CUDA Programming Model (Part 2)

ECE 277

Cheolhong An

Heterogeneous Computing

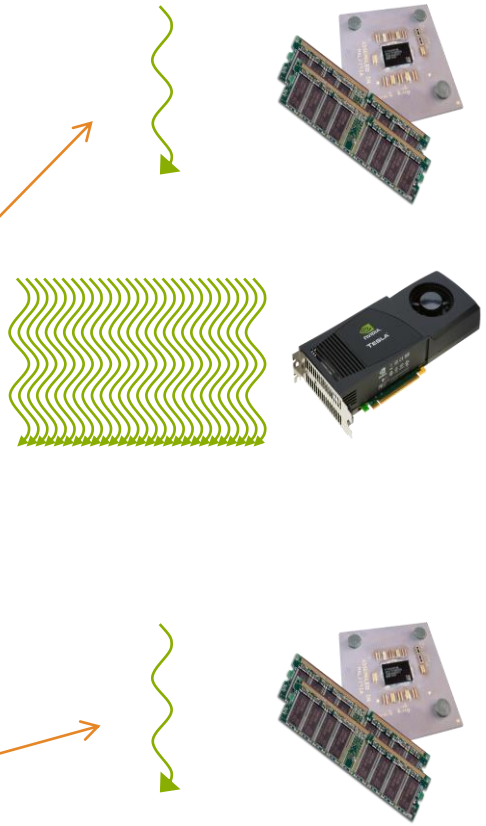
```
#include <iostream>
#include <algorithm>

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

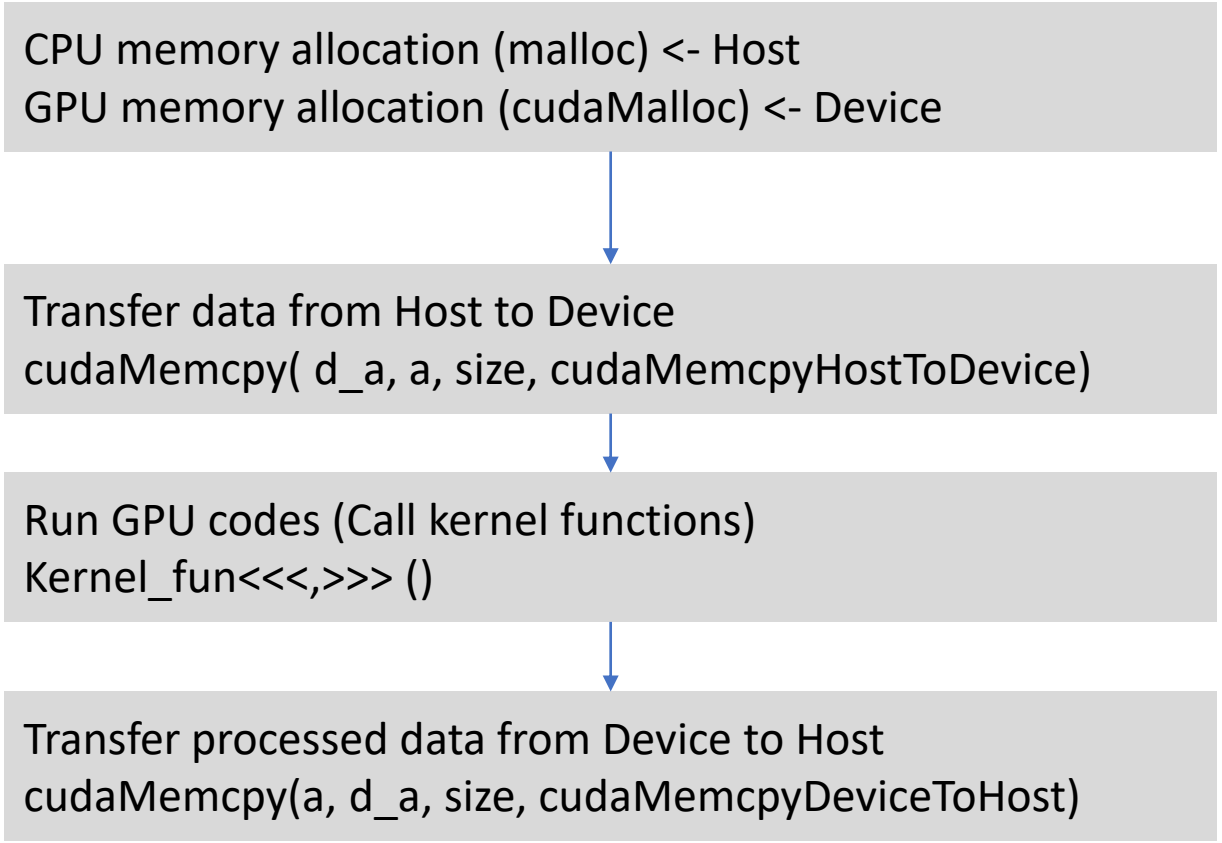
serial code



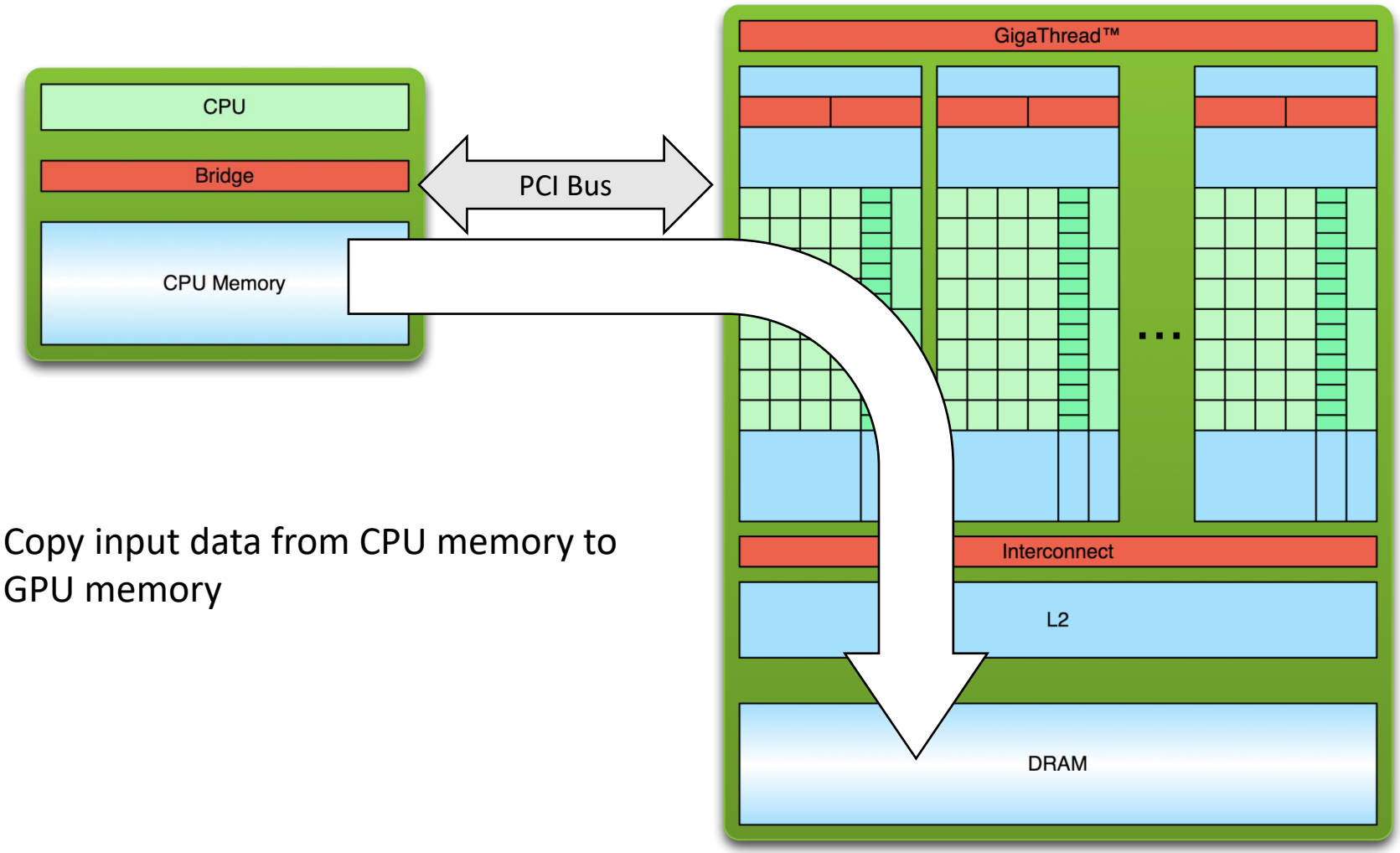
Mark Harris, **Introduction to CUDA C,** NVIDIA Corporation

Common CUDA code structure

- Ex) checkThreadIndex

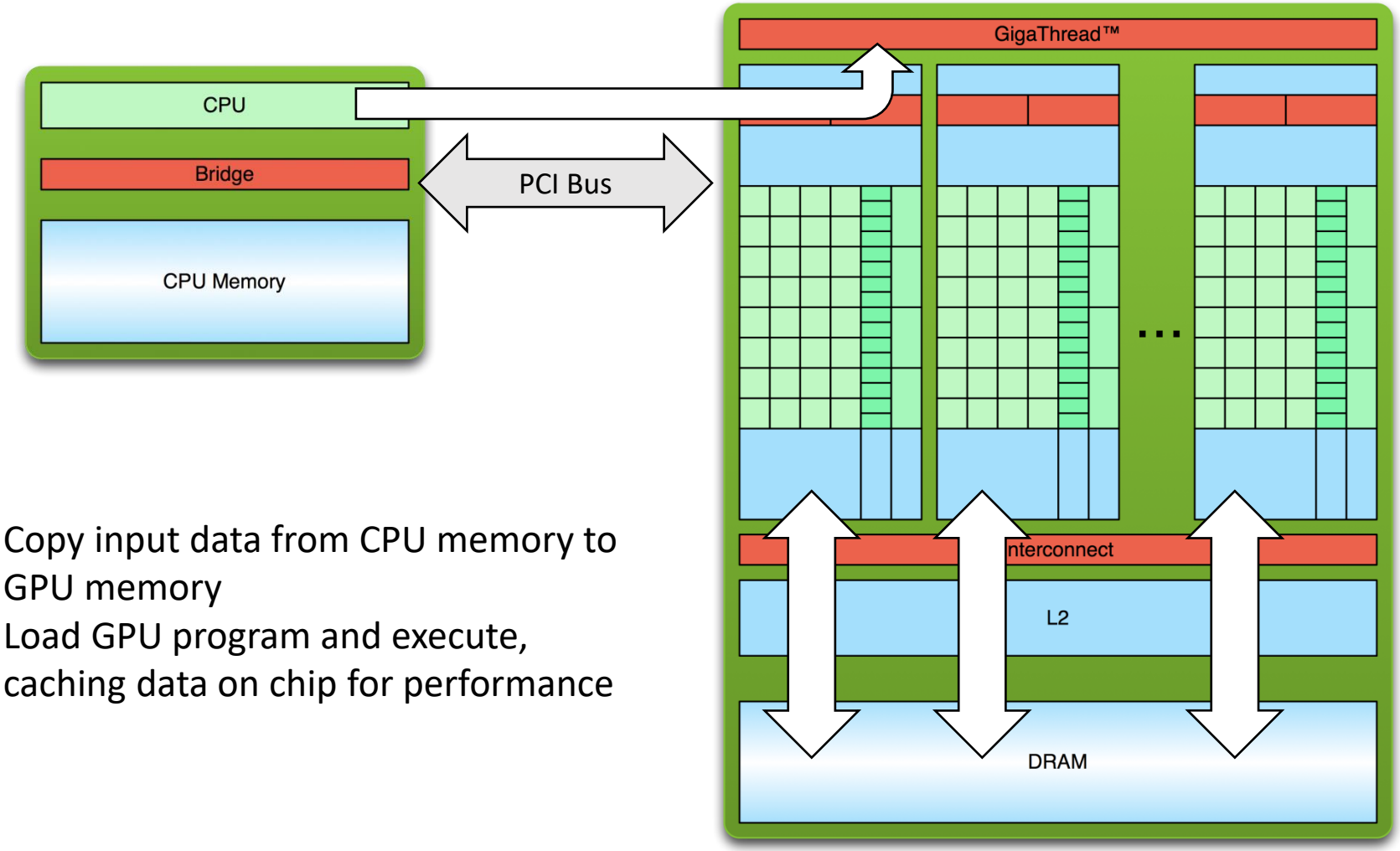


Transfer data from Host to Device



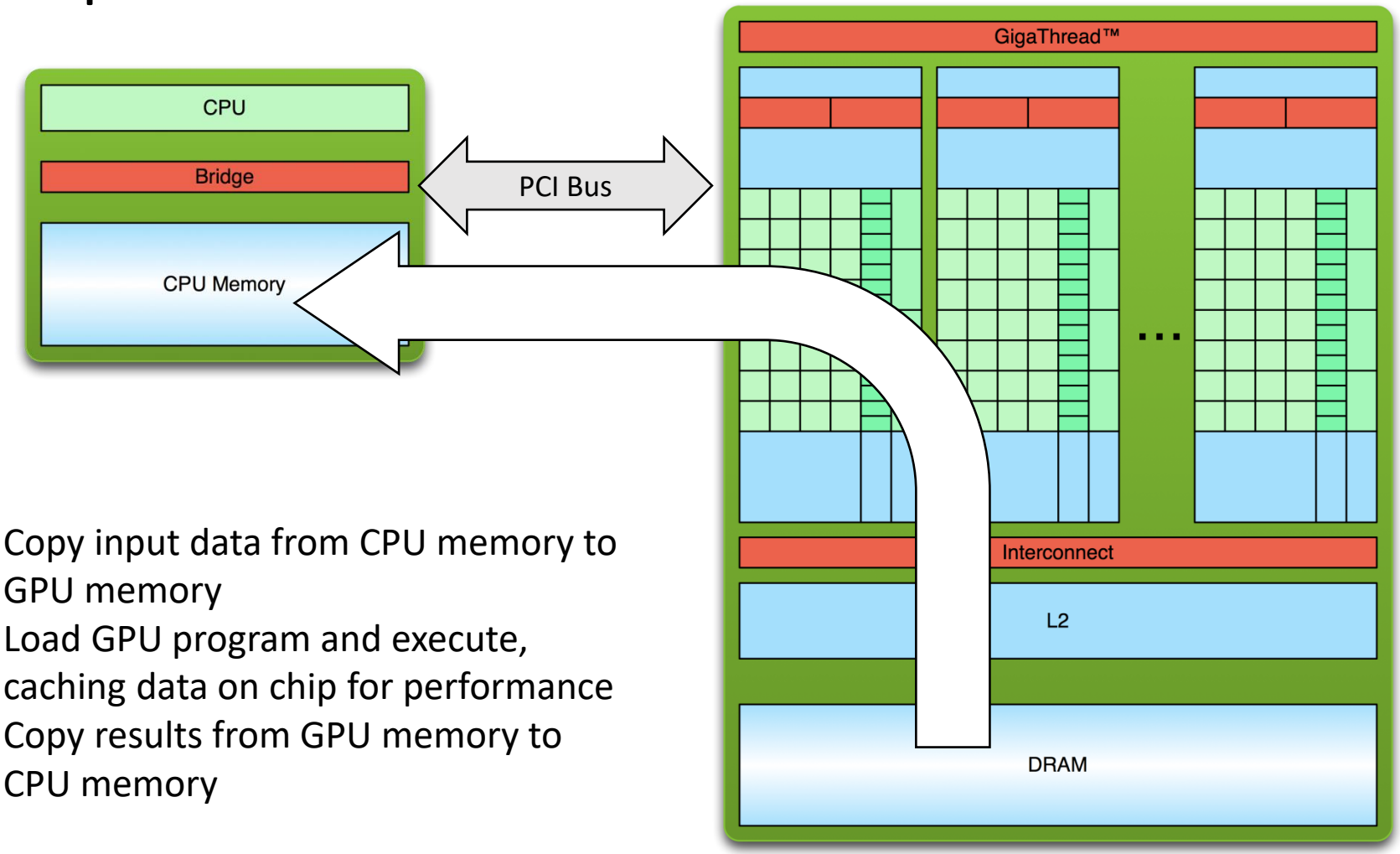
1. Copy input data from CPU memory to GPU memory

Run GPU codes



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

Transfer processed data from Device to Host



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

Mark Harris, **Introduction to CUDA C,** NVIDIA Corporation

Multidimensional threads and threadblocks

- Why do we need 3 dimensional threads and threadblocks?
- What's advantages?
  Application dependence

- We can think similar examples in C or C++.
  - 2D malloc allocation vs. 1D malloc allocation with 2D indexing

ex1) one-dimensional threads and threadblocks

```
dim3 nthreads;
ntreads.x = 4; nthreads.y = 1; nthreads.z = 1;
dim3 nblocks;
nblocks.x = 4; nblocks.y = 1; nblocks.z = 1;
kernel<<<nblocks, nthreads>>>()
```

ex2) two-dimensional threads and threadblocks

```
dim3 nthreads;
ntreads.x = 2; nthreads.y = 2; nthreads.z = 1;
dim3 nblocks;
nblocks.x = 2; nblocks.y = 2; nblocks.z = 1;
kernel<<<nblocks, nthreads>>>()
```

How to use blockIdx and threadIdx?

- **blockIdx and threadIdx are used to assign (map) data to threads**
- **blockIdx and threadIdx are used even for computation**

=> Domain knowledge is the key to know how to map effectively threads to data

For C++,

```
void copy_c(int *src_buf, *dst_buf) {
  for (int w=0; w<10; w++) {
    dst_buf[w] = src_buf[w] + w;
  }
}
```

For CUDA,

```
__global__ void copy_cuda(int *src, *dst) {
  dst[threadIdx.x] = src[threadIdx.x] + threadIdx.x;
}

main(){
  copy_c(src_buf, dst_buf);
  copy_cuda<<<1,10>>>(src_buf, dst_buf);
}
```

Mapping parallel threads to data (1/4)

- For a given matrix (n_x x n_y), we can map threads to elements of a matrix

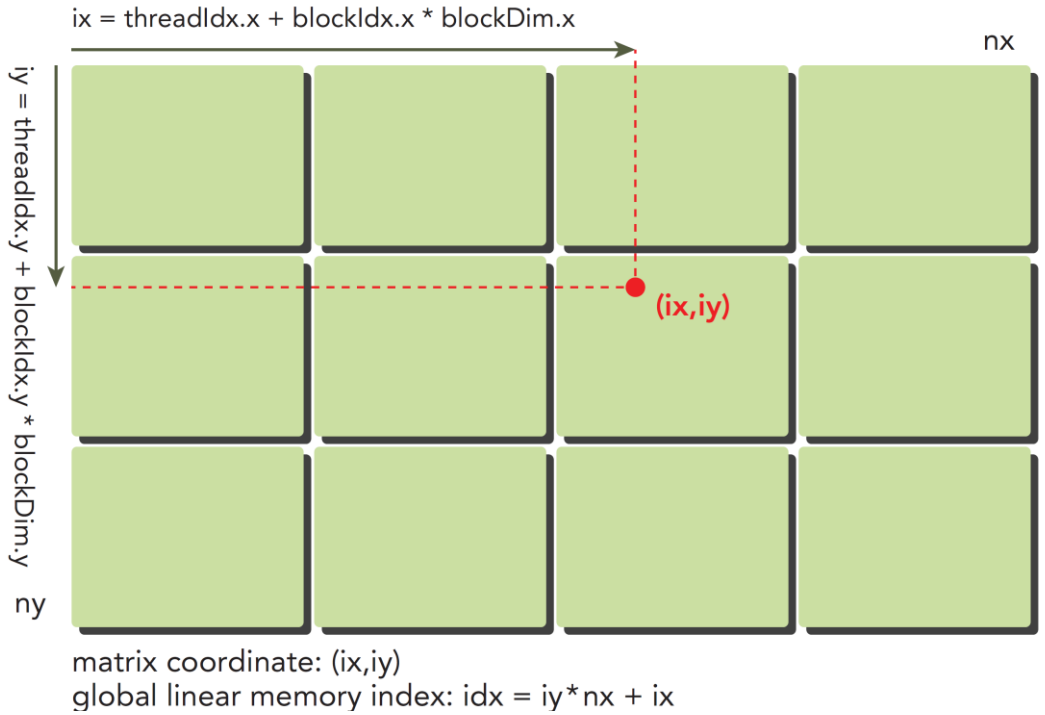
  ix = threadIdx.x + blockIdx.x * blockDim.x

  iy = threadIdx.y + blockIdx.y *blcockDim.y

The linear index

$idx = iy*n_x + ix$

- Which mapping is used in this example?
  - 1D grid and 1D blocks
  - 1D grid and 2D blocks
  - 2D grid and 1D blocks
  - 2D grid and 2D blocks ←



Mapping parallel threads to data (2/4)

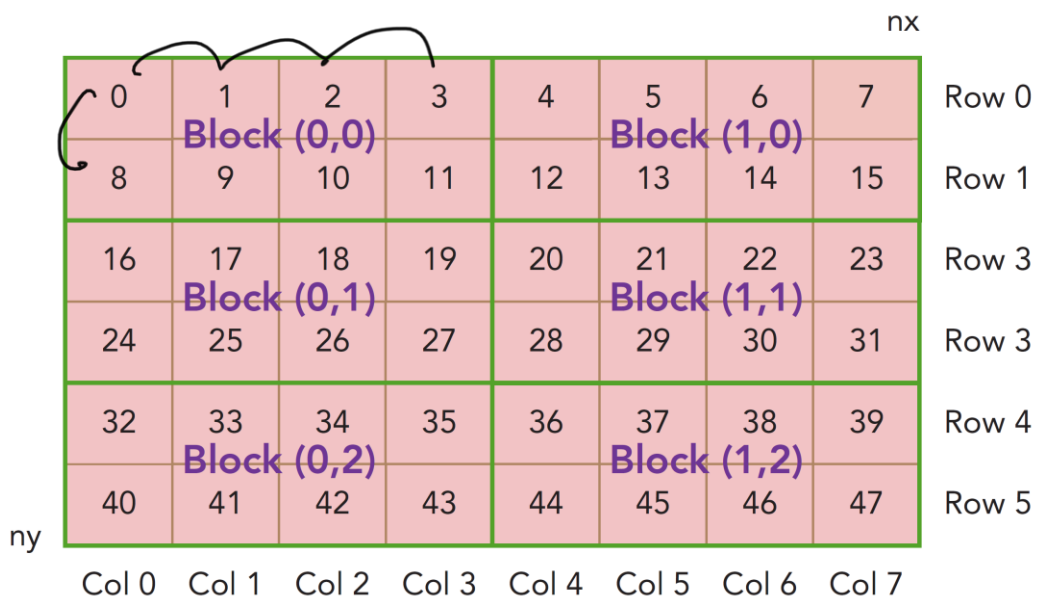
- 2D Grid, 2D Block

  kernel<<<(2,3,1), (4,2,1) >>>(dst, src, nx)

```
__global__ void kernel (int *dst, int *src, int nx) {
    idx_x = threadIdx.x + blockIdx.x*blockDim.x;
    idx_y = threadIdx.y + blockIdx.y*blockDim.y;
    linear_idx = idx_y*nx + idx_x;
    dst[linear_idx] = src[linear_idx] + 2;
}
```

If src and dst buffers are 1D array

If src and dst buffers are 2D array,
dst[idx_y][idx_x] = src[idx_y][idx_x] + 2;

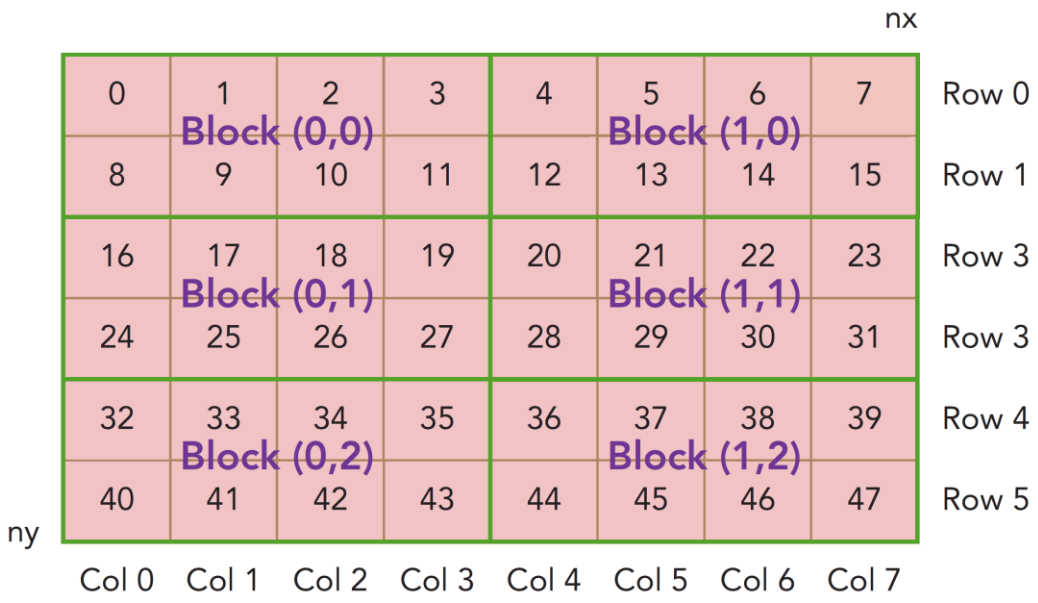


Mapping parallel threads to data (3/4)

- 2D Grid, 1D Block

  kernel<<<(2,3,1), (8,1,1) >>>(dst, src, nx)

```
__global__ void kernel (int *dst, int *src, int nx) {
    idx_x = (threadIdx.x & 0x3)+ blockIdx.x*blockDim.x;
    idx_y = (threadIdx.x >> 2) + blockIdx.y*blockDim.y;
    linear_idx = idx_y*nx + idx_x;
    dst[linear_idx] = src[linear_idx] + 2;
}
```



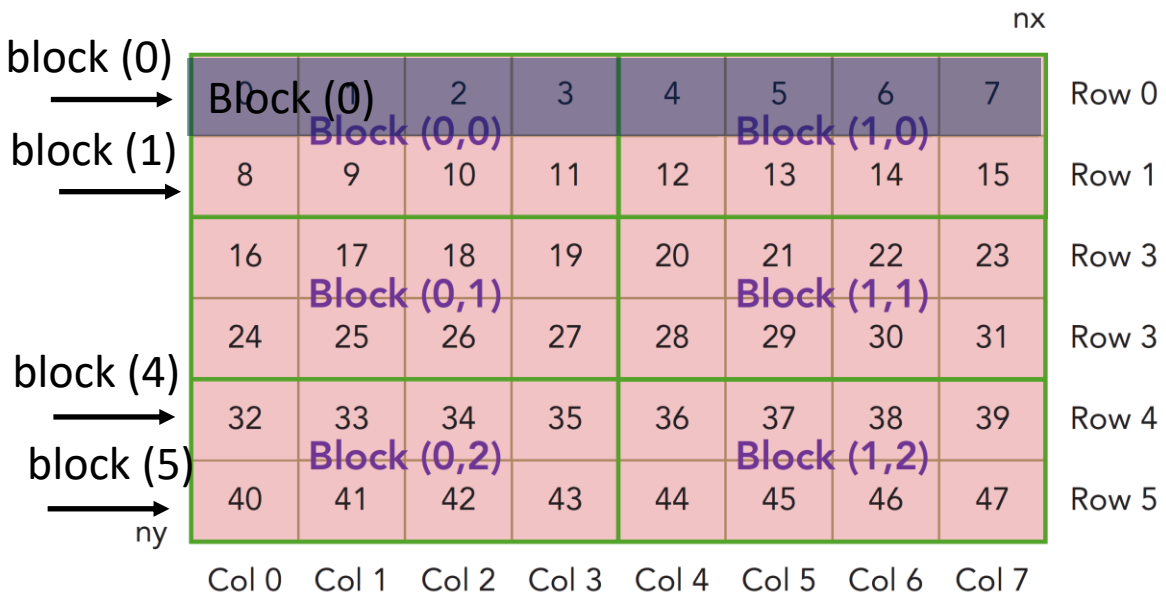
Mapping parallel threads to data (4/4)

- 1D Grid, 1D Block

  kernel<<<(6,1,1), (8,1,1) >>>(dst, src, nx)

```
__global__ void kernel (int *dst, int *src, int nx) {
    idx_x = threadIdx.x;
    idx_y = blockIdx.x;
    linear_idx = idx_y*nx + idx_x;
    dst[linear_idx] = src[linear_idx] + 2;
}
```

Class_lab: c1_checkThreadIndex



Example: Grid and block partition (1/3)

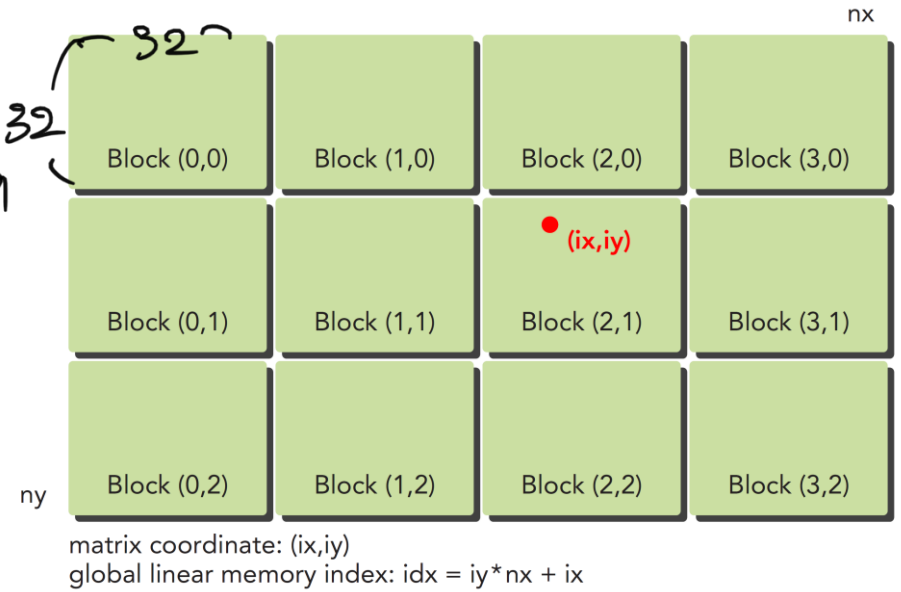
- 2D grid and 2D block partition

  Map one thread to one matrix element

  one threadblock processes a sub-block of a matrix (multiple columns and multiple rows

```
__global__ void sumMatrixOnGPU2D(float *MatA, float *MatB, float *MatC, int nx, int ny)
{
    unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;
    unsigned int iy = threadIdx.y + blockIdx.y * blockDim.y;
    unsigned int idx = iy*nx + ix;
    if (ix < nx && iy < ny)
      MatC[idx] = MatA[idx] + MatB[idx];
}

int dimx = 32;
int dimy = 32;
dim3 block(dimx, dimy);
dim3 grid((nx+block.x-1)/block.x, (ny+block.y-1)/block.y);
sumMatrixOnGPU2D <<< grid, block >>>(d_MatA, d_MatB, d_MatC, nx, ny);
```

one element/thread

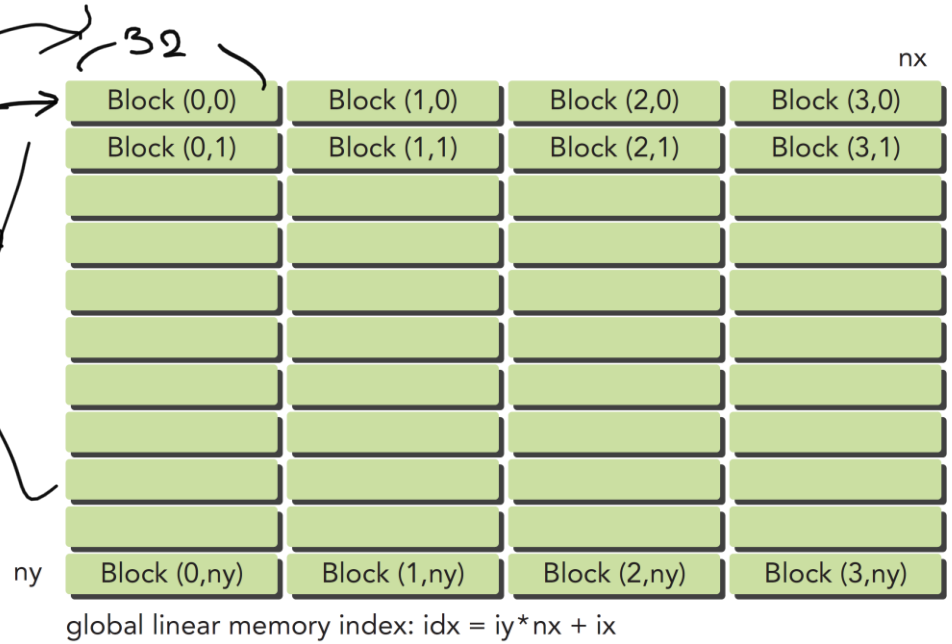


Example: Grid and block partition (2/3)

- 2D grid and 1D block partition
  Map one thread to one matrix element
  one threadblock processes multiple columns every row

```
__global__ void sumMatrixOnGPUMix(float *MatA, float *MatB, float *MatC, int nx, int ny)
{
    unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;
    unsigned int iy = blockIdx.y;
    unsigned int idx = iy*nx + ix;
    if (ix < nx && iy < ny)
        MatC[idx] = MatA[idx] + MatB[idx];
}

dim3 block(32);
dim3 grid((nx + block.x - 1) / block.x,ny);
sumMatrixOnGPUMIx <<< grid , block >>>(d_MatA, d_MatB, d_MatC, nx, ny);
```

one element/thread



Example: Grid and block partition (3/3)

- 1D grid and 1D block partition

  Map one thread to one matrix element

  However, one thread processes multiple elements

  one threadblock processes a large matrix block

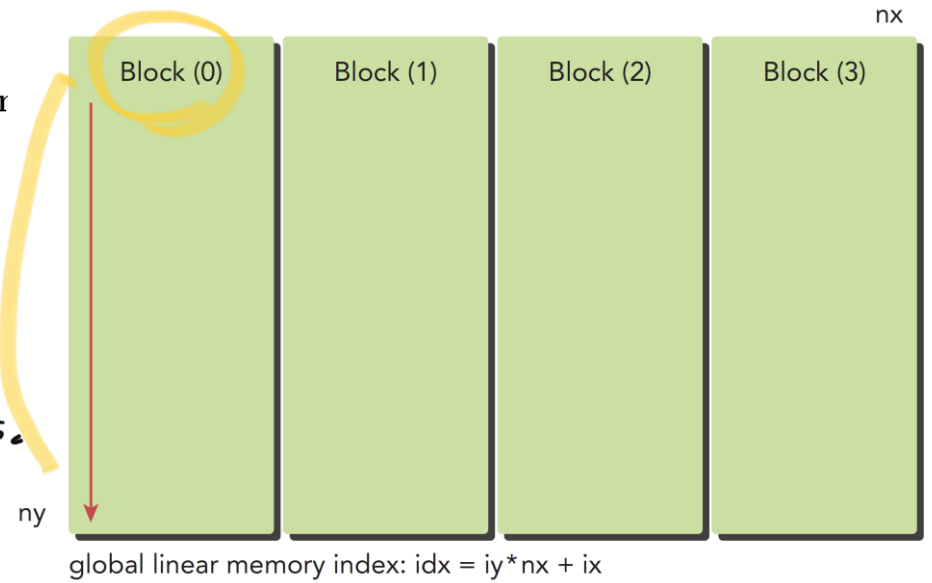
```
__global__ void sumMatrixOnGPU1D(float *MatA, float *MatB, float *MatC, ir
{
  unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;
  if (ix < nx ) {
    for (int iy=0; iy<ny; iy++) {
      int idx = iy*nx + ix;
      MatC[idx] = MatA[idx] + MatB[idx];
    }
  }
}
```

multiple processing/ thread

one element/thread

for-loop 때문에 전체 데이 처리가능

```
dim3 block(32,1);
dim3 grid((nx+block.x-1)/block.x,1);
sumMatrixOnGPU2D <<< grid, block >>>(d_MatA, d_MatB, d_MatC, nx, ny);
```



Performance vs. grid and block partitions

- The performance of GPU depends on grid and block partitions
- There is no simple rule to derive the best performance

  It depends on GPU architecture, data size, access pattern and so on

  However, **generally a single thread should process multiple elements in a threadblock with large data size**

  Data size per operation per thread should be larger 4byte

  (int, int2, int4, float, float2, float4)

16 Byte/thread

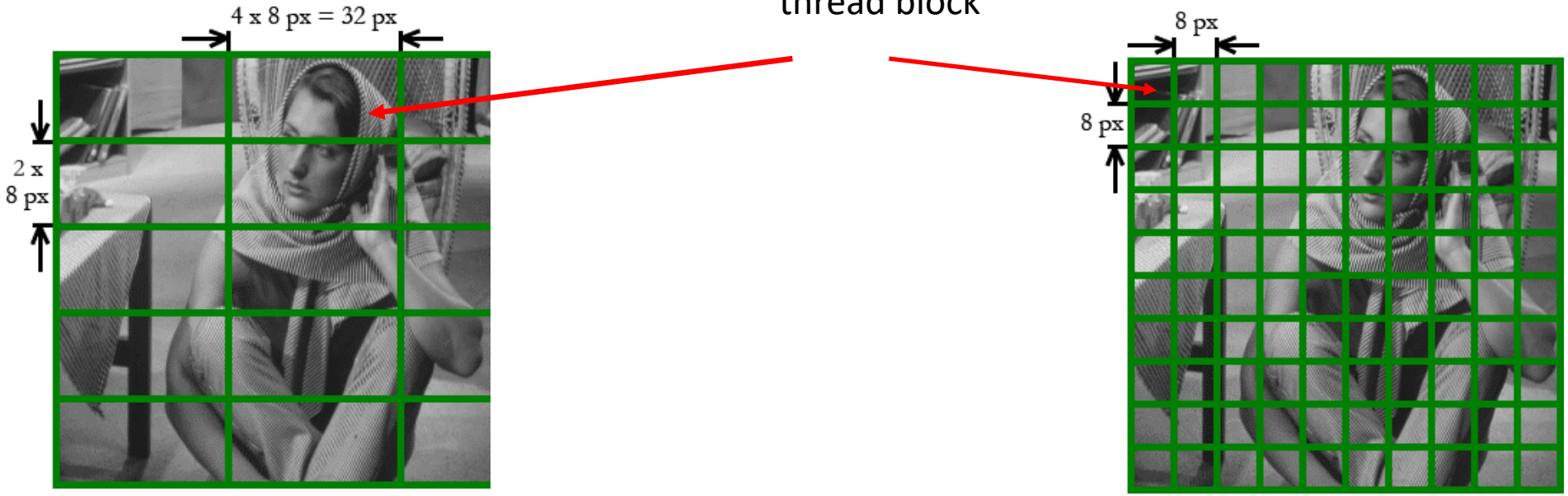
KERNEL	EXECUTION CONFIGURE	TIME ELAPSED
sumMatrixOnGPU2D	(512,1024), (32,16)	0.038041
sumMatrixOnGPU1D	(128,1), (128,1)	0.044701
sumMatrixOnGPUMix	(64,16384), (256,1)	0.030765

Example: CUDA Discrete Cosine Transform

8x4x2 threads to 8 8x8 blocks samples (8 samples/thread)
2 warps/block

vs.

8x8 threads to 8x8 block samples (one sample/thread)
2 warps/block



Kernel <<< ((W+31)/32, (H+15)/16, 1) , (4,16,1) >>>

Kernel <<< ((W+7)/8, (H+7)/8, 1) , (8,8,1) >>>