Efficient & Flexible Attention with IO-Aware Fused Kernels

Attention is Memory/IO-bound

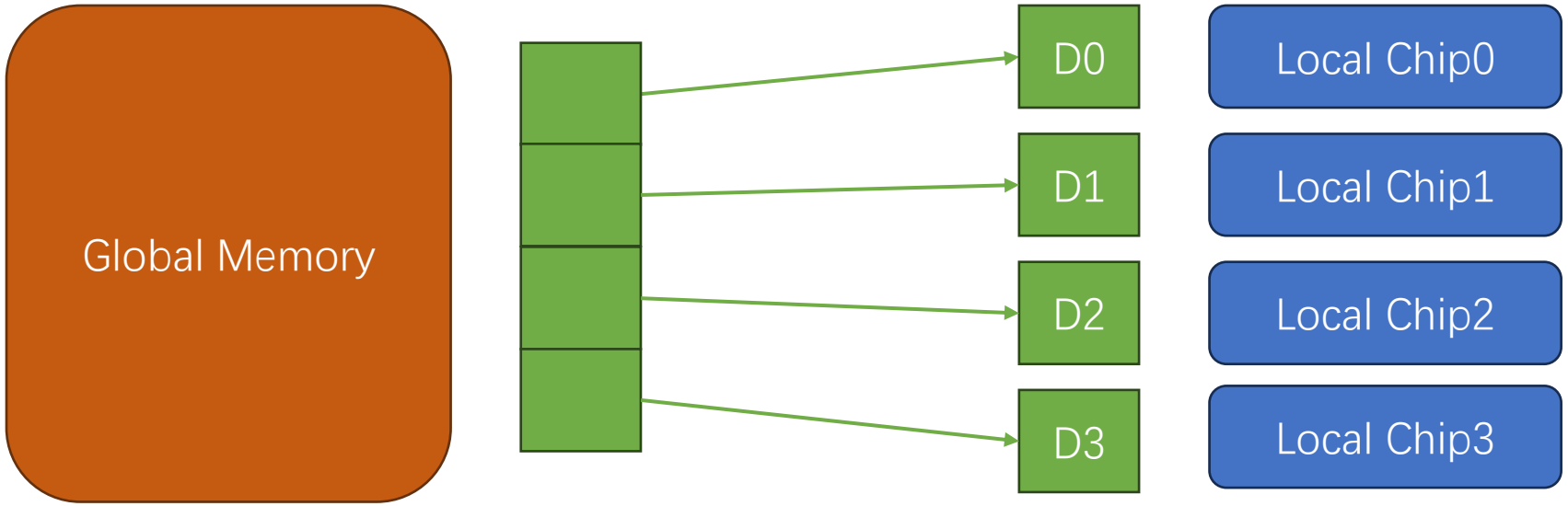
- (Self-)Attention is the key component in Transformers, which **aggregate information** from the contexts.

- Different from plain linear layers:
  - Linear: *OUTPUT*=**dot**(*INPUT*, **WEIGHT**)
    (* Weight is **fixed** for all the positions, and only needs one load.)
  - Attention: S=**dot**(*Query*, **Key**), P=**softmax**(S), O=dot(P, **Value**)
    (* Key is **input-specific, P can be large,** and may need multiple loadings/savings.)

- The Attention operation can be easily **memory- and IO-bounded**.

GPU Memory Model

- Like any computer system, the memory model of GPU has a hierarchy of:
  - **Larger** but **slower** **Global** memory (HBM)
  - **Smaller** but **faster** **Local** memory (SRAM, registers)



Plain Attention

INPUT: **Q**, **K**, **V**

OUTPUT: **O**

[Lq,D][Lk,D]

1. **S** = dot(**Q**, **K**^T)

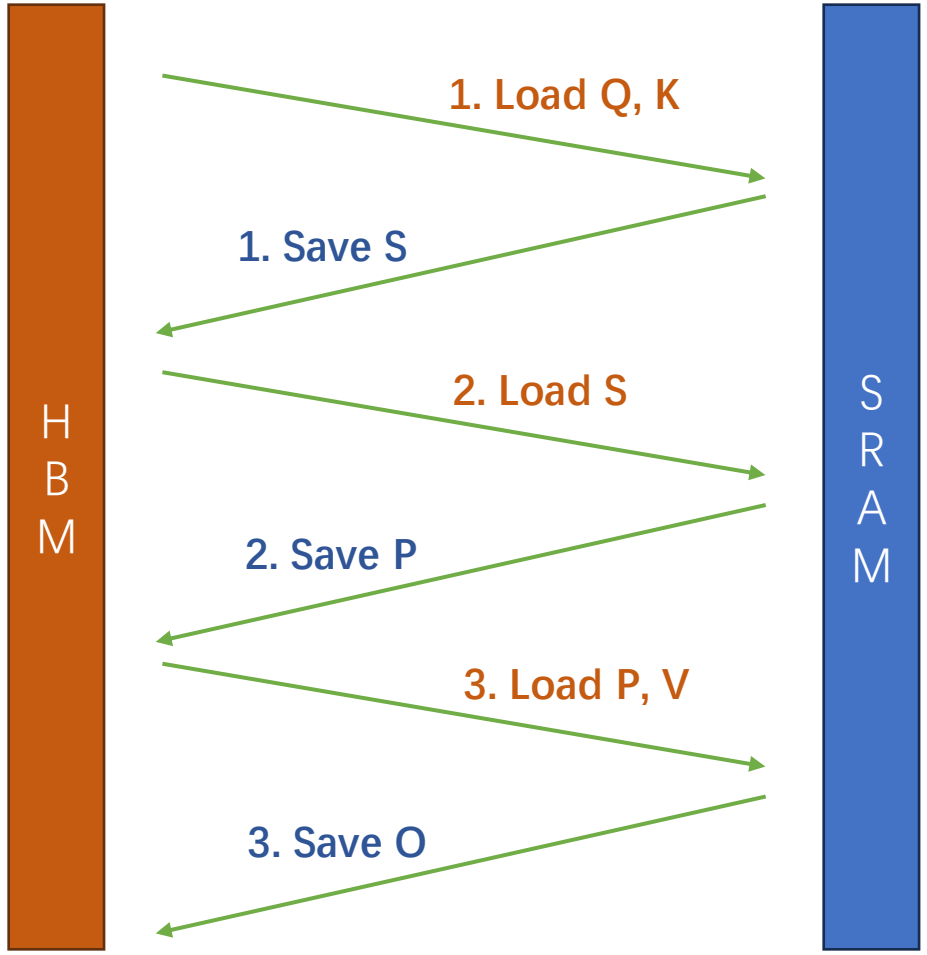
[Lq,Lk]

2. **P** = softmax(**S**)

[Lq,D] [Lk,D]

3. **O** = dot(**P**, **V**)

Intermediate results can be very large, but there is no need to be explicit!



Flash Attention

- **Flash Attention**: a **fused kernel** that eliminated the extra costs of intermediate results loading/saving. [1] [2]

INPUT: **Q**, **K**, **V**

OUTPUT: **O**

[Lq//BLOCK,D]

For **Qi, Oi** in **Q, O**:

[Lk//BLOCK,D]

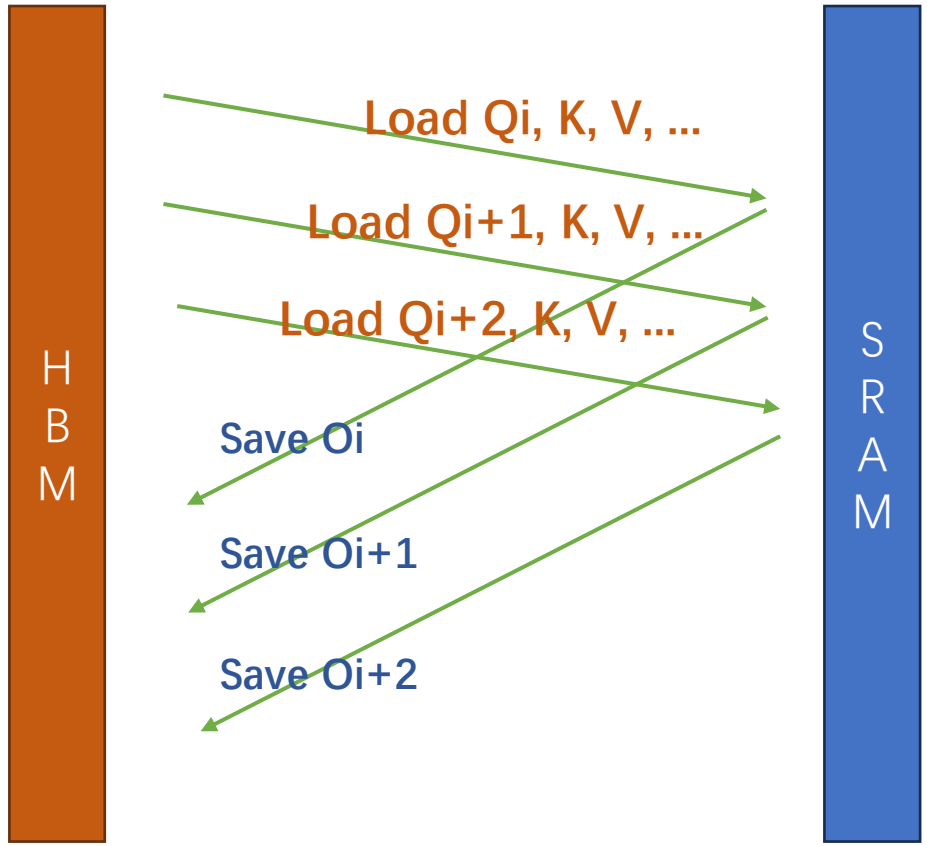
For **Kj**, **Vj** in **K**, **V**:

[Lq//BLOCK, Lk//BLOCK] Sj=dot(Qi, Kj^T)

Aj=*Combine*(Aj-1, Sj, Vj)

[Lq//BLOCK,D]

Store A to Oi



Online Softmax & Tiling

- One key technique is **online softmax** [3][4], which makes it possible to compute attention by blocks (**Tiling**).
- From [1]:

Tiling. We compute attention by blocks. Softmax couples columns of $\mathbf{K}$, so we decompose the large softmax with scaling [51, 60, 66]. For numerical stability, the softmax of vector $x \in \mathbb{R}^B$ is computed as:

$$m(x) := \max_i \; x_i, \quad f(x) := \left[e^{x_1 - m(x)} \quad \dots \quad e^{x_B - m(x)}\right], \quad \ell(x) := \sum_i f(x)_i, \quad \text{softmax}(x) := \frac{f(x)}{\ell(x)}.$$

Original m get canceled.

For vectors $x^{(1)}, x^{(2)} \in \mathbb{R}^B$, we can decompose the softmax of the concatenated $x = \left[x^{(1)} \; x^{(2)}\right] \in \mathbb{R}^{2B}$ as:

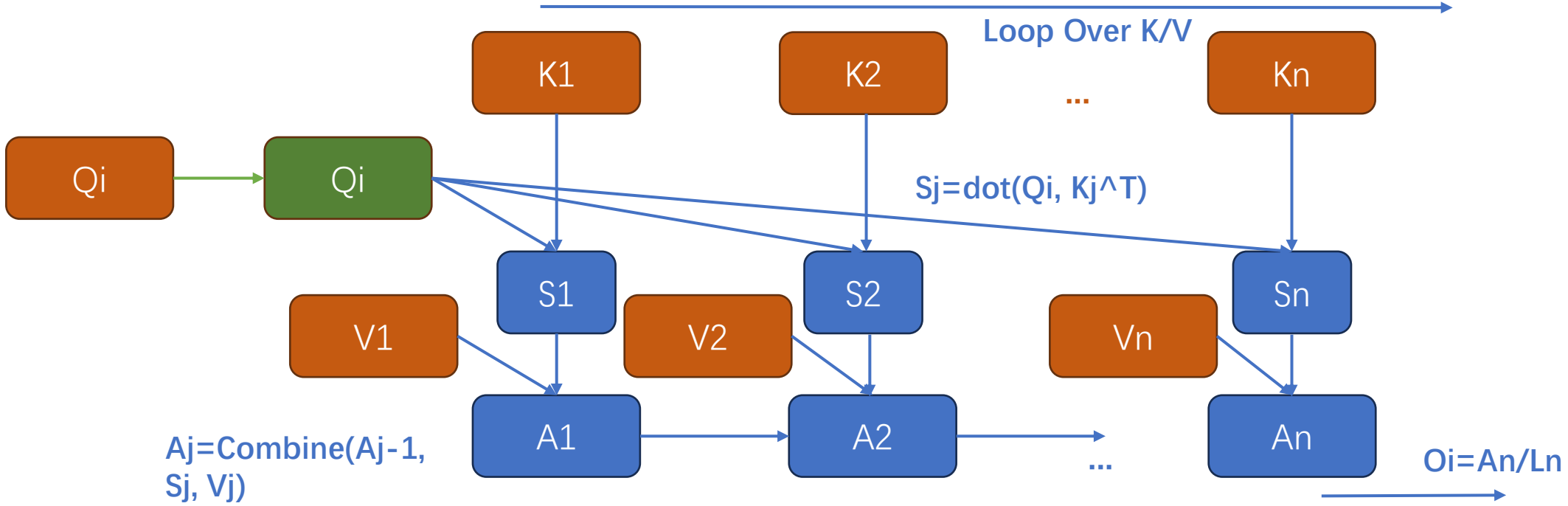
$$m(x) = m(\left[x^{(1)} \; x^{(2)}\right]) = \max(m(x^{(1)}), m(x^{(2)})), \quad f(x) = \left[e^{m(x^{(1)}) - m(x)} f(x^{(1)}) \quad e^{m(x^{(2)}) - m(x)} f(x^{(2)})\right],$$

$$\ell(x) = \ell(\left[x^{(1)} \; x^{(2)}\right]) = e^{m(x^{(1)}) - m(x)} \ell(x^{(1)}) + e^{m(x^{(2)}) - m(x)} \ell(x^{(2)}), \quad \text{softmax}(x) = \frac{f(x)}{\ell(x)}.$$

Section 3.1 From (Dao et al., 2022)

Flash Forward: Chunking

- Job chunking:
  - Q/K/V Shapes: [***B*H***, **Lq**, D], [*B*H*, Lk, D], [*B*H*, Lk, D]
  - 1. Each **Batch/Head(B*H)** can be handled individually.
  - 2. Each **Query(Lq)** is also independent to each other!



Flash Forward: Online Normalizer

Over the K/V loop, we store several things (remember that each S is a chunk rather than a single value):

- Maximum value of the score: $m_j = \max(\max(S_1), \max(S_2), \ldots, \max(S_j))$
- Sum of the exponetials: $l_j = \sum_k \exp(S_{1k} - m_j) + \sum_k \exp(S_{2k} - m_j) + \ldots + \sum_k \exp(S_{jk} - m_j)$
- Accumulated weighted values: $A_j = \sum_k \exp(S_{1k} - m_j) \cdot V_{1k} + \sum_k \exp(S_{2k} - m_j) \cdot V_{2k} + \ldots + \sum_k \exp(S_{jk} - m_j) \cdot V_{jk}$

From the index of j to $j + 1$:

- Take the maximum of both: $m_{j+1} = \max(m_j, \max(S_{j+1}))$
- Calculate a reweighting alpha: $\alpha = \exp(m_j - m_{j+1})$
- Update l: $l_{j+1} = \alpha \cdot l_j + \sum_k \exp(S_{j+1,k} - m_{j+1})$
- Update A: $A_{j+1} = \alpha \cdot A_j + \sum_k \exp(S_{j+1,k} - m_{j+1}) \cdot V_{j+1,k}$

Finally, since everyone now has the same normalizer, the final output would be:

- $O = A_{-1}/l_{-1}$

Flash Backward

- Attention score **recalculation is required** since they are not explicitly stored in the forward progress.
- Things are more complicated than forward, since there are three gradients that need to be accumulated: **dQ**, **dK**, **dV.**
- The flash-v2 (CUDA-version) chooses to **grid-split over K/V** and accumulate over Q for the loop of the kernel.
- While for the triton version, **two separate kernels** for K/V and for Q seem to be faster than load/save with accumulated dQ:
  - https://github.com/FlagOpen/FlagAttention/issues/4

Implementation with Triton

- Triton makes a combination of the **flexibility** of Python and the **efficiency** of customized kernels.

```python
@triton.jit
def add_kernel(x_ptr,  # *Pointer* to first input vector.
               y_ptr,  # *Pointer* to second input vector.
               output_ptr,  # *Pointer* to output vector.
               n_elements,  # Size of the vector.
               BLOCK_SIZE: tl.constexpr,  # Number of elements each program should process.
               # NOTE: `constexpr` so it can be used as a shape value.
               ):
    # There are multiple 'programs' processing different data. We identify which program
    # we are here:
    pid = tl.program_id(axis=0)  # We use a 1D launch grid so axis is 0.
    # This program will process inputs that are offset from the initial data.
    # For instance, if you had a vector of length 256 and block_size of 64, the programs
    # would each access the elements [0:64, 64:128, 128:192, 192:256].
    # Note that offsets is a list of pointers:
    block_start = pid * BLOCK_SIZE
    offsets = block_start + tl.arange(0, BLOCK_SIZE)
    # Create a mask to guard memory operations against out-of-bounds accesses.
    mask = offsets < n_elements
    # Load x and y from DRAM, masking out any extra elements in case the input is not a
    # multiple of the block size.
    x = tl.load(x_ptr + offsets, mask=mask)
    y = tl.load(y_ptr + offsets, mask=mask)
    output = x + y
    # Write x + y back to DRAM.
    tl.store(output_ptr + offsets, output, mask=mask)
```

- All tensors are inputted as pointers.
- Sizes are usually explicitly inputted.
- Constexpr is decided at compiling time.

The full kernel job is split into a grid of individual jobs (programs/thread-block).

```python
    # The SPMD launch grid denotes the number of kernel instances that run in parallel.
    # It is analogous to CUDA launch grids. It can be either Tuple[int], or Callable(metaparameters) ->
    # In this case, we use a 1D grid where the size is the number of blocks:
    grid = lambda meta: (triton.cdiv(n_elements, meta['BLOCK_SIZE']), )
    # NOTE:
    #  - Each torch.tensor object is implicitly converted into a pointer to its first element.
    #  - `triton.jit`'ed functions can be indexed with a launch grid to obtain a callable GPU kernel.
    #  - Don't forget to pass meta-parameters as keywords arguments.
    add_kernel[grid](x, y, output, n_elements, BLOCK_SIZE=1024)
```

Loading from Global to Local memory -> Calculating -> Store back.

From Tutorial-01

Full-Attention Comparisons

Comparisons among several choices:

1. Pytorch [explicit att]
2. FSDP math [explicit att]
3. FSDP memory efficient [implicit att]
4. Flash Attention (v2) [CUDA kernel]
5. Triton Flash Attention [Triton kernel]

40G-A100 qkv=[[16//L, 32, 1024*L, **128**] for L in [1, 2, 4, 8, 16]] + Causal + bf16

(TFlops/Sec)	1K	2K	4K	8K	16K
torch	10.6	10.39	OOM	OOM	OOM
fsdp.math	23.77	20.9	23.72	OOM	OOM
fsdp.mem	59.98	72.42	80.26	85.36	86.8
flash-cuda	106.15	135.22	156.44	170.24	177.34
flash-triton	95.61	111.78	119.93	125.22	127.25
	90.07%	82.67%	76.66%	73.55%	71.75%

40G-A100 qkv=[[16//L, 32, 1024*L, **64**] for L in [1, 2, 4, 8, 16]] + Causal + bf16

(TFlops/Sec)	1K	2K	4K	8K	16K
torch	5.48	5.3	OOM	OOM	OOM
fsdp.math	13.06	10.99	12.43	OOM	OOM
fsdp.mem	51.24	59.69	64.03	66.28	66.42
flash-cuda	94.79	122.45	142.99	156.3	161.53
flash-triton	96.18	107.54	114.69	118.72	119.9
	101.47%	87.82%	80.21%	75.96%	74.23%

Other Operations

- Basically, such kernels **fuse** multiple operations into one kernel calling, processing intermediate variables in local memories.
  - Saving **memory**: no need to store large intermediate variables
  - Saving **IO**: no need transfer those variables

- There are many other good examples of these operations:
  - Linear + Activation + Dropout
  - Softmax + Cross-entropy
  - …

Extension: Attention with Arbitrary Mask

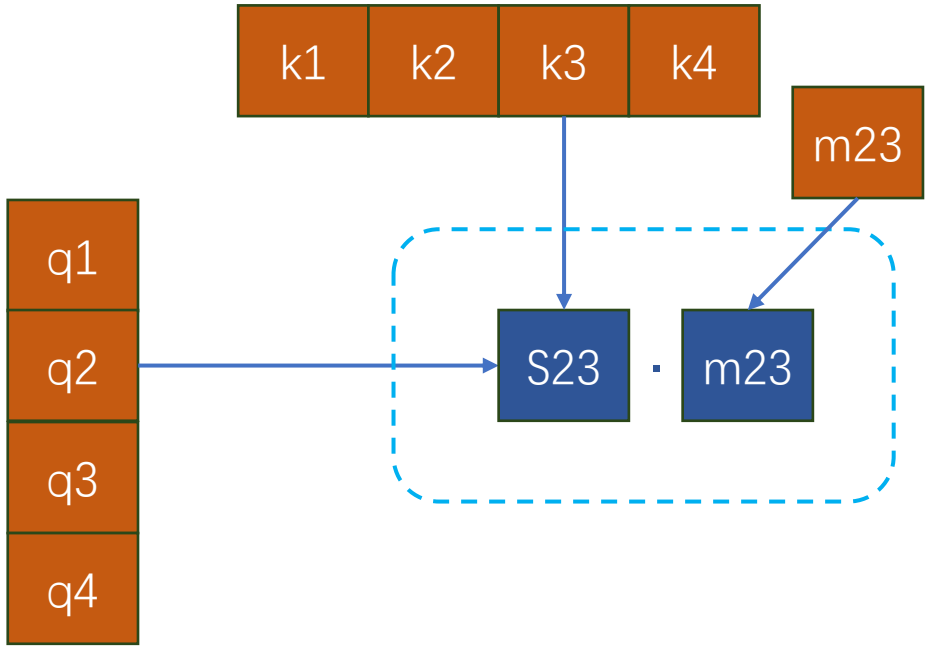
- At many times, we might need modified attention that supports **arbitrary mask**:
  - **Sliding window**: “position(q) - position(k) < window”
  - **Document individual chunks**: “doc_id(q) == doc_id(k)”
  - …
- One straight-forward way is to pre-calculate **the full mask** and then apply it to attention, but again:
  - Memory cost & IO cost

- Alternatively, we can similarly ***calculate the mask online***.

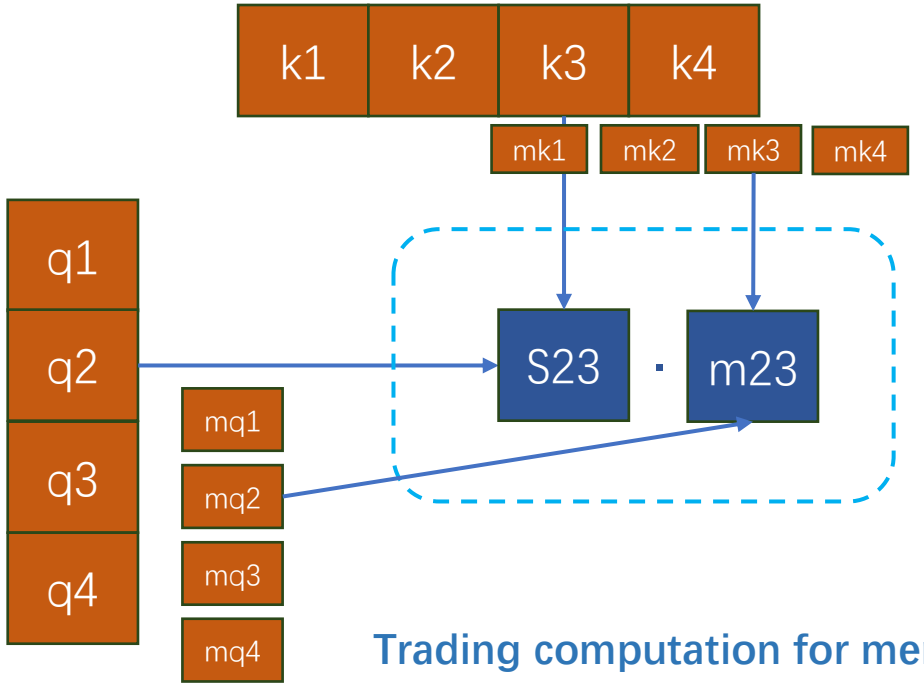
Extension: Attention with Arbitrary Mask

Exactly the same motivation as **FlexAttention**:
https://pytorch.org/blog/flexattention

Explicit Mask Applying



Online Mask Calculation



Trading computation for memory/IO.

Comparisons w/ Doc-Attention

Comparisons among several choices:

(Flash2 does not support this for now)

1. Pytorch [explicit att]
2. FSDP math [explicit att]
3. FSDP memory efficient [implicit att]
4. Triton Flash Attention [Triton kernel]

40G-A100 qkv=[[16//L, 32, 1024*L, **128**] for L in [1, 2, 4, 8, 16]] + **(Causal + Random-Doc-Attention)** + bf16

(TFlops/Sec)	1K	1KD	2K	2KD	4K	4KD	8K	8KD	16K	16KD
torch	5.48	9.44	5.3	9.23	OOM	OOM	OOM	OOM	OOM	OOM
fsdp.math	13.06	23.34	10.99	20.60	12.43	23.39	OOM	OOM	OOM	OOM
fsdp.mem	51.24	34.62	59.69	37.19	64.03	38.37	66.28	39.10	66.42	39.42
flash-cuda	94.79	N/A	122.45	N/A	142.99	N/A	156.3	N/A	161.53	N/A
flash-triton	96.18	**86.62**	107.54	**94.97**	114.69	**102.56**	118.72	**106.96**	119.9	**109.49**