IT UNIVERSITY OF CPH

Programming with CUDA

Ehsan Yousefzadeh-Asl-Miandoab

(ehyo@itu.dk)

Some slides are for later reading.

Search and read more about unfamiliar keywords.

Slides with the "Hands-On" tag indicate that the teacher will dive into the code and execute it live during the class.

Content

- Why GPUs?
- CUDA and Heterogeneous Computing
- CUDA Concepts and Hands-on Examples
- Learning how to use the University's HPC Cluster

Why GPUs?

GPUs offer much **higher instruction throughput** and **memory bandwidth** than CPUs within a similar price and power envelope, making them ideal for applications that require high parallelism. Unlike CPUs, which excel at executing a few threads sequentially, GPUs are designed to handle **thousands of threads simultaneously**, achieving greater throughput by focusing more on data processing rather than data caching and flow control.

Input: an array, Output: squared


```
length_of_array = 1024;
for(int i = 0; i < length_of_array; i++) {
    out[i] = in[i] * in[i];
}
```

```
__global__ void square(float * d_out, float * d_in) {
    int idx = threadIdx.x; // threadIdx is a cuda built-in variable
    float f = d_in[idx];
    d_out[idx] = f * f;
}
```

```
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

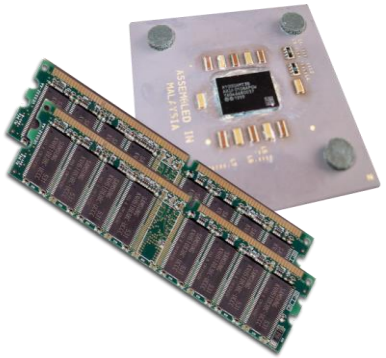
- **C**ompute **U**nified **D**evice **A**rchitecture

- 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

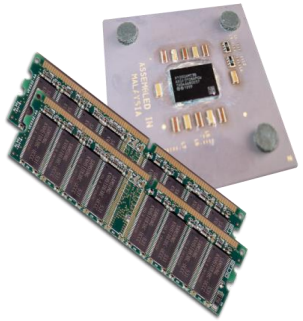
```
#include <iostream>
#include <algorithm>

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;
}
```

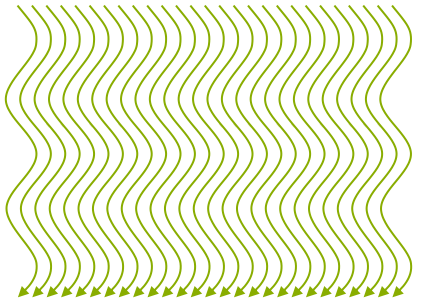
parallel fn

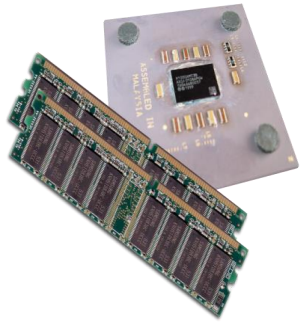
serial code

parallel code

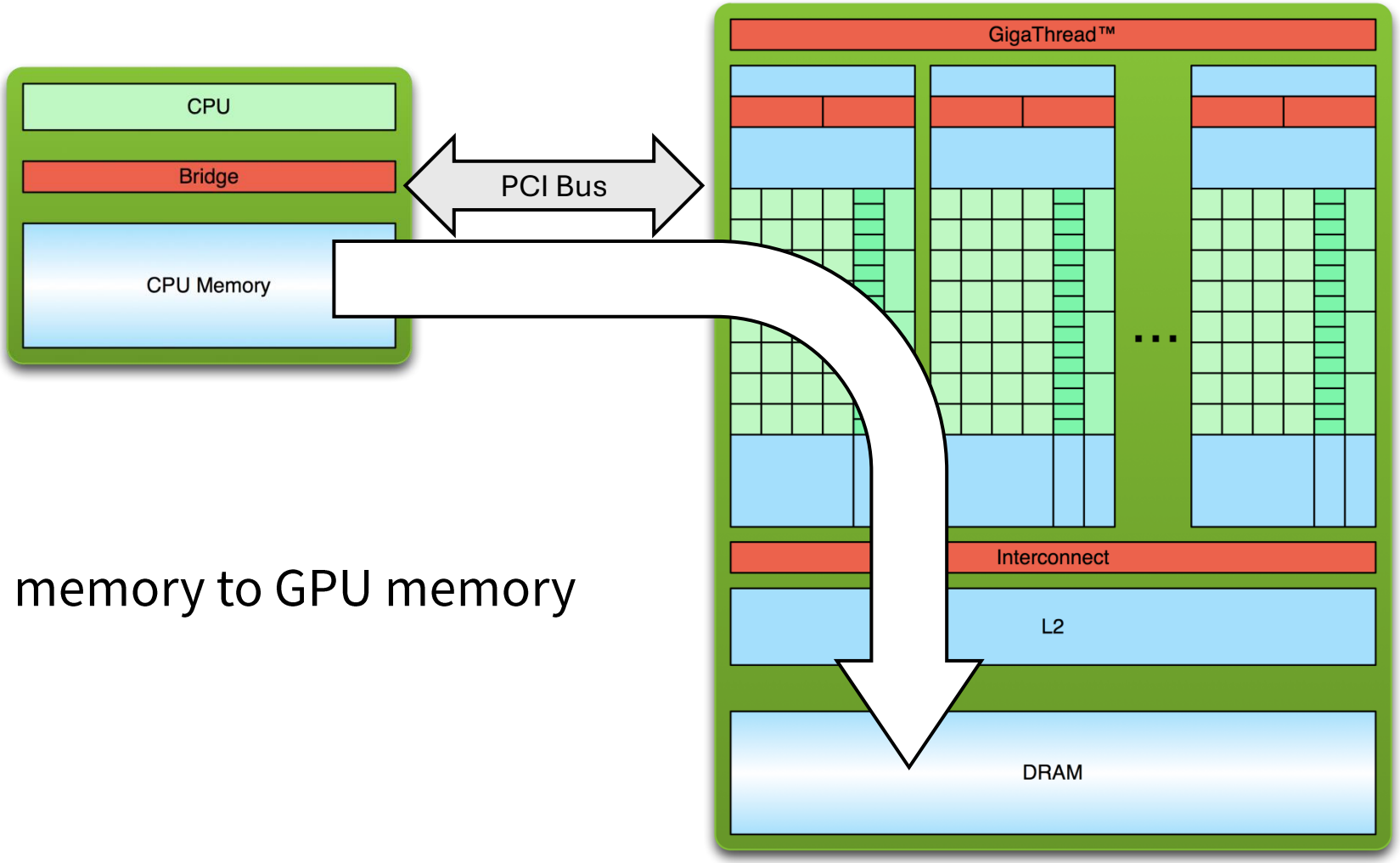
serial code





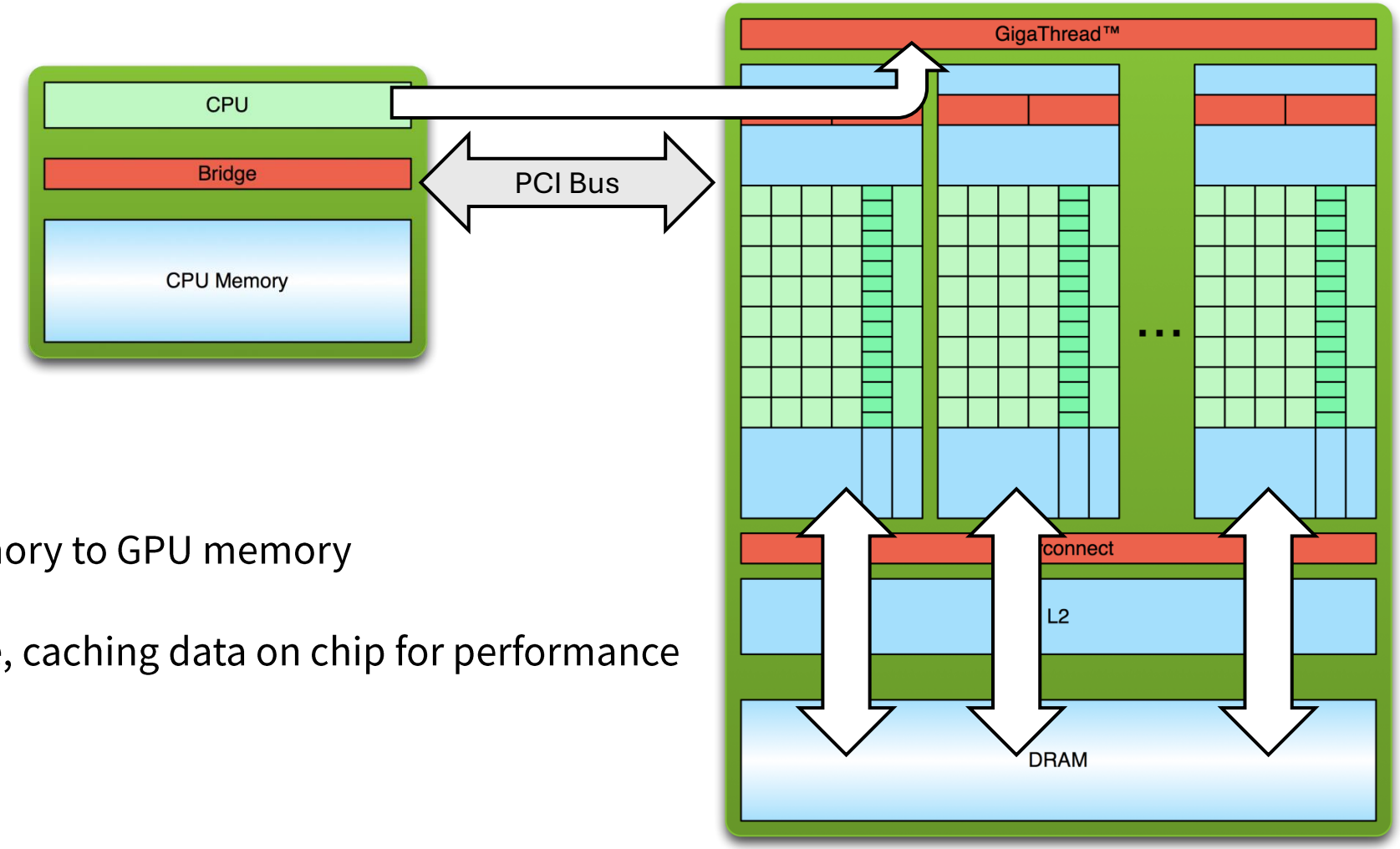


Simple Processing Flow



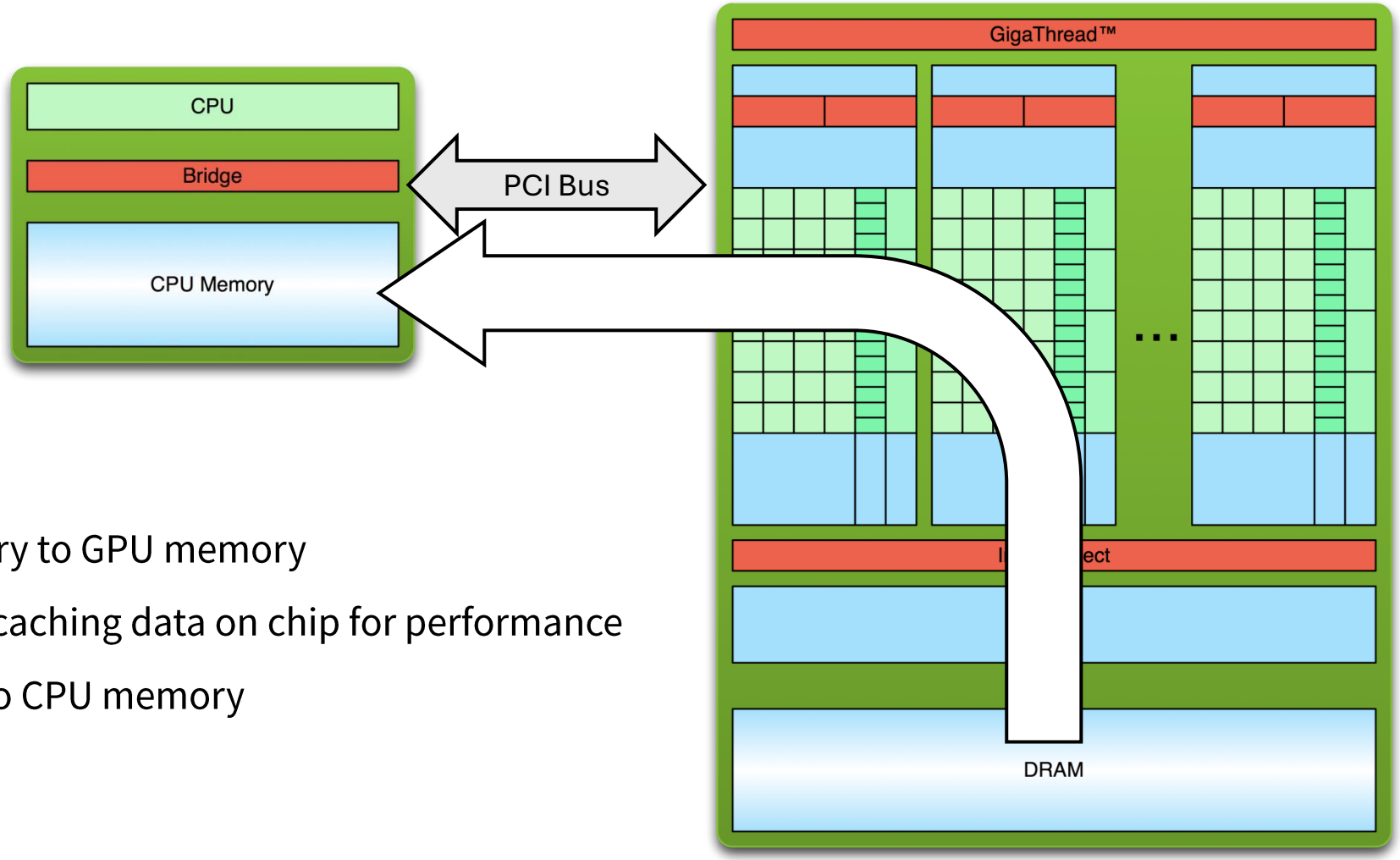
1. Copy input data from CPU memory to GPU memory

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

Hello World!

- NVIDIA compiler (`$nvcc`) can be used to compile programs with no *device* code

- Standard C ”hello world” program!

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

```
$ nvcc 00-hello_world.cu -o 00-hello_world
```

Hello World!


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

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

      return 0;
      }
```

Hello World! with Device Code

- CUDA C/C++ keyword `__global__` indicates a function that:
  - Runs on the device
  - Is called from host code

- `nvcc` separates source code into host and device components
  - Device functions (e.g. `helloWorld()`) processed by NVIDIA compiler
  - Host functions (e.g. `main()`) processed by standard host compiler

Hello World! with Device Code

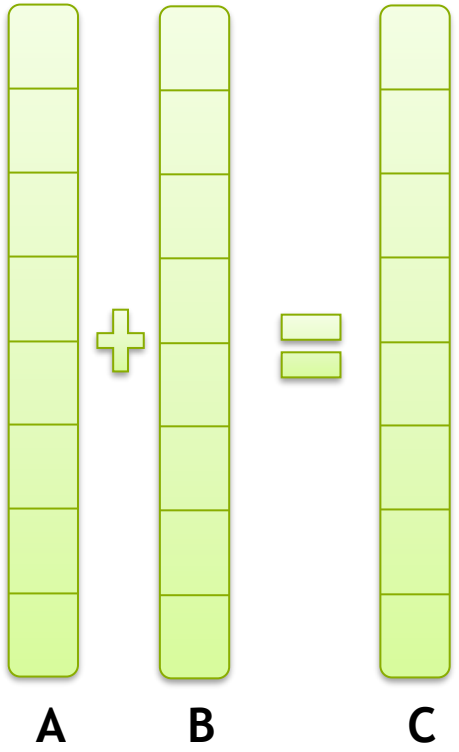
```
helloWorld<<<1,1>>>();
```

- Triple angle brackets mark a call from *host* code to *device* code
  - Also called a “kernel launch”
  - Parameters (1,1) indicate the number of thread block and the number of threads in each thread block

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

Adding two Arrays

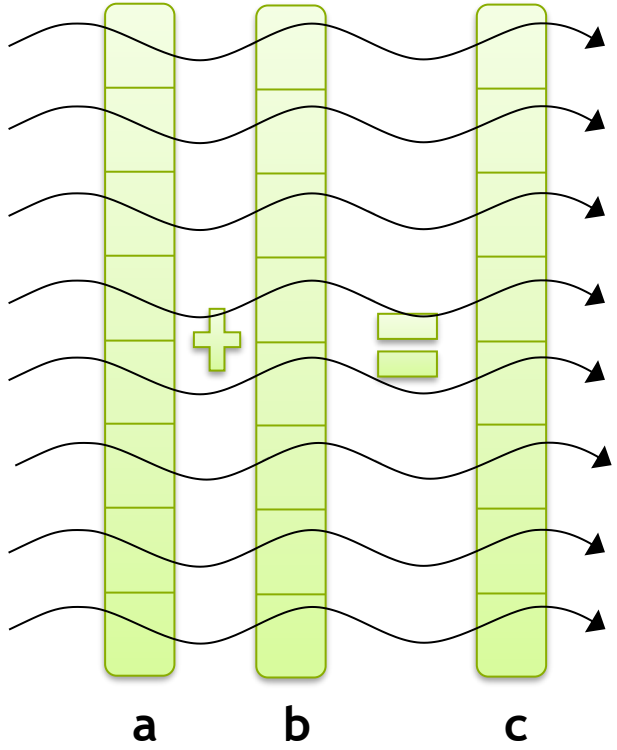
- We have two input arrays: A, and B
- Goal: calculate C = A + B with GPU
- How do we usually do it with CPU?
  - Loops
  - SIMD instructions



CPU SIMD Instructions

SIMD (Single Instruction, Multiple Data) Instructions: SIMD is a type of parallel processing in CPUs where a single instruction is executed **simultaneously** on multiple pieces of data. This is particularly useful for operations like **vector** and **matrix computations**, **image processing**, or any tasks that involve performing the same operation on large datasets. In SIMD, data is stored in vectors (arrays of elements), and special SIMD registers process multiple elements in parallel. For example, a CPU with 256-bit SIMD registers can process eight 32-bit numbers or sixteen 16-bit numbers at once. This allows significant performance improvements by leveraging **data-level parallelism**. Modern CPUs provide SIMD extensions like **Intel's SSE and AVX**, or **ARM's NEON**, which are designed to optimize workloads in fields like **scientific computing**, **multimedia processing**, and **machine learning**. SIMD helps CPUs handle tasks that require **high throughput**, enabling **faster computations** compared to processing data sequentially.

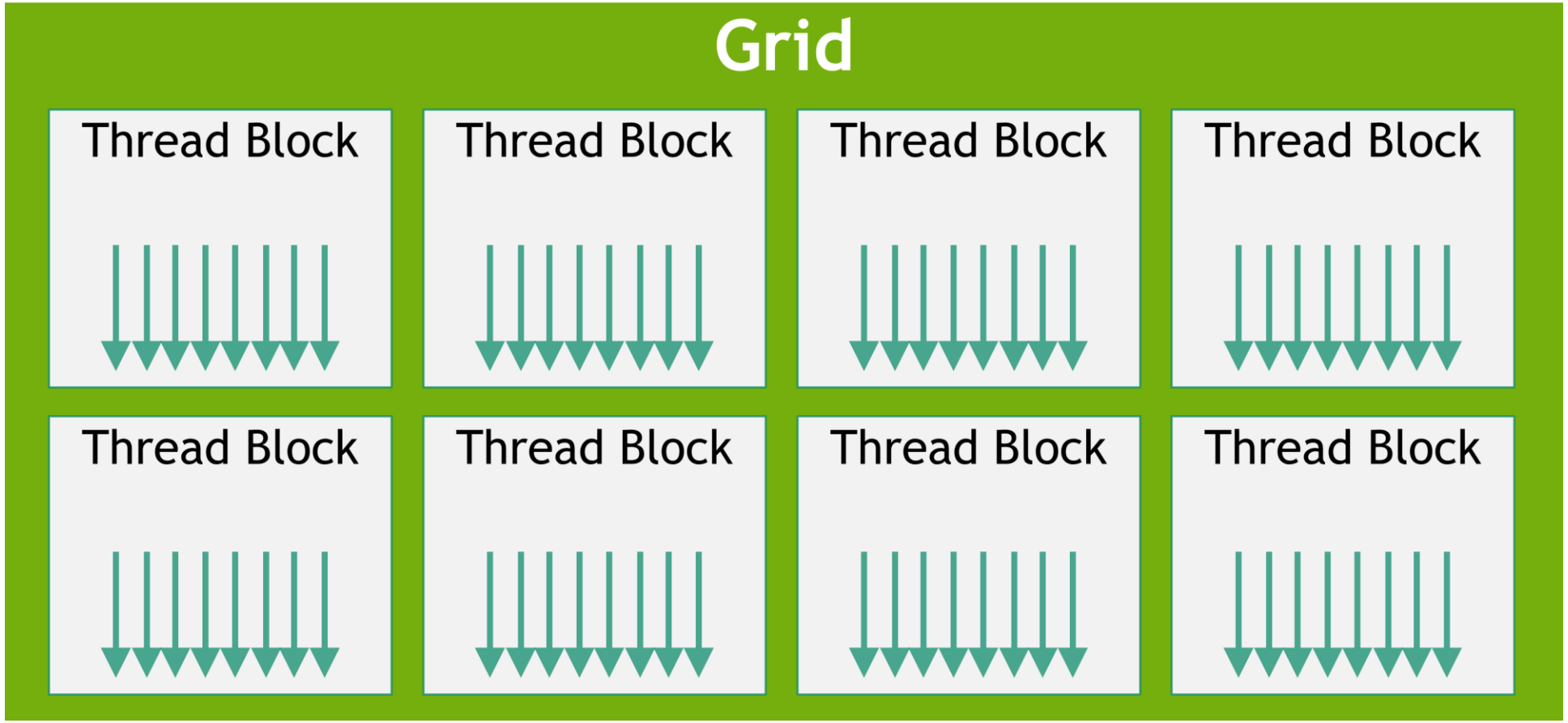
Adding two Arrays



- 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

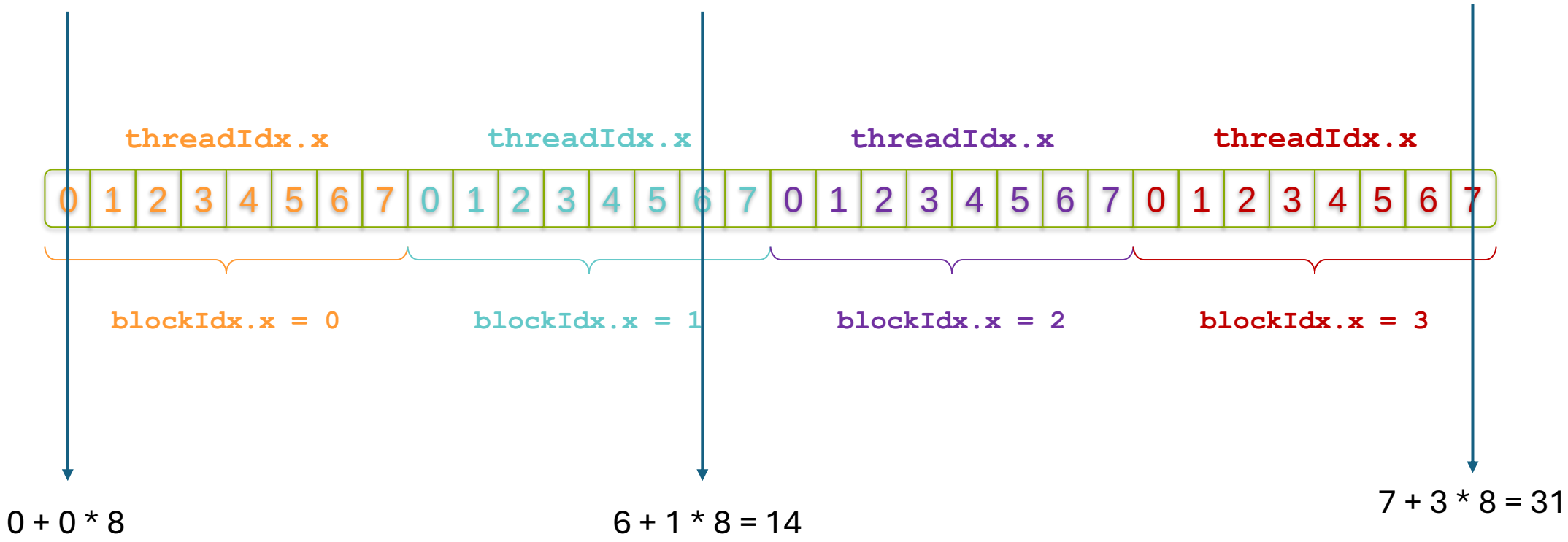


CUDA C++ Programming Guide

The logic behind the add we checked!

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

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



Basic device (GPU) memory management

- cudaMalloc()
- cudaMemcpy()
- cudaFree()

Parallel Communication Patterns

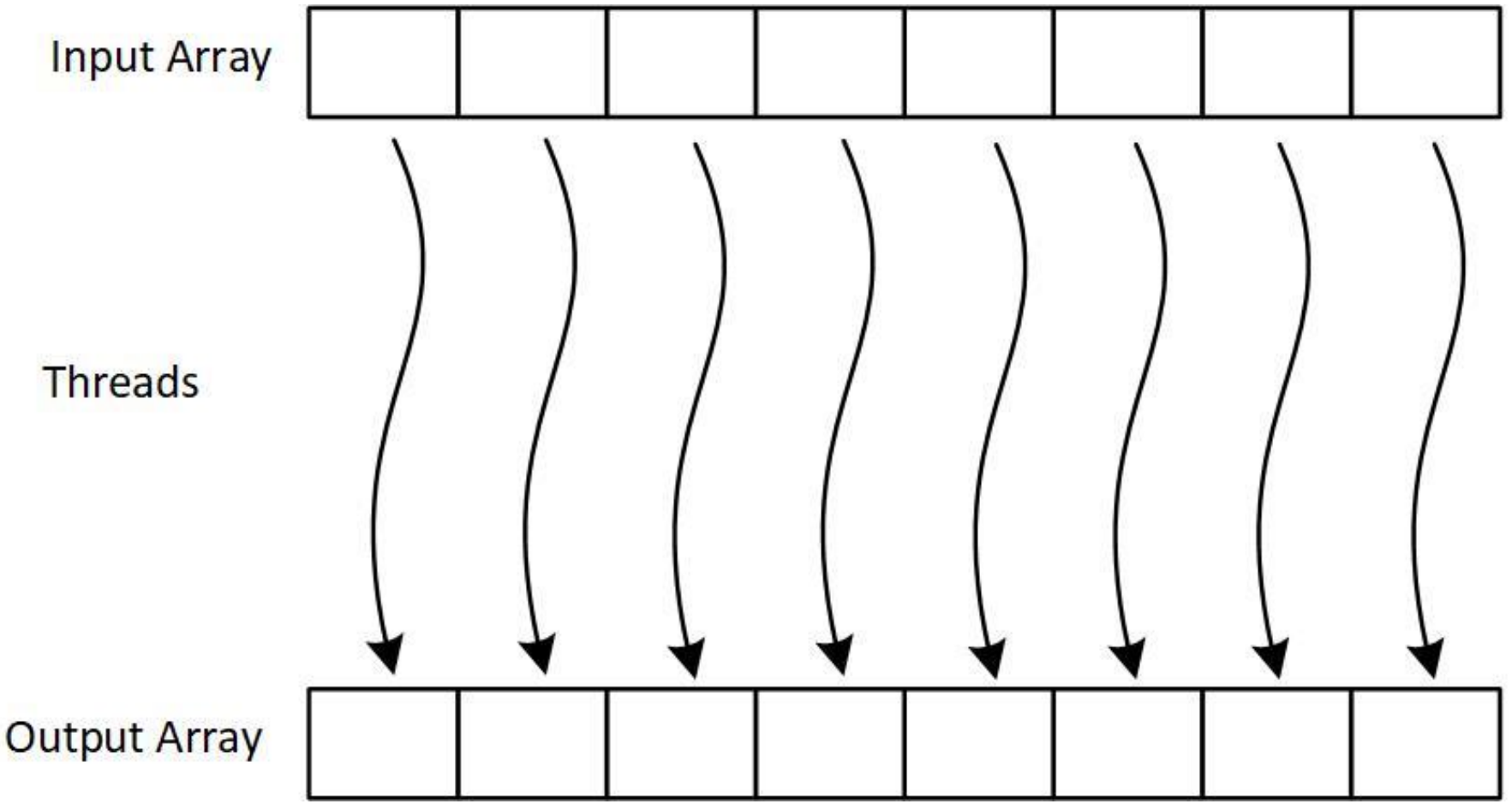
- Parallel computing is about many threads solving a problem by working together. The key to working together is **communication**. In CUDA, communication takes place through **memory**.

- There are different kinds of parallel communication patterns and they are about how to map tasks (threads) and memory.

- Some of the important patterns: **map**, **gather**, **scatter**, **stencil**, **transpose**

Map

The example of calculating the squared value of each element.



Map

- The map parallel pattern is a fundamental concept in parallel programming where the same operation is independently applied to every element of a dataset. It is an embarrassingly parallel pattern, meaning there are no dependencies between elements, so all operations can be executed in parallel without requiring communication between threads.

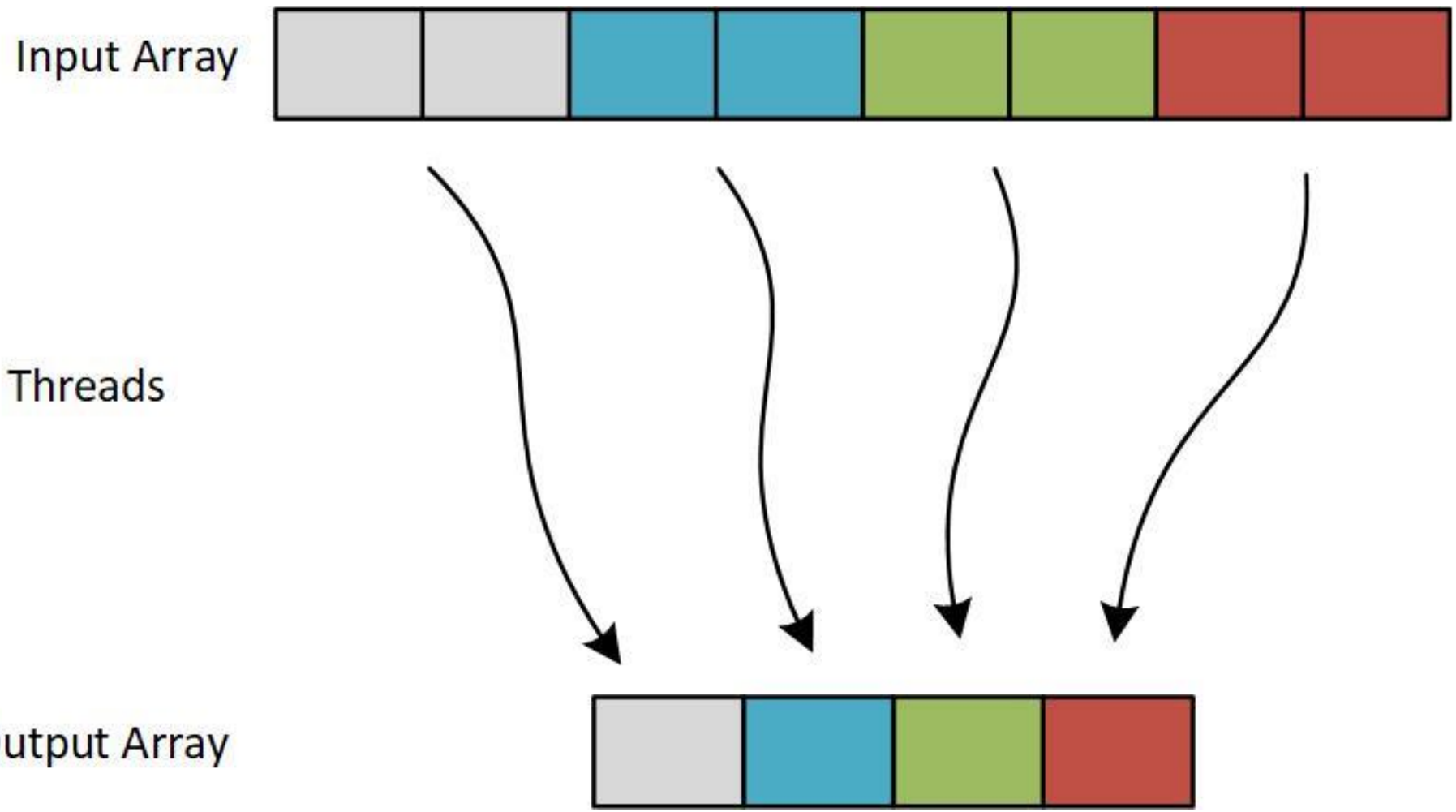
- In CUDA programming, the map pattern is often implemented by assigning one thread to process one or more elements of the dataset. The map pattern is particularly efficient because it allows maximum utilization of GPU cores by distributing work evenly across threads.

Applications of Map

- **Image Processing**: Applying filters (e.g., grayscale conversion, brightness adjustment) where each pixel can be processed independently.
- **Scientific Simulations**: Computing mathematical functions (e.g., sine, cosine, or exponential) for large arrays of input values.
- **Data Transformation**: Converting or normalizing datasets, such as scaling numerical values or applying logarithmic transformations.
- **Graphics Rendering**: Transforming vertex coordinates or applying color transformations in GPU-accelerated graphics.
- **Machine Learning**: Element-wise activation functions (e.g., ReLU, sigmoid) applied to neural network layers.

The map pattern's simplicity and lack of inter-thread communication make it a highly efficient and scalable approach for parallelizing independent computations.

Gather



Gather

The **gather parallel pattern** is a common approach in parallel programming where data is **collected from multiple memory locations into a single output dataset**. Each thread retrieves data from one or more indices of the input dataset and performs operations to produce its corresponding result. Unlike the map pattern, the gather pattern often involves **non-contiguous memory accesses**, as threads may need to access scattered input locations. In CUDA, implementing an efficient gather pattern requires **careful memory management** to minimize uncoalesced global memory access.

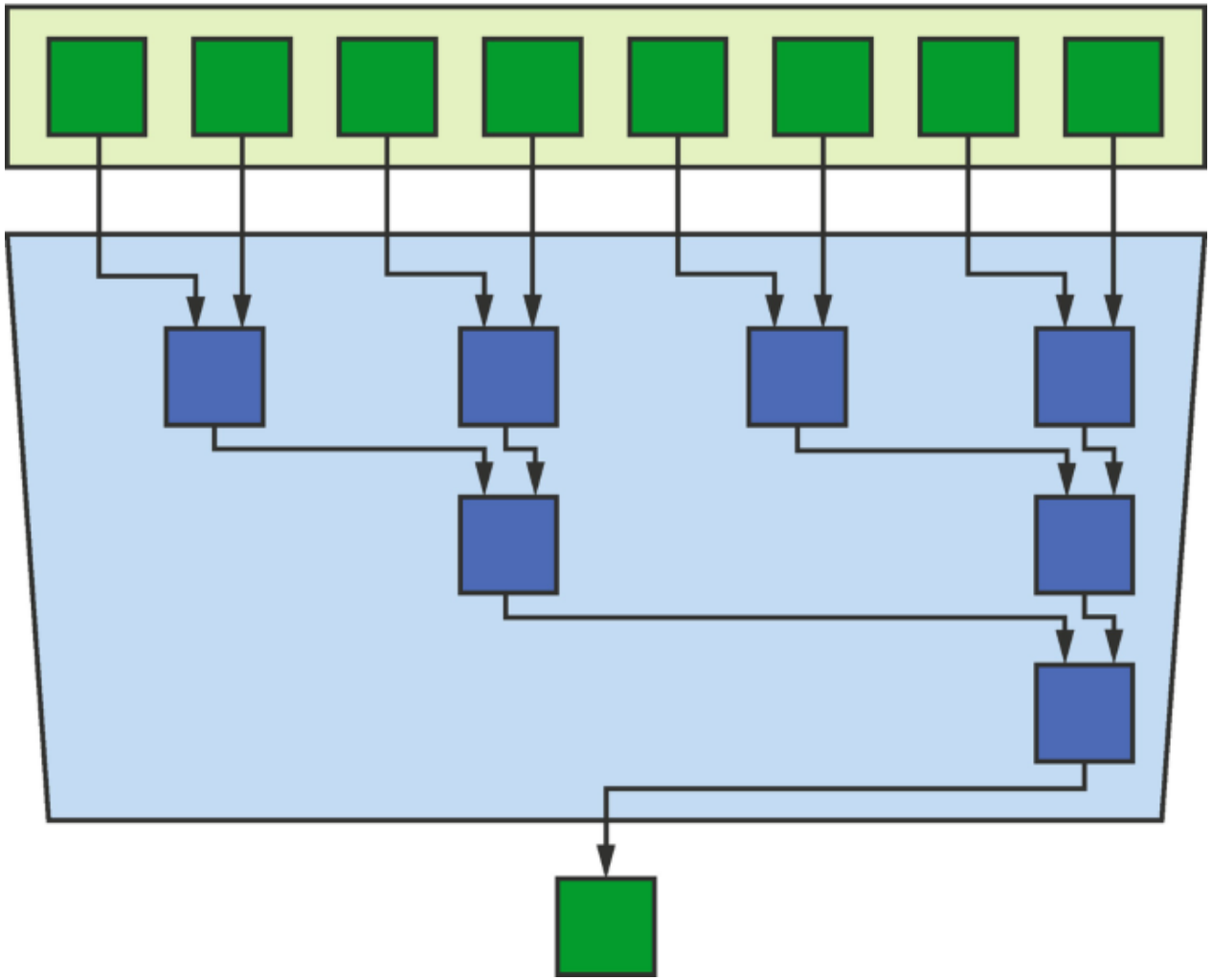
Applications of Gather

The gather pattern is frequently used in tasks where specific data needs to be extracted or rearranged:

1. **Matrix Operations**: Extracting rows, columns, or specific elements for sub-matrix computations.
2. **Image Processing**: Sampling data from scattered pixel locations (e.g., texture mapping, image warping).
3. **Scientific Simulations**: Collecting data points from irregular grids or domains for further processing.
4. **Data Rearrangement**: Reorganizing datasets (e.g., shuffling, grouping, or sorting by specific criteria).
5. **Graphics and Rendering**: Gathering vertex or texture data for 3D transformations or rendering pipelines.

The gather pattern's flexibility makes it ideal for handling irregular or scattered datasets, though optimizing memory access patterns is crucial to achieve high performance on GPUs.

Reduce



Common Parallel Patterns | SpringerLink

Reduce

The **reduce parallel pattern** involves **combining elements of a dataset into a single result** using a specified operation, such as summation, multiplication, or finding the maximum. In CUDA, this pattern is typically implemented by assigning threads to process parts of the dataset and then performing a **hierarchical reduction** in shared memory, where partial results are iteratively combined until only one result remains. Reduction is a key pattern in parallel programming as it efficiently aggregates data while minimizing global memory accesses.

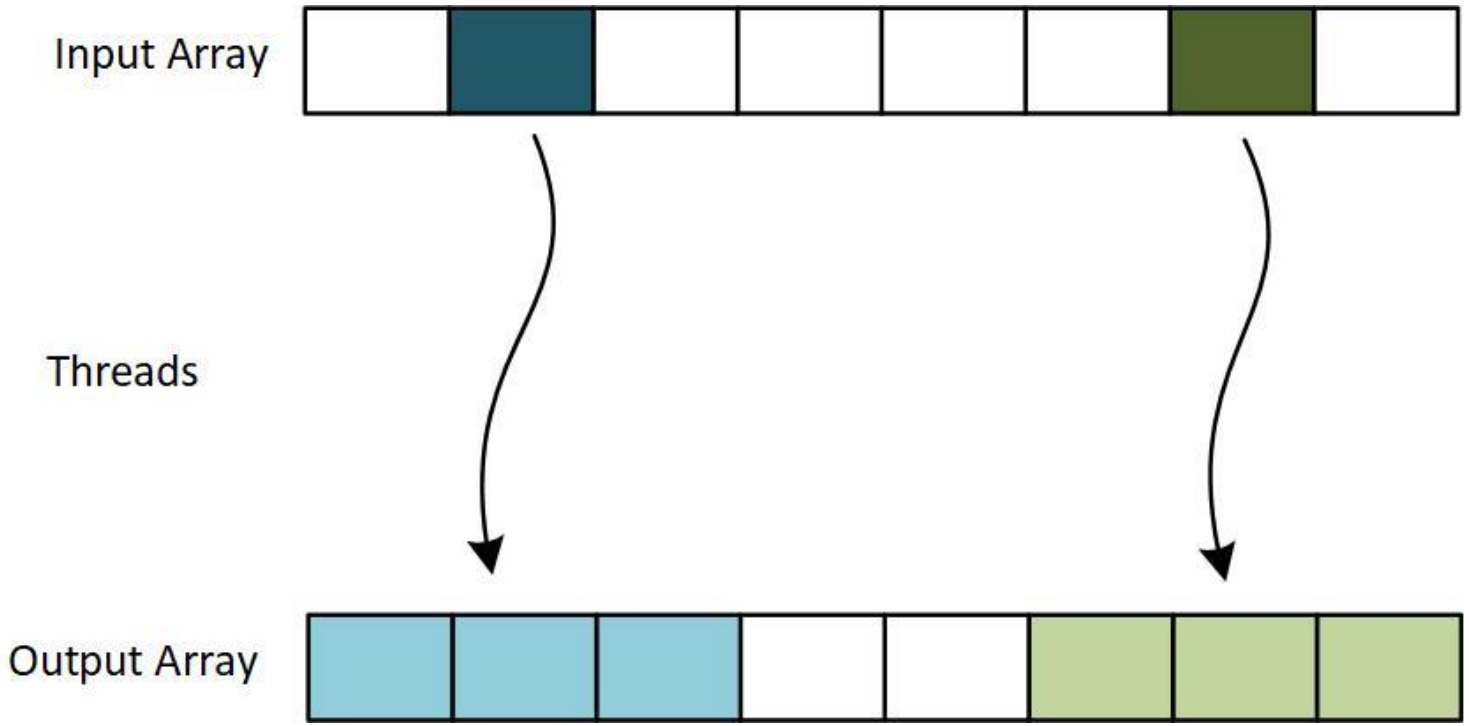
Applications of Reduce

1. **Scientific Simulations**: Summing physical quantities (e.g., energy, mass) or calculating averages over large datasets.
2. **Data Analytics**: Computing metrics like totals, maximums, minimums, or variance across datasets.
3. **Machine Learning**: Summing gradients during backpropagation or aggregating results in distributed computations.
4. **Graphics**: Calculating light intensity or pixel averages in rendering pipelines.
5. **Financial Modeling**: Aggregating transaction data for totals or risk analysis.

The reduce pattern is essential for summarizing large datasets efficiently, with shared memory and synchronization ensuring optimal performance on GPUs.

Scatter

- **Tasks compute where to write output**.
- An example is **sorting numbers** in an array because each thread will compute where to write its output.
- **Note that just two threads are shown to clearly demonstrate the pattern.**



Scatter

The **scatter parallel pattern** distributes data from a single input dataset to **specific locations in an output dataset**. Each thread takes a portion of the input and writes it to one or more indices in the output, often based on a mapping or index array. Unlike the gather pattern, which collects data, scatter focuses on **placing data in non-contiguous or irregular memory locations**. In CUDA, efficient scatter implementations require careful handling of memory writes to avoid **race conditions** and ensure coalesced access when possible.

Applications of Scatter

1. **Sorting Algorithms**: Writing elements to their correct positions in a sorted array.
2. **Sparse Matrix Representations**: Distributing values into specific non-zero locations in sparse matrices.
3. **Graphics and Rendering**: Assigning texture data or transforming vertices into frame buffers.
4. **Data Partitioning**: Splitting datasets into groups or buckets based on a condition or key.
5. **Simulation and Modeling**: Distributing particle properties (e.g., position, velocity) into spatial grids.

Scatter is vital for tasks requiring flexible data placement, with synchronization and memory optimization critical for high performance in GPU implementations.

Race Conditions

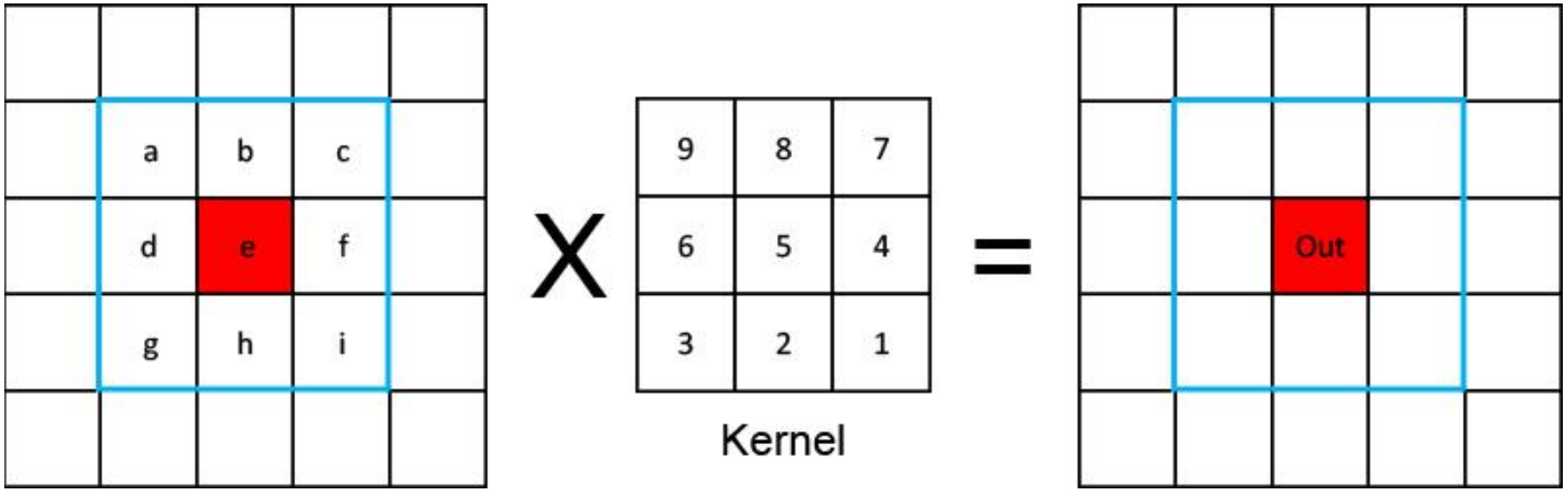
A race condition occurs when multiple threads in a parallel program attempt to access and modify the same memory location simultaneously, leading to undefined or incorrect results. This happens because threads execute independently, and without proper synchronization, one thread may overwrite the changes made by another. In CUDA programming, race conditions are common when threads in a block write to shared memory or when multiple threads across blocks write to the same global memory address. To avoid race conditions, developers can use atomic operations, which ensure only one thread modifies a memory location at a time, or synchronization mechanisms like `__syncthreads()` within blocks to control access to shared resources.

Coalesced Memory Access

Coalesced memory access occurs when threads in a warp access consecutive memory addresses, allowing the GPU to combine those requests into a single transaction with global memory. This is highly efficient because it minimizes the number of memory transactions and maximizes memory bandwidth utilization. Coalesced access is especially important in CUDA programming since uncoalesced accesses (e.g., scattered or misaligned memory requests) result in higher latency and slower performance. Ensuring coalesced access often involves structuring data in memory and aligning thread-to-data mappings so that each thread accesses a unique, contiguous address in global memory. This optimization is crucial for achieving high performance on GPUs.

Stencil

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



Stencil

The stencil parallel pattern is used for computations where each element of an output dataset depends on a specific element in the input dataset and its neighbors. Threads independently process their assigned element and access neighboring values, often defined by a fixed radius. This pattern typically involves regular grid structures and is highly parallelizable, making it well-suited for GPU acceleration. Efficient stencil implementations use shared memory to cache data and reduce redundant global memory accesses.

Applications of Stencil

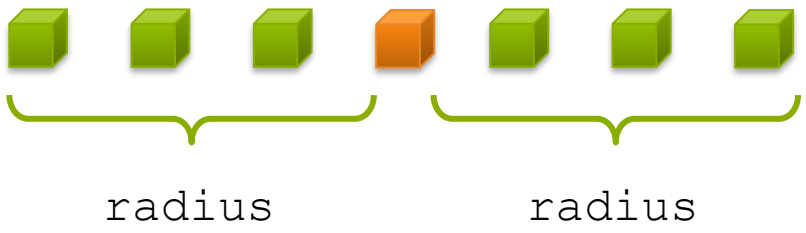
1. **Image Processing**: Applying convolution filters (e.g., Gaussian blur, edge detection) where each pixel is updated based on its neighbors.
2. **Scientific Simulations**: Modeling physical phenomena like heat diffusion, wave propagation, or fluid dynamics.
3. **Finite Difference Methods**: Solving partial differential equations (PDEs) using neighbor-based calculations.
4. **Computational Physics**: Updating grid cells in simulations like cellular automata or particle interaction grids.
5. **Weather and Climate Models**: Simulating environmental conditions using grid-based data, such as temperature or pressure updates.

- The stencil pattern is essential for tasks involving local interactions, with shared memory and efficient boundary handling playing key roles in optimizing performance.

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**

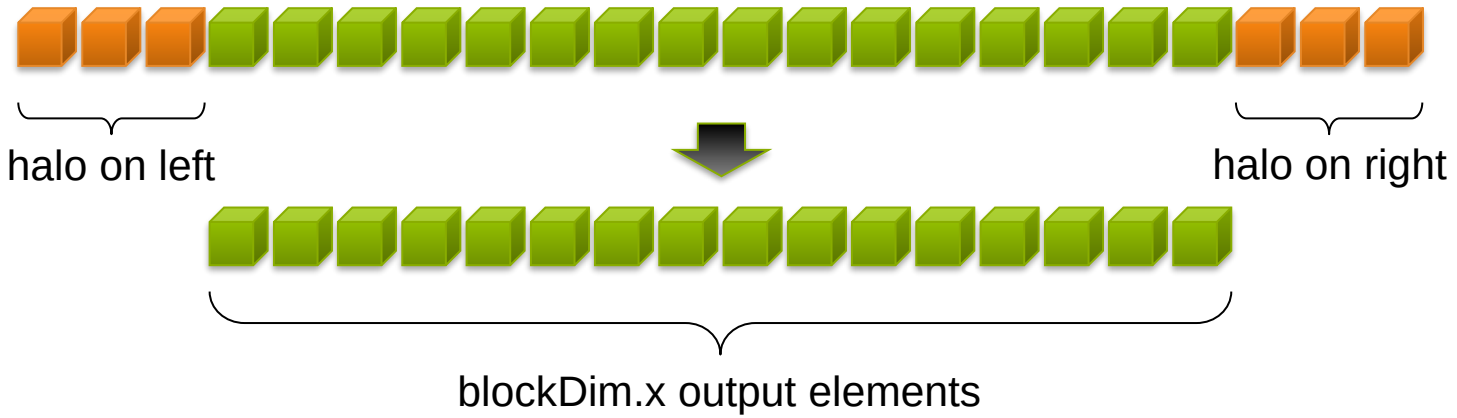
- Extremely fast on-chip memory, user-managed

- 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



Shared Memory

Shared memory is a small, fast memory space shared by all threads in a block. It is significantly faster than global memory because it resides on-chip (inside the GPU core). Shared memory allows threads within a block to collaborate by sharing 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.

Programming Assignment #2

- 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).

Where to experiment and learn?

- **Option 1**: If you have NVIDIA GPU on your laptop or PC at home
  - Do your experiment and exploration on it, it is faster

- **Option 2**: ITU's HPC cluster
  - You need to submit tasks to it, and you might also wait!

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 8 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                 # Run on either the Red or Brown queue

module load CUDA/12.1.1

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

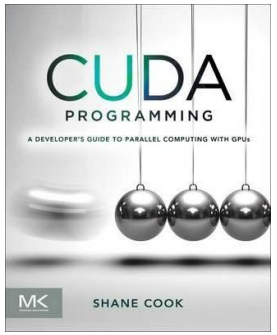
Programming Assignment #3

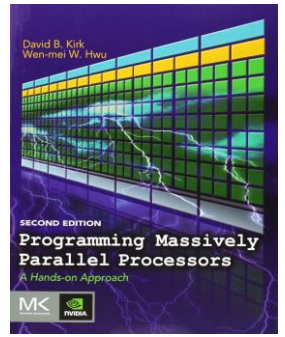
- Compute the dot product of two vectors using CUDA and compare it to the one available in the repository that uses shared memory (add the timing to both source codes). Experiment with different length of the input array, different block size.

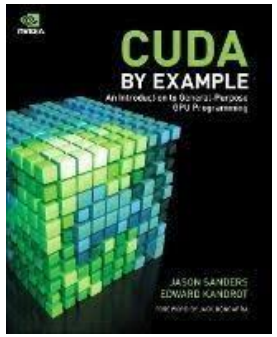
- Write your reflection on how using shared memory improves the performance.

ehsanyousefzadehasl/CUDA_for_ITU

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?