Parallel Programming

Introduction to CUDA C/C++

Part II

Phạm Trọng Nghĩa
ptnghia@fit.hcmus.edu.vn

Review: previous lecture

Structure of a simple CUDA program

- Sequential parts will run on host (CPU), massively parallel parts will run on device (GPU)
- From host, to ask device to compute in parallel:
  - Host allocates memory regions on device by calling **`cudaMalloc`**
  - Host copies necessary data to memory regions on device by calling **`cudaMemcpy`**
  - Host calls kernel function
  - Host copies results from device by calling **`cudaMemcpy`**
  - Host frees memory regions on device by calling **`cudaFree`**

Review: previous le

Structure of a simple CUDA prog

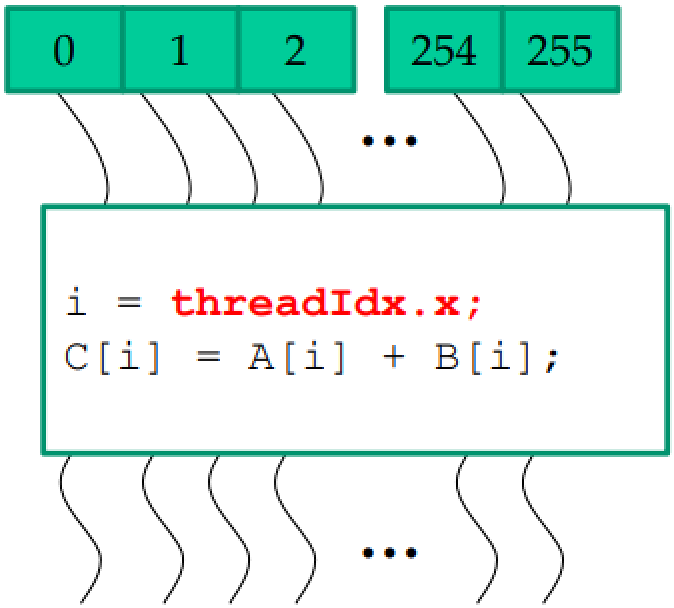
- Sequential parts will run on host (parts will run on device (GPU)
- From host, to ask device to comp
    - Host allocates memory regions on c
    - Host copies necessary data to mem **`cudaMemcpy`**
    - Host calls kernel function
    - Host copies results from device b
    - Host frees memory regions on devi

- Kernel function is executed in parallel by many threads on device
- These threads are organized as follows:
    - The grid consists of blocks (these blocks have the same size)
    - Each block consists of threads

  When host calls kernel function, host needs to specify grid size and block size
- On device, in kernel function: each thread can use built-in variables (**`blockIdx`**, **`threadIdx`**, **`blockDim`**, **`gridDim`**) to identify its data portion

Kernel Executes as Grid

- CUDA kernel executes as a grid (array) of threads.
- All threads in grid
  - Run the same kernel code; called a
  - Single Program Multiple Data (SPMD model).
- Each thread has unique index used to
  - Compute memory addresses and
  - Make control decisions.

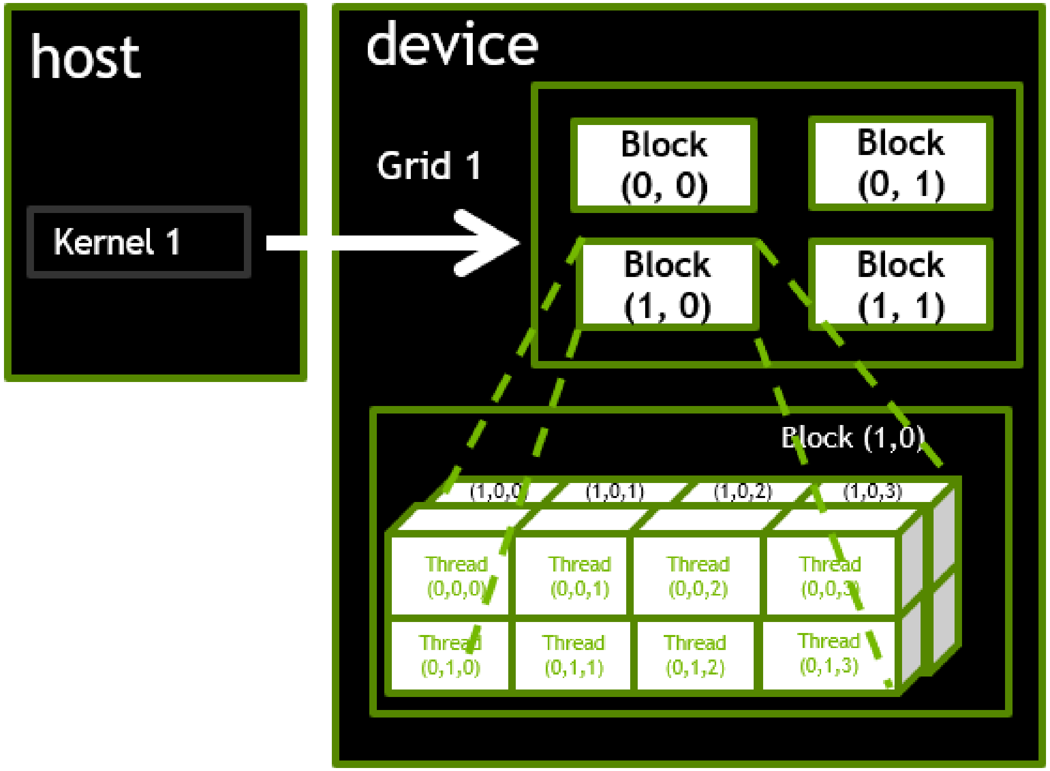


Today: working with 2D data

- Last lecture:
  - Wrote a kernel function with one-dimensional grid
- This lecture:
  - More generally at *how threads are organized* and learn how threads and blocks can be used to *process multidimensional arrays*
- Multiple examples:
  - Adding 2 matrixes
  - Converting a RGB image to grayscale (intro)
  - Blurring an image (intro)

Multidimensional grid organization

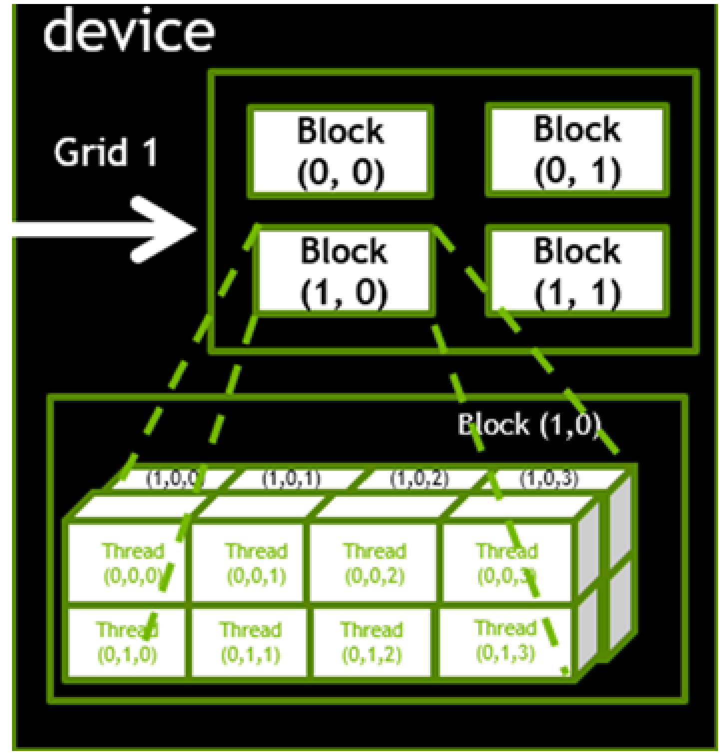
- In general, a grid is a 3D array of blocks, and each block is a 3D array of threads



- Provides a natural way to invoke computation across the elements in a domain such as a *vector*, *matrix*, or *volume*

Kernel Launch Configuration

- Determines how threads are organized and executed in parallel
- Definition of a grid
  - dim3 gridDim(2, 2, 1);
  - dim3 blockDim(4, 2, 2);
- KernelFunction<<< gridDim, blockDim >>>(...);
  - gridDim: the number of blocks in the grid.
  - blockDim: the number of threads in each block

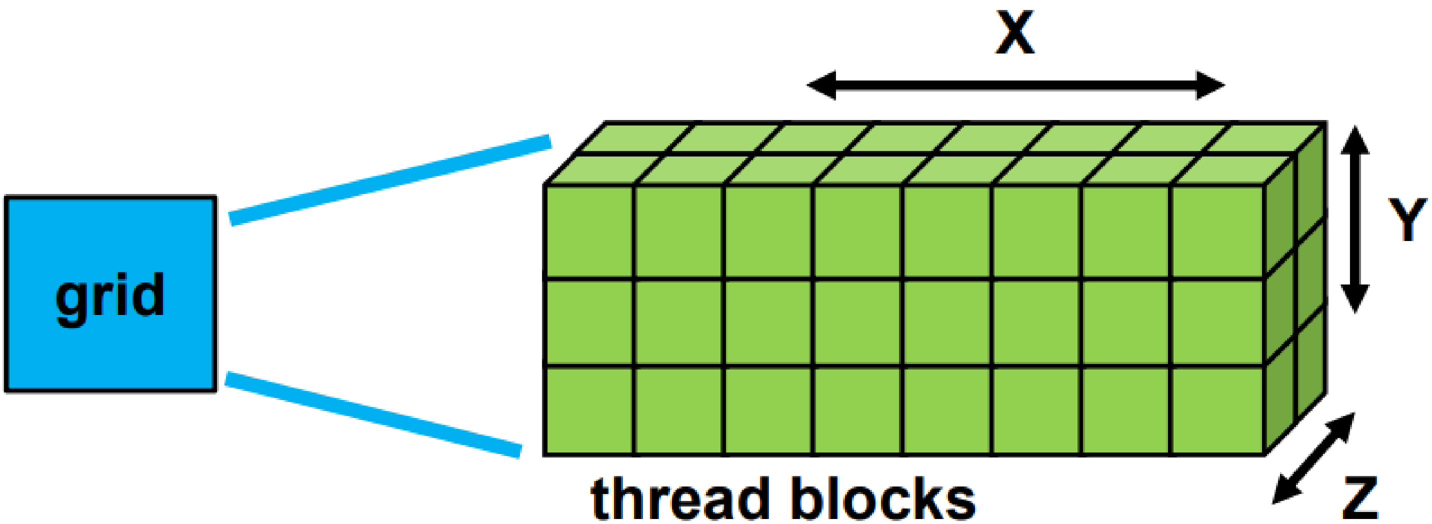


Thread Indexing

- Within a kernel, each thread can determine its unique index using ***built-in variables***.
- Build-in variables
  - **threadIdx**: the index of a thread in a block
    - threadIdx.x, threadIdx.y, and threadIdx.z
  - **blockIdx**: the index of a block in the grid
    - blockIdx.x, blockIdx.y, and blockIdx.z
  - **blockDim**: the dimensions of the block
    - blockDim.x, blockDim.y, and blockDim.z
  - **girdDim**: the dimensions of the grid
    - gridDim.x, gridDim.y, and gridDim.z

Thread Indexing

- Build-in variables: **threadIdx, blockIdx, blockDim,girdDim**



- ThreadIdx tuple is unique to each thread **WITHIN A BLOCK**.

Multidimensional grid organization

- Typical host kernel call:

```
dim3 blockSize(16, 16, 2);
dim3 gridSize(128, 1, 1);
addVec<<<gridSize, blockSize >>> (...);
```

```
dim3 blockSize(16);
dim3 gridSize(128);
addVec<<<gridSize, blockSize >>> (...);
```

```
dim3 blockSize(16);
dim3 gridSize(128);
addVec<<<128, 16 >>> (...);
```

- For 2D (and 1D blocks), simply use block dimension 1 for Z (and Y).

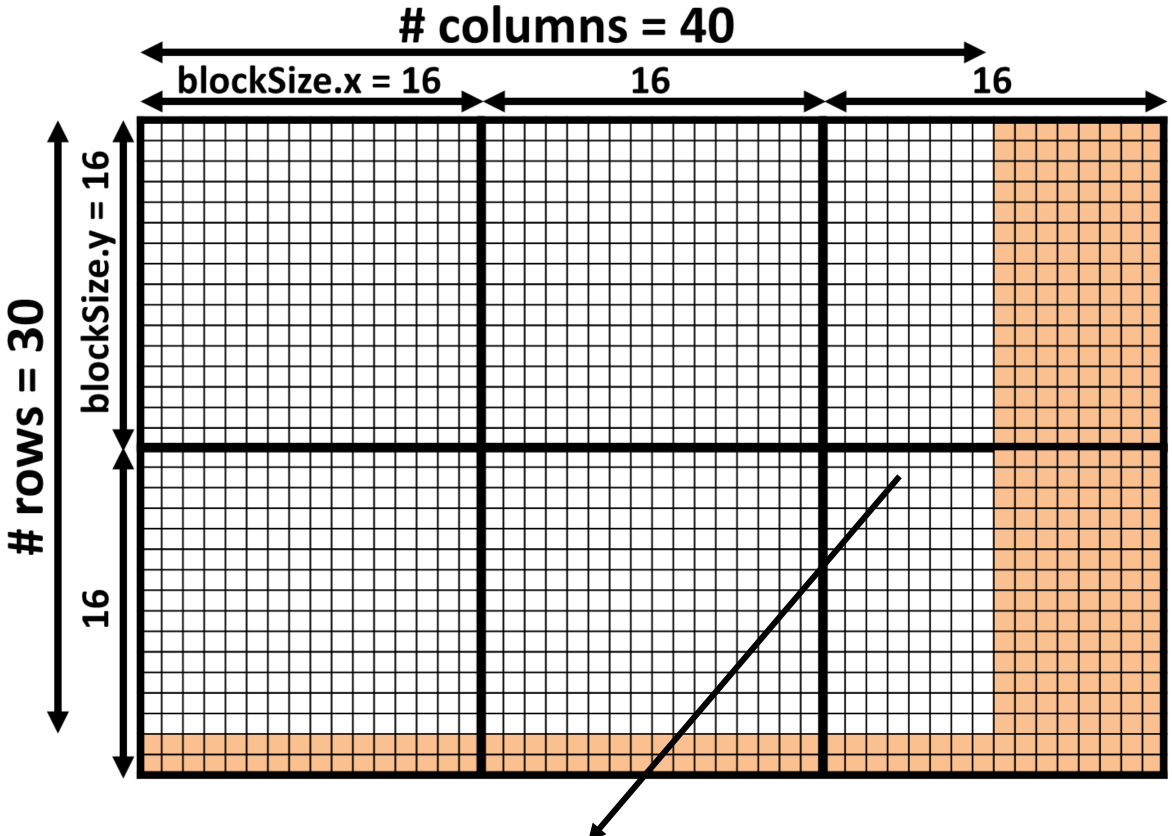
Dimension limit

```
Max gridDim.x: 2147483647
Max gridDim.y: 65535
Max gridDim.z: 65535
Max blockDim.x: 1024
Max blockDim.y: 1024
Max blockDim.z: 64
Max thread per block: 1024
```

- (512, 1, 1) ✓
- (8, 16, 4) ✓
- (32, 16, 2) ✓
- (32, 32, 2) ✘

Adding 2 matrixes

- **Sequential implementation:** easy ☺
- **Parallel implementation:** not difficult ☺
  - Let each thread compute one element in the output matrix
  - With 2D data, it's natural to organize threads into 2D blocks and 2D grid



blockIdx.x = 2   threadIdx.x = 3
blockIdx.y = 1   threadIdx.y = 1

r = blockIdx.y*blockDim.y + threadIdx.y
= 1*16 + 1
= 17

c = blockIdx.x*blockDim.x + threadIdx.x
= 2*16 + 3
= 35

Adding 2 matrixes

- **Sequential implementation:** easy ☺
- **Parallel implementation:** not difficult ☺
  - Let each thread compute one element in the output matrix
  - With 2D data, it's natural to organize threads into 2D blocks and 2D grid

- One final consideration before we code: how should we store a matrix?
  - Store as 2D array (a double pointer)

    It will cause a lot of troubles (you can try and see …)
  - Store as 1D array (a single pointer) by concatenating rows together

    It will make life easier

Row-major order

A[0][0]	A[0][1]	A[0][2]	A[0][3]
A[1][0]	A[1][1]	A[1][2]	A[1][3]
A[2][0]	A[2][1]	A[2][2]	A[2][3]

A[0][0]	A[0][1]	A[0][2]	A[0][3]	A[1][0]	A[1][1]	A[1][2]	A[1][3]	A[2][0]	A[2][1]	A[2][2]	A[2][3]
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	A[10]	A[11]

$$i = r * \#columns + c$$

When we implement functions operating on 2D array:

- We can view it directly as 1D array, but this approach doesn't work in all cases
- A more general approach is to view it as 2D array, compute row and column indexes, and when we need to access, convert row and column indexes to indexes in 1D array (we will use this approach)

Adding 2 matrixes: live coding

Today: working with 2D data

- Adding 2 matrixes
- Converting a RGB image to grayscale (intro)
- Blurring an image (intro)

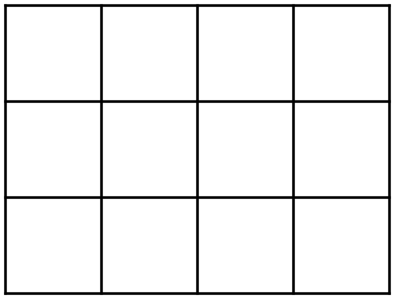
Converting a RGB image to grayscale

Input

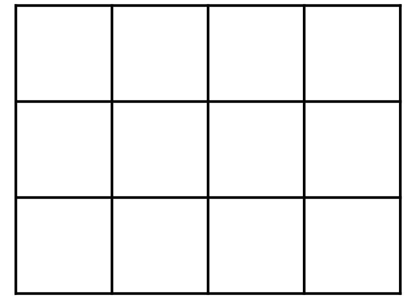
Output

Converting a RGB image to grayscale

Input



Output



Each pixel is represented by 3 numbers:

Each pixel is represented by 1 number:

red, green, blue ⟶ gray

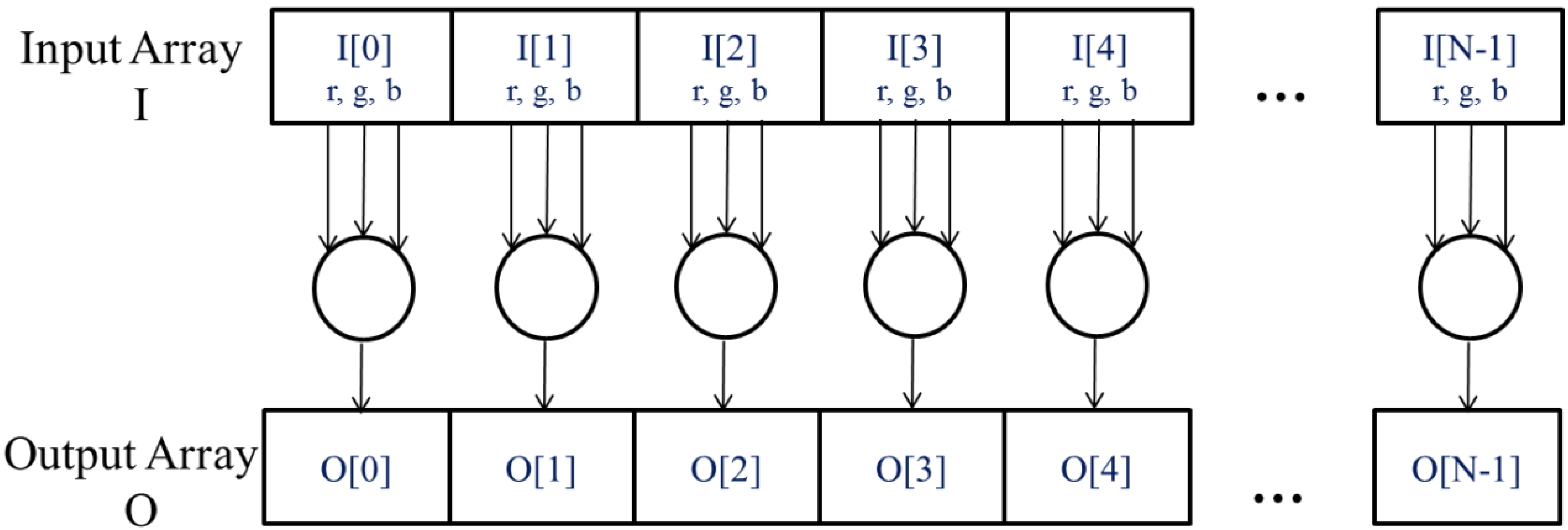
gray = 0.299*red + 0.587*green + 0.114*blue

Each number is an unsigned integer from 0-255

This number is an unsigned integer from 0-255

Converting a RGB image to grayscale

- The pixels can be calculated independently of each other



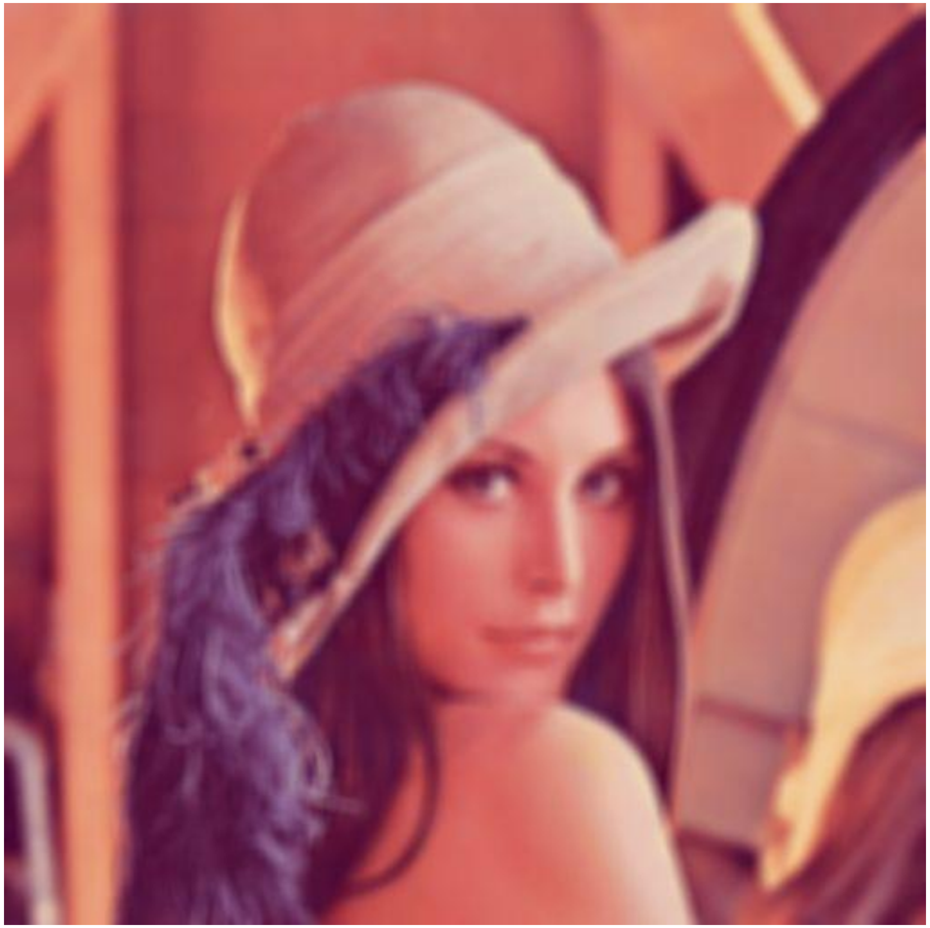
Today: working with 2D data

- Adding 2 matrixes
- Converting a RGB image to grayscale (intro)
- Blurring an image (intro)
- CUDA functions to query device info

Blurring an image

Input

Output



Blurring an image

To blur an RGB image, we blur each channel separately

→We just need to understand how to blur one channel

How to blur one channel?

Convolve it with a special filter to make adjacent pixels similar (details in the upcoming homework)

Today: working with 2D data

- Adding 2 matrixes
- Converting a RGB image to grayscale (intro)
- Blurring an image (intro)

- CUDA functions to query device info

Querying device info

In a CUDA program, we may want to query device info:

- For example, we want to query the max number of threads / block and the max number of blocks / grid on device and set block size and grid size appropriate for this device → when device is changed, code is unchanged
- Or we want to query info of all available devices in our machine and select the most powerful device to run the program
- Or we simply want to query device info and print to the screen

Querying device info

Query the number of devices

```
int devCount;
cudaGetDeviceCount(&devCount);
```

Select a device to use (in case there are many devices), e.g. device 0

```
cudaSetDevice(0);
```

Query info of a device, e.g. device 0

```
cudaDeviceProp devProp;
cudaGetDeviceProperties(&devProp, 0)
```

Check [here]() for fields of cudaDeviceProp struct. E.g., devProp.maxThreadsPerBlock indicates the max num of threads / block

Colab device's query results (now)

Device 0: Tesla T4
Compute capability: 7.5
GMEM: 15 GB
Max number of threads per block: 1024
Max size of each dimension of a block: 1024 x 1024 x 64
Max size of each dimension of a grid: 2147483647 x 65535 x 65535
…

Reference

- [1] Wen-Mei, W. Hwu, David B. Kirk, and Izzat El Hajj. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2022

THE END