CUDA SHARED MEMORY

NVIDIA Corporation

REVIEW (1 OF 2)

- Difference between *host* and *device*

Host	CPU
Device	GPU

- Using `__global__` to declare a function as device code
  - Executes on the device
  - Called from the host (or possibly from other device code)
- Passing parameters from host code to a device function

REVIEW (2 OF 2)

- Basic device memory management
  - `cudaMalloc()`
  - `cudaMemcpy()`
  - `cudaFree()`
- Launching parallel kernels
  - Launch `N` copies of `add()` with `add<<<N,1>>>(…);`
  - Use `blockIdx.x` to access block index

Shared Memory

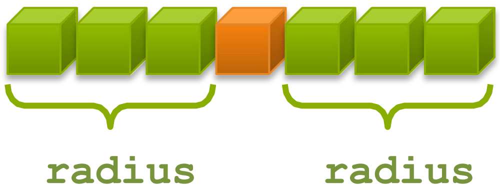
- Shared memory is equivalent to a user-managed cache: The application explicitly allocates and accesses it.

Recall: how does cache work in OS and computer organization courses

- Launching parallel kernels
  - Launch **`N`** copies of **`add()`** with **`add<<<N,1>>>(…);`**
  - Use **`blockIdx.x`** to access block index

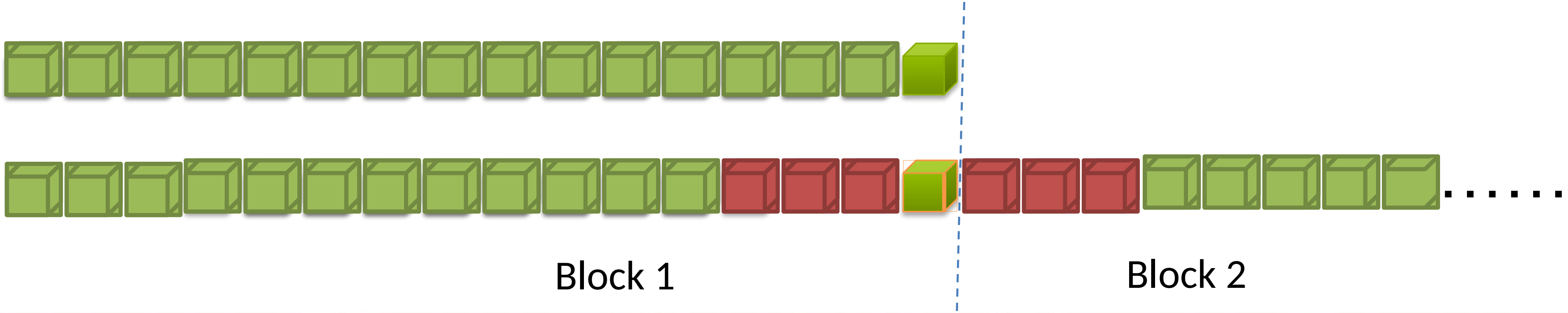
1D STENCIL

- Consider applying a 1D stencil to a 1D array of elements
  - Each output element is the sum of input elements within a radius
- If radius is 3, then each output element is the sum of 7 input elements:



IMPLEMENTING WITHIN A BLOCK

- Each thread processes one output element
  - `blockDim.x` elements per block
- Input elements are read several times
  - With radius 3, each input element is read seven times



Shared Memory: SHARING DATA BETWEEN THREADS

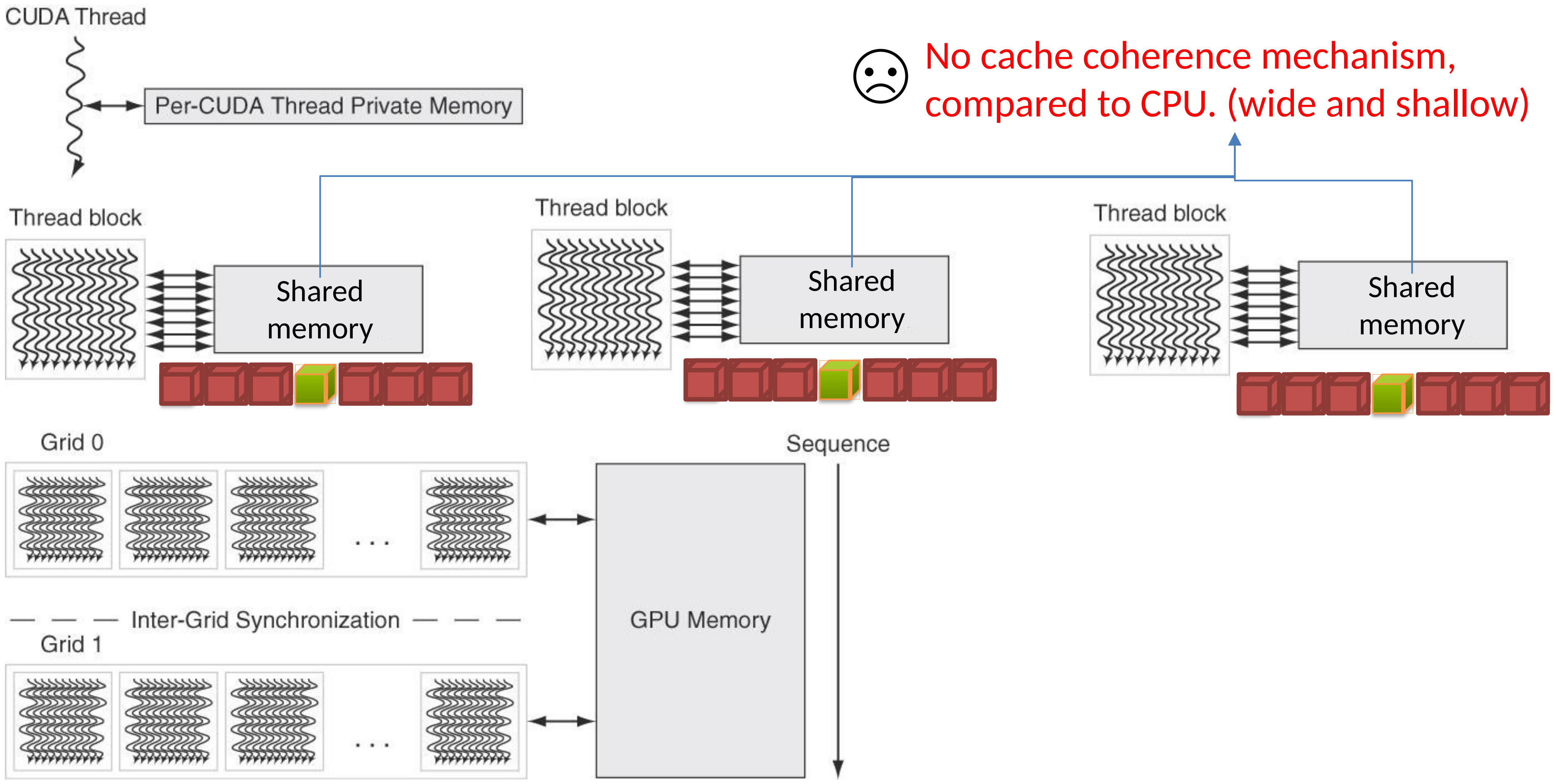
- Terminology: within a block, threads share data via shared memory
- Shared memory is equivalent to a user-managed cache:
  - The application explicitly allocates and accesses it.
- Extremely fast on-chip memory, user-managed
- Declare using `__shared__`, allocated a variable per block that:
  - Resides in the shared memory space of a thread block,
  - Has the lifetime of the block,
  - Has a distinct object per block,
  - Is only accessible from all the CUDA threads within the block,

Shared Memory: SHARING DATA BETWEEN THREADS

- Typical programming pattern:
  - Load data from device memory to shared memory
  - Synchronize with all the other threads of the block so that each thread can safely read shared memory locations that were populated by different threads,
  - Process the data in shared memory
  - Synchronize again if necessary to make sure that shared memory has been updated with the results,
  - Write the results back to device memory.

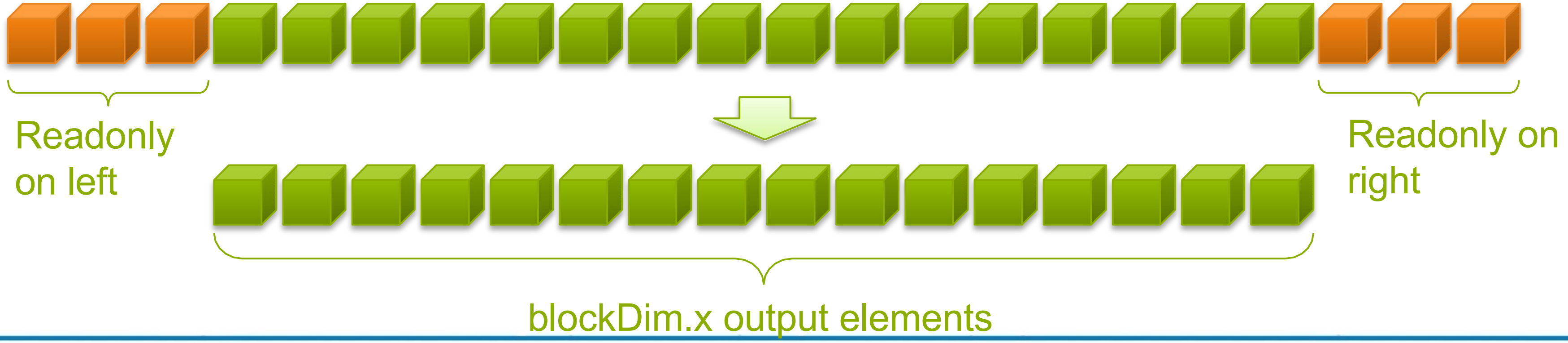
Shared Memory: SHARING DATA BETWEEN THREADS

☹ No cache coherence mechanism, compared to CPU. (wide and shallow)



IMPLEMENTING WITH SHARED MEMORY

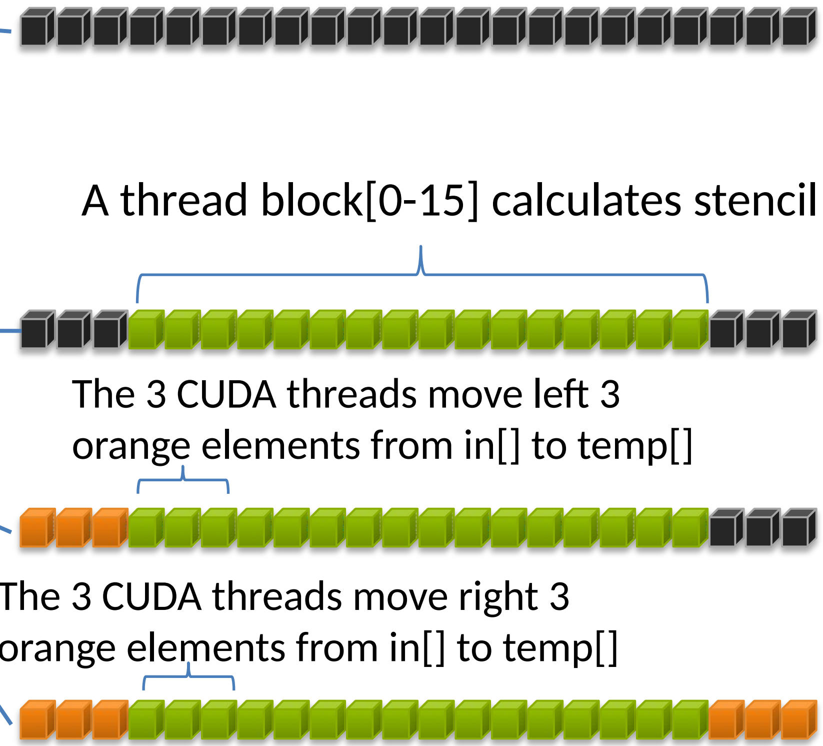
- Cache data in shared memory
  - Read `(blockDim.x + 2 * radius)` input elements from global memory to shared memory
  - Compute `blockDim.x` output elements
  - Write `blockDim.x` output elements to global memory
- Each block needs a halo of `radius` elements at each boundary



STENCIL KERNEL

```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[lindex] = in[gindex];
  if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] =in[gindex+BLOCK_SIZE];
  }
```



STENCIL KERNEL

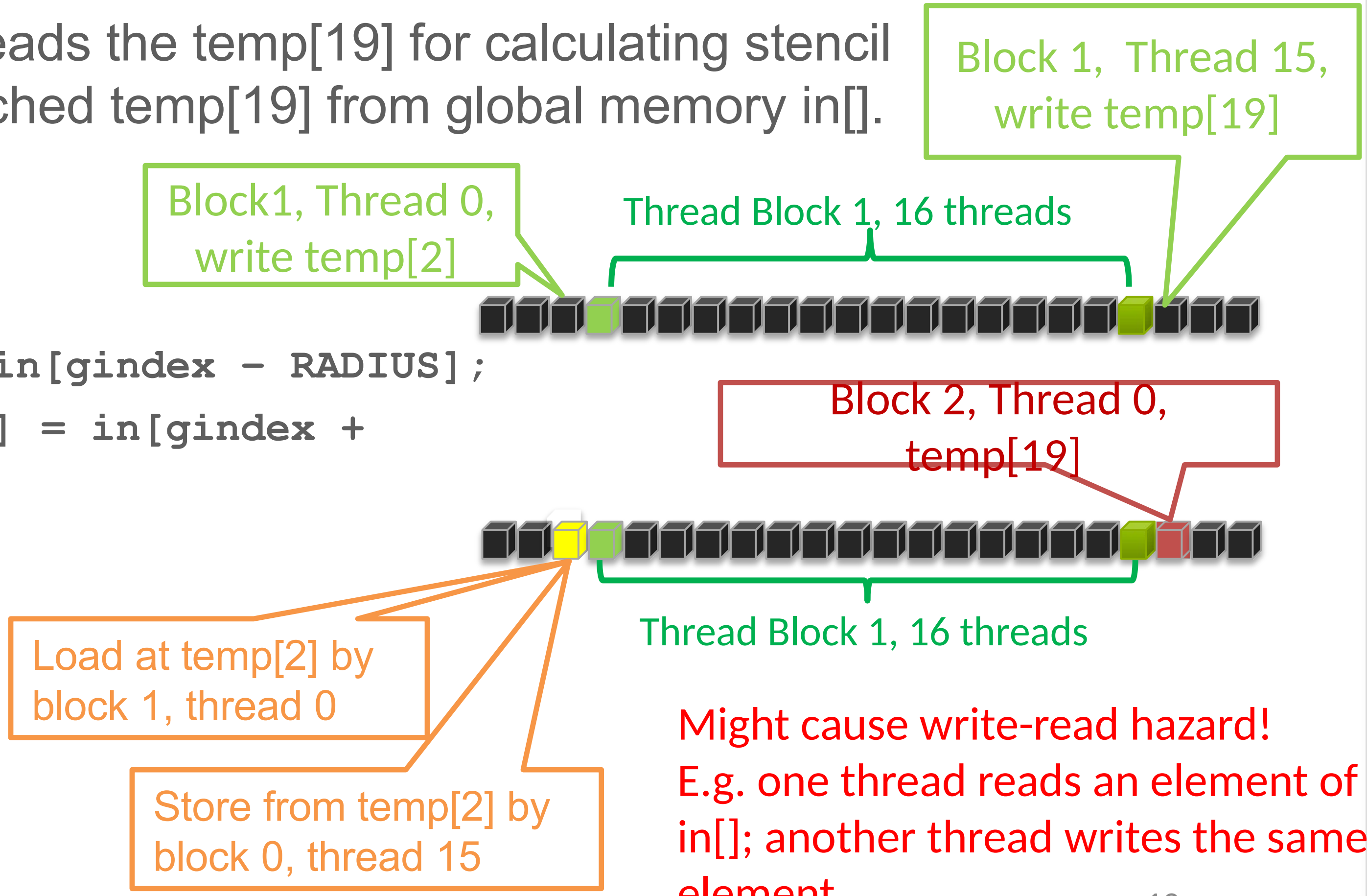
```
// Apply the stencil in shared memory
int result = 0;
for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
  result += temp[lindex + offset];

// Store the result
out[gindex] = result;
}
```

DATA RACE!

- The stencil example will not work…
- Suppose thread 15 in Block 1 reads the temp[19] for calculating stencil after thread 0 in block 2 has fetched temp[19] from global memory in[].

```
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS) {
    temp[lindex – RADIUS] = in[gindex – RADIUS];
    temp[lindex + BLOCK_SIZE] = in[gindex +
    BLOCK_SIZE];
}
int result = 0;
result += temp[lindex + 1];
```



Might cause write-read hazard!
E.g. one thread reads an element of in[]; another thread writes the same element.

__SYNCTHREADS()

- `void __syncthreads();`
- Synchronizes all threads within a CUDA block, how to explain in GPU architecture
    - Used to prevent RAW / WAR / WAW hazards
- All threads must reach the barrier
    - In conditional code, the condition must be uniform across the block

STENCIL KERNEL

```
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
     int lindex = threadIdx.x + radius;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] =
        in[gindex - RADIUS];
        temp[lindex +
        BLOCK_SIZE] = in[gindex
        + BLOCK_SIZE];
    }
    // Synchronize (ensure all
```

STENCIL KERNEL

```
    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
        result += temp[lindex + offset];

    // Store the result
    out[gindex] = result;
}
```

REVIEW

- Use __shared__ to declare a variable/array in shared memory
    - Data is shared between threads in a block
    - Not visible to threads in other blocks
- Use __syncthreads() as a barrier
    - Use to prevent data hazards

FUTURE SESSIONS

- CUDA GPU architecture and basic optimizations
- Atomics, Reductions, Warp Shuffle
- Using Managed Memory
- Concurrency (streams, copy/compute overlap, multi-GPU)
- Analysis Driven Optimization
- Cooperative Groups

FURTHER STUDY

- Shared memory:
  - https://devblogs.nvidia.com/using-shared-memory-cuda-cc/
- CUDA Programming Guide:
  - https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#shared-memory
- CUDA Documentation:
  - https://docs.nvidia.com/cuda/index.html

    https://docs.nvidia.com/cuda/cuda-runtime-api/index.html (runtime API)