# CPSC 4240/5240: Parallel Programming Techniques

## Lecture 20: Programming on GPUs

**Lecturer:** Quanquan C. Liu

Spring 2026

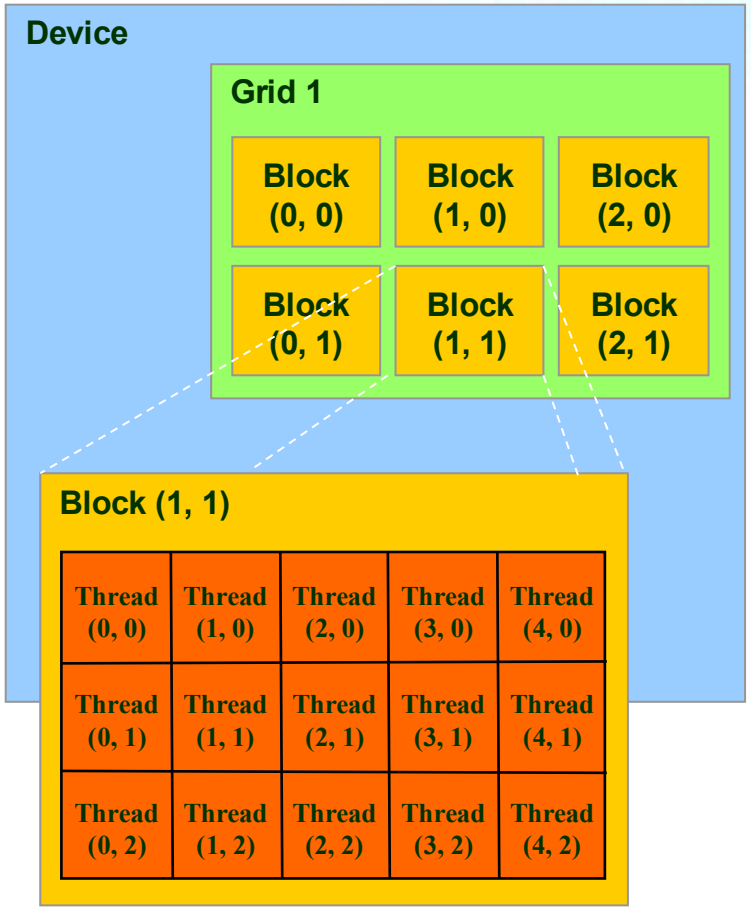
# Final Project

- If you'd like for me to comment on your final project ideas:
  - Post a private message on Ed Discussion
- Fill out the form for your final project sent via Canvas Announcement **by Friday**!
- **Additional final project guidelines posted under Files on Canvas**

# CUDA: How to Program in CUDA

# Block and Thread IDs

- Threads and blocks have IDs
  - So each thread can decide what data to work on
  - Block ID: 1D or 2D (blockIdx.x, blockIdx.y)
  - Thread ID: 1D, 2D, or 3D (threadIdx.{x,y,z})
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …




Courtesy: NDVIA

# Closer Inspection of Computation and Data Partitioning on GPU

- Define 2-d set of blocks, and 2-d set of threads per block

```
dim3 dimBlock(blocksize,blocksize);
dim3 dimGrid(N/dimBlock.x,N/dimBlock.y);
```

# Closer Inspection of Computation and Data Partitioning on GPU

- Define 2-d set of blocks, and 2-d set of threads per block

```
dim3 dimBlock(blocksize,blocksize);
dim3 dimGrid(N/dimBlock.x,N/dimBlock.y);
```

- Each thread identifies what element of the matrix it operates on

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

# Closer Inspection of Computation and Data Partitioning on GPU

- Define 2-d set of blocks, and 2-d set of threads per block

```
dim3 dimBlock(blocksize,blocksize);
dim3 dimGrid(N/dimBlock.x,N/dimBlock.y);
```

- Each thread identifies what element of the matrix it operates on

```
int i=blockIdx.x*blockDim.x+threadIdx.x;
int j=blockIdx.y*blockDim.y+threadIdx.y;
```

# Closer Inspection of Computation and Data Partitioning on GPU

- Define 2-d set of blocks, and 2-d set of threads per block

```
dim3 dimBlock(blocksize,blocksize);
dim3 dimGrid(N/dimBlock.x,N/dimBlock.y);
```

- Each thread identifies what element of the matrix it operates on

```
int i=blockIdx.x*blockDim.x+threadIdx.x;
int j=blockIdx.y*blockDim.y+threadIdx.y;
int index =i+j*N;
```

# Closer Inspection of Computation and Data Partitioning on GPU

- Define 2-d set of blocks, and 2-d set of threads per block

```
dim3 dimBlock(blocksize,blocksize);
dim3 dimGrid(N/dimBlock.x,N/dimBlock.y);
```

- Each thread identifies what element of the matrix it operates on

```
int i=blockIdx.x*blockDim.x+threadIdx.x;
int j=blockIdx.y*blockDim.y+threadIdx.y;
int index =i+j*N;
if( i <N && j <N)
  c[index]=a[index]+b[index];
```

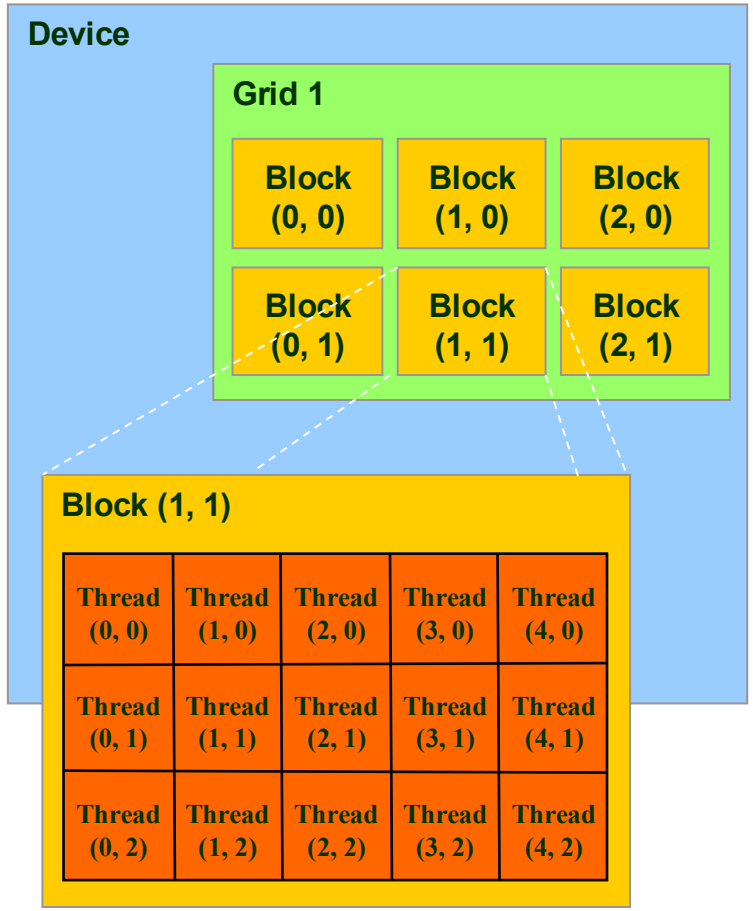
# Closer Inspection of Computation and Data Partitioning on GPU

- Define 2-d set of blocks, and 2-d set of threads per block

```
dim3 dimBlock(blocksize,blocksize);
dim3 dimGrid(N/dimBlock.x,N/dimBlock.y);
```

- Each thread identifies what element of the matrix it operates on

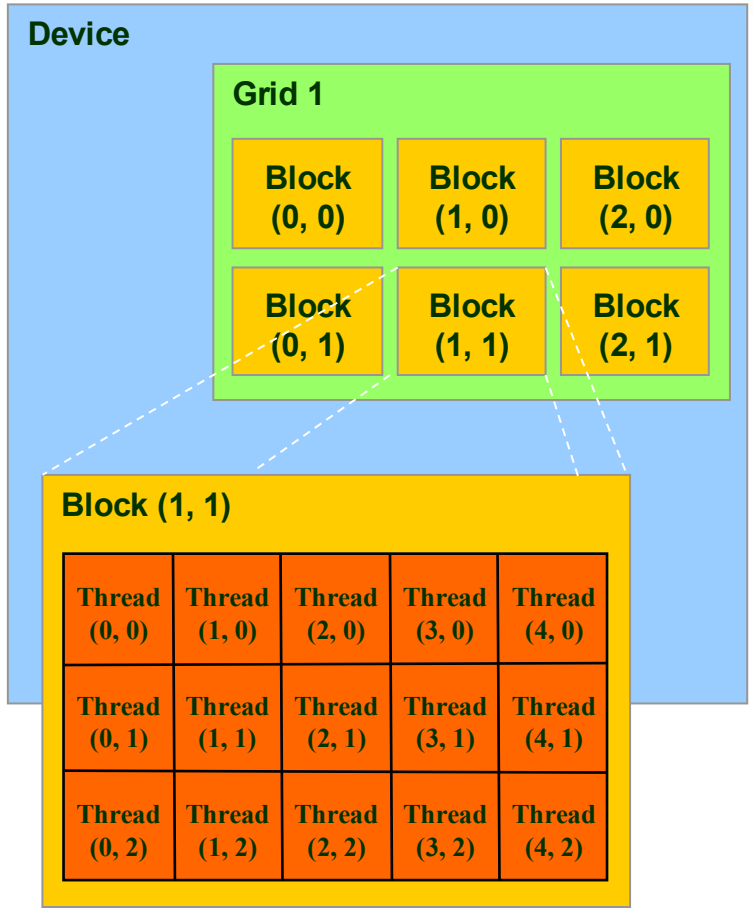
```
int i=blockIdx.x*blockDim.x+threadIdx.x;
int j=blockIdx.y*blockDim.y+threadIdx.y;
int index =i+j*N;
if( i <N && j <N)
  c[index]=a[index]+b[index];
```



# Closer Inspection of Computation and Data Partitioning on GPU

```
int i=blockIdx.x*blockDim.x+threadIdx.x;
int j=blockIdx.y*blockDim.y+threadIdx.y;
int index =i+j*N;
if( i <N && j <N)
  c[index]=a[index]+b[index];
```

**ID of Thread (2, 0)**

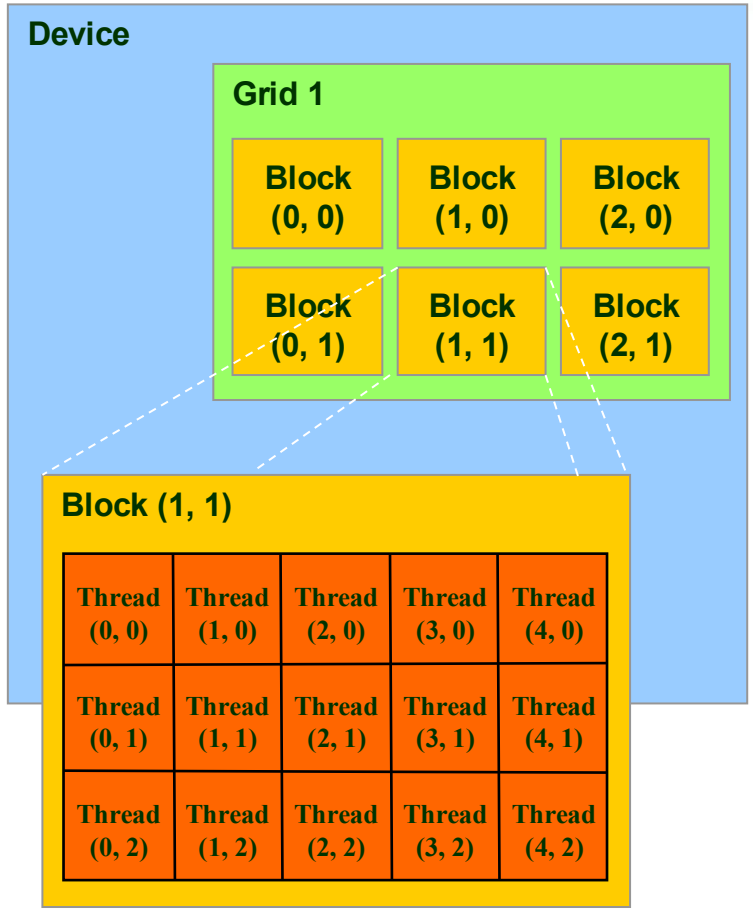


# Closer Inspection of Computation and Data Partitioning on GPU

```
int i=blockIdx.x*blockDim.x+threadIdx.x;
int j=blockIdx.y*blockDim.y+threadIdx.y;
int index =i+j*N;
if( i <N && j <N)
  c[index]=a[index]+b[index];
```

**ID of Thread (2, 0)**

**i = 4 * 5 + 2**



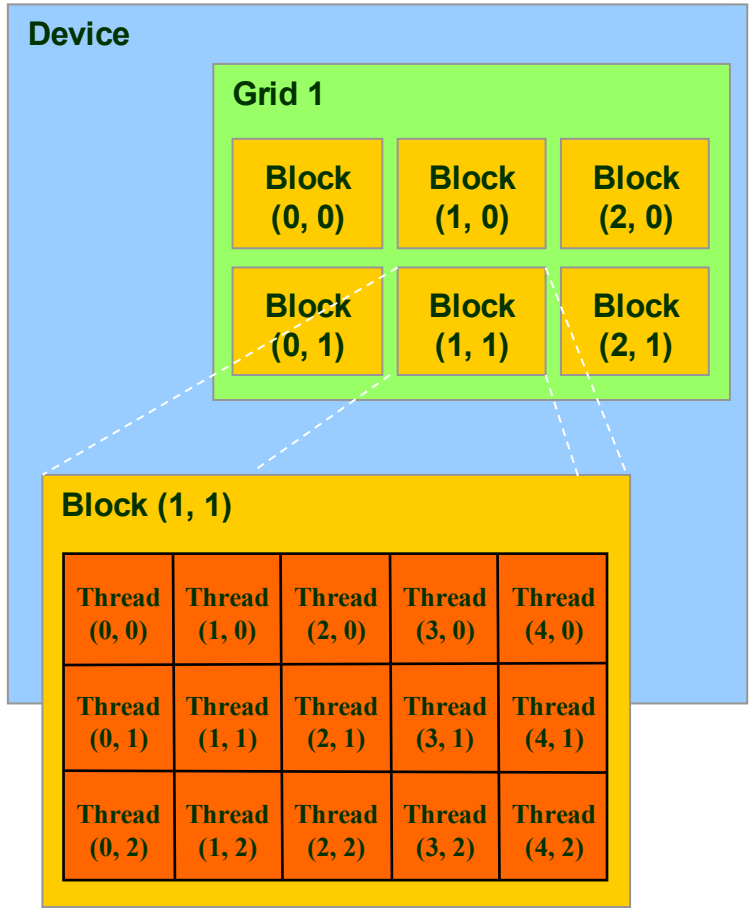
# Closer Inspection of Computation and Data Partitioning on GPU

```
int i=blockIdx.x*blockDim.x+threadIdx.x;
int j=blockIdx.y*blockDim.y+threadIdx.y;
int index =i+j*N;
if( i <N && j <N)
  c[index]=a[index]+b[index];
```

**ID of Thread (2, 0)**

**i = 4 * 5 + 2**

**j = 4 * 3 + 0**



# Closer Inspection of Computation and Data Partitioning on GPU

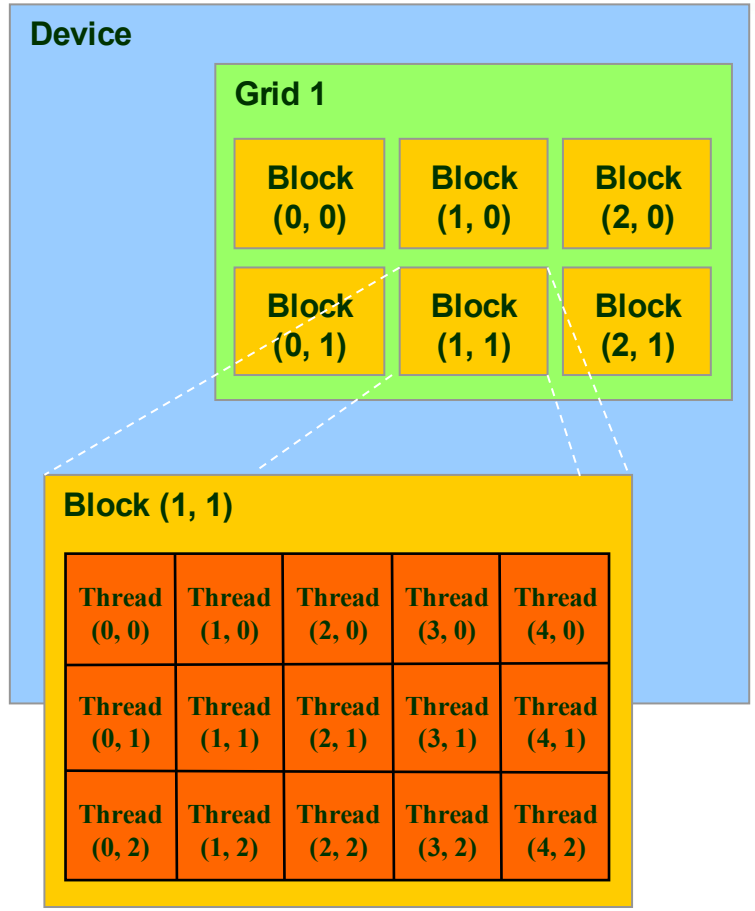
```
int i=blockIdx.x*blockDim.x+threadIdx.x;
int j=blockIdx.y*blockDim.y+threadIdx.y;
int index =i+j*N;
if( i <N && j <N)
  c[index]=a[index]+b[index];
```

**ID of Thread (2, 0)**

**i = 4 * 5 + 2**

**j = 4 * 3 + 0**

**index = 22 + 12*15 = 202**



# Reminder of How to Write CUDA Code

# CUDA Code Example

```
#include <iostream>
#include <cuda_runtime.h>

// __host__ function: Runs on the CPU
__host__ void cpuFunction() {
    std::cout << "Running on host (CPU)" << std::endl;
}

// __device__ function: Runs on the GPU; can only be called from GPU code
__device__ int deviceAdd(int a, int b) {
    return a + b;
}

// __global__ kernel: Runs on the GPU, callable from the host
__global__ void kernelFunction(int* d_result, int a, int b) {
    // Each thread can compute its own index if processing array data, for example:
    int threadId = threadIdx.x + blockIdx.x * blockDim.x;
    // For this simple example, we let one thread do the addition
    if (threadId == 0) {
        // Call the __device__ function to perform the addition
        *d_result = deviceAdd(a, b);
    }
}
```

# CUDA Code Example

```
#include <iostream>
#include <cuda_runtime.h>

// __host__ function: Runs on the CPU
__host__ void cpuFunction() {
    std::cout << "Running on host (CPU)" << std::endl;
}

// __device__ function: Runs on the GPU; can only be called from GPU code
__device__ int deviceAdd(int a, int b) {
    return a + b;
}

// __global__ kernel: Runs on the GPU, callable from the host
__global__ void kernelFunction(int* d_result, int a, int b) {
    // Each thread can compute its own index if processing array data, for example:
    int threadId = threadIdx.x + blockIdx.x * blockDim.x;
    // For this simple example, we let one thread do the addition
    if (threadId == 0) {
        // Call the __device__ function to perform the addition
        *d_result = deviceAdd(a, b);
```

# CUDA Code Example

```
#include <iostream>
#include <cuda_runtime.h>

// __host__ function: Runs on the CPU
__host__ void cpuFunction() {
    std::cout << "Running on host (CPU)" << std::endl;
}

// __device__ function: Runs on the GPU; can only be called from GPU code
__device__ int deviceAdd(int a, int b) {
    return a + b;
}

// __global__ kernel: Runs on the GPU, callable from the host
__global__ void kernelFunction(int* d_result, int a, int b) {
    int threadId = threadIdx.x + blockIdx.x * blockDim.x;
    if (threadId == 0) {
        // Call the __device__ function to perform the addition
        *d_result = deviceAdd(a, b);
    }
}
```

```
int main() {
    // Print a message from the CPU.
    cpuFunction();

    // Host-side variable to receive the result.
    int h_result;

    // Declare a pointer for device memory.
    int* d_result;

    // Allocate memory on the GPU (device)
    cudaMalloc(&d_result, sizeof(int));

    // Launch the kernel:
    // Here we use 1 block with 1 thread for simplicity.
    kernelFunction<<<1, 1>>>(d_result, 3, 4);

    // Wait for the GPU to finish its work (ensure kernel execution is complete).
    cudaDeviceSynchronize();

    // Copy the result from device memory to host memory.
    cudaMemcpy(&h_result, d_result, sizeof(int),
cudaMemcpyDeviceToHost);

    // Free the allocated device memory.
    cudaFree(d_result);

    // Display the result.
    std::cout << "Result (3 + 4): " << h_result << std::endl;

    return 0;
}
```

```
int main() {
    // Print a message from the CPU.
    cpuFunction();

    // Host-side variable to receive the result.
    int h_result;

    // Declare a pointer for device memory.
    int* d_result;

    // Allocate memory on the GPU (device)
    cudaMalloc(&d_result, sizeof(int));

    // Launch the kernel:
    // Here we use 1 block with 1 thread for
    kernelFunction<<<1, 1>>>(d_result, 3, 4);

    // Wait for the GPU to finish its work (ensure kernel execution is complete).
    cudaDeviceSynchronize();

    // Copy the result from device memory to host memory.
    cudaMemcpy(&h_result, d_result, sizeof(int),
cudaMemcpyDeviceToHost);

    // Free the allocated device memory.
    cudaFree(d_result);

    // Display the result.
    std::cout << "Result (3 + 4): " << h_result << std::endl;

    return 0;
}
```

# Working Example: Write Your Own CUDA function

# Simple working code example

- What does it do?
  - Scan elements of array of numbers (any of 0 to 9)

# Simple working code example

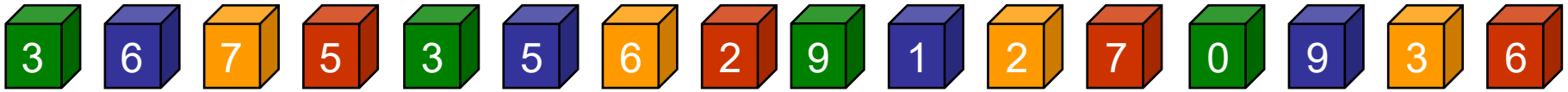
- What does it do?
  - Scan elements of array of numbers (any of 0 to 9)
  - How many times does “6” appear?

# Simple working code example

- What does it do?
  - Scan elements of array of numbers (any of 0 to 9)
  - How many times does “6” appear?
  - Array of 16 elements, each thread examines 4 elements, 1 block in grid, 1 grid

# Simple working code example

- What does it do?
  - Scan elements of array of numbers (any of 0 to 9)
  - How many times does “6” appear?
  - Array of 16 elements, each thread examines 4 elements, 16 total threads




threadIdx.x = 0 examines in_array elements 0, 4, 8, 12
threadIdx.x = 1 examines in_array elements 1, 5, 9, 13
threadIdx.x = 2 examines in_array elements 2, 6, 10, 14
threadIdx.x = 3 examines in_array elements 3, 7, 11, 15

} Known as a cyclic data distribution

# CUDA Pseudo-Code

## MAIN PROGRAM:

Initialization

- Allocate memory on host for input and output
- Assign random numbers to input array

Call ***host*** function

Calculate final output from per-thread output

Print result

# CUDA Pseudo-Code

## MAIN PROGRAM:

Initialization

- Allocate memory on host for input and output
- Assign random numbers to input array

Call ***host*** function

Calculate final output from per-thread output

Print result

## HOST FUNCTION:

Allocate memory on device for copy of ***input*** and ***output***

Copy input to ***device***

Set up grid/block

Call ***global*** function

Copy ***device*** output to host

# CUDA Pseudo-Code

## MAIN PROGRAM:

Initialization

- Allocate memory on host for input and output
- Assign random numbers to input array

Call ***host*** function

Calculate final output from per-thread output

Print result

## GLOBAL FUNCTION:

Thread scans subset of array elements

Call ***device*** function to compare with “6”

Compute local result

## HOST FUNCTION:

Allocate memory on device for copy of ***input*** and ***output***

Copy input to ***device***

Set up grid/block

Call ***global*** function

Copy ***device*** output to host

# CUDA Pseudo-Code

## MAIN PROGRAM:

Initialization

- Allocate memory on host for input and output
- Assign random numbers to input array

Call ***host*** function

Calculate final output from per-thread output

Print result

## HOST FUNCTION:

Allocate memory on device for copy of ***input*** and ***output***

Copy input to ***device***

Set up grid/block

Call ***global*** function

Copy ***device*** output to host

## GLOBAL FUNCTION:

Thread scans subset of array elements

Call ***device*** function to compare with “6”

Compute local result

## DEVICE FUNCTION:

Compare current element and “6”

Return 1 if same, else 0

# Main Program: Preliminaries

**MAIN PROGRAM:**

Initialization

- Allocate memory on host for input and output
- Assign random numbers to input array

Call ***host*** function

Calculate final output from per-thread output

Print result

```
#include <stdio.h>
#define SIZE 16
#define BLOCKSIZE 4

int main(int argc, char **argv)
{
  int *in_array, *out_array;
  …
}
```

# Main Program: Invoke Global Function

**MAIN PROGRAM:**

Initialization (OMIT)

- Allocate memory on host for input and output
- Assign random numbers to input array

Call ***host*** function

Calculate final output from per-thread output

Print result

```
#include <stdio.h>
#define SIZE 16
#define BLOCKSIZE 4
__host__ void outer_compute
        (int *in_arr, int *out_arr);
int main(int argc, char **argv)
{
  int *in_array, *out_array;
  /* initialization */ …
  outer_compute(in_array,
   out_array);
   …
}
```

# Main Program

**MAIN PROGRAM:**

Initialization (OMIT)

- Allocate memory on host for input and output
- Assign random numbers to input array

Call ***host*** function

**Calculate final output from per-thread output**

**Print result**

```
#include <stdio.h>
#define SIZE 16
#define BLOCKSIZE 4
__host__ void outer_compute     (int *in_arr,
  int *out_arr);
int main(int argc, char **argv)
{
  int *in_array, *out_array;
  int sum = 0;
  /* initialization */ …
  outer_compute(in_array, out_array);
  for (int i=0; i<BLOCKSIZE; i++) {
    sum+=out_array[i];
  }
  printf (”Result = %d\n",sum);
}
```

# Host Function: Preliminaries & Allocation

**HOST FUNCTION:**

Allocate memory on device for copy of ***input*** and ***output***

Copy input to ***device***

Set up grid/block

Call ***global*** function

Copy ***device*** output to host

```
__host__ void outer_compute (int
  *h_in_array, int *h_out_array) {

  int *d_in_array, *d_out_array;

  cudaMalloc((void **) &d_in_array,
        SIZE*sizeof(int));
  cudaMalloc((void **) &d_out_array,
        BLOCKSIZE*sizeof(int));

  …
}
```

# Host Function: Copy Data To/From Host

**HOST FUNCTION:**

Allocate memory on device for copy of ***input*** and ***output***

Copy input to ***device***

Set up grid/block

Call ***global*** function

Copy ***device*** output to host

```
__host__ void outer_compute (int
*h_in_array, int *h_out_array) {
int *d_in_array, *d_out_array;

cudaMalloc((void **) &d_in_array,
      SIZE*sizeof(int));

cudaMalloc((void **) &d_out_array,
      BLOCKSIZE*sizeof(int));

cudaMemcpy(d_in_array, h_in_array,
      SIZE*sizeof(int),
      cudaMemcpyHostToDevice);

… do computation ...

cudaMemcpy(h_out_array,d_out_array,
      BLOCKSIZE*sizeof(int),
      cudaMemcpyDeviceToHost);

}
```

# Host Function: Setup & Call Global Function

**HOST FUNCTION:**

Allocate memory on device for copy of ***input*** and ***output***

Copy input to ***device***

Set up grid/block

Call ***global*** function

Copy ***device*** output to host

```
__host__ void outer_compute (int
*h_in_array, int *h_out_array) {
int *d_in_array, *d_out_array;


cudaMalloc((void **) &d_in_array,
      SIZE*sizeof(int));

cudaMalloc((void **) &d_out_array,
      BLOCKSIZE*sizeof(int));
cudaMemcpy(d_in_array, h_in_array,
      SIZE*sizeof(int),
      cudaMemcpyHostToDevice);
msize = (SIZE+BLOCKSIZE) *
      sizeof (int);
compute<<<1,BLOCKSIZE,msize)>>>
      (d_in_array, d_out_array);
cudaMemcpy(h_out_array, d_out_array,
      BLOCKSIZE*sizeof(int),
      cudaMemcpyDeviceToHost);

}
```

# Global Function

**GLOBAL FUNCTION:**

Thread scans subset of array elements

Call ***device*** function to compare with “6”

Compute local result

```
__global__ void compute(int *d_in,int *d_out)
 {
 d_out[threadIdx.x] = 0;
 for (int i=0; i<SIZE/BLOCKSIZE;        i++)
  {
     int val = d_in[i*BLOCKSIZE +
 threadIdx.x];
    d_out[threadIdx.x] += compare(val, 6);
  }
}
```

# Device Function

**DEVICE FUNCTION:**

Compare current element and “6”

Return 1 if same, else 0

```
__device__ int
  compare(int a, int b)
  {
  if (a == b) return 1;
  return 0;
}
```

# Reductions

- This type of computation is called a ***parallel reduction***
  - Operation is applied to large data structure

# Reductions

- This type of computation is called a ***parallel reduction***
  - Operation is applied to large data structure
  - Computed result represents the aggregate solution across the large data structure

# Reductions

- This type of computation is called a ***parallel reduction***
  - Operation is applied to large data structure
  - Computed result represents the aggregate solution across the large data structure
  - Large data structure → computed result (perhaps single number) **[dimensionality reduced]**

# Reductions

- This type of computation is called a ***parallel reduction***
  - Operation is applied to large data structure
  - Computed result represents the aggregate solution across the large data structure
  - Large data structure → computed result (perhaps single number) **[dimensionality reduced]**
- Why might parallel reductions be well-suited to GPUs?

# Reductions

- This type of computation is called a ***parallel reduction***
  - Operation is applied to large data structure
  - Computed result represents the aggregate solution across the large data structure
  - Large data structure → computed result (perhaps single number) **[dimensionality reduced]**
- Why might parallel reductions be well-suited to GPUs?
- What if we tried to compute the final sum on the GPUs?

# Standard Parallel Construct

- Sometimes called “embarassingly parallel” or “pleasingly parallel”

# Standard Parallel Construct

- Sometimes called “embarassingly parallel” or “pleasingly parallel”
- Each thread is completely independent of the others

# Standard Parallel Construct

- Sometimes called “embarassingly parallel” or “pleasingly parallel”
- Each thread is completely independent of the others
- Final result copied to CPU

# Standard Parallel Construct

- Sometimes called “embarassingly parallel” or “pleasingly parallel”
- Each thread is completely independent of the others
- Final result copied to CPU
- Another example, adding two matrices:
  - A more careful examination of decomposing computation into grids and thread blocks

# Best Practices for Writing CUDA Programs

# Approach to Working with New Systems

- Approach with **caution**

# Approach to Working with New Systems

- Approach with **caution**
  - May not behave as expected.

# Approach to Working with New Systems

- Approach with **caution**
  - May not behave as expected.
  - Learn about behavior through observation and measurement.

# Approach to Working with New Systems

- Approach with **caution**
  - May not behave as expected.
  - Learn about behavior through observation and measurement.
  - Realize that **documentation is not always clear or accurate**

# Approach to Working with New Systems

- Approach with **caution**
  - May not behave as expected.
  - Learn about behavior through observation and measurement.
  - Realize that **documentation is not always clear or accurate**
- Crawl, then walk, then run

# Approach to Working with New Systems

- Approach with **caution**
  - May not behave as expected.
  - Learn about behavior through observation and measurement.
  - Realize that **documentation is not always clear or accurate**
- Crawl, then walk, then run
  - Start with something known to work.

# Approach to Working with New Systems

- Approach with **caution**
  - May not behave as expected.
  - Learn about behavior through observation and measurement.
  - Realize that **documentation is not always clear or accurate**
- Crawl, then walk, then run
  - Start with something known to work.
  - Only change one “variable” at a time.

# Debugging Using Device Emulation Mode

- An executable compiled in device emulation mode (`nvcc -deviceemu`) runs completely on the host using the CUDA runtime

# Debugging Using Device Emulation Mode

- An executable compiled in device emulation mode (`nvcc -deviceemu`) runs completely on the host using the CUDA runtime
  - No need for any device or CUDA driver

# Debugging Using Device Emulation Mode

- An executable compiled in device emulation mode (`nvcc -deviceemu`) runs completely on the host using the CUDA runtime
  - No need for any device or CUDA driver
  - Each device thread is emulated with a host thread

# Debugging Using Device Emulation Mode

- An executable compiled in device emulation mode (`nvcc -deviceemu`) runs completely on the host using the CUDA runtime
  - No need for any device or CUDA driver
  - Each device thread is emulated with a host thread
- When running in device emulation mode, one can:
  - Use host native debug support (`gdb` breakpoints, inspection, etc.)

# Debugging Using Device Emulation Mode

- An executable compiled in device emulation mode (`nvcc -deviceemu`) runs completely on the host using the CUDA runtime
  - No need for any device or CUDA driver
  - Each device thread is emulated with a host thread
- When running in device emulation mode, one can:
  - Use host native debug support (`gdb` breakpoints, inspection, etc.)
  - Access any device-specific data from host code and vice-versa

# Debugging Using Device Emulation Mode

- An executable compiled in device emulation mode (`nvcc -deviceemu`) runs completely on the host using the CUDA runtime
  - No need for any device or CUDA driver
  - Each device thread is emulated with a host thread
- When running in device emulation mode, one can:
  - Use host native debug support (`gdb` breakpoints, inspection, etc.)
  - Access any device-specific data from host code and vice-versa
  - Call any host function from device code (e.g. `printf`) and vice-versa

# Debugging Using Device Emulation Mode

- An executable compiled in device emulation mode (`nvcc -deviceemu`) runs completely on the host using the CUDA runtime
  - No need for any device or CUDA driver
  - Each device thread is emulated with a host thread
- When running in device emulation mode, one can:
  - Use host native debug support (`gdb` breakpoints, inspection, etc.)
  - Access any device-specific data from host code and vice-versa
  - Call any host function from device code (e.g. `printf`) and vice-versa
  - Detect deadlock situations caused by improper usage of `__syncthreads`

# Device Emulation Mode Pitfalls

- Emulated device threads execute sequentially, so **simultaneous accesses of the same memory location by multiple threads** could produce different results.

# Device Emulation Mode Pitfalls

- Emulated device threads execute sequentially, so **simultaneous accesses of the same memory location by multiple threads** could produce different results.
- **Dereferencing** device **pointers** on the host or host pointers on the device can produce correct results in device emulation mode, but will generate an error in device execution mode

# Device Emulation Mode Pitfalls

- Emulated device threads execute sequentially, so **simultaneous accesses of the same memory location by multiple threads** could produce different results.
- **Dereferencing** device **pointers** on the host or host pointers on the device can produce correct results in device emulation mode, but will generate an error in device execution mode
- **Results of floating-point computations** will slightly differ because of:

# Device Emulation Mode Pitfalls

- Emulated device threads execute sequentially, so **simultaneous accesses of the same memory location by multiple threads** could produce different results.
- **Dereferencing** device **pointers** on the host or host pointers on the device can produce correct results in device emulation mode, but will generate an error in device execution mode
- **Results of floating-point computations** will slightly differ because of:
  - Different compiler outputs, instruction sets

# Device Emulation Mode Pitfalls

- Emulated device threads execute sequentially, so **simultaneous accesses of the same memory location by multiple threads** could produce different results.
- **Dereferencing** device **pointers** on the host or host pointers on the device can produce correct results in device emulation mode, but will generate an error in device execution mode
- **Results of floating-point computations** will slightly differ because of:
  - Different compiler outputs, instruction sets
  - Use of extended precision for intermediate results

# Device Emulation Mode Pitfalls

- Emulated device threads execute sequentially, so **simultaneous accesses of the same memory location by multiple threads** could produce different results.
- **Dereferencing** device **pointers** on the host or host pointers on the device can produce correct results in device emulation mode, but will generate an error in device execution mode
- **Results of floating-point computations** will slightly differ because of:
  - Different compiler outputs, instruction sets
  - Use of extended precision for intermediate results
    - There are various options to force strict single precision on the host

# Run-time functions and macros

In CUDA run-time services,

cudaGetDeviceProperties(deviceProp &dp, d);

check number, type and whether device present

# Run-time functions and macros

In CUDA run-time services,

cudaGetDeviceProperties(deviceProp &dp, d);

check number, type and whether device present

In libcutil.a of Software Developers' Kit,

cutComparef (float *ref, float *data, unsigned len);

compare output with reference from CPU implementation

# Run-time functions and macros

In CUDA run-time services,

cudaGetDeviceProperties(deviceProp &dp, d);

check number, type and whether device present

In libcutil.a of Software Developers' Kit,

cutComparef (float *ref, float *data, unsigned len);

compare output with reference from CPU implementation

In cutil.h of Software Developers' Kit (with #define _DEBUG or –D_DEBUG compile flag),

CUDA_SAFE_CALL(f(<args>)), CUT_SAFE_CALL(f(<args>))

check for error in run-time call and exit if error detected

# Run-time functions and macros

In CUDA run-time services,

cudaGetDeviceProperties(deviceProp &dp, d);

check number, type and whether device present

In libcutil.a of Software Developers' Kit,

cutComparef (float *ref, float *data, unsigned len);

compare output with reference from CPU implementation

In cutil.h of Software Developers' Kit (with #define _DEBUG or –D_DEBUG compile flag),

CUDA_SAFE_CALL(f(<args>)), CUT_SAFE_CALL(f(<args>))

check for error in run-time call and exit if error detected

CUT_SAFE_MALLOC(cudaMalloc(<args>));

similar to above, but for malloc calls

# Run-time functions and macros

In CUDA run-time services,

cudaGetDeviceProperties(deviceProp &dp, d);

check number, type and whether device present

In libcutil.a of Software Developers' Kit,

cutComparef (float *ref, float *data, unsigned len);

compare output with reference from CPU implementation

In cutil.h of Software Developers' Kit (with #define _DEBUG or –D_DEBUG compile flag),

CUDA_SAFE_CALL(f(<args>)), CUT_SAFE_CALL(f(<args>))

check for error in run-time call and exit if error detected

CUT_SAFE_MALLOC(cudaMalloc(<args>));

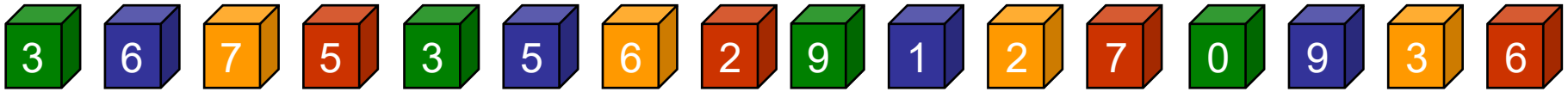
similar to above, but for malloc calls

CUT_CHECK_ERROR("error message goes here");

check for error immediately following kernel execution and if detected, exit with error message

# Simple working code example

- What does it do?
  - Scan elements of array of numbers (any of 0 to 9)
  - How many times does “6” appear?
  - Array of 16 elements, each thread examines 4 elements, 1 block in grid, 1 grid




threadIdx.x = 0 examines in_array elements 0, 4, 8, 12
threadIdx.x = 1 examines in_array elements 1, 5, 9, 13
threadIdx.x = 2 examines in_array elements 2, 6, 10, 14
threadIdx.x = 3 examines in_array elements 3, 7, 11, 15

} Known as a cyclic data distribution

# CUDA Pseudo-Code

## MAIN PROGRAM:

Initialization

- Allocate memory on host for input and output
- Assign random numbers to input array

Call ***host*** function

Calculate final output from per-thread output

Print result

## HOST FUNCTION:

Allocate memory on device for copy of ***input*** and ***output***

Copy input to ***device***

Set up grid/block

Call ***global*** function

Copy ***device*** output to host

## GLOBAL FUNCTION:

Thread scans subset of array elements

Call ***device*** function to compare with “6”

Compute local result

## DEVICE FUNCTION:

Compare current element and “6”

Return 1 if same, else 0

# Reminder: Gathering and Reporting Results

- Global, device functions and excerpts from host, main

```
__device__ int compare(int a, int b) {
    if (a == b) return 1;
    return 0;
}

__global__
  *d_out) {

  d_out[thre

  for (i=0; i<

     int val = d_in[i*BLOCKSIZE +
           threadIdx.x];

     d_out[threadIdx.x] +=
                  compare(val, 6);
   }
}
```

**Compute individual results for each thread**

**Serialize final results gathering on host**

```
int __host__ void outer_compute
   (int *h_in_array, int *h_out_array) {
   …
   compute<<<1,BLOCKSIZE,msize)>>>
     (d_in_array, d_out_array);

 cudaMemcpy(h_out_array,
d_out_array,
   BLOCKSIZE*sizeof(int),
   cudaMemcpyDeviceToHost);

main(int argc, char **argv) {
…
  for (int i=0; i<BLOCKSIZE; i++)
  {  sum+=out_array[i]; }
  printf (”Result = %d\n",sum);
}
```

# Gathering Results on GPU: Synchronization

```
void __syncthreads();
```

- Functionality: Synchronizes all threads in a block

# Gathering Results on GPU: Synchronization

```
void __syncthreads();
```

- Functionality: Synchronizes all threads in a block
  - Each thread waits at the point of this call until all other threads have reached it

# Gathering Results on GPU: Synchronization

```
void __syncthreads();
```

- Functionality: Synchronizes all threads in a block
  - Each thread waits at the point of this call until all other threads have reached it
  - Once all threads have reached this point, execution resumes normally

# Gathering Results on GPU: Synchronization

```
void __syncthreads();
```

- Functionality: Synchronizes all threads in a block
  - Each thread waits at the point of this call until all other threads have reached it
  - Once all threads have reached this point, execution resumes normally
- Why is this needed?
  - A thread can freely read the shared memory of its thread block or the global memory of either its block or grid.

# Gathering Results on GPU: Synchronization

```
void __syncthreads();
```

- Functionality: Synchronizes all threads in a block
  - Each thread waits at the point of this call until all other threads have reached it
  - Once all threads have reached this point, execution resumes normally
- Why is this needed?
  - A thread can freely read the shared memory of its thread block or the global memory of either its block or grid.
  - Allows the program to guarantee partial ordering of these accesses to prevent incorrect orderings.

# Gathering Results on GPU: Synchronization

```
void __syncthreads();
```

- Functionality: Synchronizes all threads in a block
  - Each thread waits at the point of this call until all other threads have reached it
  - Once all threads have reached this point, execution resumes normally
- Why is this needed?
  - A thread can freely read the shared memory of its thread block or the global memory of either its block or grid.
  - Allows the program to guarantee partial ordering of these accesses to prevent incorrect orderings.
- Watch out!
  - Potential for deadlock when it appears in conditionals

# Gathering Results on GPU for “Count 6”

```
__global__ void compute(int *d_in, int
   *d_out) {

  d_out[threadIdx.x] = 0;

  for (i=0; i<SIZE/BLOCKSIZE; i++) {

      int val = d_in[i*BLOCKSIZE +
              threadIdx.x];

      d_out[threadIdx.x] +=
                    compare(val, 6);

   }
}
```

# Gathering Results on GPU for “Count 6”

```
__global__ void compute(int *d_in, int
   *d_out) {

  d_out[threadIdx.x] = 0;

  for (i=0; i<SIZE/BLOCKSIZE; i++) {

      int val = d_in[i*BLOCKSIZE +
              threadIdx.x];

      d_out[threadIdx.x] +=
                    compare(val, 6);

   }
}
```

```
__global__ void compute(int *d_in, int
   *d_out, int *d_sum) {

  d_out[threadIdx.x] = 0;

  for (i=0; i<SIZE/BLOCKSIZE; i++) {

      int val = d_in[i*BLOCKSIZE +
              threadIdx.x];

      d_out[threadIdx.x] +=
                    compare(val, 6);

   }
   __synthreads();
   if (threadIdx.x == 0) {
     for 0..BKSIZE-1 {
     *d_sum += d_out[i];
   }
}
```

# Another Example: Adding Matrices

# Another Example: Adding Two Matrices

**CPU C program**

```
void add_matrix_cpu(float *a, float *b,
   float *c, int N)
{
int i, j, index;
for (i=0;i<N;i++) {
            for (j=0;j<N;j++) {
      index =i+j*N;
      c[index]=a[index]+b[index];
    }
  }
}

void main() {
   .....
   add_matrix(a,b,c,N);
}
```

# Another Example: Adding Two Matrices

**CPU C program**

```
void add_matrix_cpu(float *a, float *b,
   float *c, int N)
{
int i, j, index;
for (i=0;i<N;i++) {
            for (j=0;j<N;j++) {
      index =i+j*N;
      c[index]=a[index]+b[index];
    }
  }
}

void main() {
   .....
   add_matrix(a,b,c,N);
}
```

**CUDA C program**

```
__global__ void add_matrix_gpu(float *a, float *b, float *c, intN)
{
   int i =blockIdx.x*blockDim.x+threadIdx.x;
   int j=blockIdx.y*blockDim.y+threadIdx.y;
   int index =i+j*N;
   if( i <N && j <N)
     c[index]=a[index]+b[index];
}

void main() {
   dim3 dimBlock(blocksize,blocksize);
   dim3 dimGrid(N/dimBlock.x,N/dimBlock.y);
   add_matrix_gpu<<<dimGrid,dimBlock>>>(a,b,c,N);
}
```

# Closer Inspection of Computation and Data Partitioning

- Define 2-d set of blocks, and 2-d set of threads per block

```
dim3 dimBlock(blocksize,blocksize);
dim3 dimGrid(N/dimBlock.x,N/dimBlock.y);
```

- Each thread identifies what single element of the matrix it operates on

```
int i=blockIdx.x*blockDim.x+threadIdx.x;
int j=blockIdx.y*blockDim.y+threadIdx.y;
int index =i+j*N;
if( i <N && j <N)
  c[index]=a[index]+b[index];
```

# Closer Inspection of Computation and Data Partitioning

- Define 2-d set of blocks, and 2-d set of threads per block

```
dim3 dimBlock(blocksize,blocksize);
dim3 dimGrid(N/dimBlock.x,N/dimBlock.y);
```

- Each thread identifies what single element of the matrix it operates on

```
int i=blockIdx.x*blockDim.x+threadIdx.x;
int j=blockIdx.y*blockDim.y+threadIdx.y;
int index =i+j*N;
if( i <N && j <N)
  c[index]=a[index]+b[index];
```

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 |

- Let blocksize = 2 and N = 4

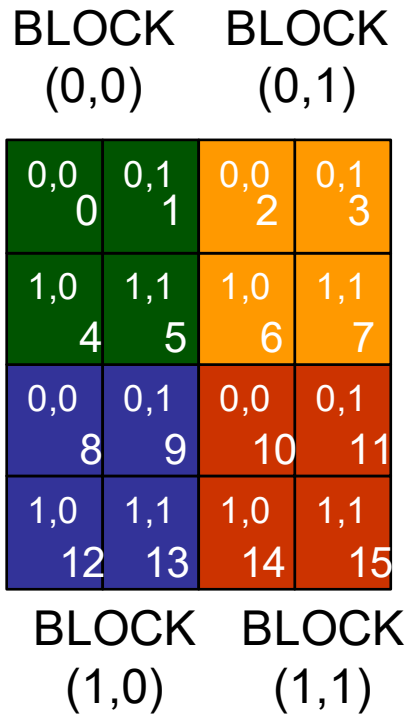
# Closer Inspection of Computation and Data Partitioning

- Define 2-d set of blocks, and 2-d set of threads per block

```
dim3 dimBlock(blocksize,blocksize);
dim3 dimGrid(N/dimBlock.x,N/dimBlock.y);
```

- Each thread identifies what single element of the matrix it operates on

```
int i=blockIdx.x*blockDim.x+threadIdx.x;
int j=blockIdx.y*blockDim.y+threadIdx.y;
int index =i+j*N;
if( i <N && j <N)
  c[index]=a[index]+b[index];
```




- Let blocksize = 2 and N = 4

# Data Distribution to Threads

- Many data parallel programming languages have mechanisms for expressing **how a distributed data structure is mapped to threads**

# Data Distribution to Threads

- Many data parallel programming languages have mechanisms for expressing **how a distributed data structure is mapped to threads**
  - That is, the **portion of the data a thread accesses** (and usually stores locally)

# Data Distribution to Threads

- Many data parallel programming languages have mechanisms for expressing **how a distributed data structure is mapped to threads**
  - That is, the **portion of the data a thread accesses** (and usually stores locally)
- Reasons for selecting a particular distribution

# Data Distribution to Threads

- Many data parallel programming languages have mechanisms for expressing **how a distributed data structure is mapped to threads**
  - That is, the **portion of the data a thread accesses** (and usually stores locally)
- Reasons for selecting a particular distribution
  - Sharing patterns: group together according to “data reuse” and communication needs

# Data Distribution to Threads

- Many data parallel programming languages have mechanisms for expressing **how a distributed data structure is mapped to threads**
  - That is, the **portion of the data a thread accesses** (and usually stores locally)
- Reasons for selecting a particular distribution
  - Sharing patterns: group together according to “data reuse” and communication needs
  - Affinity with other data

# Data Distribution to Threads

- Many data parallel programming languages have mechanisms for expressing **how a distributed data structure is mapped to threads**
  - That is, the **portion of the data a thread accesses** (and usually stores locally)
- Reasons for selecting a particular distribution
  - Sharing patterns: group together according to “data reuse” and communication needs
  - Affinity with other data
  - Locality: transfer size and capacity limit on shared memory

# Data Distribution to Threads

- Many data parallel programming languages have mechanisms for expressing **how a distributed data structure is mapped to threads**
  - That is, the **portion of the data a thread accesses** (and usually stores locally)
- Reasons for selecting a particular distribution
  - Sharing patterns: group together according to “data reuse” and communication needs
  - Affinity with other data
  - Locality: transfer size and capacity limit on shared memory
  - Access patterns relative to computation (avoid bank conflicts)

# Data Distribution to Threads

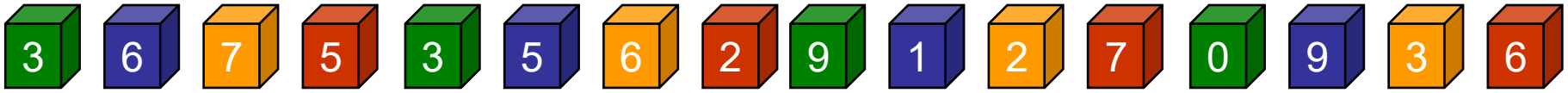
- Many data parallel programming languages have mechanisms for expressing **how a distributed data structure is mapped to threads**
  - That is, the **portion of the data a thread accesses** (and usually stores locally)
- Reasons for selecting a particular distribution
  - Sharing patterns: group together according to “data reuse” and communication needs
  - Affinity with other data
  - Locality: transfer size and capacity limit on shared memory
  - Access patterns relative to computation
  - Minimizing overhead: Cost of copying to host and between global and shared memory

# Common Data Distributions

- Consider a 1-Dimensional array

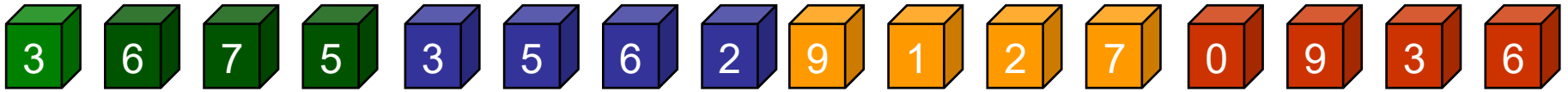
CYCLIC:

```
for (i = 0; i<BLOCKSIZE; i++)
        … d_in [i*BLOCKSIZE + threadIdx.x];
```




BLOCK:

```
for (i=threadIdx.x*BLOCKSIZE; i<(threadIdx.x+1)
    *BLOCKSIZE; i++)
    … d_in[i];
```



# Which one is better?

- **Block distribution** is simpler and often more friendly to CUDA's memory system

# Which one is better?

- **Block distribution** is simpler and often more friendly to CUDA's memory system
  - Naturally aligns well with local reuse patterns.

# Which one is better?

- **Block distribution** is simpler and often more friendly to CUDA's memory system
  - Naturally aligns well with local reuse patterns.
- However, **cyclic distribution** sometimes brings these benefits:

# Which one is better?

- **Block distribution** is simpler and often more friendly to CUDA's memory system
  - Naturally aligns well with local reuse patterns.
- However, **cyclic distribution** sometimes brings these benefits:
  - **Better load balancing**

# Which one is better?

- **Block distribution** is simpler and often more friendly to CUDA's memory system
  - Naturally aligns well with local reuse patterns.
- However, **cyclic distribution** sometimes brings these benefits:
  - **Better load balancing**
  - **Avoiding hot spots**
    - Contiguous chunks of data might cause too many memory conflicts or high load on particular parts of memory

# Which one is better?

- **Block distribution** is simpler and often more friendly to CUDA's memory system
  - Naturally aligns well with local reuse patterns.
- However, **cyclic distribution** sometimes brings these benefits:
  - **Better load balancing**
  - **Avoiding hot spots**
    - Contiguous chunks of data might cause too many memory conflicts or high load on particular parts of memory
  - **Periodic access patterns**
    - For workloads where data is processed in fixed strides, a cyclic distribution lines up more naturally with strides