**START WITH CUDA**

Absolutely! Let’s do a **step-by-step CUDA programming tutorial**, starting from the very basics and gradually building up. I’ll explain **concepts, code, and GPU behavior** along the way.

We’ll structure this in **levels** so you can follow along with coding exercises.

## **Step 1: Understanding CUDA Basics**

**CUDA** lets you run code on NVIDIA GPUs. Key concepts:

1. **Host vs Device**
   * **Host** → CPU memory and CPU code
   * **Device** → GPU memory and GPU code
2. **Kernel**
   * A function that runs on the GPU
   * Executed by **many threads in parallel**
   * Declared with \_\_global\_\_
3. **Thread, Block, Grid**
   * GPU runs **threads in blocks**, blocks form a **grid**
   * Each thread has a **unique ID** used to work on data
4. **Memory Types**
   * **Global memory** → accessible by all threads, slow
   * **Shared memory** → shared among threads in a block, fast
   * **Registers** → per-thread storage, very fast

## **Step 2: Your First CUDA Program**

We’ll start with the classic “Hello from GPU” example.

#include <stdio.h>

// GPU kernel: runs on the device

\_\_global\_\_ void helloGPU() {

printf("Hello from GPU! Thread ID: %d\n", threadIdx.x);

}

int main() {

// Launch kernel: 5 threads in 1 block

helloGPU<<<1, 5>>>();

// Wait for GPU to finish

cudaDeviceSynchronize();

printf("Hello from CPU!\n");

return 0;

}

### **Explanation:**

* \_\_global\_\_ void helloGPU() → Kernel function
* threadIdx.x → Thread index within a block
* <<<1, 5>>> → 1 block, 5 threads
* cudaDeviceSynchronize() → Wait for all GPU threads to finish before continuing CPU code

**Expected output:**

Hello from GPU! Thread ID: 0

Hello from GPU! Thread ID: 1

Hello from GPU! Thread ID: 2

Hello from GPU! Thread ID: 3

Hello from GPU! Thread ID: 4

Hello from CPU!

Yes, there **is a limit on the total number of blocks**, but it’s usually very large and depends on the **GPU architecture**. Let’s break it down carefully.

### **1️⃣ CUDA execution hierarchy recap**

* **Threads** → smallest unit of execution
* **Blocks** → a group of threads
* **Grid** → a group of blocks

When you launch a kernel:

kernel<<<numBlocks, threadsPerBlock>>>();

* threadsPerBlock → max threads per block, typically **1024**
* numBlocks → max number of blocks per grid dimension

### **2️⃣ Maximum blocks per grid**

* CUDA grids are **3D** (x, y, z)
* Maximum blocks **per dimension** (depends on GPU compute capability, common values):

|  |  |  |  |
| --- | --- | --- | --- |
| **GPU compute capability** | **Max blocks in X-dim** | **Y-dim** | **Z-dim** |
| 3.x, 5.x, 6.x, 7.x | 2³¹ − 1 (~2 billion) | 65535 | 65535 |

* **1D grid:** max ~2 billion blocks in X direction
* **2D or 3D grid:** Y and Z are limited to 65535

### **3️⃣ Total threads per kernel launch**

* totalThreads = threadsPerBlock \* numBlocks
* Example (1D grid):

threadsPerBlock = 1024;

numBlocks = 1000000;

totalThreads = 1024 \* 1,000,000 = 1,024,000,000

* This is allowed as long as **threadsPerBlock ≤ 1024** and **numBlocks ≤ max grid size**.

✅ Practically, GPUs often **run out of memory** before reaching max blocks, because each thread/block uses registers and shared memory.

### **4️⃣ Summary**

* **Threads per block**: ≤ 1024 (typical)
* **Blocks per grid (X dimension)**: ≤ 2³¹ − 1 (~2 billion)
* **Total threads per kernel**: threadsPerBlock × numBlocks
* **Practical limit**: GPU memory, registers, and shared memory per block

If you want, I can draw a **diagram showing threads → blocks → grids** so you can visually see limits and how to calculate them.

Do you want me to make that diagram?

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

No worries! Let’s break down **X, Y, Z dimensions in CUDA grids and blocks** in a simple way so it makes sense.

### **1️⃣ Concept: Threads → Blocks → Grid**

Think of **threads** like **workers**, **blocks** like **rooms**, and **grid** like a **building**.

* **Thread** → a single worker
* **Block** → a room containing multiple workers (threads)
* **Grid** → the building containing many rooms (blocks)

### **2️⃣ Dimensions**

CUDA allows **threads, blocks, and grids** to have **up to 3 dimensions**. This is useful for mapping naturally to data structures like arrays, matrices, or 3D volumes.

#### **a) Threads per block (threadIdx)**

* Threads in a block can be **1D, 2D, or 3D**
  + 1D block → threadIdx.x
  + 2D block → threadIdx.x, threadIdx.y
  + 3D block → threadIdx.x, threadIdx.y, threadIdx.z

**Example:**

dim3 threadsPerBlock(4, 2, 2); // 3D block: x=4, y=2, z=2

* Total threads in block = 4 × 2 × 2 = 16 threads

#### **b) Blocks per grid (blockIdx)**

* Grid is a collection of blocks, also **1D, 2D, or 3D**
* For 1D grid → blockIdx.x
* For 2D grid → blockIdx.x, blockIdx.y
* For 3D grid → blockIdx.x, blockIdx.y, blockIdx.z

**Example:**

dim3 numBlocks(3, 2, 1); // 3D grid: x=3, y=2, z=1

* Total blocks = 3 × 2 × 1 = 6 blocks

### **3️⃣ How we compute global thread ID**

If you have a 1D block/grid:

int tid = threadIdx.x + blockIdx.x \* blockDim.x;

* threadIdx.x → position of thread in the block
* blockIdx.x → which block we are in
* blockDim.x → total threads in a block

✅ Works for linear arrays.

If you use **2D or 3D threads/blocks**, you calculate global ID using **all dimensions**:

int tid\_x = threadIdx.x + blockIdx.x \* blockDim.x;

int tid\_y = threadIdx.y + blockIdx.y \* blockDim.y;

int tid\_z = threadIdx.z + blockIdx.z \* blockDim.z;

### **4️⃣ Simple analogy**

* 1D → row of rooms, each with workers
* 2D → a floor with rows and columns of rooms
* 3D → a building with multiple floors, rows, and columns of rooms

If you want, I can **draw a simple diagram** showing **threads inside blocks inside a 2D or 3D grid**—it makes X, Y, Z dimensions very easy to visualize.

Do you want me to do that?

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

## **Step 3: Simple Array Addition**

**Goal:** Add two arrays on GPU using 1 thread per element.

#include <stdio.h>

#include <cuda\_runtime.h>

#define N 5

\_\_global\_\_ void add(int \*a, int \*b, int \*c) {

int tid = threadIdx.x;

c[tid] = a[tid] + b[tid];

}

int main() {

int a[N] = {1,2,3,4,5};

int b[N] = {10,20,30,40,50};

int c[N];

int \*dev\_a, \*dev\_b, \*dev\_c;

cudaMalloc((void\*\*)&dev\_a, N\*sizeof(int));

cudaMalloc((void\*\*)&dev\_b, N\*sizeof(int));

cudaMalloc((void\*\*)&dev\_c, N\*sizeof(int));

cudaMemcpy(dev\_a, a, N\*sizeof(int), cudaMemcpyHostToDevice);

cudaMemcpy(dev\_b, b, N\*sizeof(int), cudaMemcpyHostToDevice);

add<<<1,N>>>(dev\_a, dev\_b, dev\_c); // 1 block, N threads

cudaDeviceSynchronize();

cudaMemcpy(c, dev\_c, N\*sizeof(int), cudaMemcpyDeviceToHost);

for(int i=0; i<N; i++)

printf("%d + %d = %d\n", a[i], b[i], c[i]);

cudaFree(dev\_a);

cudaFree(dev\_b);

cudaFree(dev\_c);

return 0;

}

### **Step-by-step explanation:**

1. Allocate GPU memory using cudaMalloc
2. Copy data from CPU → GPU using cudaMemcpy
3. Launch kernel with **N threads**
4. GPU threads add elements in parallel
5. Copy result back GPU → CPU
6. Free GPU memory

**Expected output:**

1 + 10 = 11

2 + 20 = 22

3 + 30 = 33

4 + 40 = 44

5 + 50 = 55

## **Step 4: Scaling to Many Elements**

* For large arrays (N=1000000), you can’t use 1 block → **need multiple blocks and threads per block**:

int threadsPerBlock = 256;

int blocksPerGrid = (N + threadsPerBlock - 1)/threadsPerBlock;

add<<<blocksPerGrid, threadsPerBlock>>>(dev\_a, dev\_b, dev\_c);

* Each thread calculates:

int tid = threadIdx.x + blockIdx.x \* blockDim.x;

if(tid < N)

c[tid] = a[tid] + b[tid];

* This ensures **all elements are processed**, even if N is not divisible by threadsPerBlock.

## **✅ Step 5: Key Tips**

1. Always **check memory allocation**: if(dev\_a==NULL)
2. Use cudaDeviceSynchronize() after kernel launch for debugging.
3. Use cudaMemcpy carefully: Host → Device, Device → Host.
4. Avoid large stack arrays on CPU; use malloc for big arrays.
5. Start with small arrays to **verify correctness**.

I can continue with **Step 6: Matrix Multiplication on GPU**, which introduces **2D thread blocks, shared memory, and more advanced CUDA features**.

Do you want me to continue to **matrix multiplication next**?

Absolutely! Let’s write a **fully commented and detailed CUDA matrix multiplication program**, step by step, explaining **every line and concept**. This will help you understand **how threads, blocks, memory, and computation work on the GPU**.

We’ll also include **2D thread blocks, grid mapping, and comments for every operation**.

#include <stdio.h>

#include <cuda\_runtime.h>

// -----------------------------

// Matrix size

// -----------------------------

#define N 3 // Size of NxN matrix

// -----------------------------

// CUDA Kernel Function

// -----------------------------

// This kernel multiplies two matrices A and B to produce matrix C.

// Each thread computes one element C[row][col].

// The matrices are stored in 1D arrays (row-major order).

// -----------------------------

\_\_global\_\_ void matMul(int \*A, int \*B, int \*C, int n) {

// -----------------------------

// Calculate row and column index of the element

// -----------------------------

int row = threadIdx.y + blockIdx.y \* blockDim.y; // Row index of C

int col = threadIdx.x + blockIdx.x \* blockDim.x; // Column index of C

// -----------------------------

// Boundary check

// -----------------------------

// Ensure threads do not access out-of-bound elements

if (row < n && col < n) {

int sum = 0; // Temporary variable to store sum

// -----------------------------

// Perform dot product of row from A and column from B

// -----------------------------

for (int k = 0; k < n; k++) {

// A[row\*n + k] → element in row 'row', column 'k' of A

// B[k\*n + col] → element in row 'k', column 'col' of B

sum += A[row \* n + k] \* B[k \* n + col];

}

// Store the result in C[row][col]

C[row \* n + col] = sum;

}

}

int main() {

// -----------------------------

// Declare and initialize CPU matrices

// -----------------------------

// Using 1D arrays in row-major order

int A[N\*N] = {1, 2, 3,

4, 5, 6,

7, 8, 9};

int B[N\*N] = {9, 8, 7,

6, 5, 4,

3, 2, 1};

int C[N\*N] = {0}; // Result matrix

// -----------------------------

// Declare GPU pointers

// -----------------------------

int \*dev\_A, \*dev\_B, \*dev\_C;

// -----------------------------

// Allocate memory on GPU

// -----------------------------

cudaMalloc((void\*\*)&dev\_A, N\*N\*sizeof(int)); // For matrix A

cudaMalloc((void\*\*)&dev\_B, N\*N\*sizeof(int)); // For matrix B

cudaMalloc((void\*\*)&dev\_C, N\*N\*sizeof(int)); // For matrix C

// -----------------------------

// Copy data from CPU to GPU

// -----------------------------

cudaMemcpy(dev\_A, A, N\*N\*sizeof(int), cudaMemcpyHostToDevice);

cudaMemcpy(dev\_B, B, N\*N\*sizeof(int), cudaMemcpyHostToDevice);

// -----------------------------

// Define thread block and grid dimensions

// -----------------------------

// Threads per block (2D)

dim3 threadsPerBlock(2, 2); // 2 threads in x and 2 threads in y → 4 threads per block

// Grid dimensions (enough blocks to cover all elements)

dim3 blocksPerGrid((N + threadsPerBlock.x - 1) / threadsPerBlock.x,

(N + threadsPerBlock.y - 1) / threadsPerBlock.y);

// -----------------------------

// Launch the CUDA kernel

// -----------------------------

matMul<<<blocksPerGrid, threadsPerBlock>>>(dev\_A, dev\_B, dev\_C, N);

// -----------------------------

// Wait for GPU to finish execution

// -----------------------------

cudaDeviceSynchronize();

// -----------------------------

// Copy result matrix C from GPU to CPU

// -----------------------------

cudaMemcpy(C, dev\_C, N\*N\*sizeof(int), cudaMemcpyDeviceToHost);

// -----------------------------

// Print the result matrix

// -----------------------------

printf("Result Matrix C (A x B):\n");

for (int i = 0; i < N; i++) {

for (int j = 0; j < N; j++) {

printf("%d ", C[i\*N + j]); // Row-major indexing

}

printf("\n");

}

// -----------------------------

// Free GPU memory

// -----------------------------

cudaFree(dev\_A);

cudaFree(dev\_B);

cudaFree(dev\_C);

return 0;

}

### **Detailed Explanation of Each Section**

1. **Kernel Function (\_\_global\_\_)**
   * Runs on GPU.
   * Each thread computes **one element** of C.
   * Thread position determined by threadIdx and blockIdx.
   * Boundary check prevents accessing elements outside matrix size.
2. **Thread and Block Mapping**
   * threadIdx.x/y → Thread index inside block (0,1,…)
   * blockIdx.x/y → Block index inside grid (0,1,…)
   * Global element index: row = threadIdx.y + blockIdx.y\*blockDim.y
3. **Matrix Multiplication Logic**
   * Each thread performs **dot product of a row and a column**.
   * Sum stored in C[row \* n + col].
4. **Memory Allocation**
   * CPU → GPU using cudaMalloc.
   * GPU memory must be freed after use using cudaFree.
5. **Data Transfer**
   * Host → Device: cudaMemcpyHostToDevice
   * Device → Host: cudaMemcpyDeviceToHost
6. **Grid & Block Calculation**
   * (N + threadsPerBlock - 1) / threadsPerBlock → ensures all elements are covered.
7. **Printing Result**
   * CPU prints matrix after GPU computation.

### **Expected Output**

Result Matrix C (A x B):

30 24 18

84 69 54

138 114 90

✅ **Next Steps**

1. Use **larger matrices** like 512x512 or 1024x1024.
2. Optimize using **shared memory** for faster GPU performance.
3. Learn **thread synchronization**, **warps**, and **memory coalescing**.

I can now make a **fully optimized matrix multiplication code using shared memory**, explaining **how GPU memory hierarchy and thread cooperation work**, which is much faster for large matrices.

Do you want me to do that next?

Perfect! Now let’s move to **Step 7: Optimized Matrix Multiplication using Shared Memory**.

This is the standard way to **speed up matrix multiplication on GPU**, especially for large matrices. I’ll explain **every concept, line, and memory usage**.

## **Why Shared Memory?**

* **Global memory**: accessible by all threads but **slow** (~400-600 cycles)
* **Shared memory**: small (~48 KB per block), **fast** (~1 cycle), shared among threads in a block

**Idea:**

* Each thread block loads a **tile** of A and B into **shared memory**.
* Threads compute partial results using **fast shared memory**.
* Reduces **global memory access**, boosting performance.

## **Code: Optimized Matrix Multiplication using Shared Memory**

#include <stdio.h>

#include <cuda\_runtime.h>

#define N 8 // Matrix size (NxN)

#define TILE\_SIZE 2 // Tile size (threads per block in x and y)

\_\_global\_\_ void matMulShared(int \*A, int \*B, int \*C, int n) {

// Shared memory tiles for A and B

\_\_shared\_\_ int tileA[TILE\_SIZE][TILE\_SIZE];

\_\_shared\_\_ int tileB[TILE\_SIZE][TILE\_SIZE];

// Thread row and column within the block

int tx = threadIdx.x;

int ty = threadIdx.y;

// Global row and column index in the matrix

int row = blockIdx.y \* TILE\_SIZE + ty;

int col = blockIdx.x \* TILE\_SIZE + tx;

int sum = 0;

// Loop over all tiles

for (int m = 0; m < (n + TILE\_SIZE - 1)/TILE\_SIZE; m++) {

// Load tile from global memory into shared memory

if (row < n && (m\*TILE\_SIZE + tx) < n)

tileA[ty][tx] = A[row \* n + m\*TILE\_SIZE + tx];

else

tileA[ty][tx] = 0;

if (col < n && (m\*TILE\_SIZE + ty) < n)

tileB[ty][tx] = B[(m\*TILE\_SIZE + ty) \* n + col];

else

tileB[ty][tx] = 0;

\_\_syncthreads(); // Ensure all threads have loaded tiles

// Multiply tile elements

for (int k = 0; k < TILE\_SIZE; k++)

sum += tileA[ty][k] \* tileB[k][tx];

\_\_syncthreads(); // Wait before next tile load

}

// Write result to global memory

if (row < n && col < n)

C[row \* n + col] = sum;

}

int main() {

int A[N\*N], B[N\*N], C[N\*N];

// Initialize matrices

for (int i = 0; i < N\*N; i++) {

A[i] = i+1;

B[i] = N\*N - i;

C[i] = 0;

}

int \*dev\_A, \*dev\_B, \*dev\_C;

cudaMalloc((void\*\*)&dev\_A, N\*N\*sizeof(int));

cudaMalloc((void\*\*)&dev\_B, N\*N\*sizeof(int));

cudaMalloc((void\*\*)&dev\_C, N\*N\*sizeof(int));

cudaMemcpy(dev\_A, A, N\*N\*sizeof(int), cudaMemcpyHostToDevice);

cudaMemcpy(dev\_B, B, N\*N\*sizeof(int), cudaMemcpyHostToDevice);

dim3 threadsPerBlock(TILE\_SIZE, TILE\_SIZE);

dim3 blocksPerGrid((N + TILE\_SIZE -1)/TILE\_SIZE, (N + TILE\_SIZE -1)/TILE\_SIZE);

matMulShared<<<blocksPerGrid, threadsPerBlock>>>(dev\_A, dev\_B, dev\_C, N);

cudaDeviceSynchronize();

cudaMemcpy(C, dev\_C, N\*N\*sizeof(int), cudaMemcpyDeviceToHost);

// Print result

printf("Result Matrix C (Optimized with Shared Memory):\n");

for (int i=0;i<N;i++){

for(int j=0;j<N;j++)

printf("%d ", C[i\*N + j]);

printf("\n");

}

cudaFree(dev\_A);

cudaFree(dev\_B);

cudaFree(dev\_C);

return 0;

}

## **Step-by-Step Explanation**

### **1. Shared Memory Declaration**

\_\_shared\_\_ int tileA[TILE\_SIZE][TILE\_SIZE];

\_\_shared\_\_ int tileB[TILE\_SIZE][TILE\_SIZE];

* Each block has **fast local memory** for tiles of A and B.
* Reduces repeated **global memory access**.

### **2. Thread Mapping**

int tx = threadIdx.x;

int ty = threadIdx.y;

int row = blockIdx.y \* TILE\_SIZE + ty;

int col = blockIdx.x \* TILE\_SIZE + tx;

* tx, ty → thread index inside block
* row, col → global position in matrix C

### **3. Load Tiles**

tileA[ty][tx] = A[row \* n + m\*TILE\_SIZE + tx];

tileB[ty][tx] = B[(m\*TILE\_SIZE + ty) \* n + col];

* Each thread loads **one element of A and B** into shared memory.
* Boundary check avoids **out-of-bounds access**.

### **4. Synchronize Threads**

\_\_syncthreads();

* Ensures **all threads finish loading tiles** before computation.

### **5. Multiply Tiles**

for (int k=0; k<TILE\_SIZE; k++)

sum += tileA[ty][k] \* tileB[k][tx];

* Each thread computes **partial sum** for its element.

### **6. Repeat for all tiles**

* Loop for (int m = 0; m < (n + TILE\_SIZE -1)/TILE\_SIZE; m++)
* Loads next tile of A and B and adds to sum.

### **7. Write Result**

if (row<n && col<n)

C[row\*n + col] = sum;

* Each thread writes its final sum to global memory.

### **✅ Advantages of Shared Memory**

1. Reduces **global memory access**.
2. Improves **memory coalescing and speed**.
3. Threads cooperate to compute each element **efficiently**.

### **Next Steps**

1. Experiment with **larger TILE\_SIZE**, e.g., 16 or 32 for bigger matrices.
2. Learn **cuda profiling tools** (nvprof or Nsight) to measure speed.
3. Extend this to **non-square matrices** and **multi-dimensional grids**.

I can now make a **visual diagram showing GPU threads, blocks, and shared memory tiles**, so you can **see exactly how threads cooperate to compute the matrix multiplication**.

Do you want me to draw that diagram?

Ah! got it — you want the **visual diagram showing GPU threads, blocks, and shared memory tiles** for matrix multiplication. Let me explain it clearly with a diagrammatic representation.

## **Diagram: GPU Threads, Blocks, and Shared Memory Tiles**

Imagine we have:

* Matrix A and B of size 8x8
* **Tile size = 2x2** (threads per block)

### **1. Grid and Block Layout**

Matrix C (8x8)

+--------------------------------+

| Block(0,0) | Block(0,1) | ... |

|------------+------------+-----|

| Block(1,0) | Block(1,1) | ... |

|------------+------------+-----|

| ... | ... | ... |

+--------------------------------+

* Each **block computes a tile of C**, e.g., 2x2 elements.
* **Threads inside the block** handle each element within the tile.

### **2. Threads inside a Block**

Thread indices inside Block(0,0):

+--------+

| T(0,0) | -> Computes C[0][0]

| T(0,1) | -> Computes C[0][1]

| T(1,0) | -> Computes C[1][0]

| T(1,1) | -> Computes C[1][1]

+--------+

* threadIdx.x → column index within the block
* threadIdx.y → row index within the block

### **3. Shared Memory Tiles**

Shared Memory in Block(0,0)

tileA[2][2] tileB[2][2]

+------+ +------+

| A00 | | B00 |

| A10 | | B10 |

+------+ +------+

* Each block **loads a small tile from global memory** into **shared memory**.
* Threads access this **fast memory** to compute their partial sums.
* Loop over multiple tiles if matrix > tile size.

### **4. Computation Flow**

1. Load **tileA** and **tileB** from global memory to **shared memory**.
2. Each thread computes **partial sum** for its element:

C[row][col] += tileA[rowIndex][k] \* tileB[k][colIndex]

1. Synchronize threads: \_\_syncthreads()
2. Repeat for next tiles until full row/column of C is computed.
3. Write **C[row][col]** back to global memory.

### **5. Summary of Mapping**

|  |  |
| --- | --- |
| **Concept** | **CUDA Representation** |
| Thread | Computes **one element** of C |
| Block | Computes **one tile** of C |
| Grid | Covers entire matrix |
| Shared Memory | Stores **tiles of A & B** for fast access |
| Synchronization | \_\_syncthreads() ensures threads in block finish before next tile |

If you want, I can **draw an actual ASCII-style full 8x8 diagram** showing **all threads, tiles, and memory flow**, which makes the **GPU computation visually clear**.

Do you want me to create that detailed ASCII diagram?

Perfect! Let’s create a **full ASCII-style diagram** showing **GPU threads, blocks, shared memory tiles, and computation flow** for an 8×8 matrix with **tile size = 2×2**. This will make it visually clear how threads and blocks work.

## **Matrix C (8x8) divided into 2x2 tiles**

Matrix C (Global Memory)

+------+------+------+------+

| C00 C01 | C02 C03 | ...

| C10 C11 | C12 C13 | ...

+------+------+------+------+

| C20 C21 | C22 C23 | ...

| C30 C31 | C32 C33 | ...

+------+------+------+------+

| ... | ... | ...

* Each **block computes a tile** of size 2x2.
* Number of blocks in **x-direction** = 8/2 = 4
* Number of blocks in **y-direction** = 8/2 = 4
* Total blocks = 4 × 4 = 16 blocks

## **Block Layout Example: Block(0,0)**

Block(0,0) computes tile C[0:1][0:1]

+-------+-------+

| T(0,0)| T(0,1)| -> Thread row = threadIdx.y, col = threadIdx.x

+-------+-------+

| T(1,0)| T(1,1)|

+-------+-------+

* Each thread computes **one element** of the tile:
  + T(0,0) → C[0][0]
  + T(0,1) → C[0][1]
  + T(1,0) → C[1][0]
  + T(1,1) → C[1][1]

## **Shared Memory Tiles**

Shared Memory (Fast) inside Block(0,0)

tileA[2][2] tileB[2][2]

+-----+-----+ +-----+-----+

| A00 | A01 | | B00 | B01 |

+-----+-----+ +-----+-----+

| A10 | A11 | | B10 | B11 |

+-----+-----+ +-----+-----+

* Threads in the block **load elements** from global memory into shared memory.
* Computation is done using shared memory to **reduce slow global memory access**.

## **Computation Flow in Each Block**

Step 1: Load tileA and tileB into shared memory

- Each thread loads 1 element of tileA and tileB

Step 2: Compute partial sum for C elements

- For each k in tile: sum += tileA[threadRow][k] \* tileB[k][threadCol]

Step 3: \_\_syncthreads()

- Ensure all threads finished computing before loading next tile

Step 4: Repeat steps 1-3 for next tiles in row/column

Step 5: Write final sum to C[row][col] in global memory

## **Global View**

Grid Layout (Blocks)

+--------+--------+--------+--------+

| B(0,0) | B(0,1) | B(0,2) | B(0,3) |

+--------+--------+--------+--------+

| B(1,0) | B(1,1) | B(1,2) | B(1,3) |

+--------+--------+--------+--------+

| B(2,0) | B(2,1) | B(2,2) | B(2,3) |

+--------+--------+--------+--------+

| B(3,0) | B(3,1) | B(3,2) | B(3,3) |

+--------+--------+--------+--------+

* Each **block contains 2x2 threads** → computes a 2x2 tile of C.
* Threads inside a block access **shared memory tiles of A and B**.
* Multiple blocks cover **entire matrix**.

✅ **Key Takeaways**

1. **Thread = 1 element of C**
2. **Block = 1 tile of C**
3. **Grid = All blocks covering C**
4. **Shared Memory = Tile of A and B loaded for fast computation**
5. **\_\_syncthreads() ensures correct synchronization within block**

If you want, I can **also make a step-by-step "animation-style" diagram** showing:

1. Threads loading tiles
2. Computing partial sums
3. Moving to next tile

This will **simulate exactly how GPU multiplies large matrices in tiles**.

Do you want me to make that animation-style step diagram next?