2D-convolution problem

Objective

- To learn convolution, an important method
  - Widely used in audio, image and video processing
  - Foundational to stencil computation used in many science and engineering applications
  - Basic 1D and 2D convolution kernels

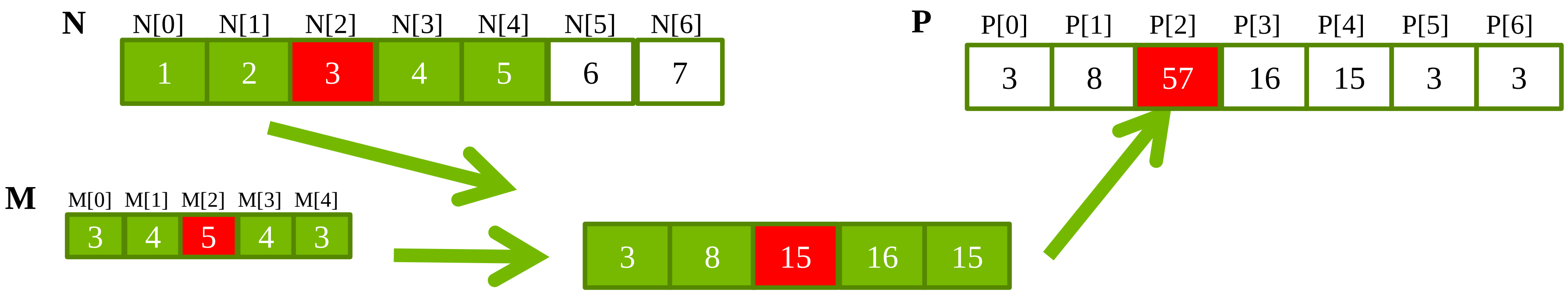
Convolution as a Filter

- Often performed as a filter that transforms signal or pixel values 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 – a computational definition

- 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 value pattern of the mask array elements defines the type of filtering done.

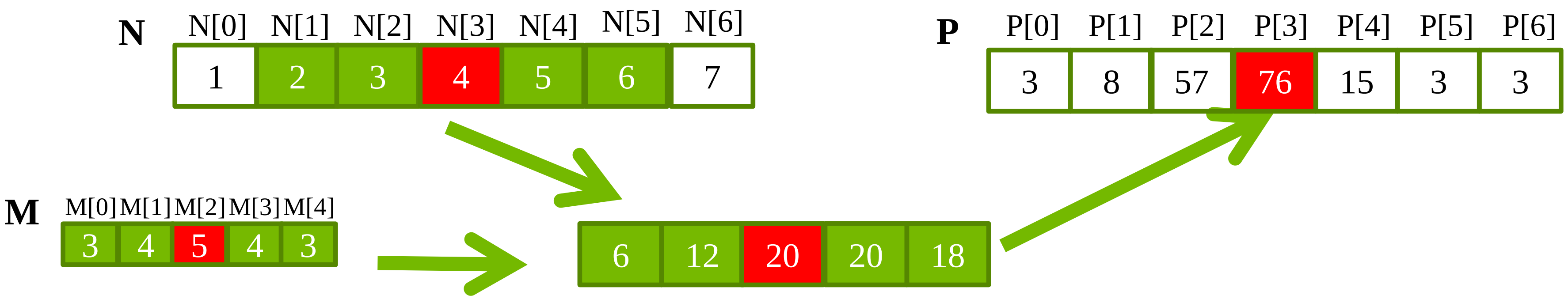
1D Convolution Example



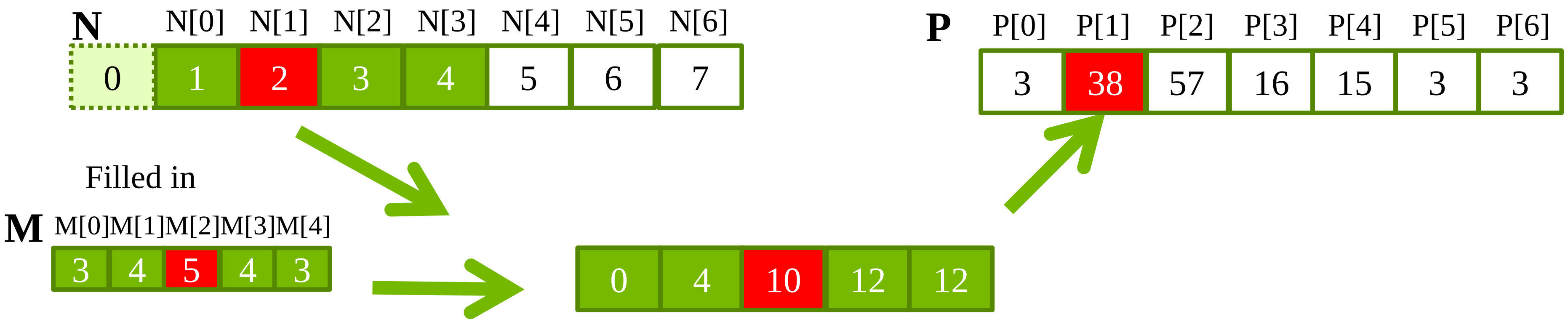
- Commonly used for audio processing
  - Mask size is usually an odd number of elements for symmetry (5 in this example)
- The figure shows calculation of P[2]

P[2] = N[0]*M[0] + N[1]*M[1] + N[2]*M[2] + N[3]*M[3] + N[4]*M[4]

Calculation of P[3]



Convolution Boundary Condition



- Calculation of output elements near the boundaries (beginning and end) of the 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 valid input range to 0

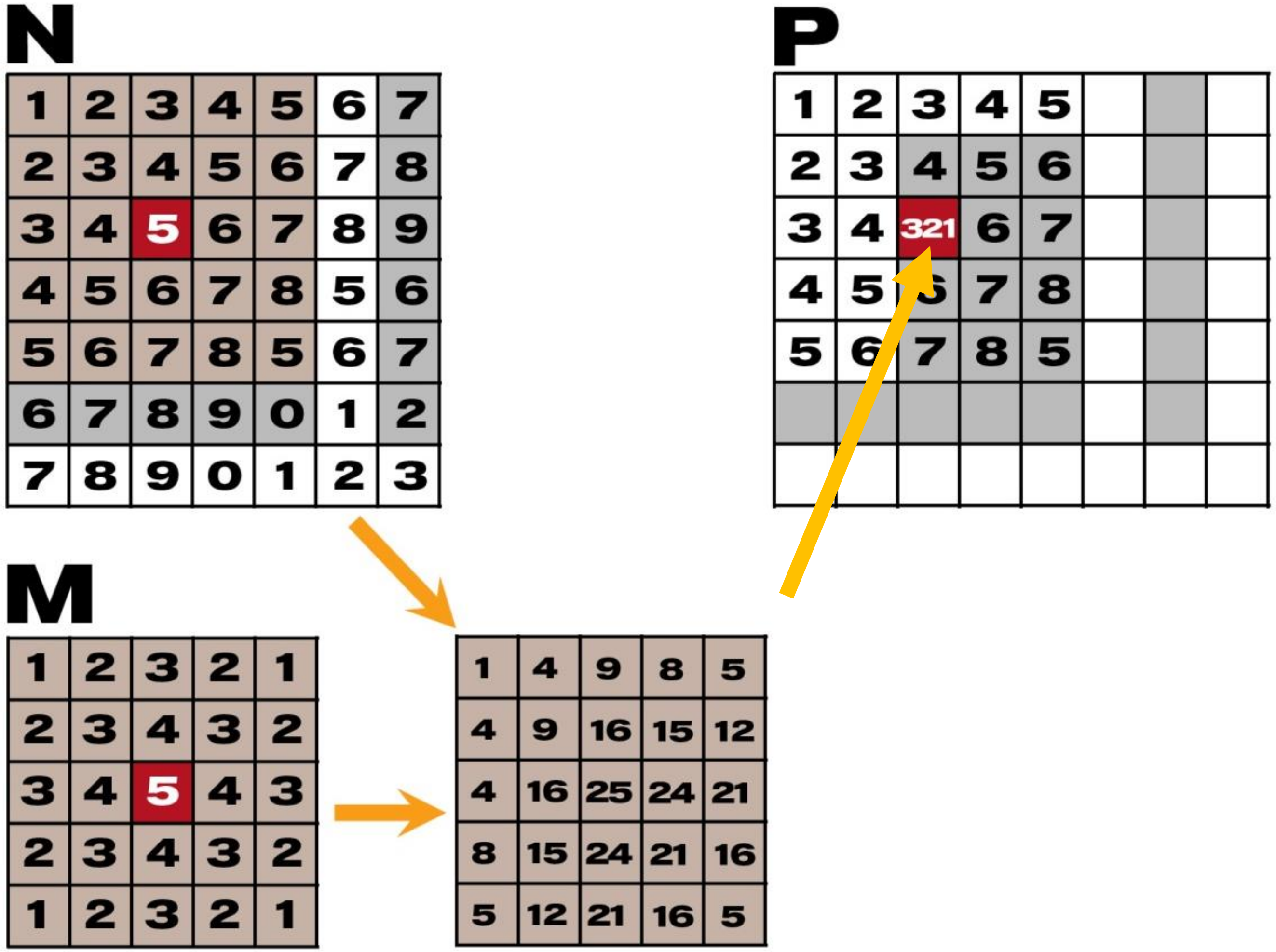
```
__global__ void convolution_1D_basic_kernel(float *N, float *M,
            float *P, int Mask_Width, int Width)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;

    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;
}
```

A 1D Convolution Kernel with Boundary Condition Handling

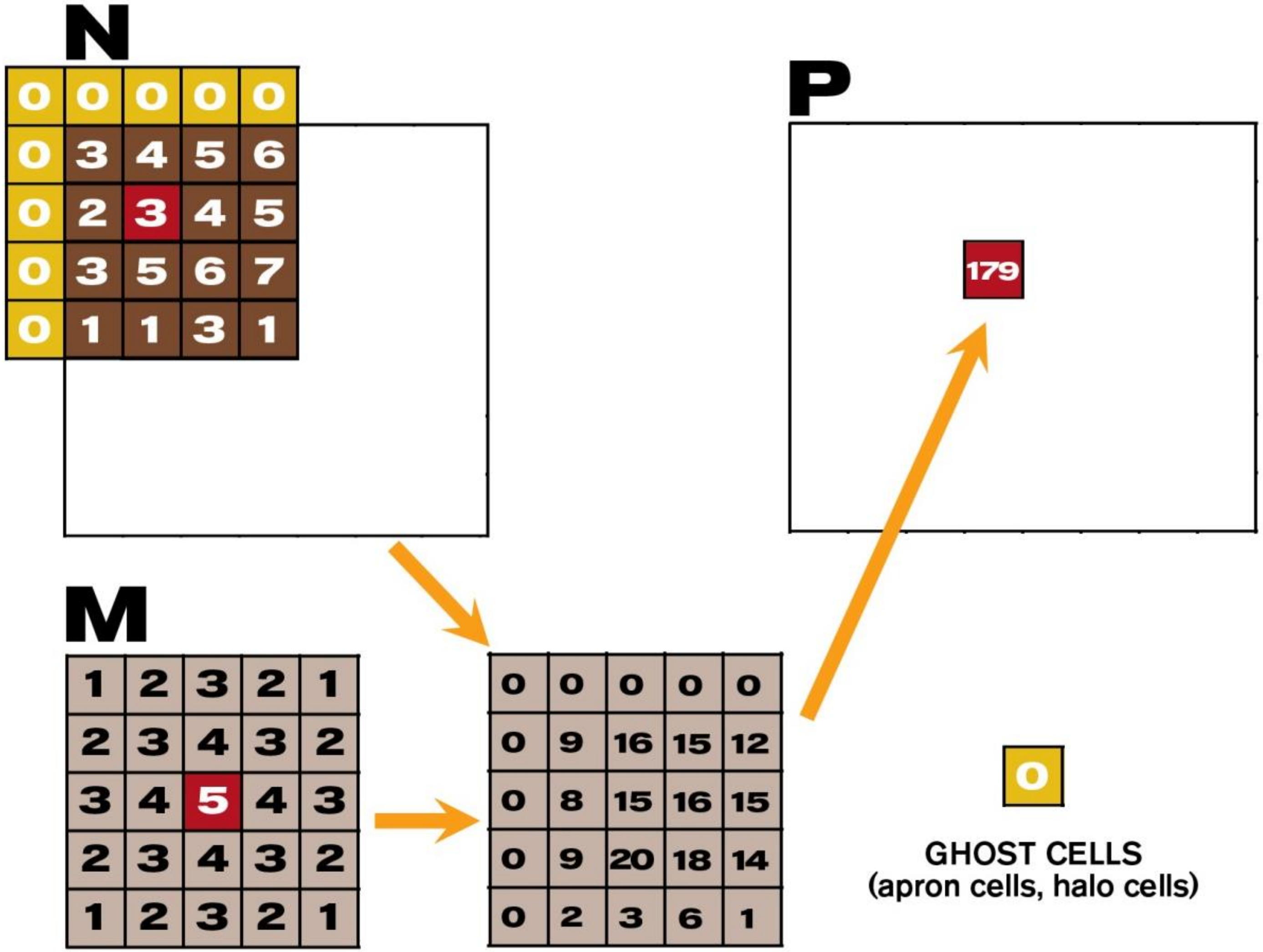
– This kernel forces all elements outside the valid input range to 0

```
__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);

   if (i < Width) {

     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;
   }
}
```

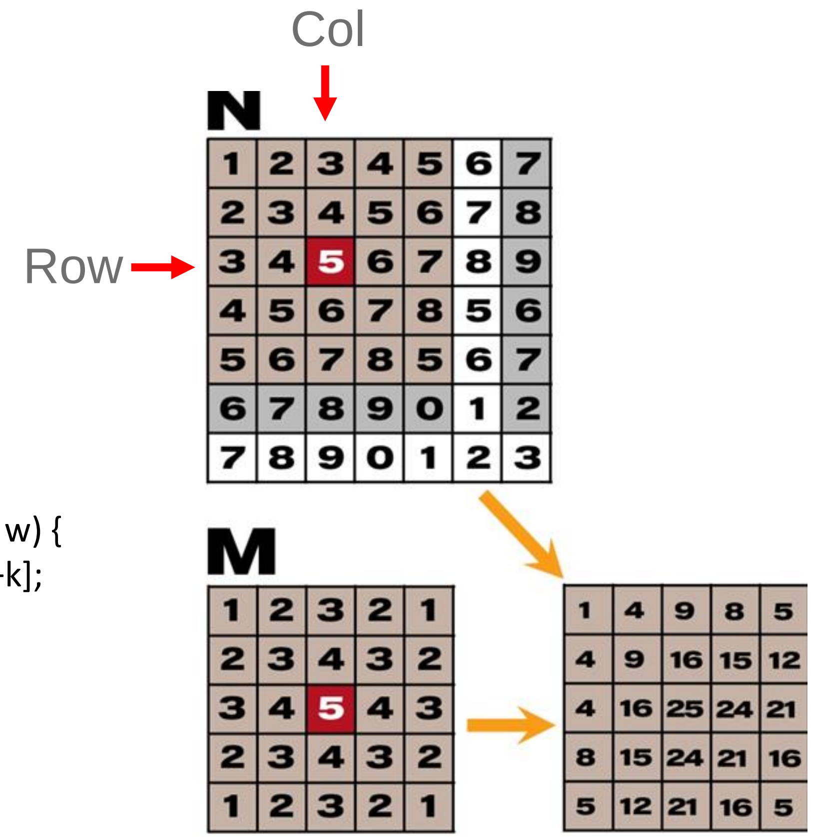
2D Convolution



2D Convolution – Ghost Cells

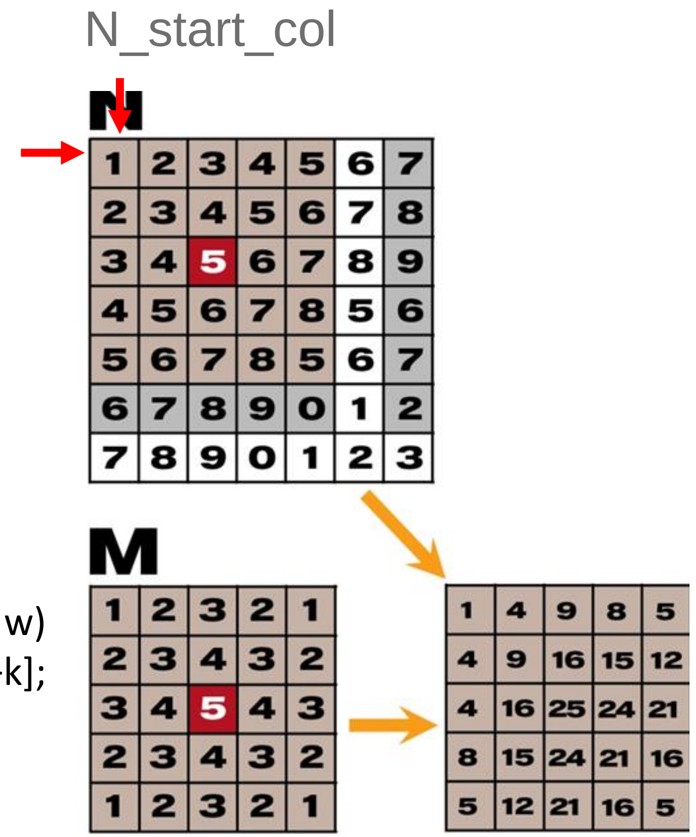


```
__global__
void convolution_2D_basic_kernel(unsigned char * in, unsigned char * mask, unsigned char * out,
                                 int maskwidth, int w, int h) {
  int Col  =   blockIdx.x * blockDim.x + threadIdx.x;
  int Row  = blockIdx.y * blockDim.y + threadIdx.y;

  if (Col < w && Row < h) {
    int pixVal = 0;

    N_start_col  = Col –   (maskwidth/2);
    N_start_row = Row – (maskwidth/2);

    // Get the of the surrounding box
    for(int j = 0; j < maskwidth; ++j) {
      for(int k = 0; k < maskwidth; ++k) {

        int curRow = N_Start_row + j;
        int curCol =   N_start_col  + k;
        // Verify we have a valid image pixel
        if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
          pixVal += in[curRow * w + curCol] * mask[j*maskwidth+k];
        }
      }
    }
    // Write our new pixel value out
    out[Row * w + Col] = (unsigned char)(pixVal);
  }
}
```



```
__global__
void convolution_2D_basic_kernel(unsigned char * in, unsigned char * mask, unsigned char * out,
                                 int maskwidth, int w, int h) {
    int Col  =   blockIdx.x * blockDim.x + threadIdx.x;
    int Row  = blockIdx.y * blockDim.y + threadIdx.y;

    if (Col < w && Row < h) {
       int pixVal = 0;

       N_start_col  = Col –   (maskwidth/2);
       N_start_row = Row – (maskwidth/2);

       // Get the of the surrounding box
       for(int j = 0; j < maskwidth; ++j) {
          for(int k = 0; k < maskwidth; ++k) {

             int curRow = N_Start_row + j;
             int curCol =   N_start_col  + k;
             // Verify we have a valid image pixel
             if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
                pixVal += in[curRow * w + curCol] * mask[j*maskwidth+k];
             }
          }
       }
       // Write our new pixel value out
       out[Row * w + Col] = (unsigned char)(pixVal);
    }
}
```

N_start_row

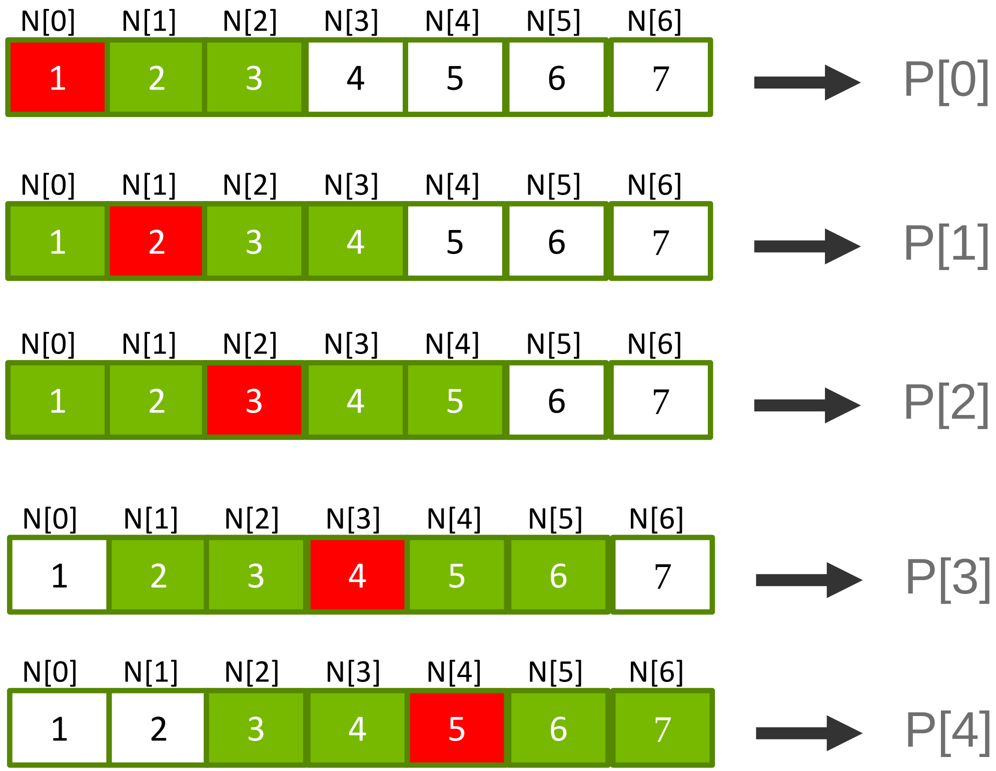


```
__global__
void convolution_2D_basic_kernel(unsigned char * in, unsigned char * mask, unsigned char * out,
                    int maskwidth, int w, int h) {
  int Col  =  blockIdx.x * blockDim.x + threadIdx.x;
  int Row  = blockIdx.y * blockDim.y + threadIdx.y;

  if (Col < w && Row < h) {
    int pixVal = 0;

    N_start_col  = Col –  (maskwidth/2);
    N_start_row = Row – (maskwidth/2);

    // Get the of the surrounding box
    for(int j = 0; j < maskwidth; ++j) {
      for(int k = 0; k < maskwidth; ++k) {

        int curRow = N_Start_row + j;
        int curCol =  N_start_col  + k;
        // Verify we have a valid image pixel
        if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
          pixVal += in[curRow * w + curCol] * mask[j*maskwidth+k];
        }
      }
    }
    // Write our new pixel value out
    out[Row * w + Col] = (unsigned char)(pixVal);
  }
}
```

Introduce tiling in 1D-convolution

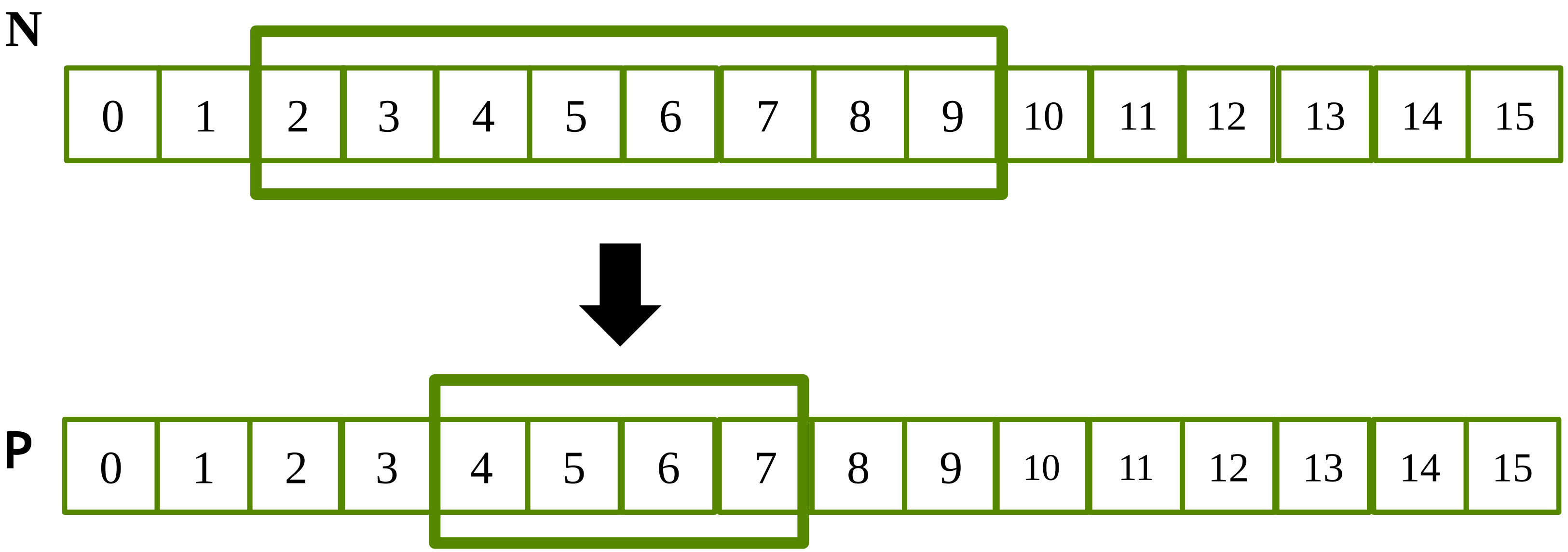
Tiling Opportunity Convolution

- Calculation of adjacent output elements involve shared input elements
  - E.g., N[2] is used in calculation of P[0], P[1], P[2]. P[3 and P[5] assuming a 1D convolution Mask_Width of width 5
- We can load all the input elements required by all threads in a block into the shared memory to reduce global memory accesses

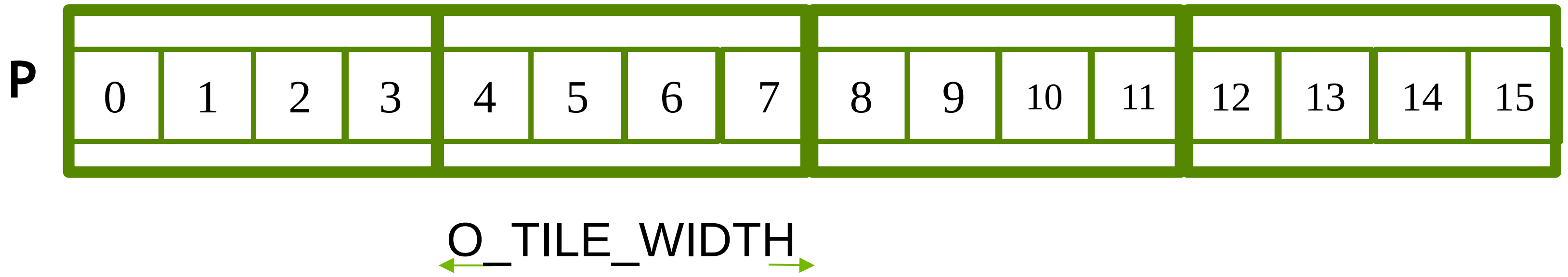


Input Data Needs

– Assume that we want to have each block to calculate T output elements
  – T + Mask_Width -1 input elements are needed to calculate T output elements
  – T + Mask_Width -1 is usually not a multiple of T, except for small T values
  – T is usually significantly larger than Mask_Width



Definition – output tile



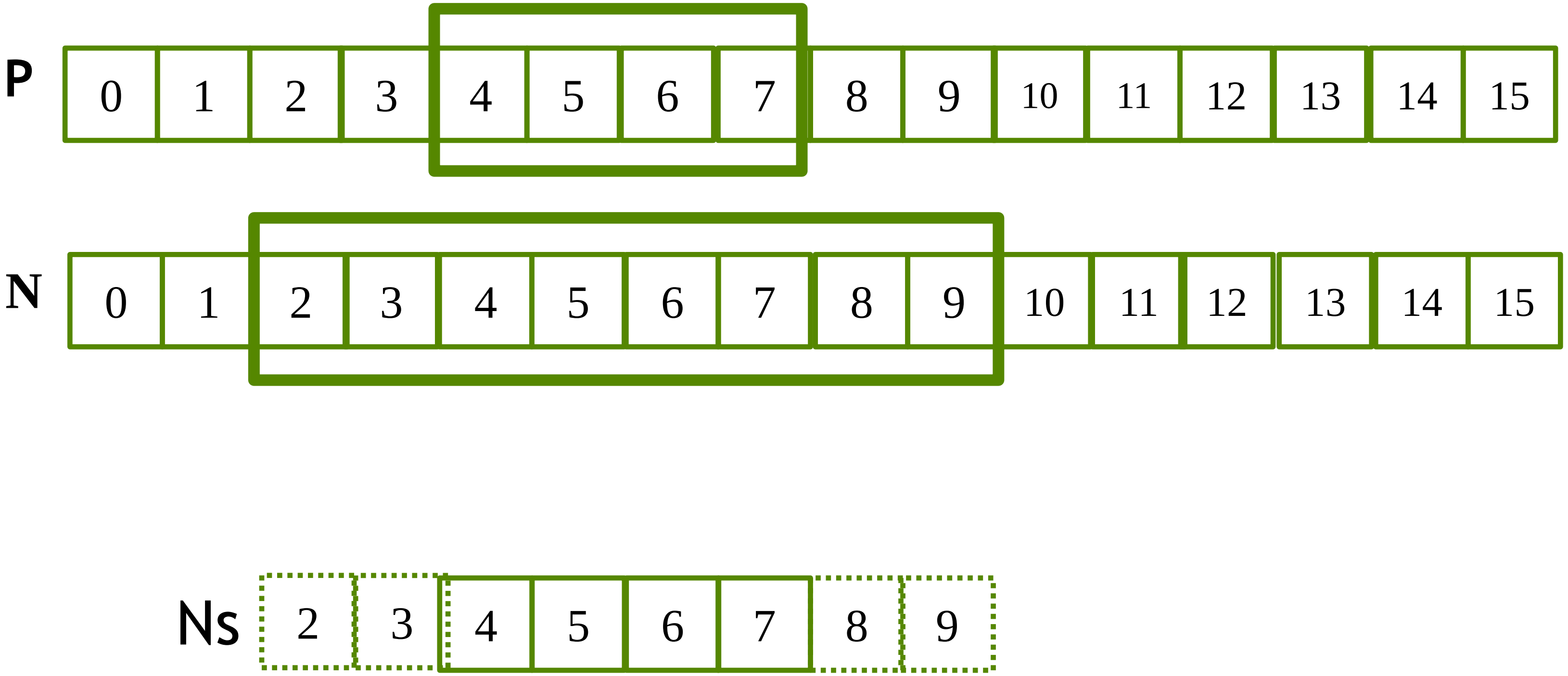
Each thread block calculates an output tile

Each output tile width is O_TILE_WIDTH

For each thread,

O_TILE_WIDTH is 4 in this example

Definition - Input Tiles



Each input tile has all values needed to calculate the corresponding output tile.

challenge

- The aim is to implement an efficient 2D-convolution algorithm in CUDA.
- The size of the mask should be parametric.
- Show the differences between the implementation with and without tiling.
- Analysis of different implementations with different tiling size: optimize the performance given a specific Colab GPU.
- Submit a google Colab file (.pynb) where you show your finding.
  - Submitting a file other than .pynb is possible, but it requires prior discussion with me
- Provide a short report (max 2 pages) where you present your finding:
  - Experimental setup
  - Performance measurements
  - Explanation of design choices
  - No screenshots of the code!
- Deadline: December 12th, midnight

Challenge rules

Groups up to 3 people

Changing team for the 2nd challenge is ok

One week of time

3 points for free in the 2nd part of the exam

Skip selected questions

Valid any time in the whole academic year