



Image Processing via Parallel Programming in CUDA


Dhruv Kohli

Mathematics & Computing Department



Topics

- GPU Programming Model
- CUDA Program Diagram
- Writing Your First CUDA Program
- Basic Parallel Communication Patterns
- Red Eye Removal



GPU Programming Model



Hardware Methods to decrease the running time

- Create a processor with faster clock
- Create a processor that can perform more work per clock cycle
- Parallel Computing

Difference between CPU and GPU

Traditional CPU

- Complex control hardware
- High flexibility and performance
- Expensive in terms of power

GPUs

- Simpler control hardware
- More hardware for computation
- Restricted Programming Model

Challenge

- **CPU Programming** : Quite Simple
- **GPU Programming** : More Complex and Mathematical



How is GPU faster than CPU?

Two Important terms

Latency -

- Amount of time needed to complete a job
- hour/ min/ sec

Throughput -

- No. of jobs completed per unit time
- $(\text{Jobs Completed}) / (\text{hour/ min/ sec})$

Example

A-----4500 Km-----B

- **Car** : 2 People, 200 km/hr
- **Bus** : 40 People, 50 km/hr

Example contd..

Latency = distance / speed

- Latency(car) = $4500 / 200 = 22.5$ Hours
- Latency(Bus) = $4500 / 50 = 90$ Hours
- Throughput(Car) = $4 / 45$ people per hour
- Throughput(Bus) = $4 / 9$ people per hour

Example contd..

Group : 40 People


- Time taken by Car = No. Of People / Throughput(Car) = 450 Hours
- Time Taken by Bus = No. Of People / Throughput(Bus) = 90 Hours

Car takes 5 times more time than Bus.

- Car == CPU
- Bus == GPU

Fact

- Modern GPU Designers optimize for throughput NOT latency



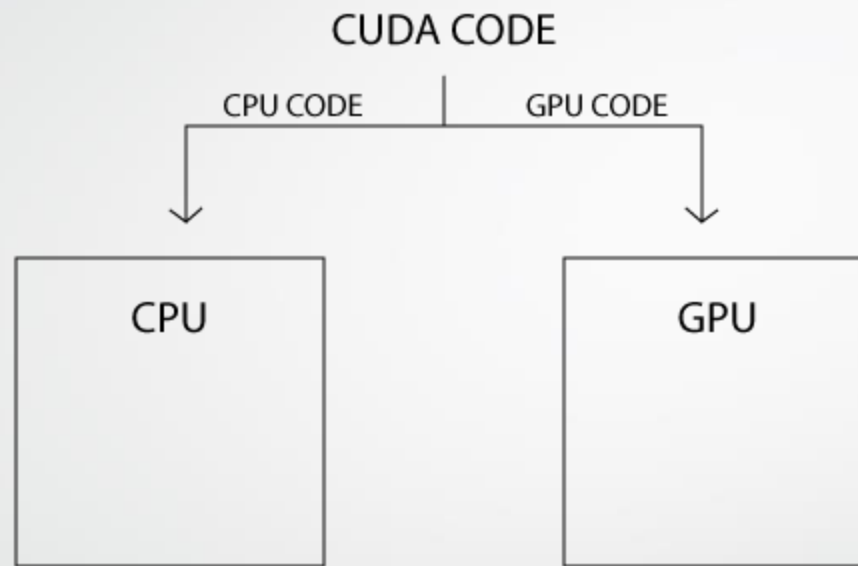
CUDA Program Diagram

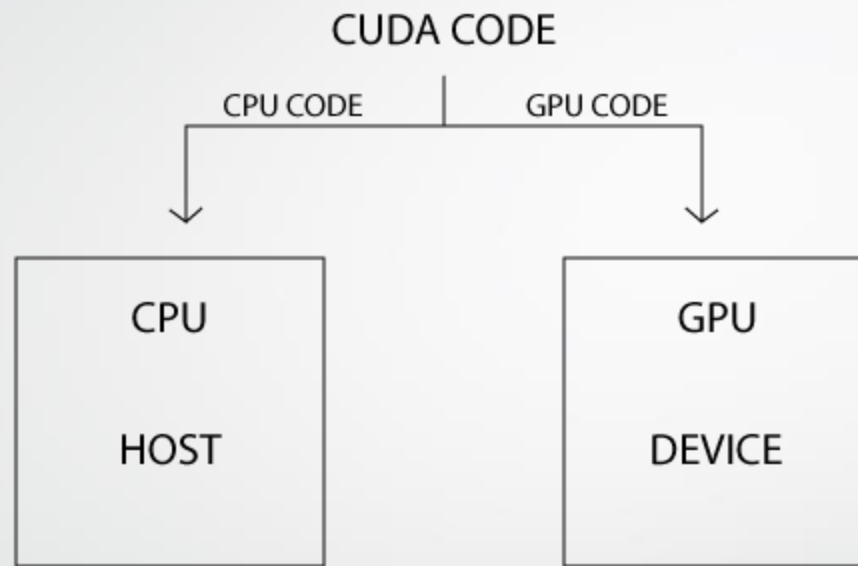


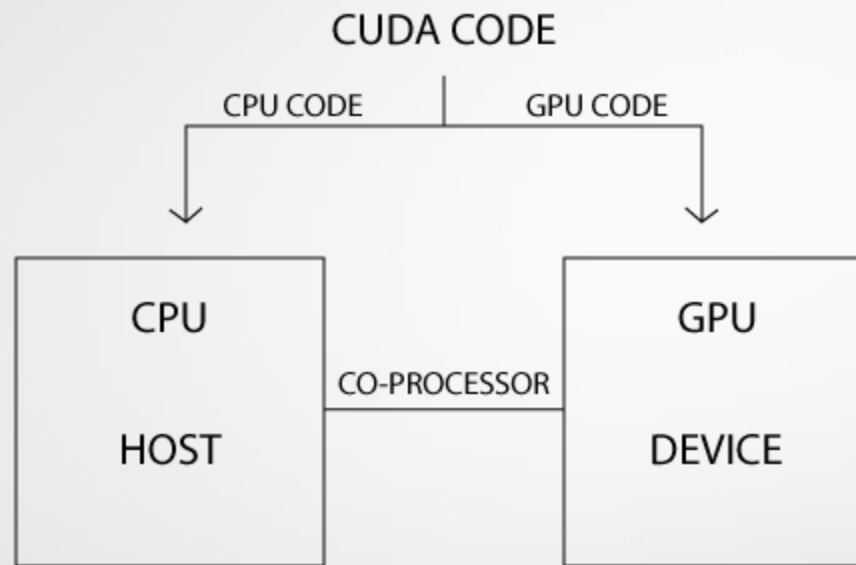
CUDA CODE

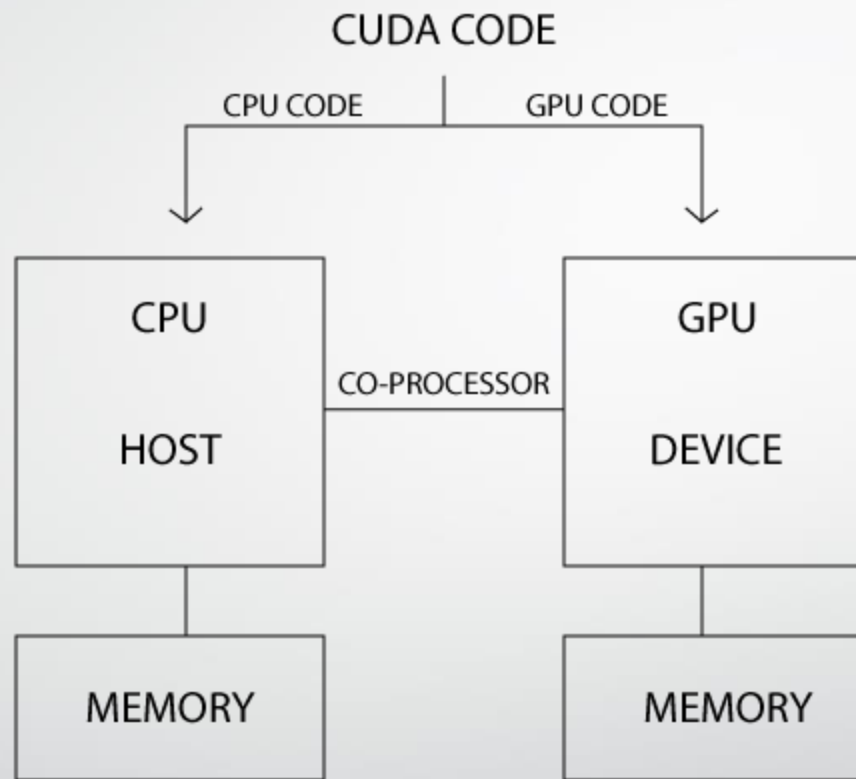
CPU

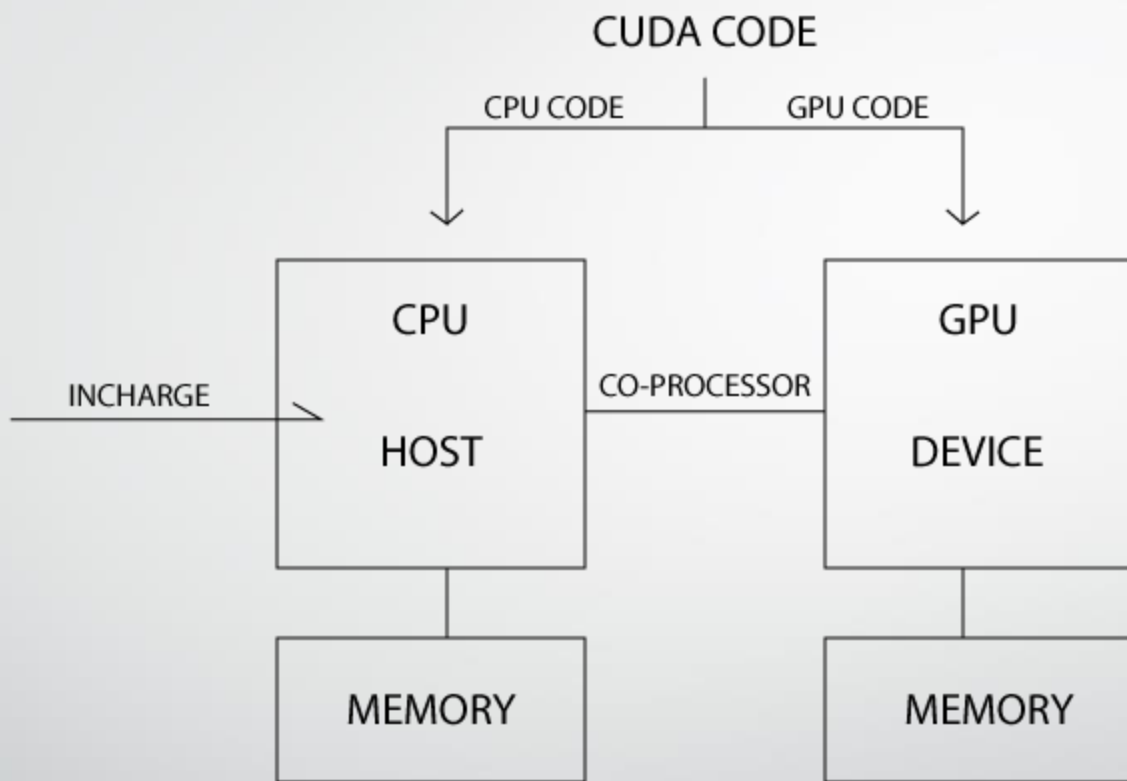
GPU

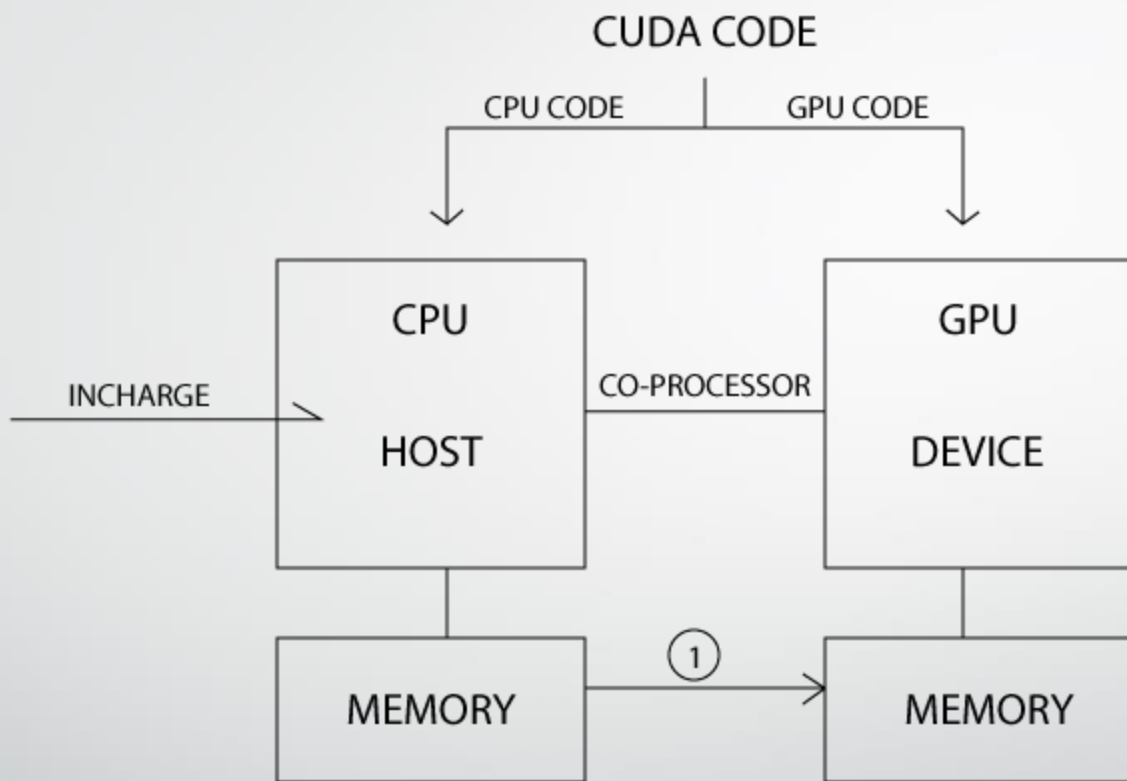


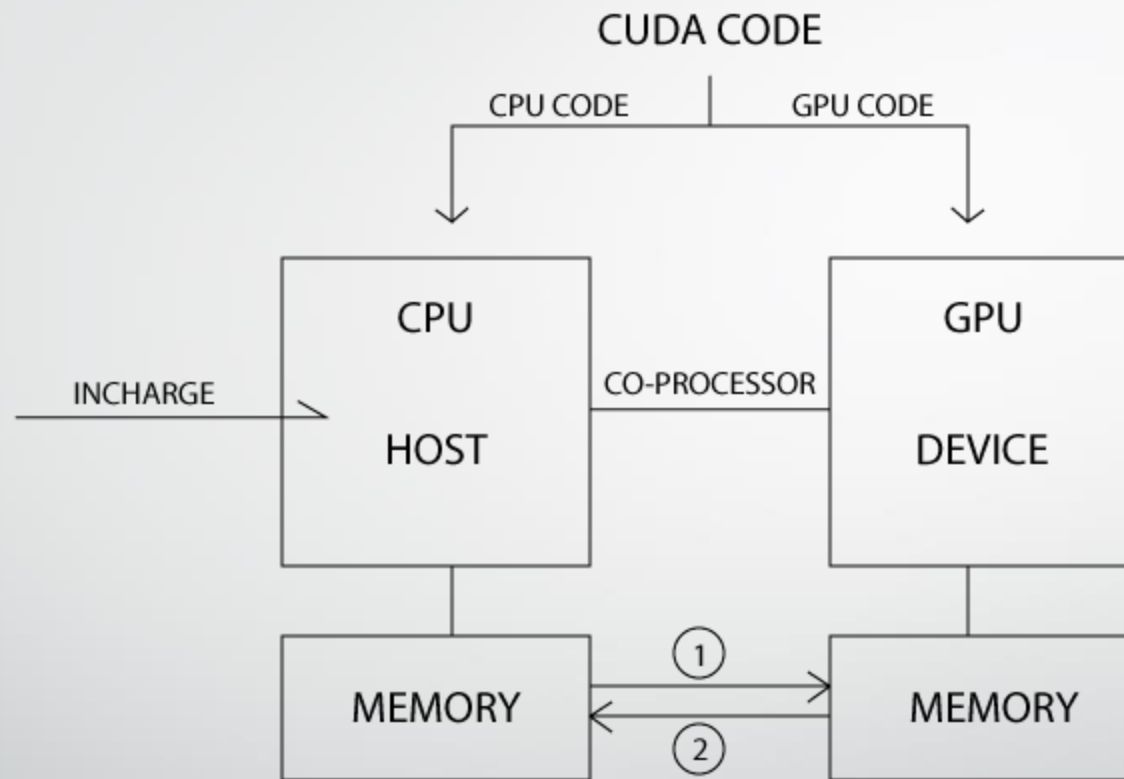


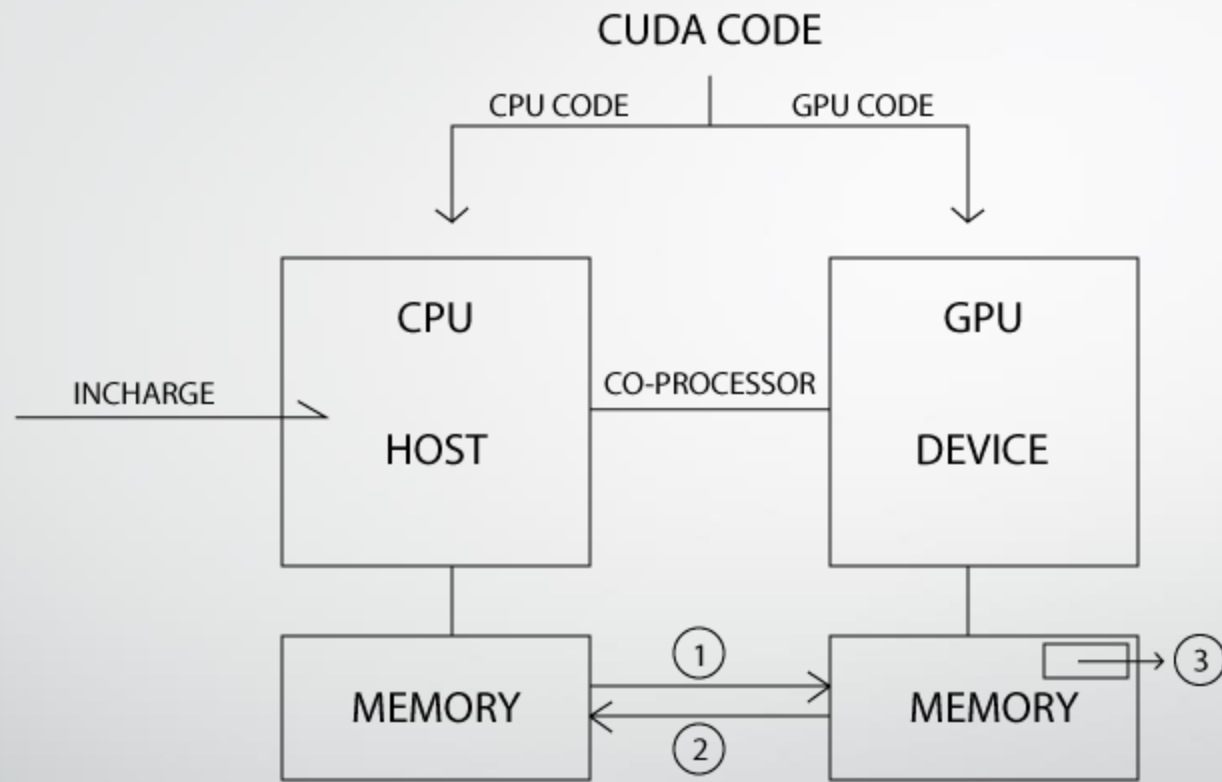


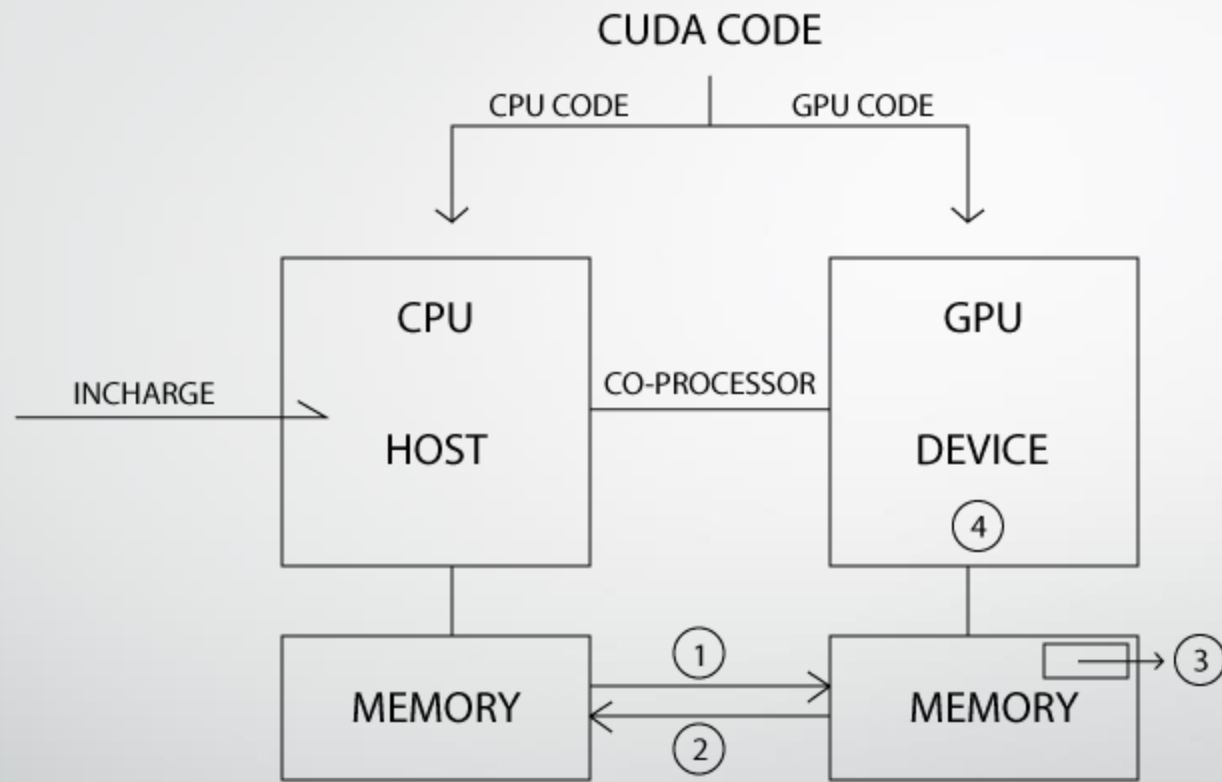














Writing Our First CUDA Program

Structure of a typical GPU program

- CPU allocates memory on GPU using `cudaMalloc`
- CPU copies input data from CPU to GPU in this allocated storage using `cudaMemcpy`
- CPU Launches the kernel on GPU which processes the data
- CPU copies back the processed data from GPU to CPU using `cudaMemcpy` again



Lot of Moving of data...

- Is this **expensive** in terms of time? Any thoughts?

BIG IDEA

- Kernel looks like a serial program
- Kernel program runs on a single thread
- What is thread ?
- An Independent path of execution
- GPU runs the kernel program on multiple threads SIMULTANEOUSLY.
- No. of the threads are defined by the user.

GPU good at..

- Launching lots of threads
- Running those threads in parallel

Example / Problem Statement

- Input - float in[] = {0, 1, , n-1}
- Output - float out[] = {0*0, 1*1,, (n-1)*(n-1)}

Example contd..

Sequential code:

- ```
for(i = 0; i < n; ++i){
 out[i] = in[i] * in[i]
```
- ```
}
```

Two things to note :

- Single thread
- No Explicit Parallelism

High level view of GPU Code

- CPU allocates memory on GPU
- CPU copies Data to / from GPU
- CPU launches the kernel on GPU which specifies the degree of parallelism
- GPU CODE contains the kernel
- Kernel program is a serial program made for a single thread
- Need to express $OUT = IN * IN$ in kernel code

Final note before our first CUDA program

```
square<<< 1, ARRAY_SIZE >>>(d_in, d_out)
```

- The Final NOTE is :
- Each thread knows which thread it is
- NOTE : Zero indexing.



LETS SEE OUR FIRST CUDA PROGRAM..

Important points in our CUDA program

`<<< gridSize, blockSize >>>`

gridSize

- `dim3 C` construct
- `x`, `y` and `z` as members
- `blockIdx.x`, `.y`, & `.z` AND `gridDim.x`, `.y` & `.z`

(Note that these are two different things, and we'll see what they are in a minute...)

blockSize

- `dim3 C` construct
- `x`, `y` and `z` as members
- `threadIdx.x`, `.y`, & `.z` AND `blockDim.x`, `.y` & `.z`

Important points contd..

Third OPTIONAL argument to Launch operator :

- Shared Memory (in the next section)

Max number of threads per block :

- 512 (Older GPUs)
- 1024 (Modern GPUs)

Max number of blocks per dimension = 65535

Important points contd..

```
<<< dim3(10, 10, 1), dim3(5, 5, 1) >>>
```

- Each thread knows : gridDim, blockDim, threadIdx, blockIdx .x, .y, .z



Basic Parallel Communication Patterns

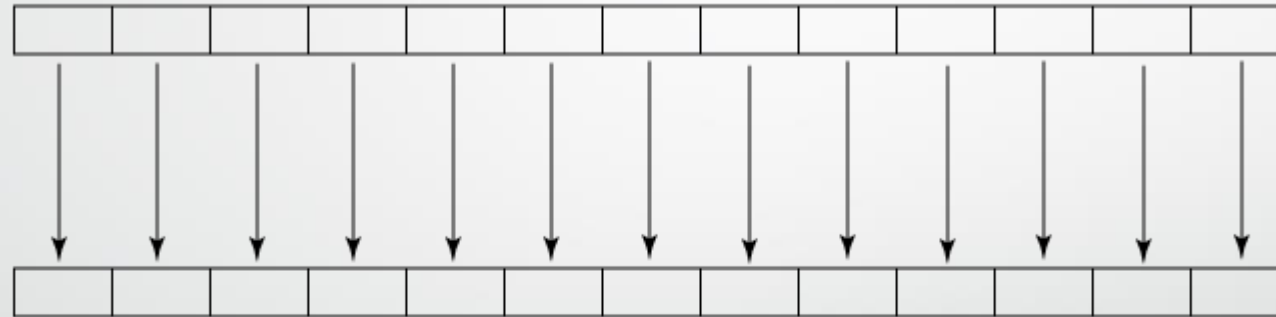
Introduction

- Threads solve problem by working together
- This working together of threads is called “Communication”

Map

- Reading from specific data elements and writing to specific data elements
- Example : Our first CUDA program, Color to Grayscale conversion
- There is a one to one correspondence between input and output elements

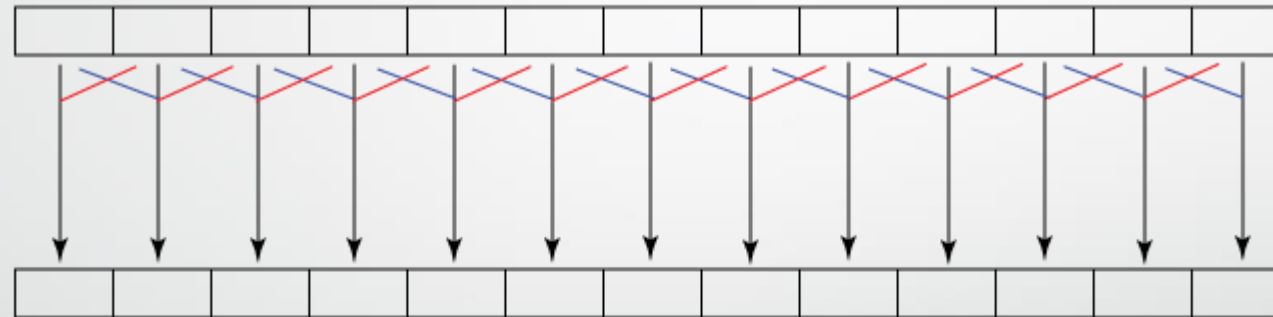
Map pattern diagram



Gather

- Reading from scattered data elements and writing to adjacent places
- Ex : Averaging set of 3 elements together and writing result to output array
- There is a many to one correspondence between input and output elements

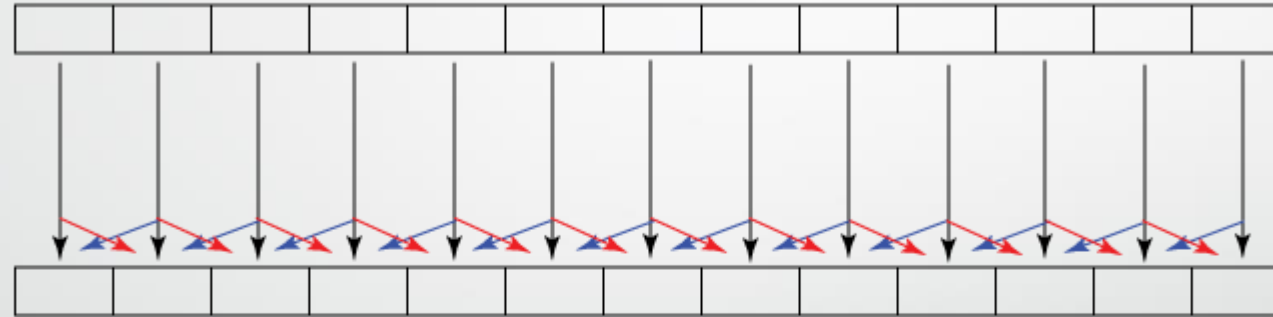
Gather pattern diagram



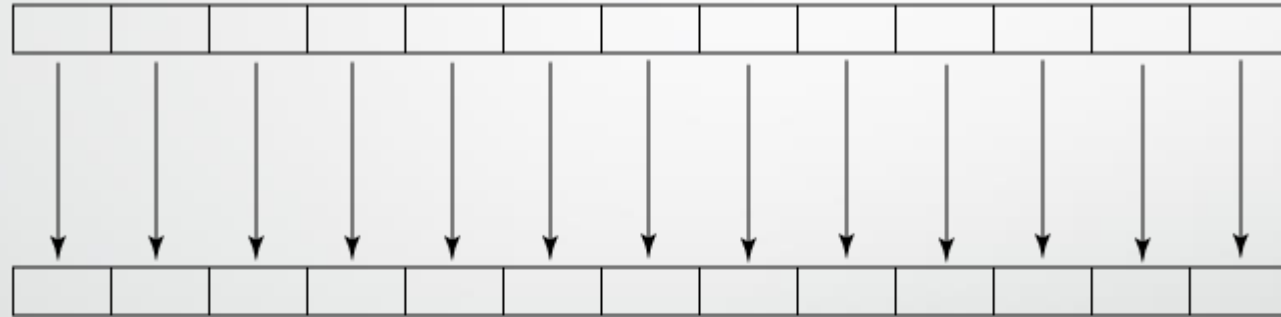
Scatter

- Reading from adjacent data elements and writing to scattered places
- Ex : Averaging set of 3 elements together and writing result to output array
- There is one to many correspondence between input and output elements
- Conflict...

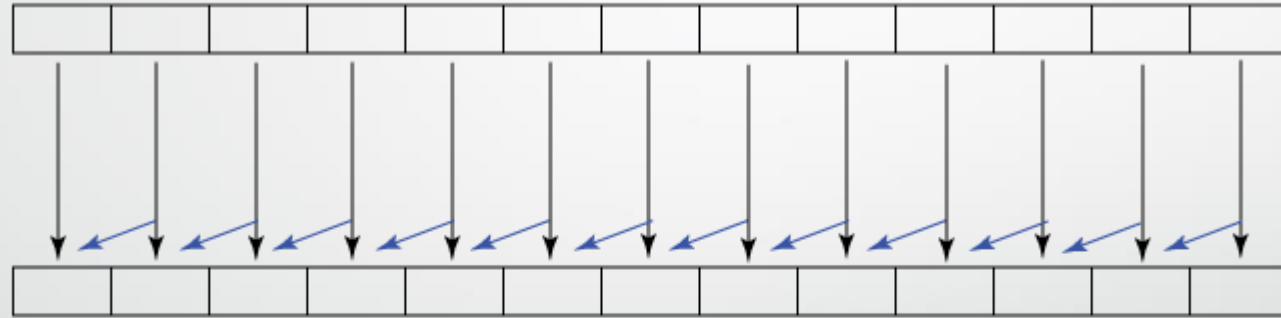
Scatter pattern diagram



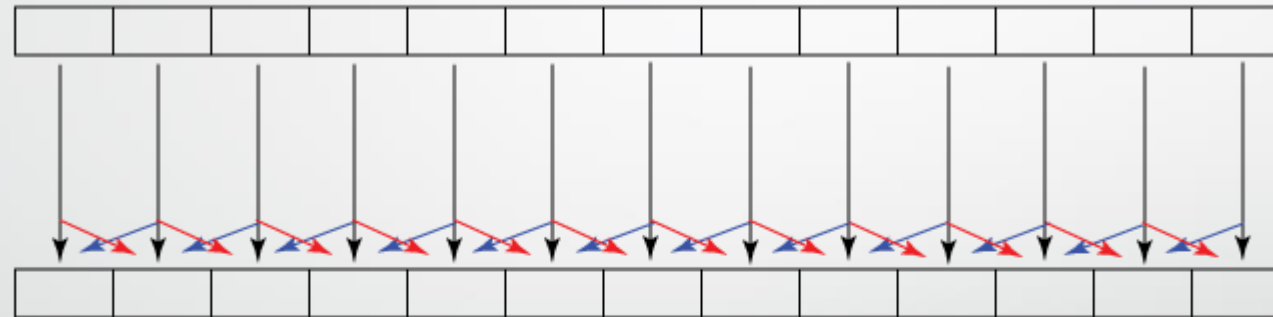
Scatter till first barrier



Scatter till second barrier



Scatter till third barrier



Stencil

- Reading from fixed neighborhood of a data element and writing to specific data element
- Image processing example : Image blurring
- There is a several to one correspondence between input and output elements

Stencil Pattern diagram

- Like gather



Von Neumann Stencils

Reduce

Takes two inputs :

- Set of elements
- Reduction operator : Binary Associative operator
- Ex : Sum the elements of an array

Optimizing parallel reduce

- Parallel reduce can be optimized using shared memory
- Shared memory can be accessed on kernel by using
 - `extern __shared__ dataType temp[]`
 - `temp[]` is local to a block and has no significance outside the scope of a block, hence we can declare shared memory variable in a block with same name
 - Optimization due to difference in speed of access...



Reduce pattern diagram

Scan

Takes three inputs :

- Set of elements
- Binary associative operator
- Identity element (I)
- INPUT = $[a_0, a_1, a_2, \dots, a_{(n-1)}]$
- Exclusive scan OUTPUT = $[I, a_0, a_0 \text{ op } a_1, \dots, a_0 \text{ op } a_1 \text{ op } \dots a_{(n-2)}]$
- Inclusive scan OUTPUT = $[a_0, a_0 \text{ op } a_1, \dots, a_0 \text{ op } a_1 \text{ op } \dots a_{(n-1)}]$

Belloch Algorithm For Exclusive Scan

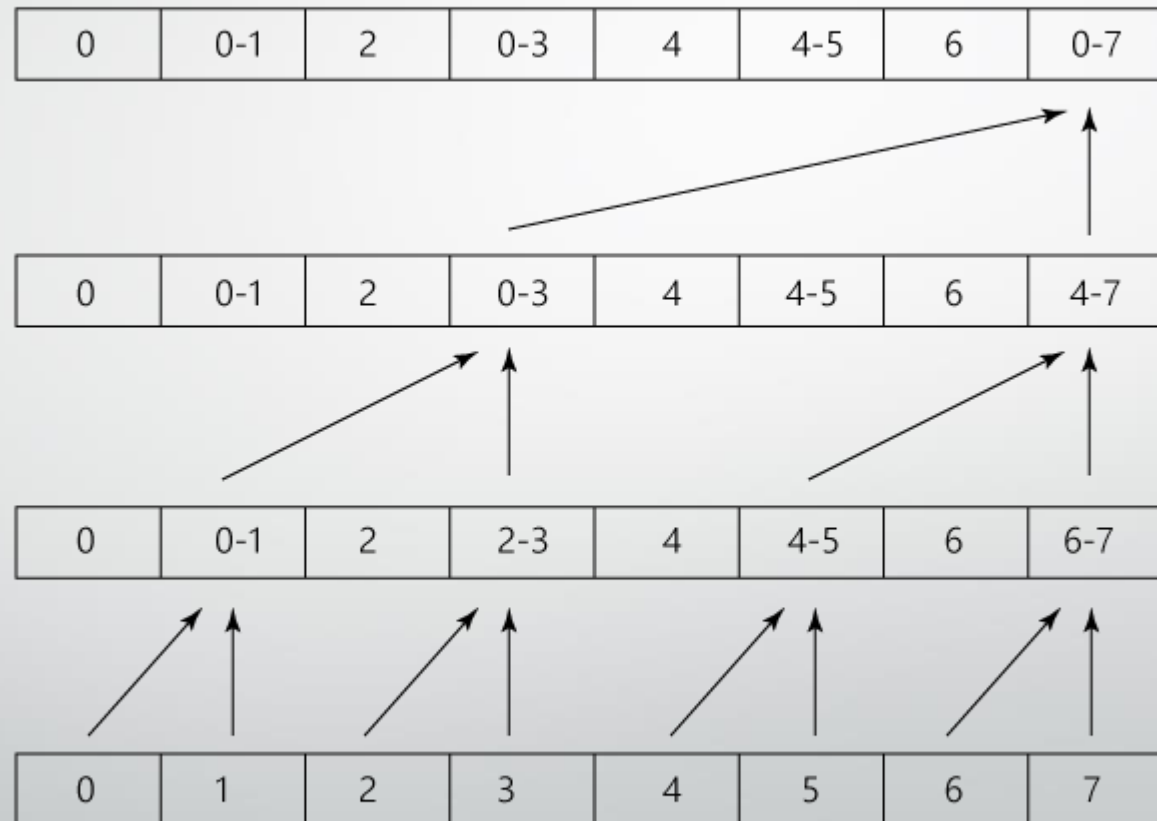
Two phases :

- Upsweep / Reduce phase
- Downsweep phase

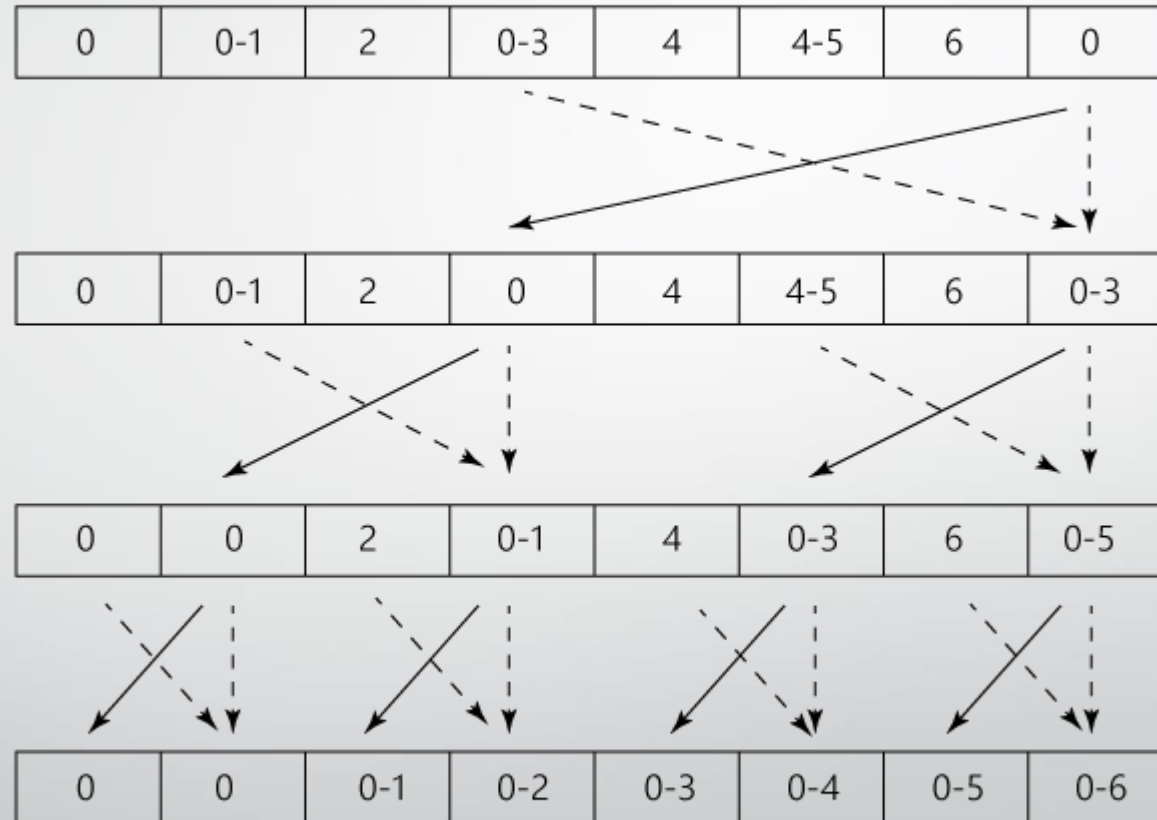
Phases


- Upsweep phase
- Traverse from leaves to root calculating the partial sums at internal nodes
- Downsweep phase
- First, insert ZERO at the end of the array
- Traverse from root to leaves by following this procedure :
 - Replace left child value of node with the value of the node
 - Replace right child value of the node with the sum of the value of the node and the former value of its left child

Upsweep phase



Downsweep Phase





Scan pattern diagram



Red Eye Removal

Note

- Once you have implemented the communication patterns and algorithms and a few more algorithms in parallel manually, then you should shift to using STL of CUDA called THRUST.
- Thrust library contains inbuilt functions to perform all types of communication patterns and basic arithmetic in parallel in an optimized manner.

Steps for Red Eye Removal

- Preprocessing
- Sorting
- Post processing

Preprocessing

Involves

- Loading RGBA image
- Splitting Channels
- Performing Normalized Cross Correlation
- Getting scores corresponding to each pixel

Sorting

Involves

- Using key sort to sort scores in an ascending order
- Key sort means :
 - As the values are sorted an array containing the initial positions of the values also gets updated.

Post processing

Involves

- Removing redevye from the image by eliminating the redness from the pixels which have highest scores
- Then Combining channels
- Finally saving image



Thank You...