GPU Teaching Kit

Accelerated Computing

Module 8.1 – Parallel Computation Patterns (Stencil)

Convolution

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

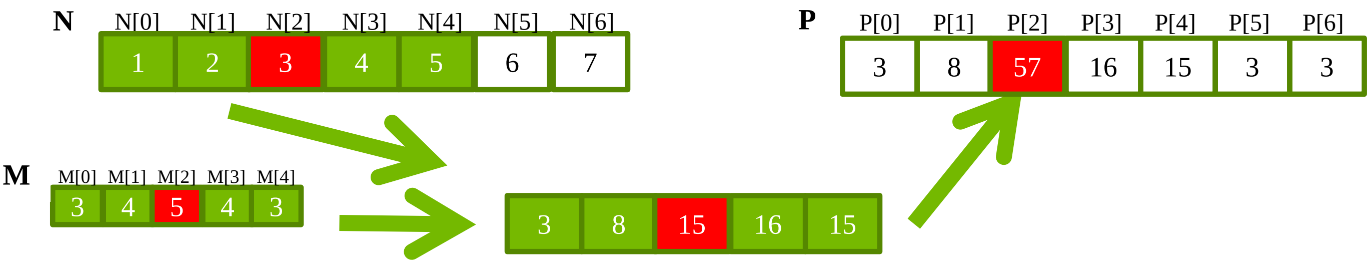
Convolution as a Filter

- 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
  - Our image blur example in Module 3 is a special case where all mask elements are of the same value and hard coded into the source code.

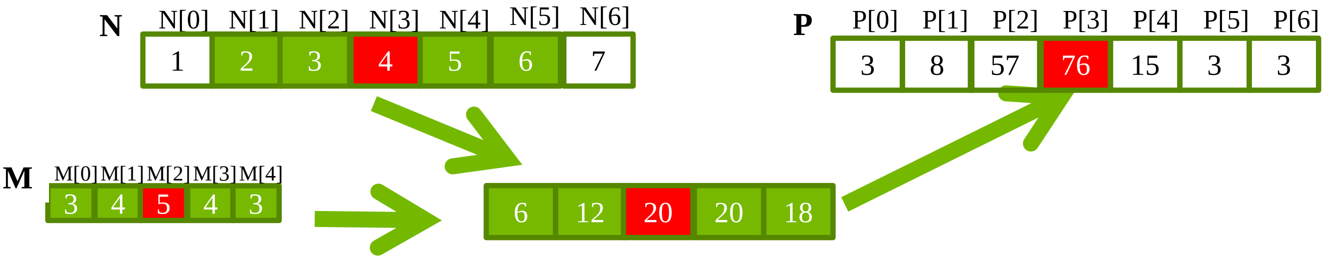
1D Convolution Example



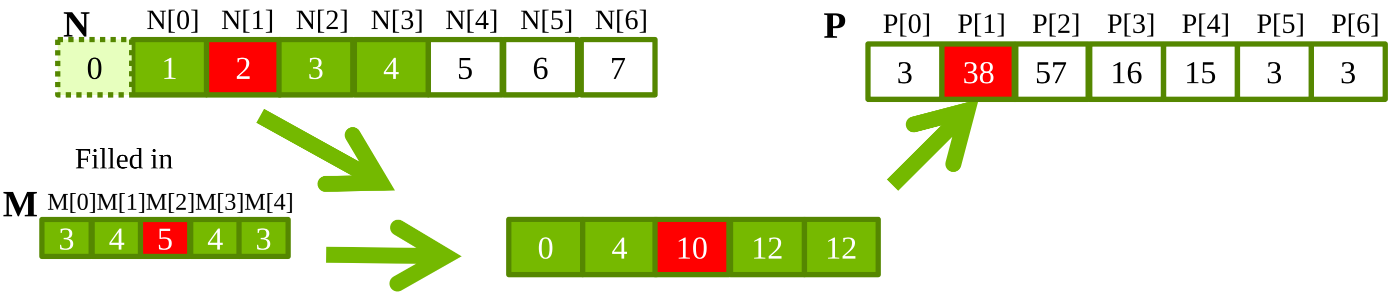
- Commonly used for audio processing
  - Mask size is usually an odd number of elements for symmetry (5 in this example)
- The figure shows calculation of P[2]

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

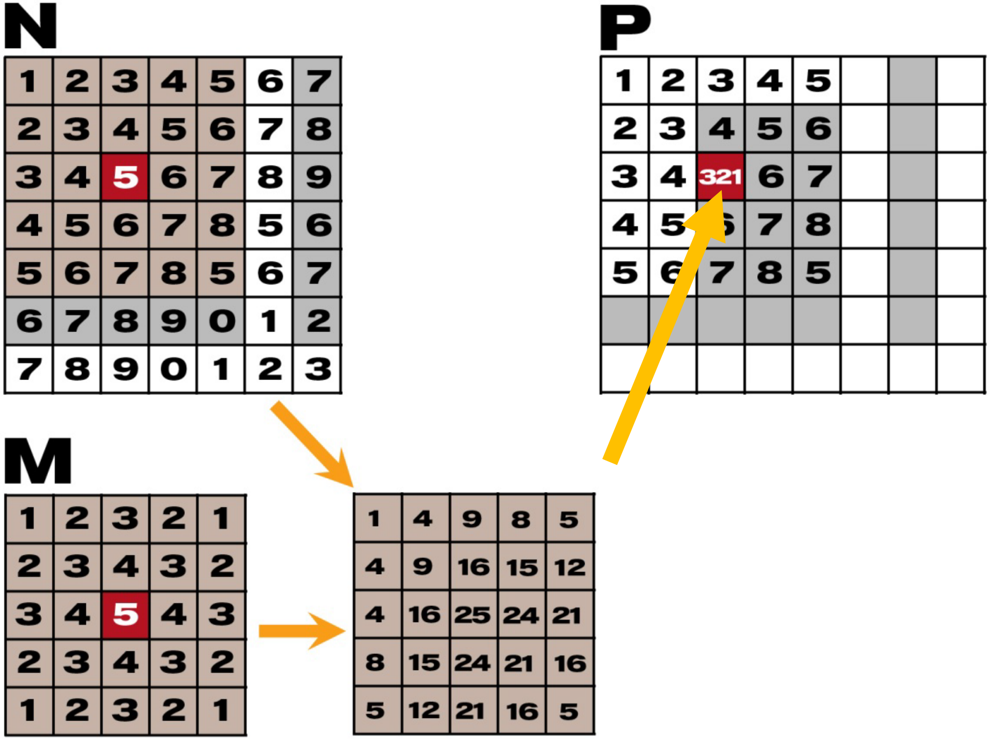
```
__global__ void convolution_1D_basic_kernel(float *N, float *M,
            float *P, int Mask_Width, int Width)
{
   int i = blockIdx.x*blockDim.x + threadIdx.x;

   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

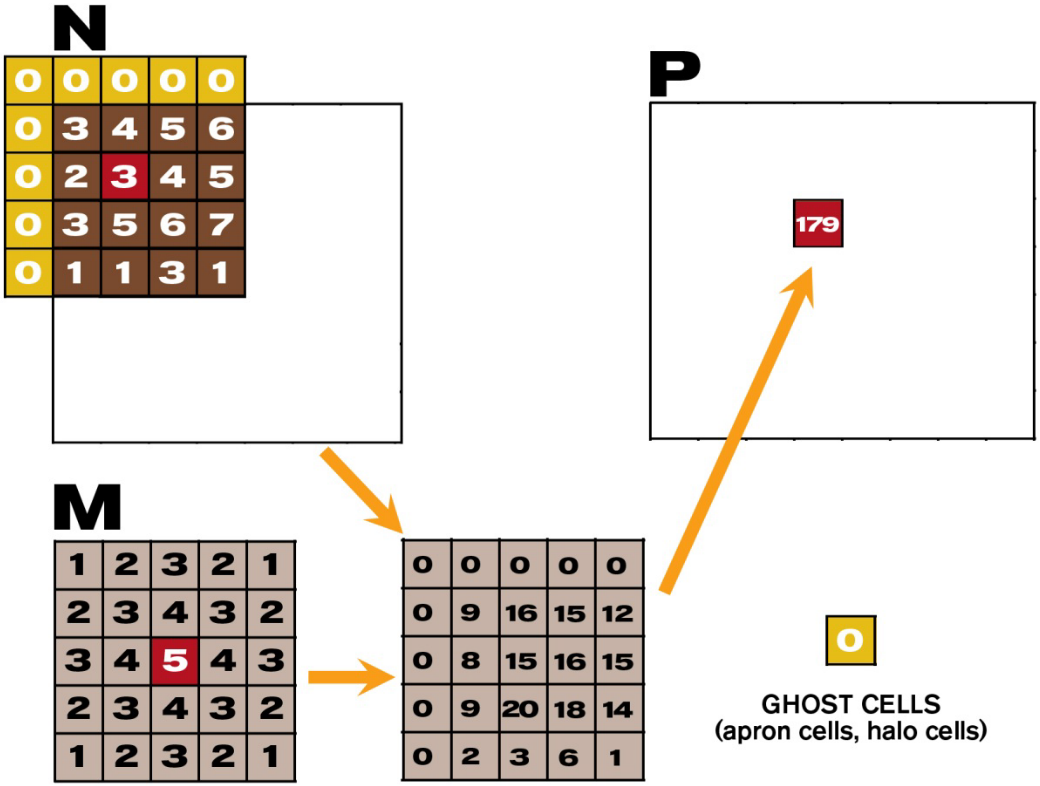
– This kernel forces all elements outside the valid input range to 0

```
__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;
  }
}
```

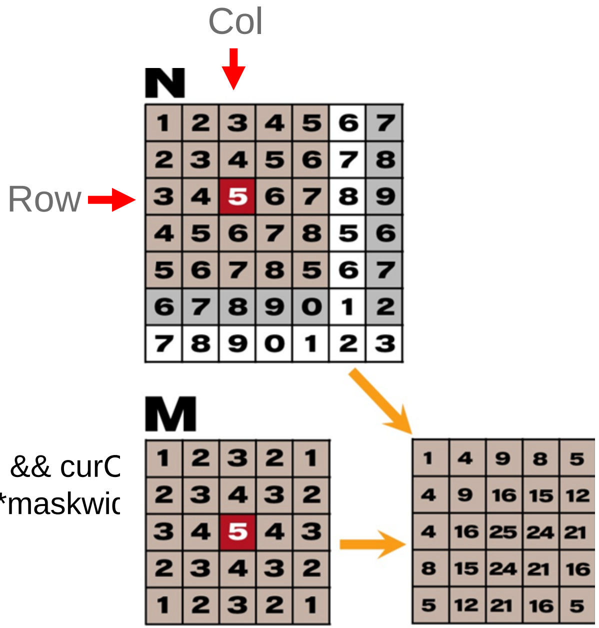
2D Convolution



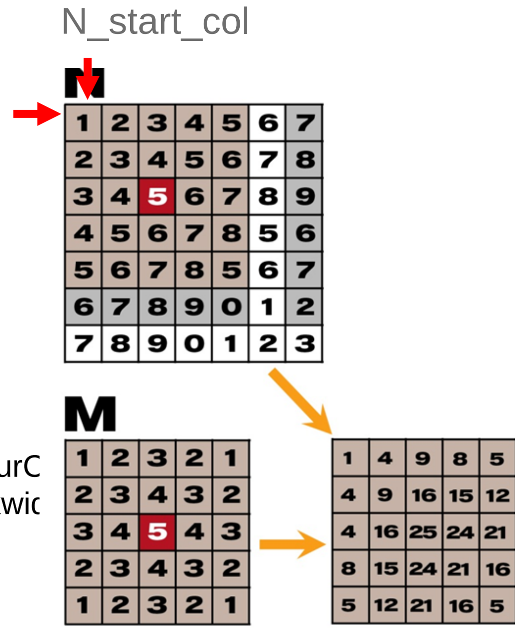
2D Convolution – Ghost Cells



```
__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 && curC
                pixVal += in[curRow * w + curCol] * mask[j*maskwic
             }
          }
       }

       // 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 && curC
                    pixVal += in[curRow * w + curCol] * mask[j*maskwic
                }
            }
        }

        // 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);
    }
}
```

GPU Teaching Kit

Accelerated Computing