Introduction to Parallel & Distributed Programming

Lec 17— CUDA Atomics & Example

Subodh Sharma | March 17, 2026

(RECAP) Shared-Memory Use

- Imagine a stencil-based computation: Naive, use of global memory

```
__global__ void stencil_naive(const float *A, float *B, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i > 0 && i < N - 1) {
        B[i] = A[i - 1] + A[i] + A[i + 1];
    }
}
```

- Load a **window of global memory in to shared memory,** including neighbour elements called *halos*

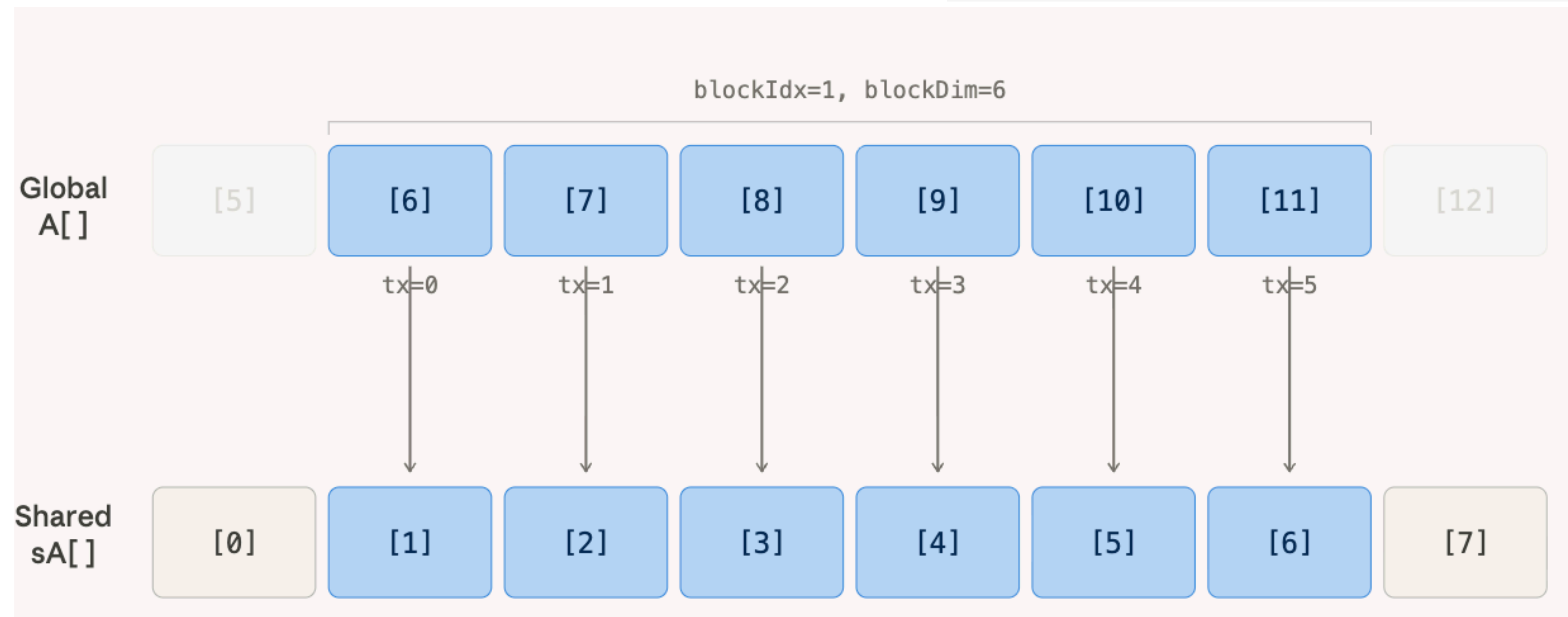
(RECAP) Shared-Memory Use

- Load Center Elements
  - For blockDim.x = 6, blockId.x = 1, i $\in [6,11]$

```
__global__ void stencil_shared(const float *A, float *B, int N) {
    __shared__ float sA[256 + 2];    // blockDim.x assumed 256

    int tx = threadIdx.x;
    int i  = blockIdx.x * blockDim.x + tx;

    // Load center element
    if (i < N) {
        sA[tx + 1] = A[i];
    }
```



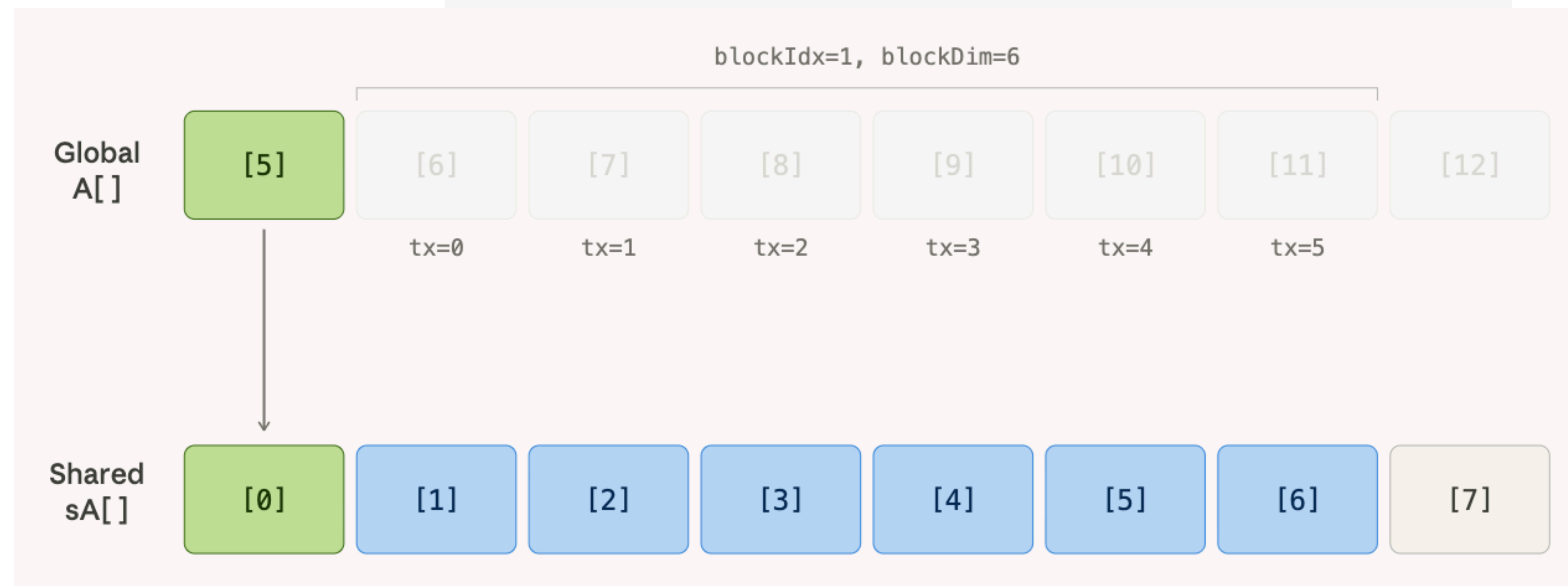
(RECAP) Shared-Memory Use

- Load left and right *halo positions*
- `_syncthreads`: Block level synchronisation.
    - **Execution Barrier:** No thread can move past until every thread has arrived here
    - **Memory Ordering**: Guarantees all writes by all threads before the barrier are visible to all other threads in the block after they pass the barrier

```
// Load left halo
if (tx == 0 && i > 0) {
    sA[0] = A[i - 1];
}

// Load right halo
if (tx == blockDim.x - 1 && i < N - 1) {
    sA[blockDim.x + 1] = A[i + 1];
}

__syncthreads();
```



(RECAP) Synchronisation Primitives

- `__syncwarp():` Warp level synchronisation
  - **Typical use**: Avoids the cost of a full block barrier when only one warp is involved

```
int lane = threadIdx.x % 32;

if (threadIdx.x < 32) {
    s[lane] = A[lane];

    __syncwarp();

    if (lane > 0) {
        B[lane] = s[lane] + s[lane - 1];
```

(RECAP) Synchronisation Primitives

- **Memory fence primitives** — Different from Barriers; **used for visibility alone**
  - `__threadfence_block()`: Calling thread's prior writes are visible to all threads of the block
  - `__threadfence()`: Calling thread's prior writes are visible to all threads of the device
  - `__threadfence_system()`: Calling thread's prior writes are visible to all threads of the system (Device + Host)
  - **Assignment** — Try a simple producer-and-consumer code with various fences and observe the effects!

Synchronisation Primitives

- **Atomics:**
  - **AtomicAdd, AtomicSub, AtomicExch, AtomicCAS, AtomicMin, AtomicMax, …**
  - Latency depends whether being applied at global memory or shared memory!
  - **When to use:** Multiple threads updating a shared mem location; implementing reductions

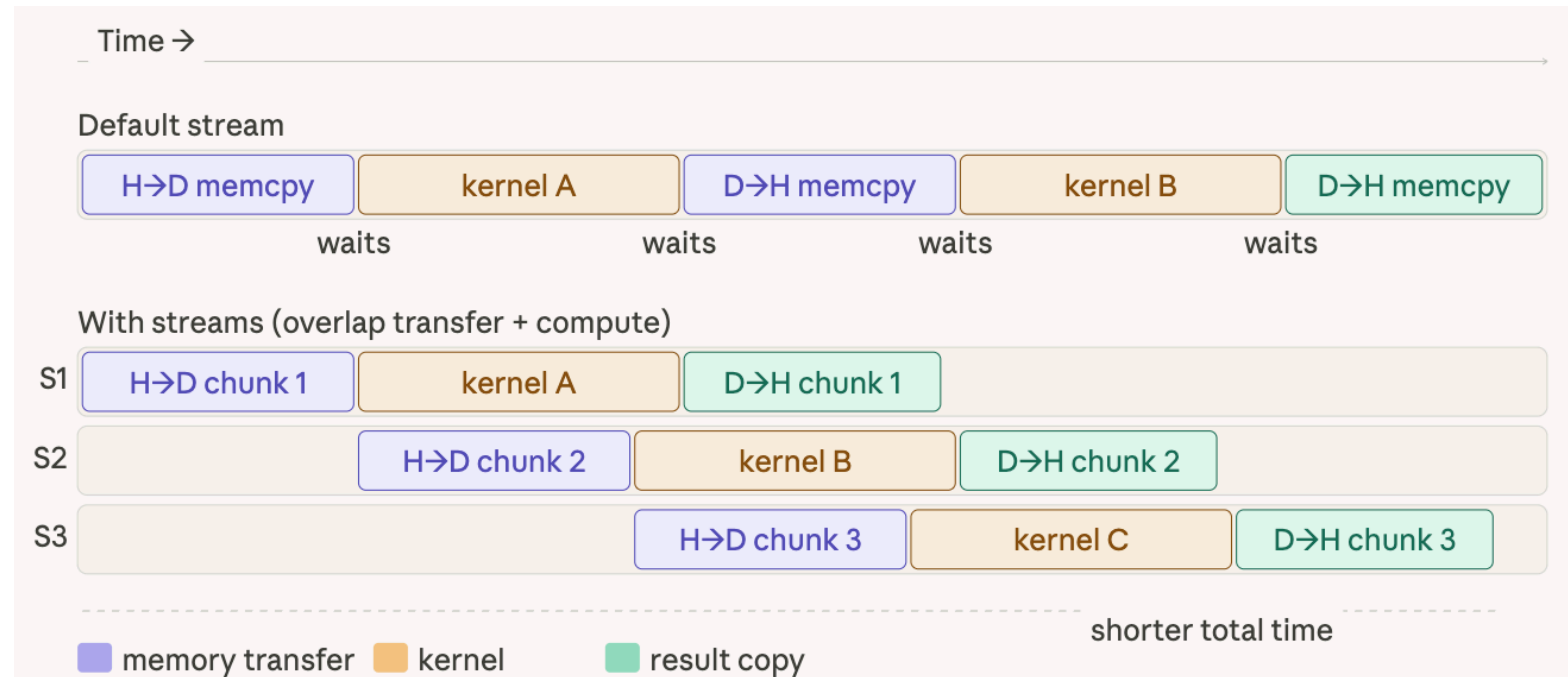
```
__global__ void sumReduce(int *input, int *result, int n) {
    __shared__ int partialSum;
    if (threadIdx.x == 0) partialSum = 0;
    __syncthreads();

    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n)
        atomicAdd(&partialSum, input[i]);
    __syncthreads();

    if (threadIdx.x == 0)
        atomicAdd(result, partialSum);
}
```

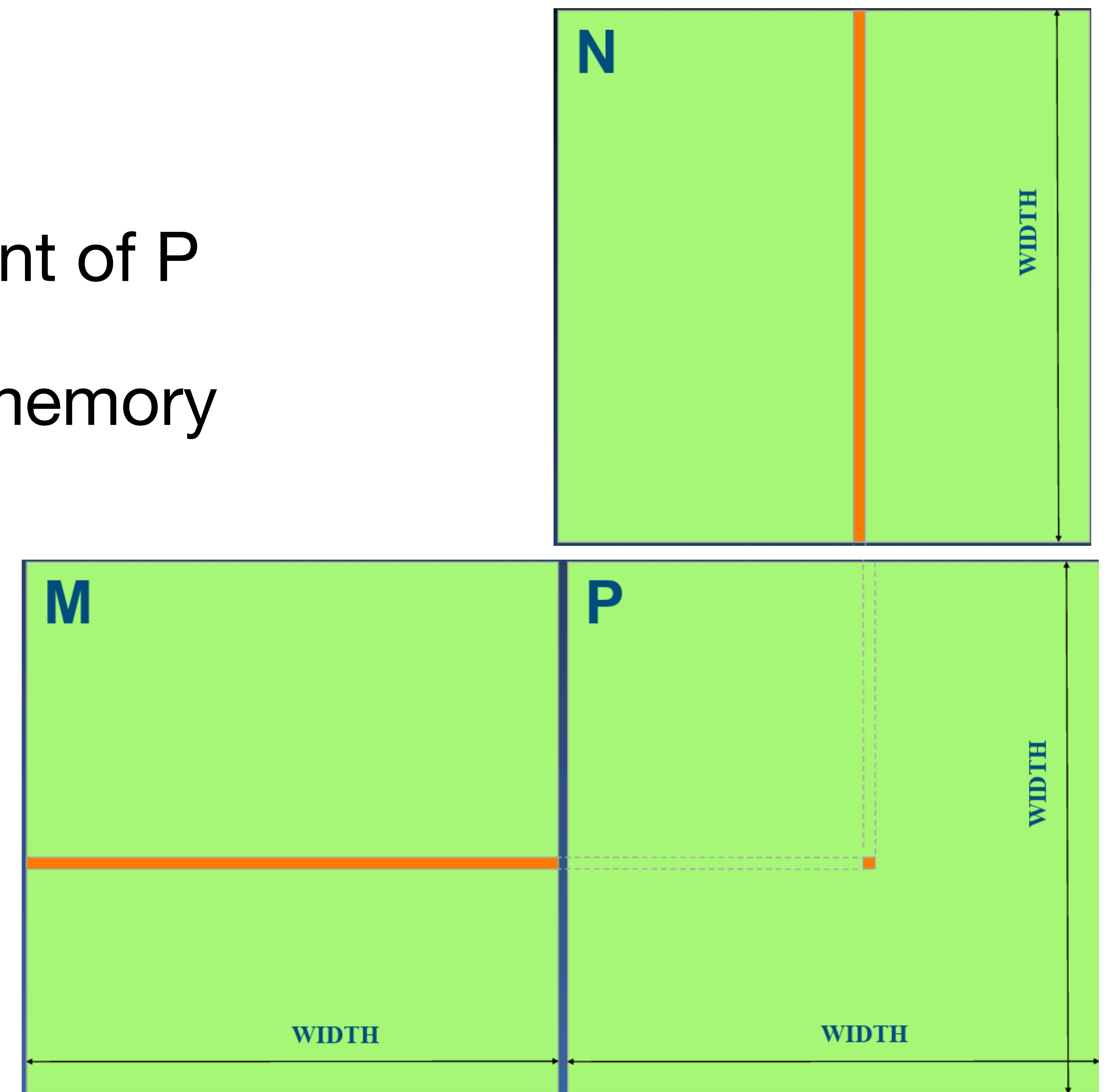
Other Important Aspects

- Explore the idea of **Streams**
  - `cudaStreamSynchronize(s)` : CPU blocks until stream s finishes.
- Explore the idea of asynchronous memory copy through `cudaMemcpyAsync`



Matrix Multiplication

- P = M*N of size WIDTH*WIDTH
- Without tiling: One thread computes one element of P
- M and N are loaded WIDTH times from global memory

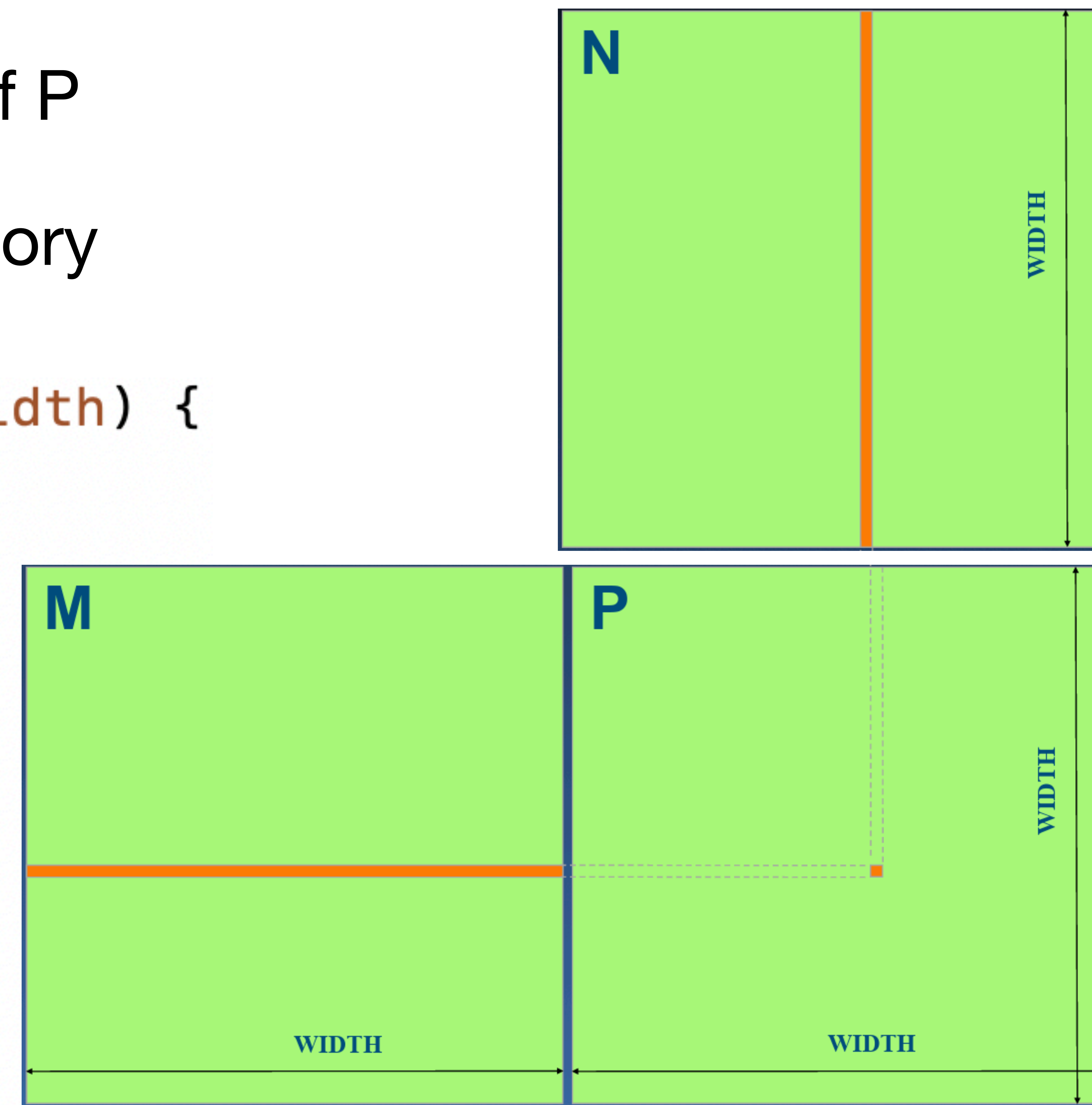


Matrix Multiplication

On Host

- P = M*N of size WIDTH*WIDTH
- Without tiling: One thread computes one element of P
- M and N are loaded WIDTH times from global memory

```
void MatrixMulOnHost(float* M, float* N, float* P, int Width) {
  for (int i = 0; i < Width; ++i)
    for (int j = 0; j < Width; ++j) {
      double sum = 0;
      for (int k = 0; k < Width; ++k) {
        double a = M[i * width + k];
        double b = N[k * width + j];
        sum += a * b;
      }
      P[i * Width + j] = sum;
    }
}
```

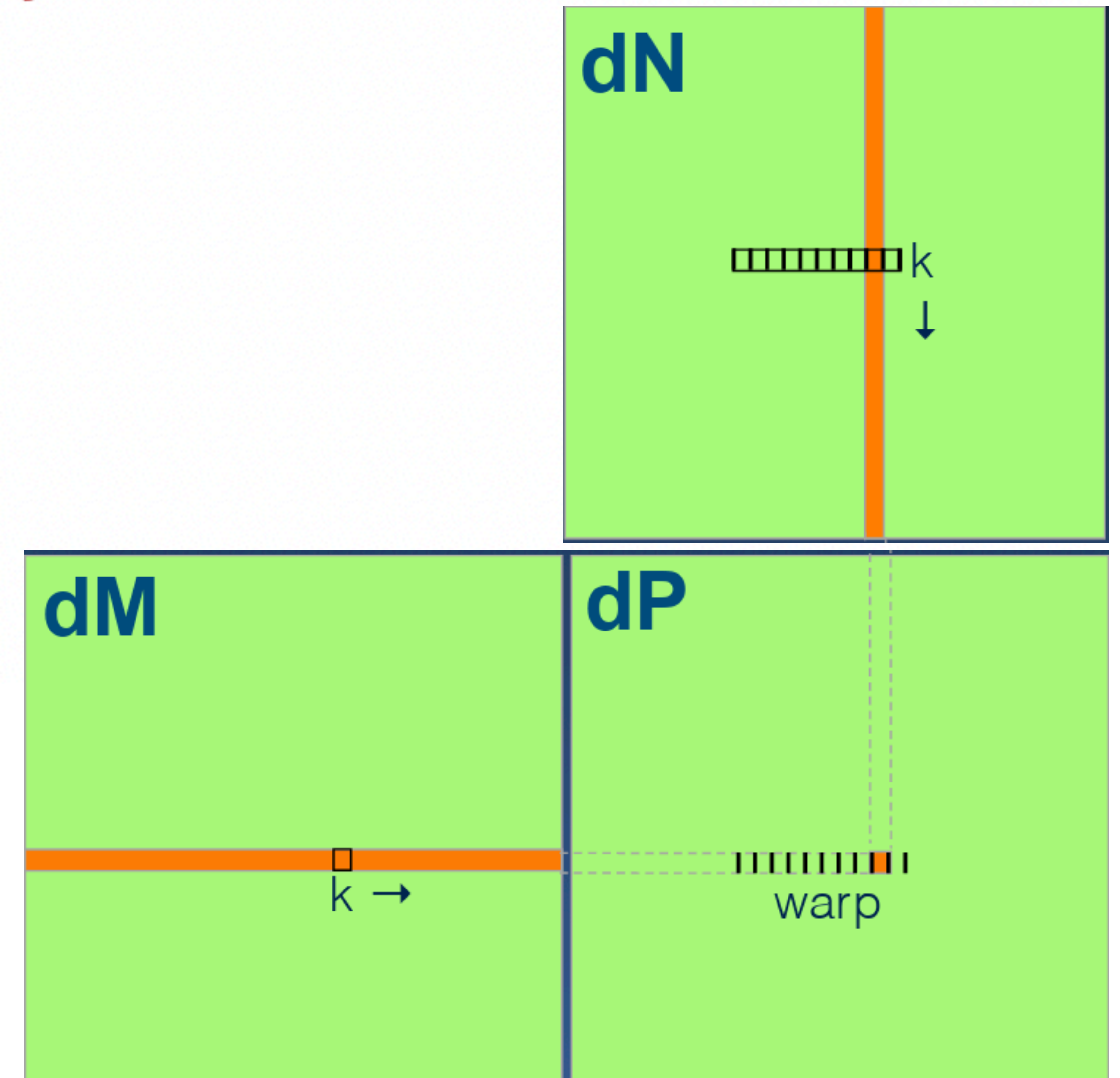


Matrix Multiplication

On Device

```
__global__ void MatrixMulKernel(float* dM, float* dN, float* dP, int Width {
    // Pvalue stores the matrix element computed by the thread
    float Pvalue = 0;
    for (int k = 0; k < Width; ++k) {
      float Melement = dM[threadIdx.y*Width + k];
      float Nelement = dN[k*Width + threadIdx.x];
      Pvalue += Melement * Nelement;
    }
    dP[threadIdx.y*Width + threadIdx.x] = Pvalue;
}
```

- Thread (i, j) computes P[I][j]
- Thread coordinates:
    - threadIdx.y → row index; threadIdx.x → column index



Matrix Multiplication

On Device — with Tiling and Shared Memory

- Each block = TILE_WIDTHxTILE_WIDTH threads
- Shared Memory — hold 1 Tile of dM and 1 Tile of dN
- Thread (`threadIdx.y, threadIdx.x`) inside a block computes dP(`row, col`)
    - `row = = blockIdx.y * TILE_WIDTH + threadIdx.y`
    - `col = blockIdx.x * TILE_WIDTH + threadIdx.x`