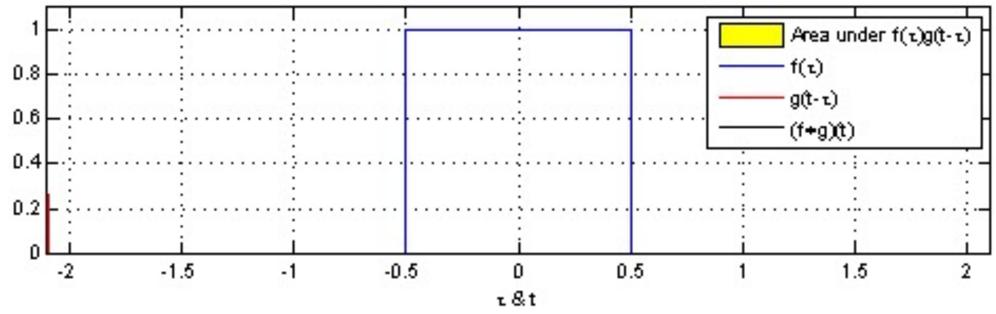
Parallel Patterns: Convolution

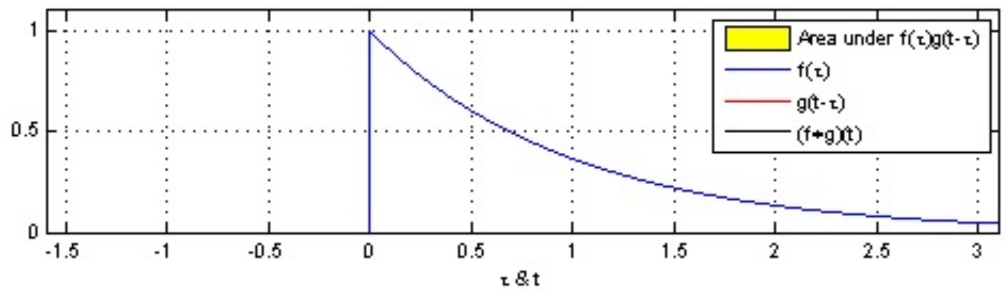
Prof. Seokin Hong

Convolution?

- A mathematical operation which describes a rule of how to combine two functions or pieces of information to form a third function.
- The convolution of *f* and *g* is written $f*g$

$$(f * g)(t) \triangleq \int_{-\infty}^{\infty} f(\tau)g(t-\tau)\,d\tau.$$

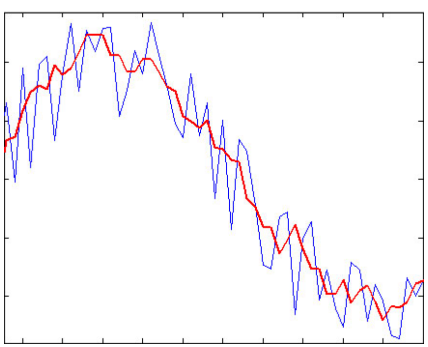


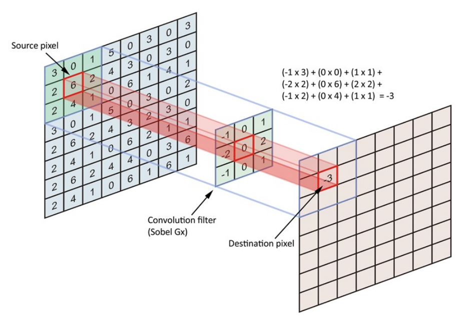


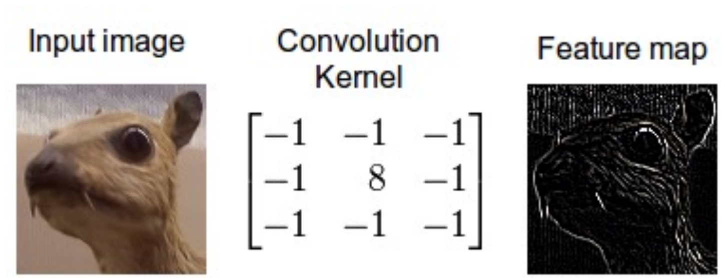
Convolution Applications

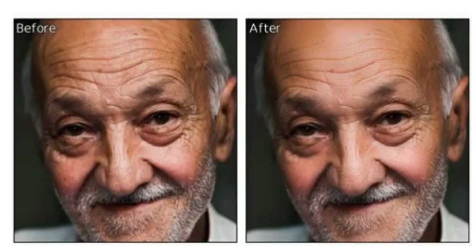
- Convolution is a popular array operation that is used in various forms in signal processing, digital recording, image processing, video processing, and computer vision.

- 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 trend
  - Gaussian filters can be used to sharpen boundaries and edges of objects in images.







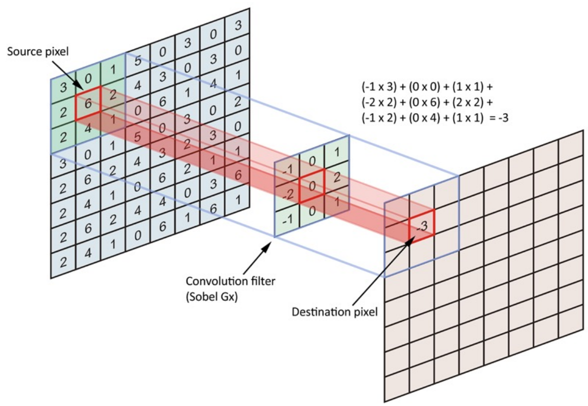


Convolution Computation

- 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 (aka filter or mask)***

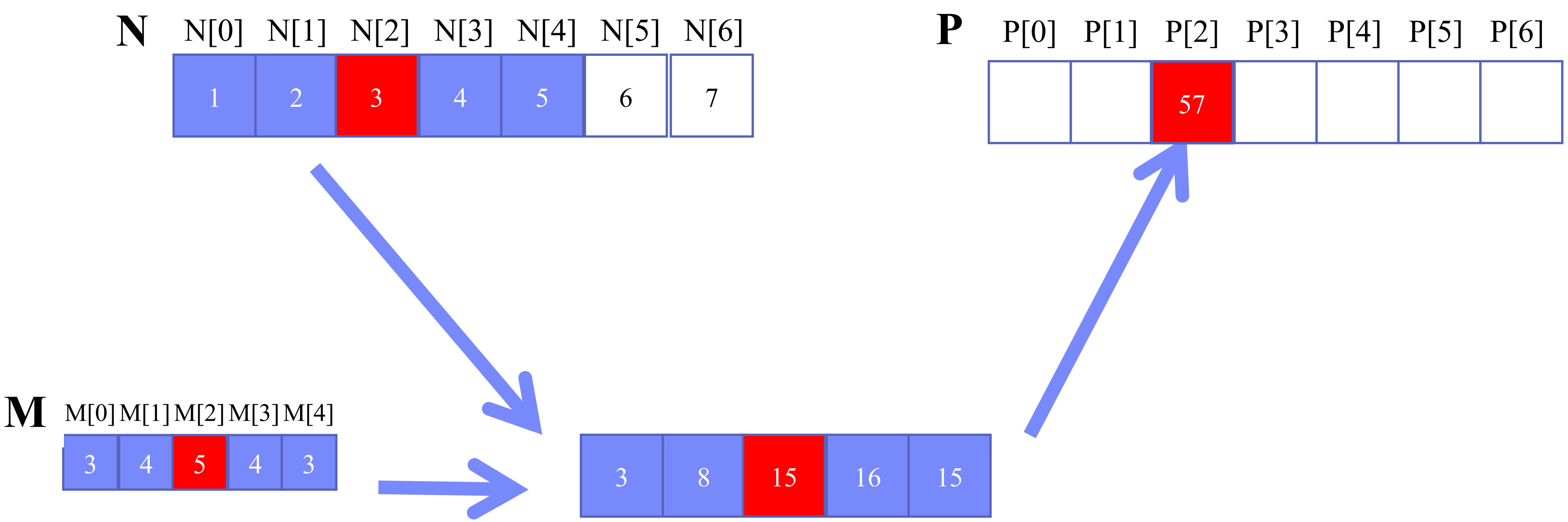
$$(f * g)(t) \triangleq \int_{-\infty}^{\infty} f(\tau) g(t-\tau) \, d\tau.$$



1D Convolution Example

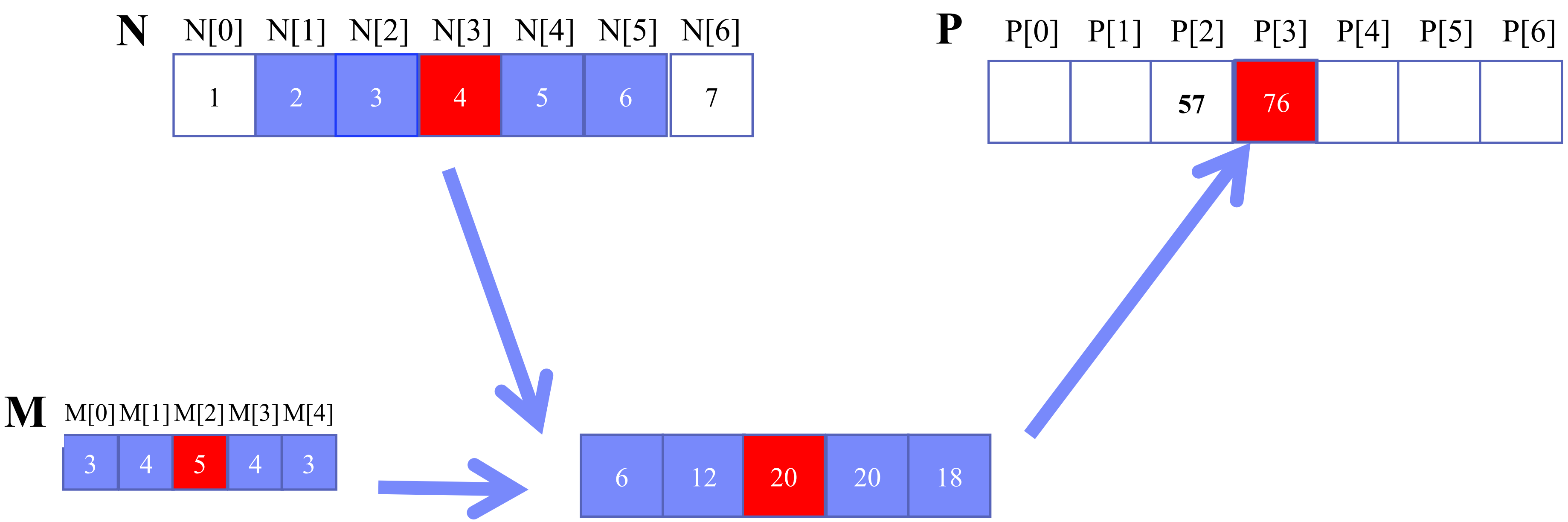
- Commonly used for audio processing
  - Mask size is usually an **odd** number of elements for symmetry (5 in this example)

- Calculation of P[2]



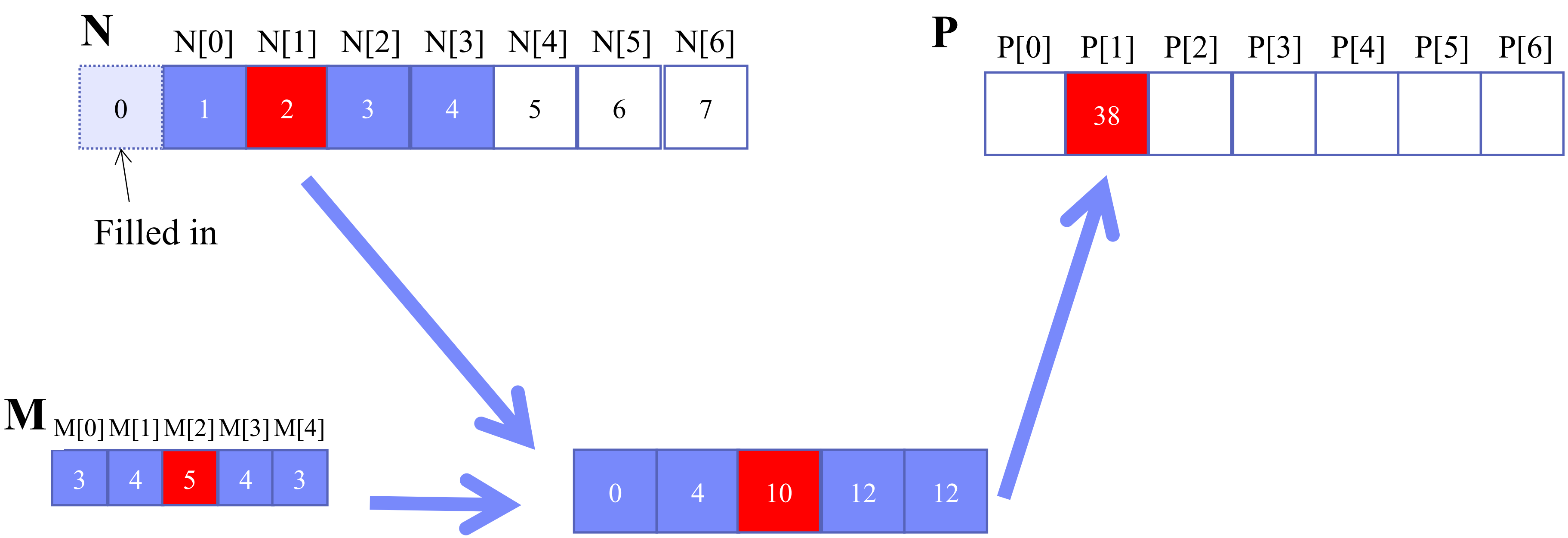
1D Convolution Example - more on inside elements

- Calculation of P[3]



1D Convolution with Boundary Condition Handling

- 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 with Boundary Condition Handling

- **1D Convolution CUDA Kernel**
  - This kernel forces all elements outside the valid data index range to 0

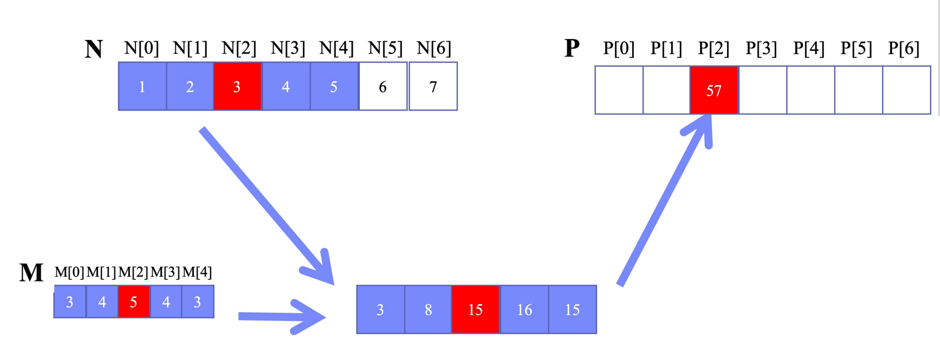
```
__global__ void convolution_1D_basic_kernel(float *N, float *M, float *P, int Mask_Width, int Width) {

  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;

}
```



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

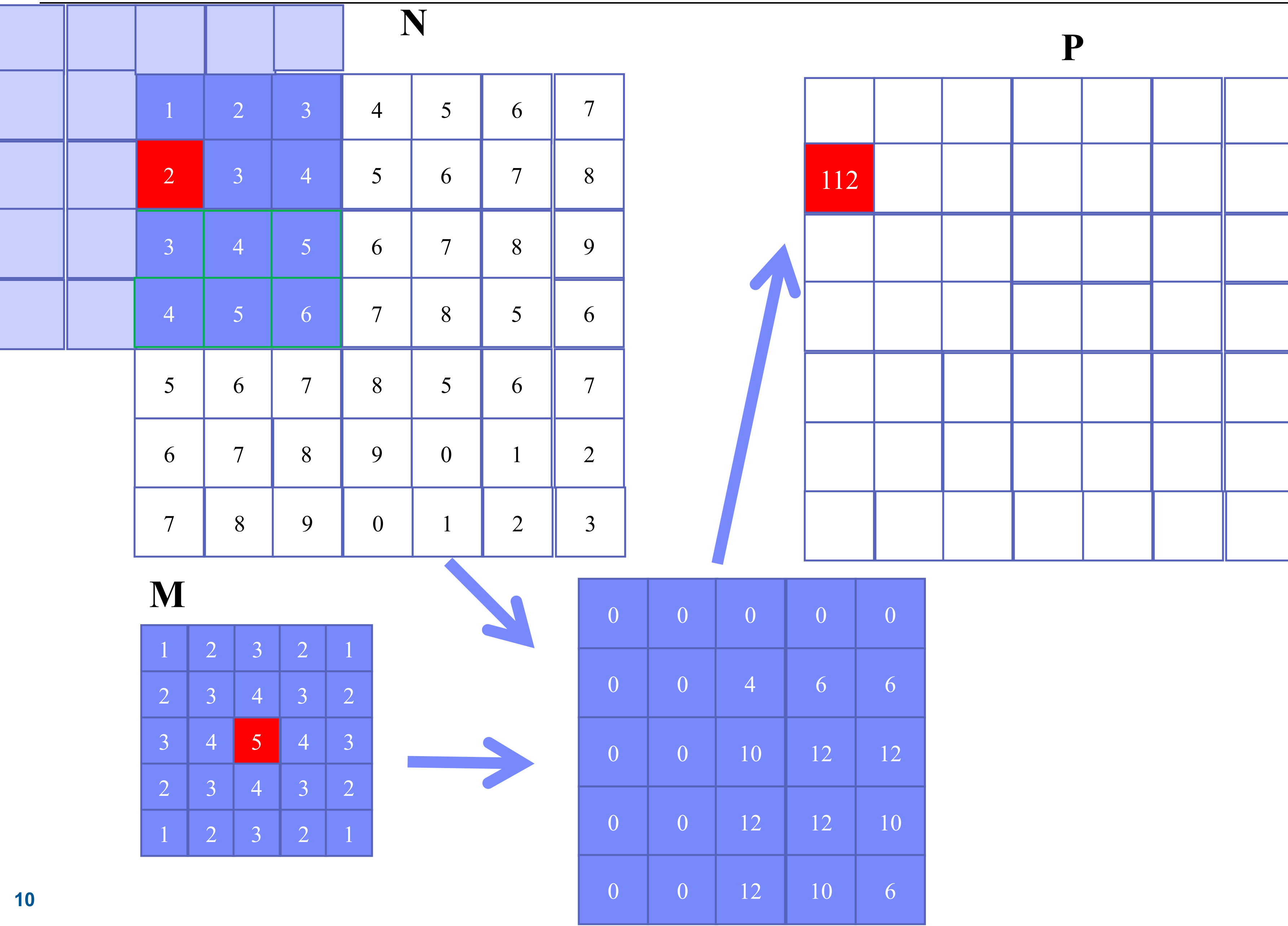
		321				

M

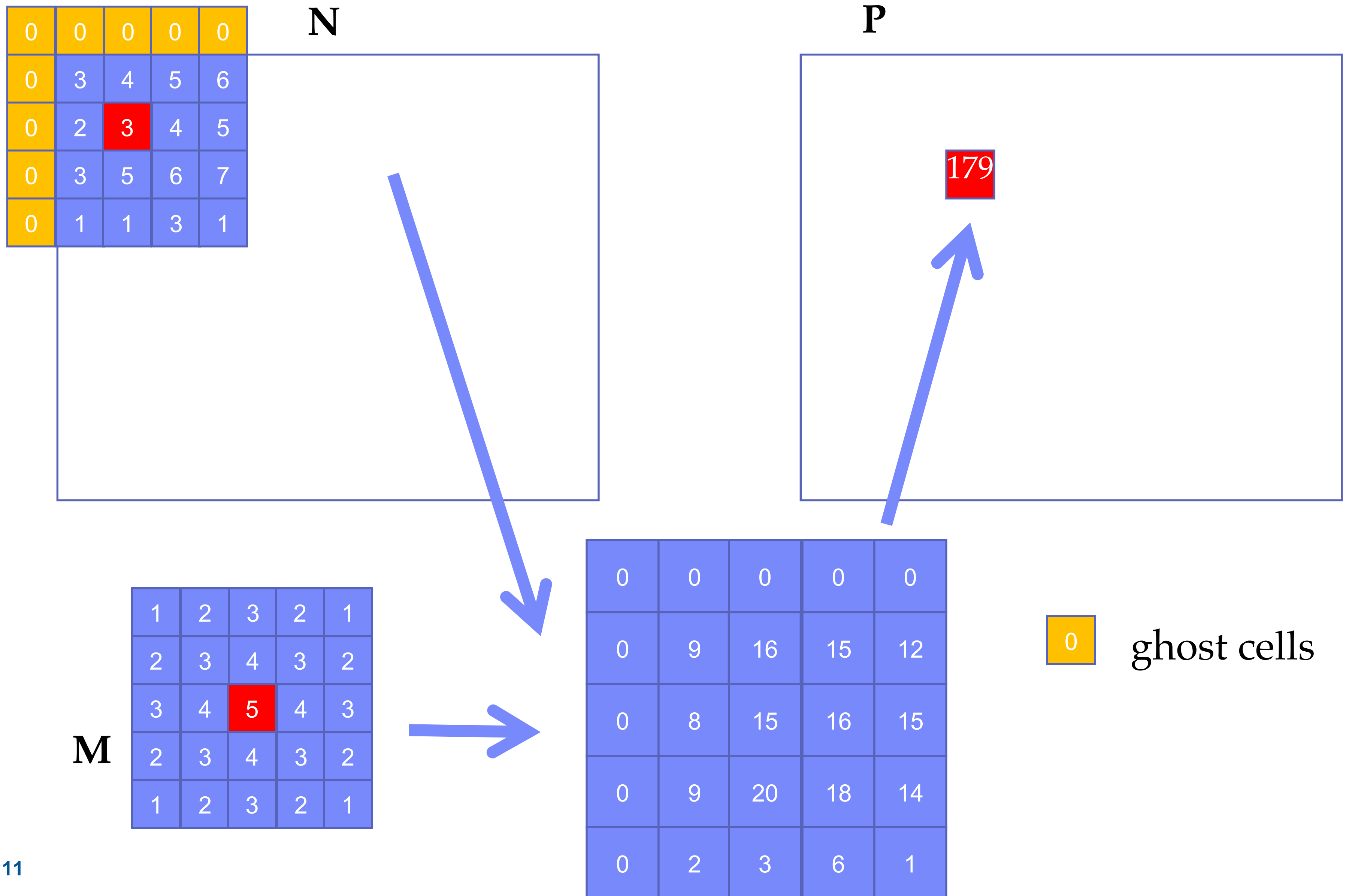
1	2	3	2	1
2	3	4	3	2
3	4	5	4	3
2	3	4	3	2
1	2	3	2	1

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

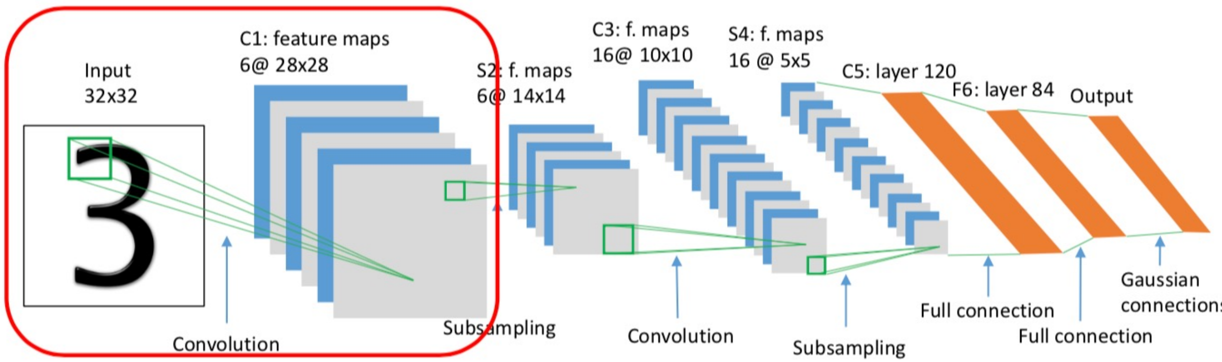
2D Convolution with Boundary Condition Handling

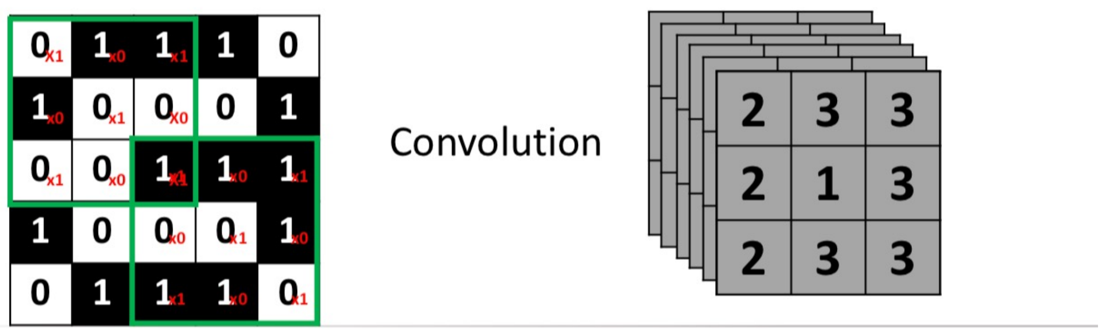


2D Convolution – Ghost Cells



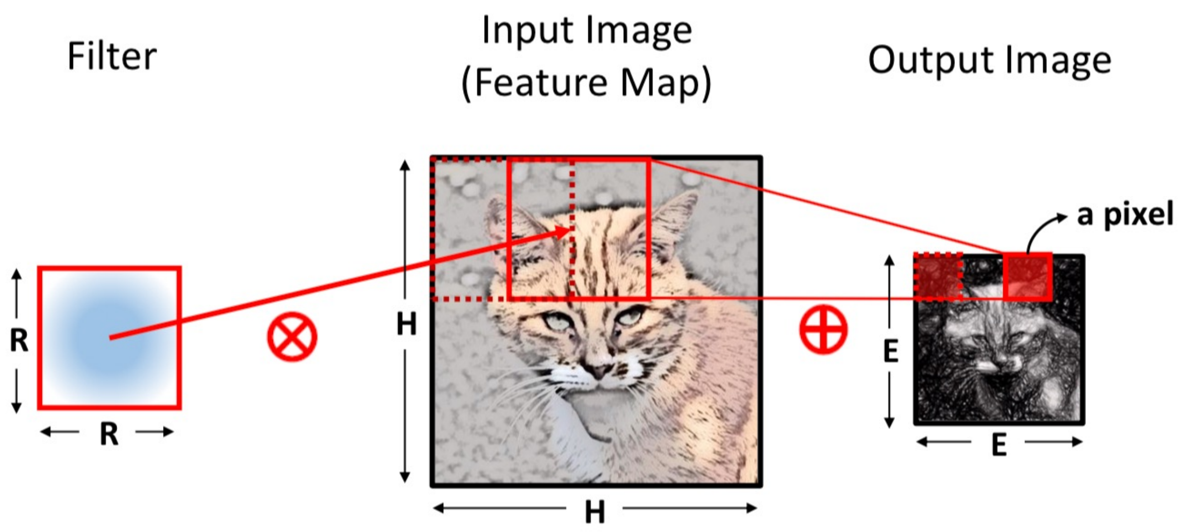
Convolution Neural Network (CNN)





Convolution Neural Network (CNN)

- Convolution



Sliding Window Processing

Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks (ISCA 2016)

Convolution Neural Network (CNN)

- Convolution Computation

```
for(r=0; r<R; r++)        //output feature map
  for(q=0; q<Q; q++)        //input feature map
    for(m=0; m<M; m++)        //row in feature map
      for(n=0; n<N; n++)        //column in feature map
        for(k=0; k<K; k++)        //row in convolution kernel
          for(l=0; l<L; l++)        //column in convolution kernel
            Y[r][m][n]+=W[r][q][k][l]*X[q][m+k][n+l];
```

Y = Y+W*X

“Convolution” account for over 90% of the CNN operations and dominates runtime

Access Pattern for M

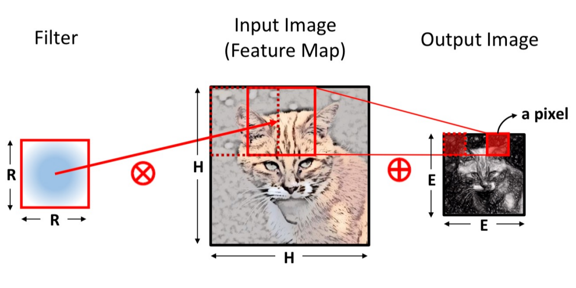
- **M** is referred to as mask (a.k.a. kernel, filter, etc.)
  - Elements of M are called mask (kernel, filter) coefficients
- Calculation of all output P elements needs **M**
- **M** is not changed during a execution of CUDA kernel that performs the convolution

- In addition, **M** elements are accessed in the same order when calculating all P elements

- M is a good candidate for **Constant Memory**

M

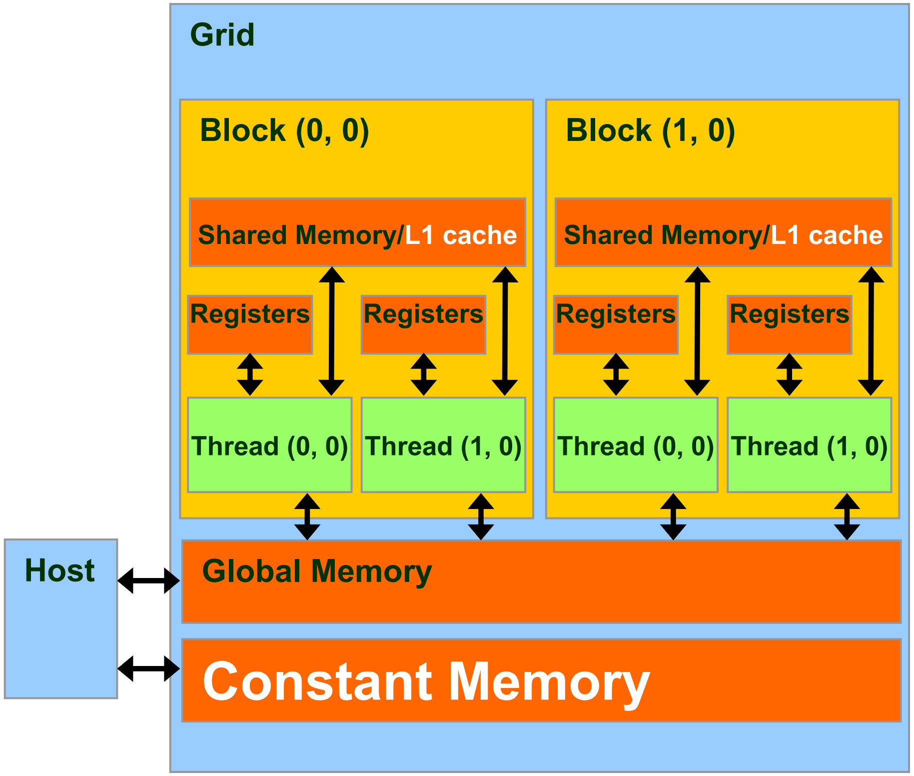
1	2	3	2	1
2	3	4	3	2
3	4	5	4	3
2	3	4	3	2
1	2	3	2	1



Sliding Window Processing

Programmer View of CUDA Memories (Review)

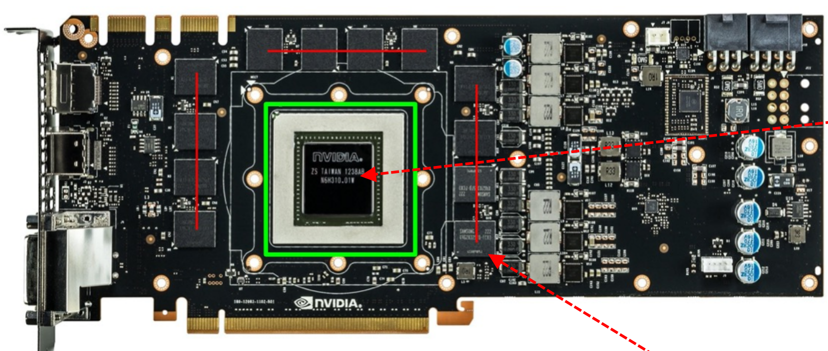
- Each thread can:
  - Read/write per-thread **registers (~1 cycle)**
  - Read/write per-block **shared memory (~5 cycles)**
  - Read/write per-grid **global memory (~500 cycles)**
  - Read/only per-grid **constant memory (~5 cycles with caching)**



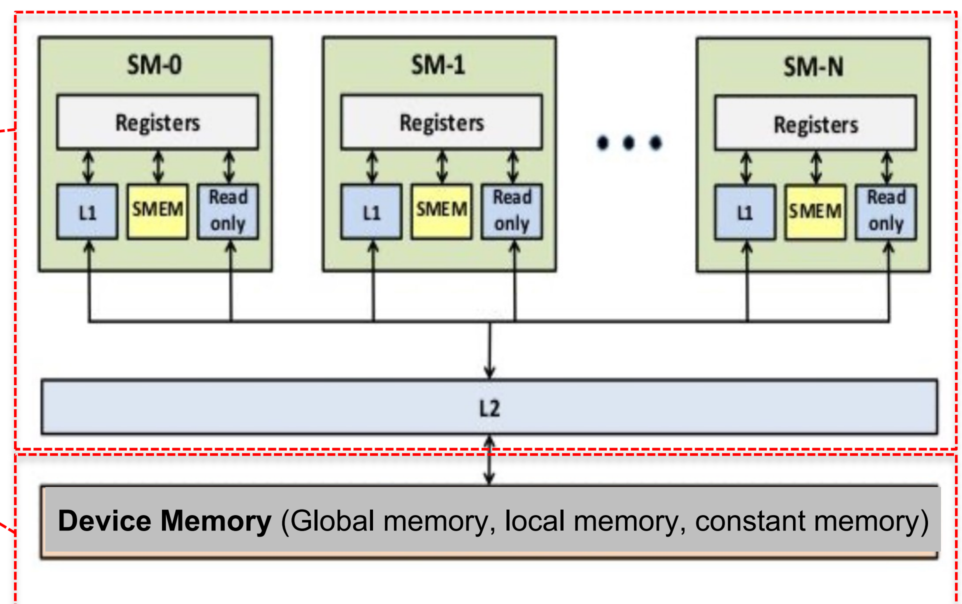
Memory Hierarchies

- If 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

- One solution: Caches



Green box is GK110, red lines are global memory



Cache (review)

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

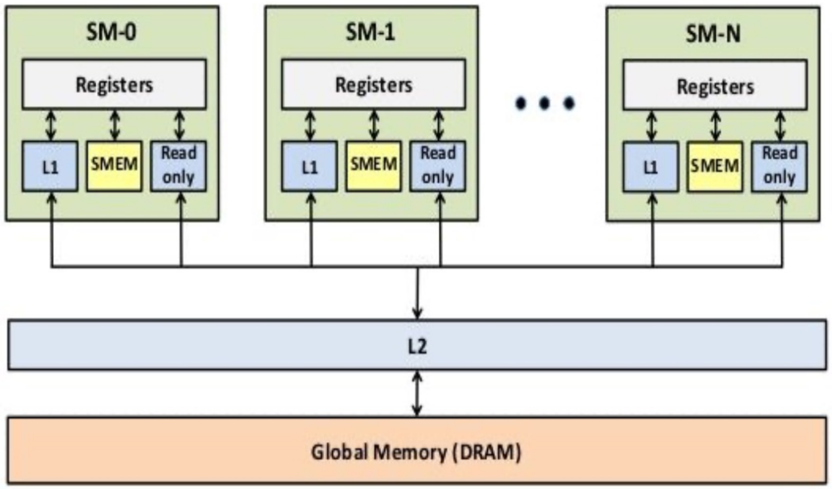
- When data is requested from the global memory, an entire cache line that includes the data being accessed is loaded into the cache, in an attempt to reduce global memory requests
  - The data in the cache is a "copy" of the original data in global memory

- **Locality**
  - **Spatial locality**: when the data elements stored in consecutive memory locations are access consecutively
  - **Temporal locality**: 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

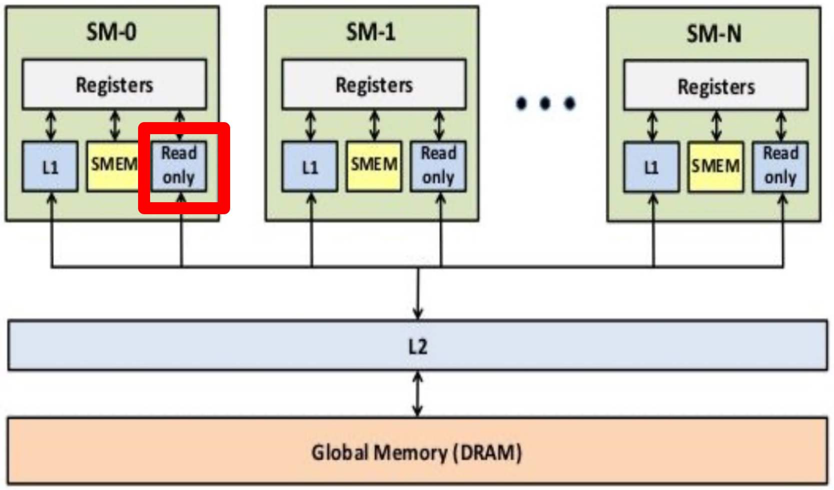
- **Scratchpad** (**shared memory** in CUDA) is another type of temporary storage used to relieve main memory contention.
  - In terms of distance from the processor, scratchpad is similar to L1 cache.
- **Unlike cache**, scratchpad does not necessarily hold a copy of data that is also in main memory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



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
  - Data declared in the constant memory will not be modified during kernel execution.



How to Use Constant Memory

- Host code allocates, initializes variables the same way as any other variables that need to 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.

Some Header File Stuff for M

```
#define MASK_WIDTH 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;
   // don't allocate memory on option 2
   if(init == 2) return M;
   int size = height * width;
   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 kernel/function
  __constant__ float Mc[MASK_WIDTH][MASK_WIDTH];
…
  // allocate N, P, initialize N elements, copy N to Nd
  Matrix  M;
  M  = AllocateMatrix(MASK_WIDTH, MASK_WIDTH, 1);

  // initialize M elements
….
  cudaMemcpyToSymbol(Mc, M.elements,
            MASK_WIDTH*MASK_WIDTH*sizeof(float));
  ConvolutionKernel<<<dimGrid, dimBlock>>>(Nd, Pd);
```

Next?

- **Tiled Convolution**