
CSIT6000V

Advanced AI Computing and Systems

· 2026 FALL ·

Tutorial 3 : CUDA Programming II

TA: Zhuo CHEN

zchenkv@connect.ust.hk



DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING

- GEMM Operator
 - Tiled GEMM
- Softmax Operator
 - Stable Softmax
 - Online Softmax
- Attention Operator
 - Basic Attention Operator
 - Flash Attention Operator

Review: Matrix Multiplication

$$Q = \begin{bmatrix} q_{11} & q_{12} & \cdots & q_{1d_k} \\ q_{21} & q_{22} & \cdots & q_{2d_k} \\ \vdots & \vdots & \ddots & \vdots \\ q_{L1} & q_{L2} & \cdots & q_{Ld_k} \end{bmatrix}, \quad K^\top = \begin{bmatrix} k_{11} & k_{21} & \cdots & k_{L1} \\ k_{12} & k_{22} & \cdots & k_{L2} \\ \vdots & \vdots & \ddots & \vdots \\ k_{1d_k} & k_{2d_k} & \cdots & k_{Ld_k} \end{bmatrix} \in \mathbb{R}^{d_k \times L}$$

$$S = QK^\top = \begin{bmatrix} \sum_{r=1}^{d_k} q_{1r} k_{1r} & \sum_{r=1}^{d_k} q_{1r} k_{2r} & \cdots & \sum_{r=1}^{d_k} q_{1r} k_{Lr} \\ \sum_{r=1}^{d_k} q_{2r} k_{1r} & \sum_{r=1}^{d_k} q_{2r} k_{2r} & \cdots & \sum_{r=1}^{d_k} q_{2r} k_{Lr} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{r=1}^{d_k} q_{Lr} k_{1r} & \sum_{r=1}^{d_k} q_{Lr} k_{2r} & \cdots & \sum_{r=1}^{d_k} q_{Lr} k_{Lr} \end{bmatrix} \in \mathbb{R}^{L \times L}$$

$$S_{ij} = Q_{i,:} K_{j,:}^\top = [q_{i1} \quad q_{i2} \quad \cdots \quad q_{id_k}] \begin{bmatrix} k_{j1} \\ k_{j2} \\ \vdots \\ k_{jd_k} \end{bmatrix} = \sum_{r=1}^{d_k} q_{ir} k_{jr} \in \mathbb{R}$$

Matrix Multiplication

$$(i, j) \mapsto S_{ij} = \sum_{r=1}^{d_k} Q_{ir} K_{jr}$$

```
// Naive CUDA GEMM: S = QK^T
// Each CUDA thread computes one S[row][col]

__global__ void gemm(float *Q, float *K, float *S,
                    int L, int dk)
{
    // Which output element does this thread compute?
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;

    // Check whether (row, col) is valid
    if (row < L && col < L)
    {
        float sum = 0.0f;

        // Row "row" of Q x row "col" of K
        // (= column "col" of K^T)
        for (int k = 0; k < dk; k++)
        {
            sum += Q[row * dk + k]
                  * K[col * dk + k];
        }

        // Store one output element S[row][col]
        S[row * L + col] = sum;
    }
}
```

Review : Matrix Operation

```
#include <stdio.h>
#include <cuda_runtime.h>

// GPU code
__global__ void matrixAdd(int *A, int *B, int *C, int rows, int cols)
{
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    if (row < rows && col < cols){
        int index = row * cols + col;
        C[index] = A[index] + B[index];
    }
}

int main()
{
    const int rows = 4, cols = 4, N = rows * cols;
    // 1. Prepare data on the CPU
    int A[N] = { 1, 2, 3, 4, 5, 6, 7, 8,
                9,10,11,12,13,14,15,16 };
    int B[N] = {10,20,30,40,50,60,70,80,
                90,100,110,120,130,140,150,160};
    int C[N] = {0};
    int *d_A, *d_B, *d_C;
    int bytes = N * sizeof(int);

    // 2. Allocate memory on the GPU
    cudaMalloc((void **)&d_A, bytes);
    cudaMalloc((void **)&d_B, bytes);
    cudaMalloc((void **)&d_C, bytes);

    // 3. Copy input data from CPU to GPU
    cudaMemcpy(d_A, A, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, B, bytes, cudaMemcpyHostToDevice);

    // 4. Launch the CUDA kernel
    dim3 threadsPerBlock(2, 2);
    dim3 blocks((cols + threadsPerBlock.x - 1) / threadsPerBlock.x,
                (rows + threadsPerBlock.y - 1) / threadsPerBlock.y);
    matrixAdd<<<blocks, threadsPerBlock>>>(d_A, d_B, d_C, rows, cols);

    // 5. Wait for the GPU to finish
    cudaDeviceSynchronize();

    // 6. Copy results from GPU to CPU
    cudaMemcpy(C, d_C, bytes, cudaMemcpyDeviceToHost);

    // Print results
    printf("C =\n");
    for (int row = 0; row < rows; row++){
        for (int col = 0; col < cols; col++) printf("%5d ", C[row * cols + col]);
        printf("\n");
    }

    // 7. Free GPU memory
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
    return 0;
}
```

```
#include <stdio.h>
#include <cuda_runtime.h>

// GPU code
__global__ void matrixMul(int *A, int *B, int *C, int rows, int cols)
{
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    if (row < rows && col < cols) {
        int index = row * cols + col;
        int sum = 0;
        for (int k = 0; k < cols; k++)
            sum += A[row * cols + k] * B[k * cols + col];
        C[index] = sum;
    }
}

int main()
{
    const int rows = 4, cols = 4, N = rows * cols;
    // 1. Prepare data on the CPU
    int A[N] = { 1, 2, 3, 4, 5, 6, 7, 8,
                9,10,11,12,13,14,15,16 };
    int B[N] = {10,20,30,40,50,60,70,80,
                90,100,110,120,130,140,150,160};
    int C[N] = {0};
    int *d_A, *d_B, *d_C;
    int bytes = N * sizeof(int);

    // 2. Allocate memory on the GPU
    cudaMalloc((void **)&d_A, bytes);
    cudaMalloc((void **)&d_B, bytes);
    cudaMalloc((void **)&d_C, bytes);

    // 3. Copy input data from CPU to GPU
    cudaMemcpy(d_A, A, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, B, bytes, cudaMemcpyHostToDevice);

    // 4. Launch the CUDA kernel
    dim3 threadsPerBlock(2, 2);
    dim3 blocks((cols + threadsPerBlock.x - 1) / threadsPerBlock.x,
                (rows + threadsPerBlock.y - 1) / threadsPerBlock.y);
    matrixMul<<<blocks, threadsPerBlock>>>(d_A, d_B, d_C, rows, cols);

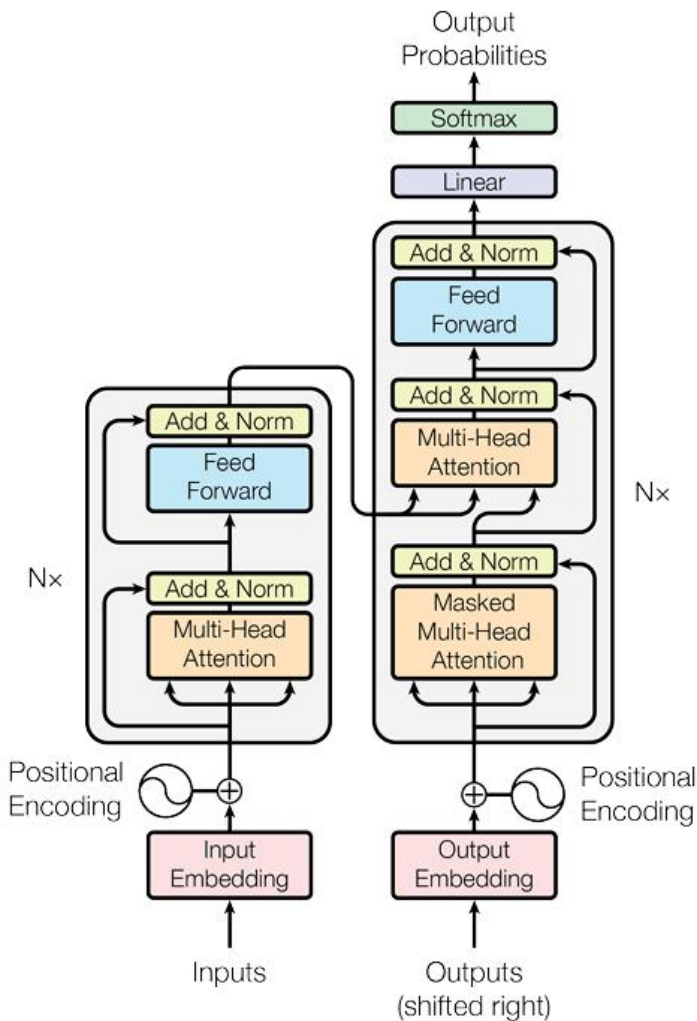
    // 5. Wait for the GPU to finish
    cudaDeviceSynchronize();

    // 6. Copy results from GPU to CPU
    cudaMemcpy(C, d_C, bytes, cudaMemcpyDeviceToHost);

    // Print results
    printf("C =\n");
    for (int row = 0; row < rows; row++){
        for (int col = 0; col < cols; col++) printf("%5d ", C[row * cols + col]);
        printf("\n");
    }

    // 7. Free GPU memory
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
    return 0;
}
```

Recondier attention



$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

2 GEMM

$$S = QK^T \text{ (gemm 1)}$$

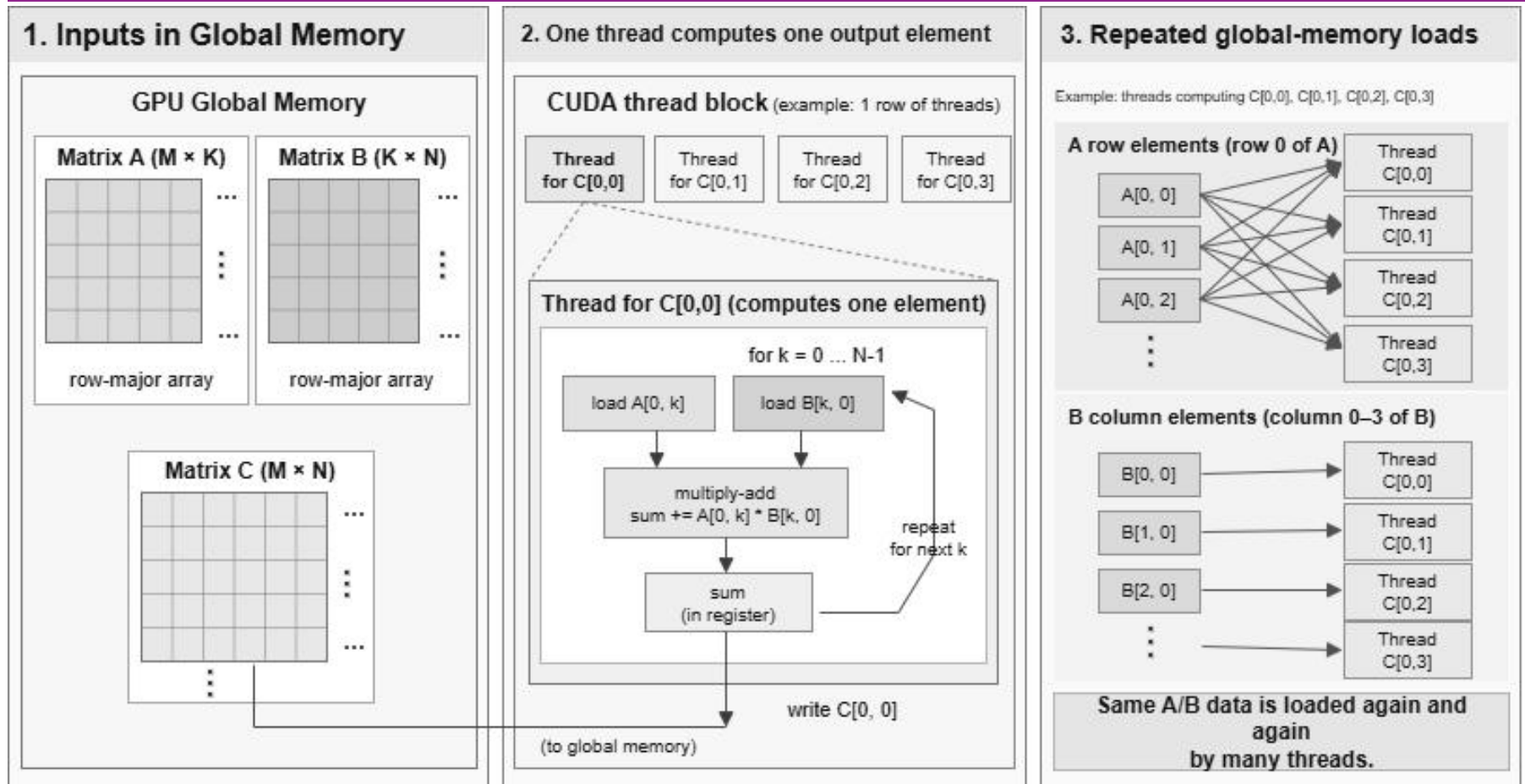
$$O = PV \text{ (gemm2)}$$

1 Softmax

$$P = softmax\left(\frac{S}{\sqrt{d_k}}\right)$$

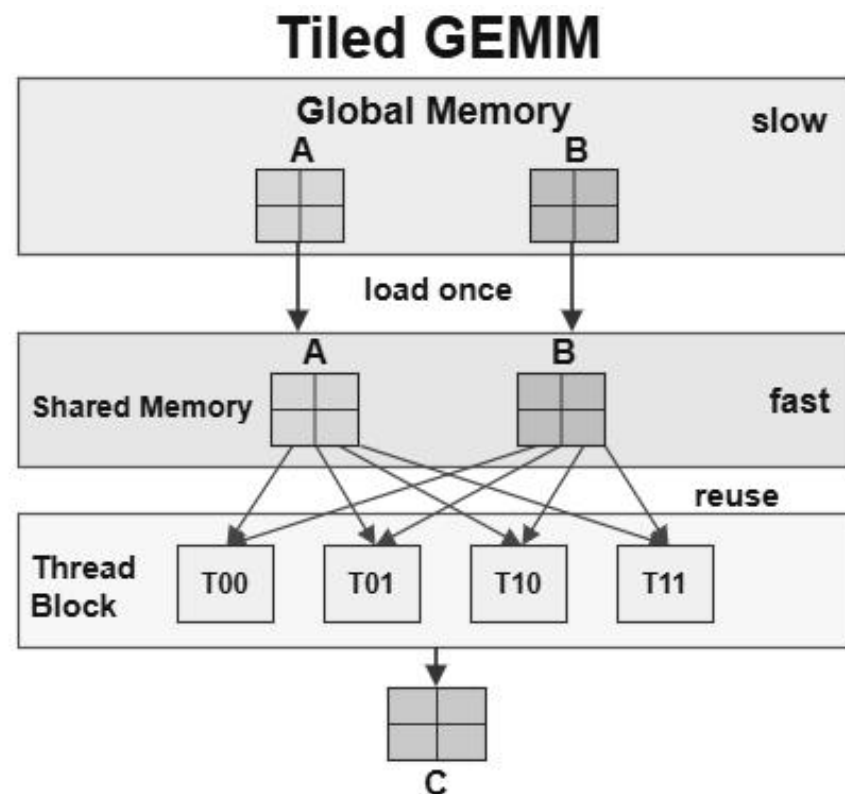
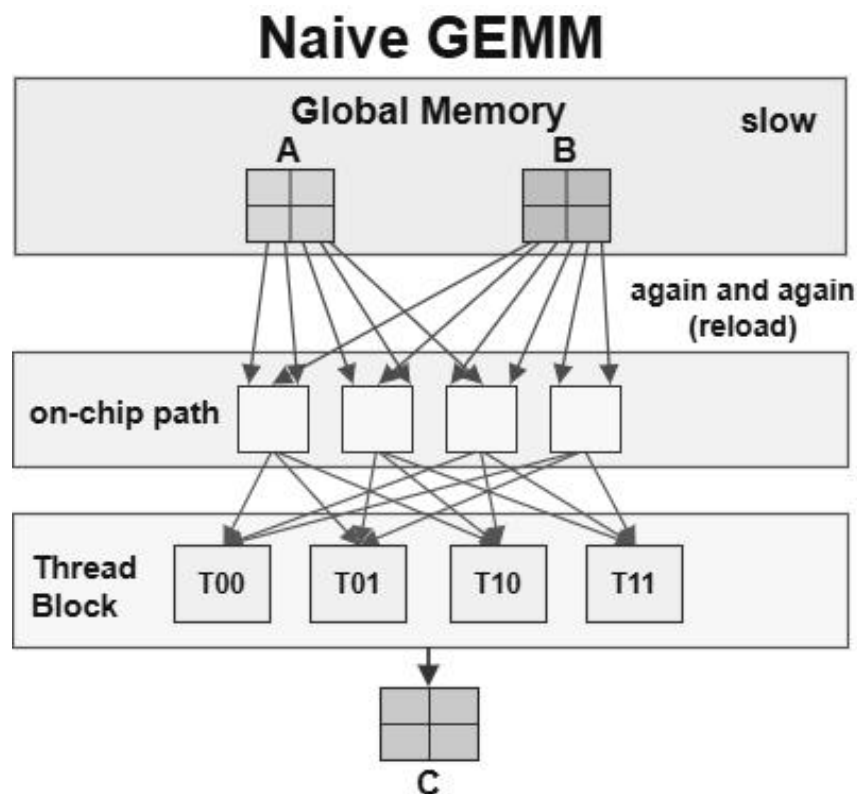
$$= softmax\left(\frac{QK^T}{\sqrt{d_k}}\right) \text{ (softmax1)}$$

Naive GEMM data path



- Each thread independently loads A and B from global memory
- The same input values are repeatedly fetched by different threads
- Computation is cheap; global-memory traffic becomes the bottleneck

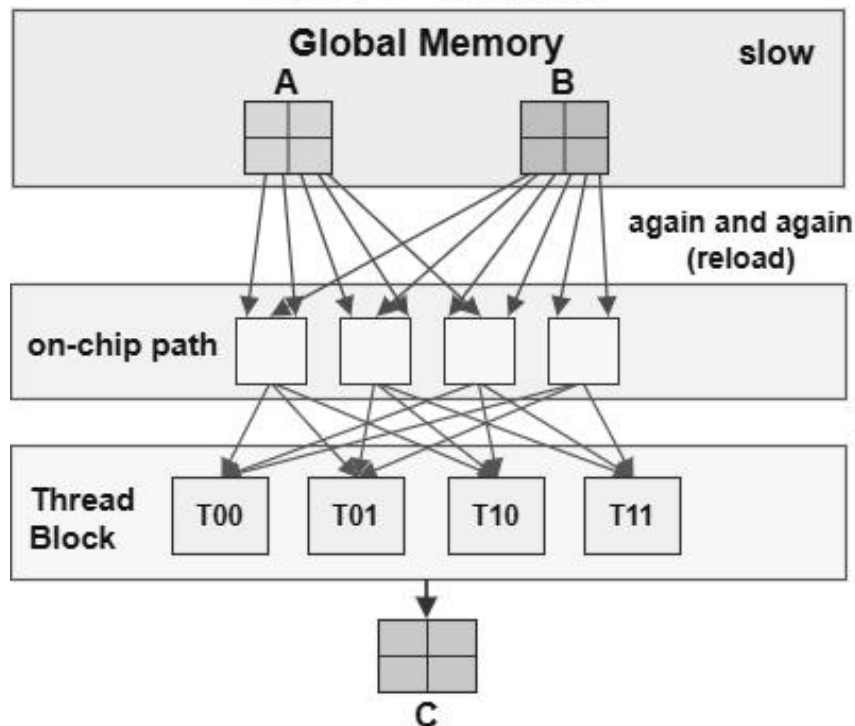
Shared-Memory Tiling GEMM



$$A \times B = \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{00} & b_{01} & b_{02} & b_{03} \\ b_{10} & b_{11} & b_{12} & b_{13} \\ b_{20} & b_{21} & b_{22} & b_{23} \\ b_{30} & b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} c_{00} & c_{01} & c_{02} & c_{03} \\ c_{10} & c_{11} & c_{12} & c_{13} \\ c_{20} & c_{21} & c_{22} & c_{23} \\ c_{30} & c_{31} & c_{32} & c_{33} \end{bmatrix}$$

Shared-Memory Tiling GEMM

Naive GEMM



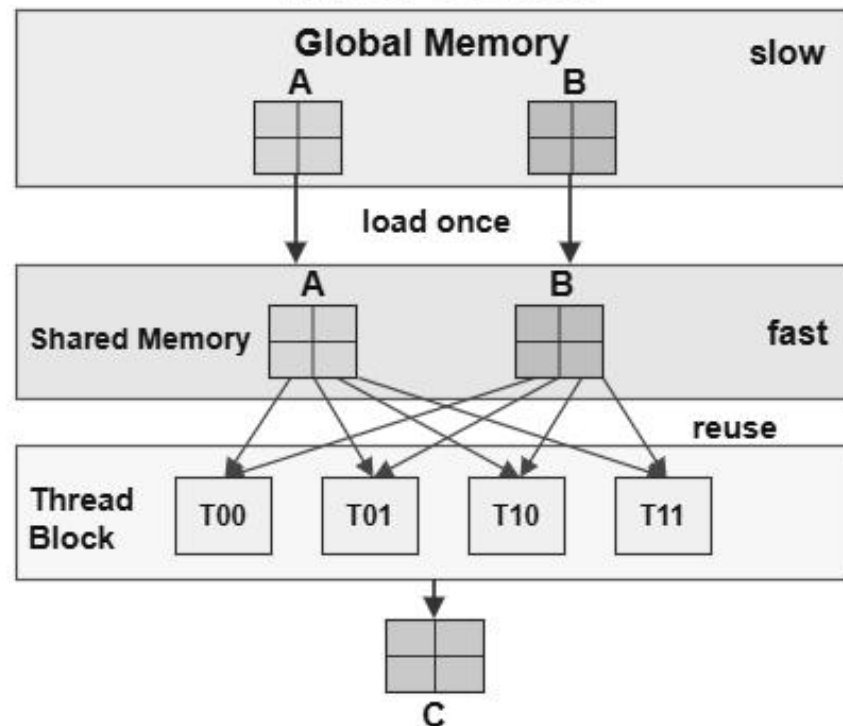
$$C_{00} = a_{00}b_{00} + a_{01}b_{10} + a_{02}b_{20} + a_{03}b_{30}$$

A: $a_{00} \ a_{01} \ a_{02} \ a_{03}$ 4 loads

B: $b_{00} \ b_{10} \ b_{20} \ b_{30}$ 4 loads

Total = 8 global loads

Tiled GEMM



$$C_{01} = a_{00}b_{01} + a_{01}b_{11} + a_{02}b_{21} + a_{03}b_{31}$$

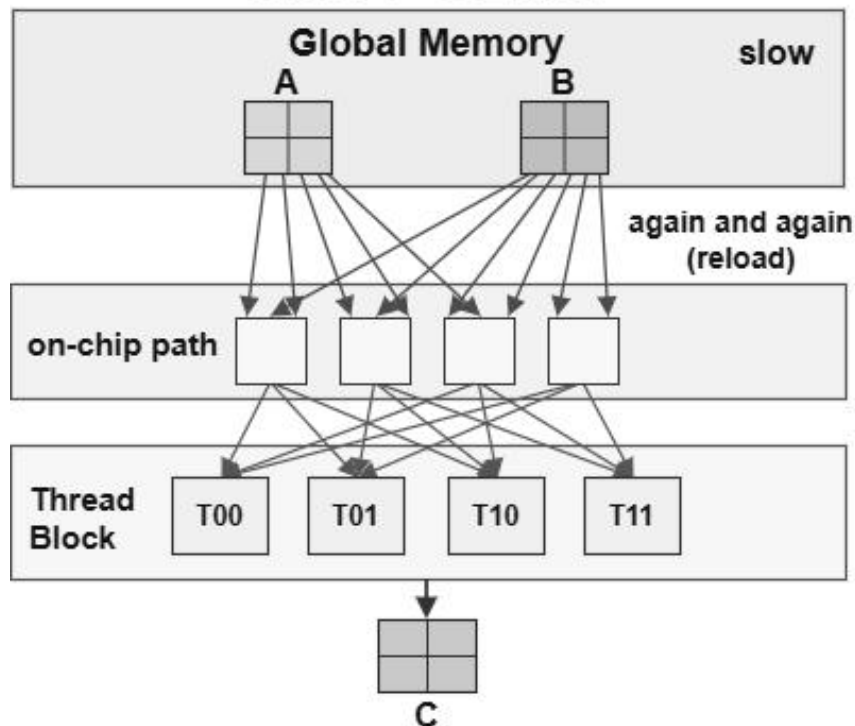
A: $a_{00} \ a_{01} \ a_{02} \ a_{03}$ 4 loads

B: $b_{01} \ b_{11} \ b_{21} \ b_{31}$ 4 loads

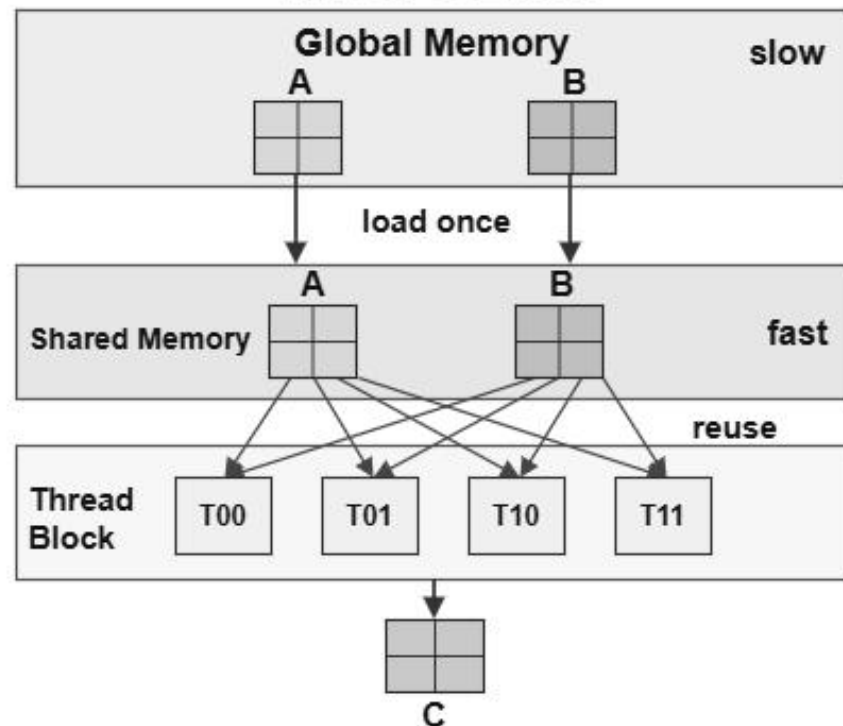
Total = 8 global loads

Shared-Memory Tiling GEMM

Naive GEMM



Tiled GEMM



$$C_{00} = a_{00}b_{00} + a_{01}b_{10} + a_{02}b_{20} + a_{03}b_{30}$$

$$C_{01} = a_{00}b_{01} + a_{01}b_{11} + a_{02}b_{21} + a_{03}b_{31}$$

A: $a_{00} \ a_{01} \ a_{02} \ a_{03}$ 4 loads

A: $a_{00} \ a_{01} \ a_{02} \ a_{03}$ 4 loads

B: $b_{00} \ b_{10} \ b_{20} \ b_{30}$ 4 loads

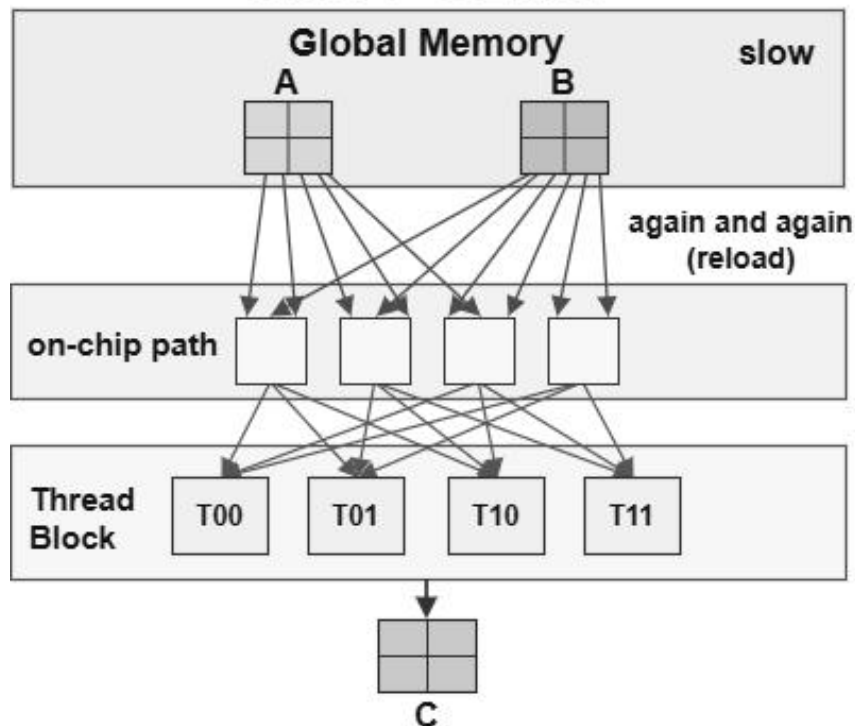
B: $b_{01} \ b_{11} \ b_{21} \ b_{31}$ 4 loads

Total = 8 global loads

Total = 8 global loads

Shared-Memory Tiling GEMM

Naive GEMM



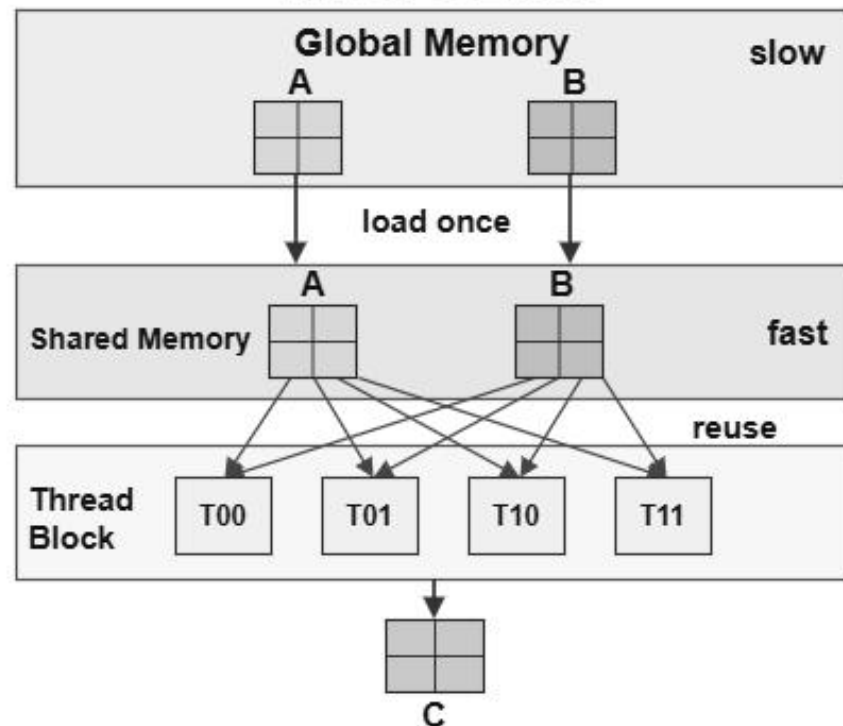
$$C_{00} = a_{00}b_{00} + a_{01}b_{10} + a_{02}b_{20} + a_{03}b_{30}$$

A: $a_{00} \ a_{01} \ a_{02} \ a_{03}$ 4 loads

B: $b_{00} \ b_{10} \ b_{20} \ b_{30}$ 4 loads

Total = 8 global loads

Tiled GEMM



$$C_{10} = a_{10}b_{00} + a_{11}b_{10} + a_{12}b_{20} + a_{13}b_{30}$$

A: $a_{10} \ a_{11} \ a_{12} \ a_{13}$ 4 loads

B: $b_{00} \ b_{10} \ b_{20} \ b_{30}$ 4 loads

Total = 8 global loads

Shared-Memory Tiling GEMM

$$A \times B = \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{00} & b_{01} & b_{02} & b_{03} \\ b_{10} & b_{11} & b_{12} & b_{13} \\ b_{20} & b_{21} & b_{22} & b_{23} \\ b_{30} & b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} c_{00} & c_{01} & c_{02} & c_{03} \\ c_{10} & c_{11} & c_{12} & c_{13} \\ c_{20} & c_{21} & c_{22} & c_{23} \\ c_{30} & c_{31} & c_{32} & c_{33} \end{bmatrix} = C$$

$$C_{00} = a_{00}b_{00} + a_{01}b_{10} + a_{02}b_{20} + a_{03}b_{30}$$

$$C_{01} = a_{00}b_{01} + a_{01}b_{11} + a_{02}b_{21} + a_{03}b_{31}$$

$$C_{10} = a_{10}b_{00} + a_{11}b_{10} + a_{12}b_{20} + a_{13}b_{30}$$

$$C_{11} = a_{10}b_{01} + a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31}$$

Thread	A loads	B loads	Total
T00	4	4	8
T01	4	4	8
T10	4	4	8
T11	4	4	8
Total	16	16	32 global loads

Shared-Memory Tiling GEMM

$$C_{00} = a_{00}b_{00} + a_{01}b_{10} + a_{02}b_{20} + a_{03}b_{30}$$

$$C_{01} = a_{00}b_{01} + a_{01}b_{11} + a_{02}b_{21} + a_{03}b_{31}$$

$$C_{11} = a_{10}b_{01} + a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31}$$

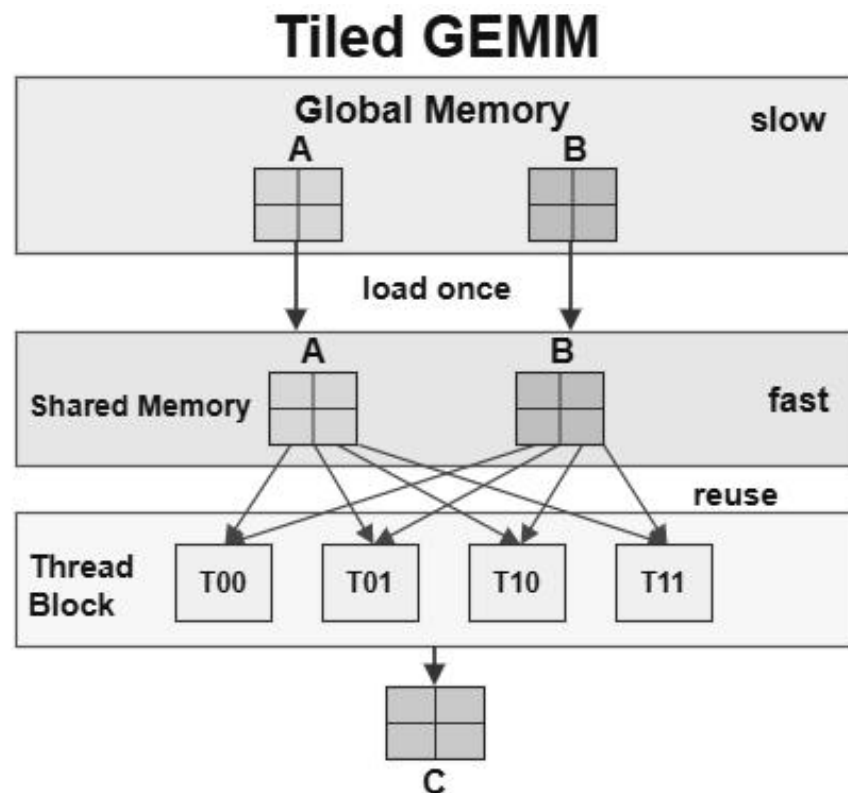
$A[0, :]$ $B[0, :]$ can be shared

$$A_{tile1} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \quad B_{tile1} = \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

$$A_{tile2} = \begin{bmatrix} a_{02} & a_{03} \\ a_{12} & a_{13} \end{bmatrix} \quad B_{tile2} = \begin{bmatrix} b_{02} & b_{03} \\ b_{12} & b_{13} \end{bmatrix}$$

$$\left. \begin{aligned} C_{00}^{(0)} &= a_{00}b_{00} + a_{01}b_{10} \\ C_{00}^{(1)} &= a_{02}b_{20} + a_{03}b_{30} \end{aligned} \right\} C_{00}$$

$$4 + 4 + 4 + 4 = 16 \text{ global loads} < 32 \text{ global loads}$$



Shared-Memory Tiling GEMM

$$C_{00} = a_{00}b_{00} + a_{01}b_{10} + a_{02}b_{20} + a_{03}b_{30}$$

$$C_{01} = a_{00}b_{01} + a_{01}b_{11} + a_{02}b_{21} + a_{03}b_{31}$$

$$C_{11} = a_{10}b_{01} + a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31}$$

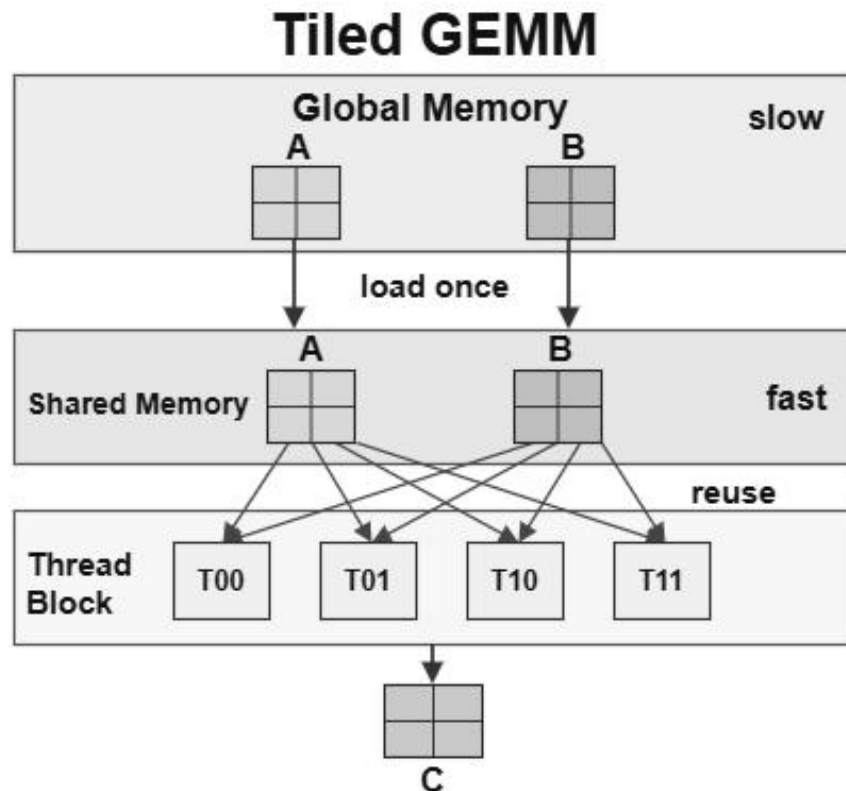
$A[0, :]$ $B[0, :]$ can be shared

$$A_{tile1} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \quad B_{tile1} = \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

$$A_{tile2} = \begin{bmatrix} a_{02} & a_{03} \\ a_{12} & a_{13} \end{bmatrix} \quad B_{tile2} = \begin{bmatrix} b_{02} & b_{03} \\ b_{12} & b_{13} \end{bmatrix}$$

$$\left. \begin{aligned} C_{00}^{(0)} &= a_{00}b_{00} + a_{01}b_{10} \\ C_{00}^{(1)} &= a_{02}b_{20} + a_{03}b_{30} \end{aligned} \right\} C_{00}$$

$$4 + 4 + 4 + 4 = 16 \text{ global loads} < 32 \text{ global loads}$$



Shared-Memory Tiling GEMM



```
#define TILE_SIZE 16

// Tiled GEMM: C = A * B
// Each CUDA thread computes one output element C[row][col]
__global__ void tiledGEMM(float *A, float *B, float *C, int N)
{
    // Shared-memory tiles
    __shared__ float As[TILE_SIZE][TILE_SIZE];
    __shared__ float Bs[TILE_SIZE][TILE_SIZE];

    // Thread position inside the block
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Output element computed by this thread
    int col = blockIdx.x * TILE_SIZE + tx;
    int row = blockIdx.y * TILE_SIZE + ty;

    float sum = 0.0f;
```

```
// Move along the K dimension tile by tile
for (int t = 0; t < (N + TILE_SIZE - 1) / TILE_SIZE; t++)
{
    // -----
    // 1. Global Memory -> Shared Memory
    // -----

    // Each thread loads one element of A
    int A_col = t * TILE_SIZE + tx;

    if (row < N && A_col < N)
        As[ty][tx] = A[row * N + A_col];
    else
        As[ty][tx] = 0.0f;

    // Each thread loads one element of B
    int B_row = t * TILE_SIZE + ty;

    if (B_row < N && col < N)
        Bs[ty][tx] = B[B_row * N + col];
    else
        Bs[ty][tx] = 0.0f;

    // Wait until the whole tile has been loaded
    __syncthreads();

    // -----
    // 2. Compute using data in Shared Memory
    // -----

    for (int k = 0; k < TILE_SIZE; k++)
    {
        sum += As[ty][k] * Bs[k][tx];
    }

    // Make sure all threads finish using this tile
    // before loading the next tile
    __syncthreads();
}

// -----
// 3. Write the final result to Global Memory
// -----

if (row < N && col < N)
{
    C[row * N + col] = sum;
}
}
```

Shared-Memory Tiling GEMM



```
#define TILE_SIZE 16

// Tiled GEMM: C = A * B
// Each CUDA thread computes one output element C[row][col]
__global__ void tiledGEMM(float *A, float *B, float *C, int N)
{
    // Shared-memory tiles
    __shared__ float As[TILE_SIZE][TILE_SIZE];
    __shared__ float Bs[TILE_SIZE][TILE_SIZE];

    // Thread position inside the block
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Output element computed by this thread
    int col = blockIdx.x * TILE_SIZE + tx;
    int row = blockIdx.y * TILE_SIZE + ty;

    float sum = 0.0f;
```

This allocates a tile for matrix A ,B in shared memory.

```
// Move along the K dimension tile by tile
for (int t = 0; t < (N + TILE_SIZE - 1) / TILE_SIZE; t++)
{
    // -----
    // 1. Global Memory -> Shared Memory
    // -----

    // Each thread loads one element of A
    int A_col = t * TILE_SIZE + tx;

    if (row < N && A_col < N)
        As[ty][tx] = A[row * N + A_col];
    else
        As[ty][tx] = 0.0f;

    // Each thread loads one element of B
    int B_row = t * TILE_SIZE + ty;

    if (B_row < N && col < N)
        Bs[ty][tx] = B[B_row * N + col];
    else
        Bs[ty][tx] = 0.0f;

    // Wait until the whole tile has been loaded
    __syncthreads();

    // -----
    // 2. Compute using data in Shared Memory
    // -----

    for (int k = 0; k < TILE_SIZE; k++)
    {
        sum += As[ty][k] * Bs[k][tx];
    }

    // Make sure all threads finish using this tile
    // before loading the next tile
    __syncthreads();
}

// -----
// 3. Write the final result to Global Memory
// -----

if (row < N && col < N)
{
    C[row * N + col] = sum;
}
}
```

Shared-Memory Tiling GEMM



```
#define TILE_SIZE 16

// Tiled GEMM: C = A * B
// Each CUDA thread computes one output element C[row][col]
__global__ void tiledGEMM(float *A, float *B, float *C, int N)
{
    // Shared-memory tiles
    __shared__ float As[TILE_SIZE][TILE_SIZE];
    __shared__ float Bs[TILE_SIZE][TILE_SIZE];

    // Thread position inside the block
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Output element computed by this thread
    int col = blockIdx.x * TILE_SIZE + tx;
    int row = blockIdx.y * TILE_SIZE + ty;

    float sum = 0.0f;
```

tx / ty is the x / y -coordinate of the current thread inside the thread block

Computes the global row / column index of the output element handled by this thread.

```
// Move along the K dimension tile by tile
for (int t = 0; t < (N + TILE_SIZE - 1) / TILE_SIZE; t++)
{
    // -----
    // 1. Global Memory -> Shared Memory
    // -----

    // Each thread loads one element of A
    int A_col = t * TILE_SIZE + tx;

    if (row < N && A_col < N)
        As[ty][tx] = A[row * N + A_col];
    else
        As[ty][tx] = 0.0f;

    // Each thread loads one element of B
    int B_row = t * TILE_SIZE + ty;

    if (B_row < N && col < N)
        Bs[ty][tx] = B[B_row * N + col];
    else
        Bs[ty][tx] = 0.0f;

    // Wait until the whole tile has been loaded
    __syncthreads();

    // -----
    // 2. Compute using data in Shared Memory
    // -----

    for (int k = 0; k < TILE_SIZE; k++)
    {
        sum += As[ty][k] * Bs[k][tx];
    }

    // Make sure all threads finish using this tile
    // before loading the next tile
    __syncthreads();
}

// -----
// 3. Write the final result to Global Memory
// -----

if (row < N && col < N)
{
    C[row * N + col] = sum;
}
}
```

Shared-Memory Tiling GEMM



```
#define TILE_SIZE 16

// Tiled GEMM: C = A * B
// Each CUDA thread computes one output element C[row][col]
__global__ void tiledGEMM(float *A, float *B, float *C, int N)
{
    // Shared-memory tiles
    __shared__ float As[TILE_SIZE][TILE_SIZE];
    __shared__ float Bs[TILE_SIZE][TILE_SIZE];

    // Thread position inside the block
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Output element computed by this thread
    int col = blockIdx.x * TILE_SIZE + tx;
    int row = blockIdx.y * TILE_SIZE + ty;

    float sum = 0.0f;
```

tile by tile

```
// Move along the K dimension tile by tile
for (int t = 0; t < (N + TILE_SIZE - 1) / TILE_SIZE; t++)
{
    // -----
    // 1. Global Memory -> Shared Memory
    // -----

    // Each thread loads one element of A
    int A_col = t * TILE_SIZE + tx;

    if (row < N && A_col < N)
        As[ty][tx] = A[row * N + A_col];
    else
        As[ty][tx] = 0.0f;

    // Each thread loads one element of B
    int B_row = t * TILE_SIZE + ty;

    if (B_row < N && col < N)
        Bs[ty][tx] = B[B_row * N + col];
    else
        Bs[ty][tx] = 0.0f;

    // Wait until the whole tile has been loaded
    __syncthreads();

    // -----
    // 2. Compute using data in Shared Memory
    // -----

    for (int k = 0; k < TILE_SIZE; k++)
    {
        sum += As[ty][k] * Bs[k][tx];
    }

    // Make sure all threads finish using this tile
    // before loading the next tile
    __syncthreads();
}

// -----
// 3. Write the final result to Global Memory
// -----

if (row < N && col < N)
{
    C[row * N + col] = sum;
}
}
```

Shared-Memory Tiling GEMM



```
#define TILE_SIZE 16

// Tiled GEMM: C = A * B
// Each CUDA thread computes one output element C[row][col]
__global__ void tiledGEMM(float *A, float *B, float *C, int N)
{
    // Shared-memory tiles
    __shared__ float As[TILE_SIZE][TILE_SIZE];
    __shared__ float Bs[TILE_SIZE][TILE_SIZE];

    // Thread position inside the block
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Output element computed by this thread
    int col = blockIdx.x * TILE_SIZE + tx;
    int row = blockIdx.y * TILE_SIZE + ty;

    float sum = 0.0f;
```

A tile → As

B tile → Bs

```
// Move along the K dimension tile by tile
for (int t = 0; t < (N + TILE_SIZE - 1) / TILE_SIZE; t++)
{
    // -----
    // 1. Global Memory -> Shared Memory
    // -----

    // Each thread loads one element of A
    int A_col = t * TILE_SIZE + tx;

    if (row < N && A_col < N)
        As[ty][tx] = A[row * N + A_col];
    else
        As[ty][tx] = 0.0f;

    // Each thread loads one element of B
    int B_row = t * TILE_SIZE + ty;

    if (B_row < N && col < N)
        Bs[ty][tx] = B[B_row * N + col];
    else
        Bs[ty][tx] = 0.0f;

    // Wait until the whole tile has been loaded
    __syncthreads();

    // -----
    // 2. Compute using data in Shared Memory
    // -----

    for (int k = 0; k < TILE_SIZE; k++)
    {
        sum += As[ty][k] * Bs[k][tx];
    }

    // Make sure all threads finish using this tile
    // before loading the next tile
    __syncthreads();
}

// -----
// 3. Write the final result to Global Memory
// -----

if (row < N && col < N)
{
    C[row * N + col] = sum;
}
}
```

Shared-Memory Tiling GEMM



```
#define TILE_SIZE 16

// Tiled GEMM: C = A * B
// Each CUDA thread computes one output element C[row][col]
__global__ void tiledGEMM(float *A, float *B, float *C, int N)
{
    // Shared-memory tiles
    __shared__ float As[TILE_SIZE][TILE_SIZE];
    __shared__ float Bs[TILE_SIZE][TILE_SIZE];

    // Thread position inside the block
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Output element computed by this thread
    int col = blockIdx.x * TILE_SIZE + tx;
    int row = blockIdx.y * TILE_SIZE + ty;

    float sum = 0.0f;
```

All threads in the block wait here until the entire A tile and B tile have been loaded into shared memory.

```
// Move along the K dimension tile by tile
for (int t = 0; t < (N + TILE_SIZE - 1) / TILE_SIZE; t++)
{
    // -----
    // 1. Global Memory -> Shared Memory
    // -----

    // Each thread loads one element of A
    int A_col = t * TILE_SIZE + tx;

    if (row < N && A_col < N)
        As[ty][tx] = A[row * N + A_col];
    else
        As[ty][tx] = 0.0f;

    // Each thread loads one element of B
    int B_row = t * TILE_SIZE + ty;

    if (B_row < N && col < N)
        Bs[ty][tx] = B[B_row * N + col];
    else
        Bs[ty][tx] = 0.0f;

    // Wait until the whole tile has been loaded
    __syncthreads();

    // -----
    // 2. Compute using data in Shared Memory
    // -----

    for (int k = 0; k < TILE_SIZE; k++)
    {
        sum += As[ty][k] * Bs[k][tx];
    }

    // Make sure all threads finish using this tile
    // before loading the next tile
    __syncthreads();
}

// -----
// 3. Write the final result to Global Memory
// -----

if (row < N && col < N)
{
    C[row * N + col] = sum;
}
}
```

Shared-Memory Tiling GEMM



```
#define TILE_SIZE 16

// Tiled GEMM: C = A * B
// Each CUDA thread computes one output element C[row][col]
__global__ void tiledGEMM(float *A, float *B, float *C, int N)
{
    // Shared-memory tiles
    __shared__ float As[TILE_SIZE][TILE_SIZE];
    __shared__ float Bs[TILE_SIZE][TILE_SIZE];

    // Thread position inside the block
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Output element computed by this thread
    int col = blockIdx.x * TILE_SIZE + tx;
    int row = blockIdx.y * TILE_SIZE + ty;

    float sum = 0.0f;
```

The thread multiplies one element from its row of A's tile with one element from its column of B's tile.

```
// Move along the K dimension tile by tile
for (int t = 0; t < (N + TILE_SIZE - 1) / TILE_SIZE; t++)
{
    // -----
    // 1. Global Memory -> Shared Memory
    // -----

    // Each thread loads one element of A
    int A_col = t * TILE_SIZE + tx;

    if (row < N && A_col < N)
        As[ty][tx] = A[row * N + A_col];
    else
        As[ty][tx] = 0.0f;

    // Each thread loads one element of B
    int B_row = t * TILE_SIZE + ty;

    if (B_row < N && col < N)
        Bs[ty][tx] = B[B_row * N + col];
    else
        Bs[ty][tx] = 0.0f;

    // Wait until the whole tile has been loaded
    __syncthreads();

    // -----
    // 2. Compute using data in Shared Memory
    // -----

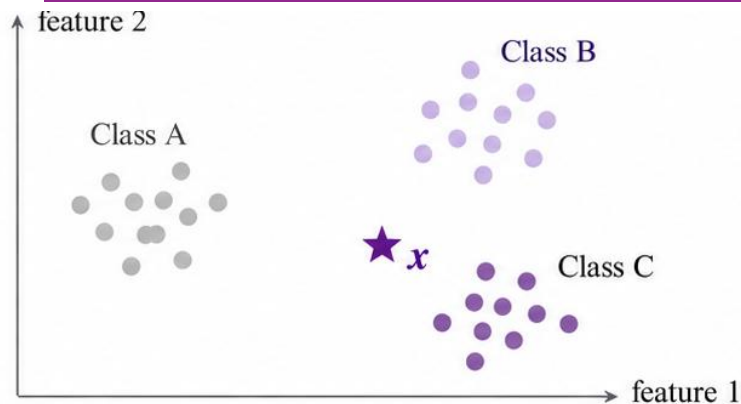
    for (int k = 0; k < TILE_SIZE; k++)
    {
        sum += As[ty][k] * Bs[k][tx];
    }

    // Make sure all threads finish using this tile
    // before loading the next tile
    __syncthreads();
}

// -----
// 3. Write the final result to Global Memory
// -----

if (row < N && col < N)
{
    C[row * N + col] = sum;
}
}
```

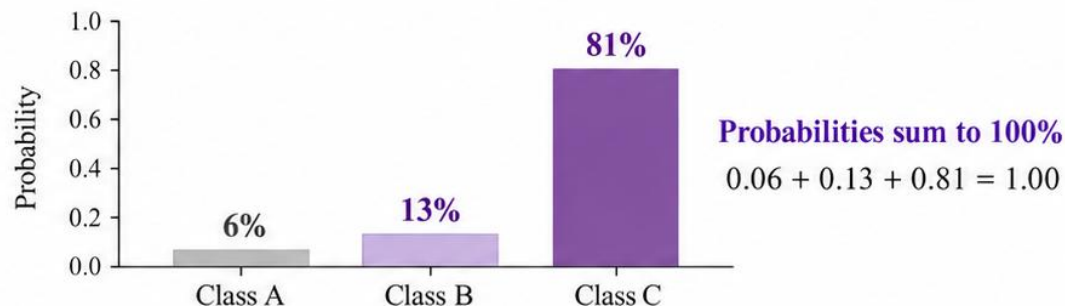
From Max to Softmax



Softmax performs a non-linear normalization.

It maps arbitrary real numbers to non-negative values that sum to 1, so they can be interpreted as probabilities.

$$\text{softmax}([1.2, 2.0, 3.8]) = [0.06, 0.13, 0.81]$$



Which class does x belong to?

scores = [1.2, 2.0, 3.8]

MAX → **choose Class C**

MAX gives only the winner.

$$\text{Softmax}(X) = \text{Softmax} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} \frac{\exp(x_1)}{\sum_{j=1}^V \exp(x_j)} \\ \frac{\exp(x_2)}{\sum_{j=1}^V \exp(x_j)} \\ \vdots \\ \frac{\exp(x_n)}{\sum_{j=1}^V \exp(x_j)} \end{bmatrix} \in \mathbb{R}^n$$

$$\sum_{i=1}^V \frac{\exp(x_i)}{\sum_{j=1}^V \exp(x_j)} = 1$$

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \in \mathbb{R}^n \xrightarrow{\text{Input}} \text{Softmax} \xrightarrow{\text{Output}} Y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} \frac{\exp(x_1)}{\sum_{j=1}^V \exp(x_j)} \\ \frac{\exp(x_2)}{\sum_{j=1}^V \exp(x_j)} \\ \vdots \\ \frac{\exp(x_2)}{\sum_{j=1}^V \exp(x_j)} \end{bmatrix} \in \mathbb{R}^n$$

problem: $\sum_{j=1}^V \exp(x_j)$ may overflow

example 1:

$$\begin{aligned} x &= (1000, 1001, 1002) \\ d &= \exp(1000) + \exp(1001) + \exp(1002) \\ \exp(1000) &\approx 1.97 \times 10^{434} \\ \max(\text{float64}) &\approx 1.8 \times 10^{308} < \exp(1000) \end{aligned}$$

From Softmax to Stable Softmax I

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \in \mathbb{R}^n \xrightarrow{\text{Input}} \text{Softmax} \xrightarrow{\text{output}} Y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} \frac{\exp(x_1)}{\sum_{j=1}^V \exp(x_j)} \\ \frac{\exp(x_2)}{\sum_{j=1}^V \exp(x_j)} \\ \vdots \\ \frac{\exp(x_2)}{\sum_{j=1}^V \exp(x_j)} \end{bmatrix} \in \mathbb{R}^n$$

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \in \mathbb{R}^n \xrightarrow{\text{Input}} \text{Stable Softmax} \xrightarrow{\text{output}} Y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} \frac{\exp(x_1)}{\sum_{j=1}^V \exp(x_j - m)} \\ \frac{\exp(x_2)}{\sum_{j=1}^V \exp(x_j - m)} \\ \vdots \\ \frac{\exp(x_2)}{\sum_{j=1}^V \exp(x_{j-m} - m)} \end{bmatrix} \in \mathbb{R}^n$$

denote $m = \max_j (x_j)$

From Softmax to Stable Softmax II

c is a constant $\in \mathbb{R}$, $\tilde{x}_i = x_i - c$

$$\text{softmax}(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^n e^{x_j}}$$

$$\text{softmax}(\tilde{x})_i = \frac{\exp(x_i - c)}{\sum_{j=1}^V \exp(x_j - c)}$$

$$= \frac{\exp(x_i)\exp(-c)}{\exp(-c) \sum_{j=1}^V \exp(x_j)}$$

$$= \frac{\exp(x_i)}{\sum_{j=1}^V \exp(x_j)}$$

$$= \text{softmax}(x)_i$$

$$\left\{ \begin{array}{l} \exp(x_i - c) = \exp(x_i)\exp(-c) \\ \sum_{j=1}^V \exp(x_j - c) = \exp(-c) \sum_{j=1}^V \exp(x_j) \end{array} \right.$$

$$\text{softmax}(\mathbf{X}) = \text{softmax}(\mathbf{X} - c\mathbf{1})$$

$$\mathbf{1} = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}$$

Translation Invariance of Softmax

To prevent overflow : require $\sum_{j=1}^V \exp(x_j - c) < \text{Float}_{\max}$

$$\exp(x_j - c) < \text{Float}_{\max}$$

$$c \geq \max_j x_j - \log \text{Float}_{\max}$$

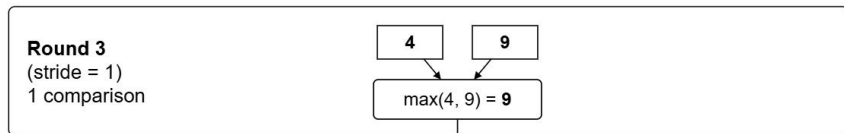
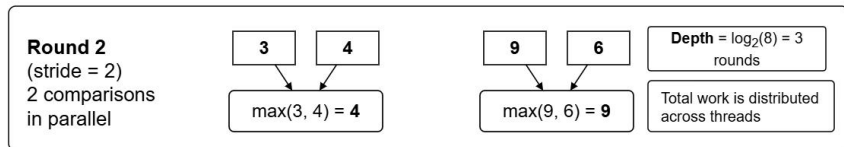
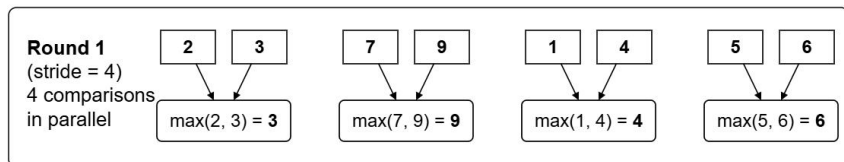
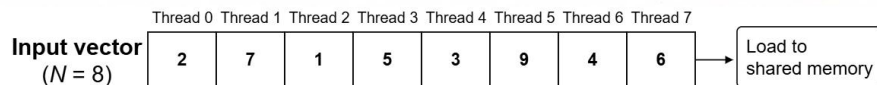
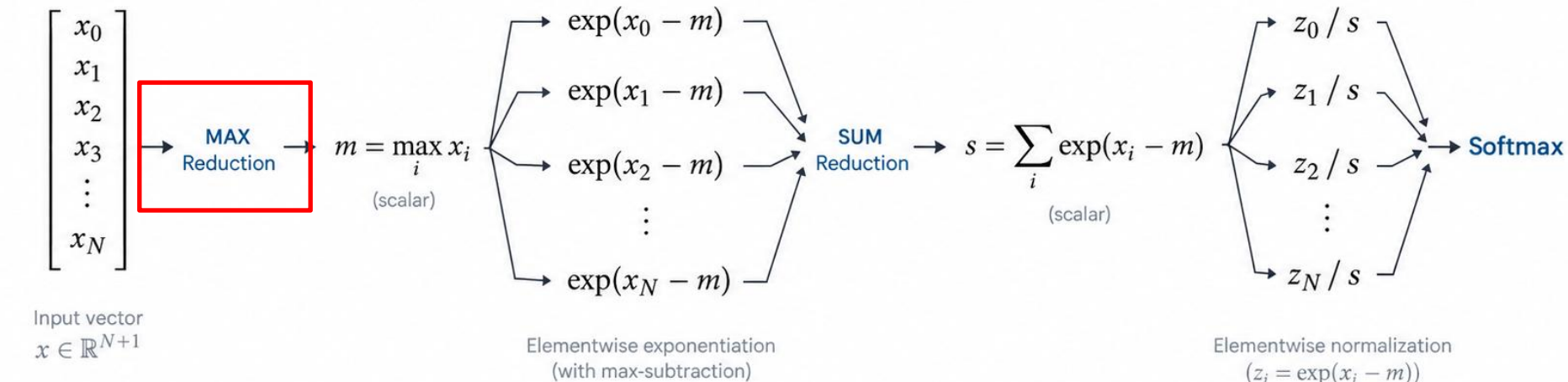
$$c = \text{Float}_{\max} \quad ?$$

To avoid catastrophic underflow : Choose c s.t. $\max_j (x_j - c) = 0$

denote the chosen c as $m = \max_j (x_j)$

$$\text{Softmax}_{\text{stable}}(X) = \text{Softmax}_{\text{stable}} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} \frac{\exp(x_1)}{\sum_{j=1}^V \exp(x_j - m)} \\ \frac{\exp(x_2)}{\sum_{j=1}^V \exp(x_j - m)} \\ \vdots \\ \frac{\exp(x_i)}{\sum_{j=1}^V \exp(x_j - m)} \end{bmatrix} \in \mathbb{R}^n, m = \max_j (x_j)$$

Stable Softmax Workflow I



Final result

Max = 9

```
__global__ void maxReduction(float *x, float *result, int N)
{
    extern __shared__ float sdata[];

    int tid = threadIdx.x;

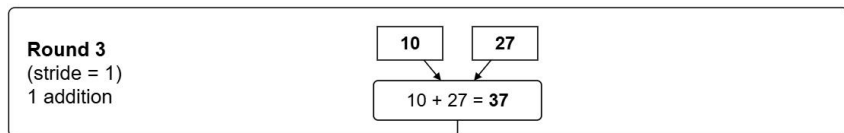
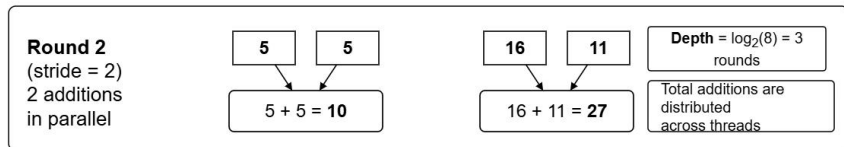
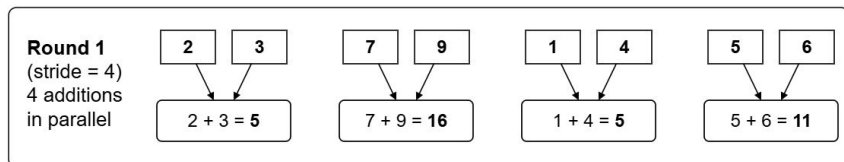
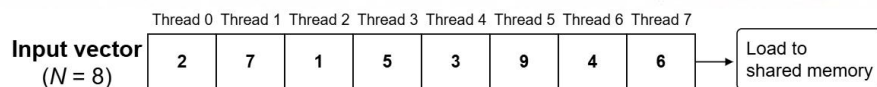
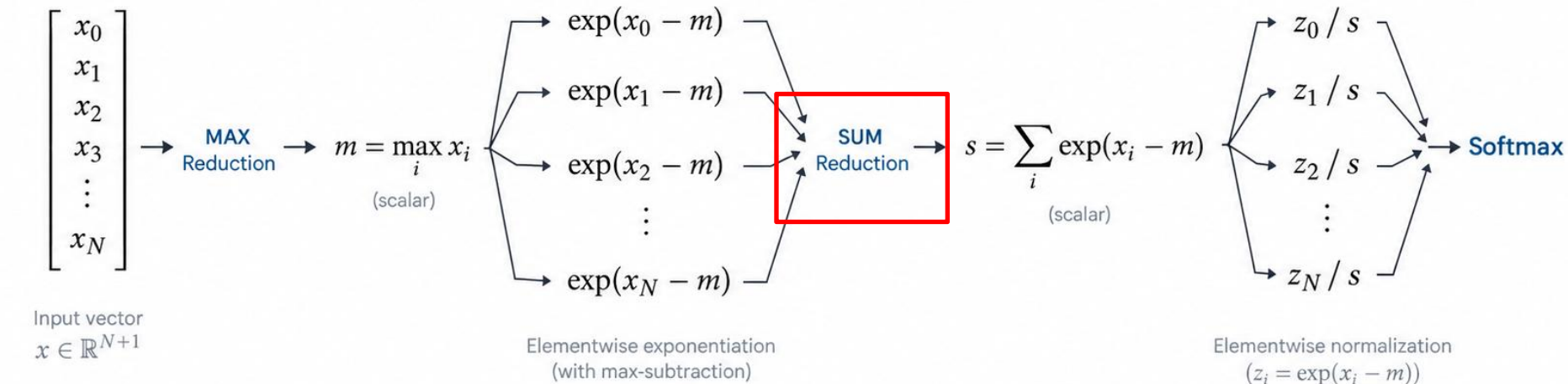
    // Each thread loads one element
    if (tid < N)
        sdata[tid] = x[tid];
    else
        sdata[tid] = -FLT_MAX;

    __syncthreads();

    // Parallel MAX reduction
    for (int stride = blockDim.x / 2; stride > 0; stride /= 2)
    {
        if (tid < stride)
            sdata[tid] = fmaxf(sdata[tid], sdata[tid + stride]);
    }
    __syncthreads();

    // Final maximum
    if (tid == 0)
        result[0] = sdata[0];
}
```

Stable Softmax Workflow II



Final result

Sum = 37

```

__global__ void sumReduction(float *z, float *result, int N)
{
    extern __shared__ float sdata[];

    int tid = threadIdx.x;

    // Each thread loads one element
    if (tid < N)
        sdata[tid] = z[tid];
    else
        sdata[tid] = 0.0f;

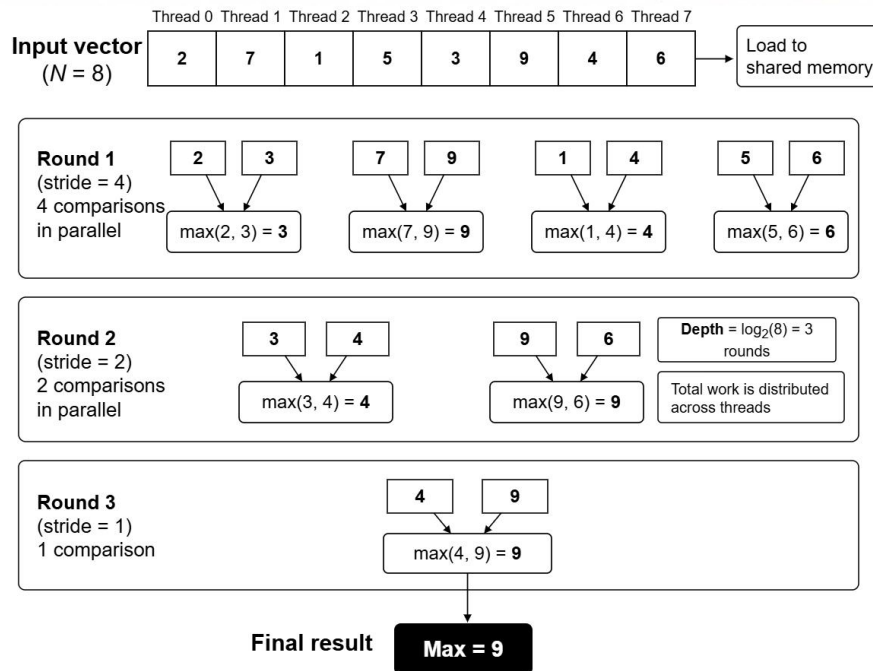
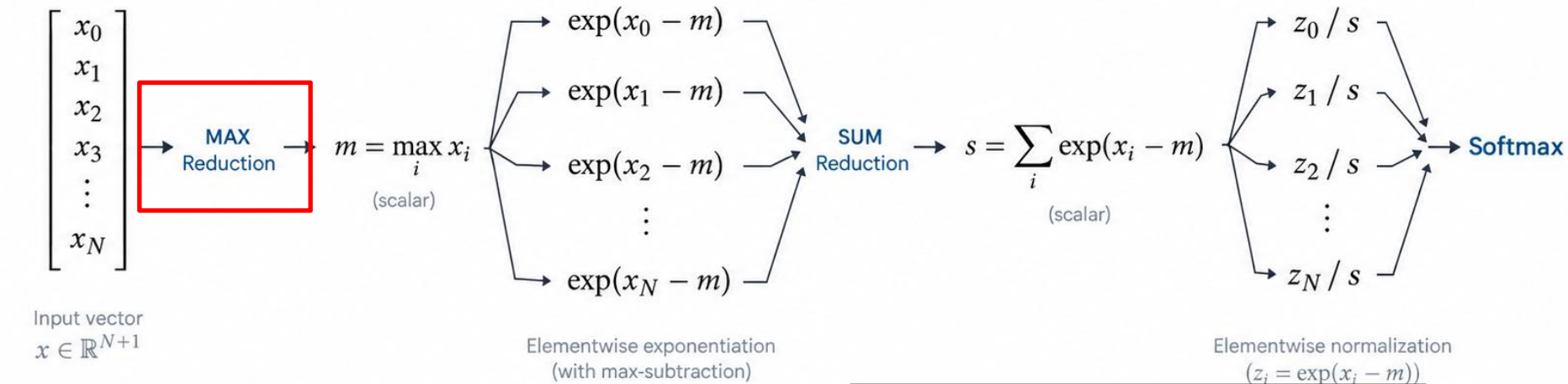
    __syncthreads();

    // Parallel SUM reduction
    for (int stride = blockDim.x / 2; stride > 0; stride /= 2)
    {
        if (tid < stride)
            sdata[tid] += sdata[tid + stride];

        __syncthreads();
    }

    // Final sum
    if (tid == 0)
        result[0] = sdata[0];
}
    
```

Stable Softmax Workflow I



```

__global__ void maxReduction(float *x, float *result, int N)
{
    extern __shared__ float sdata[];

    int tid = threadIdx.x;

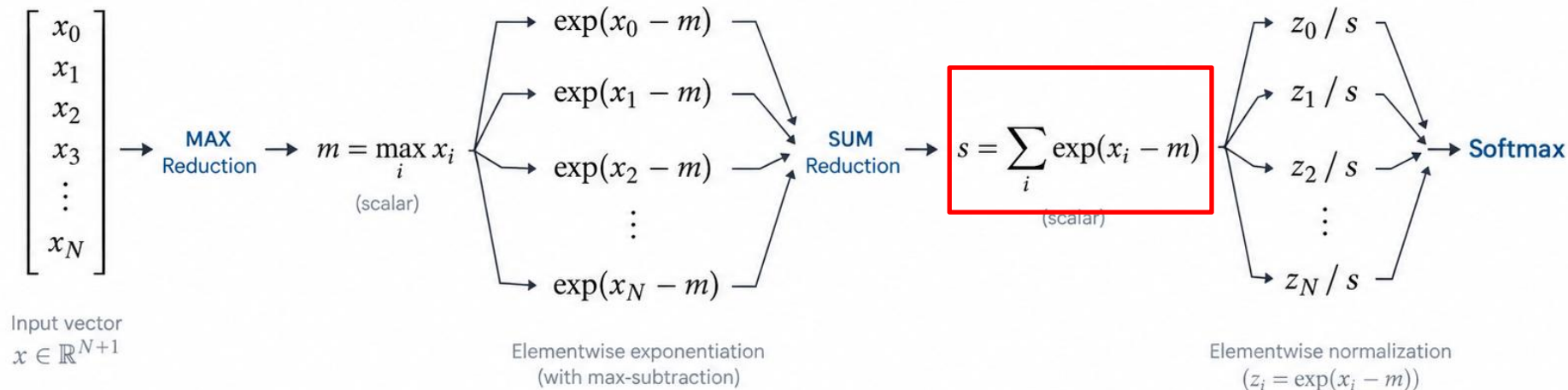
    // Each thread loads one element
    if (tid < N)
        sdata[tid] = x[tid];
    else
        sdata[tid] = -FLT_MAX;

    __syncthreads();

    // Parallel MAX reduction
    for (int stride = blockDim.x / 2; stride > 0; stride /= 2)
    {
        if (tid < stride)
            sdata[tid] = fmaxf(sdata[tid], sdata[tid + stride]);
    }
    __syncthreads();

    // Final maximum
    if (tid == 0)
        result[0] = sdata[0];
}
    
```

Stable Softmax Workflow III



```
__global__ void expKernel(float *x, float *z, float m, int N)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < N)
        z[i] = expf(x[i] - m);
}
```

Achieve Naive Stable Softmax

```
__global__ void stableSoftmax(const float* x, float* y, int B, int N)
{
    // One block processes one Softmax vector
    int row = blockIdx.x;

    if (row >= B) return;

    const float* input = x + row * N;
    float* output = y + row * N;

    // Shared by all threads in this block
    __shared__ float m;
    __shared__ float d;

    // Pass 1 and Pass 2 are still sequential
    if (threadIdx.x == 0) {
        m = -INFINITY;
        d = 0.0f;

        // Pass 1: find maximum
        for (int i = 0; i < N; i++) {
            m = fmaxf(m, input[i]);
        }

        // Pass 2: compute normalizer
        for (int j = 0; j < N; j++) {
            d += expf(input[j] - m);
        }

        // Wait until m and d are ready
        __syncthreads();

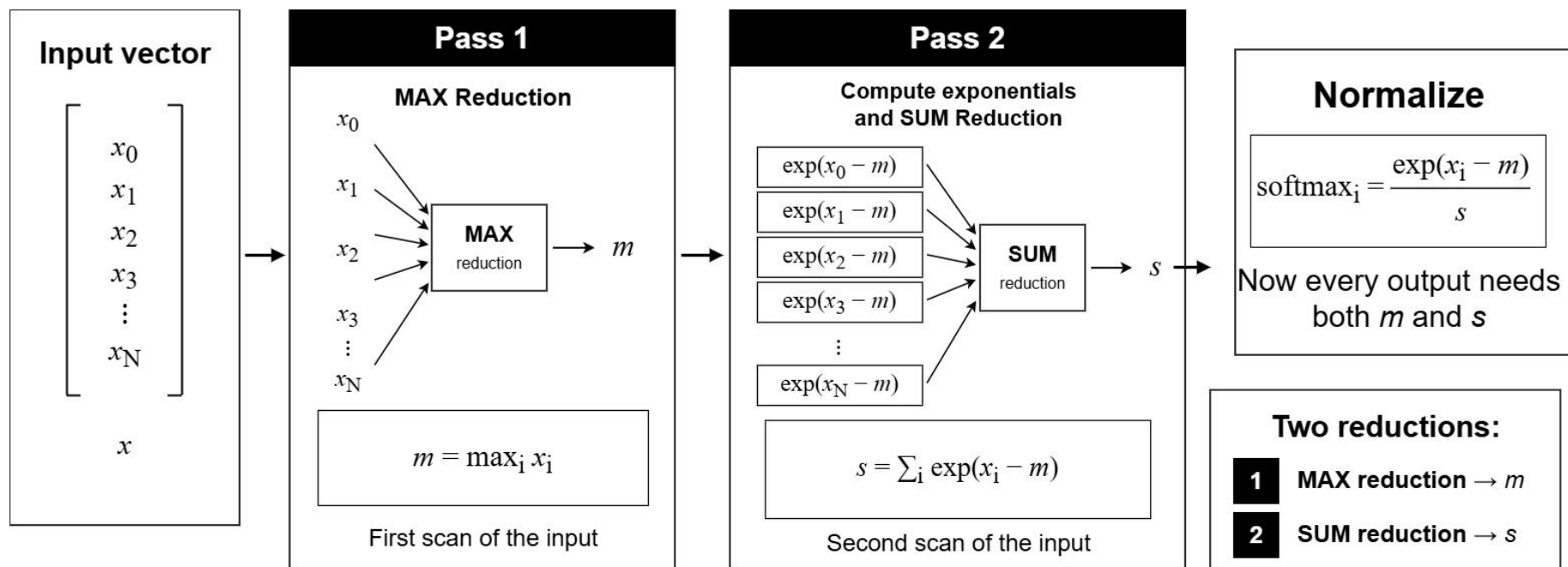
        // Pass 3: compute final Softmax values in parallel
        for (int j = threadIdx.x; j < N; j += blockDim.x) {
            output[j] = expf(input[j] - m) / d;
        }
    }
}
```

Dependency

$$\left\{ \begin{array}{l} m = \max_{i \in A} x_i \\ d = \sum_{j=1}^V \exp(x_j - m) \end{array} \right.$$

$$y_i = \frac{\exp(x_i)}{\sum_{j=1}^V \exp(x_j - m)}$$

Stable Softmax Pipeline



$$m_{j-1} = \max_{1 \leq i \leq j-1} x_i$$

$$d_{j-1} = \sum_{i=1}^{j-1} e^{x_i - m_{j-1}}$$

$$d_j = \sum_{i=1}^{j-1} e^{x_i - m_{j-1} + m_{j-1} - m_j} + e^{x_j - m_j}$$

From Softmax to Online Softmax

$$m_{j-1} = \max_{1 \leq i \leq j-1} x_i \quad \hat{d}_{j-1} = \sum_{i=1}^{j-1} e^{x_i - m_{j-1}}$$

Input $m_{j-1}, \hat{d}_{j-1}, x_j$ Output m_j, d_j

x_j : the current input element being processed.

m_{j-1} : the maximum of the first $j - 1$ input elements.

m_j : the maximum of the first j input elements.

$\hat{d}_{j-1}$: the previously stored normalizer for the first $j - 1$ elements, computed using the old maximum m_{j-1}

d_{j-1} : the previously stored normalizer for the first $j - 1$ elements, computed using the new maximum m_j

d_j : the previously stored normalizer for the first j elements, computed using the new maximum m_j

From Softmax to Online Softmax

$$m_{j-1} = \max_{1 \leq i \leq j-1} x_i \quad \hat{d}_{j-1} = \sum_{i=1}^{j-1} e^{x_i - m_{j-1}}$$

Input $m_{j-1}, \hat{d}_{j-1}, x_j$

$$m_j = \max(m_{j-1}, x_j)$$

$$\begin{aligned} d_j &= \sum_{i=1}^j e^{x_i - m_j} \\ &= \sum_{i=1}^{j-1} e^{x_i - m_j} + e^{x_j - m_j} \\ &= \sum_{i=1}^{j-1} e^{x_i - m_{j-1} + m_{j-1} - m_j} + e^{x_j - m_j} \\ &= e^{m_{j-1} - m_j} \sum_{i=1}^{j-1} e^{x_i - m_{j-1}} + e^{x_j - m_j} \\ &= e^{m_{j-1} - m_j} \hat{d}_{j-1} + e^{x_j - m_j} \\ &= d_{j-1} + e^{x_j - m_j}. \end{aligned}$$

$$d_{j-1} = e^{m_{j-1} - m_j} \hat{d}_{j-1}$$

$$m_j = \max(m_{j-1}, x_j)$$

$$d_j = \sum_{i=1}^{j-1} e^{x_i - m_{j-1} + m_{j-1} - m_j} + e^{x_j - m_j}$$

Achieve Naive Online Softmax

```
__global__ void onlineSoftmax(const float* x, float* y, int B, int N)
{
    // One block processes one Softmax vector
    int row = blockIdx.x;

    if (row >= B) return;

    const float* input = x + row * N;
    float* output = y + row * N;

    // Online Softmax state
    __shared__ float m;
    __shared__ float d;

    // Pass 1:update running maximum m and running normalizer d together
    if (threadIdx.x == 0) {
        m = -INFINITY;
        d = 0.0f;

        for (int j = 0; j < N; j++) {
            float m_new = fmaxf(m, input[j]);

            d = d * expf(m - m_new) + expf(input[j] - m_new);

            m = m_new;
        }

        // Wait until m and d are ready
        __syncthreads();

        // Pass 2:compute final Softmax values in parallel
        for (int j = threadIdx.x; j < N; j += blockDim.x){
            output[j] = expf(input[j] - m) / d;
        }
    }
}
```

$$m_j = \max(m_{j-1}, x_j)$$

$$d_j = \sum_{i=1}^{j-1} e^{x_i - m_{j-1} + m_{j-1} - m_j} + e^{x_j - m_j}$$

$$\left. \begin{aligned} m_A &= \max_{i \in A} x_i, & d_A &= \sum_{i \in A} e^{x_i - m_A} \\ m_B &= \max_{i \in B} x_i, & d_B &= \sum_{i \in B} e^{x_i - m_B} \end{aligned} \right\} \quad m = \max_{i \in A} x_i, \quad d = \sum_{i \in A \cup B} e^{x_i - m}$$

m_A : the maximum of the set A elements.

m_B : the maximum of the set B elements.

m : the maximum of the set A and B elements.

d_A : the previously stored normalizer for the set A elements

d_B : the previously stored normalizer for the set B elements

d : the new stored normalizer for the set A and B elements

$$\left. \begin{aligned} m_A &= \max_{i \in A} x_i, & d_A &= \sum_{i \in A} e^{x_i - m_A} \\ m_B &= \max_{i \in B} x_i, & d_B &= \sum_{i \in B} e^{x_i - m_B} \end{aligned} \right\} \quad m = \max_{i \in A} x_i, \quad d = \sum_{i \in A \cup B} e^{x_i - m}$$

m_A : the maximum of the set A elements.

m_B : the maximum of the set B elements.

m : the maximum of the set A and B elements.

d_A : the previously stored normalizer for the set A elements

d_B : the previously stored normalizer for the set B elements

d : the new stored normalizer for the set A and B elements

$$\left. \begin{aligned} m_A &= \max_{i \in A} x_i, & d_A &= \sum_{i \in A} e^{x_i - m_A} \\ m_B &= \max_{i \in B} x_i, & d_B &= \sum_{i \in B} e^{x_i - m_B} \end{aligned} \right\} \quad m = \max_{i \in A} x_i, \quad d = \sum_{i \in A} e^{x_i - m_A}$$

$$m = \max(m_A, m_B)$$

$$\begin{aligned} d &= \sum_{i \in A \cup B} e^{x_i - m} \\ &= \sum_{i \in A} e^{x_i - m} + \sum_{i \in B} e^{x_i - m} \\ &= e^{m_A - m} \sum_{i \in A} e^{x_i - m_A} + e^{m_B - m} \sum_{i \in B} e^{x_i - m_B} \\ &= d_A e^{m_A - m} + d_B e^{m_B - m} \end{aligned}$$

$$(m_A, d_A) \oplus (m_B, d_B) = (m, d_A e^{m_A - m} + d_B e^{m_B - m})$$

Achieve Parallel Online Softmax

```
__global__ void onlineSoftmax(const float* x, float* y, int B, int N)
{
    // One block processes one Softmax vector
    int row = blockIdx.x;

    if (row >= B)
        return;

    const float* input = x + row * N;
    float* output = y + row * N;

    // Online Softmax state
    __shared__ float mA, dA, mB, dB, m, d;
    int mid = N / 2;

    // Pass 1: update running maximum m and running normalizer together
    if (threadIdx.x == 0){
        mA = -INFINITY;
        dA = 0.0f;
        for (int j = 0; j < mid; j++){
            float m_new = fmaxf(mA, input[j]);
            dA = dA * expf(mA - m_new) + expf(input[j] - m_new);
            mA = m_new;
        }
    }

    if (threadIdx.x == 1){
        mB = -INFINITY;
        dB = 0.0f;
        for (int j = mid; j < N; j++){
            float m_new = fmaxf(mB, input[j]);
            dB = dB * expf(mB - m_new) + expf(input[j] - m_new);
            mB = m_new;
        }
    }

    // Wait until dA and dB are ready
    __syncthreads();

    // Combine dA and dB
    if (threadIdx.x == 0){
        m = fmaxf(mA, mB);
        d = dA * expf(mA - m) + dB * expf(mB - m);
    }

    // Wait until m and d are ready
    __syncthreads();

    // Pass 2: compute final Softmax values in parallel
    for (int j = threadIdx.x; j < N; j += blockDim.x){
        output[j] = expf(input[j] - m) / d;
    }
}
```

$$\underbrace{(m_{j-1}, d_{j-1})}_{\text{previous elements}} \oplus \underbrace{(x_j, 1)}_{\text{one elements}}$$

$$\underbrace{(m_A, d_A)}_{\text{one subset}} \oplus \underbrace{(m_B, d_B)}_{\text{another subset}}$$

$$(m_A, d_A) \oplus (m_B, d_B) = (m, d_A e^{m_A - m} + d_B e^{m_B - m})$$

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

Why is naive attention slow?

- Materializes the full score matrix $S = QK^T$ and probability matrix $P = \text{softmax}(S)$
- High HBM traffic: write S , read S , write P , read P
- Time and memory grow quadratically with sequence length: $O(N^2)$

$$S = QK^T, \quad P = \text{softmax}(S), \quad O = PV$$

FlashAttention: built on two ideas

1. Tiled GEMM

Compute QK^T and PV block by block instead of materializing the full $N \times N$ matrix.

$$\text{blockwise } Q_i K_j^T$$

2. Online Softmax

Maintain running statistics while scanning blocks:

$$(m_A, d_A) \oplus (m_B, d_B) = (m, d_A e^{m_A - m} + d_B e^{m_B - m})$$

Why faster?

- No full attention matrix in HBM
- Much lower memory traffic
- Exact attention, not approximation

What we have explained

Core algorithmic idea: **Tiled GEMM + Online Softmax**

- Avoid writing the full $S = QK^T$ and $P = \text{softmax}(S)$
- Maintain running m and ℓ while scanning tiles
- A small teaching kernel shows the math, not production scheduling

What real FlashAttention-1 adds

- One fused kernel: $QK^T \rightarrow \text{online softmax} \rightarrow PV$
- Careful SRAM/shared-memory/register tiling
- Backward pass with recomputation to save memory
- Causal mask, dropout, variable length, occupancy details

FlashAttention evolves with GPU architecture

FA-1

A100 / Ampere

IO-aware tiling; exact attention;
no S/P matrix in HBM

FA-2

A100 / Ampere

Less non-matmul FLOPs;
betterblock/warp partitioning;
 $\sim 2\times$ vs FA-1

FA-3

H100 / Hopper

TMA + async Tensor Cores;
warp specialization;
FP8; $1.5\text{--}2\times$ vs FA-2

FA-4

B200 / Blackwell

Async MMA pipeline; tensor
memory; 2-CTA MMA; software
exp/rescale

MHA / MQA / GQA

Share or group K, V heads.
Main win: smaller KV cache in decoding.

FlashAttention family

Exact full attention with IO-aware tiling.
Avoid materializing S and P in HBM.

PagedAttention / KV manager

Page KV cache into memory blocks.
Serving memory manager, not new math.