ECE408/CS483/CSE408 Fall 2022

Applied Parallel Programming

Lecture 7: Convolution, Constant Memory and Constant Caching

Course Reminders

- Lab updates
  - We are grading Lab 2 now, expect it to be graded soon
  - Lab 3 is due this Friday
- Take a note of Midterm 1 date
  - Details to be provided soon

Objective

- To learn convolution, an important parallel computation pattern
  - Widely used in signal, image and video processing
  - Foundational to stencil computation used in many science and engineering applications
  - Critical component of Neural Networks and Deep Learning

- Important GPU technique
  - Taking advantage of cache memories

Convolution Mathematics

$$f(x) * g(x) \quad = \quad \int_{-\infty}^{\infty} f(\tau) \cdot g(x-\tau)\, d\tau$$

$$f[x] * g[x] \quad = \quad \sum_{k=-\infty}^{\infty} f[k] \cdot g[x-k]$$

Convolution Applications

- A popular operation that is used in various forms in signal processing, digital recording, image processing, video processing, computer vision, and machine learning.

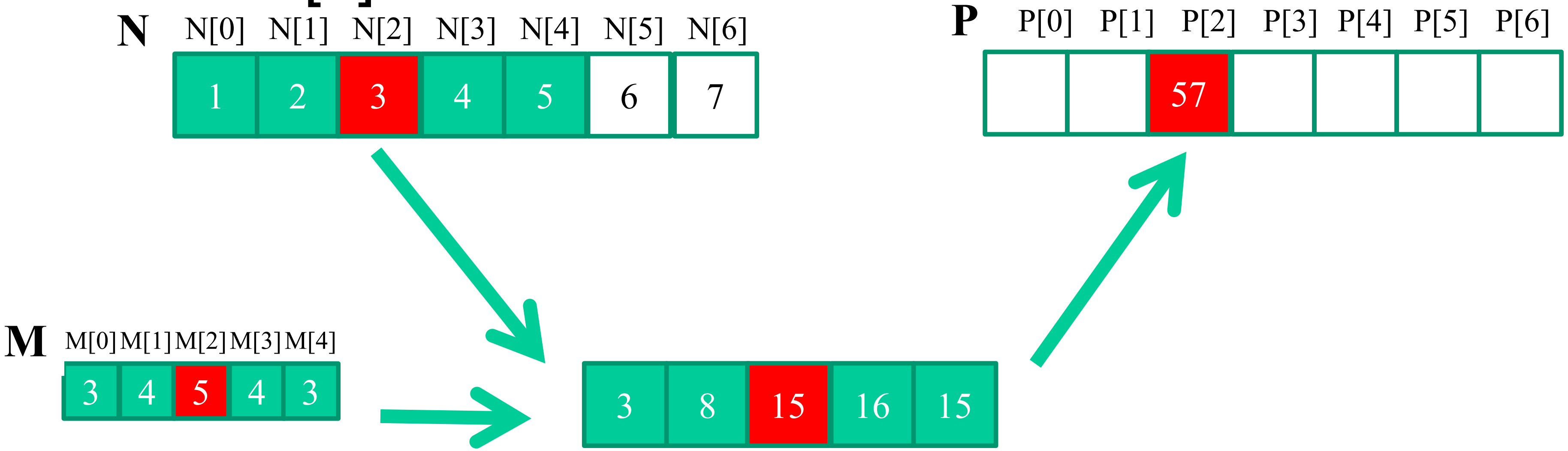
- Convolution is often performed as a **filter** that transforms the input signal (audio, video, etc) in some context-aware way.
  – Some filters smooth out the signal values so that one can see the big-picture trend
  – Or Gaussian filters to blur images, backgrounds

Convolution Computation

- An array operation where each output data element is a weighted sum of a collection of neighboring input elements
- The weights used in the weighted sum calculation are defined by an input mask array, commonly referred to as the *convolution kernel*
  - We will refer to these mask arrays as convolution masks or convolution filters to avoid confusion.
  - The same convolution mask is typically used for all elements of the array.

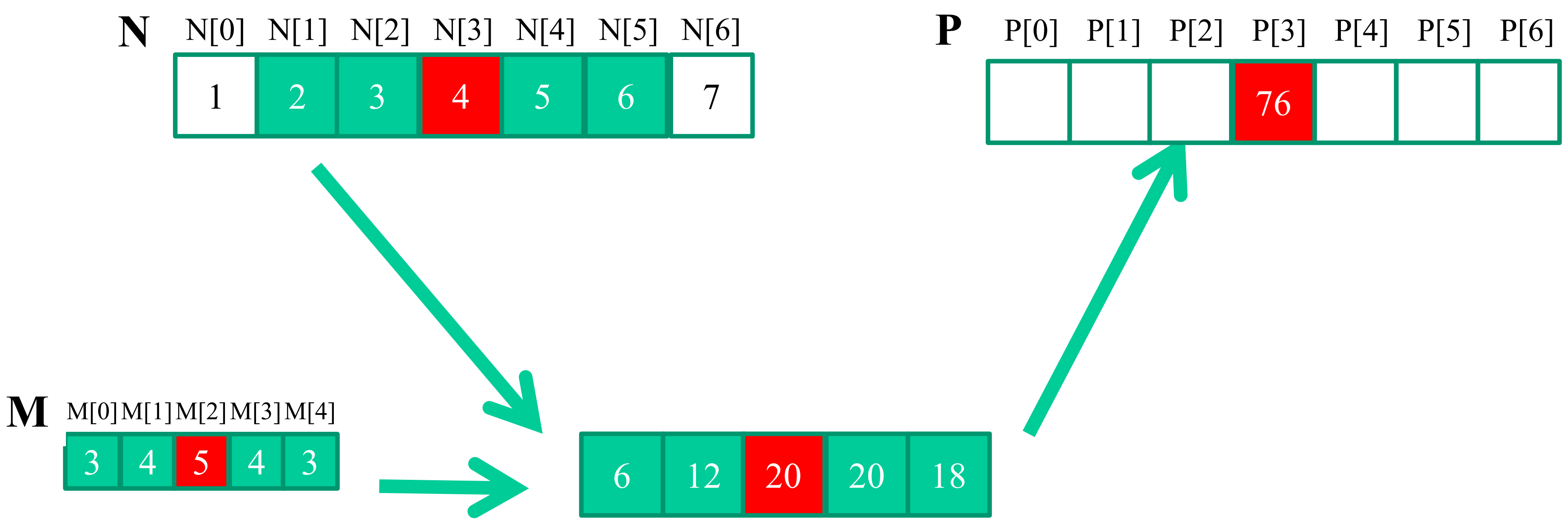
1D Convolution Example

- Commonly used for audio processing
  - MASK_WIDTH is usually an odd number of elements for symmetry (5 in this example)
  - MASK_RADIUS is the number of elements used in convolution on each side of the pixel being calculated (2 in this example).
- Calculation of P[2]:



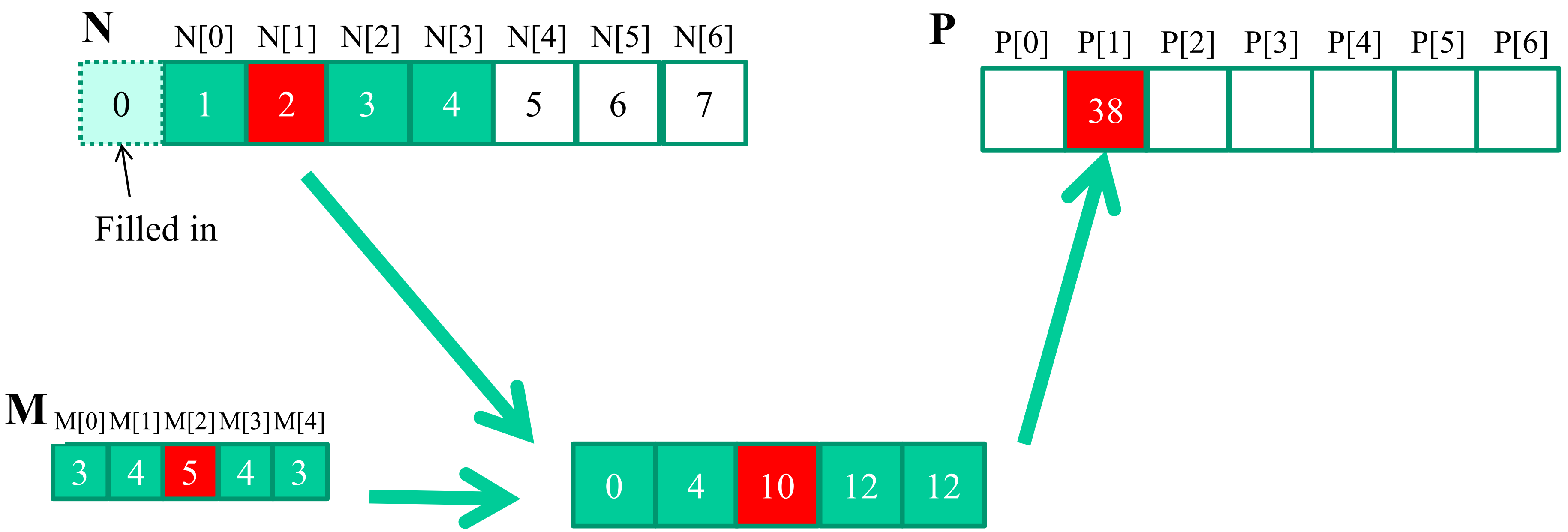
1D Convolution Example

- Calculation of P[3]



1D Convolution Boundaries

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



A 1D Convolution Kernel with Boundary Handling

- This kernel forces all elements outside the valid 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;
}
```

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

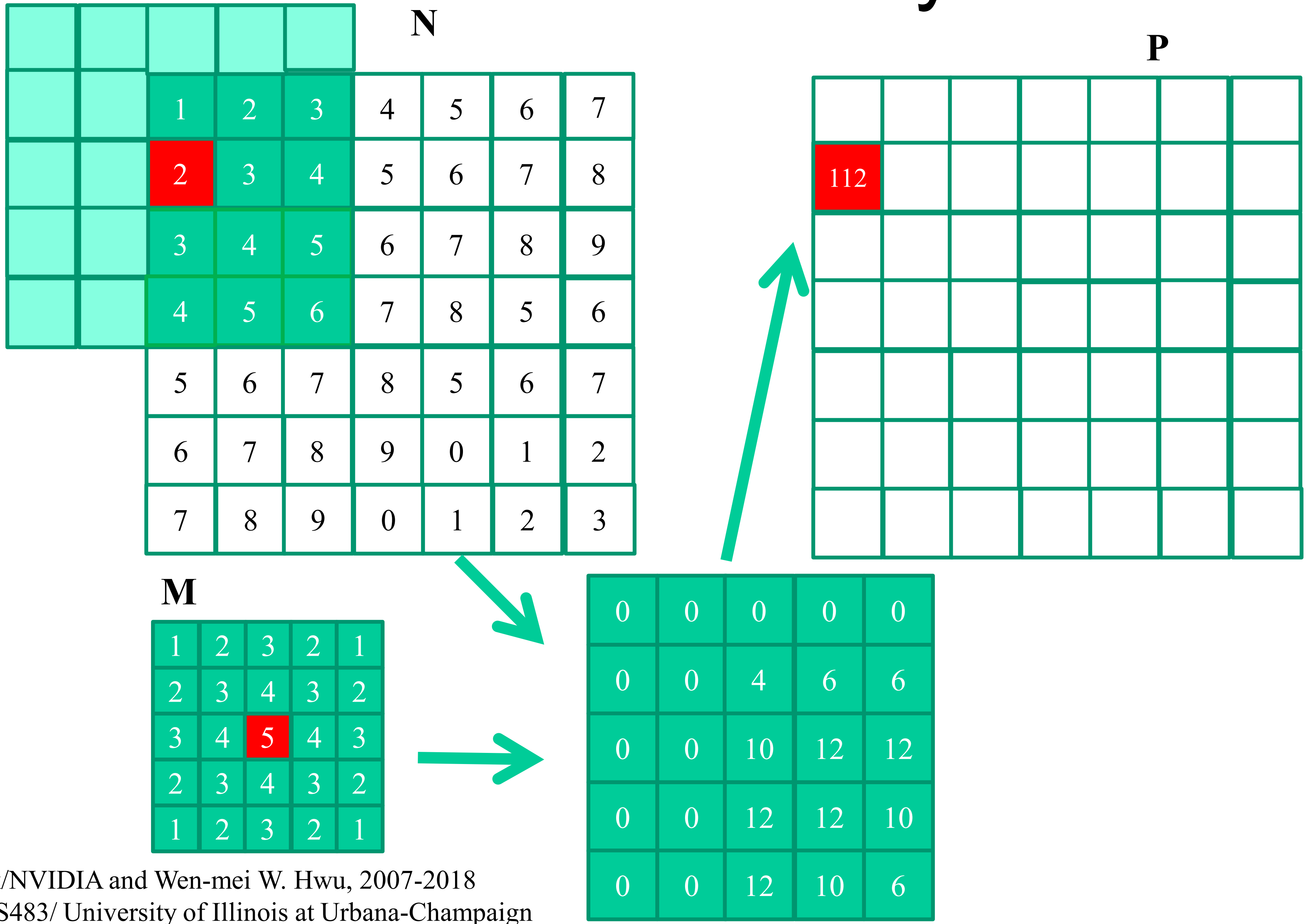
		321				

M

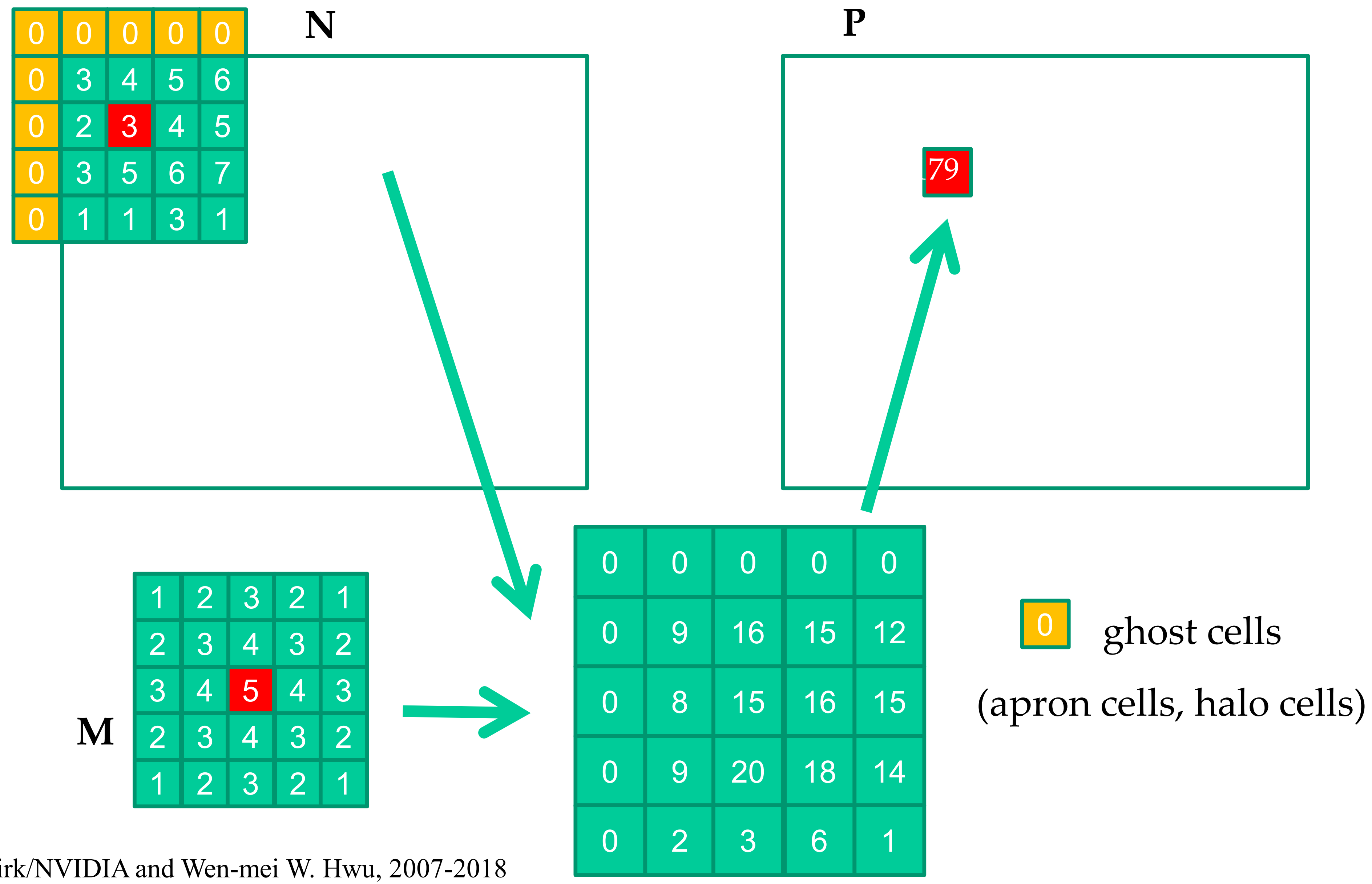
1	2	3	2	1
2	3	4	3	2
3	4	5	4	3
2	3	4	3	2
1	2	3	2	1

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

2D Convolution Boundary Condition



2D Convolution – Ghost Cells



What does this kernel accomplish?

$$\mathbf{M} = \frac{\mathbf{1}}{\mathbf{273}} \times \begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}$$

Hint: Assume input N is an image

Access Pattern for M

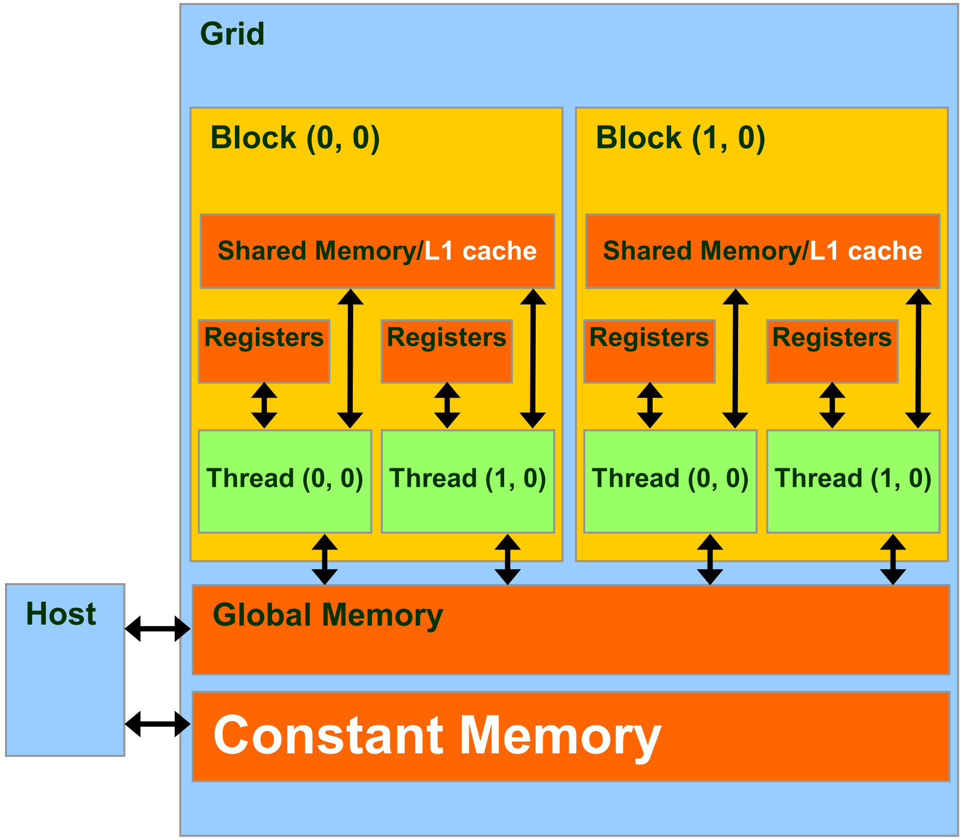
- Elements of M are called mask (kernel, filter) coefficients (weights)
  - Calculation of all output P elements needs M
  - M is not changed during grid execution

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

- M is a good candidate for Constant Memory

Programmer View of CUDA Memories (Review)

- 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

- Review: If we had to go to global memory to access data all the time, the execution speed of GPUs would be limited by the global memory bandwidth
  – We saw the use of shared memory in tiled matrix multiplication to reduce this limitation

- Another important solution: Caches

Cache

- A cache is an “array” of cache lines
  – A cache line can usually hold data from several consecutive memory addresses

- When data is requested from the global memory, an entire cache line that includes the data being accessed is loaded into the cache, in an attempt to reduce global memory requests
  – The data in the cache is a “copy” of the original data in global memory
  – Additional hardware is used to remember the addresses of the data in the cache line

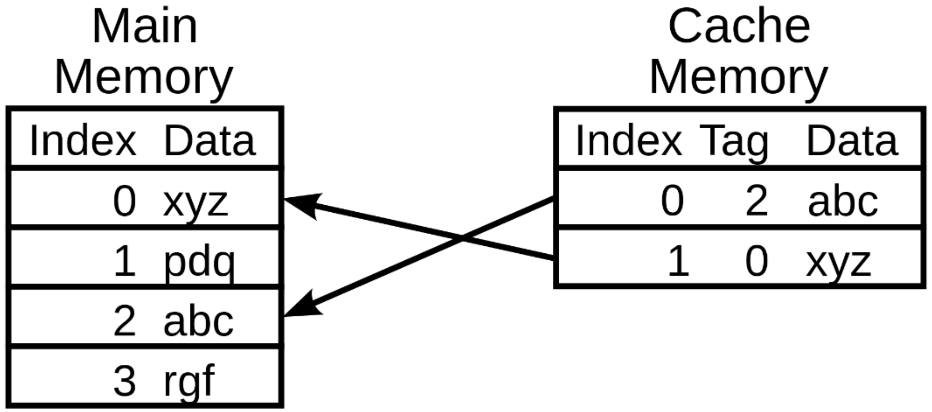
Caches Store Lines of Memory

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.



Caches and Locality

- Some definitions:
  – Spatial locality: when the data elements stored in consecutive memory locations are accessed consecutively
  – Temporal locality: when the same data element is accessed multiple times in short period of time

- Both spatial locality and temporal locality improve the performance of caches

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**:

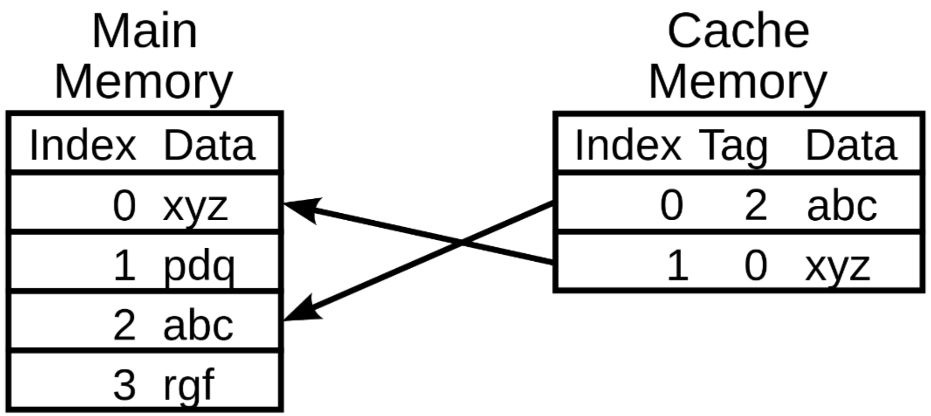
– **spatial**: accessing **X implies** accessing **X+1** (and X+2, and so forth) **soon**

– **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**.

Shared Memory vs. Cache

- Shared memory in CUDA is another type of temporary storage used to relieve main memory contention.
  - In terms of distance from the SMs, shared memory is similar to L1 cache.
- Unlike cache, shared memory does not necessarily hold a copy of data that is also in main memory
  - Shared mem requires explicit data transfer instructions into locations in the shared mem, whereas cache doesn't

Shared Memory vs. Cache - Cont'd

- 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.

Constant Cache in GPUs

- Modification to cached data needs to be (eventually) reflected back to the original data in global memory
  - Requires logic to track the modified status, etc.

- Constant cache is a special cache for constant data that will not be modified during kernel execution by a grid
  - Data declared in the constant memory not modified during kernel execution.
  - Constant cache can be accessed with higher throughput than L1 cache for some common patterns

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.

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(dest, src, size, offset = 0, kind = cudaMemcpyHostToDevice)**
- with destination defined as above.

Host Code Example

(MASK_WIDTH is the size of the mask.)

```
// global variable, outside any kernel/function
__constant__ float Mc[MASK_WIDTH][MASK_WIDTH];

// Initialize Mask
float Mask[MASK_WIDTH][MASK_WIDTH]
for(unsigned int i = 0; i < MASK_WIDTH * MASK_WIDTH; i++) {
    Mask[i] = (rand() / (float)RAND_MAX);
    if(rand() % 2) Mask[i] = - Mask[i]
}
cudaMemcpyToSymbol(Mc, Mask, MASK_WIDTH*MASK_WIDTH*sizeof(float));

ConvolutionKernel<<<dimGrid, dimBlock>>>(Nd, Pd);
```

ANY MORE QUESTIONS? READ CHAPTER 7