# ECE408/CS483/CSE408

# Applied Parallel Programming

## Lecture 7:
## Convolution and Constant Memory

# Exam 1

- Now scheduled for **Tuesday Oct 10th from 7-9pm**.
- The exam will be in-person, on paper, in ECEB.
- Room assignments will be provided.
- Please inform us immediately if you have a legitimate conflict.

# 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 Convolutional Neural Networks (CNNs)

- Important GPU technique
  - Taking advantage of cache memories

# Convolution

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

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

# 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.

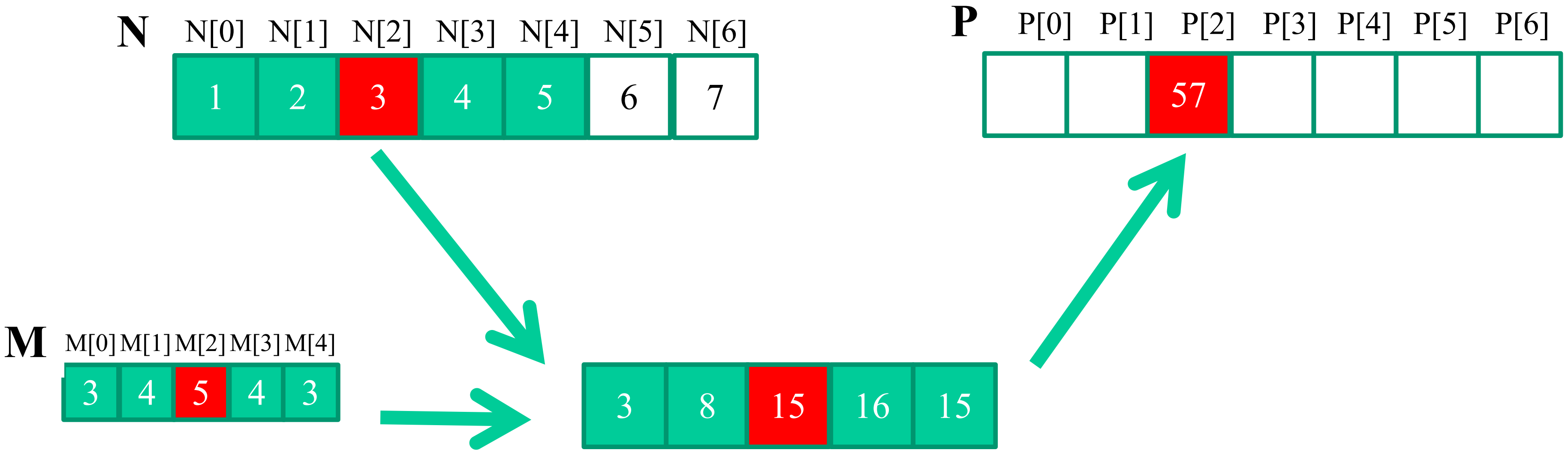
- Convolution is often performed as a **filter** that transforms the input signal (audio, video, etc) in some context-aware way.
  - Some filters smooth out the signal values so that one can see the big-picture trend
  - Or Gaussian filters to blur images, backgrounds

# 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*

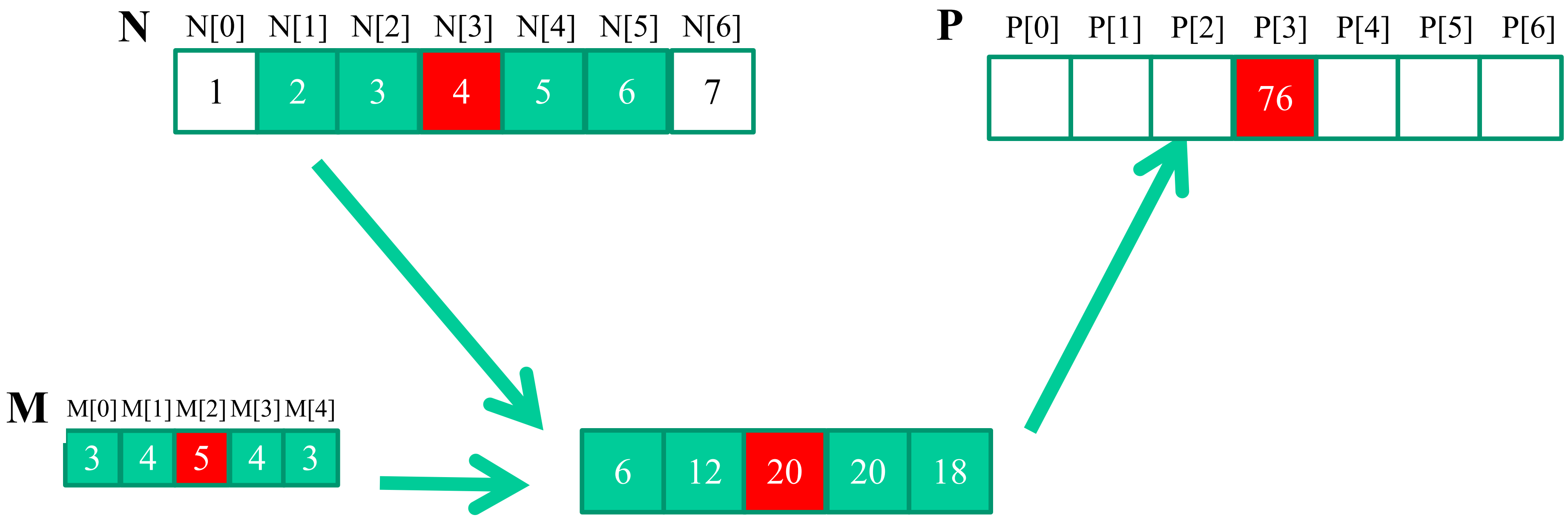
# 1D Convolution Example

- 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 on each side (2 in this example)
- Calculation of P[2]:



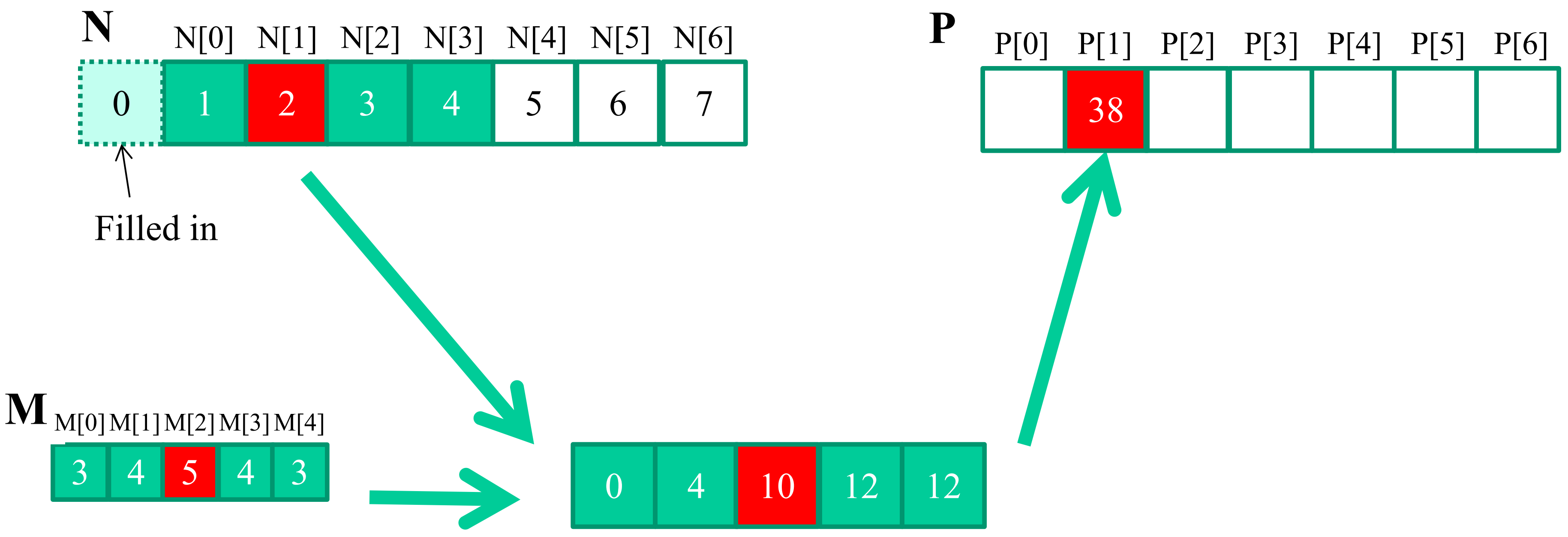
# 1D Convolution Example

- 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
- Each thread calculates one element of P

```
__global__
void convolution_1D_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**

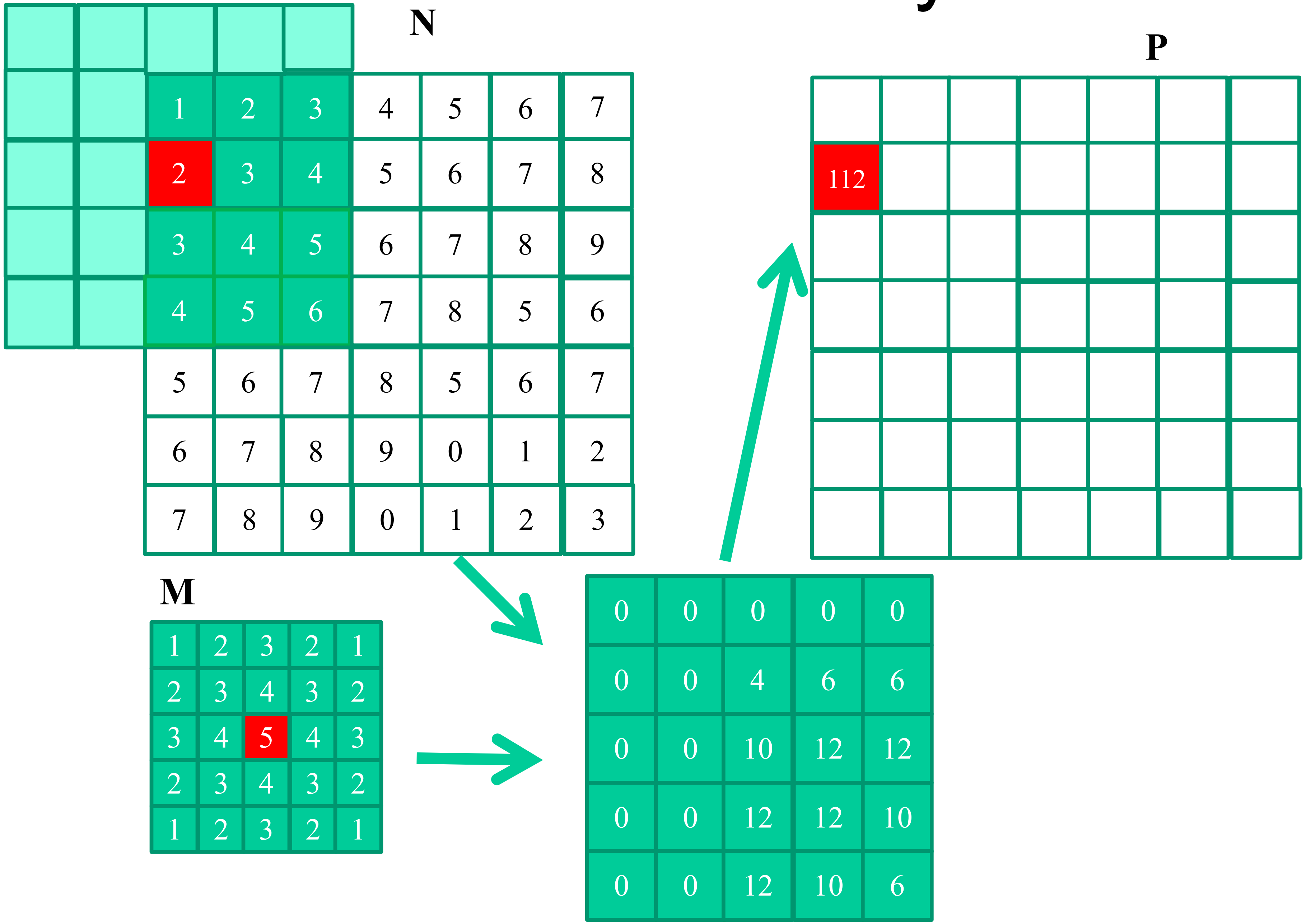
| | | | | | | |
|---|---|---|---|---|---|---|
| | | | | | | |
| | | | | | | |
| | | 321 | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |

**M**

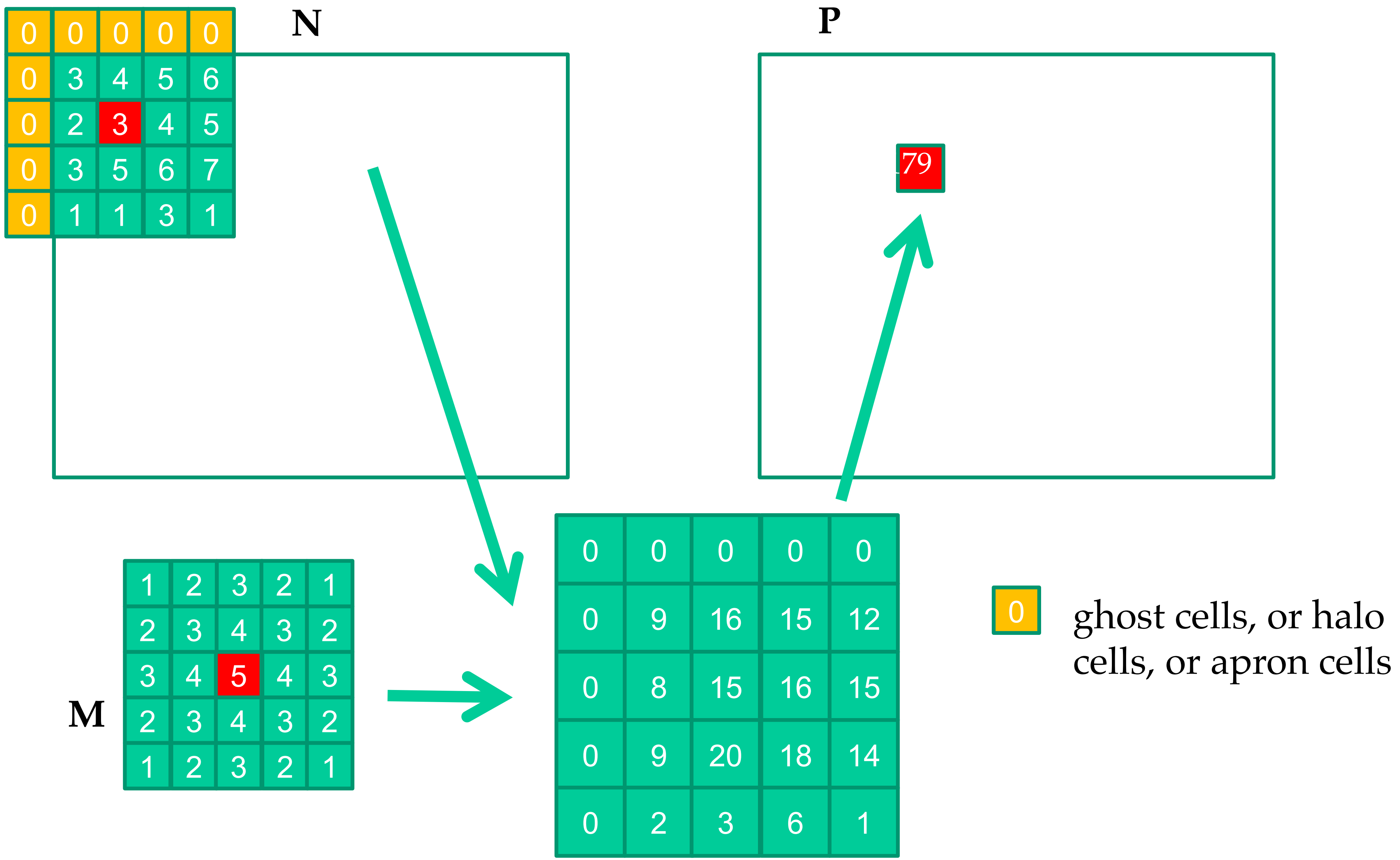
| | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | 2 | 1 |
| 2 | 3 | 4 | 3 | 2 |
| 3 | 4 | 5 | 4 | 3 |
| 2 | 3 | 4 | 3 | 2 |
| 1 | 2 | 3 | 2 | 1 |

| | | | | |
|---|---|---|---|---|
| 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 mask accomplish?

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

Assume input N is a grayscale image

# What does this mask accomplish?

$$M = \begin{bmatrix} -1 & -1 & 1 & -1 & -1 \\ -1 & -1 & 1 & -1 & -1 \\ -1 & -1 & 1 & -1 & -1 \\ -1 & -1 & 1 & -1 & -1 \\ -1 & -1 & 1 & -1 & -1 \end{bmatrix}$$

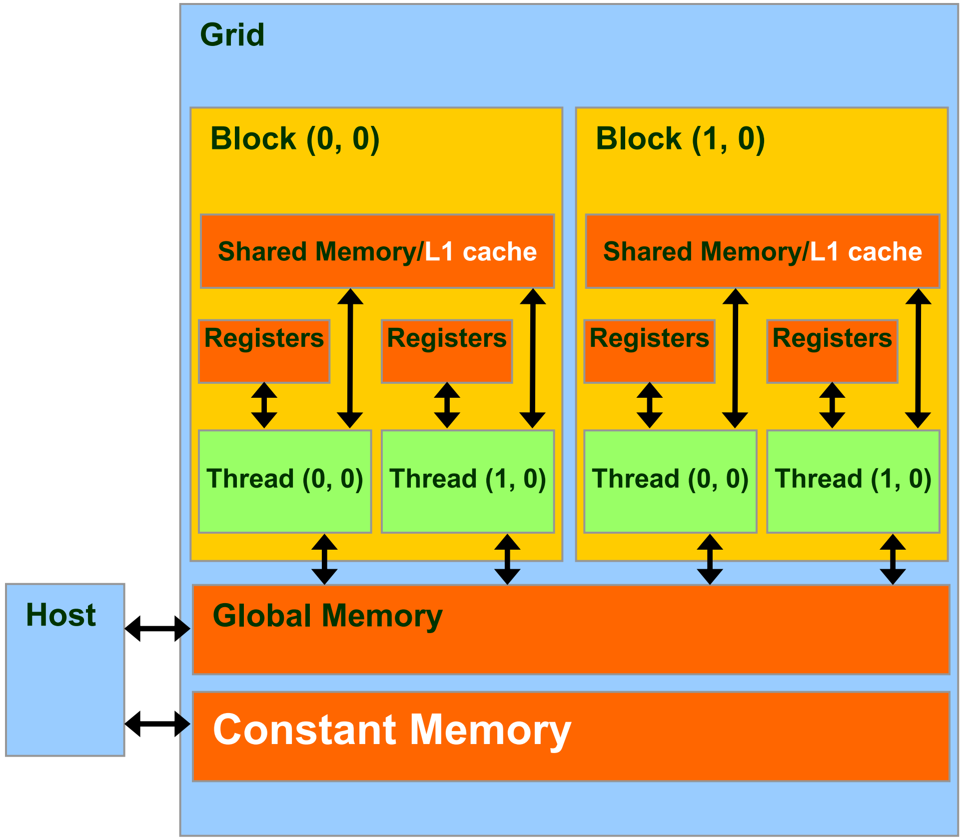
Assume input N is a grayscale image, what does the output P represent?

# Access Pattern for M

- Elements of M are called mask (kernel, filter) coefficients
  - Calculation of all elements of P need 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

- 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 all data were in global memory, the execution speed of GPUs would be limited by the global memory bandwidth

- We used shared memory in tiled matrix multiplication to reduce this limitation

- An important part of the solution: caches and constant memory

# HW Caches Store Lines of Memory

Recall: memory is optimized for bursts

- contain some number of bit, say **1024 bits** (**128B**)
- 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.

The **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.

# Shared Memory vs. Cache

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

What's the difference?

- Programmer controls shared memory contents (explicit)
- Hardware determines contents of cache (implicit).

# GPU Has Both 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 `__constant__`

# Host Code Example

```
// 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