# CS4990 – GPU Computing
## Module 7: Convolution

Hao Ji

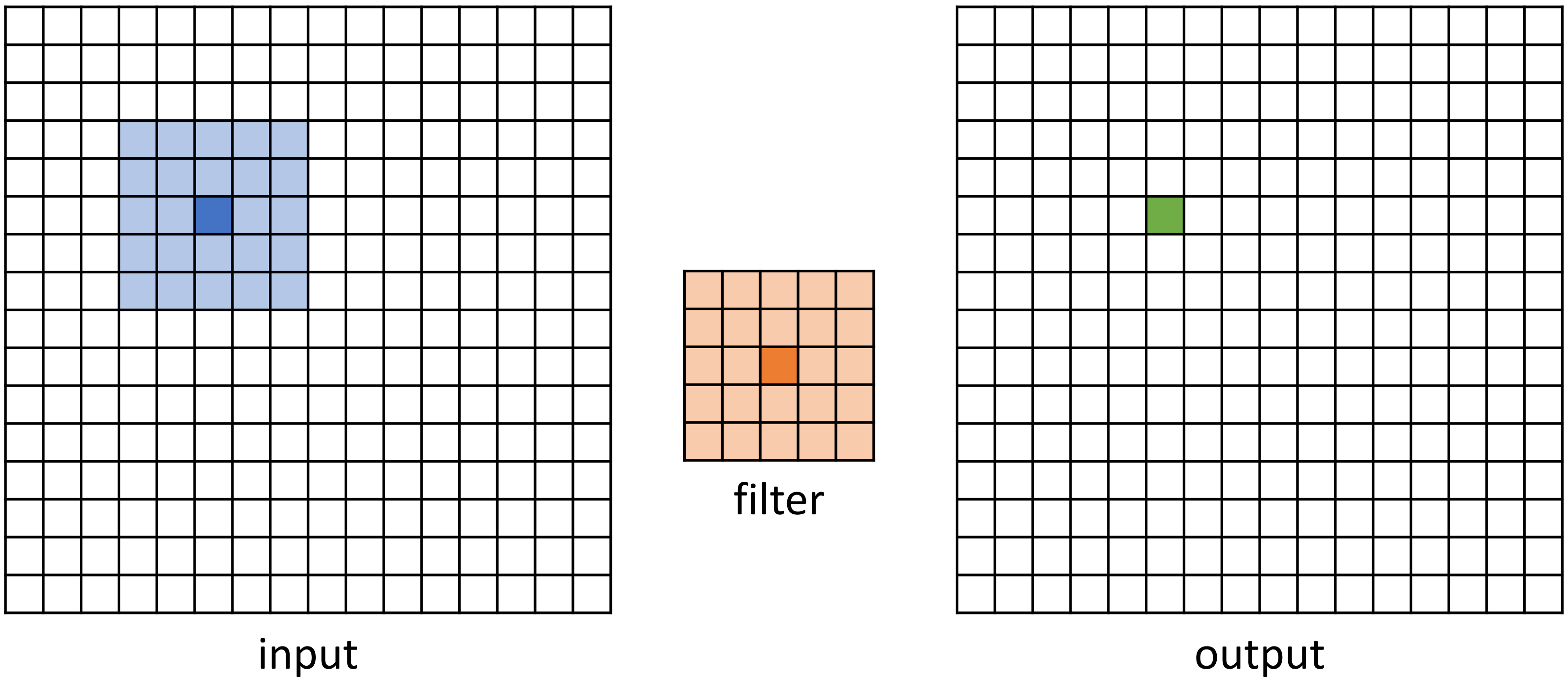
Computer Science Department

Cal Poly Pomona

# Outline

- Convolution
- Parallel Convolution: a basic algorithm
- Constant Memory and Caching
- Tiled Convolution

# 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**

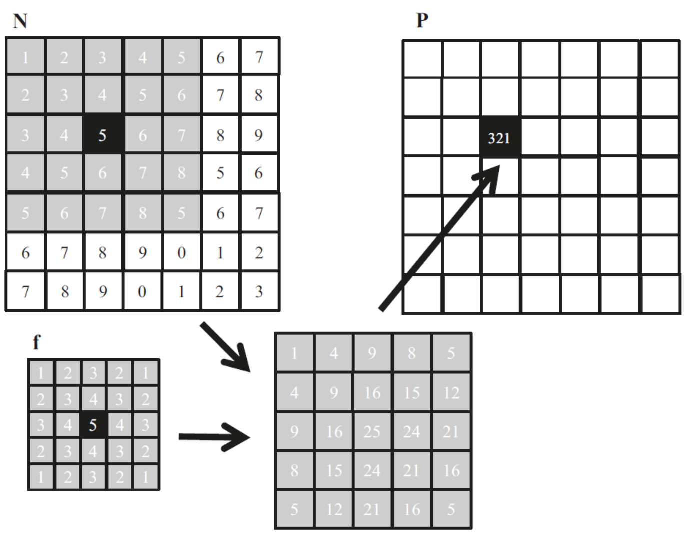
(commonly called convolution *kernel*, but we will use *filter* to avoid confusion with CUDA kernels)

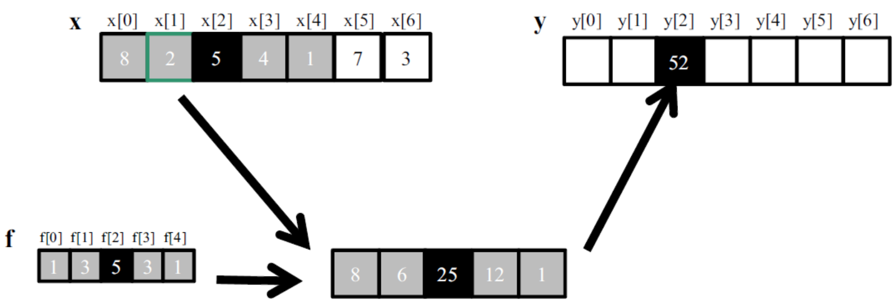
# Applications of Convolution

- Commonly used in signal processing, image processing, video processing, etc.

- Usually used to transform signals or pixels to more desirable values
  - e.g., Gaussian blur, sharpen, edge detection, etc.
  - Transformation depends on the weights in the filter

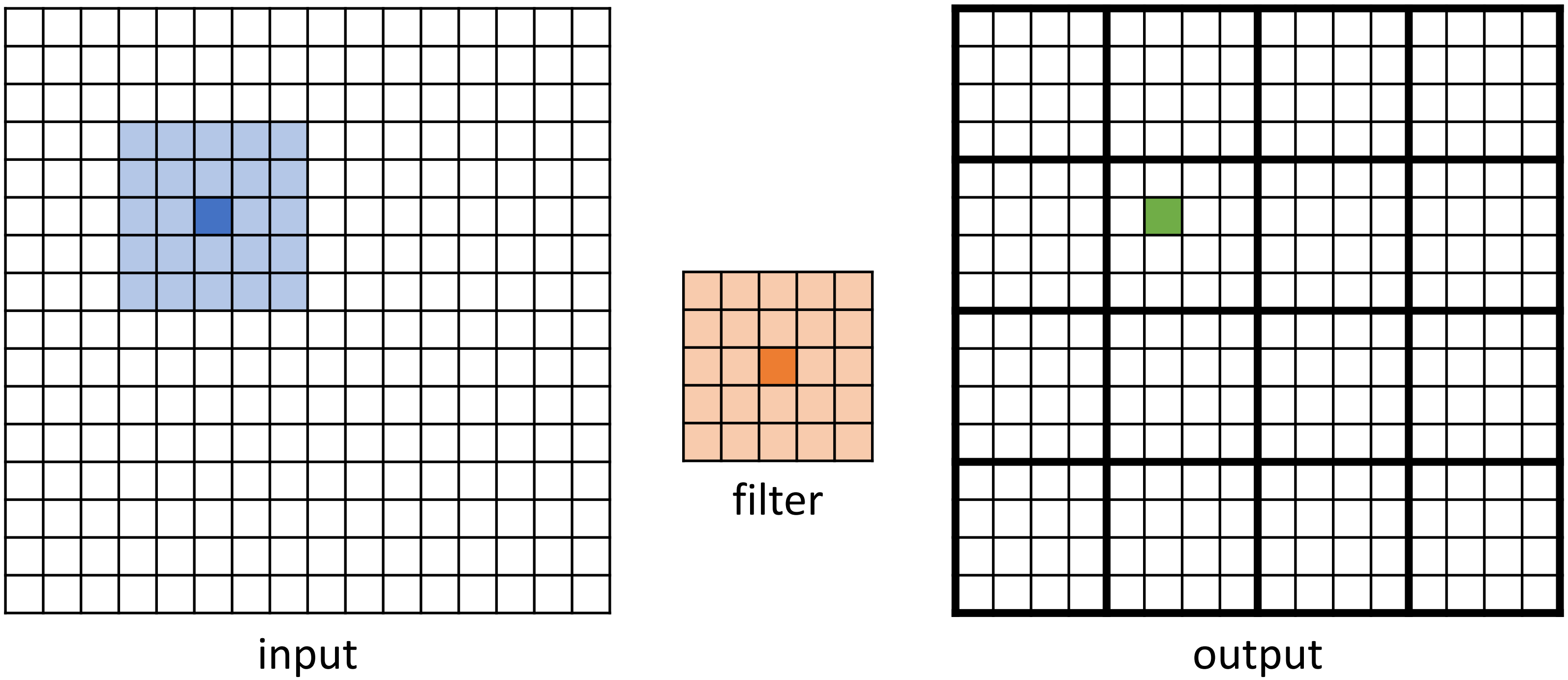
- Using 2D as an example, but can also be 1D or 3D



# Outline

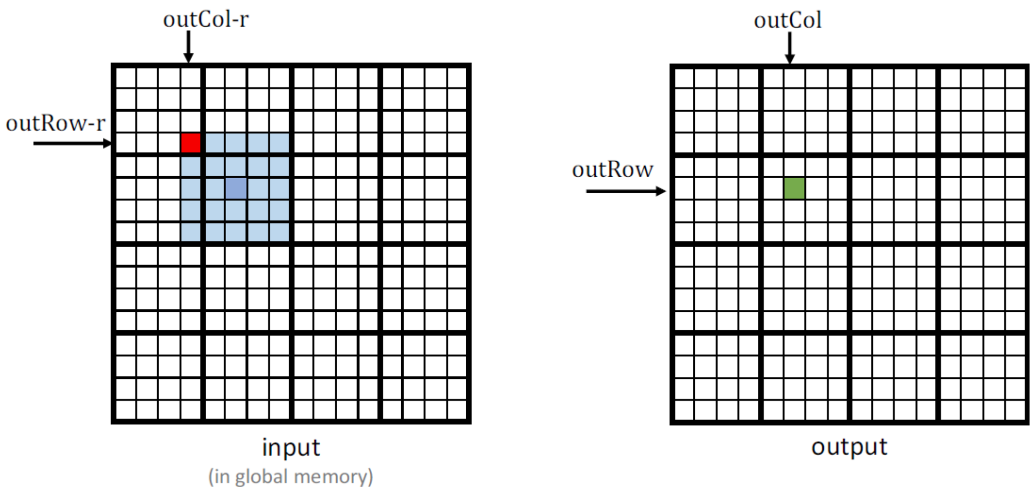
- Convolution
- **Parallel Convolution: a basic algorithm**
- Constant Memory and Caching
- Tiled Convolution




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

# Parallelizing Convolution

- Based on our experience in image smoothing and matrix multiplication, we can quickly write a simple parallel convolution kernel.

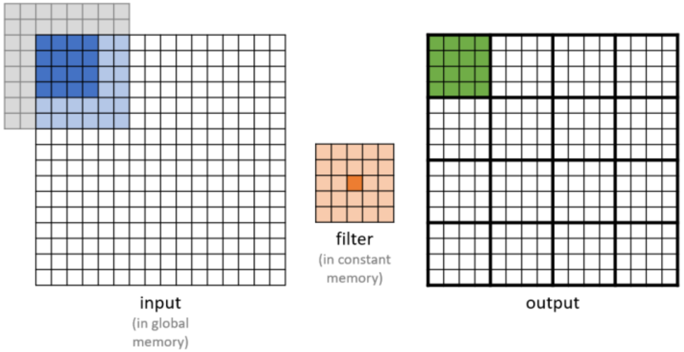



***N***: a pointer to the input array
***F***: a pointer to the filter
***P***: a pointer to the output array
***r***: the radius of the square filter
***width***: the width of the input and output arrays
***height***: the height of the input and output arrays

```
__global__ void convolution_2D_basic_kernel(float *N, float *F, float *P,
    int r, int width, int height) {
    int outCol = blockIdx.x*blockDim.x + threadIdx.x;
    int outRow = blockIdx.y*blockDim.y + threadIdx.y;
    float Pvalue = 0.0f;
    for (int fRow = 0; fRow < 2*r+1; fRow++) {
      for (int fCol = 0; fCol < 2*r+1; fCol++) {
          inRow = outRow - r + fRow;
          inCol = outCol - r + fCol;
          if (inRow >= 0 && inRow < height && inCol >= 0 && inCol < width) {
              Pvalue += F[fRow][fCol]*N[inRow*width + inCol];
          }
        }
    }
    P[outRow][outCol] = Pvalue;
}
```

- Two concerns, regarding performance considerations,
  - **Control Divergence**
    - The threads that calculate the output elements near the four edges of the *P* array will need to handle ghost cells. Therefore, control divergence occurs in the if-statement.
    - However, for large input arrays and small filters, control divergence occurs only in computing a small portion of the output elements, which will keep the effect of control divergence small.
  - **Memory Bandwidth**
    - The compute to global memory access ratio is only about 0.25 OP/B
      - 2 operations for every 8 bytes loaded (on line 10).
    - We will need to **reduce the number of global memory accesses**.

```
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 }
```



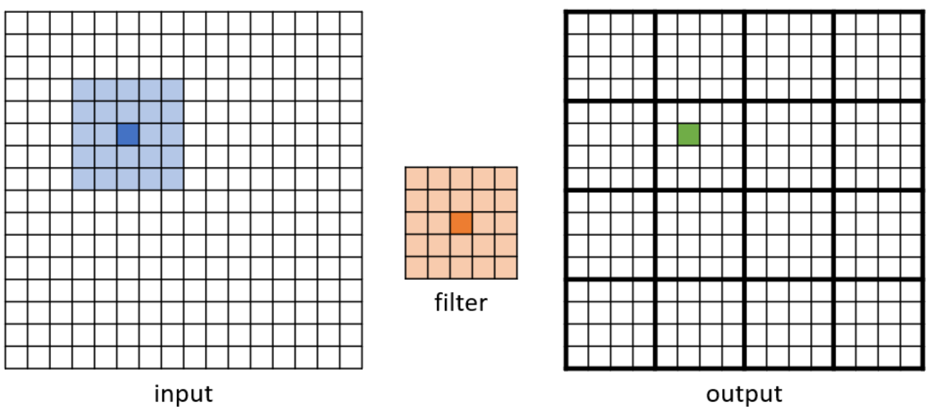
# Outline

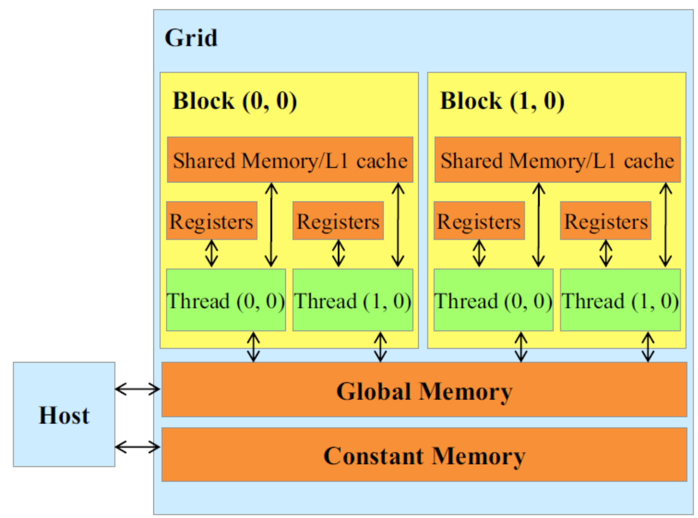
- Convolution
- Parallel Convolution: a basic algorithm
- **Constant Memory and Caching**
- Tiled Convolution

# Storing the Filter

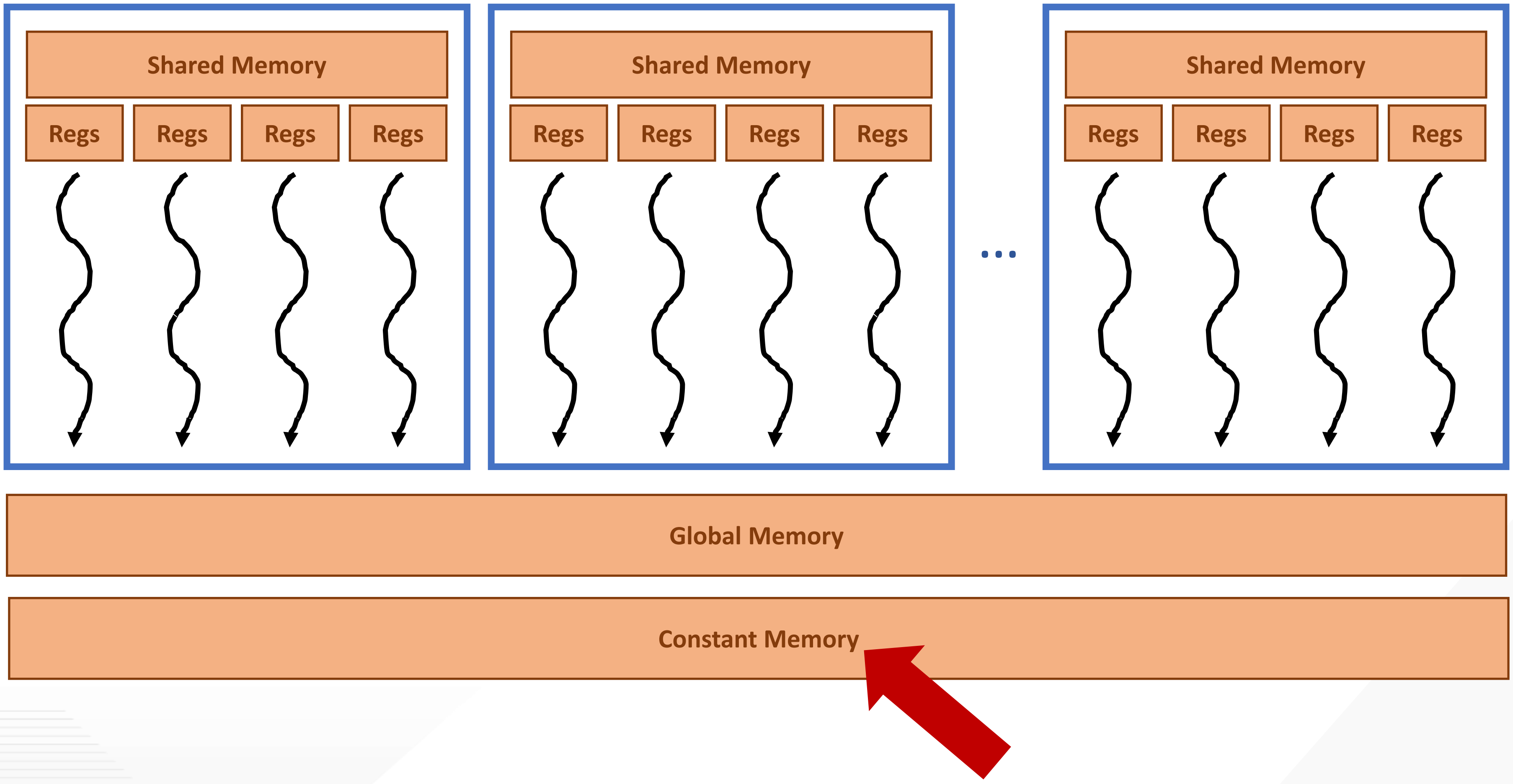
- Observations:
  - The filter is typically small
  - The filter is constant (weights do not change)
  - The filter is accessed by all threads in the grid
- These three properties make the filter an excellent candidate for constant memory and caching.

- Optimization: store the filter in **constant memory** for quicker access

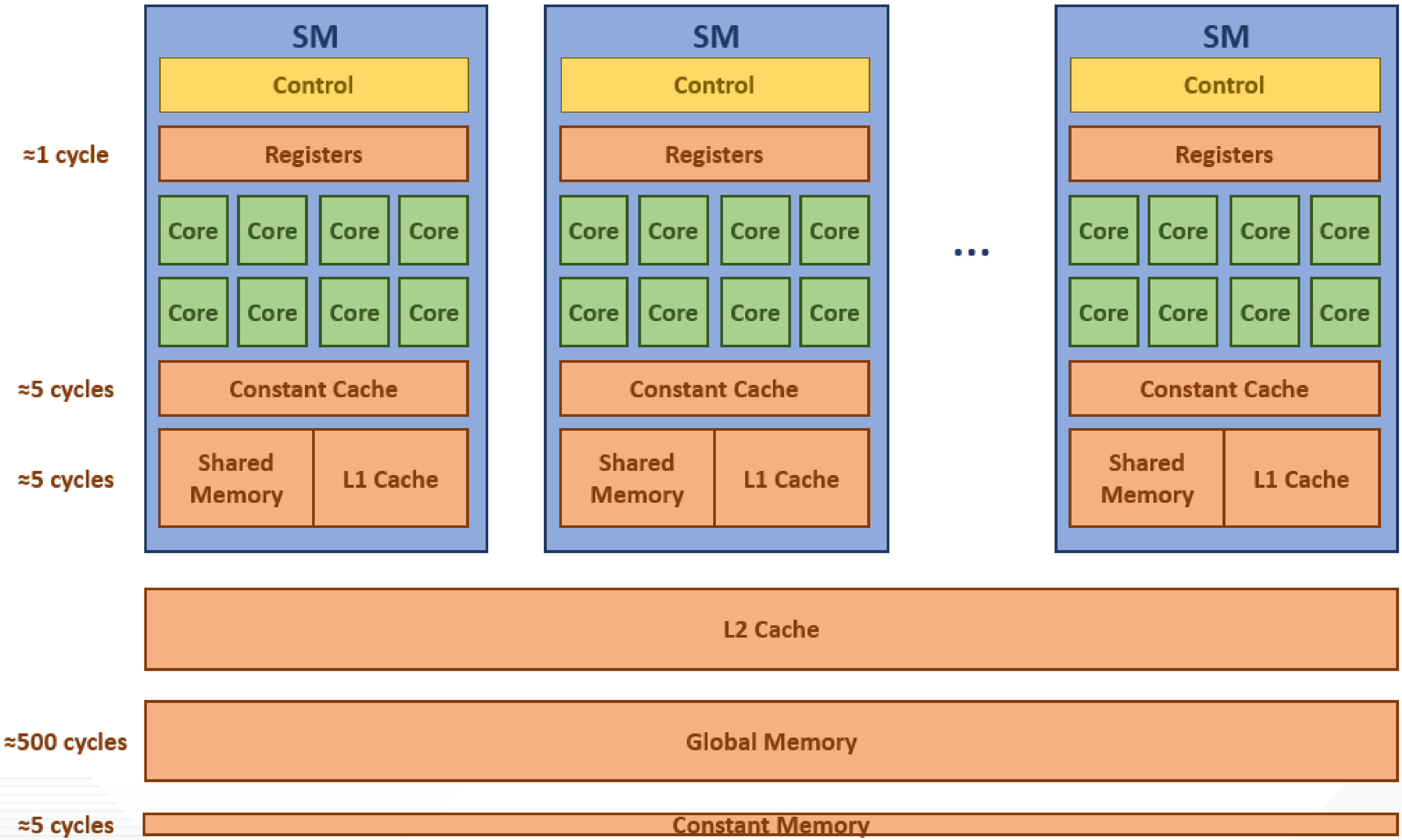


# Recall: Memory in the CUDA Programming Model



# Recall: Memory in the CUDA Programming Model



# Using Constant Memory

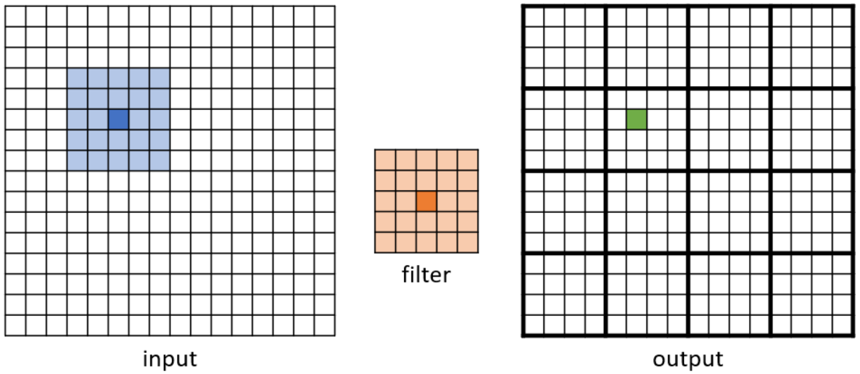
- Declare constant memory array as global variable

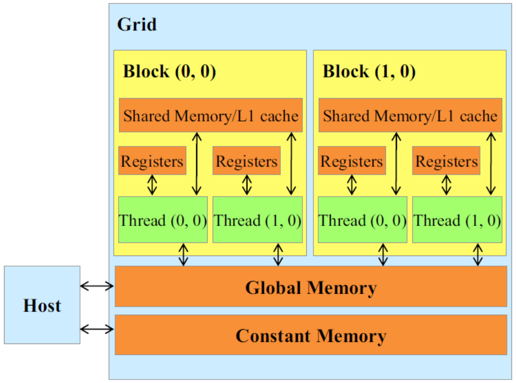
```
#define FILTER_RADIUS 2
__constant__ float F[2*FILTER_RADIUS+1][2*FILTER_RADIUS+1];
```

- Must initialize constant memory from the host:
  - Cannot modify during execution

```
cudaMemcpyToSymbol(F, F_h, (2*FILTER_RADIUS+1)*(2*FILTER_RADIUS+1)*sizeof(float));
```

- Can only allocate up to 64KB
  - Otherwise, input is also constant, but it is too large to put in constant memory

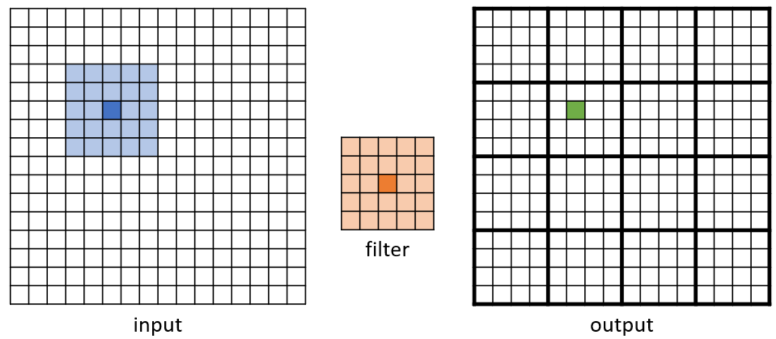


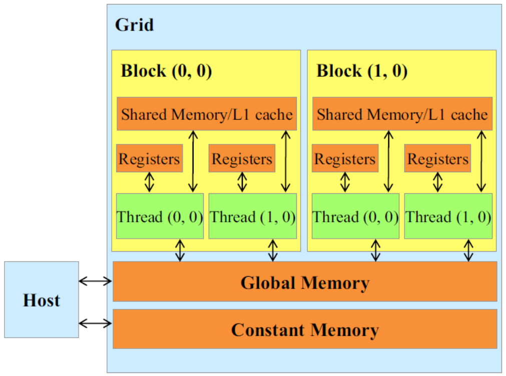



# Using Constant Memory

- A 2D convolution kernel **using constant memory for the filter *F***
  - Note: Kernel functions access constant memory variables as global variables. Therefore, their pointers do not need to be passed to the kernel as arguments.

```
__global__ void convolution_2D_const_mem_kernel(float *N, float *P, int r,
      int width, int height) {
    int outCol = blockIdx.x*blockDim.x + threadIdx.x;
    int outRow = blockIdx.y*blockDim.y + threadIdx.y;
    float Pvalue = 0.0f;
    for (int fRow = 0; fRow < 2*r+1; fRow++) {
      for (int fCol = 0; fCol < 2*r+1; fCol++) {
         inRow = outRow - r + fRow;
         inCol = outCol - r + fCol;
         if (inRow >= 0 && inRow < height && inCol >= 0 && inCol < width) {
            Pvalue += F[fRow][fCol]*N[inRow*width + inCol];
         }
      }
    }
    P[outRow*width+outCol] = Pvalue;
}
```



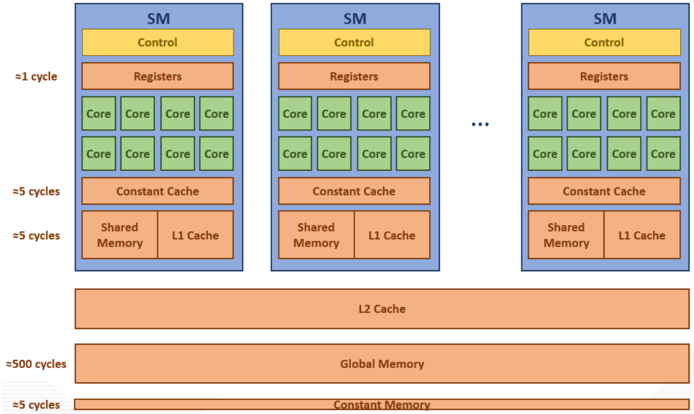
# Motivation for Constant Memory

- Constant data: easier to build an efficient cache
  - No need to support write back

- Constant Cache
  - A small, specialized cache can be highly effective in capturing the heavily used constant memory variable for each kernel.

- (Slightly Improved) Memory Bandwidth
  - The compute to global memory access ratio is now about 0.5 OP/B
    - 2 operations for every 4 bytes loaded (on line 10).
  - However, **further optimization is needed to reduce the number of global memory accesses.**

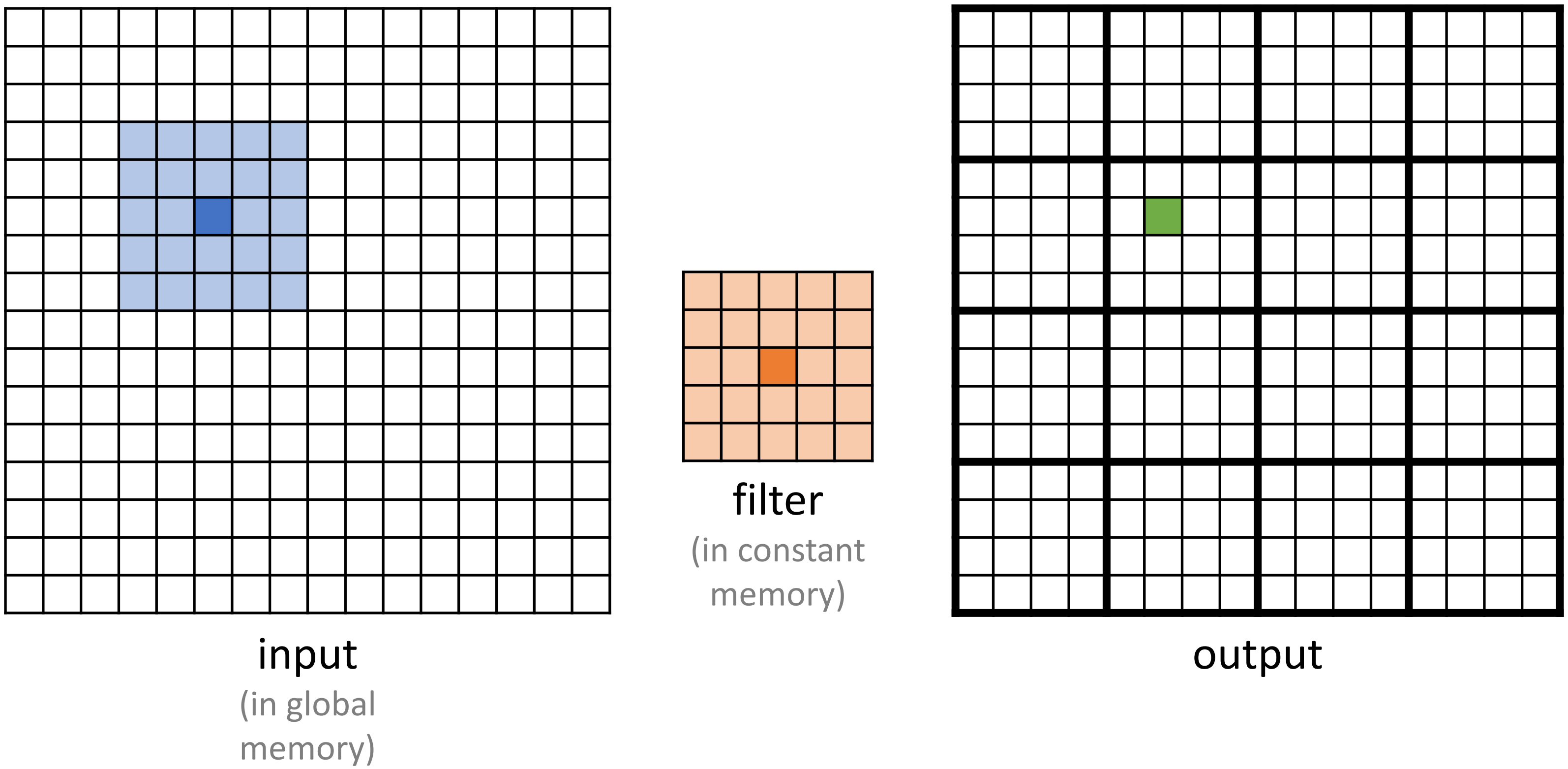
```
01 __global__ void convolution_2D_const_mem_kernel(float *N, float *P, int r,
      int width, int height) {
02    int outCol = blockIdx.x*blockDim.x + threadIdx.x;
03    int outRow = blockIdx.y*blockDim.y + threadIdx.y;
04    float Pvalue = 0.0f;
05    for (int fRow = 0; fRow < 2*r+1; fRow++) {
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 }
```



# Outline

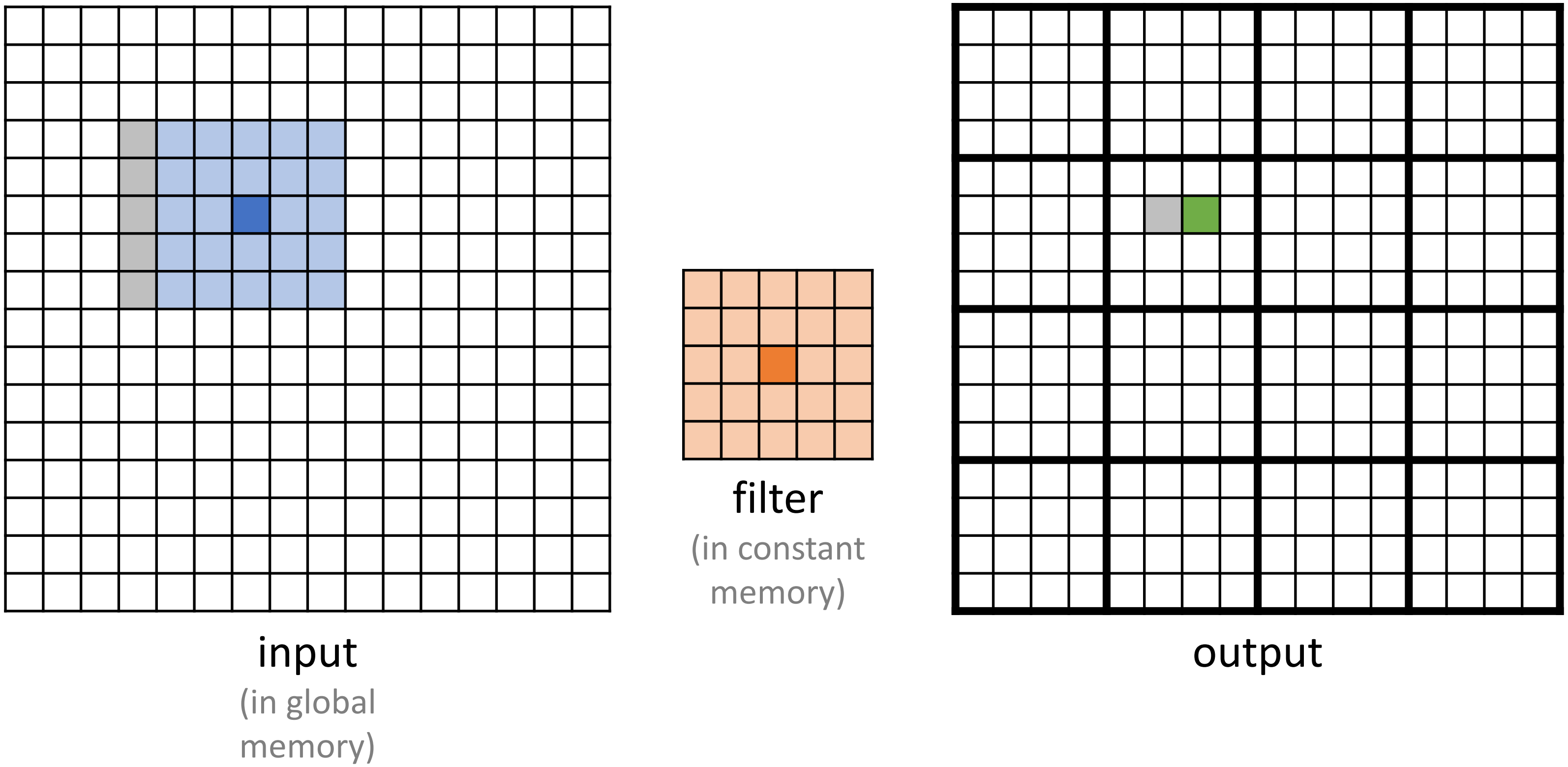
- Convolution
- Parallel Convolution: a basic algorithm
- Constant Memory and Caching
- **Tiled Convolution**

# Data Reuse in Convolution



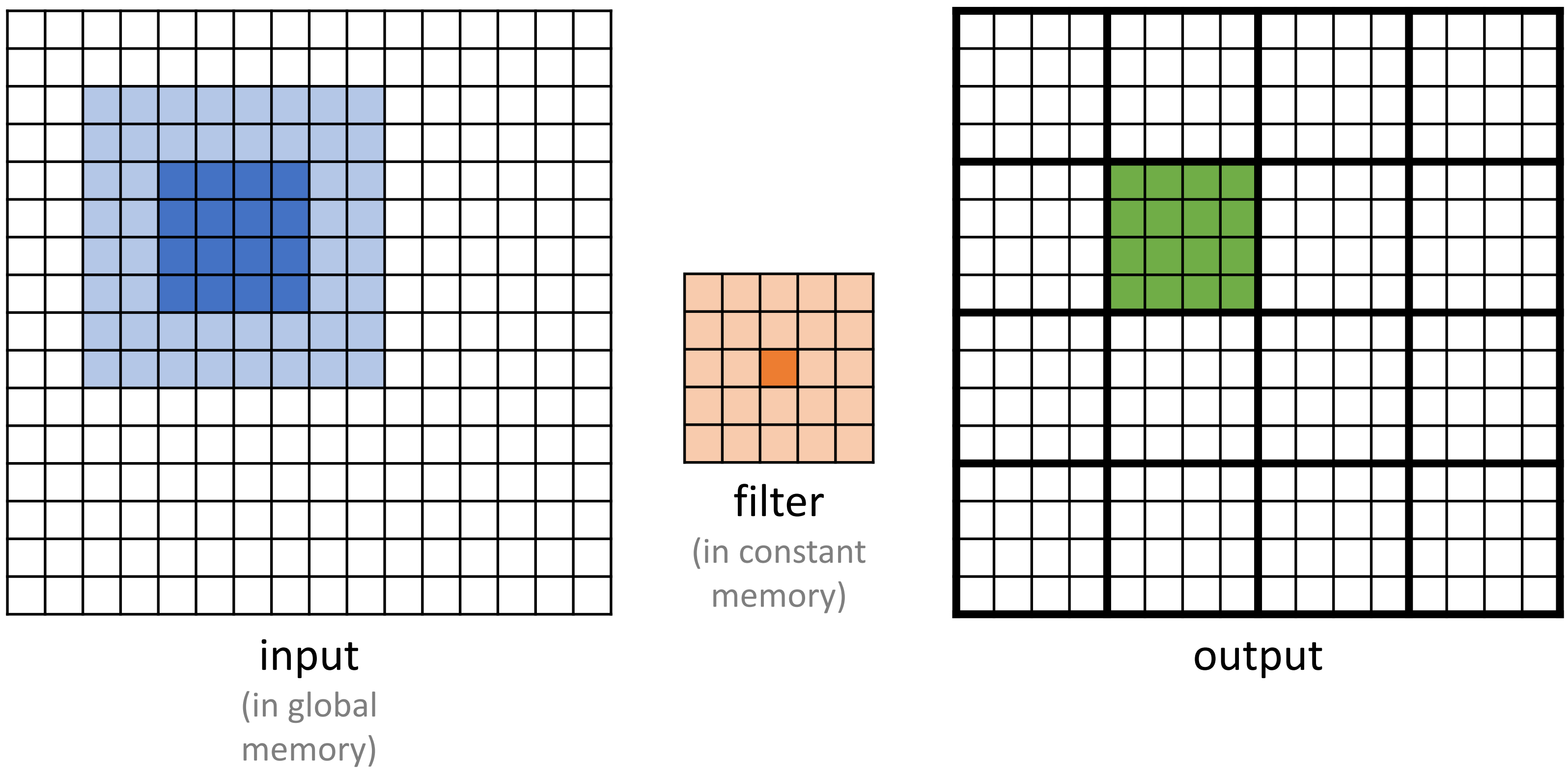

**<u>Observation:</u>** Threads in the same block load some of the same input elements

# Data Reuse in Convolution




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

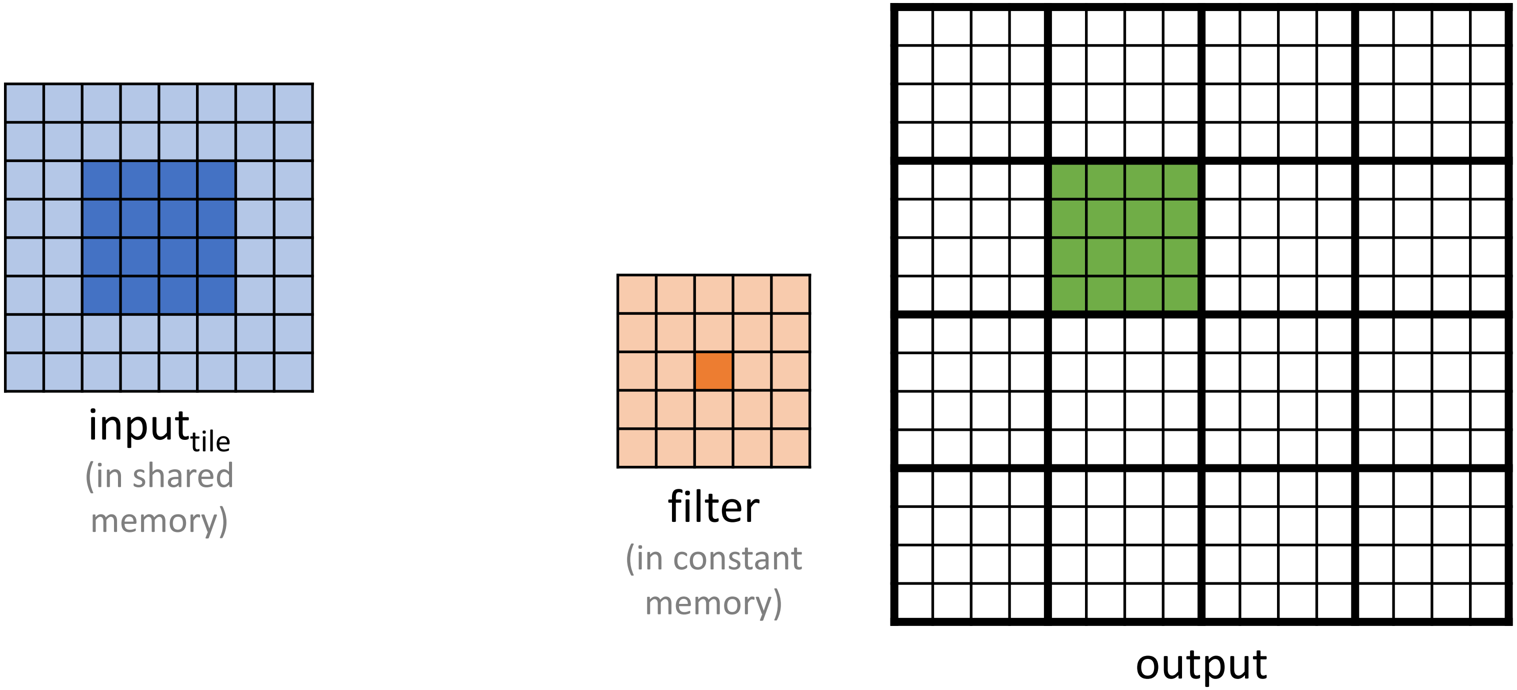
# Data Reuse in Convolution




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

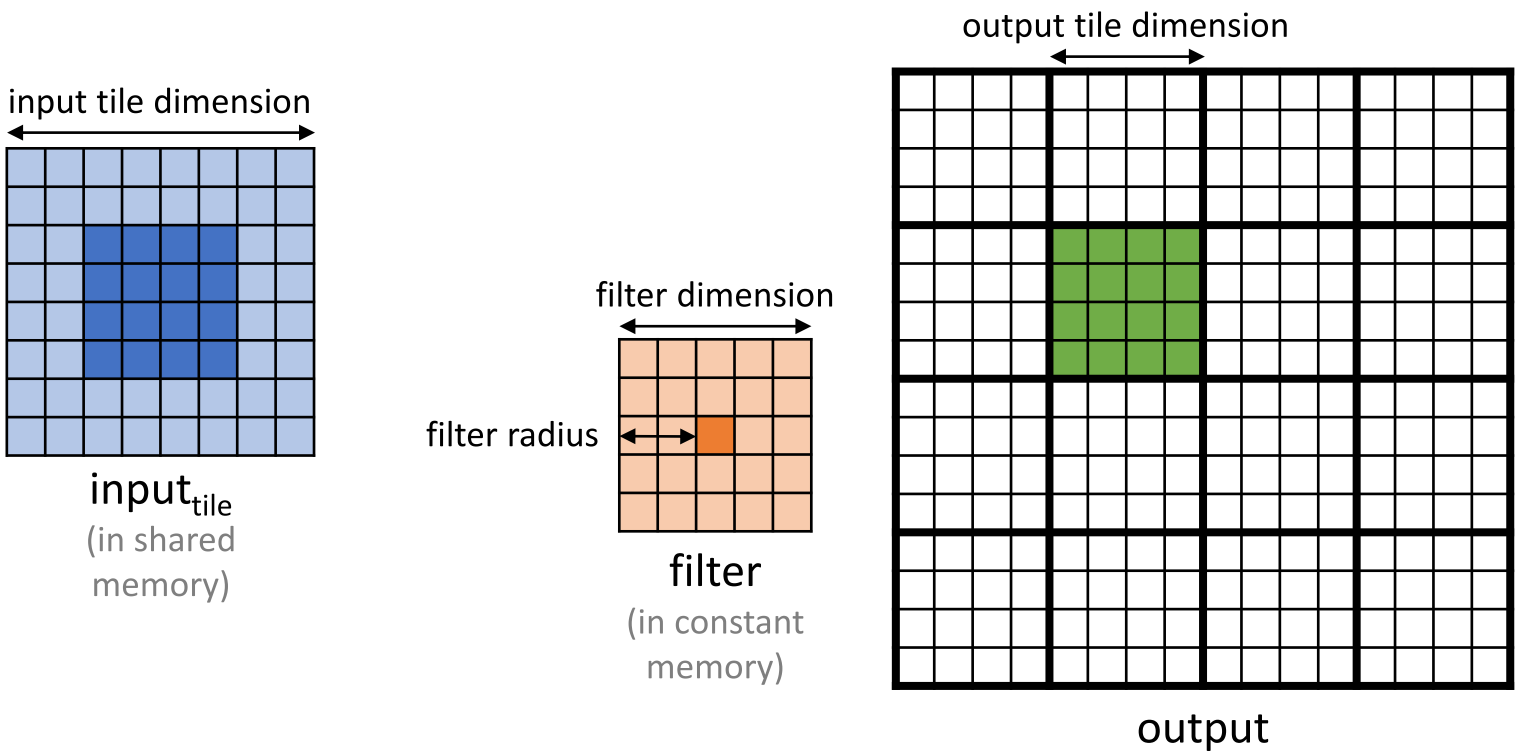
**Optimization:** Each thread loads one input element to shared memory and other threads access the element from shared memory

# Convolution with Tiling




**<u>Optimization:</u>** Each thread loads one input element to shared memory and other threads access the element from shared memory

# Difference in Tile Sizes

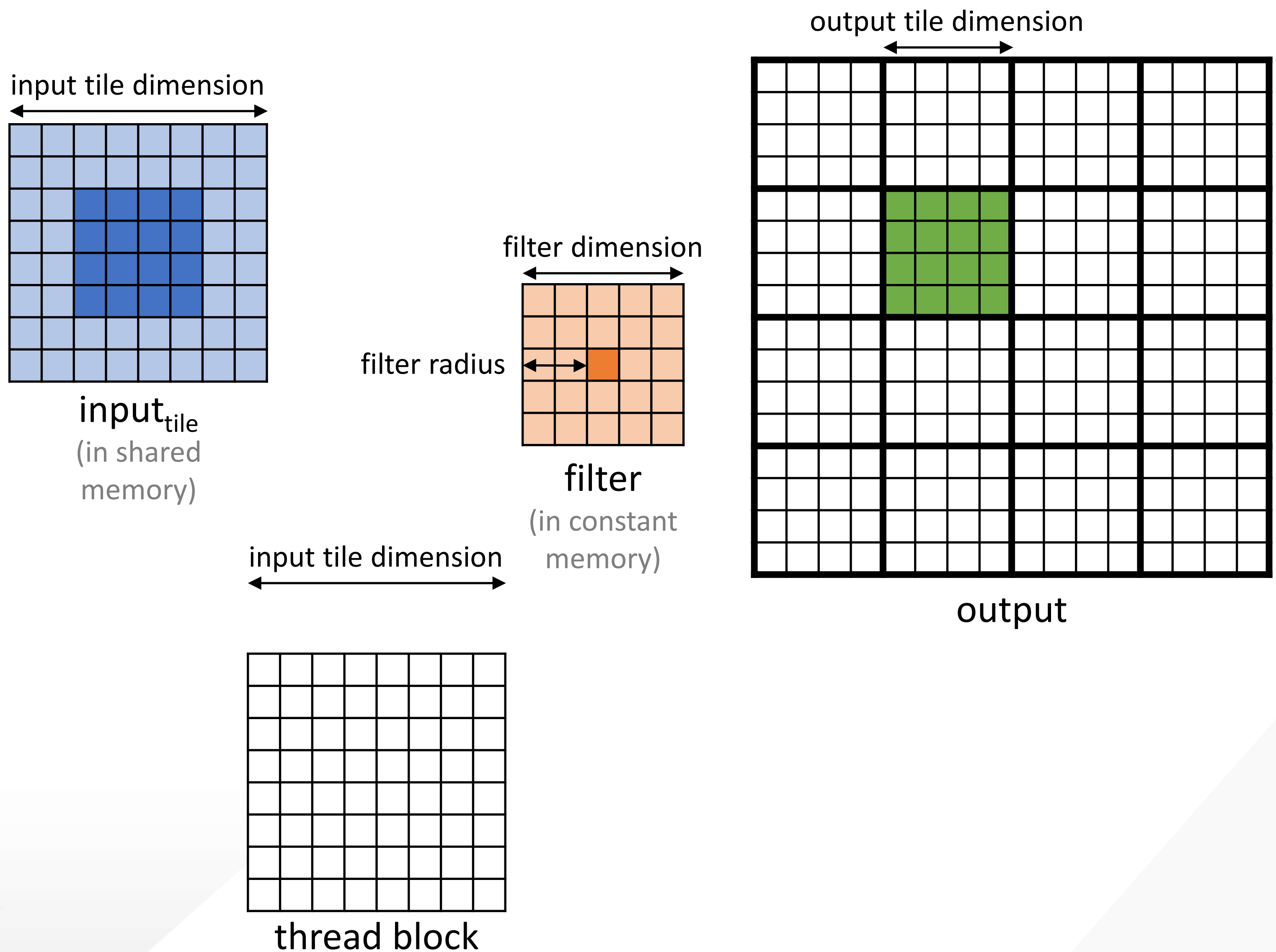



**Challenge:** Input and output tiles have different dimensions

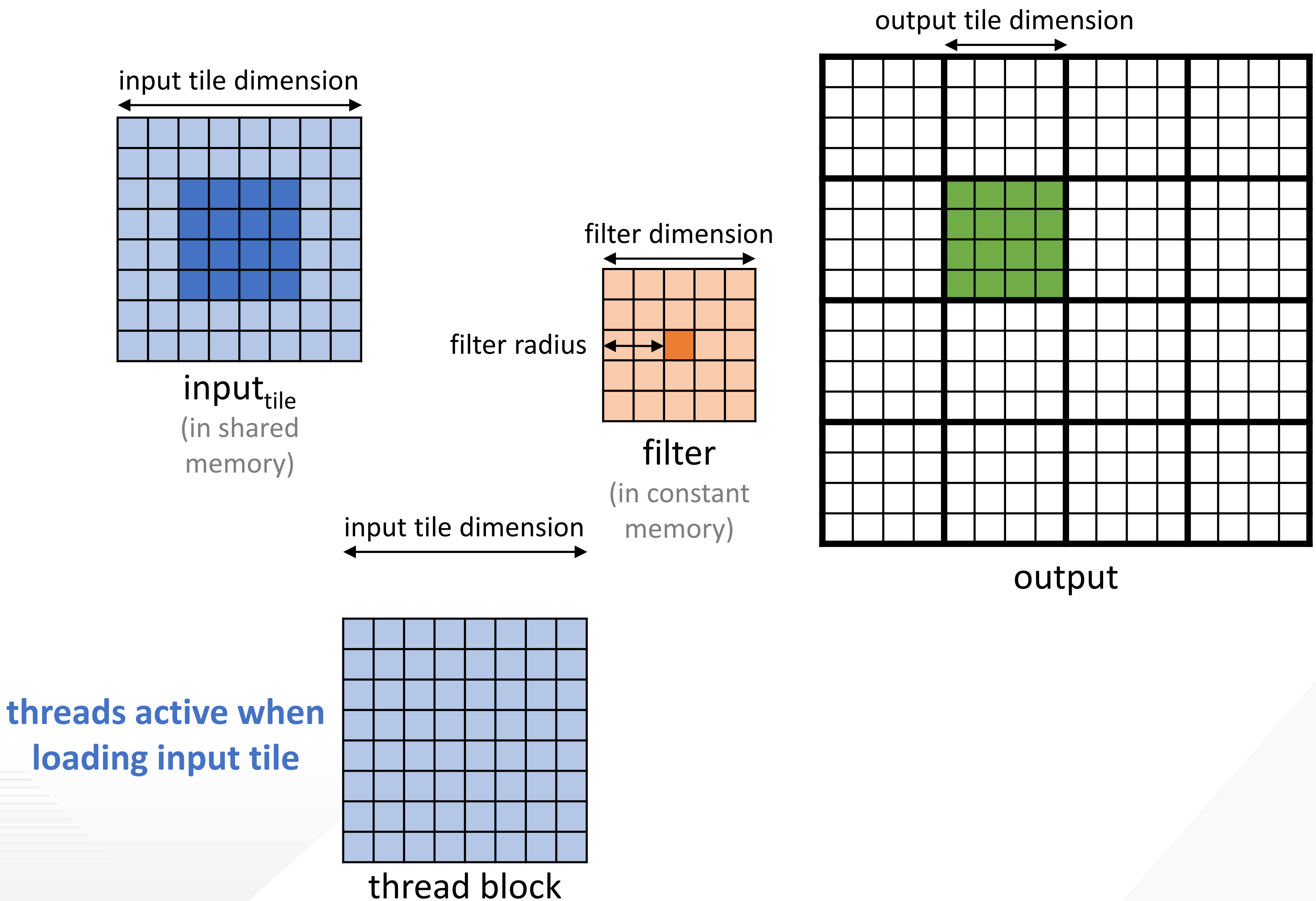
( input tile dimension = output tile dimension + 2 × filter radius )

**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

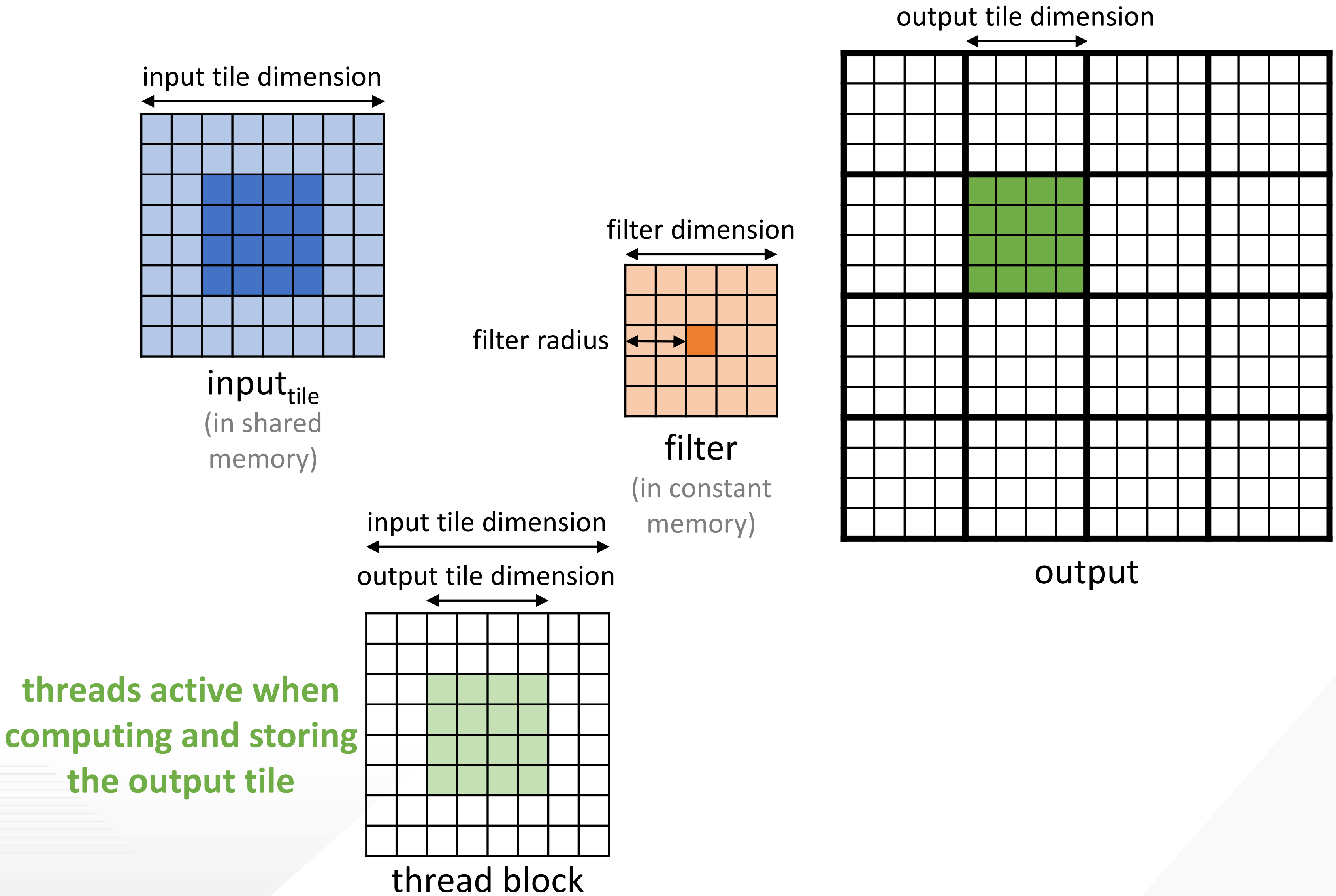
# Difference in Tile Sizes



# Difference in Tile Sizes



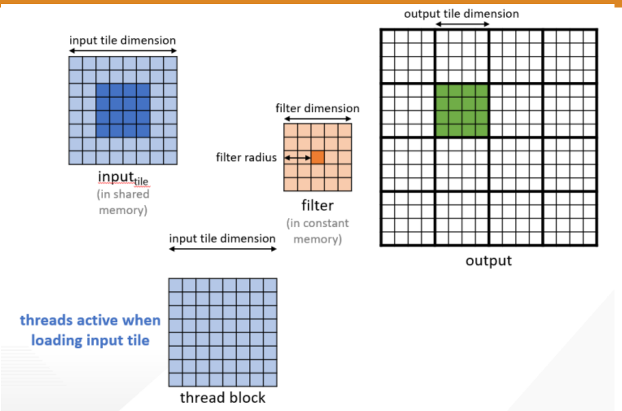
# Difference in Tile Sizes



# Convolution with Tiling

- A tiled 2D convolution kernel

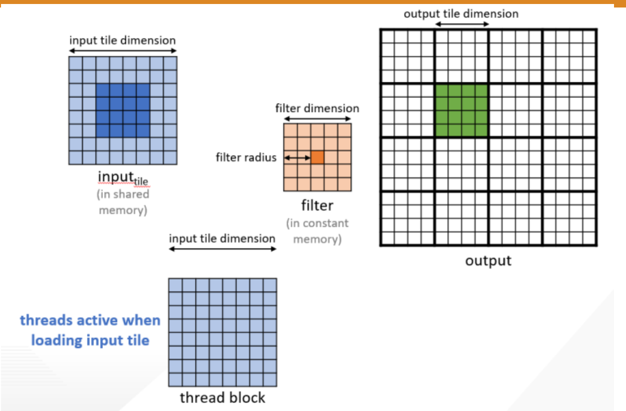
```
#define IN_TILE_DIM 32
#define OUT_TILE_DIM ((IN_TILE_DIM) - 2*(FILTER_RADIUS))
__constant__ float F[2*FILTER_RADIUS+1][2*FILTER_RADIUS+1];
__global__ void convolution_tiled_2D_const_mem_kernel(float *N, float *P,
                                                      int width, int height) {
  int col = blockIdx.x*OUT_TILE_DIM + threadIdx.x - FILTER_RADIUS;
  int row = blockIdx.y*OUT_TILE_DIM + threadIdx.y - FILTER_RADIUS;
  //loading input tile
    __shared__ N_s[IN_TILE_DIM][IN_TILE_DIM];
  if(row>=0 && row<height && col>=0 && col<width) {
    N_s[threadIdx.y][threadIdx.x] = N[row*width + col];
  } else {
    N_s[threadIdx.y][threadIdx.x] = 0.0;
  }
    __syncthreads();
  // Calculating output elements
  int tileCol = threadIdx.x - FILTER_RADIUS;
  int tileRow = threadIdx.y - FILTER_RADIUS;
  // turning off the threads at the edges of the block
  if (col >= 0 && col < width && row >=0 && row < height) {
    if (tileCol>=0 && tileCol<OUT_TILE_DIM && tileRow>=0
                    && tileRow<OUT_TILE_DIM){
      float Pvalue = 0.0f;
      for (int fRow = 0; fRow < 2*FILTER_RADIUS+1; fRow++) {
        for (int fCol = 0; fCol < 2*FILTER_RADIUS+1; fCol++) {
          Pvalue += F[fRow][fCol]*N_s[tileRow+fRow][tileCol+fCol];
        }
      }
      P[row*width+col] = Pvalue;
    }
  }
}
```

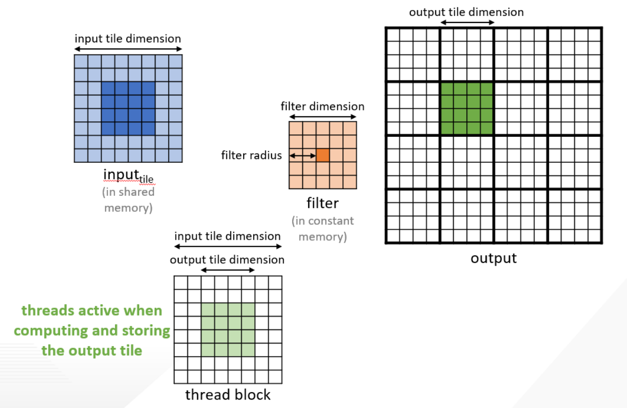


# Convolution with Tiling

- A tiled 2D convolution kernel

```
#define IN_TILE_DIM 32
#define OUT_TILE_DIM ((IN_TILE_DIM) - 2*(FILTER_RADIUS))
__constant__ float F[2*FILTER_RADIUS+1][2*FILTER_RADIUS+1];
__global__ void convolution_tiled_2D_const_mem_kernel(float *N, float *P,
                                                      int width, int height) {
  int col = blockIdx.x*OUT_TILE_DIM + threadIdx.x - FILTER_RADIUS;
  int row = blockIdx.y*OUT_TILE_DIM + threadIdx.y - FILTER_RADIUS;
  //loading input tile
    __shared__ N_s[IN_TILE_DIM][IN_TILE_DIM];
  if(row>=0 && row<height && col>=0 && col<width) {
    N_s[threadIdx.y][threadIdx.x] = N[row*width + col];
  } else {
    N_s[threadIdx.y][threadIdx.x] = 0.0;
  }
    __syncthreads();
  // Calculating output elements
  int tileCol = threadIdx.x - FILTER_RADIUS;
  int tileRow = threadIdx.y - FILTER_RADIUS;
  // turning off the threads at the edges of the block
  if (col >= 0 && col < width && row >=0 && row < height) {
    if (tileCol>=0 && tileCol<OUT_TILE_DIM && tileRow>=0
                 && tileRow<OUT_TILE_DIM){
      float Pvalue = 0.0f;
      for (int fRow = 0; fRow < 2*FILTER_RADIUS+1; fRow++) {
        for (int fCol = 0; fCol < 2*FILTER_RADIUS+1; fCol++) {
          Pvalue += F[fRow][fCol]*N_s[tileRow+fRow][tileCol+fCol];
        }
      }
      P[row*width+col] = Pvalue;
    }
  }
}
```

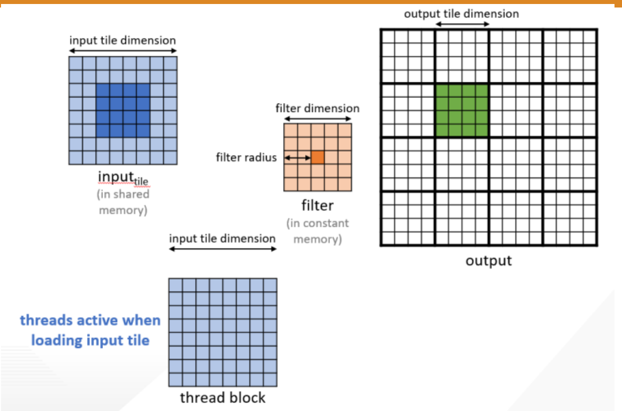


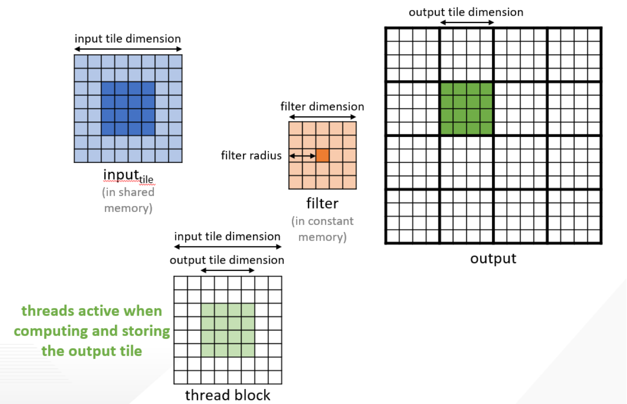



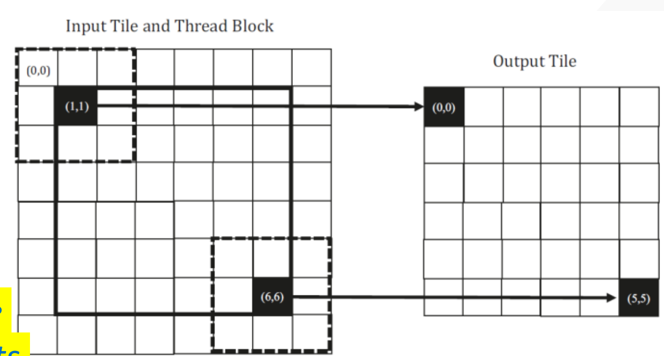
# Convolution with Tiling

- A tiled 2D convolution kernel

```
#define IN_TILE_DIM 32
#define OUT_TILE_DIM ((IN_TILE_DIM) - 2*(FILTER_RADIUS))
__constant__ float F[2*FILTER_RADIUS+1][2*FILTER_RADIUS+1];
__global__ void convolution_tiled_2D_const_mem_kernel(float *N, float *P,
                                                       int width, int height) {
  int col = blockIdx.x*OUT_TILE_DIM + threadIdx.x - FILTER_RADIUS;
  int row = blockIdx.y*OUT_TILE_DIM + threadIdx.y - FILTER_RADIUS;
  //loading input tile
    __shared__ N_s[IN_TILE_DIM][IN_TILE_DIM];
  if(row>=0 && row<height && col>=0 && col<width) {
    N_s[threadIdx.y][threadIdx.x] = N[row*width + col];
  } else {
    N_s[threadIdx.y][threadIdx.x] = 0.0;
  }
    __syncthreads();
  // Calculating output elements
  int tileCol = threadIdx.x - FILTER_RADIUS;
  int tileRow = threadIdx.y - FILTER_RADIUS;
  // turning off the threads at the edges of the block
  if (col >= 0 && col < width && row >=0 && row < height) {
    if (tileCol>=0 && tileCol<OUT_TILE_DIM && tileRow>=0
                 && tileRow<OUT_TILE_DIM){
      float Pvalue = 0.0f;
      for (int fRow = 0; fRow < 2*FILTER_RADIUS+1; fRow++) {
        for (int fCol = 0; fCol < 2*FILTER_RADIUS+1; fCol++) {
          Pvalue += F[fRow][fCol]*N_s[tileRow+fRow][tileCol+fCol];
        }
      }
      P[row*width+col] = Pvalue;
    }
  }
}
```



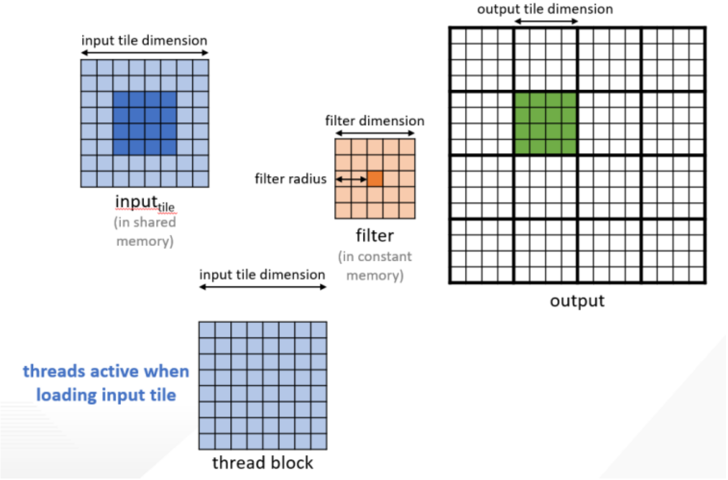

*A small example that illustrates the thread organization for using the input tile elements in the shared memory to calculate the output tile elements.*

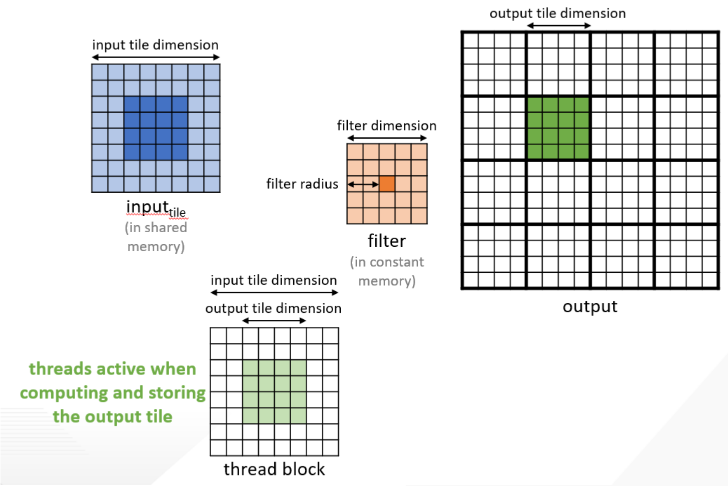
# Arithmetic to Global Memory Access Ratio

- With tiling:
  - Considering output tile dimension:
    - Input = IN_TILE_DIM
    - Output = OUT_TILE_DIM
  - Global loads per block: **$IN\_TILE\_DIM^2*4 = (OUT\_TILE\_DIM+2*FILTER\_RADIUS)^2*4$**
    - Each thread that is assigned to an input tile element loads one 4-byte input value.
  - Operations per block: **$OUT\_TILE\_DIM^2*(2*FILTER\_RADIUS+1)^2*2$**
    - Every thread that is assigned to an output tile element, performs 1 multiplication and 1 addition for every element of the filter
  - Ratio:

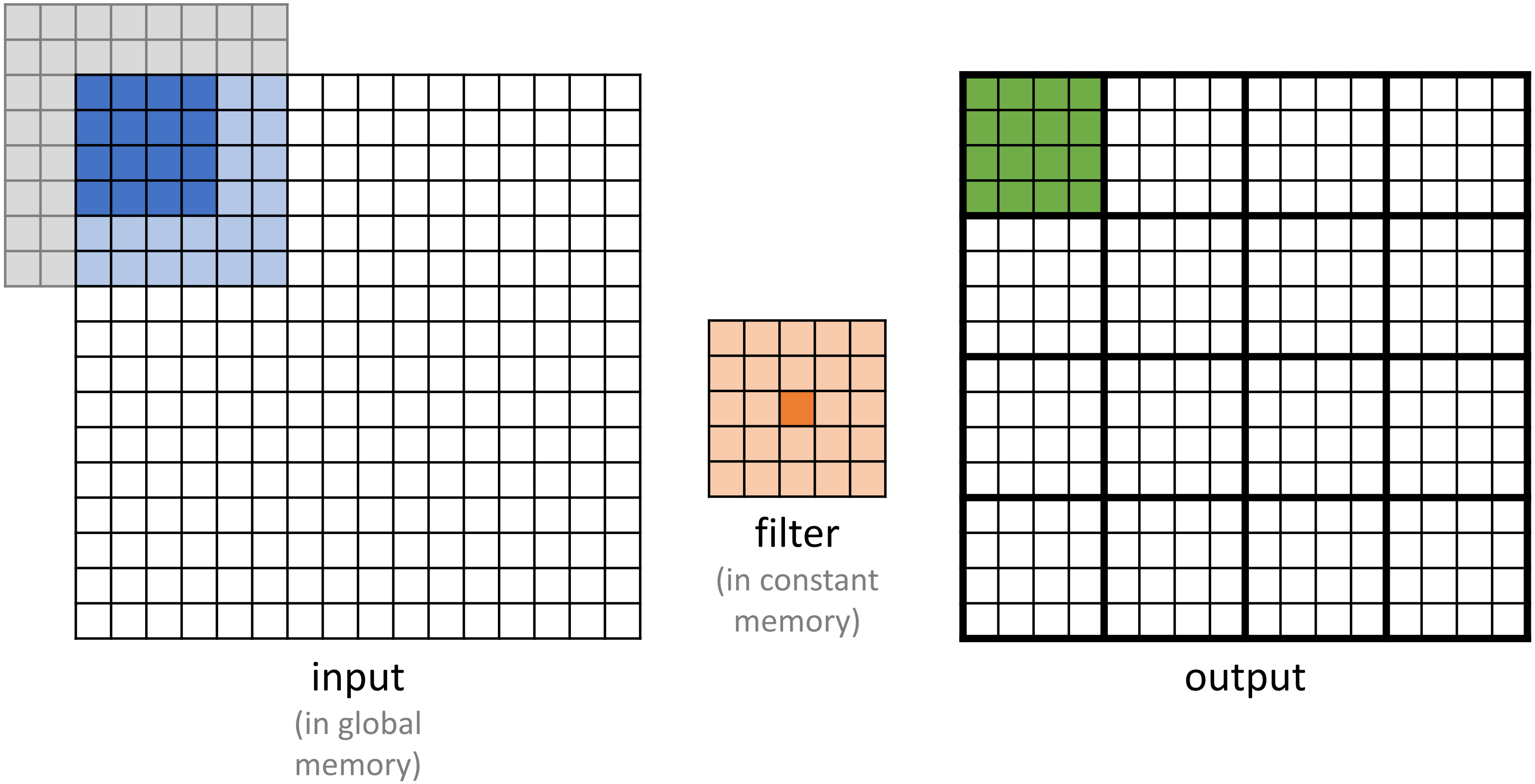
$$\frac{OUT\_TILE\_DIM^2*(2*FILTER\_RADIUS+1)^2*2}{(OUT\_TILE\_DIM+2*FILTER\_RADIUS)^2*4}$$

- For example, when FILTER_RADIUS =2 and OUT_TILE_DIM=28, the ratio is 9.57 OP/B (≈19× improvement!)



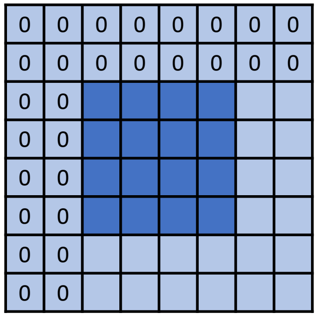



# Boundary Conditions

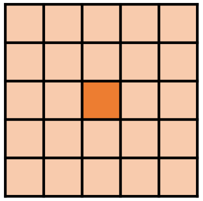



Threads computing output elements at the boundary access input elements that are out of bounds (also called *ghost* elements)

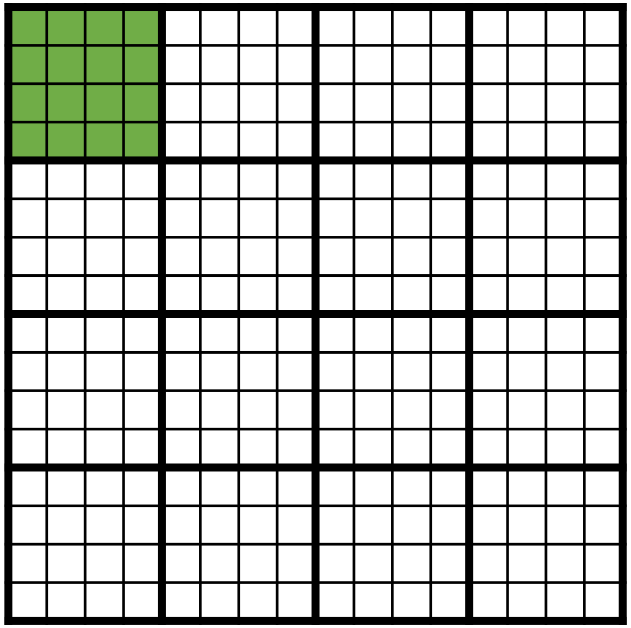
# Boundary Conditions




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)

**<u>Solution:</u>** Store zero to shared memory tile for our of bounds input elements

- Wen-mei W. Hwu, David B. Kirk, and Izzat El Hajj. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2022.