Dissecting a Transformer: Variants of Attention

Mausam

(Based on slides of Mitesh Khapra, Arun Prakash A)

Motivation: Long Context

- Quadratic attention expensive as sentence length increases
- Do we NEED “all pair” attention?

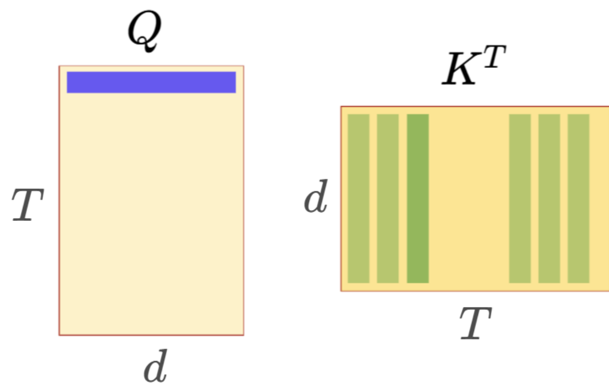
- Example use cases:
- Summarize a 64000 word book
- Generate a 5000 word story
- Find an answer in a long instruction manual
- Sequence length of 10s audio signal can be 80000 inputs (based on 8kHz sampling rate)
- Sequence length of 100x100 color image is 30000

Scaling Up Context Length

Model	Context length
LLAMA-2	4K
Mistral-7B	8K
GPT-4, Gemini 1.0	32K
Mosaic ML MPT	65K
Anthropic Claude	100K
Gemini 1.5 pro, GPT-4 Turbo	128K
Claude 2.1	200K
Gemini 1.5 pro (limited)	2 million (for public) 10 million (experimental)

Complexity of Self Attention

Time Complexity of Self Attention



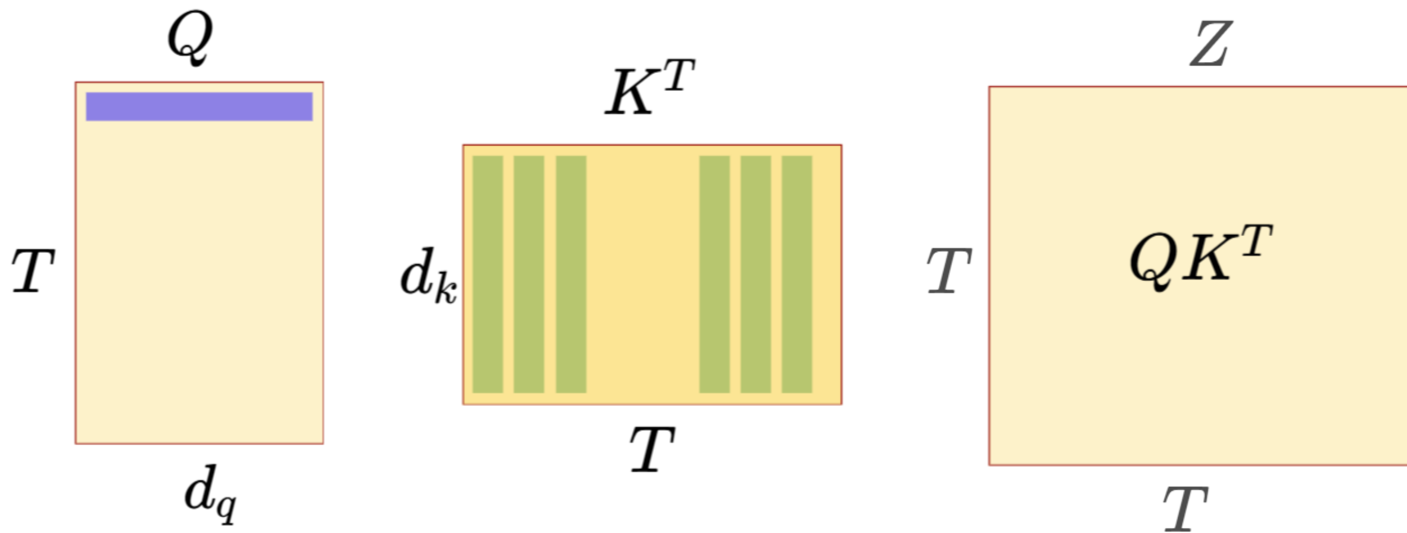
One dot product between Q_iK_j requires:

d multiplications

d-1 additions

- Context Window Size: T
- The size of the (projected) embeddings: $d = d_q = d_k$
- Number of operations for computing one row of unnormalized attention weights $Z \in \mathbb{R}^{T \times T}$ is
  - T.d multiplications, T.(d-1) additions
- To compute QK^T we need $O(T^2d)$ operations

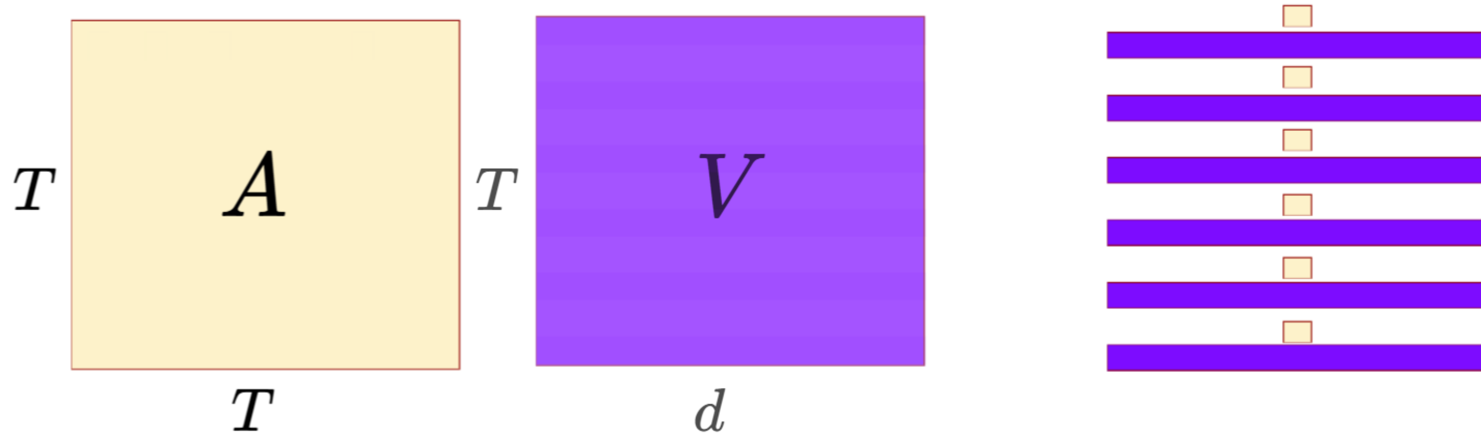
Time Complexity of Self Attention



- Normalize each row of Z: O(T)
- T rows in Z. ➔ computing softmax: $O(T^2)$
- Computational Complexity of A: $O(T^2d)$

$$A = softmax(\frac{QK^T}{\sqrt{d}})$$

Time Complexity of Self Attention



- Attention Score A multiplied with value matrix V: $O(T^2d)$
- Entire self attention block is quadratic $O(T^2d)$ in length of input

Space Complexity of Self Attention

B: Batch Size

$|\mathcal{V}|$: Number of tokens in the vocabulary

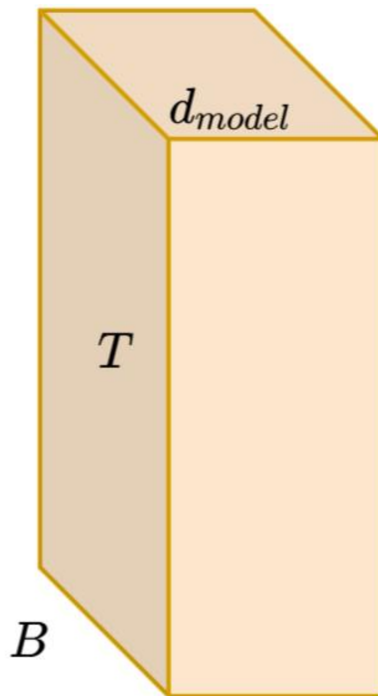
T : Context (sequence) Length

d_{ff} : Dimension of feed-forward layer

d_{model} : Dimension of input embedding, $d_{att} = 512 = \frac{d_{ff}}{4}$

n_h : Number of attention heads

N : Number of parameters in the model



Space Complexity of Self Attention

- Total Memory required is the sum of memory for
  - Storing Activation values
  - Storing Parameters
  - Storing Gradients

- Which of these do you think takes the most memory?

Space Complexity of Transformer Layer

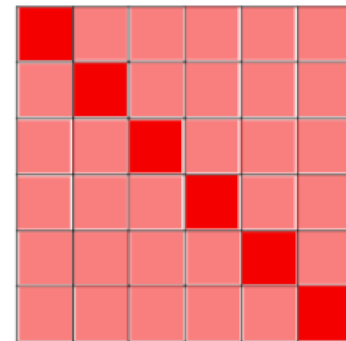
- Parameters
  - W^Q, W^K, W^V, W^O : $4n_h.(d_{model}xd)$
  - FF layer: $2(d_{model}xd_{ff})$
  - LayerNorm: $4d_{model}$
- For a single sample (MHA)
  - mha = Q ; K ; V
  - mha = $3n_h.(Txd)$
- For Batch: $3BT(n_h.d)$

- $Z = QK^T$
  - Bn_hT^2
- $A=softmax(QK^T)$
  - Bn_hT^2
- Projection via W^O
  - $BT(n_hd + d_{model})$
- FF
  - $BT(d_{ff} + d_{model})$
- LayerNorm
  - $2BT(d_{model})$

Transformer is linear in batch size and quadratic in context window length T.

How to reduce complexity?

Objective



- What we have is the full attention mechanism with $O(T^2d)$.
- What we want is sub-quadratic complexity O(cTd).
  - How do we achieve that?
- A study on BERT in NLP tasks concluded that the neighbouring inner products are extremely important in self-attention and not all heads attend to all tokens.

What Does BERT Look At?
An Analysis of BERT's Attention

Kevin Clark[†] Urvashi Khandelwal[†] Omer Levy[‡] Christopher D. Manning[†]

Attention Patterns

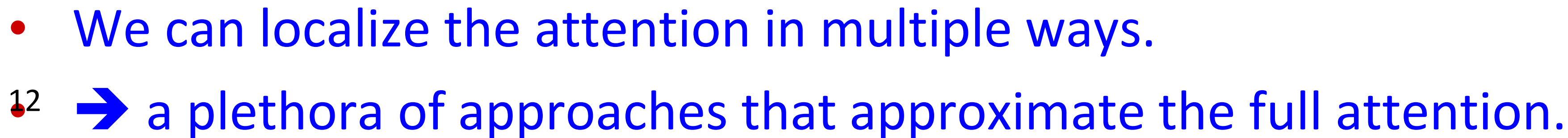
Head 1-1
Attends broadly

Head 3-1
Attends to next token

Head 8-7
Attends to [SEP]

Head 11-6
Attends to periods

... found in taiwan . [SEP] the wingspan is 24 – 28 mm . [SEP]

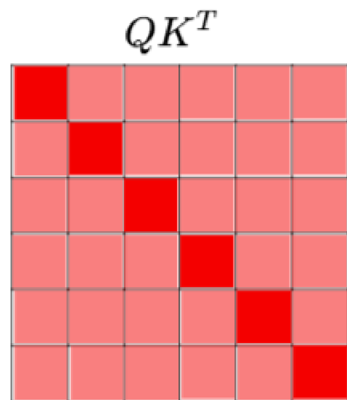


- We can localize the attention in multiple ways.
- ➔ a plethora of approaches that approximate the full attention.

Full Attention → Sparse Attention

- Computational Complexity: $O(T^2d)$
- Space Complexity: $O(T^2)$
- Goal: reduce computational complexity

- Approach:
  - Have one word attend to subset of words



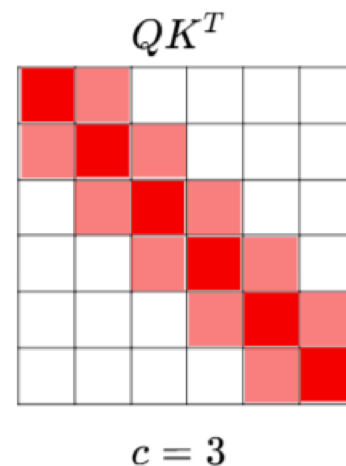
(the score for the word itself ($q_i k_i^T$))

Sparse Attention

Local (Sliding Window) Attention

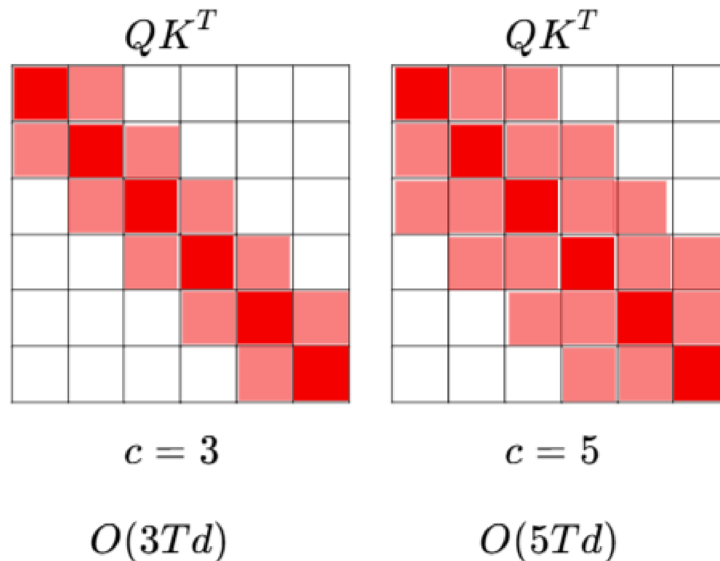
- Each word attends to $\left\lfloor \frac{c}{2} \right\rfloor$ words to the left and to $\left\lfloor \frac{c}{2} \right\rfloor$ words to the right.

- Time Complexity: O(cTd)

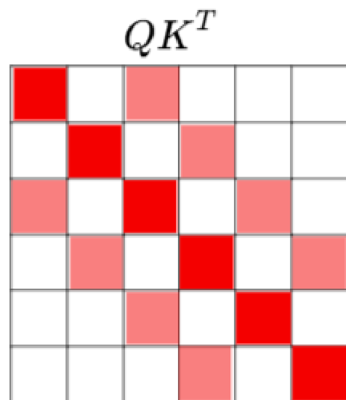


Local (Sliding Window) Attention

We could vary c for each layer of the model such that topmost layer uses the full global attention.



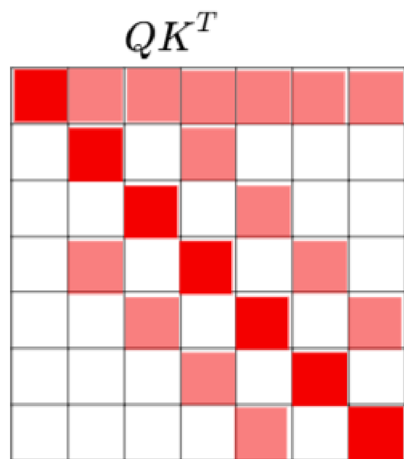
Dilated Attention



- Similar to dilated convolutions
- Different dilation rates in diff heads/layers

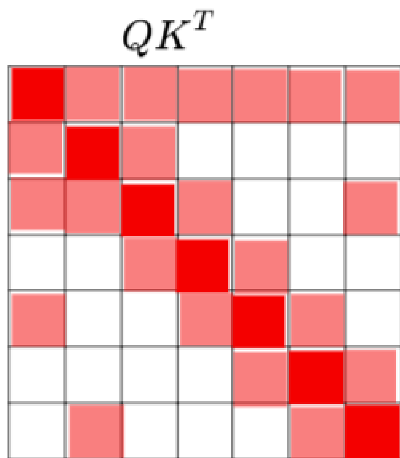
- Receptive Field View
  - Sliding Window: layer1: 5, layer2: 10, layer3: 15
  - Dilated:
    - layer1 (0): 5, layer2 (5): 25, layer3 (25): 125

Global + Local (Dilated) Attention



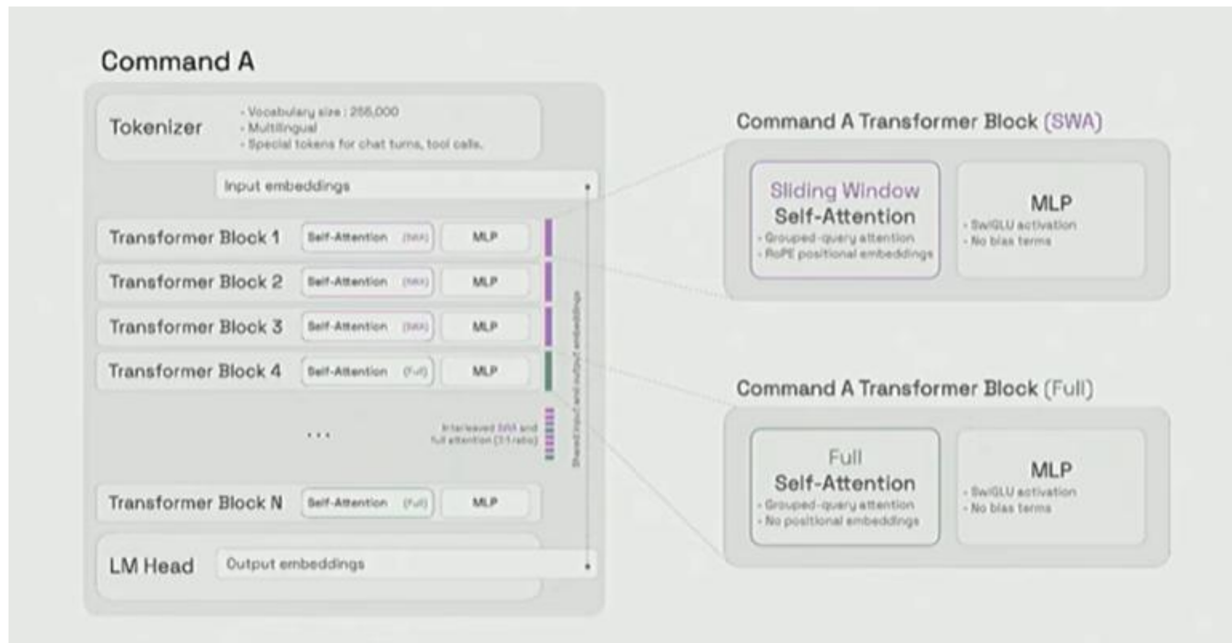
- A few words are global, rest as before.
- Tokens like [cls] require full attention to get good performance in downstream tasks.
- We manually set tokens that require global attention based on the task.

BigBird: Global + Random + Local (S.W.) Attention



- Intersperse random tokens
- Along with other ideas.

Recent Strategy

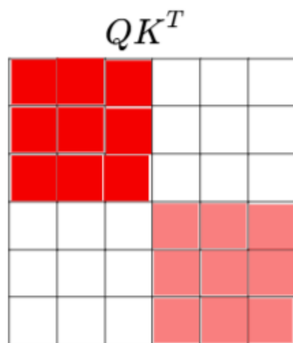


- Every 4th layer is full self-attention without position embeddings

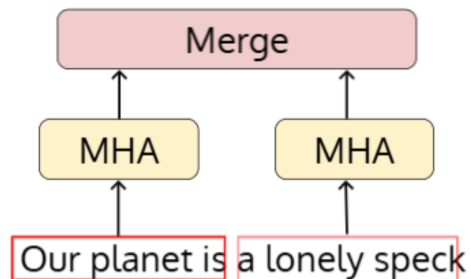
Sliding Window → Block Attention

- Intuition:
  - How to process a paragraph?
    - Split into sentences
  - How to process a sentence?
    - Split into blocks
- Approach: all pair attention within a block

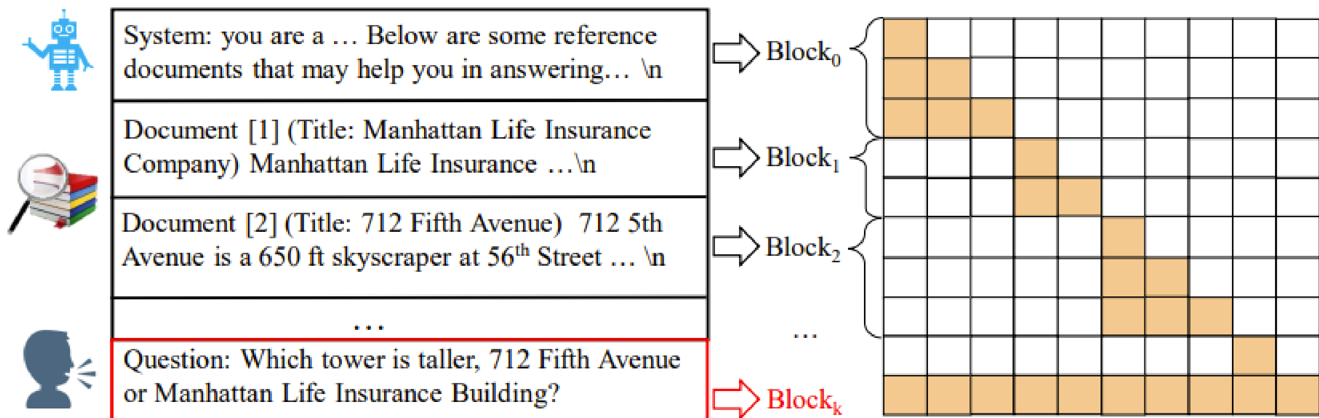
Local Block Attention



Computational complexity $O(\frac{T^2}{2})$



Local Block Attention



90%+ compute saving at ~no accuracy loss

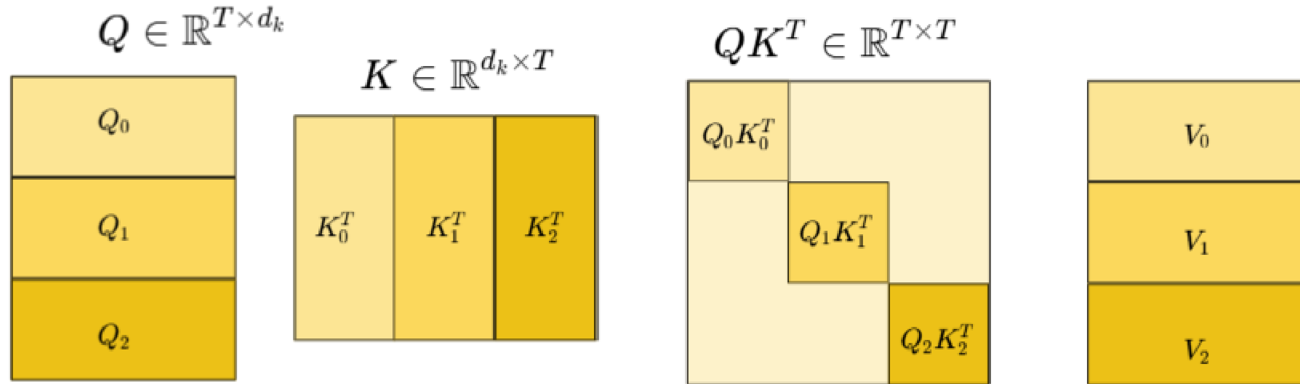
BLOCK-ATTENTION FOR EFFICIENT PREFILLING

Dongyang Ma*
Tencent
dongyangma@tencent.com

Yan Wang*†
Tencent
yanwang.branden@gmail.com

Tian Lan
lantiangmftby@gmail.com

Block Attention



- We can divide Q and K matrices into n blocks each. This will result in n^2 blocks in QK^T as shown in the figure above.
- Can we generalize this?
  - Idea: permute blocks in different heads.

Block Attention

Consider the permutation,

$\pi = \text{perm}(0, 1, 2, \cdots, n-1)$

Let the k-th element of π be $\pi(k)$. We define the masking matrix M of size $T \times T$

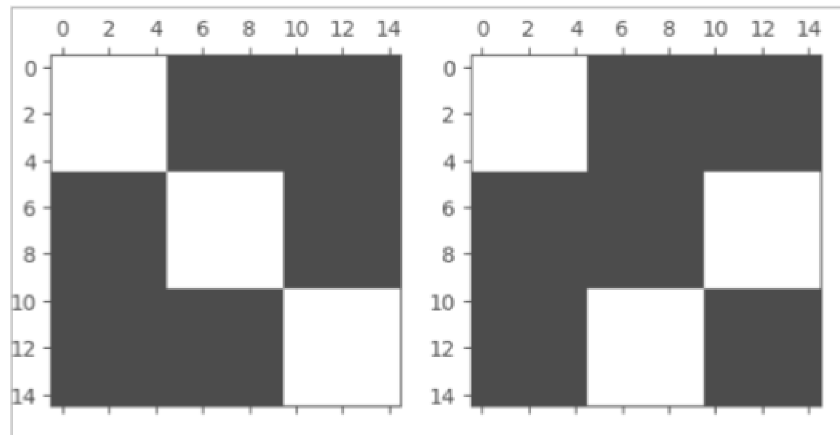
$$M_{ij} = \begin{cases} 1 & \text{if} \quad \pi(\lfloor \frac{in}{T} \rfloor) = \lfloor \frac{jn}{T} \rfloor \\ 0 & \text{otherwise} \end{cases}$$

Block Attention is given by,

$$Z = softmax(QK^T \odot M)$$

For example, suppose context length $T = 15$ and the number of blocks $n = 3$.

Then we would have 3! ways of permuting the blocks.



$Identity: \pi = (0, 1, 2)$ $n = 3, T = 15$ $\pi = (0, 2, 1)$ $n = 3, T = 15$

Write Q,K^T, and V into $n = 3$ block matrices (following the blocks in M)

$$\mathbf{Q} = (Q_0, Q_1, Q_2) \in \mathbb{R}^{5 \times d}$$

$$\mathbf{K} = (K_{\pi(0)}, K_{\pi(1)}, K_{\pi(2)}) \in \mathbb{R}^{5 \times d}$$

$$\mathbf{V} = (V_{\pi(0)}, V_{\pi(1)}, V_{\pi(2)}) \in \mathbb{R}^{5 \times d}$$

Block-wise Multi-Head Attention

Using these we can compute Block Attention as follows

$$\text{Block-wise Attention}(Q, K, V, M) = \begin{bmatrix} softmax(Q_0 K_{\pi(0)}^T) V_{\pi(0)} \\ \vdots \\ softmax(Q_n K_{\pi(n-1)}^T) V_{\pi(n-1)} \end{bmatrix} \begin{matrix} \rightarrow \frac{T}{n} \times \frac{T}{n} \times d \\ \\ \rightarrow \frac{T}{n} \times \frac{T}{n} \times d \end{matrix}$$

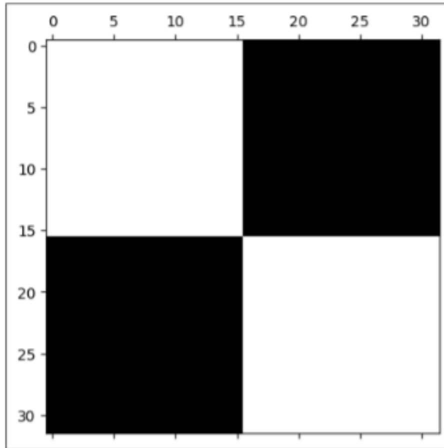
$$= O(\frac{T}{n} \times \frac{T}{n} \times n \times d)$$

$$= O(\frac{T^2 d}{n})$$

Use different permutation per MHA head.

What Happens if we increase n?

$T = 32, n = 2$

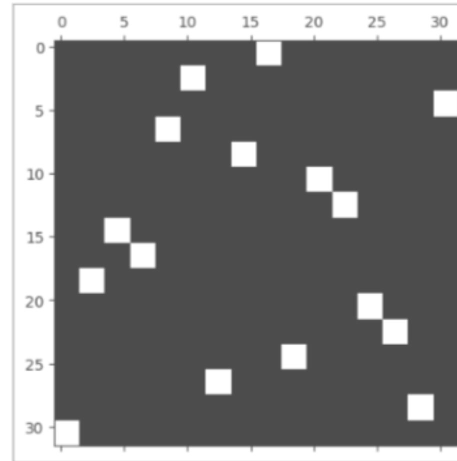


$Identity : (0, 1)$

50% non-zero values

(50% sparse)

$T = 32, n = 16$



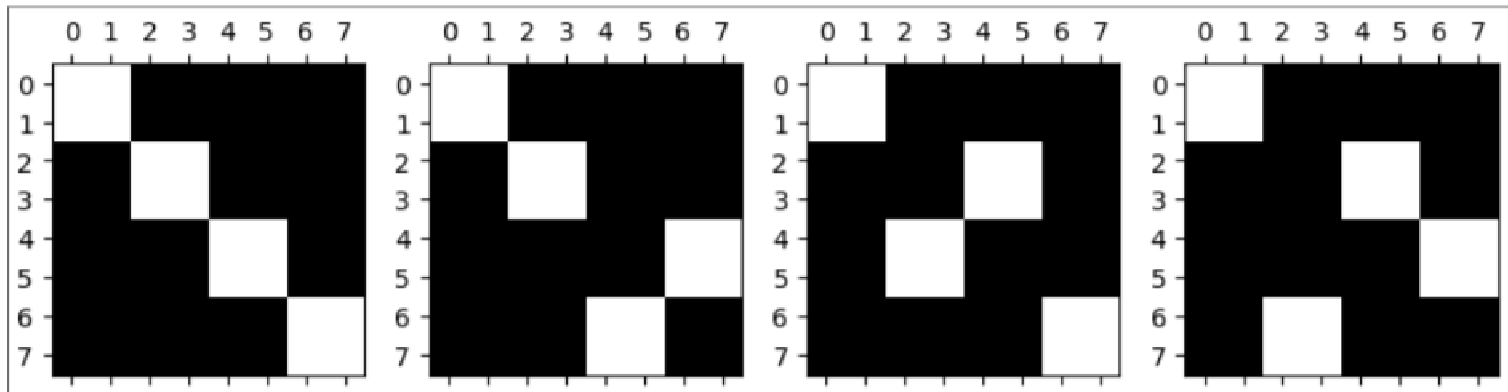
$(8, 5, 15, 4, 7, 10, 11, 2, 3, 1, 12, 13, 9, 6, 14, 0)$

6.25% non-zero values

(93% sparse)

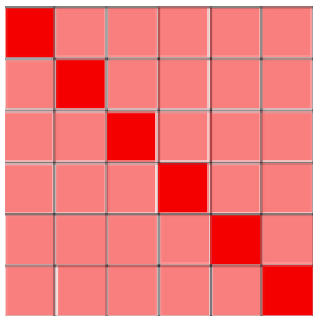
Capturing Long Range Dependency

Blockwise sparsity captures both local and long-distance dependencies in a memory-efficient way using different permutations

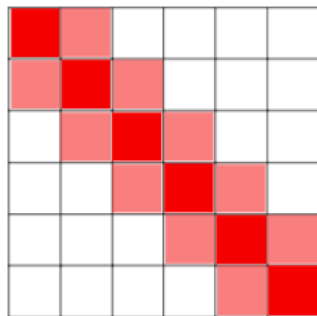


Empirically, it is observed that the identity permutation is more important than other permutations

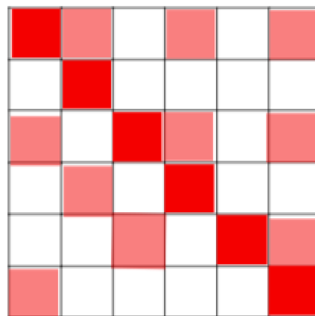
Summary (so far)



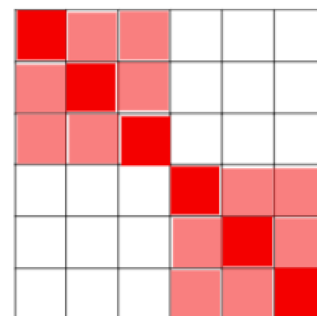
$O(T^2d)$



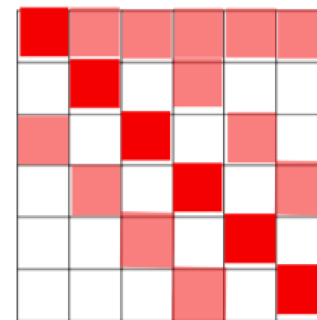
Strided Local Attention



Random Local Attention



Sparse Block Attention



Global+local attention

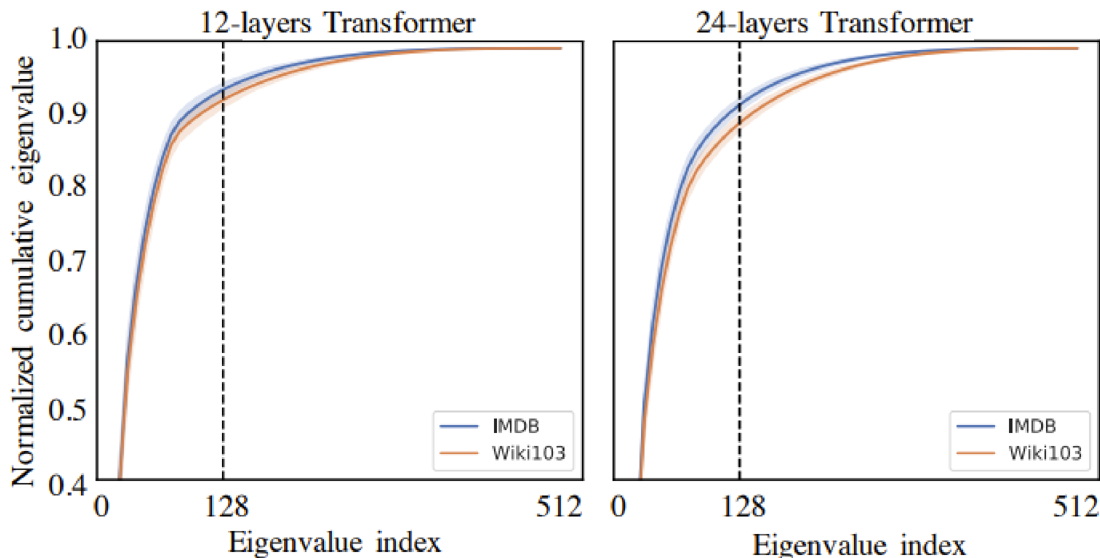
Weaknesses of Sparse Attention

- Rigid attention patterns
- Harder to implement efficiently in GPUs
- Limited benefit in practice

Sparse → Dense (Simple) Attention

- $A=QK^T$ matrix is assumed to structurally sparse
- What if the matrix is dense, but… simple?
- $A \approx UV^T$ where U, V are $\mathbb{R}^{T\times r}$ where r << T
  - Although A is TxT it lives in a small subspace
- Intuition: All attention patterns are combinations of a **small number of global patterns**. Example:
  - Attend to previous token
  - Attend to nouns
  - Attend to sentence start…

Empirical Observation



- Apply SVD on $A=QK^T$ across different layers and different heads of the model, and plot the normalized cumulative singular value averaged over 10k sentences.

➔ QK^T is low-rank

LinFormer

$$Q \in \mathbb{R}^{T \times d} \quad K \in \mathbb{R}^{T \times d} \quad V \in \mathbb{R}^{T \times d}$$

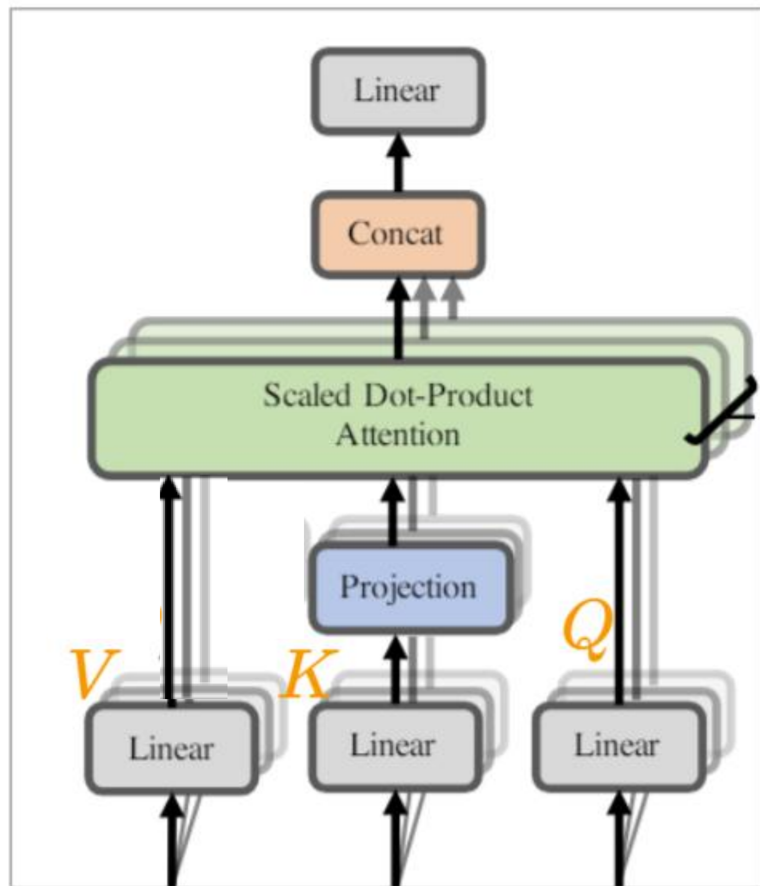
Introduce two learnable linear projection matrices

$$E, F \in \mathbb{R}^{k \times T}$$

$$A = softmax(\frac{Q(EK)^T}{\sqrt{d}})$$

$$A \in \mathbb{R}^{T \times k}$$

- How to multiply with V?



LinFormer

$$Q \in \mathbb{R}^{T \times d} \quad K \in \mathbb{R}^{T \times d} \quad V \in \mathbb{R}^{T \times d}$$

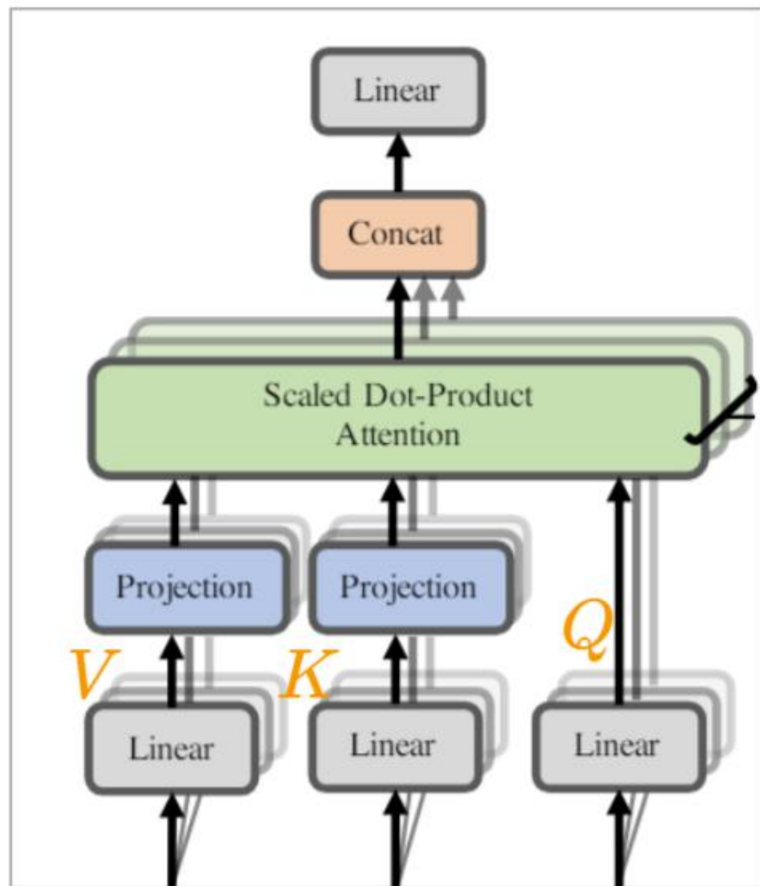
Introduce two learnable linear projection matrices

$$E, F \in \mathbb{R}^{k \times T}$$

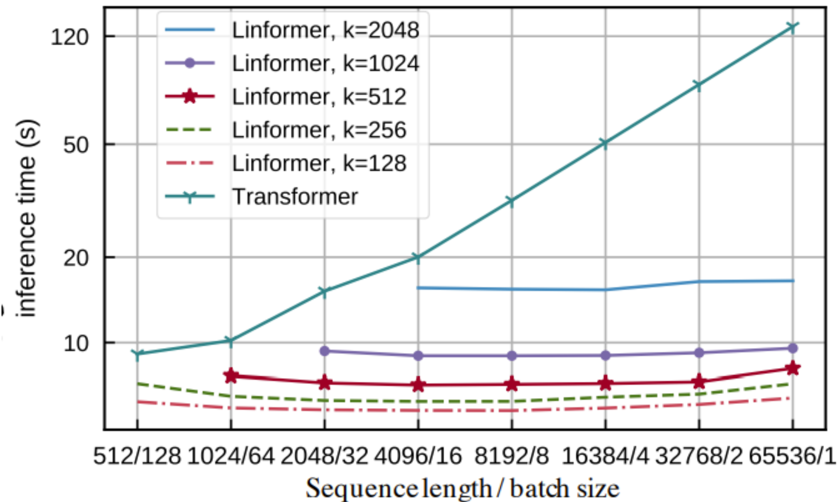
$$A = softmax(\frac{Q(EK)^T}{\sqrt{d}})$$

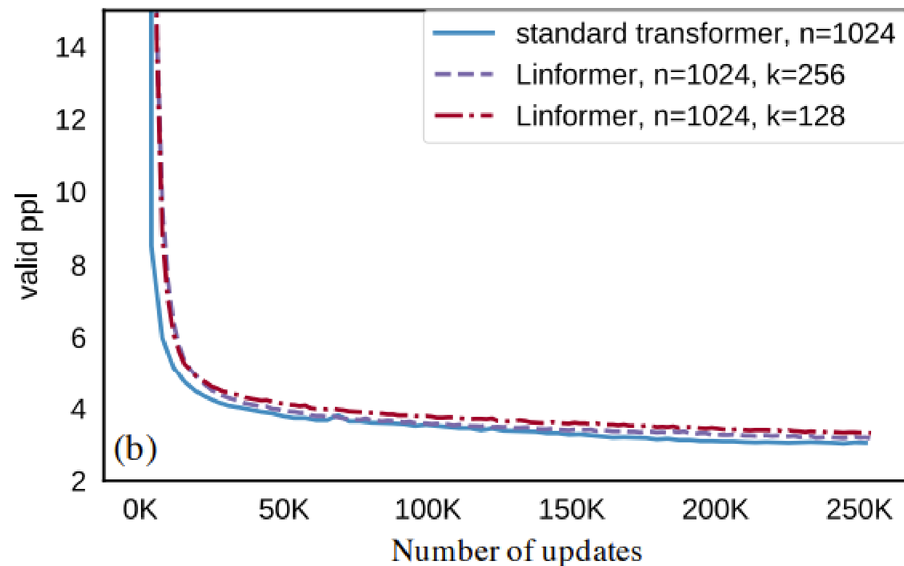
$$A \in \mathbb{R}^{T \times k}$$

- AV → AFV
- Complexity: O(kTd) based on rank k



LinFormer: Results





Weakness of LinFormer

- Projection matrices are
  - Dependent on length
  - May lose information: quality drops for long range dependencies
- Objective: instead of approximating matrix (QK^T), let us approximate (factorize) the softmax itself!

PerFormer

Rethinking Attention with Performers

Krzysztof Choromanski*[1], Valerii Likhosherstov*[2], David Dohan*[1], Xingyou Song*[1]
Andreea Gane*[1], Tamas Sarlos*[1], Peter Hawkins*[1], Jared Davis*[3], Afroz Mohiuddin[1]
Lukasz Kaiser[1], David Belanger[1], Lucy Colwell[1,2], Adrian Weller[2,4]
[1]Google [2]University of Cambridge [3]DeepMind [4]Alan Turing Institute

All that we need is an attention matrix such that each row sums up to one

$$Atten = softmax(\frac{QK^T}{\sqrt{d}})$$

$$Atten = D^{-1}A$$

where,

$$A = exp(QK^T/\sqrt{d})$$

$$D = diag(A\mathbf{1})$$

$\mathbf{1}$ is a vector of all ones of length T

Fundamentally, the exponential function in A takes in the query and key vectors and outputs a positive number.

- $A(i,j) = \exp(q_i.k_j/\sqrt{d})$

$$A(i,j) = \mathbb{E}(\phi(q_i)^T\phi(k_j))$$

$$\phi(x): \mathbb{R}^d \rightarrow \mathbb{R}_+^r, r > 0$$

Where $\phi(x)$ is some randomized feature mapping and $\mathbb{E}$ is an expectation operator.

Note that $r > 0$, and it is not necessary that $r < d$.

For example, $T = 64k, d = 4k$, then $r = 8k$ is also reasonable

How do we compute $\phi(x)$? $\phi(x) : \mathbb{R}^d \rightarrow \mathbb{R}^r_+$

The generalized form of a kernel transformation is given by,

$$\phi(x) = \frac{h(x)}{\sqrt{r}}\left(f_1(w_1^T x), \cdots, f_1(w_r^T x), \cdots, f_l(w_1^T x), \cdots, f_l(w_r^T x)\right)$$

where,

$$f_1, f_2, \cdots, f_l : \mathbb{R} \rightarrow \mathbb{R}$$

$w_1, \cdots, w_r : \mathbb{R}^d$ where w_i is sampled from $\mathcal{N}(\mathbf{0}_d, \mathbf{I}_d)$

$w_i^T x$ is a random projection, that is, the vector x is projected in a random direction given by w_i vector.

We can construct a random feature vector of size r by setting $l = 1$, considering $f_1 = exp$ and $h(x) = exp(\frac{-||x||^2}{2})$ (Note, this particular setting is theoretically motivated among many possible choices.)

Therefore,

$$\phi(x) = \frac{exp(\frac{-||x||^2}{2})}{\sqrt{r}} \left(exp(w_1^T x), \cdots, exp(w_r^T x) \right)$$

$$w_i \sim \mathcal{N}(\mathbf{0}_d, \mathbf{I}_d)$$

We can collect all w_is in a matrix $\mathbf{W}$

$$\phi(X) = \frac{1}{\sqrt{r}} exp\left(\mathbf{W}^T X - \frac{||X||^2}{2} \right)$$

where $\mathbf{W} \in \mathbb{R}^{d \times r}$,$X \in \mathbb{R}^{d \times T}$ and $||X||^2$ is the $L2-$norm of column vectors in X.

The $\mathbf{W}$ matrix is orthogonalized using Gram-Schmidt orthogonalization procedure.

Orthogonalization of W helps reduce the variance of the softmax estimator for any dimensionality of d. This requires $r \leq d$.

For all the queries q_is in Q, we can compute Q' as follows

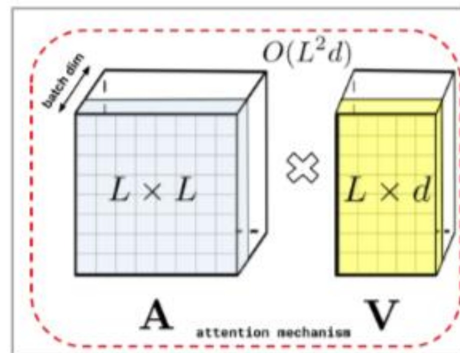
$$Q' = \frac{1}{\sqrt{r}} exp\left(\mathbf{QW^T} - \frac{||Q||^2}{2} \right) \qquad Q' \in \mathbb{R}^{T \times r}$$

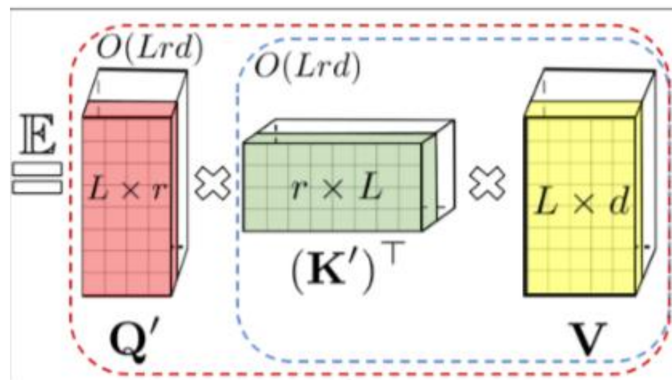
here, $\mathbf{W} \in \mathbb{R}^{r \times d}$,$Q \in \mathbb{R}^{T \times d}$ and $||Q||^2$ is the $L2-$norm of row vectors in Q.

Similary, we can compute $K' \in \mathbf{R}^{T \times r}$.

PerFormer

$L = T$ in the figure below





The approximated attention matrix for the given query and key is given by

$$\hat{A}(i,j) = \frac{1}{r} exp(q_i \mathbf{W}^T - \frac{||q_i||^2}{2}) exp(k_j \mathbf{W}^T - \frac{||k_j||^2}{2})^T$$

and the vectorized version is given by

$$\hat{A} = Q' K'^T$$

The time complexity is $O(rTd)$.

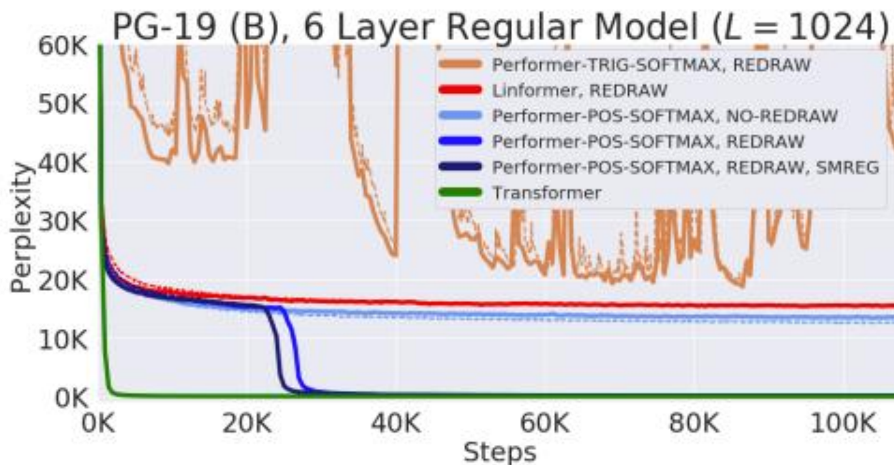
$$\mathrm{Att}_{\leftrightarrow}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \mathbf{D}^{-1}\mathbf{A}\mathbf{V}, \quad \mathbf{A} = \exp(\mathbf{Q}\mathbf{K}^\top / \sqrt{d}), \quad \mathbf{D} = \mathrm{diag}(\mathbf{A}\mathbf{1}_L).$$

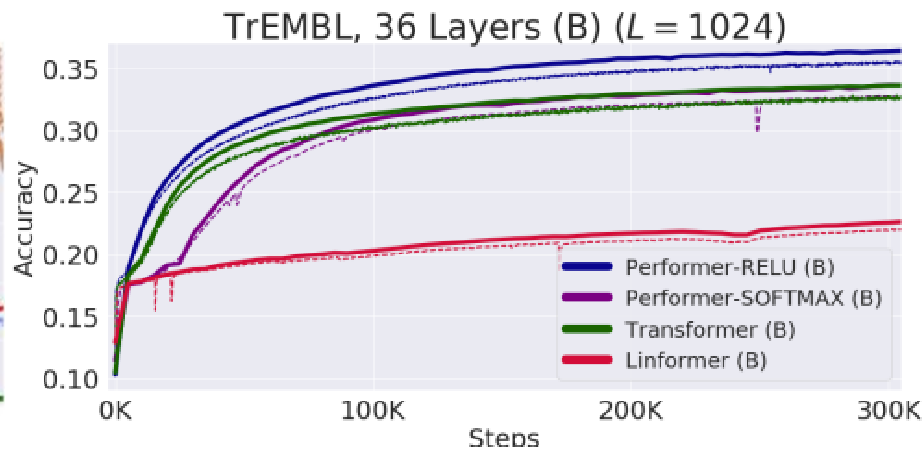
$$\widehat{\mathrm{Att}_{\leftrightarrow}}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \widehat{\mathbf{D}}^{-1}(\mathbf{Q}'((\mathbf{K}')^\top \mathbf{V})), \quad \widehat{\mathbf{D}} = \mathrm{diag}(\mathbf{Q}'((\mathbf{K}')^\top \mathbf{1}_L)).$$

Comparison

LinFormer	PerFormer
Approximates attention matrix	Factorizes softmax computation
Assumes low rank attention	Uses kernel feature maps
Projections are learned	Features are random directions
Compresses the attention table	Changes how similarity is computed so the attention table doesn't need to exist

Experiments





Weaknesses of Performer

- Approximation noise
  - Random features induce variance: hurts training stability, perplexity
- Kernel tricks break causal caching
  - KV caching hard with Performer
- Hardware advances beat algorithmic tricks

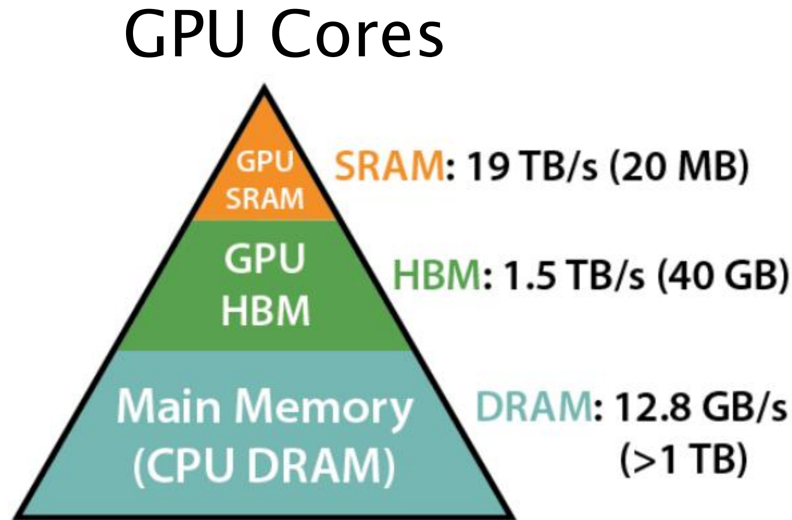
Hardware Advances

Profiling Self-Attention on GPU Hardware

	Operation	FLOPs	Wall Time
1.	$A = \frac{QK^T}{\sqrt{d}}$	$O(T^2 d)$	16%
2.	$A = A + mask$	$O(T^2)$	**36%**
3.	$A = softmax(A)$	$O(T^2)$	**32%**
4.	$output = AV$	$O(T^2 d)$	16%

Masking and Softmask take more wall time even though they are relatively compute light.
Why?

A Closer Look at GPU



$$A = \frac{QK^T}{\sqrt{d}}$$

- Read Q an V of sizes (T, d) from HBM to SRAM
- Compute A of size (T, T) in SRAM
- Write A to HBM from SRAM

I/O operations - $T^2 + 2Td$

Memory I/O Profiling

	Operation	FLOPs	Wall Time	Mem I/O
1.	A			
2.	$A =$			

FLASHATTENTION: Fast and Memory-Efficient Exact Attention with IO-Awareness

Tri Dao[†], Daniel Y. Fu[†], Stefano Ermon[†], Atri Rudra[‡], and Christopher Ré[†]

$d + T^2$

DATA MOVEMENT IS ALL YOU NEED: A CASE STUDY ON OPTIMIZING TRANSFORMERS

Andrei Ivanov [*1] Nikoli Dryden [*1] Tal Ben-Nun [1] Shigang Li [1] Torsten Hoefler [1]

I/O Aware Attention

- Problem – Transformers are bottlenecked by memory access.
- Main Culprit – Realization of (T, T) matrices during attention computation leads to under-utilization of GPU.
- Solution – Compute attention without realizing (T, T) matrices → FlashAttention

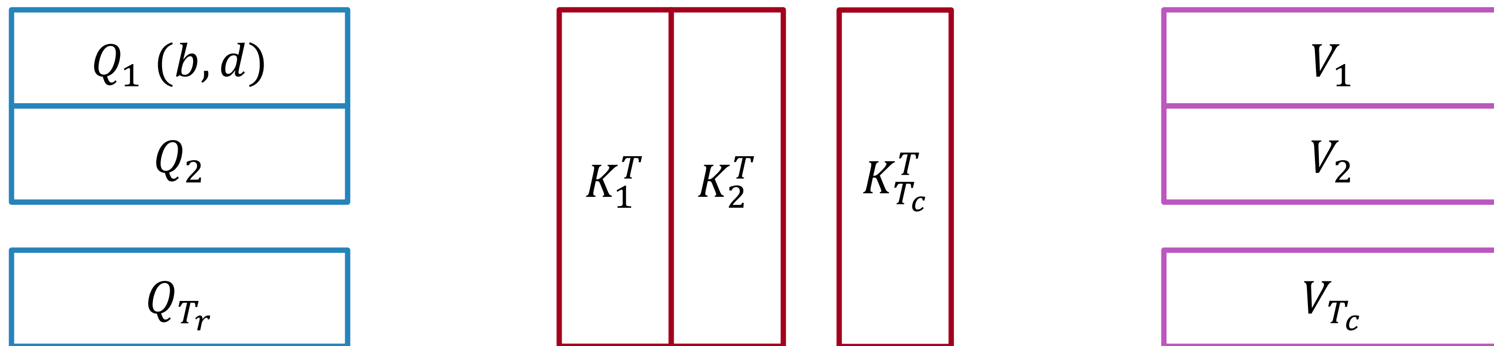
FlashAttention

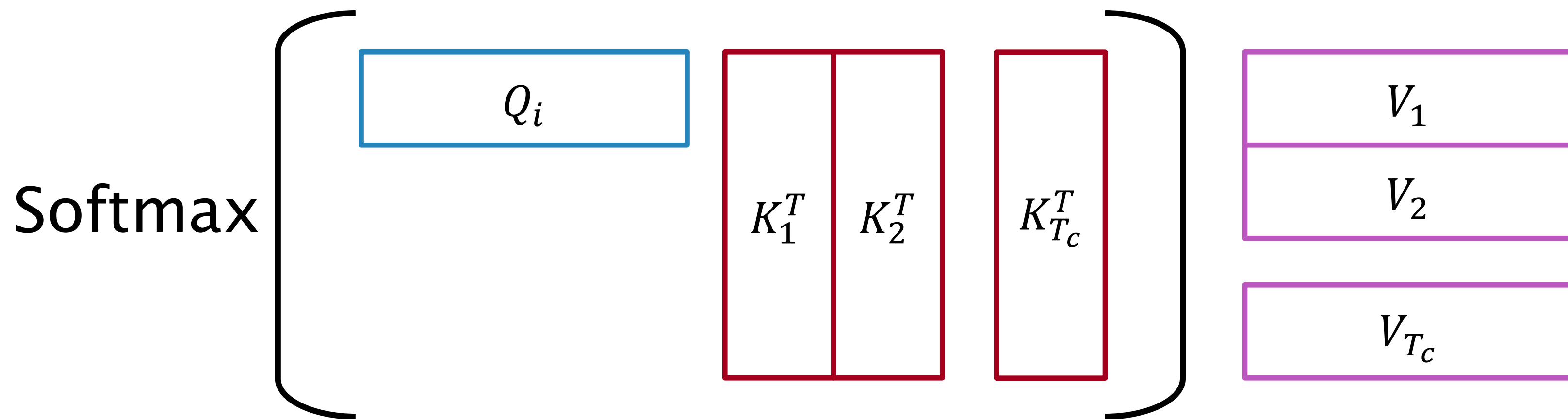
Computes exact QKV attention (no approximation) efficiently on a GPU.

- Tiling and Online Softmax – Perform blockwise computation to reduce I/O to HBM
- Fused Kernels – Implementation strategy to gain low-level memory control

Tiling

- Split the Q, K and V matrices in blocks of size b
- Perform attention computation block wise in SRAM





Let's ignore softmax for now..

$$O_i = \sum_{j=1}^{T_c} \left(Q_i \; K_j^T \right) V_j$$

for $j \in [1 \ldots T_c]$ do
  Load block K_j and V_j from HBM to SRAM
  for $i \in [1 \ldots T_r]$ do
    Load O_i and Q_i from HBM to SRAM
    Compute $A_{ij} = Q_i K_j^T$ in SRAM
    Update $O_i = O_i + A_{ij} V_j$ in SRAM
    Save O_i back to HBM
  end for
end for

Why is this efficient?

No more T^2 reads/writes to HBM

Why would such algorithm not work with softmax?

Softmax requires complete input

Online Softmax

- Input – $[a_1, a_2, .., a_{T_c}]$ (blocks of) unnormalized attention scores
- Output – $[p_1, p_2, \ldots, p_{T_c}]$ (blocks of) softmaxed attention scores
- Constraint – we receive a_j in a streaming fashion, i.e, global information is not available till the end.

Online Softmax – Iteratively builds to the final solution. Let at t-th iteration

- $m_t = \max(a_1, a_2, \ldots, a_t)$
- $l_t = \sum_{j=1}^{t} e^{(a_j - m_t)}$

Numerically stable version

$= softmax(a_1 - \mathrm{m}_{\mathrm{T_c}}, a_2 - \mathrm{m}_{\mathrm{T_c}}, \dots, a_{T_c} - \mathrm{m}_{\mathrm{T_c}}) P_1, p_2, \dots p_{T_c} = softmax(a_1, a_2, \dots, a_{T_c})$

Thus,

$$P_j = \frac{e^{a_j - m_{T_c}}}{l_{T_c}}$$

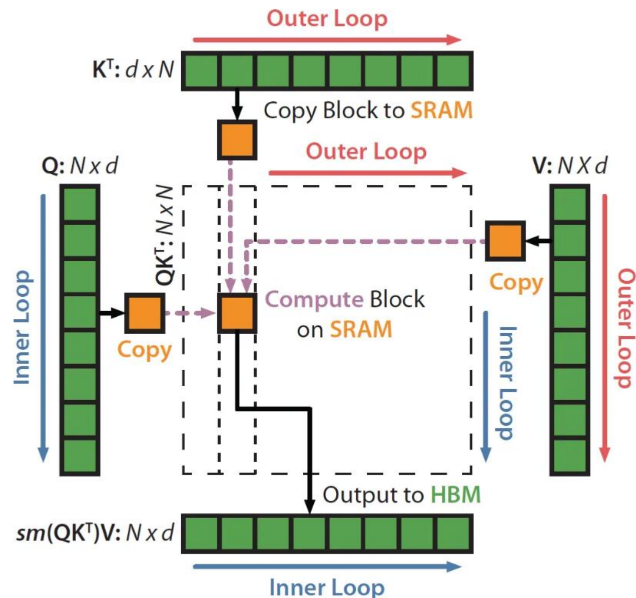
Now

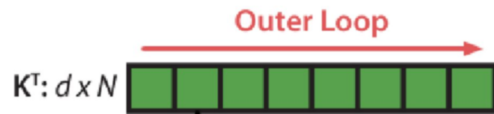
$$l_{T_c} = \sum_{j=1}^{\mathrm{T_c}} e^{a_j - m_{T_c}} = \sum_{j=1}^{\mathrm{T_c}-1} e^{a_j - m_{T_c}} + e^{a_{\mathrm{T_c}} - m_{T_c}} = \mathrm{e}^{\mathrm{m}_{\mathrm{T_c}-1} - m_{T_c}} \sum_{j=1}^{\mathrm{T_c}-1} e^{a_j - m_{T_c-1}} + e^{a_{\mathrm{T_c}} - m_{T_c}}$$

$$\boldsymbol{l_{T_c} = (e^{m_{T_c-1} - m_{T_c}}) l_{T_c-1} + e^{a_{T_c} - m_{T_c}}}$$

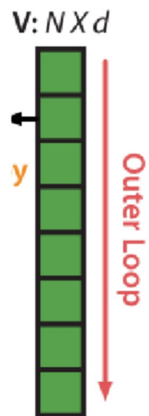
Further, $$p_j = \frac{e^{a_j - m_{T_c}}}{l_{T_c}} = (e^{m_{T_{c-1}} - m_{T_c}}) \frac{e^{a_j - m_{T_{c-1}}}}{l_{T_c}}$$

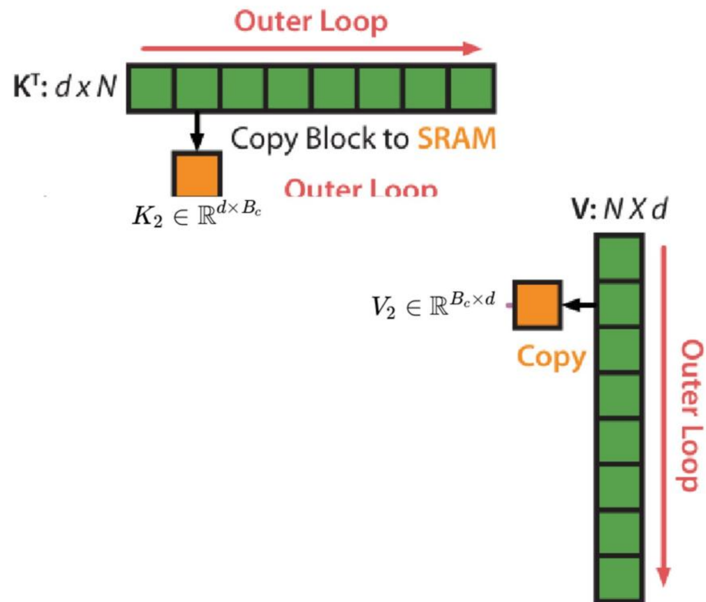
for $j \in [1 \ldots T_c]$ **do**
  Load block K_j and V_j from HBM to SRAM
  for $i \in [1 \ldots T_r]$ **do**
    Load O_i, Q_i, m_i, ℓ_i from HBM to SRAM
    Compute $A_{ij} = Q_i K_j^T$ in SRAM
    Compute new max: $\tilde{m}_{ij} = \max(m_i, \text{row_max}(A_{ij}))$
    Compute safe exponentials: $P_{ij} = \exp(A_{ij} - \tilde{m}_{ij})$
    Compute new sum: $\tilde{\ell}_{ij} = \exp(m_i - \tilde{m}_{ij})\ell_i + \text{row_sum}(P_{ij})$
    Update $O_i = \exp(m_i - \tilde{m}_{ij})O_i + P_{ij}V_j$ in SRAM
    Update trackers: $m_i = \tilde{m}_{ij}, \ \ell_i = \tilde{\ell}_{ij}$
    Save O_i, m_i, ℓ_i back to HBM
  end for
end for
for $i \in [1 \ldots T_r]$ **do**
  Final Normalization: $O_i = \text{diag}(\ell_i)^{-1} O_i$
end for





$$\text{for } 1 \leq j \leq T_c :$$

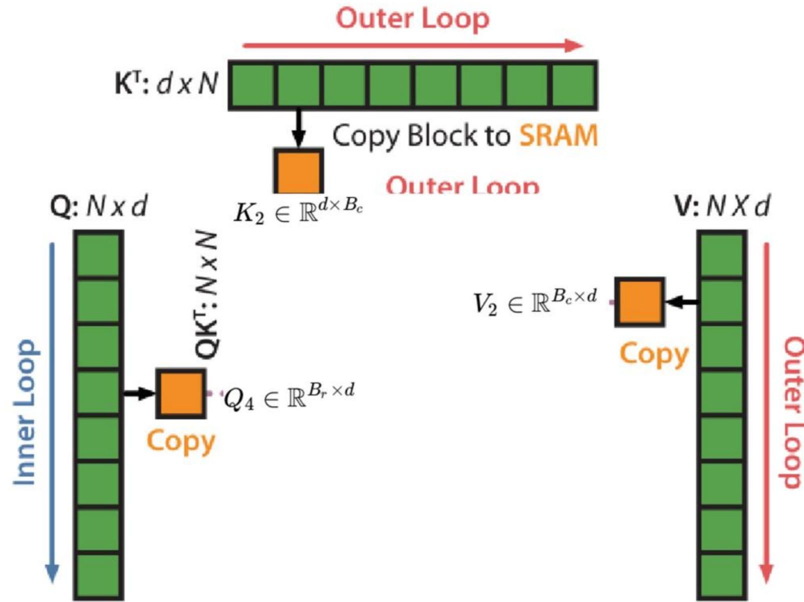




for $1 \leq j \leq T_c$:

Load, K_j, V_j from HBM to SRAM

for $1 \leq i \leq T_r$:



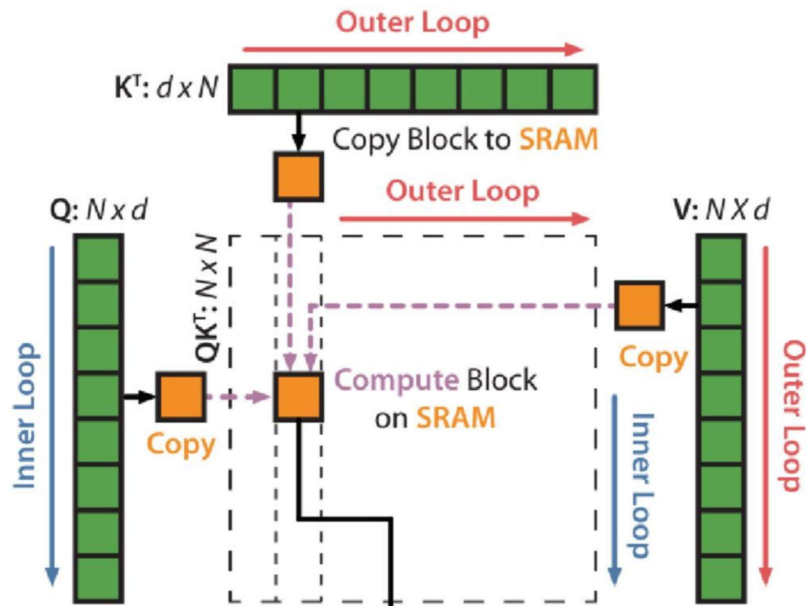
$S = QK^T$ $\quad P = Softmax(S)$ $\quad O = PV$

for $1 \leq j \leq T_c$:

Load, K_j, V_j from HBM to SRAM

for $1 \leq i \leq T_r$: #iterate over blocks of Q

Load, Q_i, O_i, l_i, m_i from HBM to SRAM



$S = QK^T \qquad P = Softmax(S) \qquad O = PV$

for $1 \leq j \leq T_c$:

Load, K_j, V_j from HBM to SRAM

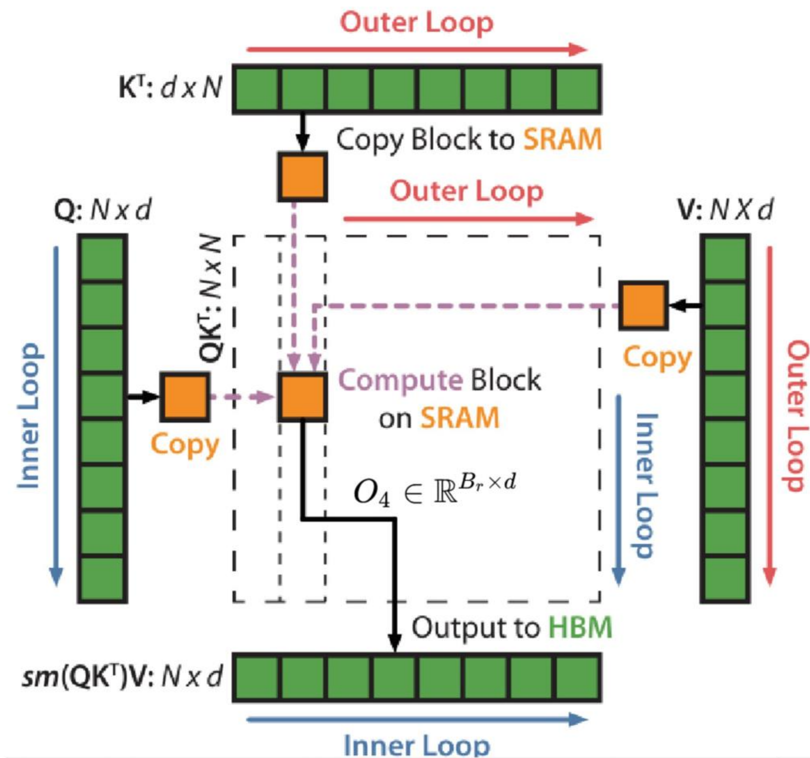
for $1 \leq i \leq T_r$: #iterate over blocks of Q

Load, Q_i, O_i, l_i, m_i from HBM to SRAM

on-chip, compute $S_{ij} = Q_i K_j^T \in \mathbb{R}^{B_r \times B_c}$

on-chip, compute block-wise statistics m_{ij}, l_{ij}, P_{ij}

on-chip, compute new values for m_i, l_i



$S = QK^T \qquad P = Softmax(S) \qquad O = PV$

for $1 \leq j \leq T_c$:

Load, K_j, V_j from HBM to SRAM

for $1 \leq i \leq T_r$: #iterate over blocks of Q

Load, Q_i, O_i, l_i, m_i from HBM to SRAM

on-chip, compute $S_{ij} = Q_i K_j^T \in \mathbb{R}^{B_r \times B_c}$

on-chip, compute block-wise statistics m_{ij}, l_{ij}, P_{ij}

on-chip, compute new values for m_i, l_i

Write, $O_i \leftarrow f(O_i, P_{ij}, V_j, m_i, l_i)$ to HBM

Write, $m_i \leftarrow m_i, l_i \leftarrow l_i$, to HBM

end for

end for

Return O

Fused Kernels

- To implement FlashAttention, we need to decide the data movement between SRAM and HBM.
- Operations such as online-softmax, do not need to store full intermediate attention scores, i.e., they are **fused** into attention computation
- High-level libraries such as PyTorch do not allow fine-grained GPU memory control
- Thus, FlashAttention is implemented as low-level CUDA Fused Kernel.

Results

Table 18: Forward pass runtime (ms) of various exact/approximate/sparse attention mechanisms by sequence length. Best in **bold**, second best underlined.

Attention Method	128	256	512	1024	2048	4096	8192	16384	32768	65536
PyTorch Attention	0.21	0.22	0.43	1.27	4.32	16.47	67.77	-	-	-
Megatron	0.24	0.26	0.42	1.33	4.28	-	-	-	-	-
Reformer	1.77	2.82	5.01	9.74	20.03	41.11	87.39	192.40	-	-
Local Attention	0.48	0.57	0.80	1.90	5.76	11.56	23.13	46.65	94.74	-
Linformer	0.46	0.36	0.45	**0.50**	1.09	2.09	4.01	7.90	15.70	35.40
Smyrf	1.94	1.96	3.01	5.69	11.26	22.23	44.21	88.22	-	-
LSformer	1.21	1.34	1.34	3.31	11.01	21.71	43.27	86.32	172.85	-
Block Sparse	0.96	1.04	1.66	2.16	5.41	16.15	-	-	-	-
Longformer	0.99	0.98	0.99	1.56	4.79	11.07	32.98	-	-	-
BigBird	0.96	1.02	1.02	1.48	5.05	11.59	34.16	-	-	-
FlashAttention	**0.08**	**0.09**	**0.18**	0.68	2.40	8.42	33.54	134.03	535.95	2147.05
Block-Sparse FlashAttention	0.56	0.52	0.63	0.65	**0.61**	**0.96**	**1.69**	**3.02**	**5.69**	**11.77**

Question: Why would FlashAttention discussed so far fail on training?

Recomputation

- FlashAttention does not realize the attention scores
- But, attention scores are needed during backpropagation.
- Solution: Recompute them when needed.

Attention	Standard	FLASHATTENTION
GFLOPs	66.6	75.2
HBM R/W (GB)	40.3	4.4
Runtime (ms)	41.7	7.3

Results – Training

Table 20: Forward pass + backward pass runtime (ms) of various exact/approximate/sparse attention mechanisms by sequence length. Best in **bold**, second best underlined.

Attention Method	128	256	512	1024	2048	4096	8192	16384	32768	65536
PyTorch Attention	0.67	0.70	1.18	3.67	13.22	50.44	-	-	-	-
Megatron	0.74	0.65	1.23	3.80	13.21	-	-	-	-	-
Reformer	3.93	7.01	13.15	25.89	52.09	105.00	215.13	-	-	-
Local Attention	1.09	1.27	1.99	5.38	18.32	36.77	73.67	147.29	296.35	-
Linformer	1.31	1.25	1.30	1.29	3.20	6.10	11.93	23.39	46.72	100.52
Smyrf	2.98	4.23	7.78	15.12	29.96	59.45	118.60	237.02	-	-
LSformer	3.03	3.05	4.26	10.70	30.77	60.15	118.33	234.94	-	-
Block Sparse	2.39	2.40	3.31	5.02	12.25	35.94	-	-	-	-
Longformer	2.36	2.34	2.38	2.94	9.83	21.35	58.12	-	-	-
BigBird	2.35	2.35	2.37	3.25	10.36	22.57	60.63	-	-	-
FlashAttention	**0.31**	**0.31**	**0.73**	2.29	7.64	30.09	118.50	470.51	1876.08	7492.85
Block-Sparse FlashAttention	0.74	0.77	0.82	**0.88**	**1.71**	**3.21**	**6.56**	**12.60**	**24.93**	**50.39**

Supporting Longer Contexts

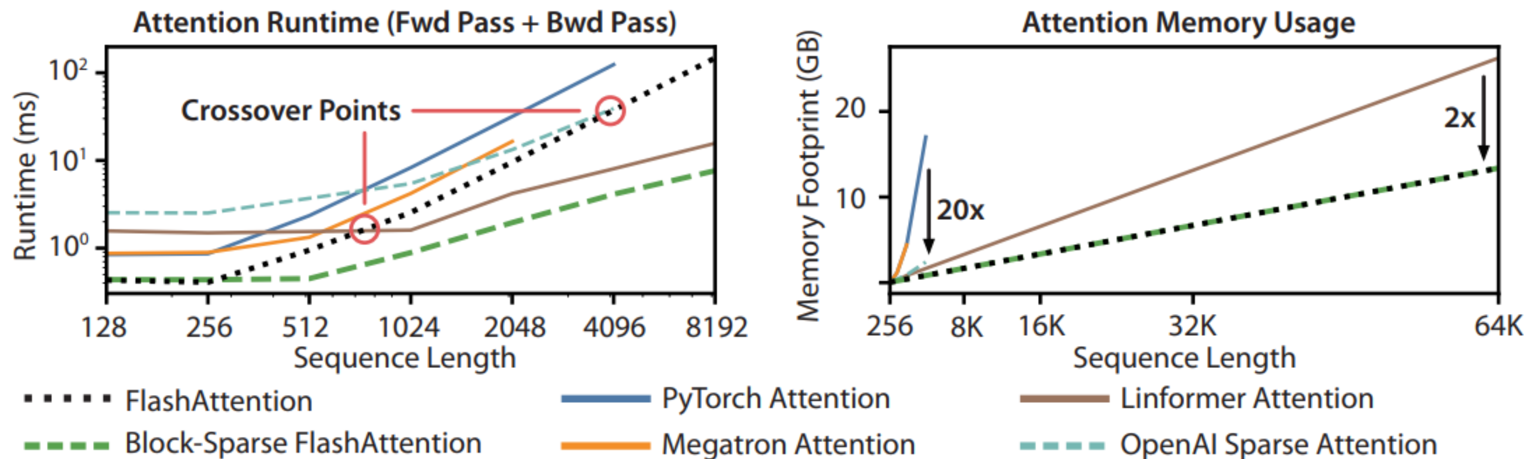


Figure 3: **Left:** runtime of forward pass + backward pass. **Right:** attention memory usage.

Impact

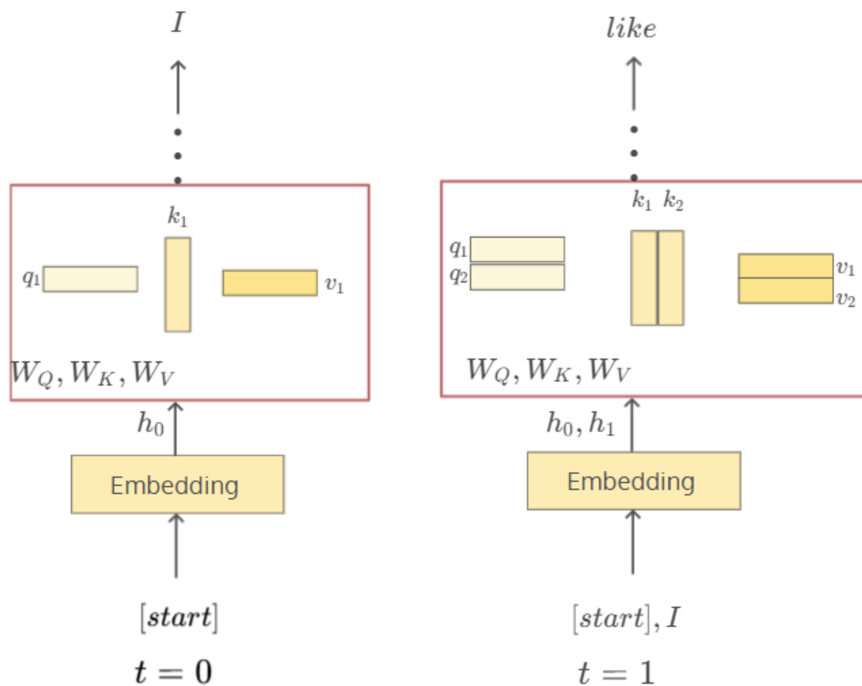
FlashAttention highlights

- Exact computation
- Faster run times
- Lower memory usage

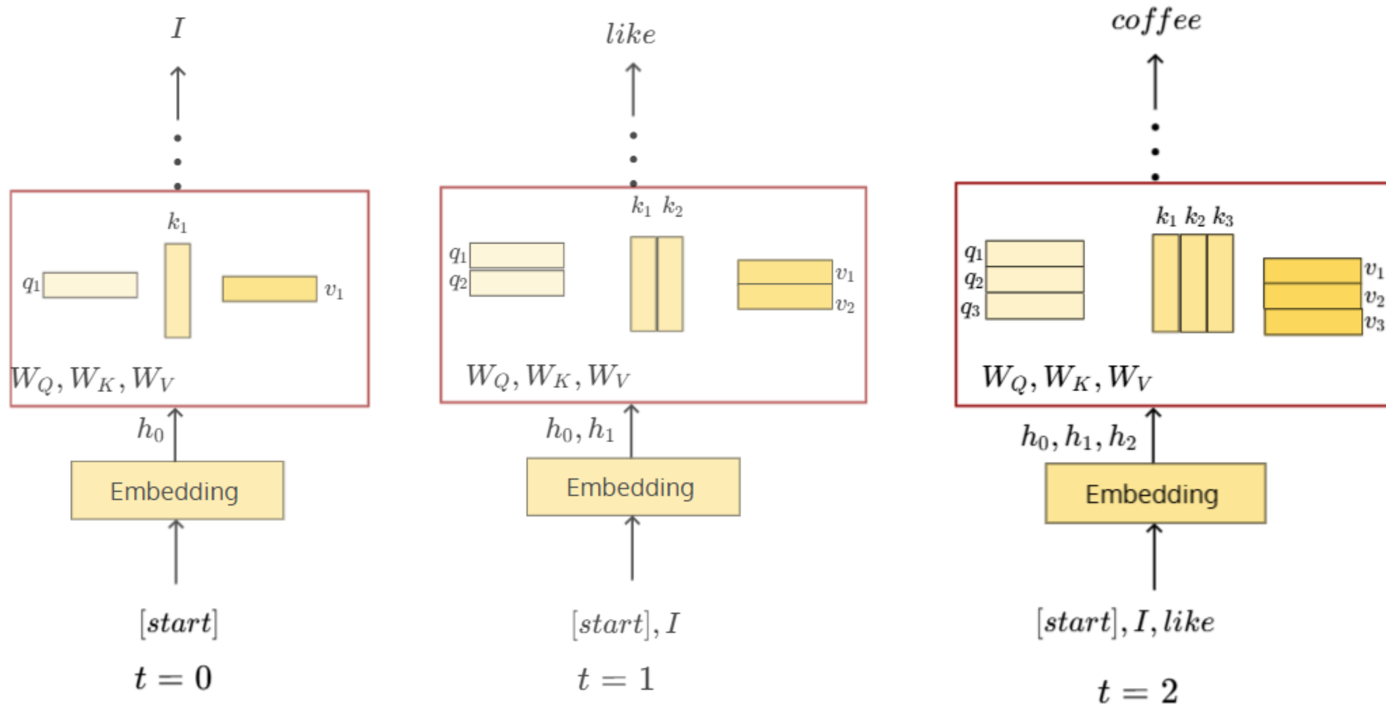
- Train LLMs over longer context and/or more data faster under same resource budget → high quality models
- Fast generation from LLMs → large scale deployments
- New modalities → Develop models for high-res images, audio, video, MRI, time series and so on.

Fast Attention during Inference Time

Naïve Implementation of Autoregressive Inference



Naïve Implementation of Autoregressive Inference

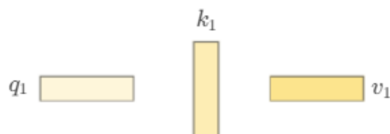


Can you spot inefficiency?

- Compute all keys and values from previous queries as well!

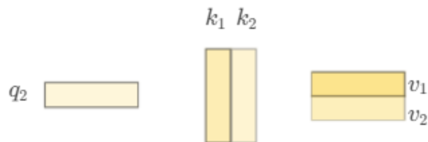
KV Caching

In the first time step, we start with a query, compute its key-value pair for the given input token when running the model in autoregressive mode.

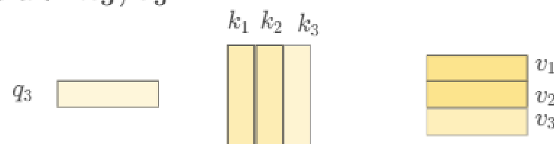


Store the key value in cache (usually GPU RAM which has HBM).

For the new query q_2 in time step 2, we reload k_1, v_1 from HBM and compute k_2, v_2



For the new query q_3 in time step 3, we reload keys and values k_1, v_1, k_2, v_2 from previous times steps from HBM and compute k_3, v_3

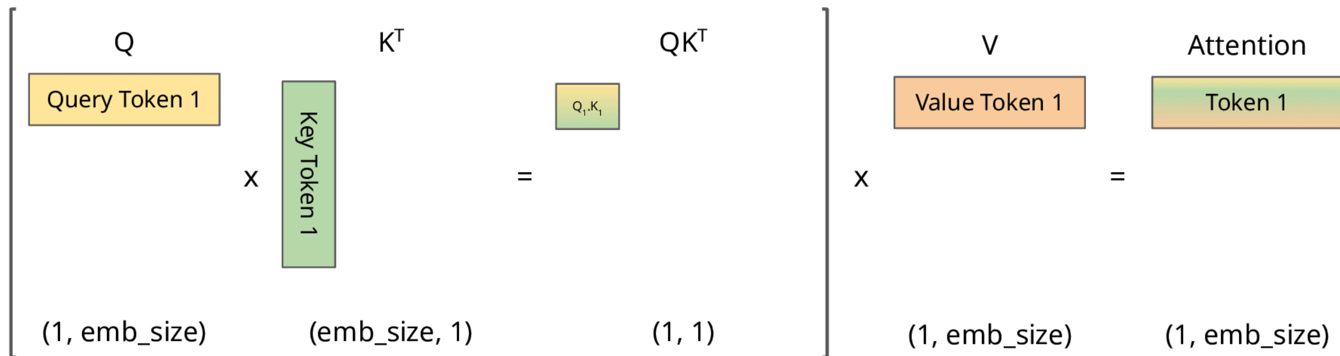


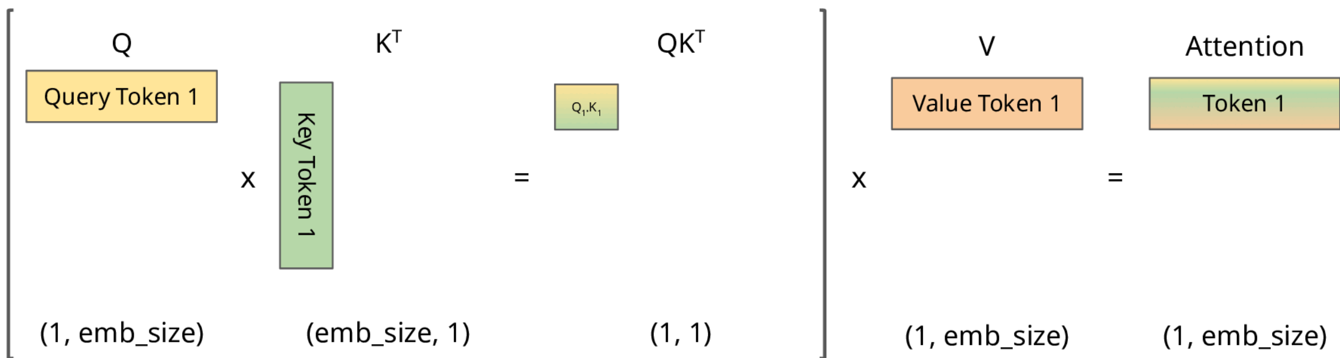
In this way, we trade off the memory for compute and the compute scales linearly $O(n)$

Since KV-caching uses GPU RAM where the model weights are stored, we cannot let the memory for KV-cache to grow indefinitely!

Let 's take an example.

Step 1





Values that will be masked   Values that will be taken from cache

https://medium.com/@joaolages/kv-caching-explained-276520203249

Example

KV cache per token (in B) =

$2 \times$ num_layers $\times$ (num_heads $\times$ dim_head)

$\times$ precision_in_bytes

Model GPT-3

- num_layers = 96
- num_heads = 96
- dim_head = 128
- precision_in_bytes=4

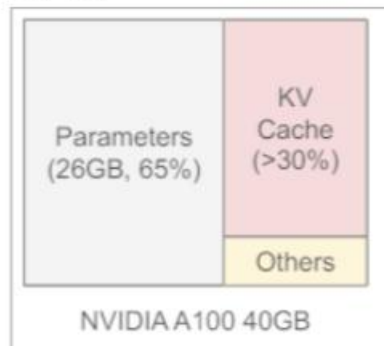
KV cache per token = $2 \times 96 \times (96 \times 128) \times 4$

= 9.4 MB

For 2K context length = 2048×9.4 MB

= 19.3 GB

to generate a sequence of length 2K requires 19.3 GB of memory for KV caching alone (we can't even use an A100 40 GB GPU).



A practical solution is to drop the keys and values of tokens that occured in the distant past (act similar to local windowed attention).

The other approach is to introduce model level modifications!

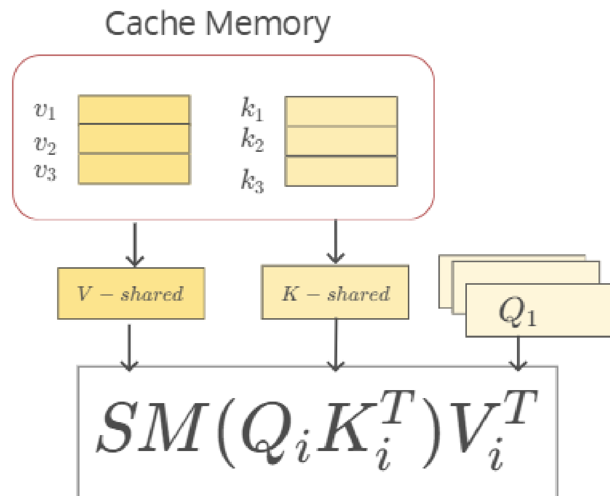
Multi-Query Attention

Select one hyper-parameter out of three hyperparameters that we could exploit in the formula to reduce the memory requirement is ____

KV cache per token (in B) =
2 × num_layers × (num_heads × dim_head) × precision_in_bytes

Lets set num_heads = 1. I.e., we want to access key-value pairs for all heads in const. time. How?

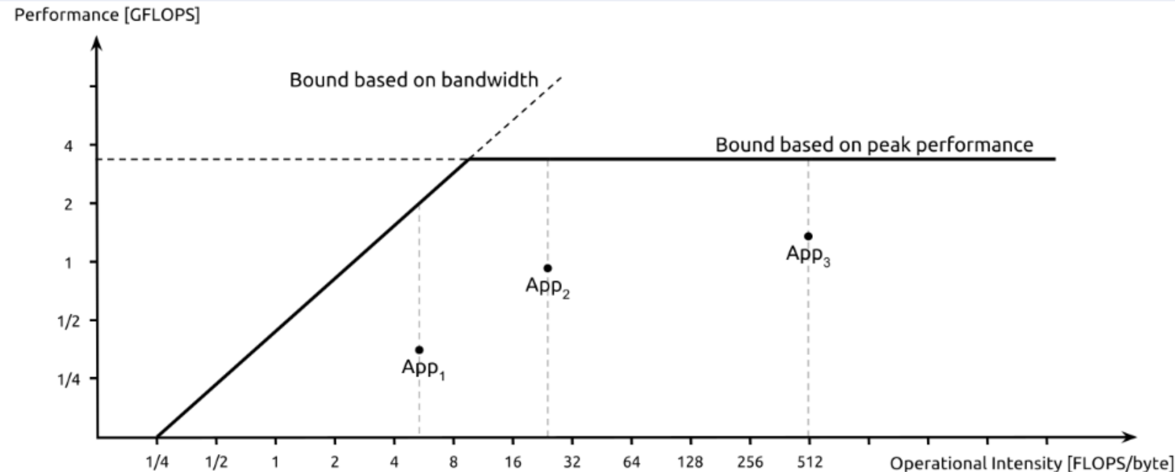
Share all the key-value pairs across the heads.



This greatly reduces the cache memory. However, the performance degrades significantly.

Moreover, this requires us to re-train the model with at least 5% of training data

Memory Bound vs. Compute Bound Jobs



Roofline Model (https://en.wikipedia.org/wiki/Roofline_model)

Computations are memory-bandwidth bound until they reach a high enough arithmetic intensity (determined by the processor's ratio of compute to memory capability). Beyond this point, they become **compute-bound**, and the full arithmetic performance of the processor is achieved.

MQA and Arithmetic Intensity

- Arithmetic Intensity = #FLOPS/#Memory Accesses

The effect of multi-query attention on arithmetic intensity is two-fold:

1. MQA reduces the number of bytes read from memory per arithmetic operation in the attention computation, thus increasing arithmetic intensity. This leads to a faster and more efficient attention computation.

2. MQA also reduces the number of bytes that must be stored in memory for KV-cache values. This extra space allows us to **increase the batch size** (N in the above pseudocode), which has a similar effect of increasing arithmetic intensity for the program.

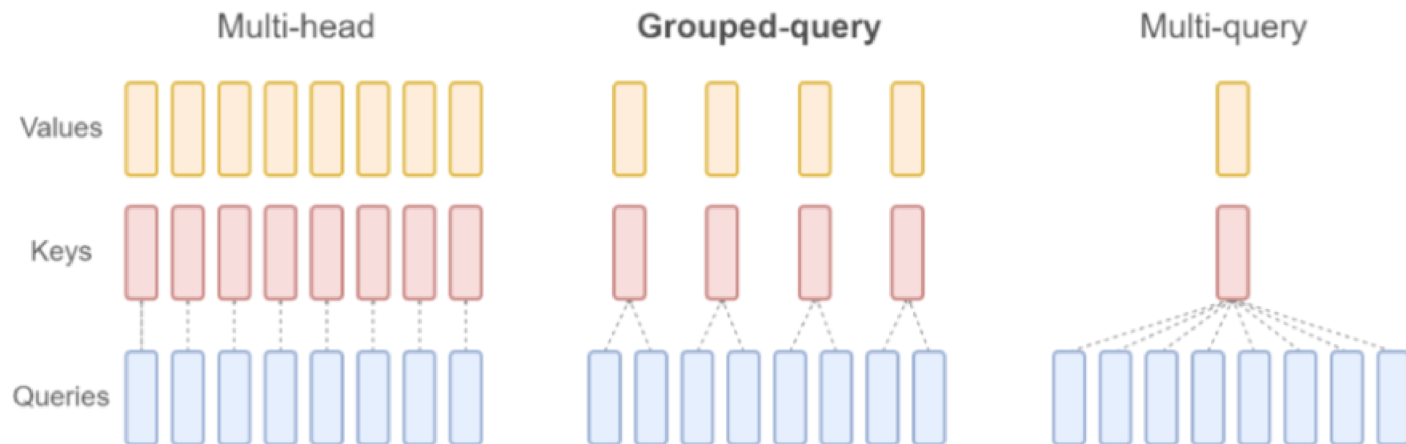
https://fireworks.ai/blog/multi-query-attention-is-all-you-need

Grouped Query Attention (GQA)

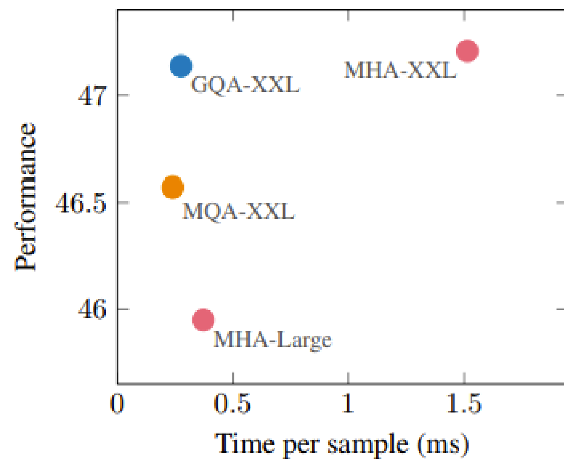
On one extreme we have Multi-Head Attention where each head has a separate query, key and value vector.

On the other extreme we have Multi-Query Attention where each head has separate query but keys and values are shared across all heads.

Grouped Query Attention is just a middle ground between the two, where a group of queries share key-value pairs that result in performance close to MHA



Results

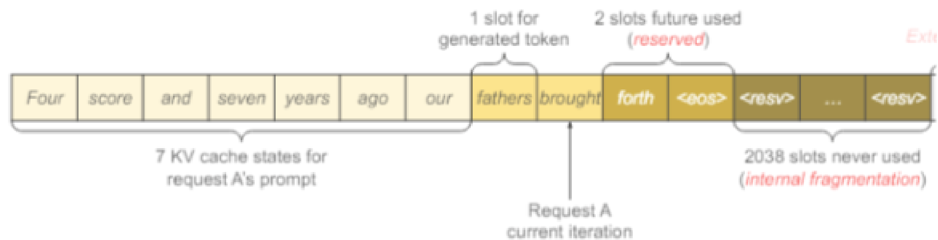


Paged Attention

Typically, fixed and contiguous memory is reserved for KV-caching (say, based on context length of the model).

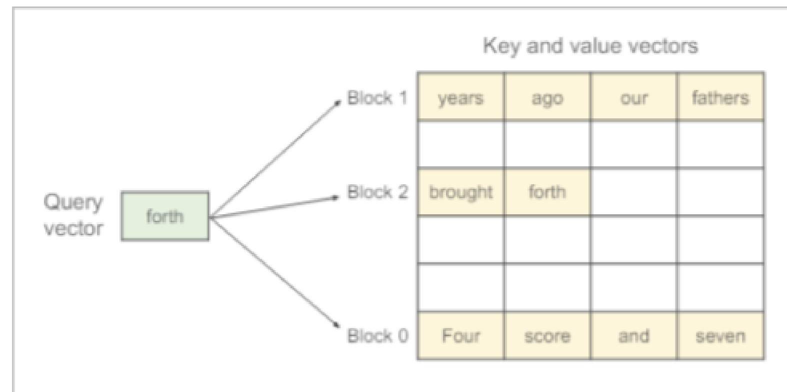
However, we do not know the length of the sequence that the model generates a priori.

The memory during the generation process is wasted if the model generated a sequence that is much smaller than the pre-fixed length.



A solution is to use a block of non-contiguous memory locations.

That is what Paged Attention does



A Look into Attention Formulation

- Input: $X\ (T, d)$

Observation: Caching options are tied to attention formulation

What makes this formula so special?
- It scales.

But, is this the best possible formulation?
- Not necessarily

Choices for Key caching
- $X\ (\mathrm{T}, \mathrm{d})$
- $XW_K^h\ (\mathrm{T}, \mathrm{d_h})\mathrm{n_h}$

Choices for Value caching
- $X\ (\mathrm{T}, \mathrm{d})$
- $XW_\mathrm{V}^h\ (\mathrm{T}, \mathrm{d_h})\mathrm{n_h}$

Changing Attention Formulation

$$O_h = softmax\left(\left(XW_Q^h\right)\left(X(W_C W_{DK})W_K^h\right)^T\right)\left(XW_V^h\right)$$

Where $W_C(d, d_c)$ and $W_{DK}(d_c, d)$ are head independent matrices with $(d_c < d)$

Choices for Key Caching:

- X (T, d)
- XW_c (T, d_c)
- XW_cW_{DK} (T, d)
- $XW_cW_{DK}W_K^h$ $(T, d_h) * n_h$

Caching Values

$$O = softmax\left(\left(XW_Q^h\right)\left(\boldsymbol{X}(\boldsymbol{W_C}\boldsymbol{W_{D\mathrm{K}}})\boldsymbol{W_K^h}\right)^T\right)\left(\boldsymbol{X}(\boldsymbol{W_C}\boldsymbol{W_{DV}})\boldsymbol{W_V^h}\right)$$

DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model

DeepSeek-AI

research@deepseek.com

Multihead Latent Attention (MLA) Inference

At each layer,

- Compute compressed vectors $Z = XW_C$ and cache them
- Compute keys $\mathrm{K_h} = ZW_{DK}W_K^h$ and values $\mathrm{V_h} = ZW_{D\mathrm{V}}W_{\mathrm{V}}^h$ for head h on the fly.
- Perform attention as usual.

Question: Why is it okay to perform more computation to reduce cache operations?

- As in FlashAttention, memory/data movement is the bottleneck.

Matrix Absorption

$$\left(XW_Q^h\right)\left(X(W_C W_{DK})W_K^h\right)^T$$
$$= X\left(W_Q^h W_K^{hT} W_{DK}^T\right)Z^T$$
$$= XW_{QDK}^h Z^T$$

Use W_{QDK}^h $(\mathrm{d}, \mathrm{d_c})$ during inference

Matrix Absorption - Values

$$\mathrm{O_h} = softmax\left(\left(XW_Q^h\right)\left(X(W_C W_{DK})W_K^h\right)^T\right)\left(X(W_C W_{DV})W_V^h\right)$$

$\mathrm{n_h}$

DeepSeek V2 uses compression for Query as well.

$$= \sum_{\mathrm{h=1}} \mathrm{A}\left(X(W_C W_{DV})W_V^h\right)\mathrm{W_o^h} = \sum_{\mathrm{h=1}} \mathrm{AZ}W_{DV}W_V^h\mathrm{W_o^h}$$

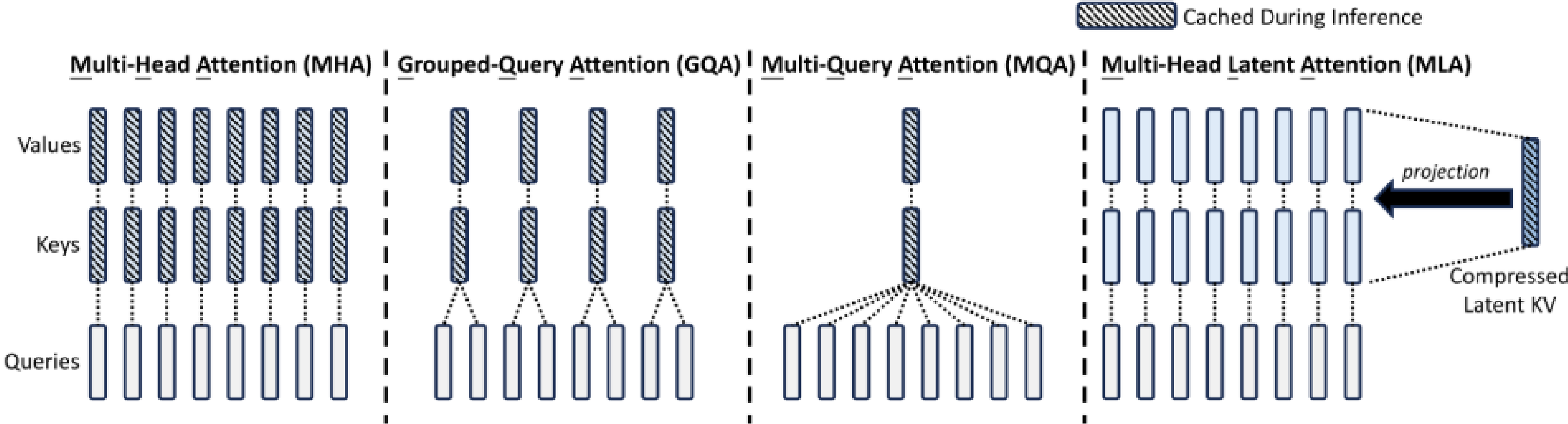
$$= \sum_{\mathrm{h=1}}^{\mathrm{n_h}} \mathrm{AZW_{ODV}^h}$$

Use $\mathrm{W_{ODV}^h}$ at inference

Results

Benchmark (Metric)	# Shots	Small MoE w/ MHA	Small MoE w/ MLA	Large MoE w/ MHA	Large MoE w/ MLA
# Activated Params	-	2.5B	2.4B	25.0B	21.5B
# Total Params	-	15.8B	15.7B	250.8B	247.4B
KV Cache per Token (# Element)	-	110.6K	15.6K	860.2K	34.6K
BBH (EM)	3-shot	37.9	**39.0**	46.6	**50.7**
MMLU (Acc.)	5-shot	48.7	**50.0**	57.5	**59.0**
C-Eval (Acc.)	5-shot	**51.6**	50.9	57.9	**59.2**
CMMLU (Acc.)	5-shot	52.3	**53.4**	60.7	**62.5**

Multi-Head Latent Attention

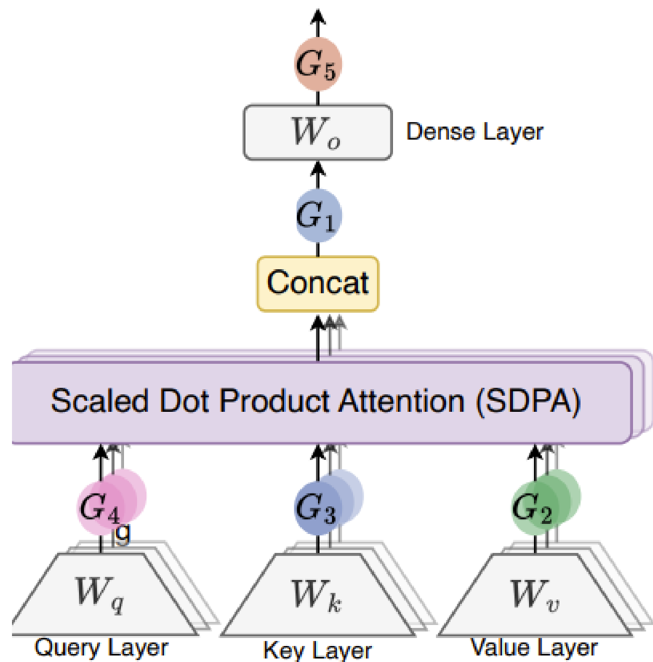


Different KV caching techniques.

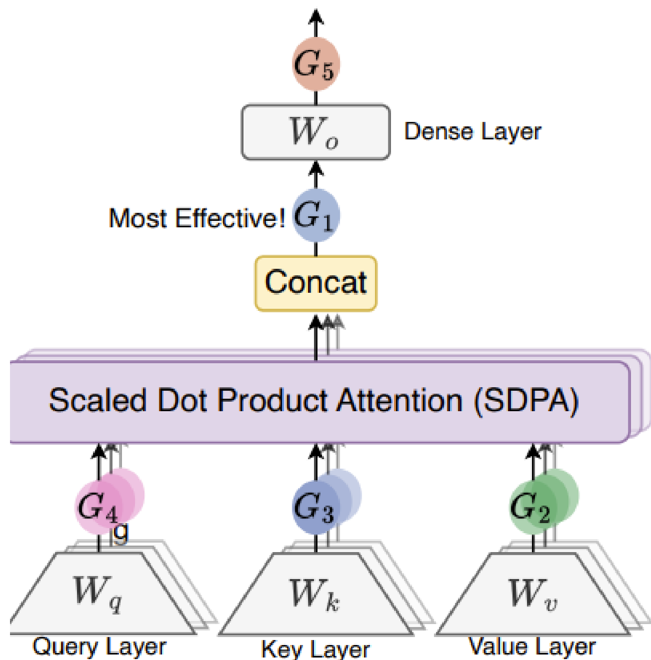
https://liorsinai.github.io/machine-learning/2025/02/22/mla.html

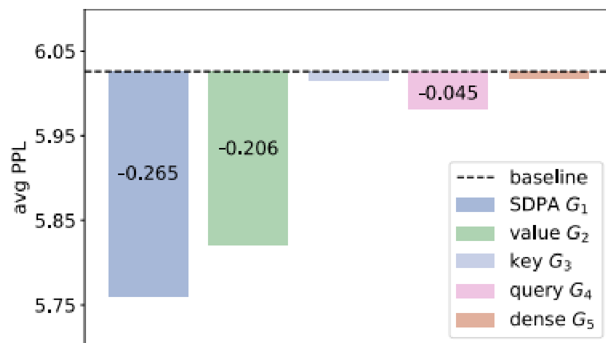
Gated Attention

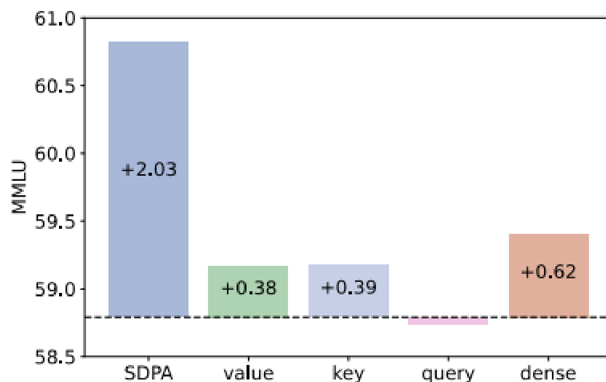
Gated Attention

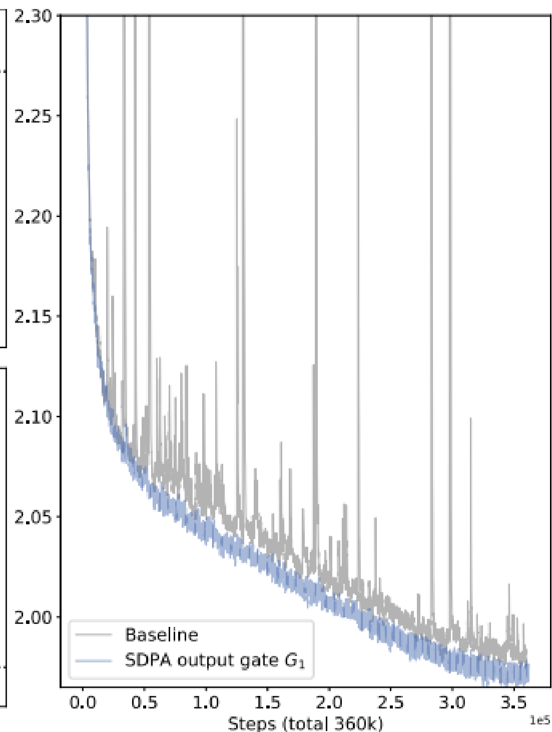


Gated Attention









Gating Mechanism

$$Y' = g(Y, X, W_\theta, \sigma) = Y \odot \sigma(XW_\theta)$$

- Granularity:
  - Headwise: A single scalar gating score modulates the entire output of an attention head.
  - Elementwise: Gating scores are vectors with the same dimensionality as Y, enabling fine-grained, per-dimension modulation.
- Head Specific or Shared.
  - Head-Specific: each attention head has its specific gating scores, enabling independent modulation for each head.
  - Head-Shared: W_θ and gating scores are shared across heads.

Intuition

- Expressiveness Argument

$$o_i^k = \left(\sum_{j=0}^{i} S_{ij}^k \cdot X_j W_V^k\right) W_O^k = \sum_{j=0}^{i} S_{ij}^k \cdot X_j \left(W_V^k W_O^k\right)$$

$$o_i^k = \left(\sum_{j=0}^{i} S_{ij}^k \cdot \text{Non-Linearity-Map}(X_j W_V^k)\right) W_O^k$$

- Attention Sink Argument

Mitigates Attention Sink

- Conclusions:
  - enhances non-linearity
  - introduces input-dependent sparsity
  - eliminates inefficiencies like the ‘attention sink’ phenomenon