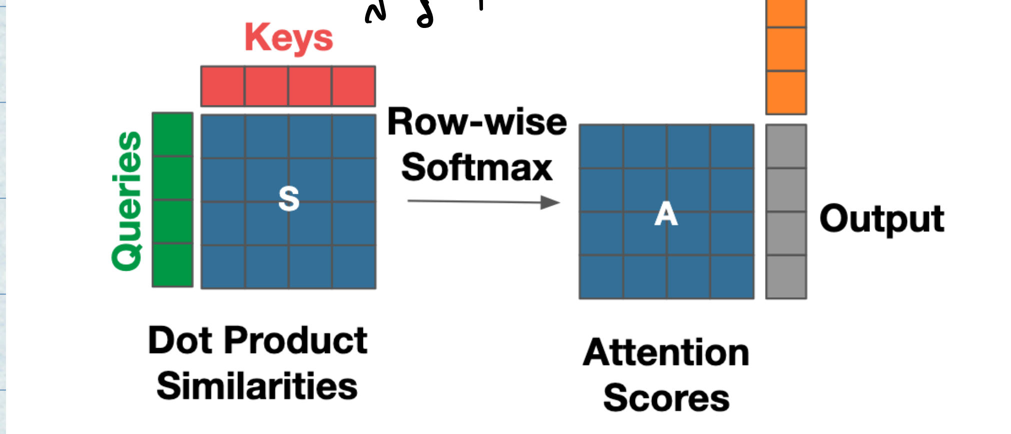
→ backprop is 2x larger than fw
(4BHN vs. 2BHN)

(2) Revisit scaling in normal, softmax, attention

→ recall: Q, K, V matrices are N×D dimension

→ therefore: final attention scores are N×N

$$A = \text{Softmax}\left(\frac{\overbrace{QK^T}^{N\times d \;\; d\times N}}{\sqrt{d}}\right) \longrightarrow N \times N$$



→ high level takeaway (though we went through math in the last class: Attention MEMORY and COMPUTE scales quadratically $O(N^2)$ in the INPUT SEQUENCE LENGTH

→ but it would be nice if our models could scale to long sequences:

- thousands of words in books or docs
- long-range dependencies between definitions and concepts at multiple scales (within a problem, chapter, across entire book)
- thousands of timesteps in single second of raw audio; long-range dependencies across audio sequences...

→ next time: we will look at SW methods to improve this complexity. but today:

*** IS THERE A FAST EXACT ATTENTION ALGORITHM? ****

(3) How is attention executed on GPUs today?

- kernel = some "unit" of operation a GPU; involves:
  1. Load data from HBM to SRAM/registers (local to each streaming multiprocessor
  2. compute (kernel can also store intermediate data in SRAM)
  3. write data (result) back to HBM

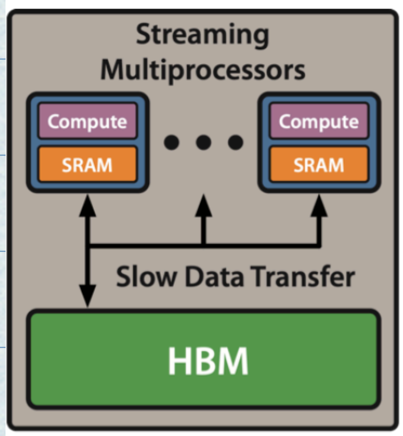


Photo: Dao et al., 2022

- GPUs can execute diff. kernels in parallel across streaming multiprocessors!

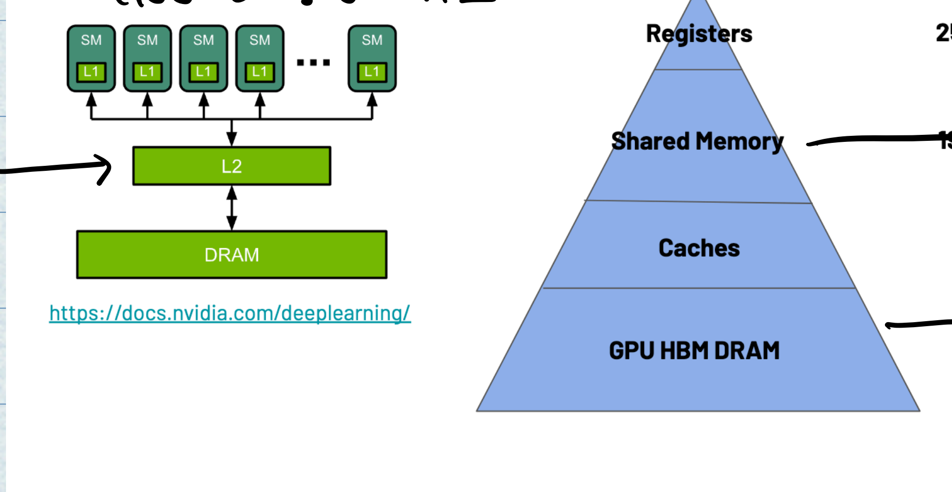
- as we explored in a previous lecture, SRAM/HBM form a memory hierachy:

on A100:

- SRAM = fast but small — ~3x smaller, but order of magnitude faster
- HBM = slow but larger

there's OTHER levels of memory hierachy, but we'll focus on just SRAM vs. HBM

(108 SMs on A100



Shared Memory → 192 KB/SM

GPU HBM DRAM → 40GB

- how does attention currently run on a GPU?

- in most regimes - memory is the bottleneck in attention computation (only compute bound when generating 2048 tokens, as we saw last class)

	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

compute BW is ~300 Flops/s

HBM memory BW is ~2 TB/s

- in GPU - compute speed outpaces memory speed!

- 2 further issues:

1. Attention implementations do not carefully account for

reads and writes to dif. levels of GPU memory (e.g. SRAM vs. HBM)

2. Popular DL frameworks (Pytorch) do not expose ways for finegrained memory management

* you can't fit attention computation into SRAM - so you end up doing many reads and writes between S.M.s and HBM.

- as a reminder: what is happening after Q, K, V are calculated:

N = seq. length B = # attn heads
D = dimension

Memory Accesses:	sizes	read and write?	fp16	total
· Read Q, K	2NDB	1	2	4NDB
· Write QK^T	NNB	1	2	$2N^2B$
· read/write masking	* you will fill in table during PLQ:			
· read/write Dropout				
· read/write Softmax				
· read V				
· write multiply by V				

Compute:	Op:	Flops:
· Compute QK^T (N×D, D×N)	Matmul	B·N·D·N·2 → notes made and take in lecture!
masking, dropout, softmax	elementwise / or reduction	B·N·N·2 → N^2 matrix
· multiply by V (N×N, N×D)	matmul	B·N·N·D·2

softmax = element-wise exponent, sum, divide

- note that we are JUST talking about **attention**
  not feed-forward/MLP layers

- we are also ignoring Q, K, V projections:
  or creating $Q = W_Q \cdot inp$, $K = W_K \cdot inp$, $V = W_V \cdot inp$

* observation: attention op has memory-boundedness because it is a series of VERY low-compute ops (masking, dropout, softmax) w/ reads /writes between these ops!

* Let's Profile Attention!

  · recall : $AI = \frac{\#ops}{\#\text{ bytes accessed}}$

  if $AI > \frac{\text{HW compute BW}}{\#\text{HW mem BW}}$: compute bound

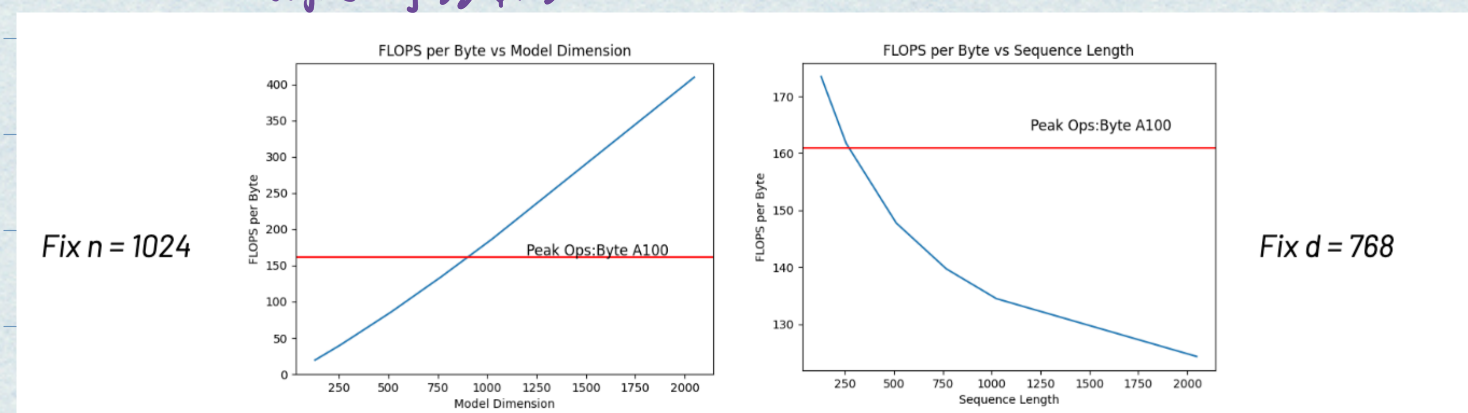
  if $AI < \frac{\text{HW compute BW}}{\text{HW mem BW}}$: mem bound

  · for A100 : peak ratio is 161 $\left(\frac{312 \text{ TFlops}}{1935 \text{ GB/s}}\right)$

  fp16/bf16 at tensor core

  ↳ PCIe HBM mem BW

- we can (ignoring B) plug in values for n and D above:



* observation: as sequence length increases, AI ↓

LESS than peak ratio for A100 (whenever N > 250)

↳ but in full attn block - $Q \cdot K^T$ could be compute bound, mat mul w/ V could be compute bound

- feedforward networks could be mem. bound

· recall AI of matmul:

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

what happens as N↑?

· K = M = 8192, N = 128: AI = 124.1

↳ mem. bound

- K = M = N = 8192: AI = 2730.6

↳ compute bound!

*takeaway: as N↑ for attn block, becomes more mem bound, but for mat. mul, it becomes more compute bound (imagine mat mul for QK^T or w/ V)

3) RECAP OF PRINCIPLES OF ADDRESSING ATTENTION BOTTLENECK

· Fusion → save trips to/from memory by performing composite data ops on processing units

· Tiling: assign subsets of computation to parallel workers - ends up reducing memory traffic

↳ remember: GPU has multiple SMs (streaming multiprocessors)

- thread blocks = given to independent SMs to execute

↳ blocks usually organized into 1D, 2D, 3D grids

- blocks contain threads
- SMs schedule warps, or groups of 32 threads, to execute same instr.

max # active threads =

$$(\#SMs) \cdot \left(\frac{\text{max blocks}}{SMs}\right) \cdot \left(\frac{\text{max threads}}{\text{block}}\right)$$

→ Tiling:

- recall: memory access within a thread block:
  - global memory: seen by all threads (in HBM)
  - shared memory: seen by threads in SAME block
    - ↳ one thread can pull in data multiple threads need to see
  - registers: private per thread

*what is an efficient mem. loading rule of thumb?

1) coalesced mem. access - threads in same warp should access adjacent cells of memory

2) Tiling: partition data into subsets so a subset fits into shared mem.

computation on these tiles should be performed INDEPENDENTLY

https://nichijou.co/cuda7-tiling/

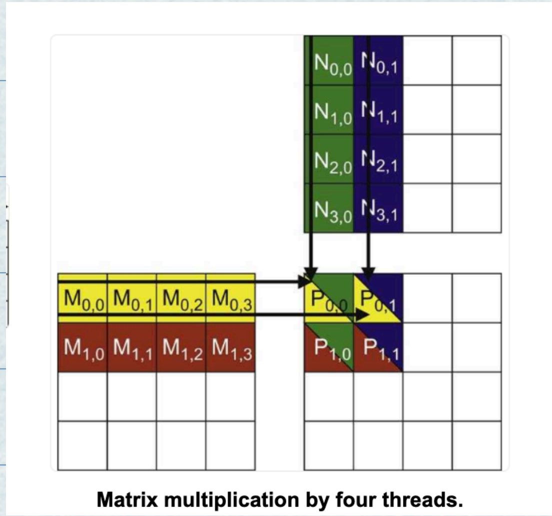
each elt. accessed twice!

Access order

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

→ observe that the threads overlap in some of the data they access

→ if they can collaborate (e.g. share data in same SM's shared mem) - saves 1/2 of mem accesses



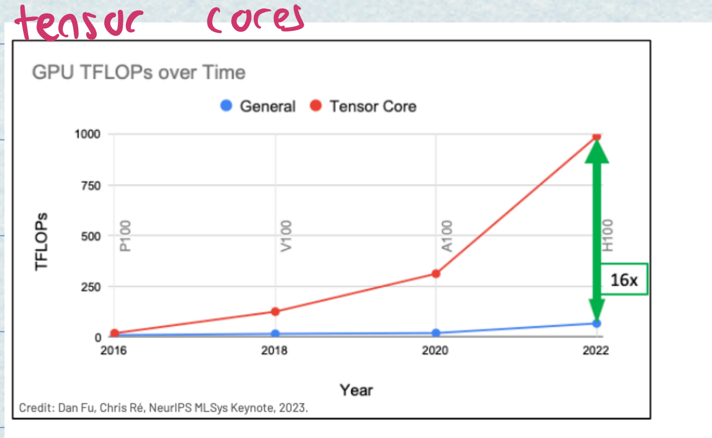
Matrix multiplication by four threads.

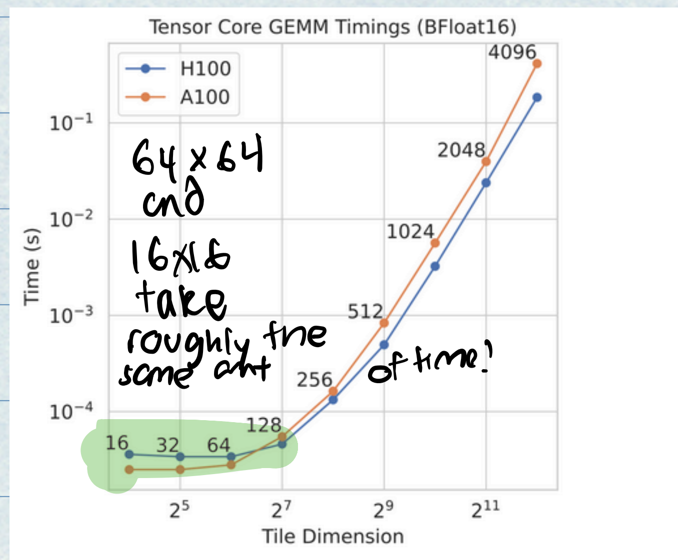
→ use Faster Specialized HW units (e.g.) tensor cores

- modern GPUs contain tensor cores - or specialized compute for performing MMs (GEMMs, D = AB + C)
- tensor cores can generally multiply 16x16 tiles

*key idea: algorithms should leverage tensor cores. On later and later GPUs, increasing speed dif. w/ w/o tensor cores

*key idea: select tile sizes to keep tensor cores busy (launching takes times)





→ Idea 4: Caching and Recompute

- Backprop requires storing intermediate values from FW pass, adding to reads/writes

- when memory bound (reads/writes expensive relative to compute) - recompute activations
- when compute bound (compute expensive relative to reads/writes) - store activations

- **FLASH ATTENTION**
  - overview: new implementation of standard attention (exactly) that carefully accounts for reads/writes by using above ideas:
    1) recompute for memory bound workloads, cache for compute bound workloads
    2) Tiling
    3) Fusion
  - for simplicity, we'll only discuss: $\mathrm{Softmax}(QK^T) \cdot V$
    ↳ you can sometimes add masking / dropout along w/ softmax, can be fused together
  - overview of results:
    → HBM accesses, given N = sequence length, d = dimension, M = SRAM size

flash attn: $O\left(\frac{N^2 d^2}{M}\right)$

standard attn: $O(Nd + N^2)$ → about 9x fewer HBM writes

Attention	Standard	FlashAttention
GFLOPs	66.6	75.2 (↑13%)
HBM reads/writes (GB)	40.3	4.4 (↓9x)
Runtime (ms)	41.7	7.3 (↓6x)

Model implementations	OpenWebText (ppl)	Training time (speedup)
GPT-2 small - Huggingface [87]	18.2	9.5 days (1.0×)
GPT-2 small - Megatron-LM [77]	18.2	4.7 days (2.0×)
GPT-2 small - FlashAttention	18.2	**2.7 days (3.5×)**
GPT-2 medium - Huggingface [87]	14.2	21.0 days (1.0×)
GPT-2 medium - Megatron-LM [77]	14.3	11.5 days (1.8×)
GPT-2 medium - FlashAttention	14.3	**6.9 days (3.0×)**

- 3x faster training compared to HuggingFace!
- 15% faster pre-training than BERT pre-training record in MLperf (the olympics of efficient model training)
- also in some cases results in higher quality models - can train on longer sequences!

- Let's incrementally build flash attn:
- approach: lets avoid materializing the full NxN matx during the attn. computation

→ Try 1: what if we loaded all of Q and all of K into stream-multiprocessor's SRAM to compute attn output?

↳ an NxN matrix needs $2N^2$ space

↳ but a s.m. has 20-40 MB of SRAM

Context Length (N)	Size of $O(2N^2)$
500	0.5
1000	2MB
4000	32 MB

But we could be trying to fit multiple BATCHES of inputs