ECE408 / CS483 / CSE408
Summer 2025

Applied Parallel Programming

Lecture 8: Convolution, Constant Memory and Constant Caching

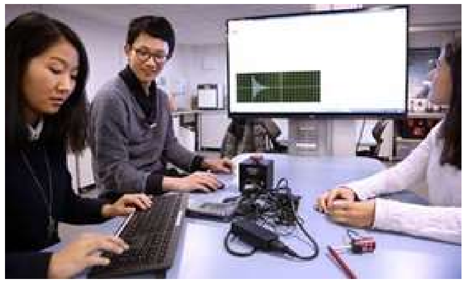
What Will You Learn Today?

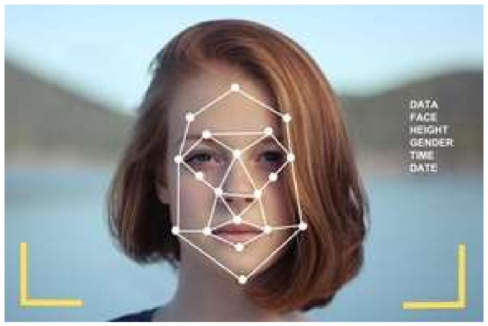
- convolution, an important parallel computation pattern
  - widely used in signal, image and video processing
  - basis for stencil computation used in many science and engineering applications
  - critical component of Neural Networks and Deep Learning

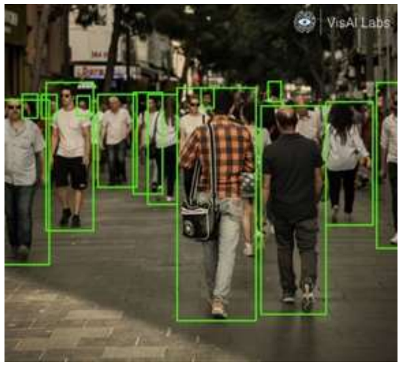
- important technique: leveraging cached memories

Convolution is a Widely-Used Array Operation

- **Convolution**: a **common array operation** used in
  - signal processing,
  - digital recording,
  - image processing,
  - video processing,
  - computer vision, and
  - machine learning.







How Does Convolution Work?

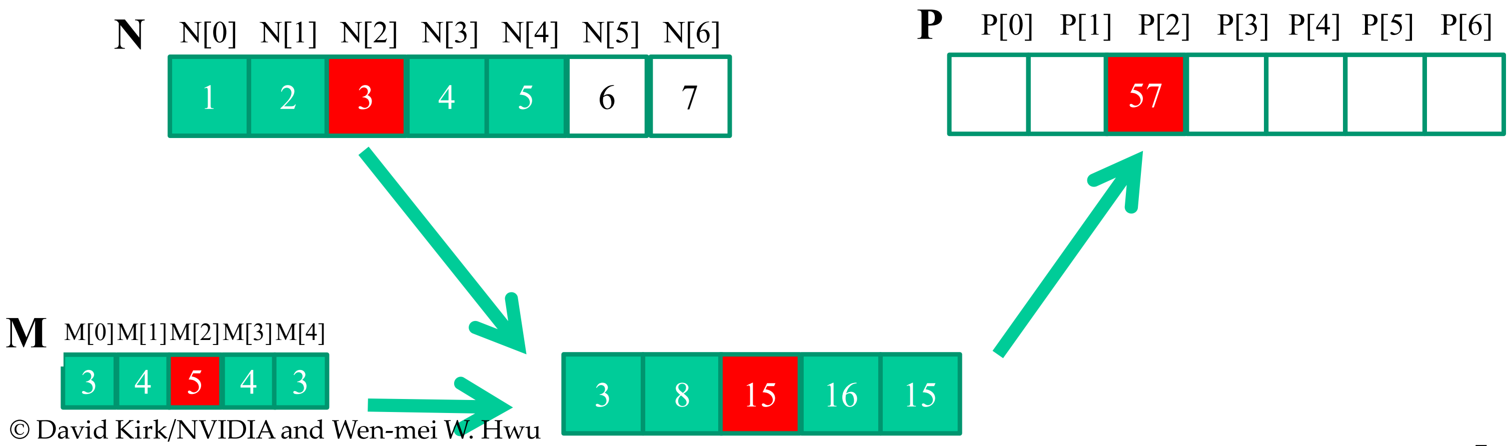
- Convolution **converts an array into another array**.
  - Each output data element
  - is a weighted sum
  - of nearby input elements.
- **Definition of “nearby” and weights** on necessary inputs
  - Together **form a convolution mask** (also filter, kernel),
  - a **constant array with** the **same dimensionality** as input data.
- Generally, the **same mask** is **used for all input** elements.

Convolution Applications are Numerous

- In Discrete Signal Processing,
  - **any filter** (low-pass, high-pass, band-pass, notch, or any other)
  - **can be implemented as** a **convolution**.
- In **image processing**, **applications** of convolution **include**
  - **blurring/smoothing,**
  - **sharpening,**
  - **edge detection, and**
  - **feature identification**.
- In scientific computing, **solution of discretized partial differential equations** are **implemented as convolutions**.

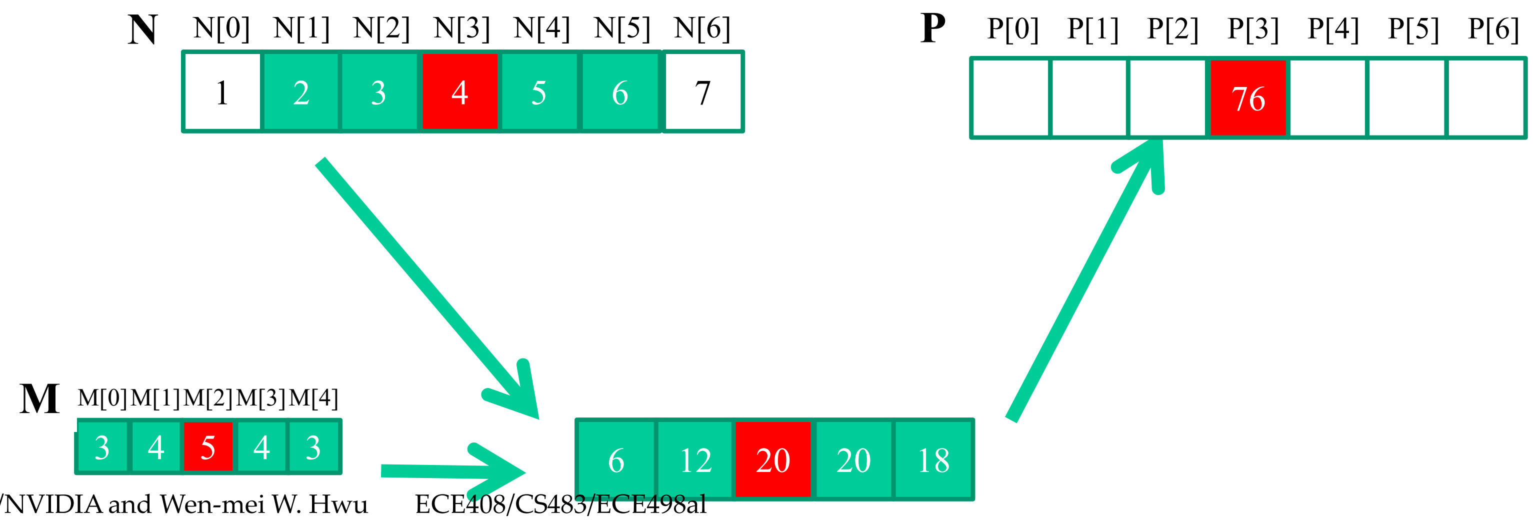
Let’s Start with a 1D Example

- Commonly used for audio processing
  - **Mask_Width**: **number of elements** in mask; **typically odd** for symmetry (5 in this example)
  - **Mask_Radius**: **number of elements on each side** of pixel being calculated (2 in this example).
- Calculation of P[2]:



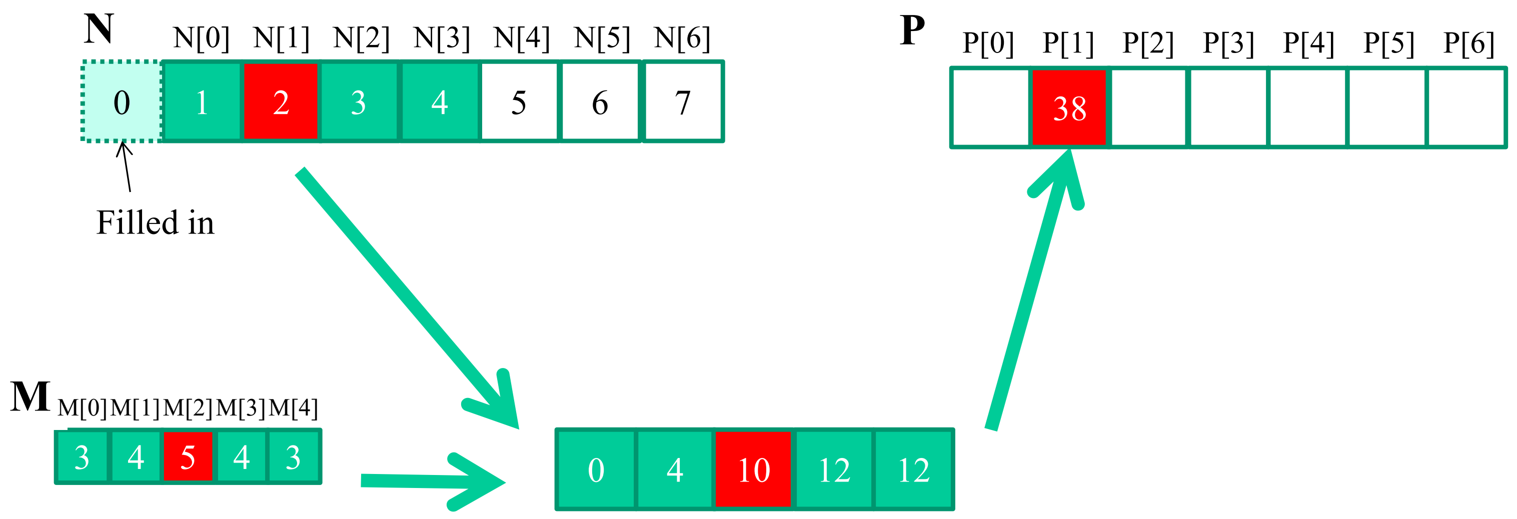
And a Second Output…

- Calculation of P[3]


N
N[0] N[1] N[2] N[3] N[4] N[5] N[6]
1 2 3 4 5 6 7
P
P[0] P[1] P[2] P[3] P[4] P[5] P[6]
76
M
M[0] M[1] M[2] M[3] M[4]
3 4 5 4 3
6 12 20 20 18

What Happens at Boundaries?

- Calculation of output elements near the boundaries (beginning and end) of the input array need to deal with "ghost" elements
  - Different policies (0, replicates of boundary values, etc.)



1D Convolution Kernel with Boundary Condition Handling

- This kernel forces all elements outside the valid data index range to 0

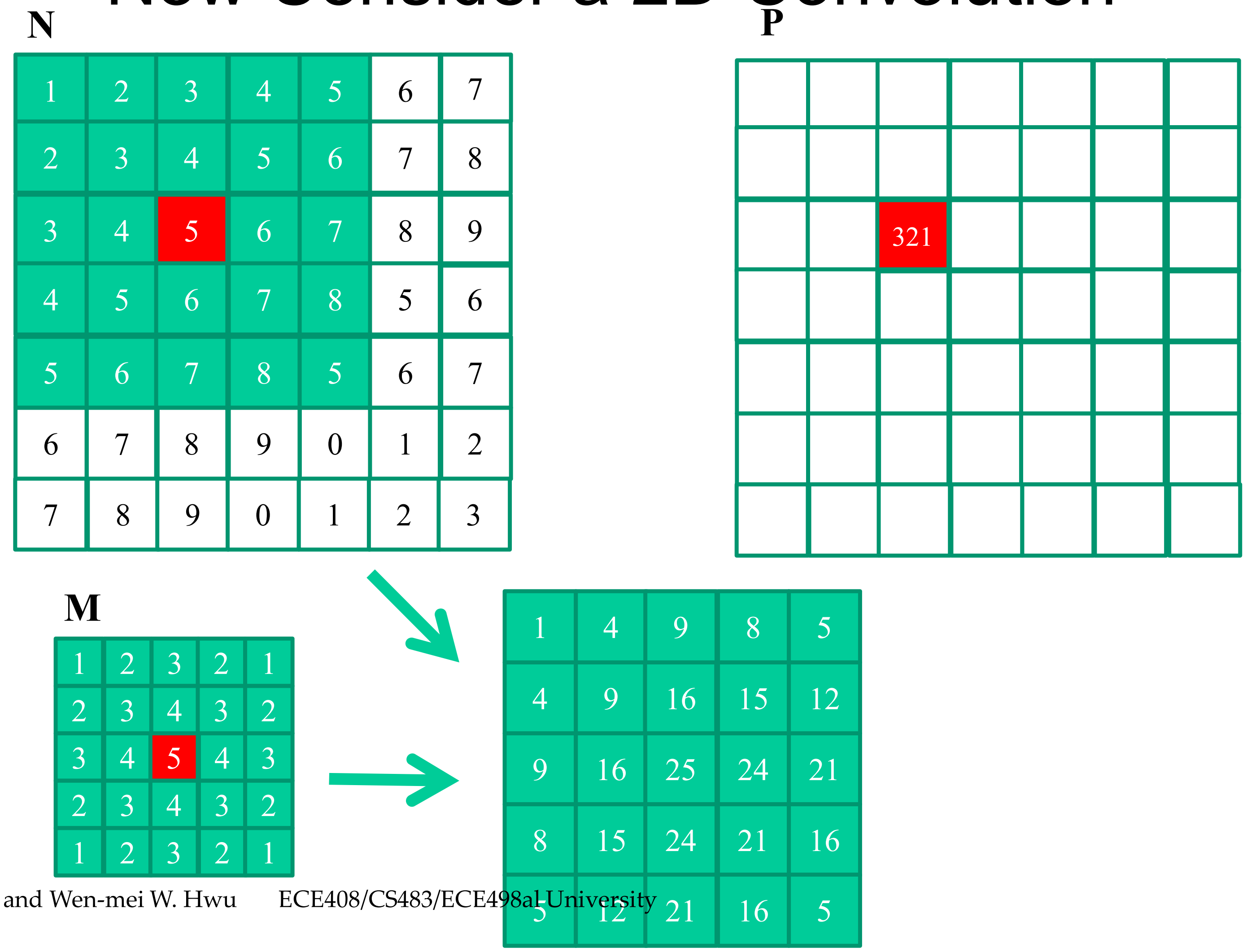
```
__global__ void convolution_1D_basic_kernel(float *N, float *M, float *P,
  int Mask_Width, int Width) {

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

  float Pvalue = 0;
  int N_start_point = i - (Mask_Width/2);
  for (int j = 0; j < Mask_Width; j++) {
    if (N_start_point + j >= 0 && N_start_point + j < Width) {
      Pvalue += N[N_start_point + j]*M[j];
    }
  }
  P[i] = Pvalue;

}
```

Now Consider a 2D Convolution



N

1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	5	6
5	6	7	8	5	6	7
6	7	8	9	0	1	2
7	8	9	0	1	2	3

P

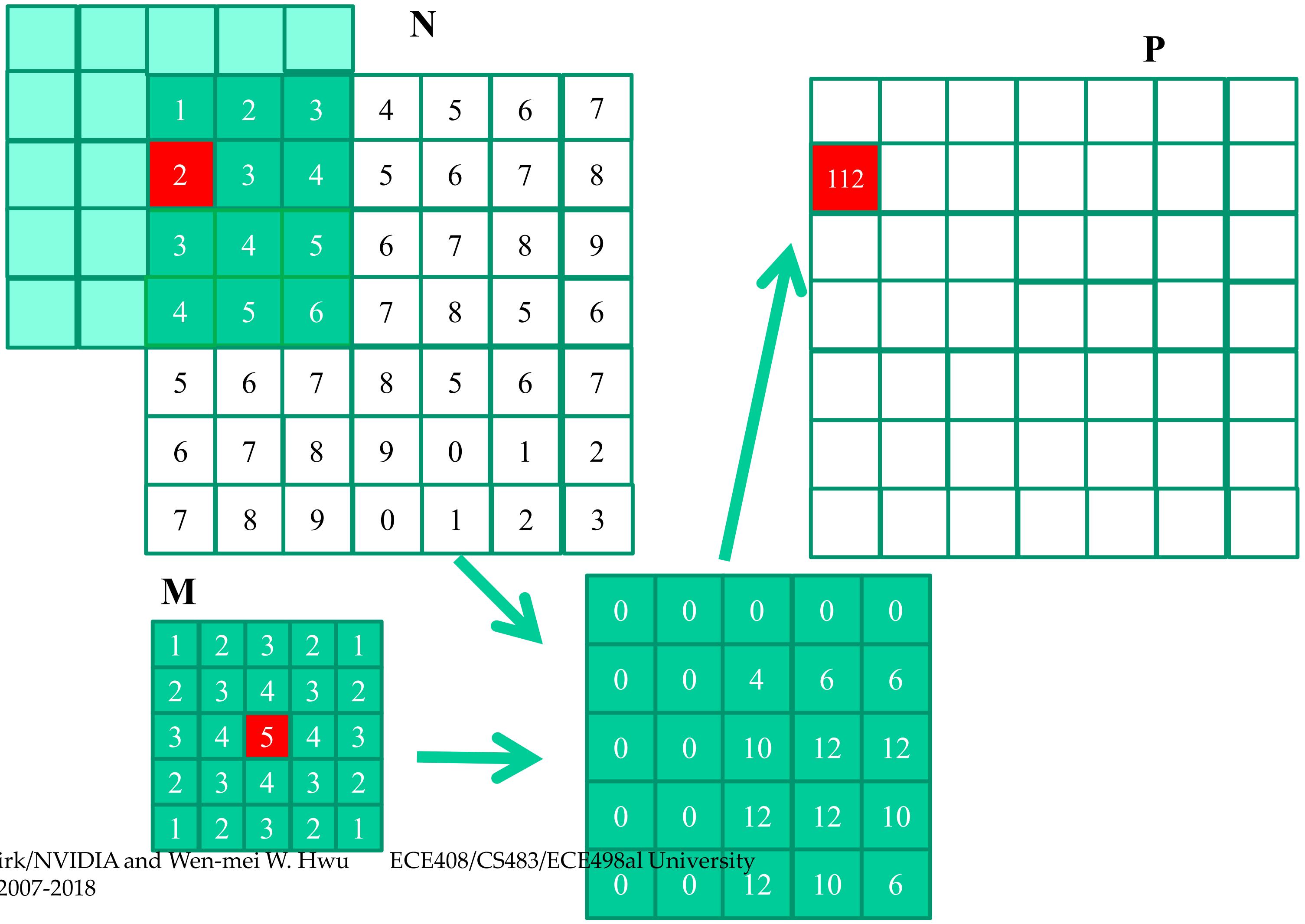
		321				

M

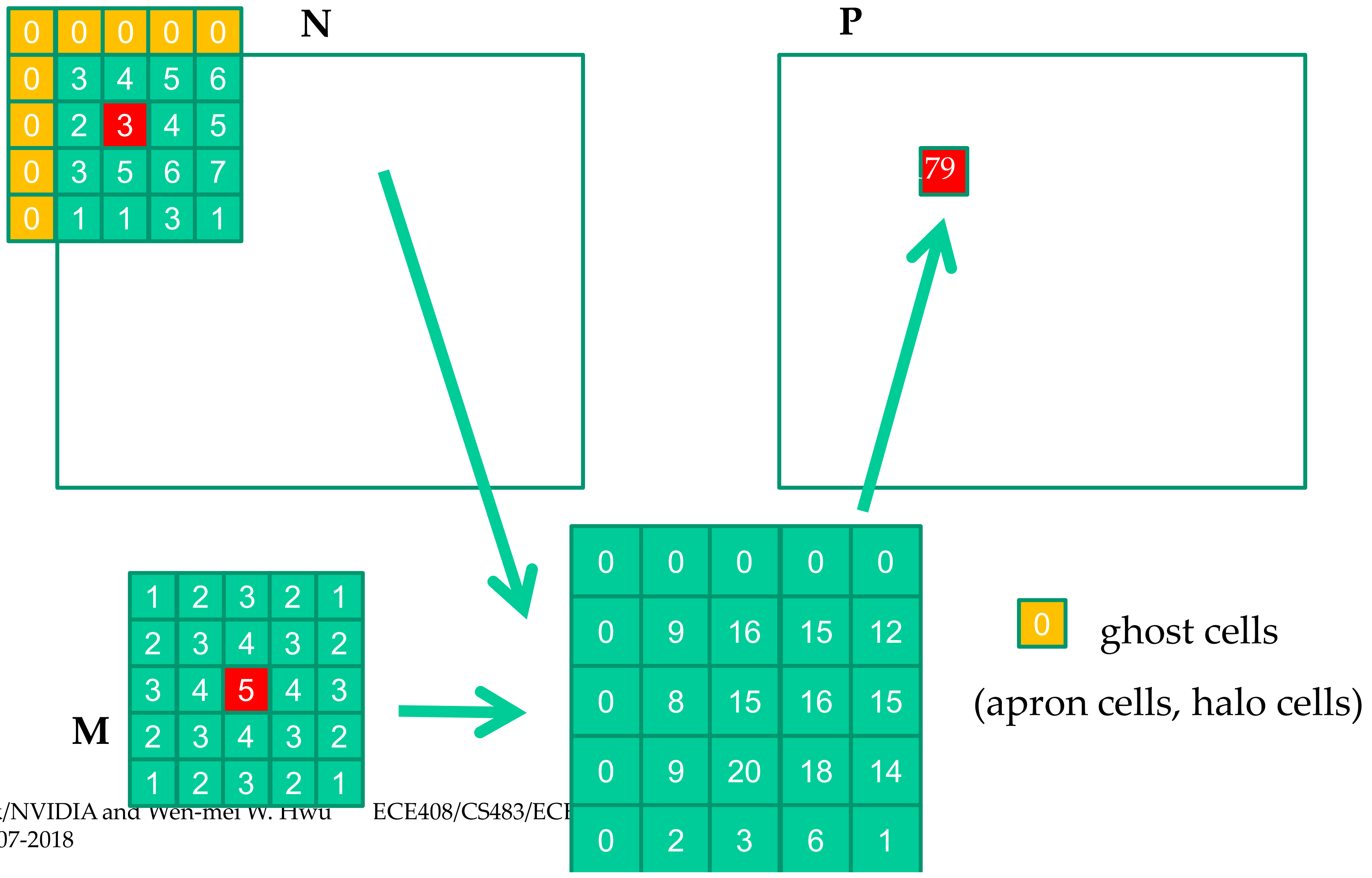
1	2	3	2	1
2	3	4	3	2
3	4	5	4	3
2	3	4	3	2
1	2	3	2	1

1	4	9	8	5
4	9	16	15	12
9	16	25	24	21
8	15	24	21	16
5	12	21	16	5

We'll Handle Boundaries in the Same Way



2D Convolution – Ghost Cells



Access Pattern for M

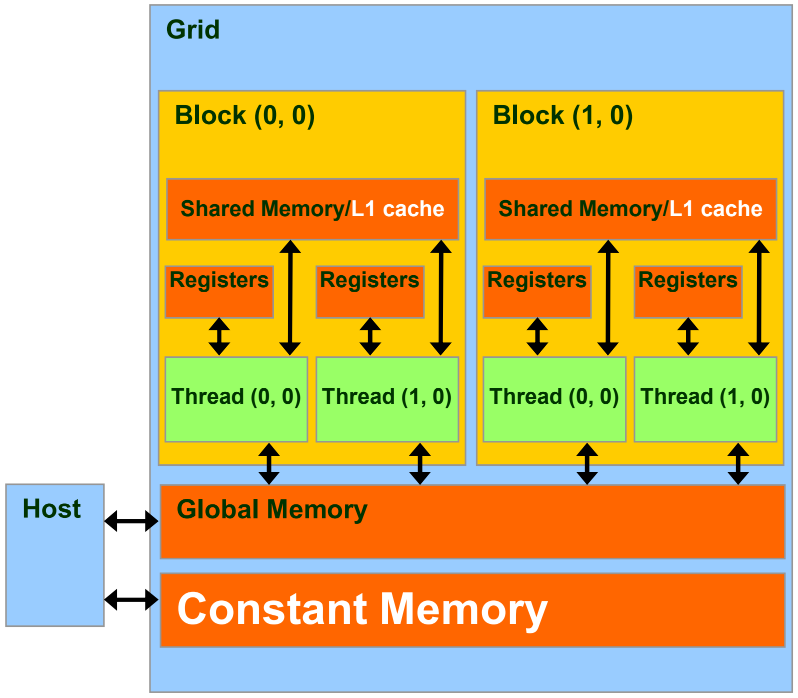
- Elements of M are called mask (kernel, filter) coefficients (weights)
  - Calculation of all output P elements needs M
  - M is not changed during grid execution

- Bonus - M elements are accessed in the same order when calculating all P elements

- M is a good candidate for Constant Memory

Review: Programmer View of CUDA Memories

- Each thread can:
  - Read/write per-thread **registers (~1 cycle)**
  - Read/write per-block **shared memory (~5 cycles)**
  - Read/write per-grid **global memory (~500 cycles)**
  - Read/only per-grid **constant memory (~5 cycles with caching)**



Memory Hierarchies

- Remember:
  - kernel execution time can be limited by global memory bandwidth, so
  - we need to reuse data brought onto the chip from memory.
- With matrix multiplication
  - we were able to leverage a scratchpad, shared memory,
  - and implement a tiled version of the algorithm.

- Another important solution: Caches

Caches Store Lines of Memory

Recall: memory bursts

- contain around **1024 bits** (**128B**) from
- consecutive (linear) addresses.
- Let's call a single burst a **line**.

What's a **cache**?

- An **array of cache lines** (and tags).
- Memory **read produces** a **line**,
- **cache stores** a **copy** of the line, and
- tag records line's memory address.

Memory Accesses Show Locality

An executing program

- loads and store data from memory.
- **Consider sequence of addresses** accessed.

Sequence usually **shows** two types of **locality**:

- **spatial**: accessing **X implies** accessing **X+1** (and X+2, and so forth) **soon**
- **temporal**: accessing **X** implies accessing **X again soon**

(Caches improve performance for both types.)

Caches Can’t Hold Everything

Caches are smaller than memory.

When cache is full,

- must make room for new line,
- usually by **discard**ing **least recently used line**.

GPU Has Constant and L1 Caches

To support writes (modification of lines),

- **changes** must be **copied back to memory**, and
- cache must **track** modification **status**.
- **L1 cache** in GPU (for global memory accesses) **supports writes**.

Cache for constant / texture **memory**

- Special case: **lines are read-only**
- Enables higher-throughput access than L1
  for common GPU kernel access patterns.

Cache vs. Scratchpad (GPU Shared Mem.)

- Caches vs. shared memory
  - Both on chip*, with similar performance
  - (As of Volta generation, both using the same physical resources, allocated dynamically!)

What's the difference?

- **Programmer controls shared memory** contents (called a scratchpad)
- **Microarchitecture** automatically **determines contents of cache**.

*Static RAM, not DRAM, by the way—see ECE120/CS233.

How to Use Constant Memory

Host code is **similar** to previous versions, but…

Allocate device memory for **M** (the mask)

- **outside of all functions**
- **using `__constant__`**
  (tells GPU that caching is safe).

For copying to device memory, **use**

- **cudaMemcpyToSymbol(dst, src, size)**
- with destination defined as above.

Example of Host Code

(MASK_WIDTH is the size of the mask.)

```
// outside of any kernel/function
static __constant__ float Mc[MASK_WIDTH][MASK_WIDTH];

// allocate N, P, initialize N elements, copy N to Nd

// in host code:
    float* M; // host memory copy of mask
    // initialize M
    cudaMemcpyToSymbol(Mc, M,
        MASK_WIDTH * MASK_WIDTH * sizeof(M[0]));
    ConvolutionKernel<<<dimGrid, dimBlock>>>(Nd, Pd);
    // (note that file-scope Mc is visible to kernel)
```

QUESTIONS?

READ CHAPTER 7!

Problem Solving

Recall that caches can benefit from both spatial and temporal locality. Let’s think about the order in which we make use of a 2D mask array. Specifically, which order is a better choice for performance?

1. For every row R { For every column C { read $\text{mask}_{R,C}$ } }
2. For every column C { For every row R { read $\text{mask}_{R,C}$ } }

A: Remember that in C/CUDA, arrays are laid out in row-major order, meaning that elements in adjacent columns are adjacent in memory and offer spatial locality, making #1 the best choice.

(Elements in adjacent rows are NOT adjacent in memory.)

Problem Solving

Making good use of computation resources on a GPU requires using each datum loaded from memory many times. Consider a 2D convolution using a 5×5 mask array. Given an element of the input data chosen far from all boundaries, how many output data elements require the given input element as part of their computation?

A: This question requires mentally flipping the problem around: rather than looking at the input set that affects an output, we have to go the other way.

The relationship is symmetric, though: for each mask element, the given input must be multiplied with that mask element to compute some unique output element, thus we have one dependent output for each mask element, for a total of 5×5 = 25.