

Programmazione di Sistemi ~~Embedded e~~ Multicore

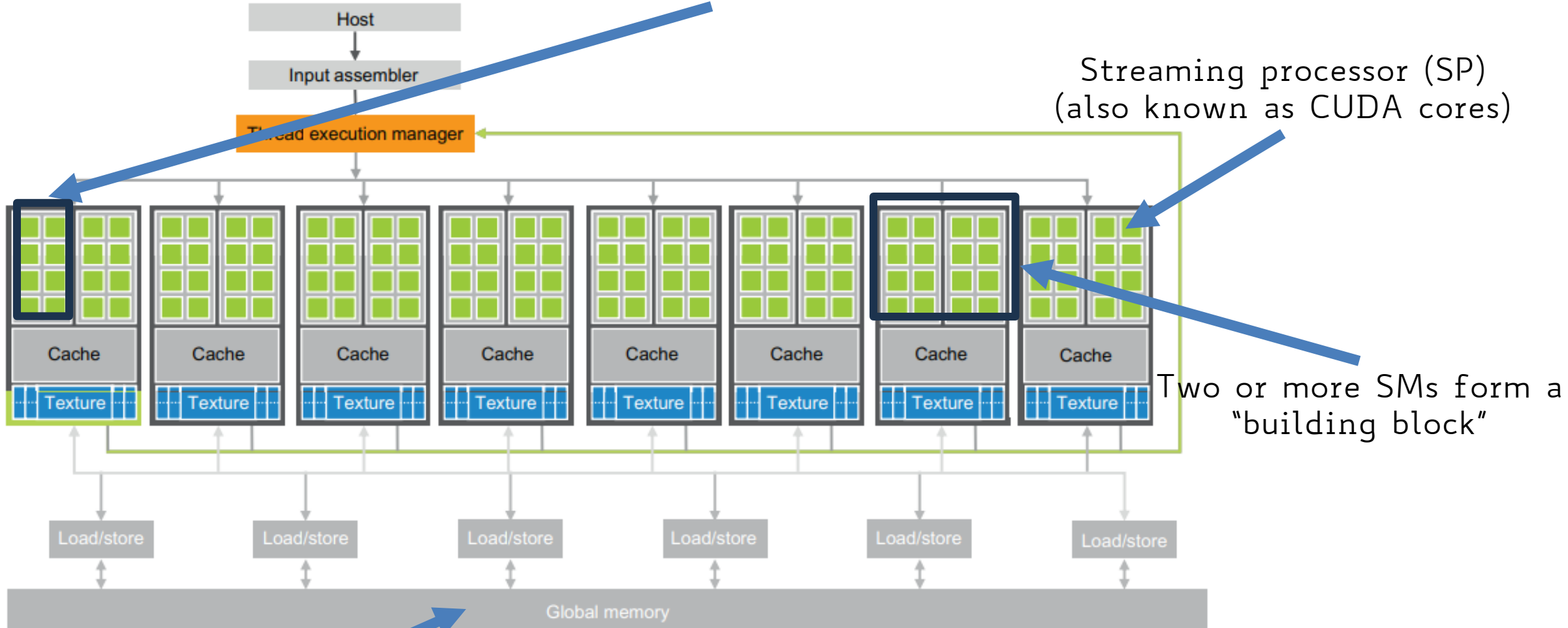
Teacher: Daniele De Sensi

Recap

Architecture of a CUDA-capable GPU

Streaming multiprocessor (SM)

(SPs on the same SM share control logic and instruction cache)



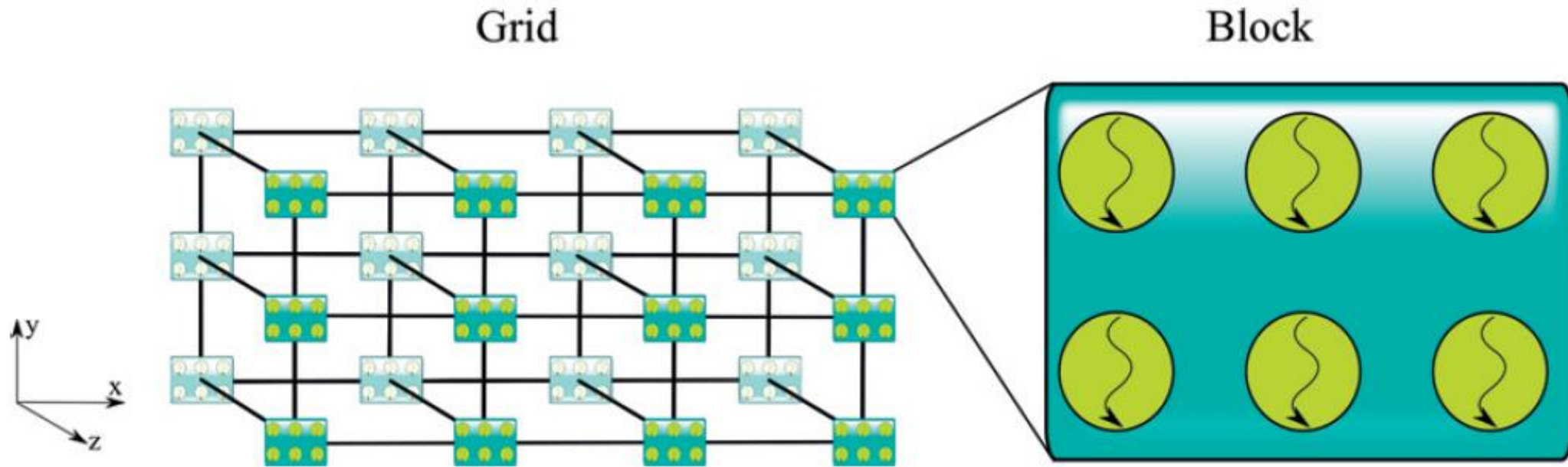
Streaming processor (SP)
(also known as CUDA cores)

Two or more SMs form a
"building block"

High-Bandwidth Memory

How many CUDA cores?
15,000 on NVIDIA H100

CUDA Execution Model



- CUDA organizes the threads in a 6-D structure (lower dimensions are also possible).
- Each thread has a position in a 1D, 2D, or 3D *block*
- Each block has a position in a 1D, 2D or 3D *grid*
- Each thread is aware of its position in the overall structure, via a set of **intrinsic variables**/structures. With this information a thread can map its position to the subset of data that it is assigned to.

Hello World in CUDA

Can be called from the host or the device
Must run on the device

A kernel is always declared as void
Computed result must be copied explicitly from GPU to host memory

```
// File : hello.cu
#include <stdio.h>
#include <cuda.h>

__global__ void hello()
{
    printf("Hello world\n");
}

int main()
{
    hello<<<1,10>>>();
    cudaDeviceSynchronize();
    return 1;
}
```

printf in device
code is supported
by CC 2.0 and above

Blocks until the CUDA kernel terminates
(like a barrier, kernel launch is asynchronous)

Compile for
CC 2.0

- To compile and run:
\$ nvcc -arch=sm_20 hello.cu -o hello
\$./hello

Thread Scheduling

- Each thread runs on a streaming processor (CUDA core)
- Cores on the same SM share the control unit (i.e., they must **synchronously** execute the same instruction)
- Different SMs can run different kernels
- Each block runs on an SM (i.e., I can't have a block spanning over multiple SMs, but I can have more blocks running on the same SM)
- Once a block is fully executed, the SM will run the next one
- Not all the threads in a group run concurrently

Warps

- They are executed in groups called *warps* (in current GPUs, the size of a warp is 32 – might change in the future, check the *warpSize* variable)
- Thread in a block are split into warps according to their intra-block ID (i.e., the first 32 threads in a block belong to the same warp, the next 32 threads to a different warp, etc...)
- All threads in a warp executed according to the Single Instruction, Multiple Data (SIMD) model—i.e., at any instant in time, one instruction is fetched and executed for all threads in the warp. Consequently, all threads in a warp will always have the same execution timing.
- Several warp schedulers (e.g., 4) can be present on each SM. I.e., multiple (e.g., 4) warps can run at the same time, each possibly following a different execution path

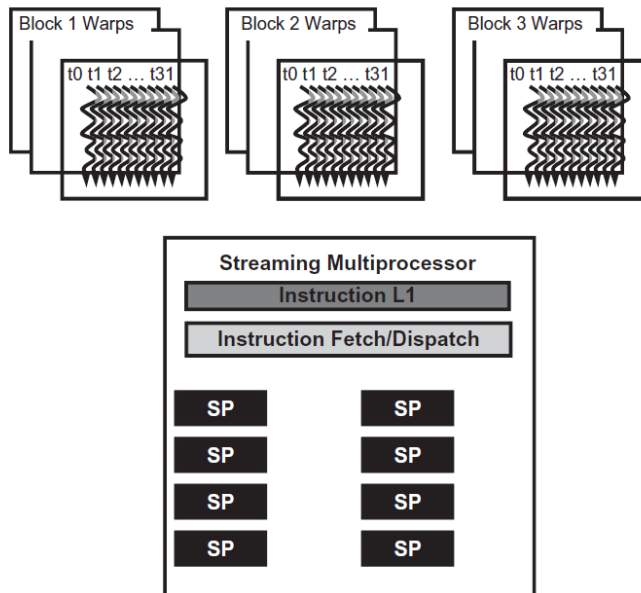
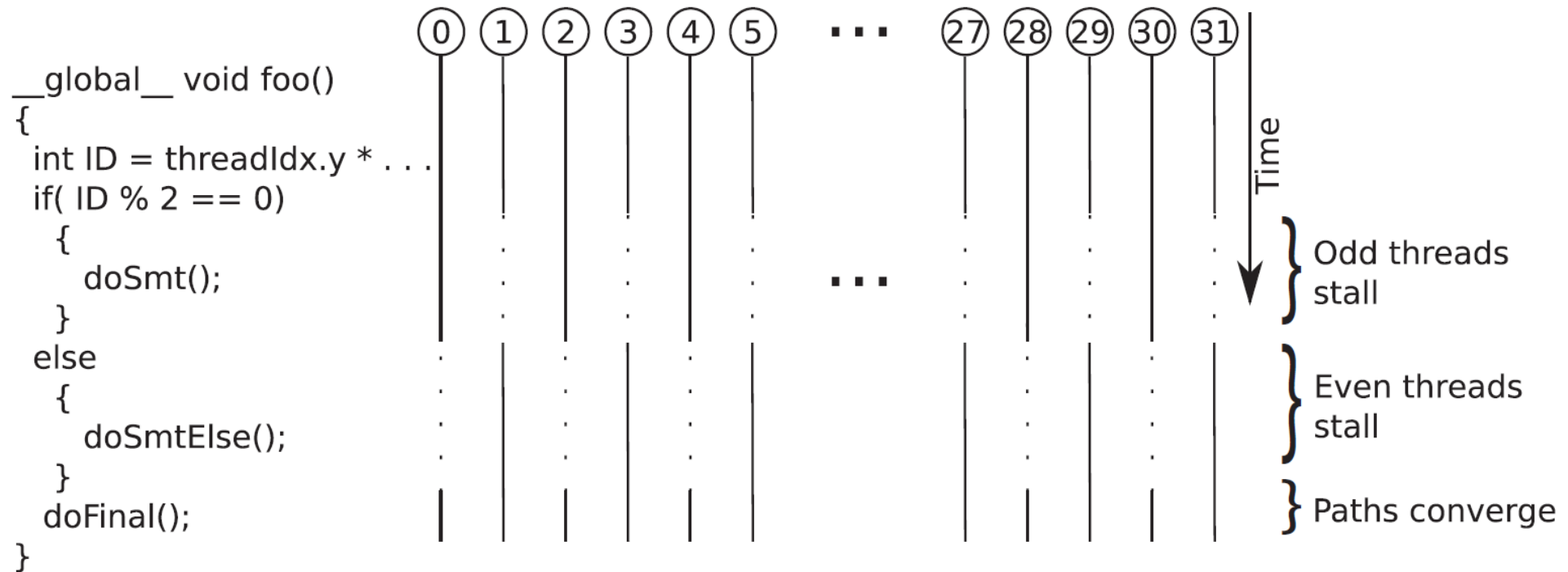


FIGURE 3.13

Blocks are partitioned into warps for thread scheduling.

Warp Divergence

- All threads in a warp executed according to the Single Instruction, Multiple Data (SIMD) model—i.e., at any instant in time, one instruction is fetched and executed for all threads in the warp. Consequently, all threads in a warp will always have the same execution timing.
- What happens if the result of a conditional operation leads them to different paths?
- All the divergent paths are evaluated (if threads branch into them) in sequence until the paths merge again.
- The threads that do not follow the path currently being executed are stalled.




How to install CUDA

<https://docs.nvidia.com/cuda/cuda-installation-guide-linux/>

Account Creation for the Cluster


Account Cluster PMC 24/25

desensi@di.uniroma1.it [Switch account](#)  Draft saved

* Indicates required question


Email *

☐ Record desensi@di.uniroma1.it as the email to be included with my response

 This is a required question


Username (lowercase) *

Your answer

 This is a required question

Password *

Your answer

 This is a required question

A copy of your responses will be emailed to desensi@di.uniroma1.it.

[Submit](#) [Clear form](#)

https://docs.google.com/forms/d/e/1FAIpQLScJbGoKOKeAjatPkBRB6bcKDe3_IQTYSHVzA7vFnjpYmBxsA/viewform?usp=sf_link

Questions?

Context Switching

- Usually a SM has more *resident* blocks/warps than what it is able to concurrently run
- Each SM can switch seamlessly between warps
- Each thread has its own private execution context that is maintained on-chip (i.e., context switch comes for free)
- When an instruction to be executed by a warp needs to wait for the result of a previously initiated long-latency operation, the warp is not selected for execution (e.g., memory read, long latency floating-point operations).
- Instead, another resident warp that is no longer waiting for results will be selected for execution.
- This mechanism of filling the latency time of operations with work from other threads is often called "latency tolerance" or "latency hiding"
- Given a sufficient number of warps, the hardware will likely find a warp to execute at any point in time, thus making full use of the execution hardware in spite of these long-latency operations.
- With warp scheduling, the long waiting time of warp instructions is "hidden" by executing instructions from other warps.
- This ability to tolerate long-latency operations is the main reason GPUs do not dedicate nearly as much chip area to cache memories and branch prediction mechanisms as do CPUs.

Example 1

- A CUDA device allows up to 8 blocks and 1024 threads per SM, and 512 threads per block
- Shall we use 8x8, 16x16, or 32x32 thread blocks?
- 8x8 blocks:
 - We would have 64 threads per block
 - To fill the 1024 threads we can have for each SM, we would need $1024/64 = 16$ blocks
 - However, we can have at most 8 blocks per SM, thus we would end with only $64 \times 8 = 512$ threads per SM
 - We are not fully utilizing the resources. Most likely, at some time, the scheduler might not find threads to schedule when some thread is waiting for long-latency operations
- 16x16
 - We would have 256 threads per block
 - To fill the 1024 threads we can have for each SM, we would need $1024/256 = 4$ blocks
 - This would allow to have 1024 threads on the SM, so there will be a lot of opportunities for latency hiding
- 32x32
 - We would have 1024 threads per block, which is higher than the 512 threads per block we can have
- In practice, it is more complicated than this, the usage of other resources as registers and shared memory must be taken into account (we will see how)

Example 2

- A CUDA device allows up to 8 blocks and 1536 threads per SM, and 1024 threads per block
- Shall we use 8x8, 16x16, or 32x32 thread blocks
- 8x8 blocks:
 - We would have 64 threads per block
 - To fill the 1536 threads we can have for each SM, we would need $1536/64 = 24$ blocks
 - Only 512 threads (8x8x8) would go into each SM
- 16x16
 - We would have 256 threads per block
 - To fill the 1536 threads we can have for each SM, we would need $1536/256 = 6$ blocks
 - We achieve full capacity (unless other resources constraints come into play)
- 32x32
 - We would have 1024 threads per block. Only one block can fit (two would bring the number of threads to 2048, which is higher than 1536)
 - Thus, we would use only 2/3 of the thread capacity of the SM (1024 out of 1536)

Example 3

- A grid of $4 \times 5 \times 3$ blocks, each made of 100 threads
- The GPU has 16 SMs
- Thus, we have $4 \times 5 \times 3 = 60$ blocks, that need to be distributed over 60 16SMs
 - Let's assume that they are distributed round-robin
 - 12 SMs will receive 4 blocks, and 6 SMs will receive 3 blocks
 - Inefficient! While the first 12 SMs process the last block, the other 6 SMs are idle
- A block contains 100 threads, which are divided into $100/32 = 4$ warps
 - The first three warps have 32 threads, and the last one have 4 threads ($32+32+32+4$)
 - Assume we can only schedule a warp at a time (e.g., because we have 32 CUDA cores per SM)
 - The last warp would only use 4 out of the 32 available cores (87.5% of the cores on each SM will be unused)
- Take home message: pay attention to how you define the grid and block sizes (we will come back to this again)
- General hint: The number of threads in a block should be a multiple of 32 (in general, a multiple of the warp size)

Device properties

Example : Listing all the GPUs in a system

```
int deviceCount = 0;
cudaGetDeviceCount(&deviceCount);
if(deviceCount == 0)
    printf("No CUDA compatible GPU exists.\n");
else
{
    cudaDeviceProp pr;
    for(int i=0;i<deviceCount;i++)
    {
        cudaGetDeviceProperties(&pr, i);
        printf("Dev #%i is %s\n", i, pr.name);
    }
}
```

Device Properties

```
struct cudaDeviceProp{
    char name[256]; // A string identifying the device
    int major;      // Compute capability major number
    int minor;      // Compute capability minor number
    int maxGridSize [3];
    int maxThreadsDim [3];
    int maxThreadsPerBlock;
    int maxThreadsPerMultiProcessor;
    int multiProcessorCount;
    int regsPerBlock; // Number of registers per block
    size_t sharedMemPerBlock;
    size_t totalGlobalMem;
    int warpSize;
    . . .
};
```

Memory Management

Memory Accesses

- Data allocated on host memory is not visible from the GPU, and viceversa

```
int *mydata = new int[N];  
. . . // populating the array  
foo<<<grid, block>>>(mydata, N);    // NOT POSSIBLE!
```

- Instead, it must explicitly be copied from/to host to GPU

Memory Allocation and Copy

```
// Allocate memory on the device.
cudaError_t cudaMalloc ( void** devPtr, // Host pointer address,
                                // where the address of
                                // the allocated device
                                // memory will be stored
                                size_t size ) // Size in bytes of the
                                                // requested memory block

// Frees memory on the device.
cudaError_t cudaFree ( void* devPtr ); // Parameter is the host
                                         // pointer address, returned
                                         // by cudaMalloc

// Copies data between host and device.
cudaError_t cudaMemcpy ( void* dst, // Destination block address
                        const void* src, // Source block address
                        size_t count, // Size in bytes
                        cudaMemcpyKind kind ) // Direction of copy.
```

`cudaError_t` is an enumerated type. If a CUDA function returns anything other than `cudaSuccess (0)`, an error has occurred.

Memory Copy Kind

The `cudaMemcpyKind` parameter of `cudaMemcpy` is also an enumerated type. The `kind` parameter can take one of the following values:

- `cudaMemcpyHostToHost = 0`, Host to Host
- `cudaMemcpyHostToDevice = 1`, Host to Device
- `cudaMemcpyDeviceToHost = 2`, Device to Host
- `cudaMemcpyDeviceToDevice = 3`, Device to Device (for multi-GPU configurations)
- `cudaMemcpyDefault = 4`, used when Unified Virtual Address space capability is available (see [Section 6.7](#))

Caveat

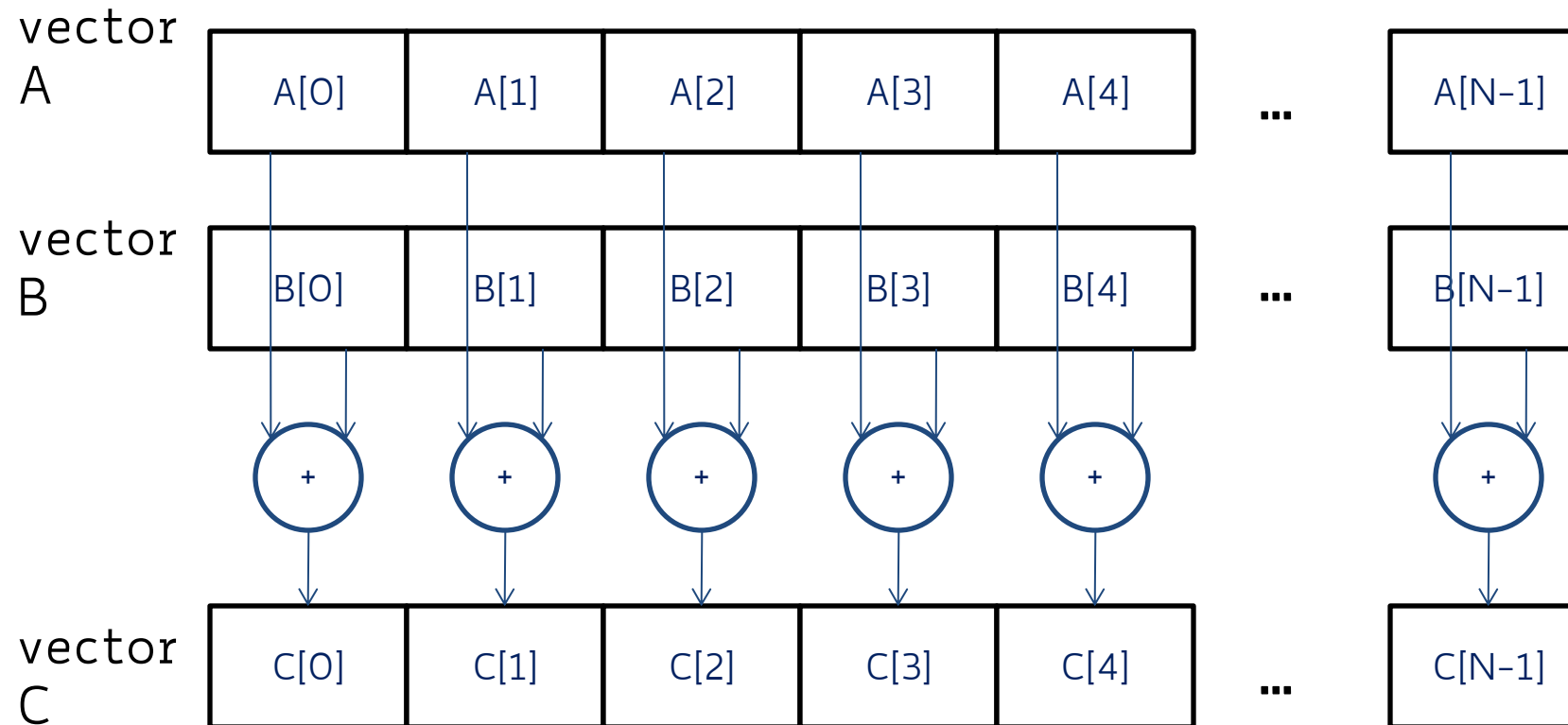
- Pointers allocated through `malloc()` and `cudaMalloc()` belong to two different address spaces: a pointer to the GPU memory cannot be accessed from the host, and vice-versa
- **Unified Memory** (from CUDA 6) allows accessing CPU and GPU memory using a single addressing space

Questions?

Example: Vector addition

Vector Addition – Conceptual View

ATTENTION: Don't make any assumption on the order in which threads are executed
(Depends on how the GPU decides to schedule the blocks)



Example: Vector Addition

h_* to indicate
data allocated
on host memory

d_* to indicate data
allocated on device
(GPU) memory

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
    int size = n * sizeof(float);
    float *d_A, *d_B, *d_C;

    cudaMalloc((void **) &d_A, size);
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_B, size);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    cudaMalloc((void **) &d_C, size);

    // Kernel invocation code - to be shown later
    ...

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory for A, B, C
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
}
```

FIGURE 2.10

A more complete version of `vecAdd()`.

Example: Vector Addition – Error check

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
    int size = n * sizeof(float);
    float *d_A, *d_B, *d_C;


    cudaMalloc((void **) &d_A, size);
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_B, size);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    cudaMalloc((void **) &d_C, size);

    // Kernel invocation code - to be shown later
    ...

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory for A, B, C
    cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}
```



```
cudaError_t err=cudaMalloc((void **) &d_A, size);
if (error !=cudaSuccess) {
    printf("%s in %s at line %d\n", cudaGetErrorString(err),__
FILE__,__LINE__);
    exit(EXIT_FAILURE);
}
```

Example: Vector Addition – Better Error check

```
#define CUDA_CHECK_RETURN(value) { \
    cudaError_t _m_cudaStat = value; \
    if (_m_cudaStat!= cudaSuccess) { \
        fprintf(stderr, "Error %s at line %d in file %s\n", \
            cudaGetErrorString(_m_cudaStat), \
            __LINE__, __FILE__); \
        exit(1); \
    } }
```

```
CUDA_CHECK_RETURN (cudaMalloc ((void **) &da, sizeof (int) * N));
```

Example: Vector Addition

```
void vecAdd(float* A, float* B, float* C, int n)
{
    int size = n * sizeof(float);
    float *d_A, *d_B, *d_C;

    cudaMalloc((void **) &d_A, size);
    cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_B, size);
    cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);

    cudaMalloc((void **) &d_C, size);

    vecAddKernel<<<ceil(n/256.0), 256>>>(d_A, d_B, d_C, n);

    cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory for A, B, C
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
}
```

- What if n is not a multiple of 256?
 - I might run more threads than number of elements in the vector
 - Each thread must check if it needs to process some elements or not
-
- Each block has 256 threads
 - We have $n/256$ blocks

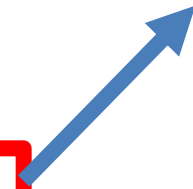



FIGURE 2.15

A complete version of the host code in the vecAdd.function.

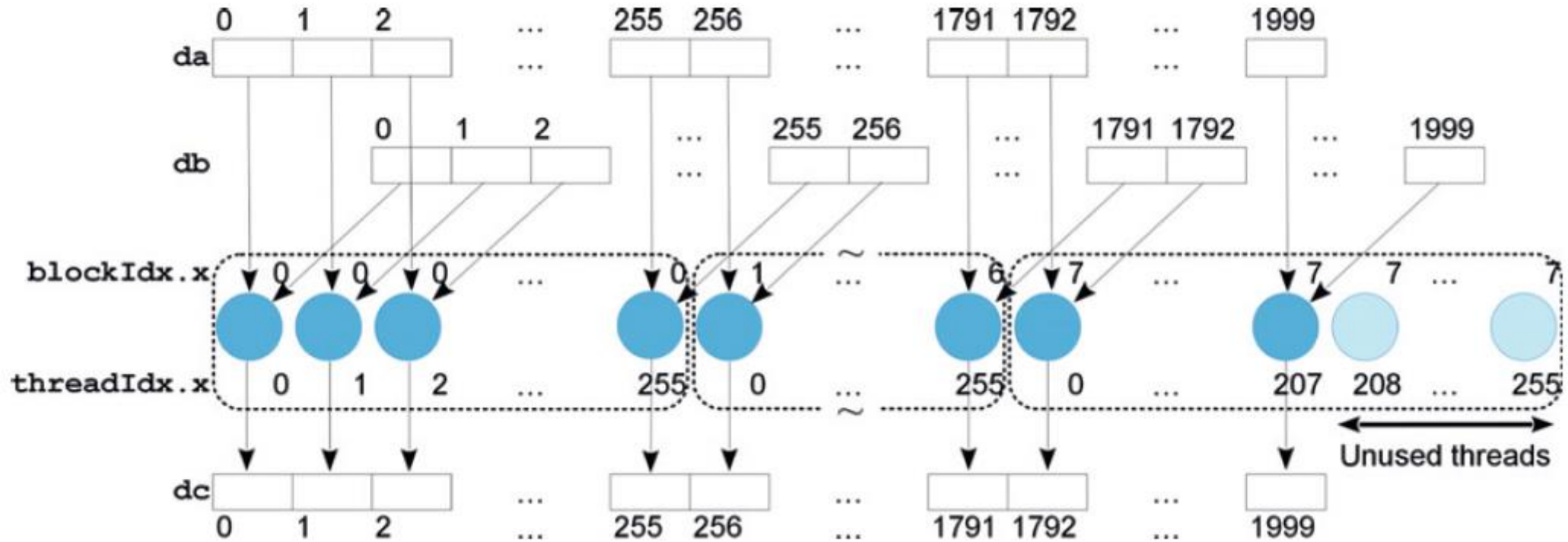
Example: Vector Addition (Kernel)

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__
void vecAddKernel(float* A, float* B, float* C, int n)
{
    int i = blockDim.x*blockIdx.x + threadIdx.x;
    if(i<n) C[i] = A[i] + B[i];
}
```



We might have more threads than elements in the array (if the number of elements is not a multiple of the block size)

Example: Vector Addition

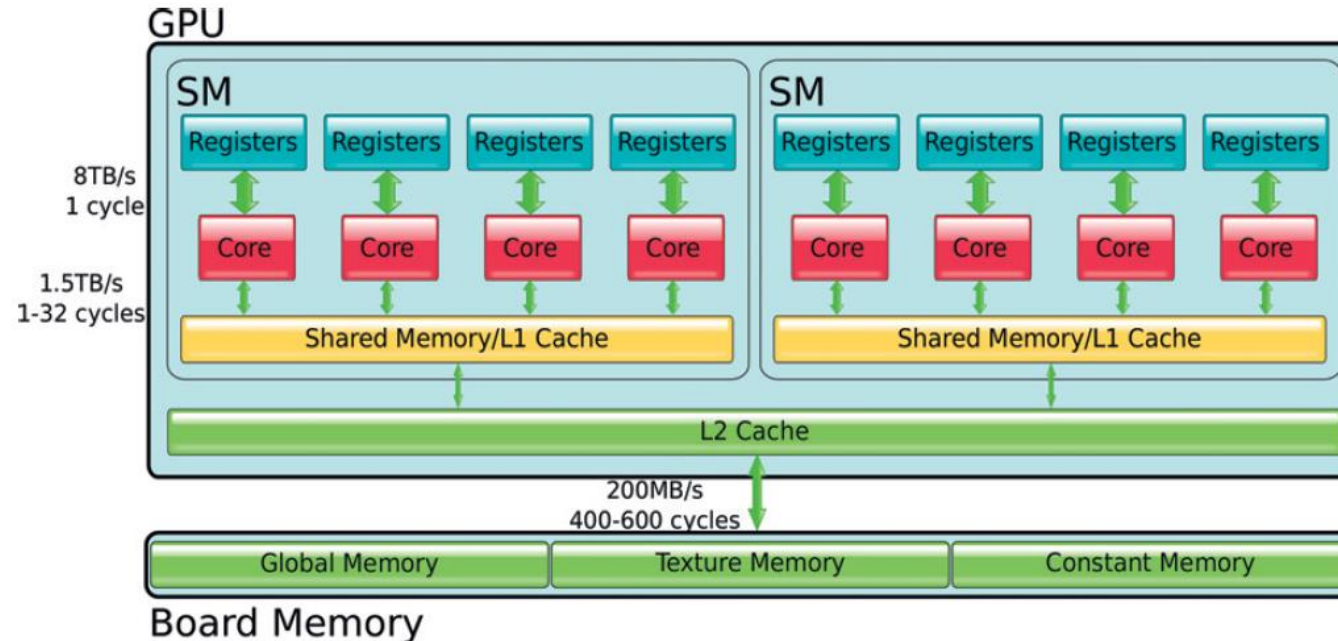


Questions?

Memory Types

Memory Types

There are multiple types of memories, some on-chip and some off-chip.



- **Registers:** Holding local variables
- **Shared Memory:** Fast on-chip memory to hold frequently used data. Can be used to exchange data between the cores of the same SM.
- **L1/L2 Cache:** Transparent to the programmer
- **Global memory:** Main part of the off-chip memory. High capacity but relatively slow. The only part accessible by the host through CUDA functions
- **Texture and surface memory:** Content managed by special hardware that permits fast implementation of some filtering/interpolation operator
- **Constant memory:** Can only store constants. It is cached, and allows broadcasting of a single value to all threads in a warp (less appealing on newer GPUs that have a cache anyway)

Memory Types

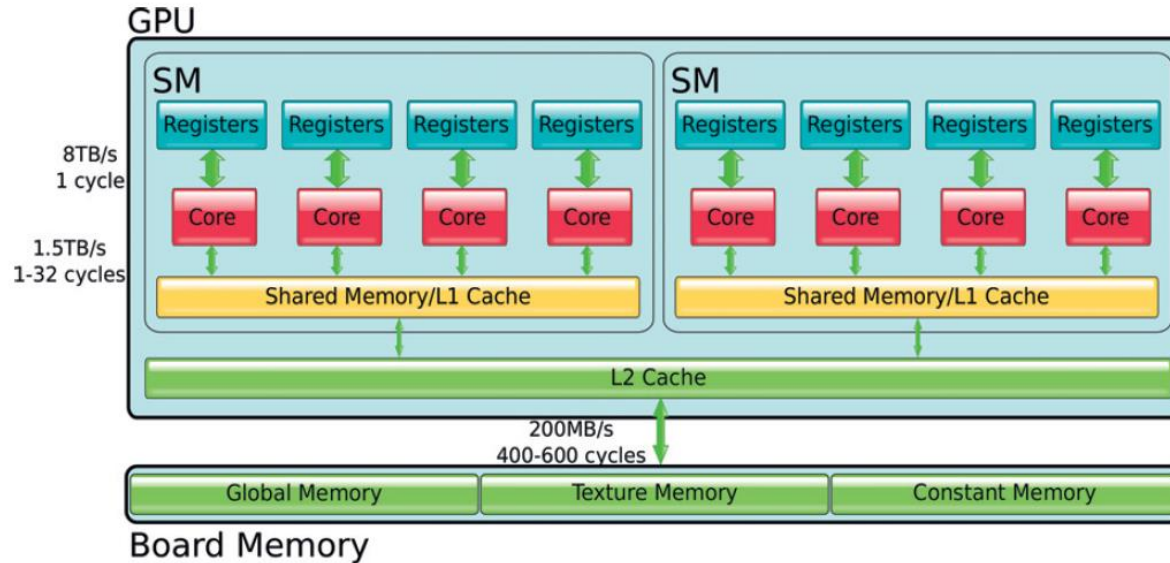


Table 4.1 CUDA Variable Type Qualifiers

Variable declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	Thread	Kernel
Automatic array variables	Global	Thread	Kernel
<code>__device__ __shared__ int SharedVar;</code>	Shared	Block	Kernel
<code>__device__ int GlobalVar;</code>	Global	Grid	Application
<code>__device__ __constant__ int ConstVar;</code>	Constant	Grid	Application

Registers

- Used to store local variables to a thread
- Registers on a core are split among the resident threads.
- Compute capability determines the maximum number of registers that can be used per thread.
- If this number is exceeded, local variables are allocated in the global off-chip memory (slow)
 - Spilled variables could be cached in the L1 on-chip cache.
- The compiler will decide which variables will be allocated in the registers and which will spill over to global memory

```
$ nvcc -Xptxas -v -arch=sm_20 warpFixMultiway.cu
ptxas info      : 0 bytes gmem, 14 bytes cmem[2]
ptxas info      : Compiling entry function '_Z3foov' for 'sm_20'
ptxas info      : Function properties for _Z3foov
      8 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info      : Used 9 registers, 32 bytes cmem[0], 4 bytes cmem[16]
```

Registers

- The number of maximum registers per thread influences the maximum number of resident threads we can have on an SM
- E.g.,
 - A kernel is using 48 registers
 - It is invoked as blocks of 256 threads
 - Each block requires $48 \times 256 = 12,288$ registers
 - Let's assume the target GPU has 32,000 registers per SM
 - Let's assume the target GPU can have up to 1,536 resident threads per SM
 - Thus, each SM could have 2 resident blocks ($12,288 \times 2 < 32,000$)
 - I.e., each SM could have $2 \times 256 = 512$ resident threads
 - This is much below that the maximum limit of 1,536 threads
 - This undermines the possibility to hide latency
- NVIDIA defines as *occupancy* the ratio of resident warps over the maximum possible resident warps

$$\text{occupancy} = \frac{\text{resident_warps}}{\text{maximum_warps}} = \frac{2 \cdot \frac{256 \text{ threads}}{32 \text{ threads/warp}}}{48 \text{ warps}} = \frac{16}{48} = 33.3\%$$

Registers

- An occupancy close to 1 is desirable (the closer to 1, the higher the opportunities to swap between threads and hide latencies)
- The occupancy of a given kernel can be analyzed through a profiler (we will see how)
- How to increase occupancy?
 - Reduce the number of registers required by the kernel (e.g., by avoiding to have too many temporary variables, a high distance between reuse of the same variables, etc...)
 - Use a GPU with higher registers per thread limits

Questions?

Shared Memory (on-chip)

- On-chip memory
 - Different from registers. Registers data is private to threads, shared memory is shared among threads
- Can be seen as a user-managed L1 cache (scratchpad)
- Can be used as:
 - A place to hold frequently used data that would otherwise require a global memory access
 - As a way for cores on the same SM to share data
- The `__shared__` specifier can be used to indicate that some data must go in the shared on-chip memory rather than on the global memory
- Shared memory vs. L1 cache:
 - Both are on-chip. The former is managed by the programmer the latter automatically
 - In some cases, managing it manually (i.e., using the shared memory), might provide better performance
 - E.g., you do not have any guarantee that the data you need will be in the L1 cache, but with the explicitly managed shared memory, you can control that

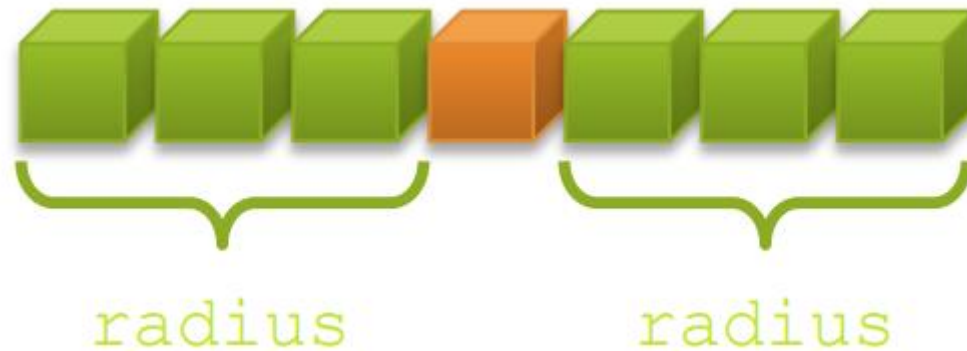
Shared Memory Example: 1D Stencil

- ▶ 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:



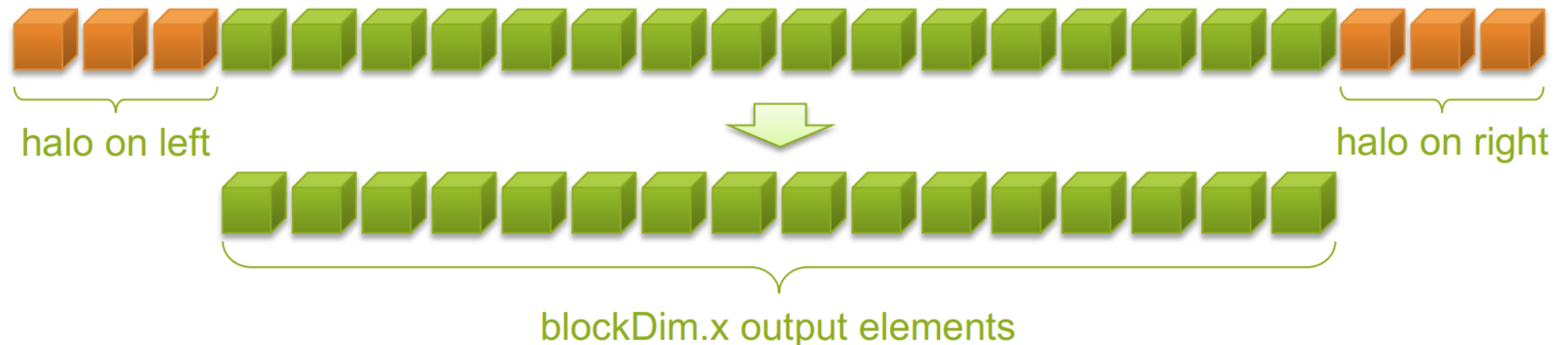
Shared Memory Example: 1D Stencil

- ▶ 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 Example: 1D Stencil

- ▶ Cache data in shared memory
 - ▶ Read $(\text{blockDim.x} + 2 * \text{radius})$ 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



Shared Memory Example: 1D Stencil

```
__global__ void stencil_1d(int *in, int *out) {  
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];  
    ...  
}
```



Shared Memory Example: 1D Stencil

```
__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;
```



Shared Memory Example: 1D Stencil

```
__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];  
}
```



Shared Memory Example: 1D Stencil

```
__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];  
    }
```



Shared Memory Example: 1D Stencil

```
__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];  
    }  
}
```



Shared Memory Example: 1D Stencil

```
__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];
    }

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

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



What if the first warp of the block gets executed, and thread 31 starts computing the result before thread 32 copies in temp the element in position 32?

We need to synchronize the threads, to guarantee that the stencil is applied only after the data has been loaded in the shared memory

Shared Memory Example: 1D Stencil

- ▶ `void __syncthreads();`
- ▶ Synchronizes all threads within a block
 - ▶ Used to prevent RAW / WAR / WAW hazards
- ▶ All threads must reach the barrier

Shared Memory Example: 1D Stencil (Correct)

```
__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 the data is available)
    __syncthreads();

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

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

Questions?