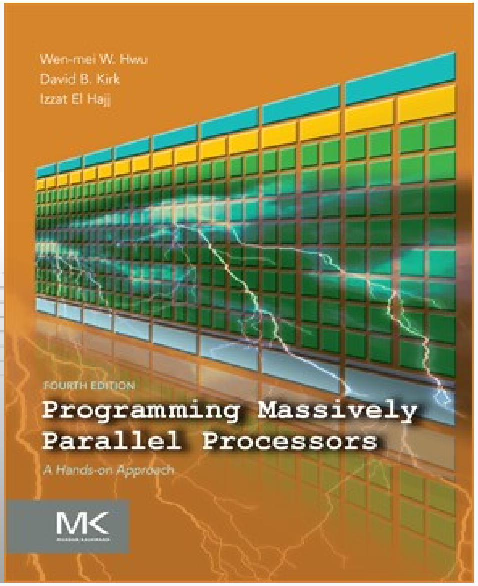
Programming Massively Parallel Processors

A Hands-on Approach

CHAPTER 8 › Stencil

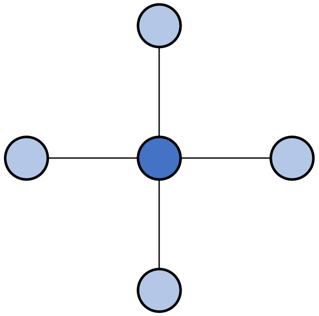


Copyright © 2022 Elsevier

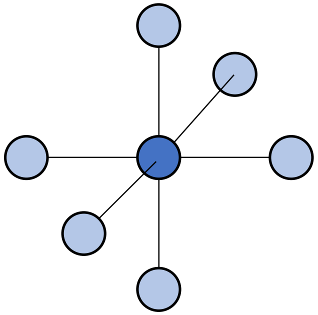
Stencil

- The **stencil** computation pattern refers to a class of computations on a grid where the value at a grid point is computed based on neighboring points
  - Typically used in solving partial differential equations in domains such as fluid dynamics, heat conductance, combustion, weather forecasting, etc.

- Example:



5-point stencil (2D)



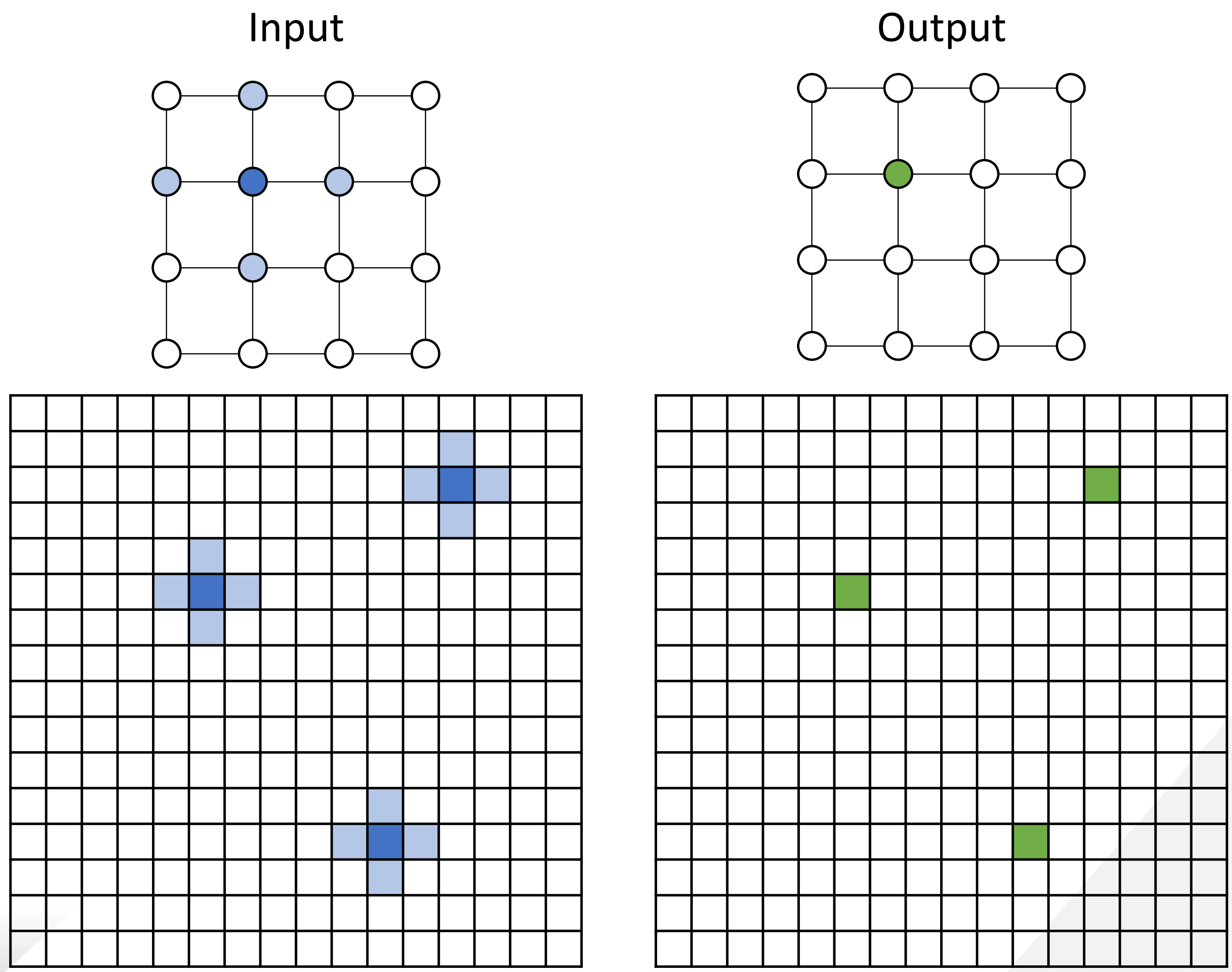
7-point stencil (3D)

Copyright © 2022 Elsevier

Stencil (5-point, 2D)

Grid of points:
(4x4 grid shown)

Stored as 2D array:
(16x16 grid shown)



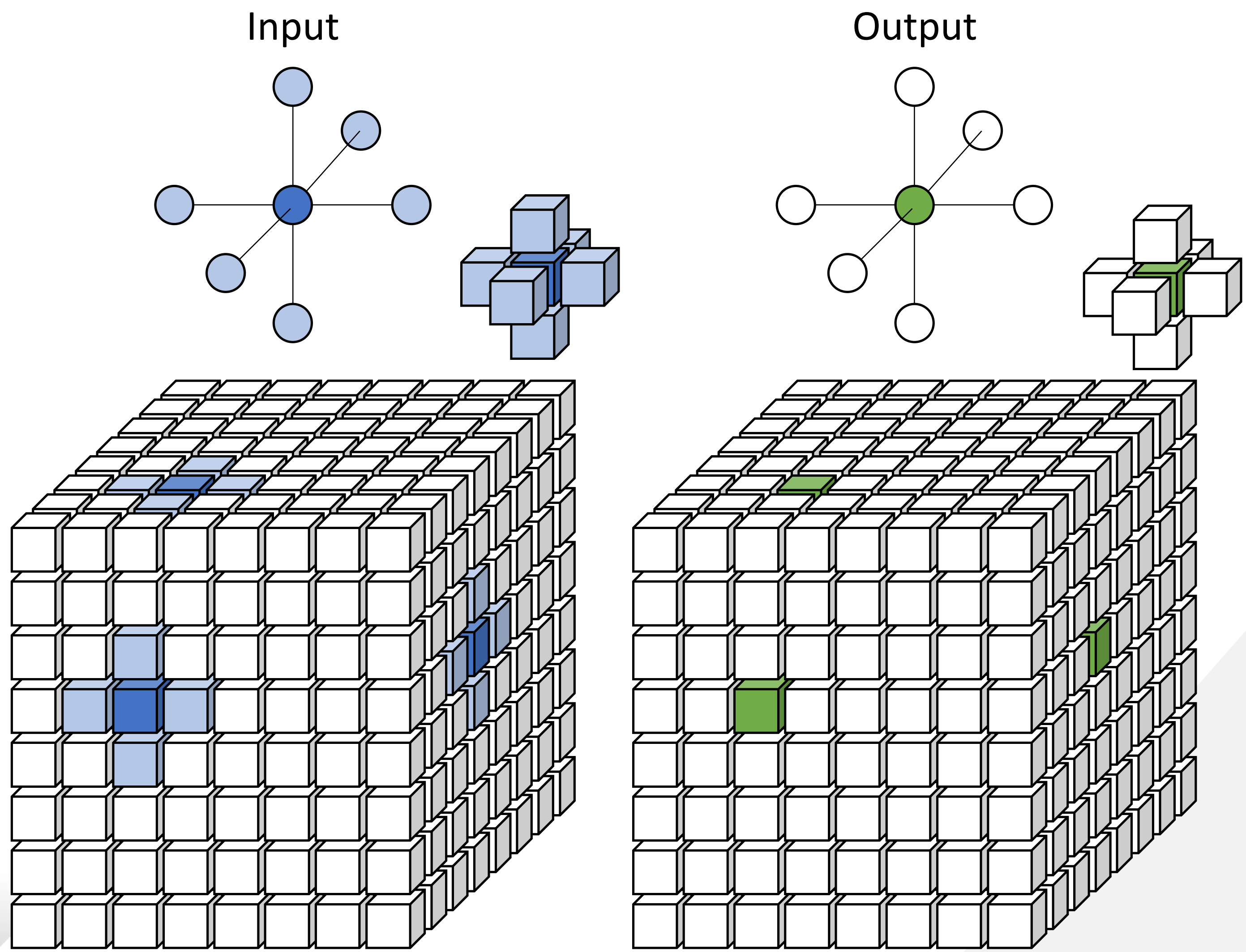
The **output value** is computed based on the **corresponding and neighboring input values**

Copyright © 2022 Elsevier

Stencil (7-point, 3D)

Grid of points:
(one stencil shown)

Stored as 3D array:
(8x8x8 grid shown)



The **output value** is computed based on the **corresponding and neighboring input values**

Copyright © 2022 Elsevier

Parallelizing Stencil (2D)

Input

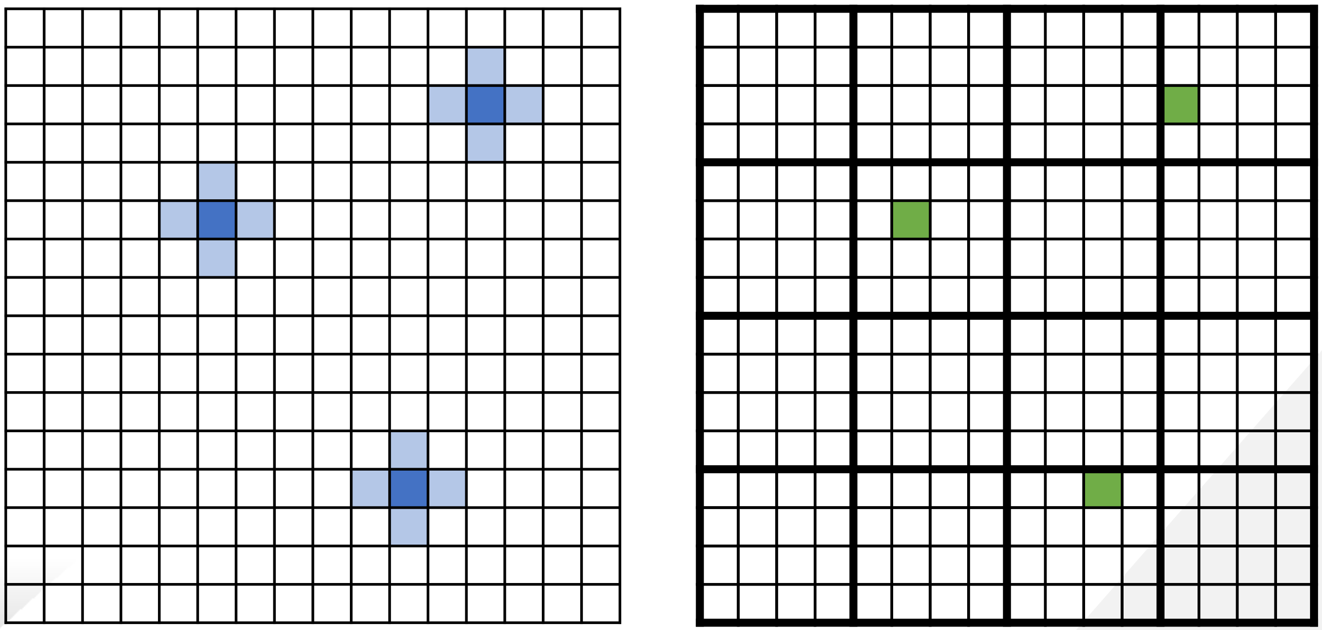
Output

Grid of points:

(4x4 grid shown)

Stored as 2D array:

(16x16 grid shown)



Parallelization Approach:

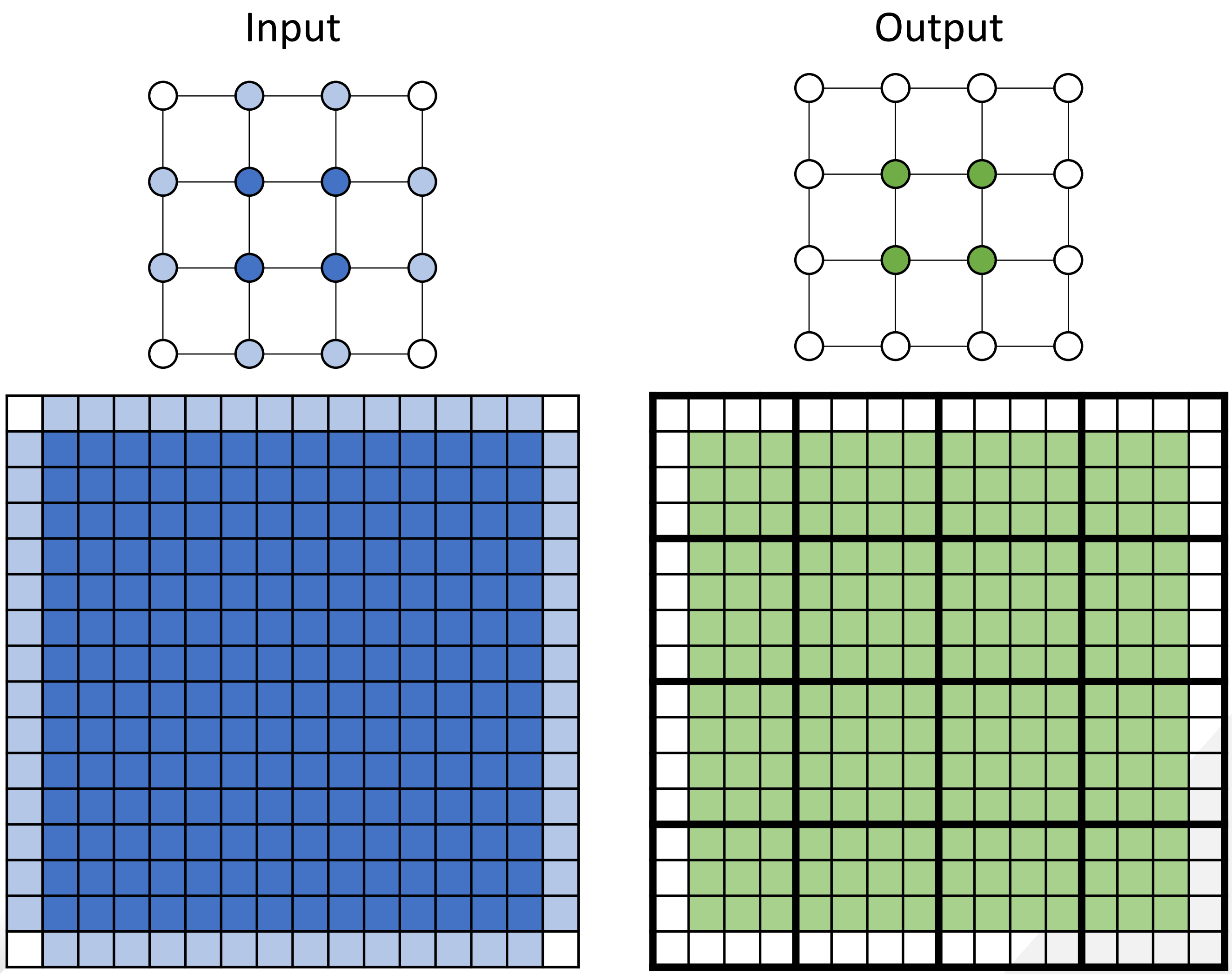
Assign one thread per output grid point (use 2D grid of threads)

Copyright © 2022 Elsevier

Boundary Conditions

Grid of points:
(4x4 grid shown)

Stored as 2D array:
(16x16 grid shown)



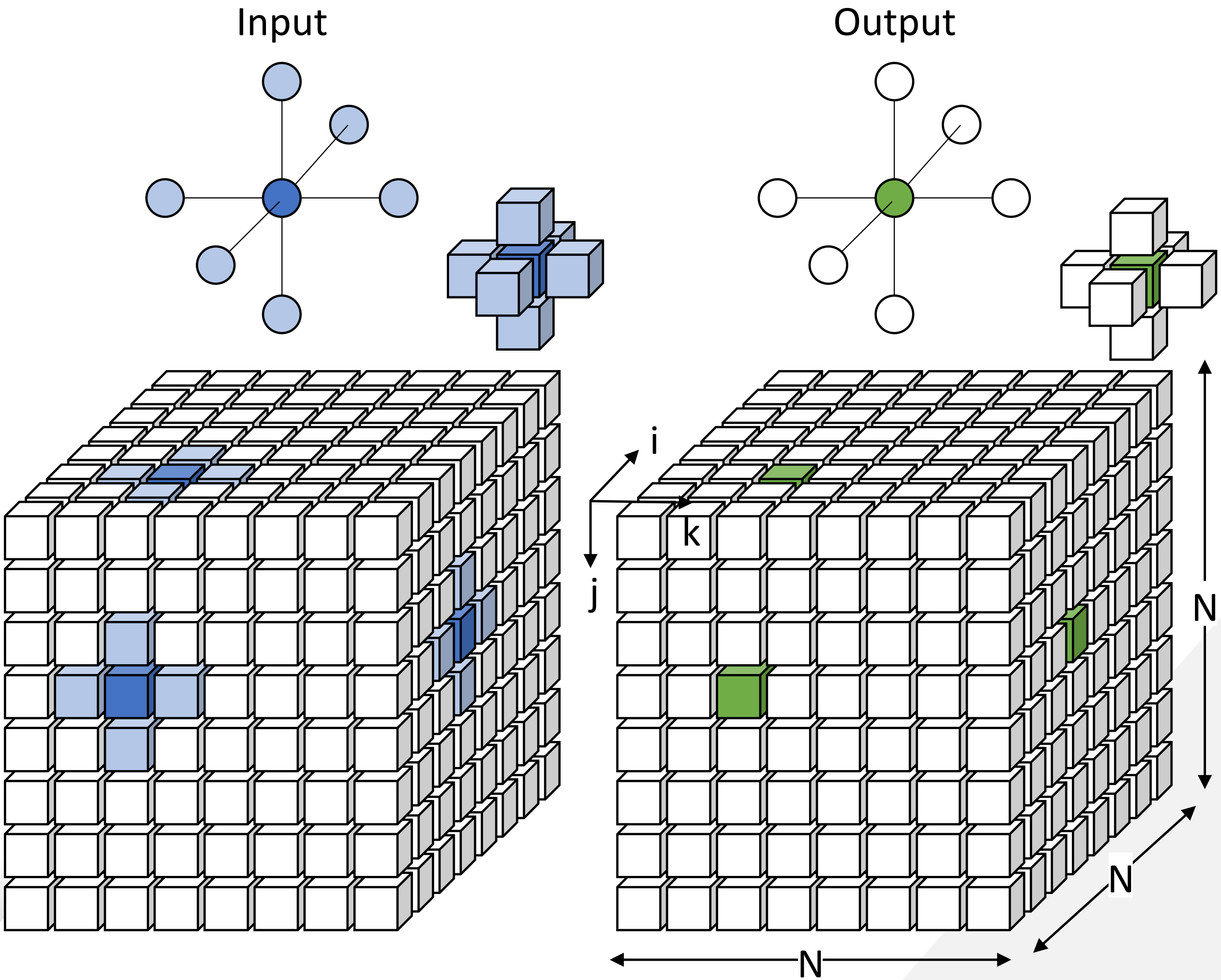
Only compute internal output values such that all input values are in bounds
(input values at the boundary typically store boundary conditions)

Copyright © 2022 Elsevier

Parallelizing Stencil (3D)

Grid of points:
(one stencil shown)

Stored as 3D array:
(8x8x8 grid shown)



Parallelization Approach:
Assign one thread per output grid point (use 3D grid of threads)

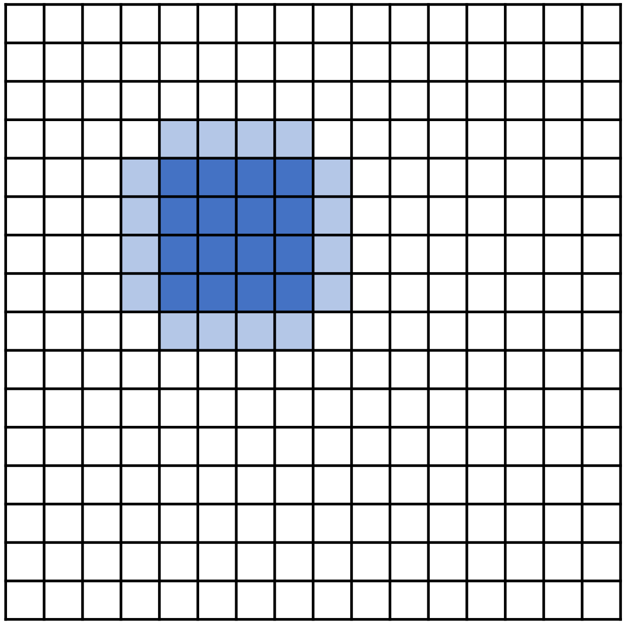
Copyright © 2022 Elsevier

Stencil Code

```
#define BLOCK_DIM  8

__global__ void stencil_kernel(float* in, float* out, unsigned int N) {
    unsigned int i = blockIdx.z*blockDim.z + threadIdx.z;
    unsigned int j = blockIdx.y*blockDim.y + threadIdx.y;
    unsigned int k = blockIdx.x*blockDim.x + threadIdx.x;
    if(i >= 1 && i < N - 1 && j >= 1 && j < N - 1&& k >= 1 && k < N - 1) {
        out[i*N*N + j*N + k] = C0*in[i*N*N + j*N + k]
                             + C1*in[i*N*N + j*N + (k - 1)]
                             + C2*in[i*N*N + j*N + (k + 1)]
                             + C3*in[i*N*N + (j - 1)*N + k]
                             + C4*in[i*N*N + (j + 1)*N + k]
                             + C5*in[(i - 1)*N*N + j*N + k]
                             + C6*in[(i + 1)*N*N + j*N + k];
    }
}
```

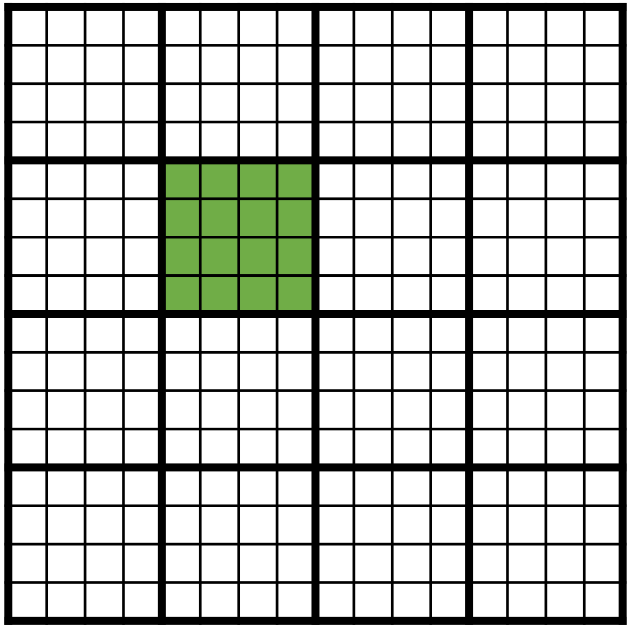
Copyright © 2022 Elsevier

Data Reuse in Stencil (2D)



input

(in global memory)

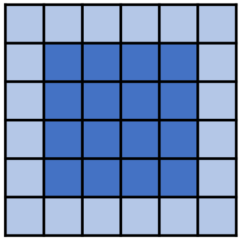


output

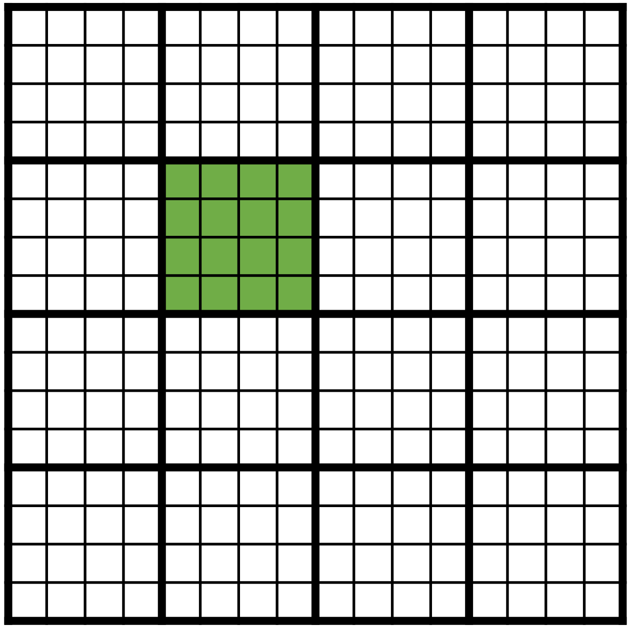
Observation: Threads in the same block load some of the same input elements

Copyright © 2022 Elsevier

Shared Memory Tiling (2D)



$input_{tile}$
(in shared memory)



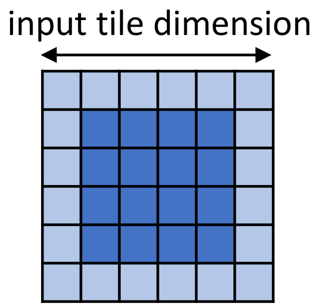
output

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

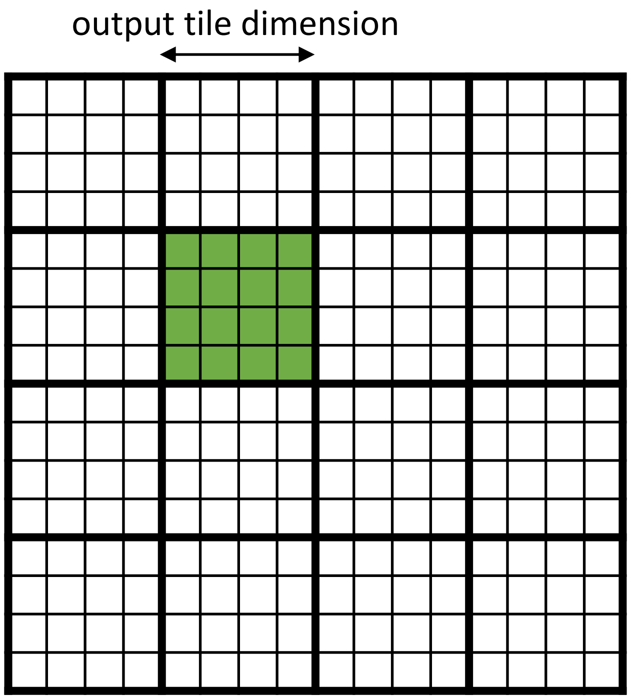
Copyright © 2022 Elsevier

Different Tile Dimensions



$input_{tile}$

(in shared memory)



output

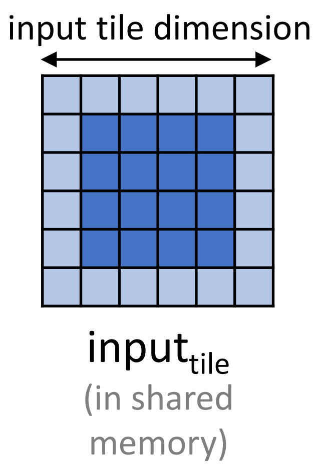
Challenge: Input and output tiles have different dimensions

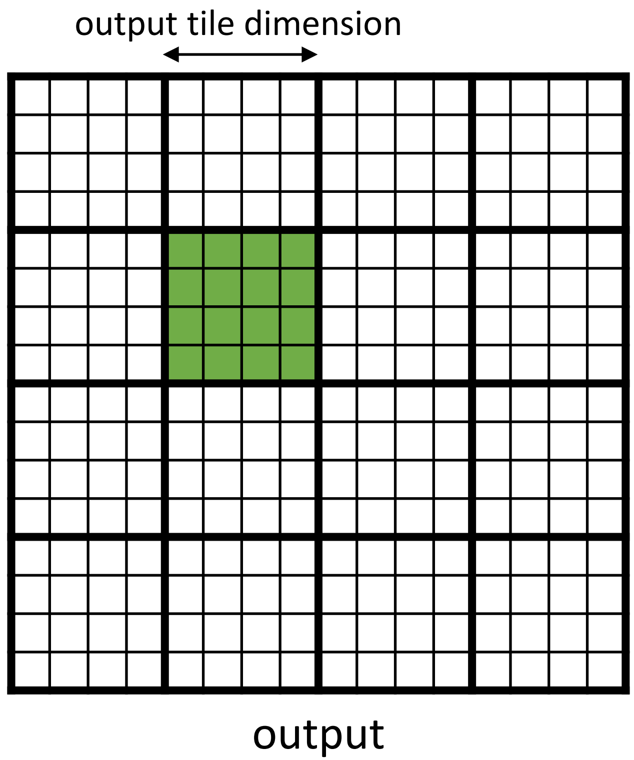
(output tile dimension = input tile dimension – 2)

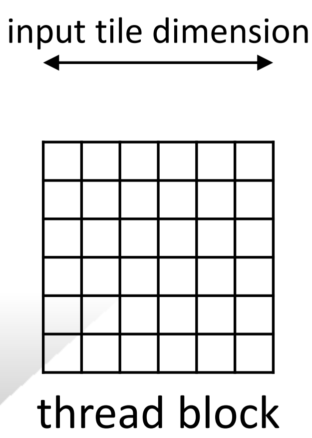
Solution: Launch enough threads per block to load the input tile to shared memory, then use a subset of them to compute and store the output tile

Copyright © 2022 Elsevier

Different Tile Dimensions

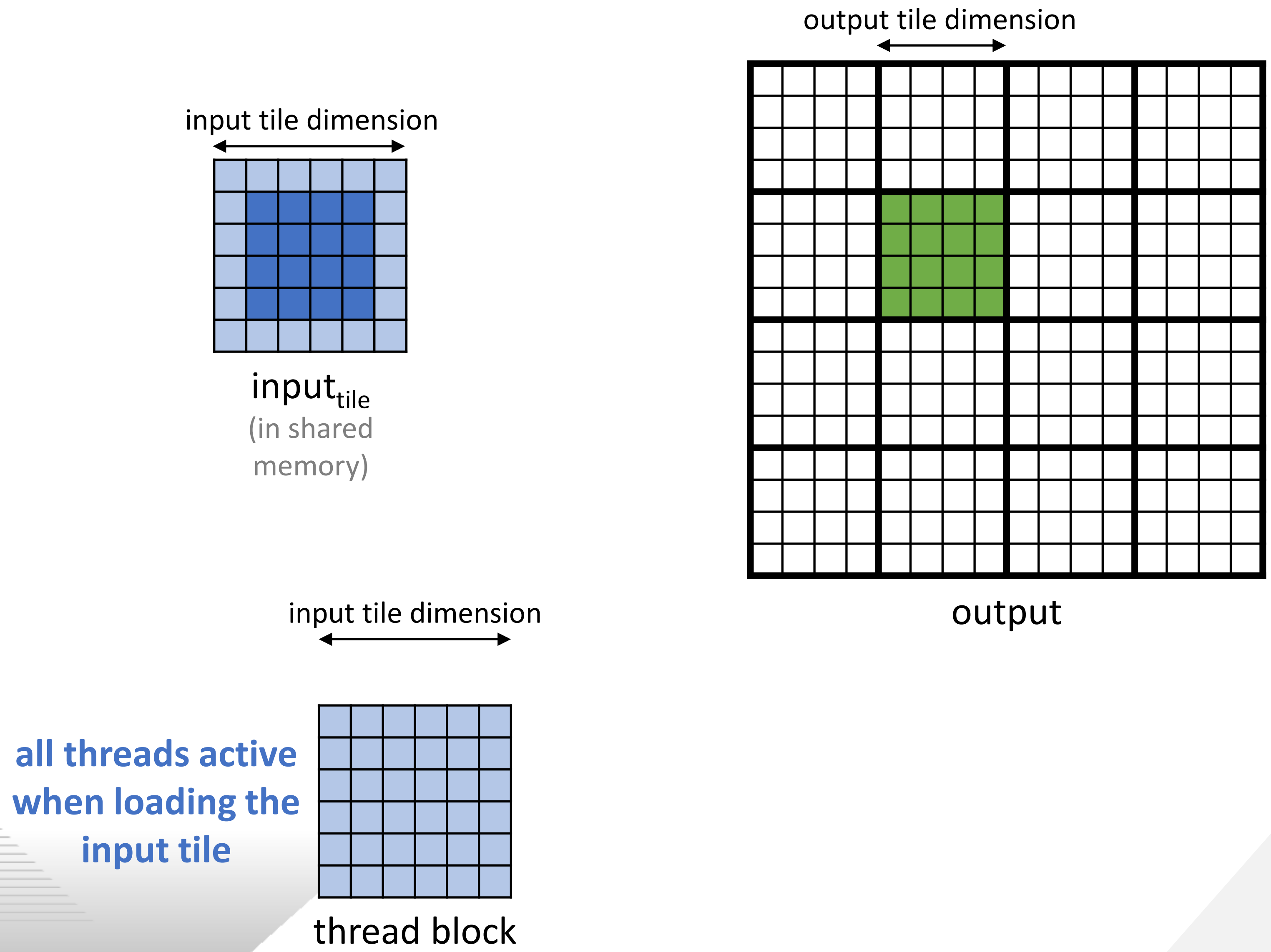






Copyright © 2022 Elsevier

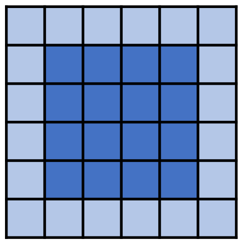
Different Tile Dimensions



Copyright © 2022 Elsevier

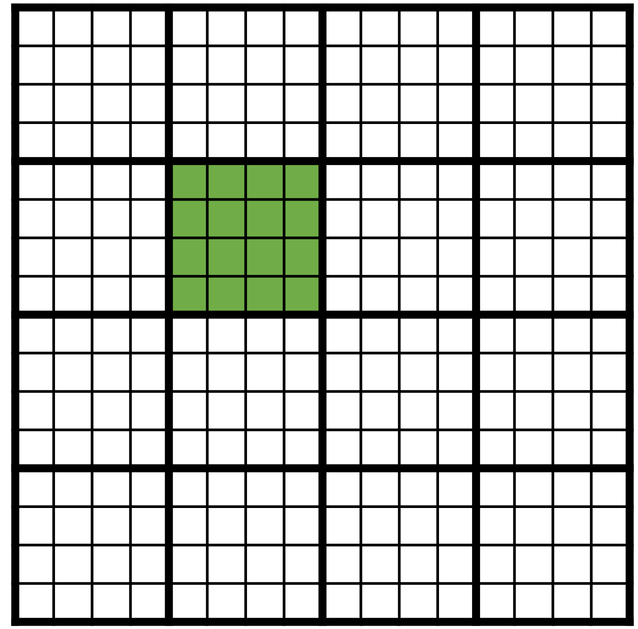
Different Tile Dimensions

input tile dimension



input$_{tile}$
(in shared memory)

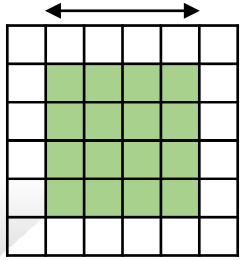
output tile dimension



output

input tile dimension

output tile dimension

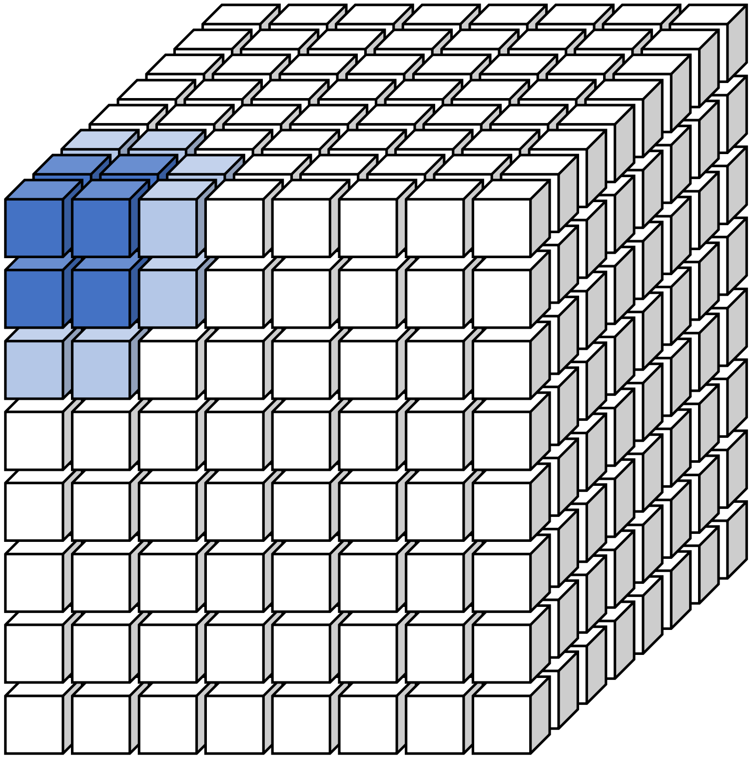


thread block

only internal threads active when computing and storing the output tile

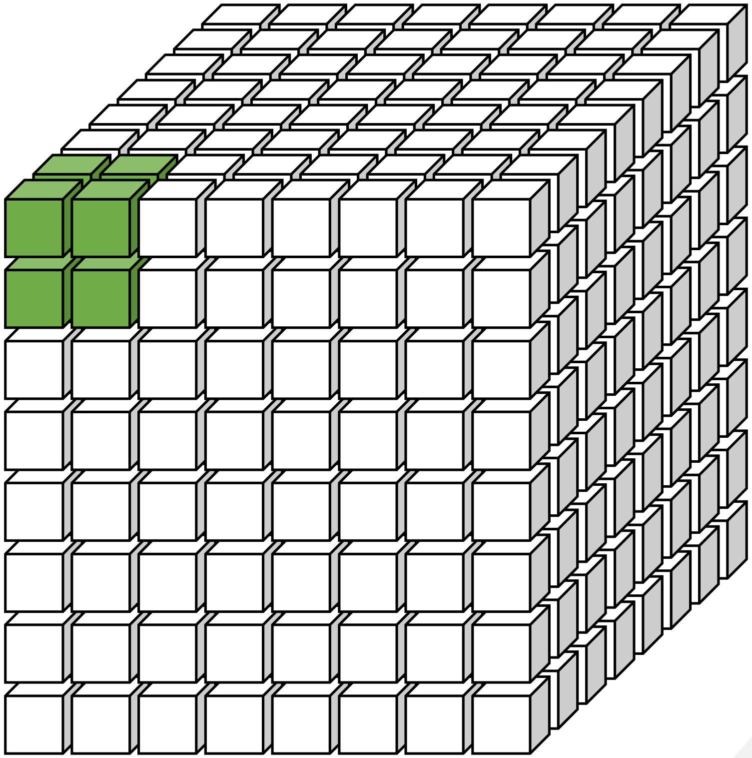
Copyright © 2022 Elsevier

Data Reuse in Stencil (3D)



input

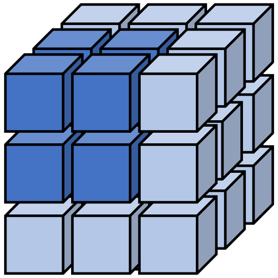
(in global memory)

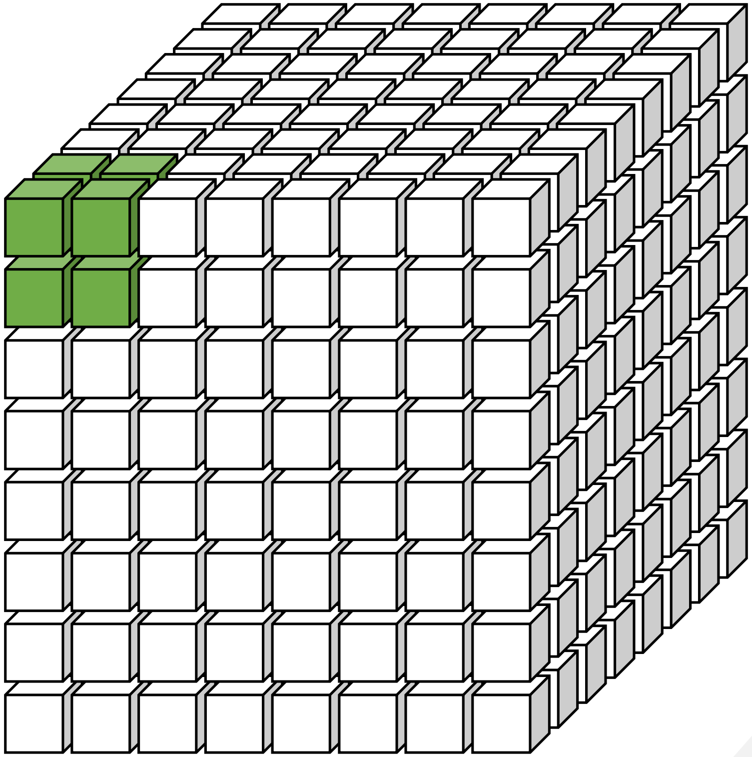


output

Copyright © 2022 Elsevier

Shared Memory Tiling





Copyright © 2022 Elsevier

Stencil with Shared Memory Tiling Code

```
#define BLOCK_DIM 8
#define IN_TILE_DIM BLOCK_DIM
#define OUT_TILE_DIM (IN_TILE_DIM - 2)

__global__ void stencil_kernel(float* in, float* out, unsigned int N) {

    int i = blockIdx.z*OUT_TILE_DIM + threadIdx.z - 1;
    int j = blockIdx.y*OUT_TILE_DIM + threadIdx.y - 1;
    int k = blockIdx.x*OUT_TILE_DIM + threadIdx.x - 1;

    __shared__ float in_s[IN_TILE_DIM][IN_TILE_DIM][IN_TILE_DIM];
    if(i >= 0 && i < N && j >= 0 && j < N && k >= 0 && k < N) {
        in_s[threadIdx.z][threadIdx.y][threadIdx.x] = in[i*N*N + j*N + k];
    }
    __syncthreads();

    if(i >= 1 && i < N - 1 && j >= 1 && j < N - 1 && k >= 1 && k < N - 1) {
         if(threadIdx.z >= 1 && threadIdx.z < IN_TILE_DIM - 1 && threadIdx.y >= 1
            && threadIdx.y < IN_TILE_DIM - 1 && threadIdx.x >= 1 && threadIdx.x < IN_TILE_DIM - 1) {
            out[i*N*N + j*N + k] = c0*in_s[threadIdx.z][threadIdx.y][threadIdx.x]
                                 + c1*in_s[threadIdx.z][threadIdx.y][threadIdx.x - 1]
                                 + c2*in_s[threadIdx.z][threadIdx.y][threadIdx.x + 1]
                                 + c3*in_s[threadIdx.z][threadIdx.y - 1][threadIdx.x]
                                 + c4*in_s[threadIdx.z][threadIdx.y + 1][threadIdx.x]
                                 + c5*in_s[threadIdx.z - 1][threadIdx.y][threadIdx.x]
                                 + c6*in_s[threadIdx.z + 1][threadIdx.y][threadIdx.x];
        }
    }

}
```

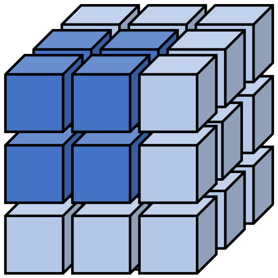
Copyright © 2022 Elsevier

Arithmetic to Global Memory Access Ratio

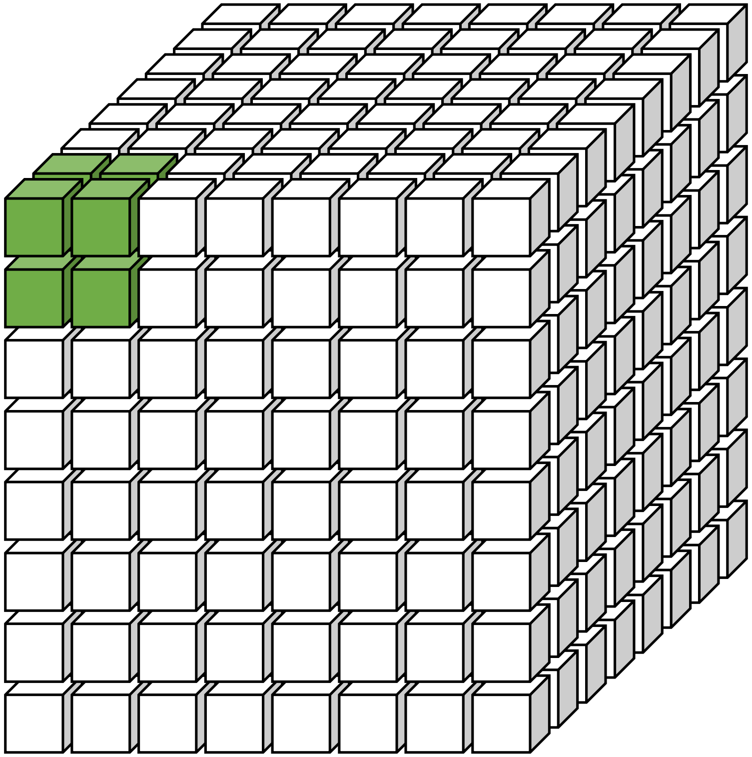
- Original kernel:
  - Each thread performs 13 OPs (6 FP adds and 7 FP muls)
  - Each thread loads 28 B from global memory (7 FP values)
  - Ratio: (13 OPs)/(28 B) = 0.46 OP/B
- Tiled kernel:
  - Assume the input tile size is T (output tile size is T – 2)
  - Each block performs (13 OPs)*$(T-2)^3 = 13(T-2)^3$ OPs
  - Each block loads (4 B)*T^3
  - Ratio: $[13*(T-2)^3]/[4*T^3] = 3.25*(1-2/T)^3$
    - For T=8, the ratio is 1.37 OP/B
    - Increasing T will improve the ratio
      - For T=32, ratio is 2.68 (≈2× improvement)
      - Intuition: boundary elements have lower data reuse, and increasing tile size decreases the ratio of boundary elements to total elements

Copyright © 2022 Elsevier

Small Tile Size



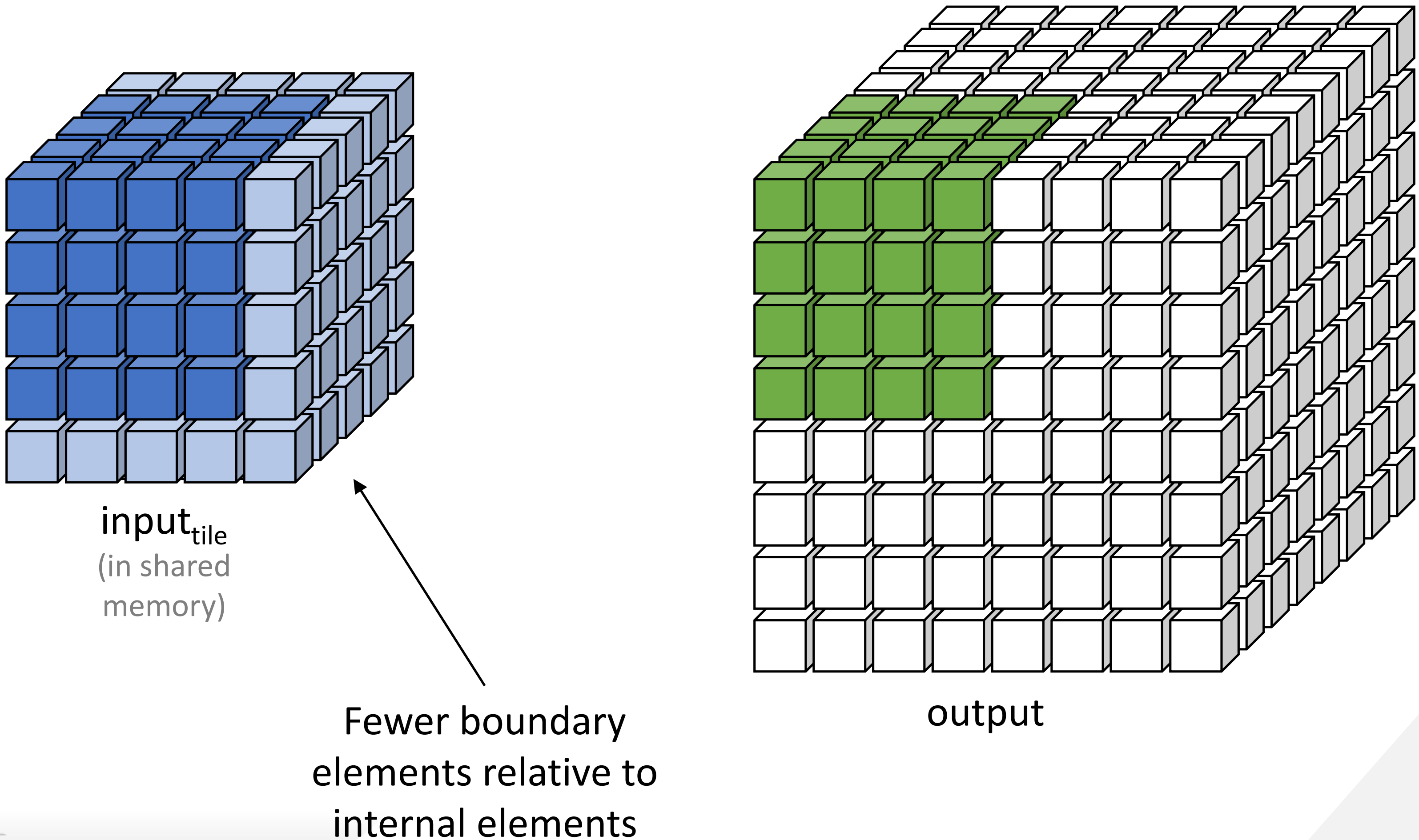
$input_{tile}$
(in shared memory)



output

Copyright © 2022 Elsevier

Large Tile Size



Copyright © 2022 Elsevier

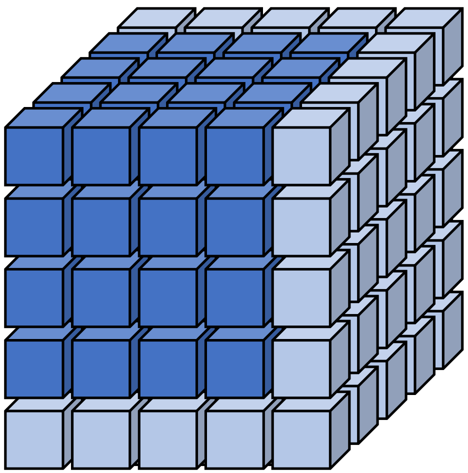
Increasing Input Tile Size

- Challenges with increasing input tile size:
  - Block size limit: The input tile size in the current implementation is the same as the block size which is limited by the hardware
  - Shared memory capacity limit: The input tile size in the current implementation determines the shared memory usage per block which is limited by hardware
    - Even if the limit is not exceeded, using too much shared memory may hurt occupancy

- Solutions:
  - Block size limit: Use thread coarsening to process a larger input/output tile without using more threads
    - Price of parallelization here is redundant loading of boundary elements
  - Shared memory capacity limit: Only keep needed slices of the input tile in shared memory

Copyright © 2022 Elsevier

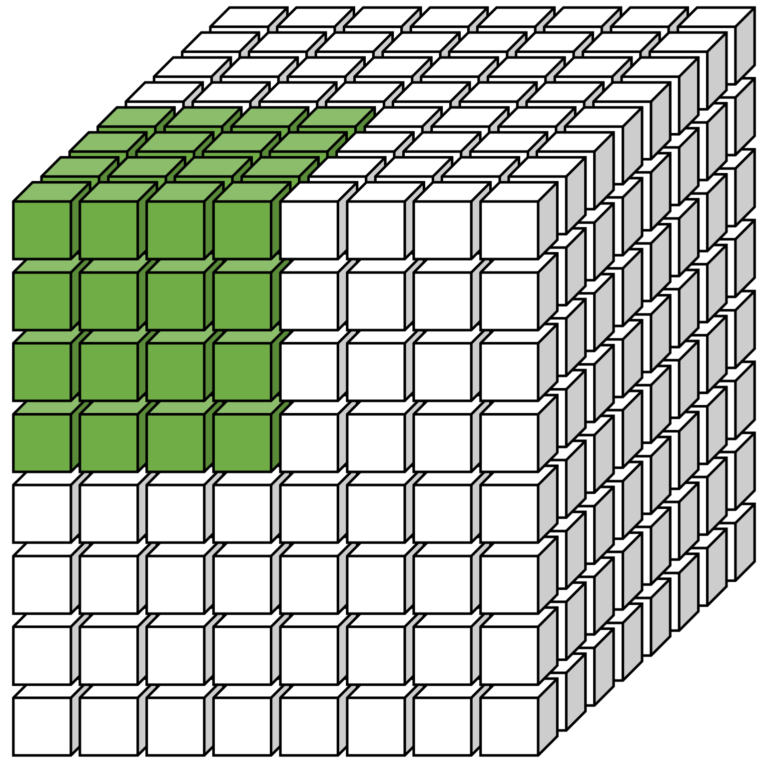
Limits Exceeded



input$_{tile}$
(in shared memory)

<u>Challenge:</u>
Input tile requires too much shared memory (hurts occupancy or even exceeds limit)

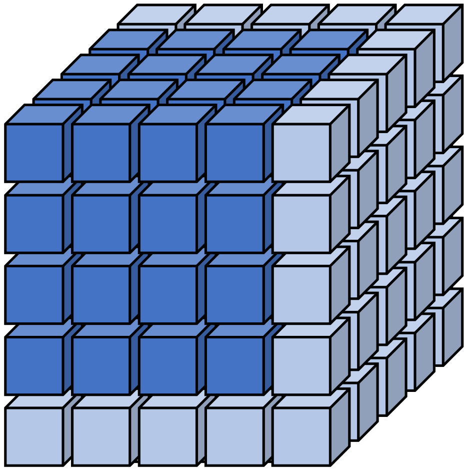
<u>Challenge:</u>
Maximum threads per block exceeded



output

Copyright © 2022 Elsevier

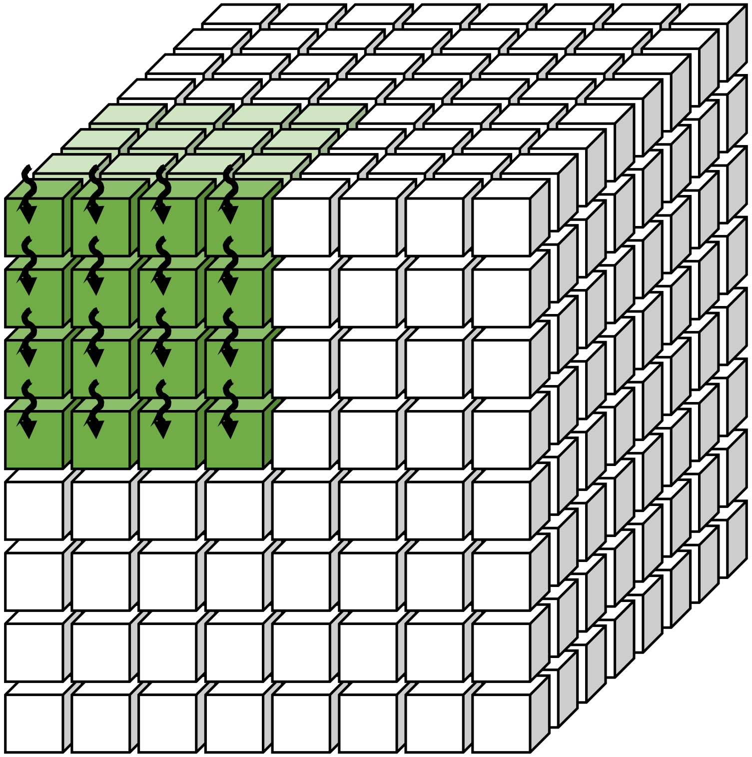
Thread Coarsening and Tile Slicing



input$_{tile}$
(in shared memory)

Challenge:
Input tile requires too much shared memory (hurts occupancy or even exceeds limit)

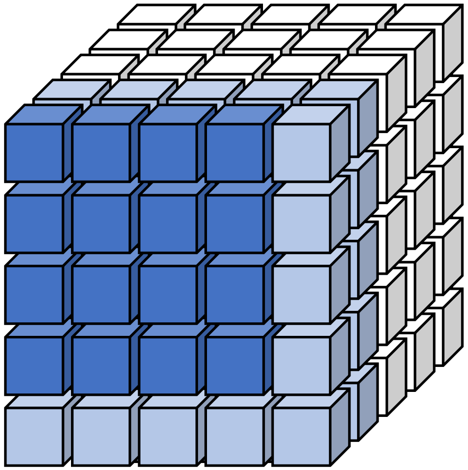
Solution:
Assign enough threads for loading one input plane and processing one output plane



output

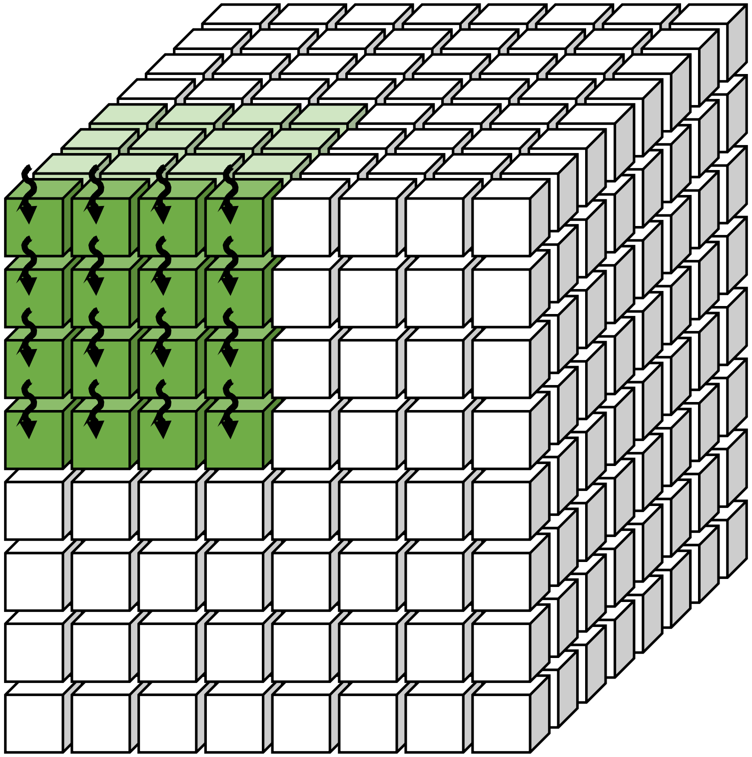
Copyright © 2022 Elsevier

Thread Coarsening and Tile Slicing



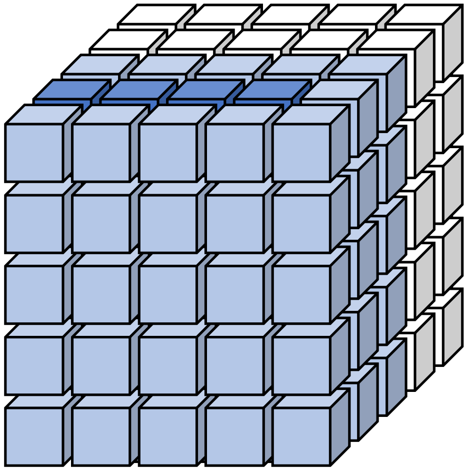
<u>Solution:</u>
Only store the three input planes needed by the output plane at a time

<u>Solution:</u>
Assign enough threads for loading one input plane and processing one output plane



Copyright © 2022 Elsevier

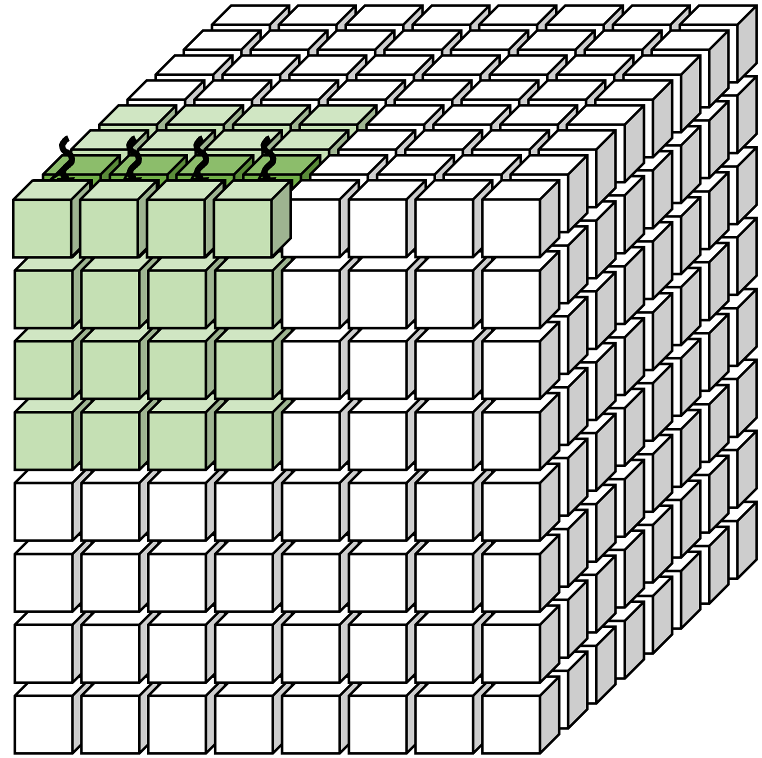
Thread Coarsening and Tile Slicing



$input_{tile}$
(in shared memory)

Solution:
Only store the three input planes needed by the output plane at a time

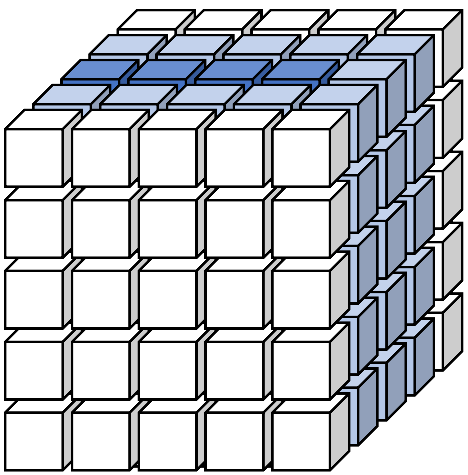
Solution:
Assign enough threads for loading one input plane and processing one output plane



output

Copyright © 2022 Elsevier

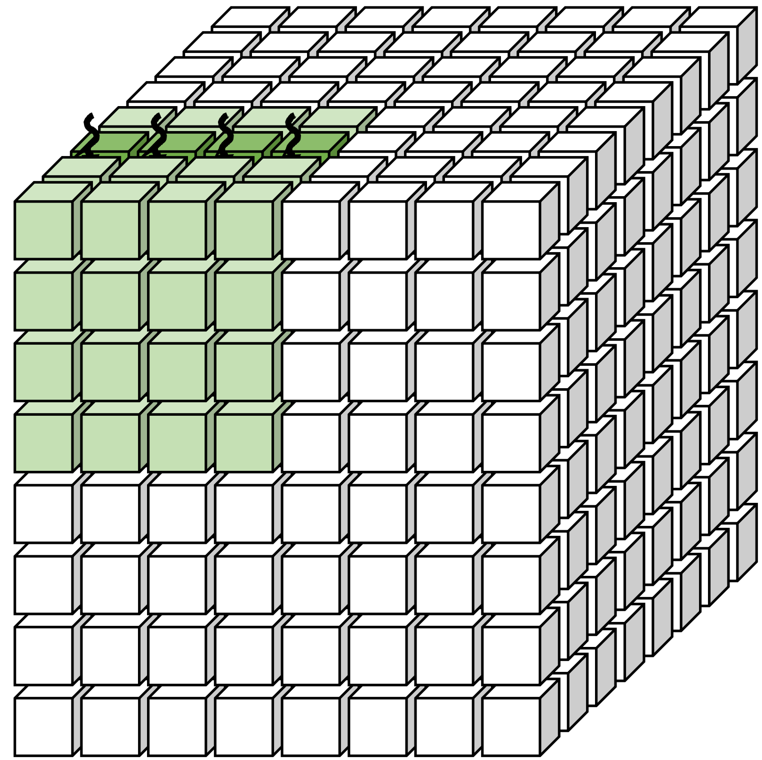
Thread Coarsening and Tile Slicing



$\text{input}_{\text{tile}}$
(in shared memory)

Solution:
Only store the three input planes needed by the output plane at a time

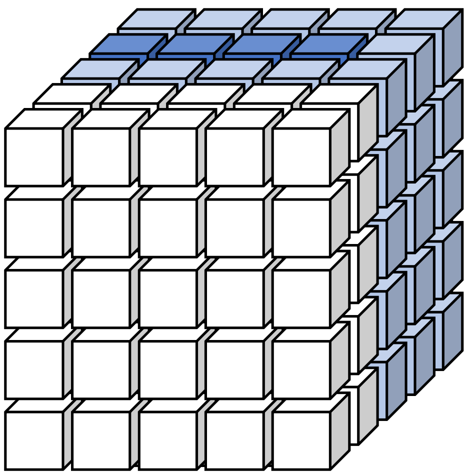
Solution:
Assign enough threads for loading one input plane and processing one output plane



output

Copyright © 2022 Elsevier

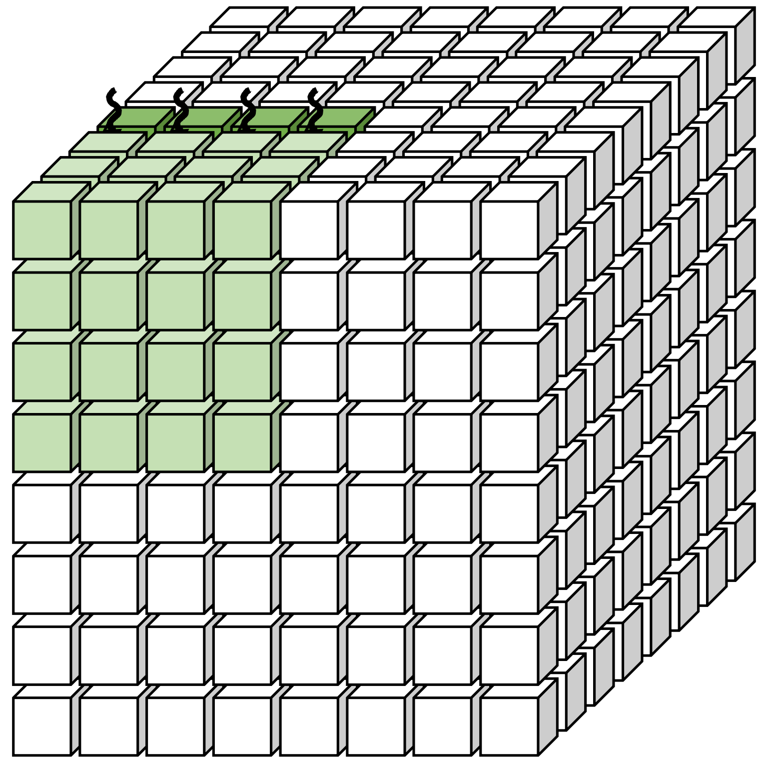
Thread Coarsening and Tile Slicing



input$_{tile}$
(in shared memory)

Solution:
Only store the three input planes needed by the output plane at a time

Solution:
Assign enough threads for loading one input plane and processing one output plane



output

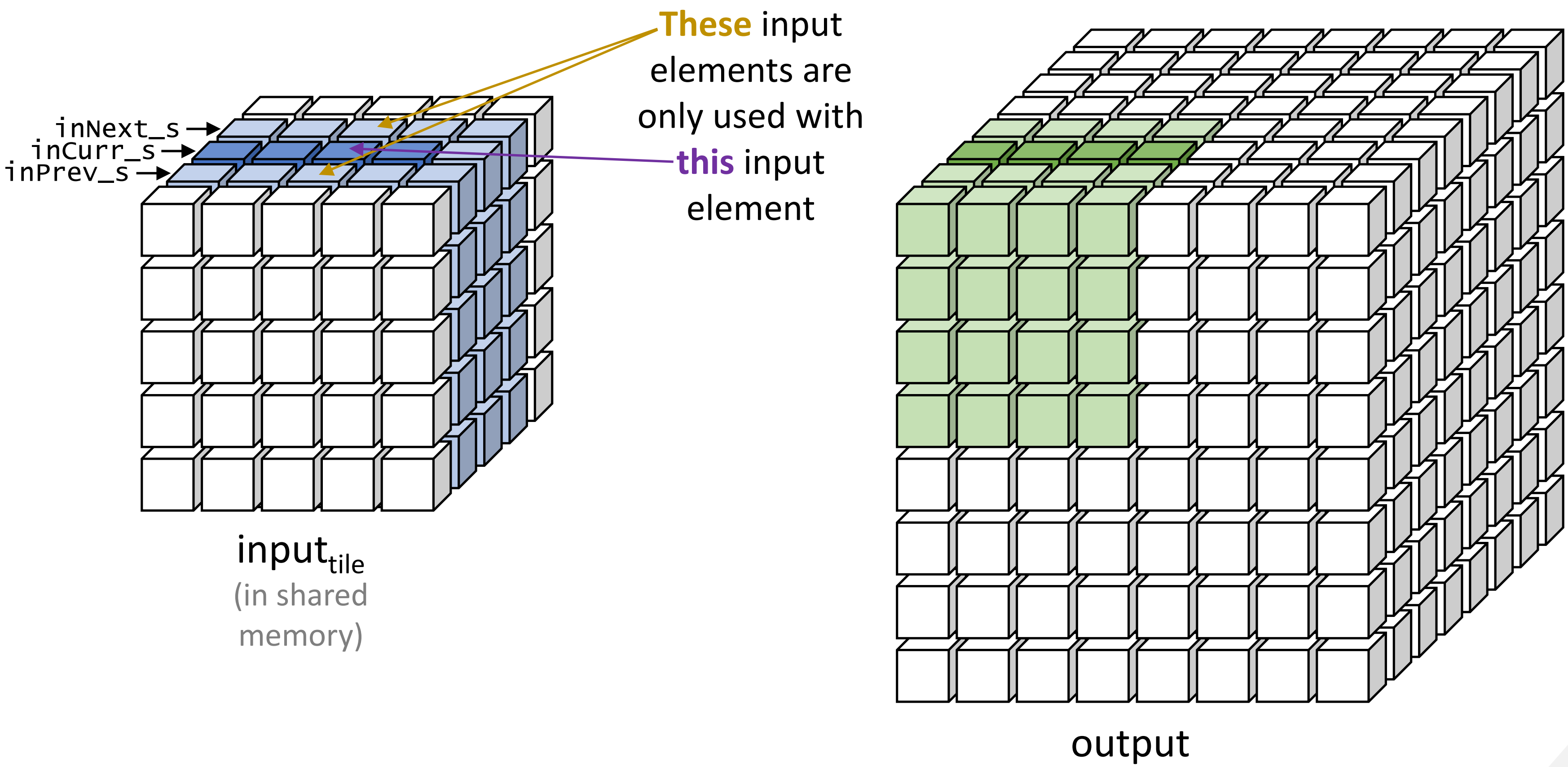
Copyright © 2022 Elsevier

Stencil with Thread Coarsening Code

```
__global__ void stencil_kernel(float* in, float* out, unsigned int N) {

    int iStart = blockIdx.z*OUT_TILE_DIM;
    int j = blockIdx.y*OUT_TILE_DIM + threadIdx.y - 1;
    int k = blockIdx.x*OUT_TILE_DIM + threadIdx.x - 1;

    __shared__ float inPrev_s[IN_TILE_DIM][IN_TILE_DIM];
    __shared__ float inCurr_s[IN_TILE_DIM][IN_TILE_DIM];
    __shared__ float inNext_s[IN_TILE_DIM][IN_TILE_DIM];
    if(iStart - 1 >= 0 && iStart - 1 < N && j >= 0 && j < N && k >= 0 && k < N) {
        inPrev_s[threadIdx.y][threadIdx.x] = in[(iStart - 1)*N*N + j*N + k];
    }
    if(iStart >= 0 && iStart < N && j >= 0 && j < N && k >= 0 && k < N) {
        inCurr_s[threadIdx.y][threadIdx.x] = in[iStart*N*N + j*N + k];
    }

    for(int i = iStart; i < iStart + OUT_TILE_DIM; ++i) {
        if(i + 1 >= 0 && i + 1 < N && j >= 0 && j < N && k >= 0 && k < N) {
            inNext_s[threadIdx.y][threadIdx.x] = in[(i + 1)*N*N + j*N + k];
        }
        __syncthreads();
        if(i >= 1 && i < N - 1 && j >= 1 && j < N - 1 && k >= 1 && k < N - 1) {
            if(threadIdx.y >= 1 && threadIdx.y < IN_TILE_DIM - 1
               && threadIdx.x >= 1 && threadIdx.x < IN_TILE_DIM - 1) {
                out[i*N*N + j*N + k] = c0*inCurr_s[threadIdx.y][threadIdx.x]
                                     + c1*inCurr_s[threadIdx.y][threadIdx.x - 1]
                                     + c2*inCurr_s[threadIdx.y][threadIdx.x + 1]
                                     + c3*inCurr_s[threadIdx.y - 1][threadIdx.x]
                                     + c4*inCurr_s[threadIdx.y + 1][threadIdx.x]
                                     + c5*inPrev_s[threadIdx.y][threadIdx.x] +
                                     + c6*inNext_s[threadIdx.y][threadIdx.x];
            }
        }
        __syncthreads();
        inPrev_s[threadIdx.y][threadIdx.x] = inCurr_s[threadIdx.y][threadIdx.x];
        inCurr_s[threadIdx.y][threadIdx.x] = inNext_s[threadIdx.y][threadIdx.x];
    }
}
```

Copyright © 2022 Elsevier

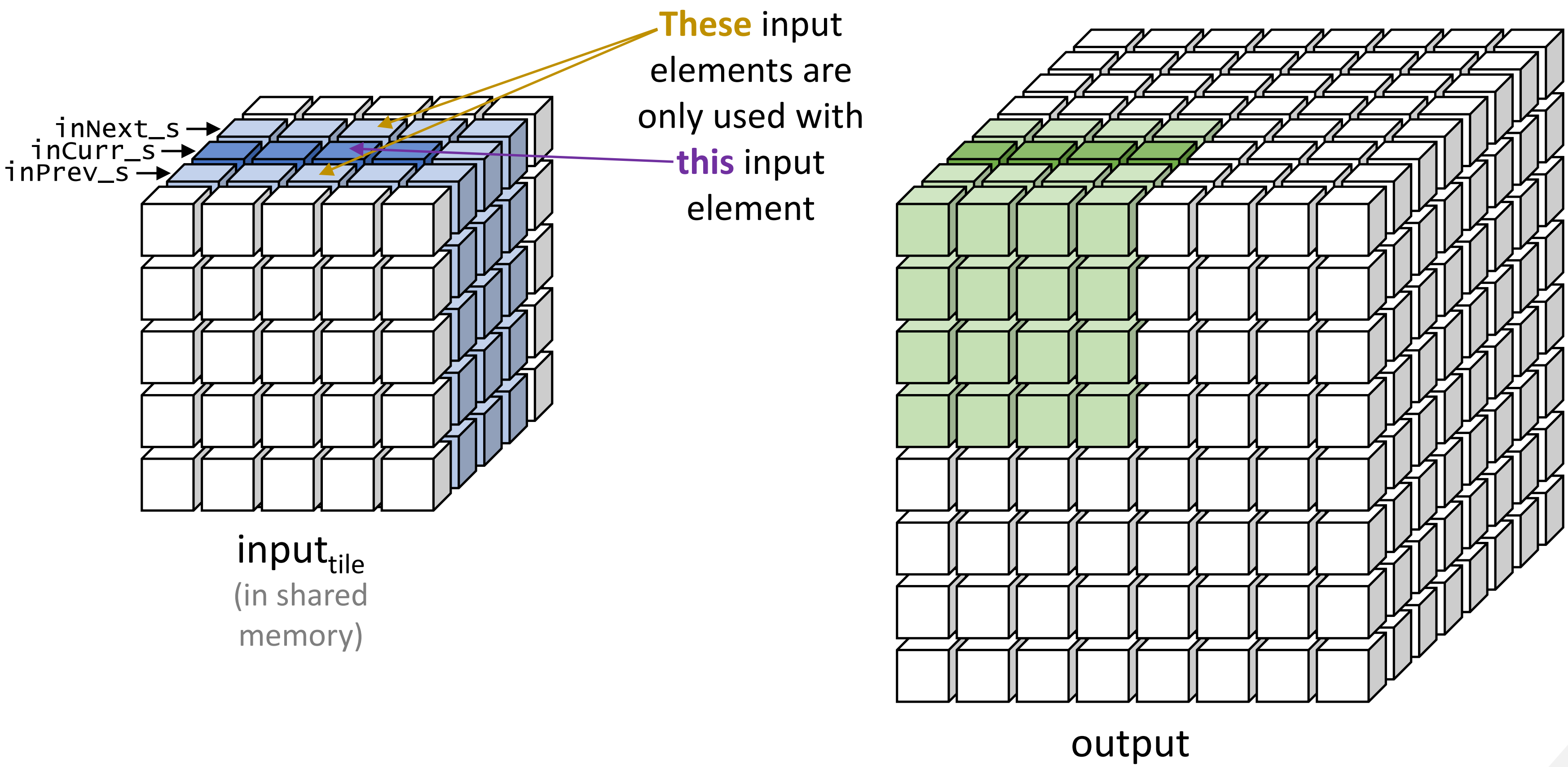
Unshared Slices



Observation: Only the current slice is truly shared by the threads. The previous and next slice elements are only needed by the thread that loaded them.

Copyright © 2022 Elsevier

Register Tiling



Optimization: Save shared memory by putting the next slice elements in registers, moving them to shared memory when they become the current slice, then moving them back to registers when they become the previous slice. We only need enough shared memory for one slice

The registers of the different threads collectively form a tile (called register tiling).

Copyright © 2022 Elsevier

Stencil with Thread Coarsening Code

```
__global__ void stencil_kernel(float* in, float* out, unsigned int N) {

    int iStart = blockIdx.z*OUT_TILE_DIM;
    int j = blockIdx.y*OUT_TILE_DIM + threadIdx.y - 1;
    int k = blockIdx.x*OUT_TILE_DIM + threadIdx.x - 1;

    __shared__ float inPrev_s[IN_TILE_DIM][IN_TILE_DIM];
    __shared__ float inCurr_s[IN_TILE_DIM][IN_TILE_DIM];
    __shared__ float inNext_s[IN_TILE_DIM][IN_TILE_DIM];
    if(iStart - 1 >= 0 && iStart - 1 < N && j >= 0 && j < N && k >= 0 && k < N) {
        inPrev_s[threadIdx.y][threadIdx.x] = in[(iStart - 1)*N*N + j*N + k];
    }
    if(iStart >= 0 && iStart < N && j >= 0 && j < N && k >= 0 && k < N) {
        inCurr_s[threadIdx.y][threadIdx.x] = in[iStart*N*N + j*N + k];
    }

    for(int i = iStart; i < iStart + OUT_TILE_DIM; ++i) {
        if(i + 1 >= 0 && i + 1 < N && j >= 0 && j < N && k >= 0 && k < N) {
            inNext_s[threadIdx.y][threadIdx.x] = in[(i + 1)*N*N + j*N + k];
        }
        __syncthreads();
        if(i >= 1 && i < N - 1 && j >= 1 && j < N - 1 && k >= 1 && k < N - 1) {
            if(threadIdx.y >= 1 && threadIdx.y < IN_TILE_DIM - 1
               && threadIdx.x >= 1 && threadIdx.x < IN_TILE_DIM - 1) {
                out[i*N*N + j*N + k] = c0*inCurr_s[threadIdx.y][threadIdx.x]
                                     + c1*inCurr_s[threadIdx.y][threadIdx.x - 1]
                                     + c2*inCurr_s[threadIdx.y][threadIdx.x + 1]
                                     + c3*inCurr_s[threadIdx.y - 1][threadIdx.x]
                                     + c4*inCurr_s[threadIdx.y + 1][threadIdx.x]
                                     + c5*inPrev_s[threadIdx.y][threadIdx.x]
                                     + c6*inNext_s[threadIdx.y][threadIdx.x];
            }
        }
        __syncthreads();
        inPrev_s[threadIdx.y][threadIdx.x] = inCurr_s[threadIdx.y][threadIdx.x];
        inCurr_s[threadIdx.y][threadIdx.x] = inNext_s[threadIdx.y][threadIdx.x];
    }
}
```

Copyright © 2022 Elsevier

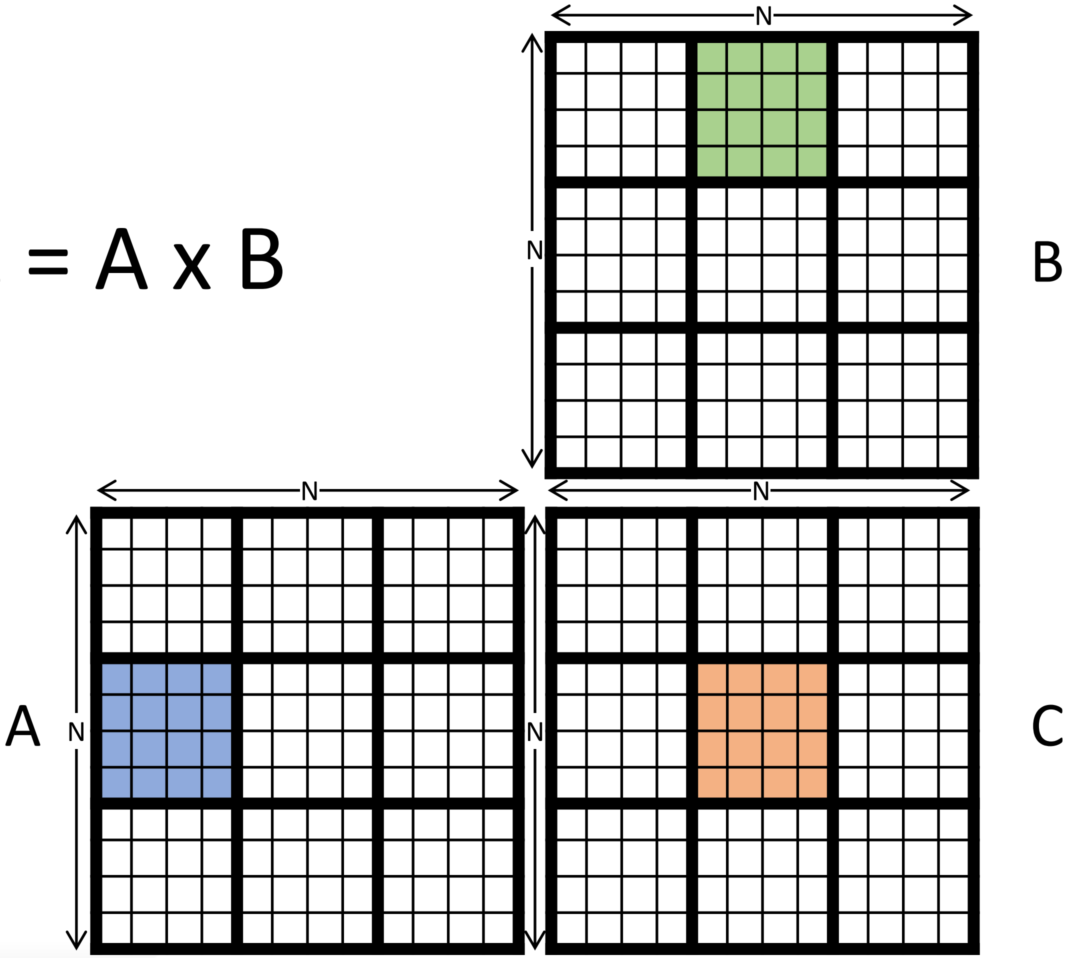
Stencil with Register Tiling Code

```
__global__ void stencil_kernel(float* in, float* out, unsigned int N) {

    int iStart = blockIdx.z*OUT_TILE_DIM;
    int j = blockIdx.y*OUT_TILE_DIM + threadIdx.y - 1;
    int k = blockIdx.x*OUT_TILE_DIM + threadIdx.x - 1;

    float inPrev;
    __shared__ float inCurr_s[IN_TILE_DIM][IN_TILE_DIM];
    float inNext;
    if(iStart - 1 >= 0 && iStart - 1 < N && j >= 0 && j < N && k >= 0 && k < N) {
        inPrev = in[(iStart - 1)*N*N + j*N + k];
    }
    if(iStart >= 0 && iStart < N && j >= 0 && j < N && k >= 0 && k < N) {
        inCurr_s[threadIdx.y][threadIdx.x] = in[iStart*N*N + j*N + k];
    }

    for(int i = iStart; i < iStart + OUT_TILE_DIM; ++i) {
        if(i + 1 >= 0 && i + 1 < N && j >= 0 && j < N && k >= 0 && k < N) {
            inNext = in[(i + 1)*N*N + j*N + k];
        }
        __syncthreads();
        if(i >= 1 && i < N - 1 && j >= 1 && j < N - 1 && k >= 1 && k < N - 1) {
            if(threadIdx.y >= 1 && threadIdx.y < IN_TILE_DIM - 1
               && threadIdx.x >= 1 && threadIdx.x < IN_TILE_DIM - 1) {
                out[i*N*N + j*N + k] = c0*inCurr_s[threadIdx.y][threadIdx.x]
                                     + c1*inCurr_s[threadIdx.y][threadIdx.x - 1]
                                     + c2*inCurr_s[threadIdx.y][threadIdx.x + 1]
                                     + c3*inCurr_s[threadIdx.y - 1][threadIdx.x]
                                     + c4*inCurr_s[threadIdx.y + 1][threadIdx.x]
                                     + c5*inPrev
                                     + c6*inNext;
            }
        }
        __syncthreads();
        inPrev = inCurr_s[threadIdx.y][threadIdx.x];
        inCurr_s[threadIdx.y][threadIdx.x] = inNext;
    }
}
```

Copyright © 2022 Elsevier

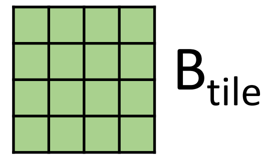
Have we seen register tiling before?

$C = A \times B$

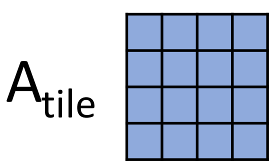


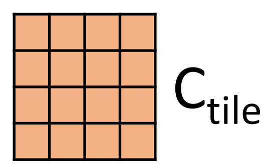
Copyright © 2022 Elsevier

Have we seen register tiling before?



$$C_{tile} = A_{tile} \times B_{tile}$$





The A and B input tiles were stored in shared memory.

The C output tile was stored in the registers of the threads collectively.

Stencil just made register tiling more apparent because the same tile was sometimes stored in registers and other times in shared memory.

Copyright © 2022 Elsevier

References

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

Copyright © 2022 Elsevier