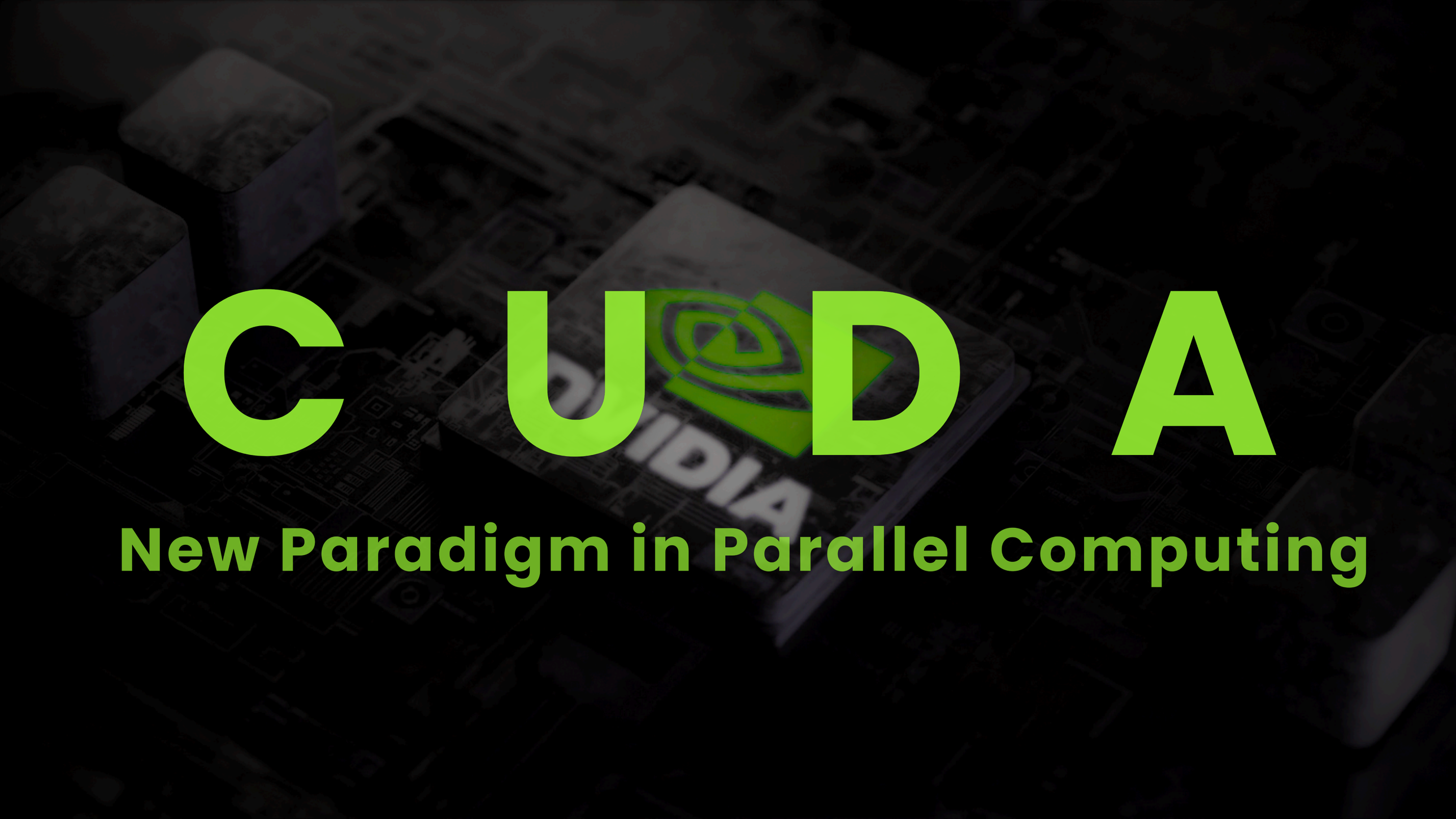

C U D A
New Paradigm in Parallel Computing

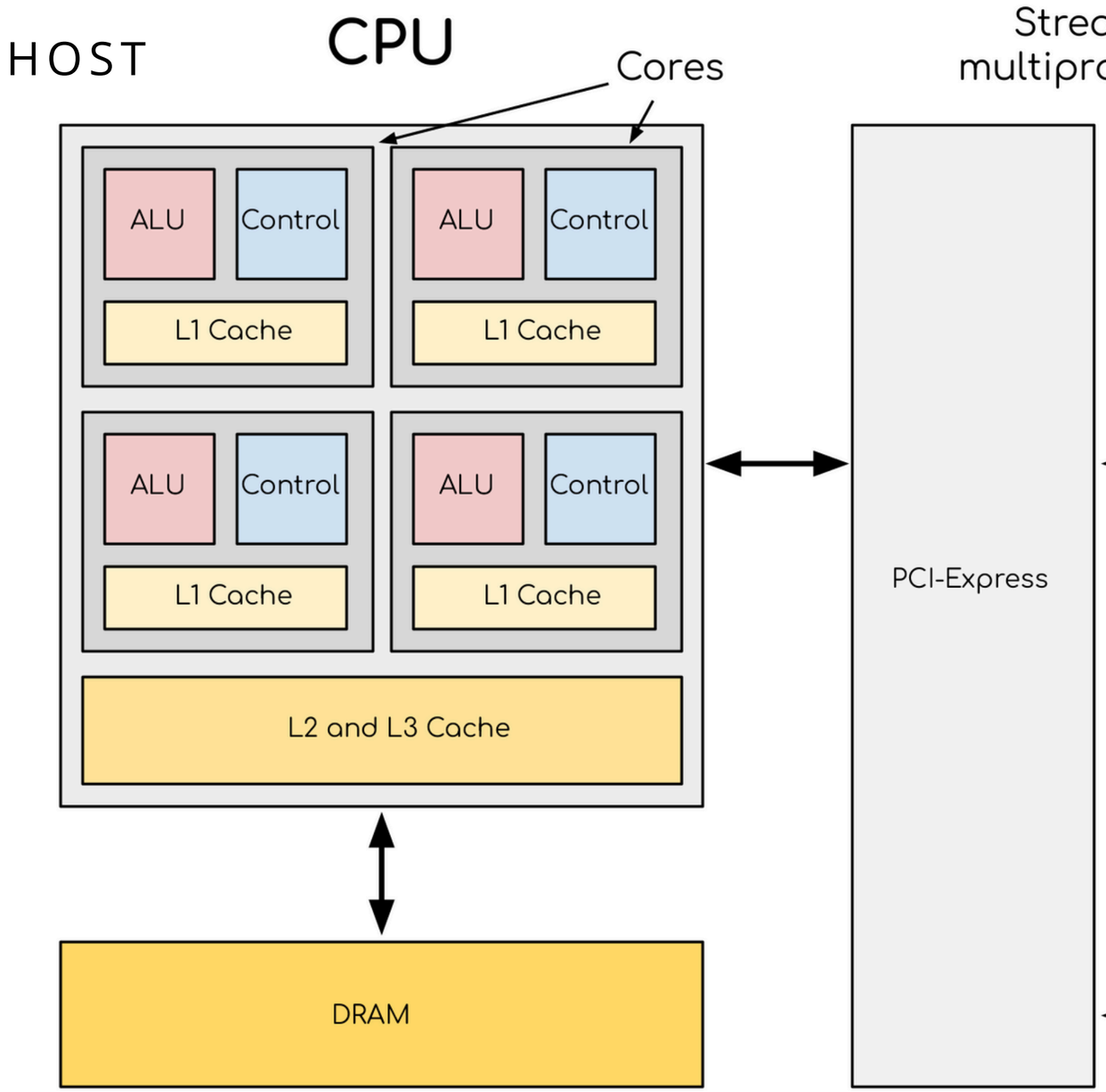
Parallel Computation Before GPUs

Single Core - Multi Threaded single core manages multiple threads in a round robin fashion

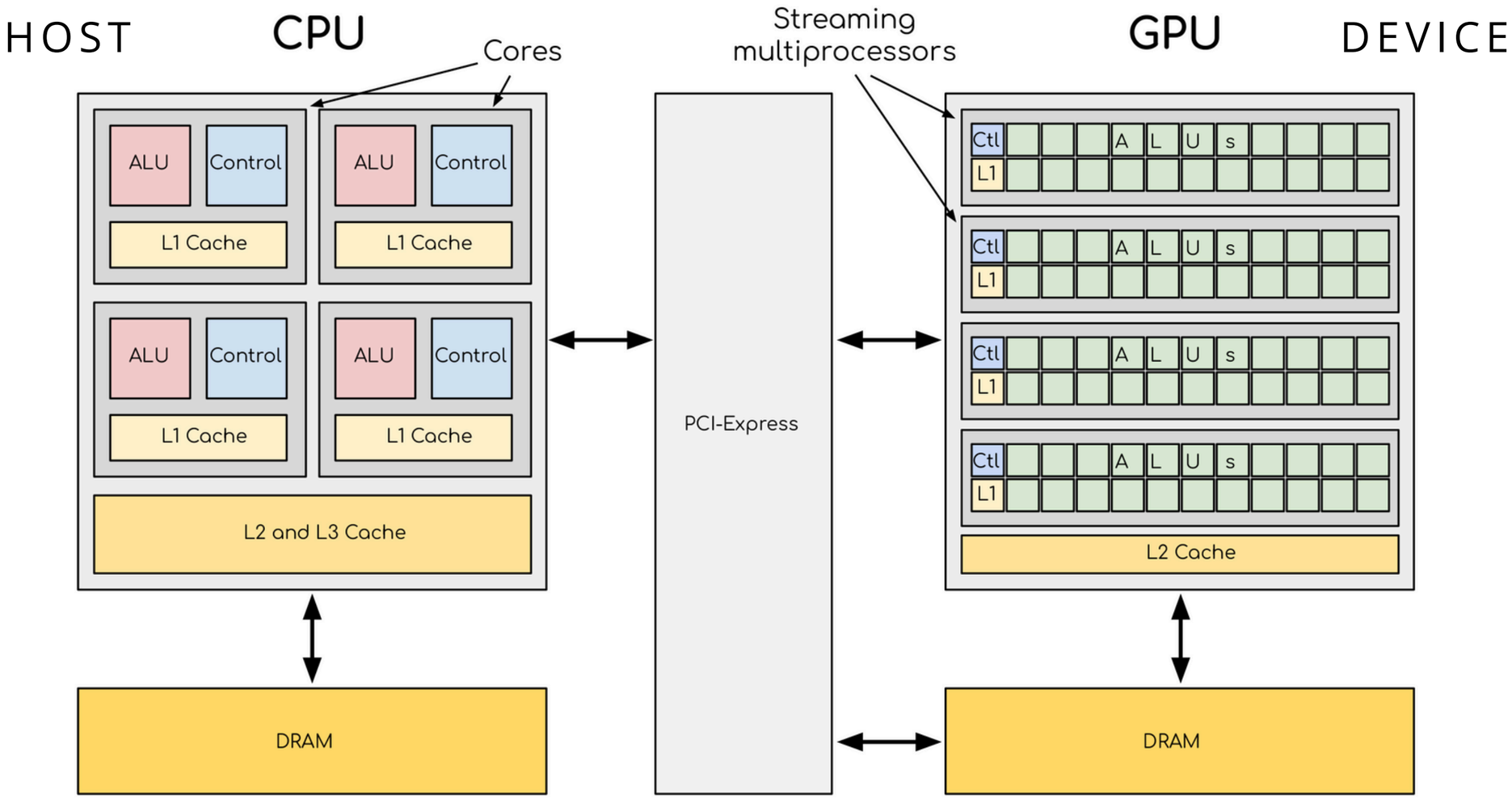
Large Cache size used for managing big **sequential processes**

Complex Arithmetic Logic Unit to perform complex tasks

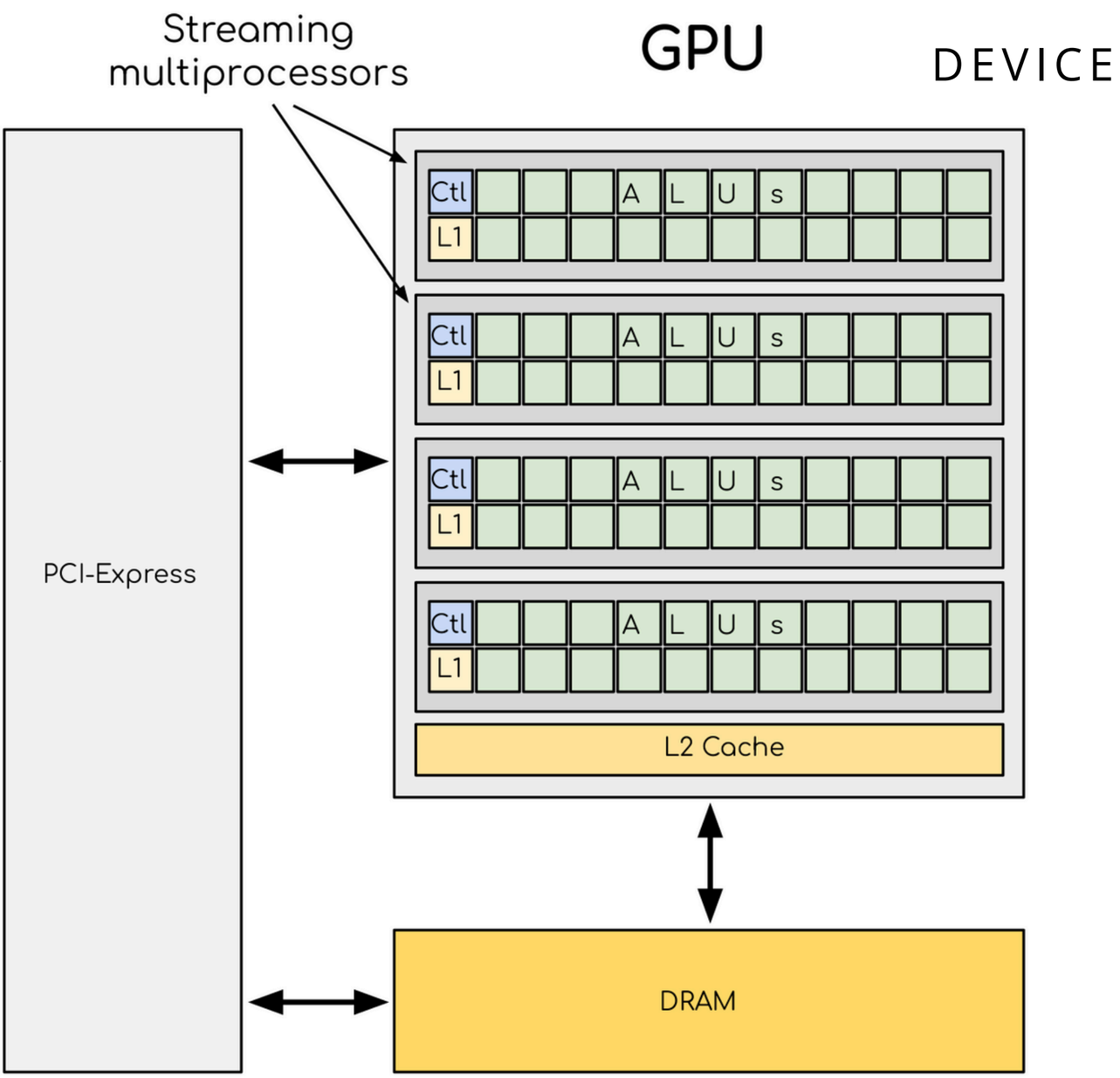
Simpler architecture to code and distribute work



What is the need for GPU ?



To execute small identical work in parallel

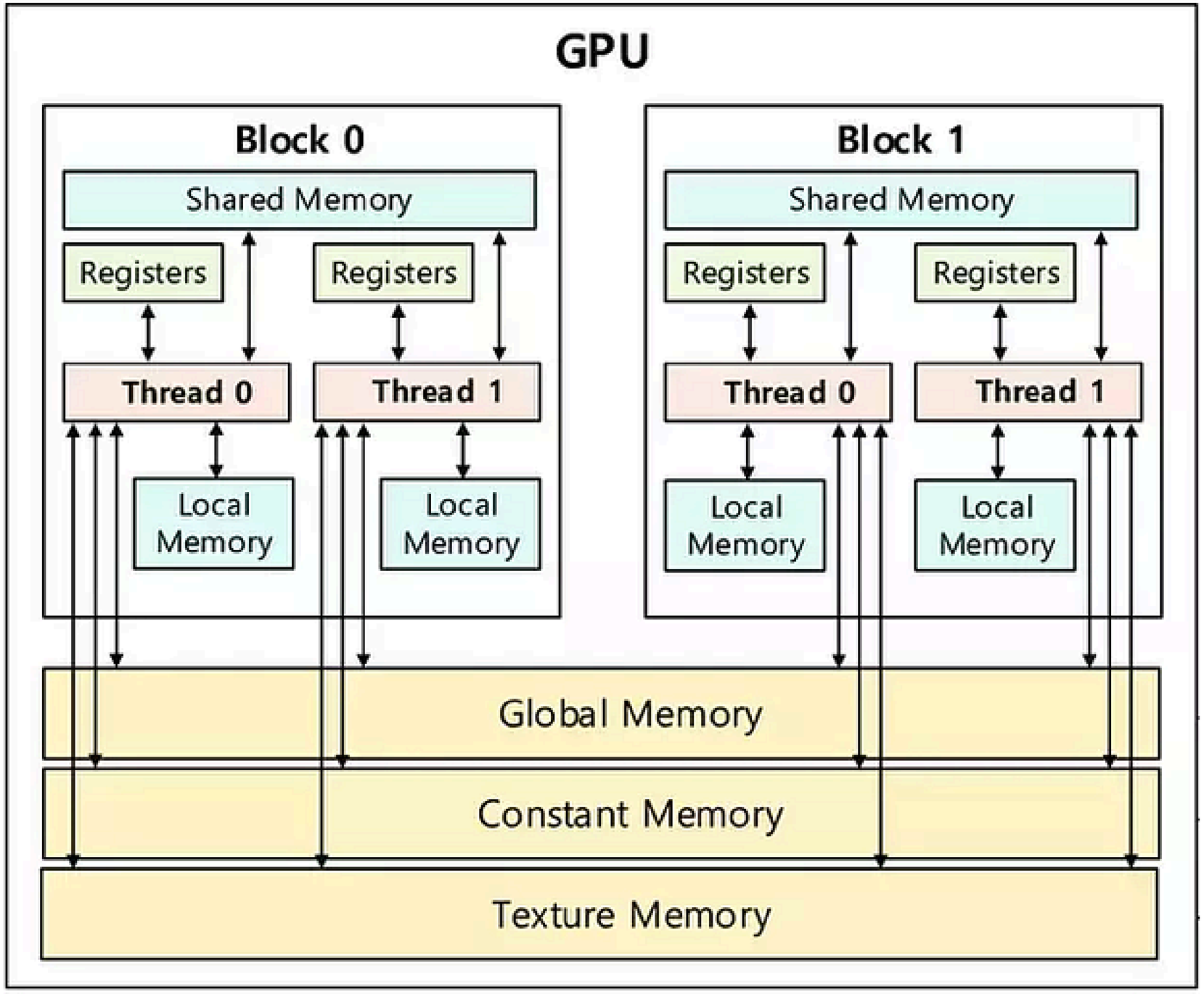


Parallel Computation After GPUs

Large number of **Cuda Cores** to execute large number of threads.

Less Powerful ALU ensuring no computation and power wastage

Multi Level Storage to ensure low latency in data access


GPU
Block 0
Shared Memory
Registers
Registers
Thread 0
Thread 1
Local Memory
Local Memory
Block 1
Shared Memory
Registers
Registers
Thread 0
Thread 1
Local Memory
Local Memory
Global Memory
Constant Memory
Texture Memory

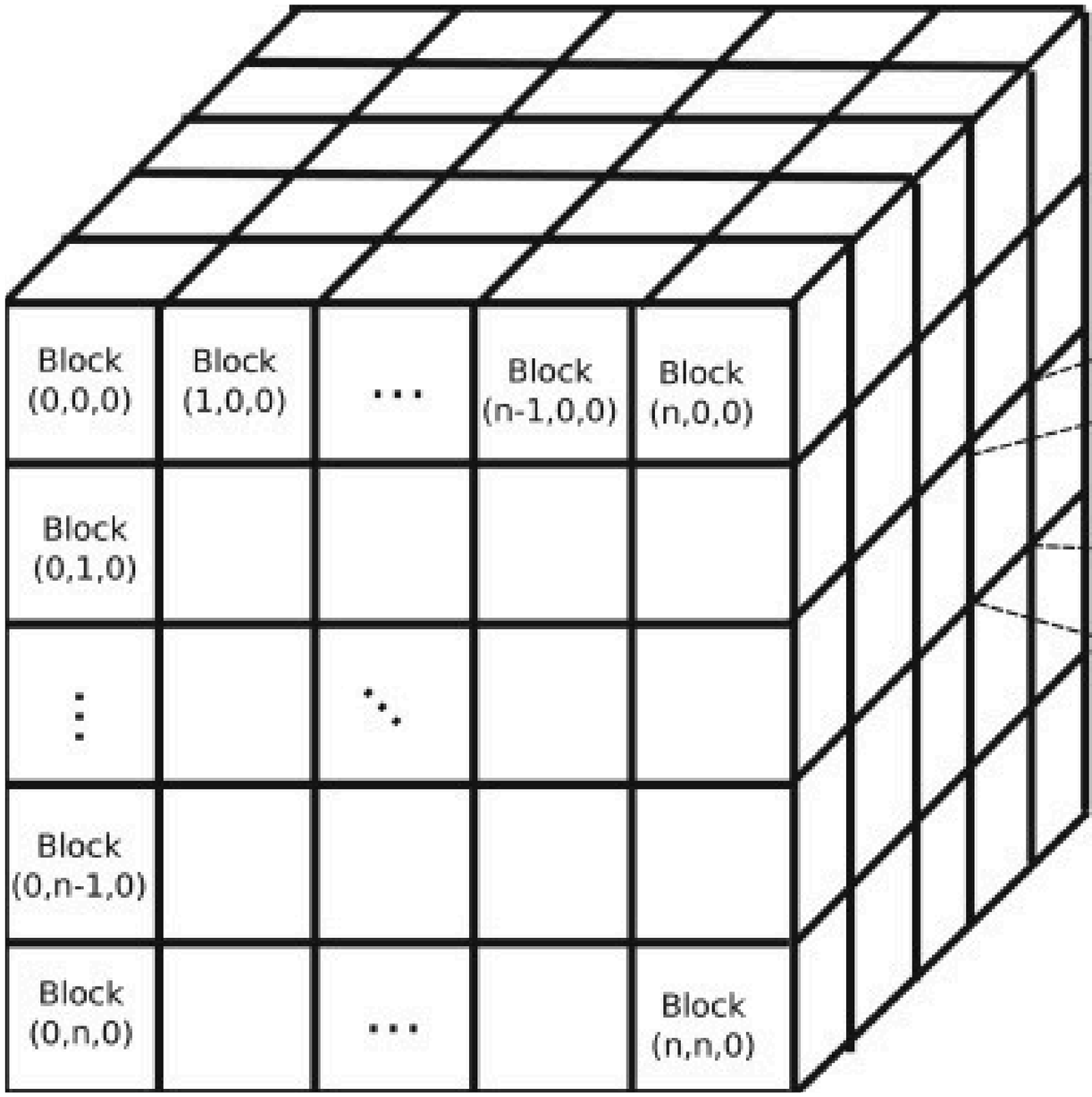
Basic of CUDA Programming

Kernel is a function executed when control of the code is handed from CPU to GPU.

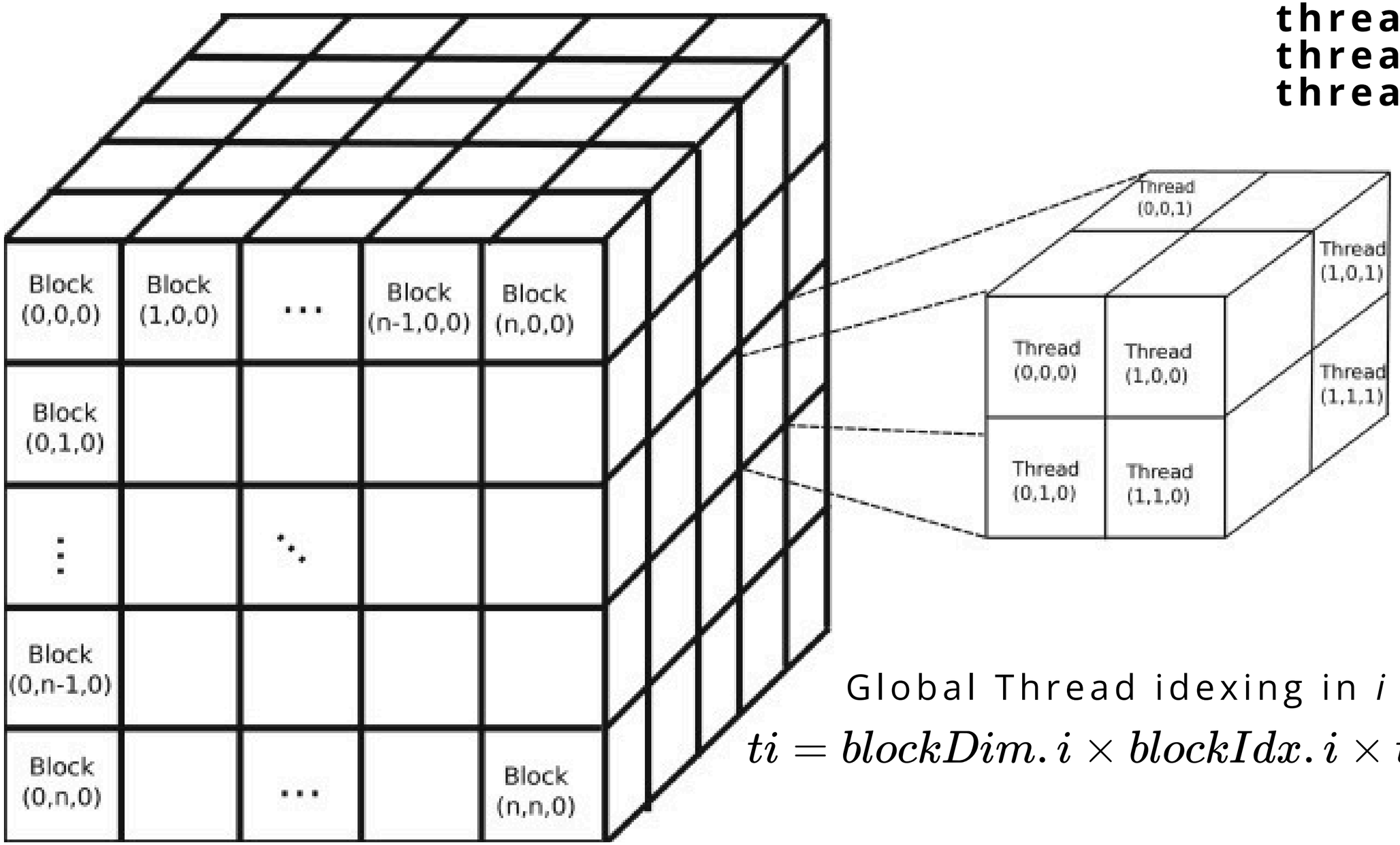
Each Kernel launches a **GRID** which is 3D arrangement of **Blocks**

Each block has 3D arrangement of **Threads**

blockIdx.x
blockIdx.y
blockIdx.z



threadIdx.x
threadIdx.y
threadIdx.z

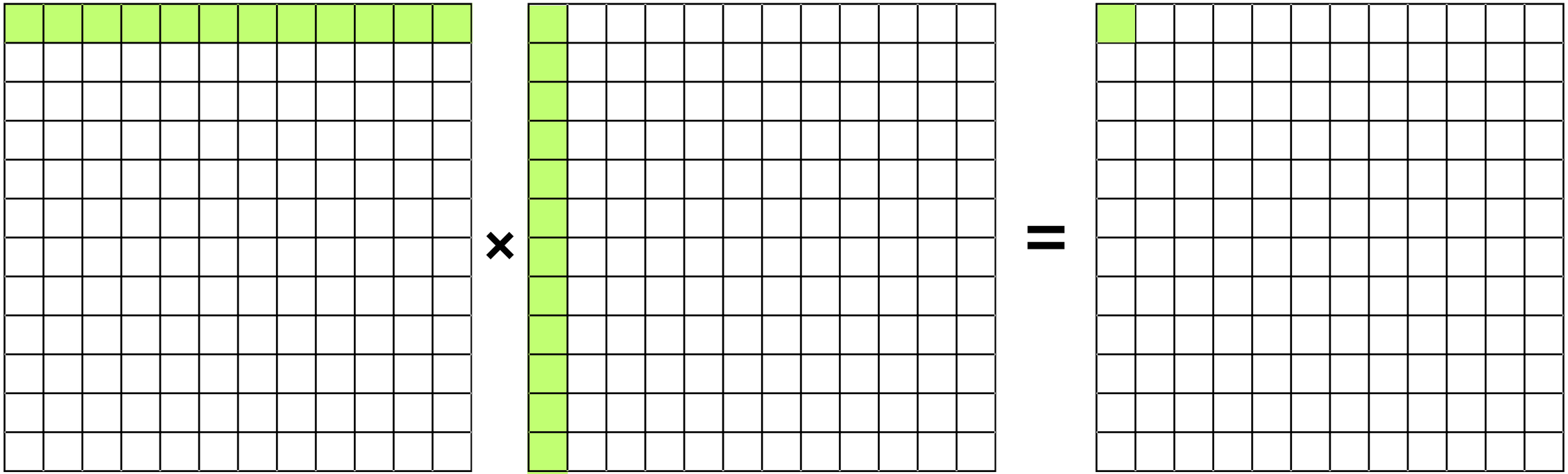


Global Thread idexing in *i* dimension

$$ti = blockDim.i \times blockIdx.i \times threadIdx.i$$

Best Example to Start with Parallelism

Matrix Multiplication

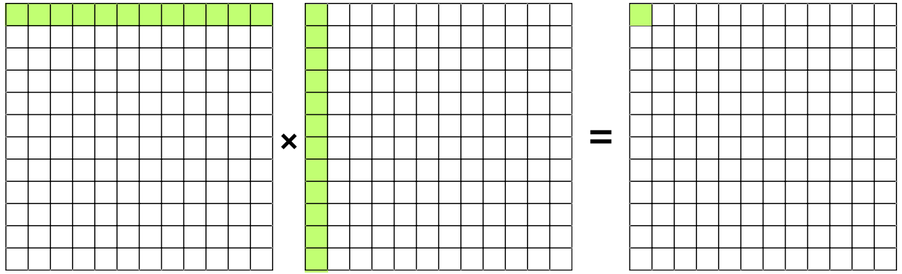


each Thread does calculation for the specific Element,

```
void serialMatMul(
    int *arr1,
    int *arr2,
    int *arr3,
    int r1, int c1,
    int r2, int c2
) {
    for (int i = 0; i < r1; i++) {
        for (int j = 0; j < c2; j++) {
            int sum = 0;

            for (int k = 0; k < c1; k++)
                sum += arr1[i * c1 + k] + arr2[k * c2 + j];

            arr3[i * c2 + k] = sum;
        }
    }
}
```



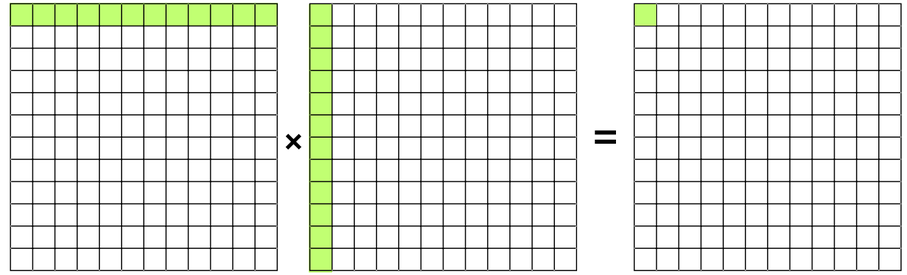
$$O\left(n^3\right)$$

$$C\left[i\right]\left[j\right] = \sum_{k}^{n} = A\left[i\right]\left[k\right] \times B\left[k\right]\left[j\right]$$

```cpp
__global__ void kernalMatMul(
    int *a, int *b,int *c,
    int r1, int c1,
    int r2, int c2
) {

// local to global thread indexing
    int tidx = blockIdx.x * blockDim.x + threadIdx.x;
    int tidy = blockIdx.y * blockDim.y + threadIdx.y;

    if (tidx < r1 && tidy < c2) {
        int sum = 0;
        for (int i = 0; i < c1; i++) {
            sum += a[tidx * c1 + i] * b[i * c2 + tidy];
        }
        c[tidx * c2 + tidy] = sum;
    }
}
```



$$C[tx][ty] = \sum_{k}^{n} = A[tx][k] \times B[k][ty]$$

$$O(n)$$

Host Pointer pointing to Device

```
 // declare pointers for the Device variables
int *d_a;
int *d_b;
int *d_c;
```

Memory Allocation on Device

```
// Device variables memory allocation
cudaMalloc(&d_a, sizeof(int) * ROW1 * K);
cudaMalloc(&d_b, sizeof(int) * K * COL2);
cudaMalloc(&d_c, sizeof(int) * ROW1 * COL2);
```

Host to Device Data Transfer

```
// Data copying from Host to Device
cudaMemcpy(d_a, h_a, sizeof(int) * ROW1 * K, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, h_b, sizeof(int) * K * COL2, cudaMemcpyHostToDevice);
```

Grid and Block Dimension

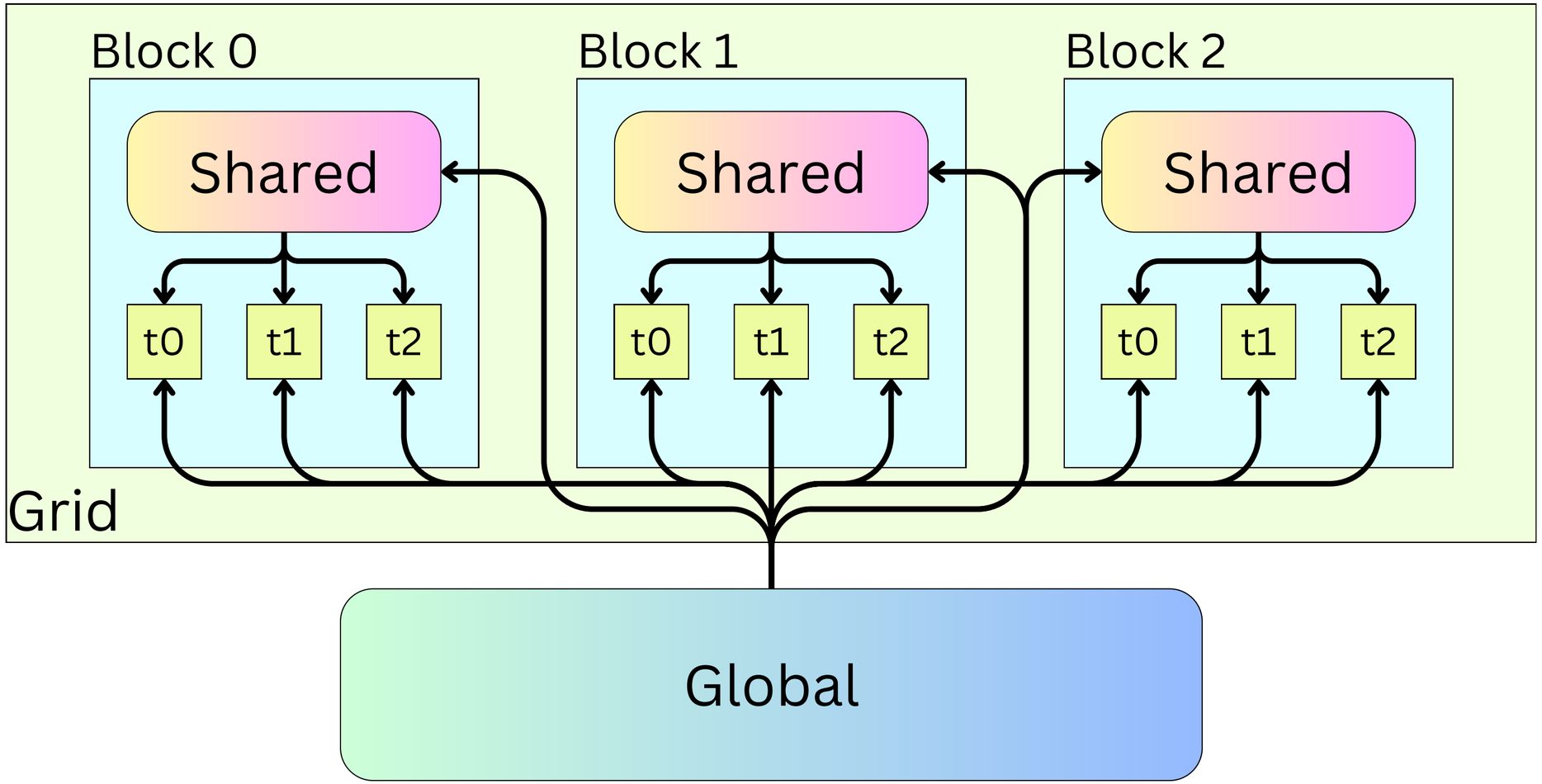
```
// declare Kernel Dimension
dim3 gridDim(1, 1, 1);
dim3 blockDim(ROW1, COL2, 1);
```

```
// Kernel Launch
kernalMatMul <<< gridDim, blockDim >>> (d_a, d_b, d_c, ROW1, K, K, COL2);
cudaDeviceSynchronize();
```

Device to Host Data Transfer

```
// Data copying from Device to Host
cudaMemcpy(p_h_c, d_c, sizeof(int) * ROW1 * COL2, cudaMemcpyDeviceToHost);
```

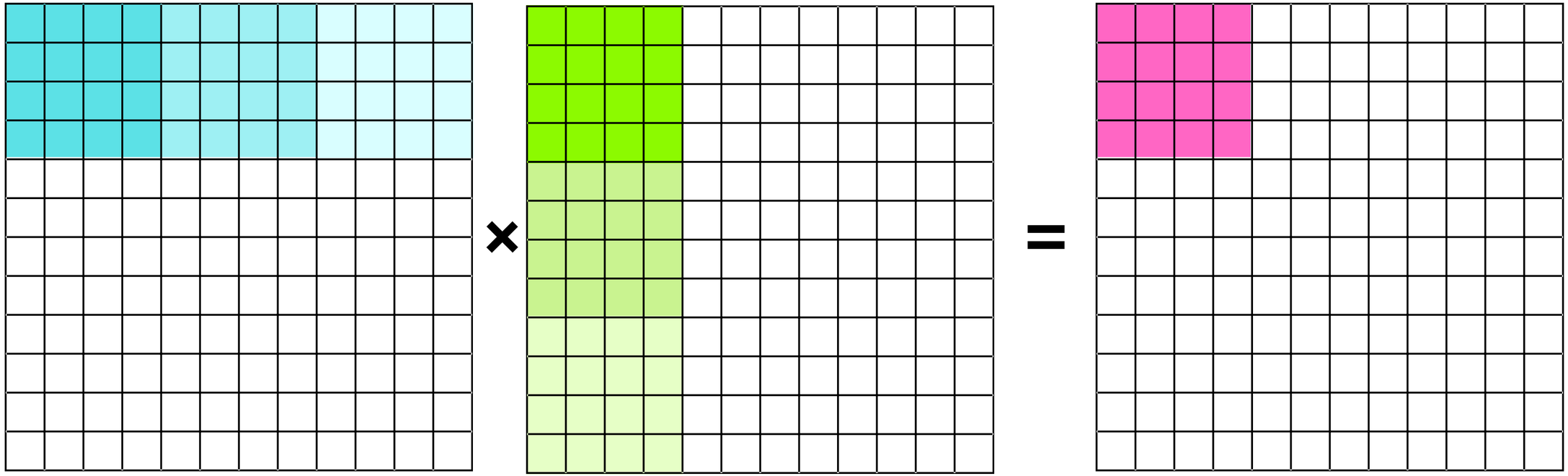
each Thread fetches required data from Global & writes back to Global



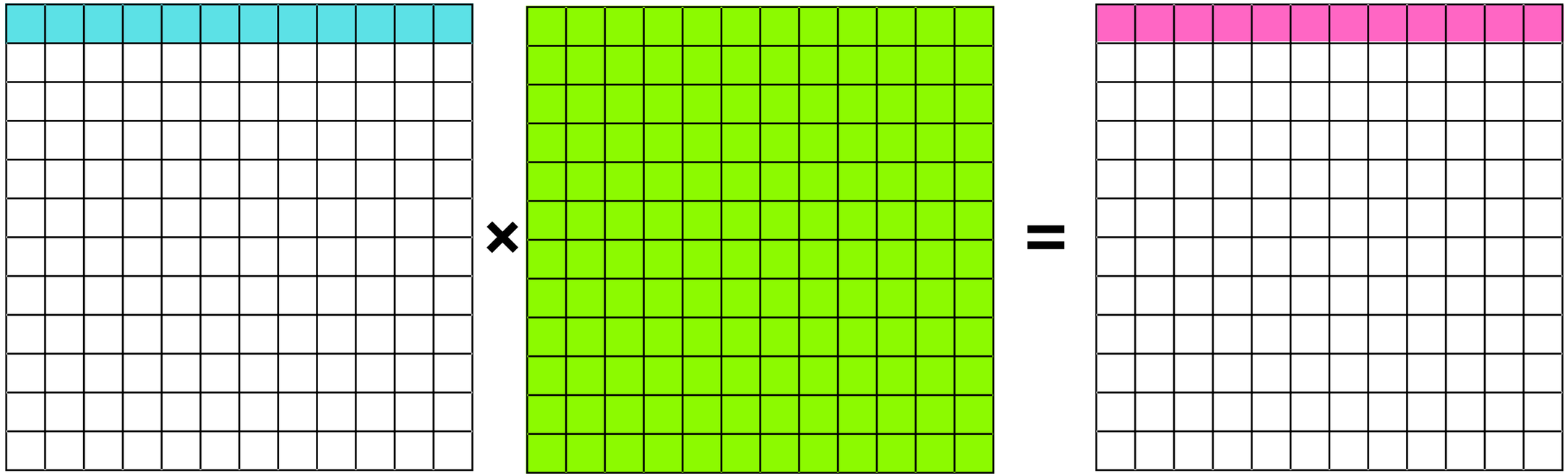
But Global is very slow

Optimized Parallelism with **Tiled Matrix Multiplication**

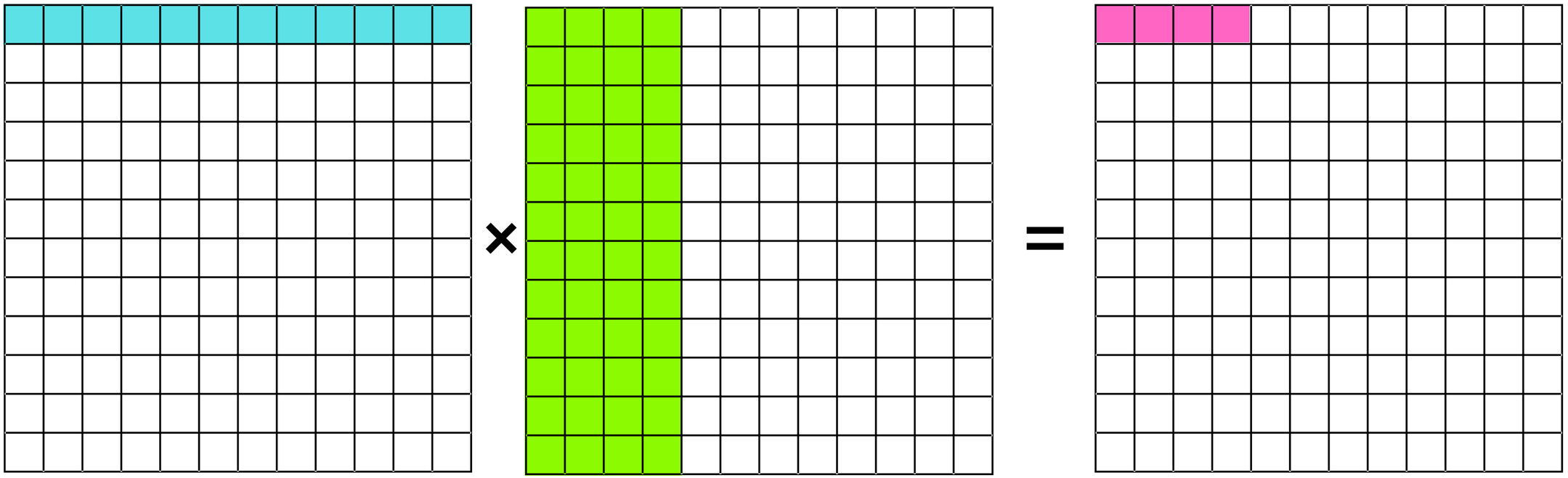
Basic Idea is to perform calculation in form of TILES
along with
Copying data for Faster Retreival



A[*i*][:] B[:][:] C[*i*][:]

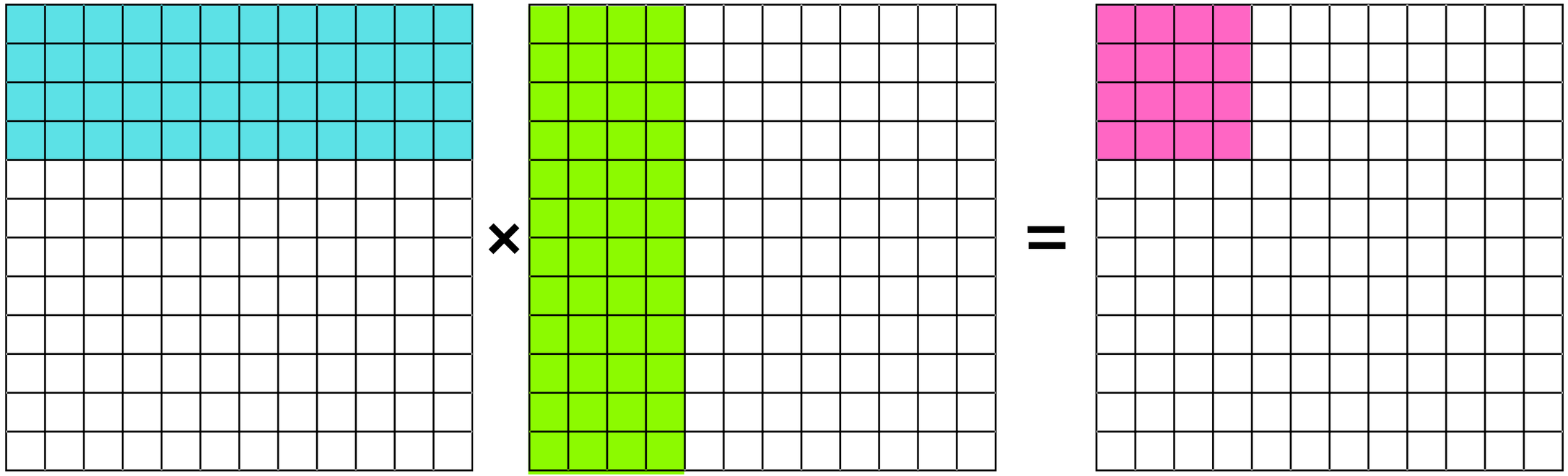


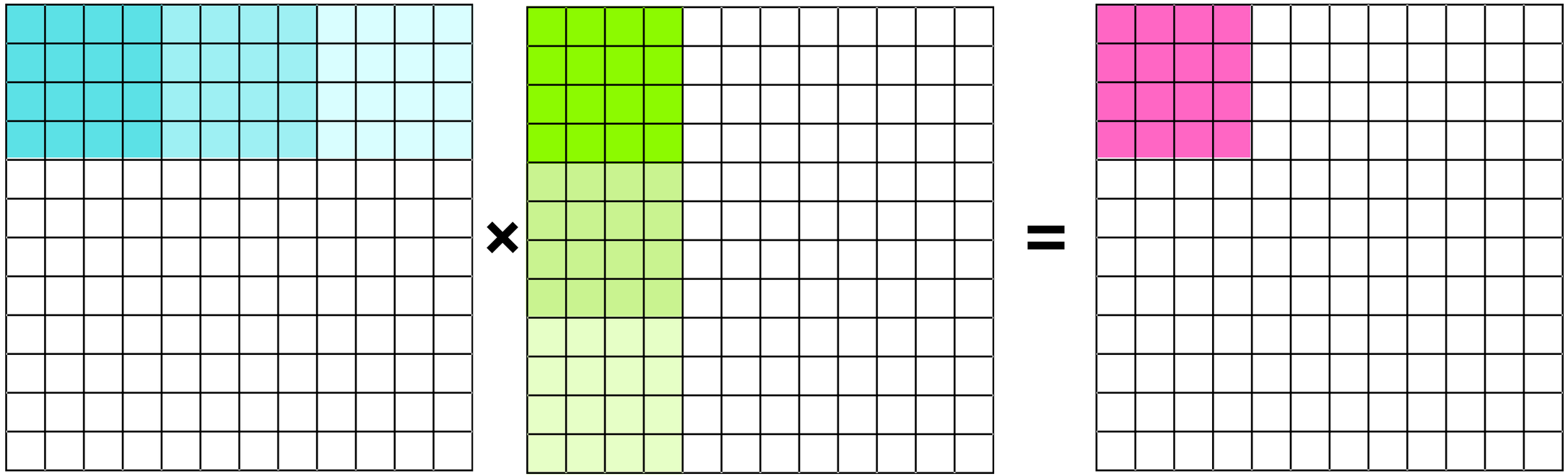
A[*i*][:] B[:][:*T*] C[*i*][:*T*]

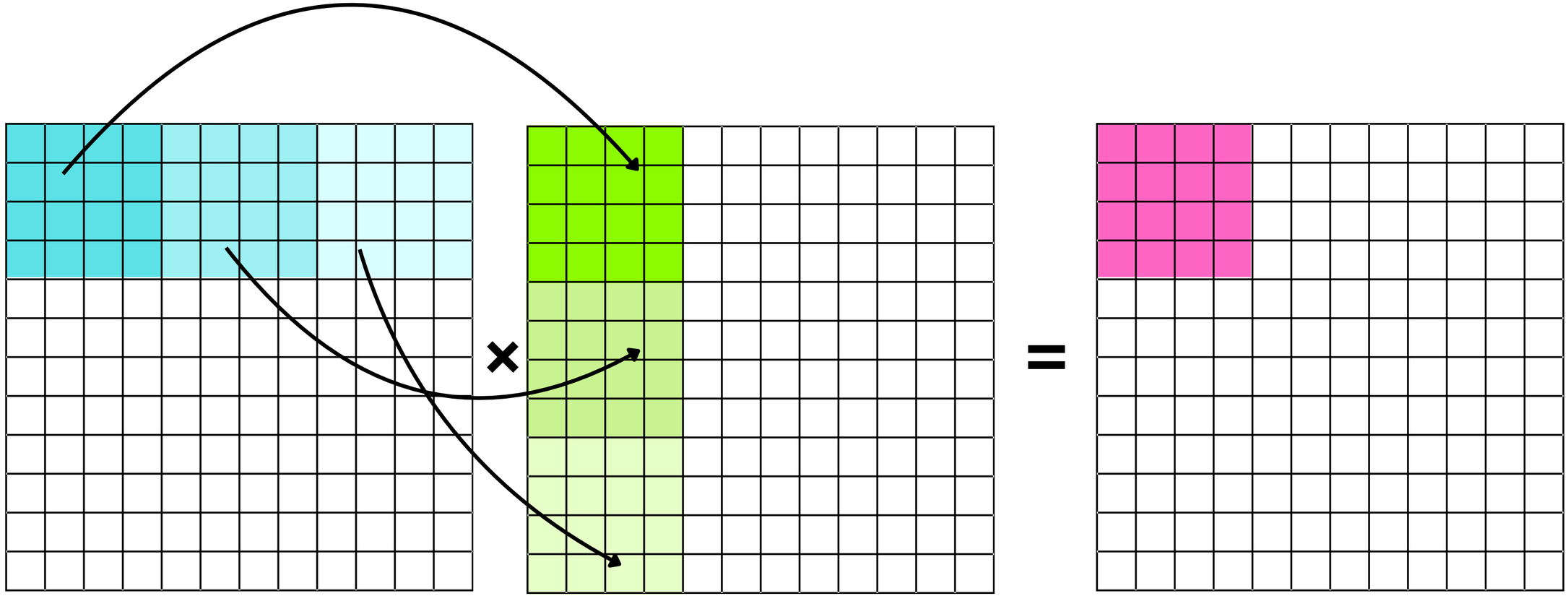


O(n)

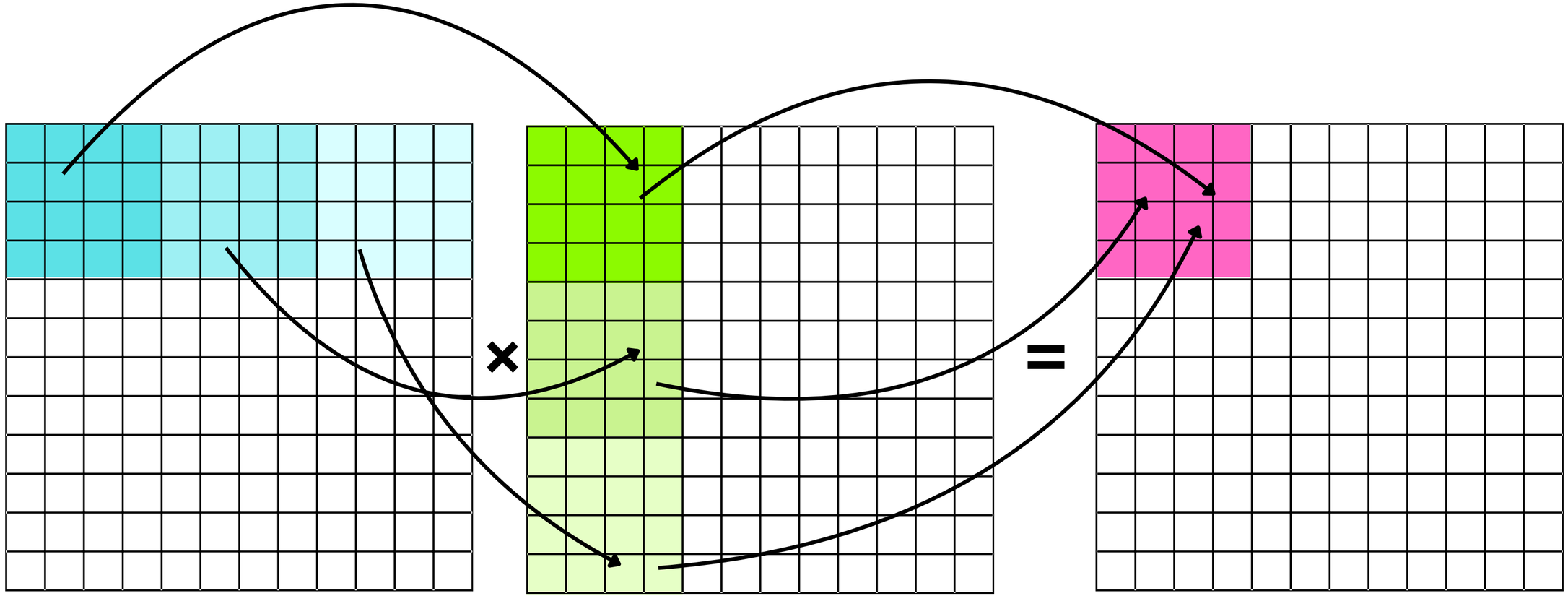
A[:T][:] B[:][:*T*] C[:*T*][:*T*]



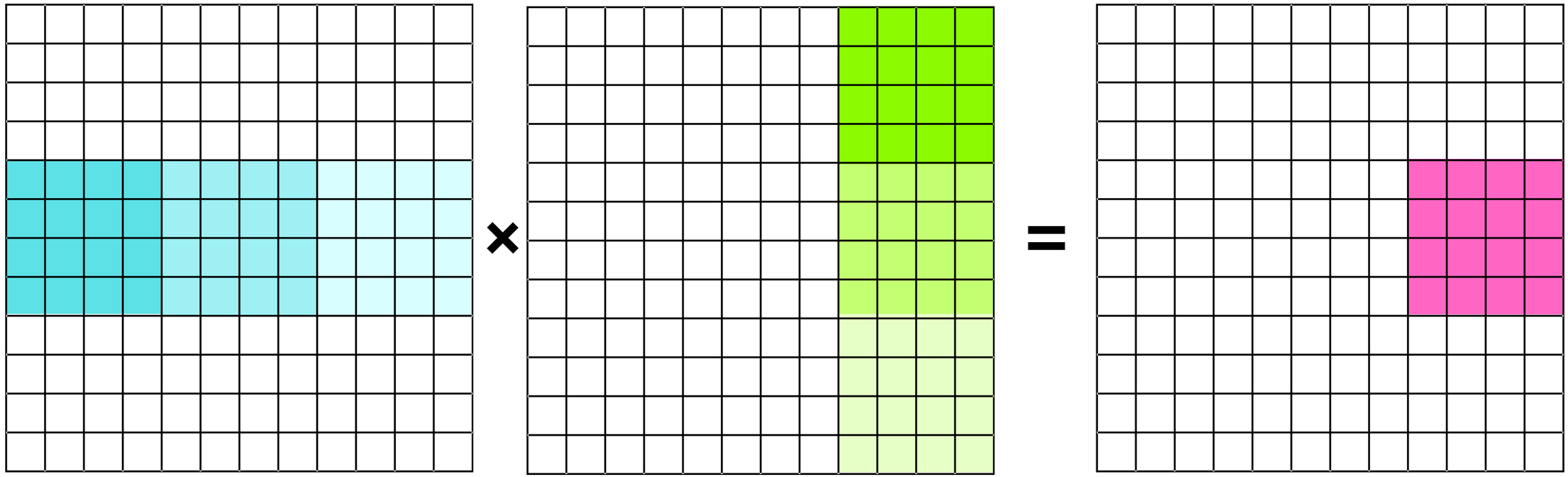


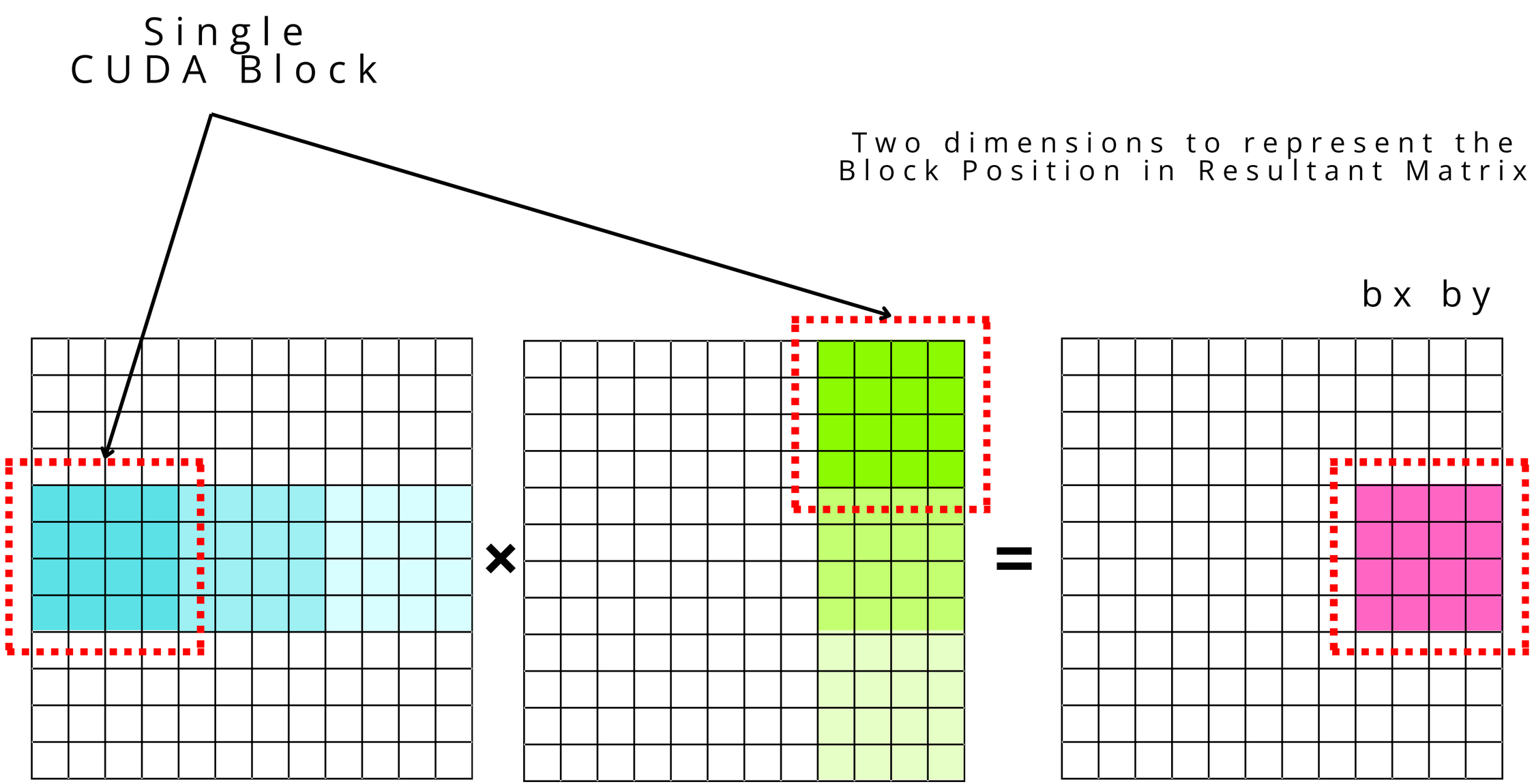


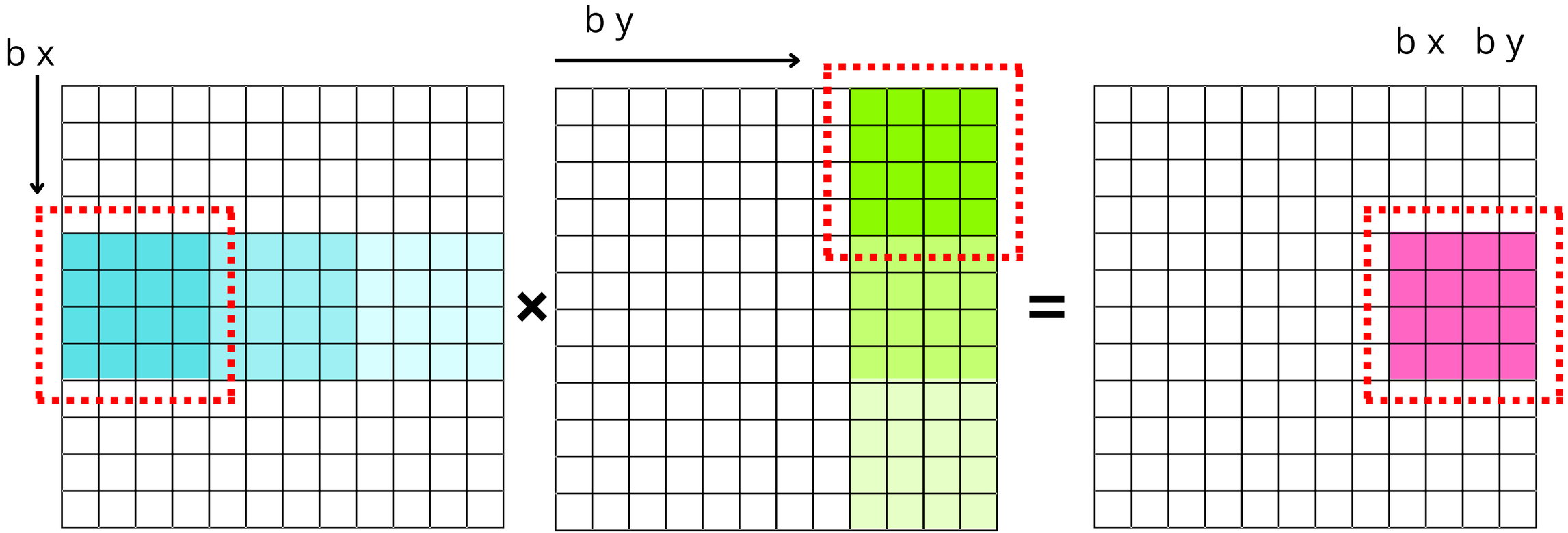
O(1)



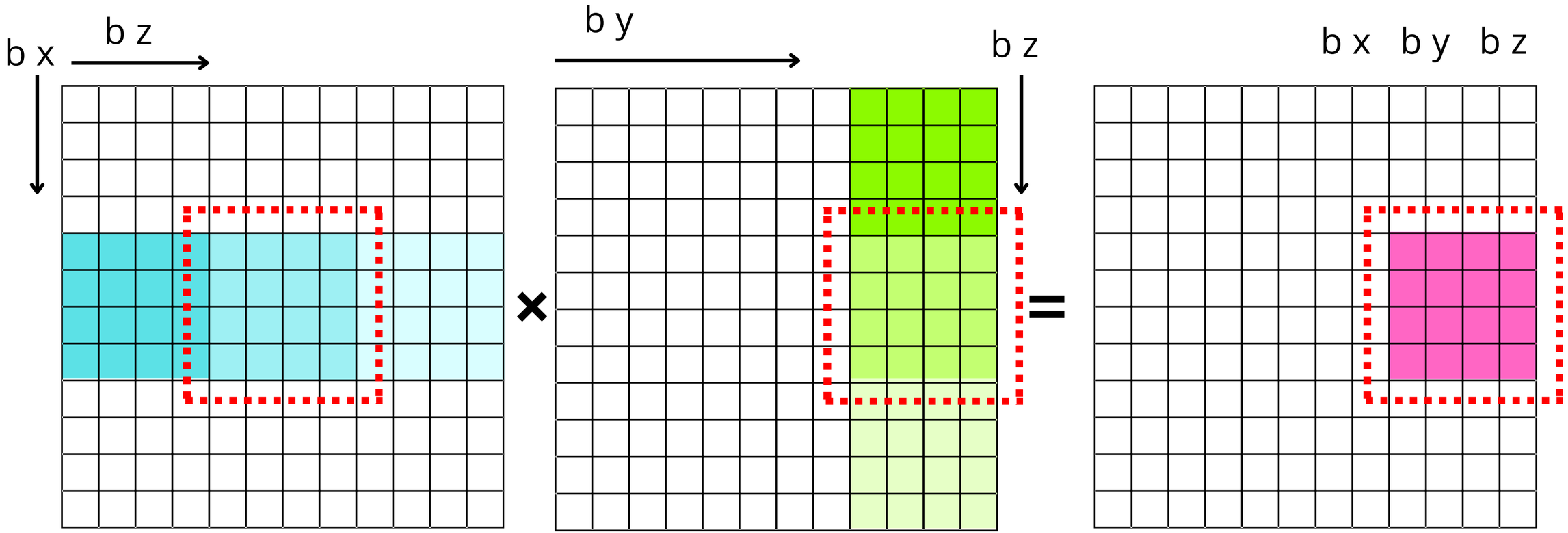
How to design the algorithm?



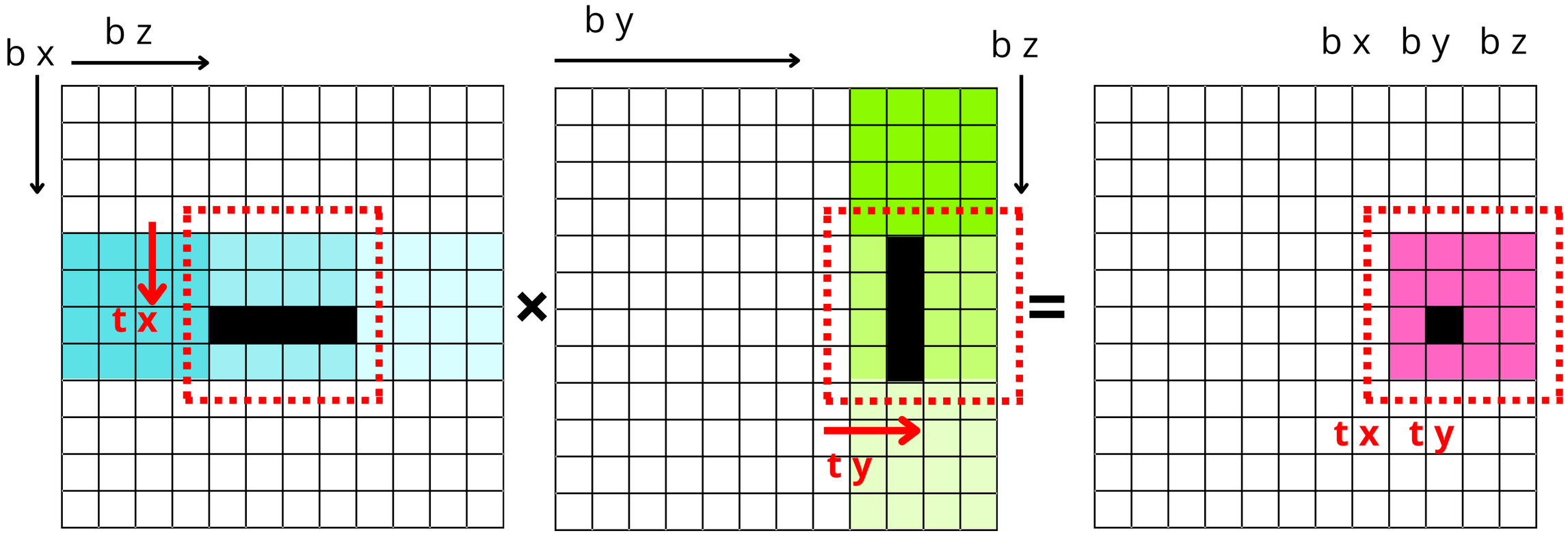

Single
CUDA Block
Two dimensions to represent the
Block Position in Resultant Matrix
bx by


b x
b y
b x b y

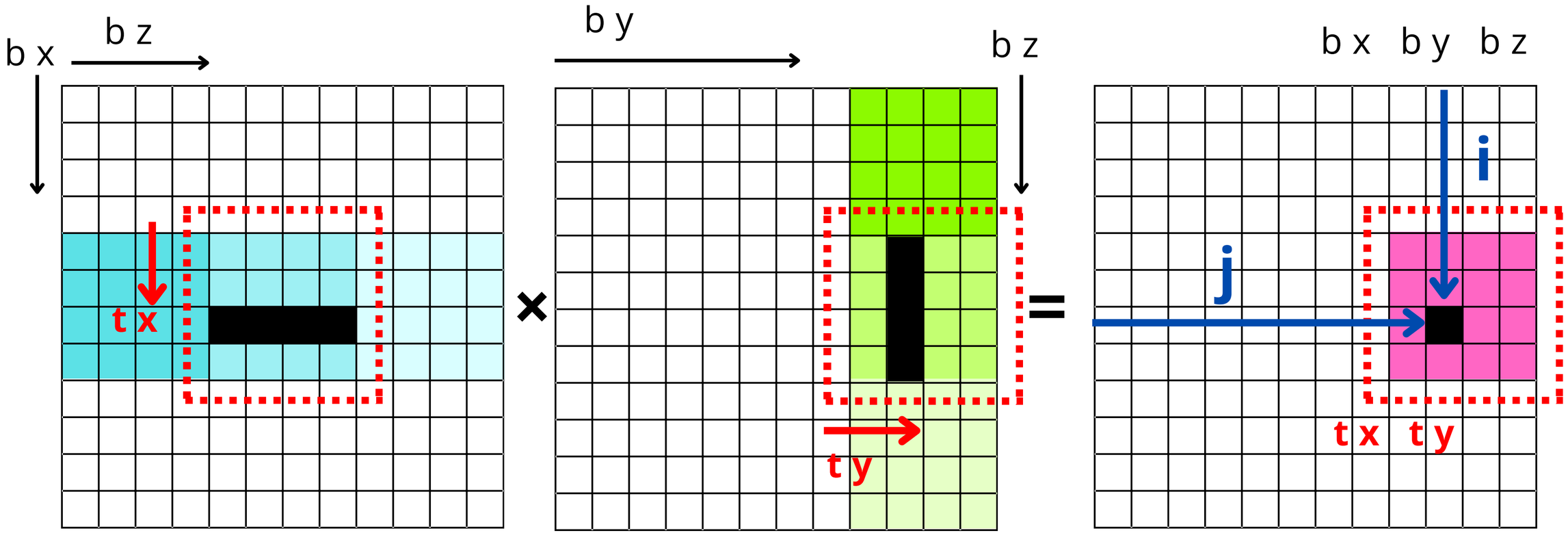
Every Block (bx, by, bz)
does computation on independent set of copied data for
Block (bx,by)



Every Block (bx, by, bz)
has Threads (tx,ty)



every Thread (tx,ty) is mapped w.r.t global indexing



```
__global__ void Tiled_Mat_Multi
  (
    int * a, int * b, int * c, int N
  ) {

    int bx = blockIdx.x;
    int by = blockIdx.y;
    int bz = blockIdx.z;

    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // C[i][y]
    int i = bx * blockDim.x + tx;
    int j = by * blockDim.y + ty;

    __shared__ int sh_a[T * T];
    __shared__ int sh_b[T * T];
    __syncthreads();
```

shared variable initialization (local to a Block)

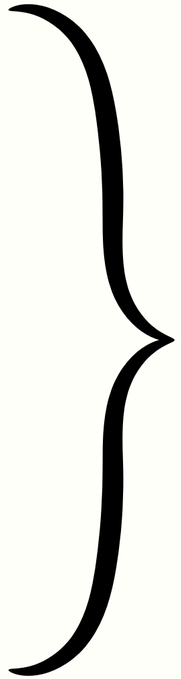
```
__shared__ int sh_a[T * T];
__shared__ int sh_b[T * T];
__syncthreads();

// copying data only once per block
if (tx == 0 && ty == 0) {
  int offset_a = T * (bx * N + bz);
  int offset_b = T * (bz * N + by);

  for (int x = 0; x < T; x++) {
    // copying each row in Tile
    for (int y = 0; y < T; y++) {
      sh_a[x * T + y] = a[offset_a + y];
      sh_b[x * T + y] = b[offset_b + y];
    }
    // offset to next row in a the Tile
    offset_a += N;
    offset_b += N;
  }

}

__syncthreads();
```



Copying data done by only one thread

```
    // sum specific to a Block
    int local_sum = 0;
    for (int k = 0; k < T; k++) {
      local_sum += sh_a[tx * T + k] * sh_b[k * T + ty];
    }

    // add local sum to Global resultant matrix using Atomic Add
    atomicAdd( & c[i * N + j], local_sum);
}
```

Each thread performing calculation

Writing result to Global resultant matrix

```
dim3 gridDim (N/T, N/T, N/T);
dim3 blockDim (T, T, 1);
Tiled_Mat_Multi <<<gridDim,blockDim>>> (d_a, d_b, d_c, N);
cudaDeviceSynchronize();
```

References

Tiled Matrix Multiplication
http://alvinwan.com/how-to-tile-matrix-multiplication/

Memory Hierarchy
https://giahuy04.medium.com/memory-types-in-gpu-6373b7a0ca47

Code

Implementation (personal repo)
https://github.com/Harshtherocking/learn-cuda


Thanks