CUDA Programming

Ehsan Yousefzadeh-Asl-Miandoab

(ehyo@itu.dk)

Input: an array, Output: squared

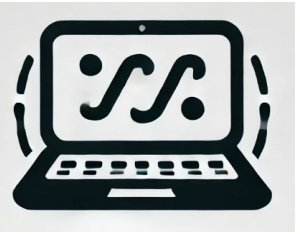

```cpp
void squareArrayCPU(int *input, int *output, int size) {
    for (int i = 0; i < size; i++) {
        output[i] = input[i] * input[i];
    }
}
```

```cpp
__global__ void squareArrayGPU(int *d_input, int *d_output, int size) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < size) {
        d_output[idx] = d_input[idx] * d_input[idx];
    }
}
```

Execution time = 1024 * 2 ns = 2048 ns

Execution Time = 10 ns + 2 * (Data Transfer Overhead) + (Kernel Launch Overhead)

Programming Assignment #1



- Get the given assignment code running, which are for CPU, and GPU

CUDA_for_ITU/assignments/01-CPU_GPU_difference at main · ehsanyousefzadehasl/CUDA_for_ITU

- Read the code carefully to understand its functionality. Then, experiment with different input arguments and observe the differences in execution time between the CPU and GPU implementations.

Number of elements in the input array	CPU Time	GPU Time

CUDA

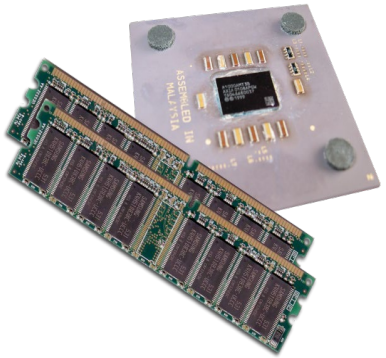
- **C**ompute **U**nified **D**evice **A**rchitecture

- CUDA C/C++
  - Based on standard C/C++
  - Set of extensions enabling heterogeneous programming

- Pre-requisites:
  - Experience with C/C++

Heterogeneous Computing

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



Host

Device

Heterogeneous Computing

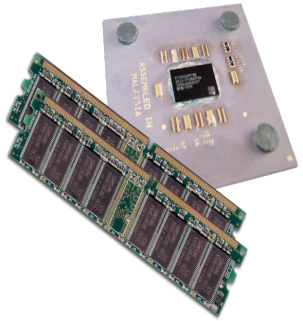
```
#include <iostream>
#include <algorithm>

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;
}
```

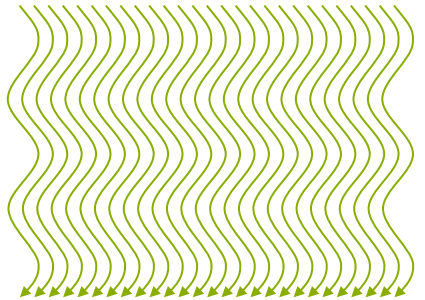
parallel fn

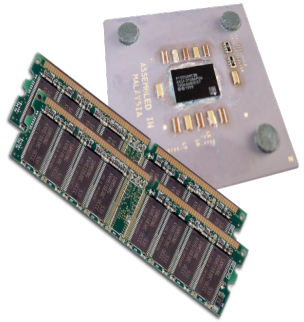
serial code

parallel code

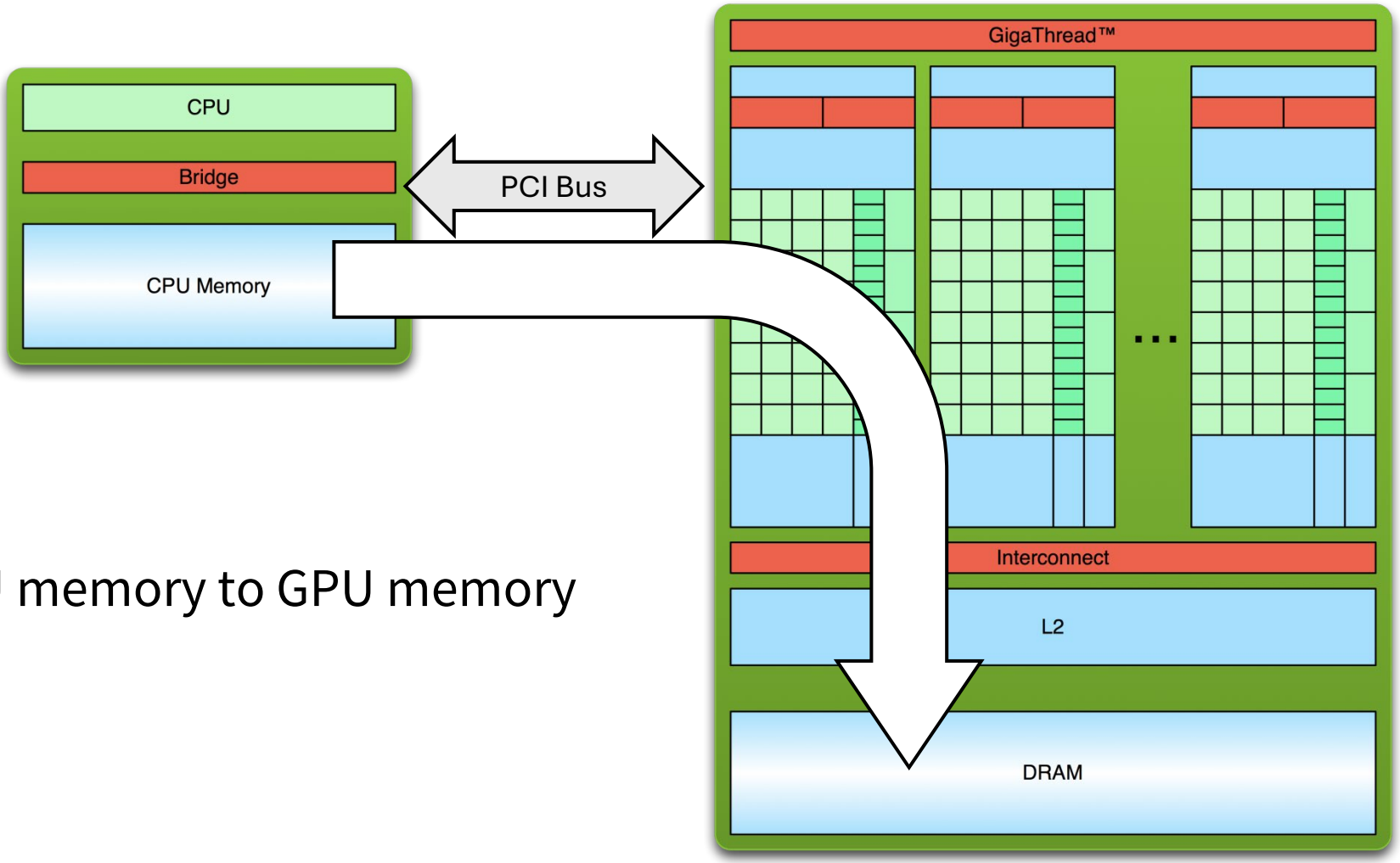
serial code





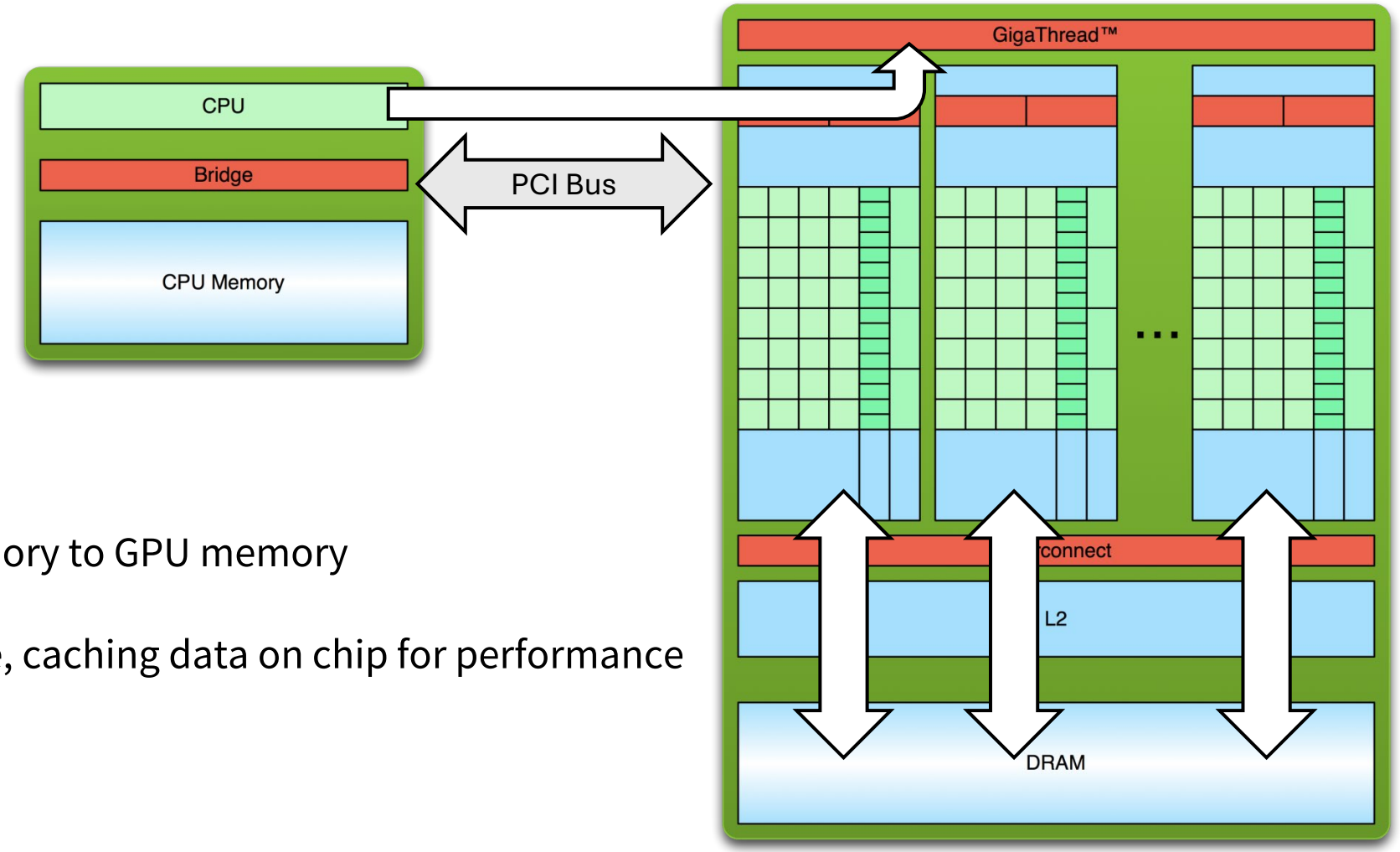


Simple Processing Flow



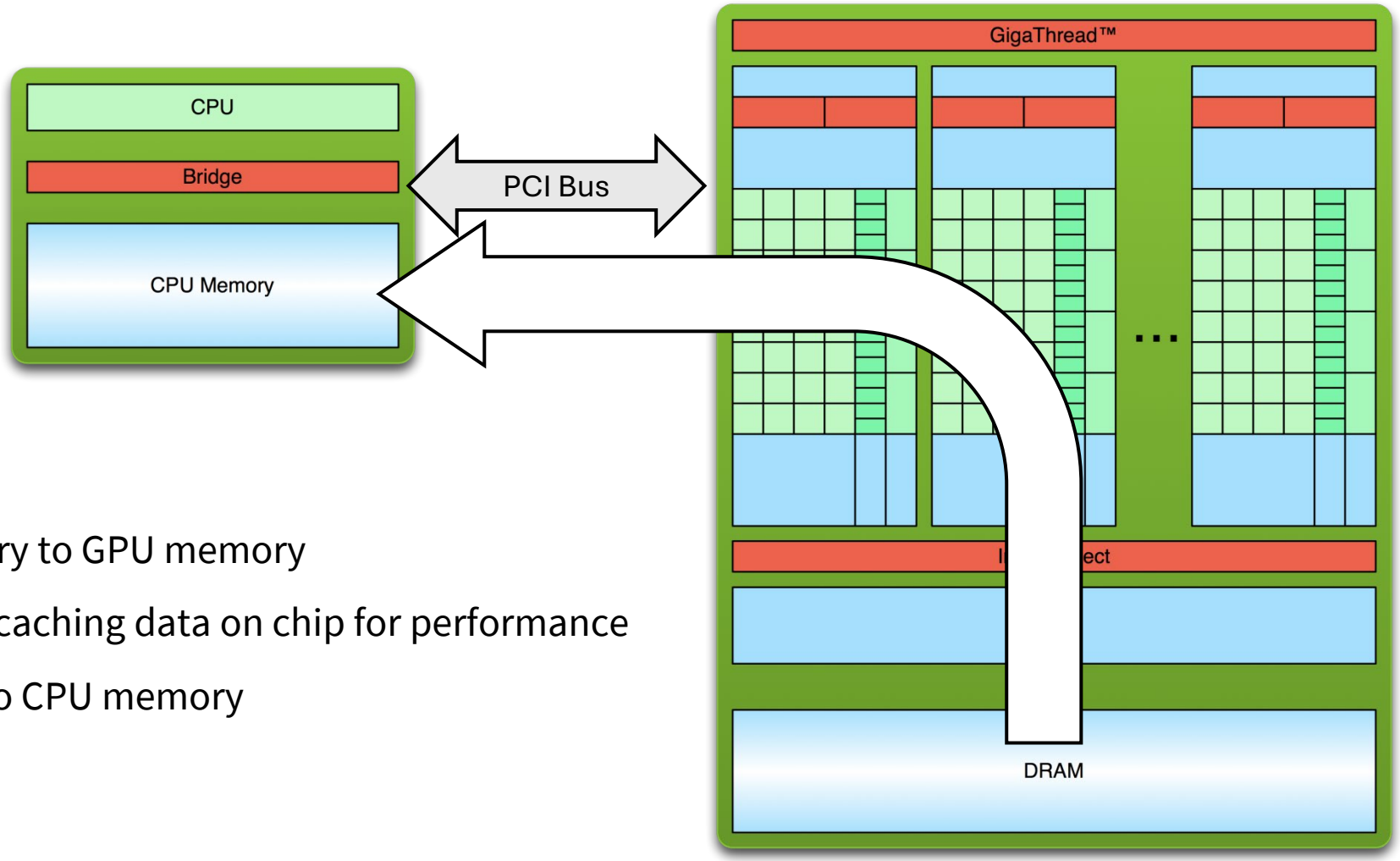
1. Copy input data from CPU memory to GPU memory

Simple Processing Flow



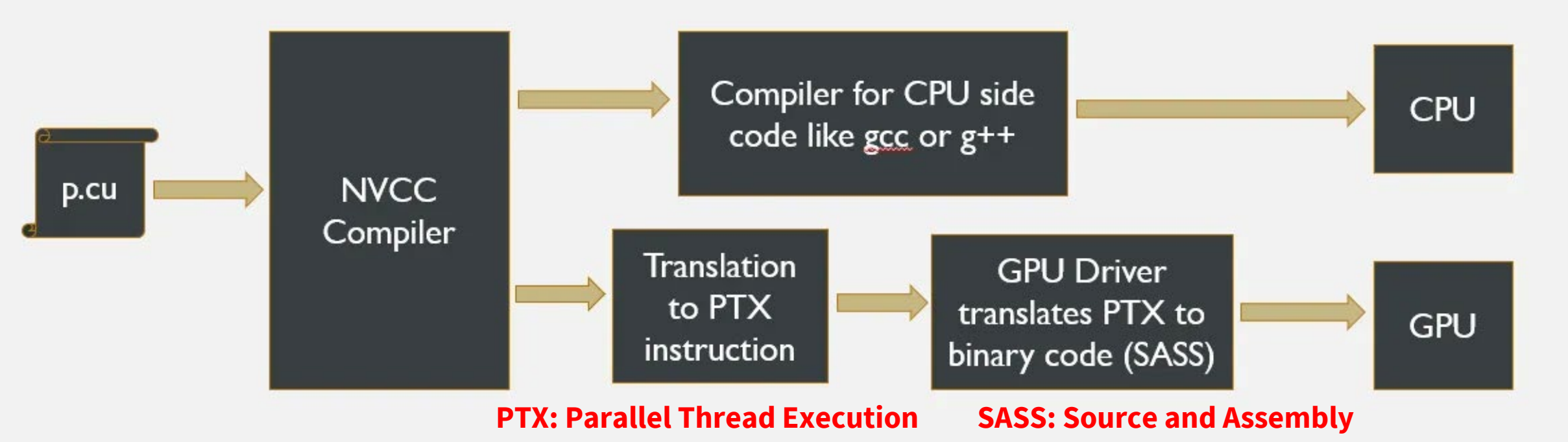
1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

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

From Code to Execution: The CUDA Process



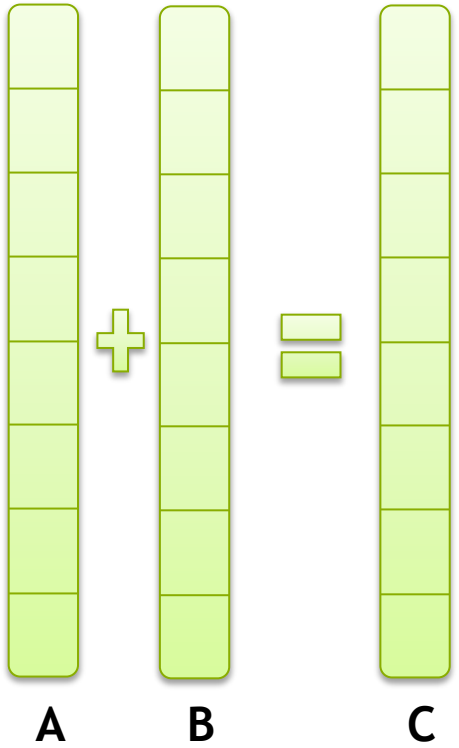

```
$ nvcc my_program.cu -o my_program
```

Hello World!


```
__global__ void helloWorld(void) {
      printf("Hello from thread %d from block %d\n",
            threadIdx.x, blockIdx.x);
}

int main(void) {
       helloWorld<<<10,100>>>();

      return 0;
      }
```

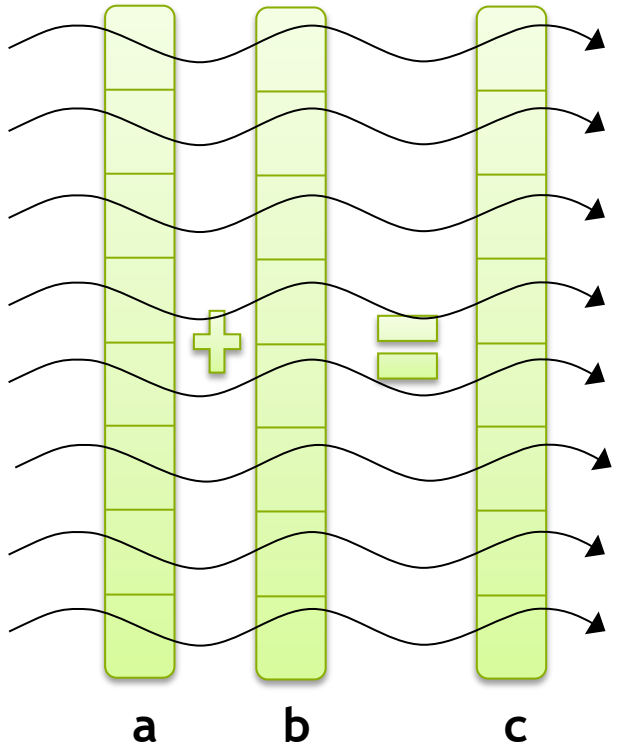
Adding two Arrays

- We have two input arrays: A, and B
- Goal: calculate C = A + B with GPU
- How do we usually do it with CPU?
  - Loops
  - SIMD instructions



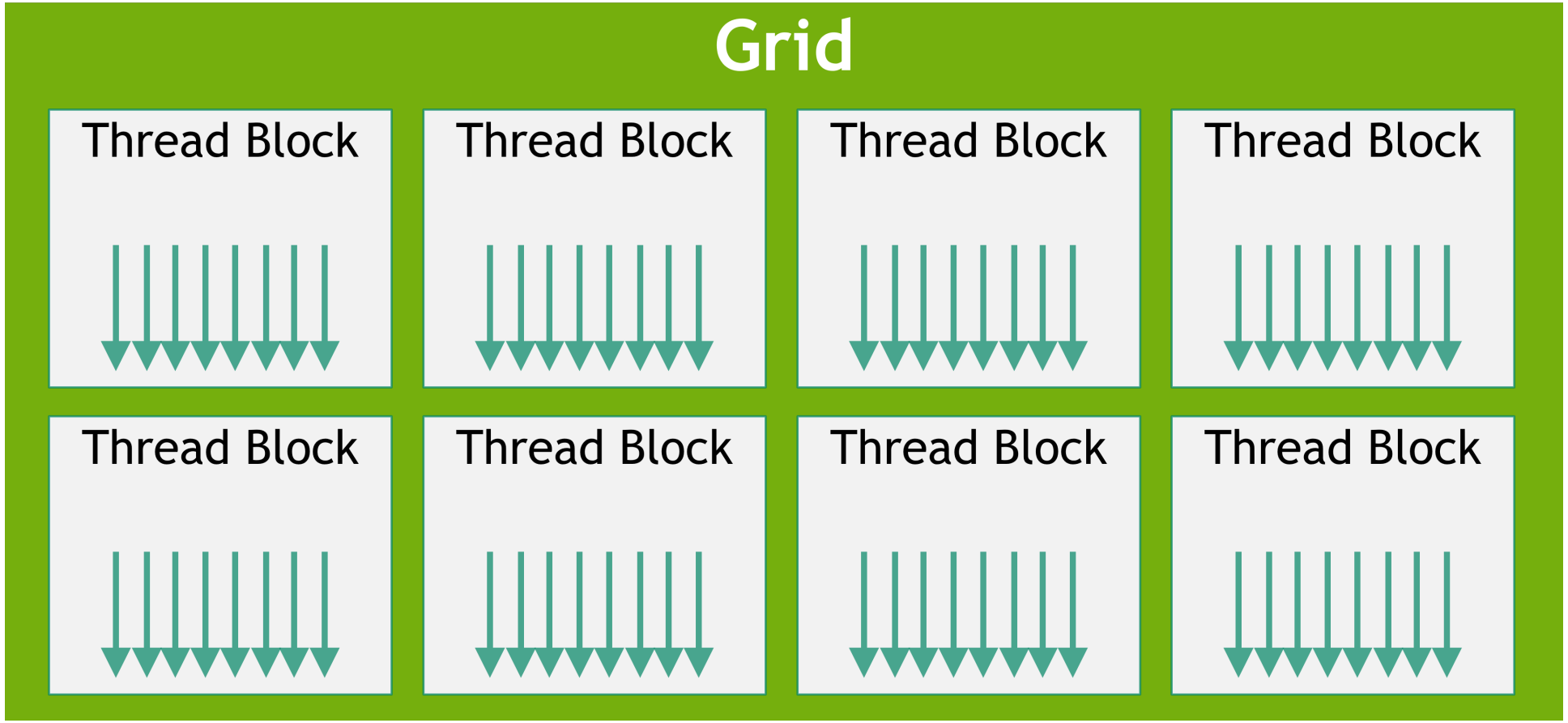
Adding two Arrays

- Having an independent thread
  - For each add operation



1. Arrays on the CPU (host) memory
2. Allocate memory on GPU (device) memory
3. Copy arrays to device memory
4. Launch the kernel (pass it the arguments)
5. Copy back the results from device memory to host memory

Terminology

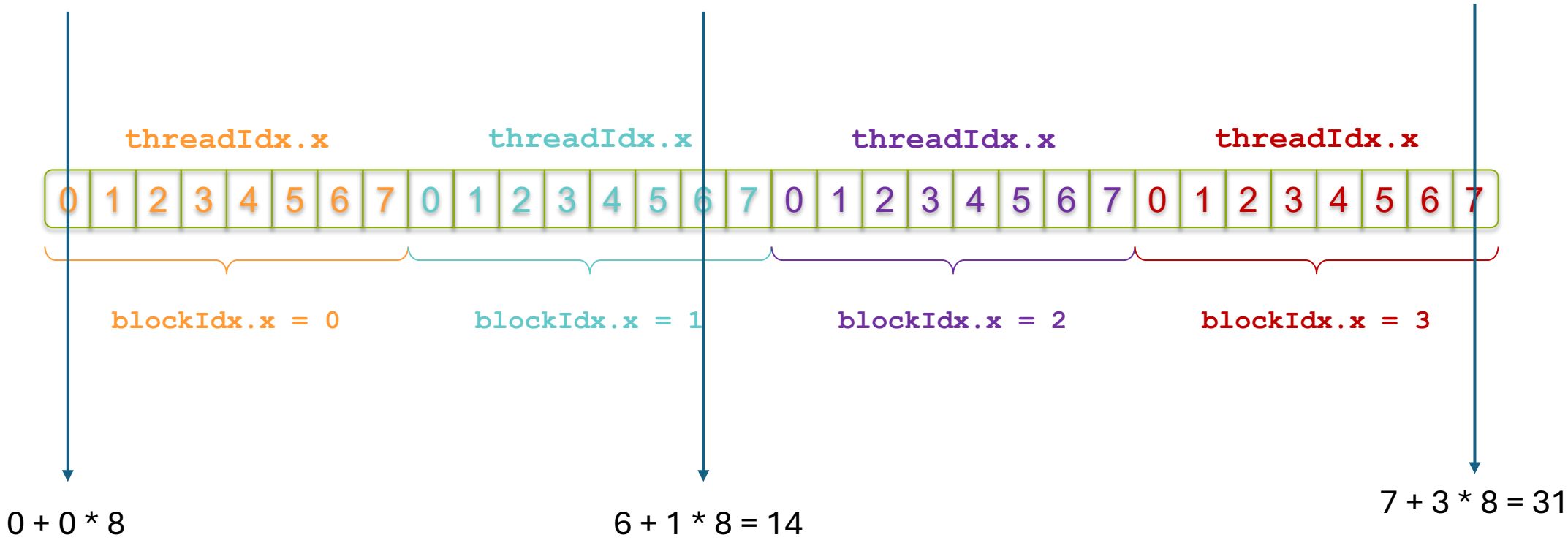


[CUDA C++ Programming Guide]

The logic behind the add we checked!

- Assume **`add<<<4, 8>>>(A, B, C, N)`**

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



ITU’s HPC Cluster

- The cluster is a shared computing resource used by **students** and **staff** for running computational tasks, simulations, and data processing workloads, etc. efficiently. It is designed to handle **multiple users simultaneously** while ensuring **fair access** to resources.

- The cluster uses **SLURM** (Simple Linux Utility for Resource Management) as the scheduler. SLURM is responsible for: **Assigning tasks to available compute nodes** in the cluster. **Managing queues** to ensure tasks are executed in the appropriate order based on **priority** and **resource availability**.

ITU HPC

HPC.ITU.DK ITU HPC Documentation

How to login!

Step 1:

```
$ ssh your_username@hpc.itu.dk
```

Step 2:

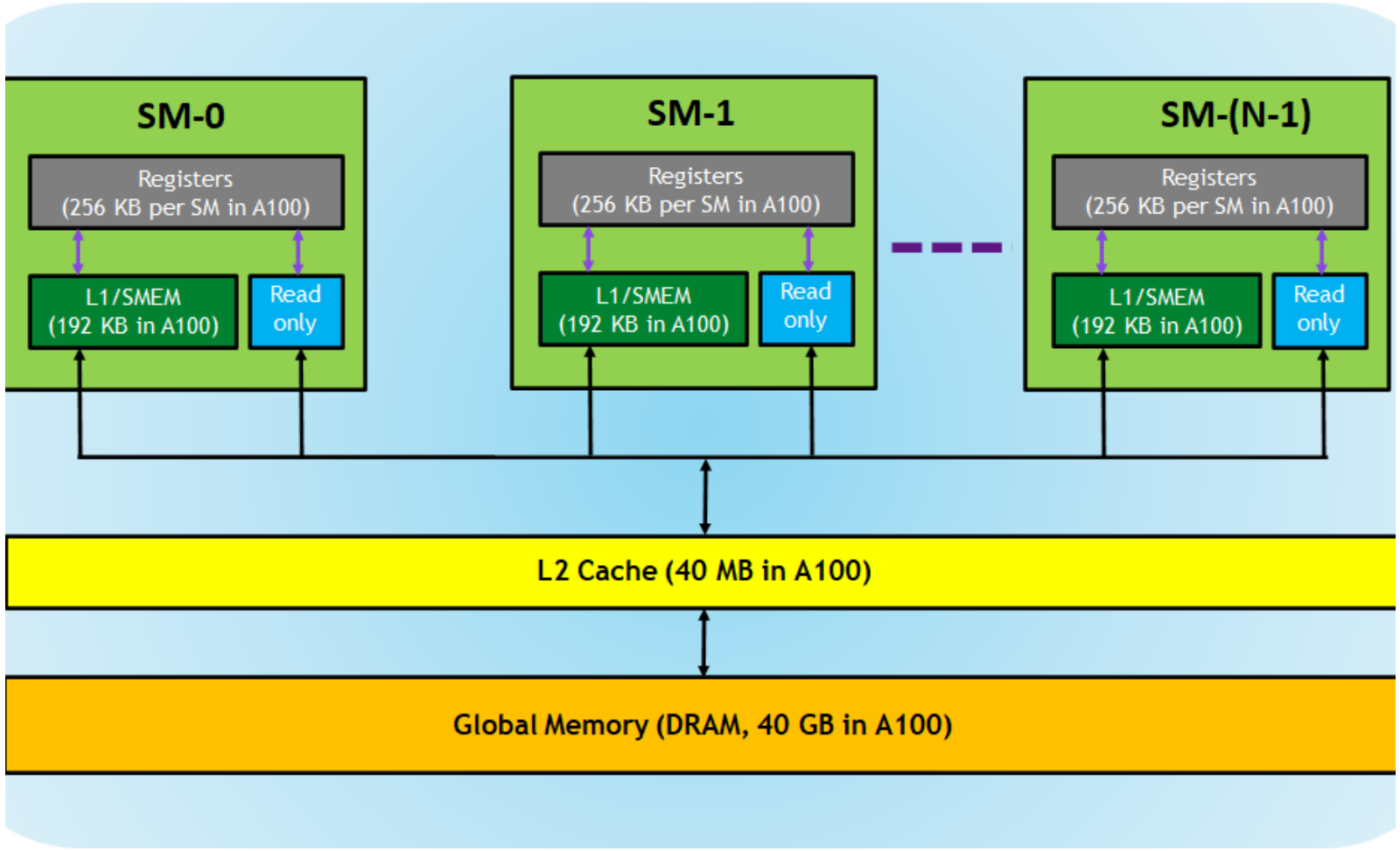
```
Enter your password:
```

EXPLORE AROUND AND SEE WHAT WE HAVE ON THE CLUSTER!

How to submit a task!


```
#!/bin/bash

#SBATCH --job-name=cuda_test_job_name        # Job name
#SBATCH --output=cuda_test_output_name       # output file name
#SBATCH --cpus-per-task=1                    # Schedule 1 cores (includes hyperthreading)
#SBATCH --gres=gpu                           # Schedule a GPU, it can be on 2 gpus like gpu:2
#SBATCH --time=00:05:00                      # Run time (hh:mm:ss) - run for one hour max
#SBATCH --partition=scavenge

module load CUDA/12.1.1

nvcc test_cuda.cu -o test_cuda
./test_cuda
```

Shared Memory



https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/

Shared Memory

Shared memory is a small, fast memory space shared by all threads in a block.

Significantly faster than global memory

Resides

Shmem enables threads within a block to share data, which can lead to substantial performance improvements by:

- **Reducing Redundant Memory Accesses**:
  - If multiple threads need the same data, they can share it in shared memory instead of each thread fetching it separately from global memory.
- **Minimizing Global Memory Latency**:
  - Global memory access is slow compared to shared memory. Using shared memory avoids repeatedly accessing global memory for frequently used data.
- **Enabling Thread Collaboration**:
  - Threads can share intermediate results via shared memory, making it easier to implement cooperative algorithms.

Maximum Example

```
for (int stride = blockDim.x / 2; stride > 0; stride >>= 1) {
    if (tid < stride) {
        sharedData[tid] = max(sharedData[tid], sharedData[tid + stride]);
    }
    __syncthreads();
}

if (tid == 0) {
    *finalMax = sharedData[0];
}
```

Programming Assignment #2

- Compute the dot product of two vectors using CUDA and compare it to the one available in the repository that uses shared memory (add the timing to both source codes). Experiment with different length of the input array, different block size.

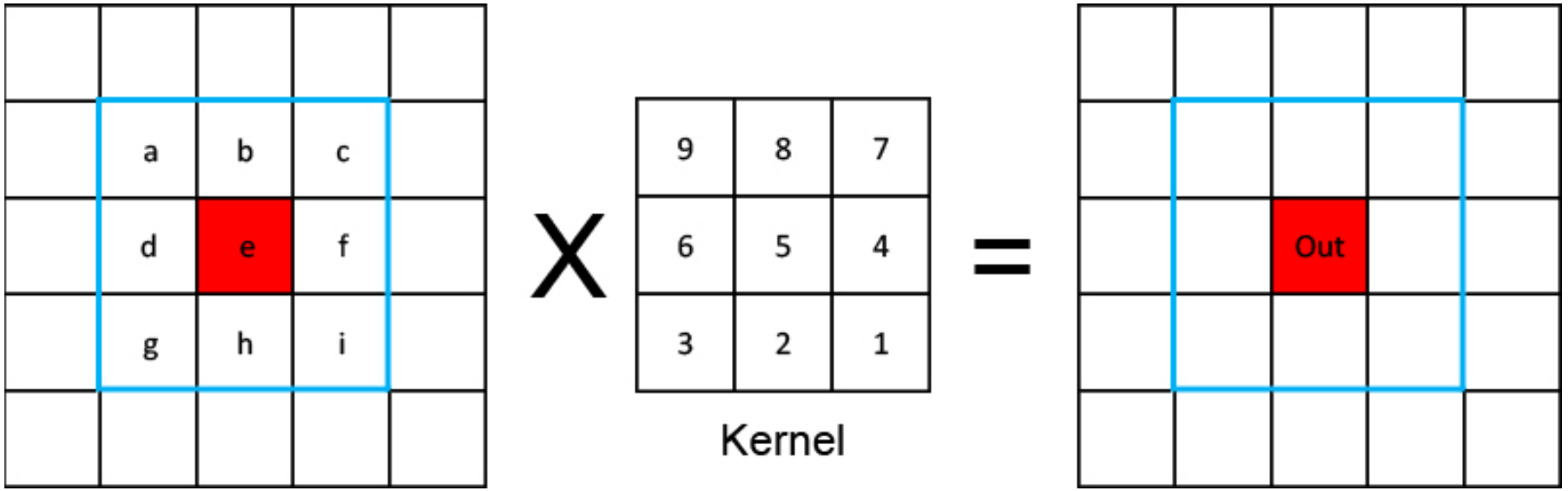
- Write your reflection on how using shared memory improves the performance.

ehsanyousefzadehasl/CUDA_for_ITU

OPTIONAL

Stencil

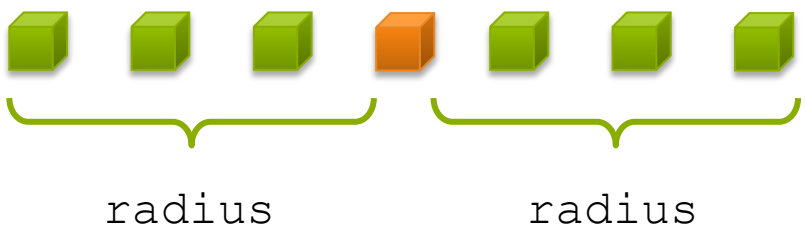
- Tasks read input from a fixed neighborhood in an array.
- Convolution operations in Convolution Neural Networks (CNNs).



1D Stencil Example

- 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:



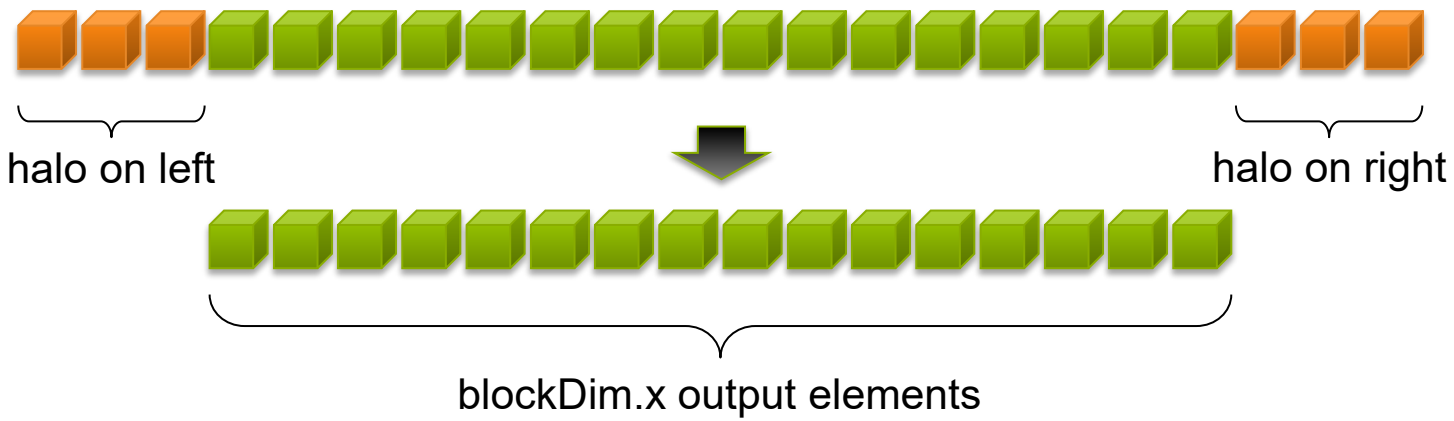
1D Stencil Example

- 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
- 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

1D Stencil Example

- 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

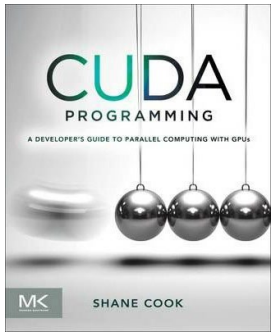


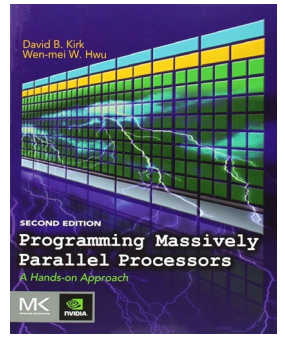
Programming Assignment (Optional)

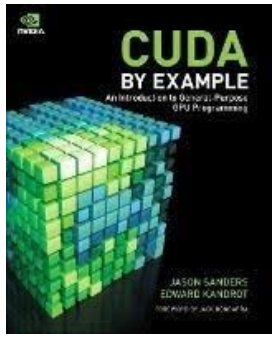
- Develop a CUDA kernel that performs the same **1D stencil operation** without using shared memory. Next, design and execute an experiment to compare the performance of the shared memory version with the non-shared memory version. Highlight the advantages of shared memory by demonstrating reduced global memory accesses and improved execution time in the shared memory approach, especially for large input sizes (compare the time).

Links to learn more!

- NVIDIA - CUDA C++ Programming Guide







- "**CUDA by Example: An Introduction to General-Purpose GPU Programming**" by Jason Sanders and Edward Kandrot
- "**Programming Massively Parallel Processors: A Hands-on Approach**" by David B. Kirk and Wen-mei W. Hwu
- "**CUDA Programming: A Developer's Guide to Parallel Computing with GPUs**" by Shane Cook

Questions?