More on GPUs, tiling and FlashAttention

Xingda Wei

IPADS, Shanghai Jiao Tong University

https://www.sjtu.edu.cn

Credits: AI-sys Sp22, CMU15-418， Ming Cong @IPADS

Review: huge computation needed by the AI applications

Case study: training LLM w/ Stochastic Gradient Descent

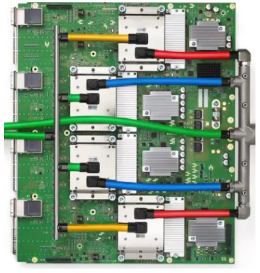
- Expected computation: 6 * #parameters * Batch size

Known fact: GPT-4 has 1.76 trillion parameters [1]

- This is 1,760,000,000,000 parameters
- So this is 10,560,000,000,000 calculations for a single input of **a single iteration**!!

[1] https://the-decoder.com/gpt-4-has-a-trillion-parameters/

Spectrum of computation device available



Intel i5-9600K, single core: 6.3GFLOPS Multiple cores: 37.7GFLOPS

Mate60 GPU 2.3 TFLOPs

Apple M2 GPU 3.6 TFLOPs

NVIDIA A100 GPU 19.5 TFLOPs

Google TPUv4 275 TFLOPS

Performance

Review: parallelism on a single device

3 parallelism strategies on a single device

- Single core+: pipeline + super scalar with instruction level parallelism (ILP)
- Single core++: added SIMD support
- Multiple core: a single core (single core, single core+, single core++) can be glued together !

Question: what's next? Single core+++++?

- Solution (Out of the scope of this lecture): domain-specific accelerators !

The era of domain-specific accelerators

Accelerators (even on general-purpose computing devices)

- Hardware designed to fulfill a single task
- Typically are not general-purpose, e.g., not programmable

CPU:

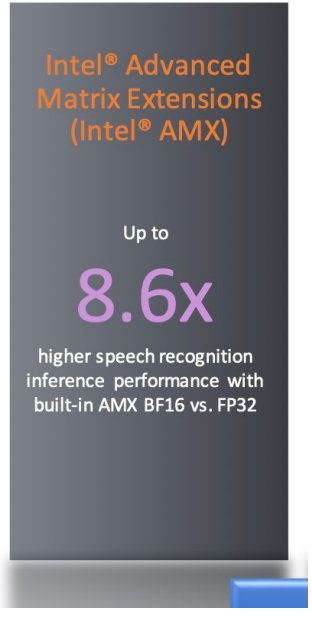
- SIMD + Matrix accelerators (e.g., Intel's AMX accelerators)

GPU:

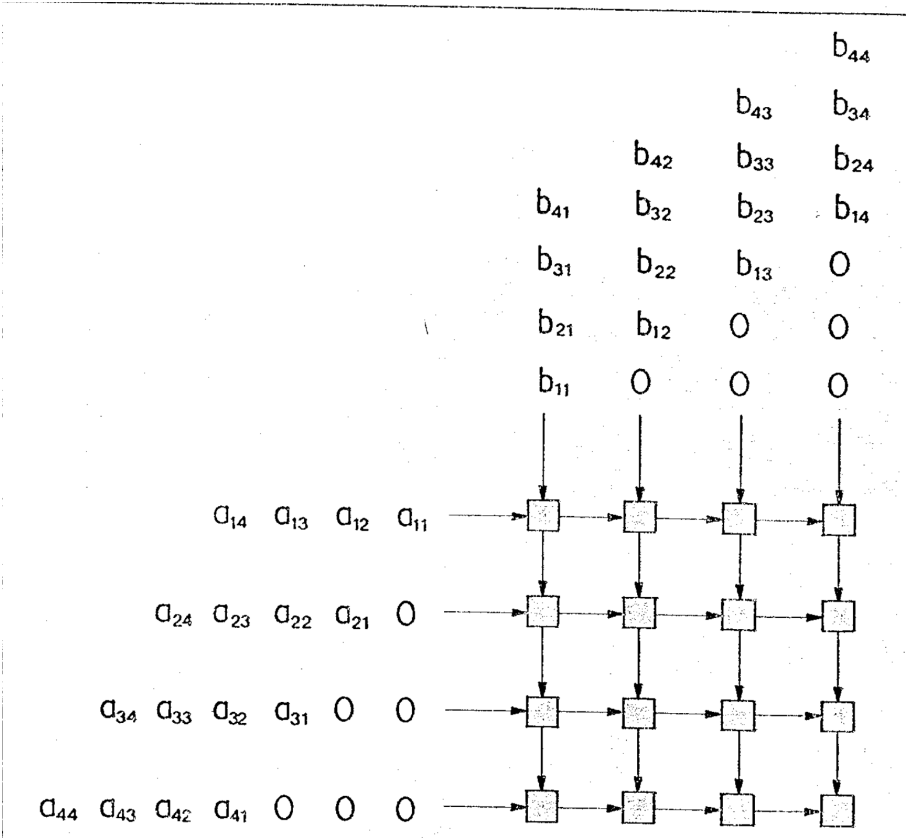
- Tensorcore: accelerators for matrix operations

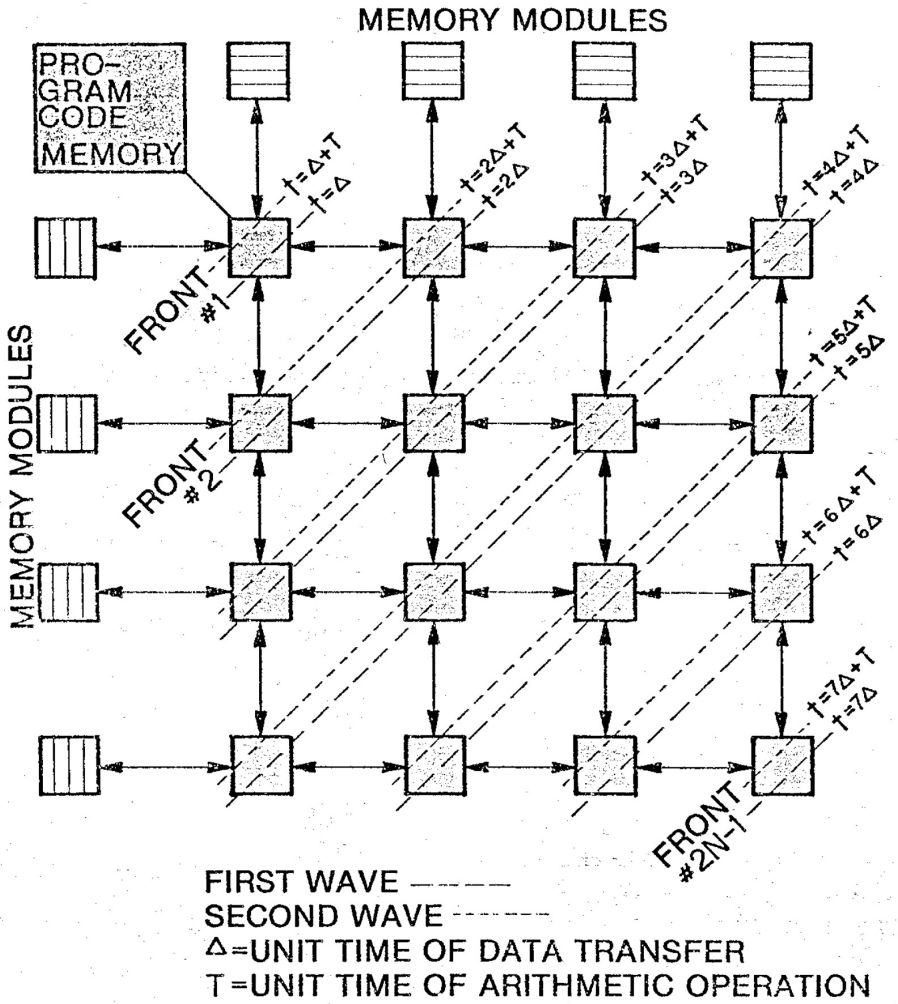
TPU (Tensor processing unit):

- tensor process (optimized for large matrix operations)

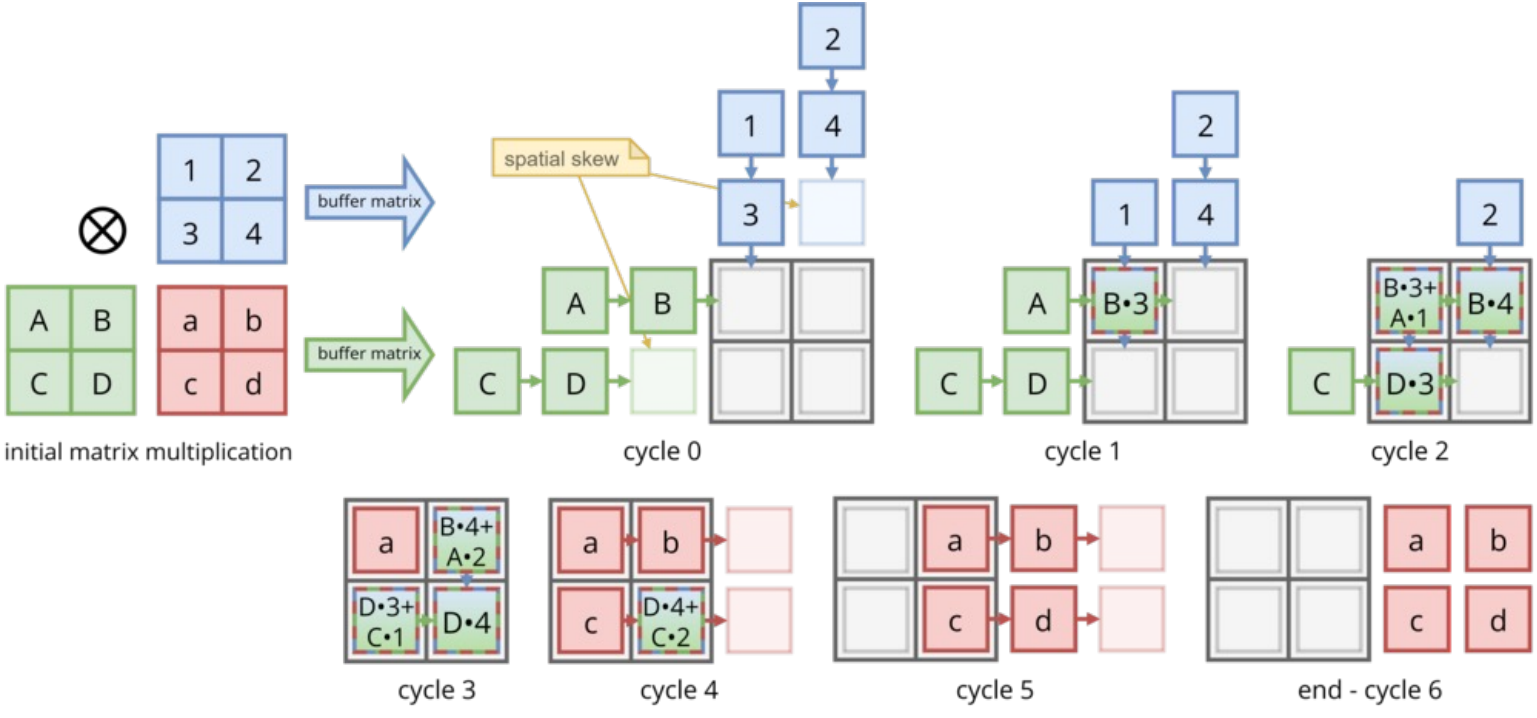


Behind the tensor core: Systolic Array





Example of Systolic Array



Source: https://en.wikipedia.org/wiki/Systolic_array

Case study: **GPU**

More reading: https://modal.com/gpu-glossary/device-hardware/special-function-unit

GPU and (GP)GPU

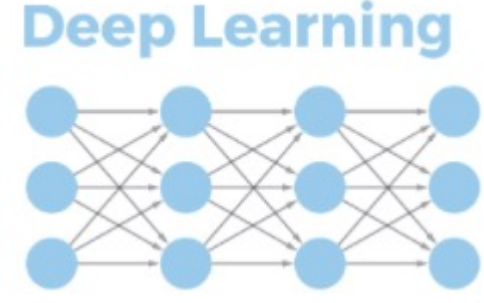
GPU

– Graphics Processing Unit

GPGPU

– General purpose GPU

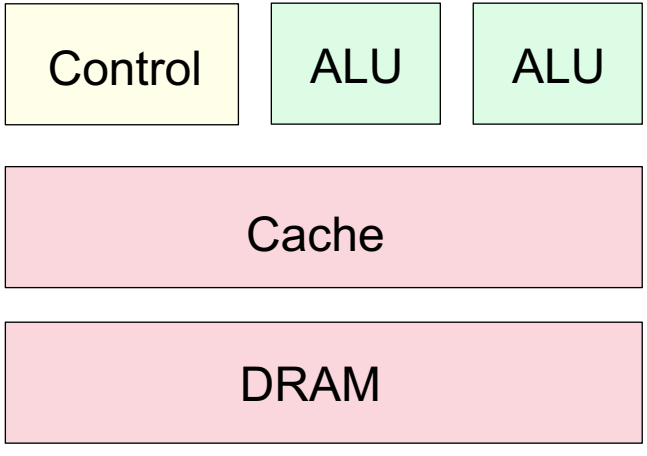
- Traditionally, GPU is primarily designed for image rendering
- Nowadays, we can use GPU for general purpose computing



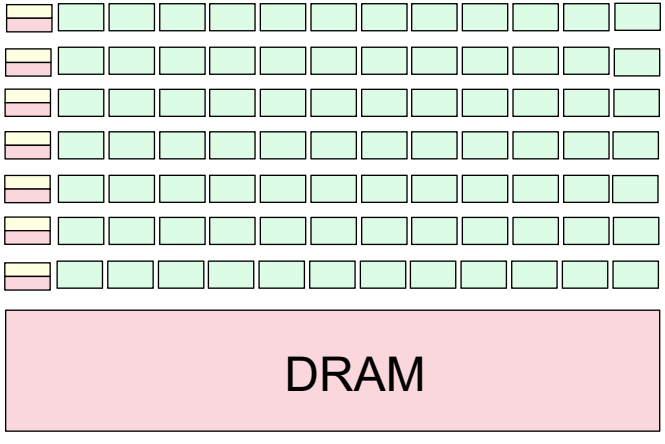
GPU is designed for more computing power than CPU

GPU supports more ALUs and they execute in parallel

- E.g., CPU uses 30% of circuits for control
- E.g., GPU may use 80% of circuits for computing (ALU)



Simplified architecture of CPU



Simplified architecture of GPU

Basic building block: SM

Streaming Multiprocessor (SM)

- Each SM has multiple warps (the detailed number depends on the generations!)

All cores in a warp (e.g., 32 threads) are executing the same instruction

- SIMD execution
- Each core has its own registers

Example

- The architecture of SM of GTX 1080

More example: H100

The most advanced GPU we can buy according to US's law

- H200 is an extension to H100

There are two notable changes here

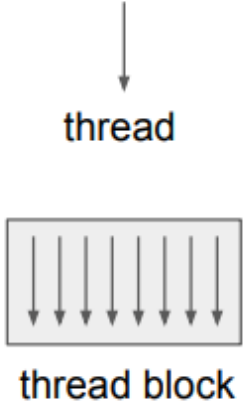
1. The introduction of the tensor cores
2. The number of unit differs, e.g., less SFU

Due to the introduction of tensorcore, the FP32 or other units are typically called CUDA cores

CUDA programming model: SIMT

Abstraction: SIMT

– Single instruction, multiple threads



Concrete language: C code

– Programmer writes code for a single thread using C

– All threads executes the same code

  • But can take different path (control flow supported)

Threads are grouped in a block (or cooperative thread arrays, CTA in NVIDAI's terminology)

CUDA programming model: SIMT

Abstraction: SIMT

– Single instruction, multiple threads

Concrete language : C code *

Threads are grouped in a block

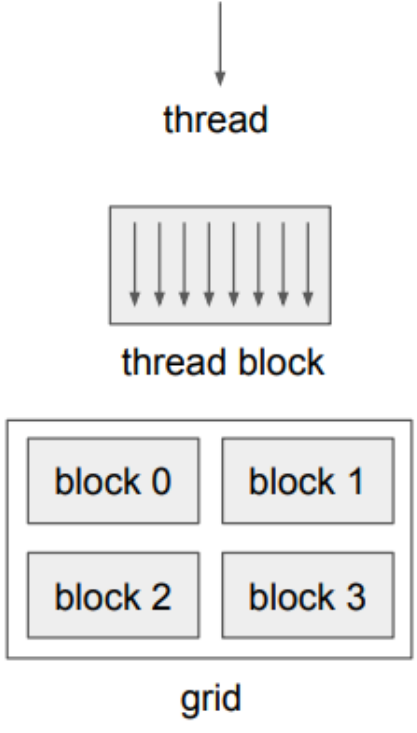
– Threads within the same block can synchronize execution

Blocks are grouped in a grid

– Blocks are independently scheduled on the GPU, can be executed in any order

Kernel (do the real computation, e.g., matrix multiplication)

– A grid of blocks of threads

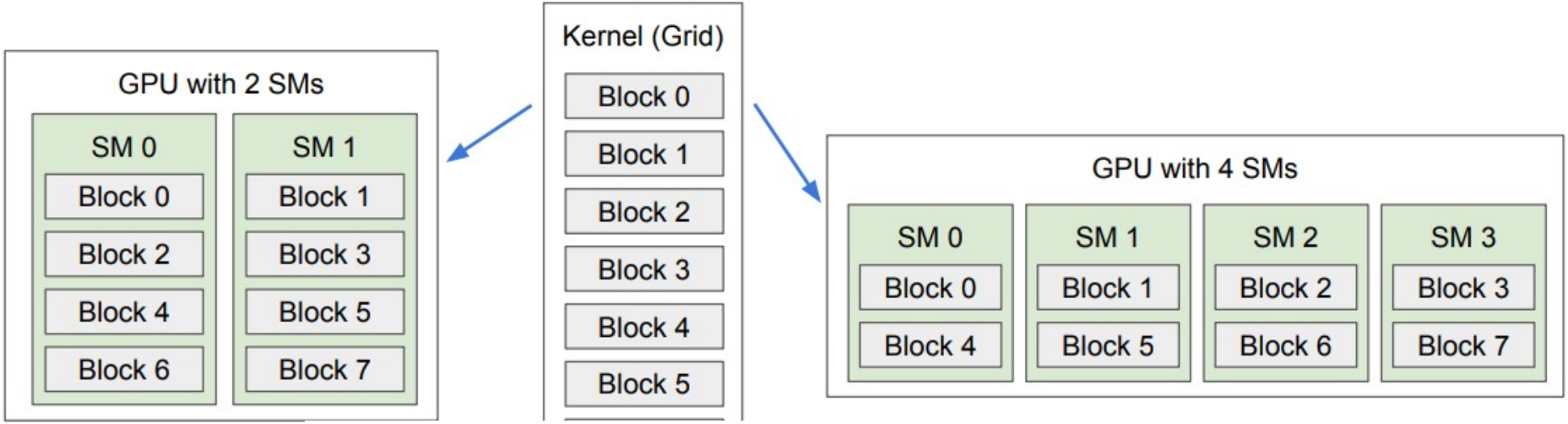


CUDA kernel execution

Each block is executed by one SM and does not migrate

Several concurrent block can reside on one SM

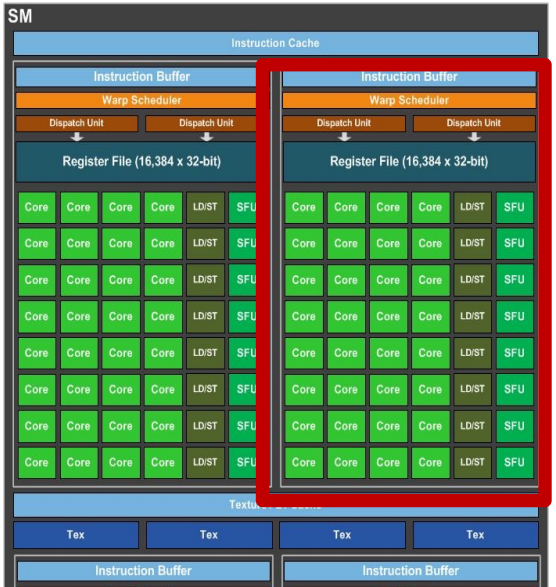
- Depending on block's memory requirement & the SM memory capacity
- The scheduling of block execution is one of the key technique of GPU designs

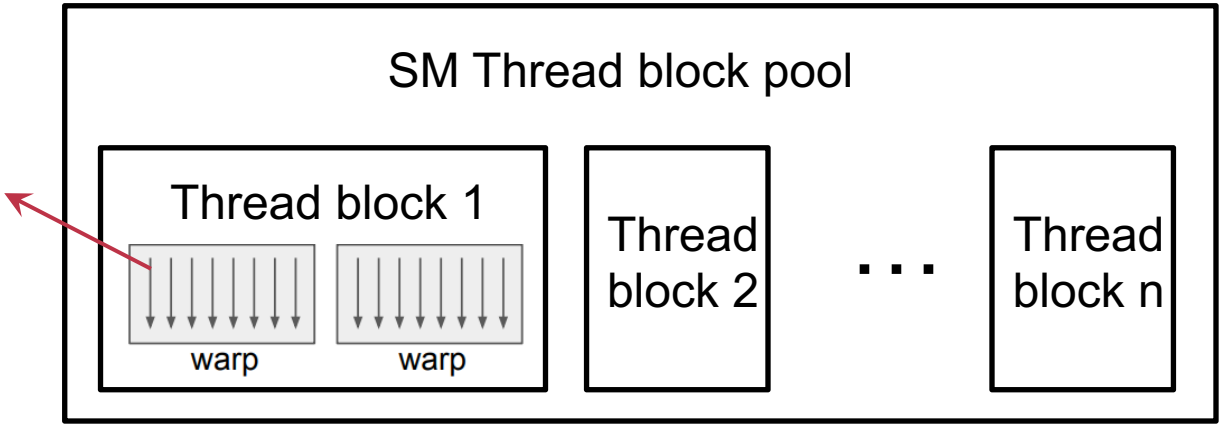


CUDA kernel execution

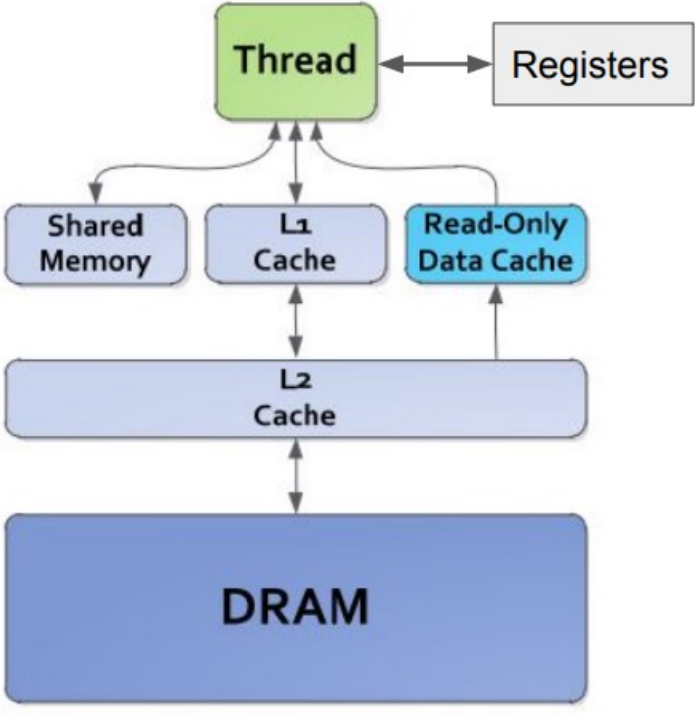
To execute a kernel

- The thread block is divided into 32-thread warps
- Each cycle, a warp scheduler selects one ready warps and dispatches the warps to CUDA cores to execute.





GPU memory access latency



Registers: R 0 cycle / R-after-W ~20 cycles

L1/texture cache: 92 cycles
Shared memory: 28 cycles
Constant L1 cache: 28 cycles

L2 cache: 200 cycles

DRAM: 350 cycles

(for Nvidia Maxwell architecture)

DRAM: HBM (global memory)

More on CUDA kernel execution: control flow

Every threads in a warp share the same program counter

```
100: ...
101: if (condition) {
102:     ...
103: } else {
104:     ...
105: }
```

More on CUDA kernel execution: control flow

Every threads in a warp share the same program counter

```
100: ...
101: if (condition) {
102:     ...
103: } else {
104:     ...
105: }
```

More on CUDA kernel execution: control flow

Every threads in a warp share the same program counter

```
100: ...
101: if (condition) {
102:     ...
103: } else {
104:     ...
105: }
```

More on CUDA kernel execution: control flow

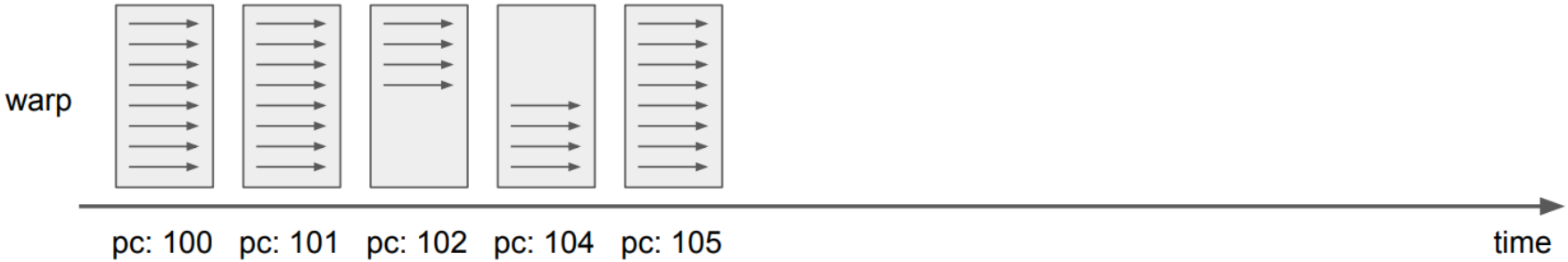
Every threads in a warp share the same program counter

```
100: ...
101: if (condition) {
102:     ...
103: } else {
104:     ...
105: }
```

More on CUDA kernel execution: control flow

Every threads in a warp share the same program counter

```
100: ...
101: if (condition) {
102:     ...
103: } else {
104:     ...
105: }
```



More on CUDA kernel execution: control flow

Every threads in a warp share the same program counter (PC)

- Though recent generation of GPU allows different PC
- But has limitations: e.g., each warp only has one decoder
- Question: why?

Consequently, programming CUDA efficiently should take branch predictions seriously

> *"Hence, it is valid to state that low control flow efficiency always indicates a performance issue. "**

*https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/branchstatistics.htm

CUDA program problem: dead lock

Can easily cause deadly lock if not carefully programmed

– “Generalized deadlock”: two threads preventing each other from execution

```
if (threadIdx.x == 0) {
    consume();
} else {
    produce();
}
```

CUDA kernel execution: why we need the block?

Block in CUDA is also called “Cooperative Thread Array”

#1. Block-level shared memory is efficient

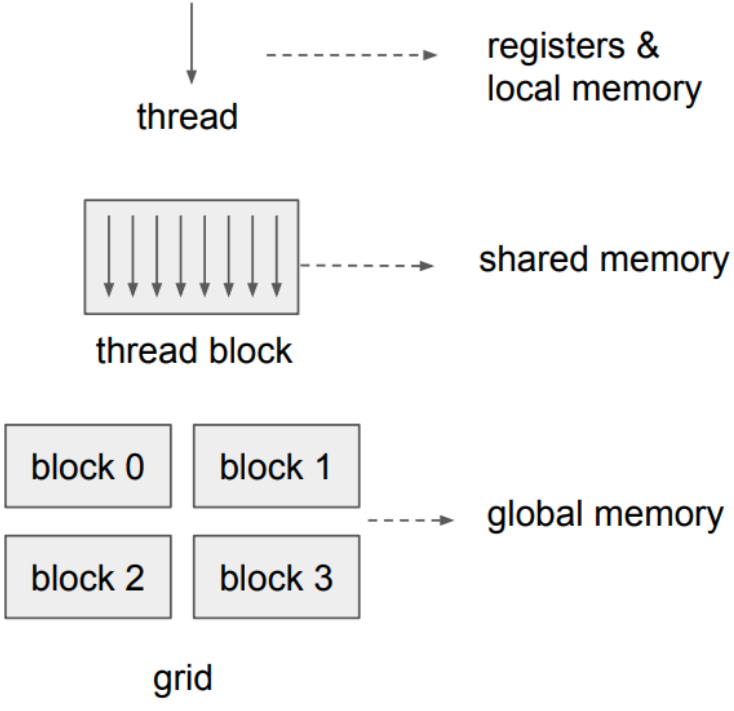
- E.g., `__shared__ a[N];`
- The same performance (latency, throughput) as L1 cache

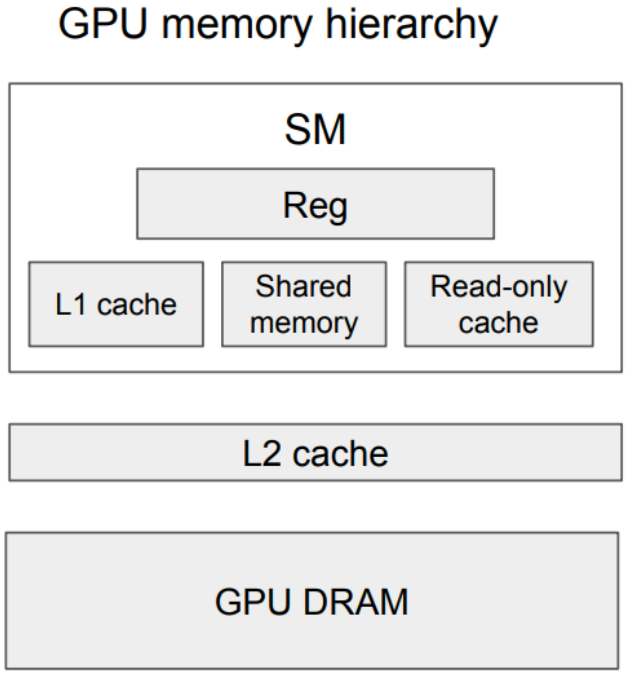
#2. Block-level synchronization barrier

- E.g., `__syncthreads();`
- Lightweight and fast
- Question: why fast?

More on CUDA kernel execution: memory access

Thread Hierarchy & Memory Hierarchy





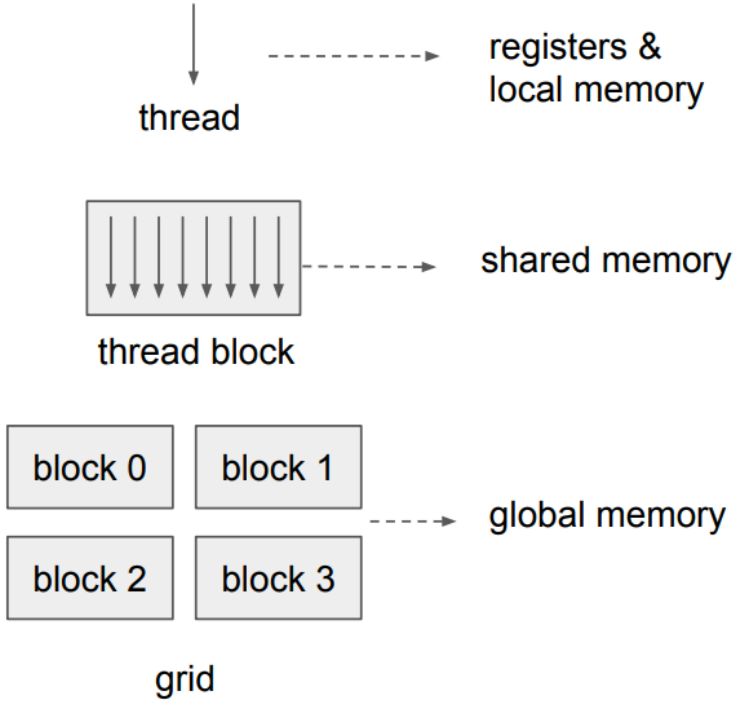
More on CUDA kernel execution: memory access

Thread Hierarchy & Memory Hierarchy

- Global memory accesses from a warp are performed by memory transactions

Memory transaction

- a read/write to a continuous global memory region
- memory transaction size: 32, 64, 128B
- multiple memory transactions from a warp are **serialized**
- Latency: e.g., 350 cycles



Example revisited: vector add

Compute vector sum

– C = A + B

```
Void vecAdd_cpu(const float* A, const float* B, float* C, int n) {
    for (int i = 0; i < n; ++i)
        C[i] = A[i] + B[i];
}
```

How to parallel it with GPU?

– Each thread only compute the sum of one entry

Example revisited: vector add

Compute vector sum

– C = A + B

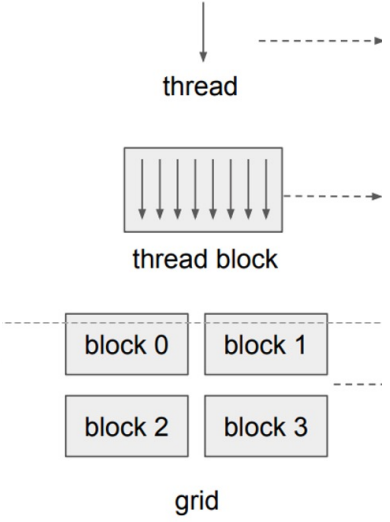
```
Void vecAdd_cpu(const float* A, const float* B, float* C, int n) {
    for (int i = 0; i < n; ++i)
        C[i] = A[i] + B[i];
}
```

```
__global__ void vecAddKernel(const float* A, const float* B, float* C, int n) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < n) {
        C[i] = A[i] + B[i];
    }
}
```

Example revisited: vector add

Compute vector sum

– C = A + B




```
Void vecAdd_cpu(const float* A, const float* B, float* C,
    for (int i = 0; i < n; ++i)
        C[i] = A[i] + B[i];
}
```

```
__global__ void vecAddKernel(const float* A, const float* B, float* C, int n) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < n) {
        C[i] = A[i] + B[i];
    }
}
```

Compute the global thread index

Example revisited: vector add

Compute vector sum

– C = A + B

```
Void vecAdd_cpu(const float* A, const float* B, float* C, int n) {
    for (int i = 0; i < n; ++i)
        C[i] = A[i] + B[i];
}
```

```
__global__ void vecAddKernel(const float* A, const float* B, float* C, int n) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < n) {
        C[i] = A[i] + B[i];
    }
}
```

Each thread performs one pair-wise addition

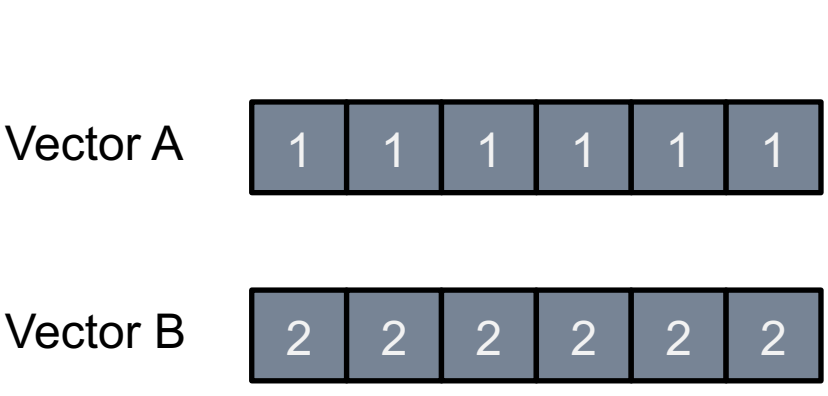
Memory layout: row-wise or column-wise?

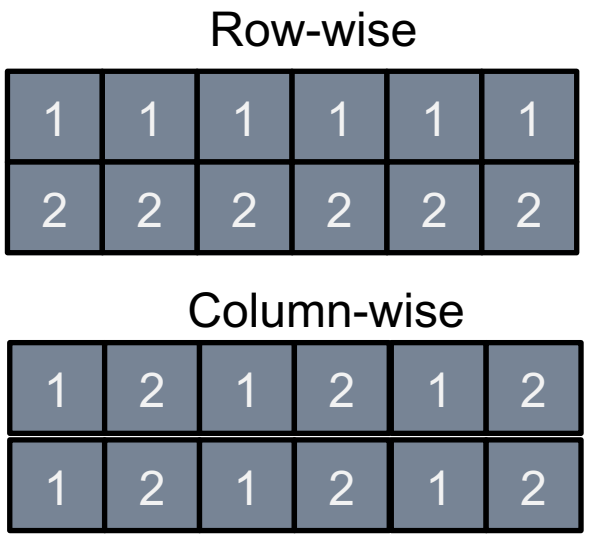
Compute vector sum

– C = A + B

Row-wise: store content of a vector continuously

Column-wise: store the content of a vector with other vectors



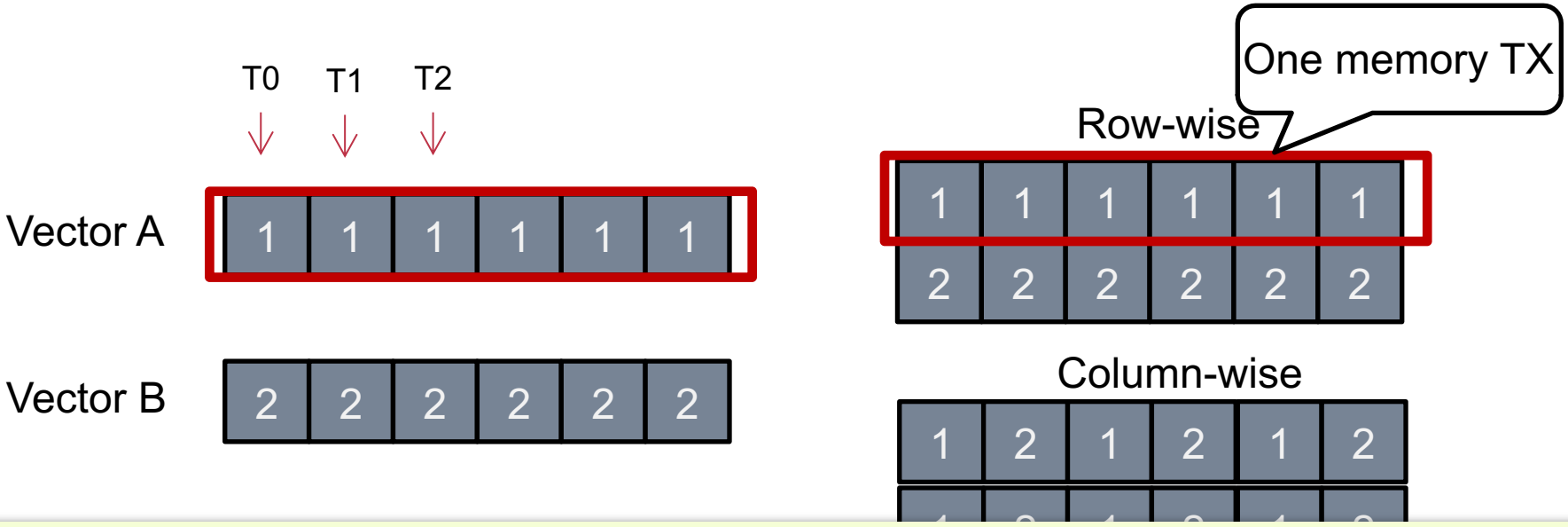


Row-wise causes extra memory stalls!

Compute vector sum (Assuming three working threads)

– C = A + B

Under row-wise storage, the warp must first fetch vector A & then B



Question: how many threads can work after one memory TX?

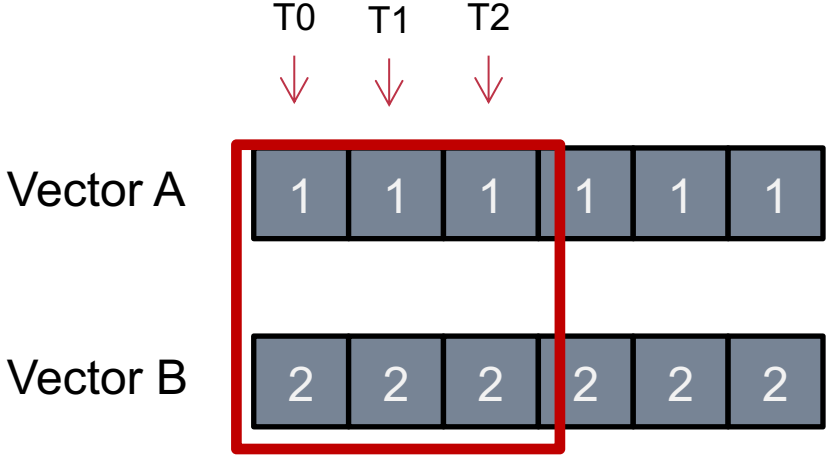
Row-wise causes extra memory stalls!

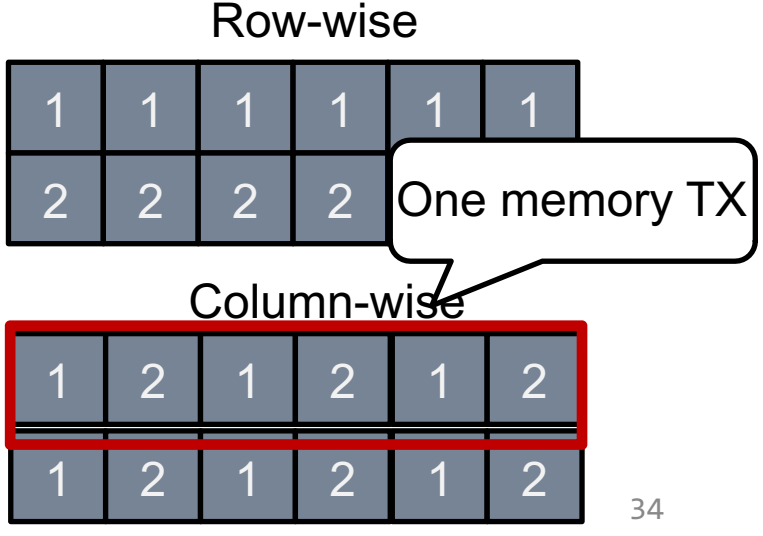
Compute vector sum (Assuming three working threads)

– C = A + B

Under column-wise storage

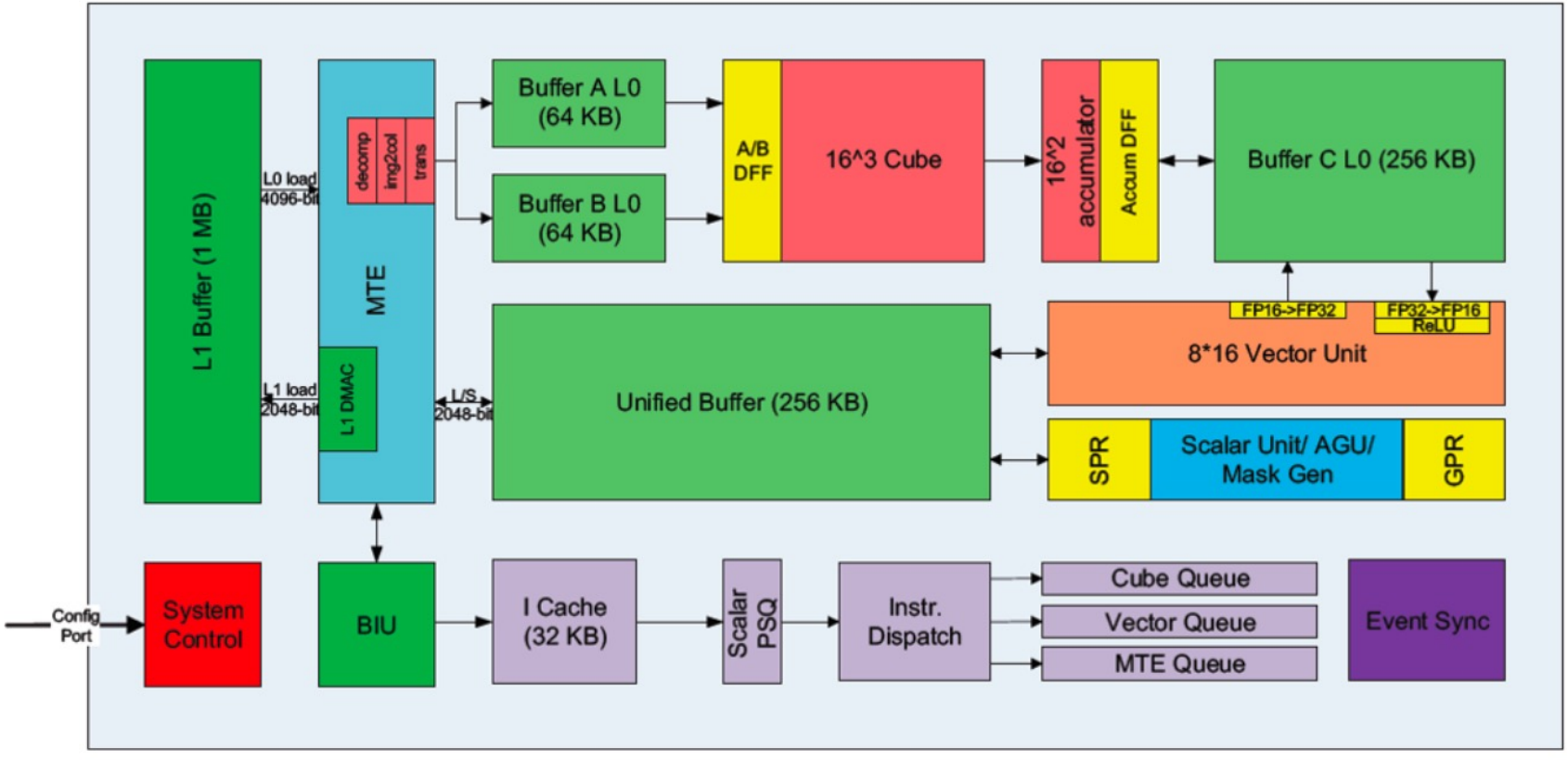
– The warp can immediately start the execution after one memory TX





Other accelerators

Ascend: Huawei's accelerators for AI



Source: Ascend: a Scalable and Unified Architecture for Ubiquitous Deep Neural Network Computing, HPCA'21

Ascend: Huawei's accelerators for AI

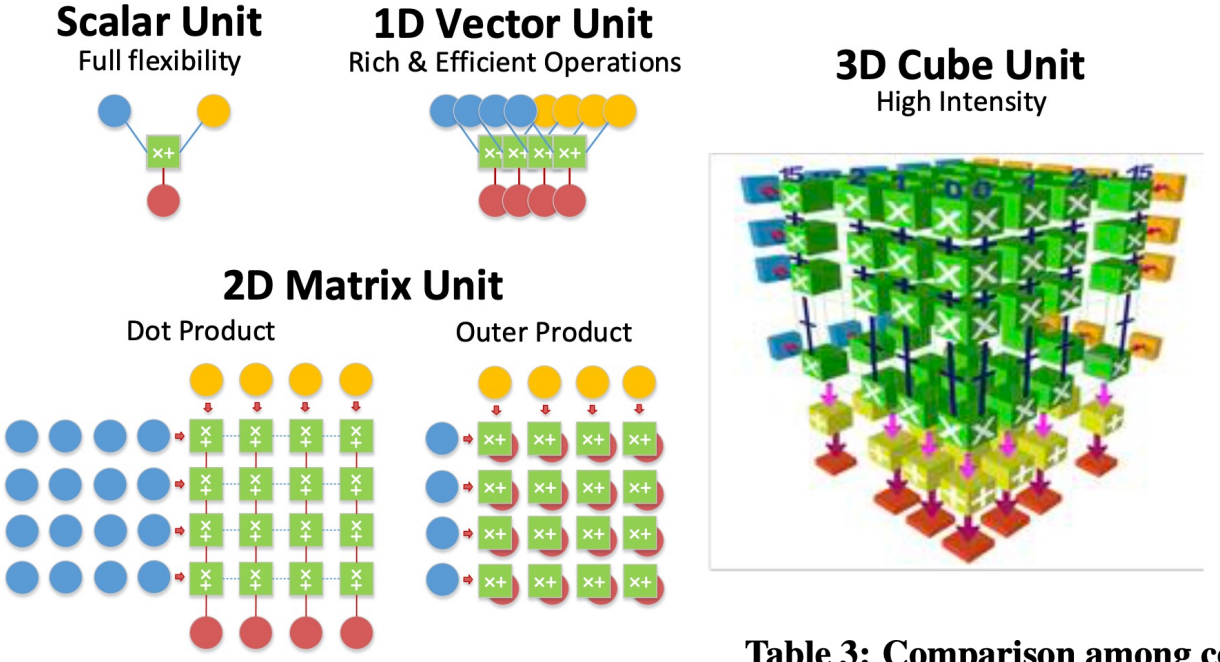


Table 2: Operations for each unit.

Unit Name	Typical Operations
Scalar	Control, Scalar Computation
Vector	Normalize, Activation, Format Transfer, CV Operators (RPN, etc.)
Cube	Convolution, FC, MatMul

Table 3: Comparison among computing units

Unit Name	**Scalar**	**Vector**	**Cube**
Performance (FLOPS)	2G	256G	8T
Power (W)	/	0.46	3.13
Area (mm^2, 7nm)	0.04	0.70	2.57
Perf./Power (TFLOPS/W)	/	0.56	2.56
Perf./Area (TFLOPS/mm^2)	0.05	0.36	3.11

Optimizing attention computation:

Analysis from a memory access perspective

Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com

Noam Shazeer* Google Brain noam@google.com

Niki Parmar* Google Research nikip@google.com

Jakob Uszkoreit* Google Research usz@google.com

Llion Jones* Google Research llion@google.com

Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu

Łukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin* ‡ illia.polosukhin@gmail.com

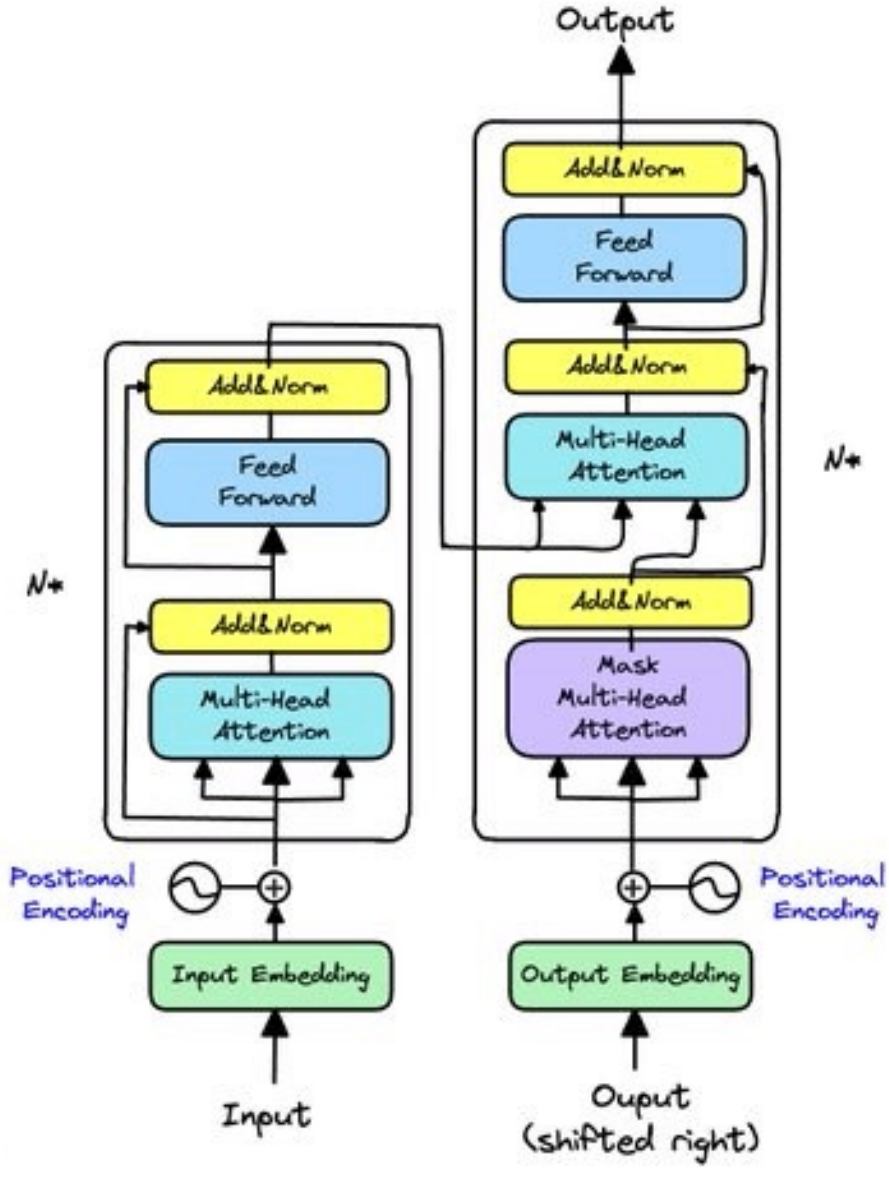
Abstract

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train. Our model achieves 28.4 BLEU on the WMT 2014 English-to-German translation task, improving over the existing best results, including ensembles, by over 2 BLEU. On the WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.8 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature. We show that the Transformer generalizes well to other tasks by applying it successfully to English constituency parsing both with large and limited training data.

*Equal contribution. Listing order is random. Jakob proposed replacing RNNs with self-attention and started the effort to evaluate this idea. Ashish, with Illia, designed and implemented the first Transformer models and has been crucially involved in every aspect of this work. Noam proposed scaled dot-product attention, multi-head attention and the parameter-free position representation and became the other person involved in nearly every detail. Niki designed, implemented, tuned and evaluated countless model variants in our original codebase and tensor2tensor. Llion also experimented with novel model variants, was responsible for our initial codebase, and efficient inference and visualizations. Lukasz and Aidan spent countless long days designing various parts of and implementing tensor2tensor, replacing our earlier codebase, greatly improving results and massively accelerating our research.

†Work performed while at Google Brain.

‡Work performed while at Google Research.



Attention basics: what does it do?

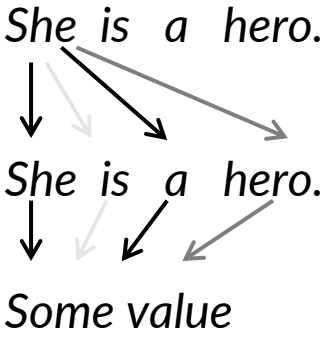
Goal: output a prediction considering the relationship between inputs

Input: *She is a hero.*

0 1 2 3

Output:

For i in N



Original paper: Attention is all you need, NIPS'17

Attention basics: example implementation

Case study: the naïve GPT-2 code (simplified from [source] in Pytorch)

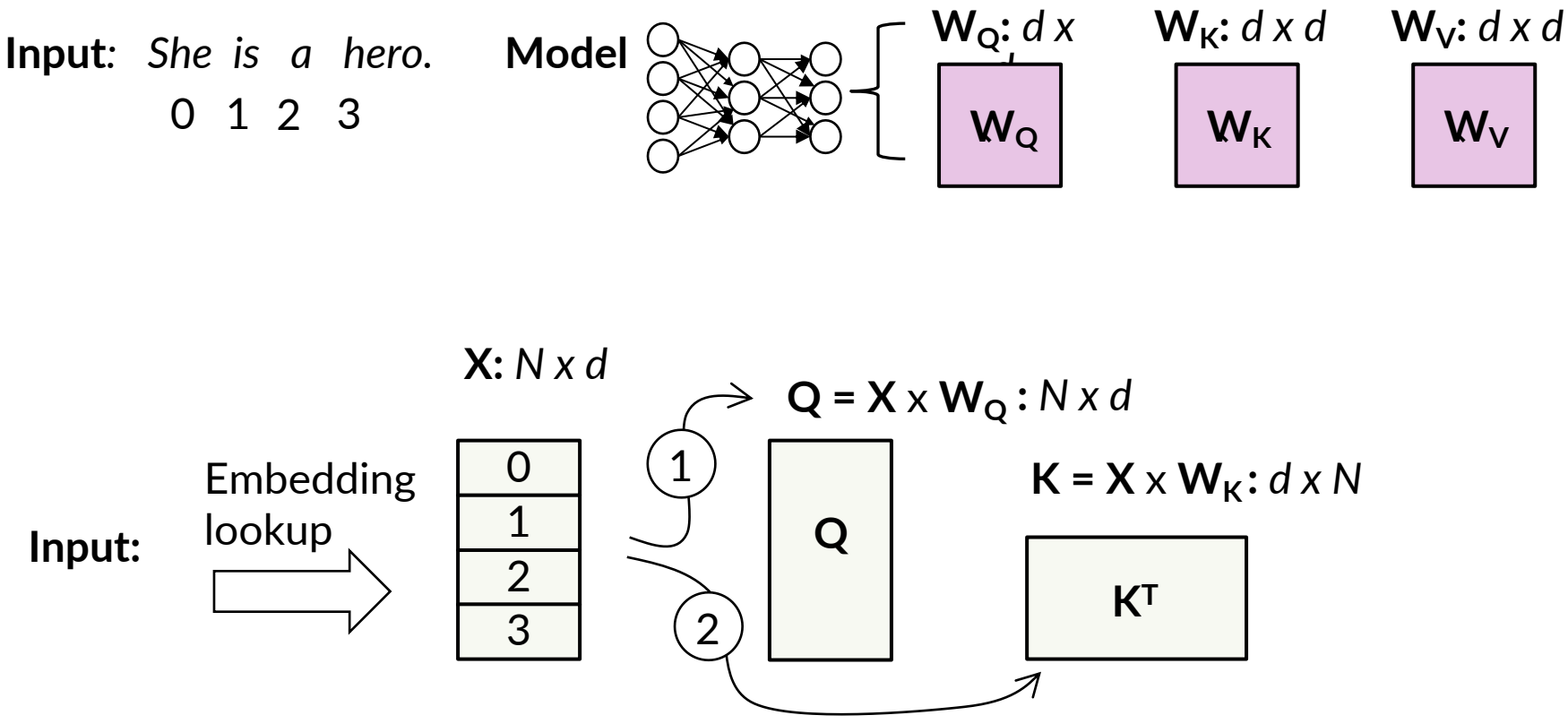
```
class Attention(nn.Module):
    def __init__(self, head_size):
        super().__init__()
        self.query = nn.Linear(n_embd, head_size,bias=False)
        self.key = nn.Linear(n_embd, head_size,bias=False)
        self.value = nn.Linear(n_embd, head_size,bias=False)
        self.register_buffer("tril",torch.tril(torch.ones(block_size, block_size)))

    def forward(self, x):
        B,N,D = x.shape  # Batch, Number of tokens, Dimension
        Q = self.query(x)   1
        K = self.key(x)     2

        weight = Q @ K.transpose(-2,-1)   3
        weight = weight.masked_fill(self.tril[:N,:N] == 0, float('-inf' ))   4 + 5
        return weight @ self.value(x)   6
```

Source: Reproducing GPT-2, https://www.youtube.com/watch?v=l8pRSuU81PU

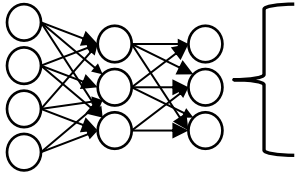
Attention basics: the computation in the model



Attention basics: the computation in the model

Input: *She is a hero.*
0 1 2 3

Model

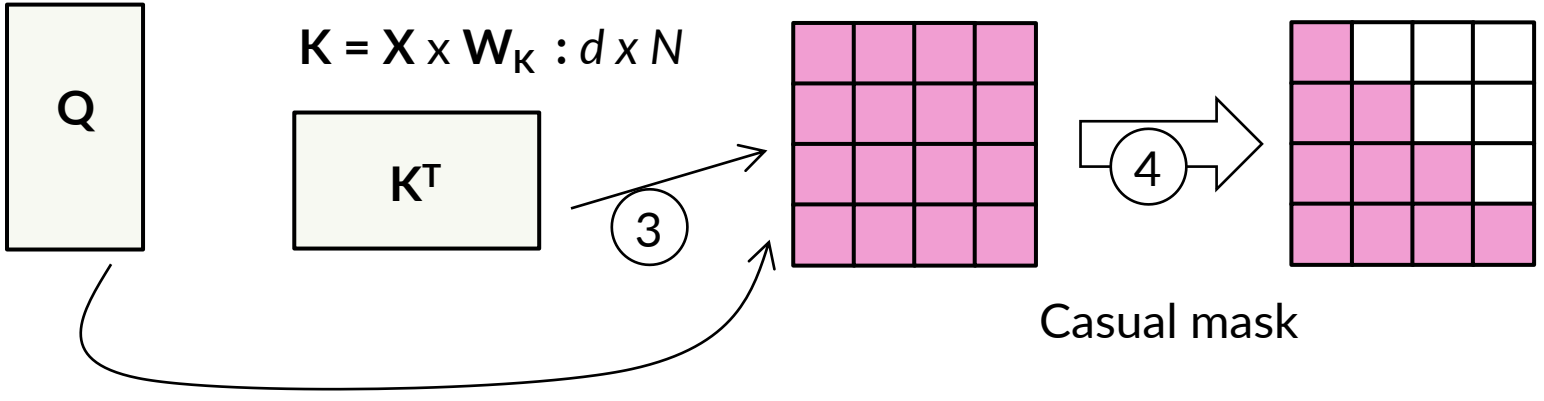


W_Q: *d x* W_K: *d x d* W_V: *d x d*

$Q = X \times W_Q : N \times d$

$K = X \times W_K : d \times N$

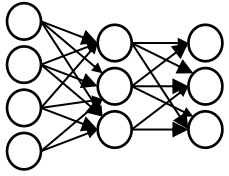
$Weight = Q \times K^T : N \times N$



Attention basics: the computation in the model

Input: *She is a hero.*
0 1 2 3

Model

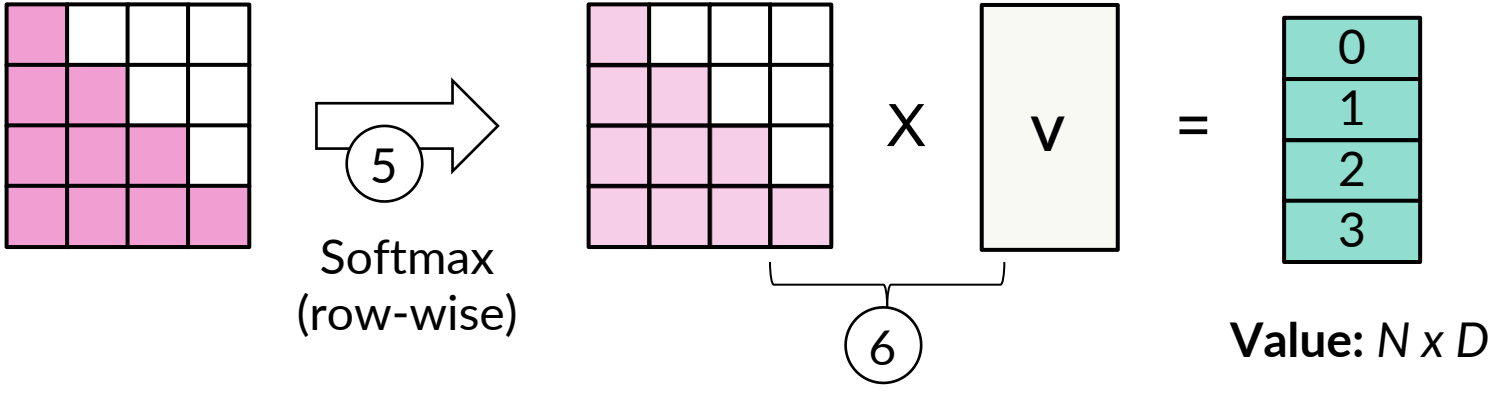


W_Q: *d x* W_K: *d x d* W_V: *d x d*

W_Q W_K W_V

$$s\left(x_i\right) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

$V = X \times W_v$: *N x d*



Question: how to optimize the attention operator?

Time: slow due to the complex interactions between different operators

- i.e., matrix multiplication and softmax

Recall: the time for a computation operator depends on two factors

- First, how many FLOP do the operator need?
- Second, how many memory access do the GPU need to load?

FLOP for the multiplication and softmax

- Really constant, e.g., $O(N^3)$ for two matrix multiplication in almost all implementations

Memory load

- Depends on how the operation is implemented

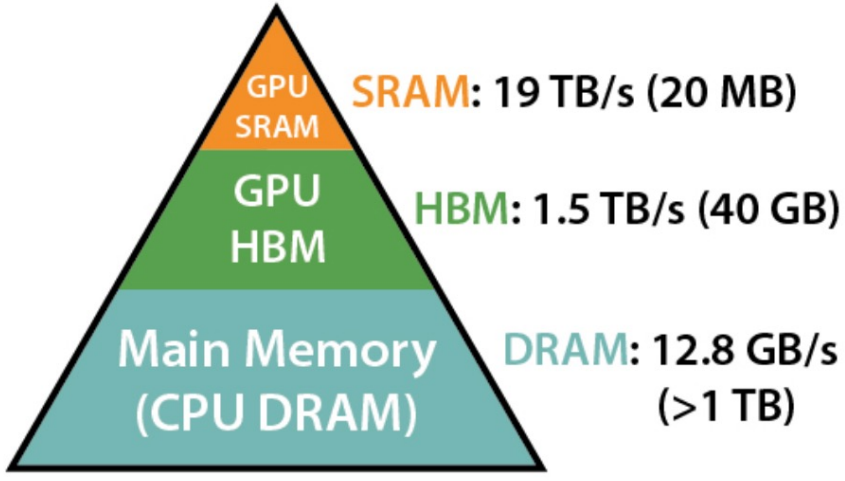
Memory hierarchy in modern GPUs

GPU also has memory hierarchy

- SRAM: e.g., registered files, ultra fast but small
- HBM: relative fast, sufficiently large

SRAM bandwidth is not the bottleneck

Goal: minimize the HBM load



Memory Hierarchy with Bandwidth & Memory Size

Source: https://github.com/Dao-AILab/flash-attention?tab=readme-ov-file

Question: what is the HBM load for basic attention?

Recap: each HBM load will load a block of data to the SRAM

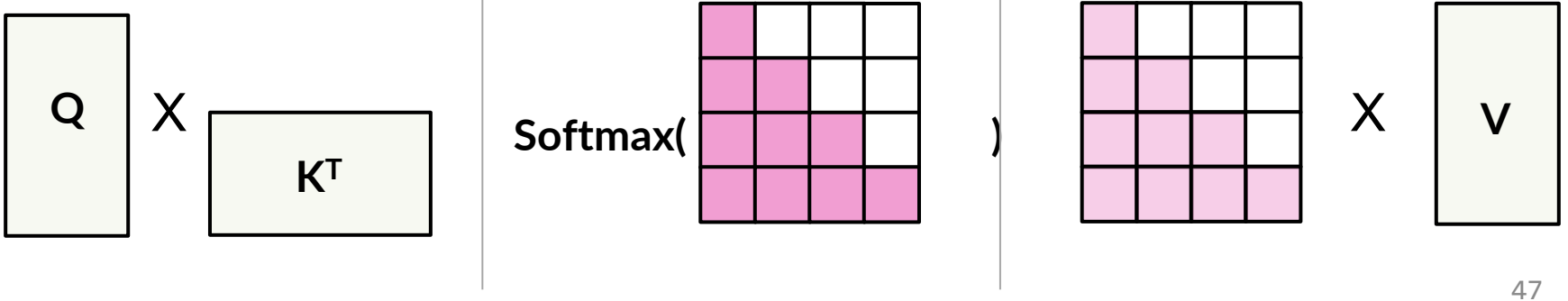
- Like load cacheline in the CPU

But, how many loads depends on how the operator is implemented

- So we should break down the operator

Key operators in the basic attention

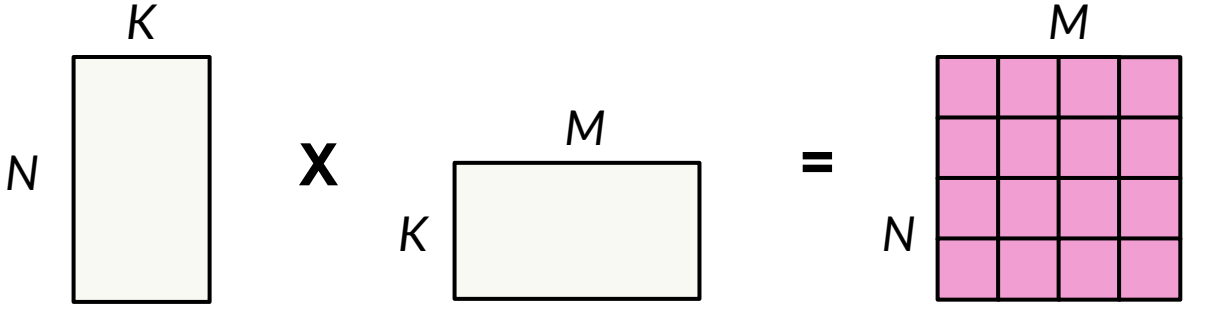
- **Matrix multiplications** & softmax



What is the memory loads of matrix multiplication?

A long story, actually ☺ (worth an at-least 45-minute lecture)

– We will cover three methods in this lecture: the naïve & an optimized one



Approach #1. naïve loop

```
for i in 0..N:
    for j in 0..M:
        for k in 0..K:
            C[i][j] += A[i][k] * B[k][j]
```

Question:

– How many block accesses do the above code need?

Assumptions

Matrixes are stored in row-wise

- Each memory access load at least **BB bytes**

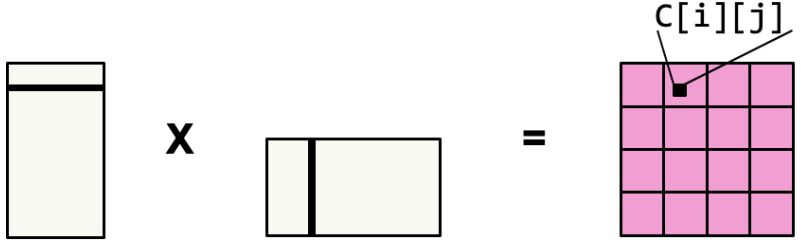
HBM loads in row order (will be relaxed later)

- i.e., a load load_HBM(A[i][j]) will load A[i][j:j + B] into the cache

Analysis: naïve loop (w/o considering the cache)

```
for i in 0..N:
    for j in 0..M:
        for k in 0..K:
            HBM_read(A[i][k])
            HBM_read(B[k][j]
            C[i][j] += A[i][k] * B[k][j]
        HBM_write(C[i][j])
```

Overall: needs $O(NMK * BB)$ HBM reads!



Analysis: naïve loop (w/ considering the cache)

```
for i in 0..N:
    for j in 0..M:
        for k in 0..K:
            if k % BB == 0:
                HBM_read(A[i][k]) # will read A[i][k:k+BB]
            HBM_read(B[k][j])
            C[i][j] += A[i][k] * B[k][j]
     if j % BB == 0:
        HBM_write(C[i][j]) # will write C[i][j:j+BB]
```

Overall: still needs $O(NMK * BB)$ HBM bytes !

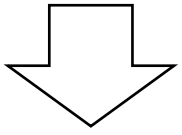
Question: can we improve the performance?

Analysis: change the loop order

```
for i in 0..N:
     for j in 0..M:
          for k in 0..K:
               C[i][j] += A[i][k] * B[k][j]
```

Observation:

- The add is **commutative**



```
for i in 0..N:
    for k in 0..K:
        for j in 0..M:
            C[i][j] += A[i][k] * B[k][j]
```

Analysis: change the loop order (w/o cache)

```
for i in 0..N:
    for k in 0..K:
        HBM_read(A[i][k])
        for j in 0..M:
            HBM_read(B[k][j])
            HBM_read(C[i][j])
            C[i][j] += A[i][k] * B[k][j]
            HBM_write(C[i][j])
```

Overall: still needs $O(NMK)$ HBM accesses, does not change?

– Not so far yet, we need to think about the cache.

Analysis: change the loop order (w cache)

```
for i in 0..N:
    for k in 0..K:
        HBM_read(A[i][k])
        for j in 0..M:
            if j % BB == 0:
                HBM_read(B[k][j])
                HBM_read(C[i][j])
            C[i][j] += A[i][k] * B[k][j]
        if j % BB == 0:
            HBM_write(C[i][j])
```

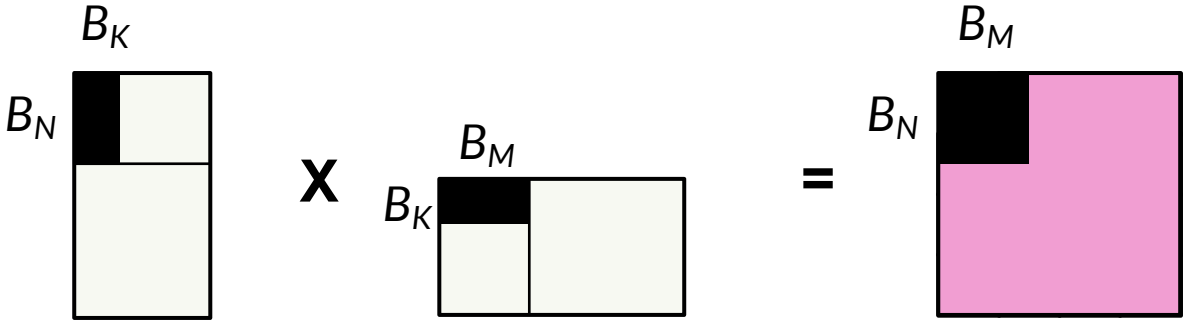
Result: the above code will roughly give $O(NMK)$, a great improvement!

Approach #3: Tiling

Question: can we improve HBM reads w/o adding more writes?

– Yes. We can compute the end matrix in a block-version

```
for i in 0..N/B_N.step(B_N):
    for j in 0..M/B_M.step(B_M):
        for k in 0..K/B_K.step(B_K):
            C[i:i+ B_N][j:j+B_M] += A[i:i+B_N][k:k+B_K]
                                 * B[k+B_K][j:j+B_M]
```



Analysis: tiling (w/o cache)

```
for i in 0..N/B_N.step(B_N):
    for j in 0..M/B_M.step(B_M):
        for k in 0..K/B_K.step(B_K):
            load(A[i:i+B_N][k:k+B_K])
            load(B[k+B_K][j:j+B_M])
            C[i:i+B_N][j:j+B_M] += A[i:i+B_N][k:k+B_K]
                                 * B[k+B_K][j:j+B_M]
        write(C[i:i+B_N][j:j+B_M])
```

The total number is roughly: $O(\frac{N}{BN} \times \frac{M}{BM} + 2 \times \frac{N}{BN} \times \frac{M}{BM} \times \frac{K}{BK})$

What about considering the cache? A little complicated

– We can make some simplification: assume the block fit the cache

Analysis: tiling (w cache)

```
for i in 0..N/B_N.step(B_N):
    for j in 0..M/B_M.step(B_M):
        for k in 0..K/B_K.step(B_K):
            load(A[i:i+B_N][k:k+B_K])
            load(B[k+B_K][j:j+B_M])
            C[i:i+B_N][j:j+B_M] += A[i:i+B_N][k:k+B_K]
                                 * B[k+B_K][j:j+B_M]
        write(C[i:i+B_N][j:j+B_M])
```

Under the assumption that three blocks can reside in the cache

- i.e., by tuning the number of the block sizes, e.g., B_N

The total number is still roughly: $O(\frac{N}{BN} \times \frac{M}{BM} + 2 \times \frac{N}{BN} \times \frac{M}{BM} \times \frac{K}{BK})$

HBM accesses (w/o cache): #number vs. #bytes in tiling

```
for i in 0..N/B_N.step(B_N):
    for j in 0..M/B_M.step(B_M):
        for k in 0..K/B_K.step(B_K):
            load(A[i:i+B_N][k:k+B_K])
            load(B[k+B_K][j:j+B_M])
            C[i:i+B_N][j:j+B_M] += A[i:i+B_N][k:k+B_K]
                                 * B[k+B_K][j:j+B_M]
        write(C[i:i+B_N][j:j+B_M])
```

The total number is roughly: $O(\frac{N}{BN} \times \frac{M}{BM} + 2 \times \frac{N}{BN} \times \frac{M}{BM} \times \frac{K}{BK})$

The total bytes read is: $O(\frac{NMK}{BM} + \frac{NMK}{BN})$

The total bytes written is: $O(\mathbf{N} \times \mathbf{M})$

Tiling: change the order of i,j,k

```
for i in 0..N/B_N.step(B_N):
    for j in 0..M/B_M.step(B_M):
        for k in 0..K/B_K.step(B_K):
            load(A[i:i+B_N][k:k+B_K])
            load(B[k+B_K][j:j+B_M])
            C[i:i+B_N][j:j+B_M] += A[i:i+B_N][k:k+B_K]
                                * B[k+B_K][j:j+B_M]
        write(C[i:i+B_N][j:j+B_M])
```

Recall: in previous example, we iterate over i,j,k order

- The C's block is reused upon computation
- But our current matrix C is small, we can let A to be re-used

Short quiz: can tiling change the I,j,k order?

A little more complex in analyzing the memory accessed

Further question: how to determine the order (& block sizes?)

- Solution: auto tune !

Question: what is the memory accessed for the basic attention?

Depend on how the operator is implemented

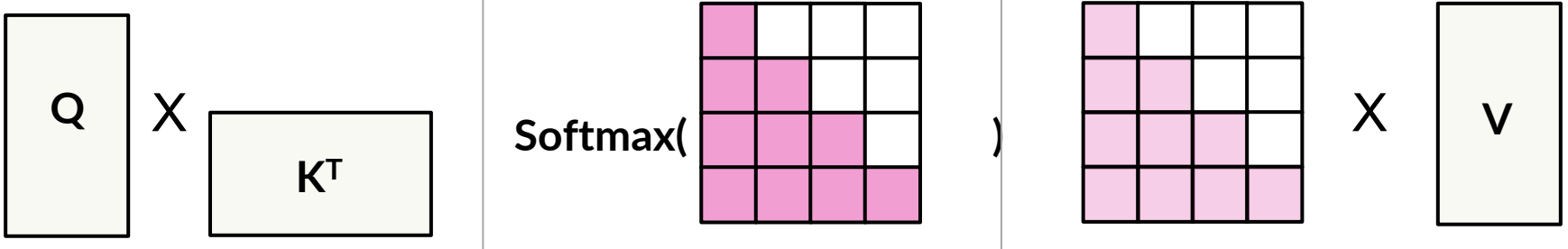
- So we should break down the operator

Key operators in the basic attention

- Matrix multiplications & softmax

We can calculate these using a divide and conquer approach

Why not considering the computation? Few space for acceleration



Analysis: The Q X K^T

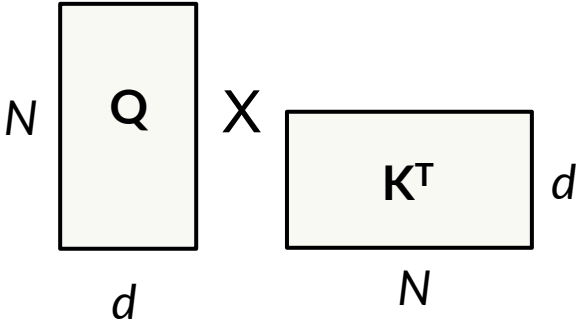
Assuming we are using tiling using the i,j,k iteration order

- The standard matrix multiplication acceleration method

Recall: for matrix multiplication of (N,K) X (K,M), we have

- The total bytes read is: $O(\frac{NMK}{B\mathrm{M}} + \frac{NMK}{BN})$
- The total bytes written is: $O(\mathbf{N}\times\mathbf{M})$

We have N = N, K = d, M = N => O(N^2) + O($\frac{N^2d}{BN} + \frac{N^2d}{BN}$)



Considering BN can be chosen to be larger than d in the attention (why?), we can simplify the calculation as:

O(N^2)

Analysis: Softmax

$$s\left(x_i\right) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

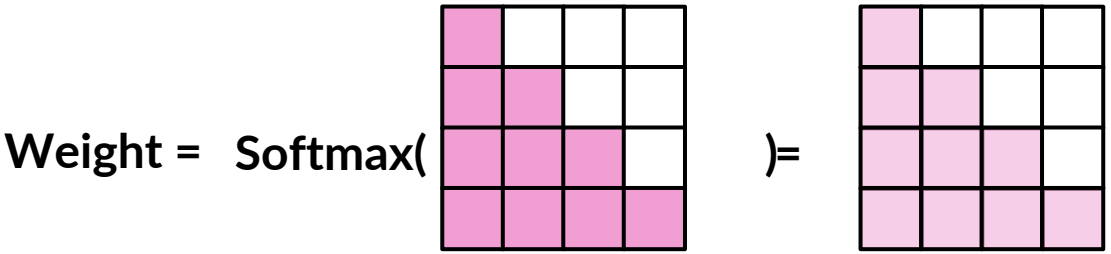
The softmax in transformer compute in a casually row-wise fashion

- Row-wise
- Casual: the x_i is 0 for $i > row\_id$

Softmax computation is typically decomposed into two steps (for stability)

- First, calculate the maximum of each row (x_{max})
- Then , let $x_i = xi - x_{max}$, then calculate the softmax

Question: how many HBM accesses? Not so hard to see: O(N^2)

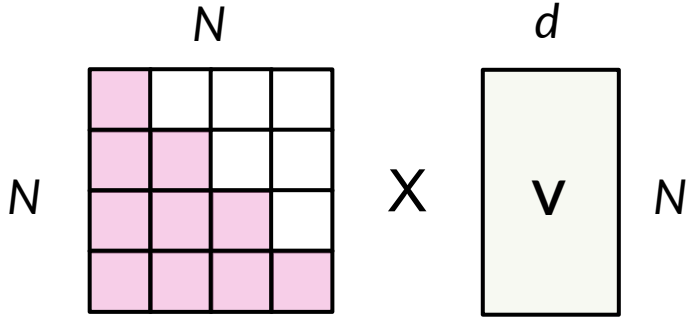


Analysis: Weight X V (still using tiling with i,j,k)

Recall: for matrix multiplication of (N,K) X (K,M), we have

- The total bytes read is: $O(\frac{NMK}{B\mathrm{M}} + \frac{NMK}{BN})$
- The total bytes written is: $O(\mathbf{N}\times\mathbf{M})$

We have N = N, K = N, M = d => O(Nd) + O($\frac{N^2d}{BN} + \frac{N^2d}{Bd}$) ~= O($N^2$)



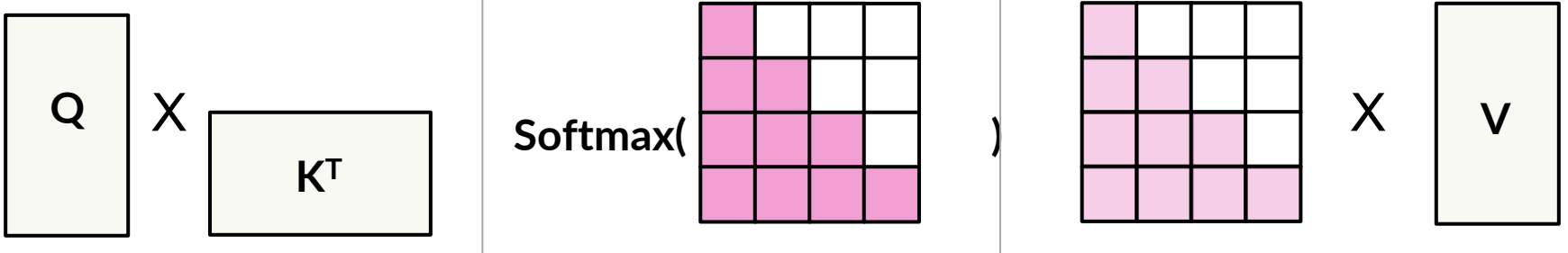
Summary: memory accessed in attention

Key operators in the basic attention

- Matrix multiplications & softmax

Our current analysis shows that the accessed in bytes are $O(N^2)$

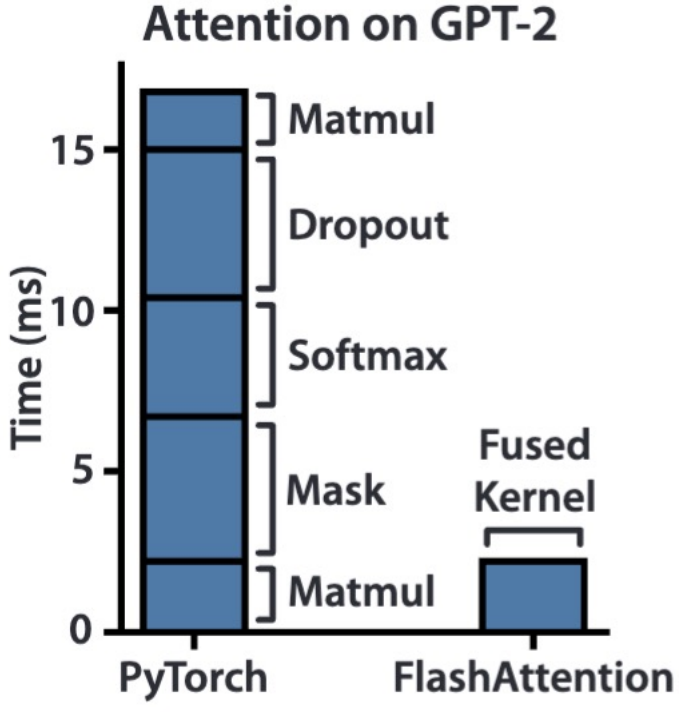
- And we have little optimization space if the operators are decoupled
- i.e., run each operator one-by-one
- Why? Because the softmax operator has little optimization space!



Idea: fusing the operators

Fuse: execute multiple operators together

– Can dramatically reduce the memory accessed of the overall computation!

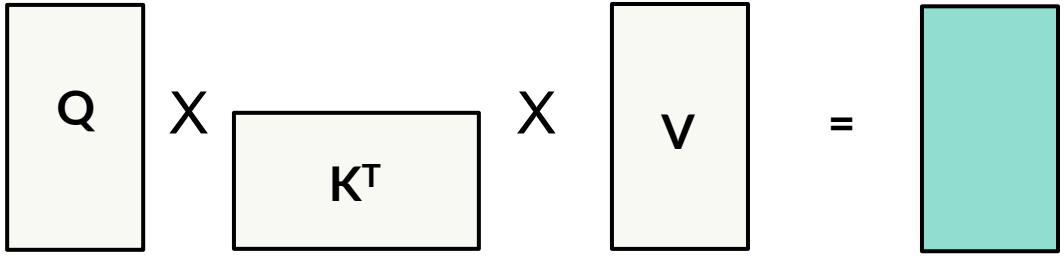


Source: FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness, NIPS'22

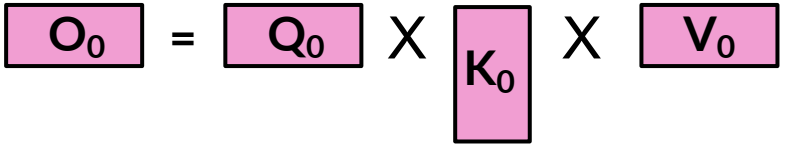
Step #1: fuse QK^TV (let's forget the softmax for now)

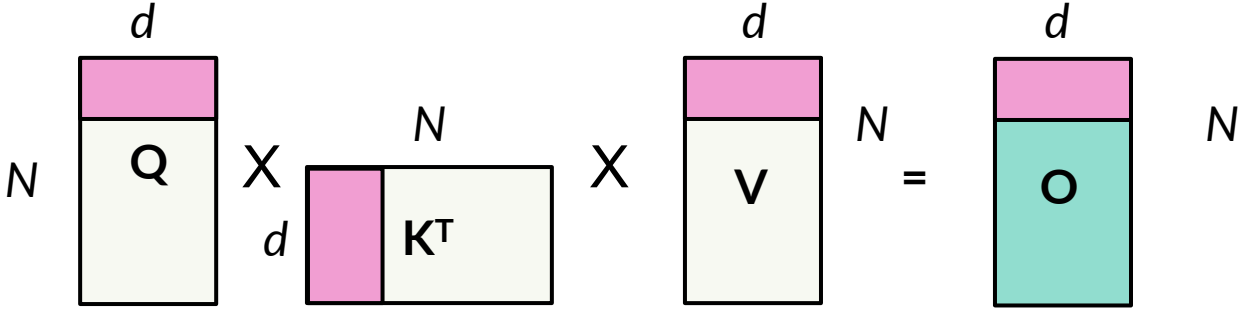
It turns out that we can have a very efficient way of doing multiple matrixes

- Because the same value can be reused (not needed to be loaded to the HBM and then read it)



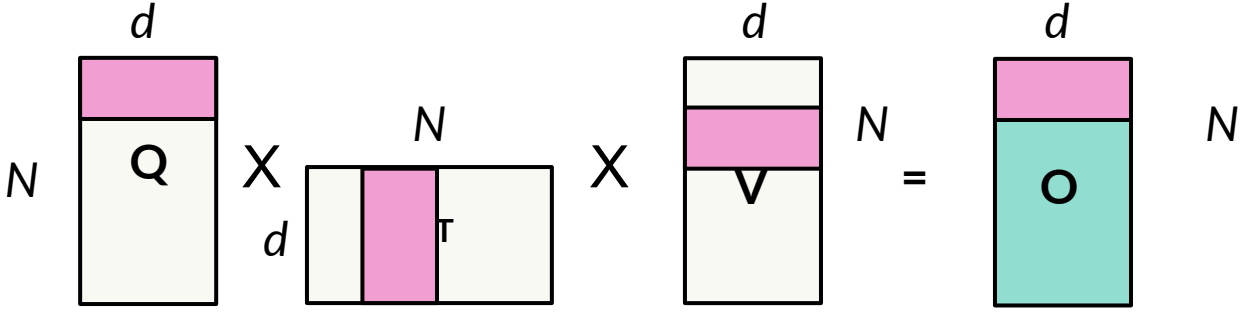
Example of reusing





Example of reusing

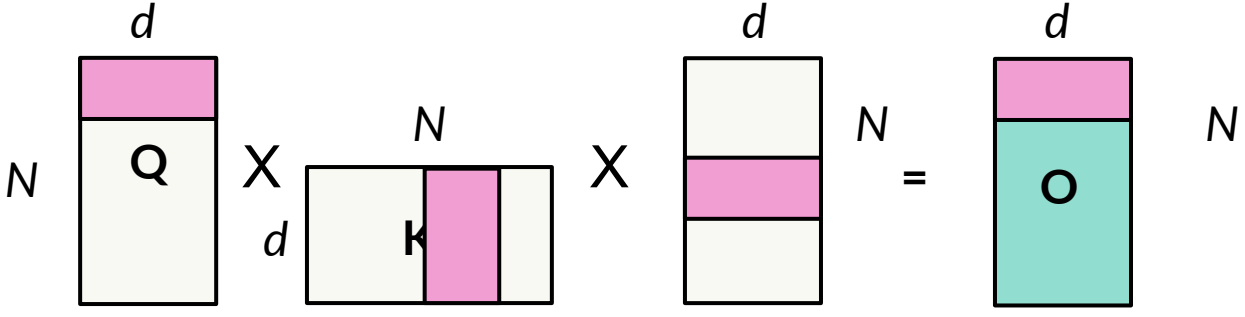
$$O_0 = Q_0 \times K_0 \times V_0 + Q_0 \times K_1 \times V_1$$



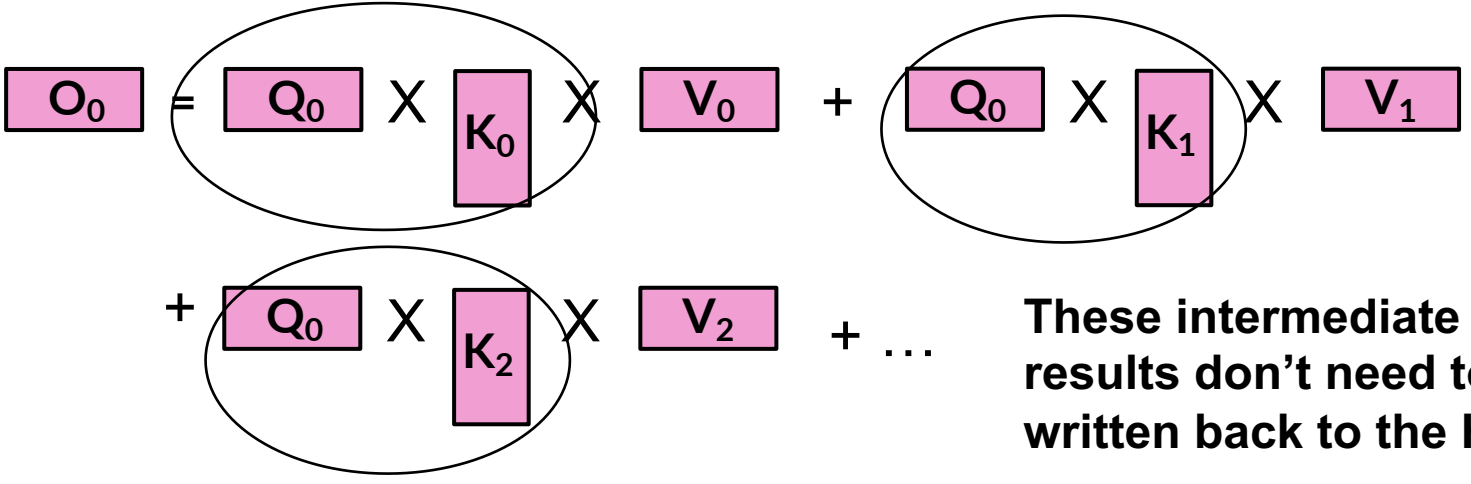
Example of reusing

$$O_0 = Q_0 \times K_0 \times V_0 + Q_0 \times K_1 \times V_1$$

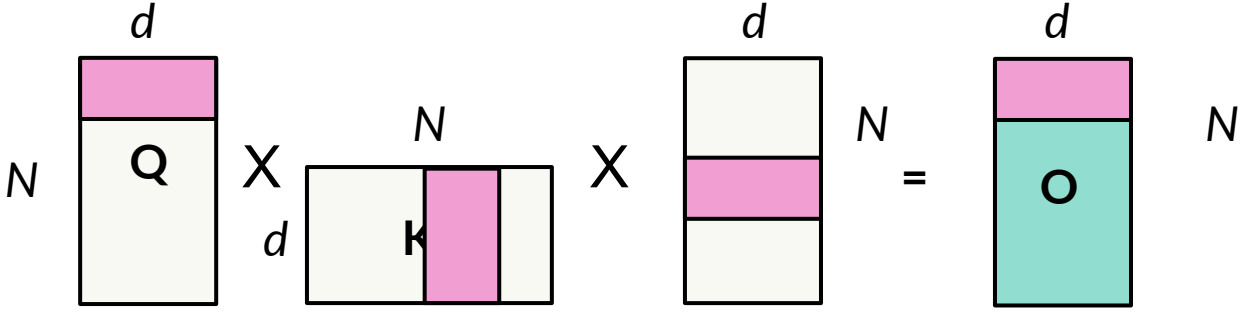
$$+ Q_0 \times K_2 \times V_2 + \dots$$



Why reducing the #HBM accessed?



These intermediate results don't need to be written back to the HBM!



Fused QK^TV: the code (under tiling)

```
for i in 0..N/B_Q.step(B_Q):  ## Iterate over the Q
    O[i:i + B_Q , 0:d] = 0
    for j in 0..N/B_K.step(B_K):
         O[i,0:d] += Q[i:i+B_Q,0:d] * K^T[0:d,j:j+ B_K ]
                     * V[j:j+ B_K, 0:d]
```

Analysis: fuse QK^TV: the code (under tiling)

```
for i in 0..N/B_Q.step(B_Q):  ## Iterate over the Q
    load(Q[i:i + B_Q , 0:d])
    write(O[i:i + B_Q , 0:d])
    O[i:i + B_Q , 0:d] = 0
    for j in 0..N/B_K.step(B_K):
         load(K^T[0:d,j:j+ B_K ])
         load(V[j:j+ B_K, 0:d])
         O[i:i+B_Q ,0:d] += Q[i:i+B_Q,0:d] * K^T[0:d,j:j+ B_K ]
                         * V[j:j+ B_K, 0:d]
    write(O[i:i + B_Q , 0:d])
```

Assumption: d is a small value (e.g., 768)

Analysis: fuse QK^TV: the code (under tiling)

Q: $N/B_Q * B_Q * d = N * d$

O is the same as Q: $N * d$

K: $N/B_Q * N/B_K * d * B_K = \frac{N^2 d}{B_Q}$

V is the same as K

Oveall: O($\frac{N^2 d}{B_Q}$**)**

Assuming the cache size is M, then we have $B_Q * d \sim= M$**, then we have**

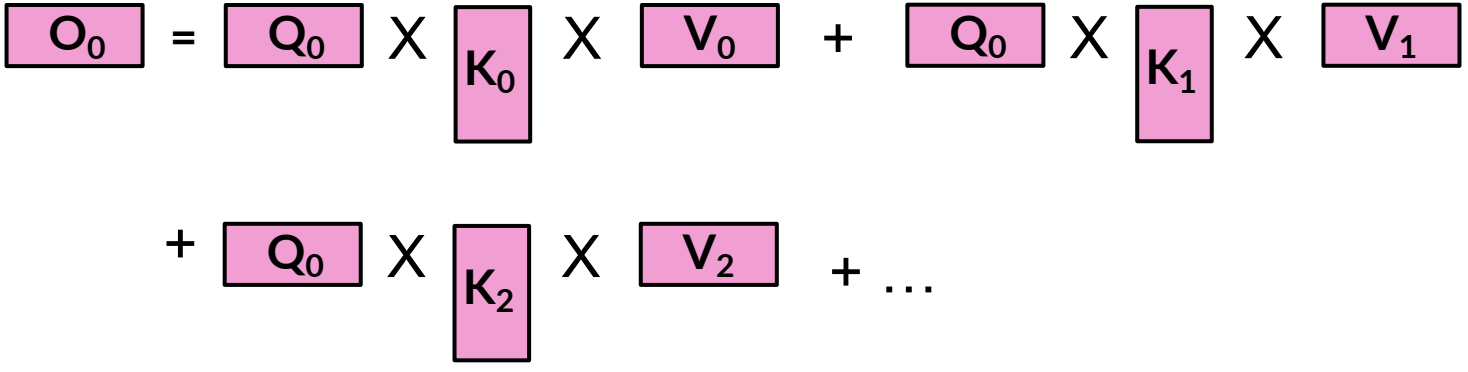
- Overall: O($\frac{N^2 d^2}{M}$)
- This is consistent with the original Flashattention[1] paper

[1] FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness, NIPS'22

Step #2: fused QK^TV + Softmax

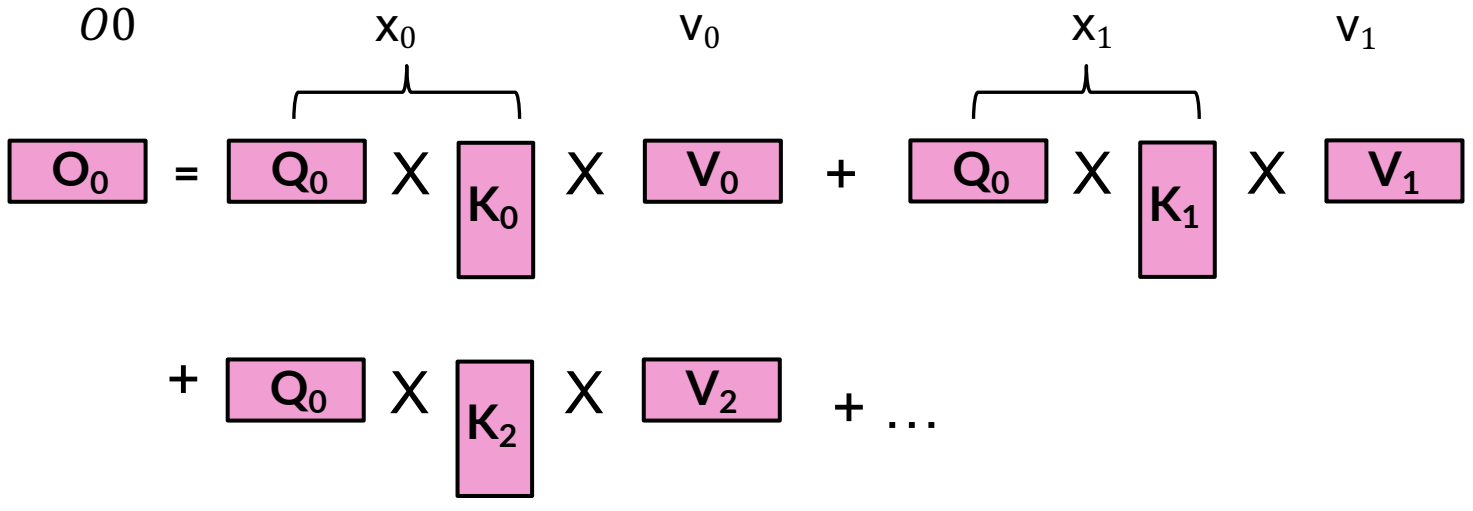
Question: can we apply the same mythology?

– Yes, though needs a little extension



Step #2: fused QK^TV + Softmax

To better illustrate the algorithm, we made some extension



Step #2: fused QK^TV + Softmax

$$s\left(x_i\right) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

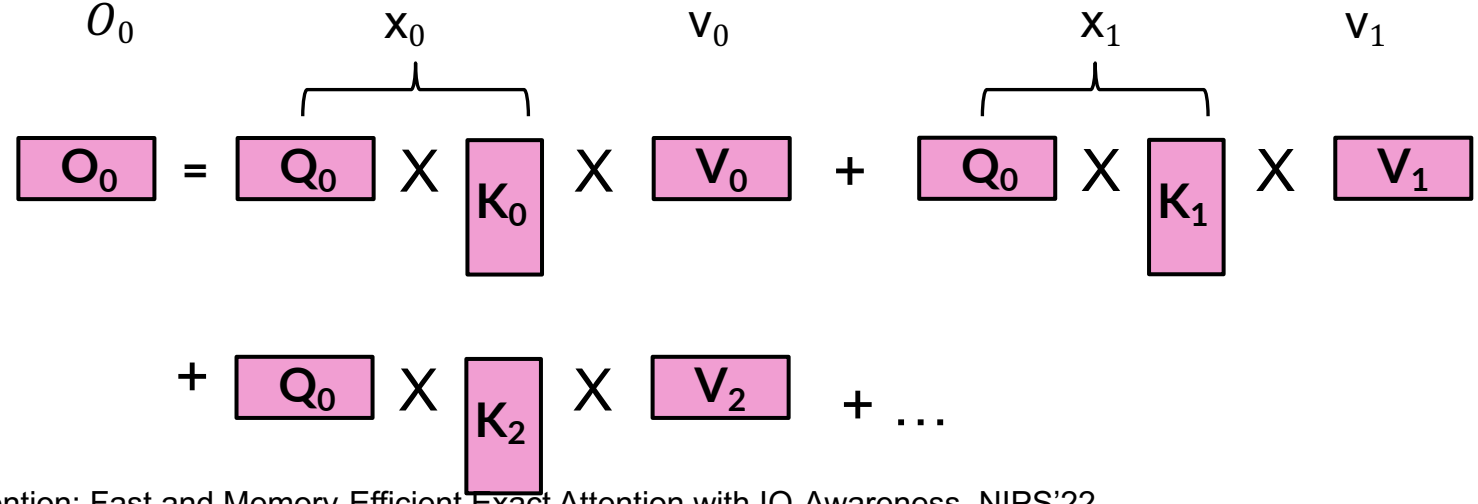
Softmax

To better illustrate the algorithm, we made some extension

- Fused QK^TV: O0 = X0 * V0 + X1 * V1
- Fused QK^TV + Softmax: O0 = **Softmax(X0, X)** * V0 + **Softmax(X1, X)** * V1

Question: how to make the softmax operators combined together?

- Check [1] fort the **online softmax** if you are interested!



[1] FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness, NIPS'22

More to be optimized

FLASHATTENTION-2:
Faster Attention with Better Parallelism and Work Partitioning

FlashAttention-3:
Fast and Accurate Attention with Asynchrony and Low-precision

Jay Shah*[1], Ganesh Bikshandi*[1], Ying Zhang[2], Vijay Thakkar[3,4], Pradeep Ramani[3], and Tri Dao[5,6]
[1]Colfax Research [2]Meta [3]NVIDIA [4]Georgia Tech [5]Princeton University [6]Together AI

FlashAttention-4震撼来袭，原生支持Blackwell GPU，英伟达的护城河更深了？

机器之心
人工智能等 2 个话题下的优秀答主

+ 关注

收录于 · 机器之心 ›

122 人赞同了该文章 ›

机器之心报道，编辑：Panda

More: the future trend

Nvidia Roadmap							
	2022	2023	2024	2025		2026	2027
Chip and Package Level							
	Hopper		**Blackwell**			**Rubin**	
Accelerator	H100 (SXM)	H200	B200/ GB200	GB300 (Ultra)	B300 (single die, B300A)	VR200	VR300 (Ultra)
GPU TDP (W)	700	700	700/1200	1,400	600	1,800	3,600
Foundry Node	4N		4NP			N3P (3NP)	
Logic Die Configuration	1 x Reticle Sized GPU		2 x Reticle Sized GPU			2 x Reticle Sized GPU, 2x I/O chiplet	4 x Reticle Sized GPU, 2x I/O chiplet
FP4 PFLOPs - Dense (per Package)	4*		10	15	4.6	50	100
HBM	80GB HBM3	141GB HBM3E	192GB HBM3E	288GB HBM3E	144GB HBM3E	288GB HBM4	1024GB HBM4E
HBM Stacks	5	6	8		4	8	16
HBM Bandwidth	3.35TB/s	4.8TB/s	8TB/s		4TB/s	13TB/s	32TB/s
Packaging	CoWoS-S		CoWoS-L			CoWoS-L	
SerDes speed (Gb/s uni-di)	112G		224G			224G	224G
Nvidia CPU	Grace					Vera	

More: the future trend

		FLOPS (norm.)	BW		FLOPS/BW
Hopper	H100	4	80	**0.05**	5
Hopper	H200	4	141	**0.0283688**	2.8368794
Blackwell	B200/GB200	10	192	**0.0520833**	5.2083333
Blackwell	GB300 (Ultra	15	288	**0.0520833**	5.2083333
Blackwell	B300 (single	4.6	144	**0.0319444**	3.1944444
Rubin	VR200	50	288	**0.1736111**	17.361111
Rubin	VR300 (Ultra	100	1024	**0.0976563**	9.765625

HBM is challenging to scale (on a single chip)