CMPE 214
GPU Architecture & Programming

Lecture 4.
Parallel Computation Patterns (2)

Haonan Wang

SJSU SAN JOSÉ STATE UNIVERSITY

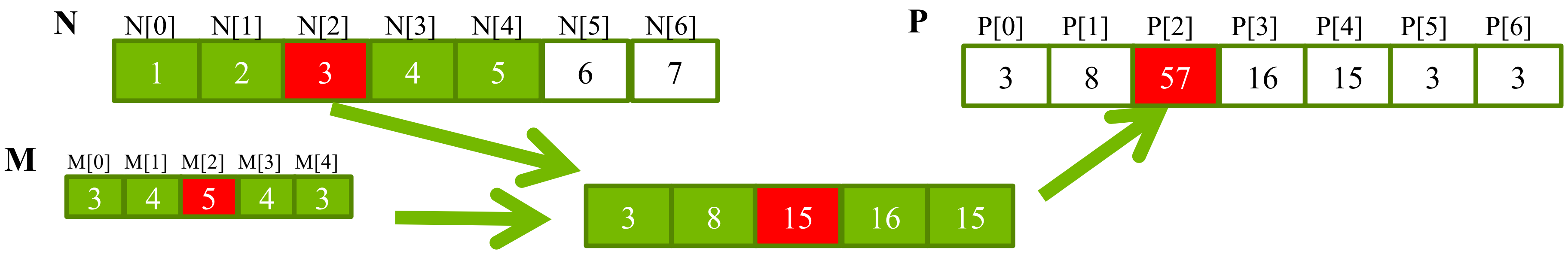
Stencil (aka. Convolution)

- **Widely used in audio, image and video processing**
  - Often performed as a filter that transforms signal or pixel values into more desirable values.
  - E.g., some filters smooth out the signal values so that one can see the big-picture trend

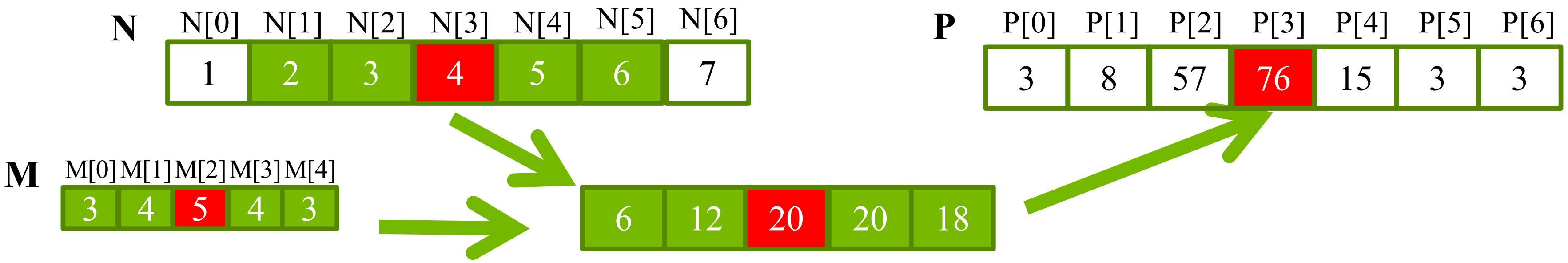
Original

Emboss

1D Convolution Example

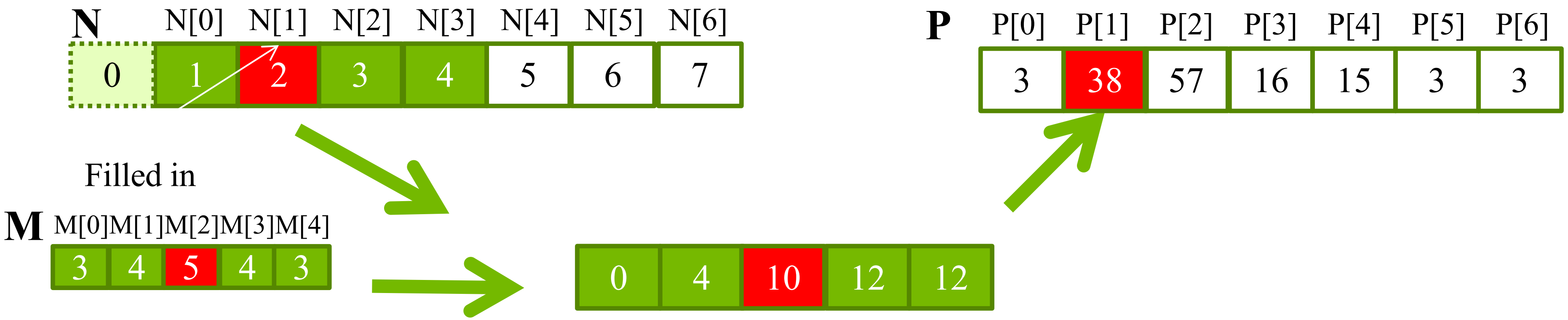


- **The output is is a weighted sum of a collection of neighboring input elements**
  - E.g., P[2] = N[0]*M[0] + N[1]*M[1] + N[2]*M[2] + N[3]*M[3] + N[4]*M[4]
  - M is an mask array containing the weights (aka. Convolution kernel)
  - Different effects can be achieved by using different masks
  - Performed on all elements:



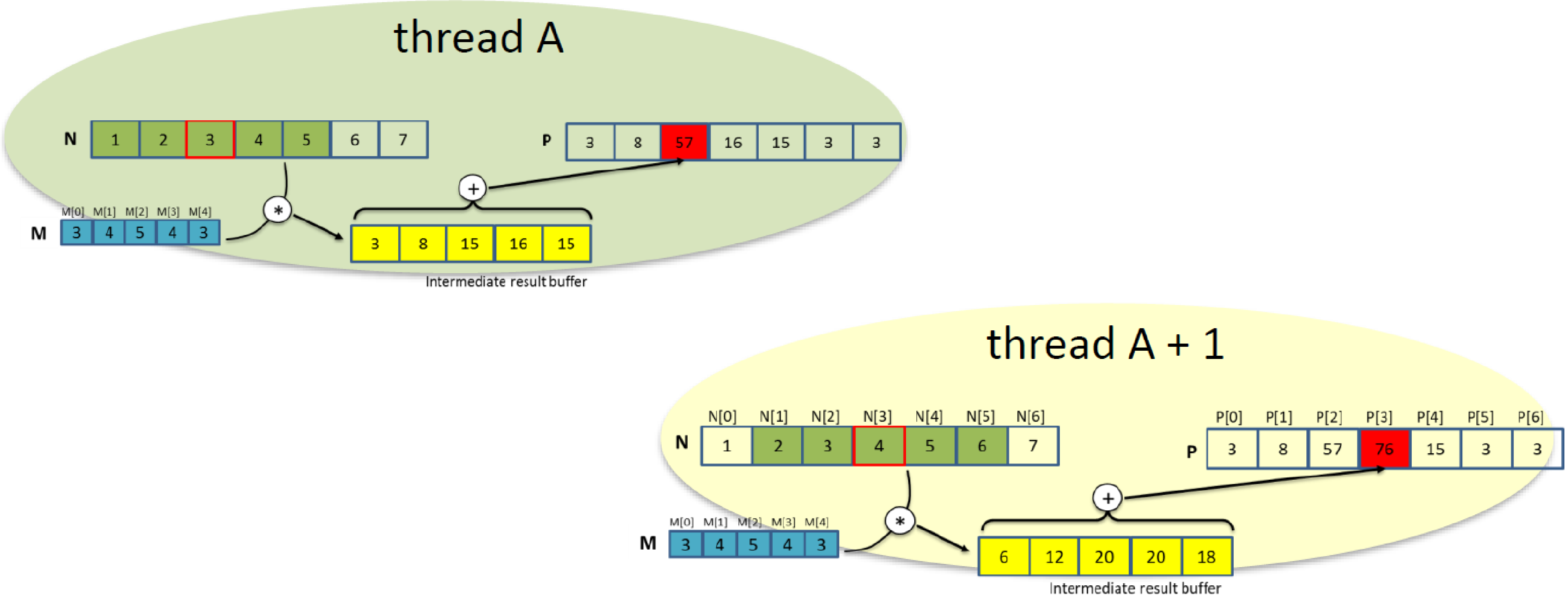
Convolution Boundary Condition

- **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.)



1D Convolution Thread Organization

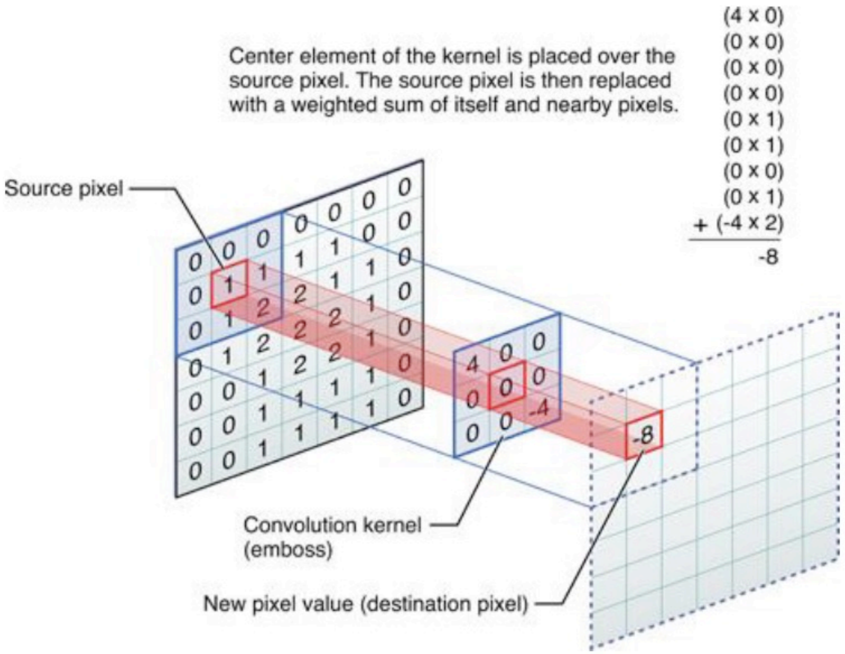
- **Each thread can take care of one output entry**



1D Convolution Kernel

```
__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 Example



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

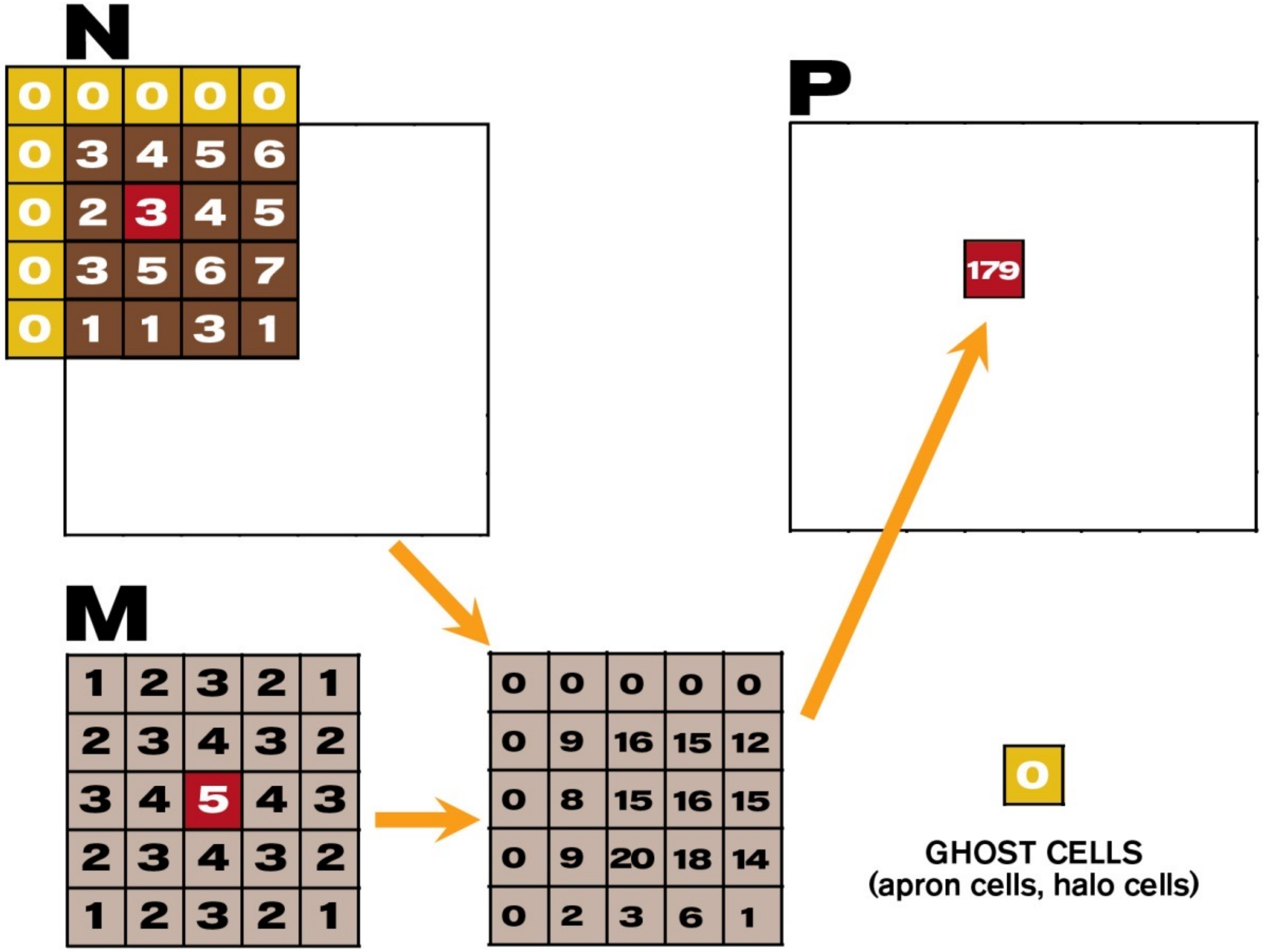
1	2	3	4	5			
2	3	4	5	6			
3	4	321	6	7			
4	5	6	7	8			
5	6	7	8	5			

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
4	16	25	24	21
8	15	24	21	16
5	12	21	16	5

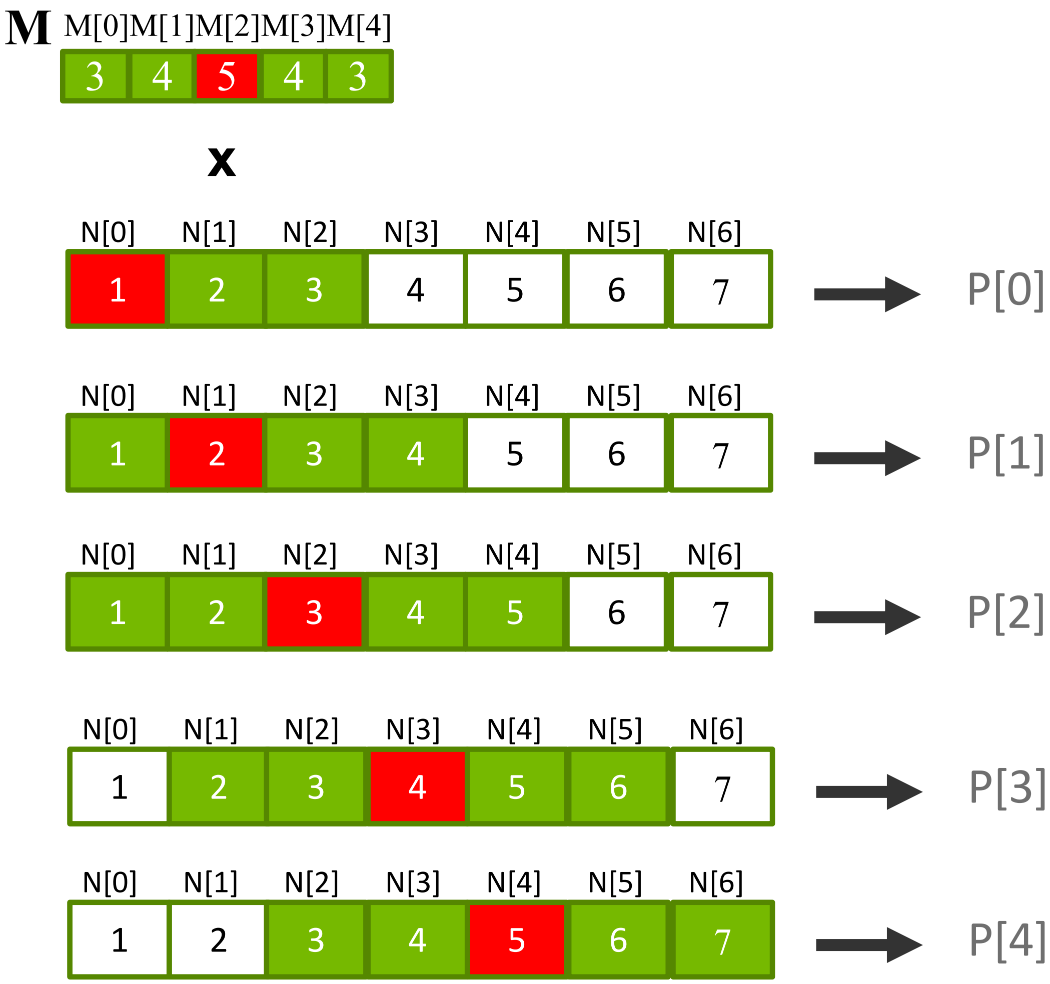
2D Convolution Boundary Condition



2D Convolution Kernel

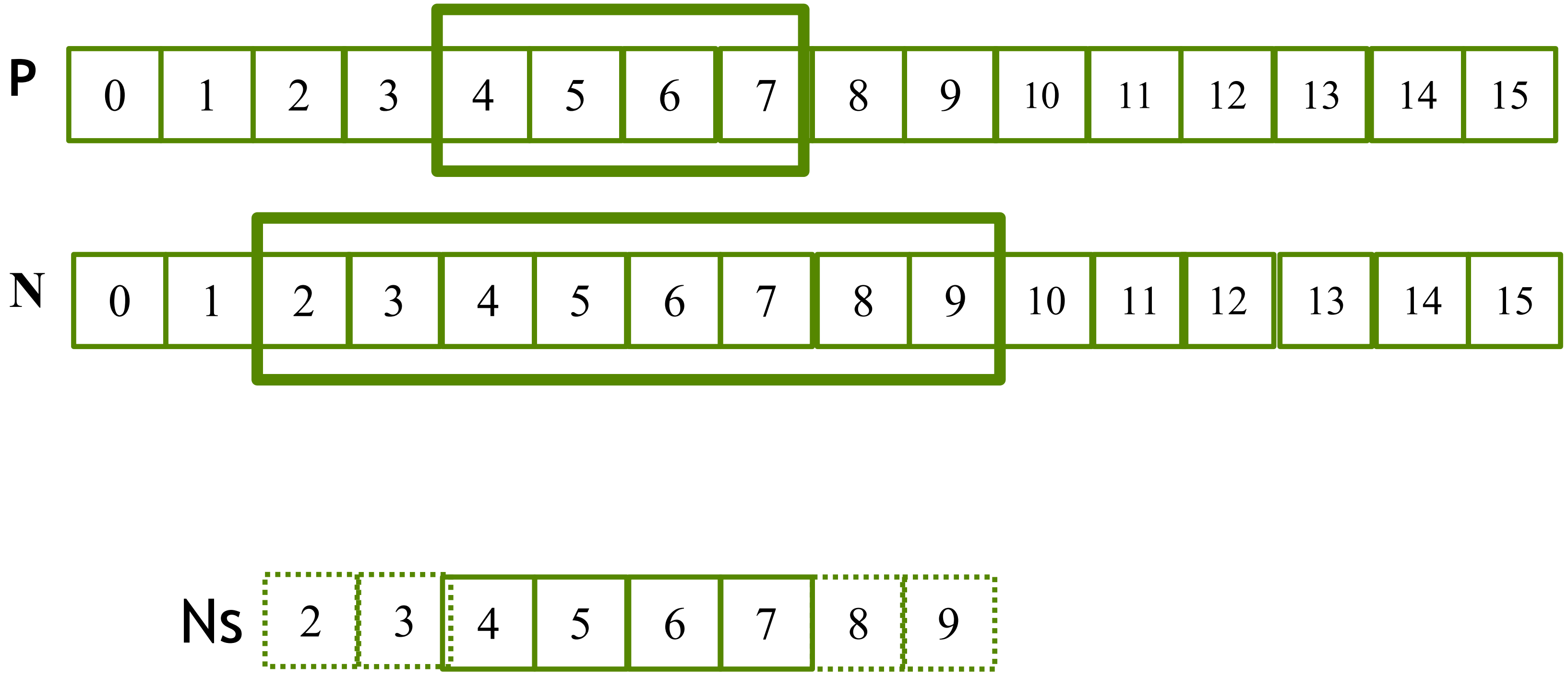
```
 __global__ void convolution_2D_basic_kernel(unsigned char * in, unsigned char * mask, unsigned char *
out, int maskwidth, int w, int h) {
      int Col  =   blockIdx.x * blockDim.x + threadIdx.x;
      int Row  = blockIdx.y * blockDim.y + threadIdx.y;

      if (Col < w && Row < h) {
          int pixVal = 0;
          N_start_col  = Col - (maskwidth/2);
          N_start_row = Row - (maskwidth/2);

          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;
                  if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) // Verify valid pixel
                      pixVal += in[curRow * w + curCol] * mask[j * maskwidth + k];
              }
          }
          out[Row * w + Col] = (unsigned char)(pixVal);
      }
  }
```

Optimization Opportunities



- **M does not change during kernel**
  - All elements used by all threads
  - Also always accessed in the same order
  - Good for Constant Memory
- **N also does not change during kernel**
  - Not all threads use the same elements, but have some overlap in nearby threads
  - Larger capacity requirement
  - Good for Shared Memory

Tiled 1D Convolution



- **How many threads in a block?**
  - 4: Some threads need to load more than one input element into the shared memory
  - 8: Some threads will not participate in calculating output elements

Tiled 1D Convolution Kernel

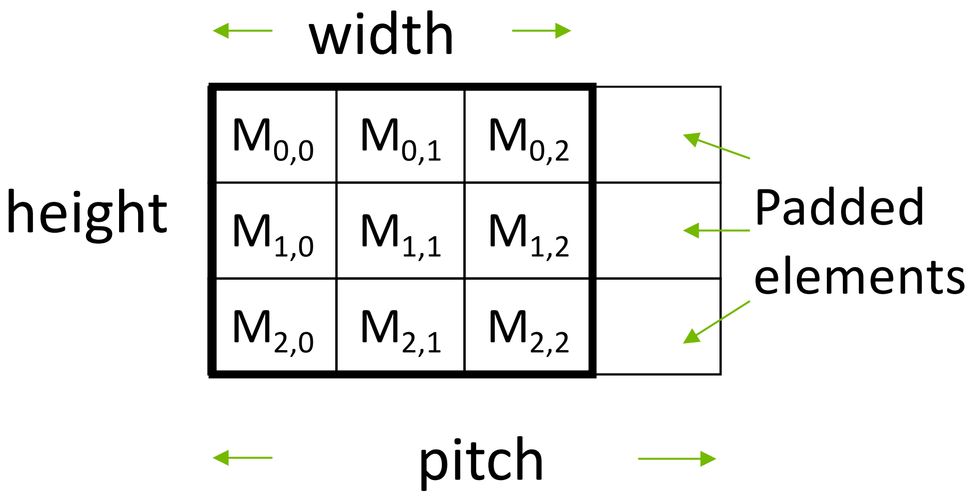
```
…
float output = 0.0f;
index_o= blockIdx.x* O_TILE_WIDTH + threadIdx.x;
index_i= index_o-Mask_Width/2;

if((index_i>= 0) && (index_i< Width)) {
    Ns[threadIdx.x] = N[index_i];
}else{
    Ns[threadIdx.x] = 0.0f;
}

if (threadIdx.x < O_TILE_WIDTH){
    output = 0.0f;

    for(j = 0; j < Mask_Width; j++) {
        output += M[j] * Ns[j+threadIdx.x];
    }
    P[index_o] = output;
}
…
```

Tiled 2D Convolution Kernel

```
…
float output = 0.0f;
__shared__ float Ns[TILE_WIDTH+MASK_WIDTH-1][TILE_WIDTH+MASK_WIDTH-1];

if((row_i>= 0) && (row_i< N.height) && (col_i>= 0)  && (col_i< N.width) )
    Ns[ty][tx] = N.elements[row_i*N.width+ col_i];
else
    Ns[ty][tx] = 0.0f;

if(ty < TILE_WIDTH && tx< TILE_WIDTH){
    for(i= 0; i< 5; i++)
        for(j = 0; j < 5; j++)
            output += Mc[i][j] * Ns[i+ty][j+tx];
}

if(row_o< P.height&& col_o< P.width)
    P.elements[row_o* P.width+ col_o] = output;
…
```

Enhancing Memory Efficiency

- **It is sometimes desirable to pad each row of a 2D matrix to multiples of cache line size**
- **cudaMallocPitch (void** devPtr, size_t* pitch, size_t width, size_t height)**
- **Example: a 3X3 matrix padded into a 3X4 matrix**



Reduction

- **Partition and Summarize**
  - Requirement: No required order of processing elements in a data set (associative and commutative)
  - E.g., Max, Min, Sum, Product

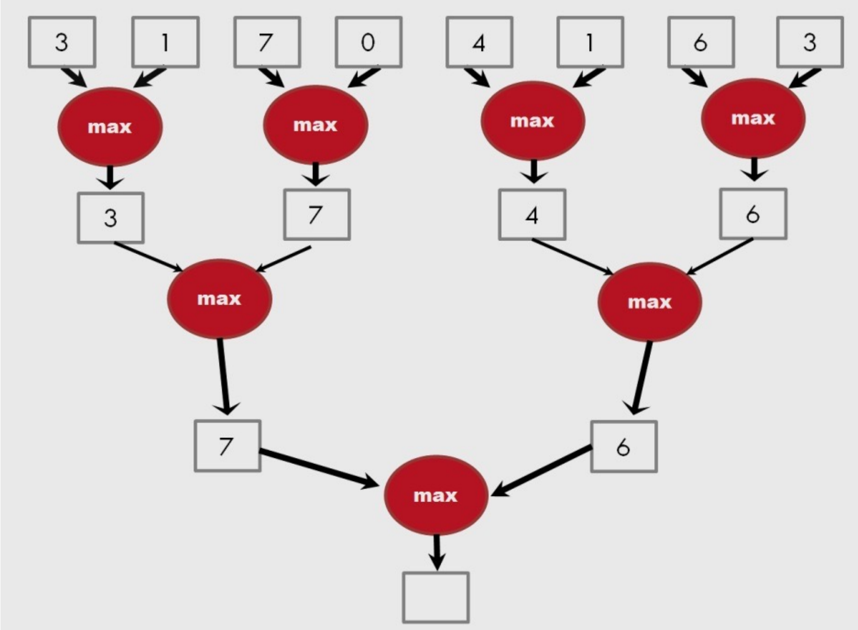
- **Steps:**
  - Initialize the result as an identity value for the reduction operation
  - Partition the data set into smaller chunks
  - Have each thread to process a chunk
  - Use a reduction tree to summarize the results from each chunk into the final answer

- **Often used in combination with other patterns**
  - Clean up after some commonly used parallelizing transformations
  - Privatization

Reduction Performance Analysis

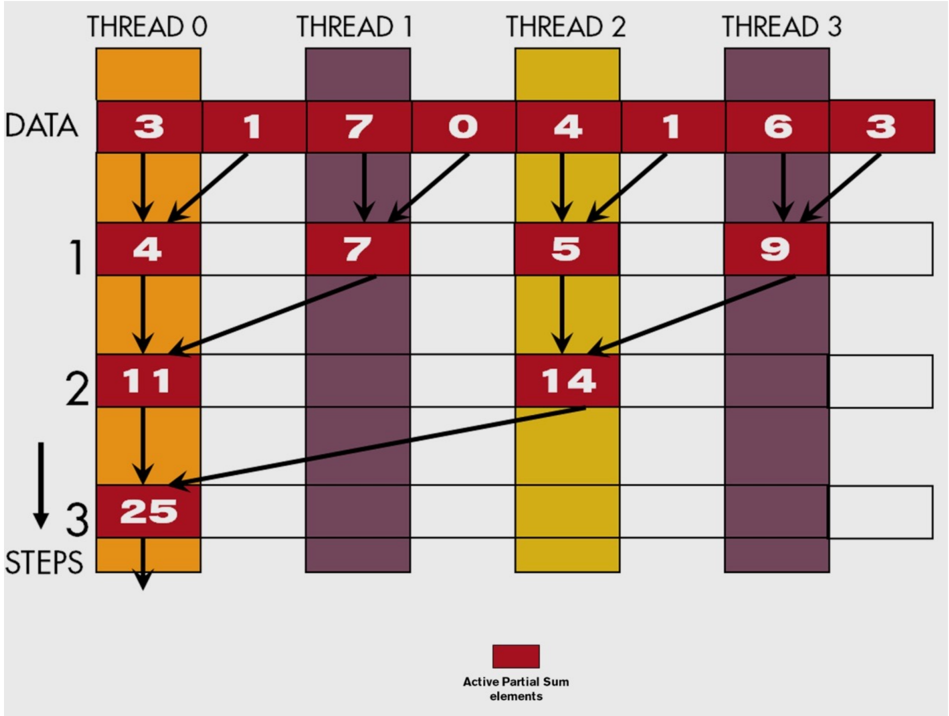
- **Sequential reduction:** Iterate through the input and perform the reduction operation between the result value and the current input value
  - N reduction operations performed for N input values: O(N)

- **Parallel reduction:** Log(N) steps for N input values
  - Operations: (1/2)N + (1/4)N + (1/8)N + … 1 = N – 1
    - A work-efficient parallel algorithm
  - Average Parallelism: (N-1) / Log(N)
    - However, first step requires N/2 nodes
    - Not resource efficient



Naïve Reduction

- Each thread is responsible for an even-index location of the partial sum vector
- After each step, half of the threads are no longer needed
- One of the inputs is always from the same location
- In each step, one of the inputs comes from an increasing distance away



SJSU SAN JOSÉ STATE UNIVERSITY

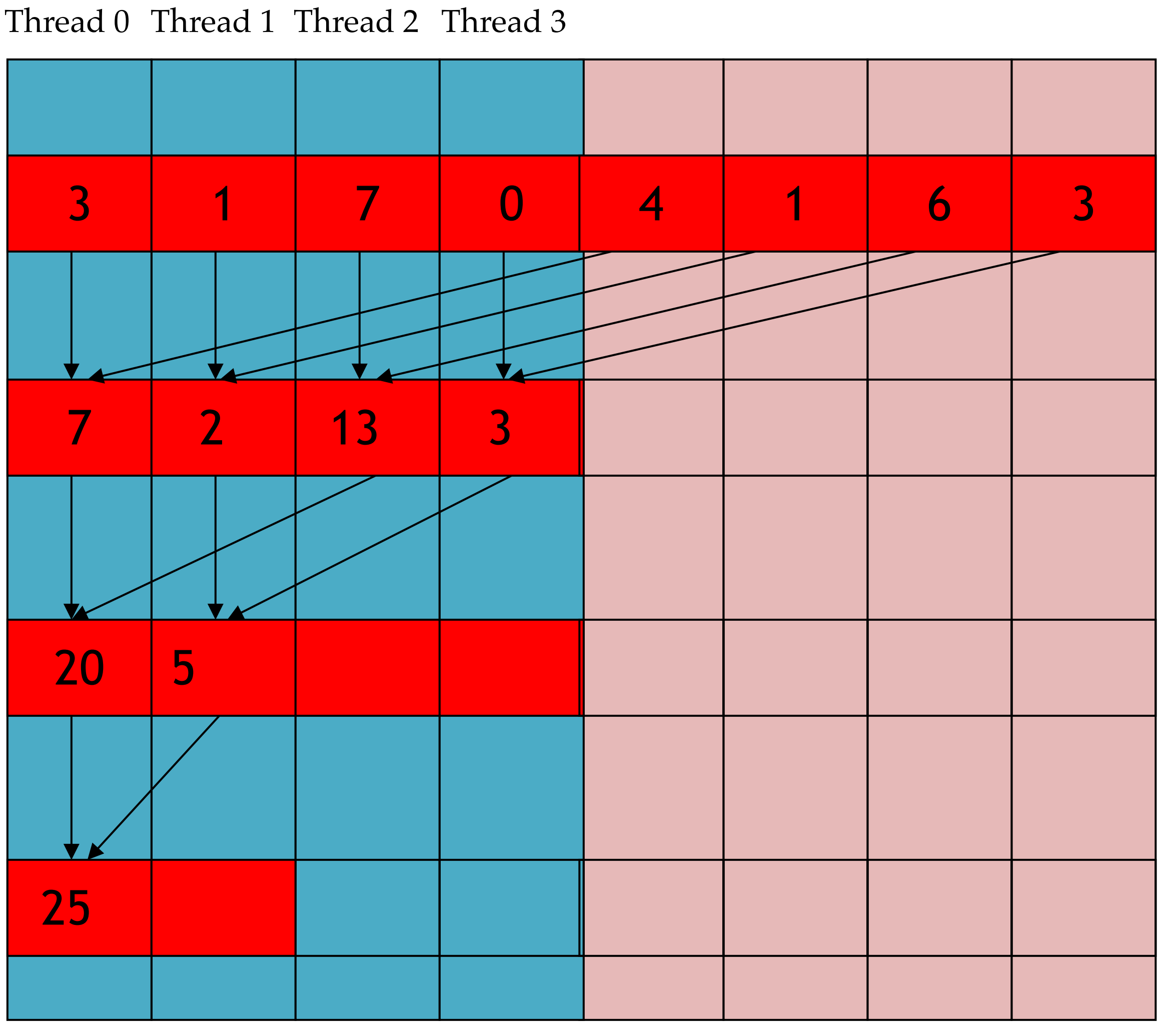
Naïve Reduction Kernel

```
…

__shared__ float partialSum[2*BLOCK_SIZE];

unsigned int t = threadIdx.x;
unsigned int start = 2 * blockIdx.x * blockDim.x;
partialSum[t] = input[start + t];
partialSum[blockDim+t] = input[start + blockDim.x + t];

for (unsigned int stride = 1; stride <= blockDim.x; stride *= 2){
   __syncthreads();

   if (t % stride == 0)
     partialSum[2 * t]+= partialSum[2 * t + stride];
 }
…
```

Problems with Naïve Reduction

- **In each iteration, two control flow paths will be sequentially traversed for each warp**
  - Threads that perform addition and threads that do not
  - Threads that do not perform addition still consume execution resources
- **Half or fewer of threads will be executing after the first step**
  - All odd-index threads are disabled after first step
  - After the 5th step, entire warps in each block will fail the if test, poor resource utilization but no divergence
  - This can go on for a while, up to 6 more steps (stride = 32, 64, 128, 256, 512, 1024), where each active warp only has one productive thread until all warps in a block retire
- **Solution?**
  - Keep the active threads consecutive

A better Thread Organization



SAN JOSÉ STATE UNIVERSITY *powering* SILICON VALLEY