- Agenda:
  - more review of Roofline model
  - Basic Principles for Achieving High perf. on Hardware (w/ focus on some computations in attention
  - Perf. Analysis of Transformers
  - If time: KV caching, advanced techniques for inference efficiency
- Sources:
  - End of cs229s lecture on HW: https://docs.google.com/presentation/d/14hK7SmkUNfSEIRGyptFD2bGO7K9sJOTnwjAVg3vgg6g/edit?usp=sharing
  - cs229s Lecture on Analyzing Transformers: https://docs.google.com/presentation/d/1PV0cKnzcHRCAbJy3OtER1UdJME7T7WBEqLLYmKWJR4g/edit?usp=sharing
  - Blogpost on roofline model: https://dando18.github.io/posts/2020/04/02/roofline-model

* Part 1: review of roofline model
  - remember units of arithmetic intensity
  - $AI = \frac{\text{Ops / time}}{\text{memory traffic}} \rightarrow$ eg. $\frac{\text{Flops/second}}{\text{bytes/second}} = \frac{\text{flops}}{\text{byte}}$
  - this metric is important because in general: memory ops are much slower than compute ops
    - → on a CPU:
      - → FLOP like add or multiply: ~1 cycle

→ I/O from register: ~ 1 cycle

→ I/O to L1 cache: ~ 1 ns (2 cycles)

→ I/O to L2 cache: ~ 4 ns (8 cycles)

→ I/O to main memory: 100 ns (200 cycles)

for 2 GHz processor: 2 cycles = 1 ns

- in general in our algorithm we should strive to do as much computation before loading next data

- high AI: lots of computation per load
- low AI: low amt of computation per load

* consider Axpy operation:

$y = ax + y \quad a \in \mathbb{R} \quad x, y \in \mathbb{R}^n$

→ to impl in C:

```
void daxpy (size_t n, double a, const *double x,
                         const *double y):

    for ( size_t i=0 ; i+=1 ; i<n ):
        y[i] = a·x[i] + y[i]
```

· number of flops: $2 \cdot n$ → 1 add and 1 multiply per loop iteration

· memory traffic: $\left(2 \text{ loads} \cdot \frac{8 \text{ bytes}}{\text{load}} + 1 \text{ store} \cdot \frac{8 \text{ bytes}}{\text{load}}\right) \cdot n$

$= 24n$

· Arithmetic intensity: $\frac{2n}{24n} = \frac{1}{12}$

* consider matrix multiply for matrix $(M \times K) \cdot (K \cdot N)$

→ from last time: $\frac{2MNK}{(MK + KN + MN)}$

~generally: MM will do more math per byte for larger matrix sizes

*roofline model

· from HW specs we know:

- Peak Bandwidth: bytes /s
- Peak compute perf: flops /s

· the Actual perf. of an algorithm is:

$$\min \begin{cases} AI \cdot \text{Peak bandwidth} \rightarrow \frac{flops}{byte} \cdot \frac{bytes}{s} = \frac{flops}{s} \\ \text{Peak compute perf} \rightarrow \frac{flop}{s} \end{cases}$$

· intuition: no alg can have higher flop /s than hardware limit, but alg can be limited by BW

*note that in reality, there might not be 1 BW slope or 1 compute ceiling.

https://people.eecs.berkeley.edu/~kubitron/cs252/handouts/papers/RooflineVyNoYellow.pdf (original roofline paper)

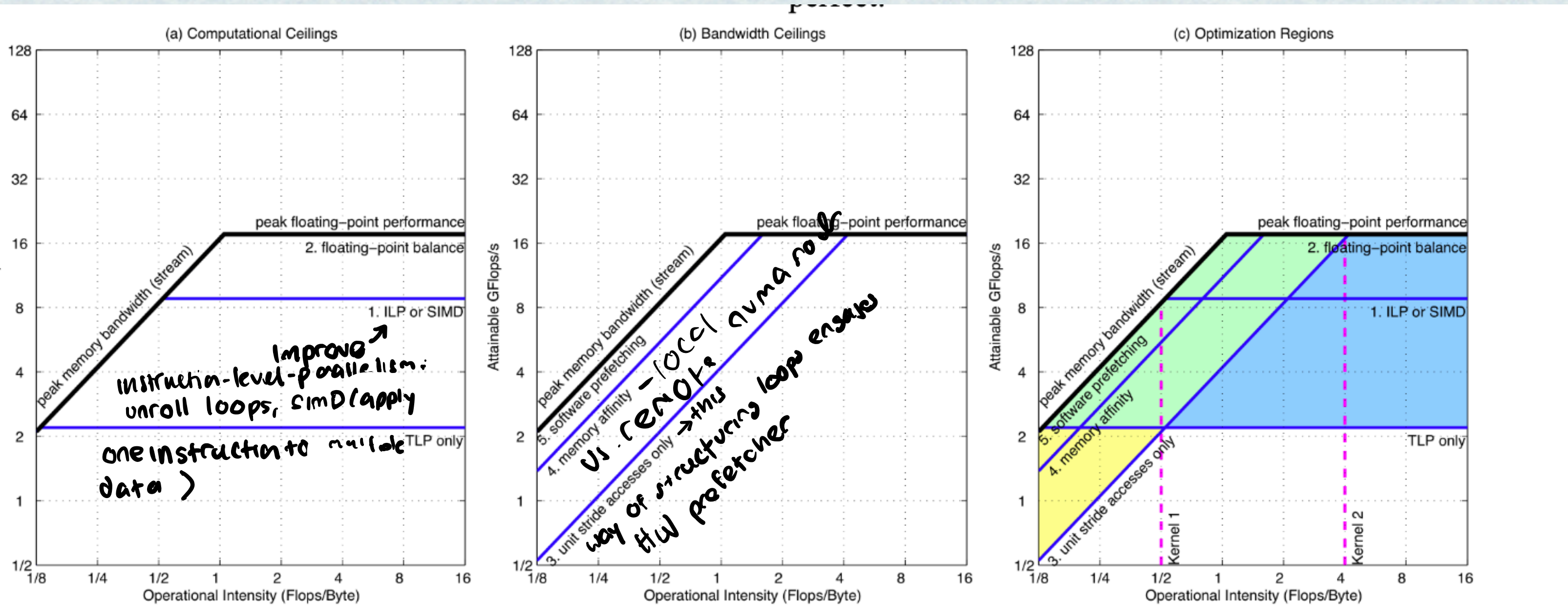
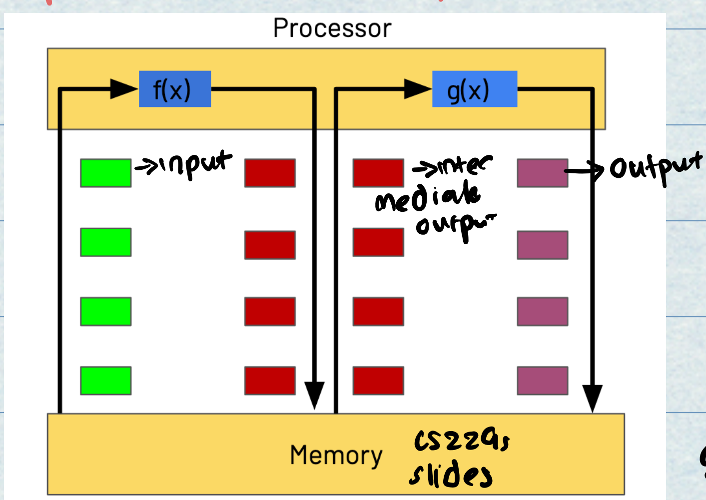


Figure 2. Roofline Model with Ceilings for Opteron X2.

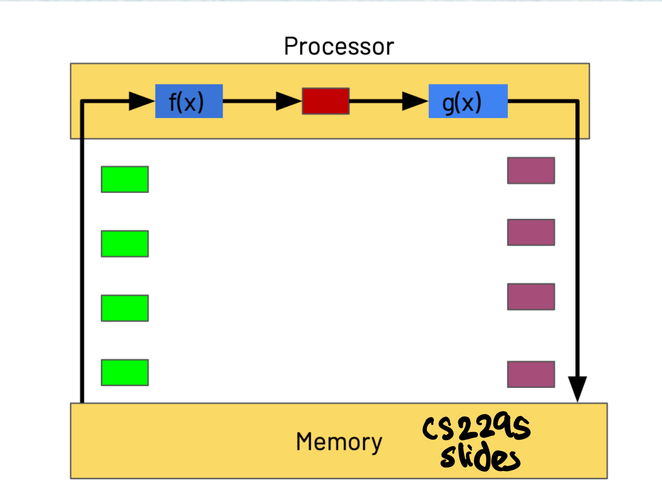
- Depending on where exactly your algorithm falls - roofline als. suggests a series of compute and memory optimizations you can apply to reach peak of both memory BW and flops/second

*Part 2 of Lecture: Basic Principles of Achieving High Perf. on Hardware

1) Fusion: save trips to/from memory by performing composite data on processing units



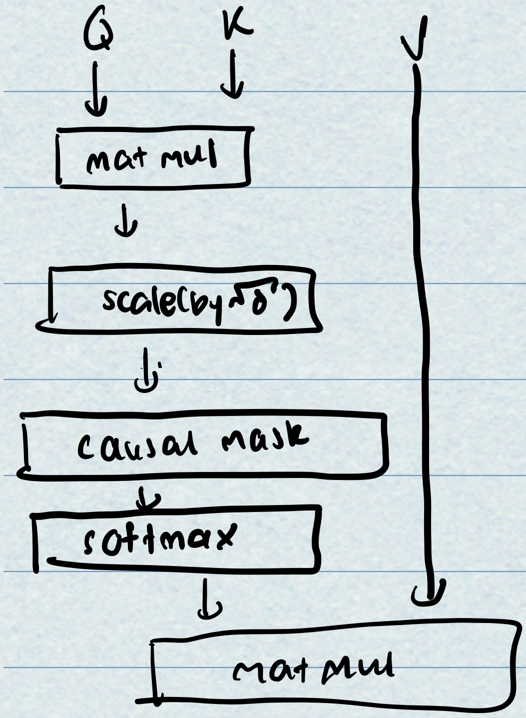
→ we write outputs of f(x) back to memory, load them back, before then computing g(x)

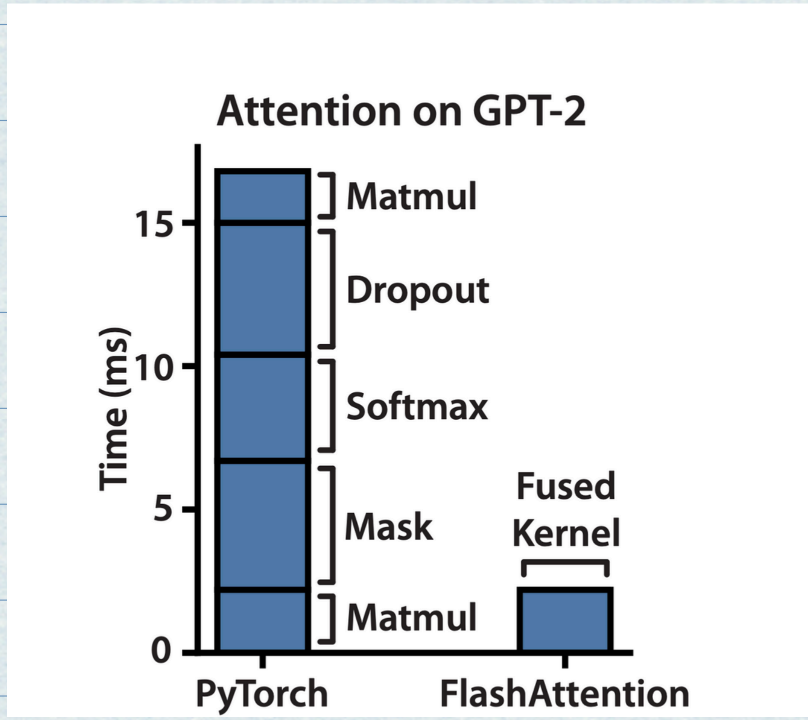


- Fusion prevents unecessary trips to HBM
- critical for I/O bound ops!

* Fusion for attention (flash attention!)

recall:

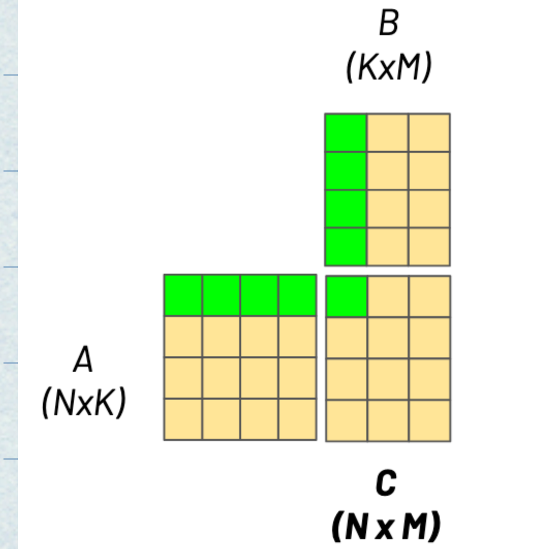




[FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness]()

2) Parallelization (in context of matrix multiply)

- recall: A100 GPU has 108 streaming multiprocessors
- how do we ensure each SM has as much work as possible
- idea: perhaps parallelize across OUTPUT cells
  - ↳ ideally atleast 108 output items



→ we can create one thread PER output cell value and put each thread in a sp. thread block - the dif. thread blocks will go to dif. SMs

→ what would arithmetic intensity be?

→ compute: Dot product of row of A and col. of B (K adds, K multiplies)

= 2K Flops

→ mem.

1) read row of A, col. of B (K) (K)

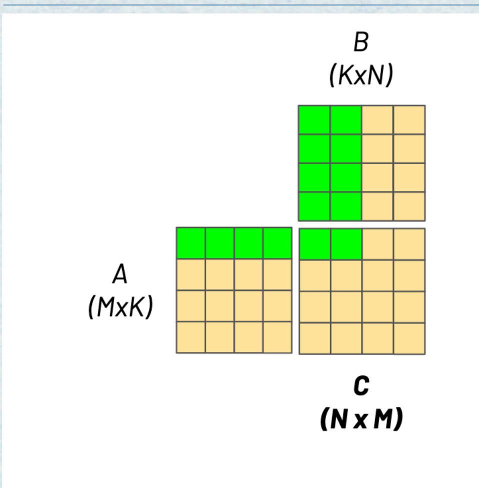
2) write one val. for C (1)

] = 2K + 1 memory accesses

→ arithmetic intensity = $\frac{2K}{2(2K+1)}$ * multiply by 2 for fp16

2) Idea: Blocking / Tiling w/ 1D tiling

- each thread computes 2 tile items in output matrix



→ compute: dot product of 2 cols of B, one row of A = 4K

→ memory:

1) read row of A = K

2) read 2 cols of B = 2K

3) write 2 outputs of C = 2

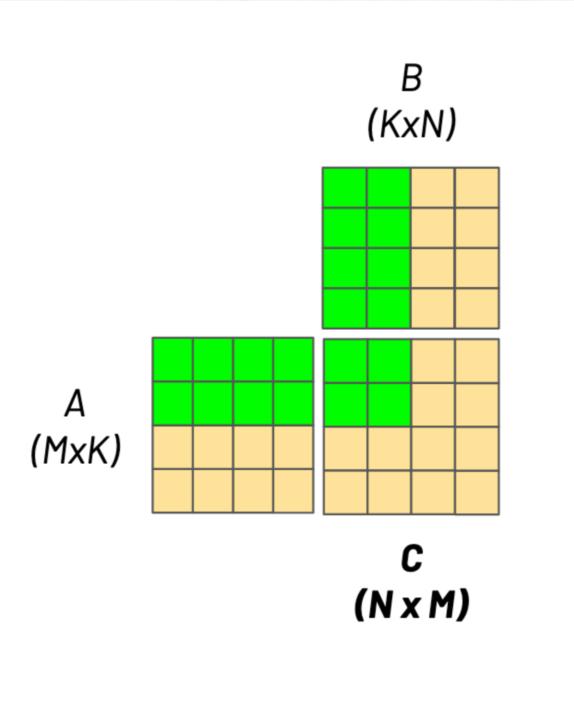
] = 3K+2 memory accesses

→ new AI : $\frac{4K}{2(3K+2)}$

* depending on problem size - might get luck w/ caching

2) 2D tiling

→ each thread computes 2D tile of output C



→ compute = 8K flops (4 output cells)

→ mem. access = 4K + 4 → but n fp16

AI: $\frac{8K}{2(4K+4)}$

→ general idea of blocking/tiling: assign TILES of work to each parallel processing unit - exploits some mem. locality (reuse of memory being loaded)

- note that AI does depend on K, though

3) Idea 3: caching and Recomputation

- for compute bound workloads - it makes sense to CACHE

- for memory bound workloads -

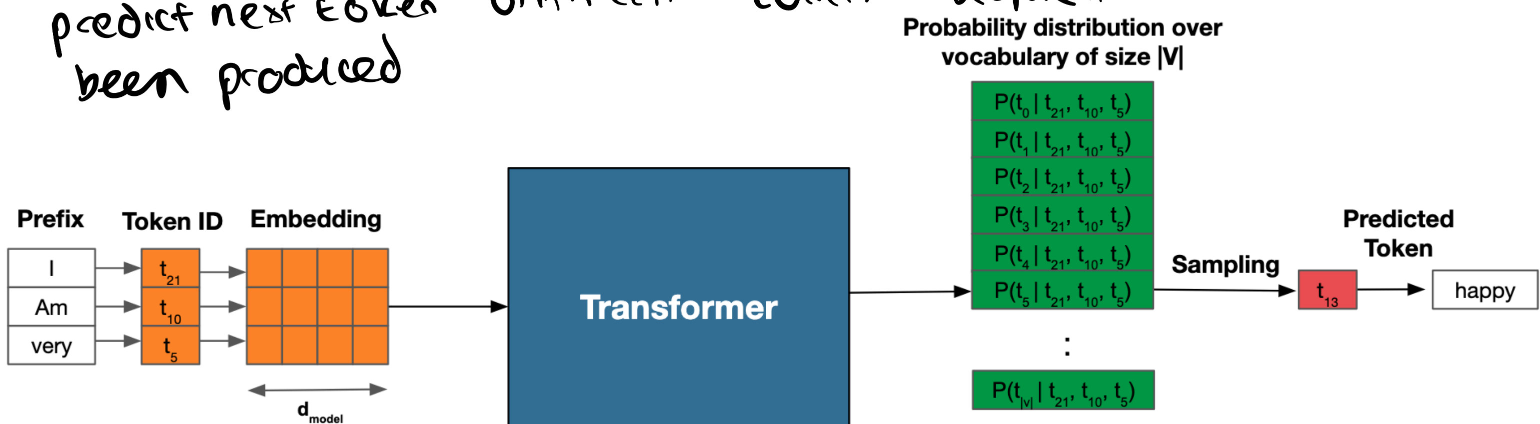
instead of caching - lets recompute
- recall auto diff. for backprop.
- we need to store activations to compute backword pass - gradients

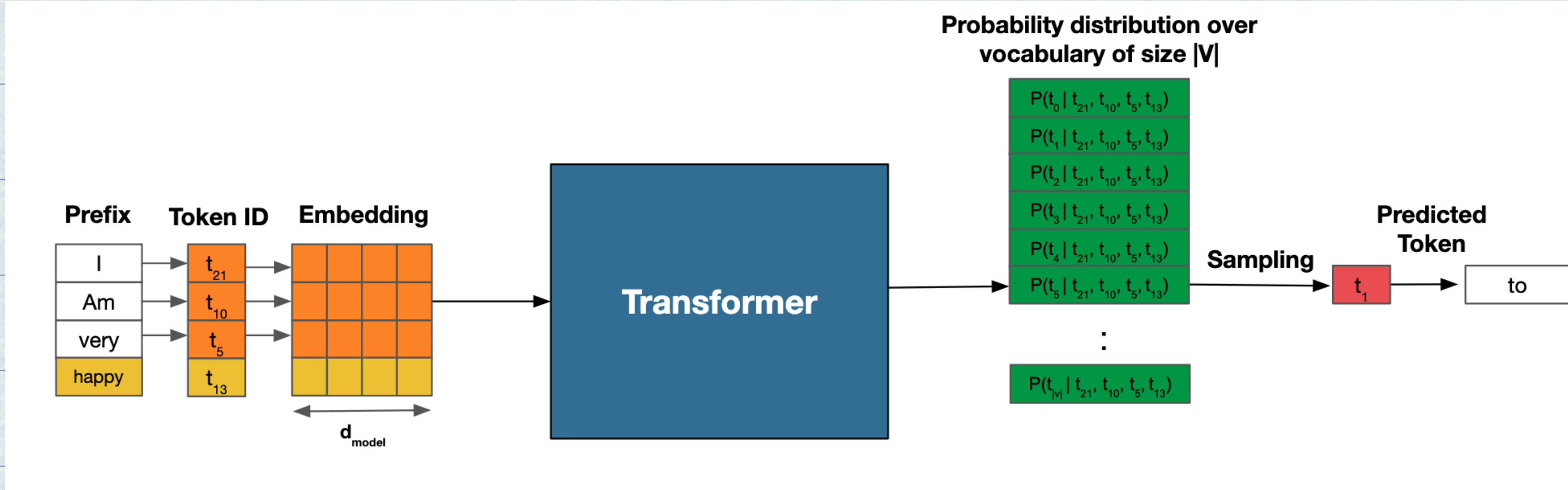
*in some cases - maybe don't store all intermediate activations; maybe better to actually recompute forward passes!

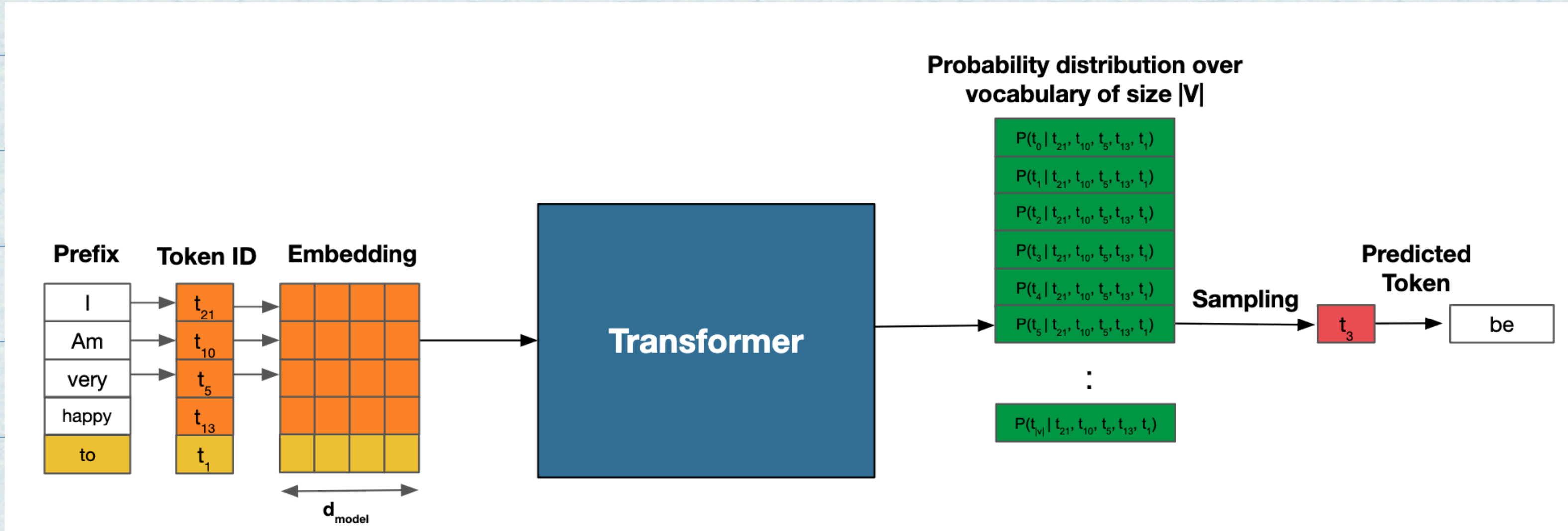
*Let's look at caching vs. recomputation in context of ** AUTOREGRESSIVE GENERATION **

- in inference - we take current prompt sequence - and predict next word

autoregressive = "predict one token at a time" - can't predict next token until entire current sequence has been produced

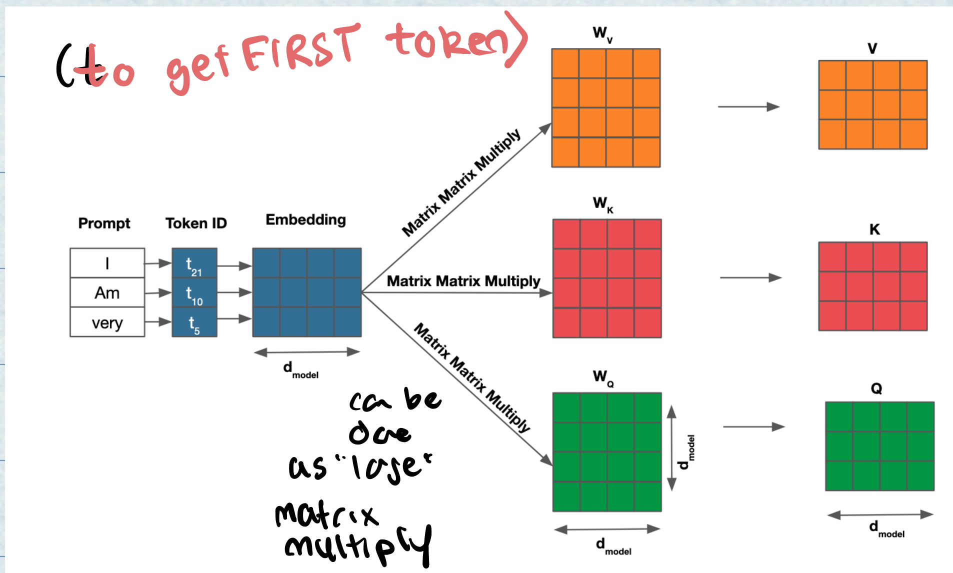






How does this manifest interms of math being done?

(to get FIRST token)



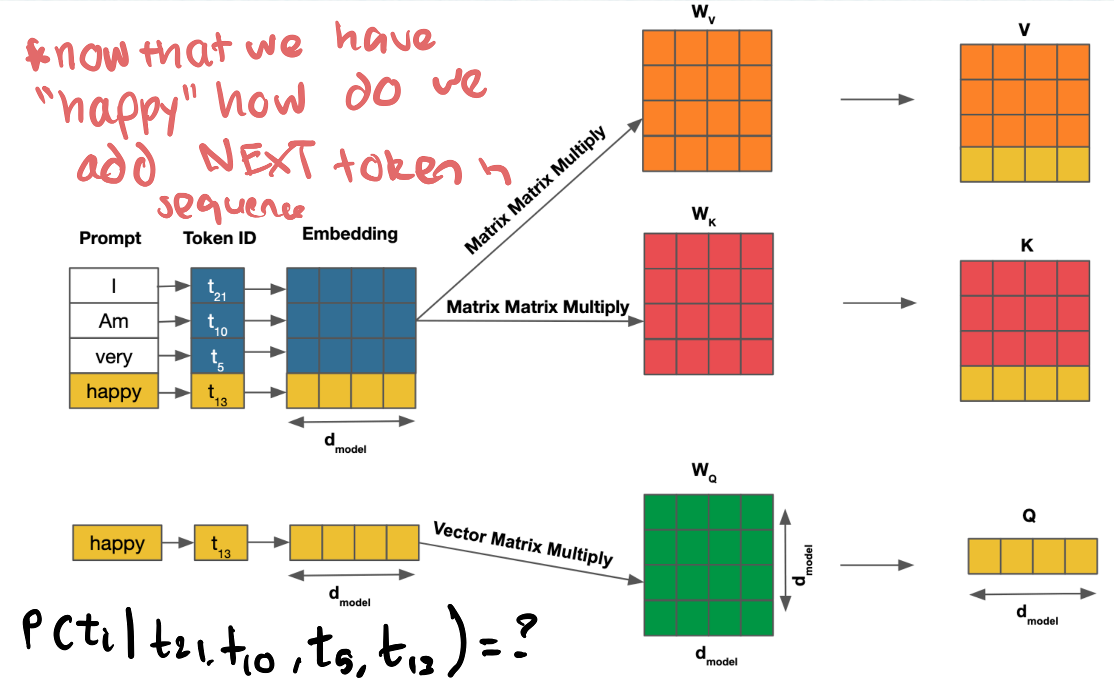
can be done as "large" matrix multiply

→for each token insequence we calculate:
- key and query vectors
- for each token t:
  - find attention scores of all tokens in sequence
- use attention scores to get value matrices

(for initial input prompt)

- But what happens when we predict next token in sequence?

*now that we have "happy" how do we add NEXT token in sequence

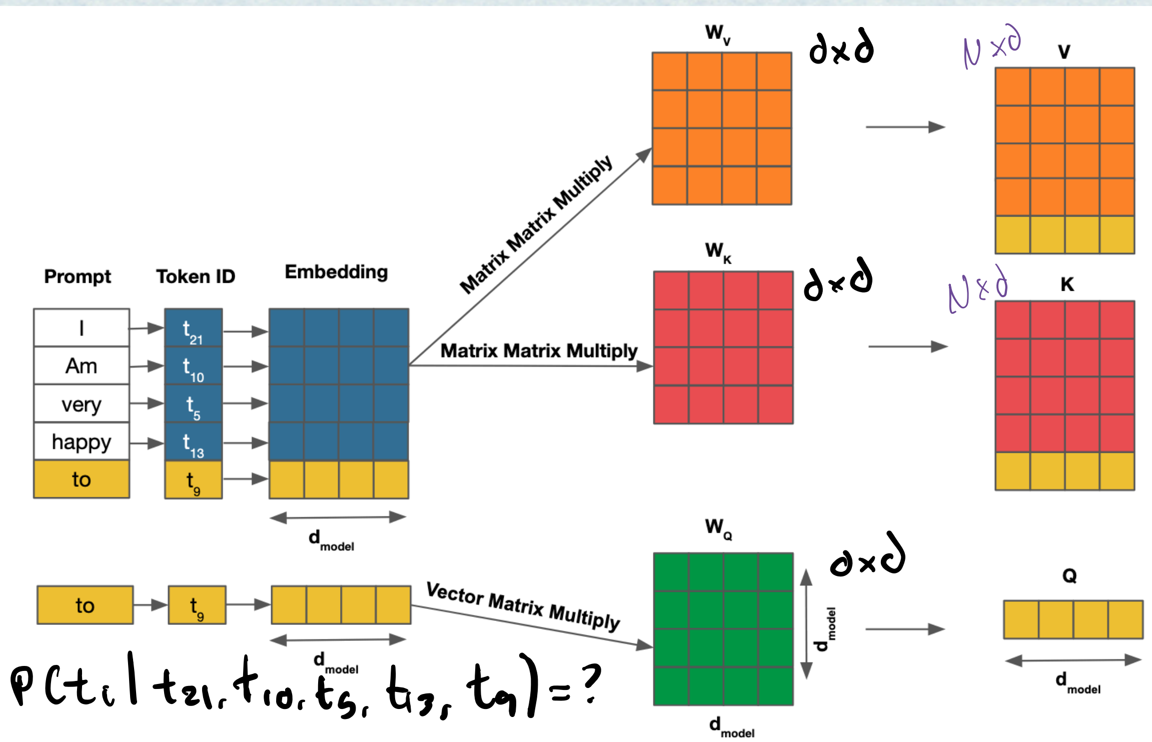


$P(t_i \mid t_{21}, t_{10}, t_5, t_{13}) = ?$

→ for next token:
- we only need to calculate one new query vector, one additional key vector; and one resulting value vector

↓

similarly:



$d \times d$ $N \times d$

$d \times d$ $N \times d$

$d \times d$

$P(t_i \mid t_{21}, t_{10}, t_5, t_{13}, t_9) = ?$

- again:
  - we only need to generate one query vector
  - and add one row each to value and key matrices

*what are the FLOPs in the attention calculation?

→ lets say we are pre-processing input sequence to produce <u>FIRST</u> token prediction?

- computing K: $O(N \cdot d \cdot d) = O(2Nd^2)$
- computing Q: $O(N \cdot d \cdot d) = O(2Nd^2)$
- multiplying Q and K: $S = QK^T = O(2N^2d)$
- letting $A = \text{softmax}(S)$ = had to account for exactly, but not bottleneck
- computing V: $O(2Nd^2)$

→ now we've predicted happy. how do we predict next word?

1. compute K for full sequence: $O(2Nd^2)$
2. compute new query vector: $O(2d^2)$

   vector matrix multiply →
   $1 \times d$ $\quad d \times d$

3. compute $S : \underset{1 \times d}{Q} \; \underset{d \times N}{K^T} = 1 \times N$ $= O(2Nd)$
4. $A = \text{softmax}(S)$: had to account for exactly
5. compute V for entire sequence so far: $O(2Nd^2)$
6. Compute $\underset{1 \times N}{A} \times \underset{N \times d}{V} = 1 \times d$ = attention weights for next pred. : $O(2Nd)$

* key idea: there's a TON of repeated computation if we do this for EACH NEW TOKEN

- Introduce concept of ** KV CACHING **

  → in producing any subsequent token after the FIRST one, we DON'T need to calculate all of K and V again (as most of it is unchanged):

→ we should reuse K and V as much as possible:

- new protocol for each NEW token:

1. compute K/Q/V VECTORs for new token:

→ all $1 \cdot d \times d \times d$ MMs =

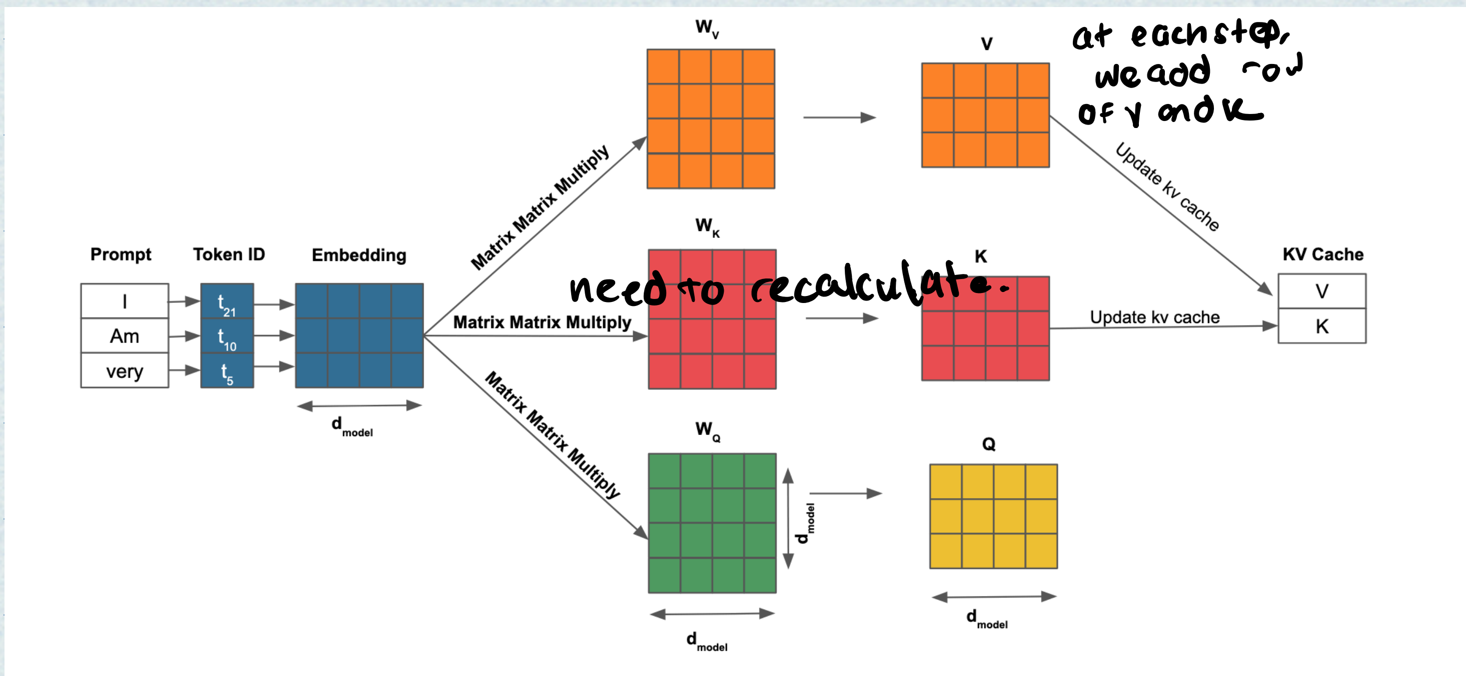
$O(3 \cdot 2d^2)$

2. multiplying $\underset{1 \cdot d}{Q} \cdot \underset{d \cdot N}{K^T} = O(2Nd)$

3. A = softmax (S): need to account for exactly

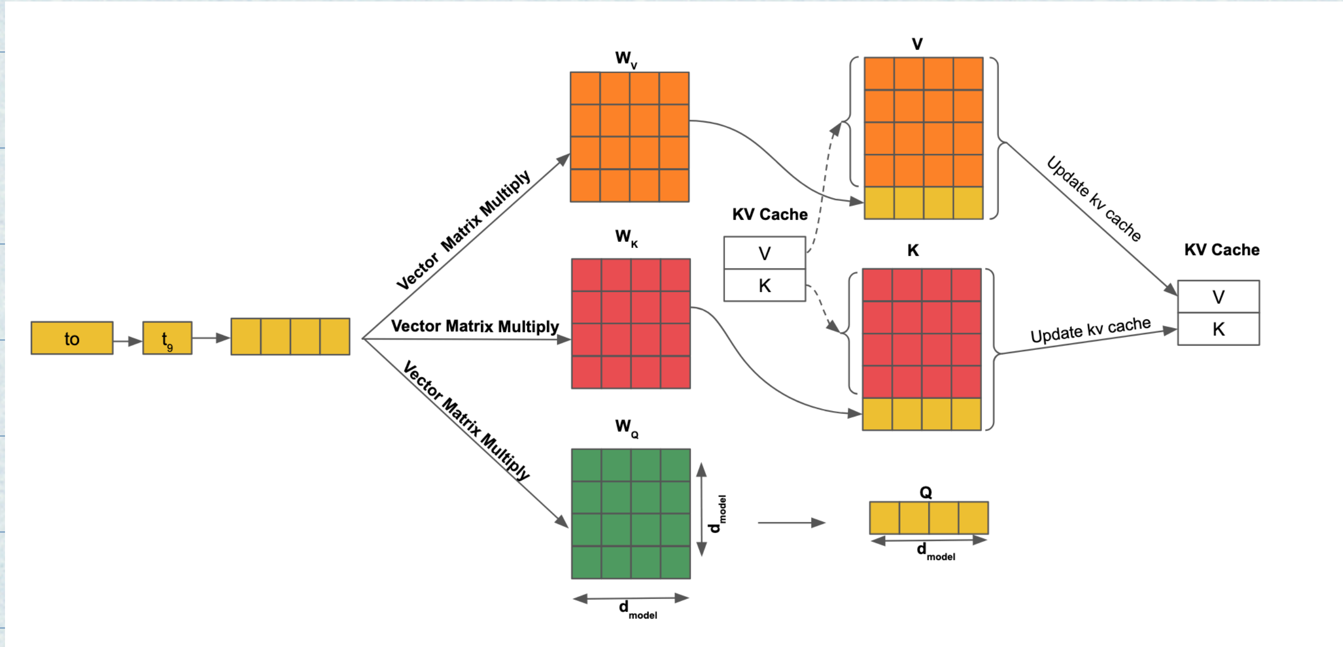
4. multiplying $\underset{1 \times N}{A}$ by $\underset{N \times d}{V}$: $O(2Nd)$

new picture:



↓

keep V/K cached in GPU DRAM so we don't recalculate!



* this was all for SINGLE HEAD attention, would be dif. for multihead

* In summary, for each new token our compute is: (w/ kv-caching)

$$O(3\cdot 2d^2) + O(2Nd) + O(2Nd)$$

→ lets try to understand WHEN KV caching is worth it:

→ total compute for calculating KV PER TOKEN

$$O(2\cdot n\_layers \cdot 2d^2)$$

k and v

attention layers

matrix multiply for (1xd)·(dxd)

. 1x

→ total storage for k and v vector $\boxed{\text{per token}}$

attention layers

dimension of k and v vectors

$$O(2\cdot 2\cdot n\_layers \cdot d)$$

k and v fp16

* so this needs to be stored PER TOKEN

→ recall on A100:

compute = 312^{12} Flop/s

mem BW = 1.5^{12} Bytes/s

→ How do we calculate arithmetic intensity of calculating $W_k \cdot x$ and $W_v \cdot y$

1) compute → compute term above:

$O(2 \cdot n_{layers} \cdot 2d^2)$

↳ represents $W_K \cdot x$ and $W_V \cdot x$

2) memory:

1) Loading W_k and W_v weights is:

$$O(\underset{k,v}{2} \cdot \underset{fp16}{2} \cdot \underset{\#\text{ attn layers}}{n_{layers}} \cdot \underset{W_k \text{ and } W_v \text{ are } d \times d \text{ matrices}}{d_{model}^2})$$

2) Writing output of $W_k \cdot x$ and $W_v \cdot x$ (activation) is from plugging term above:

$$O(2 \cdot 2 \cdot n_{layers} \cdot d)$$

- since loading weights is more, we will just consider that for simplicity:

$$T_{compute} = \frac{4 n_{layers} \cdot d^2}{312 \cdot 10^{12} \text{ FLOP/s}}$$

$$T_{mem} = \frac{4 n_{layers} d^2}{1.5 \cdot 10^{12} \text{ Bytes/s}}$$

compute to mem. ratio $= \frac{T_{mem}}{T_{math}} = \frac{312 \cdot 10^{12}}{1.5 \cdot 10^{12}}$

$\boxed{= 208}$

* what does 208 mean here?

- once we've loaded weights into memory, it is worth it to compute 208 tokens (run computation 208 times)

- fewer than 208: memory bound
- greater than: compute bound!

we will explore in a later class - another way to exploit this is via batching across requests!

* so when does KV caching make sense?

- if we are compute bound - KV cache makes sense! (trade off computation for the extra storage)