Introduction to Parallel & Distributed Computing

GPU Programming in HIP (4): misc. & convolutions

Lecture 9, Spring 2024

Instructor: 罗国杰

gluo@pku.edu.cn

Occupancy

- On current GPUs, work groups get mapped to compute units
- Enough resources ⇒ multiple work groups mapped at the same time
  - Warp/wavefronts from another work group can be swapped in to hide latency
- Resources are fixed per compute unit
  - number of registers
  - local memory size
  - maximum number of threads
- The term *occupancy* is used to describe how well the resources of the compute unit are being utilized

Occupancy – Registers

- The availability of registers is one of the major limiting factor for larger kernels
- Example: Consider a GPU with 16384 registers per compute unit running a kernel that requires 35 registers per thread
  - Each compute unit can execute at most 468 threads
  - This affects the choice of workgroup size
    - A workgroup of 512 is not possible
    - Only 1 workgroup of 256 threads is allowed at a time, even though 212 more threads could be running
    - 3 workgroups of 128 threads are allowed, providing 384 threads to be scheduled, etc.

Occupancy – Registers

- **Consider another example:**
  - **A GPU has 16384 registers per compute unit**
  - **The work group size of a kernel is fixed at 256 threads**
  - **The kernel currently requires 17 registers per thread**
- **Given the information, each work group requires 4352 registers**
  - **This allows for 3 active work groups if registers are the only limiting factor**
- **If the code can be restructured to only use 16 registers, then 4 active work groups would be possible**

Occupancy – Local Memory

- GPUs have a limited amount of local memory on each compute unit
  - 32KB of local memory on AMD GPUs
  - 32-48KB of local memory on NVIDIA GPUs
- Local memory limits the number of active work groups per compute unit
- Depending on the kernel, the data per workgroup may be fixed regardless of number of threads (e.g., histograms), or may vary based on the number of threads (e.g., matrix multiplication, convolution)

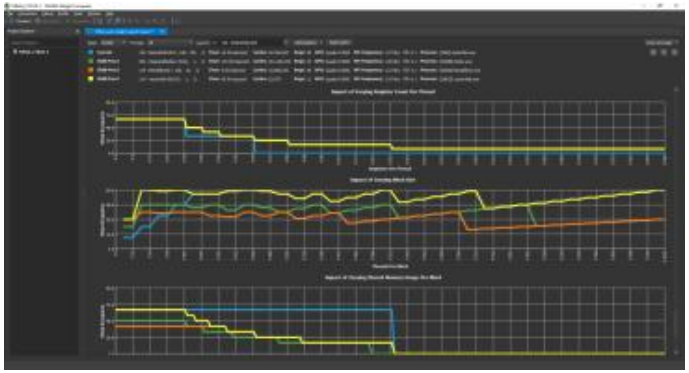
Occupancy – Threads

- GPUs have hardware limitations on the maximum number of threads per work group
  - 256 threads per WG on AMD GPUs
  - 512 threads per WG on NVIDIA GPUs
- NVIDIA GPUs have per-compute-unit limits on the number of active threads and work groups (depending on the GPU model)
  - 768 or 1024 threads per compute unit
  - 8 or 16 warps per compute unit
- AMD GPUs have GPU-wide limits on the number of wavefronts
  - 496 wavefronts on the 5870 GPU (~25 wavefronts or ~1600 threads per compute unit)

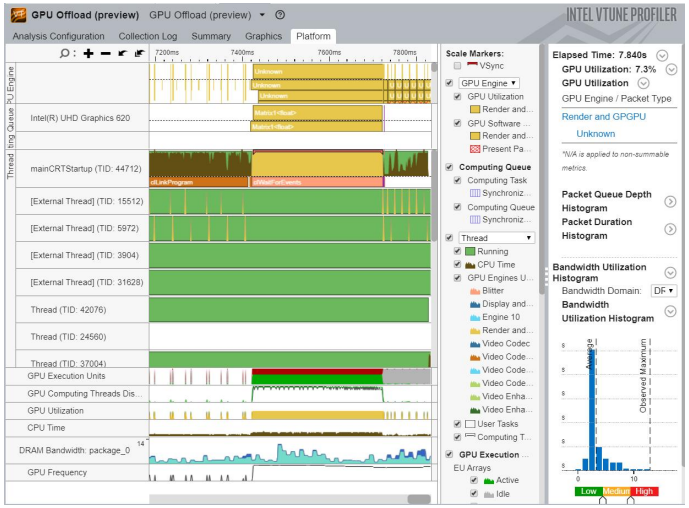
Occupancy – Limiting Factors

- The minimum of these three factors is what limits the active number of threads (or occupancy) of a compute unit
- The interactions between the factors are complex
  - The limiting factor may have either thread or wavefront granularity
  - Changing work group size may affect register or shared memory usage
  - Reducing any factor (such as register usage) slightly may have allow another work group to be active

- NVIDIA Nsight Compute



- Intel VTune Profiler



GPU-Specific Optimizations: Summary

- Although writing a simple SYCL program is relatively easy, optimizing code can be very difficult
  - Improperly mapping loop iterations to SYCL threads can significantly degrade performance
- When creating work groups, hardware limitations (number of registers, size of local memory, etc.) need to be considered
  - Work groups must be sized appropriately to maximize the number of active threads and properly hide latencies
- Vectorization is an important optimization for AMD GPU hardware
  - Though not covered here, vectorization may also help performance when targeting CPUs

Convolution-1D/2D Convolution,Constant Memory and Constant Caching

Objective

- To learn convolution, an important parallel computation pattern
  - -Widely used in signal,image,and video processing
  - -Foundational to stencil computation used in many science and engineering
- Important techniques
  - -Taking advance of cache memories

Convolution and its Applications

Convolution is a widely-used operation in signal processing, image processing, video processing, and computer vision

Convolution applies a filter or mask or kernel* on each element of the input (e.g.,a signal, an image, a frame) to obtain a new value, which is a weighted sum of a set of neighboring input elements

- Smoothing, sharpening, or blurring an image
- Finding edges in an image
- Removing noise,etc.

Applications in machine learning and artificial intelligence

- Convolutional Neural Networks (CNN or ConvNets)

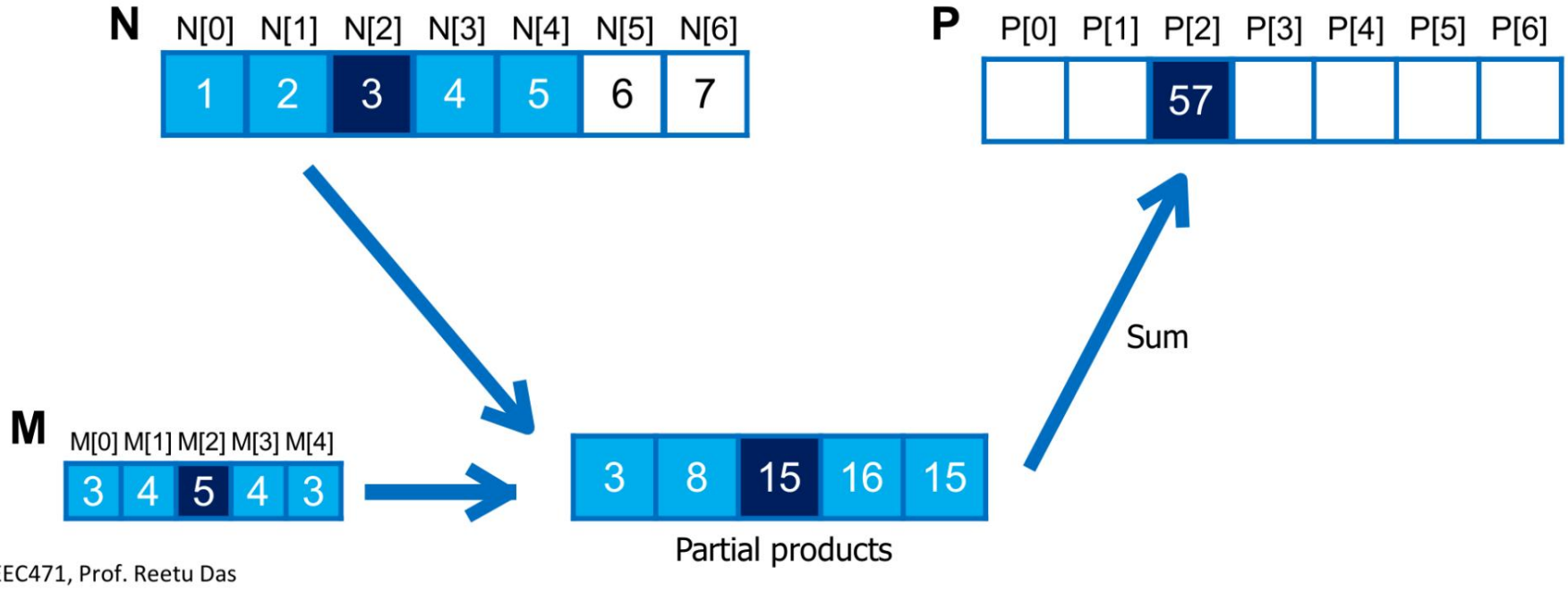
*The term "kernel"may create confusion in the confexR SfDus redalear CUoAGiPii kernel is a function executed by a GPU)

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]:

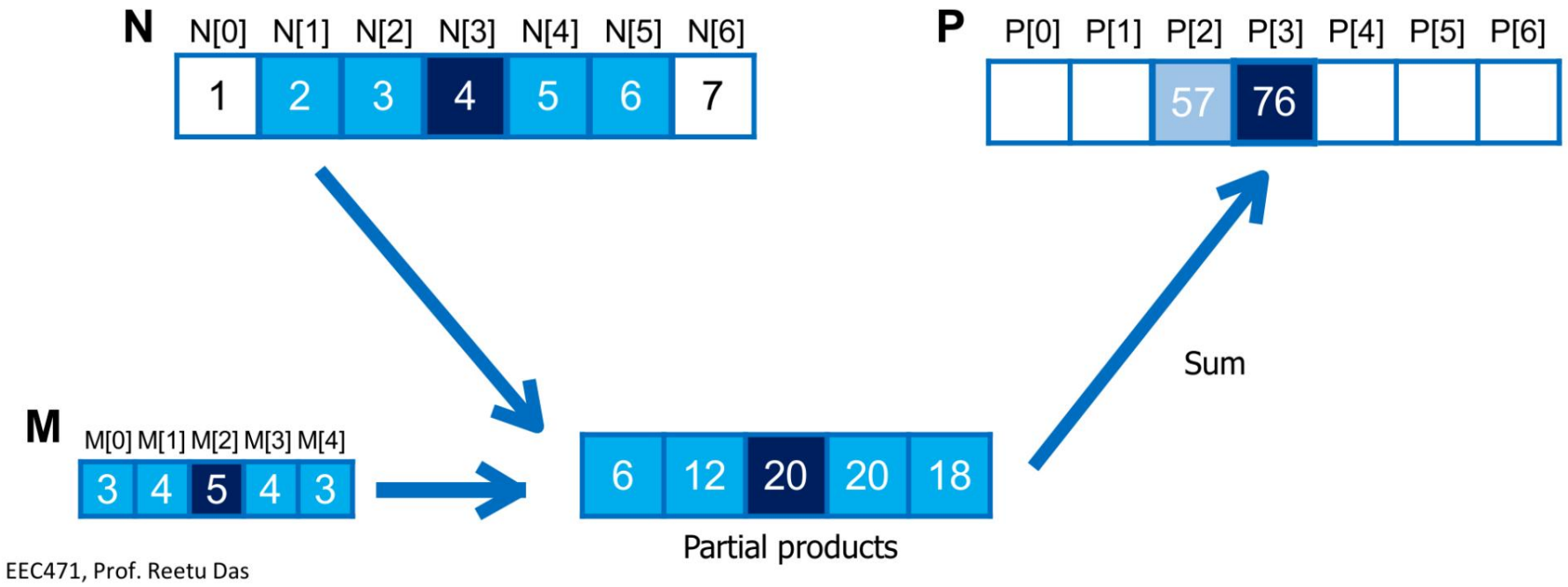


EEC471, Prof. Reetu Das

1D Convolution Example:Next Element

Calculation of P[3]

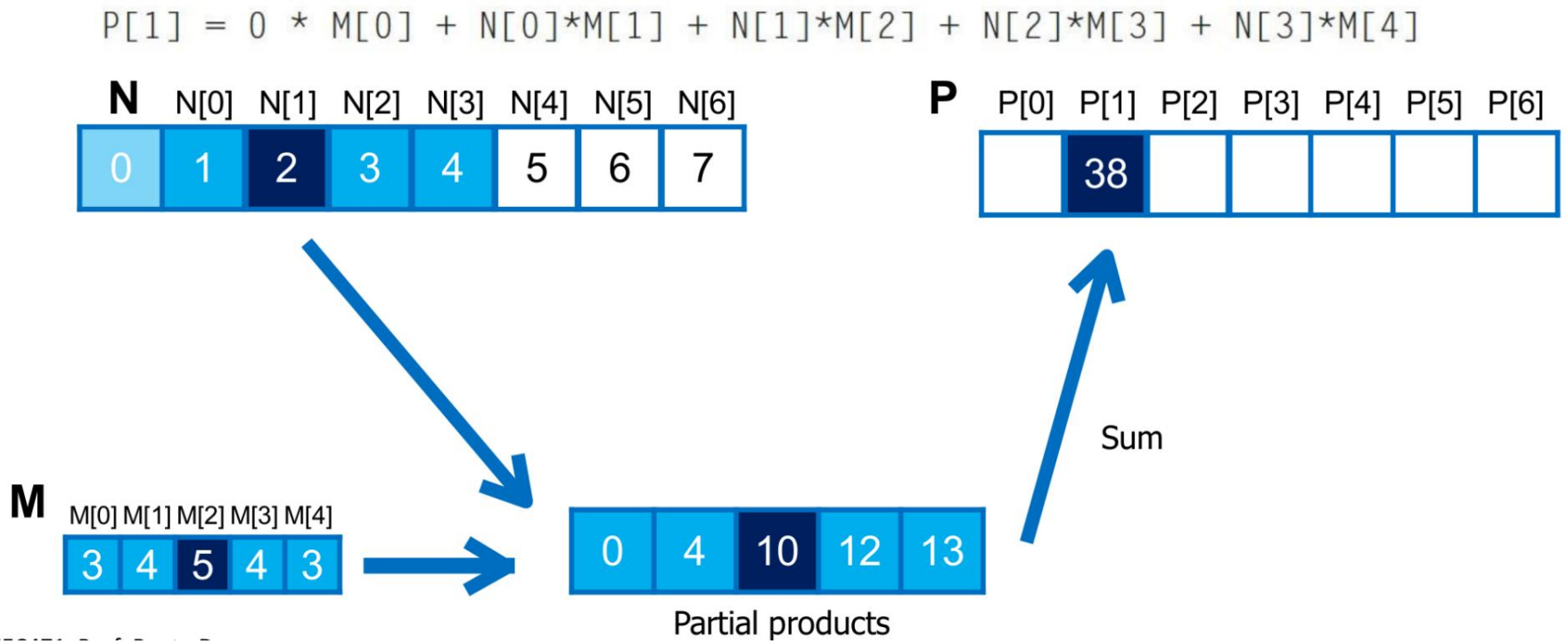
```
P[3] = N[1]*M[0] + N[2]*M[1] + N[3]*M[2] + N[4]*M[3] + N[5]*M[4]
```



EEC471, Prof. Reetu Das

1D Convolution Boundary Condition

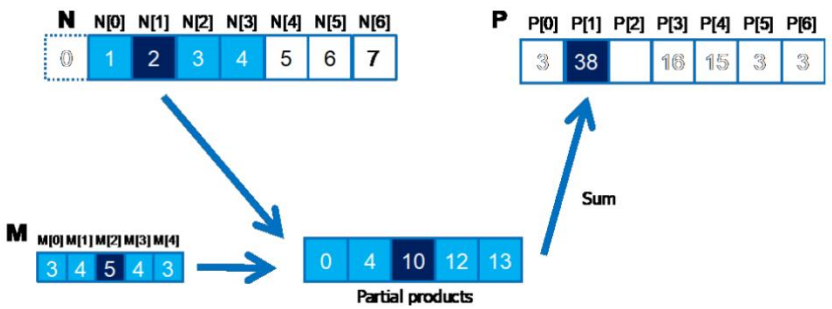
Calculation of output elements near the boundaries (beginning and end)of the input array need to deal with "ghost"elements



1D Convolution Kernel

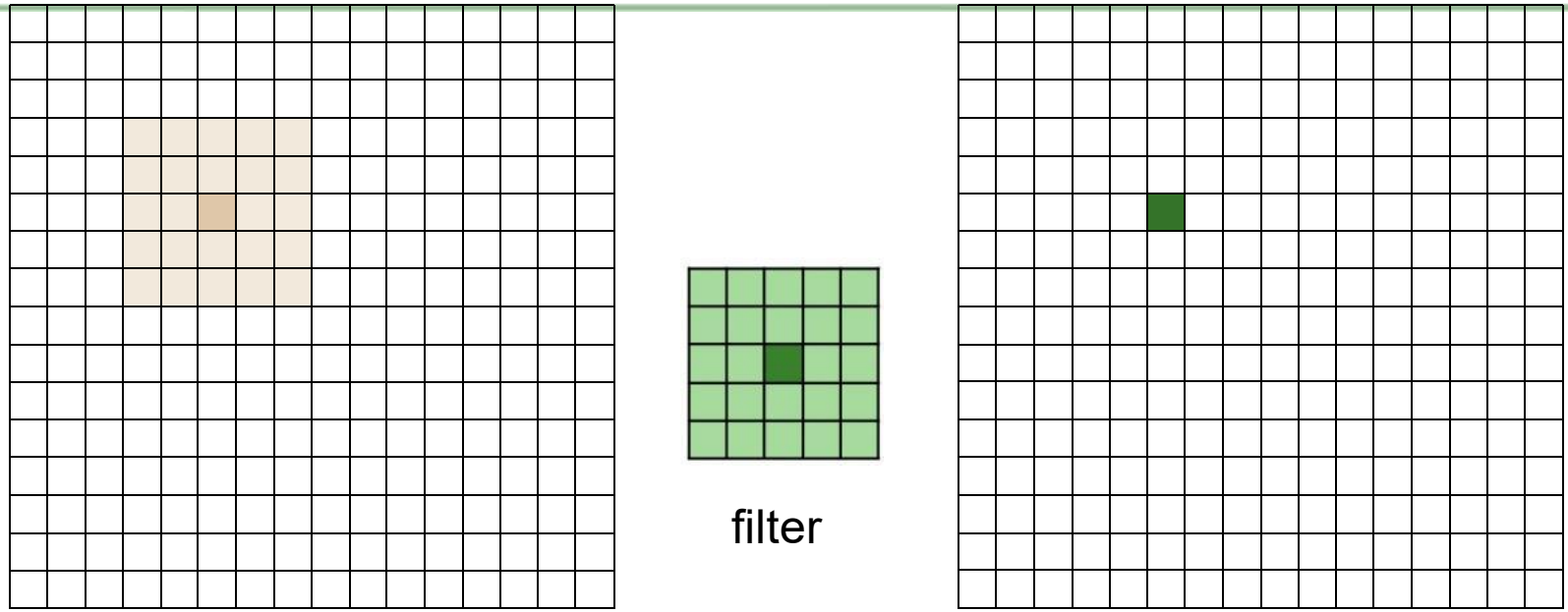
with Boundary Condition Handling*

```
__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; // Index of output element

  float Pvalue = 0;

  int N_start_point = i - (Mask_Width/2); // Index of first neighbor

  for(int j = 0; j < Mask_Width; j++) {

    // Check the boundaries
    if(N_start_point + j >= 0 && N_start_point + j < Width) {

      Pvalue += N[N_start_point + j] * M[j]; // Multiply and accumulate
    }
  }

  P[i] = Pvalue; // Store output element
}
```



*All elements outside the image are O in this ker6eE P590,Dr.Hari Sadasivan

2D Convolution



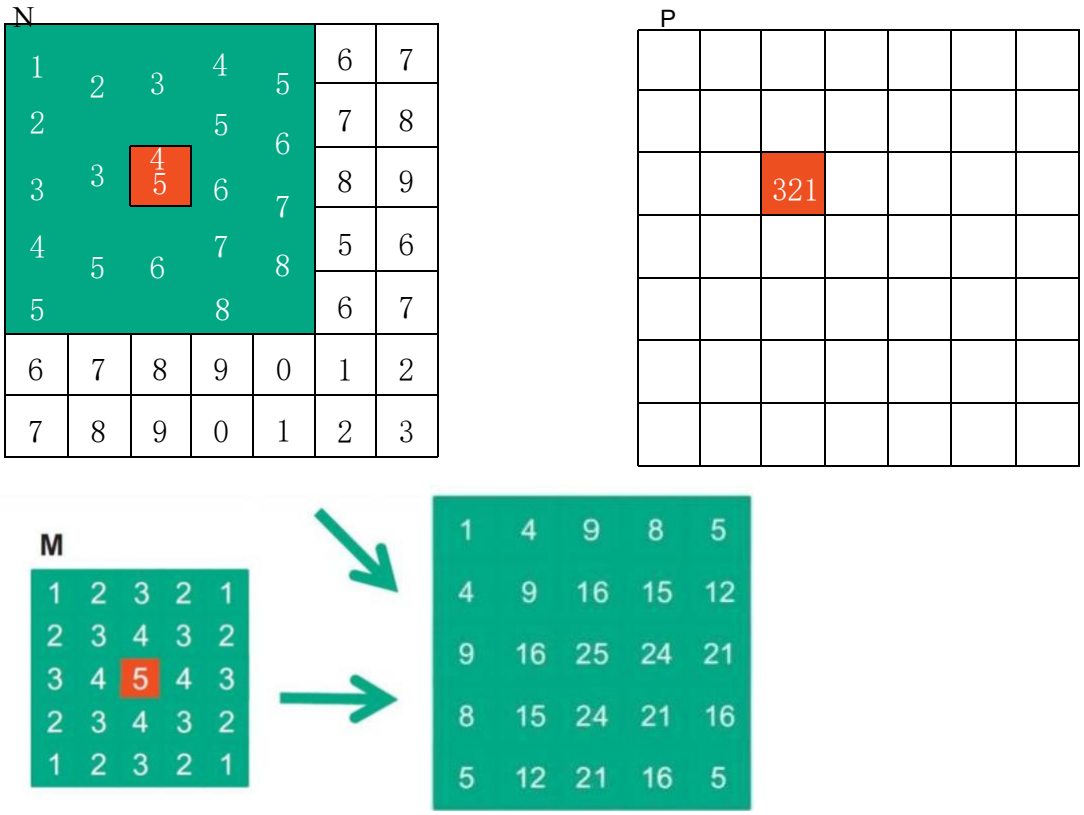
- Every output element is a weighted sum of the neighboring **input elements**

·Image blur seen before was a special case where all weights are the same

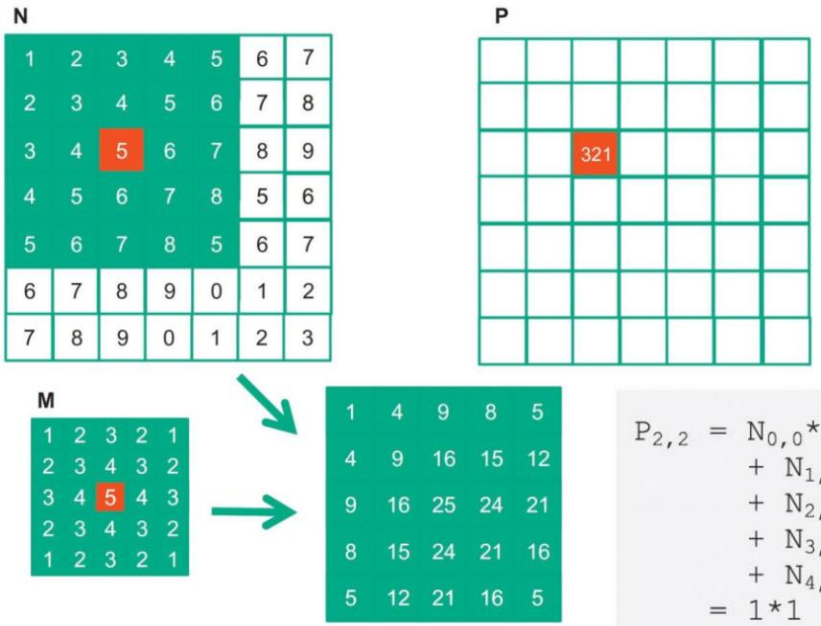
·In general,weights are determined by a convolution **filter**

(commonly called convolution kernel,but we will use filter to avoid confusion with CUDA kernels)

2D Convolution

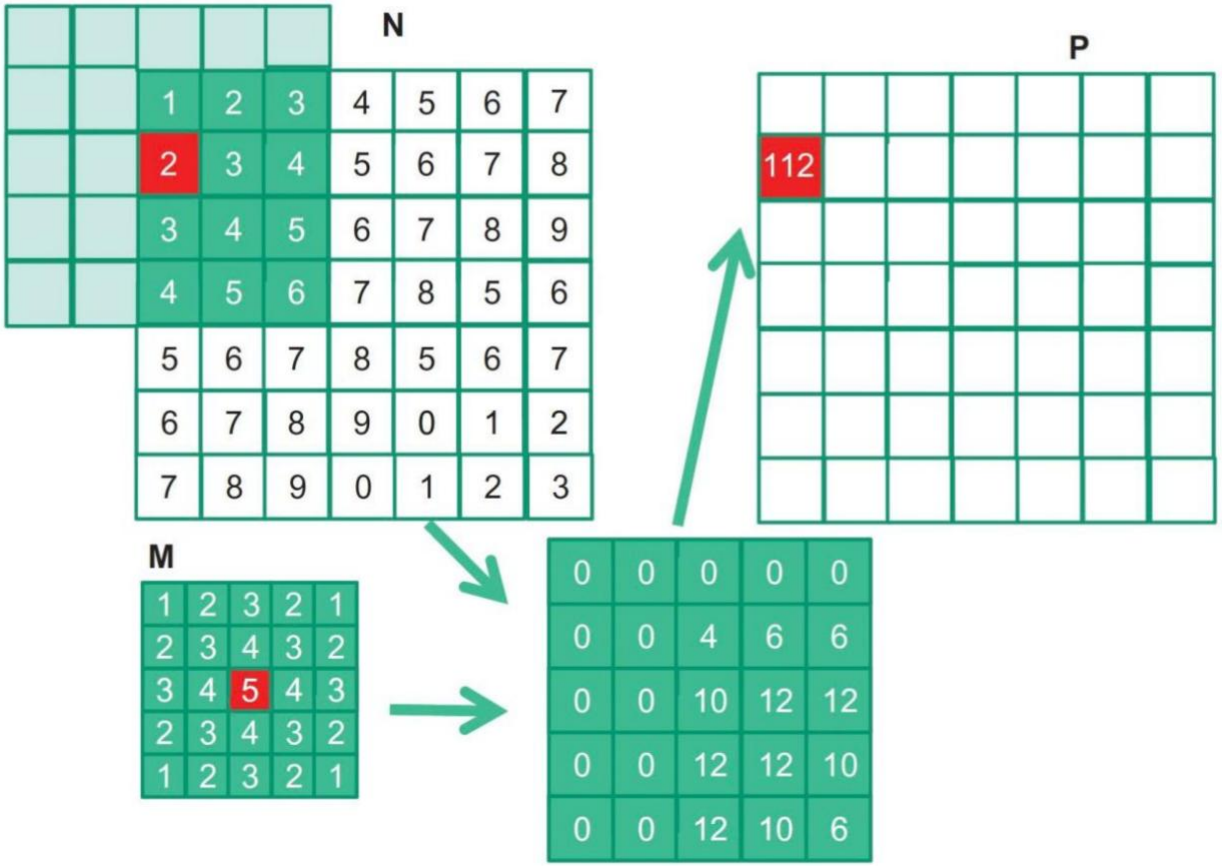


2D Convolution

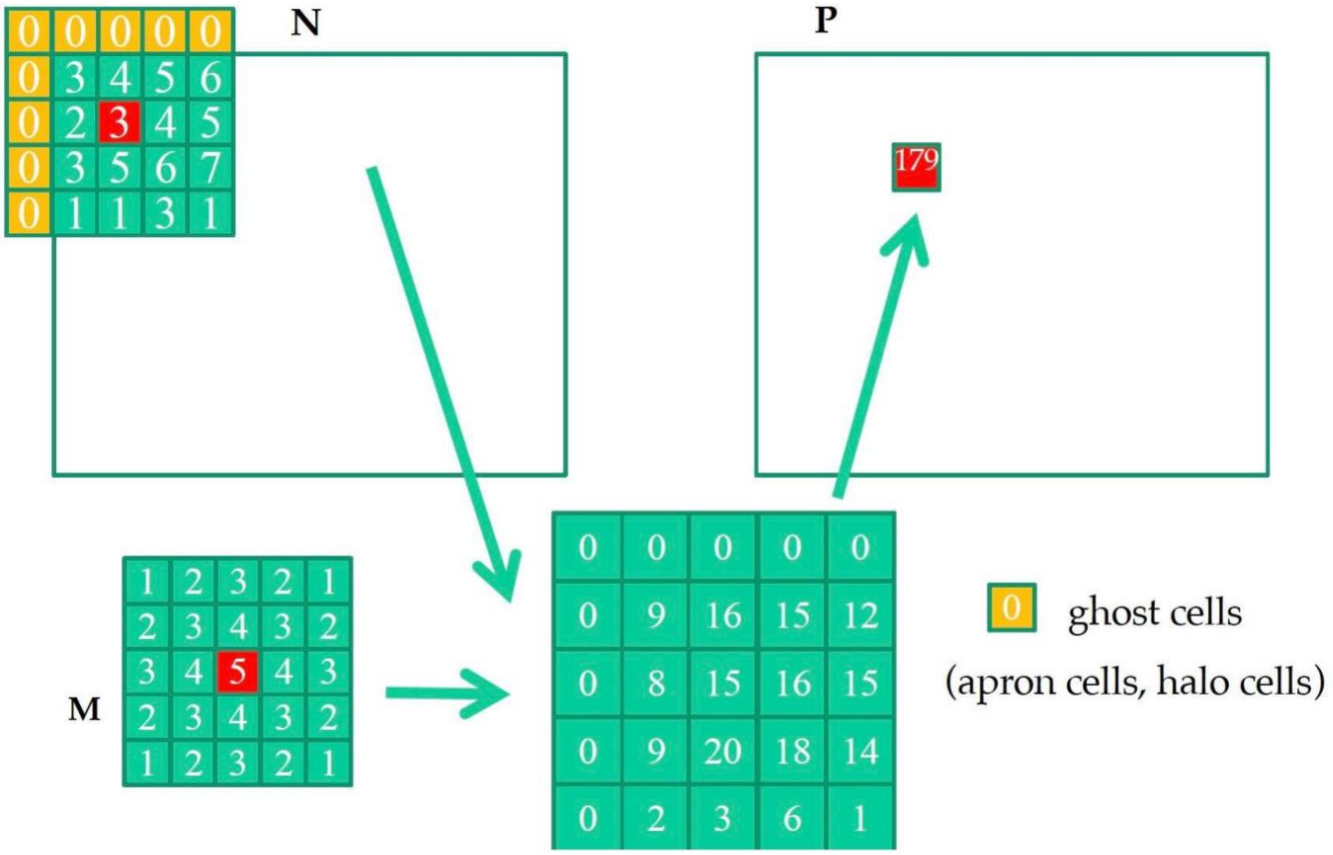


$$
\begin{aligned}
P_{2,2} &= N_{0,0}*M_{0,0} + N_{0,1}*M_{0,1} + N_{0,2}*M_{0,2} + N_{0,3}*M_{0,3} + N_{0,4}*M_{0,4} \\
&\quad + N_{1,0}*M_{1,0} + N_{1,1}*M_{1,1} + N_{1,2}*M_{1,2} + N_{1,3}*M_{1,3} + N_{1,4}*M_{1,4} \\
&\quad + N_{2,0}*M_{2,0} + N_{2,1}*M_{1,1} + N_{2,2}*M_{2,2} + N_{2,3}*M_{2,3} + N_{2,4}*M_{2,4} \\
&\quad + N_{3,0}*M_{3,0} + N_{3,1}*M_{3,1} + N_{3,2}*M_{3,2} + N_{3,3}*M_{3,3} + N_{3,4}*M_{3,4} \\
&\quad + N_{4,0}*M_{4,0} + N_{4,1}*M_{4,1} + N_{4,2}*M_{4,2} + N_{4,3}*M_{4,3} + N_{4,4}*M_{4,4} \\
&= 1*1 + 2*2 + 3*3 + 4*2 + 5*1 \\
&\quad + 2*2 + 3*3 + 4*4 + 5*3 + 6*2 \\
&\quad + 3*3 + 4*4 + 5*5 + 6*4 + 7*3 \\
&\quad + 4*2 + 5*3 + 6*4 + 7*3 + 8*2 \\
&\quad + 5*1 + 6*2 + 7*3 + 8*2 + 5*1 \\
&= 1 + 4 + 9 + 8 + 5 \\
&\quad + 4 + 9 + 16 + 15 + 12 \\
&\quad + 9 + 16 + 25 + 24 + 21 \\
&\quad + 8 + 15 + 24 + 21 + 16 \\
&\quad + 5 + 12 + 21 + 16 + 5 \\
&= 321
\end{aligned}
$$

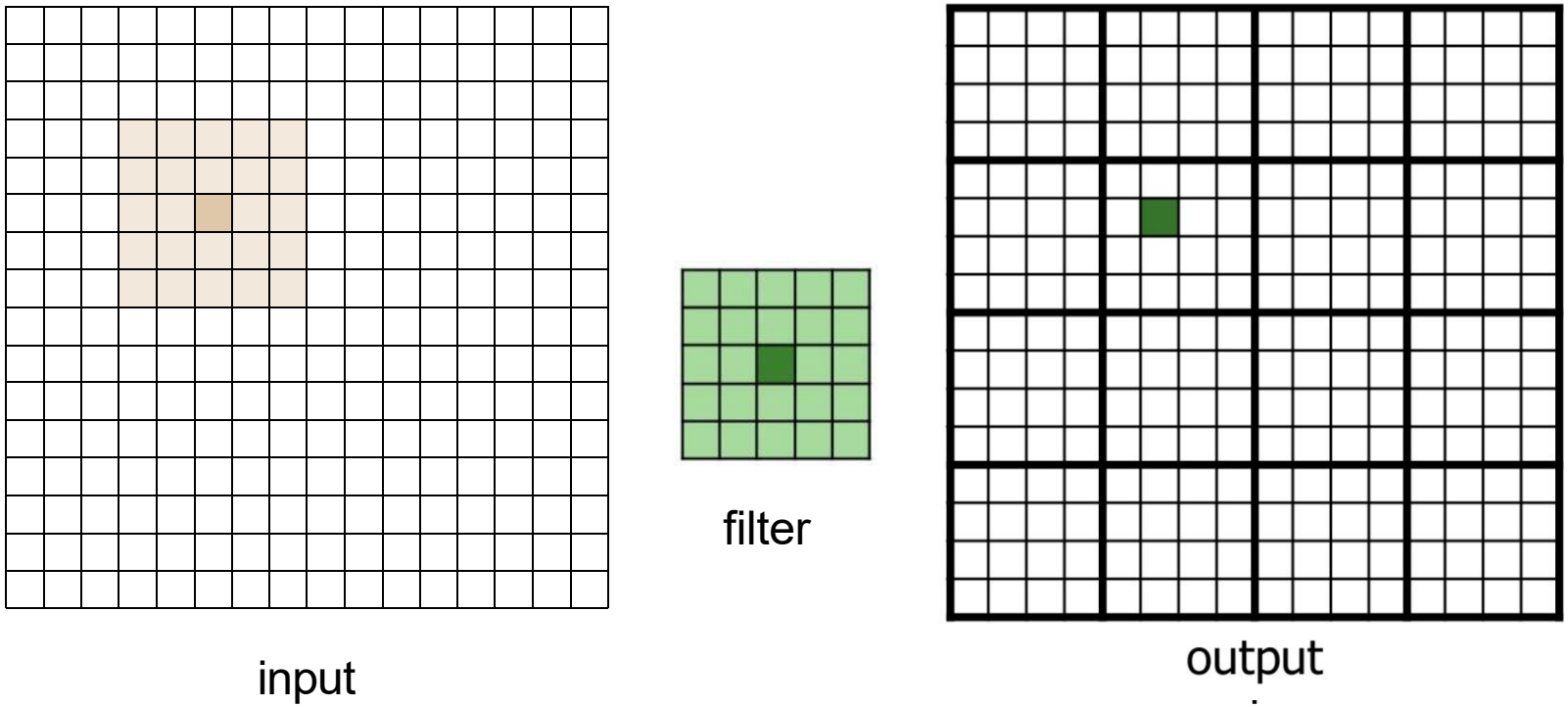
2D Convolution Boundary Condition



2D Convolution -Ghost Cells



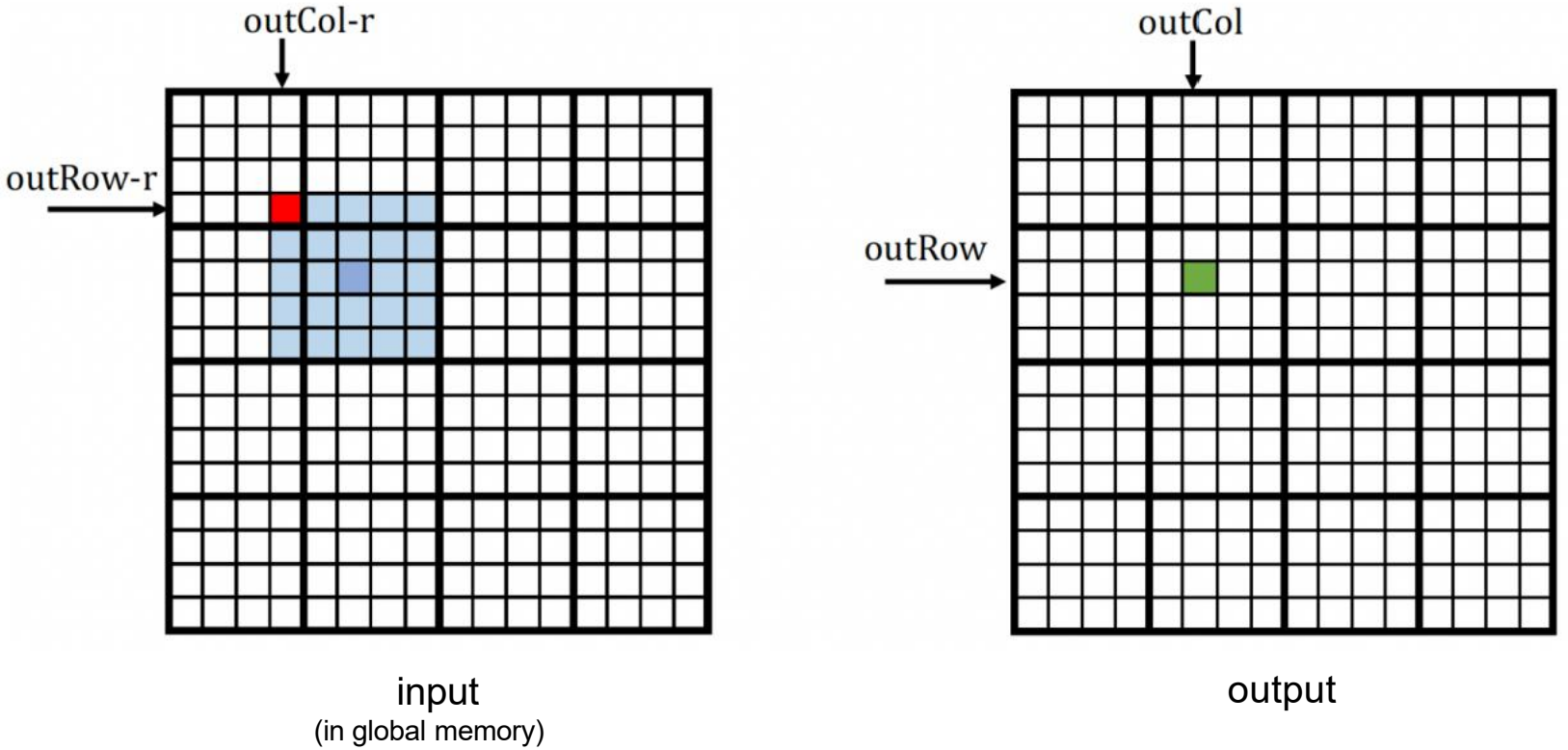
2D Convolution-Parallel Approach



Parallelization approach: Assign one thread to compute each output element by looping over input elements and filter weights

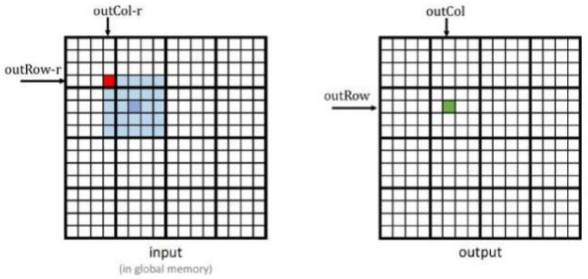
2D Convolution-Parallel Approach

Input area may be different from the output area



Parallelization approach: Assign one thread to compute each **output element** by looping over input elements and filter weights

2D Convolution -Basic Kernel



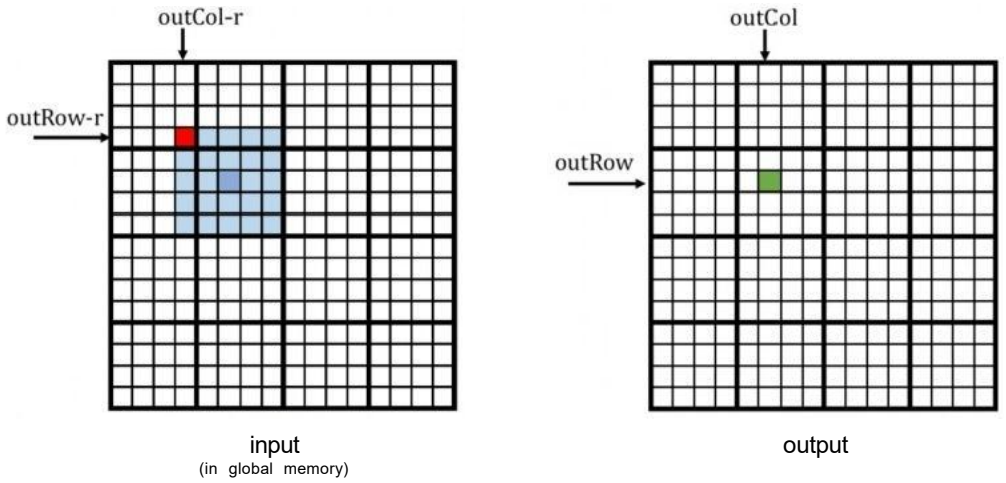

```
01  __global__ void convolution_2D_basic_kernel(float *N, float *F, float *P,
      int r, int width, int height) {
02    int outCol = blockIdx.x*blockDim.x + threadIdx.x;
03    int outRow = blockIdx.y*blockDim.y + threadIdx.y;
04    float Pvalue = 0.0f;
05    for (int fRow = 0; fRow < 2*r+1; fRow++) {
06      for (int fCol = 0; fCol < 2*r+1; fCol++) {
07          inRow = outRow - r + fRow;
08          inCol = outCol - r + fCol;
09          if (inRow >= 0 && inRow < height && inCol >= 0 && inCol < width) {
10             Pvalue += F[fRow][fCol]*N[inRow*width + inCol];
11          }
12        }
13    }
14    P[outRow][outCol] = Pvalue;
15  }
```

Parallelization approach: Assign one thread to compute each output element by looping over input elements and filter weights

2D Convolution -Basic Kernel

Memory Access: The ratio of floating-point The arithmetic calculation to global memory accesses is only about 0.25 OP/B(2 operations for every 8 bytes loaded on line 10)

```
08          inCol = outCol - r + fCol;
09          if (inRow >= 0 && inRow < height && inCol >= 0 && inCol < width) {
10             Pvalue += F[fRow][fCol]*N[inRow*width + inCol];
11          }
```



Parallelization approach: Assign one thread to compute each output element by looping over input elements and filter weights

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 need M

M is not changed during kernel

M elements are accessed in the same order when calculating all P elements

M is a good candidate for Constant Memory

Programmer View of CUDA Memories

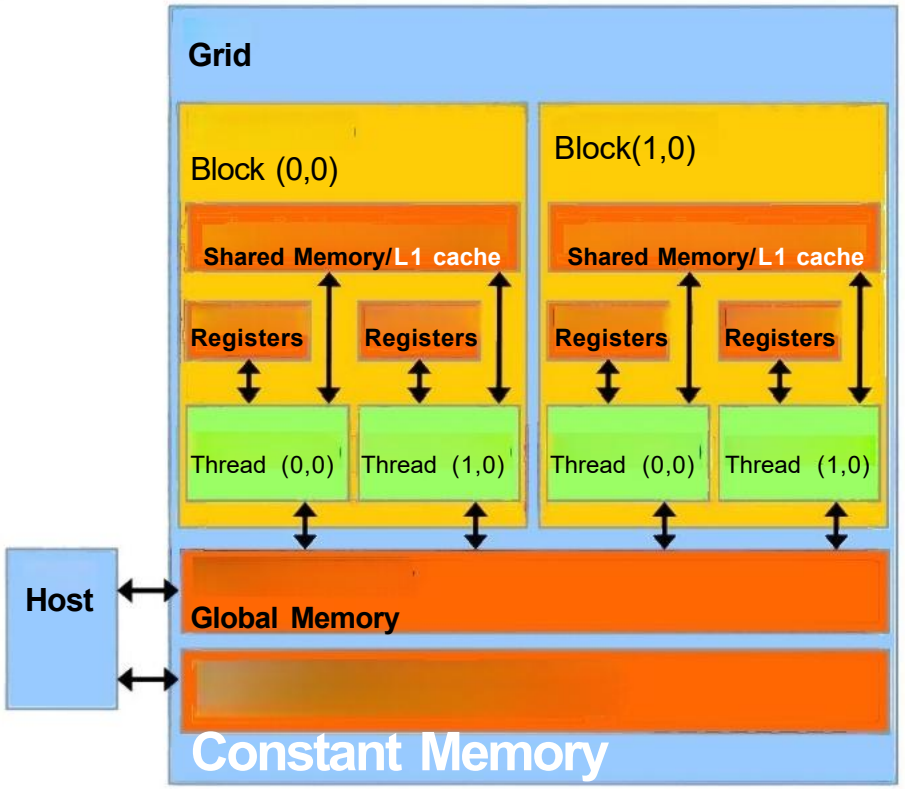
·Each thread can:

-Read/write per-thread registers (~1 cycle)

-Read/write per-block shared memory (~60 cycles)

-Read/write per-grid global memory (~1000 cycles)

-Read-only per-grid constant memory (~50 cycles with caching)



Constant Memory in GPUs

Modification to cached data needs to be(eventually) reflected to the original data in global memory

-Requires logic to track the modified status,etc.

Constant memory 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

-Constant cache can be accessed with higher throughput than L1 cache for some common patterns

How to Use Constant Memory

Host code allocates,and initializes variables the same way as any other variables that need to be copied to the device

```
#define MAX_MASK_WIDTH 10

__constant__float   M[MAX_MASK_WIDTH];      (1D)
__constant__float   M[MAX_MASK_WIDTH][MAX_MASK_WIDTH];      (2D)
```

Use cudaMemcpyToSymbol(dest,src,size) to copy the variable into the device memory

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

1D Convolution Kernel with Constant Memory

The kernel is the same as one w/o constant caching

```
__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 with Constant Memory

The host code is different

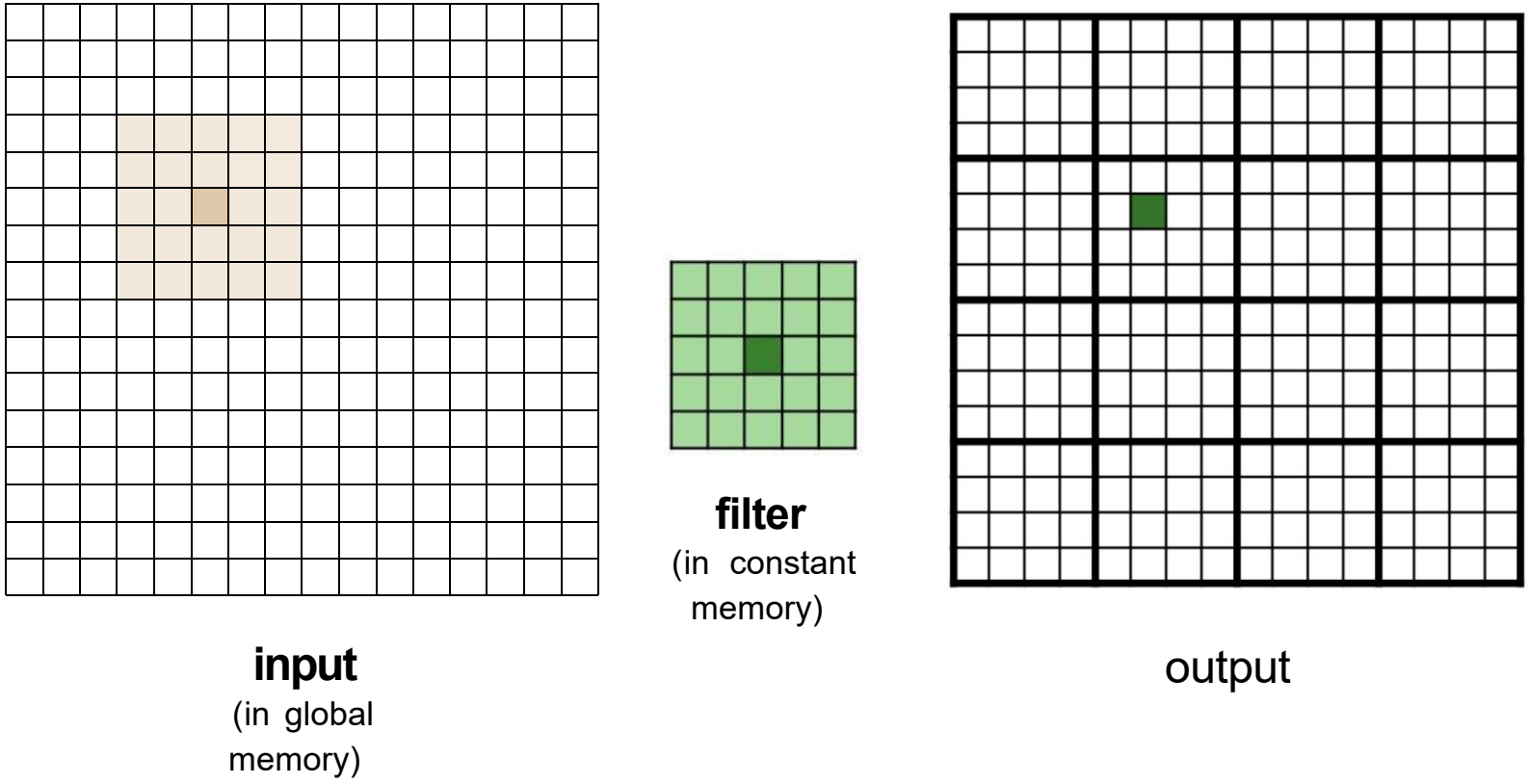
```
// global variable, outside any kernel/function
#define MAX_MASK_WIDTH 10
__constant__ float Mc[MAX_MASK_WIDTH];

// allocate N, P, initialize N elements, copy N to Nd, initialize Mask

....
 // copy Mask to constant memory variable Mc
cudaMemcpyToSymbol(Mc, Mask,
            MASK_WIDTH*sizeof(float));

//kernel launch
convolution_1D_basic_kernel <<<dimGrid, dimBlock>>>(Nd, Mc, Pd, MASK_WIDTH, width);
```

Compute to Memory **Ratio**

- How many global memory accesses?
- How many constant memory access?
  - -Latency of constant memory access?
- Compute to memory ratio
  - –2
- Kernel with no constant memory access -Compute to memory ratio?
  - –1
- Doubled performance

```
__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;

}
```

Parallelizing Convolution



input
(in global memory)

filter
(in constant memory)

output

Parallelization anproach: Assign one thread to compute each output element by looping over input elements and filter weights

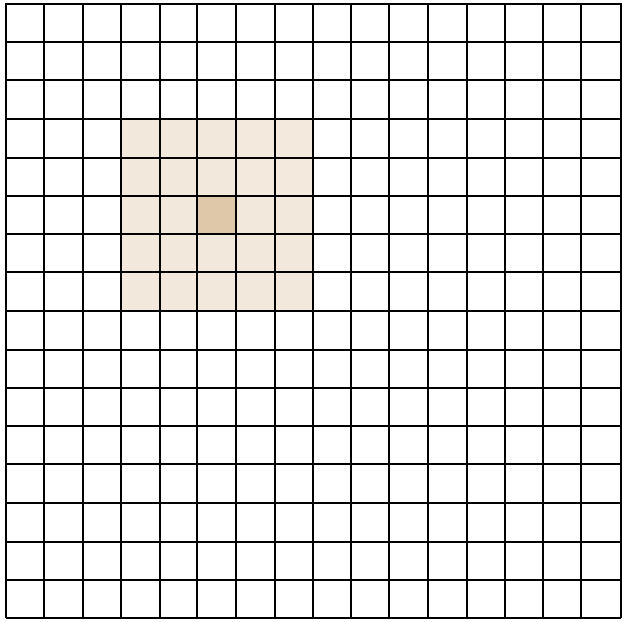
2D Convolution Kernel Code

```
01  __global__ void convolution_2D_basic_kernel(float *N, float *F, float *P,
      int r, int width, int height) {
02    int outCol = blockIdx.x*blockDim.x + threadIdx.x;
03    int outRow = blockIdx.y*blockDim.y + threadIdx.y;
04    float Pvalue = 0.0f;
05    for (int fRow = 0; fRow < 2*r+1; fRow++) {
06      for (int fCol = 0; fCol < 2*r+1; fCol++) {
07          inRow = outRow - r + fRow;
08          inCol = outCol - r + fCol;
09          if (inRow >= 0 && inRow < height && inCol >= 0 && inCol < width) {
10             Pvalue += F[fRow][fCol]*N[inRow*width + inCol];
11          }
12        }
13    }
14    P[outRow][outCol] = Pvalue;
15  }
```

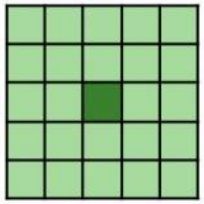
2D Convolution Kernel Code with Constant Memory

```
01 __global__ void convolution_2D_const_mem_kernel(float *N, float *P, int r,
       int width, int height) {
02    int outCol = blockIdx.x*blockDim.x + threadIdx.x;
03    int outRow = blockIdx.y*blockDim.y + threadIdx.y;
04    float Pvalue = 0.0f;
05    for (int fRow = 0; fRow < 2*r+1; fRow++) {
06      for (int fCol = 0; fCol < 2*r+1; fCol++) {
07          inRow = outRow - r + fRow;
08          inCol = outCol - r + fCol;
09          if (inRow >= 0 && inRow < height && inCol >= 0 && inCol < width) {
10             Pvalue += F[fRow][fCol]*N[inRow*width + inCol];
11          }
12        }
13    }
14    P[outRow*width+outCol] = Pvalue;
15 }
```

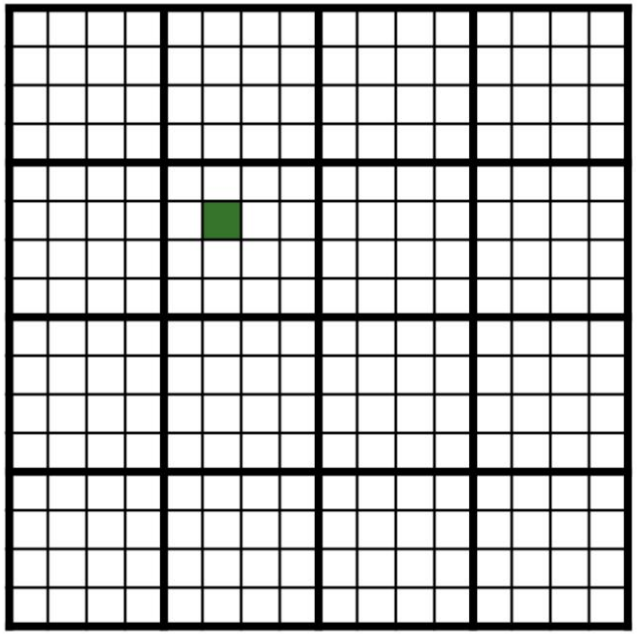
Data Reuse in Convolution



input
(in global memory)



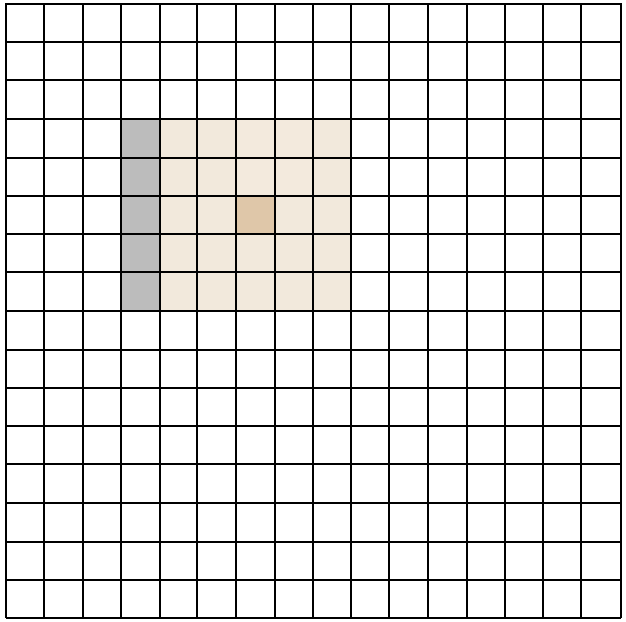
filter
(in constant memory)



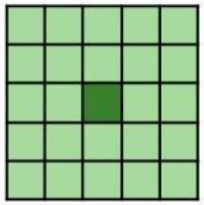
output

Observation: Threads in the same block load some of the same input elements

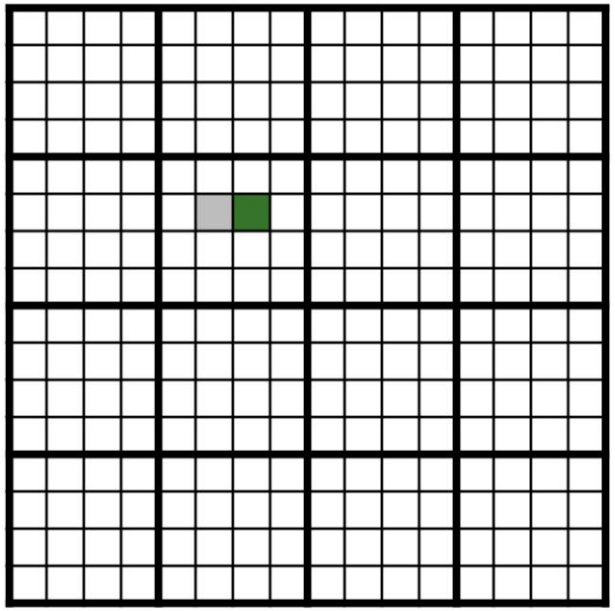
Data Reuse in Convolution



input
(in global memory)



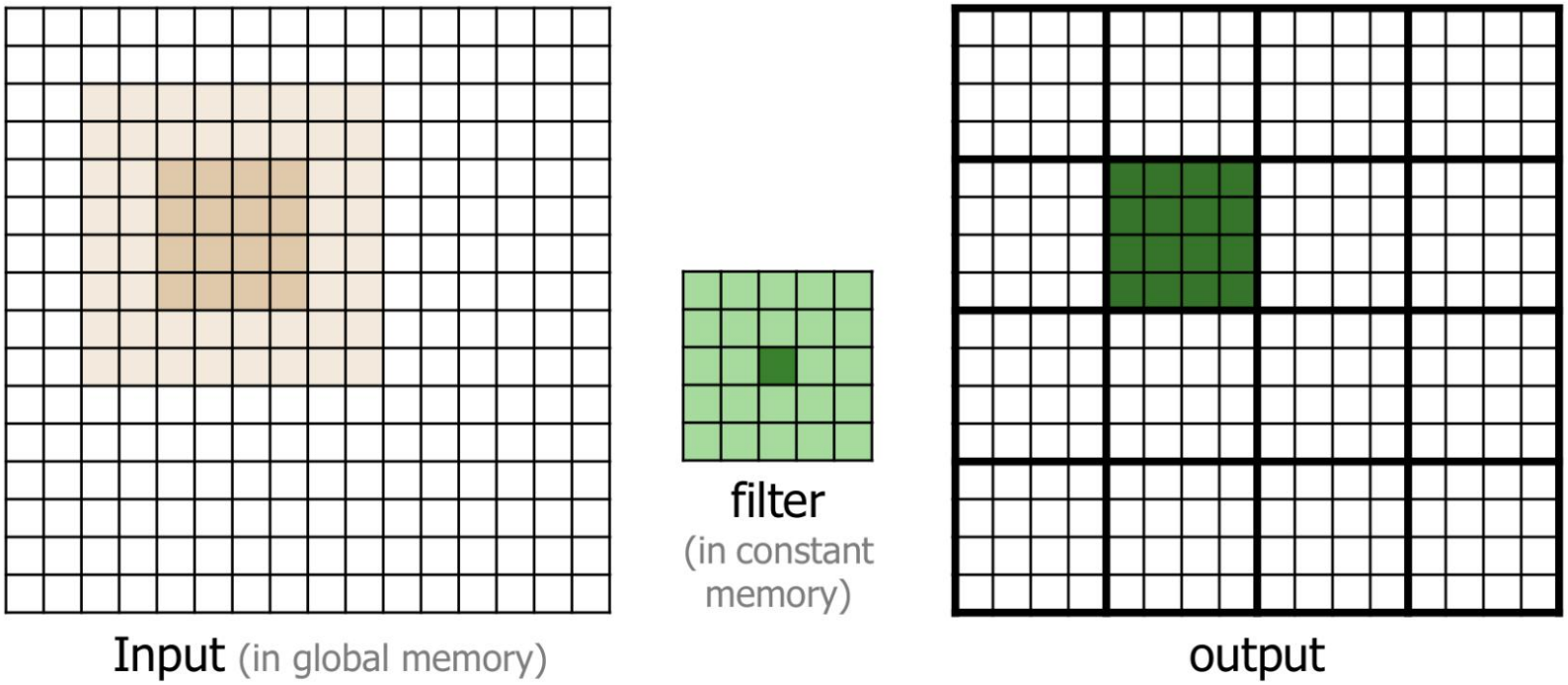
filter
(in constant memory)



output

Observation: Threads in the same block load some of the same input elements

Data Reuse in Convolution



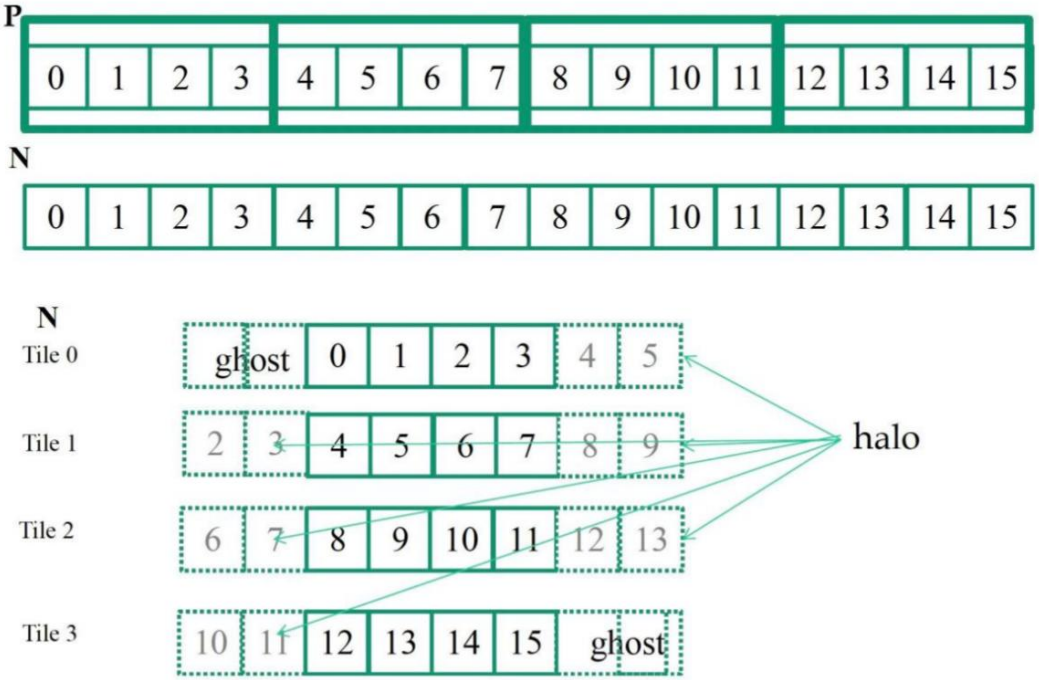
Qbservation: Threads in the same block load some of the same input elements

Qptimization: Each thread loads one input element to shared memory and other threads access the element from shared memory

Tiled Convolution

Tiled 1D Convolution

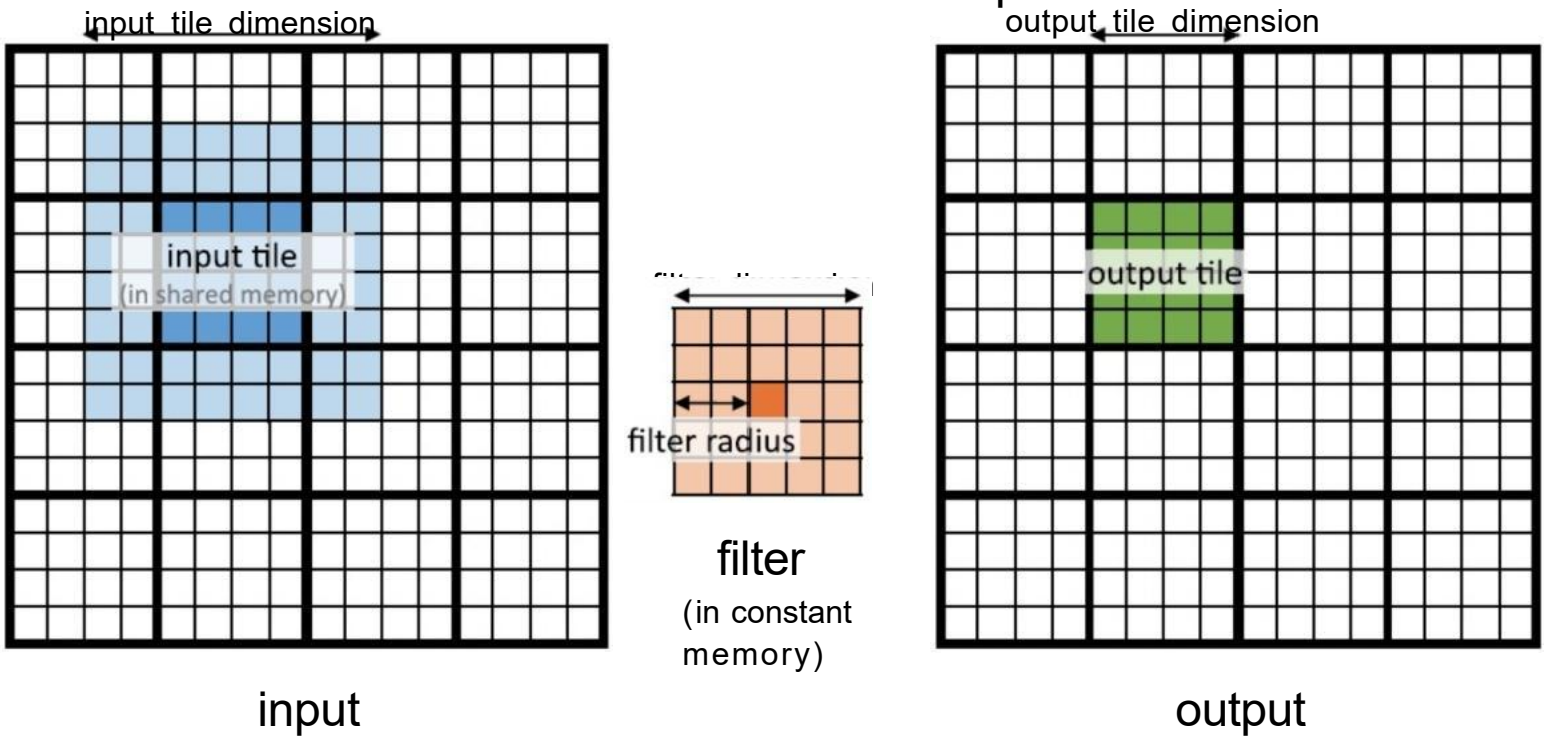
Memory bandwidth bottleneck: use tiled convolution algorithm.
Input and output tiles: Collection of output elements processed by each block called output tile.An input tile is the input N elements needed to ca' [illegible]



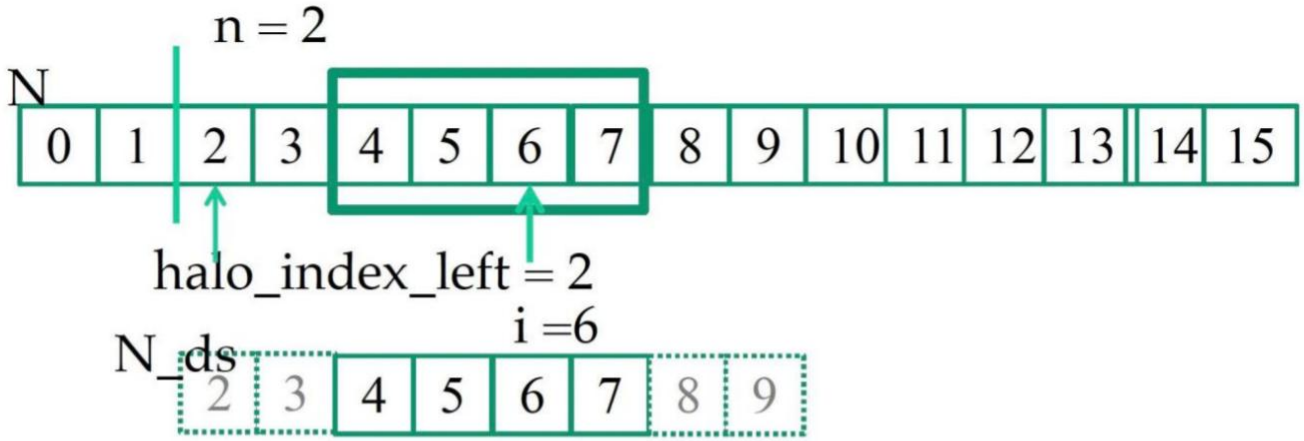
Tiled 2D Convolution

Memory bandwidth bottleneck: use tiled convolution algorithm.

Input and output tiles: Collection of output elements processed by each block called output tile. An input tile is the input N elements needed to calculate the P elements in an output tile.

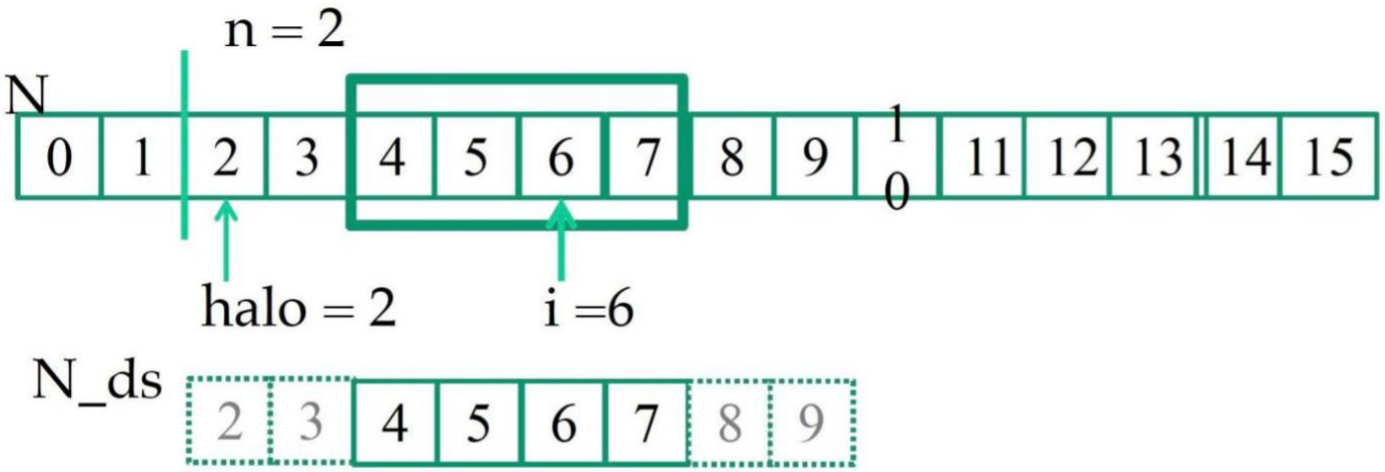


Loading the left halo



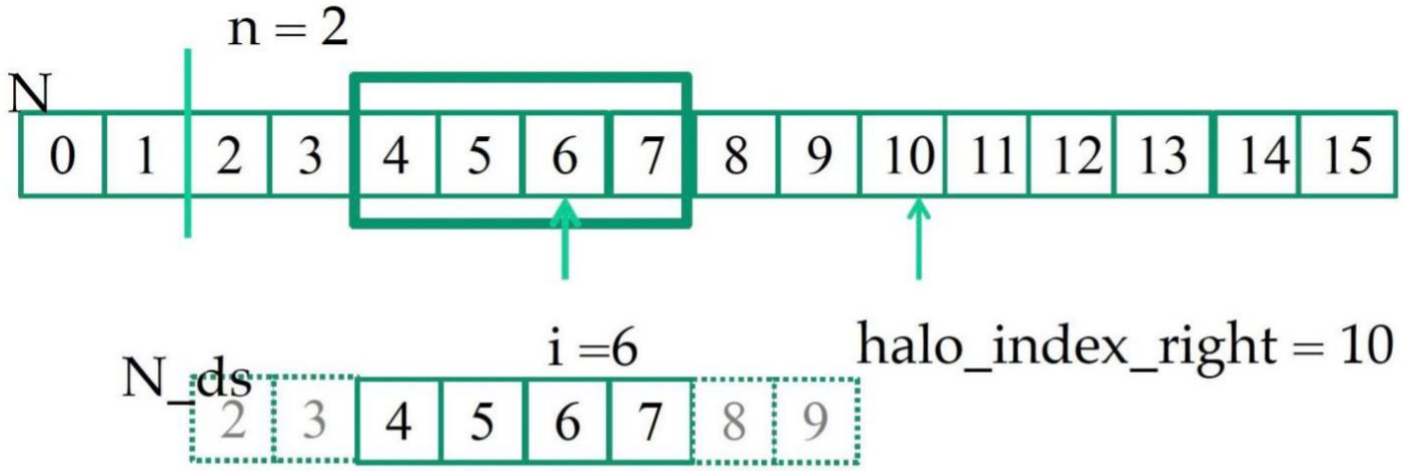

```
int n = Mask_Width/2;
int halo_index_left = (blockIdx.x - 1)*blockDim.x + threadIdx.x;
  if (threadIdx.x >= blockDim.x - n) {
    N_ds[threadIdx.x - (blockDim.x - n)] =
      (halo_index_left < 0) ? 0 : N[halo_index_left];
  }
```

Loading the internal elements



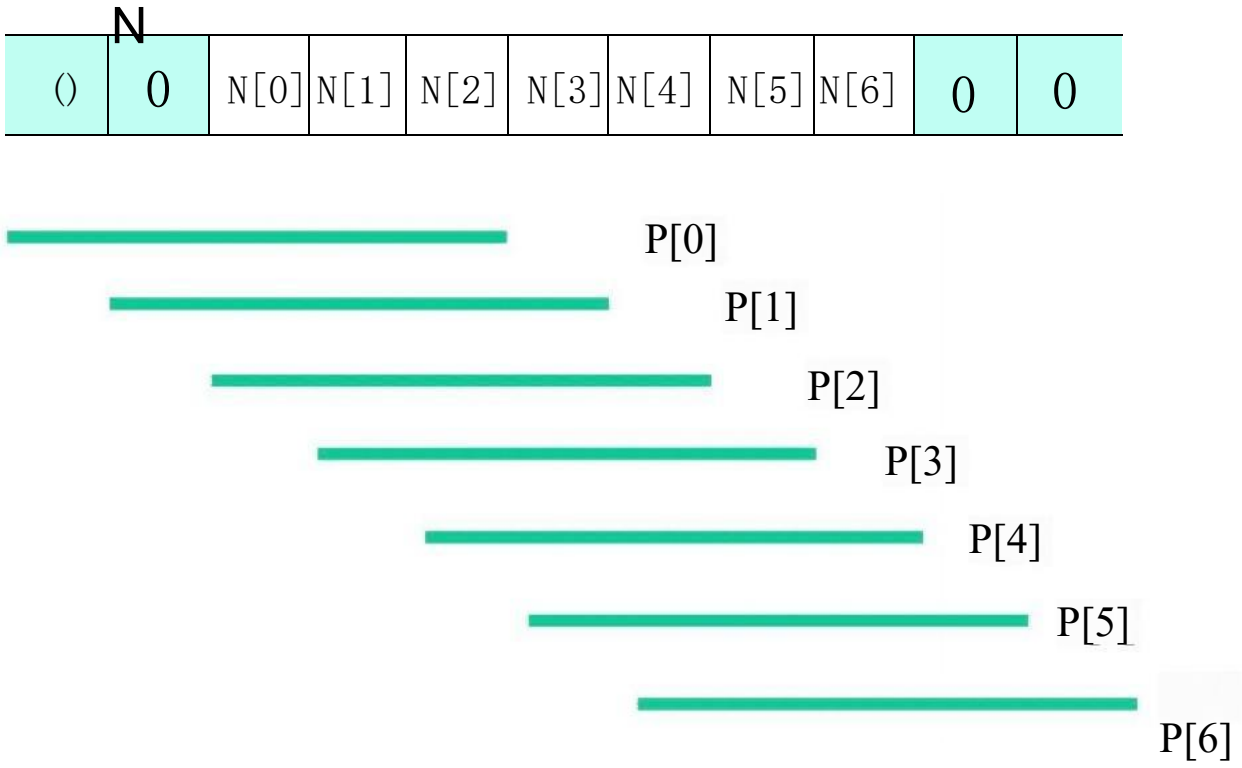

```
N ds[n + threadIdx.x] = N[blockIdx.x*blockDim.x + threadIdx.x];
```

Loading the right halo




```
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];
  }
```

Ghost Cells



1D Convolution Tiled 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;

}
```

1:

2:

3:

Shared Memory Data Reuse

- Element 2 is used by thread 4 (1X)
- Element 3 is used by threads 4, 5 (2X)
- Element 4 is used by threads 4, 5, 6 (3X)
- Element 5 is used by threads 4, 5, 6, 7 (4X)
- Element 6 is used by threads 4, 5, 6, 7 (4X)
- Element 7 is used by threads 5, 6, 7 (3X)
- Element 8 is used by threads 6, 7 (2X)
- Element 9 is used by thread 7 (1X)

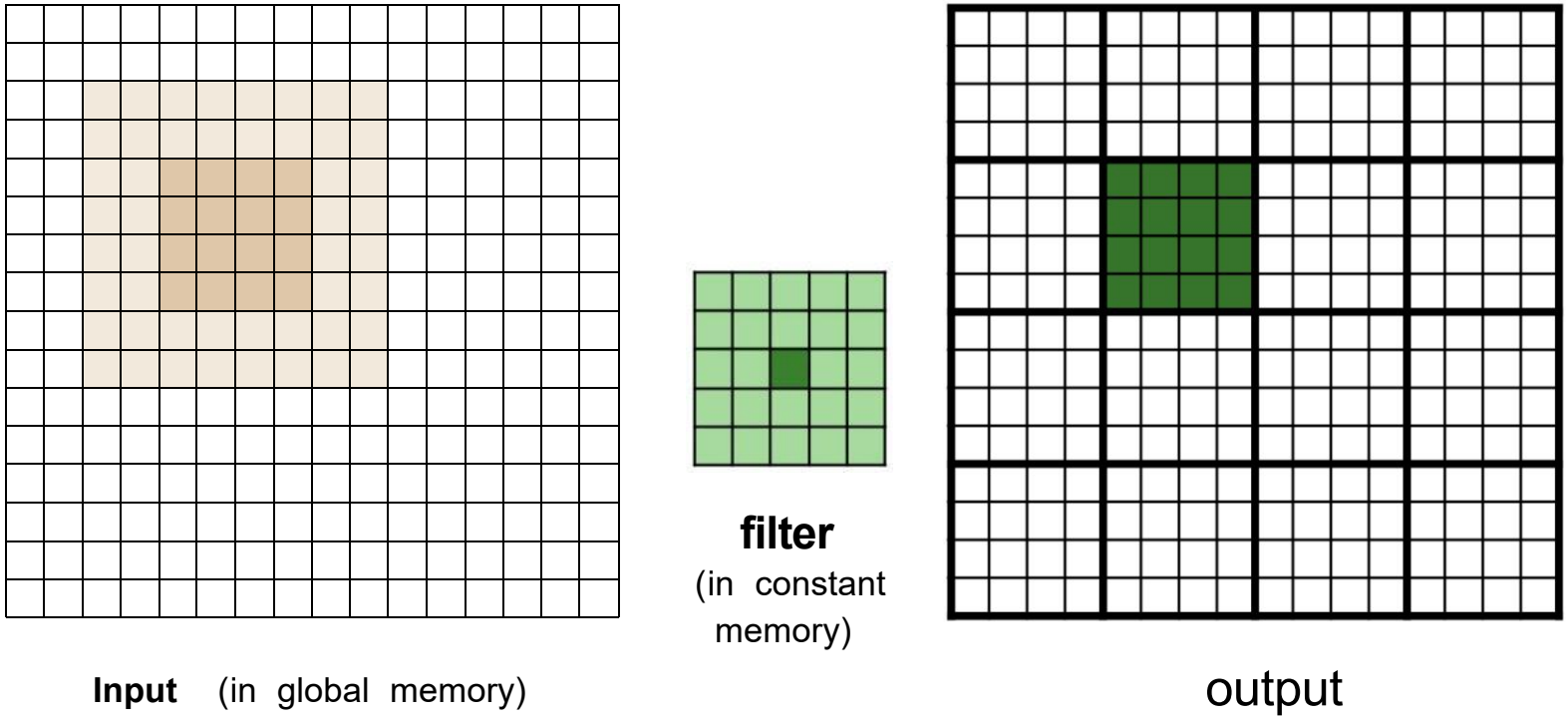
1D Convolution Tiled Kernel with Caching

The halo cells of an input tile of a block are also the internal elements of neighboring tiles

```
__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;

}
```

2D Tiled Convolution

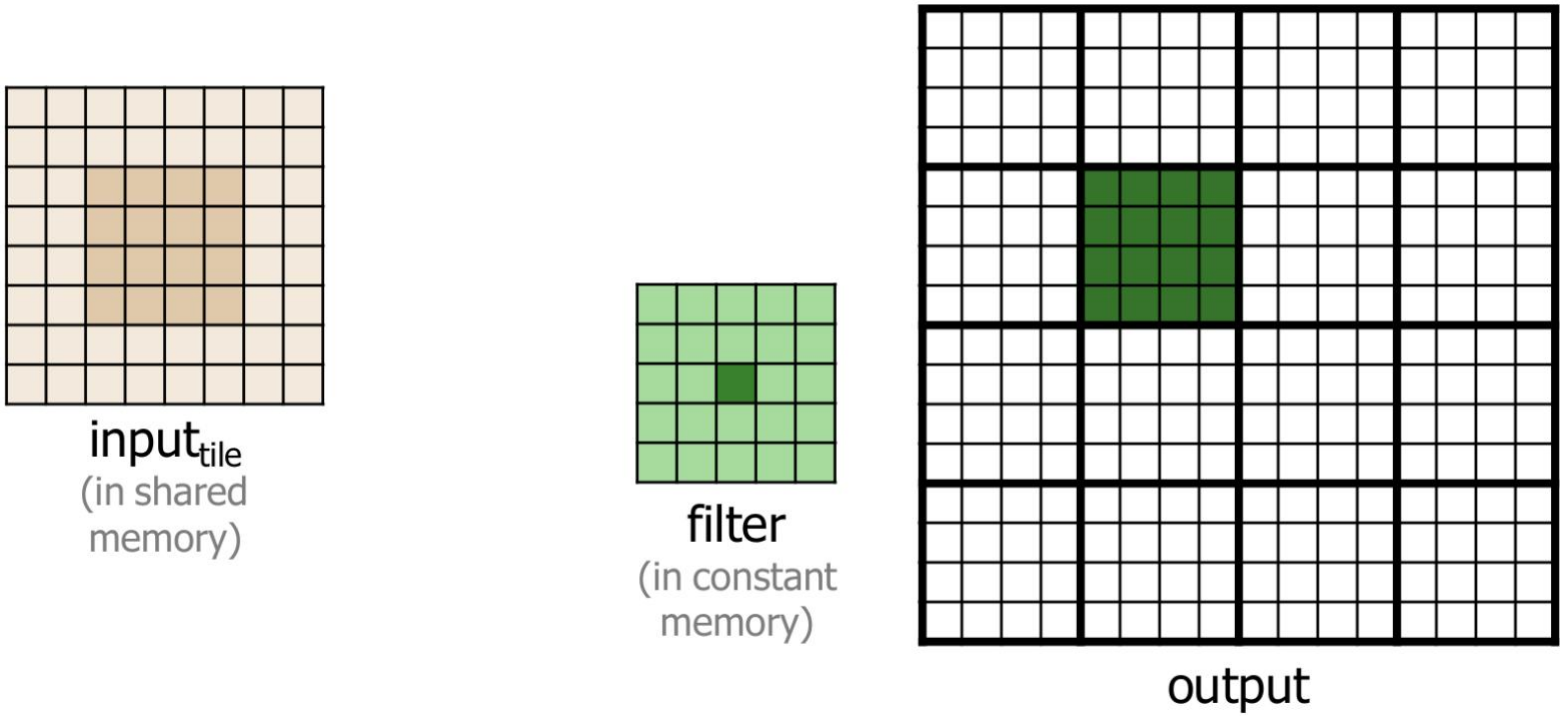
Data Reuse in Convolution



Observation: Threads in the same block load some of the same input elements

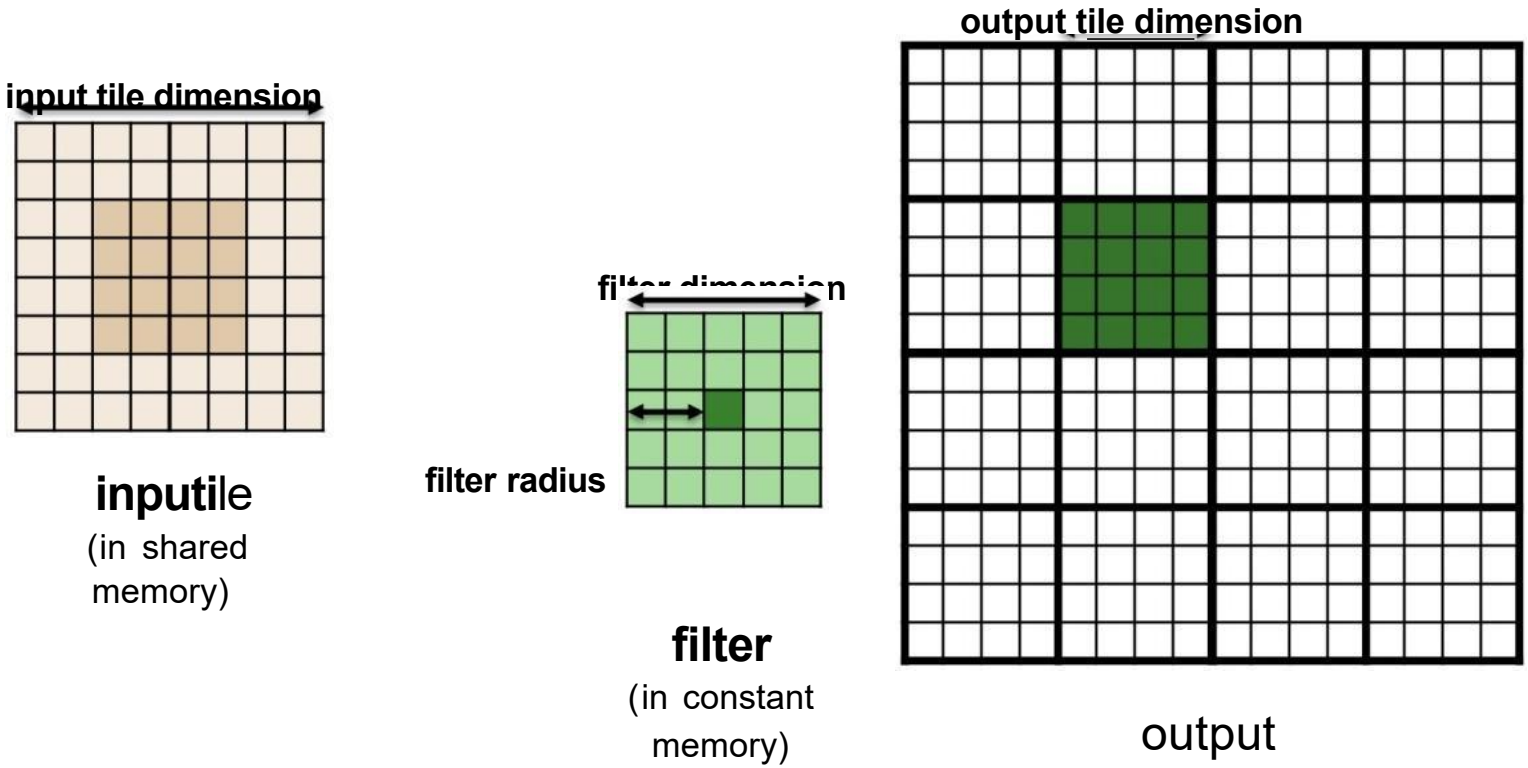
Optimization: Each thread loads one input element to shared memory and other threads access the element from shared memory

Convolution with Tiling



Qptimization: Each thread loads one input element to shared memory and other threads access the element from shared memory

Difference in Tile Sizes

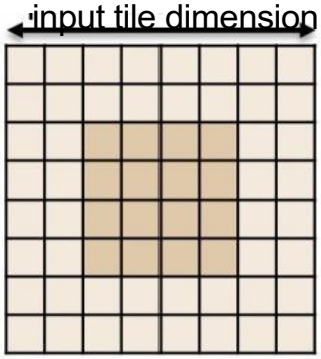


Challenge: Input and output tiles have different dimensions

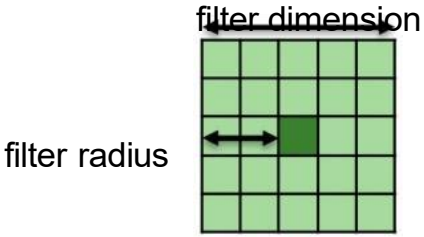
(input tile dimension =output tile dimension +2×filter radius)

Solution: Launch enough threads per block to load the input tile to shared memory, then use a subset of them to compute and store the output tile

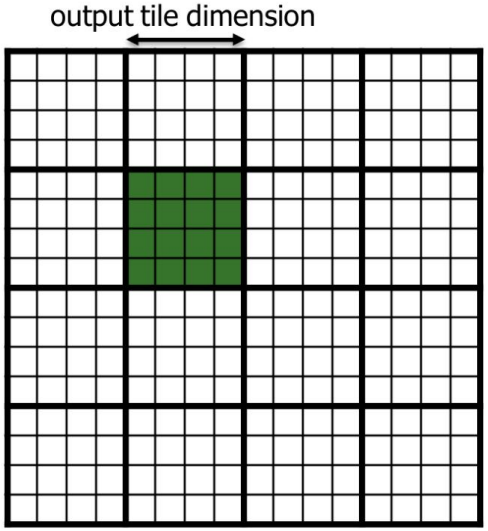
Difference in Tile Sizes



inputtile
(in shared
memory)

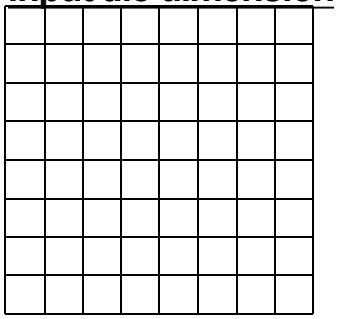


filter
(in constant
memory)



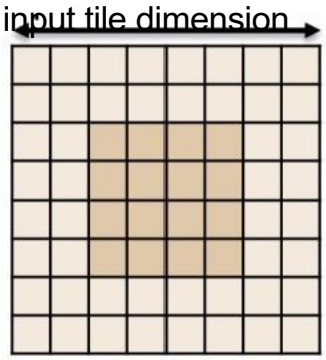
output

input tile dimension

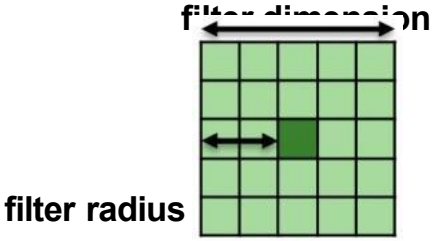


thread block

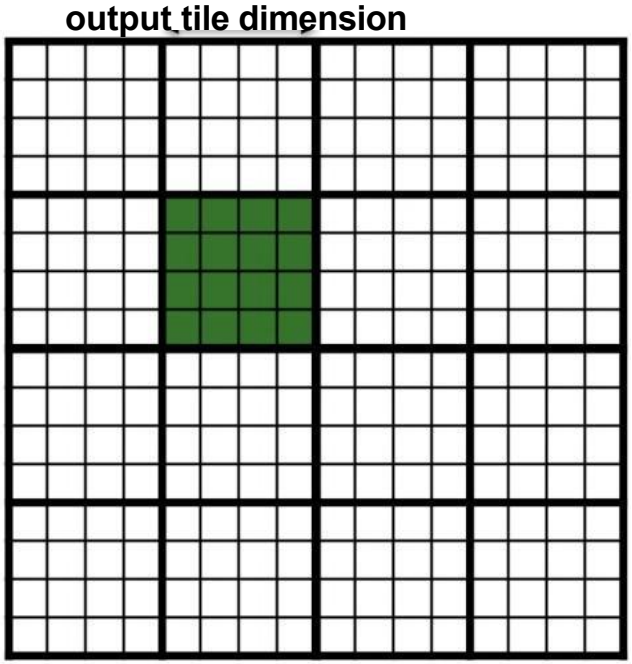
Difference in Tile Sizes



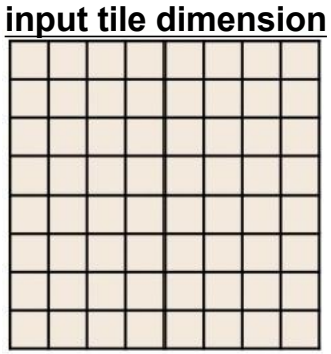
inputtile
(in shared memory)



filter (in constant memory)



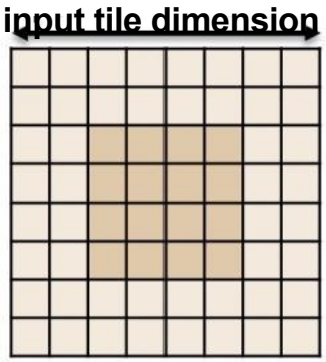
output



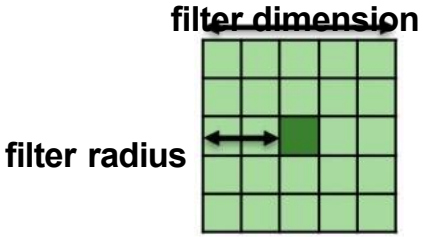
threads active when loading input tile

thread bleck

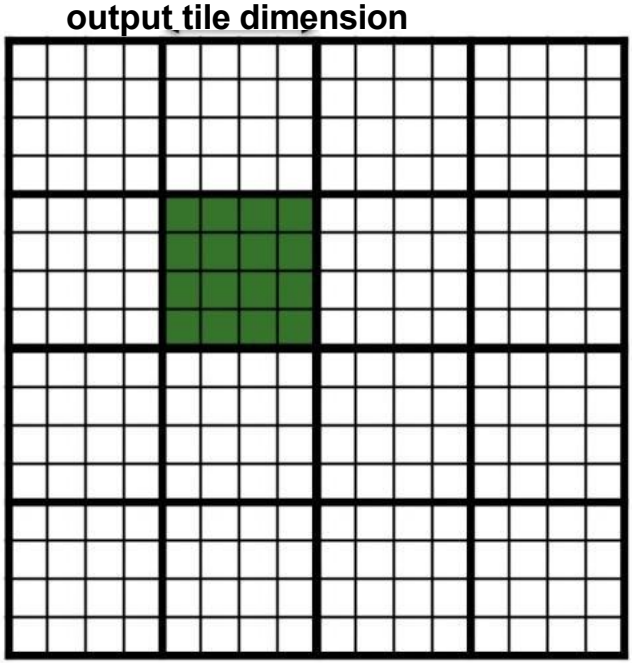
Difference in Tile Sizes



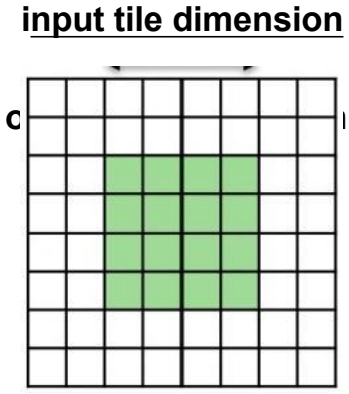
inputtile (in shared memory)



filter (in constant memory)



output



thread block

threads active when computing and storing the output tile

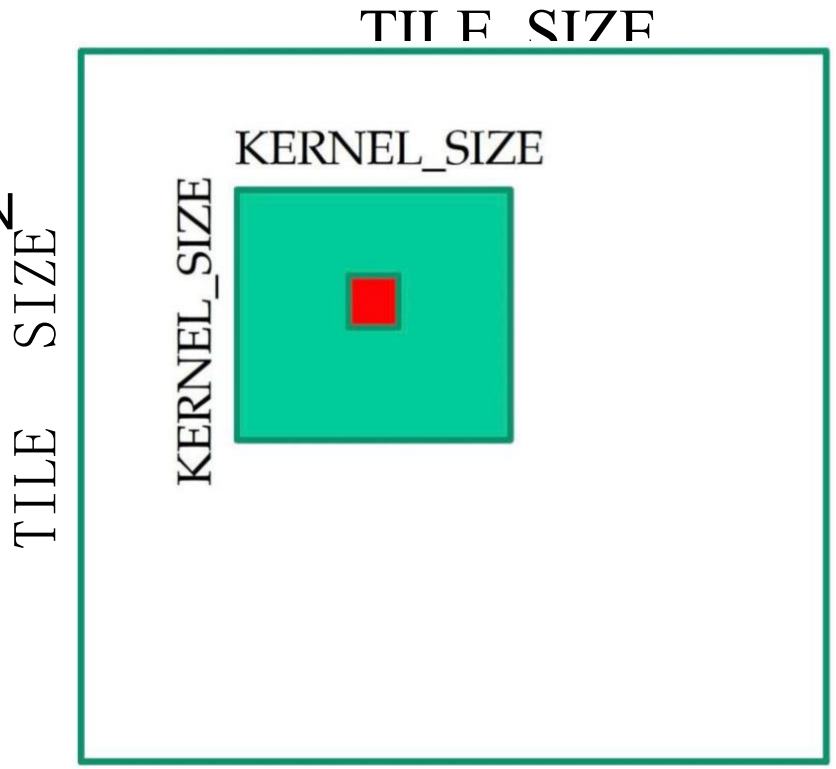
Arithmetic to Global Memory Access Ratio

- Considering an MxM filter
- Without tiling:
  - Operations per thread: M^2 adds $+M^2$ muls = $2M^2$ OP
  - Global loads per thread: $M^2 \times 4$ B = $4M^2$ B
  - Ratio: $(2M^2OP)/(4M^2B) = 0.5$ OP/B
- With tiling:
  - Considering tile dimensions: input =T, output =T-M+1
  - Operations per block: $(T-M+1)^2 \times 2M^2$ OP
  - Global loads per block: $T^2 \times 4$ B
  - Ratio: $((T-M+1)^2 \times 2M^2OP)/(T^2 \times 4\ B) = 0.5M^2(1-(M-1)/T)^2$

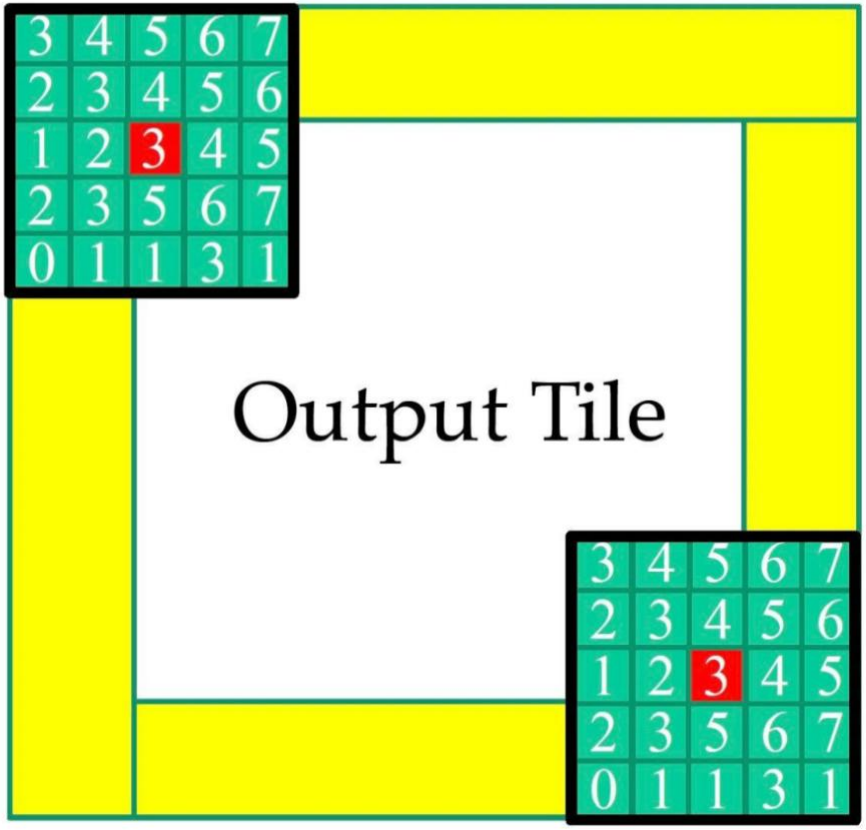
For M=5 and T=64:10.98 OP/B(≈22×improvement!)

High-Level Tiling Strategy-Input Tiling

- Load a tile of N into LDS
  - -All threads participate in loading
  - -A subset of threads then use each N elements in SM



Input tiles need to be larger than output tiles



Input Tile

We will use a strategy where the input tile will be loaded into the shared memory

Dealing with Mismatch

- Use a thread block that matches the input tile

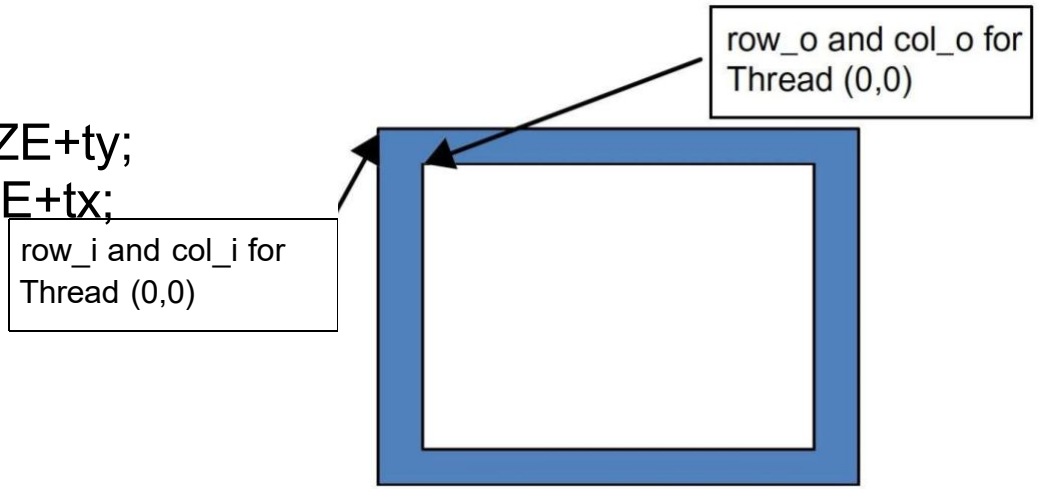
  -Each thread loads one element of the input tile

  -Some threads do not participate in calculating the output

  > There will be "if"statements and control divergence

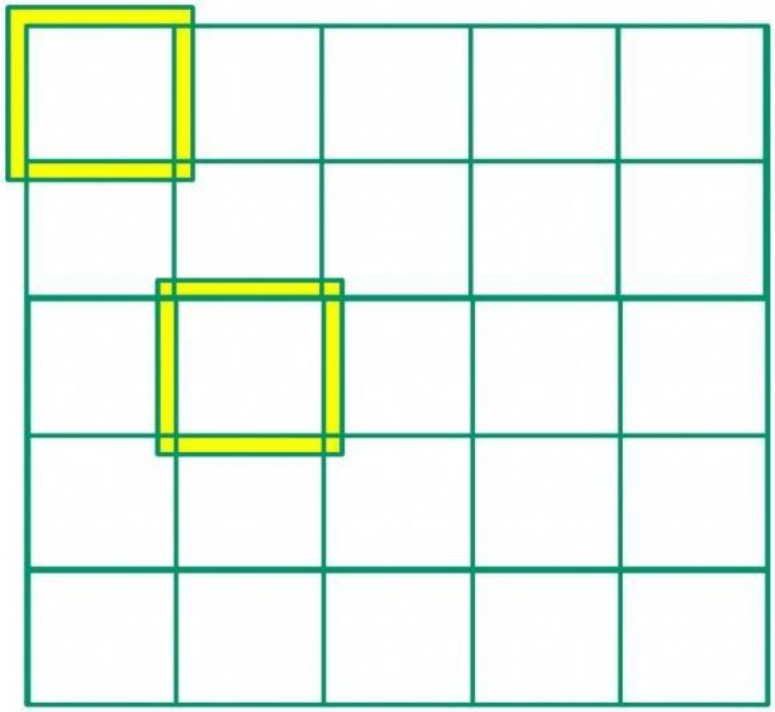
Shifting from output coordinates to input coordinate

```
int  tx  =threadIdx.X;
int  ty  =threadIdx.y;

int  row_o=blockIdx.y*TILE_SIZE+ty;
int  col_o=blockIdx.x*TILE_SIZE+tx;

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

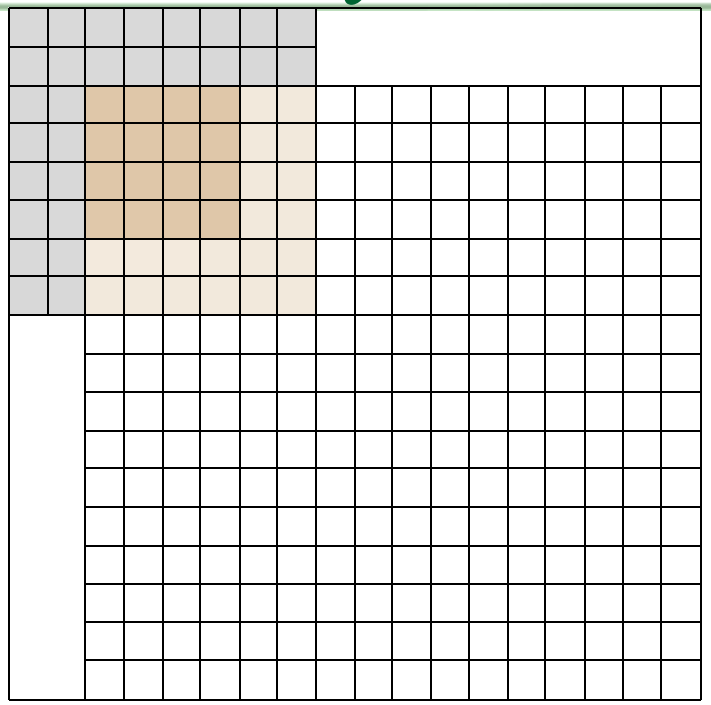


Loading halos

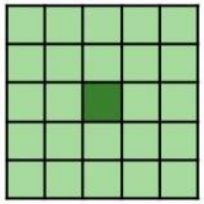
m Threads that load halos outside N should return 0.0



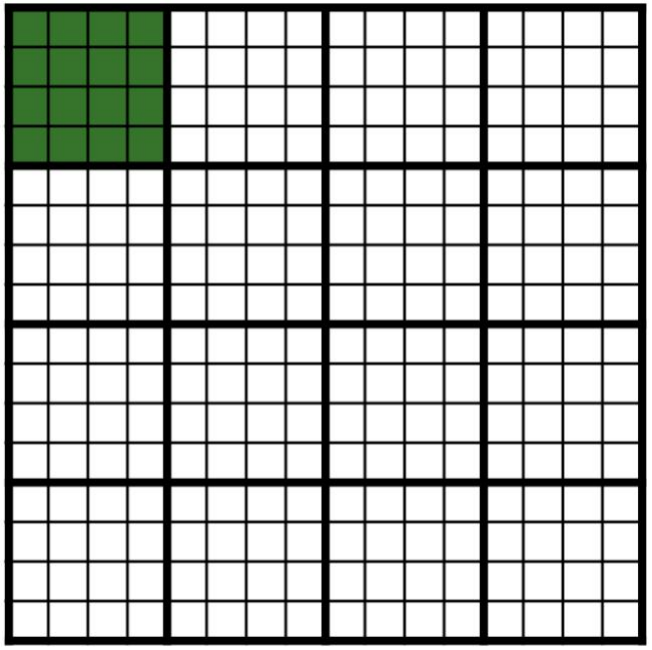
Boundary Conditions



input
(in global memory)



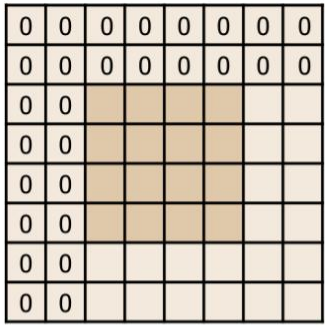
filter
(in constant memory)



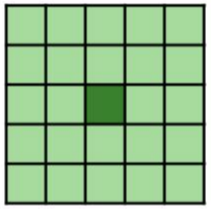
output

Threads computing output elements at the boundary access input elements that are out of bounds (also called ghost elements)

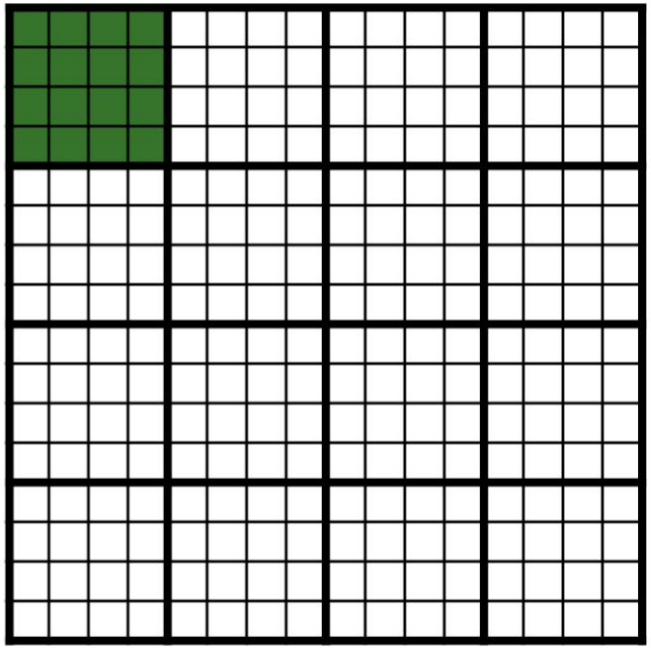
Boundary Conditions



$input_{tile}$
(in shared memory)



filter
(in constant memory)



output

Threads computing output elements at the boundary access input elements that are out of bounds (also called ghost elements)

Solution: Store zero to shared memory tile for our of bounds input elements

Boundary Conditions

```
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;
```

Calculating Output

Some threads do not participate in calculating the output

```
If(ty<TILE_SIZE && tx<TILE_SIZE){
  for(i=0; i<5; i++){
    for(j=0; j<5; j++){
      Output += Mc[i][j] * Ns[i+ty][j+tx];
```

Writing Output

Some threads do not write output

```
lf (row_o<Pheight  && col_o <Pwidth){

P.elements[row_o * Pwidth+col_o] = output;
```

2D Convolution with Tiling and Constant Memory

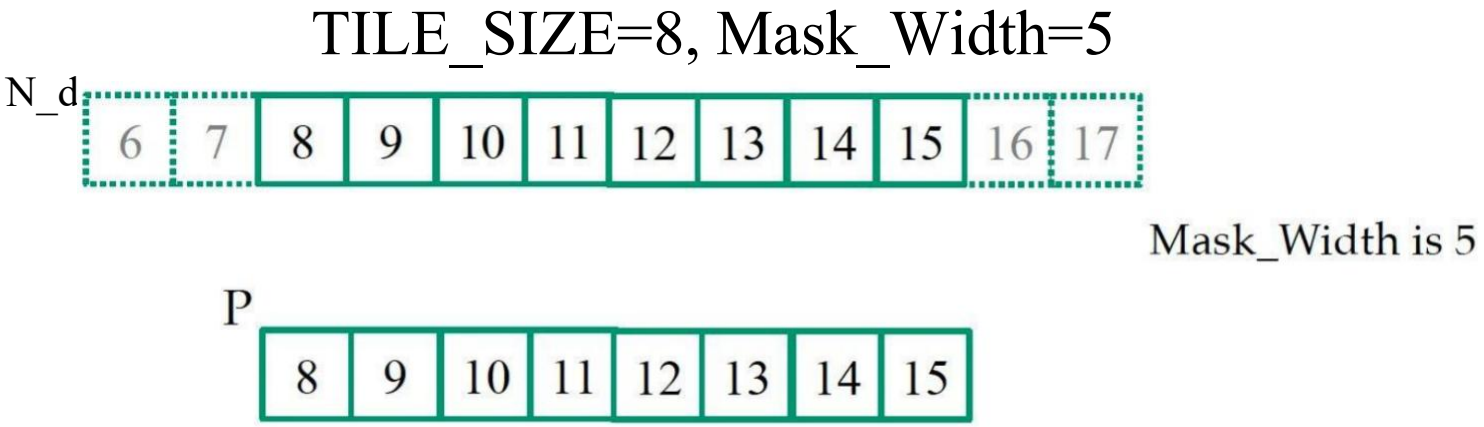
```
01  #define IN_TILE_DIM 32
02  #define OUT_TILE_DIM ((IN_TILE_DIM) - 2*(FILTER_RADIUS))
03  __constant__ float F_c[2*FILTER_RADIUS+1][2*FILTER_RADIUS+1];
04  __global__ void convolution_tiled_2D_const_mem_kernel(float *N, float *P,
05                                                        int width, int height) {
06    int col = blockIdx.x*OUT_TILE_DIM + threadIdx.x - FILTER_RADIUS;
07    int row = blockIdx.y*OUT_TILE_DIM + threadIdx.y - FILTER_RADIUS;
08    //loading input tile
09      shared   N s[IN TILE DIM][IN TILE DIM];
10    if(row>=0 && row<height && col>=0 && col<width) {
11      N_s[threadIdx.y][threadIdx.x] = N[row*width + col];
12    } else {
13      N_s[threadIdx.y][threadIdx.x] = 0.0;
14    }
15    __syncthreads();
16    // Calculating output elements
17    int tileCol = threadIdx.x - FILTER_RADIUS;
18    int tileRow = threadIdx.y - FILTER_RADIUS;
19    // turning off the threads at the edges of the block
20    if (col >= 0 && col < width && row >=0 && row < height) {
21      if (tileCol>=0 && tileCol<OUT_TILE_DIM && tileRow>=0
22                     && tileRow<OUT_TILE_DIM){
23        float Pvalue = 0.0f;
24        for (int fRow = 0; fRow < 2*FILTER_RADIUS+1; fRow++) {
25          for (int fCol = 0; fCol < 2*FILTER_RADIUS+1; fCol++) {
26            Pvalue += F[fRow][fCol]*N_s[tileRow+fRow][tileCol+fCol];
27          }
28        }
29        P[row*width+col] = Pvalue;
30      }
31    }
32  }
```

2D Convolution with Tiling &Constant Memory with Caches

```
01  #define TILE_DIM 32
02  __constant__ float F_c[2*FILTER_RADIUS+1][2*FILTER_RADIUS+1];
03    global   void convolution_cached_tiled_2D_const_mem_kernel(float *N,
                                        float *P, int width, int height) {
04    int col = blockIdx.x*TILE_DIM + threadIdx.x;
05    int row = blockIdx.y*TILE_DIM + threadIdx.y;
      //loading input tile
06    __shared__ N_s[TILE_DIM][TILE_DIM];
07    if(row<height && col<width) {
08      N_s[threadIdx.y][threadIdx.x] = N[row*width + col];
09    } else {
10      N_s[threadIdx.y][threadIdx.x] = 0.0;
11    }
12    __syncthreads();
      // Calculating output elements
      // turning off the threads at the edges of the block
13    if (col < width && row < height) {
14      float Pvalue = 0.0f;
15      for (int fRow = 0; fRow < 2*FILTER_RADIUS+1; fRow++) {
16        for (int fCol = 0; fCol < 2*FILTER_RADIUS+1; fCol++) {
17          if (threadIdx.x-FILTER_RADIUS+fCol >= 0 &&
18              threadIdx.x-FILTER_RADIUS+fCol < TILE_DIM &&
19              threadIdx.y-FILTER_RADIUS+fRow >= 0 &&
20              threadIdx.y-FILTER_RADIUS+fRow < TILE_DIM){
21            Pvalue += F[fRow][fCol]*N_s[threadIdx.y+fRow][threadIdx.x+fCol];
22          }
23          else {
24            if (row-FILTER_RADIUS+fRow >= 0 &&
25                row-FILTER_RADIUS+fRow < height &&
26                col-FILTER_RADIUS+fCol >=0 &&
27                col-FILTER_RADIUS+fCol < width) {
24              Pvalue += F[fRow][fCol]*
25                                        N[(row-FILTER_RADIUS+fRow)*width+col-
FILTER_RADIUS+fCol];
26            }
27          }
28        }
29        P[row*width+col] = Pvalue;
30      }
31    }
32  }
```

Tiled Reuse Analysis

1D Example

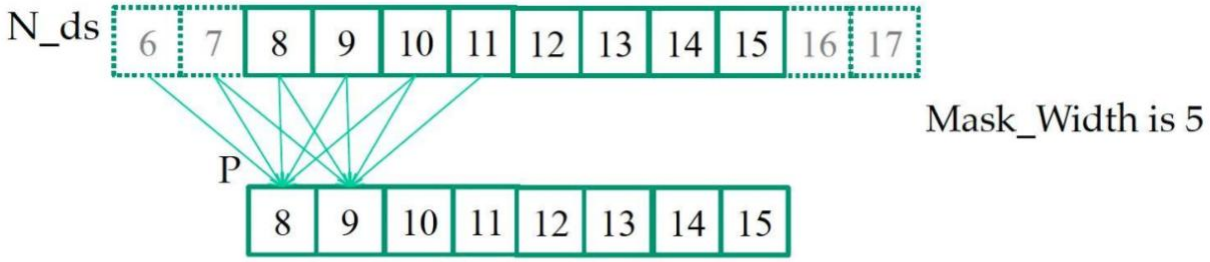


- output and input tiles for block 1

- For Mask_Width=5, each block loads 8+5-1=12 elements(12 memory loads)

1D Example

Each output P element uses 5 elements (in N_ds)



- 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] uses 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]

A Total of 8 * 5 N elements are used for the output tile.

1D Example

A simple way to calculate tiling benefit

1. Elements loaded with shared memory: (8+2+2)=12
2. Global memory accesses without shared memory: 8×5

3. Bandwidth reduction: 40/12=3.3

1D Example

Elements loaded with shared memory:

TILE_SIZE+Mask_Width-1

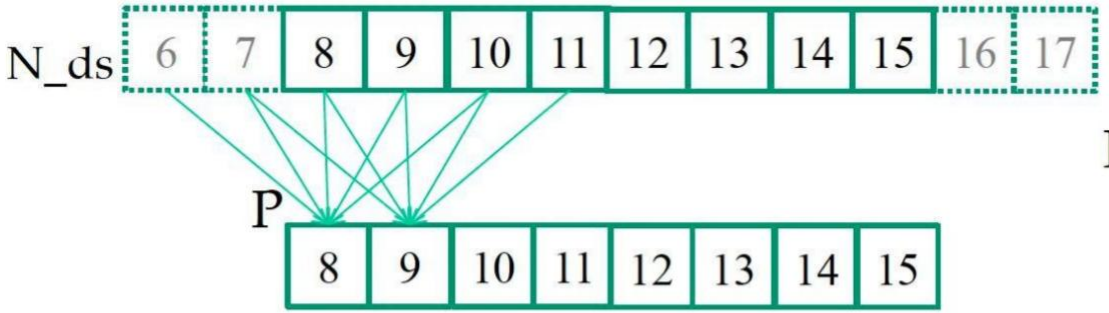
Global memory accesses without shared memory:

TILE_SIZE×Mask_Width

Bandwidth reduction:

$$\frac{\text{TILE\_SIZE+Mask\_Width-1}}{\text{TILE\_SIZE×Mask\_Width}}$$

Reuse



Mask_Width is 5

- N_ds[6] is used by P[8] (1X)
- N_ds[7] is used by P[8], P[9] (2X)
- N_ds[8] is used by P[8], P[9], P[10] (3X)
- N_ds[9] is used by P[8], P[9], P[10], P[11] (4X)
- N_ds[10] is used by P[8], P[9], P[10], P[11], P[12] (5X)
- … (5X)
- N_ds[14] is uses by P[12], P[13], P[14], P[15] (4X)
- N_ds[15] is used by P[13], P[14], P[15] (3X)
- N_ds[16] is used by P[14], P[15] (2X)
- N_ds[17] is used by P[15[(1X)

Reuse

Each time an N_ds element is used,it replaces access to the global memory.

- The total number of global memory accesses without using shared memory:

1+2+3+4+5*(8-5+1)+4+3+2+1

=10+20+10

=40

Bandwidth reduction: 40/12 =3.3

Ghost elements change ratios

Load for a boundary tile:TILE_SIZE + (Mask_Width-1)/2 elements

-10 in our example of Tile_Width =8 and Mask_Width=5

·Computing boundary elements do not access global memory for ghost cells

-Total accesses is 3+4+6×5 =37 accesses

The reduction is 37/10 =3.7

1D,internal tiles

- The total number of global memory accesses to the (TILE_SIZE+Mask_Width-1)N elements replaced by shared memory accesses is

1+2+…+Mask_Width-1+Mask_Width*(TILE_SIZE-Mask_Width+1)+
Mask_Width-1+…+2+1

=((Mask_Width-1)*Mask_Width)/2+Mask_Width*(TILE_SIZE-Mask_Width+1)+((Mask_Width-1)*Mask_Width)/2

=(Mask_Width-1) *Mask_Width+Mask_Width*(TILE_SIZE-Mask_Width+1)

=Mask_Width*(TILE_SIZE)

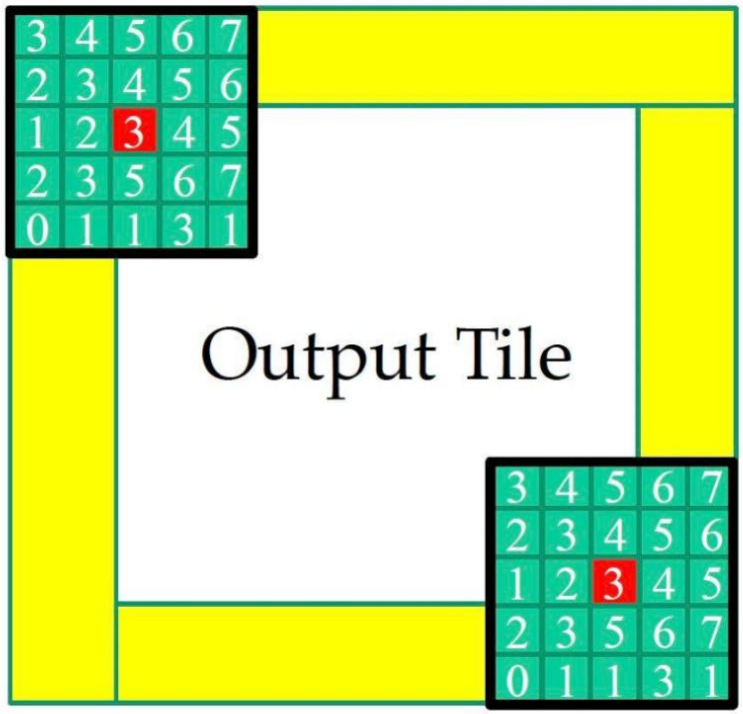
Bandwidth Reduction for 1D

- The reduction: $\frac{Mask\_Width \times TILE\_SIZE}{TILE\_SIZE+Mask\_Width-1}$

TILE_SIZE	16	32	64	128	256
Reduction Mask_Width =5	4.0	4.4	4.7	4.9	4.9
Reduction Mask_Width =9	6.0	7.2	8.0	8.5	8.7

2D Tiling

Input tiles need to be larger than output tiles

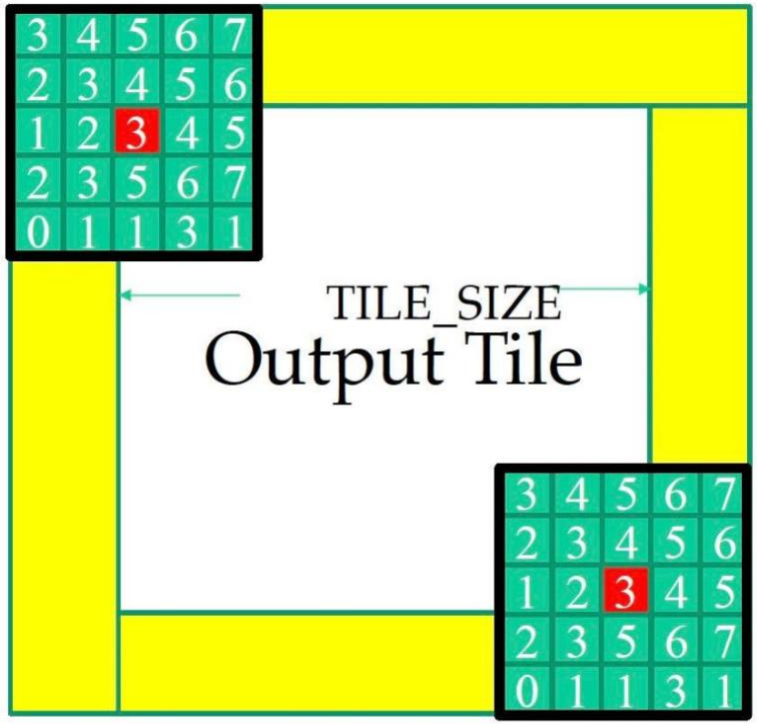


Input Tile

We will use a strategy where the input tile will be loaded into the shared memory.

2D Tiling

Input tiles need to cover halo elements.



Mask_Width =5

Input Tile

2D Tiling:Analysis for a small 8X8 output tile

8X8 output tile

Mask_Width =5

Need to be loaded into shared memory: $12 \times 12 = 144$ elements.

The calculation of each P element needs to access 25 elements

Without shared memory:the number of global memory accesses: $8 \times 8 \times 25 = 1,600$

The reduction: 1, 600/144 = 11X

2D Tiling

- Need to load into shared memory: (TILE_SIZE+Mask_Width-1)² Elements

  The calculation of each P element: needs to access Mask_Width² elements

  Global memory accesses are converted into shared memory accesses: TILE_SIZE²×Mask_Width²

- The reduction: $\frac{\text{TILE\_SIZE}^2 \times \text{Mask\_Width}^2}{(\text{TILE\_SIZE}+\text{Mask\_Width}-1)2}$

Bandwidth Reduction for 2D

- The reduction: $$\frac{\text{TILE\_SIZE}^2 \times \text{Mask\_Width}^2}{(\text{TILE\_SIZE}+\text{Mask\_Width}-1)2}$$

TILE SIZE	8	16	32	64
Reduction Mask Width =5	11.1	16	19.7	22.1
Reduction Mask_Width=9	20.3	36	51.8	64