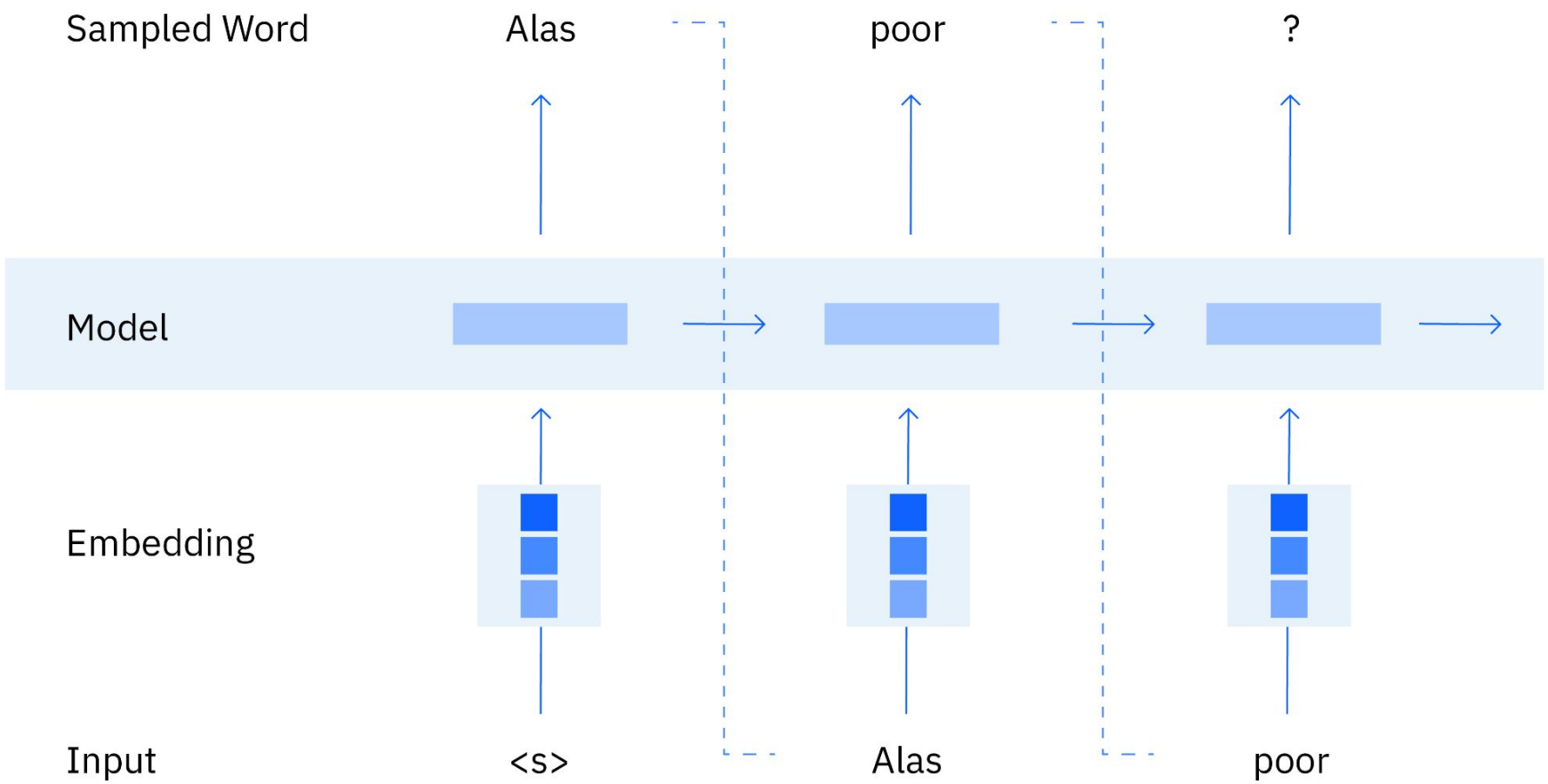
Flex Attention

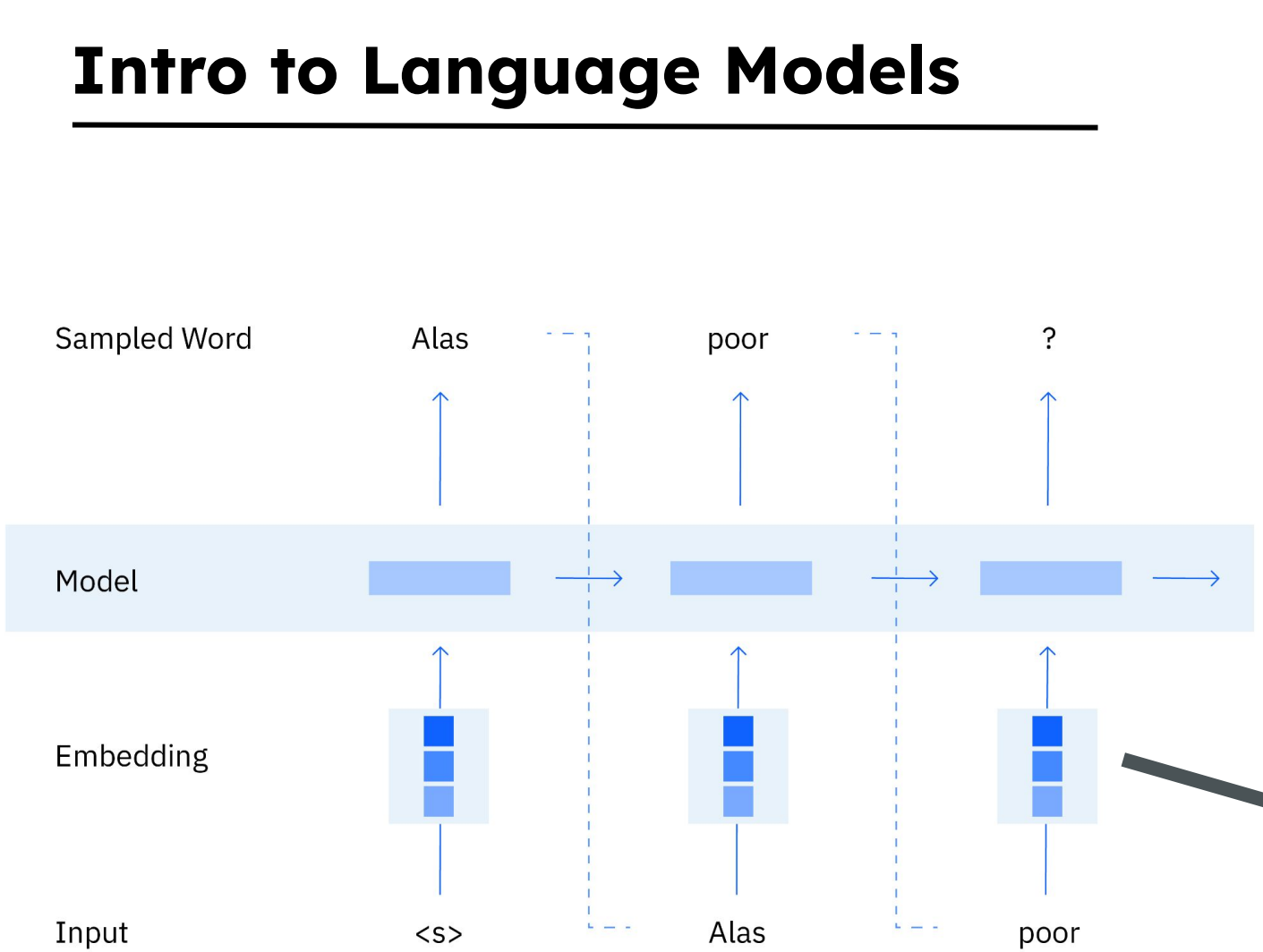
Automatic Optimised Attention Kernels

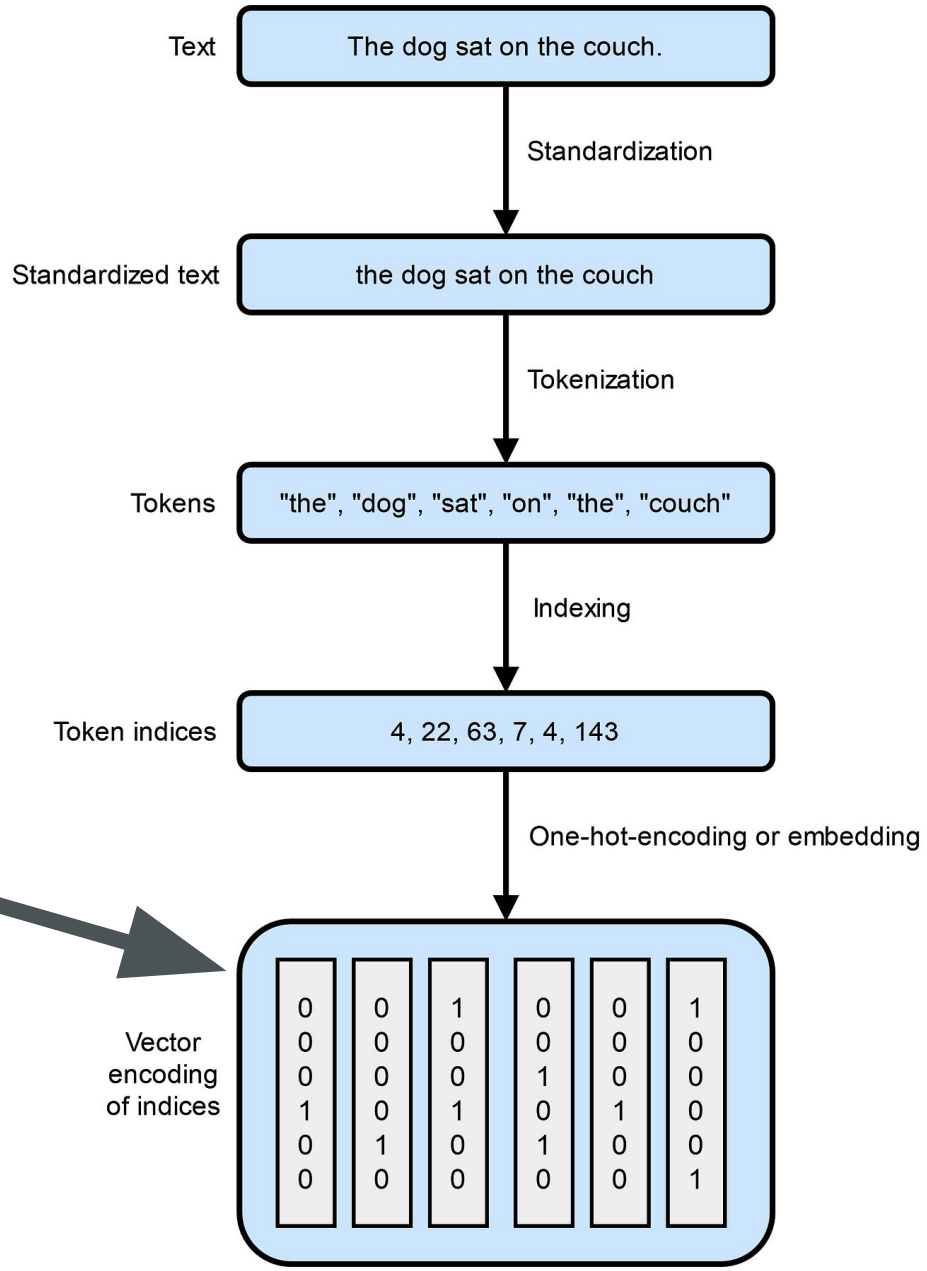
By: Bhavye, Satyam, Zain, Charlie, Kushal

Intro to Language Models



Intro to Language Models





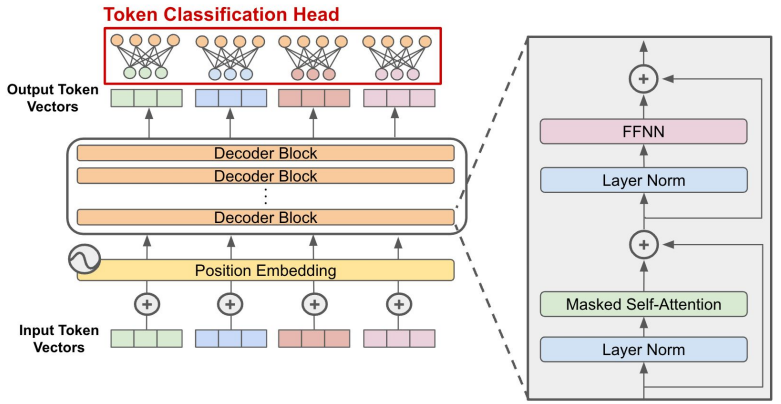
Current Day Language Models

Transformer Architecture

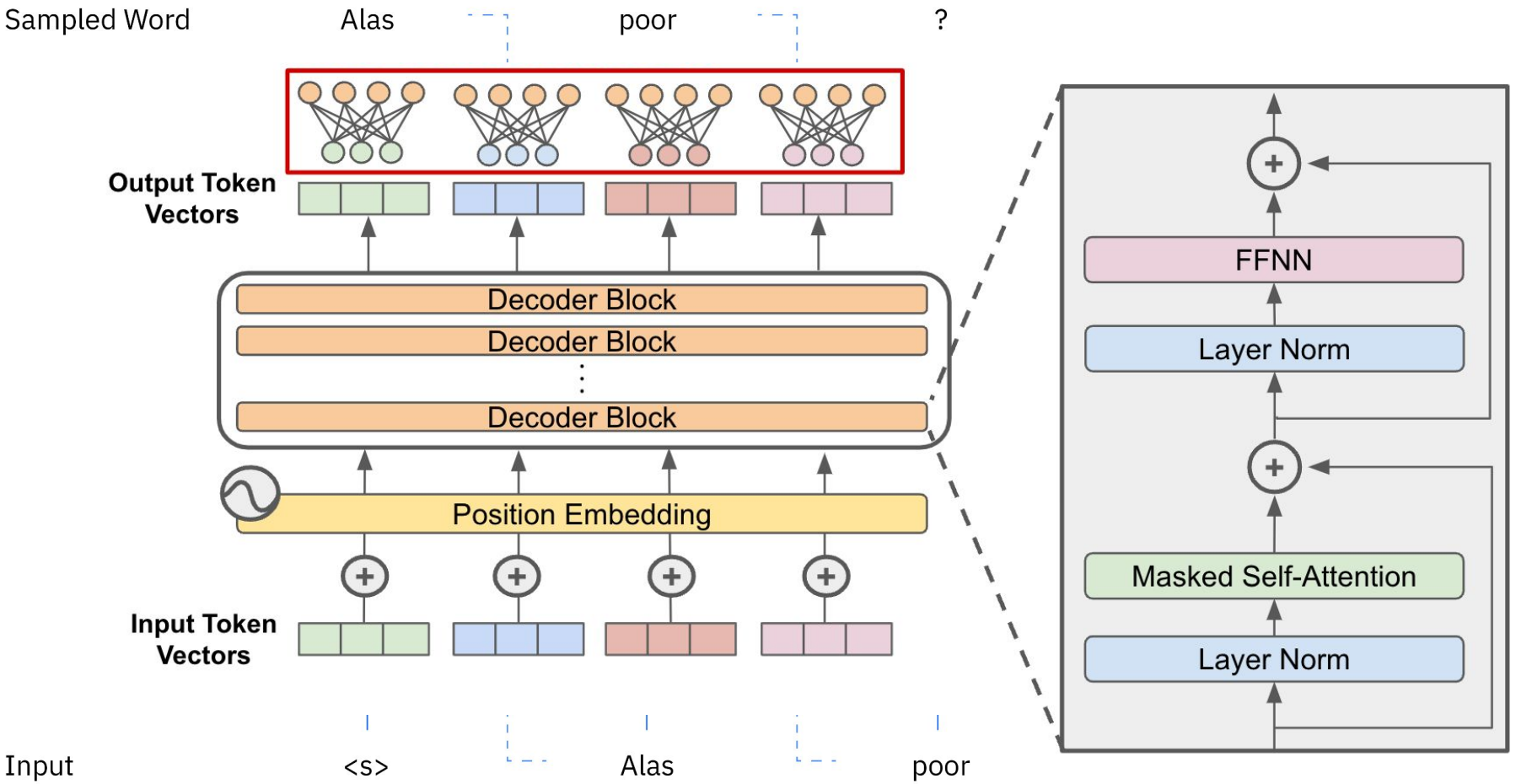
Current Day Language Models

Gemini

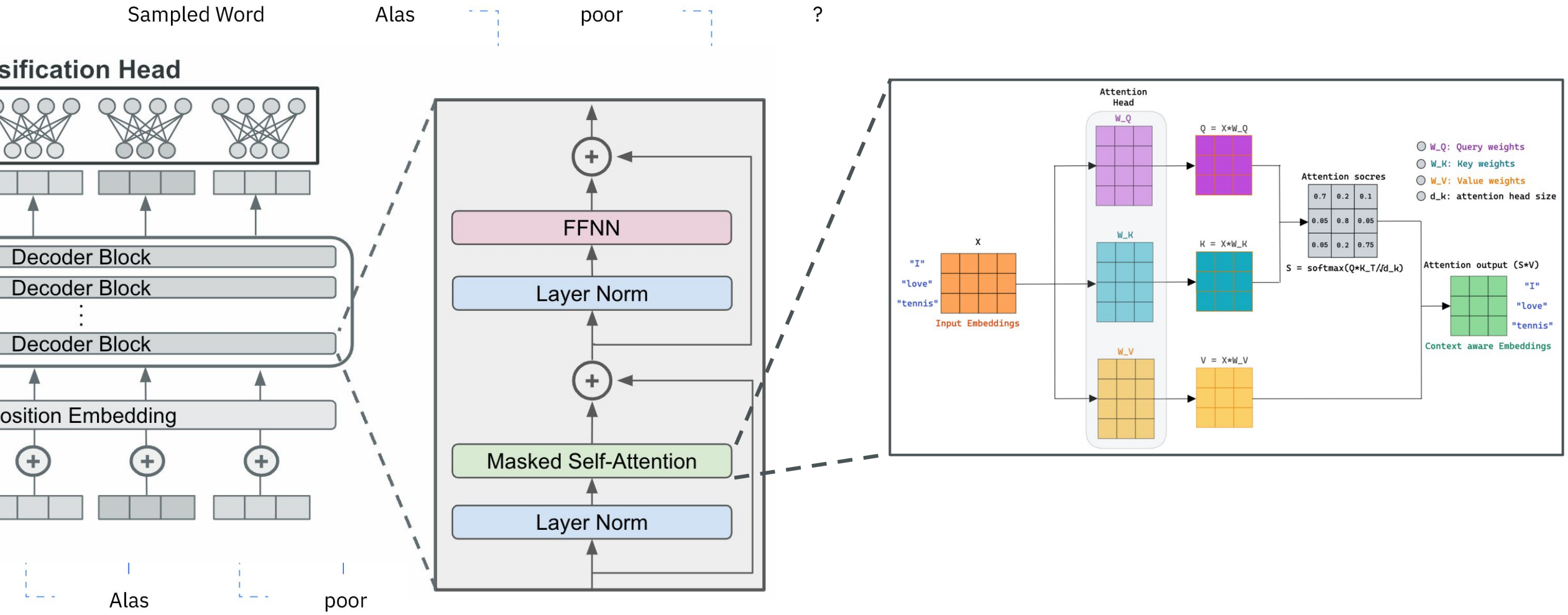
Claude



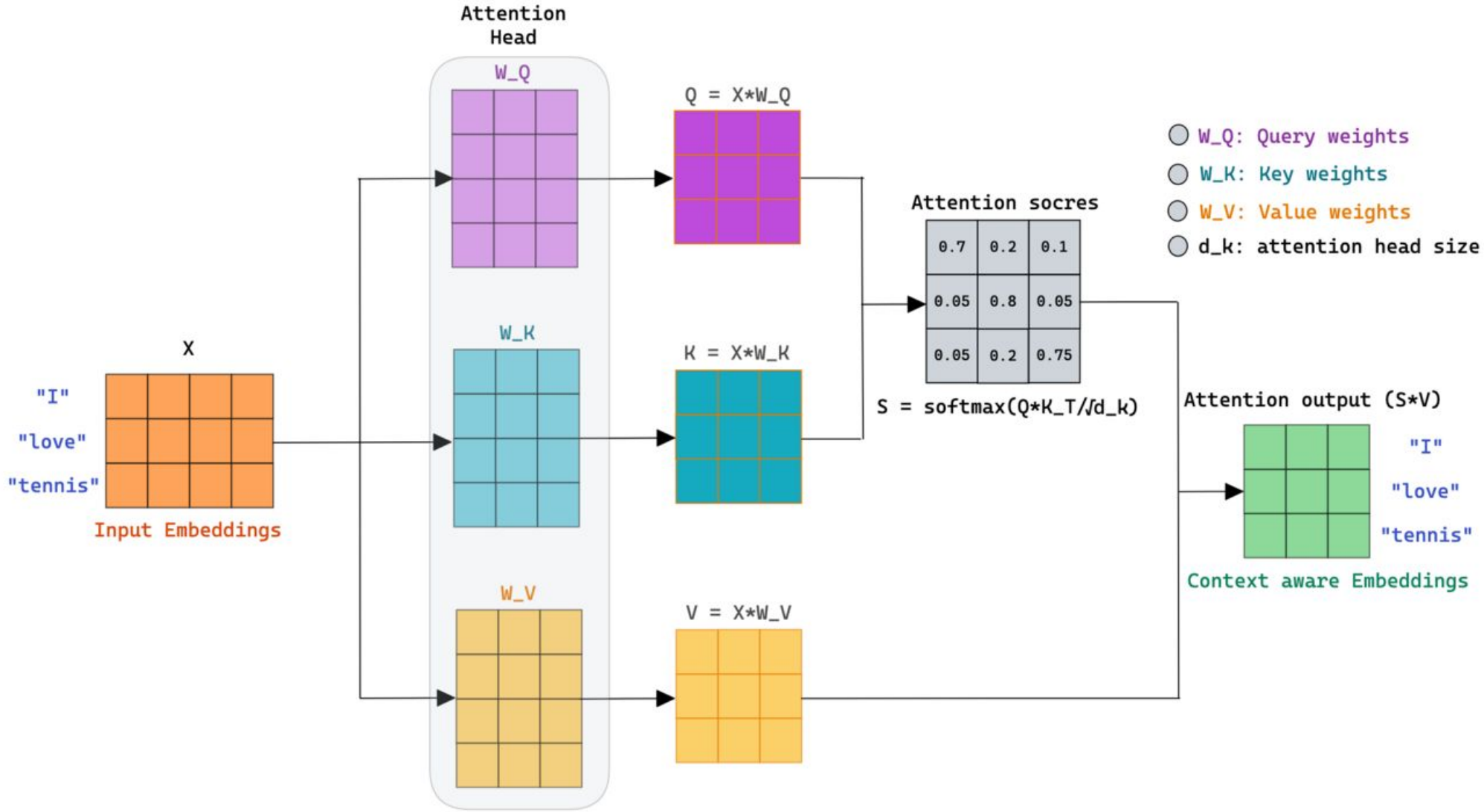
Transformer Block



Transformer Block

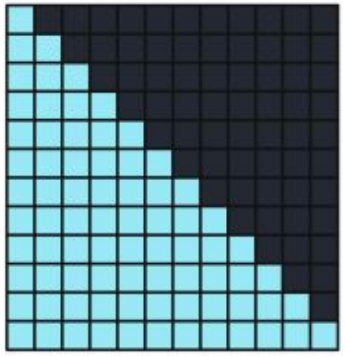


Attention

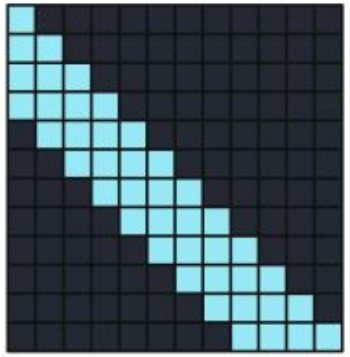

Attention Head
W_Q
Q = X*W_Q
W_K
K = X*W_K
W_V
V = X*W_V
X
"I"
"love"
"tennis"
Input Embeddings
Attention socres
0.7 0.2 0.1
0.05 0.8 0.05
0.05 0.2 0.75
S = softmax(Q*K_T/√d_k)
W_Q: Query weights
W_K: Key weights
W_V: Value weights
d_k: attention head size
Attention output (S*V)
"I"
"love"
"tennis"
Context aware Embeddings

Attention Variants

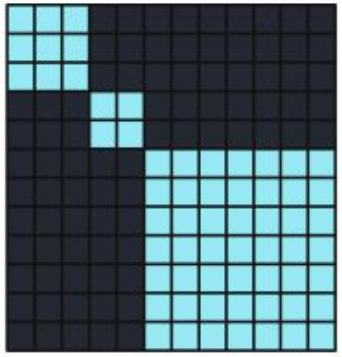
- Masks are applied to the attention matrix.



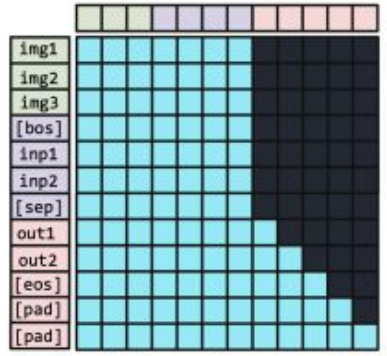
```
def causal_mask(b, h, q, kv):
    return q >= kv
```



```
def sliding_window_mask(b, h, q, kv):
    return q <= kv + window and q >= kv
```



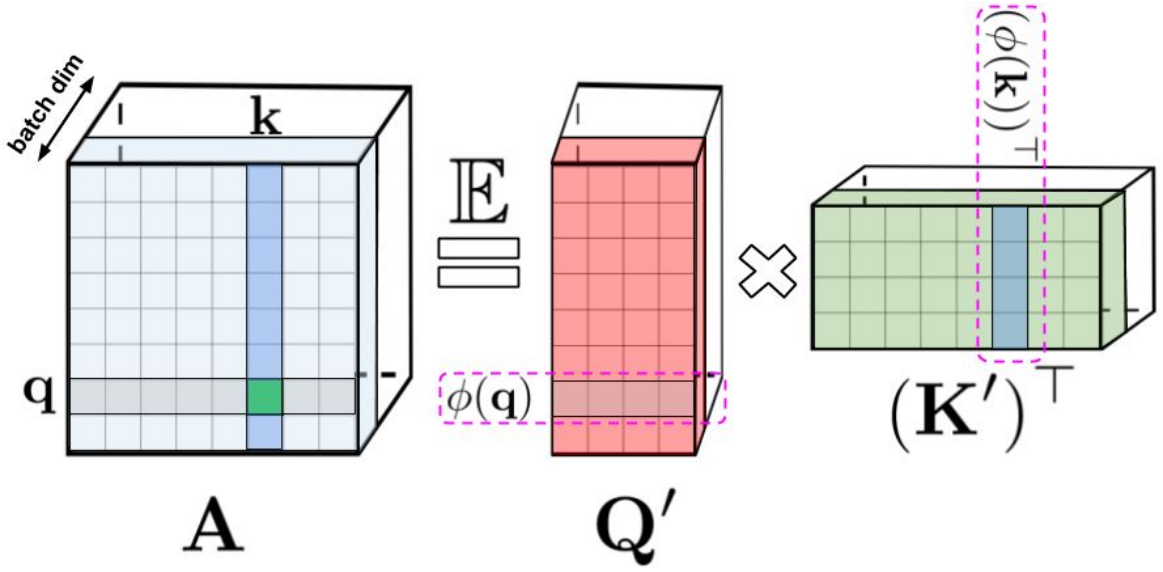
```
def document_mask(b, h, q, kv):
    return doc_id[q] == doc_id[kv]
```

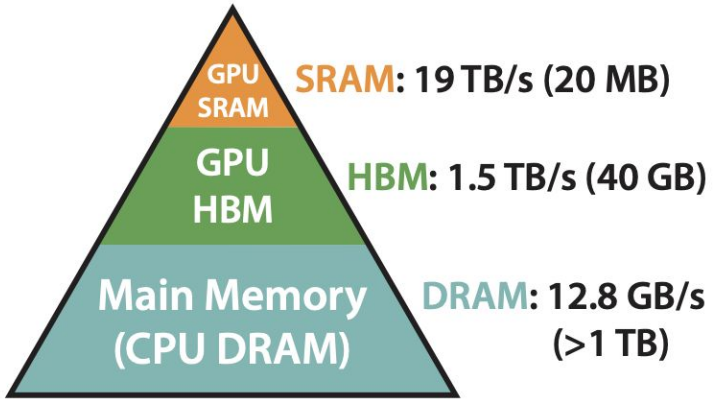



```
def prefix(b, h, q_idx, kv_idx):
    return kv_idx <= prefix_length[b]
prefixLM = or_mask(prefix, causal_mask)
```

Flash Attention: Context

Memory is main bottleneck in attention computation

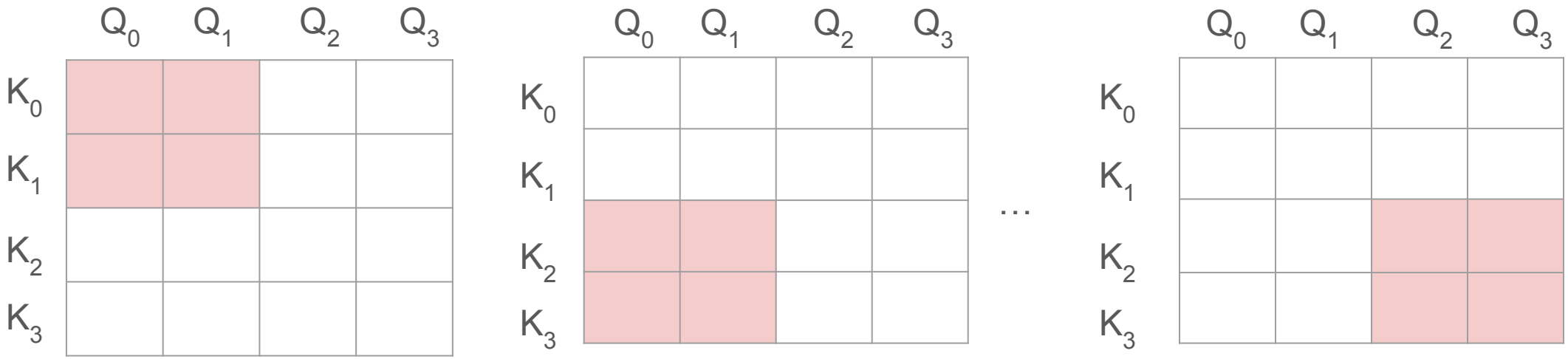




Memory Hierarchy with Bandwidth & Memory Size

Flash Attention: Solution

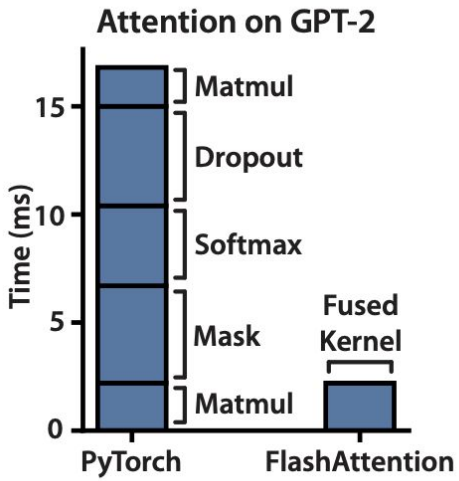
Key: Avoid HBM access by tiling

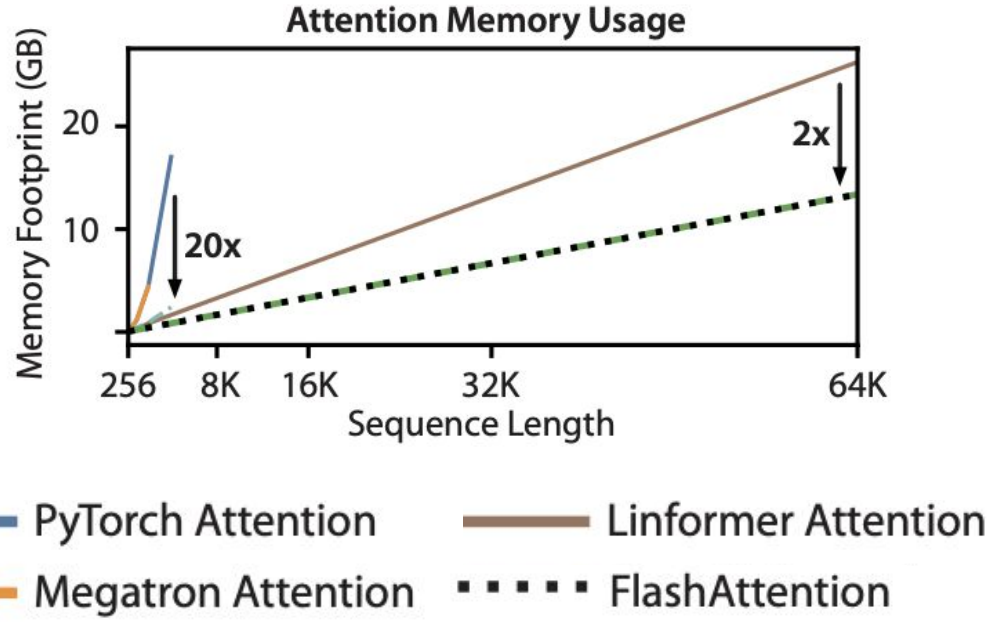


$$\text{Col\_Max}_i = \max(Q_i K_j)$$
$$\text{Col\_Sum\_Exp}_i = \text{sum}(\exp(Q_i K_j - \text{Col\_Max}_i)) \Rightarrow O_{ij} = \frac{\exp(Q_i K - \text{Col\_Max}_i) * V_j}{\text{Col\_Sum\_Exp}_i}$$

Flash Attention: Results

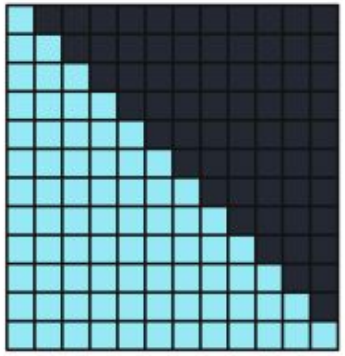
Noticeable speedups in runtime



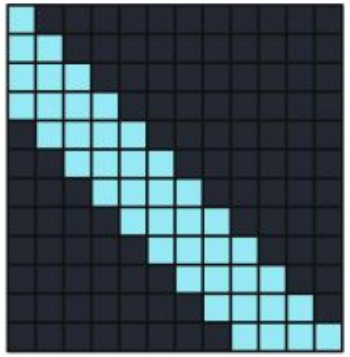


Flash Attention: Limitation

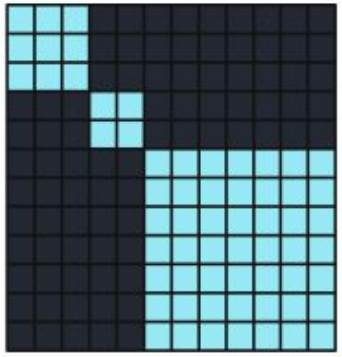
- Only supports standard, rigid attention
- If researchers need to experiment with a attention variant, they need to develop new CUDA kernels entirely from scratch



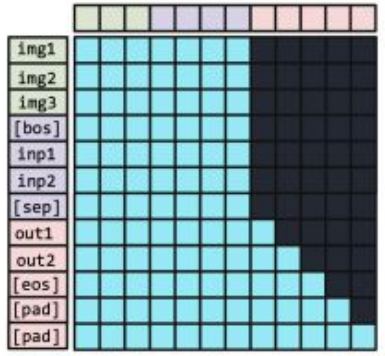
```
def causal_mask(b, h, q, kv):
    return q >= kv
```



```
def sliding_window_mask(b, h, q, kv):
    return q <= kv + window and q >= kv
```

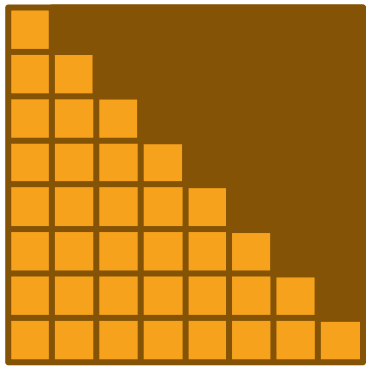


```
def document_mask(b, h, q, kv):
    return doc_id[q] == doc_id[kv]
```

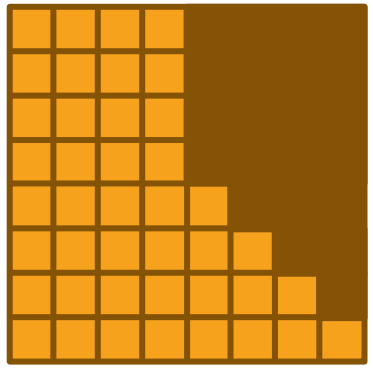



```
def prefix(b, h, q_idx, kv_idx):
    return kv_idx <= prefix_length[b]
prefixLM = or_mask(prefix, causal_mask)
```

Block Masks + Logical Fusions

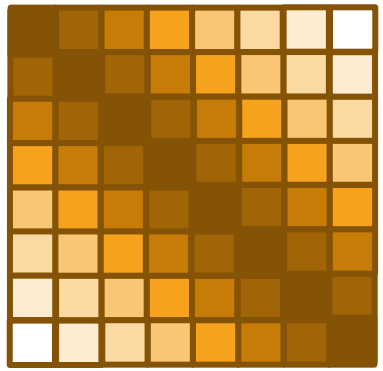


```
def causal_mask(batch: int, head: int, q_idx: int, kv_idx: int):
    return q_idx >= kv_idx
```

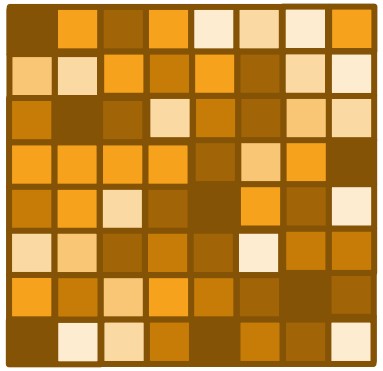


```
def prefix(batch: int, head: int, q_idx: int, kv_idx: int):
    return kv_idx <= prefix_length[batch]

prefix_mask = or_mask(prefix, causal_mask)
```

Score Modification Functions



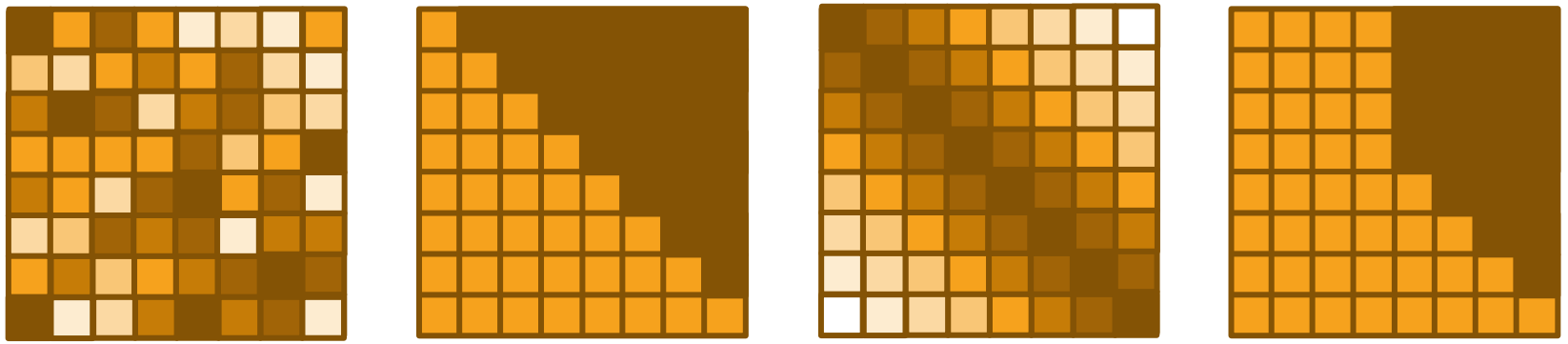
```
def alibi_bias(score: double, batch: int, head: int, q_idx: int, kv_idx: int):
    return score + alibi_bias[head] * (q_idx – kv_idx)
```



```
def tanh_mod(score: double, batch: int, head: int, q_idx: int, kv_idx: int):
    return tanh(score)
```

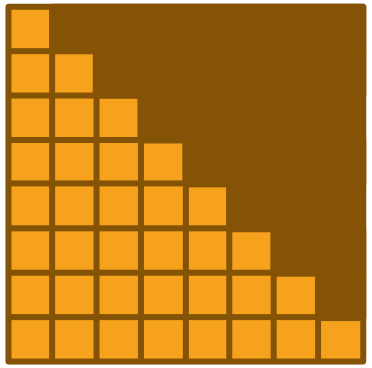
Unified Attention Abstraction

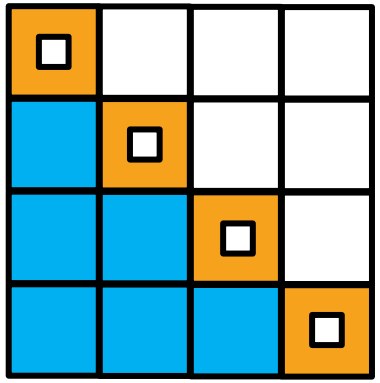
```
def mask_mod(batch: int, head: int, q_idx: int, kv_idx: int)
        → bool

def score_mod(score: double, batch: int, head: int, q_idx: int, kv_idx: int)
        → double
```



Why Mask Modifications?

"attention variants usually show high sparsity such as 50%"





full blocks ⟶ ignore mask

partial blocks ⟶ apply mask element-wise

empty blocks ⟶ skip computation

autotuned + configurable block sizes

Flex Attention Overview

Implemented new attention variant via 2 simple functions

Abstracted into

- Mask Mod: specifies if scalar value should be masked out
- Score Mod: applies arb., fine-grained update to score scalars

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}})\mathbf{V}$$

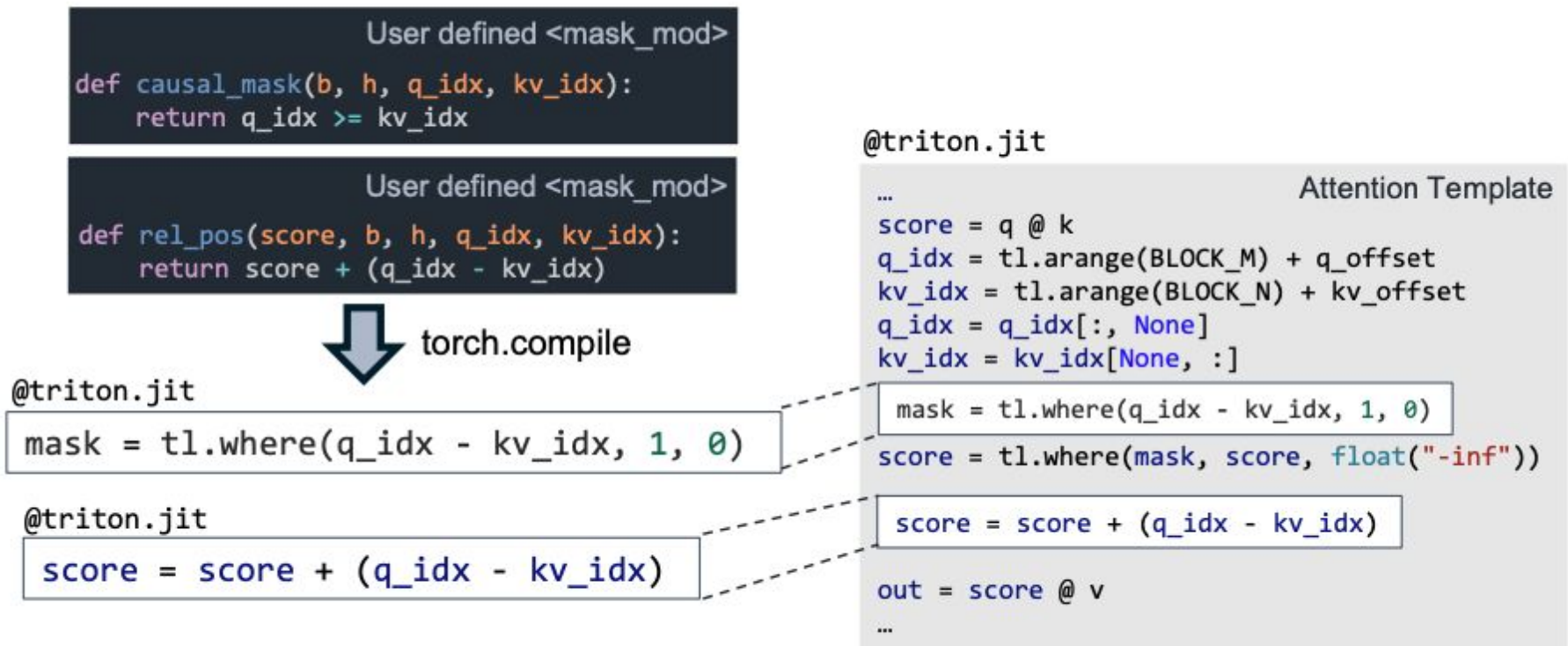
$$\text{FlexAttention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}(\text{mod}(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}))\mathbf{V}$$

Mod Signatures

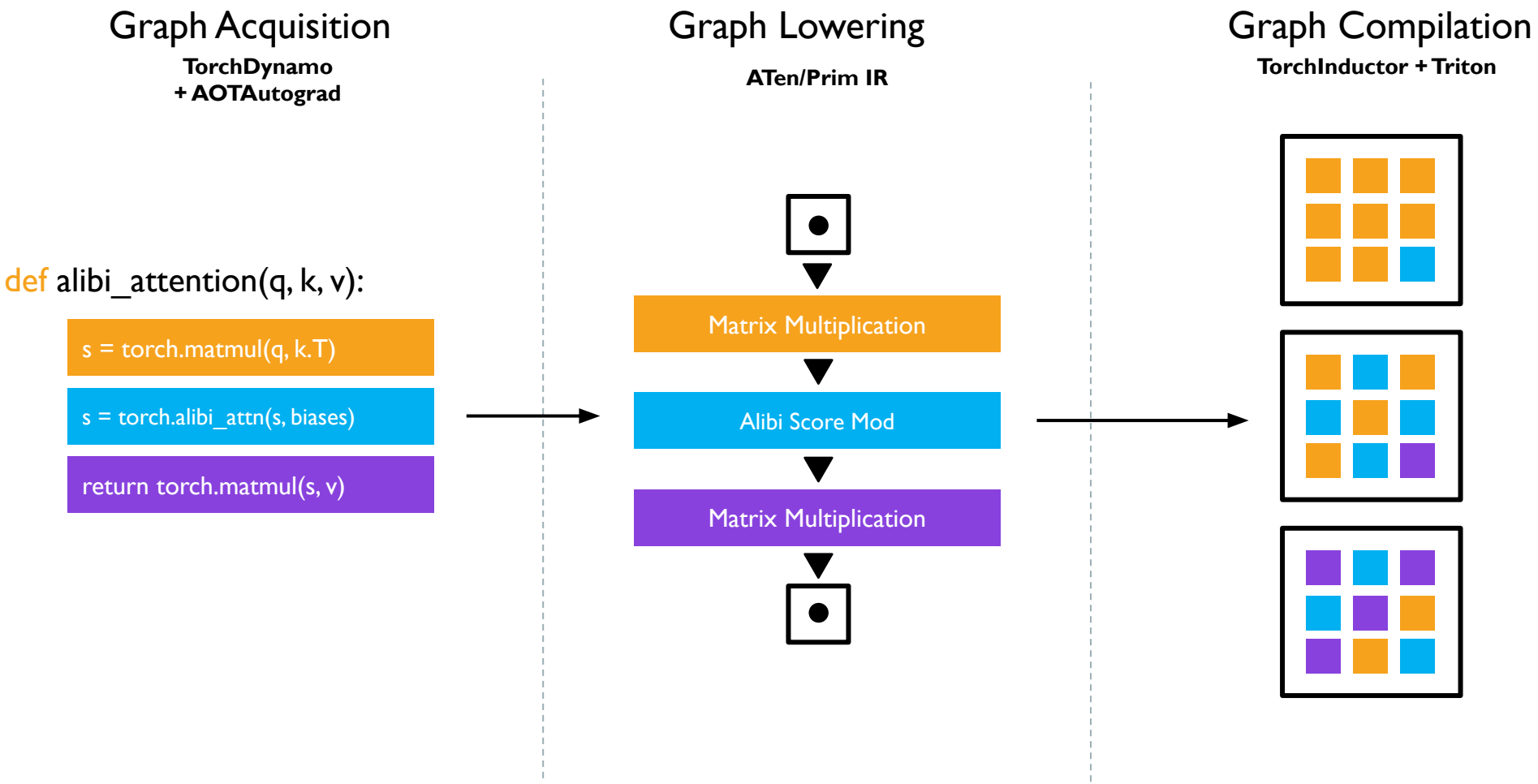
```
def mask_mod(batch_idx: int, head_idx: int,
     q_idx: int, kv_idx: int) -> bool
```

```
def score_mod(score: T, batch_idx: int,
    head_idx: int, q_idx: int, kv_idx: int)
     -> T
```

Code Compilation



Automatic Kernel Compilation



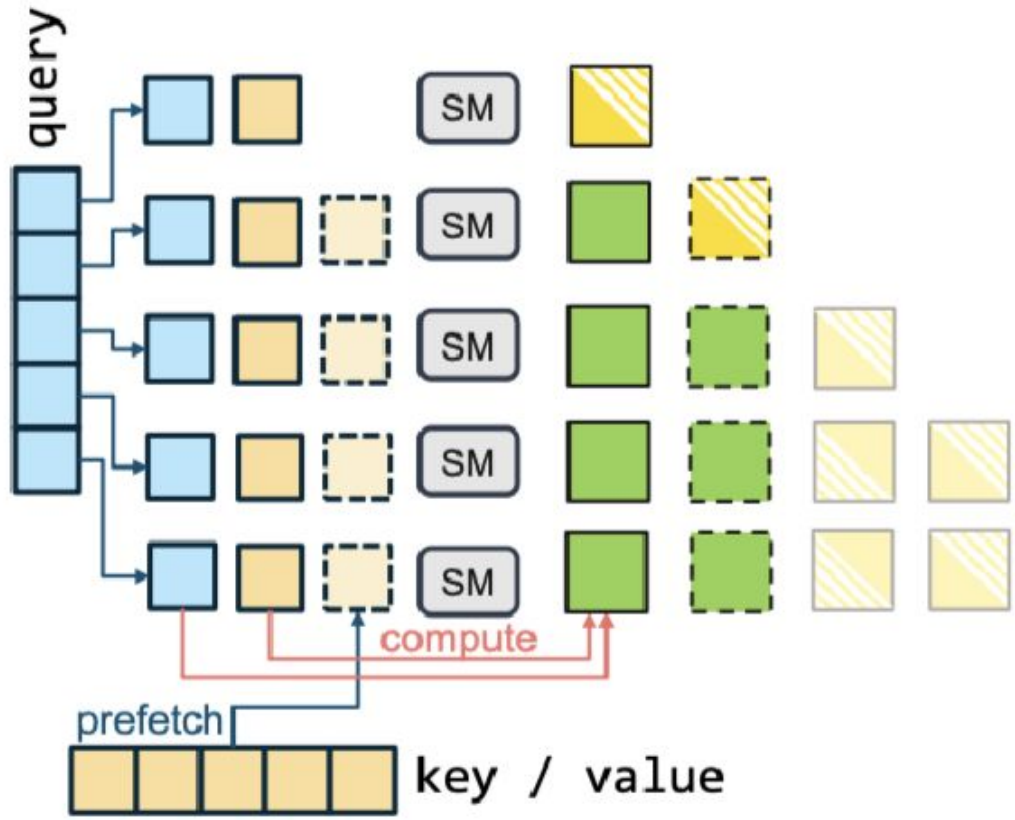
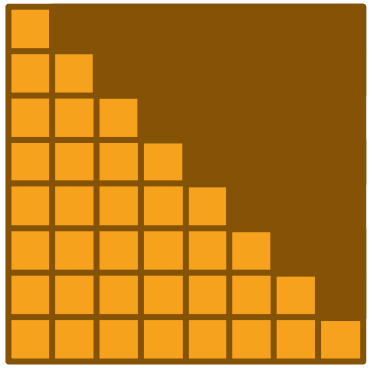
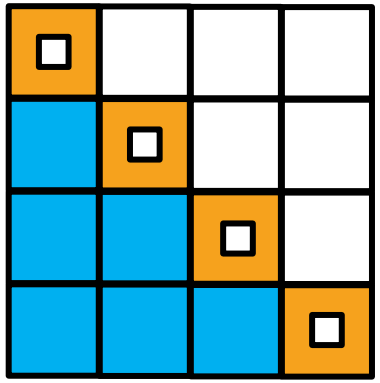


Figure 4. Scheduling full and partial blocks to SM.

Why Mask Modifications?

"attention variants usually show high sparsity such as 50%"





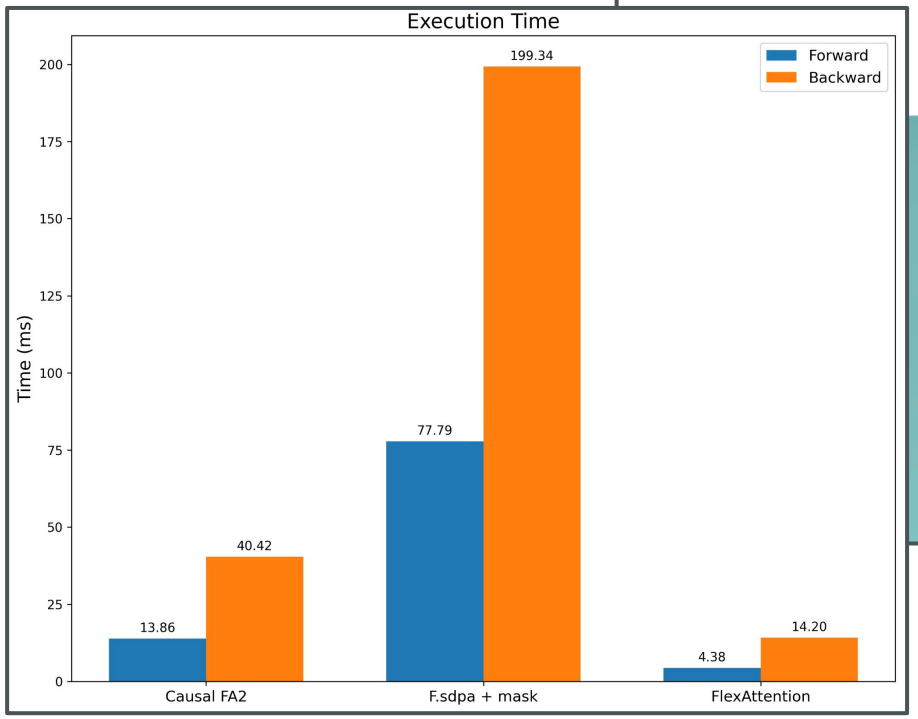
full blocks ⟶ ignore mask

partial blocks ⟶ apply mask element-wise

empty blocks ⟶ skip computation

autotuned + configurable block sizes

Results

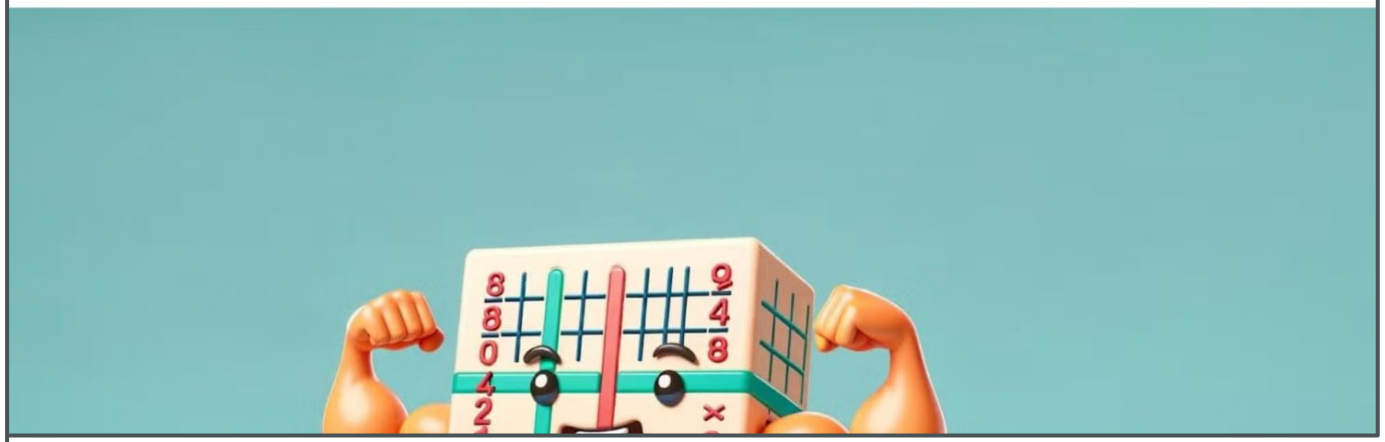


PyTorch Learn Community Projects Docs Blog & News About JOIN

Blog

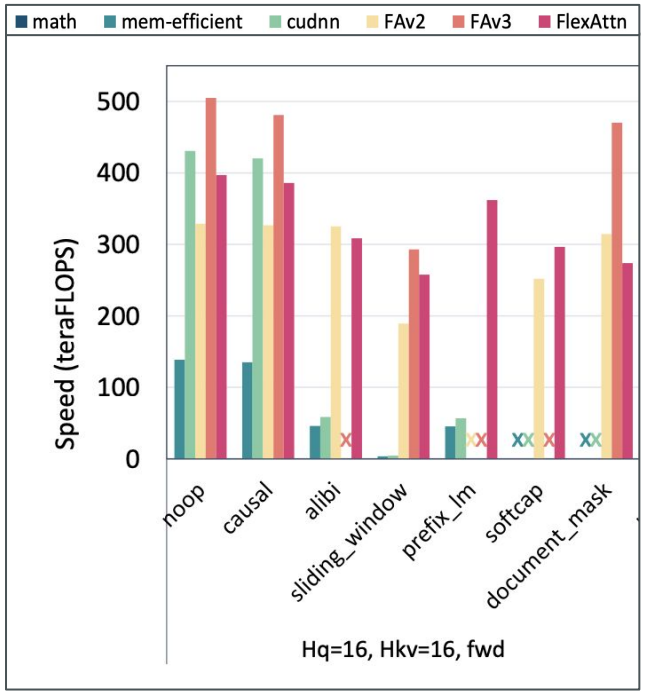
FlexAttention: The Flexibility of PyTorch with the Performance of FlashAttention

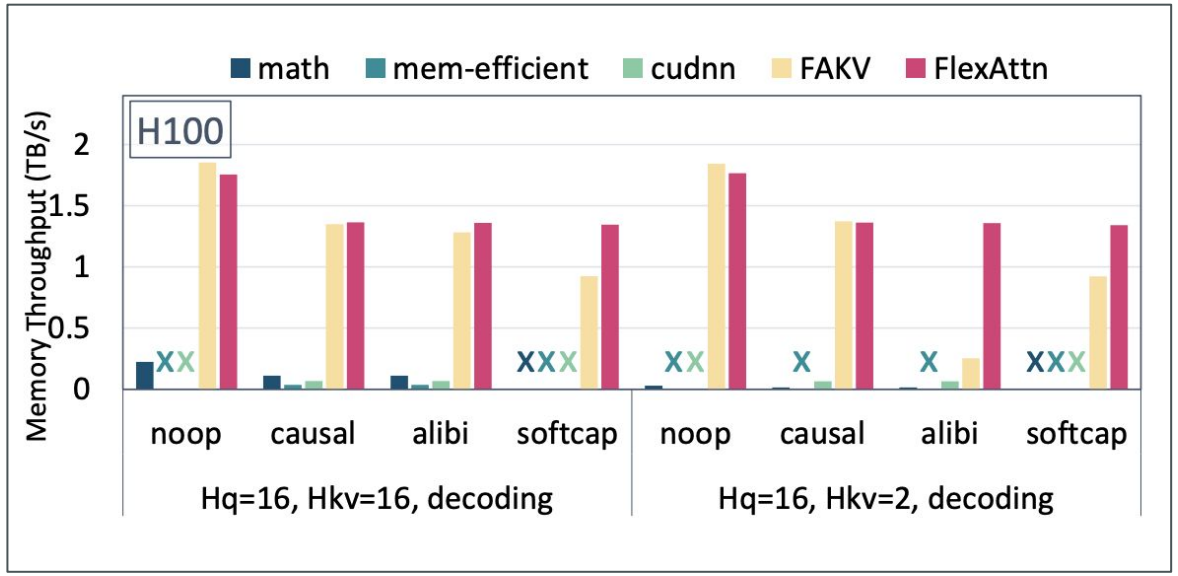
By Team PyTorch: Driss Guessous, Yanbo Liang, Joy Dong, Horace He | August 7, 2024

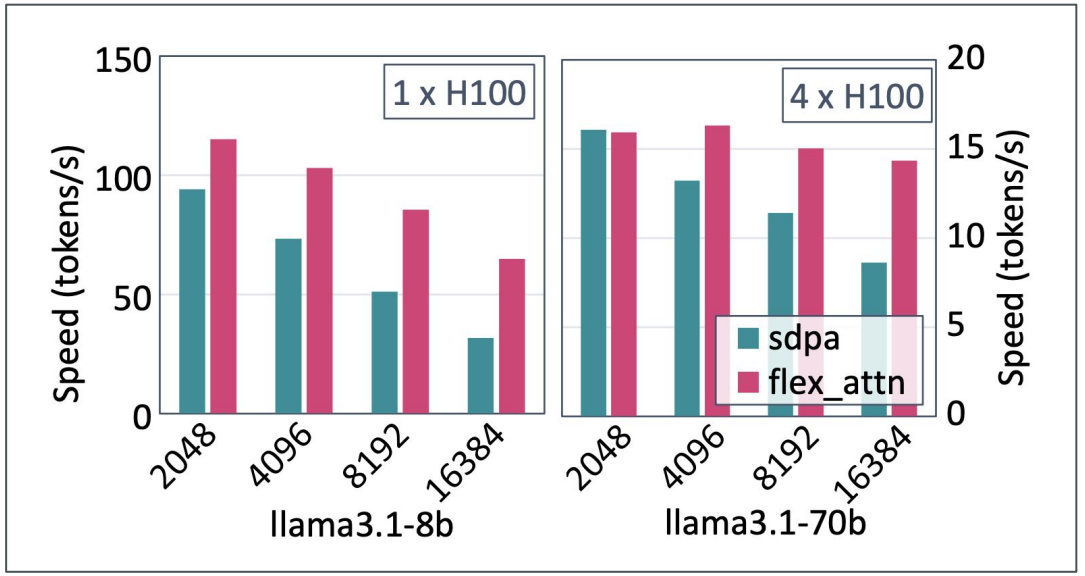


Results

- FlexAttention is an improvement on FlashAttention that enables more attention variants and often turns out to be more efficient than eager implementations.







Limitations / Future Scope

- FlexAttention is an improvement that allows for more attention variants than FlashAttention does, but on common variants (causal, noop, etc) it is actually slower than FlashAttention or hand-written cuda implementations.

- The paper loosely concludes that it hopes to help “explore new attention variants without being held back by what hand-written kernels support.” A future scope could include not only adapting new attention variants, but optimizing for the most common variants as well.

- Similarly, the paper could continue to show performance on more new variants and show a stronger use case for FlexAttention over FlashAttention, given widespread adoption is the goal.