CUDA Tutorial

ECSE 420: Fall 2019

By: Mohammad Mushfiqur Rahman

Outline

- TA Information
- Parallel Programming
- What is CUDA?
- CUDA Concepts
  - Heterogenous Computing
  - Blocks
  - Threads
  - Indexing
  - Cooperating Threads
  - Managing the Device
- References

TA Information

1. Name: Mohammad Mushfiqur Rahman (**Mushfique**)
   Email: mohammad.rahman4@mcgill.ca

2. Name: **Ankit** Malhotra
   Email: ankit.malhotra@mail.mcgill.ca

- **TA Office Hours = Lab Sessions**
- For emailing, please use **"ECSE 420"** in the subject for faster response!

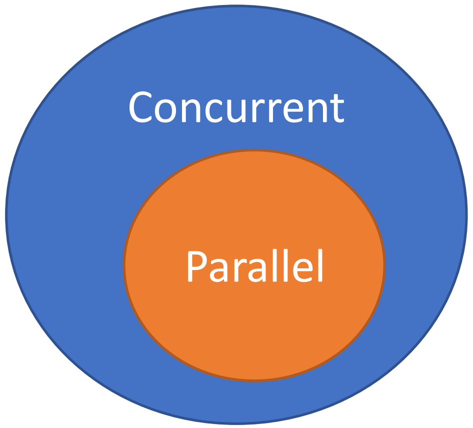
Paralleling Programming

- What is **parallel** programming?

  - Programs are executed ***simultaneously*** on separate hardware, independent of each other.

- What is **concurrent** programming and how is that different?

  - Programs ***seem*** to run simultaneously on the same/separate hardware.
  Ex – parallel programs, task switching, etc.



- Amdahl's Law:

$$\boldsymbol{Speedup} = \frac{\mathbf{1}}{\frac{\boldsymbol{P}}{\boldsymbol{n}} + (\mathbf{1} - \boldsymbol{P})}$$

  - P = parallelizable portion (0 to 1)
  - n = processing elements (ex – CPU cores)

What is CUDA?

- CUDA Architecture
  - Expose GPU parallelism for general-purpose computing
  - Retain performance

- CUDA C/C++
  - Based on industry-standard C/C++
  - Small set of extensions to enable heterogeneous programming
  - Straightforward APIs to manage devices, memory etc.

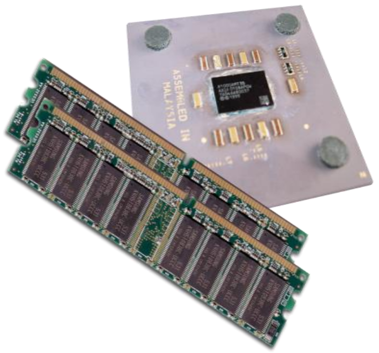
- This session introduces CUDA C/C++

Prerequisites

- You (probably) need experience with C or C++, or other programming languages such as python, Matlab, etc.
- You don’t need GPU experience
- You don’t need parallel programming experience
- You don’t need graphics experience

CUDA Concept: HETEROGENOUS COMPUTING

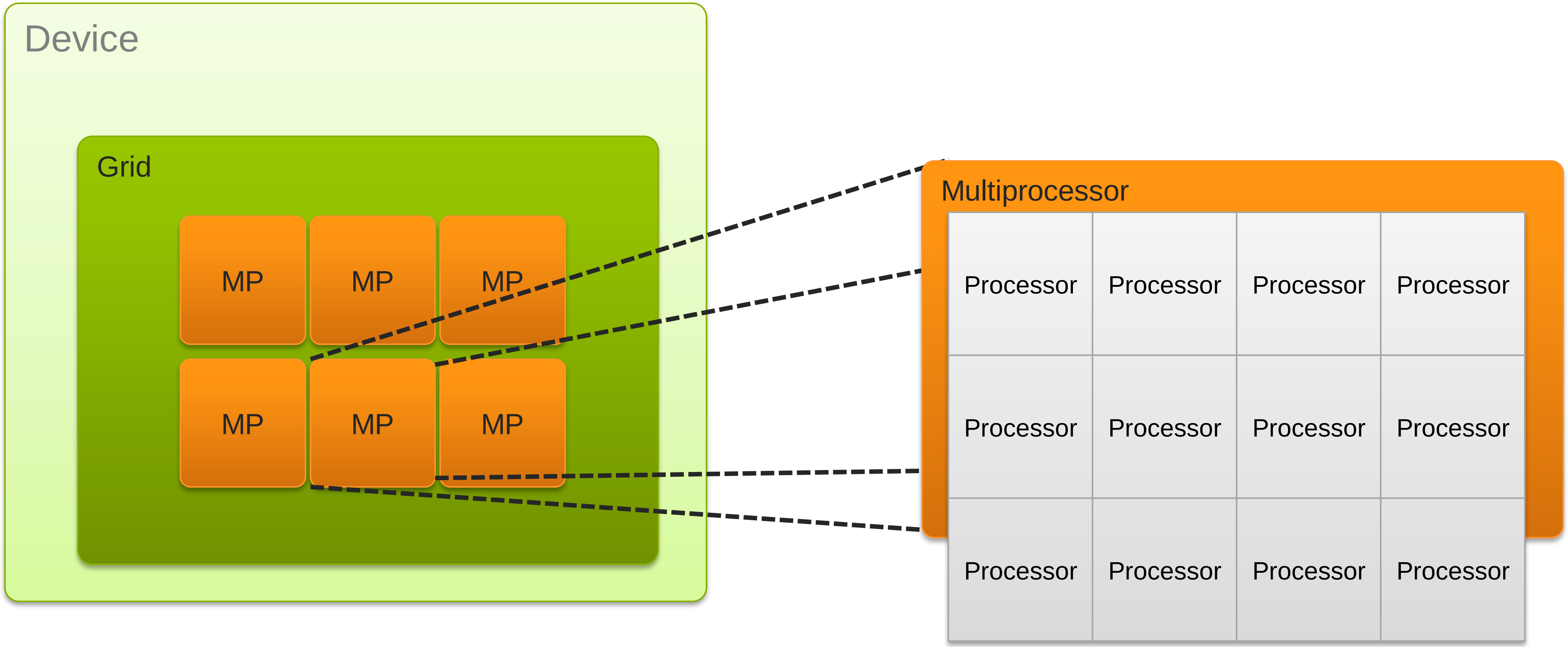
- Terminology:
  - *Host* The CPU and its memory (host memory)
  - *Device* The GPU and its memory (device memory)



Host

Device

GPU Architecture



Heterogeneous Computing

Device code

Host code

```
#include <iostream>
#include <algorithm>

using namespace std;

#define N          1024
#define RADIUS     3
#define BLOCK_SIZE 16

__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;
}

void fill_ints(int *x, int n) {
        fill_n(x, n, 1);
}

int main(void) {
        int *in, *out;          // host copies of a, b, c
        int *d_in, *d_out;      // device copies of a, b, c
        int size = (N + 2*RADIUS) * sizeof(int);

        // Alloc space for host copies and setup values
        in  = (int *)malloc(size); fill_ints(in,  N + 2*RADIUS);
        out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

        // Alloc space for device copies
        cudaMalloc((void **)&d_in,  size);
        cudaMalloc((void **)&d_out, size);

        // Copy to device
        cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

        // Launch stencil_1d() kernel on GPU
        stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS,
d_out + RADIUS);

        // Copy result back to host
        cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

        // Cleanup
        free(in); free(out);
        cudaFree(d_in); cudaFree(d_out);
        return 0;
}
```

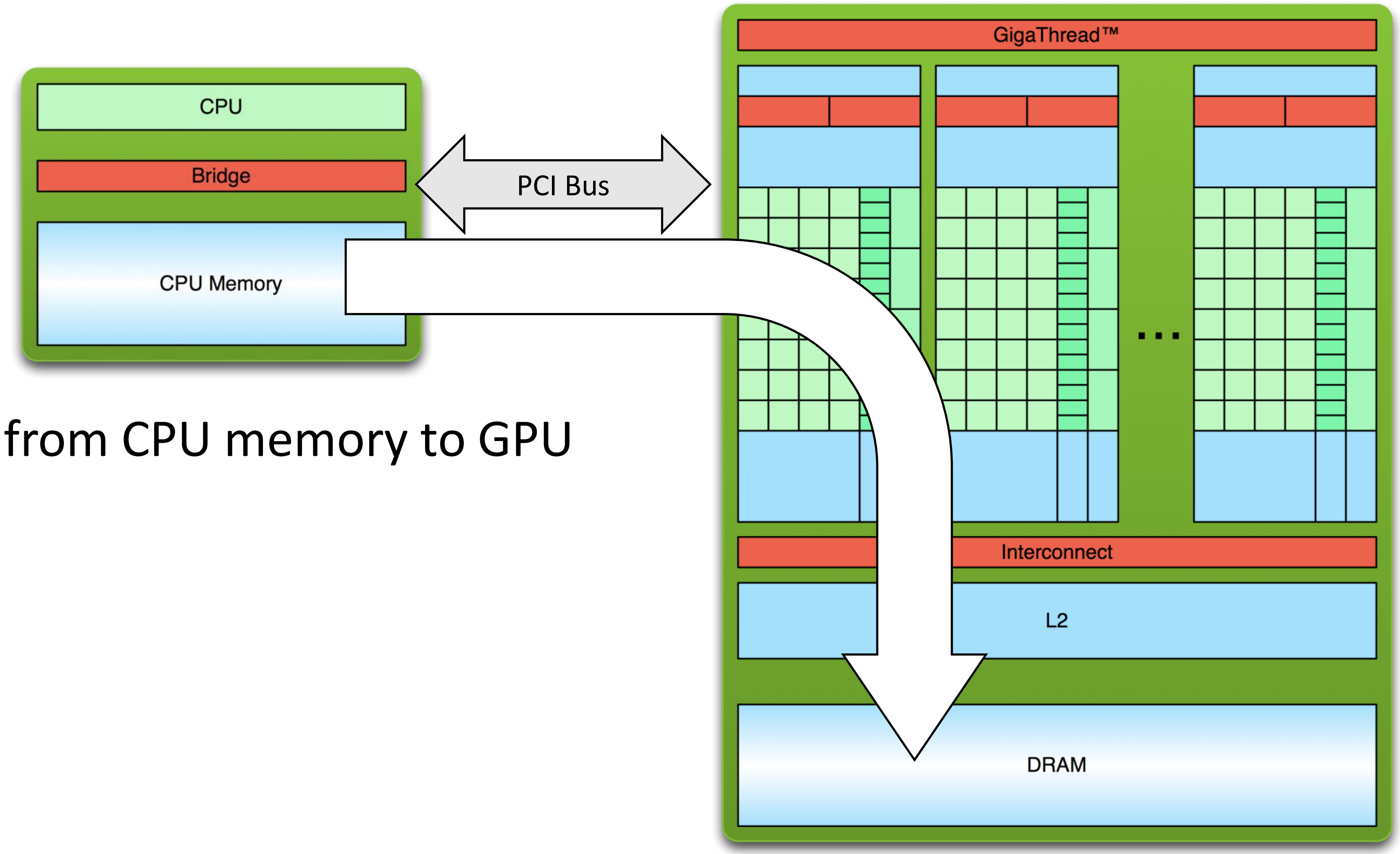
parallel function

serial code

parallel code

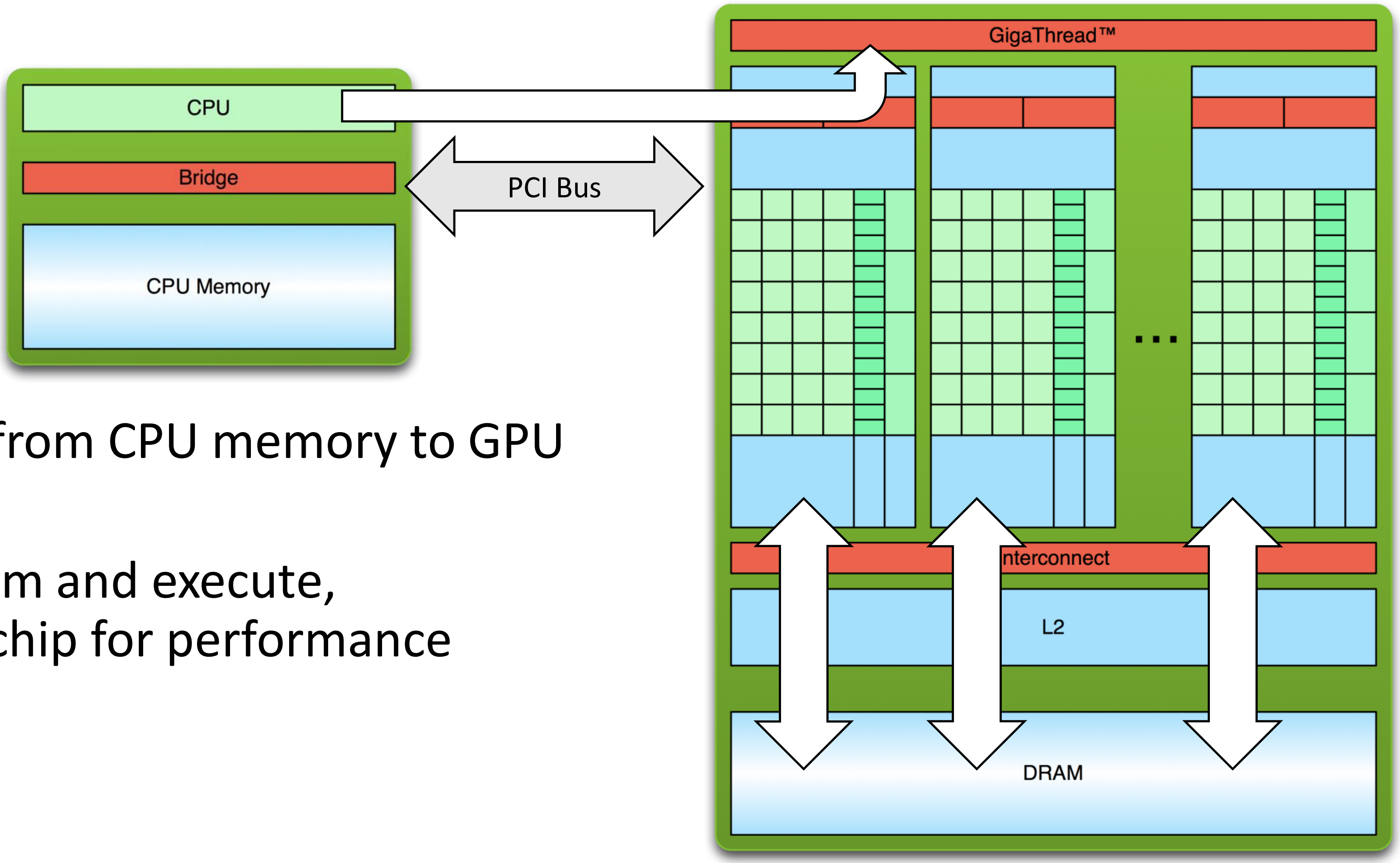
serial code

Simple Processing Flow



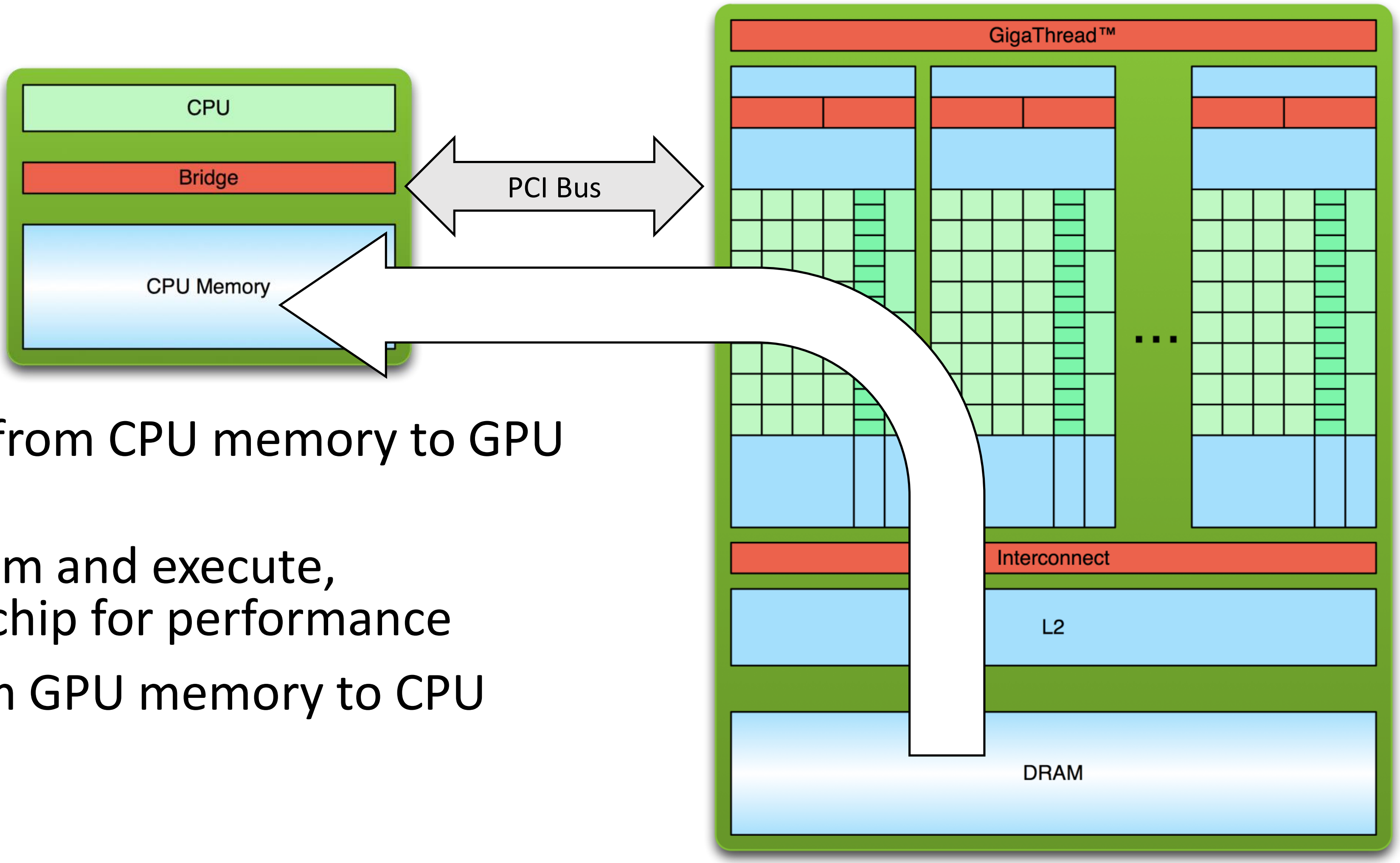
1. Copy input data from CPU memory to GPU memory

Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

Simple Processing Flow

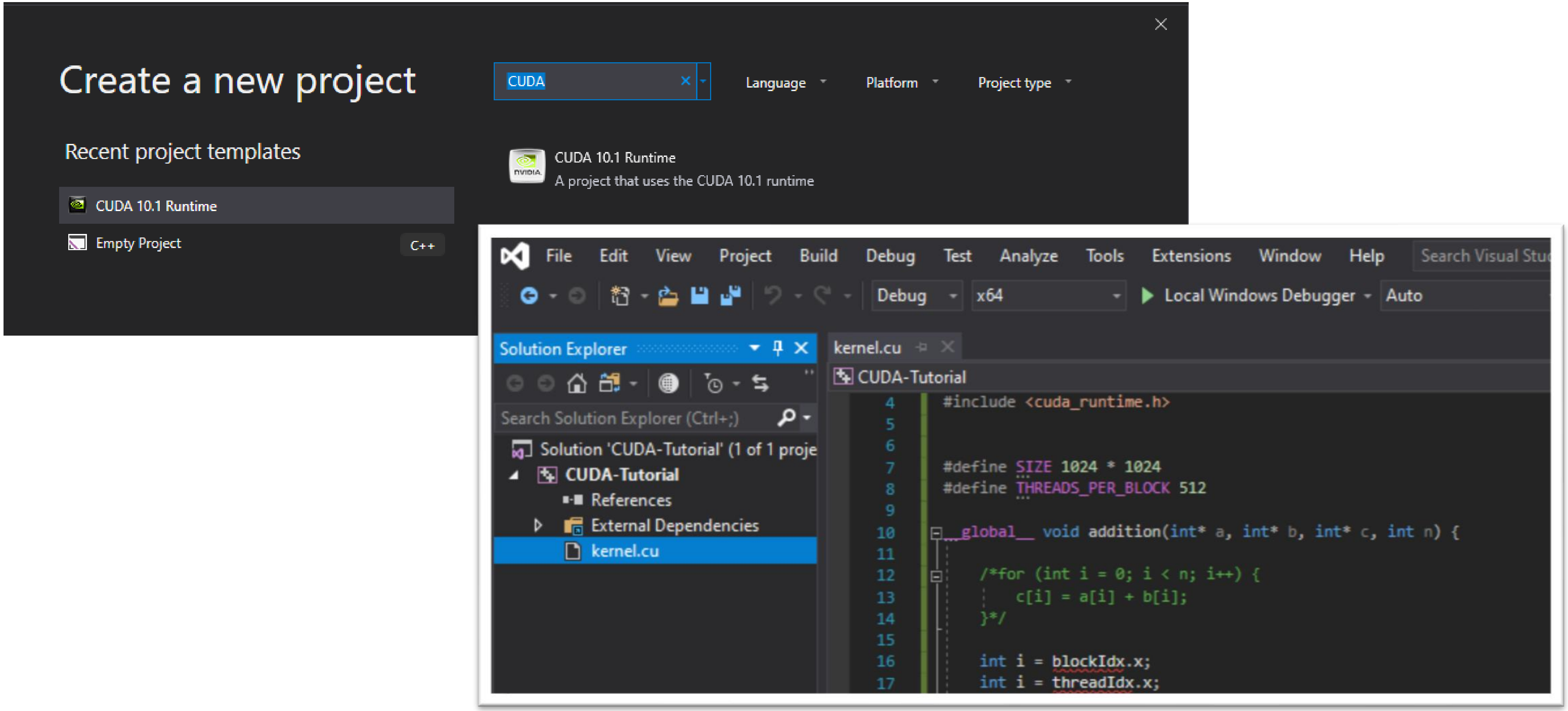


1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

IDE, Toolkit and GPUs in the Labs

- IDE: **Microsoft Visual Studio 2019** (Community Version), **MATLAB** 2019b
- Toolkit: **CUDA 10.1**
- GPU: **NVIDIA GTX 1050 Ti**

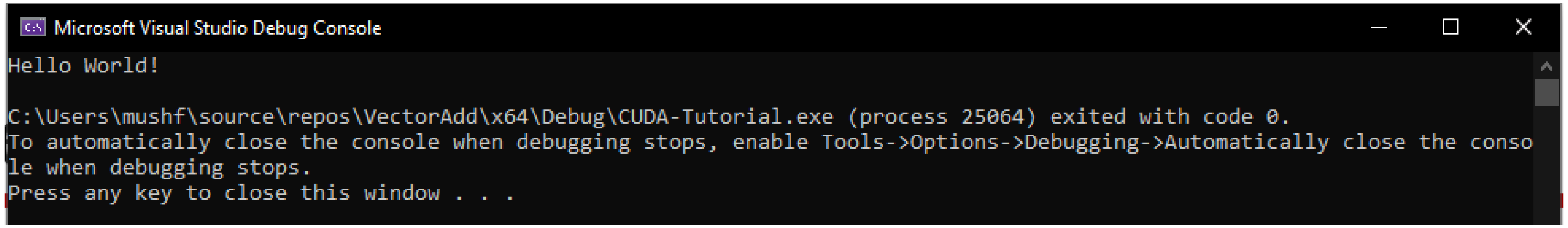
Creating Your First CUDA Project



Example 1: Hello World!

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
        printf("Hello World!\n");
        return 0;
}
```

- Standard C running on Host
- NVIDIA compiler (nvcc) can be used to compile programs with no device code
- **Output:**



Example 1: Hello World! (With Device Code)

```
#include <stdio.h>
#include <stdlib.h>
// CUDA runtime
#include <cuda_runtime.h>        🡐=================== Handles all the CUDA Syntax
__global__ void myKernel(void) {
        printf("Hello World!\n");
}
int main(void) {
        myKernel <<<1, 1 >>> ();
        return 0;
}
```

Output:

```
Microsoft Visual Studio Debug Console
Hello World!

C:\Users\mushf\source\repos\VectorAdd\x64\Debug\CUDA-Tutorial.exe (process 25064) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

Example 1: Hello World! (With Device Code)

```
__global__ void myKernel(void) {
```

- CUDA C/C++ keyword `__global__` indicates a function that:
  - Runs on the device
  - Is called from host code
- **Kernels** – Functions that run on device (GPU) and are called from host (CPU). Ex – myKernel
- `nvcc` separates source code into host and device components
  - Device functions (e.g. **`mykernel()`**) processed by NVIDIA compiler
  - Host functions (e.g. **`main()`**) processed by standard host compiler
    - **`gcc`, `cl.exe`**

Example 1: Hello World! (With Device Code)

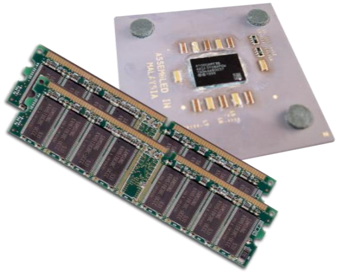
```
myKernel <<<1, 1 >>> ();
```

- Triple angle brackets mark a call from *host* code to *device* code
  - Also called a “kernel launch”
  - We’ll return to the parameters (1,1) in a moment

- That’s all that is required to execute a function on the GPU!

Memory Management

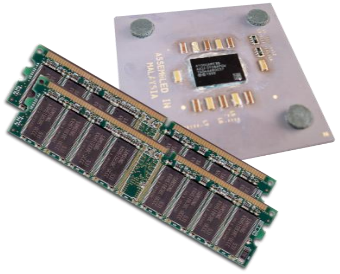
- Host and device memory are separate entities
  - *Device* pointers point to GPU memory
    - May be passed to/from host code
    - May *not* be dereferenced in host code
  - *Host* pointers point to CPU memory
    - May be passed to/from device code
    - May *not* be dereferenced in device code



- Simple CUDA API for handling device memory
  - **`cudaMalloc(),cudaFree(),cudaMemcpy()`**
  - Similar to the C equivalents: **`malloc(),free(),memcpy()`**

Steps to remember

1. Allocate *host* memory and initialized host data
2. Allocate *device* memory
3. Transfer input data from *host* to *device* memory
4. Execute kernels
5. Transfer output from *device* memory to *host* memory



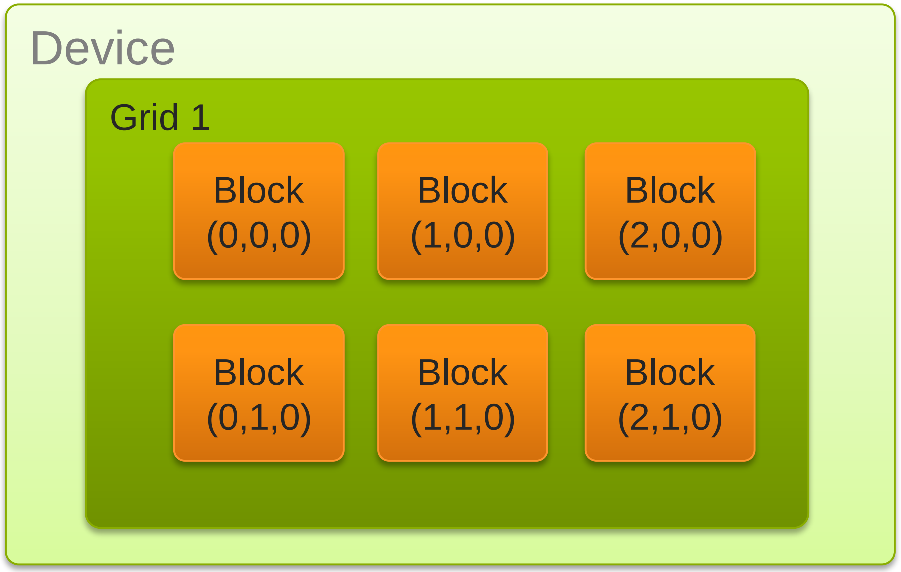
Example 2: Integer Addition

```
__global__ void add(int *a, int *b, int *c) {
        *c = *a + *b;
}
int main(void) {
        int a, b, c;                // host copies of a, b, c
        int *d_a, *d_b, *d_c;       // device copies of a, b, c
        int size = sizeof(int);

        // Allocate space for device copies of a, b, c
        cudaMalloc((void **)&d_a, size);
        cudaMalloc((void **)&d_b, size);
        cudaMalloc((void **)&d_c, size);

        // Setup input values
        a = 2;
        b = 7;
```

Example 2: Integer Addition

```
// Copy inputs to device
        cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

        // Launch add() kernel on GPU
        add<<<1,1>>>(d_a, d_b, d_c);

        // Copy result back to host
        cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

        // Cleanup
        cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
        return 0;
}
```

CUDA Concept:
BLOCKS



Moving to Parallel

- GPU computing is about massive parallelism
  - So how do we run code in parallel on the device?

```
addition<<< 1, 1 >>>();

addition<<< N, 1 >>>();
```

- Instead of executing `addition()` once, execute N times in parallel

Example 3: Vector Addition using Blocks

- With `add()` running in parallel we can do vector addition

- Terminology: each parallel invocation of `addition()` is referred to as a block
  - The set of blocks is referred to as a grid
  - Each invocation can refer to its block index using `blockIdx.x`

```
__global__ void addition(int *a, int *b, int *c) {
        c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- By using `blockIdx.x` to index into the array, each block handles a different index

Example 3: Vector Addition using Blocks

- With `add()` running in parallel we can do vector addition

- Terminology: each parallel invocation of `add()` is referred to as a block
  - The set of blocks is referred to as a grid
  - Each invocation can refer to its block index using `blockIdx.x`

```
__global__ void add(int *a, int *b, int *c) {
        c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- By using `blockIdx.x` to index into the array, each block handles a different index

Block 0	Block 1	Block 2	Block 3
`c[0]  = a[0] + b[0];`	`c[1]  = a[1] + b[1];`	`c[2]  = a[2] + b[2];`	`c[3]  = a[3] + b[3];`

Example 3: Vector Addition using Blocks

```
#include <stdio.h>
#include <stdlib.h>
// CUDA runtime
#include <cuda_runtime.h>
#define SIZE 1024

__global__ void addition(int* a, int* b, int* c, int n) {

        /*for (int i = 0; i < n; i++) {  ←=================Conventional way of Vector Addition with single thread
        c[i] = a[i] + b[i];
        }*/

        int i = blockIdx.x;    ←================Using GPU Blocks for Vector Addition
        if (i < n) {
                c[i] = a[i] + b[i];
        }
}
```

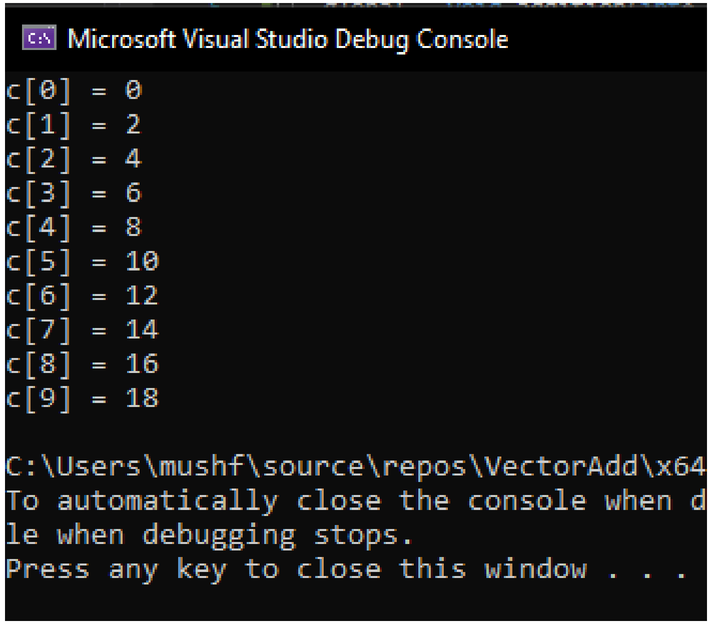
Example 3: Vector Addition using Blocks

```
// Printing an Array
void printArray(int* a, int n) {
        for (int i = 0; i < n; i++) {
                printf("c[%d] = %d\n", i, a[i]);
        }
}

int main(void) {
        int* a, * b, * c;

        // Allocate space in Unified Memory for a,b,c
        cudaMallocManaged((void**)&a, SIZE * sizeof(int)); ←===Different than cudaMalloc()
        cudaMallocManaged((void**)&b, SIZE * sizeof(int));
        cudaMallocManaged((void**)&c, SIZE * sizeof(int));
```

Example 3: Vector Addition using Blocks

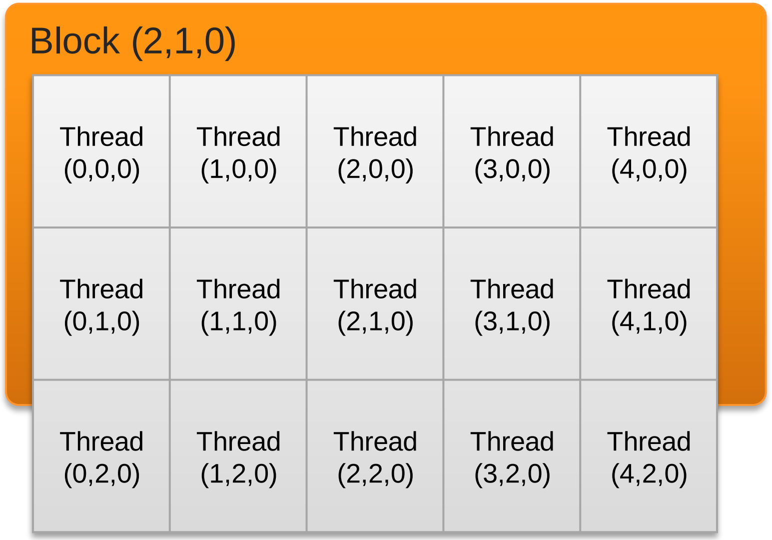
```
    // Initializing the values
    for (int i = 0; i < SIZE; i++) {
            a[i] = i;
            b[i] = i;
            c[i] = 0;
    }
    // Launch addition() kernel on GPU with N blocks
    addition<<< SIZE, 1 >>>(a, b, c, SIZE);

    // Wait for GPU threads to complete
    cudaDeviceSynchronize(); ←====New Function to sync
                                 (more on that later..)

    // Printing the Output Array
    printArray(c, 10);

    // Cleanup
    cudaFree(a);cudaFree(b); cudaFree(c);

    return 0;
}
```

Output:


Microsoft Visual Studio Debug Console

```
c[0] = 0
c[1] = 2
c[2] = 4
c[3] = 6
c[4] = 8
c[5] = 10
c[6] = 12
c[7] = 14
c[8] = 16
c[9] = 18

C:\Users\mushf\source\repos\VectorAdd\x64
To automatically close the console when d
le when debugging stops.
Press any key to close this window . . .
```

CUDA Concept: THREADS



CUDA Threads

- Terminology: a block can be split into parallel threads
- Let's change `addition()` to use parallel *threads* instead of parallel *blocks*

```
__global__ void addition(int* a, int* b, int* c, int n) {
        int i = threadIdx.x;

        if (i < n) {
                c[i] = a[i] + b[i];
        }
}
```

- We use **`threadIdx.x`** instead of **`blockIdx.x`**
- Need to make one change in **`main()`**…

Example 4: Vector Addition using Threads

```
#include <stdio.h>
#include <stdlib.h>
// CUDA runtime
#include <cuda_runtime.h>
#define SIZE 1024

__global__ void addition(int* a, int* b, int* c, int n) {

        /*for (int i = 0; i < n; i++) {  🡐=================Conventional way of Vector Addition with single thread
        c[i] = a[i] + b[i];
        }*/

        int i = threadIdx.x;    🡐=================Using GPU Threads for Vector Addition
        if (i < n) {
                c[i] = a[i] + b[i];
        }
}
```

Example 4: Vector Addition using Threads

```
// Printing an Array
void printArray(int* a, int n) {
        for (int i = 0; i < n; i++) {
                printf("c[%d] = %d\n", i, a[i]);
        }
}

int main(void) {
        int* a, * b, * c;

        // Allocate space in Unified Memory for a,b,c
        cudaMallocManaged((void**)&a, SIZE * sizeof(int)); 🡐===Different than cudaMalloc()
        cudaMallocManaged((void**)&b, SIZE * sizeof(int));
        cudaMallocManaged((void**)&c, SIZE * sizeof(int));
```

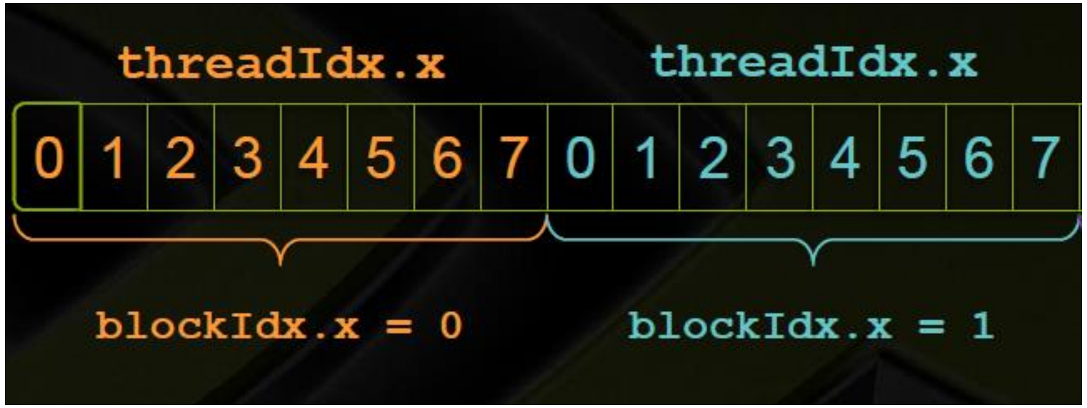
Example 4: Vector Addition using Threads

```
        // Initializing the values
        for (int i = 0; i < SIZE; i++) {
                a[i] = i;
                b[i] = i;
                c[i] = 0;
        }
        // Launch addition() kernel on GPU with N blocks
        addition<<< 1, SIZE >>>(a, b, c, SIZE);

        // Wait for GPU threads to complete
        cudaDeviceSynchronize(); <====New Function to sync
                                     (more on that later..)

        // Printing the Output Array
        printArray(c, 10);

        // Cleanup
        cudaFree(a);cudaFree(b); cudaFree(c);

        return 0;
}
```

Output:

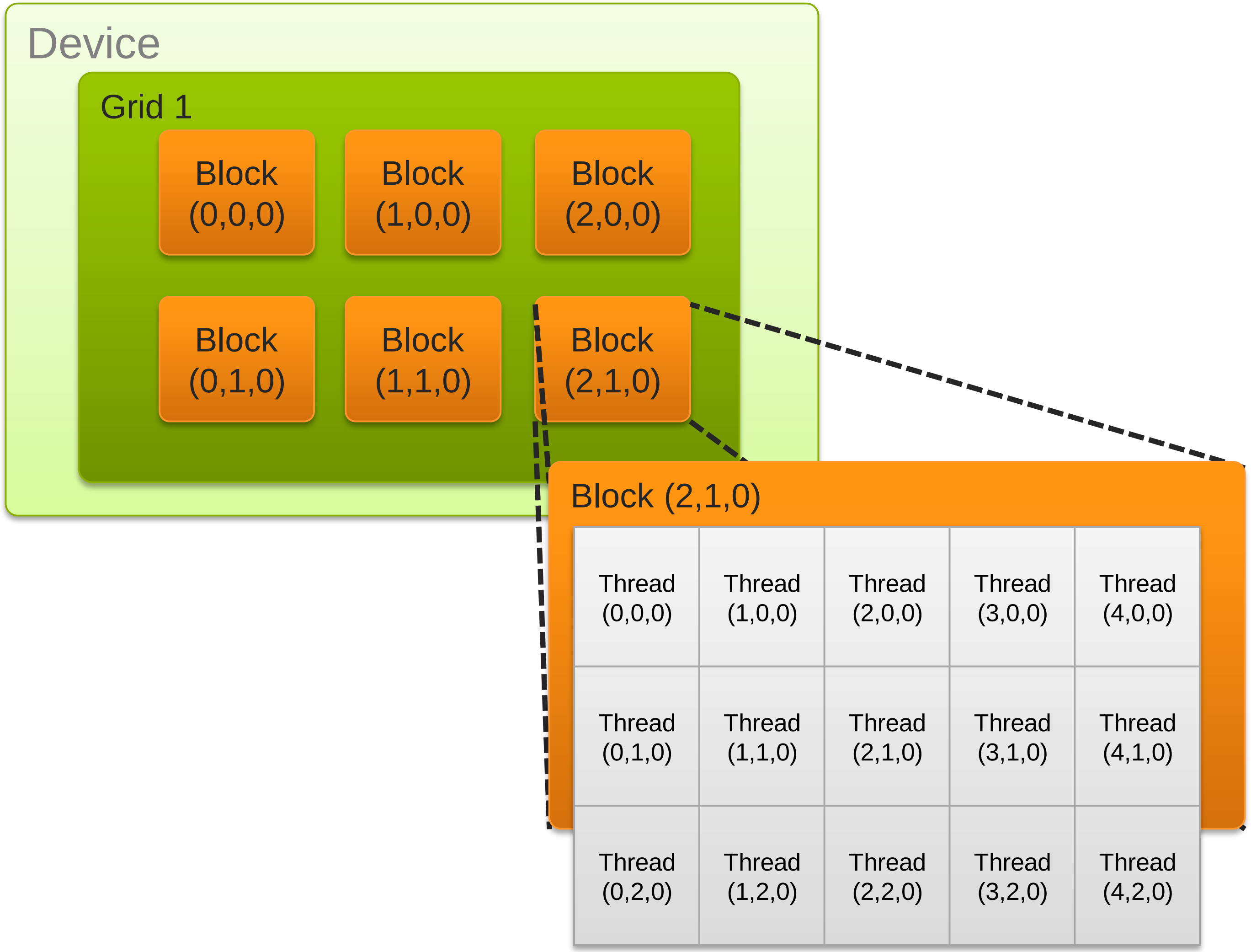
```
Microsoft Visual Studio Debug Console
c[0] = 0
c[1] = 2
c[2] = 4
c[3] = 6
c[4] = 8
c[5] = 10
c[6] = 12
c[7] = 14
c[8] = 16
c[9] = 18

C:\Users\mushf\source\repos\VectorAdd\x64
To automatically close the console when d
le when debugging stops.
Press any key to close this window . . .
```

CUDA Concept: INDEXING



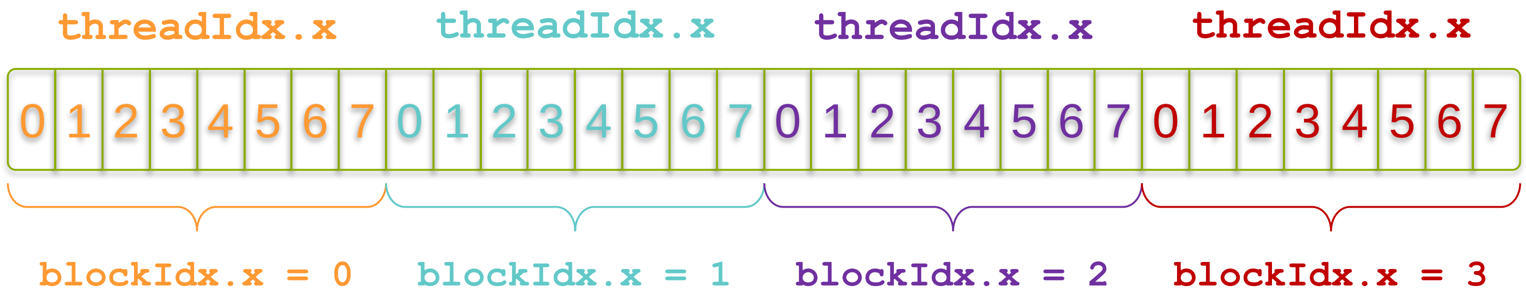
Indices and Dimensions

- A kernel is launched as a grid of blocks of threads
  - `blockIdx` and `threadIdx` are 3D
  - **We will only show one dimension (x),** the same logic applies for the rest (y, z)
- Built-in variables:
  - `threadIdx`
  - `blockIdx`
  - `blockDim`
  - `gridDim`



Indexing Arrays with Blocks and Threads

- We've seen parallel vector addition using:
  - Many **blocks** with *one thread* each
  - *One block* with many **threads**
- No longer as simple as using `blockIdx.x` and `threadIdx.x`
  - Consider indexing an array with one element per thread (8 threads/block)

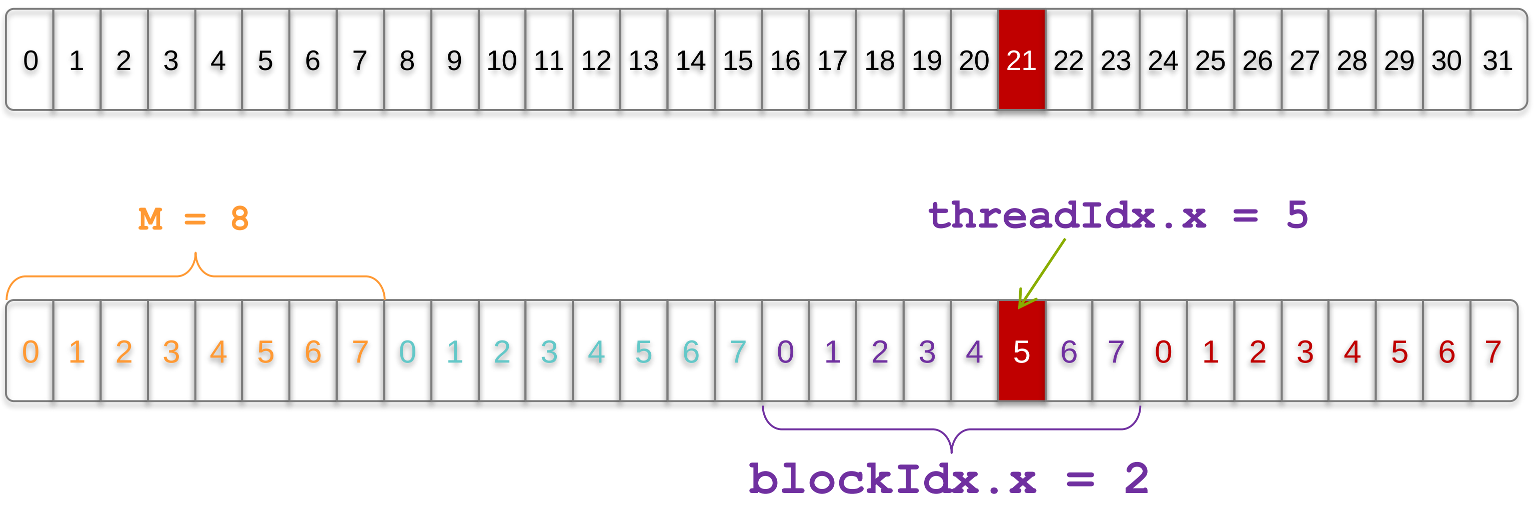


- With M threads/block a unique index for each thread is given by:

```
int index = threadIdx.x + blockIdx.x * M;
```

Example 5: Indexing Arrays

- Which thread will operate on the red element?




```
int index =   threadIdx.x   + blockIdx.x * M;
          =             5   +      2     * 8;
          =   21;
```

Example 6: Vector Addition with Blocks and Threads

- Use the built-in variable `blockDim.x` for threads per block

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

- Combined version of `addition()` to use parallel threads *and* parallel blocks

```
__global__ void addition(int *a, int *b, int *c) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    c[index] = a[index] + b[index];
}
```

- What changes need to be made in `main()`?

Example 6: Vector Addition with Blocks and Threads

```
#define SIZE (2048*2048)
#define THREADS_PER_BLOCK 512

// Printing an Array
void printArray(int* a, int n) {
        for (int i = 0; i < n; i++) {
                printf("c[%d] = %d\n", i, a[i]);
        }
}

int main(void) {
        int* a, * b, * c;

        // Allocate space in Unified Memory for a,b,c
        cudaMallocManaged((void**)&a, SIZE * sizeof(int));
        cudaMallocManaged((void**)&b, SIZE * sizeof(int));
        cudaMallocManaged((void**)&c, SIZE * sizeof(int));
```

Example 6: Vector Addition with Blocks and Threads

```
    // Initializing the values
    for (int i = 0; i < SIZE; i++) {
            a[i] = i;
            b[i] = i;
            c[i] = 0;
    }
    // Launch addition() kernel on GPU with N blocks
    addition<<< SIZE/ THREADS_PER_BLOCK, THREADS_PER_BLOCK >>>(a, b, c, SIZE);

    // Wait for GPU threads to complete
    cudaDeviceSynchronize();

    // Printing the Output Array
    printArray(c, 10);

    // Cleanup
    cudaFree(a);cudaFree(b); cudaFree(c);

    return 0;
}
```

Handling Arbitrary Vector Sizes

- Typical problems are not friendly multiples of `blockDim.x`
- Avoid accessing beyond the end of the arrays:

```
__global__ void add(int *a, int *b, int *c, int n) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < n)
        c[index] = a[index] + b[index];
}
```

- Update the kernel launch:

```
add<<<(SIZE + M-1) / M,M>>>(d_a, d_b, d_c, SIZE);
```

- M = THREADS_PER_BLOCK

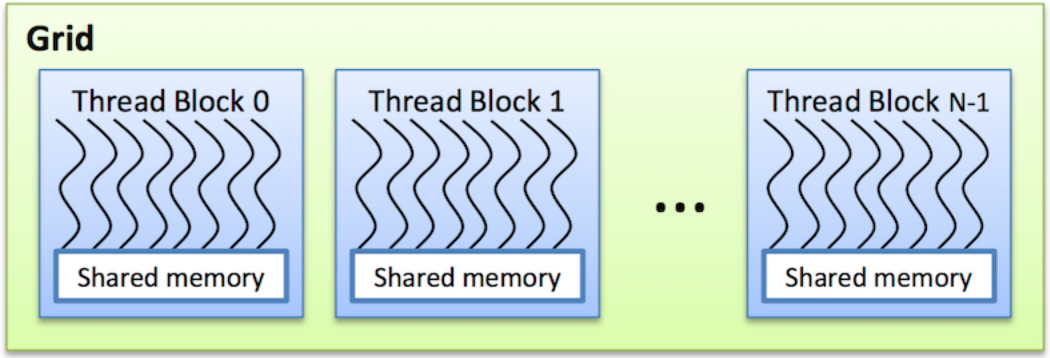
Why Bother with Threads?

- Threads seem unnecessary
  - They add a level of complexity
  - What do we gain?

- Unlike parallel blocks, threads have mechanisms to:
  - Communicate
  - Synchronize

- To look closer, we need a new example…

CUDA Concept:
COOPERATING THREADS

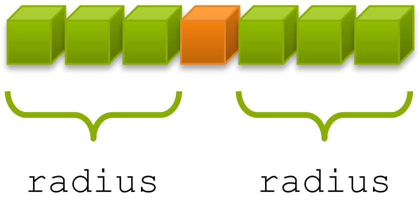


1D Stencil

- Stencil codes are a class of iterative kernels which update array elements according to some fixed pattern, called a stencil.

- Consider applying a 1D stencil to a 1D array of elements
  - Each output element is the sum of input elements within a radius

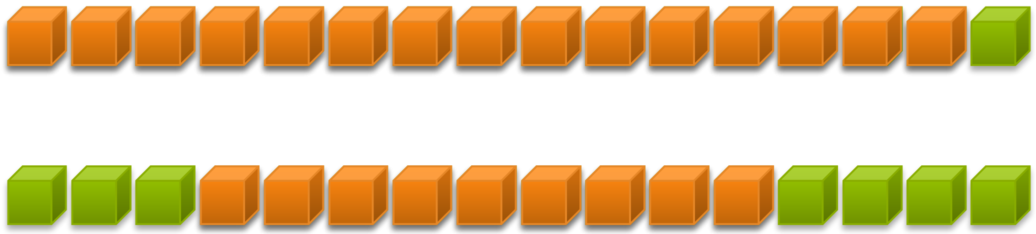
- If radius is 3, then each output element is the sum of 7 input elements:



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



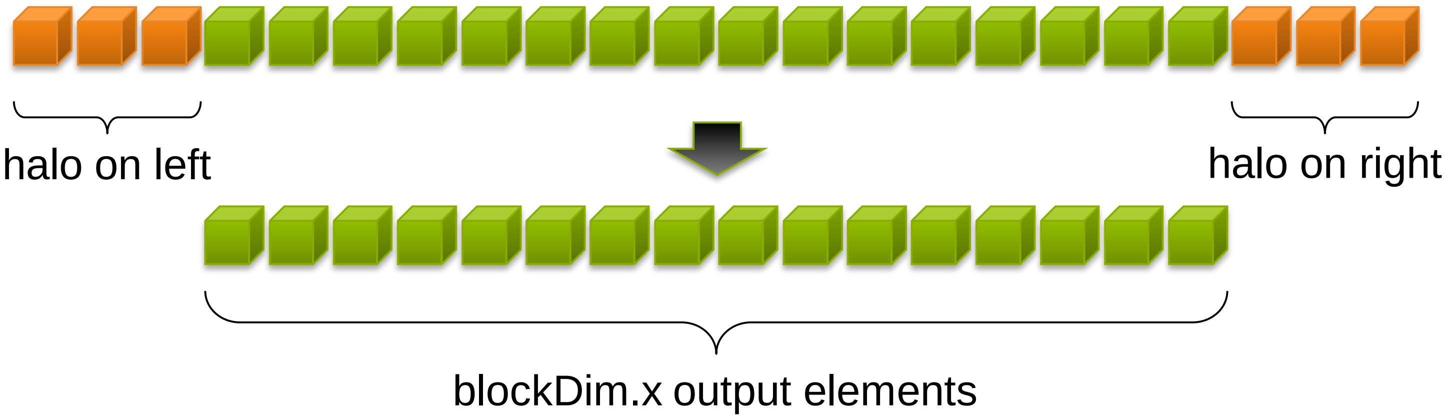
Sharing Data Between Threads

- 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 blocks

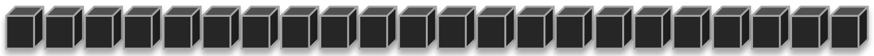
Implementing With Shared Memory

- 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

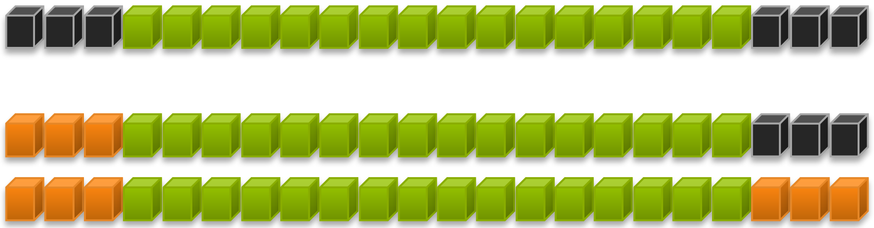
  - Each block needs a halo of radius elements at each boundary



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
  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 reads the halo before thread 0 has fetched it…

```
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];
```

Store at temp[18]

Skipped, threadIdx > RADIUS

Load from temp[19]

__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

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

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;
}
```

CUDA Concept: MANAGING THE DEVICE

Coordinating Host & Device

- Kernel launches are asynchronous
  - Control returns to the CPU immediately (while GPU is still running some tasks)

- CPU needs to synchronize before consuming the results

`cudaMemcpy()`	Blocks the CPU until the copy is complete Copy begins when all preceding CUDA calls have completed
`cudaMemcpyAsync()`	Asynchronous, does not block the CPU
`cudaDeviceSynchronize()`	Blocks the CPU until all preceding CUDA calls have completed

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 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

Final Tips and Warnings!!!

- Trying exploring the given sample CUDA projects in the lab computers
  - C:\ProgramData\NVIDIA Corporation\CUDA Samples
- **Some** CUDA syntax might be marked red by Visual Studio. For example:

```
int i = blockIdx.x;
int i = threadIdx.x;
<<< 1,1>>>
```

- Don't worry too much. As long as the code compiles you are all good!

References:

Material taken from NVIDIA:

https://developer.nvidia.com/

https://devblogs.nvidia.com/easy-introduction-cuda-c-and-c/

https://medium.com/@erangadulshan.14/1d-2d-and-3d-thread-allocation-for-loops-in-cuda-e0f908537a52

For programming API reference:

C: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

C++: https://devblogs.nvidia.com/even-easier-introduction-cuda/

Python: https://developer.nvidia.com/how-to-cuda-python
https://documen.tician.de/pycuda/,
https://devblogs.nvidia.com/cudacasts-episode-12-programming-gpus-cuda-python/

Java: http://www.jcuda.org/

Matlab: https://www.mathworks.com/solutions/gpu-computing/getting-started.html