Agenda

→ Recap of last time: memory and FLOPs complexity of softmax attn

→ A few Attn-Approximation Algorithms

→ Flash Attn (which we started to introduce last lecture

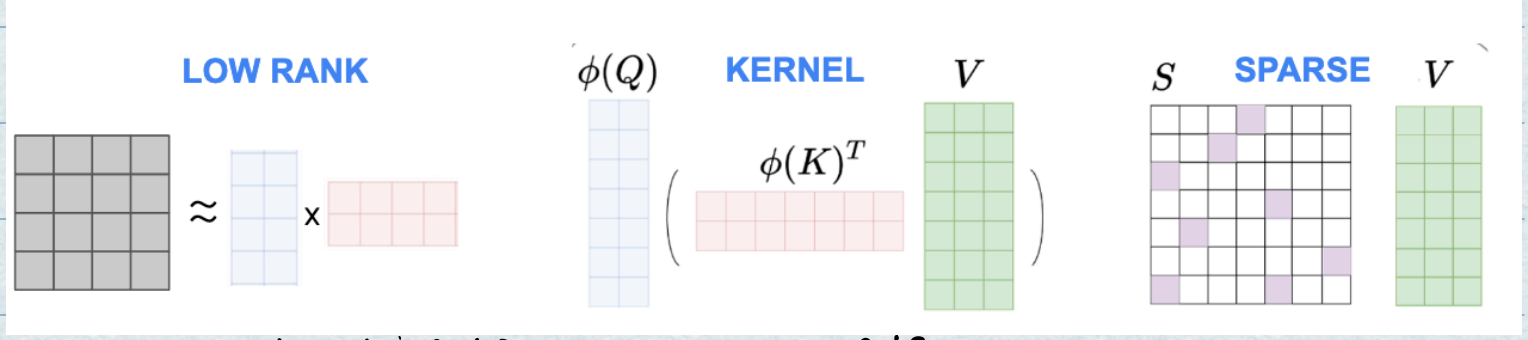
→ Intro to project 2 (released today!)

Recap: memory and compute complexity of softmax attn:

1. compute QK^T ($N \times d$ · $d \times N$ = $N \times N$ matrix)
2. Causal mask, softmax
3. multipy result by V (weighted sum of value vectors w.r.t attn scores)

*problem - as size of inp. sequence (N) grows - the $N \times N$ matrix QK^T grows quadratically in size ($O(N^2)$); compute is also bottlenecked by $O(N^2 d)$ term if $N > d$

* some potential ideas to make attn. more efficient



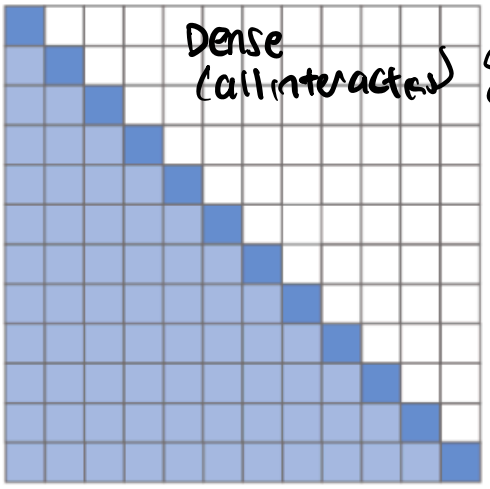
can we project matrices to smaller dimensions?

can we compute similarities b/w tokens w/o full $N^2 d$ complexity? w/o softmax?

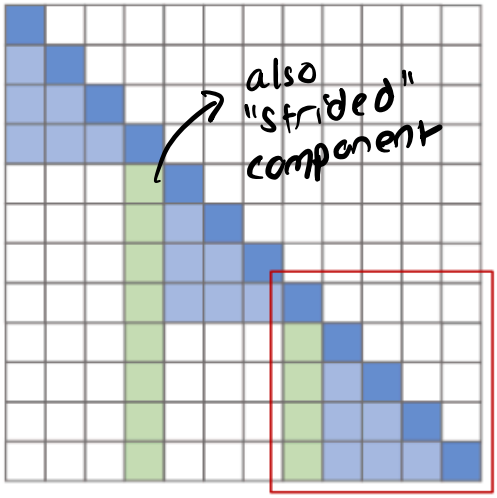
can we compute attn only between a SUBSET of tokens?

1) Sparsity

- Let's not keep full matrix (even in memory)
- let's just consider a subset



(a) Transformer



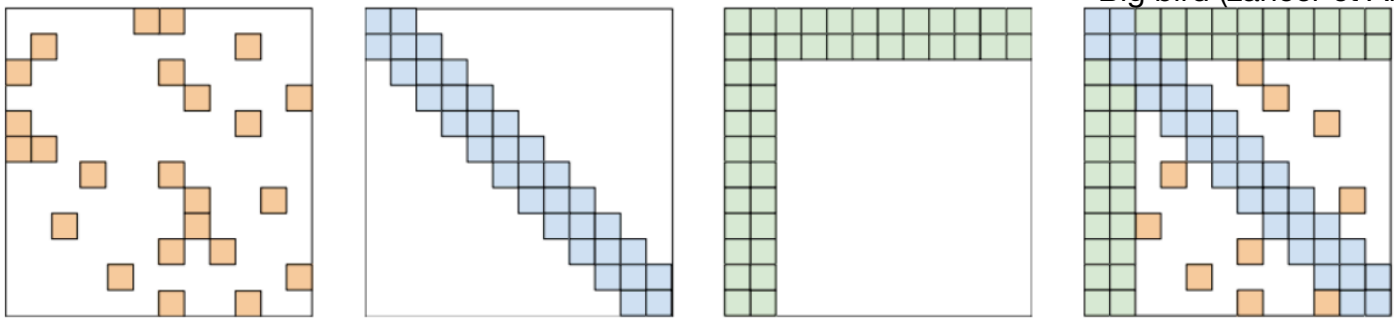
(b) Sparse Transformer

Efficient Transformers; A Survey. Tay Et. Al. 2022

- Compute attn in "local" windows
  - drop a bunch of interactions
  - here: computing interactions mostly btwn neighboring tokens
- compute attn in local windows.. → $O(n\sqrt{n})$
- also strided component

*memory complexity and compute scale w/ window size

Big bird (zaheer et Al, 2021)



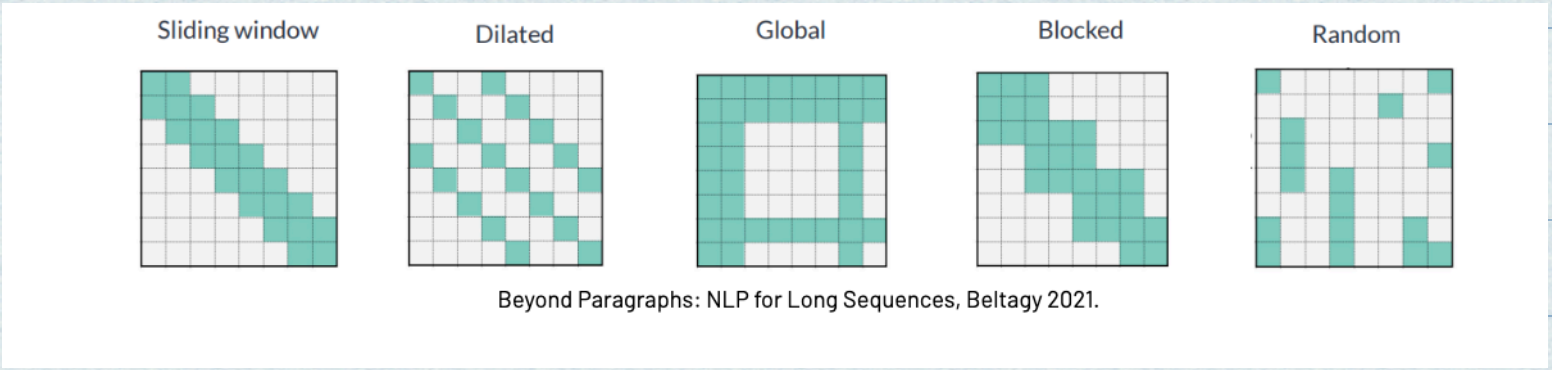
(a) Random attention (b) Window attention (c) Global Attention (d) BIGBIRD

BigBird: combines 3 dif. types of sparsity:

(1) window: local attn as tokens close together are likely to be related

(2) global tokens - attend to <u>All</u> tokens in sequence

(3) random - reduces distance btwn nodes - attn pattern can be viewed as fully connected graphs



Beyond Paragraphs: NLP for Long Sequences, Beltagy 2021.

A few common fixed sparsity patterns: → compute. product

(1) sliding window: each token "attends" w/ tokens in local window

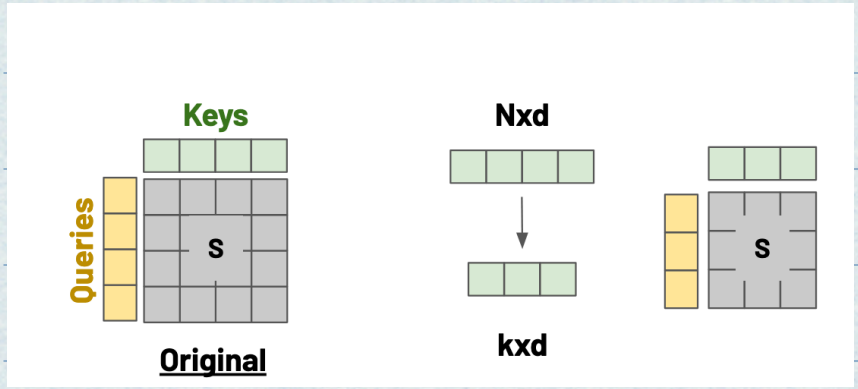
(2) Dilated: fixed interval

(3) Global: few tokens are GLOBALLY attended to

(4) Blocked Pattern: split seq. into fixed blocks

(5) Random

LOW RANK: NxN matrices could be represented by FEWER dimension NOT all values are important - maybe we can project matrices down to smaller dimensions to take most informative direction



Linformer (Wanget al):

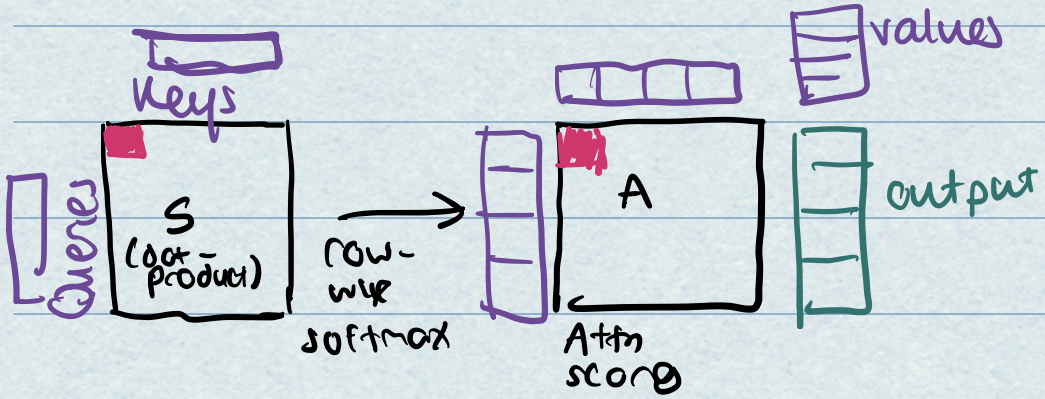
→ project queries/keys down from nxd to kxd

→ so now QK^T is kxk, not nxn

- if $k \ll n$ (or $k \sim \sqrt{n}$) → attn computation becomes $O(n)$
- key question: how do we project q, k, v such that you get a good low rank approx?

KERNELIZATION

- kernel = similarity function btwn two datapoints
- w/ normal softmax attn - we're calculating similarity w/ dot products (of query/key vectors) and softmax
- could we compute similarity in dif. way?



Think of this as "sim(qi, kj)" → the output cell in A.

* Linear Transformers - Katharopoulos et al, 2021

→ Key idea:

- $sim(q_i, k_j) \approx f(q_i) \cdot f(k_j)$

where "f" is a feature map

- the idea is that the product of these feature maps approximates similarity

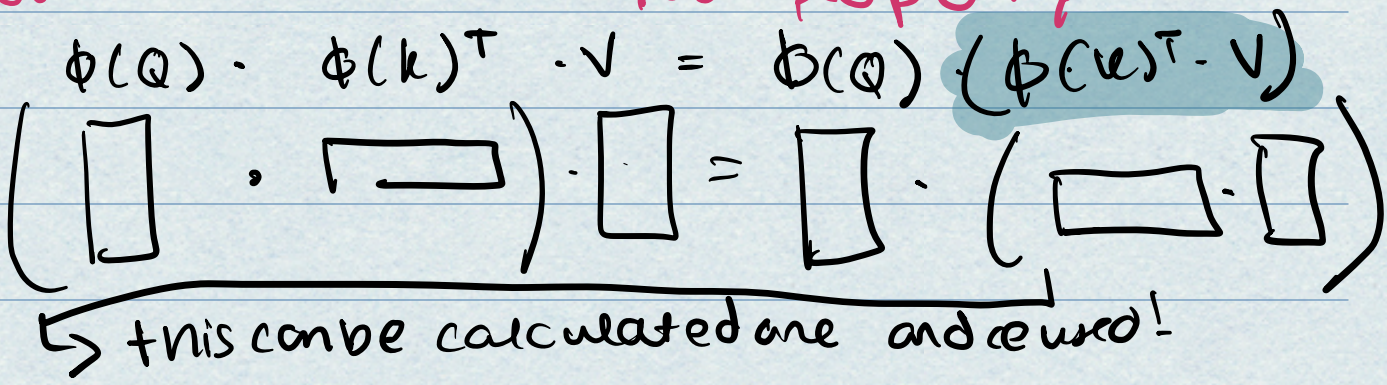
→ This ends up working out to:

$$\cancel{\text{softmax}}(QK^T) \cdot V = \frac{\phi(Q) \cdot \phi(K)^T \cdot V}{\sum_{i=1}^{N} (\phi(Q) \cdot \phi(K)^T)_i}$$

(eliminate softmax)

- now use associative property:

$$\phi(Q) \cdot \phi(K)^T \cdot V = \phi(Q) \cdot (\phi(K)^T \cdot V)$$



↳ this can be calculated once and reused!

result: $O(ND^2)$ complexity

usually $D \ll N$ → hence "linear"

- General issue: are these methods widely adopted??
  - no
  - 2 problems:

→ can have LARGE accuracy drops in favor of speed

→ even if they REDUCE FLOP count, they may not have proportional speedup

- Recap of last lecture: ITS HARD to fully utilize GPU

↳ in many cases - memory can be bottleneck - NOT FLOPS

→ attn is $O(N^2 d)$ in terms of memory accesses

↳ main issue: full $O(N \times N)$ matrix can't be loaded into local SRAM of GPU

*Is there a fast, EXACT, attn algorithm?

↳ yes, there is! ** FLASH ATTENTION **

- and it takes advantage of following efficient HW principles:

① Fusion - composite operations on data saves trips to/from memory

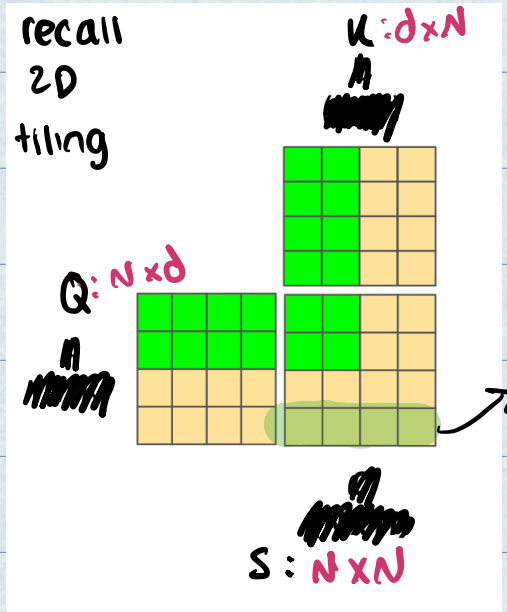
② Tiling - assign subsets of independent work to dif. parallel processing units

③ Caching vs. recomputation - cache for compute bound workloads, recompute for memory bound

④ Hardware specific optimizations - use bf16/fp16 tensor cores effectively?

→ key idea: lets load Q,K intiles and compute full attn output w.r.t those tiles.

- we have todo softmax (QK^T)
- we can definitely do MM in tiles!



But what about softmax?

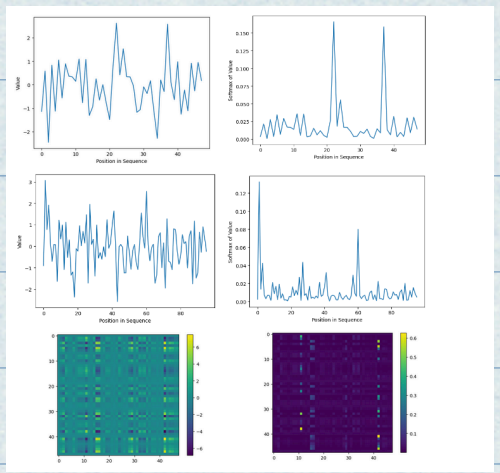
- recall that softmax is on a entire ROW of the output
- if we only have blocks, NOT full row, now do we compute softmax?

→ recap of softmax

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

- exponentiate vector, divide by sum of exponentiated vectors
- in order to give probability distribution

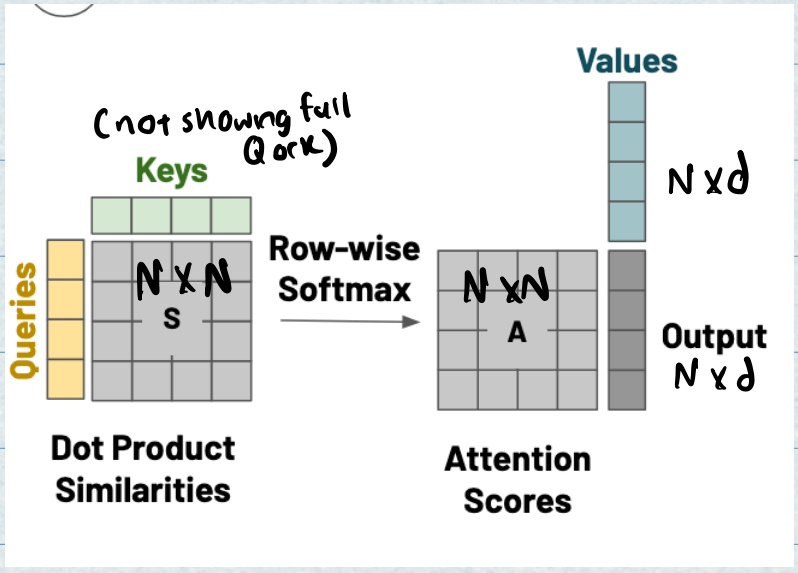
- Intuitively. softmax takes RAW scores on left and makes them spikier (isolating most important token interactions for attention, suppresses unimportant ones)



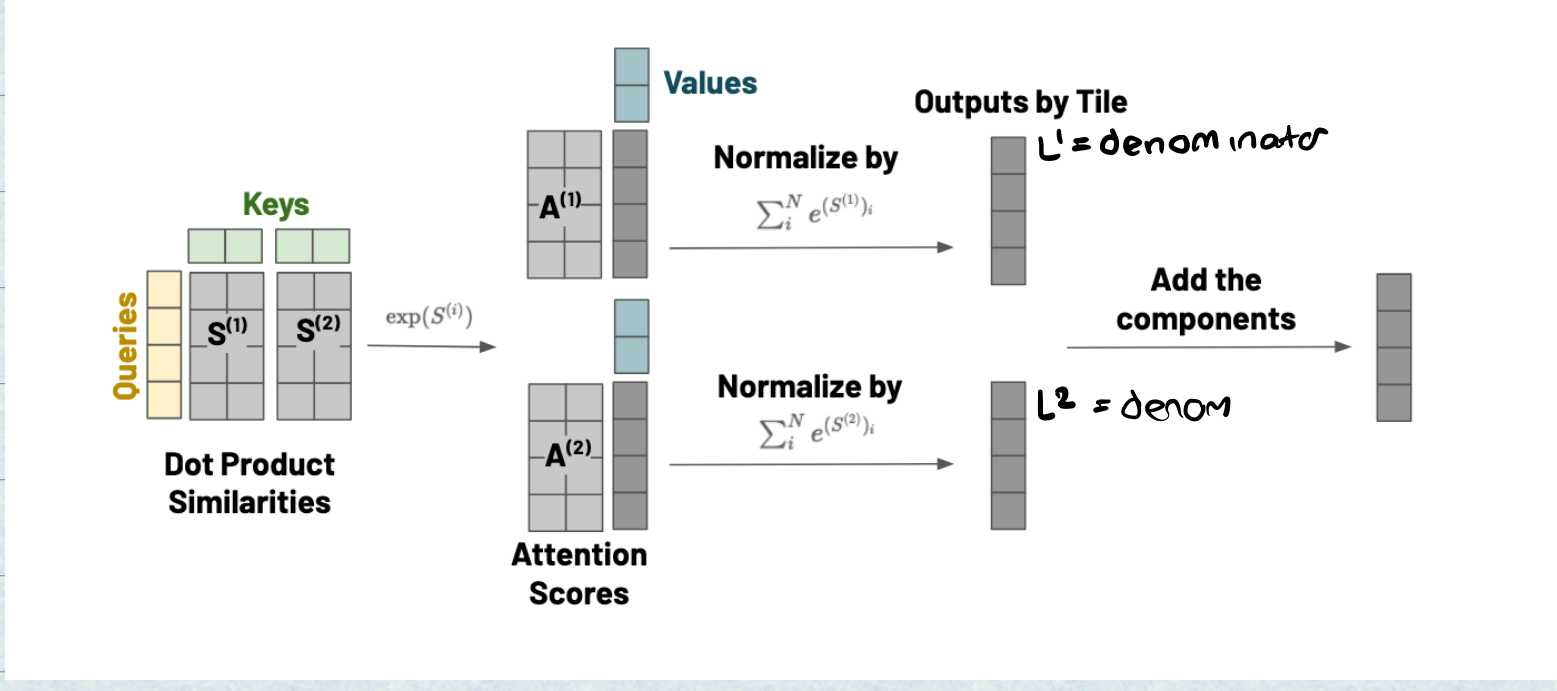
But sadly, we need access to whole row!

→ how could we potentially tile the attention computation?

- visually, if this is untiled attn:



- in tiled attn, we could split K and V



Softmax would work out to:

$$\left(\frac{(A^{(1)} + A^{(2)})\ V}{L^{(1)} + L^{(2)}}\right)$$

But in above pic we are computing:

$$\frac{A^{(1)}}{L^{(1)}} \cdot V^{(1)} + \frac{A^{(2)} \cdot V^{(2)}}{L^{(2)}}$$

* these are not the same ï

- remember:

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

$\rightarrow A_i^{(1)} = e^{(s^{(1)})_i}$

$A^{(2)}{}_i = e^{(s^{(2)})_i}$

$\rightarrow L^{(1)} = \sum_{i=1}^{N} e^{s^{(1)}{}_i}$

$L^{(2)} = \sum_{i=1}^{N} e^{s^{(2)}{}_i}$

- we want: denom $= L = \sum_{i=1}^{N} \left(e^{(s^1)_i} + e^{(s^2)_i} \right)$

* Key idea: Rescaling.

1) Tile 1 gets $V^{(1)}$: $O^{(1)} = \left(\frac{A^{(1)}}{L^{(1)}} \right) \cdot V^{(1)}$

2) Update $O^{(1)}$ by dividing by denom. we want:

$$O^{(1)} = O^{(1)} \cdot \frac{L^{(1)}}{L^{(1)} + L^{(2)}}$$

$\rightarrow$ rescaling factor

- for 2nd tile, we similarly set:

$$O^{(2)} = \frac{A^{(2)} \cdot V^{(2)}}{L^{(1)} + L^{(2)}}$$

$\rightarrow$ now $O = O^{(1)} + O^{(2)}$

* Quick Detour: Numerical stability

```
def softmax(x):
    # assumes x is a vector
    return np.exp(x) / np.sum(np.exp(x))

x = np.array([1.2, 2000, -4000, 0.0])
softmax(x)
```

✓ 0.0s

Problem: there can be instabilities due to limits of FPs

```
/tmp/ipykernel_1217769/580451730.py:3: RuntimeWarning: overflow encountered in exp
  return np.exp(x) / np.sum(np.exp(x))
/tmp/ipykernel_1217769/580451730.py:3: RuntimeWarning: invalid value encountered in divide
  return np.exp(x) / np.sum(np.exp(x))

array([ 0., nan,  0.,  0.])
```

↳ we can improve these instabilities by SUBTRACTING max value of vector in

soft max:

$$\text{softmax}(x)_i = \frac{e^{x_i - \max(x)}}{\sum_j e^{x_j - \max(x)}}$$

- How do we keep track of max (x) w/o full access to x?

- suppose we've computed:

Tile 1: $e^{x_1 - \max(x_1)}$

Tile 2: $e^{x_2 - \max(x_2)}$

Tile N: $e^{x_N - \max(x_N)}$

→ we can RESCALE by overall max: (max)

Tile 1: $e^{(x_1 - \max(x_1))} \cdot e^{(\max(x_1) - \max)}$

$= e^{x_1 - \max}$

similarly for

Tile 2: $e^{x_n - \max(x_n)} \cdot e^{(\max(x_n) - \max)}$

$= e^{x_n - \max}$

- Final problem: How do we do backwards pass?

↳ usually, we store: QK^T

$\text{softmax}(QK^T)$ → $N \times N$ matrices we HAVEN'T materialized

→ in order to compute gradients wrt Q, K, V

- Trick: cache rescaling vectors! (to recompute attention)

→ for block 1: $L^{(1)}$

→ for Block 2: $\frac{L^{(1)}}{L^{(1)} + L^{(2)}}$

→ for Block 3: $\frac{L^{(1)}}{L^{(1)} + L^{(2)}}$ $\frac{L^{(1)} + L^{(2)}}{L^{(1)} + L^{(2)} + L^{(3)}}$

→ we are storing these successive sums to be able to get each $A^{(i)}$ and $O^{(i)}$

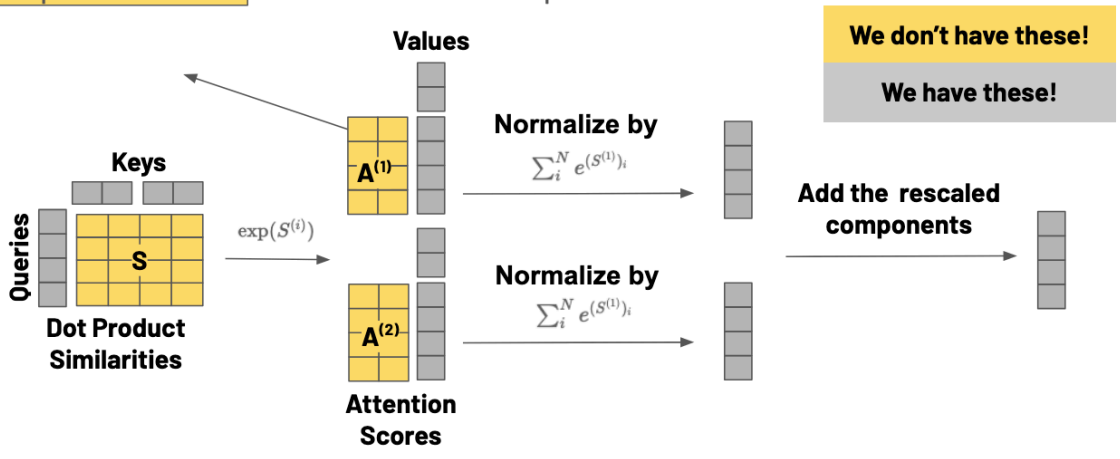
- also cache max per block:

$$M(x) = \max\left[x^{(1)}, x^{(2)} \dots x^{(n)}\right]$$

* now we can RECOMPUTE attn matrix (and output of softmax) in BW pass.

↳ this will have size N (per row):

By storing the softmax normalization from the forward pass (size N), quickly recompute attention in backward from inputs in SRAM...



we now have ↗ recomputation in BW pass

Results:

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)

Key Takeaway: Lower FLOPs != Wall Clock Speedup

- what about flash attention 2?

- limitations of FA1: suboptimal managing of work partitioning across threads/warps

*main updates:

1) use new CUTLASS primitives - which takes care of memory management, helps mitigate concurrency issues

(2) Parallelize over more dimensions:

- sequence length (queries)
- batch size (dif inputs)
- # attn heads (dif Q, K, V)

* see flash attn paper for more details!

-FURTHER REFERENCES:

CS229s Lecture 5 on hardware aware algorithm design (where this lecture came from):https://docs.google.com/presentation/d/1kY5MOOJfWny7SvV4D7YeSAulOTMnnVJTKKG5DQYLid0/edit#slide=id.g24c4f25c1b8_0_2

Efficient Transformers, A Survey: https://arxiv.org/pdf/2009.06732 (has notes on sparsity)

Linformer (low rank approximation of transformers): https://arxiv.org/pdf/2006.04768.

Linear Transformers (Kernel Based): https://arxiv.org/pdf/2006.16236

Flash Attention: https://arxiv.org/pdf/2205.14135

Flash attention version 2: https://arxiv.org/pdf/2307.08691