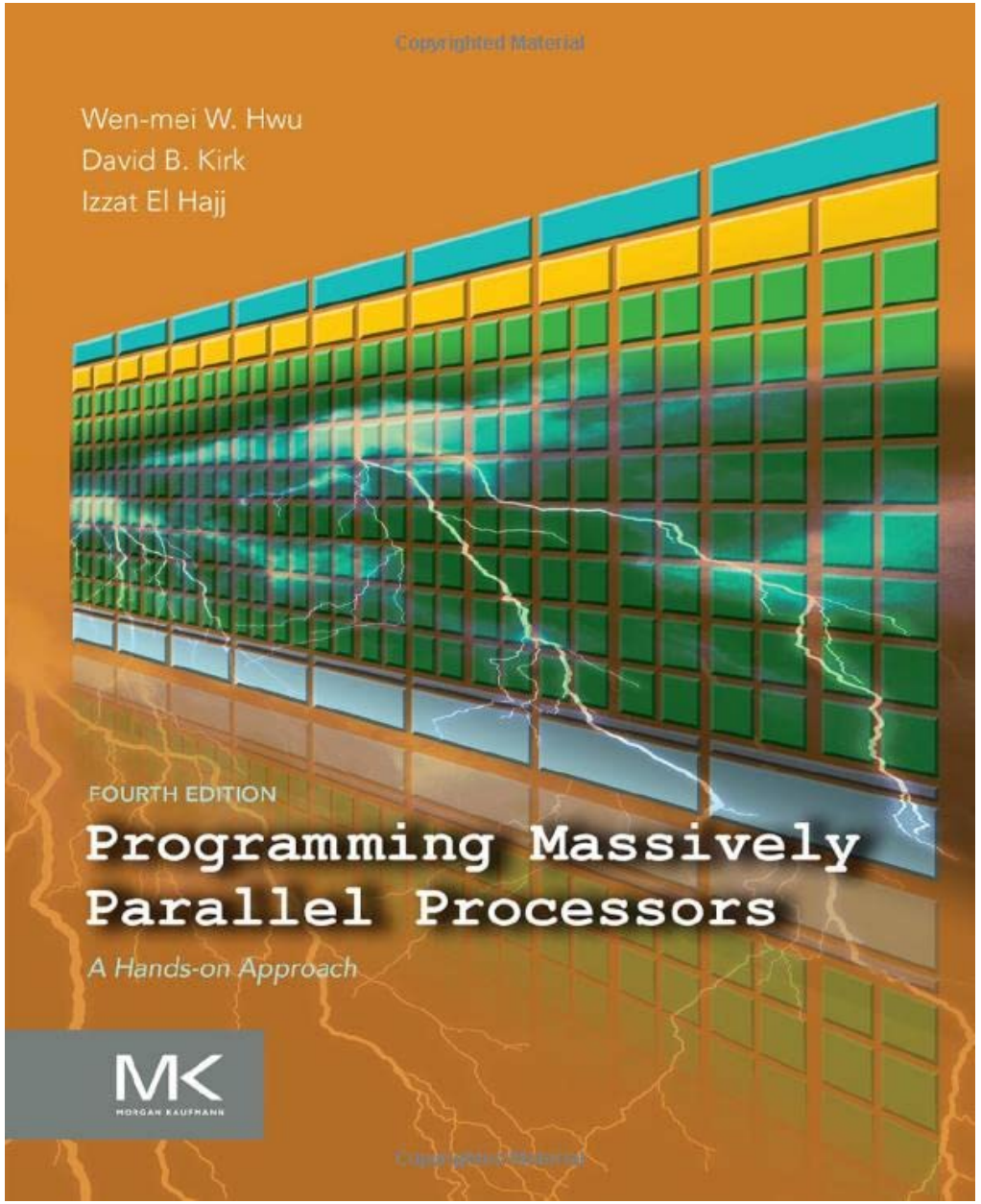


pmpp book ch. 1-3

CUDA-MODE

Lecture 2

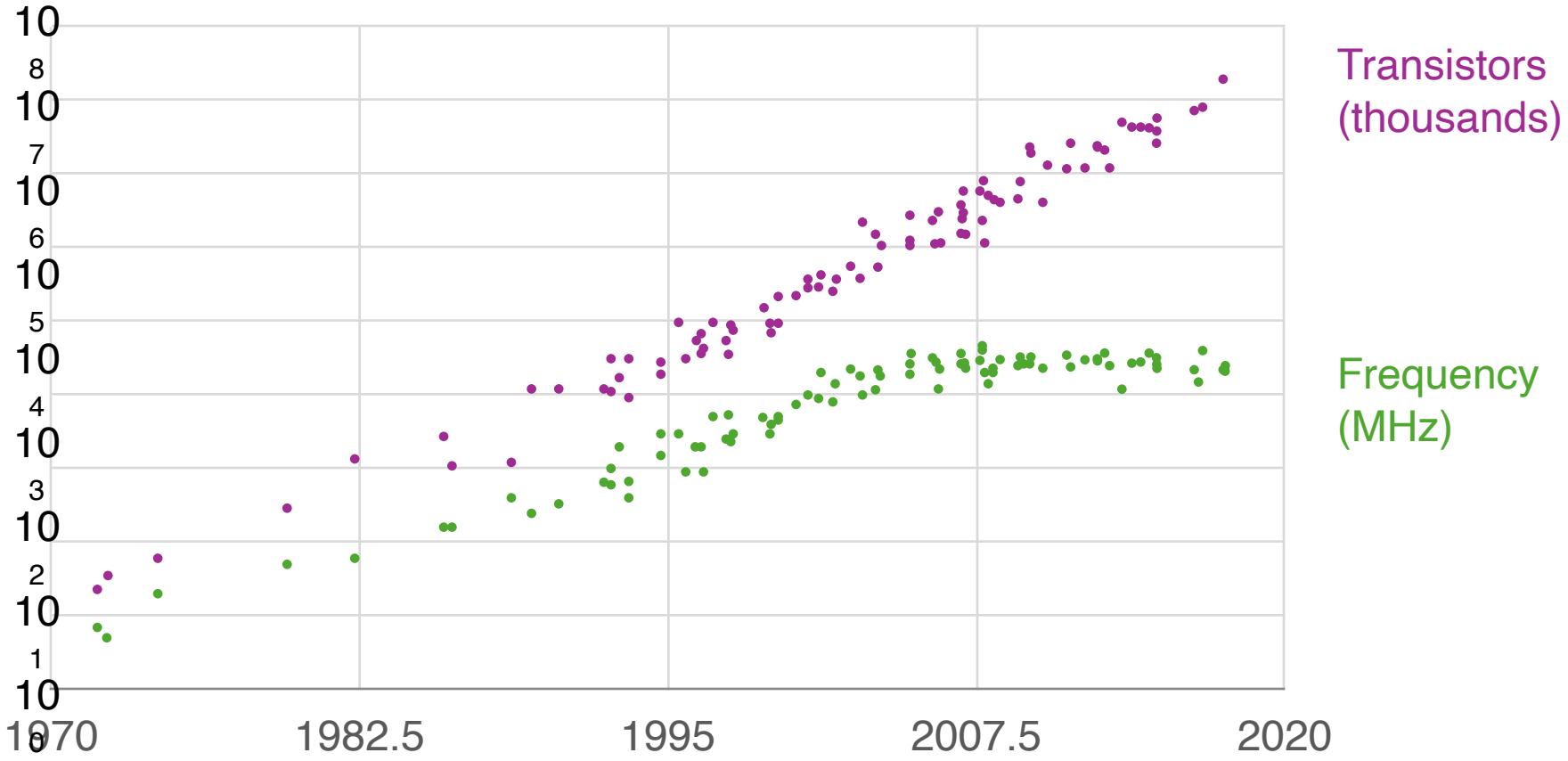
Agenda for Lecture 2

- 1: Introduction
- 2: Heterogeneous data parallel computing
- 3: Multidimensional grids and data

Ch 1: Introduction

- motivation: GPU go brrr, more FLOPS please
- Why? Simulation & world-models (games, weather, proteins, robotics)
- Bigger models are smarter -> AGI (prevent wars, fix climate, cure cancer)
- GPUs are the backbone of modern deep learning
- classic software: sequential programs
- higher clock rate trend for CPU slowed in 2003: energy consumption & heat dissipation
- multi-core CPU came up
- developers had to learn multi-threading (deadlocks, races etc.)

The Power Wall



Source: M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, C. Batten (1970-2010). K. Rupp (2010-2017).

(increasing frequency further would make the chip too hot to cool feasibly)

The rise of CUDA

- CUDA is all about parallel programs (modern software)
- GPUs have (much) higher peak FLOPS than multi-core CPUs
- main principle: divide work among threads
- GPUs focus on execution throughput of massive number of threads
- programs with few threads perform poorly on GPUs
- CPU+GPU: sequential parts on CPU, numerical intensive parts on GPU
- CUDA: Compute Unified Device Architect
- GPGPU: Before CUDA tricks were used to compute with graphics APIs (OpenGL or Direct3D)
- GPU programming is now attractive for developers (thanks to massive availability)

Amdahl's Law

- speedup = slow_sys_time / fast_sys_time
- achievable speedup is limited by the parallelizable portion ***p*** of programs

$$speedup < \frac{1}{1 - p}$$

  - e.g., if ***p*** is 90%, speedup < 10×

- Fortunately, for many real applications, ***p*** > 99% especially for large datasets, and speedups >100× are attainable

Challenges

- "if you do not care about performance, parallel programming is very easy"
- designing parallel algorithms in practice harder than sequential algorithms
  e.g. parallelizing recurrent computations requires nonintuitive thinking (like prefix sum)
- speed is often limited by memory latency/throughput (memory bound)
- perf of parallel programs can vary dramatically based on input data characterists
- not all apps are "embarassingly parallel" - synchronization imposes overhead (waits)

Main Goals of the Book

1. Parallel programming & computational thinking
2. Correct & reliable: debugging function & performance
3. Scalability: regularize and localize memory access

- PMPP aims to build up the foundation for parallel programming in general
- GPUs as learning vehicle - techniques apply to other accelerators
- concepts are introduced hands-on as concrete CUDA examples

Ch 2: Heterogeneous data parallel computing

- heterogeneous: CPU + GPU
- data parallelism: break work down into computations that can be executed independently
- Two examples: vector addition & kernel to convert an RGB image to grayscale
- Independence: each RGB pixel can be converted individually
- L = r*0.21 + g*0.72 + b*0.07 (L=luminance)
- simple weighted sum

RGB->Grayscale, data independence

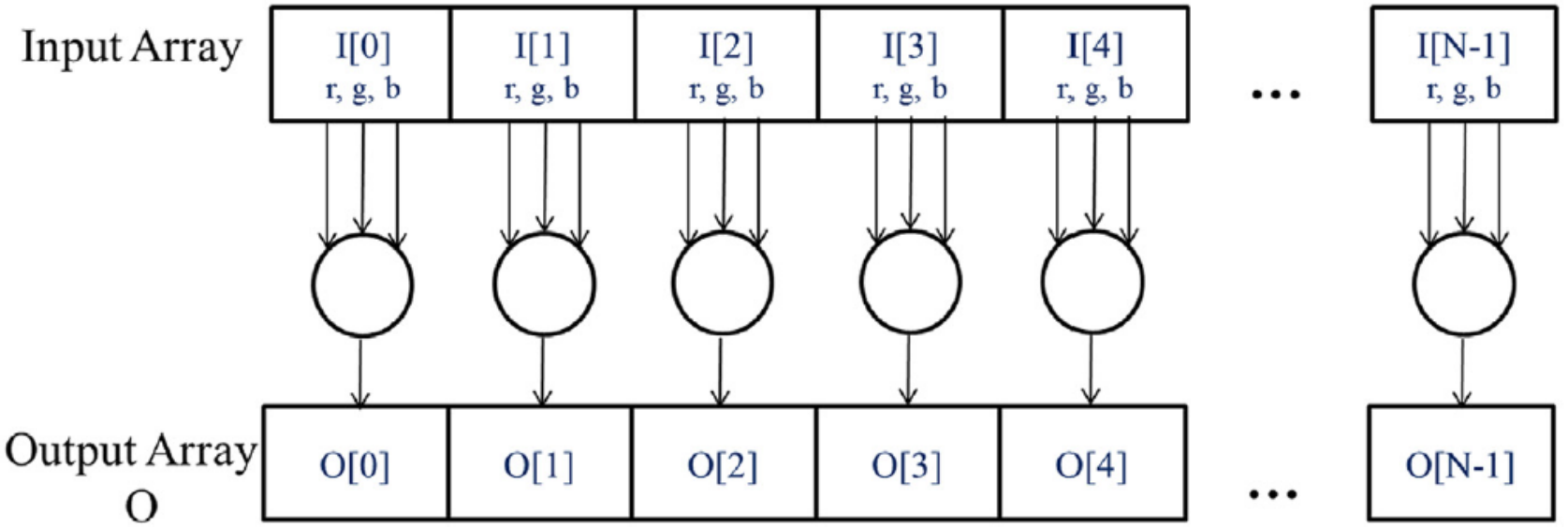


FIGURE 2.2

Data parallelism in image-to-grayscale conversion. Pixels can be calculated independently of each other.

CUDA C

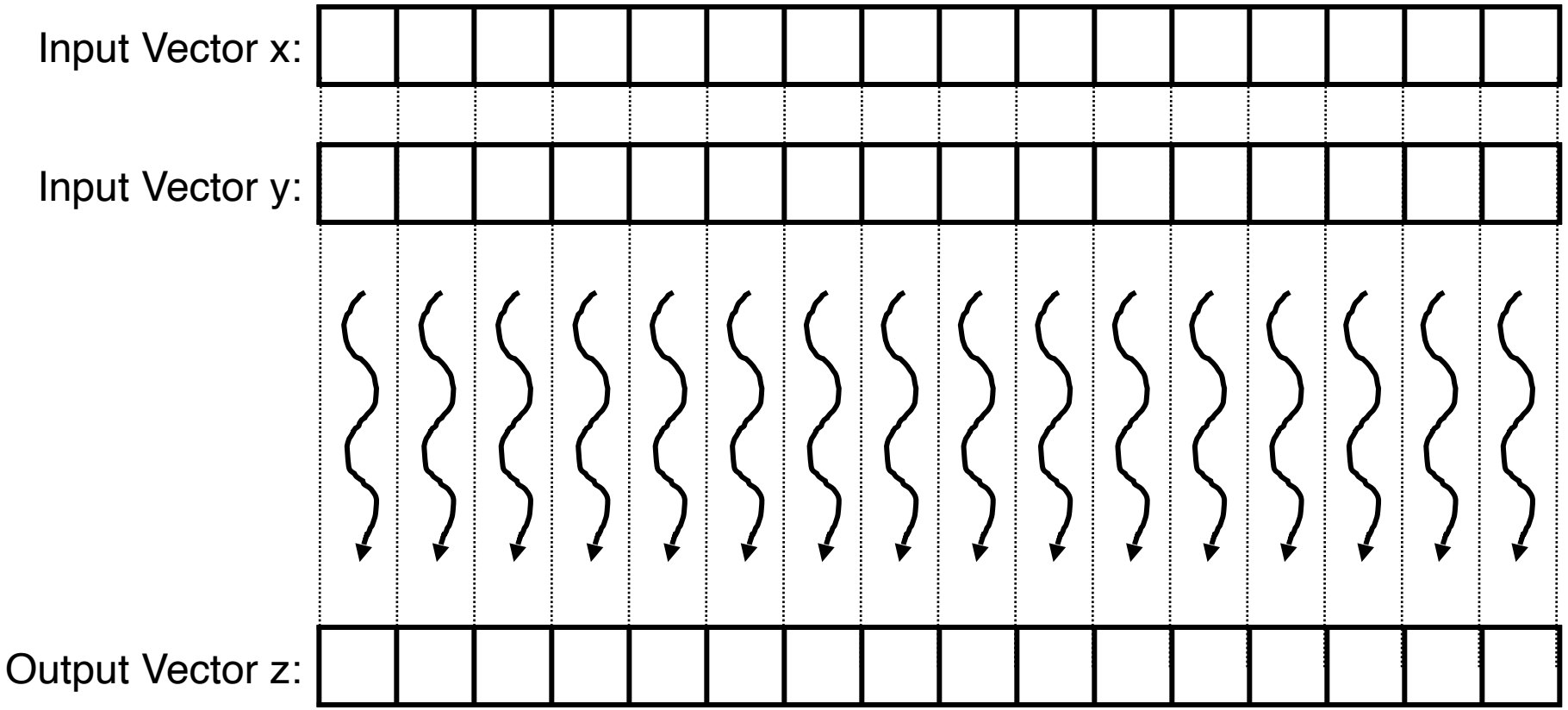
- extends ANSI C with minimal new syntax
- Terminology: CPU = host, GPU = device
- CUDA C source can be mixture of host & device code
- device code functions: kernels
- grid of threads: many threads are launched to execute a kernel
- CPU & GPU code runs concurrently (overlapped)
- on GPU: don't be afraid of launching many threads
- e.g. one thread pre (output) tensor element is fine

Example: Vector Addition

- vector addition example:
  - main concept loop -> threads
  - Easily parallelizable: all additions can be computed independently

- Naïve GPU vector addition:
  1. Allocate device memory for vectors
  2. Transfer inputs host -> device
  3. Launch kernel and perform additions
  4. Copy device -> host back
  5. Free device memory

- normally we keep data on the gpu as long as possible to asynchronously schedule many kernel launches



- One thread per vector element

CUDA Essentials: Memory allocation

- nvidia devices come with their own DRAM (device) global memory
  (in Ch 5 we learn about other mem types)

- `cudaMalloc` & `cudaFree`:

```
float *A_d;
size_t size = n * sizeof(float); // size in bytes
cudaMalloc((void**)&A_d, size);  // pointer to pointer!
...
cudaFree(A_d);
```

cudaMemcpy: Host <-> Device Transfer

- Copy data from CPU memory to GPU memory and vice versa

```
// copy input vectors to device (host -> device)
cudaMemcpy(A_d, A_h, size, cudaMemcpyHostToDevice);
cudaMemcpy(B_d, B_h, size, cudaMemcpyHostToDevice);

...

// transfer result back to CPU memory (device -> host)
cudaMemcpy(C_h, C_d, size, cudaMemcpyDeviceToHost);
```

CUDA Error handling

- CUDA functions return `cudaError_t` .. if not `cudaSuccess` we have a problem ...
- always check returned error status ☺

Kernel functions fn<<>>

- Launching kernel = grid of threads is launched
- All threads execute the same code: Single program multiple-data (SPMD)
- Threads are hierarchically organized into **grid blocks** & **thread blocks**
- up to 1024 threads can be in a thread block

Kernel Coordinates

- built-in variables available inside the kernel: **blockIdx**, **threadIdx**
- these "coordinates" allow threads (all executing the same code) to identify what to do (e.g. which portion of the data to process)
- each thread can be uniquely identified by threadIdx & blockIdx
- telephone system analogy: think of `blockIdx` as the area code and `threadIdx` as the local phone number
- built-in `blockDim` tells us the number of threads in a block
- for vector addition we can calculate the array index of the thread

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

Threads execute the same kernel code

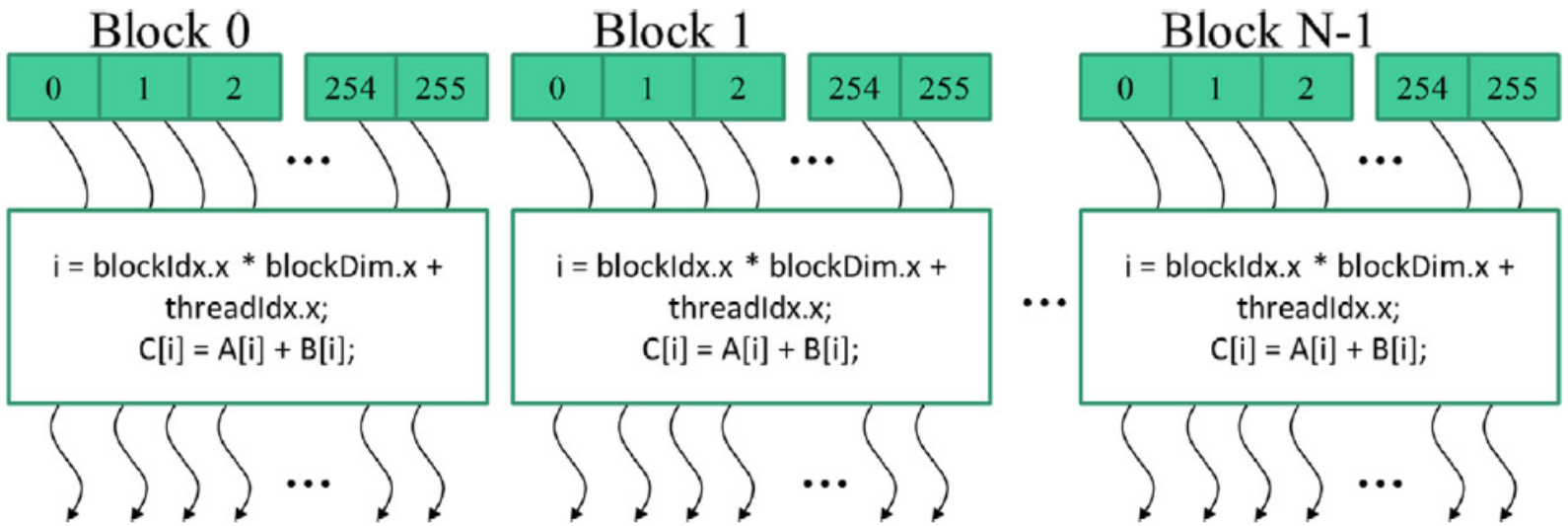


FIGURE 2.9

All threads in a grid execute the same kernel code.

__global__ & __host__

- declare a kernel function with `__global__`
- calling a `__global__` function -> launches new grid of cuda threads
- functions declared with `__device__` can be called from within a cuda thread
- if both `__host__` & `__device__` are used in a function declaration CPU & GPU versions will be compiled

Qualifier Keyword	Callable From	Executed On	Executed By
__host__ (default)	Host	Host	Caller host thread
__global__	Host (or Device)	Device	New grid of device threads
__device__	Device	Device	Caller device thread

FIGURE 2.11

CUDA C keywords for function declaration.

Vector Addition Example

- general strategy: replace loop by grid of threads!
- data sizes might not perfectly divisible by block sizes: always check bounds
- prevent threads of boundary block to read/write outside allocated memory

```
01 // compute vector sum C = A + B
02 // each thread peforms one pair-wise addition
03 __global__
04 void vecAddKernel(float* A, float* B, float* C, int
n) {
05  int i = threadIdx.x + blockDim.x * blockIdx.x;
06  if (i < n) {    // check bounds
07    C[i] = A[i] + B[i];
08  }
09 }
```

Calling Kernels

- kernel configuration is specified between `<<<` and `>>>`
- number of blocks, number of threads in each block

```
dim3 numThreads(256);
dim3 numBlocks((n + numThreads - 1) / numThreads);
vecAddKernel<<<numBlocks, numThreads>>>(A_d, B_d, C_d,
n);
```

- we will learn about additional launch parameters (shared-mem size, cudaStream) later

Compiler

- nvcc (NVIDIA C compiler) is used to compile kernels into PTX
- Parallel Thread Execution (PTX) is a low-level VM & instruction set
- graphics driver translates PTX into executable binary code (SASS)

Ch 3: Multidimensional grids and data

- CUDA grid: 2 level hierarchy: **blocks**, **threads**
- Idea: map threads to multi-dimensional data
- all threads in a grid **execute the same kernel**
- threads in same block can access the same shared mem
- max block size: 1024 threads
- built-in 3D coordinates of a thread: **blockIdx**, **threadIdx -** identify which portion of the data to process
- shape of grid & blocks:
  - **gridDim**: number of blocks in the grid
  - **blockDim**: number of threads in a block

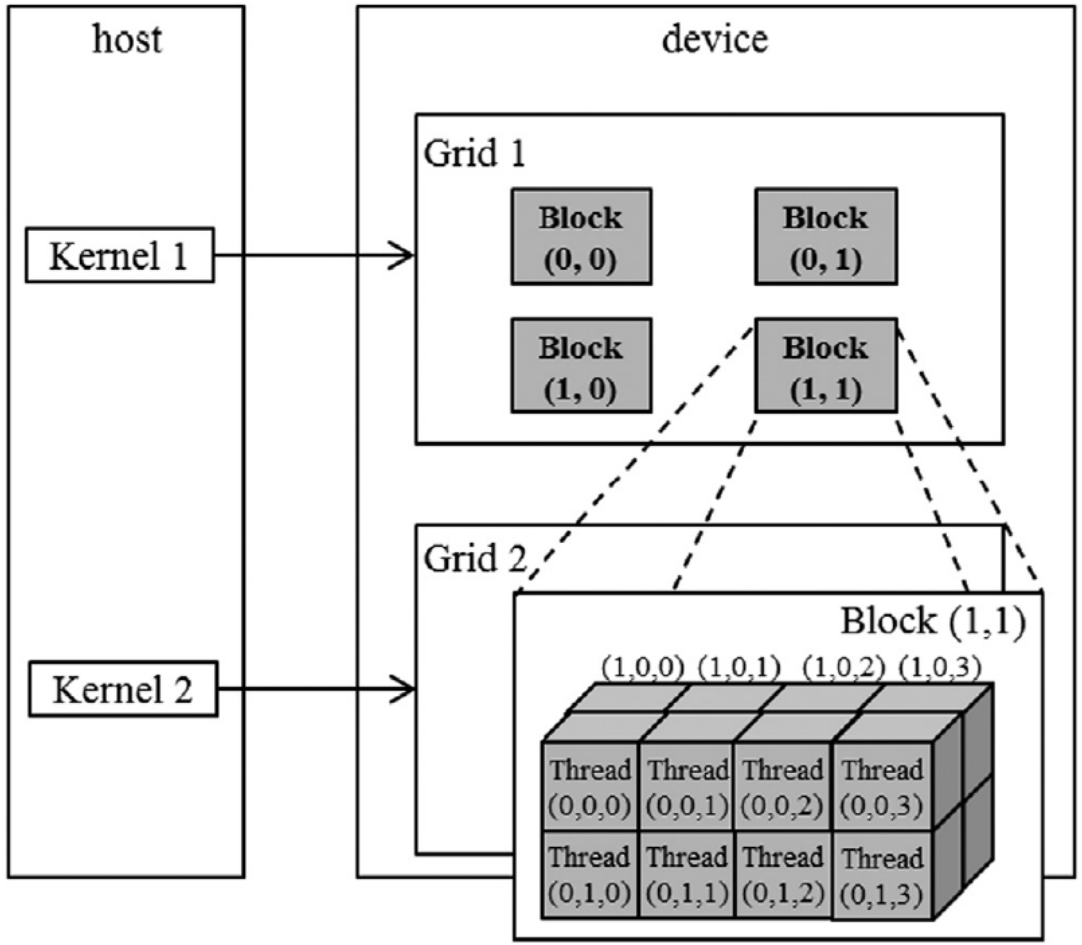


FIGURE 3.1

A multidimensional example of CUDA grid organization.

Grid continued

- grid can be different for each kernel launch, e.g. dependent on data shapes
- typical grids contain thousands to millions of threads
- simple strategy: one thread per output element (e.g. one thread per pixel, one thread per tensor element)
- **threads can be scheduled in any order**
- can use fewer than 3 dims (set others to 1)
- e.g. 1D for sequences, 2D for images etc.

```
dim3 grid(32, 1, 1);
dim3 block(128, 1, 1);
kernelFunction<<<grid, block>>>(..);
// Number of threads: 128 * 32=4096
```

Built-in Variables

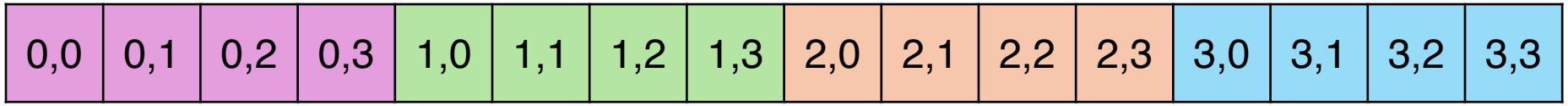
- Built-in variables inside kernels:

```
blockIdx      // dim3 block coordinate
threadIdx     // dim3 thread coordinate
blockDim      // number of threads in a block
gridDim             // number of blocks in a grid
```

- (blockDim & gridDim have the same values in all threads)

nd-Arrays in Memory

Actual layout in memory



Logical view of data

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3
3,0	3,1	3,2	3,3

- memory of multi-dim arrays under the hood is flat 1d
- 2d array can be linearized different ways:

```
- row-major:
  A B C
  D E F
  G H I
```

```
- column-major:
  A D G
  B E H
  C F I
```

- torch tensors & numpy ndarrays use strides to specify how elements are laid out in memory.

Image blur example (3.3, p. 60)

- mean filter example blurKernel
- each thread writes one output element, reads multiple values
- single plane in book, can be extended easily to multi-channel
- shows row-major pixel memory access (in & out pointers)
- track of how many pixels values are summed
- handles boundary conditions in ln 5 & 25

FIGURE 3.6

An original image (*left*) and a blurred version (*right*).

Handling Boundary Conditions

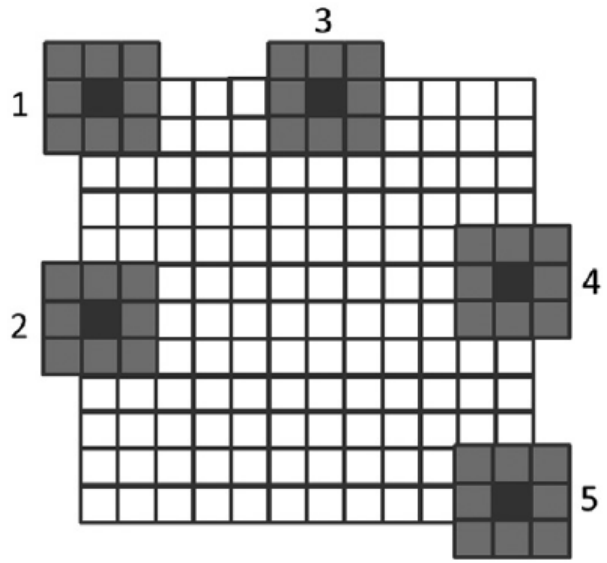


FIGURE 3.9

Handling boundary conditions for pixels near the edges of the image.

```
__global__
void mean_filter_kernel(unsigned char* output, unsigned char* input, int width, int height, int radius) {
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int channel = threadIdx.z;

    int baseOffset = channel * height * width;
    if (col < width && row < height) {

        int pixVal = 0;
        int pixels = 0;

        for (int blurRow=-radius; blurRow <= radius; blurRow += 1) {
            for (int blurCol=-radius; blurCol <= radius; blurCol += 1) {
                int curRow = row + blurRow;
                int curCol = col + blurCol;
                if (curRow >= 0 && curRow < height && curCol >=0 && curCol < width) {
                    pixVal += input[baseOffset + curRow * width + curCol];
                    pixels += 1;
                }
            }
        }

        output[baseOffset + row * width + col] = (unsigned char)(pixVal / pixels);
    }
}
```

Matrix Multiplication

- staple of science & engineering (and deep learning)
- compute inner-products of rows & columns
- Strategy: 1 thread per output matrix element
- Example: Multiplying square matrices (rows == cols)

```
01    __global__ void MatrixMulKernel(float* M, float* N,
02                                    float* P, int Width) {
03        int row = blockIdx.y*blockDim.y+threadIdx.y;
04        int col = blockIdx.x*blockDim.x+threadIdx.x;
05        if ((row < Width) && (col < Width)) {
06            float Pvalue = 0;
07            for (int k = 0; k < Width; ++k) {
08                Pvalue += M[row*Width+k]*N[k*Width+col];
09            }
10            P[row*Width+col] = Pvalue;
11        }
12    }
```

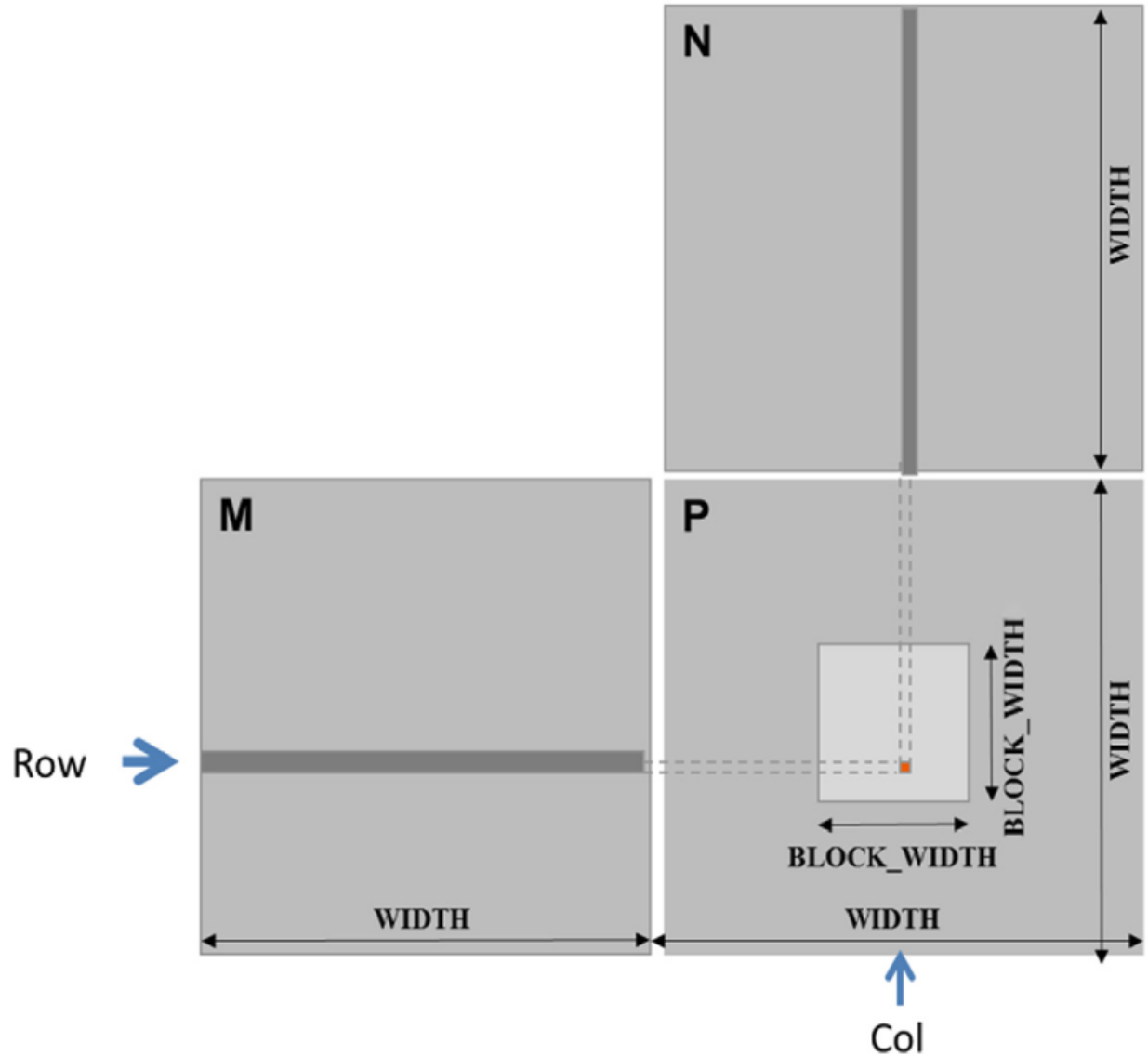


FIGURE 3.10

Matrix multiplication using multiple blocks by tiling P.