KOÇ UNIVERSITY

Memory Hierarchy in GPUs

Case Study: Stencil Computation

Didem Unat

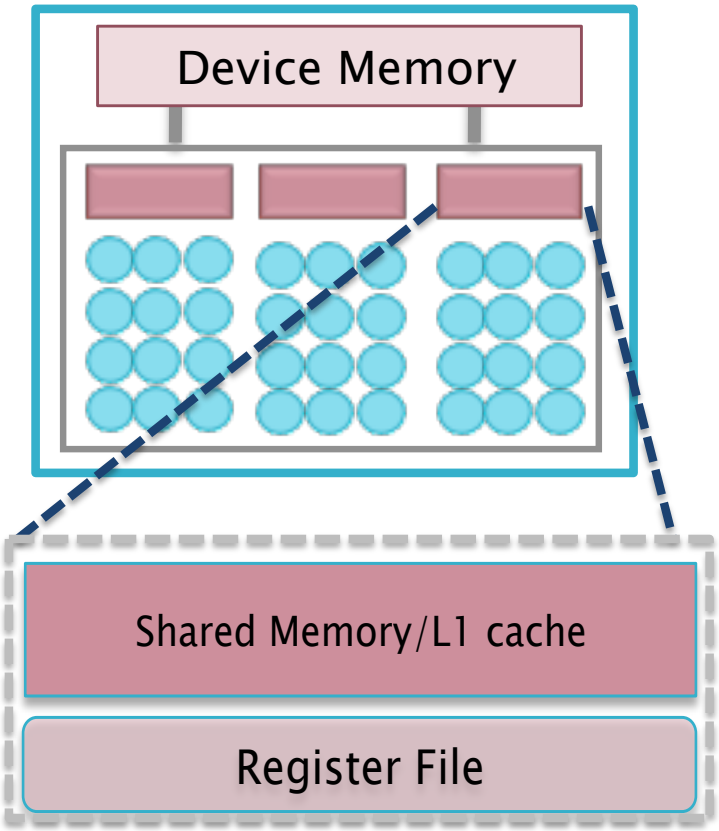
dunat@ku.edu.tr

http://parcorelab.ku.edu.tr

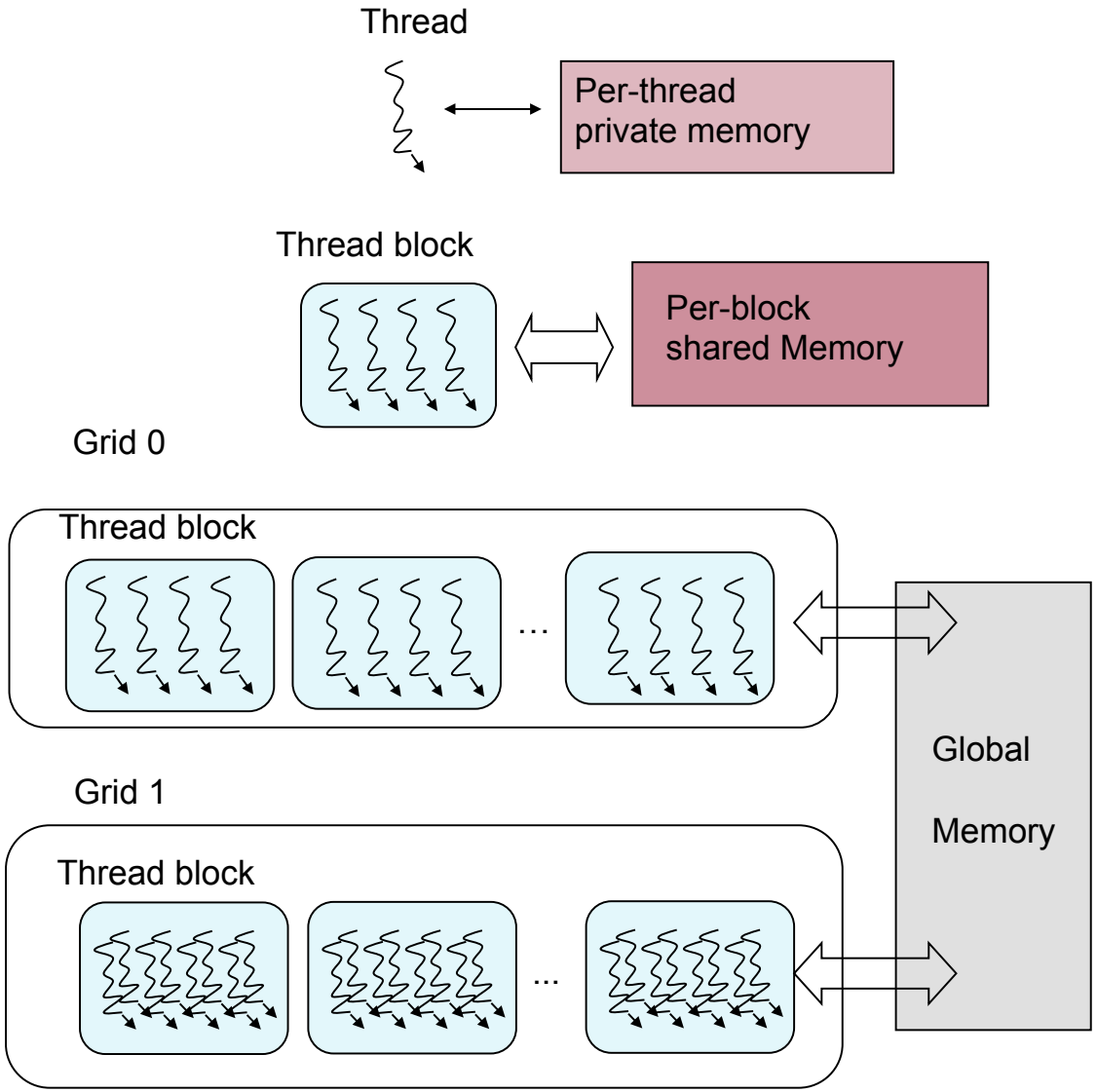
- Heterogeneous Computing
- Kernels
- Memory Management
- Threads
- Thread Blocks
- **Memory Hierarchy**
- Shared Memory
- Handling errors
- Managing devices

Memory Hierarchy on GPUs

- Device Memory
  - Also called **global memory (GPU's DRAM)**
  - Shared by all threads
- Explicitly managed on-chip memory
  - Also called **local storage or shared memory** (unfortunate naming)
  - Private to a block and incoherent
- L1 cache is configurable
  - ¼ is cache ¾ is explicitly managed or vice versa
- Register file
  - Private to a thread



GPU Memory Structures



Registers
(thread-private)

Software-managed Cache
(shared by only a block
of threads)

Device (Global) memory
(shared by all threads)

Language Extensions: Variable Type Qualifiers

Variable declaration	Memory	Scope	Lifetime
`int var;`	register	thread	thread
`int array_var[10];`	local	thread	thread
`__shared__ int shared_var;`	shared	block	block
`__device__ int global_var;`	global	grid	application
`__constant__ int constant_var;`	constant	grid	application

- "automatic" scalar variables without qualifier reside in a register
  - compiler will spill to thread local memory
- "automatic" array variables without qualifier reside in thread-local memory

CUDA Variable Type Performance

Variable declaration	Memory	Penalty
`int var;`	register	1x
`int array_var[10];`	local	100x
`__shared__ int shared_var;`	shared	1x
`__device__ int global_var;`	global	100x
`__constant__ int constant_var;`	constant	1x

- scalar variables reside in fast, on-chip registers
- shared variables reside in fast, on-chip memories
- thread-local arrays & global variables reside in uncached off-chip memory
- constant variables reside in cached off-chip memory

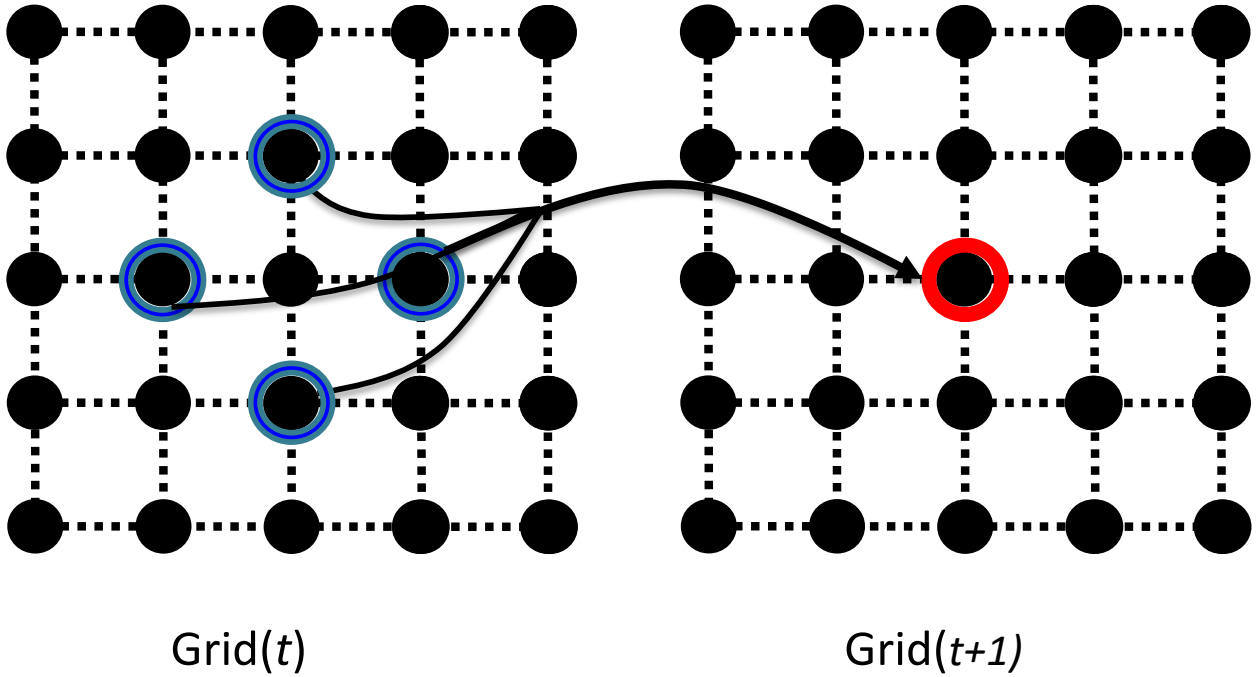
CUDA Variable Type Scale

Variable declaration	Instances	Visibility
`int var;`	100,000s	1
`int array_var[10];`	100,000s	1
`__shared__ int shared_var;`	100s	100s
`__device__ int global_var;`	1	100,000s
`__constant__ int constant_var;`	1	100,000s

- 100Ks per-thread variables, Read/Written by 1 thread
- 100s shared variables, each Read/Written by 100s of threads
- 1 global variable is Read/Written by 100Ks threads
- 1 constant variable is readable by 100Ks threads

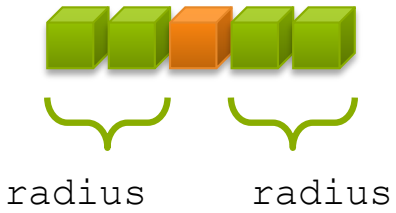
Case Study: Stencil Computation

- Structured grid problems
  - Iterative Finite Difference Techniques
  - Updates each point with weighted contributions from a subset of its neighbors in both in time and space

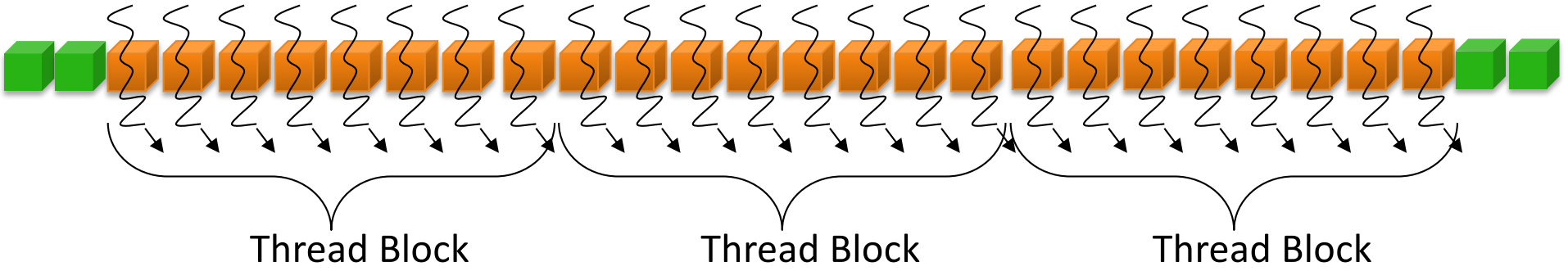


1D Stencil

- Let’s consider first a 1D problem
- 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 2, then each output element is the sum of 5 input elements:



Naïve Implementation



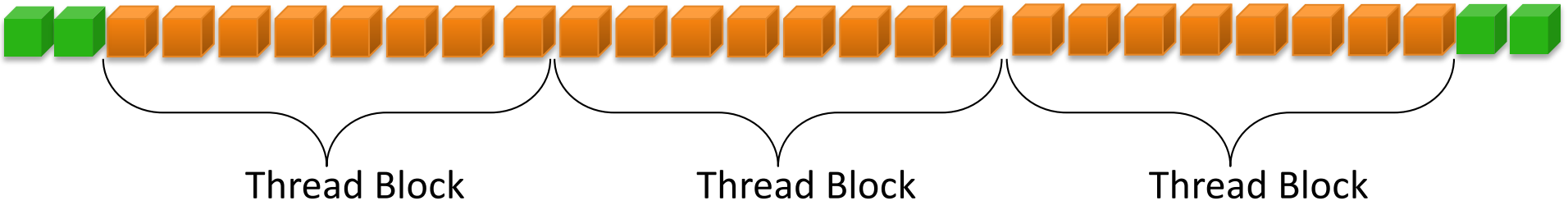
- In global memory, we have halos for boundaries
- Each thread processes one output element
  - `blockDim.x` elements per block
- Input elements are read several times
  - With radius 2, each input element is read 5 times

Lab-4 1D Stencil

- Copy the lab from
  - /kuacc/users/dunat/COMP429/GPU/Hands-on-Labs/labs/4_Stencil1D
- Slides are for RADIUS 2, the lab uses arbitrary RADIUS
- 1st kernel: stencil_1D_simple
  - Performs stencil with a radius 3 on 1D array
- 2nd kernel: stencil_1D_improved
  - Uses shared memory to eliminate redundant memory accesses
  - Fix the problems in the code one by one

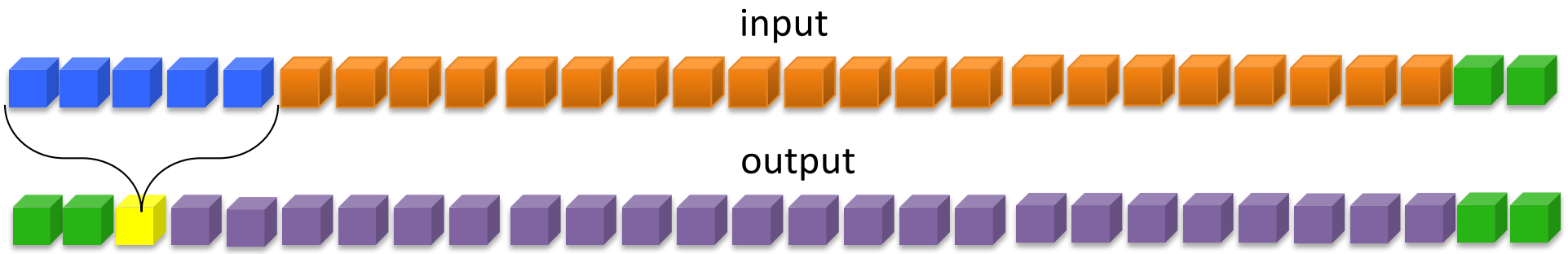
Simple 1D Stencil

```
// a simple version of 1D stencil
__global__ void Stencil_1D_simple(int *out, int *in, int NUM_ELEMENTS)
{
  // compute this thread's global index
  unsigned int i = blockDim.x * blockIdx.x + threadIdx.x + 2;

  if(i < NUM_ELEMENTS + 2 ){
    out[i] = alpha*(in[i-2]+ in[i-1]+
                    in[i+1]+ in[i+2]) +
                beta * in[i];
  }
}
```



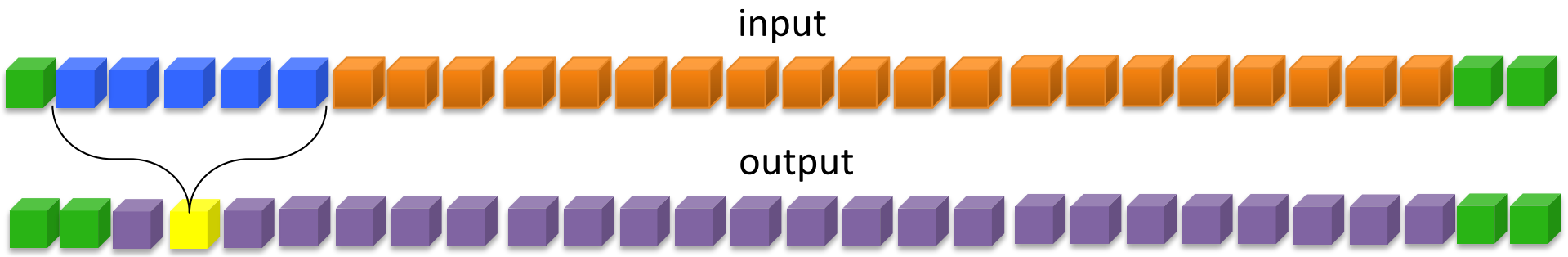
Simple 1D Stencil

```
// a simple version of 1D stencil
__global__ void Stencil_1D_simple(int *out, int *in, int NUM_ELEMENTS)
{
  // compute this thread's global index
  unsigned int i = blockDim.x * blockIdx.x + threadIdx.x + 2;

  if(i < NUM_ELEMENTS + 2 ){
    out[i] = alpha*(in[i-2]+ in[i-1]+
                    in[i+1]+ in[i+2]) +
                beta * in[i];
  }
}
```



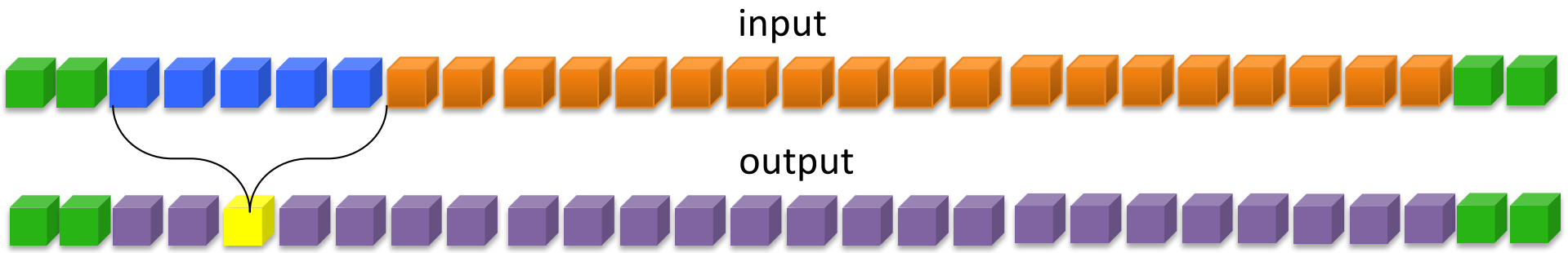
Simple 1D Stencil

```
// a simple version of 1D stencil
__global__ void Stencil_1D_simple(int *out, int *in, int NUM_ELEMENTS)
{
  // compute this thread's global index
  unsigned int i = blockDim.x * blockIdx.x + threadIdx.x + 2;

  if(i < NUM_ELEMENTS + 2 ){
    out[i] = alpha*(in[i-2]+ in[i-1]+
                    in[i+1]+ in[i+2]) +
                beta * in[i];
  }
}
```



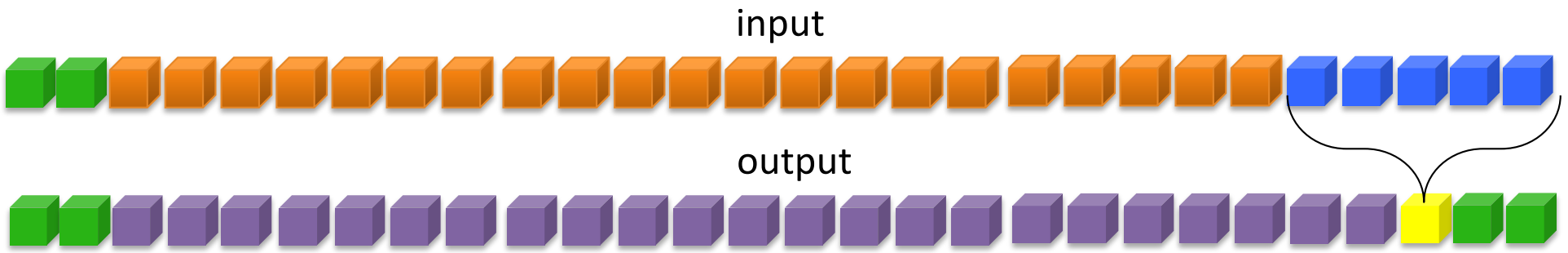
Simple 1D Stencil

```
// a simple version of 1D stencil
__global__ void Stencil_1D_simple(int *out, int *in, int NUM_ELEMENTS)
{
  // compute this thread's global index
  unsigned int i = blockDim.x * blockIdx.x + threadIdx.x + 2;

  if(i < NUM_ELEMENTS + 2 ){
    out[i] = alpha*(in[i-2]+ in[i-1]+
                    in[i+1]+ in[i+2]) +
                beta * in[i];
  }
}
```

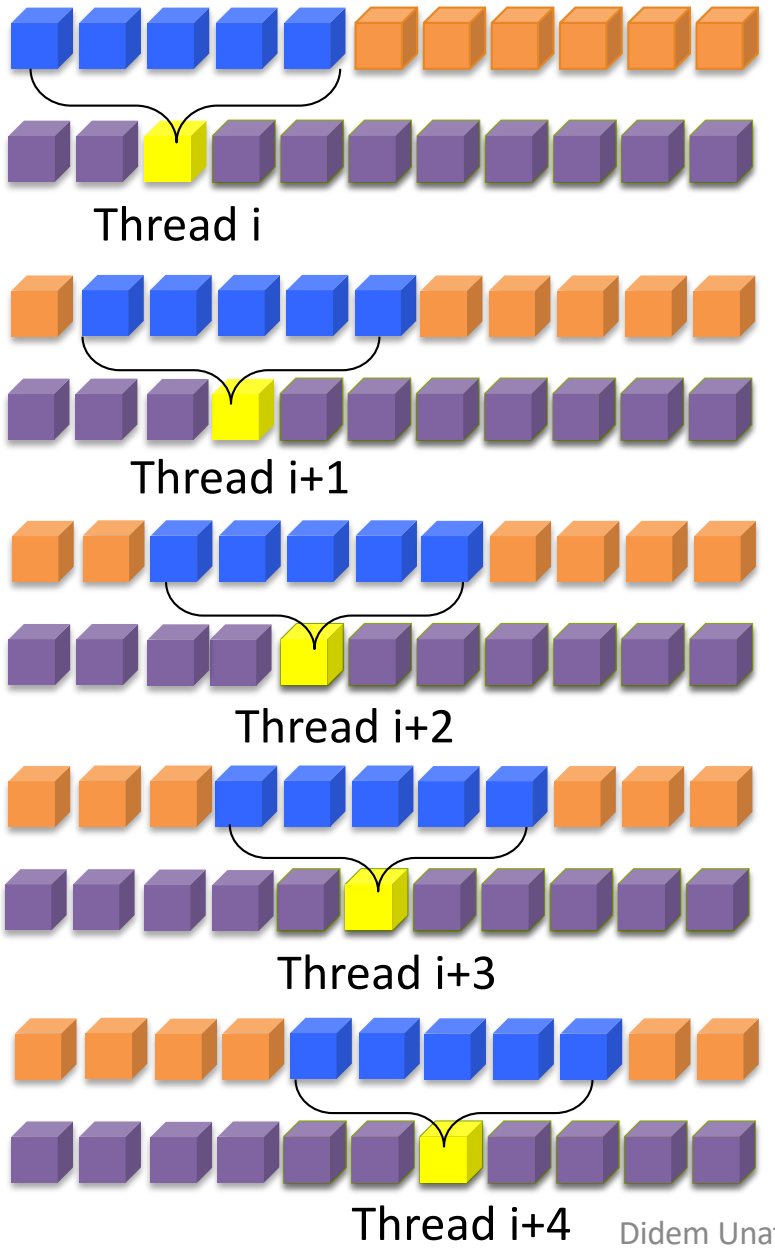


Simple 1D Stencil

```
// a simple version of 1D stencil
__global__ void Stencil_1D_simple(int *out, int *in, int NUM_ELEMENTS)
{
  // compute this thread's global index
  unsigned int i = blockDim.x * blockIdx.x + threadIdx.x + 2;

  if(i < NUM_ELEMENTS + 2 ){
    out[i] = alpha*(in[i-2]+ in[i-1]+
                    in[i+1]+ in[i+2]) +
                beta * in[i];
  }
}
```

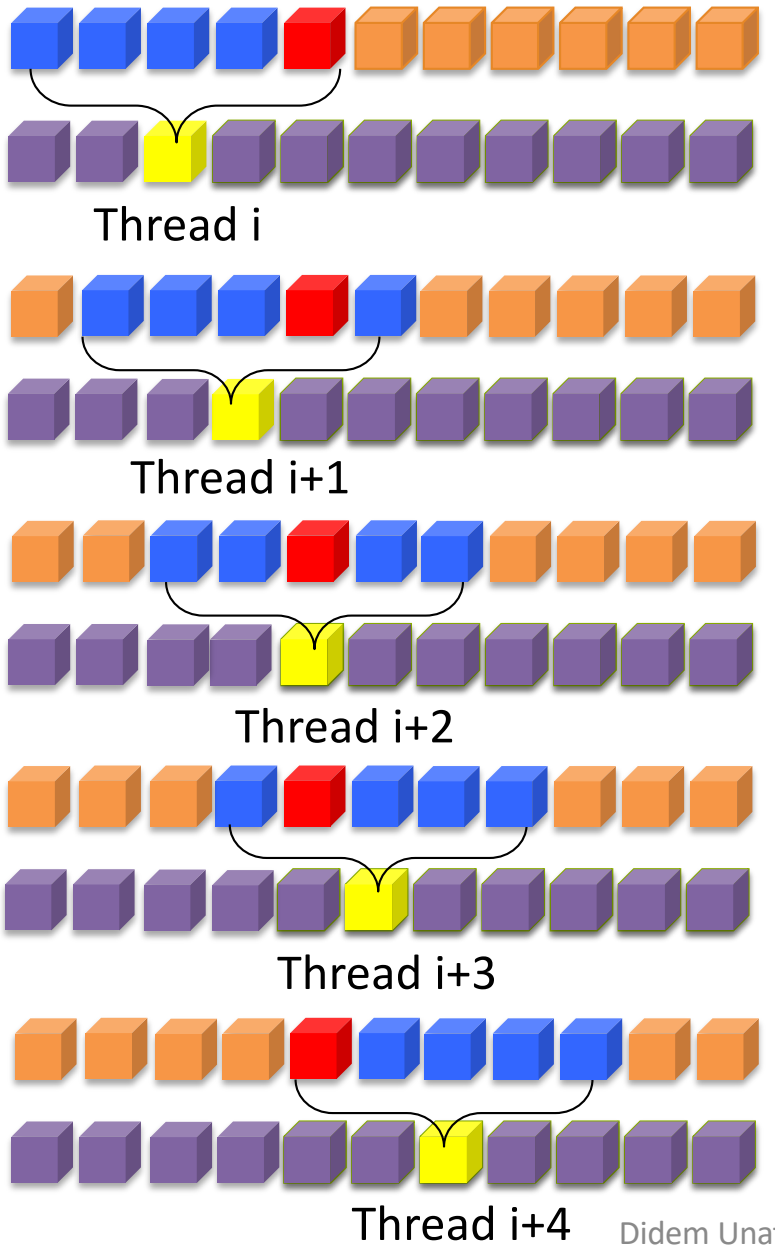


Redundant Memory Reads



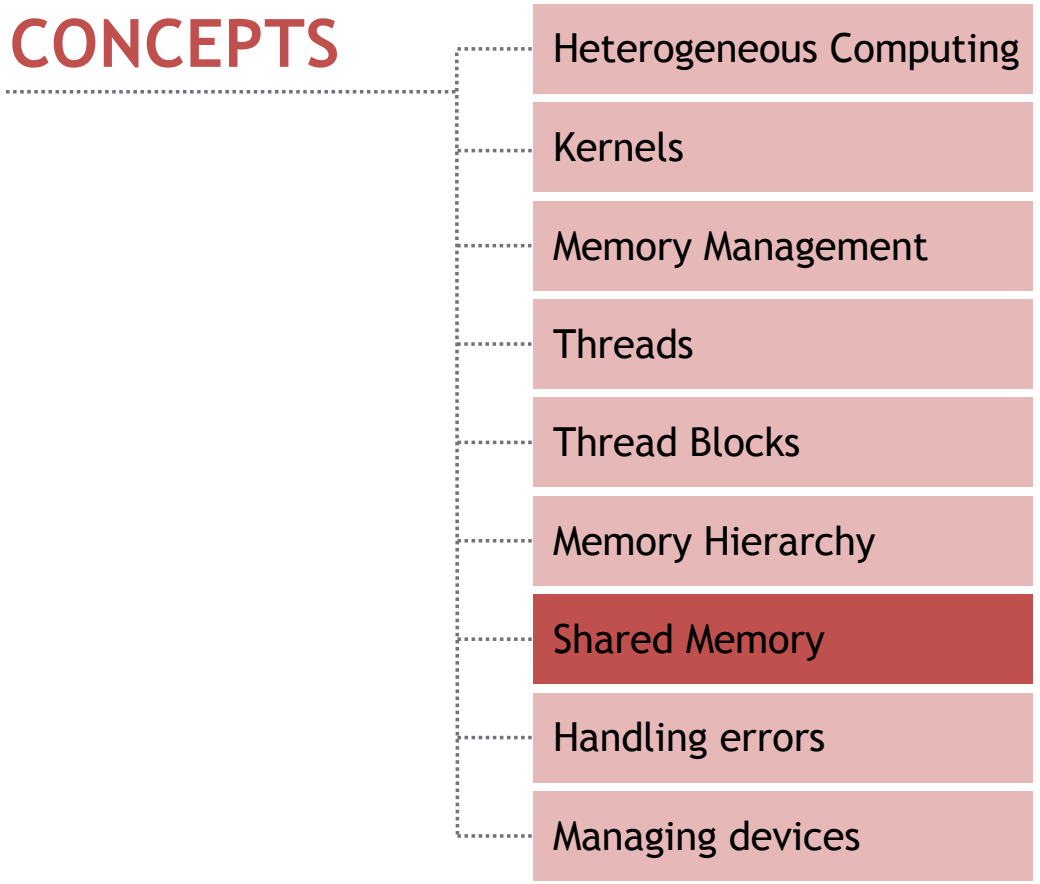
- Input elements are read several times
  - With radius 2, each input element is read five times;
  - Too many memory reads
- Redundant memory access leads to low performance
- Need to fix this issue

Redundant Memory Reads



- Input elements are read several times
  - With radius 2, each input element is read five times;
  - Too many memory reads
- Redundant memory access leads to low performance
- For example, the red element is read five times for different outputs

CONCEPTS


Heterogeneous Computing
Kernels
Memory Management
Threads
Thread Blocks
Memory Hierarchy
Shared Memory
Handling errors
Managing devices

Sharing Data Between Threads

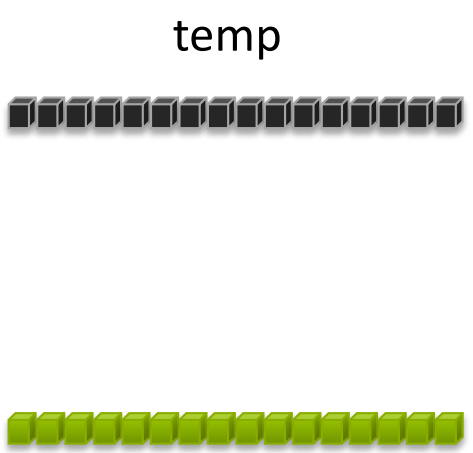
- Terminology: within a block, threads share data via **shared memory**
- Extremely fast on-chip memory, user-managed
- Declare using `__shared__`, allocated per block
- Data is not visible to threads in other thread blocks

Using Shared Memory

```
__global__ void stencil_1d(int *in, int *out)
{

    __shared__  int  temp[BLOCK_SIZE],

    //compute global indices from thread ids and block ids

    //compute local indices to reference temp in shared
      memory

    //read an element of a block from 'in' into shared
      memory

    // compute 1D stencil

    // store the result back to global memory

}
```

Stencil Kernel with Shared Memory

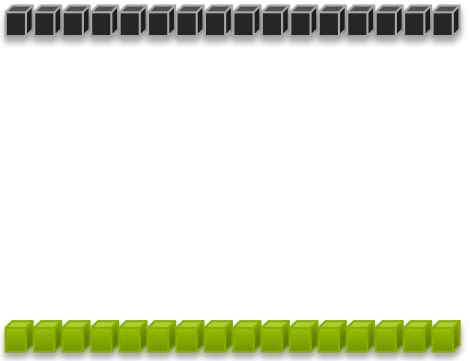
```
__global__ void stencil_1d(int *in, int *out) {

  __shared__ int temp[BLOCK_SIZE];
  int gi = threadIdx.x + blockIdx.x * blockDim.x;
  int li = threadIdx.x;

  // Read input elements into shared memory
  temp[li] = in[gi];
```



Compute using Shared Memory

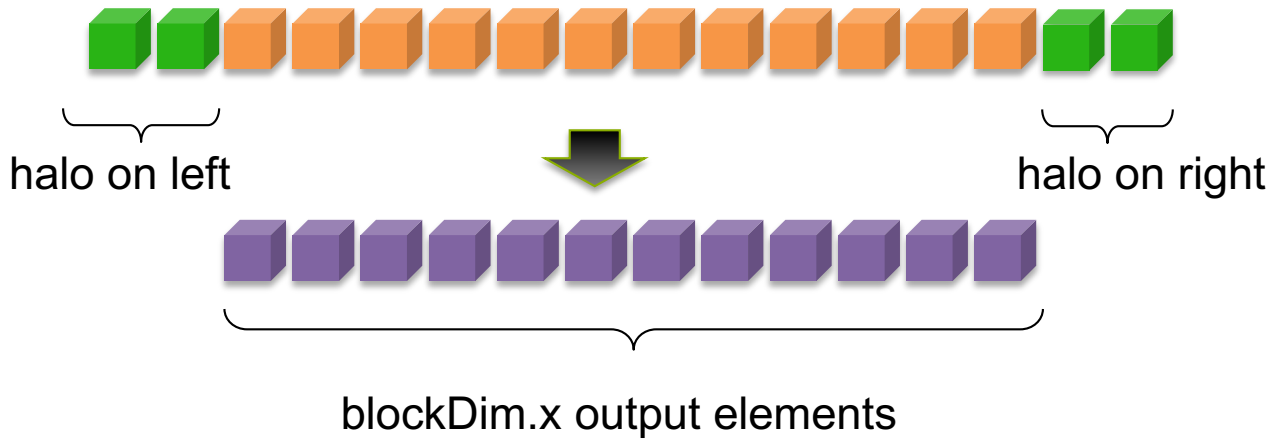
```
__global__ void stencil_1d(int *in, int *out) {

  __shared__ int temp[BLOCK_SIZE];
  int gi = threadIdx.x + blockIdx.x * blockDim.x;
  int li = threadIdx.x;

  // Read input elements into shared memory
  temp[li] = in[gi];

  // Apply the stencil
  int result = 0;
  result = alpha*(temp[li-2]+ temp[li-1]+ temp[li+1]+ temp[li+2])
           + beta * temp[li];

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

temp



Loading Ghost Cells/Halos into 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



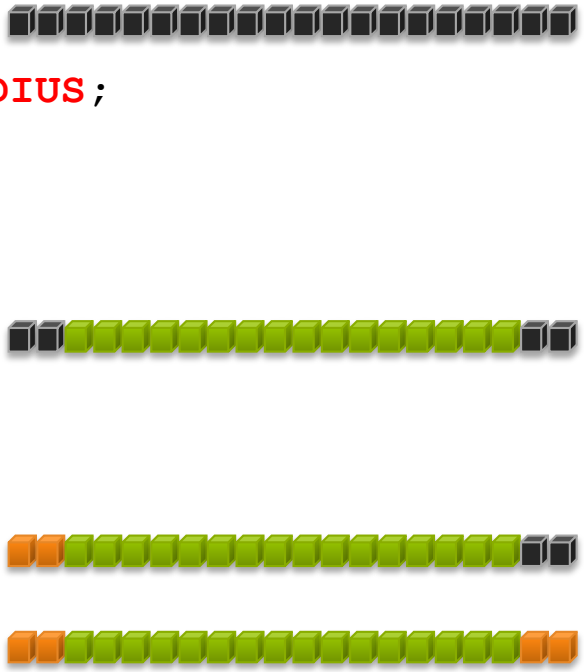
Using Shared Memory

```
__global__ void stencil_1d(int *in, int *out)
{

    __shared__  int  temp[BLOCK_SIZE + 2 * RADIUS],

    //compute global indices from thread ids and block ids

    //compute local indices to reference temp

    //read an element of a block from 'in' into shared
      memory

    //some threads need to load ghost cells as well

    // compute 1D stencil

    // store the result back to global memory

}
```

Halos (Ghost Cells)

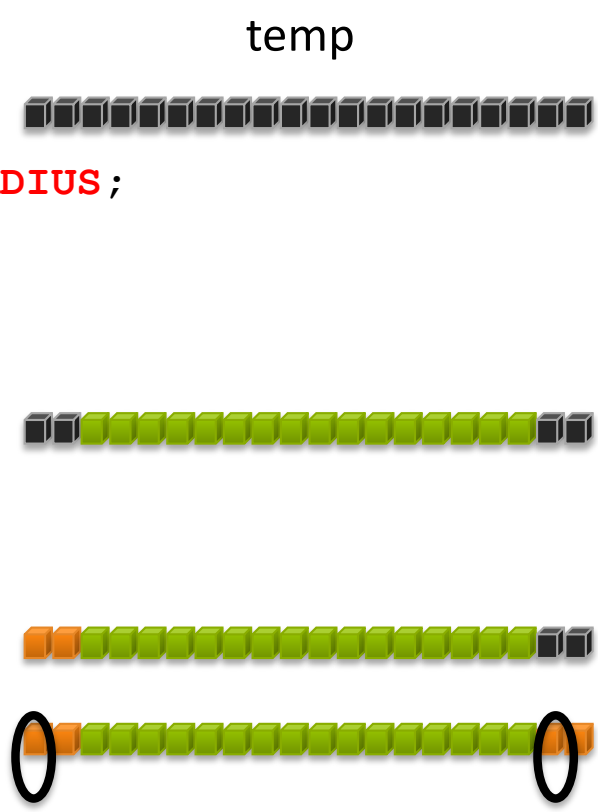
```
__global__ void stencil_1d(int *in, int *out) {

  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gi = threadIdx.x + blockIdx.x * blockDim.x + RADIUS;
  int li = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[li] = in[gi];

  // Load ghost cells (halos)
  if (threadIdx.x < RADIUS) {
    temp[li - RADIUS] = in[gi - RADIUS];
    temp[li + BLOCK_SIZE] =  in[gi + BLOCK_SIZE];
  }
```

temp



Halos (Ghost Cells)

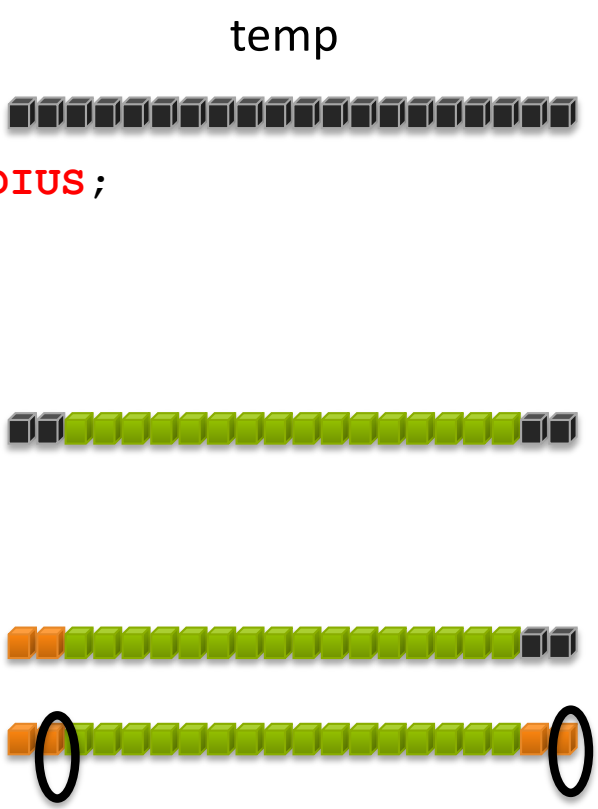
```
__global__ void stencil_1d(int *in, int *out) {

  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gi = threadIdx.x + blockIdx.x * blockDim.x + RADIUS;
  int li = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[li] = in[gi];

  // Load ghost cells (halos)
  if (threadIdx.x < RADIUS) {
    temp[li - RADIUS] = in[gi - RADIUS];
    temp[li + BLOCK_SIZE] =  in[gi + BLOCK_SIZE];
  }
```



These two elements are loaded by Thread 0

Halos (Ghost Cells)

```
__global__ void stencil_1d(int *in, int *out) {

  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gi = threadIdx.x + blockIdx.x * blockDim.x + RADIUS;
  int li = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[li] = in[gi];

  // Load ghost cells (halos)
  if (threadIdx.x < RADIUS) {
    temp[li - RADIUS] = in[gi - RADIUS];
    temp[li + BLOCK_SIZE] =  in[gi + BLOCK_SIZE];
  }
```



These two elements are loaded by Thread 1

Problem: Data Races

- The stencil example will not work…

- Suppose thread 15 reads the halo before thread 0 has fetched it…

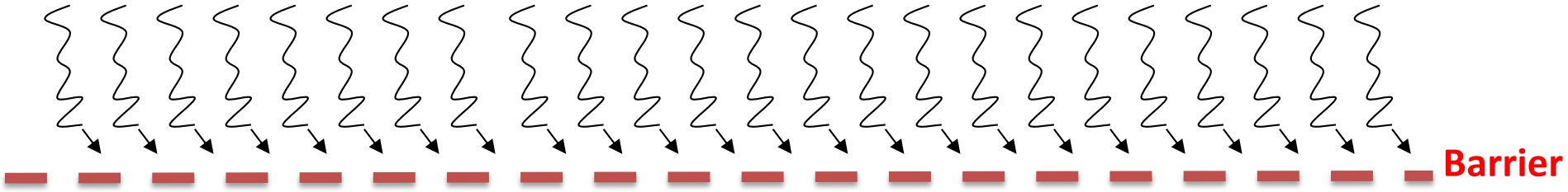
```
temp[li] = in[gi];                          Store at temp[17]
if (threadIdx.x < RADIUS) {
  temp[li – RADIUS = in[gi – RADIUS];          Skipped, threadIdx > RADIUS
  temp[li + BLOCK_SIZE] = in[gi + BLOCK_SIZE];    because my thread ID is 15
}

int result = 0;
result = temp[li + 2] ...;          Read from temp[18]
```

This red element hasn't been loaded into shared memory by thread 0

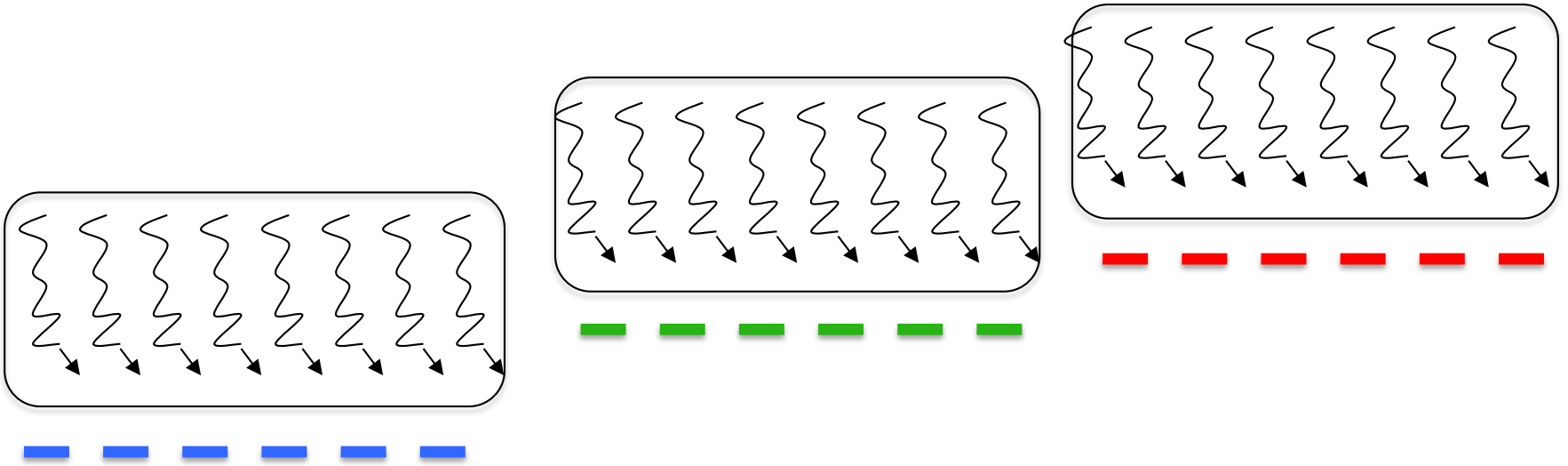
__syncthreads()

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

Global Barrier vs Thread Block Barrier

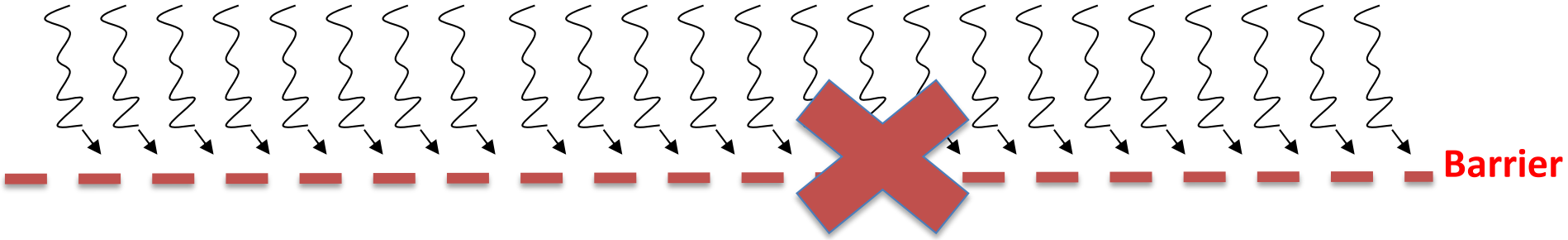


- Threads reaching a barrier point wait for all other threads to proceed

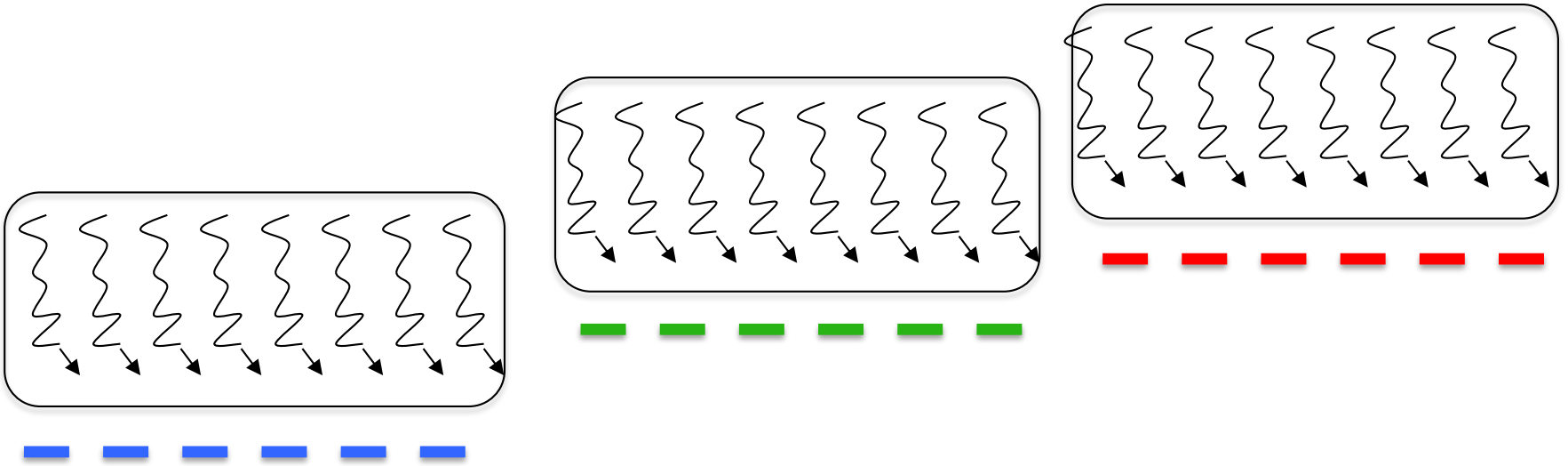


- Threads in a thread block can synchronize with `__syncthreads()`

Global Barrier vs Thread Block Barrier



- Threads reaching a barrier point wait for all other threads to proceed



- In GPUs, there is **NO a global barrier** to synchronize threads in ALL thread blocks within a kernel – kernel launch is required.
- Need to use **cooperative thread blocks** to synchronize within a kernel (new feature)

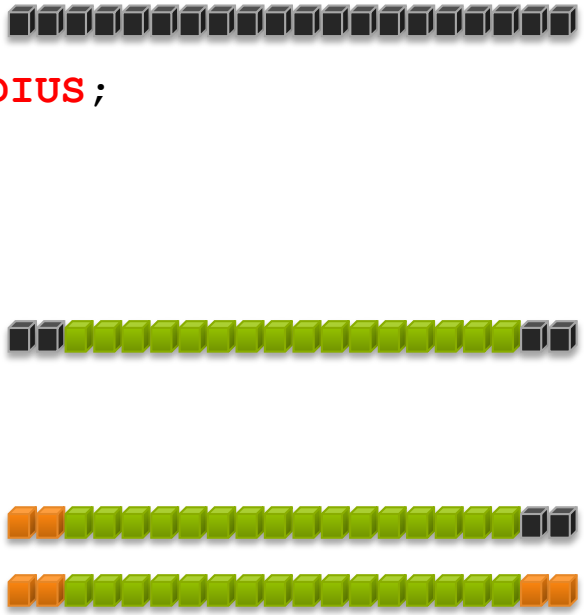
Using Shared Memory

```
__global__ void stencil_1d(int *in, int *out)
{

    __shared__  int  temp[BLOCK_SIZE + 2 * RADIUS],

    //compute global indices from thread ids and block ids

    //compute local indices to reference temp

    //read an element of a block from 'in' into shared
      memory

    //some threads need to load ghost cells as well

    //synchronize to avoid data races

    // compute 1D stencil

    // store the result back to global memory

}
```

Halos (Ghost Cells)

```
__global__ void stencil_1d(int *in, int *out) {

  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gi = threadIdx.x + blockIdx.x * blockDim.x + RADIUS;
  int li = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[li] = in[gi];

  // Load ghost cells (halos)
  if (threadIdx.x < RADIUS) {
    temp[li - RADIUS] = in[gi - RADIUS];
    temp[li + BLOCK_SIZE] =  in[gi + BLOCK_SIZE];
  }
  // Synchronize (ensure all the data is available)
  __syncthreads();
```

temp



Compute after synchronizing

```
 // Apply the stencil
int result = 0;
result = alpha*(temp[li-2]+ temp[li-1]+
                temp[li+1]+ temp[li+2])
         + beta * temp[li];

// Store the result
out[gi] = result;
```

Review (1 of 2)

- Launching parallel threads
  - Launch `N` blocks with `M` threads per block with `kernel<<<N,M>>>(…);`
  - Use `blockIdx.x` to access block index within grid
  - Use `threadIdx.x` to access thread index within block

- Allocate elements to threads:

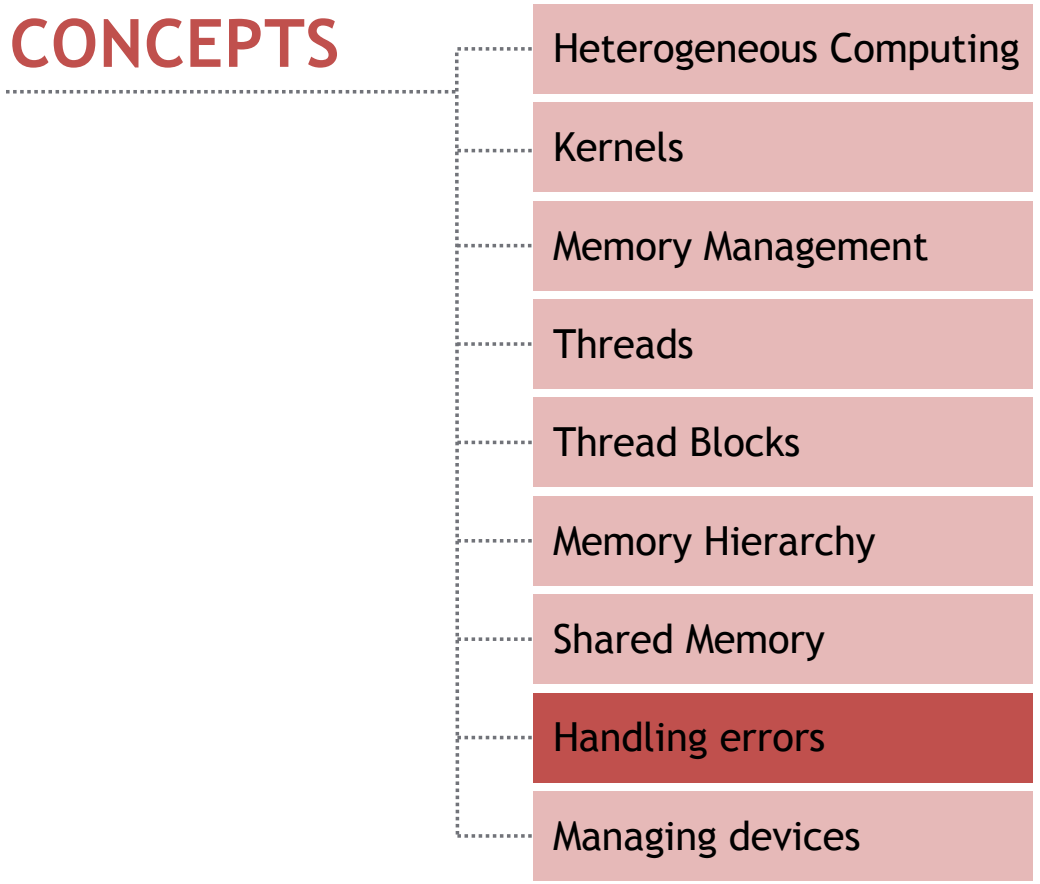
```
int index = threadIdx.x + blockIdx.x * blockDim.x
```

Review (2 of 2)

- 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

CONCEPTS



Reporting Errors

- All CUDA API calls return an error code (`cudaError_t`)
  - Error in the API call itself

    OR
  - Error in an earlier asynchronous operation (e.g. kernel)

- Get the error code for the last error:

```
cudaError_t cudaGetLastError(void)
```

- Get a string to describe the error:

```
char *cudaGetErrorString(cudaError_t)
```

```
printf("%s\n",
cudaGetErrorString(cudaGetLastError()));
```

Device Management

- Application can query and select GPUs

```
cudaGetDeviceCount(int *count)
cudaSetDevice(int device)
cudaGetDevice(int *device)
cudaGetDeviceProperties(cudaDeviceProp *prop, int device)
```

- Multiple host threads can share a device

- A single thread can manage multiple devices

  `cudaSetDevice(i)` to select current device

  `cudaMemcpy(…)` for peer-to-peer copies

requires OS and device support

Acknowledgments

- These slides are inspired and partly adapted from
  - Programming Massively Parallel Processors: A Hands On Approach, available from *http: David Kirk and Wen-mei Hwu, February 2010, Morgan Kaufmann Publishers, ISBN 0123814723.*
  - NVidia, *CUDA Programming Guide,* available from *http://docs.nvidia.com/cuda/cuda-c-programming-guide*
  - Why GPU Computing? By Mark Ebersole – Nvidia
    - https://developer.nvidia.com/cuda-education
  - CS193g Spring 2010 GPU Computing Course at Stanford
    - https://github.com/jaredhoberock/stanford-cs193g-sp2010
  - CUDA Education
    - https://developer.nvidia.com/cuda-education