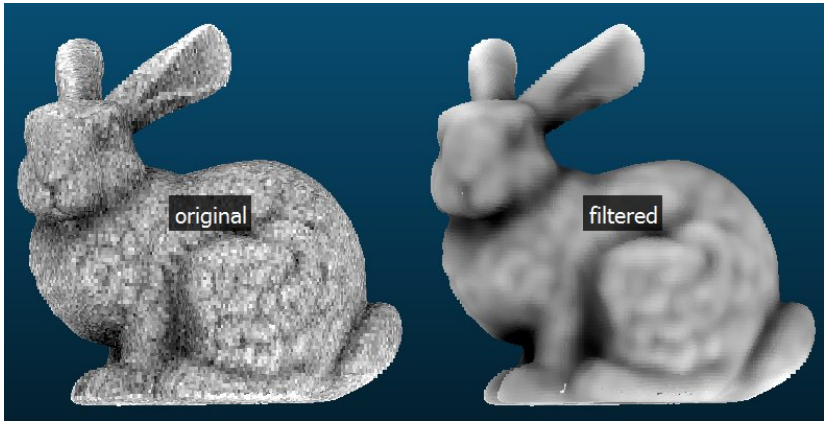
CENG443
Heterogeneous Parallel Programming

Convolution

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



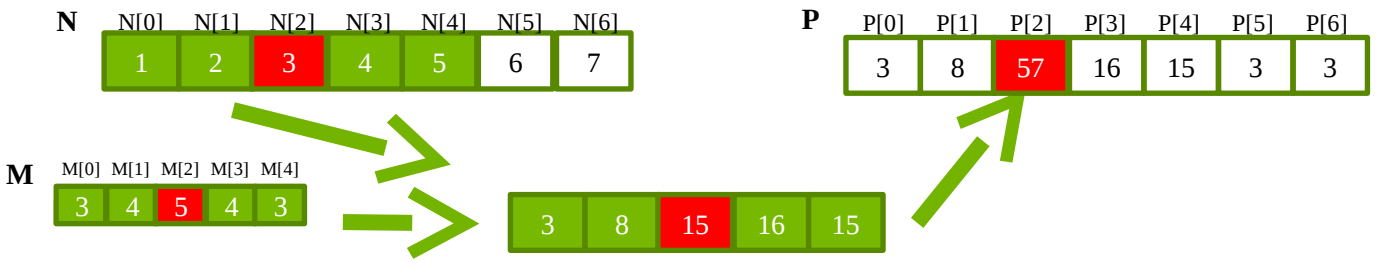
Convolution Definition

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

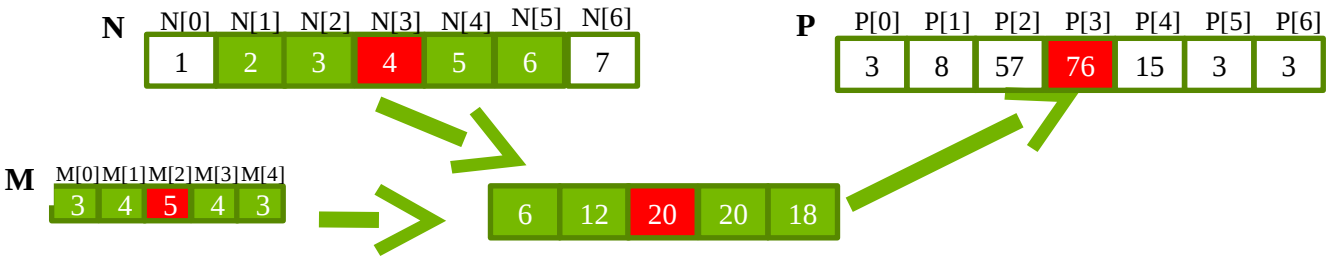
- Refer to these mask arrays as convolution masks to avoid confusion
- The value pattern of the mask array elements defines the type of filtering done
- The image blur example before is a special case where all mask elements are of the same value and hard coded into the source code

1D Convolution Example

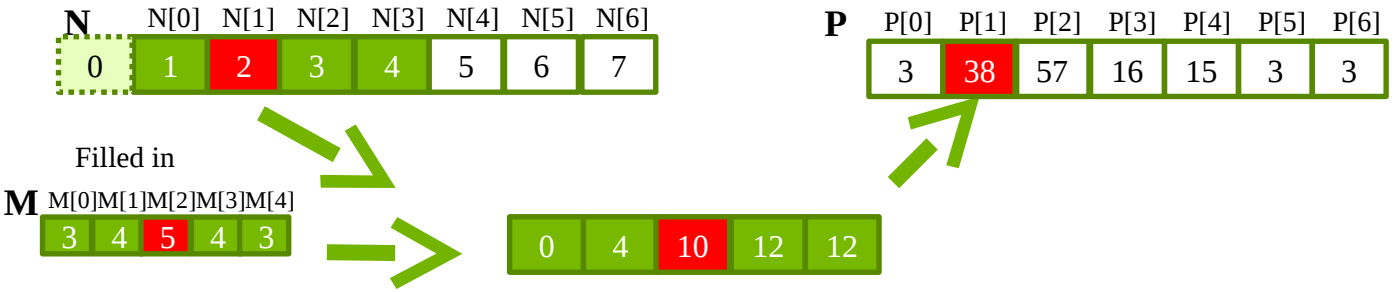


P[2] = N[0]*M[0] + N[1]*M[1] + N[2]*M[2] + N[3]*M[3] + N[4]*M[4]

1D Convolution Example



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.)

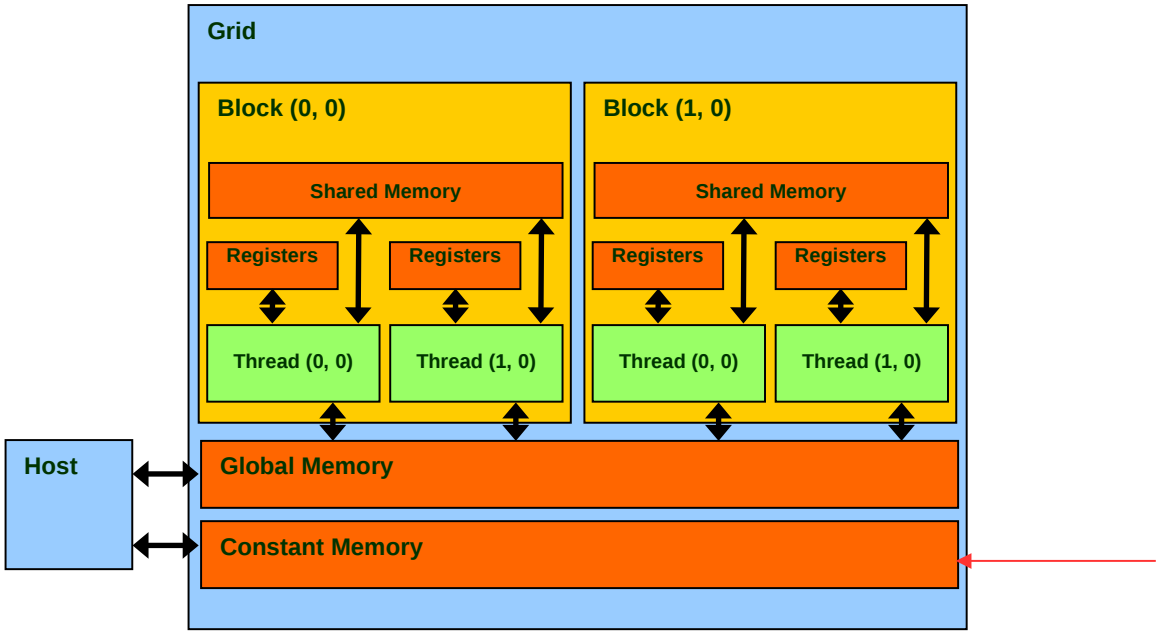
1D Convolution Kernel

```
__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;
}
```

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;
}
```

Constant Memory



- Read only
- Short latency and high bandwidth when all threads access the same location
- Visible to all thread blocks
- Limited size, 64KB

Constant Memory Usage

Mask M

- Small (less than 10 elements in each dimension (1000 in 3D))
- Contents not changed
- All threads need to access mask elements in the same order

Very suitable for constant memory

```
#define MAX_MASK_WIDTH 10
__constant__ float M[MAX_MASK_WIDTH];


cudaMemcpyToSymbol(M, M_h, Mask_Width*sizeof(float));
```

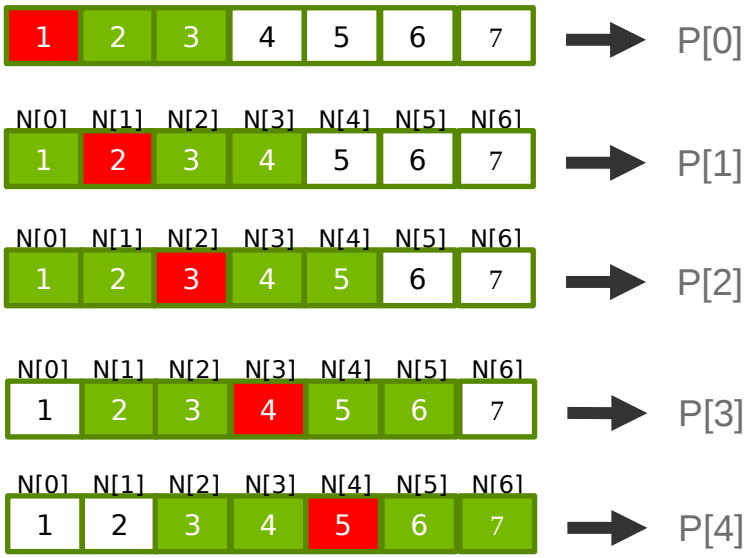
1D Convolution Kernel

```
__global__ void convolution_1D_basic_kernel(float *N, 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;
}
```

Tiling Opportunity

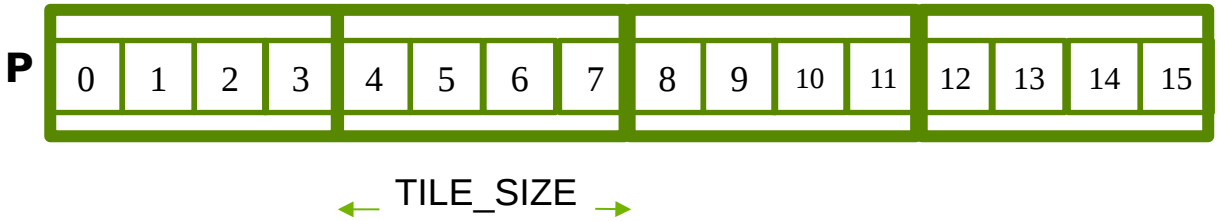
Calculation of adjacent output elements involve shared input elements

1D convolution Mask_Width of width 5



N[2] is used in calculation of P[0], P[1], P[2], P[3], P[4]

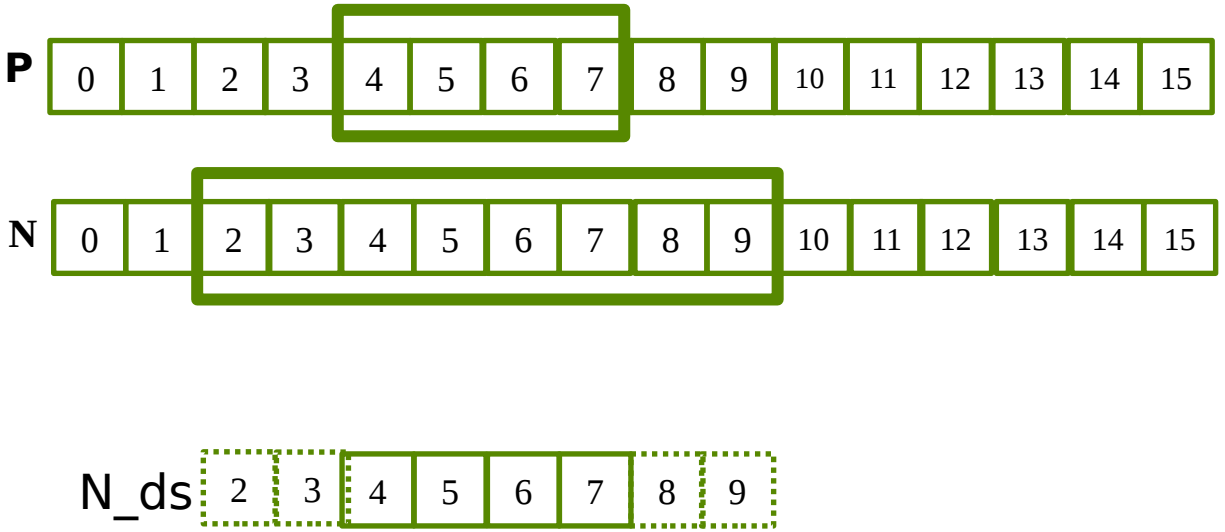
Tiled 1D Convolution – Output Tile



Each thread block calculates an output tile
Each thread calculates one output element

TILE_SIZE is 4 in this example

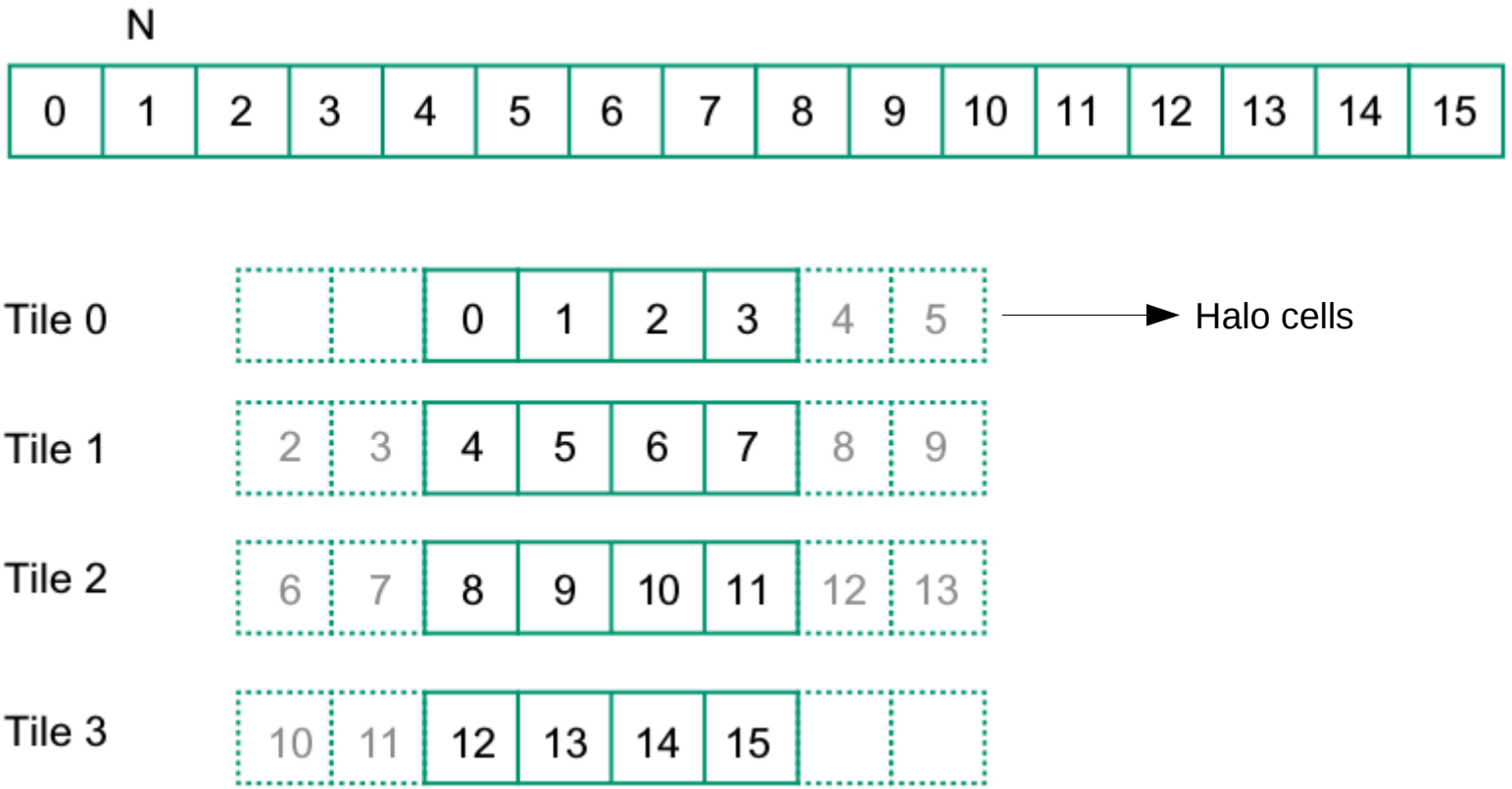
Tiled 1D Convolution – Input Tile



Each input tile needs the values to calculate the corresponding output tile

```
__shared__ float N_ds[TILE_SIZE + MAX_MASK_WIDTH - 1];
```

Tiled 1D Convolution Example



Load Left Halo Cells

The last n threads load n left halo cells (n = Mask_Width/2)

```
int halo_index_left = (blockIdx.x - 1)*blockDim.x +
threadIdx.x;
    if (threadIdx.x >= blockDim.x - n) {
     N_ds[threadIdx.x - (blockDim.x - n)] =
       (halo_index_left < 0) ? 0 : N[halo_index_left];
   }
```

Load Center Cells

Each thread loads 1 center cell

```
N_ds[n + threadIdx.x] = N[blockIdx.x*blockDim.x +
threadIdx.x];
```

Load Right Halo Cells

The first n threads load n right halo cells (n = Mask_Width/2)

```
int halo_index_right = (blockIdx.x + 1)*blockDim.x +
threadIdx.x;
    if (threadIdx.x < n) {
      N_ds[n + blockDim.x + threadIdx.x] =
      (halo_index_right >= Width) ? 0 : N[halo_index_right];
   }
```

Calculation

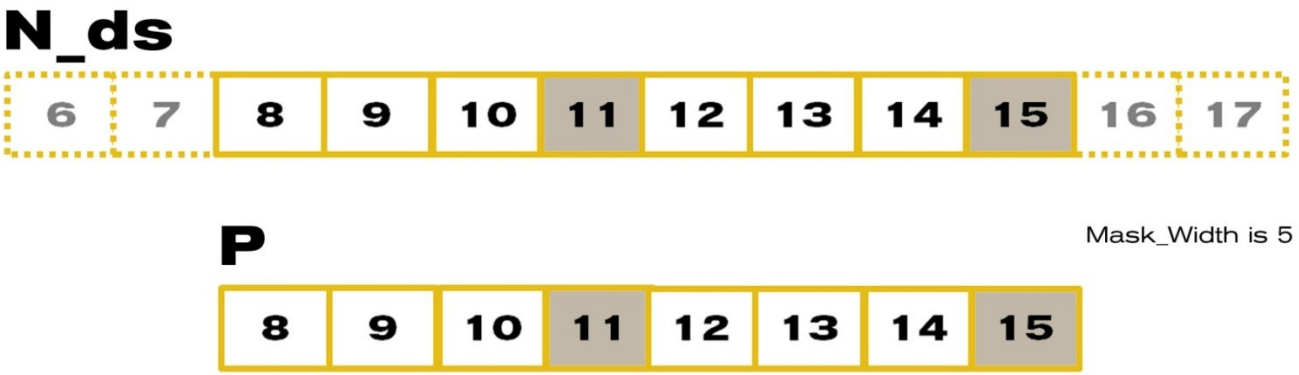
Each thread performs calculation of one output element

```
float Pvalue = 0;
for(int j = 0; j < Mask_Width; j++) {
    Pvalue += N_ds[threadIdx.x + j]*M[j];
}
P[i] = Pvalue;
```

Tiled 1D Convolution Kernel

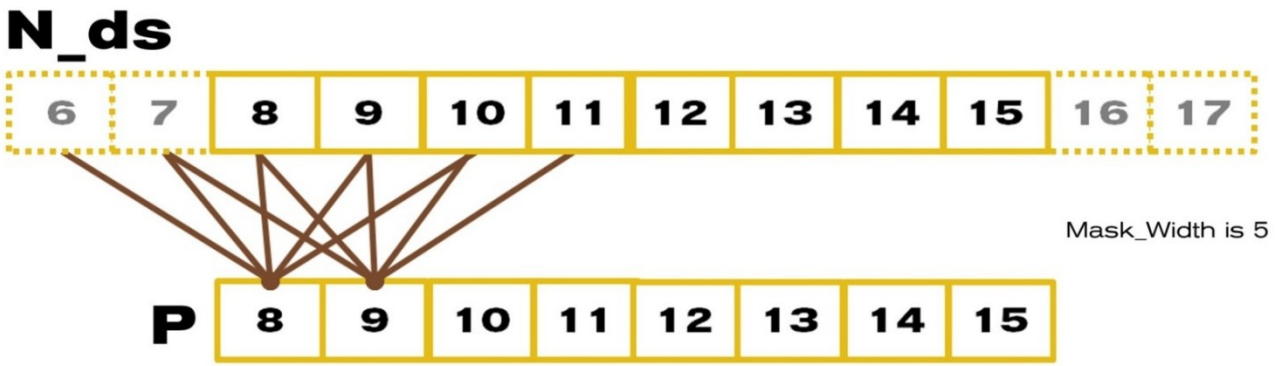
```
__global__ void convolution_1D_tiled_kernel(float *N, float *P, int Mask_Width,
  int Width) {

  int i = blockIdx.x*blockDim.x + threadIdx.x;
  __shared__ float  N_ds[TILE_SIZE + MAX_MASK_WIDTH - 1];

  int n = Mask_Width/2;

  int halo_index_left = (blockIdx.x - 1)*blockDim.x + threadIdx.x;
  if (threadIdx.x >= blockDim.x - n) {
    N_ds[threadIdx.x - (blockDim.x - n)] =
      (halo_index_left < 0) ? 0 : N[halo_index_left];
  }

  N_ds[n + threadIdx.x] = N[blockIdx.x*blockDim.x + threadIdx.x];

  int halo_index_right = (blockIdx.x + 1)*blockDim.x + threadIdx.x;
  if (threadIdx.x < n) {
    N_ds[n + blockDim.x + threadIdx.x] =
      (halo_index_right >= Width) ? 0 : N[halo_index_right];
  }

  __syncthreads();

  float Pvalue = 0;
  for(int j = 0; j < Mask_Width; j++) {
    Pvalue += N_ds[threadIdx.x + j]*M[j];
  }
  P[i] = Pvalue;

}
```

An 8-Element Convolution Tile



For Mask_Width=5, we load 8+5-1=12 elements
(12 memory loads)

Each Output element uses 5 N elements



P[8] uses N[6], N[7], N[8], N[9], N[10]
P[9] uses N[7], N[8], N[9], N[10], N[11]
P[10] use N[8], N[9], N[10], N[11], N[12]
…
P[14] uses N[12], N[13], N[14], N[15], N[16]
P[15] uses N[13], N[14], N[15], N[16], N[17]

Tiling Benefit

(8+5-1)=12 elements loaded

8*5 global memory accesses replaced by shared memory accesses

This gives a bandwidth reduction of 40/12=3.3

Caching

Recent GPUs such as Fermi provide general L1 and L2 caches, where L1 is private to each streaming multiprocessor and L2 is shared among all streaming multiprocessors

This leads to an opportunity for the blocks to take advantage of the fact that their halo cells may be available in the L2 cache

The memory accesses to these halo cells may be naturally served from L2 cache without causing additional DRAM traffic

We can leave the accesses to these halo cells in the original N elements rather than loading them into the N_ds

Simpler Tiled 1D Convolution

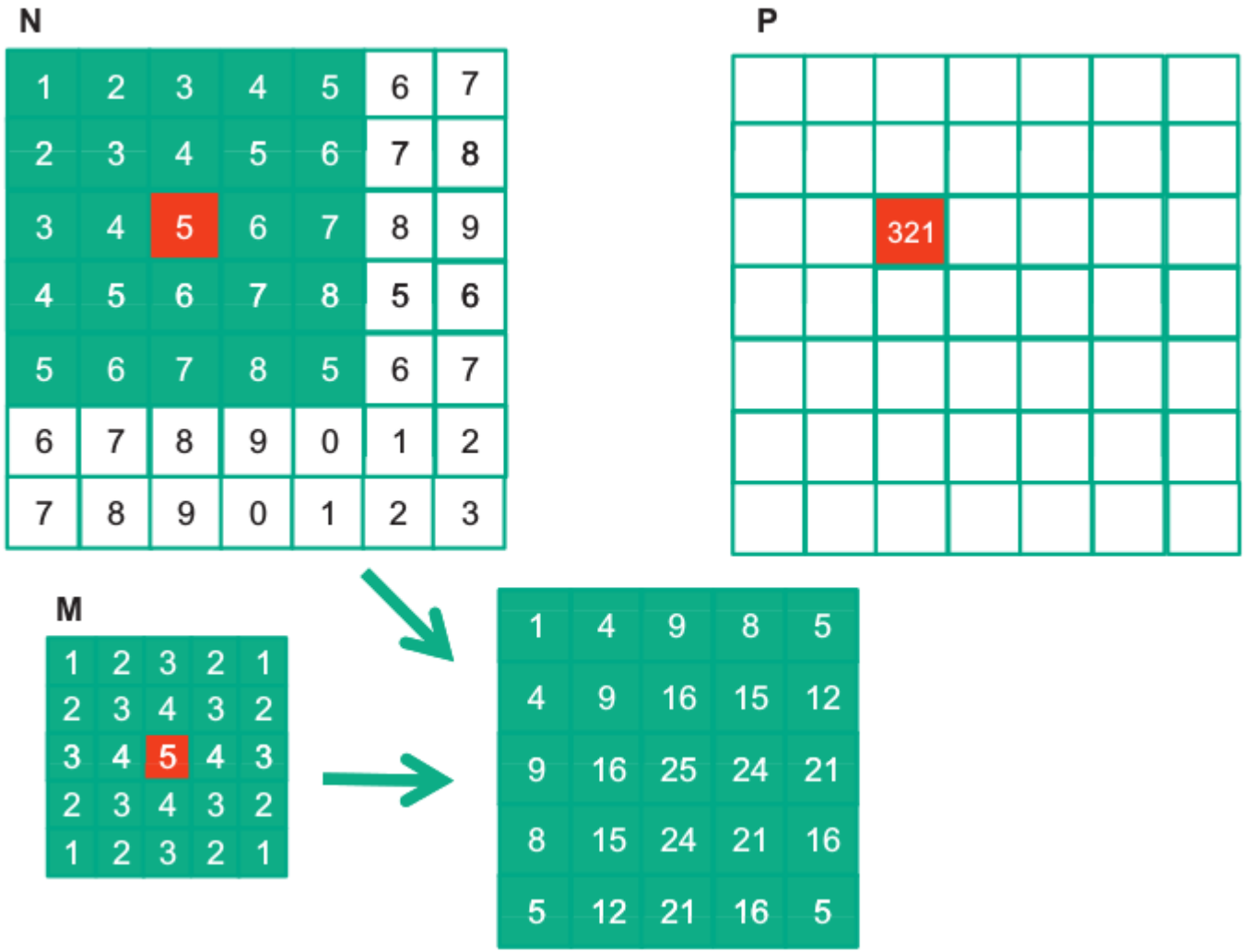
N_ds array only needs to hold the internal elements of the tile, others from global memory (potentially from L2 cache)

```
__global__ void convolution_1D_tiled_caching_kernel(float *N, float *P, int
  Mask_Width,int Width) {

  int i = blockIdx.x*blockDim.x + threadIdx.x;
  __shared__ float  N_ds[TILE_SIZE];

  N_ds[threadIdx.x] = N[i];

  __syncthreads();

  int This_tile_start_point = blockIdx.x * blockDim.x;
  int Next_tile_start_point = (blockIdx.x + 1) * blockDim.x;
  int N_start_point = i - (Mask_Width/2);
  float Pvalue = 0;
  for (int j = 0; j < Mask_Width; j++) {
     int N_index = N_start_point + j;
     if (N_index >= 0  && N_index < Width) {
       if ((N_index >= This_tile_start_point)
         && (N_index < Next_tile_start_point)) {
         Pvalue += N_ds[threadIdx.x+j-(Mask_Width/2)]*M[j];
       } else {
         Pvalue += N[N_index] * M[j];
       }
     }
  }
  P[i] = Pvalue;

}
```

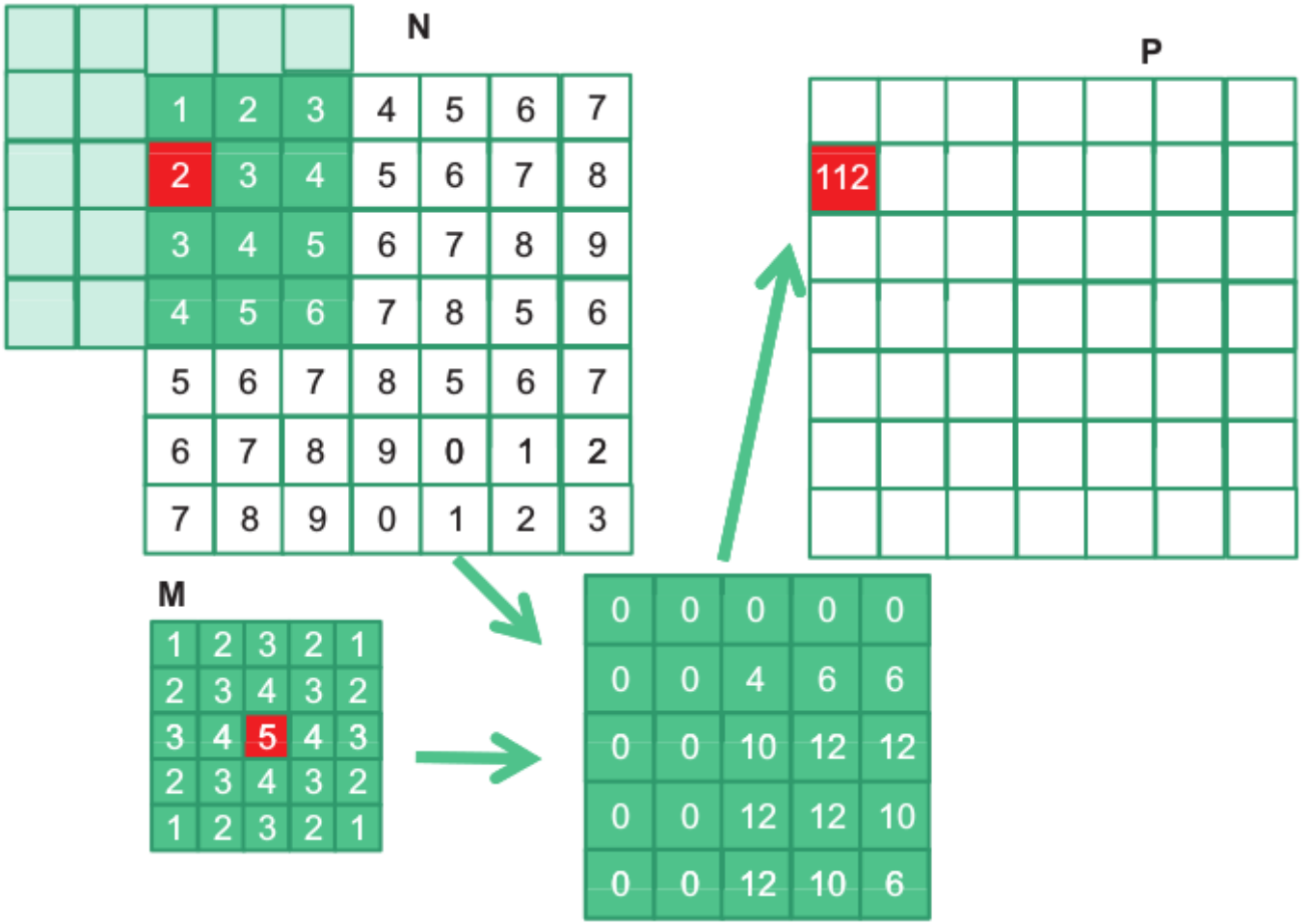
Simpler Tiled 1D Convolution

```
__syncthreads();
int This_tile_start_point = blockIdx.x * blockDim.x;
int Next_tile_start_point = (blockIdx.x + 1) * blockDim.x;
int N_start_point = i - (Mask_Width/2);
float Pvalue = 0;
for (int j = 0; j < Mask_Width; j++) {
   int N_index = N_start_point + j;
   if (N_index >= 0 && N_index < Width) {
        if ((N_index >= This_tile_start_point)&& (N_index < Next_tile_start_point)){
             Pvalue += N_ds[threadIdx.x+j-(Mask_Width/2)]*M[j];
        } else {
             Pvalue += N[N_index] * M[j];
        }
   }
}
P[i] = Pvalue;
```

2D Convolution Example



2D Convolution-Ghost Cells



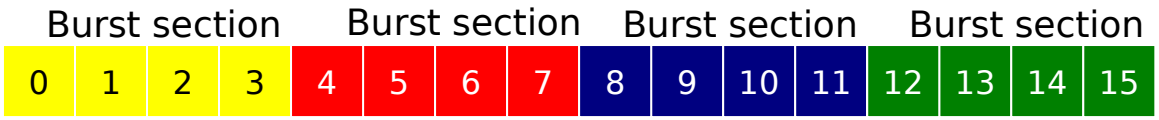
2D Convolution Kernel

```
__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);
```

2D Convolution Kernel

```
        // 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);
    }
}
```

DRAM Burst



Each address space is partitioned into burst sections

- Whenever a location is accessed, all other locations in the same section are also delivered to the processor

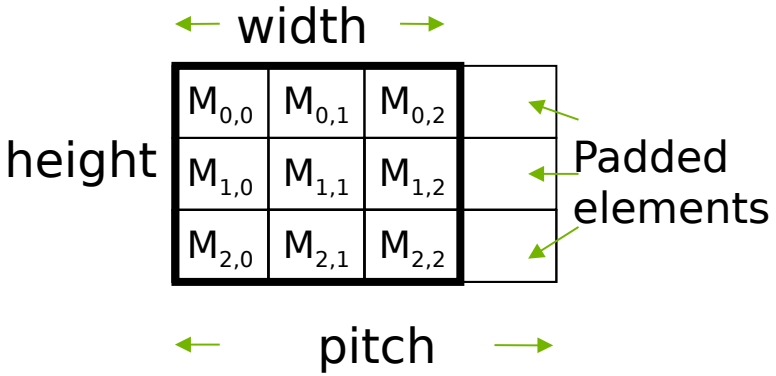
Basic example: a 16-byte address space, 4-byte burst sections

- In practice, we have at least 4GB address space, burst section sizes of 128-bytes or more

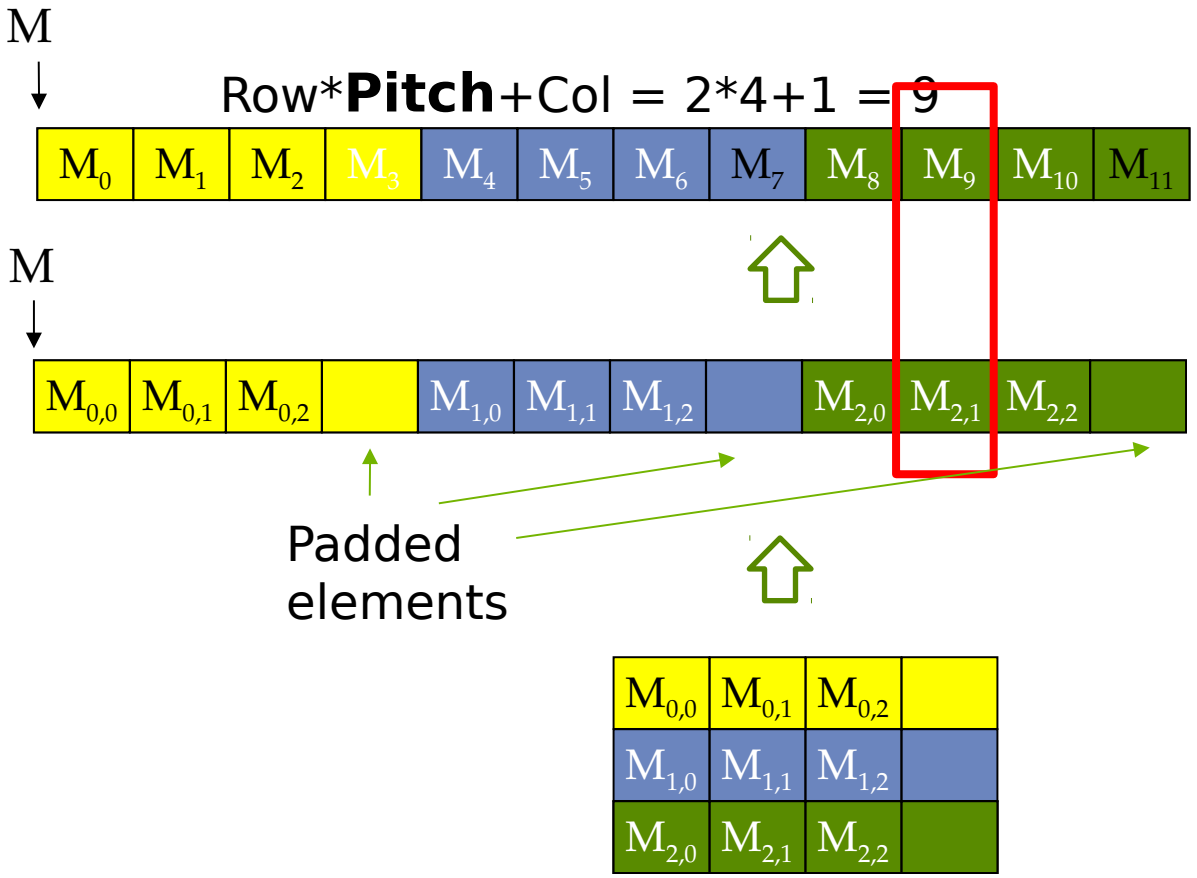
2D Image with Padding

If the matrix width is not multiple of DRAM burst size, padding is helpful

So each row starts at the DRAM burst boundary



Row-Major Layout with Pitch

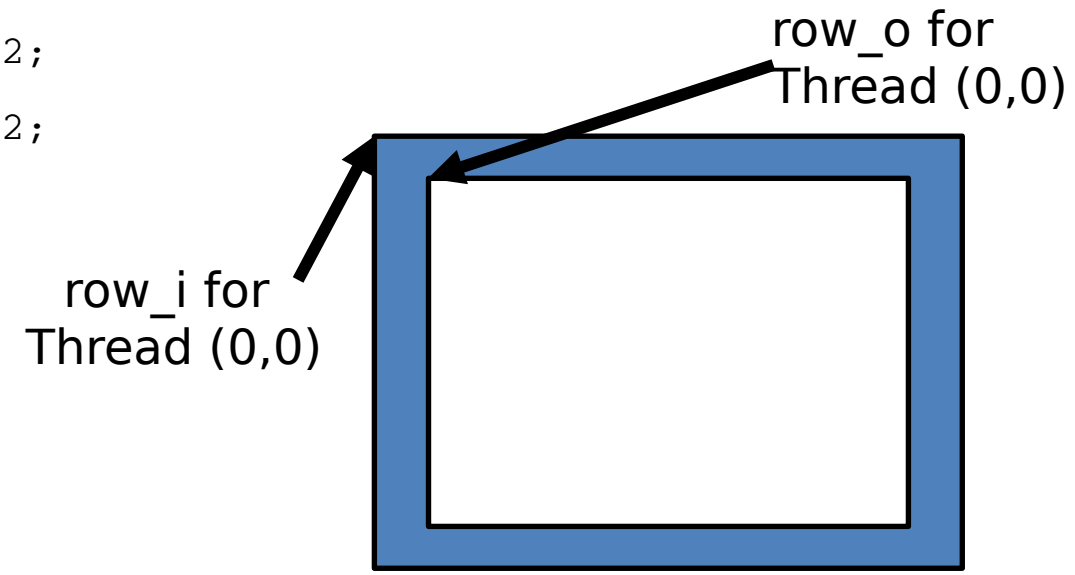


Tiled 2D Convolution

```
int tx = threadIdx.x;
int ty = threadIdx.y;
int row_o = blockIdx.y*O_TILE_WIDTH + ty;
int col_o = blockIdx.x*O_TILE_WIDTH + tx;

int row_i = row_o - Mask_Width/2;
int col_i = col_o - Mask_Width/2;
```

BLOCK_WIDTH should be

O_TILE_WIDTH + (MASK_WIDTH-1)



Data Load

```
if((row_i >= 0) && (row_i < height) && (col_i >= 0)  && (col_i < width)) {
    N_ds[ty][tx] = N[row_i * width + col_i];
  } else{
    N_ds[ty][tx] = 0.0f;
  }
```

Calculate/Write Output

```
//only the first O_TILE_WIDTH threads perform calculations
float output = 0.0f;
 if(ty < O_TILE_WIDTH && tx < O_TILE_WIDTH){
    for(i = 0; i < MASK_WIDTH; i++) {
      for(j = 0; j < MASK_WIDTH; j++) {
        output += M[i][j] * N_ds[i+ty][j+tx];
      }
    }
    if(row_o < height && col_o < width)
       P[row_o*width + col_o] = output;
 }
```