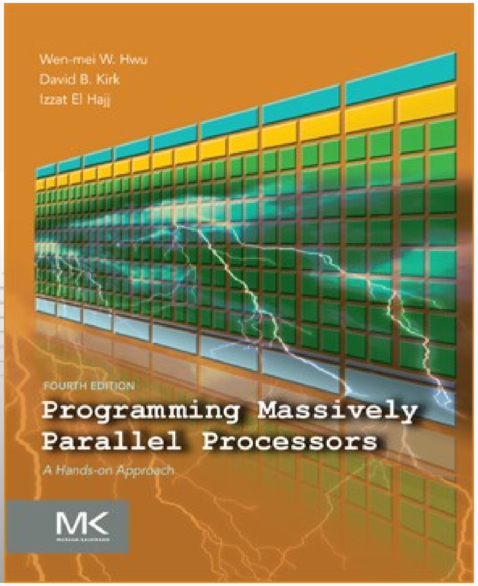
Programming Massively Parallel Processors

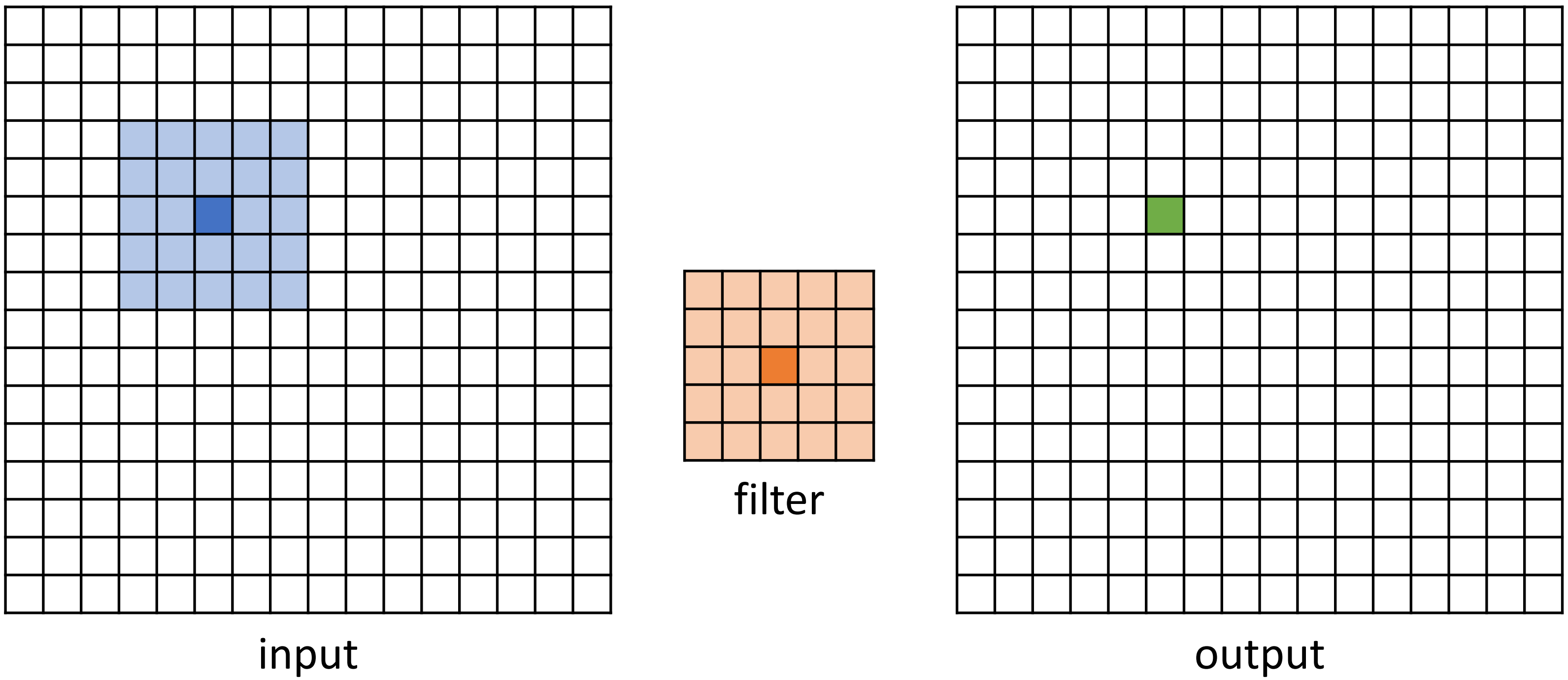
A Hands-on Approach

CHAPTER 7 › Convolution



Copyright © 2022 Elsevier

Convolution



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

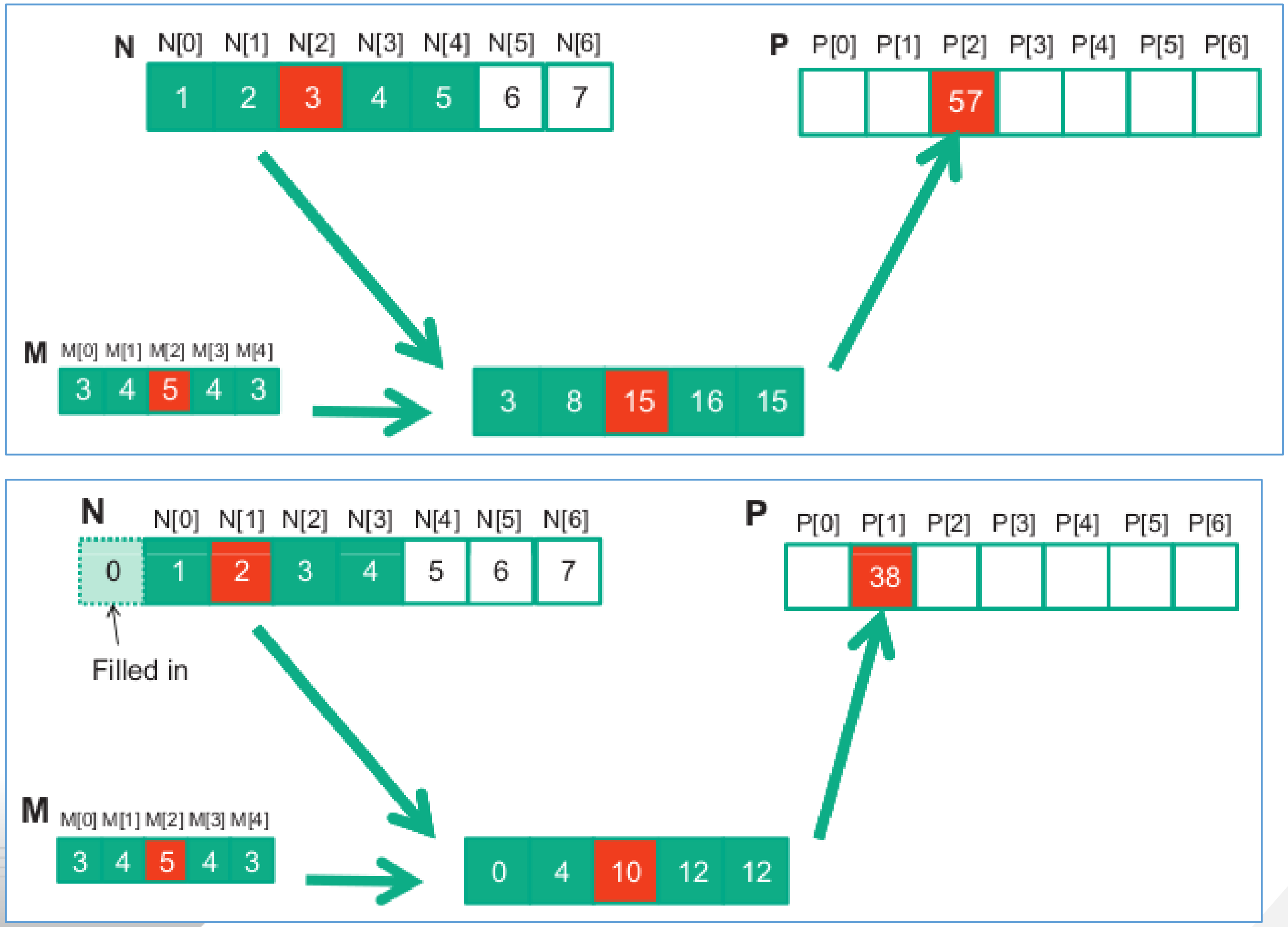
Image blur seen before was a special case where all weights are the same

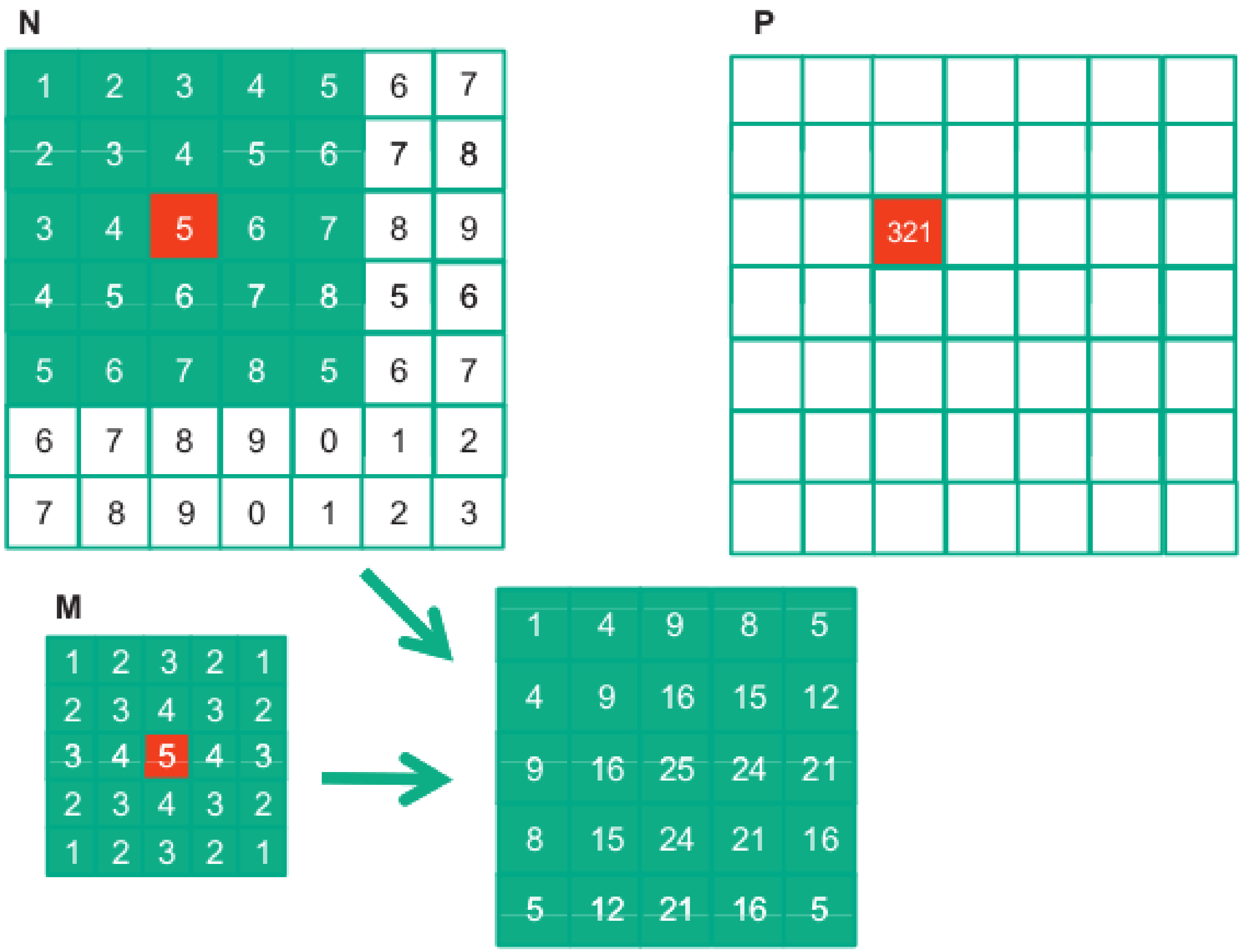
In general, weights are determined by a convolution **filter**

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

Example

- Convolution is an array operation where each output data element is a weighted sum of a collection of neighboring input elements.



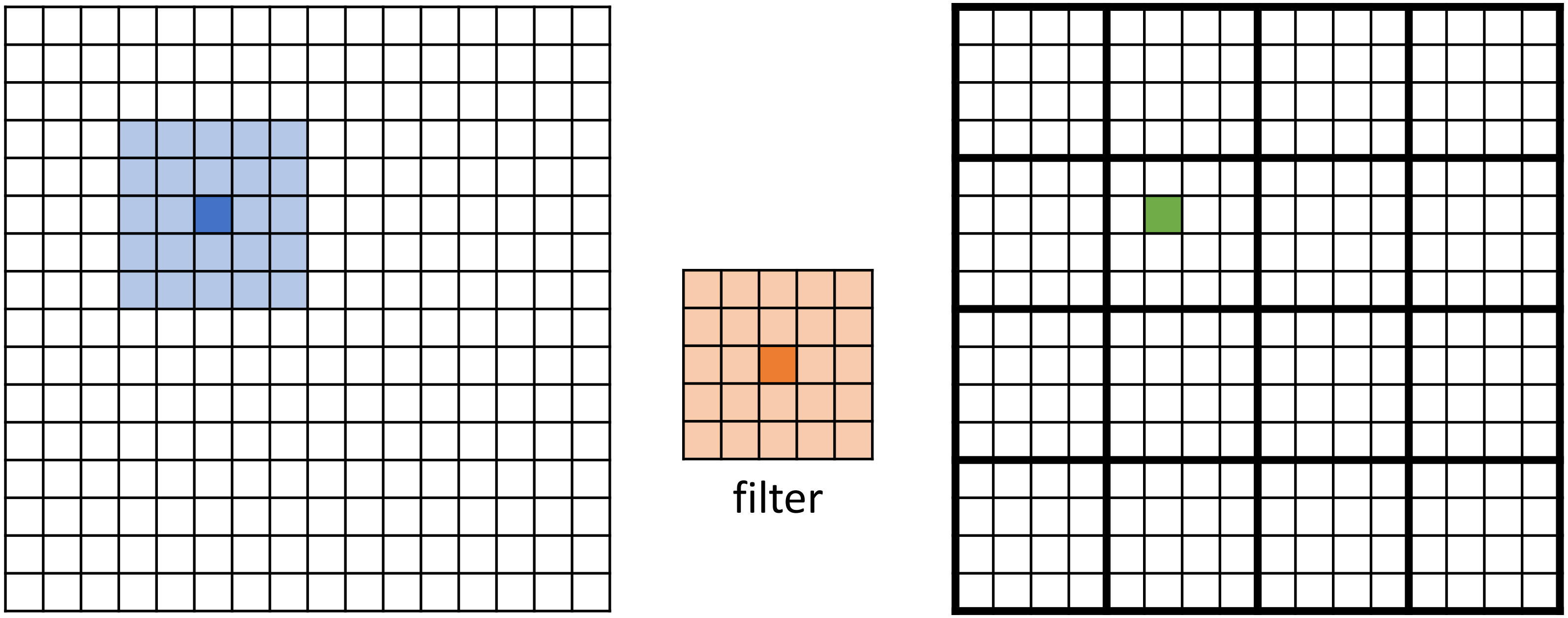

N
1 2 3 4 5 6 7
2 3 4 5 6 7 8
3 4 5 6 7 8 9
4 5 6 7 8 5 6
5 6 7 8 5 6 7
6 7 8 9 0 1 2
7 8 9 0 1 2 3
P
321
M
1 2 3 2 1
2 3 4 3 2
3 4 5 4 3
2 3 4 3 2
1 2 3 2 1
1 4 9 8 5
4 9 16 15 12
9 16 25 24 21
8 15 24 21 16
5 12 21 16 5

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



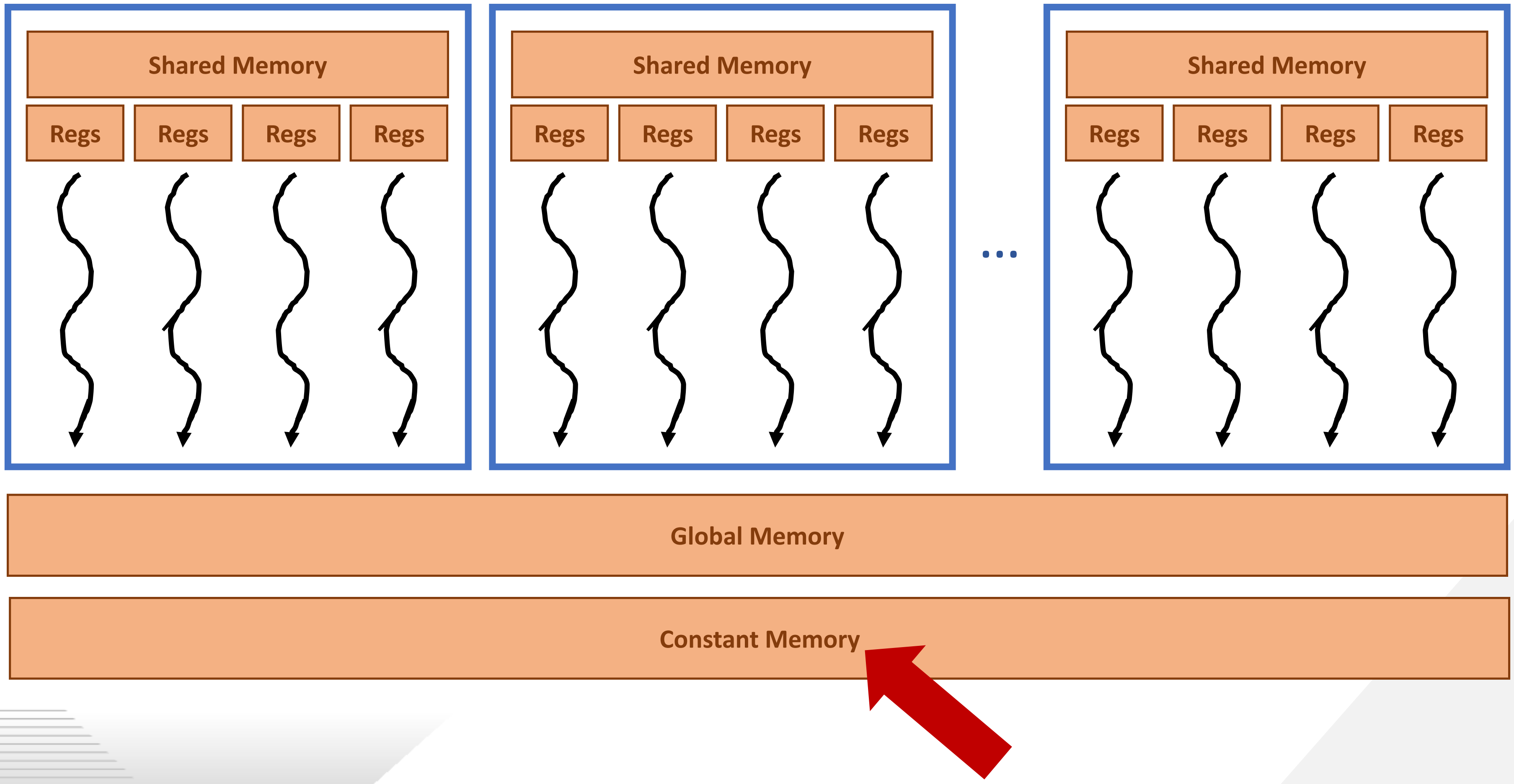
<u>Parallelization approach:</u> Assign one thread to compute each **output element** by looping over **input elements** and **filter** weights

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

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

Recall: Memory in the CUDA Programming Model



Using Constant Memory

- Declare constant memory array as global variable

```
__constant__ float filter_c[FILTER_DIM][FILTER_DIM];
```

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

```
cudaMemcpyToSymbol(filter_c, filter, FILTER_DIM*FILTER_DIM*sizeof(float));
```

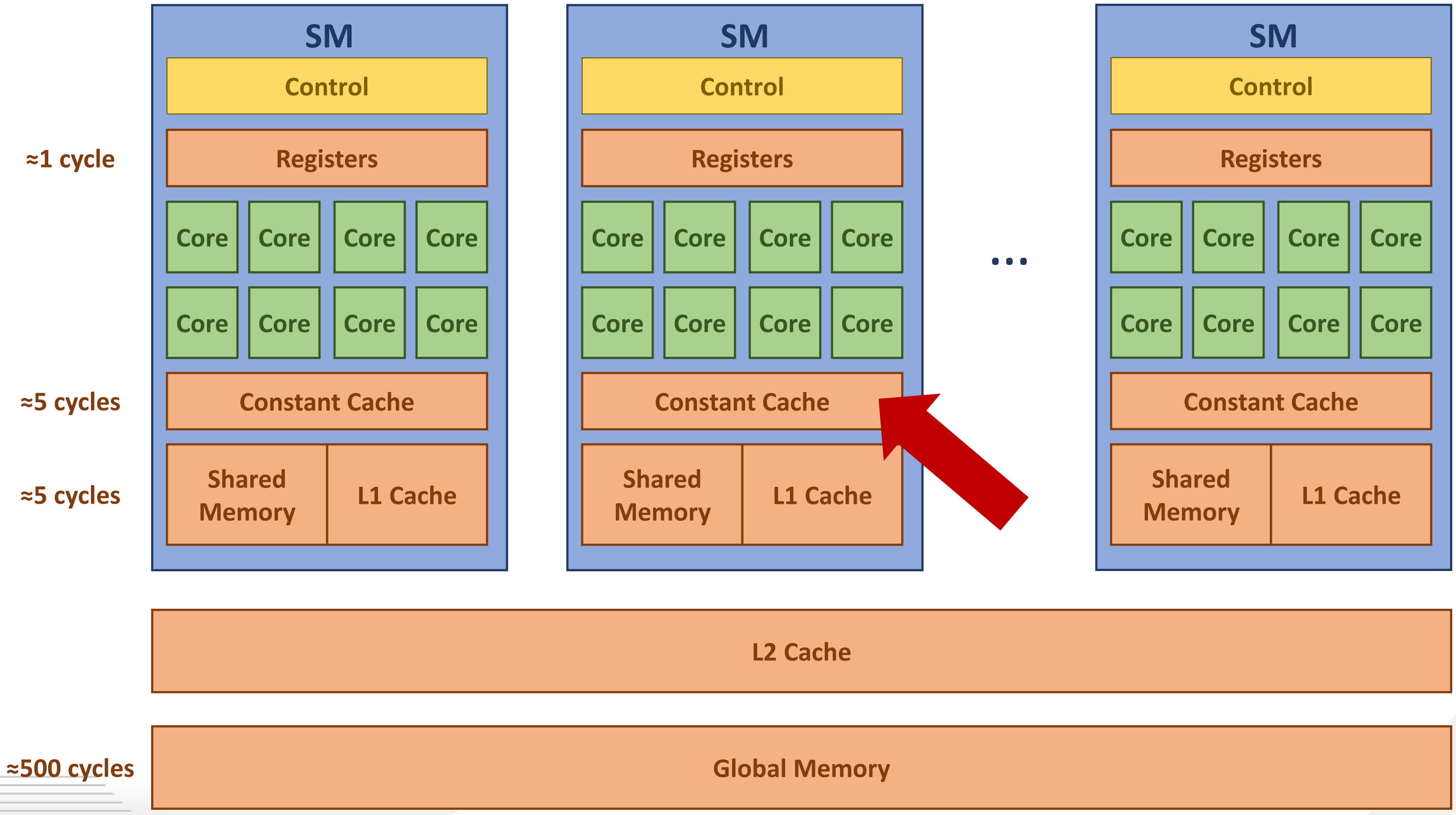
- Can only allocate up to 64KB
  - Otherwise, input is also constant, but it is too large to put in constant memory

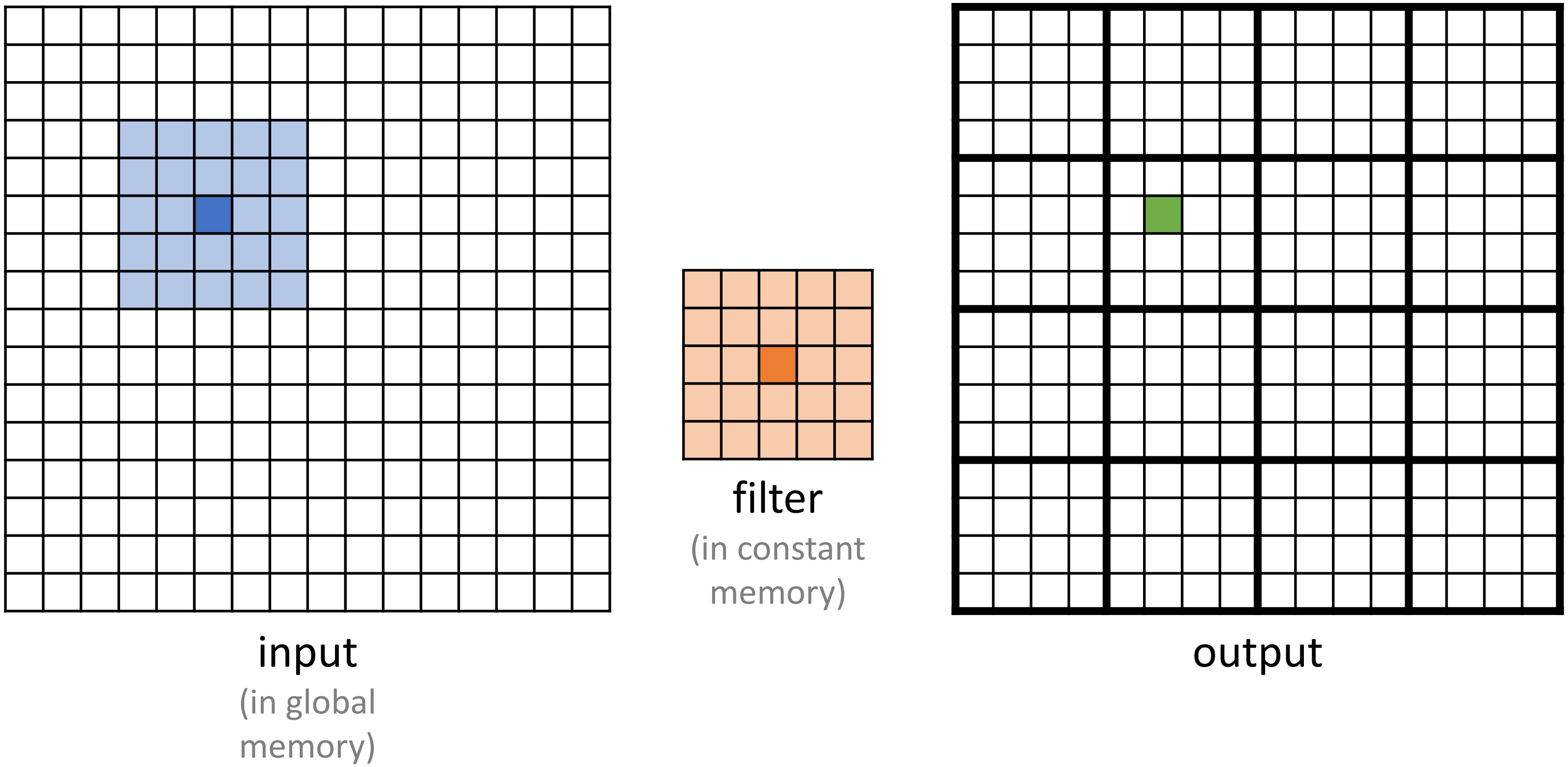
Motivation for Constant Memory

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

- Small size: minimize evictions
  - Cache for constant memory has low miss rate

Recall: Memory in the GPU Architecture



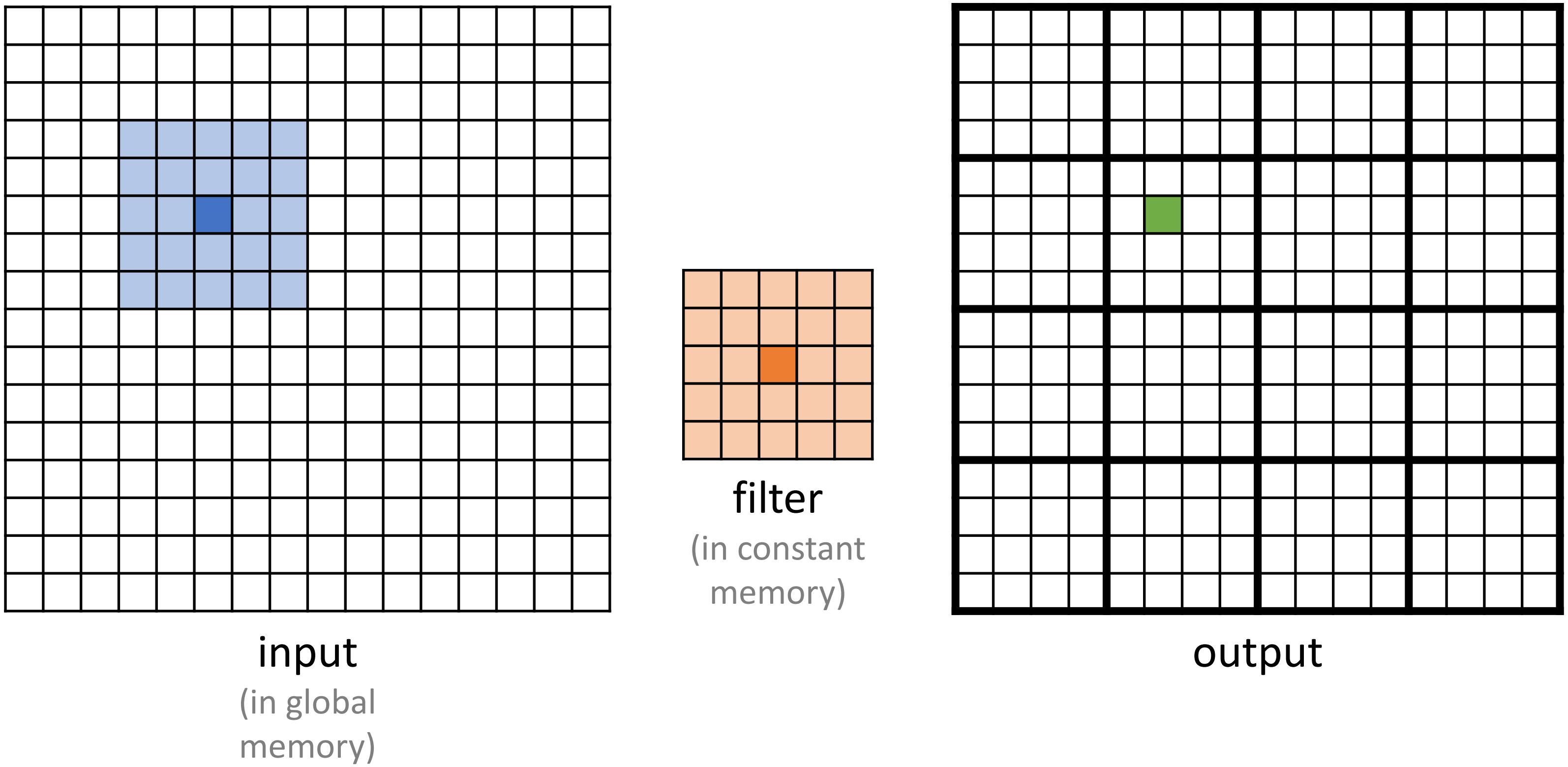


Parallelization approach: Assign one thread to compute each output element by looping over input elements and filter weights

Convolution Kernel Code

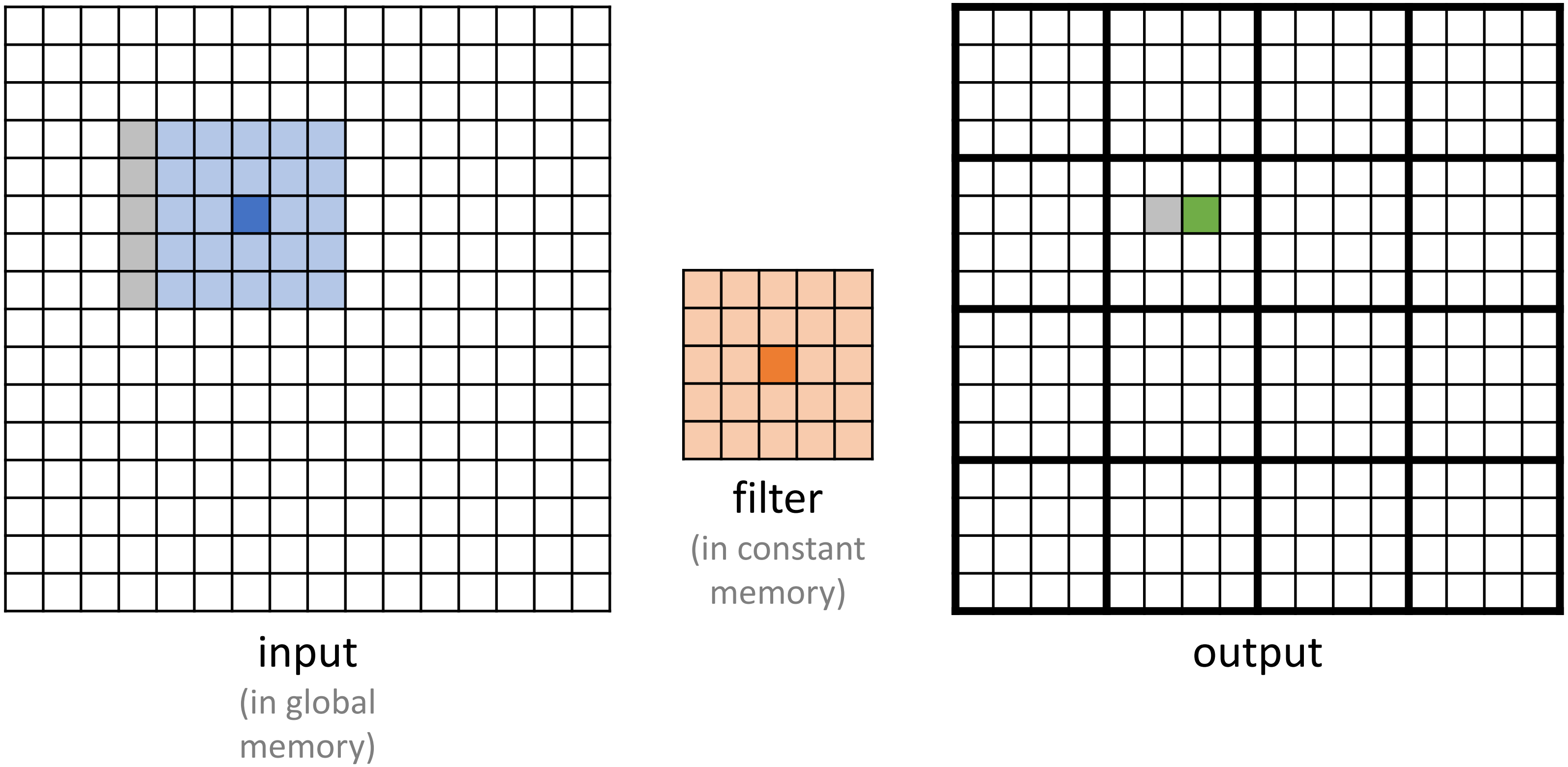
```
__global__ void convolution_kernel(float* input, float* output, unsigned int width,
                                                               unsigned int height) {
    int outRow = blockIdx.y*blockDim.y + threadIdx.y;
    int outCol = blockIdx.x*blockDim.x + threadIdx.x;
    if (outRow < height && outCol < width) {
        float sum = 0.0f;
        for(int filterRow = 0; filterRow < FILTER_DIM; ++filterRow) {
            for(int filterCol = 0; filterCol < FILTER_DIM; ++filterCol) {
                int inRow = outRow - FILTER_RADIUS + filterRow;
                int inCol = outCol - FILTER_RADIUS + filterCol;
                if(inRow >= 0 && inRow < height && inCol >= 0 && inCol < width) {
                    sum += filter_c[filterRow][filterCol]*input[inRow*width + inCol];
                }
            }
        }
        output[outRow*width + outCol] = sum;
    }

}
```

Data Reuse in Convolution



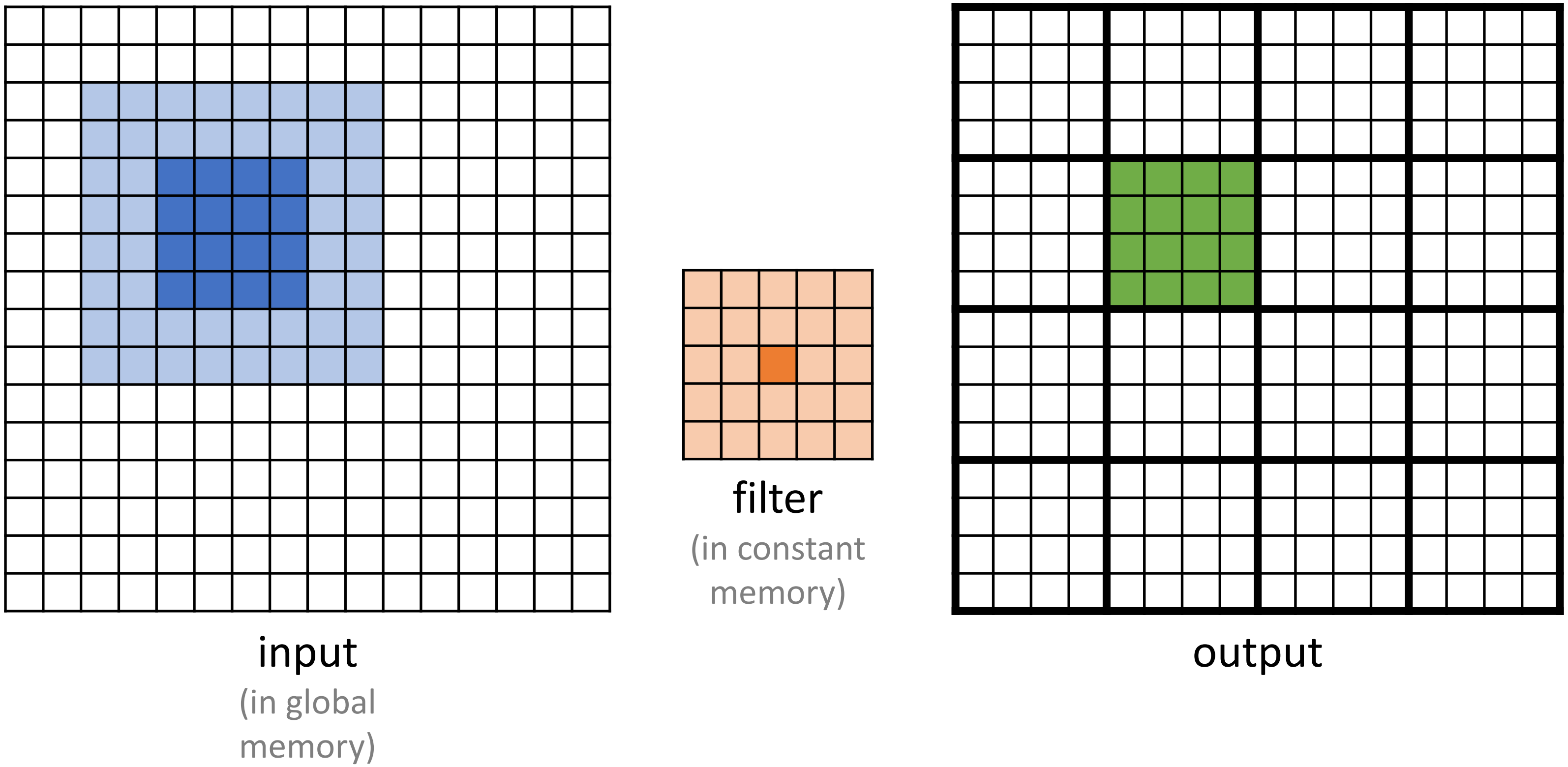
Observation: Threads in the same block load some of the same input elements

Data Reuse in Convolution



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

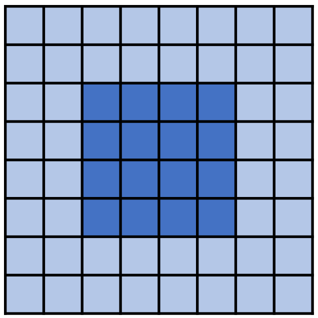
Data Reuse in Convolution



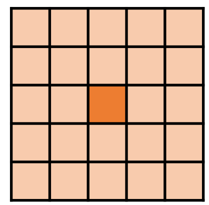
Observation: Threads in the same block load some of the same input elements

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

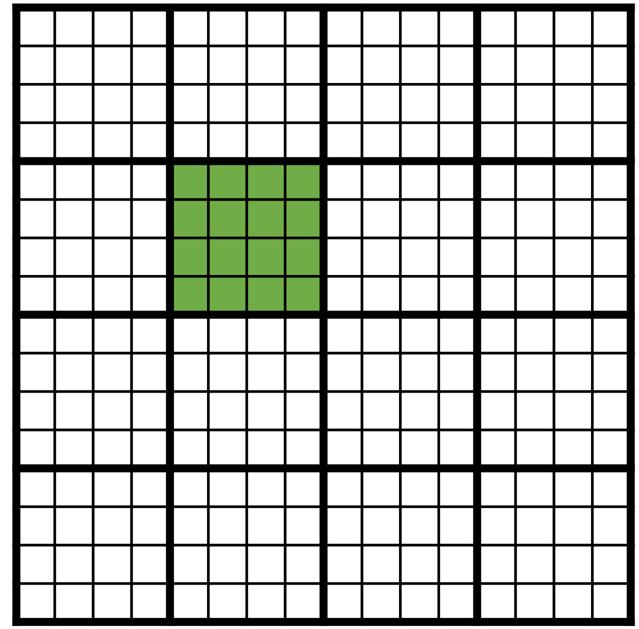
Convolution with Tiling



$input_{tile}$
(in shared memory)



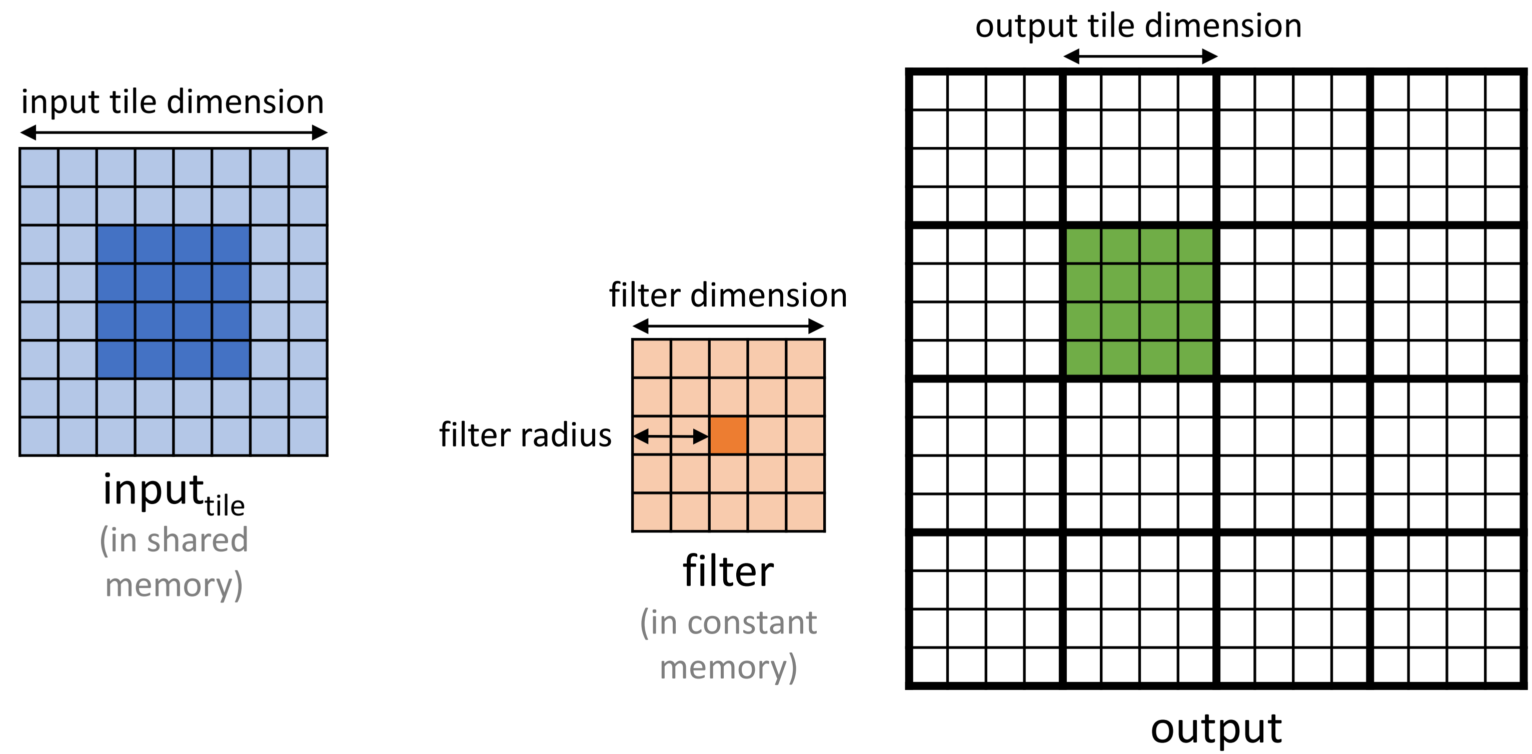
filter
(in constant memory)



output

<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

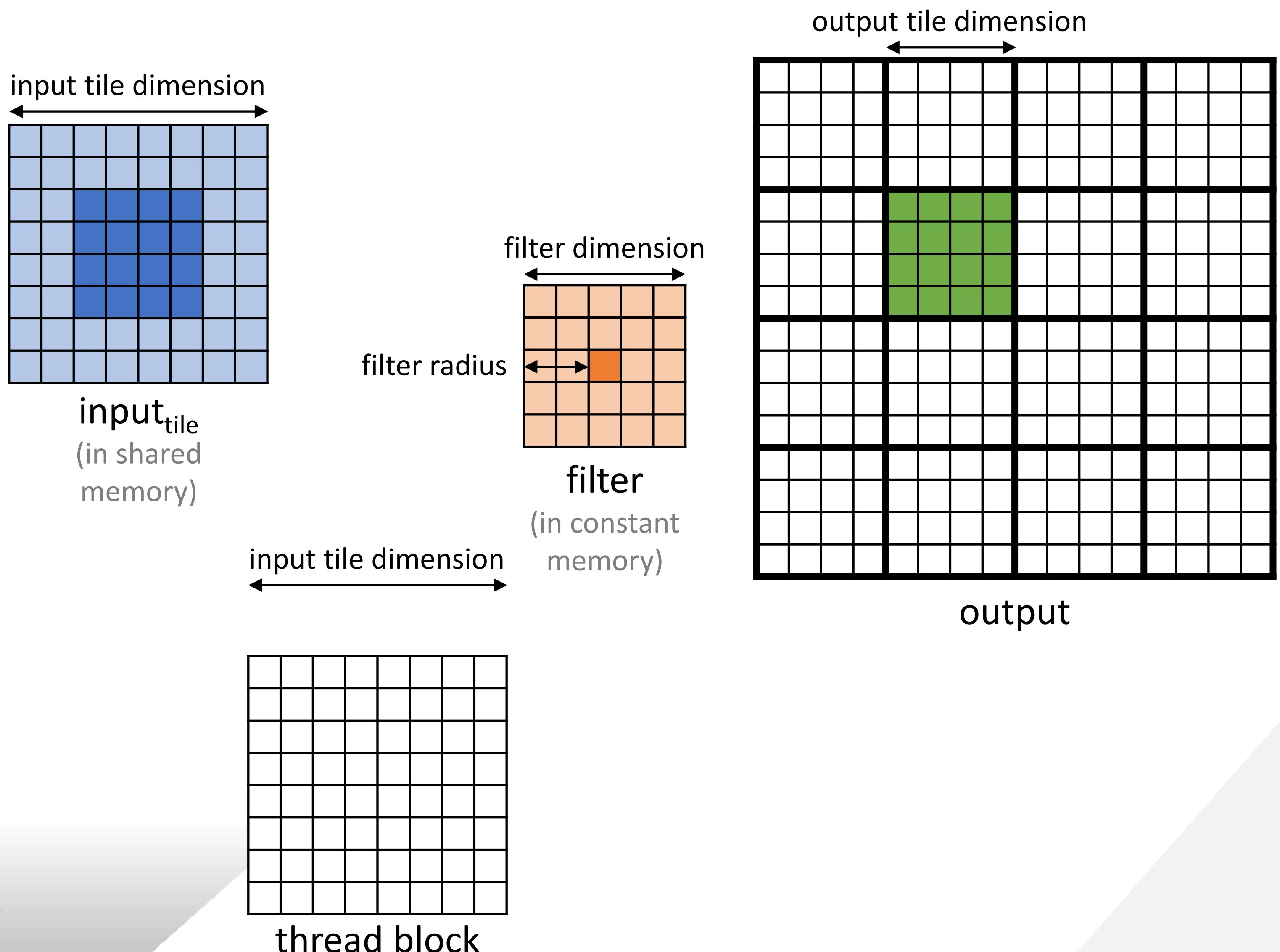


Challenge: Input and output tiles have different dimensions

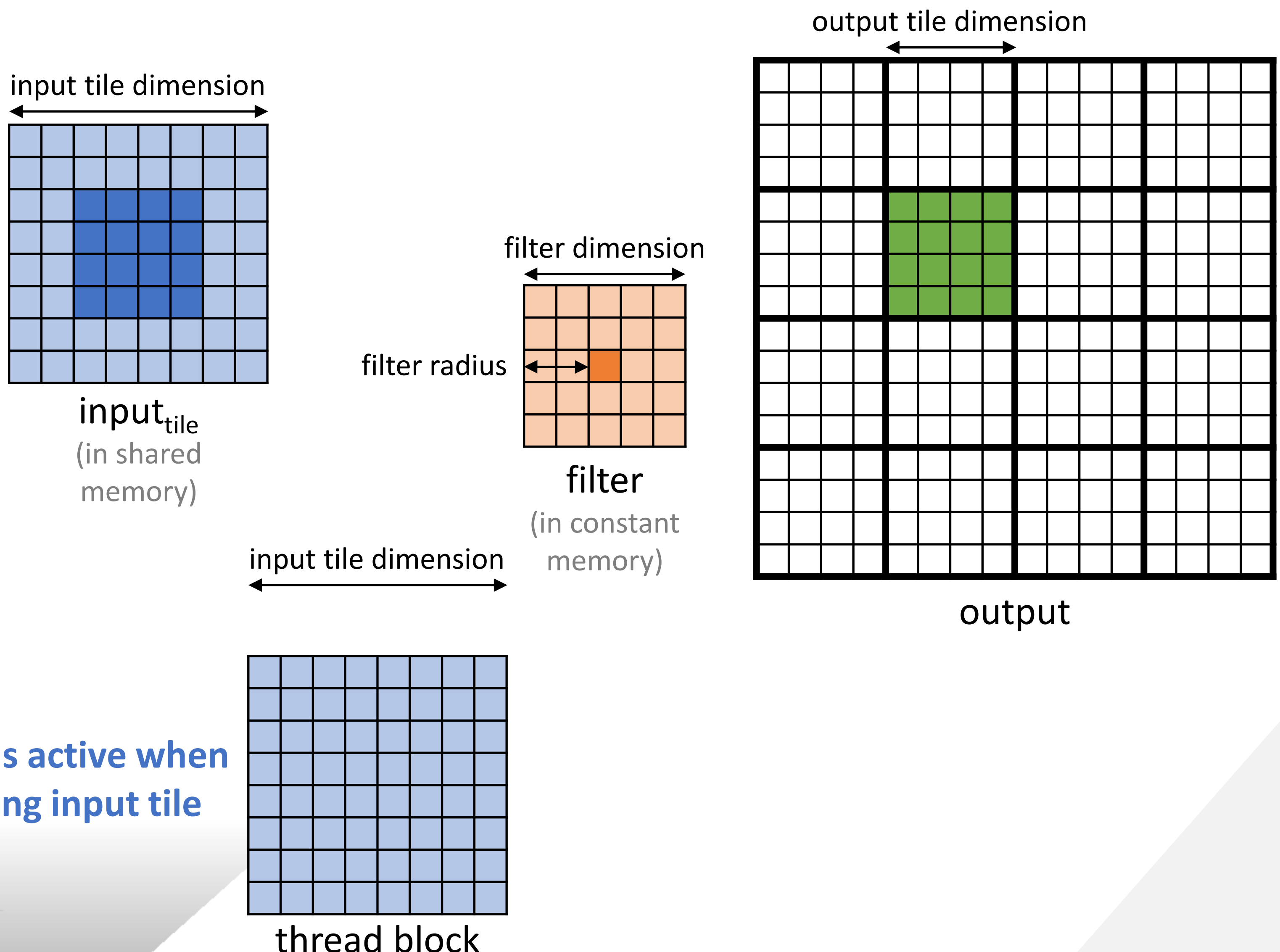
(input tile dimension = output tile dimension + 2 × filter radius)

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

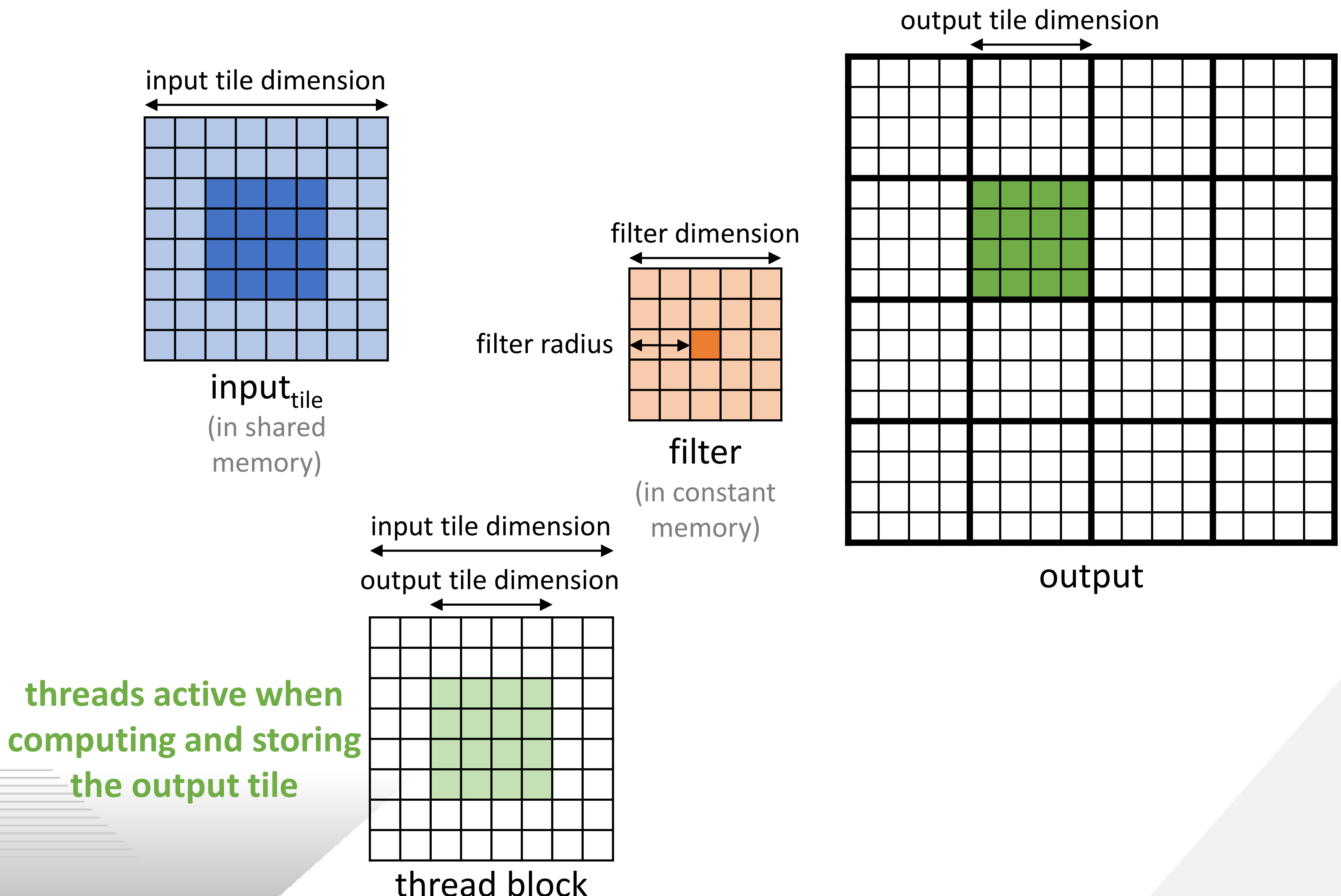


Difference in Tile Sizes



threads active when loading input tile

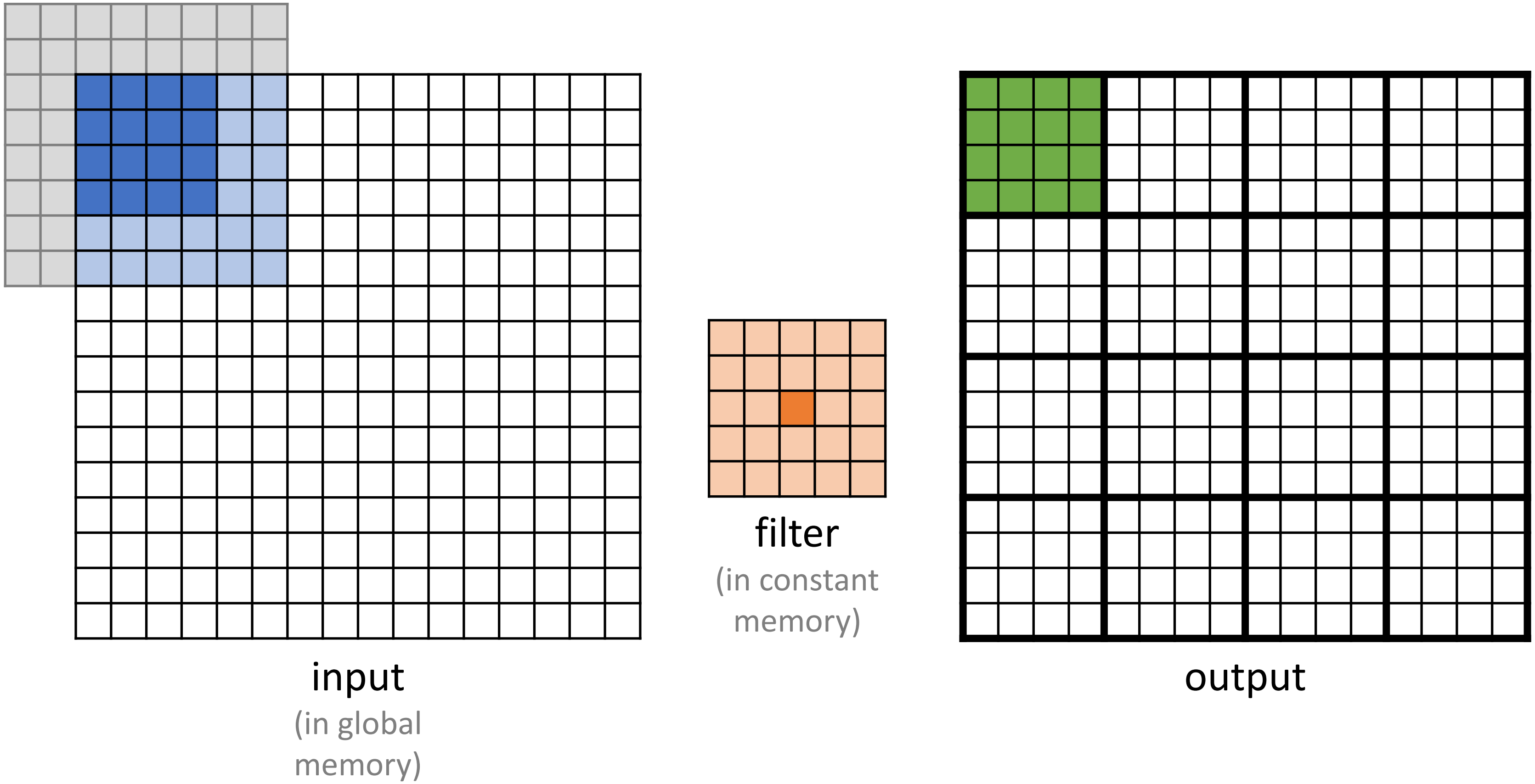
Difference in Tile Sizes



Arithmetic to Global Memory Access Ratio

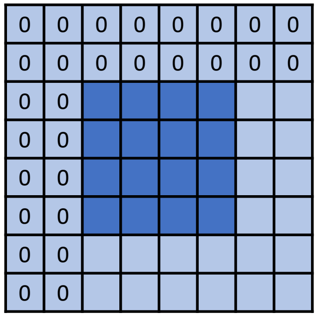
- Considering an M×M filter
- Without tiling:
  - Operations per thread: M^2 adds + M^2 muls = $2M^2$ OP
  - Global loads per thread: $M^2 \times 4$ B = $4M^2$ B
  - Ratio: $(2M^2$ OP$)/(4M^2$ B$) = 0.5$ OP/B
- With tiling:
  - Considering tile dimensions: input = T, output = T-M+1
  - Operations per block: $(T-M+1)^2 \times 2M^2$ OP
  - Global loads per block: $T^2 \times 4$ B
  - Ratio: $((T-M+1)^2 \times 2M^2$ OP$)/(T^2 \times 4$ B$) = 0.5M^2(1 - (M-1)/T)^2$
    - For M=5 and T=32: 9.57 OP/B (≈19× improvement!)

Boundary Conditions

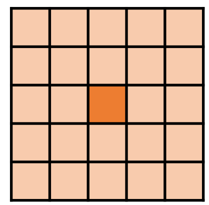


Threads computing output elements at the boundary access input elements that are out of bounds (also called *ghost* elements)

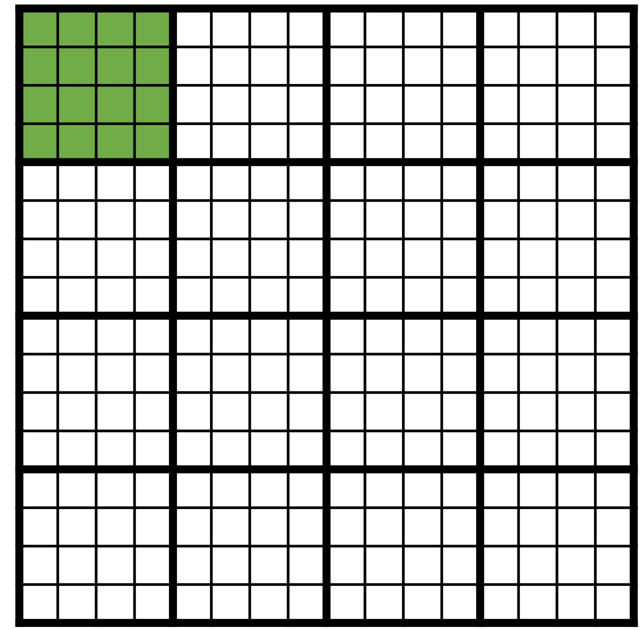
Boundary Conditions



$input_{tile}$
(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)

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

References

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