Introduction to Parallel & Distributed Programming

Lec 16— CUDA Synchronisations

Subodh Sharma | March16, 2026

(RECAP) CUDA Basics

Variable Qualifiers

- __device__ type var_name;
  –resides in GDRAM, scope is lifetime of the app

- __constant__ type var_name;
  –resides in constant memory space on the device
  –scope is the lifetime of the application

- __shared__ type var_name;
  –resides in the shared memory space of the thread block
  –scope is the lifetime of the block

(RECAP) CUDA Basics

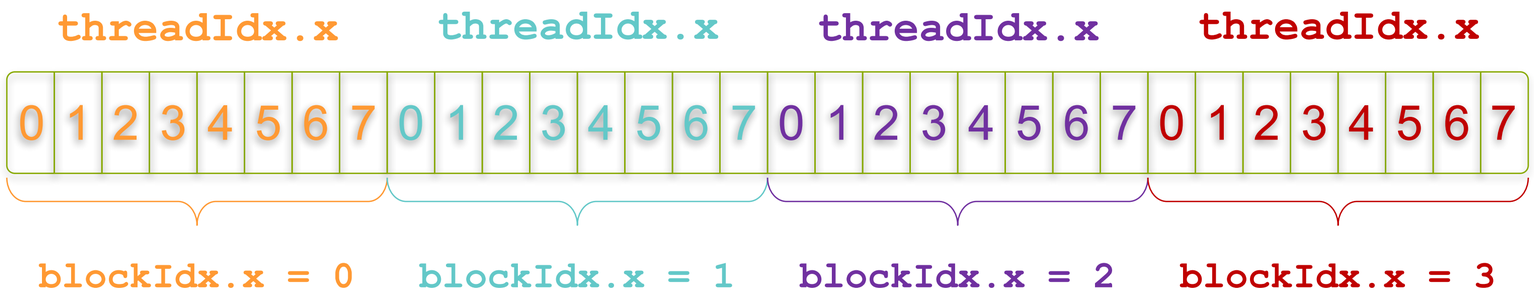
- **Kernel** - The unit of computation for a GPC

```
__global__ void vector_add(const float *A, const float *B, float *C, int N)
{

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

    if (i < N)
        C[i] = A[i] + B[i];
}
```

Explain why this check is required

Device Qualifier: On device but called from the host

Thread Indexing



- index = threadIdx.x + blockIdx.x * blockDim.x

(RECAP) CUDA Basics

- **Kernel Invocation (from a HOST function)**

```
int threads_per_block = 256;
int blocks = (N + threads_per_block - 1) / threads_per_block;

vector_add<<<blocks, threads_per_block>>>(d_A, d_B, d_C, N);

CUDA_CHECK(cudaDeviceSynchronize());
```

Wait for GPU to finish computation

Grid Dimension, Block Dimension

(RECAP) CUDA Basics

- **Data Transfer (from HOST to DEVICE)**

```
float *d_A, *d_B, *d_C;
CUDA_CHECK(cudaMalloc(&d_A, bytes));
CUDA_CHECK(cudaMalloc(&d_B, bytes));
CUDA_CHECK(cudaMalloc(&d_C, bytes));

// —— Step 3: Copy input data from CPU → GPU ————
CUDA_CHECK(cudaMemcpy(d_A, h_A, bytes, cudaMemcpyHostToDevice));
CUDA_CHECK(cudaMemcpy(d_B, h_B, bytes, cudaMemcpyHostToDevice));
printf("Copied A and B: CPU → GPU\n");

... (kernel invocation) ..

CUDA_CHECK(cudaMemcpy(h_C, d_C, bytes, cudaMemcpyDeviceToHost));
```

Heap allocation on the DEVICE

Direction of Memcpy — H2D

Revisiting the Memory Hierarchy

- **Registers**: On-chip, Private to each thread
- **Shared memory**: Fast, on-chip, shared by thread from the **same block**
  - It is a block-level scratch-pad
- **L1 Cache/ Read-only Cache/ Texture Cache**: On-chip, managed by hardware
- **L2 Cache**: Large on-chip cache, **shared across SMs**
- **Global Memory:** Large and slow

Shared-Memory Use

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

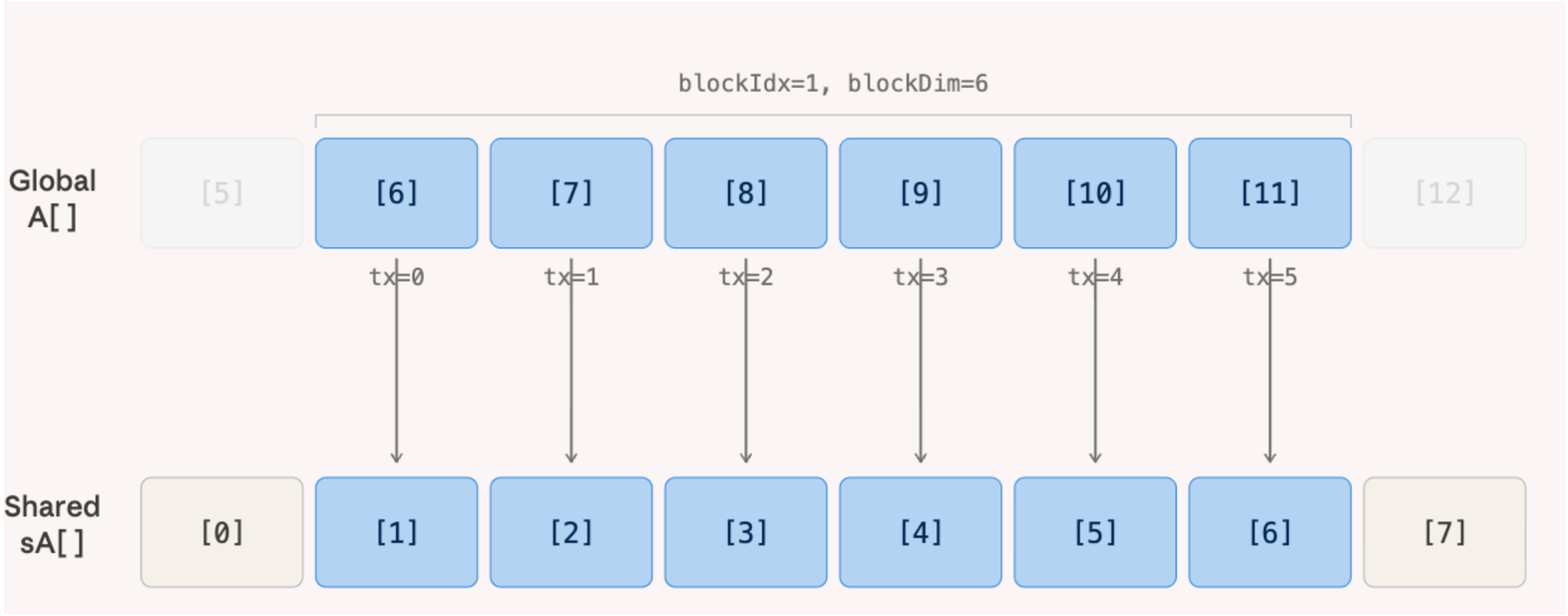
```
__global__ void stencil_naive(const float *A, float *B, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    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*

Shared-Memory Use

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

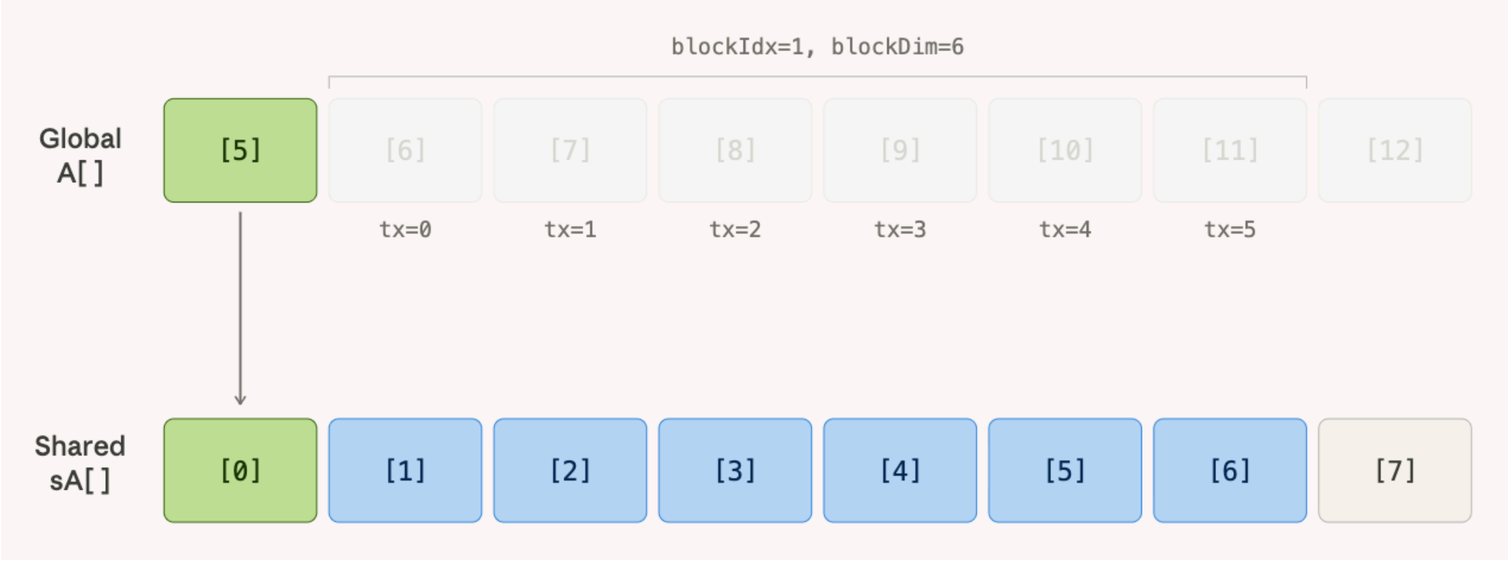
```cpp
__global__ void stencil_shared(const float *A, float *B, int N) {
    __shared__ float sA[256 + 2];   // blockDim.x assumed 256

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

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



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();
```



Common Mistakes with Shared Memory

- If there is **no reuse — don't use** it
- Don't forget to use `__syncthreads()` — otherwise race conditions can arise
- Taking up too much shared memory per block will **hit the occupancy of SM**.
- Avoid bank conflicts
  - Shared mem is organised in **banks** — concurrently accessible
    - Bank conflits occurs when the "stride" of accesses cause multiple threads to hit the same bank

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];
```

Synchronisation Primitives

- `__syncthreads()`: Block level synchronisation
  - **Typical use**: When work happens in phases, and all block threads must finish one phase before the next
  - **Do not use:**
    - When threads are independent
    - Only one warp is involved
    - Inside a divergent control flow

```
if (threadIdx.x < 128) {
    __syncthreads();   // Wrong
}
```

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!