CS/EE 217: GPU Architecture and Parallel Programming

Convolution, (with a side of Constant Memory and Caching)

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
- Taking advantage of cache memories

Convolution Applications

- A popular array operation used in signal processing, digital recording, image processing, video processing, and computer vision.
- Convolution is often performed as a filter that transforms signals and pixels into more desirable values.
  – Some filters smooth out the signal values so that one can see the big-picture trend
  – Others like Gaussian filters can be used to sharpen boundaries and edges of objects in images..

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 to avoid confusion.
  - The same convolution mask is typically used for all elements of the array.

Convolution definition

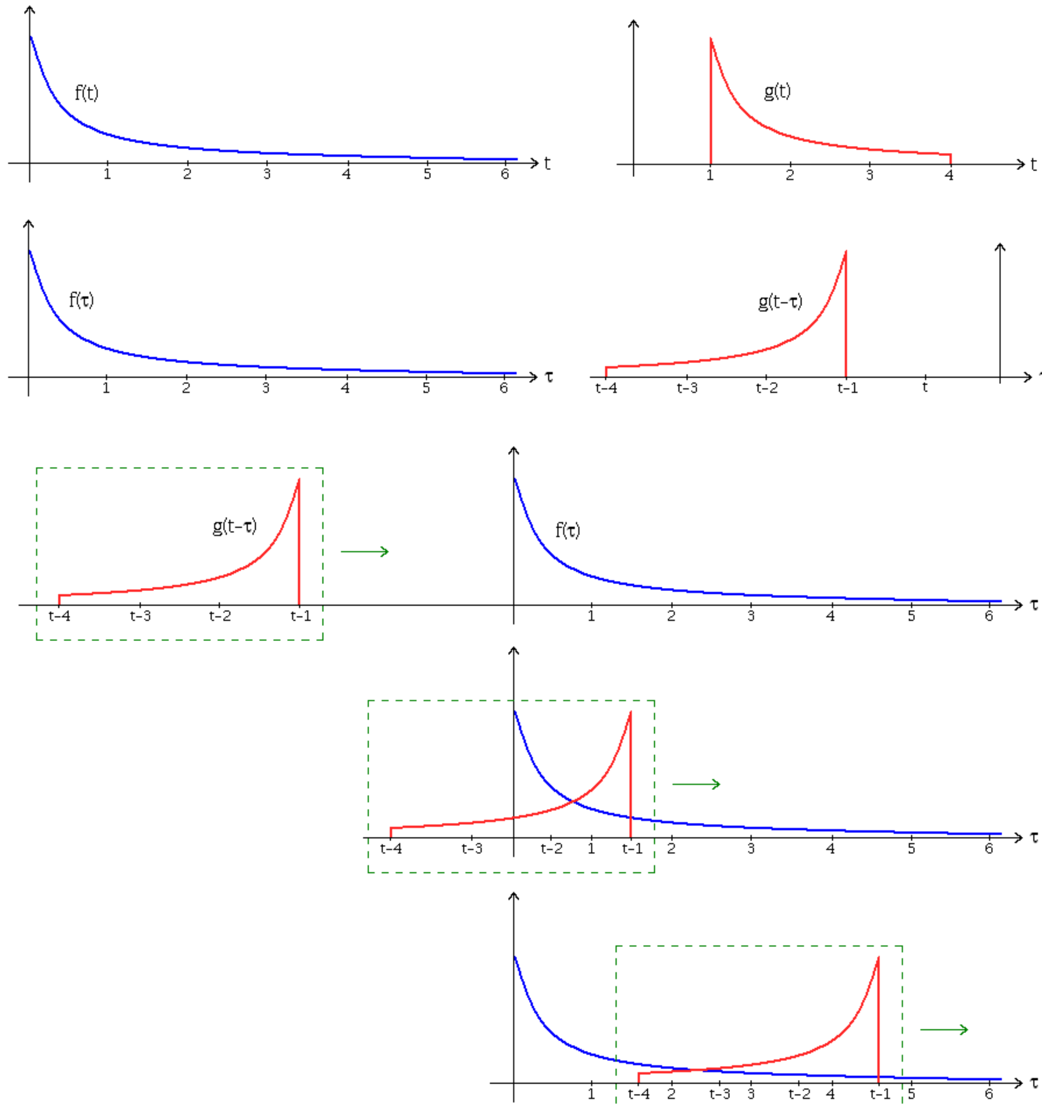
Convolution is to compute the response of linear time invariant system f(t) for the given input signal g(t).

$$(f * g)(t) = \int_0^t f(\tau)\, g(t-\tau)\, d\tau \quad \text{for} \quad f, g : [0, \infty) \to \mathbb{R}$$

In frequency domain

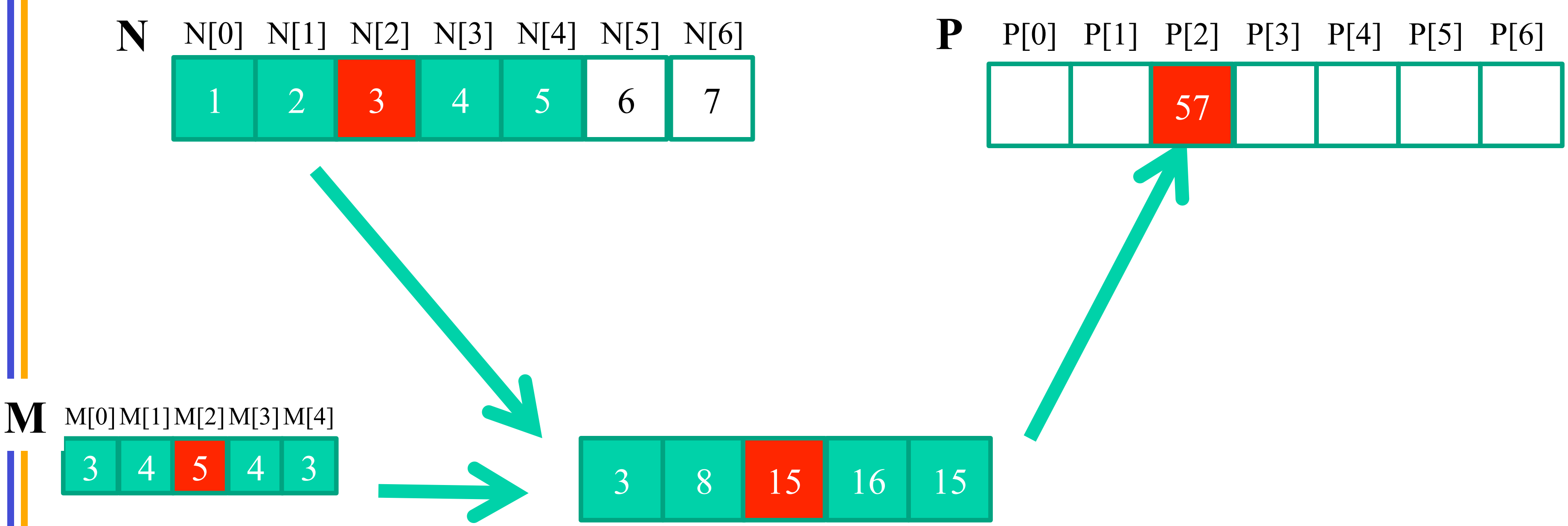
Convolution operation

- From the [Wikipedia page on convolution]



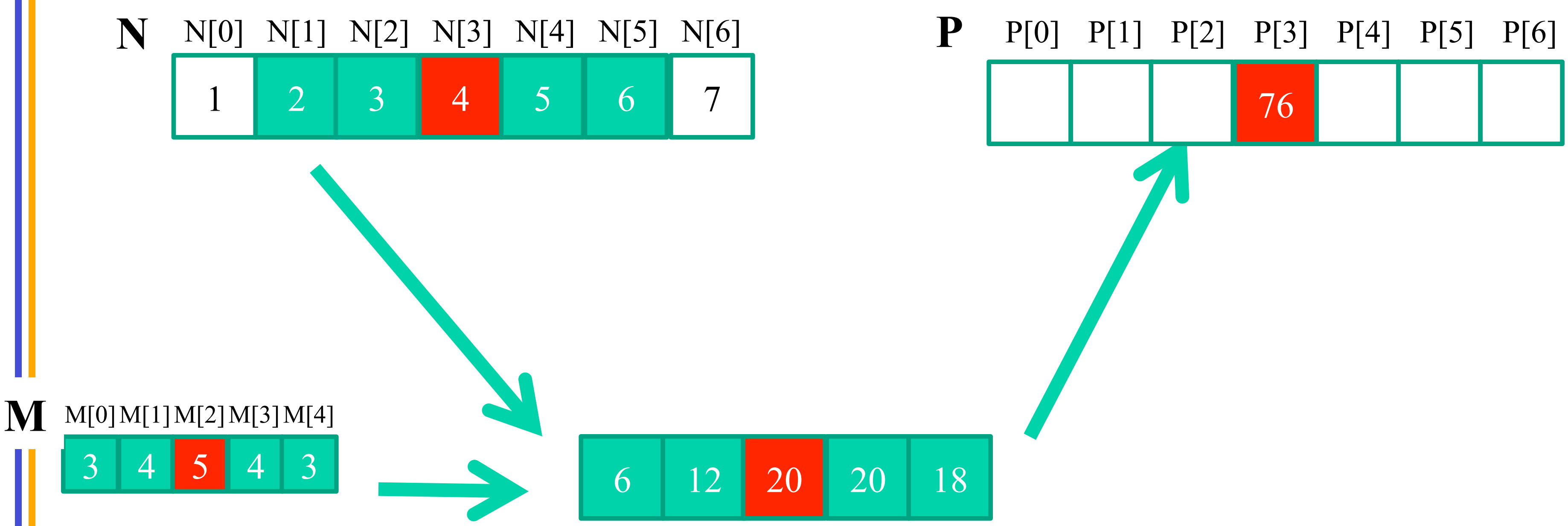
1D Convolution Example

- Commonly used for audio processing
  - Mask size is usually an odd number of elements for symmetry (5 in this example)
- Calculation of P[2]



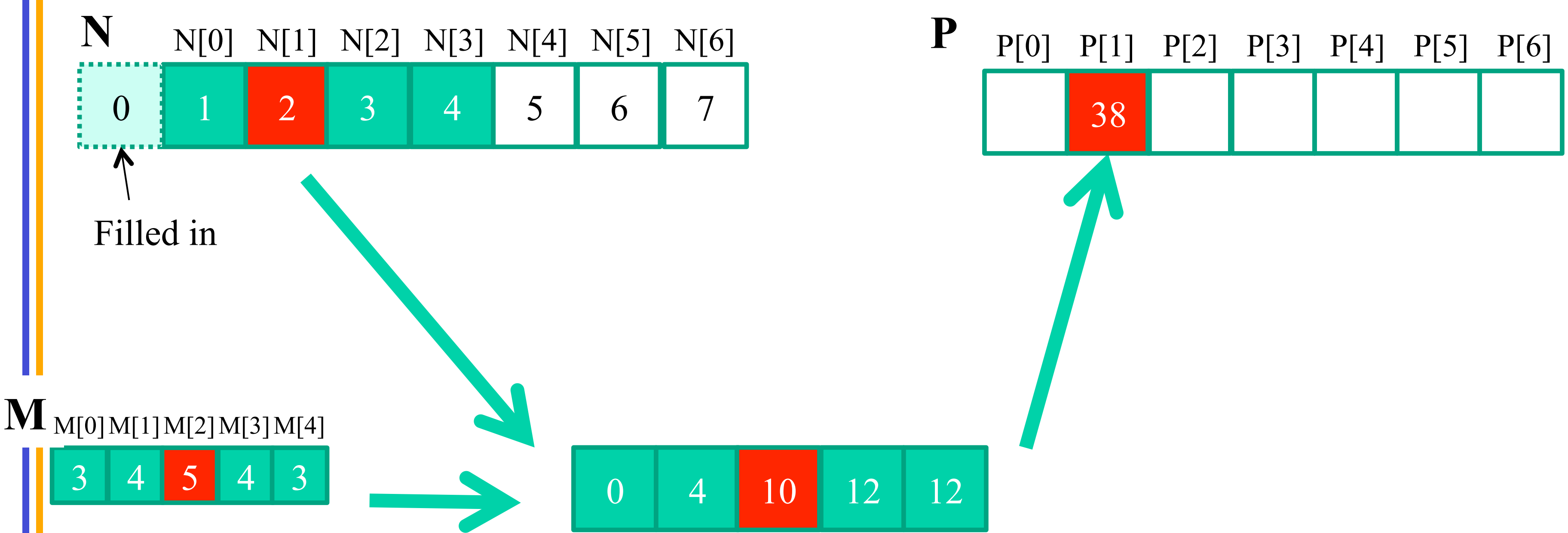
1D Convolution Example - more on inside elements

- Calculation of P[3]



1D Convolution Boundary Condition

- 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.)



A 1D Convolution Kernel with Boundary Condition Handling

- This kernel forces all elements outside the image 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

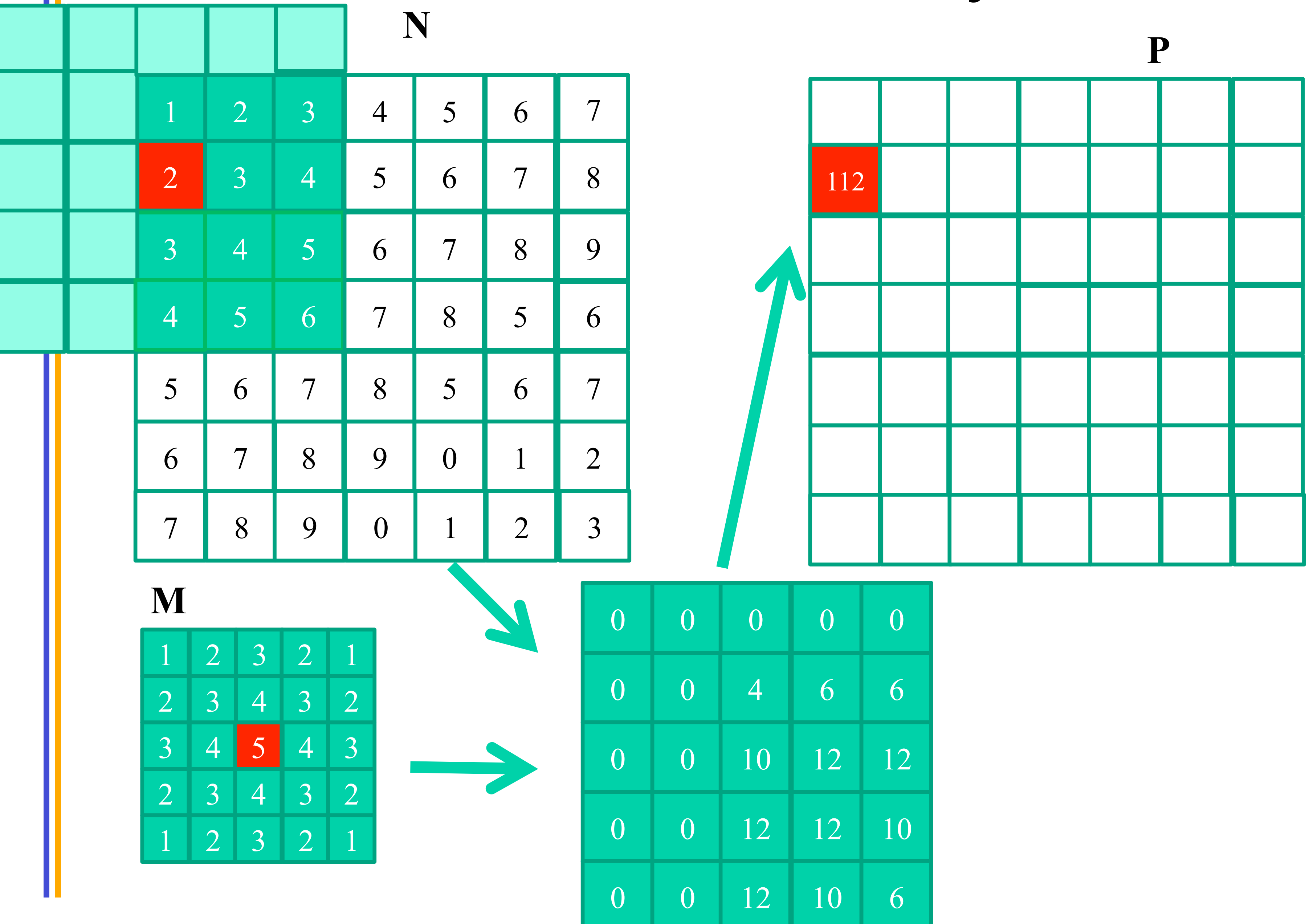
		321				

M

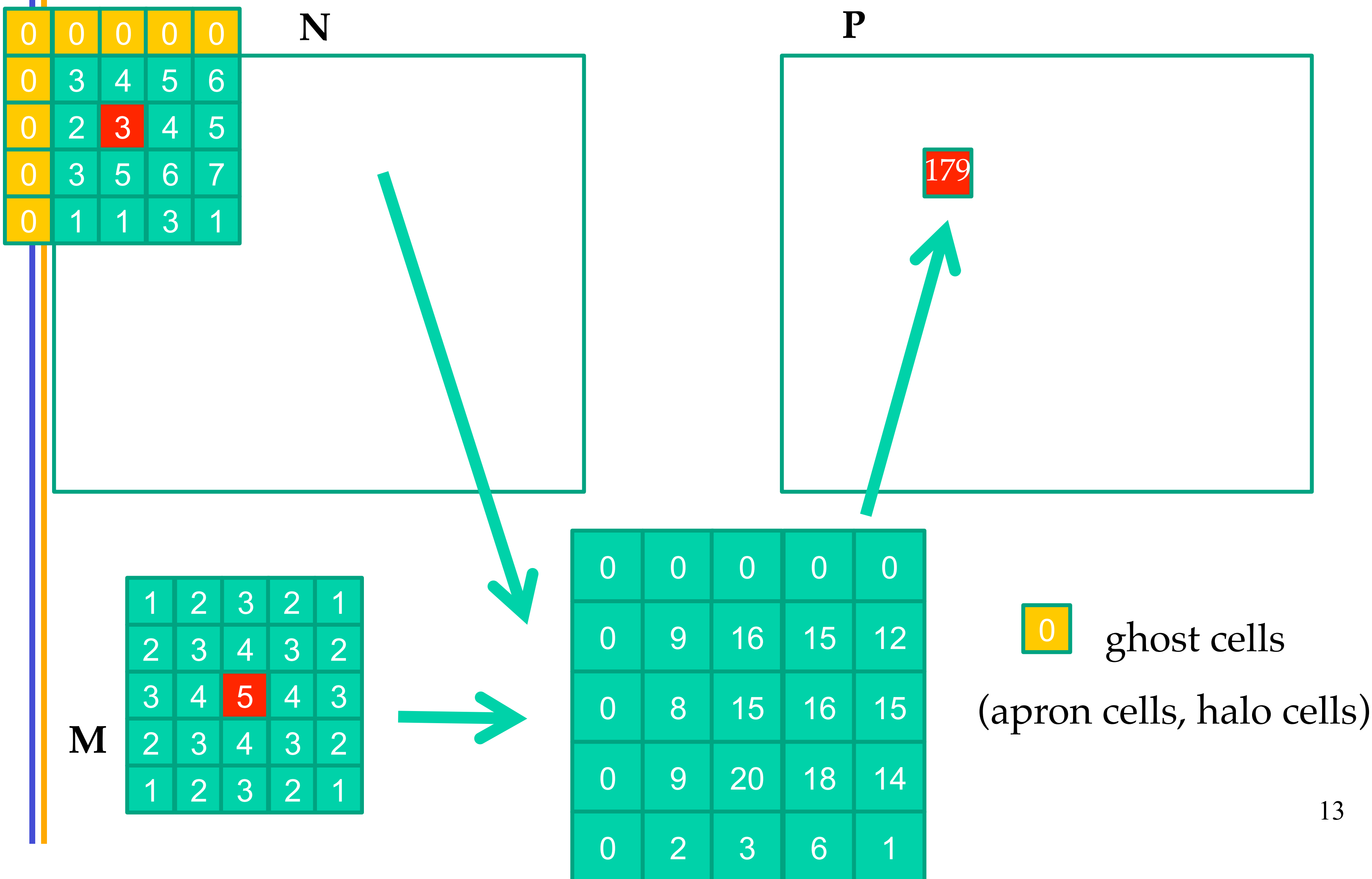
1	2	3	2	1
2	3	4	3	2
3	4	5	4	3
2	3	4	3	2
1	2	3	2	1

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



Access Pattern for M

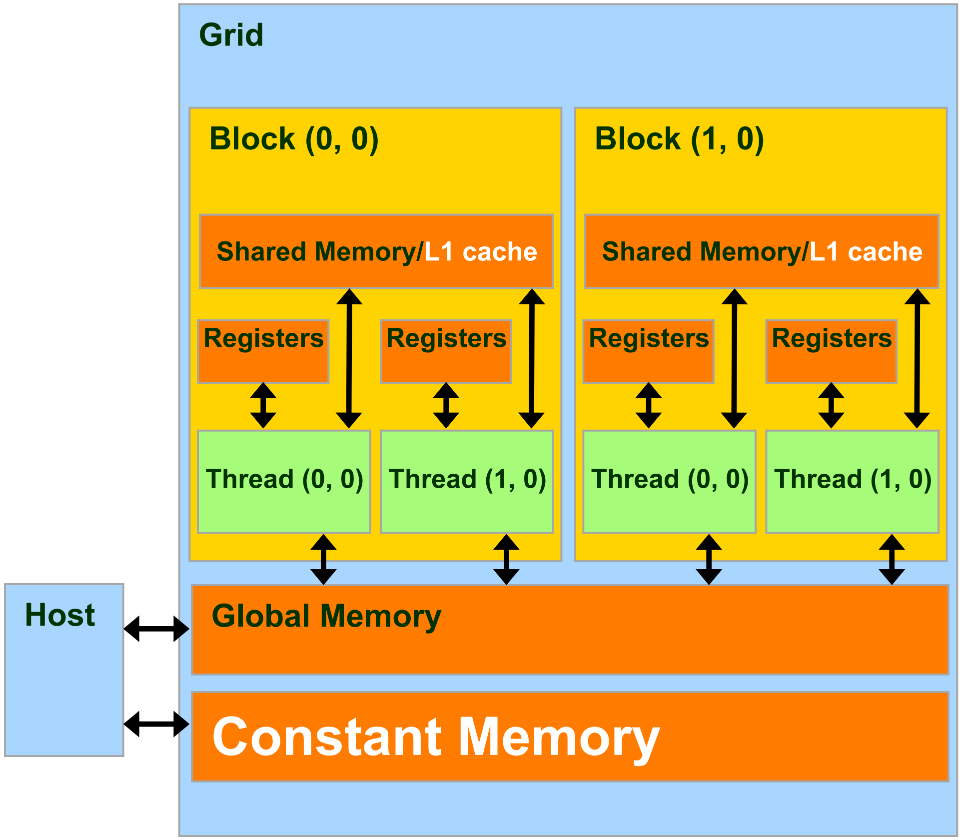
- M is referred to as mask (a.k.a. kernel, filter, etc.)
  - Elements of M are called mask (kernel, filter) coefficients
- Calculation of all output P elements need M
- M is not changed during kernel

- 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)**



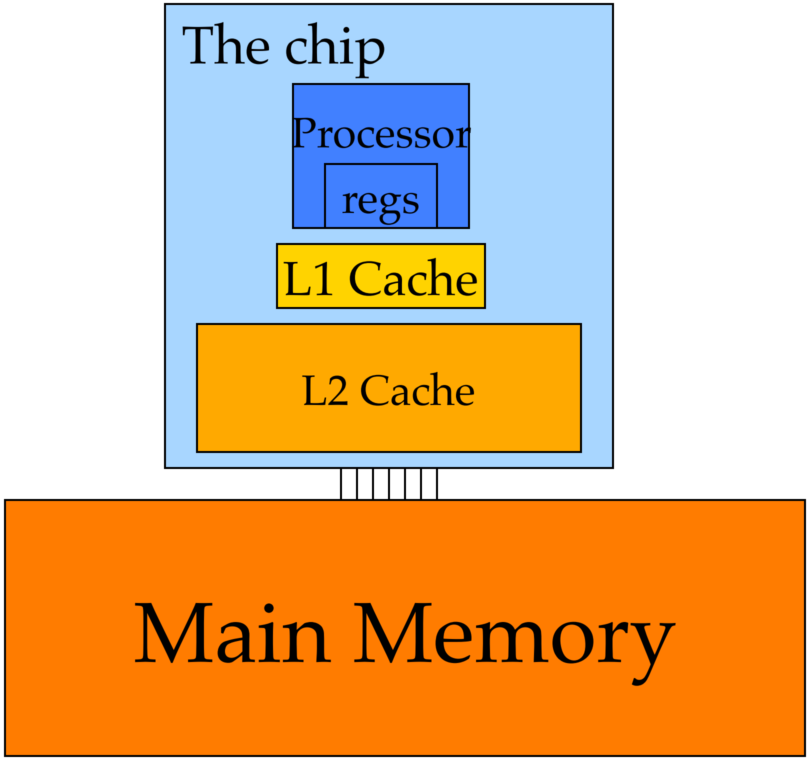
Memory Hierarchies

- If every time we needed a piece of data, we had to go to main memory to get it, computers would take a lot longer to do anything
- On today's processors, main memory accesses take hundreds of cycles

- One solution: Caches

Cache - Cont'd

- In order to keep cache fast, it needs to be small, so we cannot fit the entire data set in it



Cache - Cont’d

- Cache is unit of volatile memory storage
- A cache is an “array” of cache lines
- Cache line can usually hold data from several consecutive memory addresses
- When data is requested from memory, an entire cache line is loaded into the cache, in an attempt to reduce main memory requests

Caches - Cont'd

Some definitions:

– Spatial locality: is when the data elements stored in consecutive memory locations are access consecutively

– Temporal locality: is when the same data element is access multiple times in short period of time

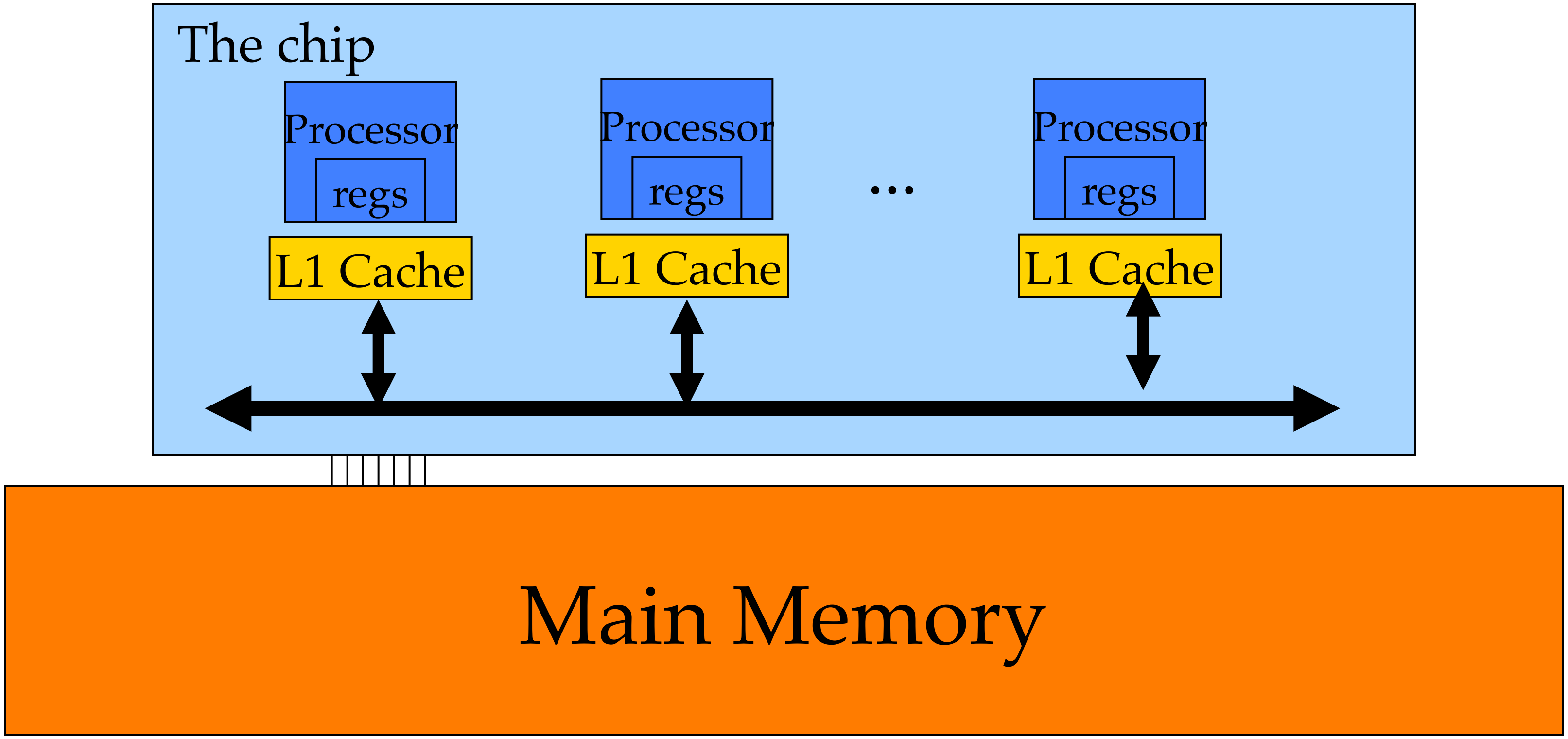
- Both spatial locality and temporal locality improve the performance of caches

Scratchpad vs. Cache

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

Cache Coherence Protocol

- A mechanism for caches to propagate updates by their local processor to other caches (processors)



CPU and GPU have different caching philosophy

- CPU L1 caches are usually coherent
  - L1 is also replicated for each core
  - Even data that will be changed can be cached in L1
  - Updates to local cache copy invalidates (or less commonly updates) copies in other caches
  - Expensive in terms of hardware and disruption of services (cleaning bathrooms at airports..)

- GPU L1 caches are usually incoherent
  - Avoid caching data that will be modified

How to Use Constant Memory

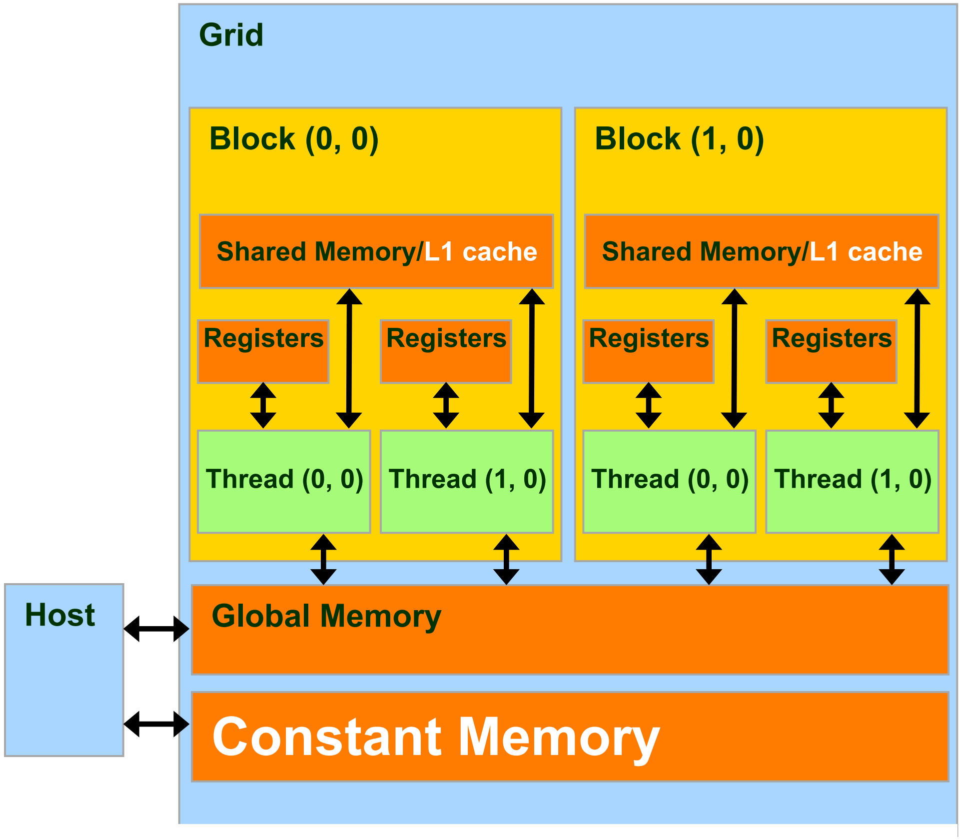
- Host code allocates, initializes variables the same way as any other variables that need o be copied to the device

- Use **cudaMemcpyToSymbol(dest, src, size)** to copy the variable into the device memory

- This copy function tells the device that the variable will not be modified by the kernel and can be safely cached.

More on Constant Caching

- Each SM has its own L1 cache
  - Low latency, high bandwidth access by all threads
- However, there is no way for threads in one SM to update the L1 cache in other SMs
  - No L1 cache coherence



This is not a problem if a variable is NOT modified by a kernel.

Using Constant memory

- When declaring variables, use *__const__ <type> restrict*
- For example:

```
    __global__ void convolution_2D_kernel(float
*P, float *N, int height, int width, int channels,
__const__ float restrict *M)
```

- In this case, we are telling the compiler that M is constant and eligible for caching

ANY MORE QUESTIONS? READ CHAPTER 8

Some Header File Stuff for M

```
#define KERNEL_SIZE 5

// Matrix Structure declaration
typedef struct {
   unsigned int width;
   unsigned int height;
   unsigned int pitch;
   float* elements;
} Matrix;
```

AllocateMatrix

```
// Allocate a device matrix of dimensions height*width
//        If init == 0, initialize to all zeroes.
//        If init == 1, perform random initialization.
//  If init == 2, initialize matrix parameters, but do not
allocate memory
Matrix AllocateMatrix(int height, int width, int init)
{
    Matrix M;
    M.width = M.pitch = width;
    M.height = height;
    int size = M.width * M.height;
    M.elements = NULL;
```

AllocateMatrix() (Cont.)

```
// don't allocate memory on option 2
  if(init == 2) return M;
  M.elements = (float*) malloc(size*sizeof(float));
  for(unsigned int i = 0; i < M.height * M.width; i++)
  {
    M.elements[i] = (init == 0) ? (0.0f) :
                    (rand() / (float)RAND_MAX);
   if(rand() % 2)    M.elements[i] = - M.elements[i]
  }
return M;
}
```

Host Code

```
// global variable, outside any function
__constant__ float Mc[KERNEL_SIZE][KERNEL_SIZE];
…
// allocate N, P, initialize N elements, copy N to Nd
Matrix  M;
M  = AllocateMatrix(KERNEL_SIZE, KERNEL_SIZE, 1);
// initialize M elements
….
cudaMemcpyToSymbol(Mc, M.elements,
        KERNEL_SIZE*KERNEL_SIZE*sizeof(float));
ConvolutionKernel<<<dimGrid, dimBlock>>>(Nd, Pd);
```