CS 677: Parallel Programming for Many-core Processors
Lecture 6

Instructor: Philippos Mordohai
Webpage: www.cs.stevens.edu/~mordohai
E-mail: Philippos.Mordohai@stevens.edu

Overview

- Parallel Patterns: Convolution
  - Constant memory
  - Cache
- Parallel Patterns: Reduction Trees
- Parallel Patterns: Parallel Prefix Sum (Scan)

Convolution Applications

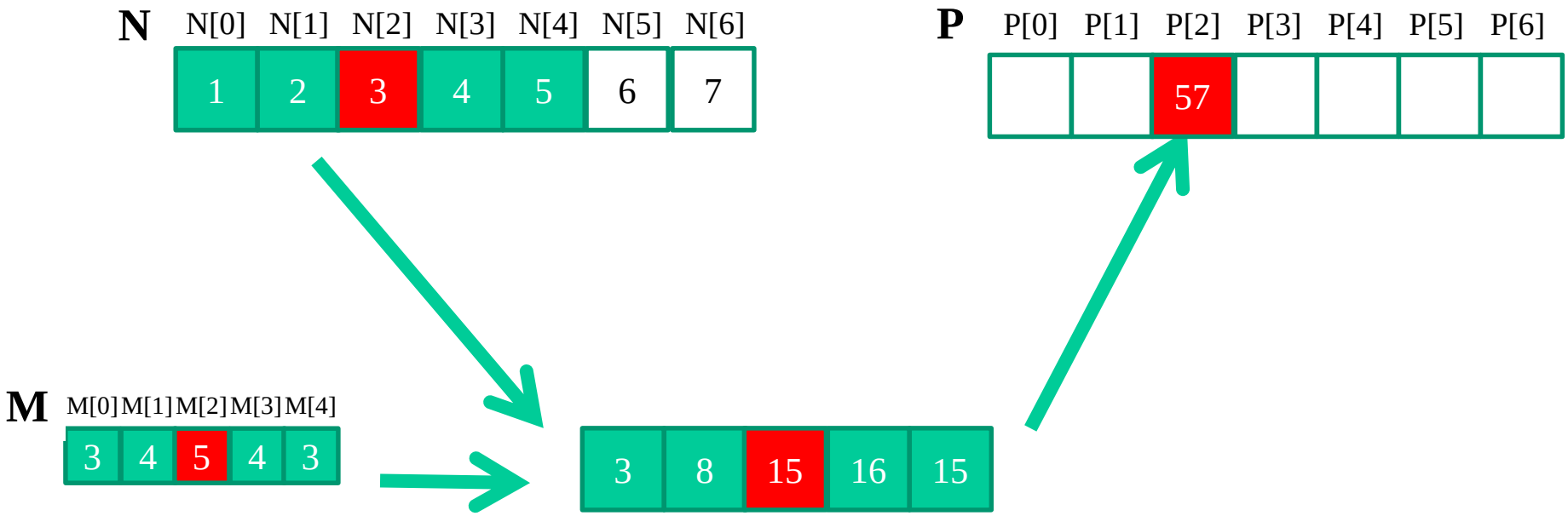
- A popular array operation that is used in various forms in signal processing, digital recording, image processing, video processing, and computer vision
- Convolution is often performed as a filter that transforms signals and pixels into more desirable values
  - Some filters smooth out the signal values so that one can see the big-picture trend
  - Others like Gaussian filters can be used to sharpen boundaries and edges of objects in images

Convolution Computation

- Array operation where each output is a weighted sum of a collection of neighboring input elements
- Weights are defined in a *mask array* a.k.a. *convolution kernel*

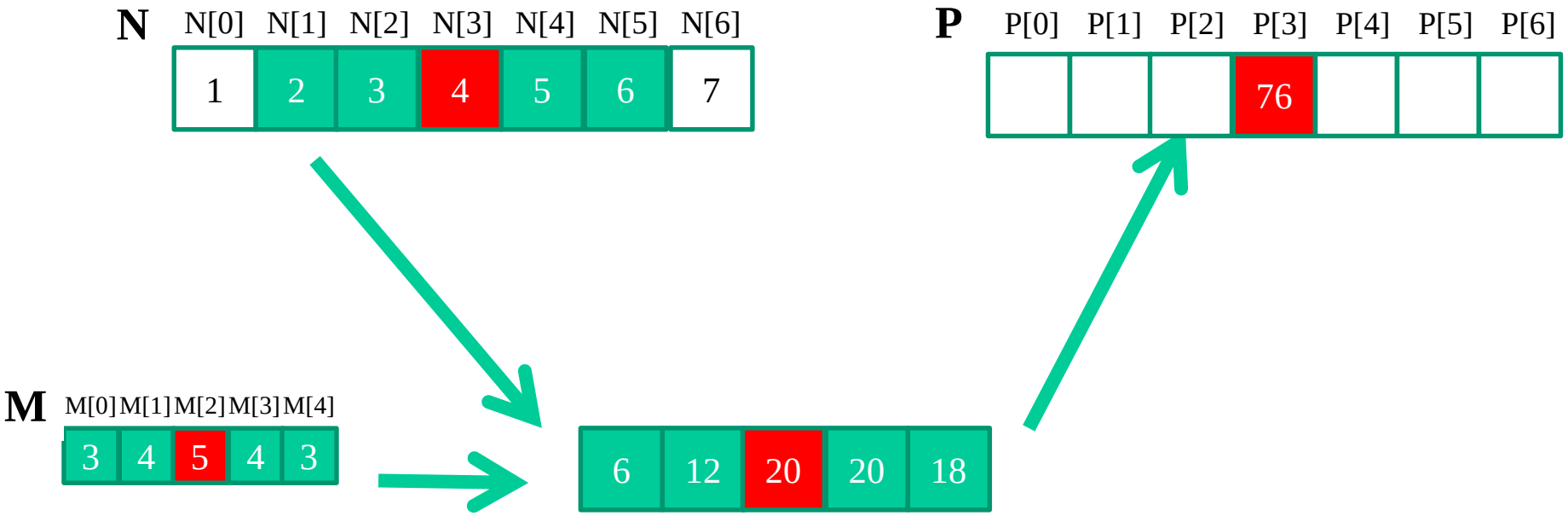
1D Convolution Example

- Commonly used for audio processing
  - Mask size is usually an odd number of elements for symmetry (5 in this example)
- Calculation of P[2]



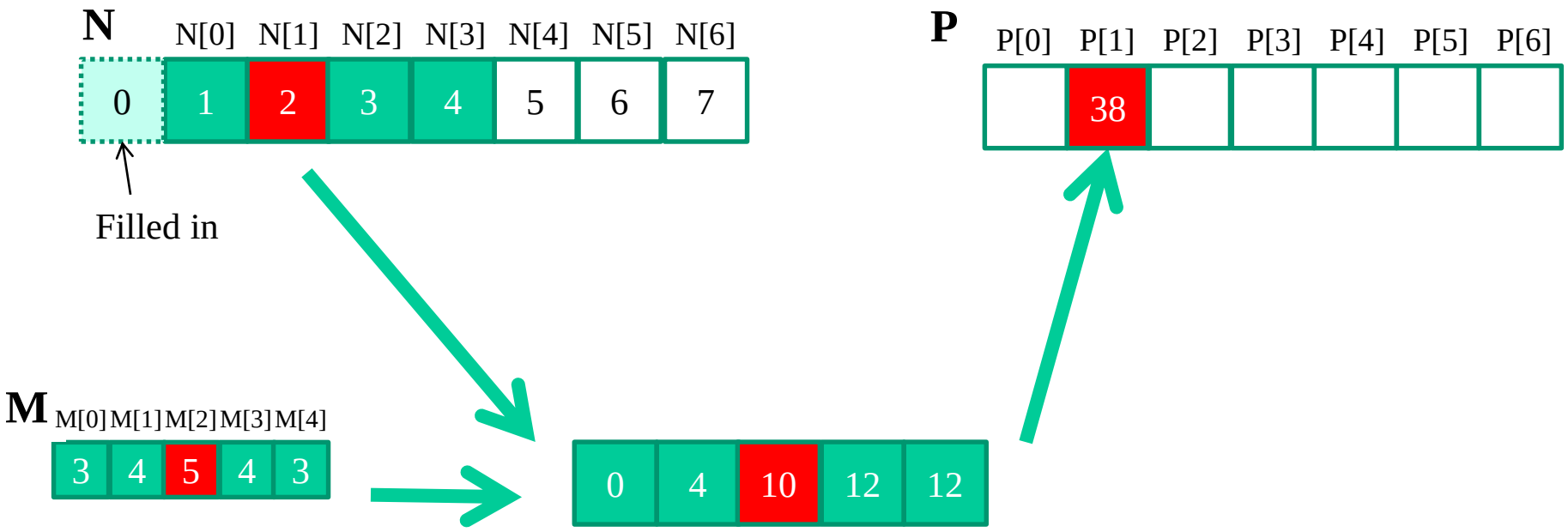
1D Convolution Example

- Calculation of P[3]



1D Convolution – Boundary Condition

- Calculation of output elements near the boundaries (beginning and end) of the input array need to deal with “ghost” elements
  - Different policies (0, replicates of boundary values, etc.)



Simple 1D Covolution Kernel

- This kernel forces all elements outside the valid data index range to 0

```
__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;

}
```

2D Convolution - Inside Cells

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

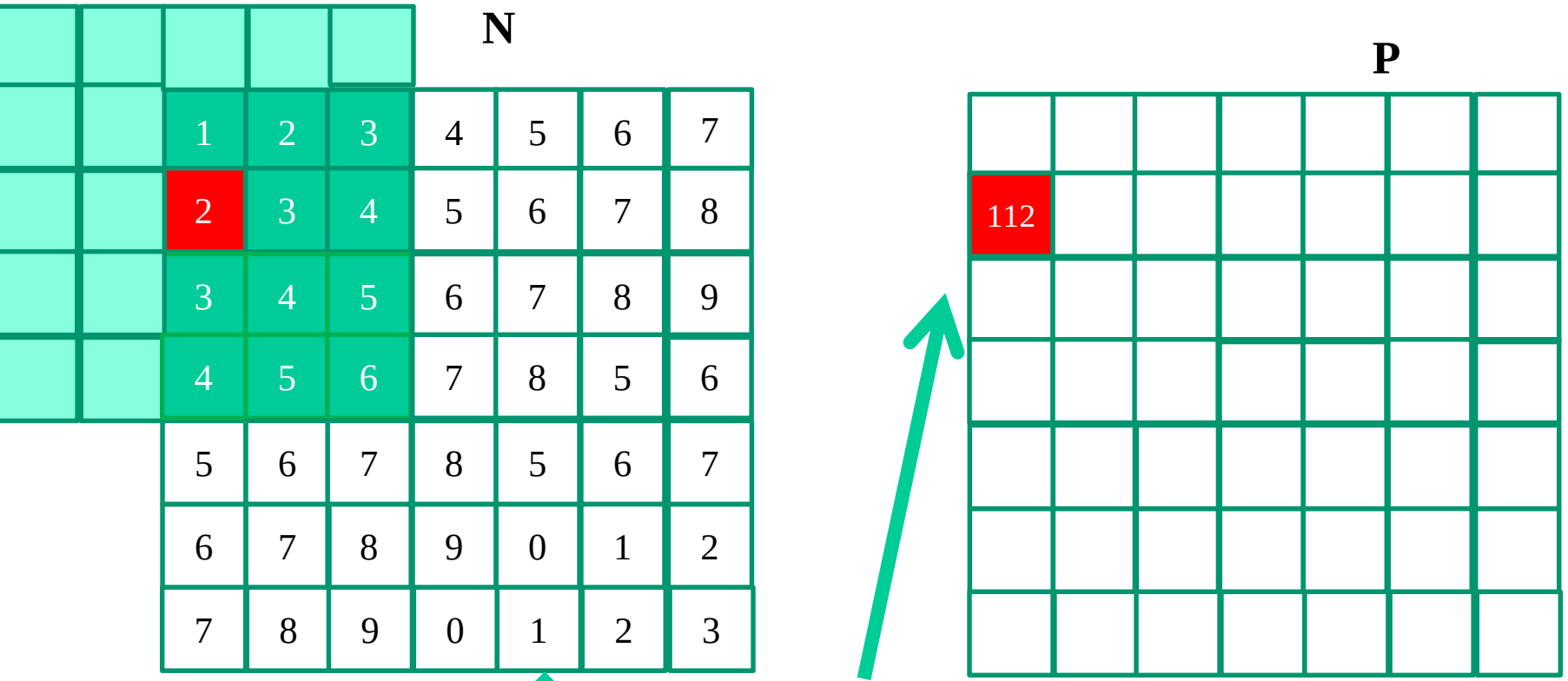
		321				

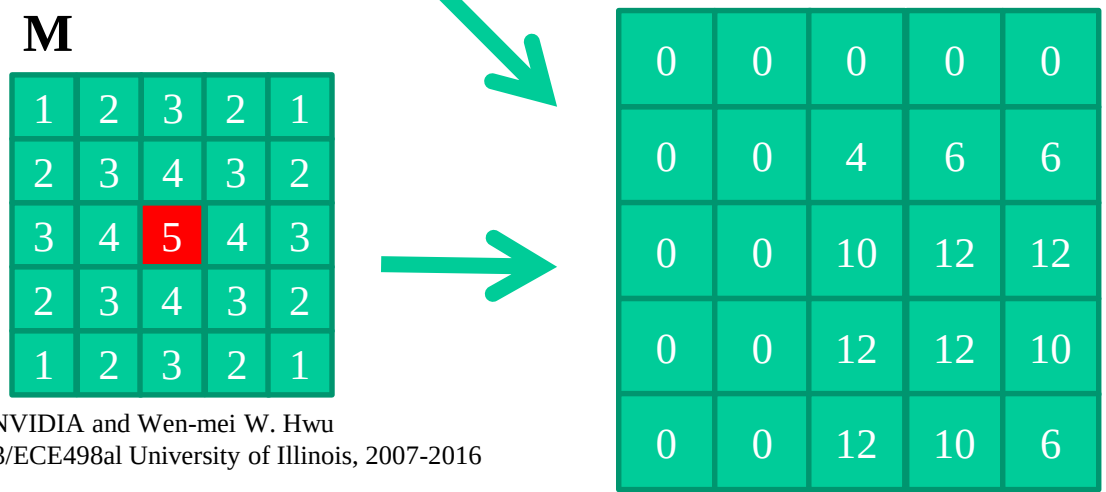
M

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

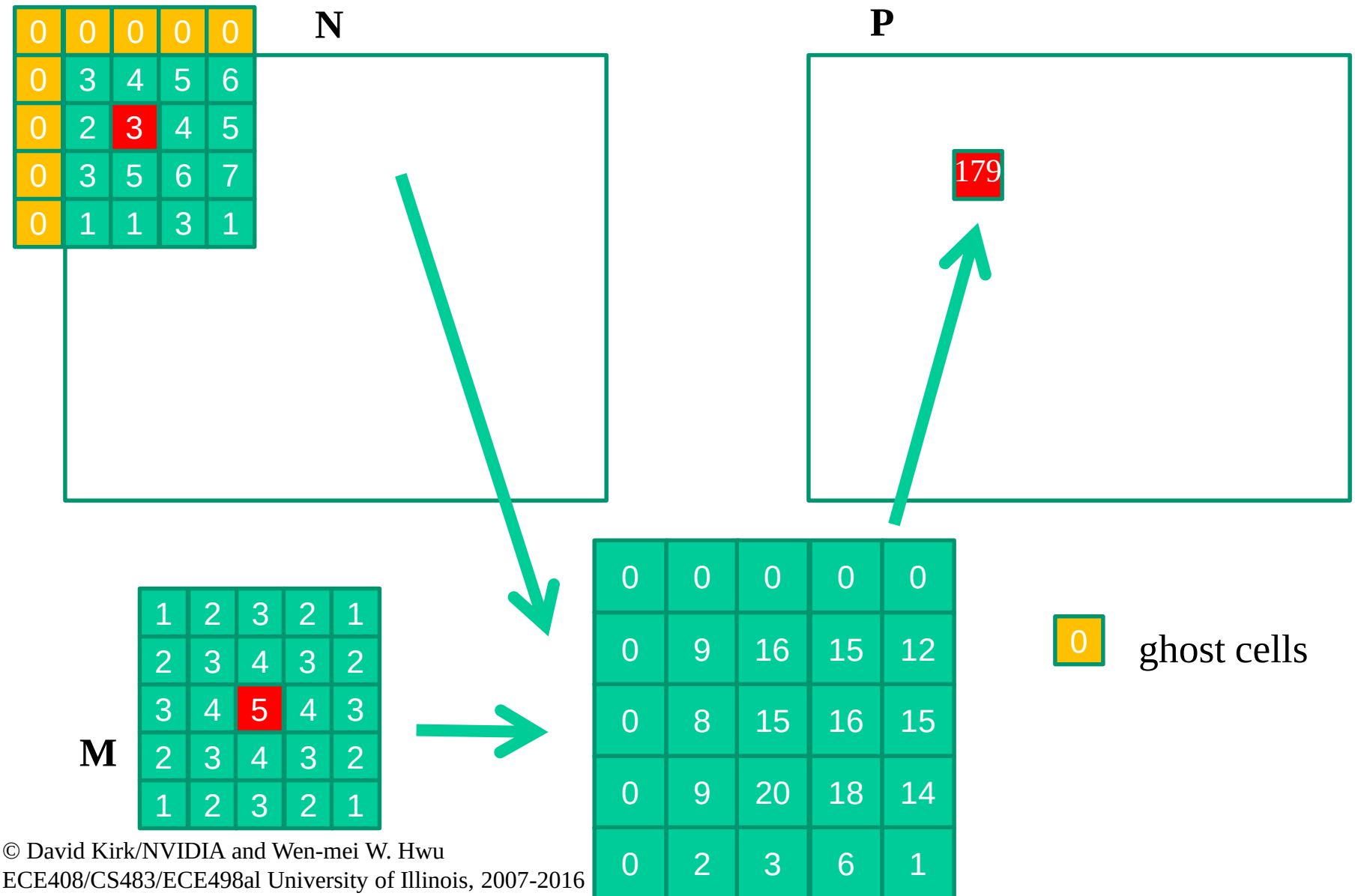
1	4	9	8	5
4	9	16	15	12
9	16	25	24	21
8	15	24	21	16
5	12	21	16	5

2D Convolution – Boundary Condition





2D Convolution - Ghost Cells



Access Pattern for M

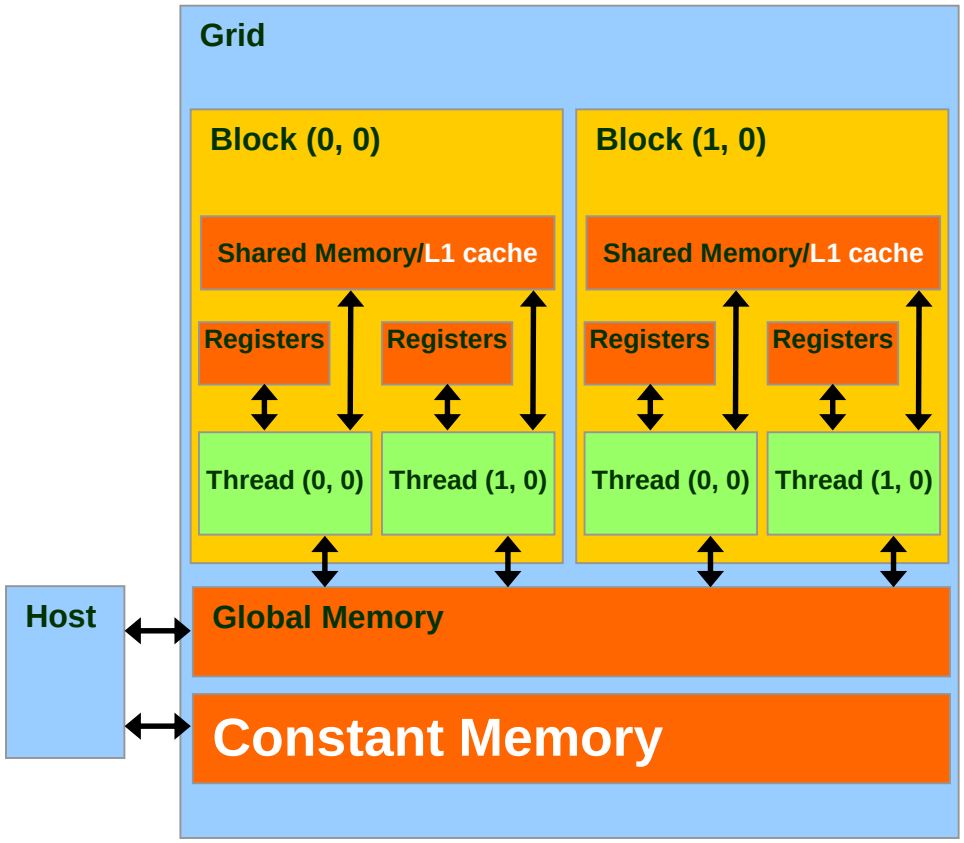
- M is referred to as mask (a.k.a. kernel, filter, etc.)
  - Elements of M are called mask (kernel, filter) coefficients
- Calculation of all output P elements need M
- M is not changed during kernel

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

- M is a good candidate for Constant Memory

Review of CUDA Memories

- Each thread can:
  - Read/write per-thread **registers (~1 cycle)**
  - Read/write per-block **shared memory (~5 cycles)**
  - Read/write per-grid **global memory (~500 cycles)**
  - Read/only per-grid **constant memory (~5 cycles with caching)**



Memory Hierarchies

- If we had to go to global memory to access data all the time, the execution speed of GPUs would be limited by the global memory bandwidth

- One solution: Caches

Cache

- A cache is an “array” of cache lines
  - A cache line can usually hold data from several consecutive memory addresses

- When data is requested from the global memory, an entire cache line that includes the data being accessed is loaded into the cache, in an attempt to reduce global memory requests
  - The data in the cache is a “copy” of the original data in global memory

Cache

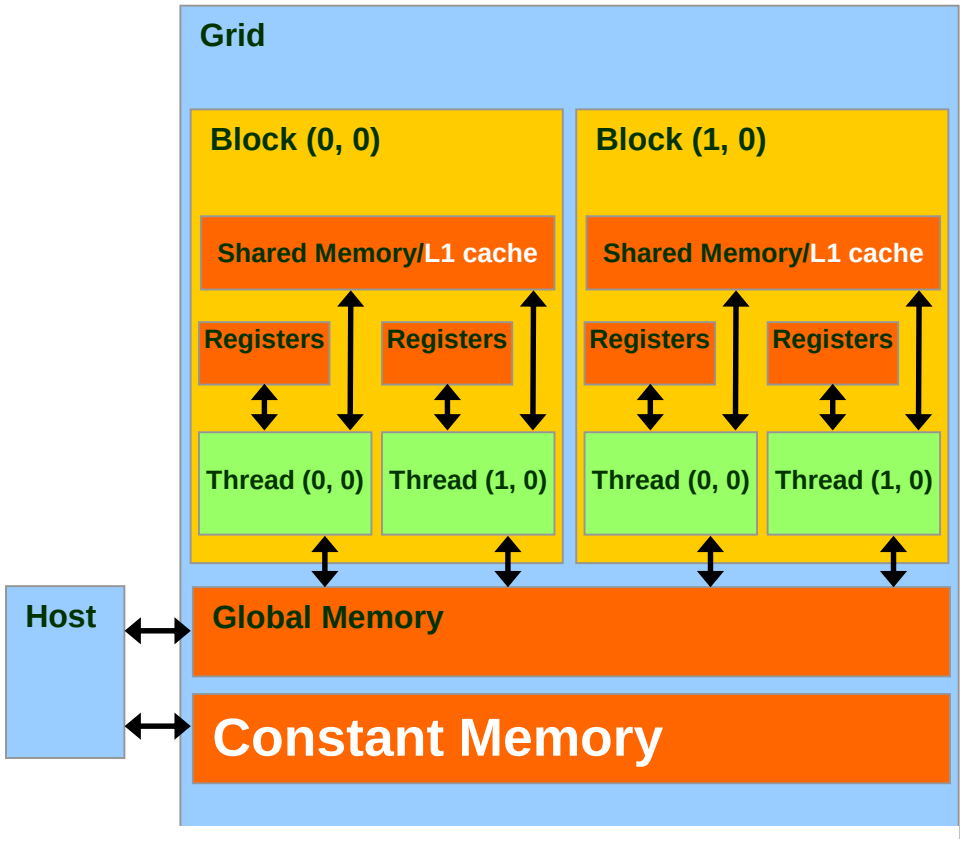
Some definitions:

- Spatial locality: when the data elements stored in consecutive memory locations are access consecutively
- Temporal locality: when the same data element is access multiple times in short period of time

- Both spatial locality and temporal locality improve the performance of caches

More on Constant Caching

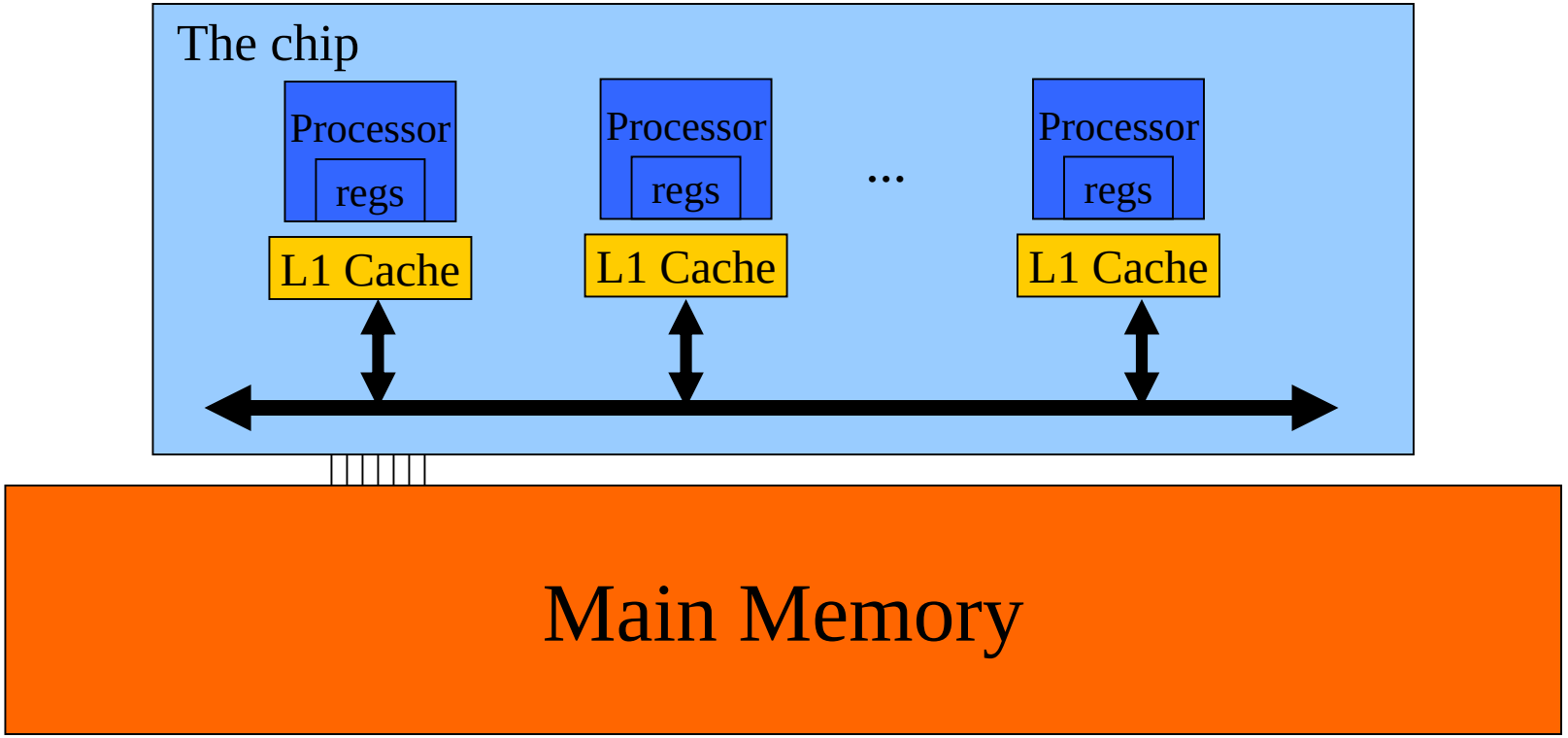
- Each SM has its own L1 cache
  - Low latency, high bandwidth access by all threads
- However, there is no way for threads in one SM to update the L1 cache in other SMs
  - No L1 cache coherence



This is not a problem if a variable is NOT modified by a kernel.

Cache Coherence Protocol

- A mechanism for caches to propagate updates by their local processor to other caches (processors)



CPU and GPU have different caching philosophy

- CPU L1 caches are usually coherent
  - L1 is also replicated for each core
  - Even data that will be changed can be cached in L1
  - Updates to local cache copy invalidate (or less commonly update) copies in other caches
  - Expensive in terms of hardware and disruption of services (cleaning bathrooms at airports..)

- GPU L1 caches are usually incoherent
  - Avoid caching data that will be modified

GPU Cache Coherence

- Current CUDA implementation:
  - Provides coherence by disabling L1 cache after writes
  - There is room for improvement
- Custom implementations
  - Temporal coherence: invalidates cache using synchronized counters without message passing
  - Stall writes to cache blocks until they have been invalidated in other caches

Scratchpad vs. Cache

- Scratchpad (shared memory in CUDA) is another type of temporary storage used to relieve main memory contention.
  - In terms of distance from the processor, scratchpad is similar to L1 cache
- Unlike cache, scratchpad does not necessarily hold a copy of data that is also in main memory
  - Scratchpad requires explicit data transfer instructions, whereas cache doesn't

Constant Cache in GPUs

- Modification to cached data needs to be (eventually) reflected back to the original data in global memory
  - Requires logic to track the modified status, etc.
- Constant cache 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, initializes variables the same way as any other variables that need to be copied to the device
- Use `cudaMemcpyToSymbol(dest,src,size)` to copy the variable into the device memory
  - Declare `__const__ float M[MASK_WIDTH]` first
- This copy function tells the device that the variable will not be modified by the kernel and can be safely cached

Header File for M

```
#define MASK_WIDTH 5

// Matrix Structure declaration
typedef struct {
    unsigned int width;
    unsigned int height;
    unsigned int pitch; // unused
    float* elements;
} Matrix;
```

AllocateMatrix

```
// Allocate a device matrix of dimensions height*width
//    If init == 0, initialize to all zeroes.
//    If init == 1, perform random initialization.
//    If init == 2, initialize matrix parameters, but
//        do not allocate memory
Matrix AllocateMatrix(int height, int width, int init)
{
    Matrix M;
    M.width = M.pitch = width;
    M.height = height;
    int size = M.width * M.height;
    M.elements = NULL;
```

AllocateMatrix

```
// don't allocate memory on option 2
  if(init == 2) return M;
  int size = height * width;
  M.elements = (float*) malloc(size*sizeof(float));
  for(unsigned int i = 0; i < M.height * M.width; i++)
  {
    M.elements[i] = (init == 0) ? (0.0f) :
              (rand() / (float)RAND_MAX);
   if(rand() % 2) M.elements[i] = - M.elements[i]
  }
return M;
}
```

Host Code

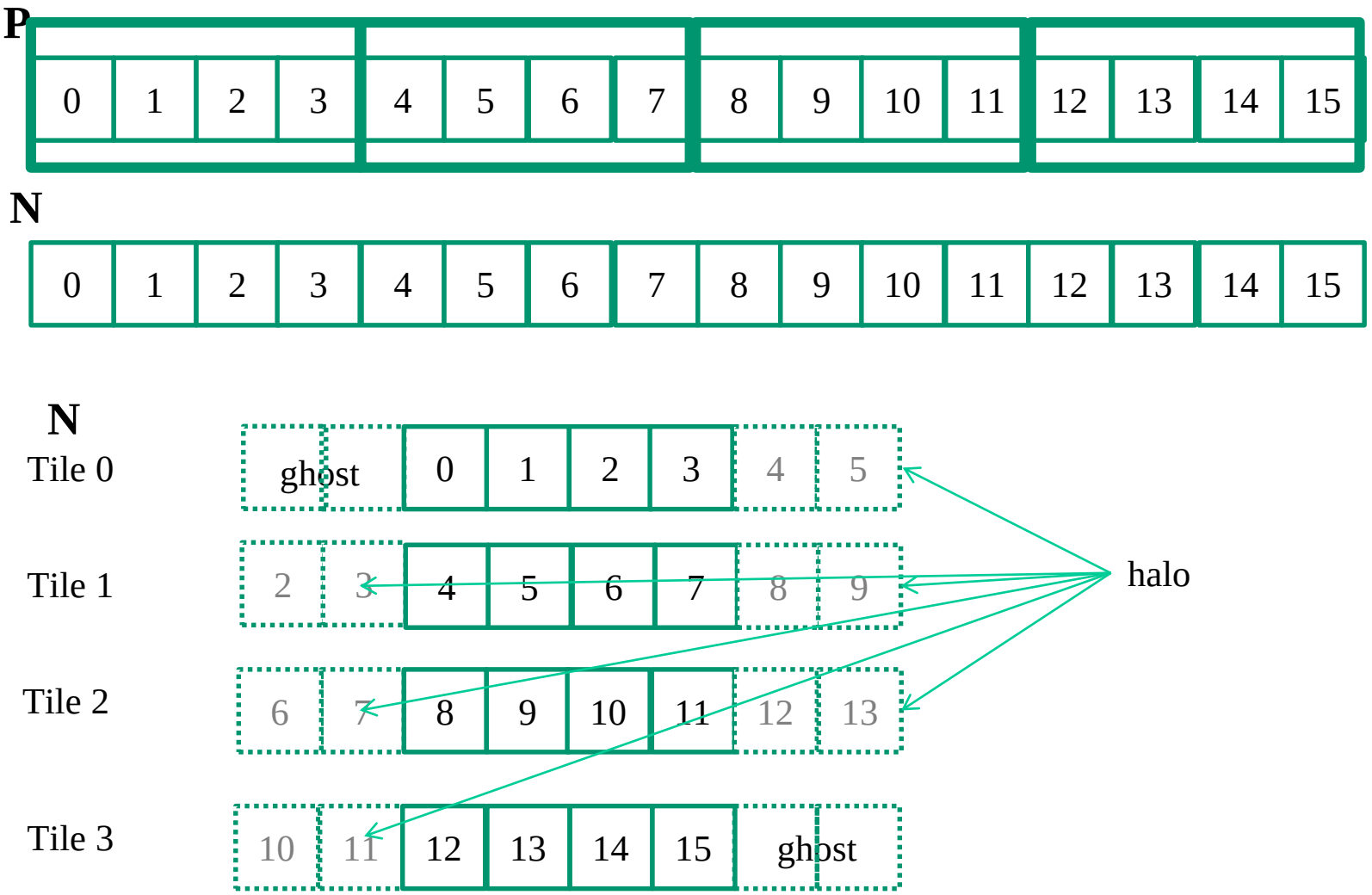
```
// global variable, outside any kernel/function
   __constant__ float Mc[MASK_WIDTH][MASK_WIDTH];
…
   // allocate N, P, initialize N elements, copy N to Nd
   Matrix  M;
   M  = AllocateMatrix(MASK_WIDTH, MASK_WIDTH, 1);
   // initialize M elements
….
   cudaMemcpyToSymbol(Mc, M.elements,
            MASK_WIDTH*MASK_WIDTH*sizeof(float));
   ConvolutionKernel<<<dimGrid, dimBlock>>>(Nd, Pd);
```

Tiled 1D Convolution

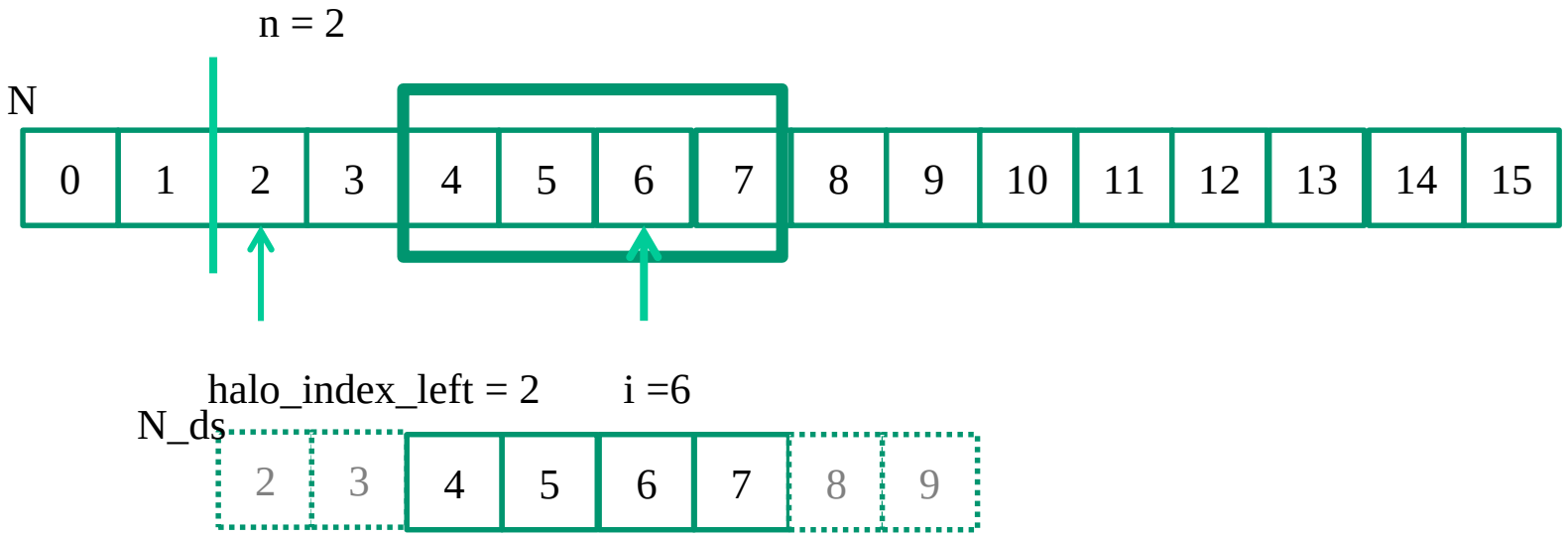
- Elements of the input vector are used in multiple computations
- Opportunity to use **shared memory**

- Shared memory tile must be larger than mask

Tiled 1D Convolution Basic Idea

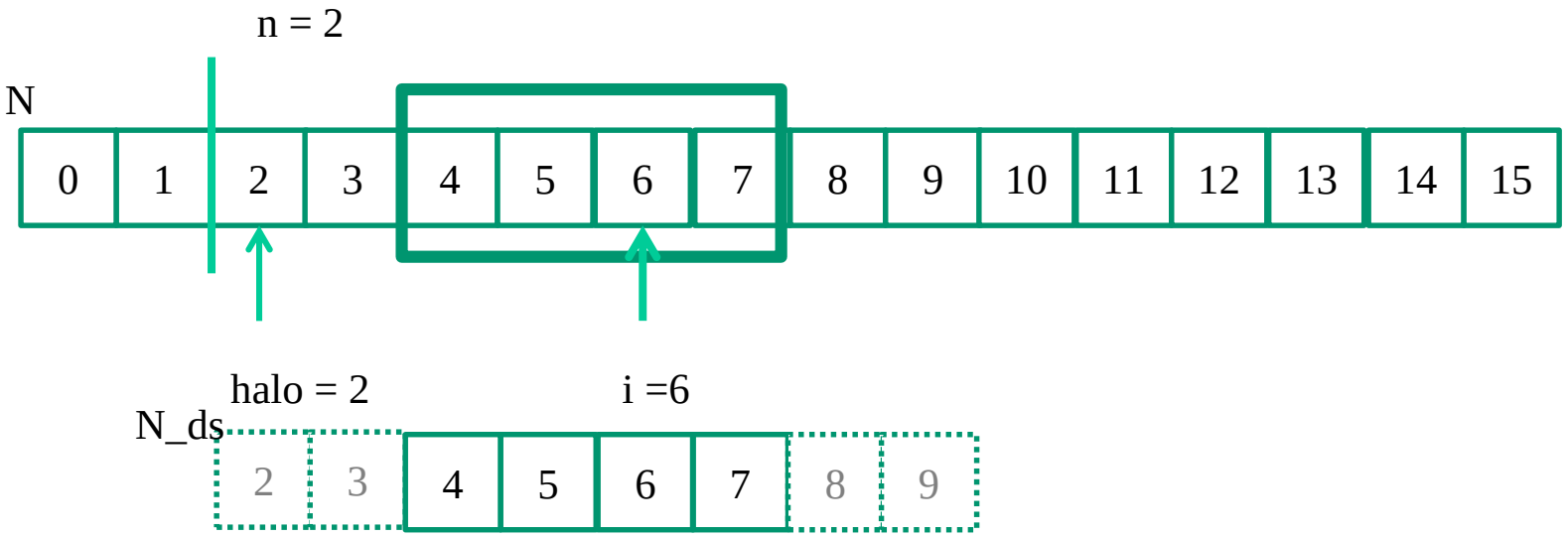


Loading Left Halo



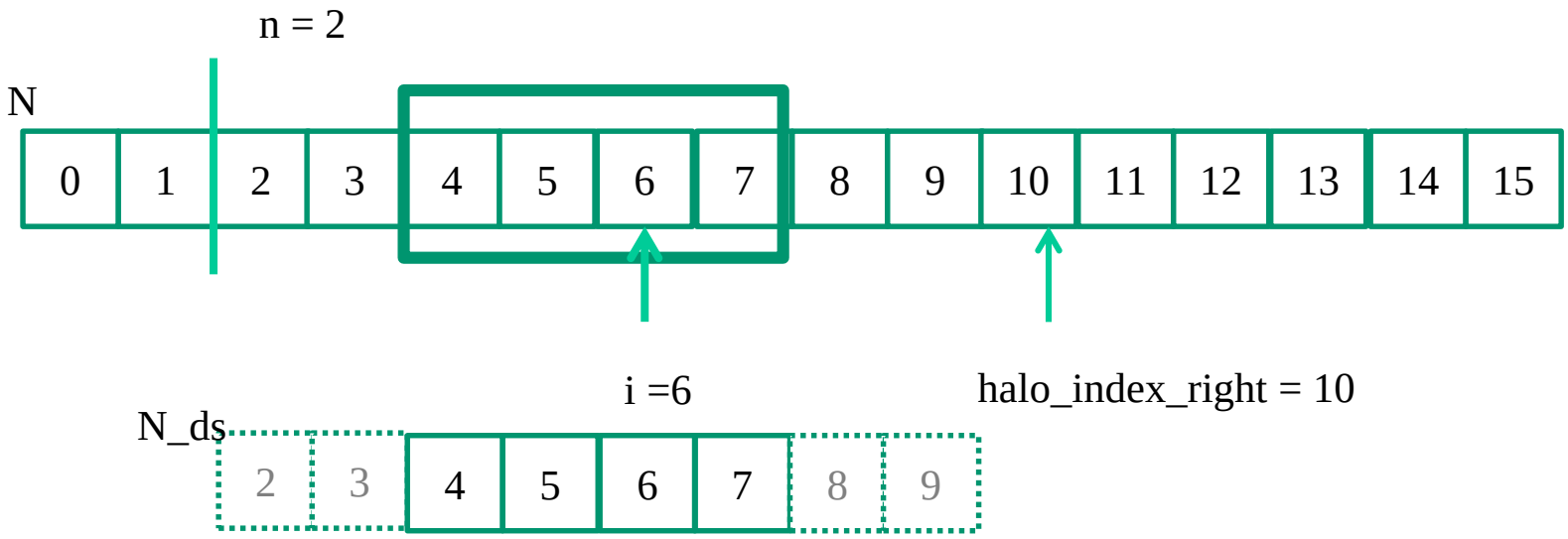

```
int n = Mask_Width/2;
int halo_index_left = (blockIdx.x - 1)*blockDim.x + threadIdx.x;
  if (threadIdx.x >= blockDim.x - n) {
    N_ds[threadIdx.x - (blockDim.x - n)] =
      (halo_index_left < 0) ? 0 : N[halo_index_left];
  }
```

Loading Internal Elements




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

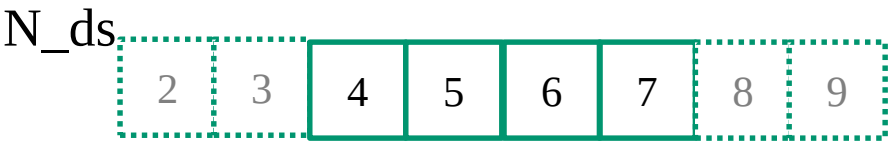
Loading Right Halo




```
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];
  }
```

```
__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;

}
```

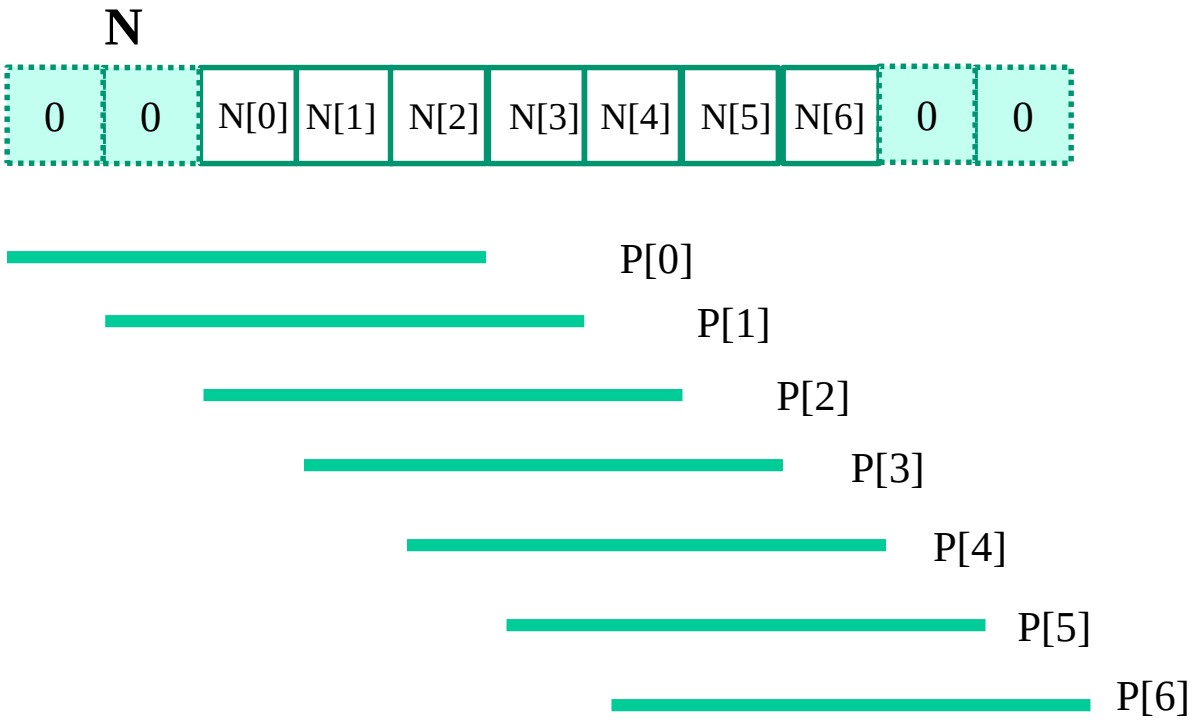
Shared Memory Data Reuse



Mask_Width is 5

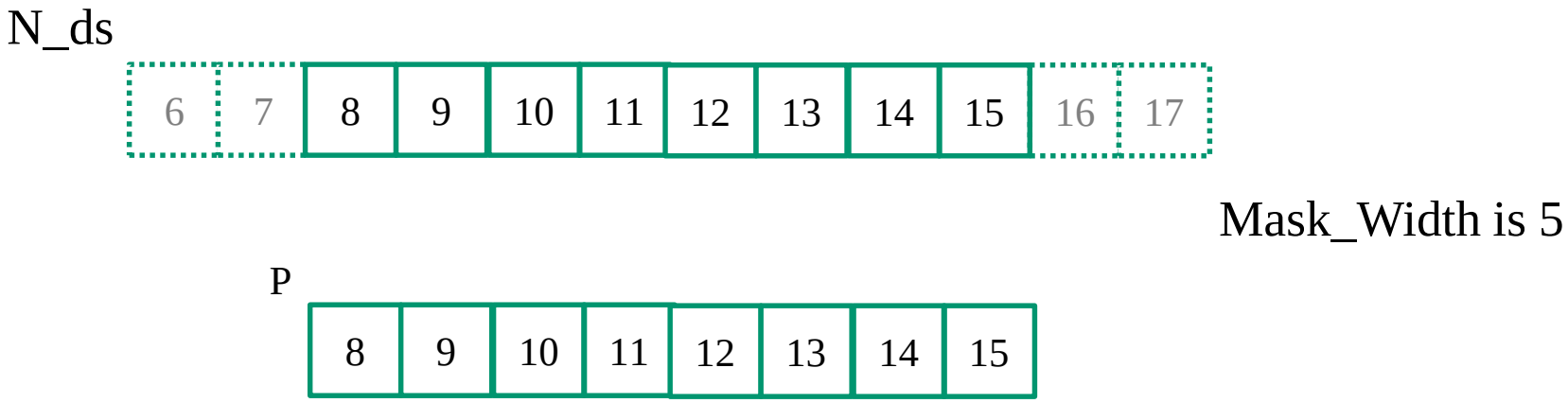
- Element 2 is used by thread 4 (1X)
- Element 3 is used by threads 4, 5 (2X)
- Element 4 is used by threads 4, 5, 6 (3X)
- Element 5 is used by threads 4, 5, 6, 7 (4X)
- Element 6 is used by threads 4, 5, 6, 7 (4X)
- Element 7 is used by threads 5, 6, 7 (3X)
- Element 8 is used by threads 6, 7 (2X)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

Ghost Cells



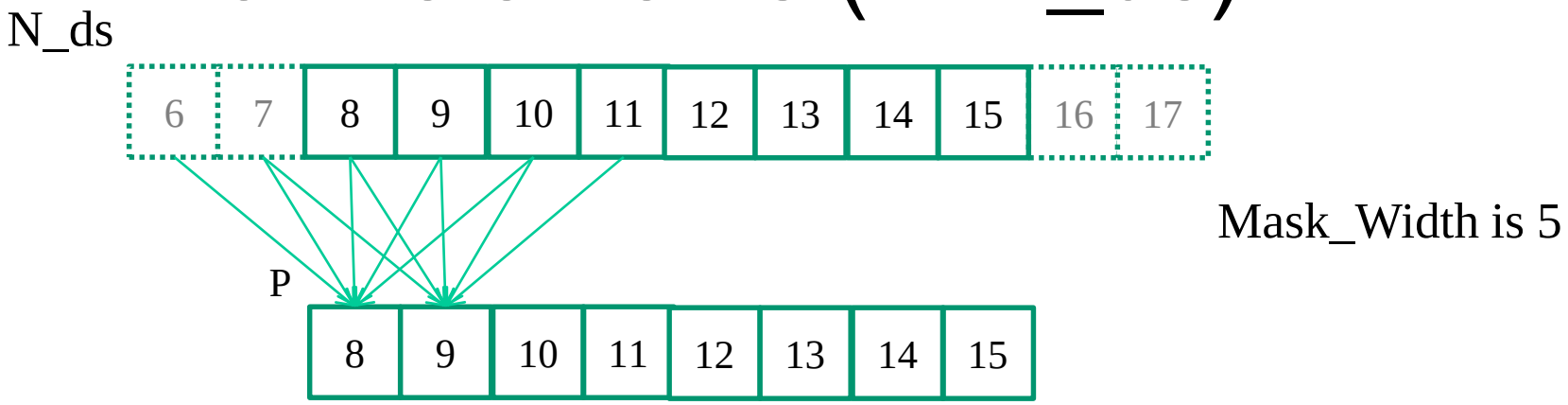
```
__global__ void convolution_1D_tiled_cache_kernel(float *N, float *P,
int Mask_Width, int Width) {

  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;

}
```

Analysis - Small 1D Example



- TILE_SIZE = 8, Mask_Width=5
- Output and input tiles for block 1
- For Mask_Width = 5, each block loads 8+5-1 = 12 elements (12 memory loads)

Each output P element uses 5 N 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.

A simple way to calculate tiling benefit

- (8+5-1)=12 elements loaded
- 8*5 global memory accesses replaced by shared memory accesses
- This gives a bandwidth reduction of 40/12=3.3

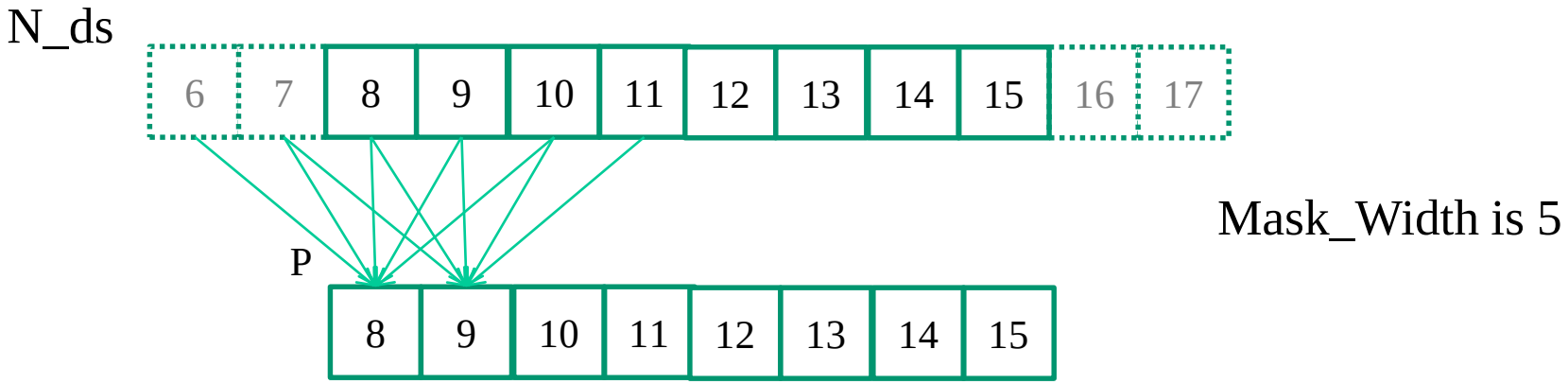
In General, in 1D

- TILE_SIZE + Mask_Width -1 elements loaded
- TILE_SIZE * Mask_Width global memory accesses replaced by shared memory access

- This gives a reduction of bandwidth by

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

Another Way to Look at Reuse



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

Another Way to Look at Reuse

- Each time an N_ds element is used, it replaces an access to the global memory N element
- The total number of global memory accesses (to the (8+5-1)=12 N elements) replaced by shared memory accesses is

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

= 10 + 20 + 10

= 40

So the reduction is

40/12 = 3.3

Ghost Elements

- For a boundary tile, we load
  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

In General for 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 in 1D

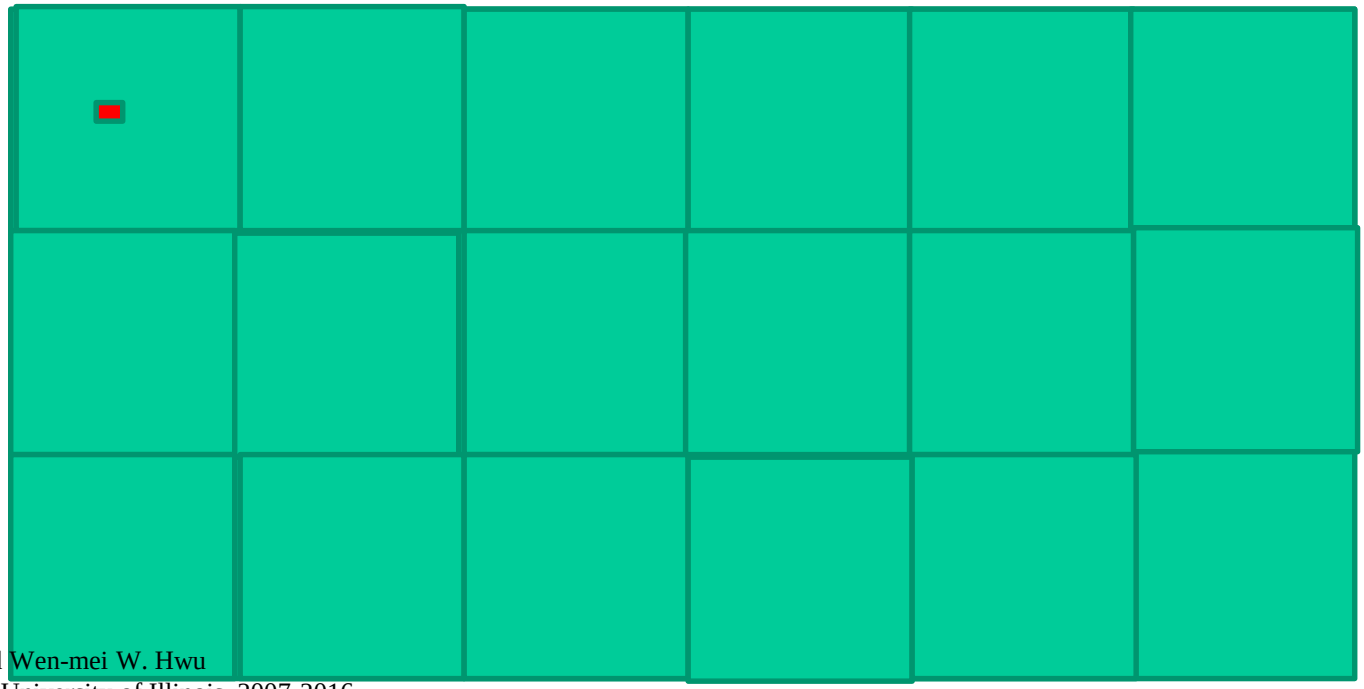
- The reduction is

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

Tile_Width	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

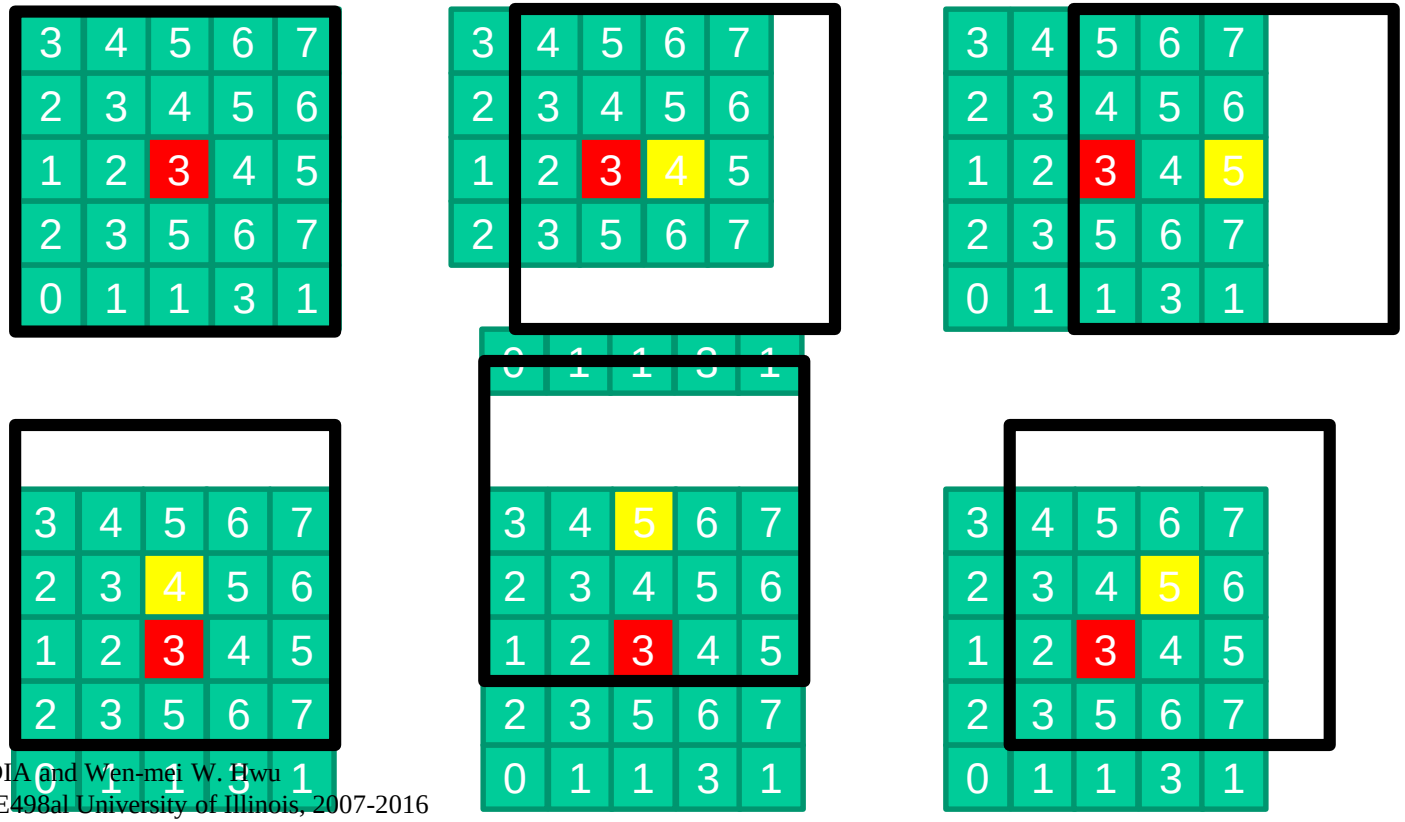
Tiling P

- Use a thread block to calculate a tile of P
  - Each output tile is of TILE_SIZE for both x and y
  - `row_o = blockIdx.y*TILE_SIZE + ty;`
  - `col_o = blockIdx.x*TILE_SIZE + tx;`

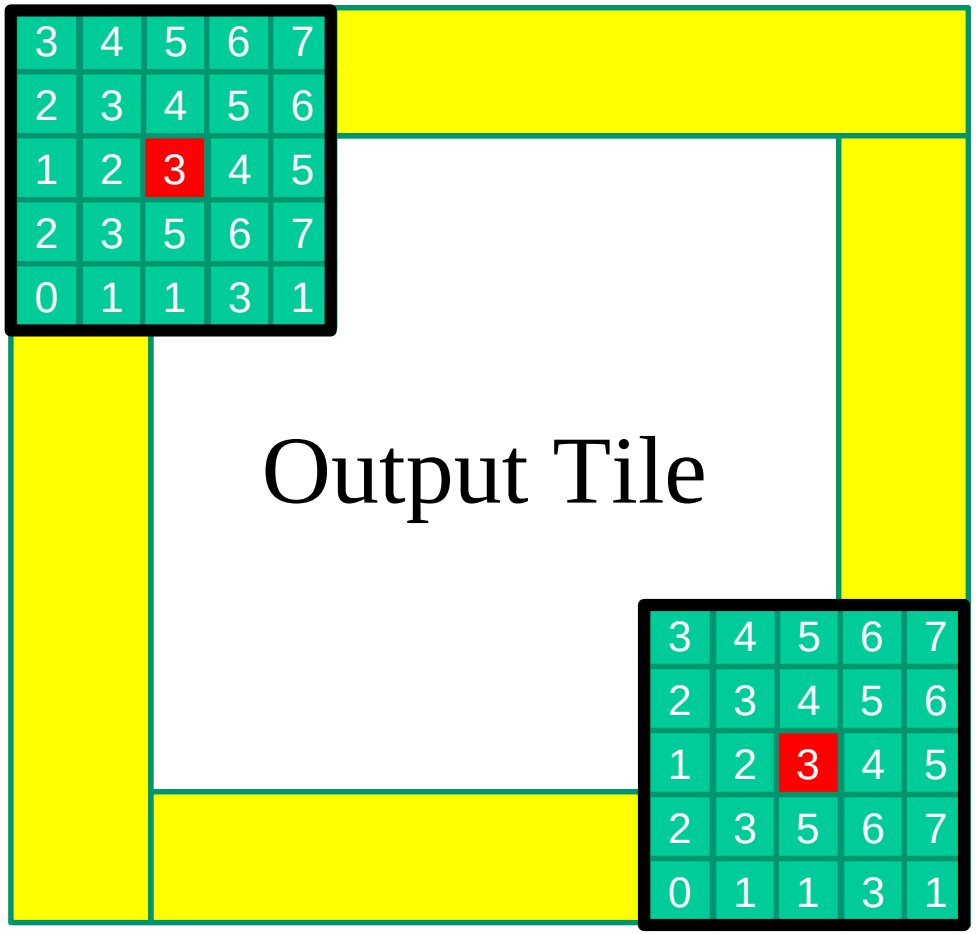


Tiling N

- Each N element is used in calculating up to KERNEL_SIZE * KERNEL_SIZE P elements (all elements in the tile)



Input tiles need to be larger than output tiles



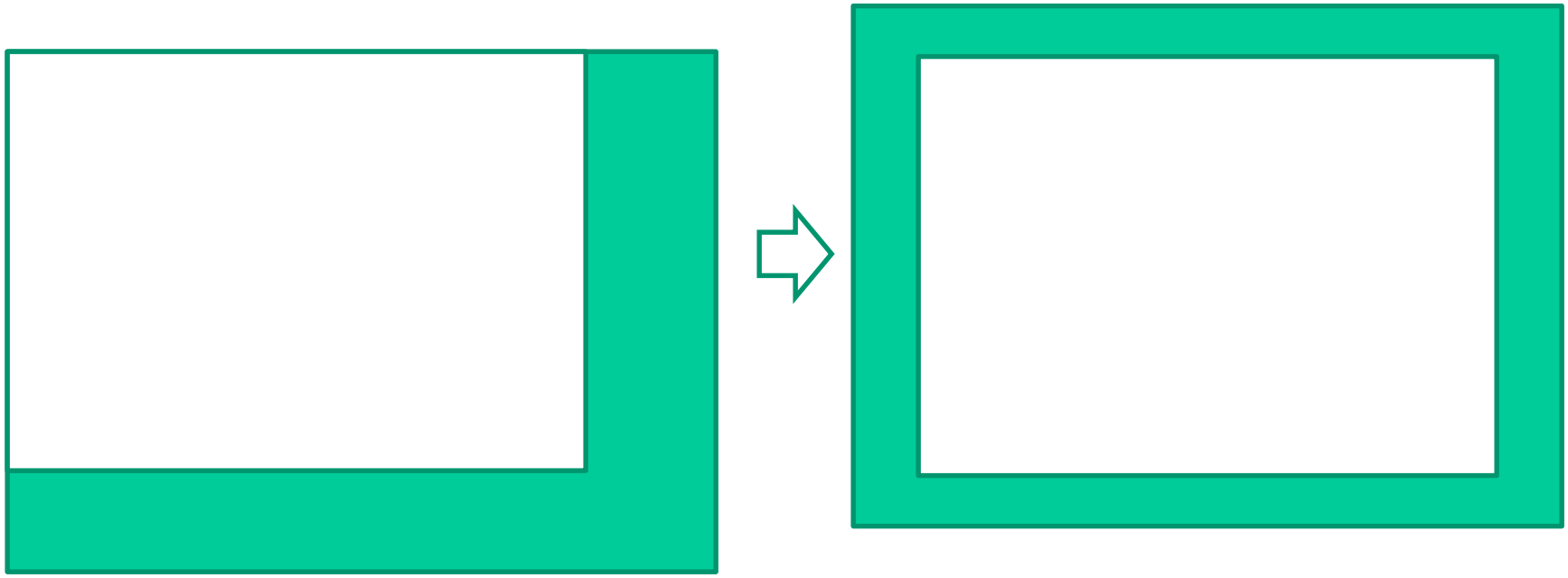
← Input Tile

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 input tile
  - Each thread loads one element of the input tile
  - Some threads do not participate in calculating output
    - There will be if statements and control divergence

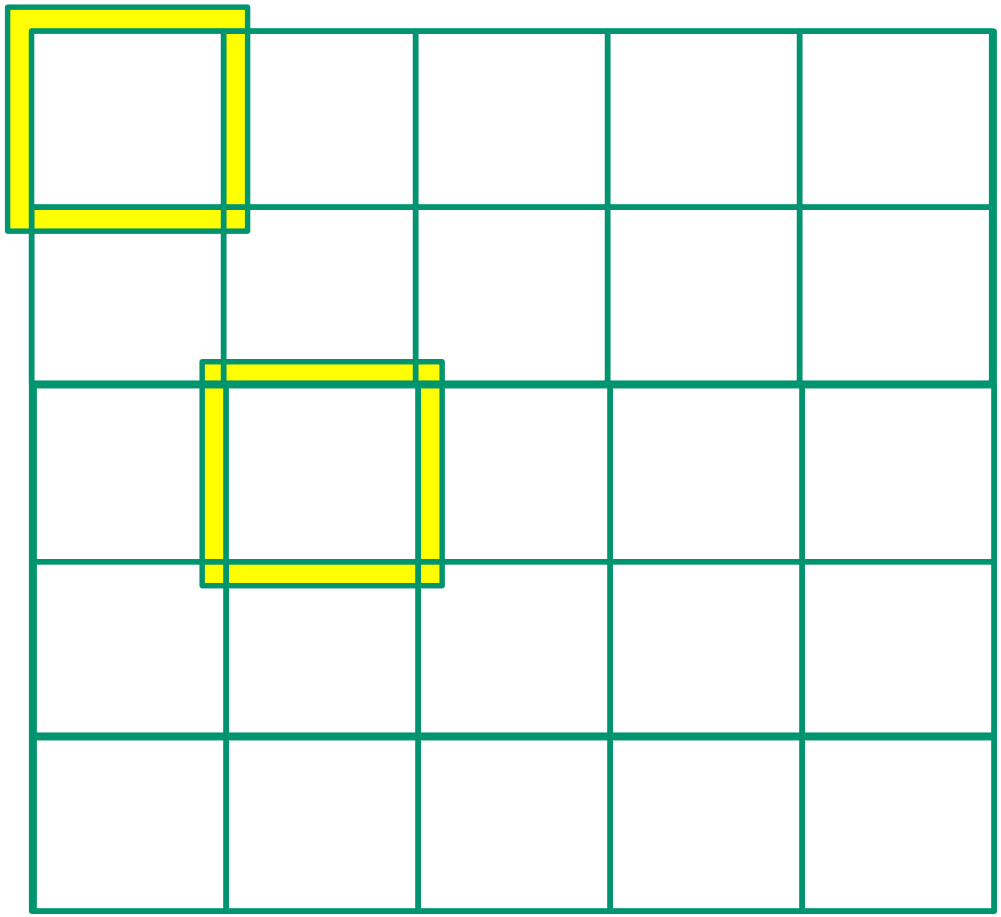
Shifting from output coordinates to input coordinates



Shifting from output coordinates to input coordinates

```
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;
```

Threads that loads halos outside N should return 0.0



Taking Care of Boundaries

```
float output = 0.0f;

  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;
  }
```

Some threads do not participate in calculating 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];
        }
      }
```

Some threads do not write output

```
 if(row_o < P.height && col_o < P.width)
     P.elements[row_o * P.width + col_o] =
                          output;
}
```

Setting Block Size

```
#define BLOCK_SIZE (TILE_SIZE + 4)

dim3 dimBlock(BLOCK_SIZE,BLOCK_SIZE);
```

In general, block size should be tile size + (kernel size -1)

```
dim3 dimGrid(N.width/TILE_SIZE,
    N.height/TILE_SIZE, 1)
```

More on Sizes

- BLOCK_SIZE is limited by the maximal number of threads in a thread block
- Input tile sizes could be could be N*TILE_SIZE + (KERNEL_SIZE-1)
  - By having each thread calculate N input points (thread coarsening)
  - N is limited is limited by the shared memory size
- KERNEL_SIZE is decided by application needs

8x8 Output Tile

- KERNEL_SIZE = 5
- 12X12=144 N elements need to be loaded into shared memory
- The calculation of each P element needs to access 25 N elements
- 8X8X25 = 1600 global memory accesses are converted into shared memory accesses
- A reduction of 1600/144 = 11X

In General in 2D

- $(TILE\_SIZE + KERNEL\_SIZE - 1)^2$ N elements need to be loaded into shared memory
- The calculation of each P element needs to access $KERNEL\_SIZE^2$ N elements
- $TILE\_SIZE^2 * KERNEL\_SIZE^2$ global memory accesses are converted into shared memory accesses
- The reduction is

$$TILE\_SIZE^2 * KERNEL\_SIZE^2 / (TILE\_SIZE + KERNEL\_SIZE - 1)^2$$

Bandwidth Reduction in 2D

- The reduction is

$$\text{TILE\_SIZE}^2 * \text{KERNEL\_SIZE}^2 / (\text{TILE\_SIZE} + \text{KERNEL\_SIZE} - 1)^2$$

TILE_SIZE	8	16	32	64
Reduction KERNEL_SIZE = 5	11.1	16	19.7	22.1
Reduction KERNEL_SIZE = 9	20.3	36	51.8	64

Reduction Trees

Partition and Summarize

- A commonly used strategy for processing large input data sets
  - There is no required order of processing elements in a data set (associative and commutative)
  - Partition the data set into smaller chunks
  - Have each thread to process a chunk
  - Use a reduction tree to summarize the results from each chunk into the final answer
- We will focus on the reduction tree step for now
- Google and Hadoop MapReduce frameworks are examples of this pattern

Reduction enables other techniques

- Reduction is also needed to clean up after some commonly used parallelizing transformations

- Privatization
  - Multiple threads write into an output location
  - Replicate the output location so that each thread has a private output location
  - Use a reduction tree to combine the values of private locations into the original output location

What is a reduction computation

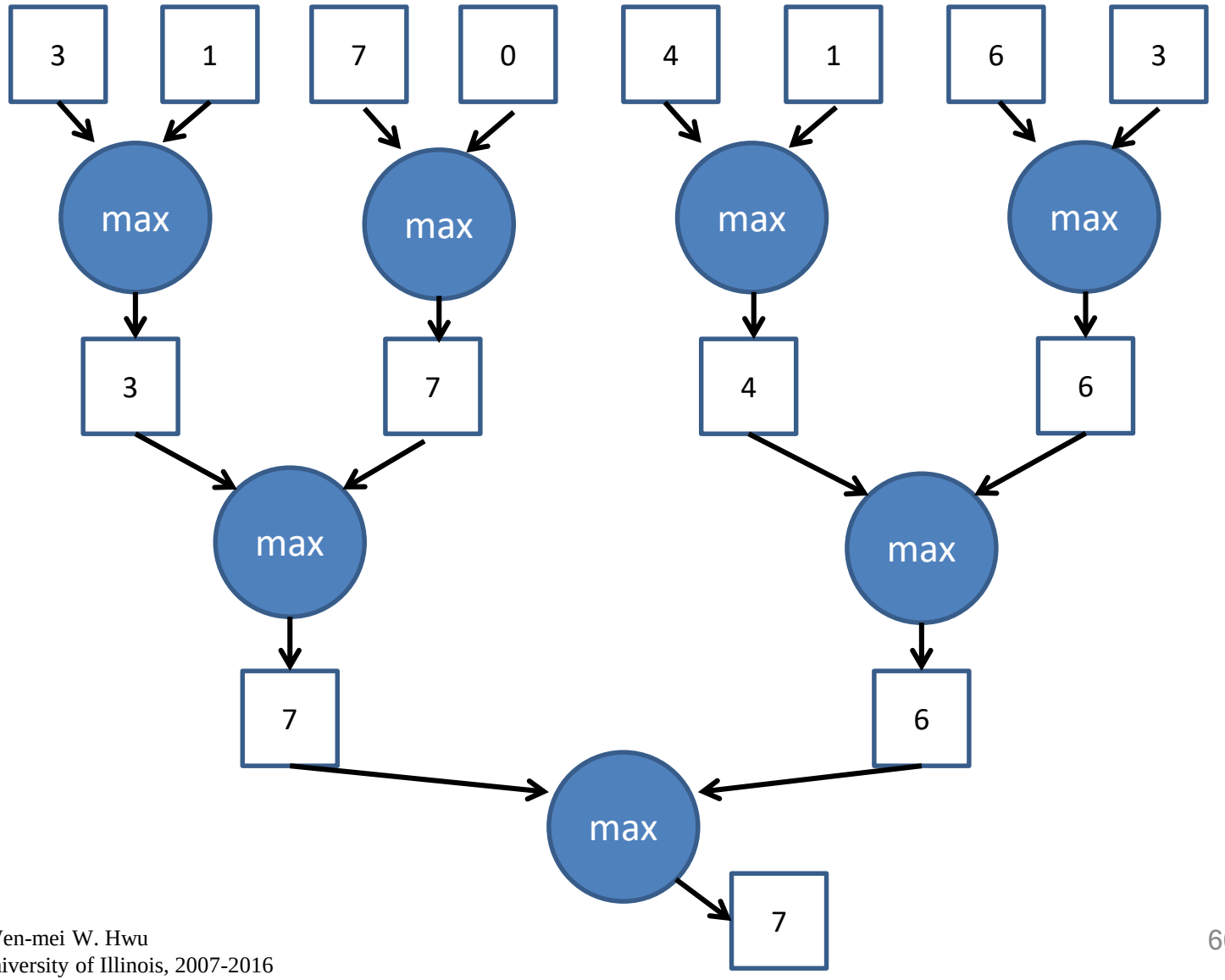
- Summarize a set of input values into one value using a “reduction operation”
  - Max
  - Min
  - Sum
  - Product
  - Often with user defined reduction operation function as long as the operation
    - Is associative and commutative
    - Has a well-defined identity value (e.g., 0 for sum)

A sequential reduction algorithm performs N operations - O(N)

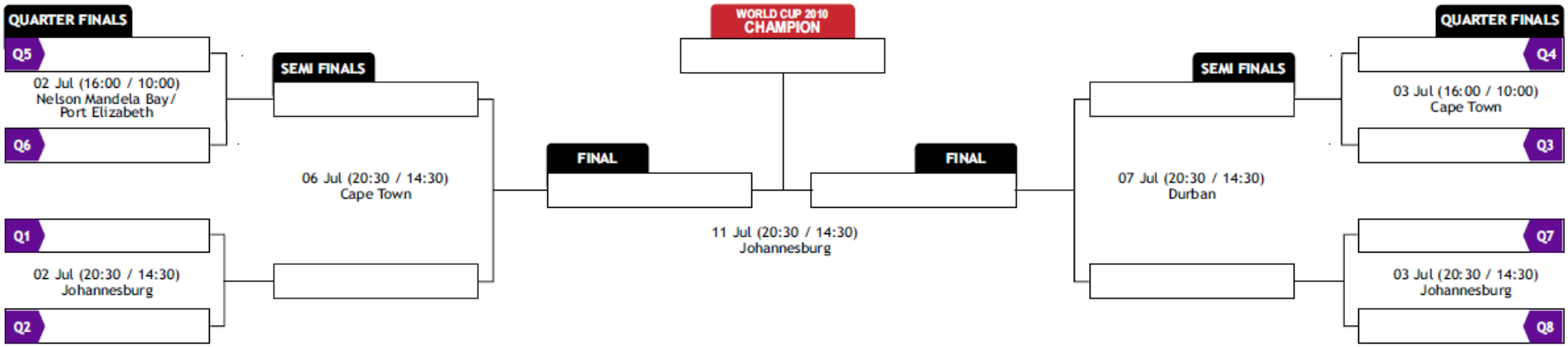
- Initialize the result as an identity value for the reduction operation
  - Smallest possible value for max reduction
  - Largest possible value for min reduction
  - 0 for sum reduction
  - 1 for product reduction

- Scan through the input and perform the reduction operation between the result value and the current input value

A parallel reduction tree algorithm performs N-1 Operations in log(N) steps



A tournament is a reduction tree with “max” operation



A Quick Analysis

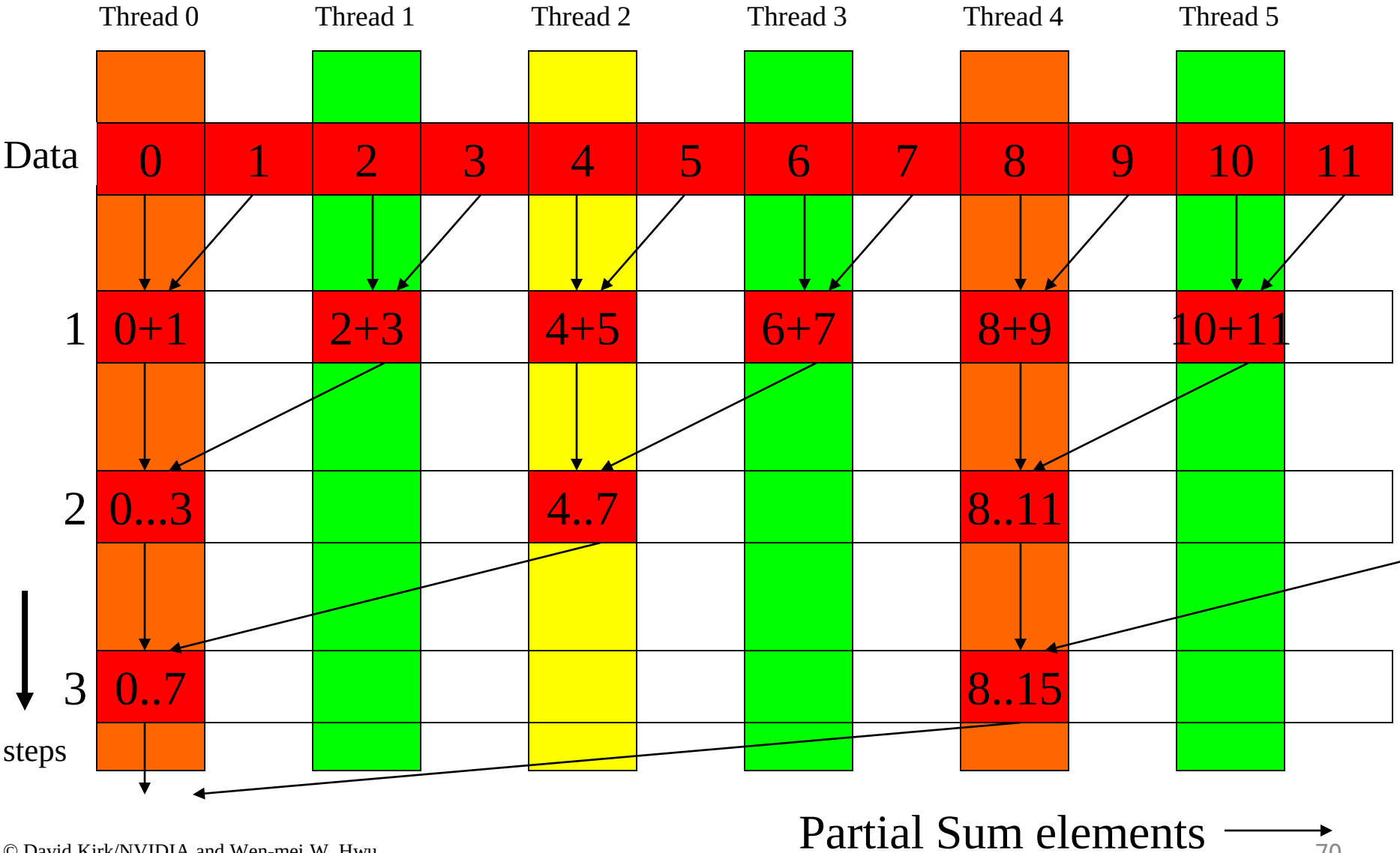
- For N input values, the reduction tree performs
  - (1/2)N + (1/4)N + (1/8)N + … (1/N) = (1- (1/N))N = N-1 operations
  - In Log (N) steps – 1,000,000 input values take 20 steps
    - Assuming that we have enough execution resources
  - Average Parallelism (N-1)/Log(N))
    - For N = 1,000,000, average parallelism is 50,000
    - However, peak resource requirement is 500,000!
- This is a work-efficient parallel algorithm
  - The amount of work done is comparable to sequential
  - Many parallel algorithms are not work efficient

A Sum Reduction Example

- Parallel implementation:
  - Recursively halve # of threads, add two values per thread in each step
  - Takes log(n) steps for n elements, requires n/2 threads

- Assume an in-place reduction using shared memory
  - The original vector is in device global memory
  - The shared memory is used to hold a partial sum vector
  - Each step brings the partial sum vector closer to the sum
  - The final sum will be in element 0
  - Reduces global memory traffic due to partial sum values

Vector Reduction with Branch Divergence



Some Observations

- In each iteration, two control flow paths will be sequentially traversed for each warp
  - Threads that perform addition and threads that do not
  - Threads that do not perform addition still consume execution resources

- No more than half of threads will be executing after the first step
  - All odd index threads are disabled after first step
  - After the 5^{th} step, entire warps in each block will fail the if test, poor resource utilization but no divergence.
    - This can go on for a while, up to 5 more steps (1024/32=16= 2^5), where each active warp only has one productive thread until all warps in a block retire

Thread Index Usage Matters

- In some algorithms, one can shift the index usage to improve the divergence behavior
  - Commutative and associative operators

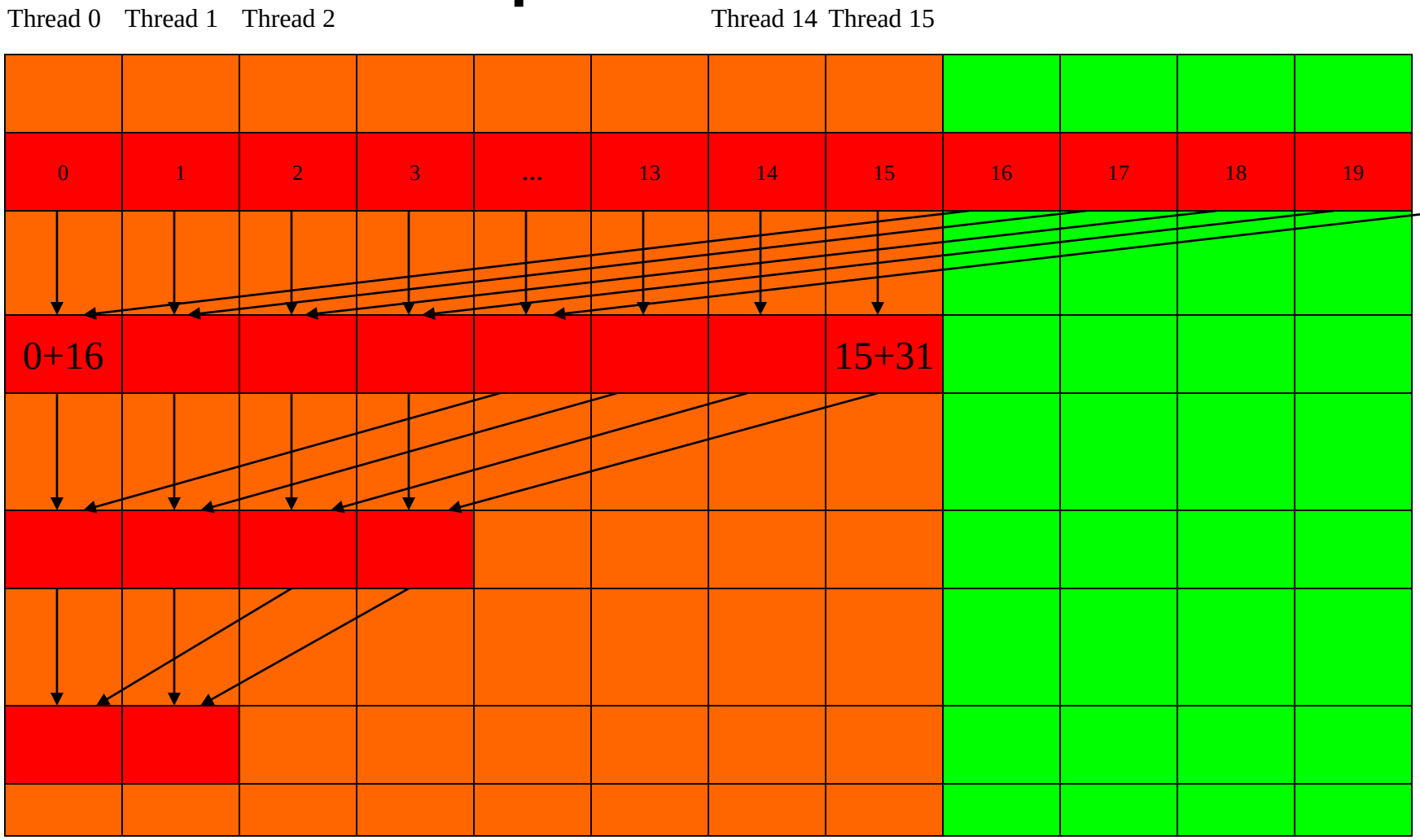
- Reduction satisfies this criterion

A Better Strategy

- Always compact the partial sums into the first locations in the partialSum[] array

- Keep the active threads consecutive

An Example of 16 threads



A Better Reduction Kernel

```
for (unsigned int stride = blockDim.x/2;
     stride >= 1;  stride >>= 1)
{
  __syncthreads();
  if (t < stride)
    partialSum[t] += partialSum[t+stride];
}
```

A Quick Analysis

- For a 1024 thread block
  - No divergence in the first 5 steps
  - 1024, 512, 256, 128, 64, 32 consecutive threads are active in each step
  - The final 5 steps will still have divergence

Parallel Algorithm Overhead

```
__shared__ float partialSum[2*BLOCK_SIZE];

unsigned int t = threadIdx.x;
unsigned int start = 2*blockIdx.x*blockDim.x;
partialSum[t] = input[start + t];
partialSum[blockDim+t] = input[start+ blockDim.x+t];
 for (unsigned int stride = blockDim.x/2;
      stride >= 1;  stride >>= 1)
 {
   __syncthreads();
   if (t < stride)
      partialSum[t] += partialSum[t+stride];
 }
```

Parallel Algorithm Overhead

```
__shared__ float partialSum[2*BLOCK_SIZE];

unsigned int t = threadIdx.x;
unsigned int start = 2*blockIdx.x*blockDim.x;
partialSum[t] = input[start + t];
partialSum[blockDim+t] = input[start+ blockDim.x+t];
 for (unsigned int stride = blockDim.x/2;
      stride >= 1;  stride >>= 1)
 {
   __syncthreads();
   if (t < stride)
      partialSum[t] += partialSum[t+stride];
 }
```

Parallel Execution Overhead

- Although the number of “operations” is N, each operation involves much more complex address calculation and intermediate result manipulation

- If the parallel code is executed on a single-thread hardware, it would be significantly slower than the code based on the original sequential algorithm

Parallel Prefix Sum (Scan)

Objectives

- Prefix Sum (Scan) algorithms
  - Frequently used for parallel work assignment and resource allocation
  - A key primitive in many parallel algorithms to convert serial computation into parallel computation
  - Based on reduction tree and reverse reduction tree
- To learn the concept of double buffering

(Inclusive) Prefix-Sum (Scan) Definition

Definition: The all-prefix-sums operation takes a binary associative operator $\oplus$, and an array of n elements

$$[x_0, x_1, \ldots, x_{n-1}],$$

and returns the array

$$[x_0, (x_0 \oplus x_1), \ldots, (x_0 \oplus x_1 \oplus \ldots \oplus x_{n-1})].$$

Example: If $\oplus$ is addition, then the all-prefix-sums operation on the array [3 1 7 0 4 1 6 3], would return [3 4 11 11 15 16 22 25].

A Inclusive Scan Application Example

- Assume that we have a 100-inch bread to feed 10 people
- We know how much each person wants in inches
  - [3 5 2 7 28 4 3 0 8 1]
- How do we cut the bread quickly?
- How much will be left

- Method 1: cut the sections sequentially: 3 inches first, 5 inches second, 2 inches third, etc.
- Method 2: calculate Prefix scan
  - [3, 8, 10, 17, 45, 49, 52, 52, 60, 61] (39 inches left)

Typical Applications of Scan

- Assigning camp slots
- Assigning farmer market space
- Allocating memory to parallel threads
- Allocating memory buffer to communication channels
- Useful for many parallel algorithms:
  - radix sort
  - quicksort
  - String comparison
  - Lexical analysis
  - Stream compaction
  - Polynomial evaluation
  - Solving recurrences
  - Tree operations
  - Histograms
  - Etc.

A Inclusive Sequential Prefix-Sum

Given a sequence $[x_0, x_1, x_2, \ldots]$

Calculate output $[y_0, y_1, y_2, \ldots]$

Such that

$$y_0 = x_0$$
$$y_1 = x_0 + x_1$$
$$y_2 = x_0 + x_1 + x_2$$
$$\ldots$$

Using a recursive definition

$$y_i = y_{i-1} + x_i$$

A Work Efficient C Implementation

```
y[0] = x[0];
for (i = 1; i < Max_i; i++)
   y[i] = y [i-1] + x[i];
```

Computationally efficient:

N additions needed for N elements - O(N)

A Naïve Inclusive Parallel Scan

- Assign one thread to calculate each y element
- Have every thread to add up all x elements needed for the y element

$$y_0 = x_0$$

$$y_1 = x_0 + x_1$$

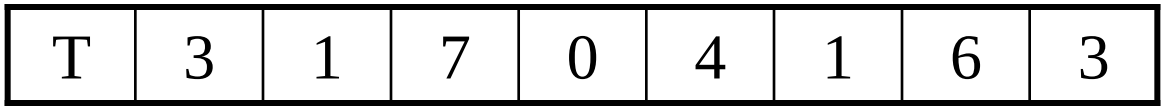
$$y_2 = x_0 + x_1 + x_2$$

"Parallel programming is easy as long as you do not care about performance."

Parallel Inclusive Scan using Reduction Trees

- Calculate each output element as the reduction of all previous elements
  - Some reduction partial sums will be shared among the calculation of output elements
  - Based on hardware added design by Peter Kogge and Harold Stone at IBM in the 1970s – Kogge-Stone Trees

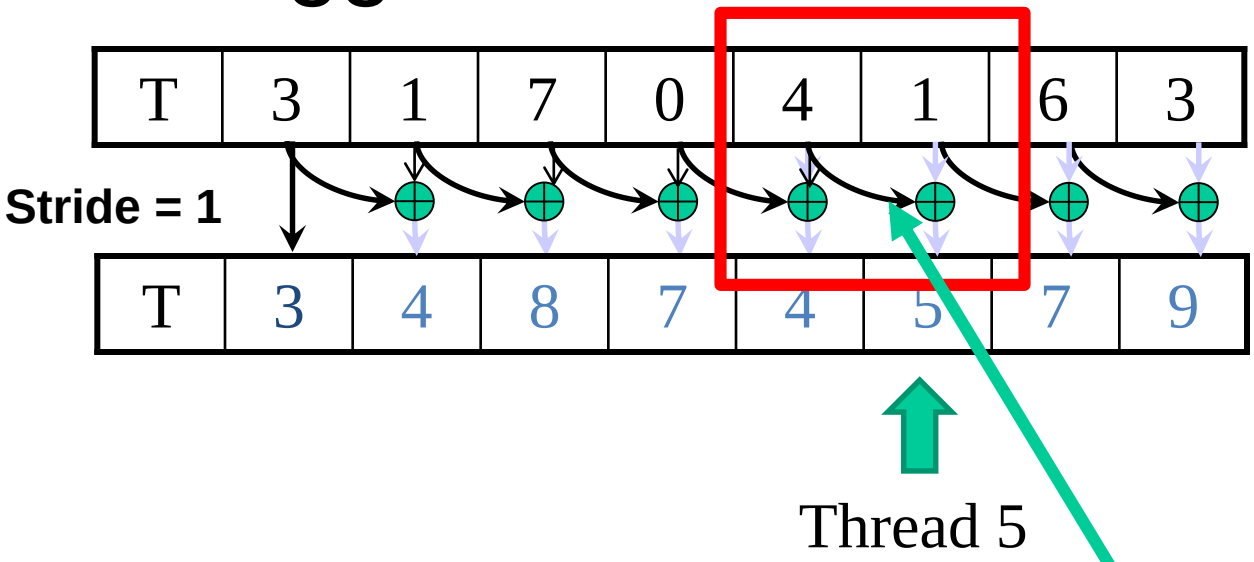
A Slightly Better Parallel Inclusive Scan Algorithm



1. Load input from global memory into shared memory array T

Each thread loads one value from the input (global memory) array into shared memory array T.

A Kogge-Stone Parallel Scan Algorithm

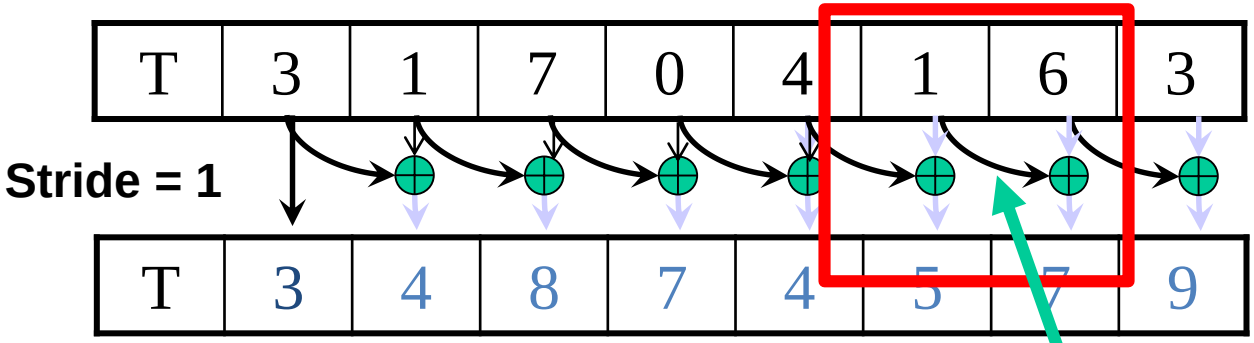


1. (previous slide)
2. Iterate log(n) times, stride from 1 to ceil(n/2.0). Threads *stride* to *n-1 active:* add pairs of elements that are s*tride* elements apart.

- Active threads: *stride* to *n*-1 (*n-stride* threads)
- Thread *j* adds elements *j* and *j-stride* from T and writes result into shared memory buffer T
- Each iteration requires two syncthreads
  - syncthreads(); // make sure that input is in place
  - float temp = T[j] + T[k - stride];

Iteration #1
Stride = 1

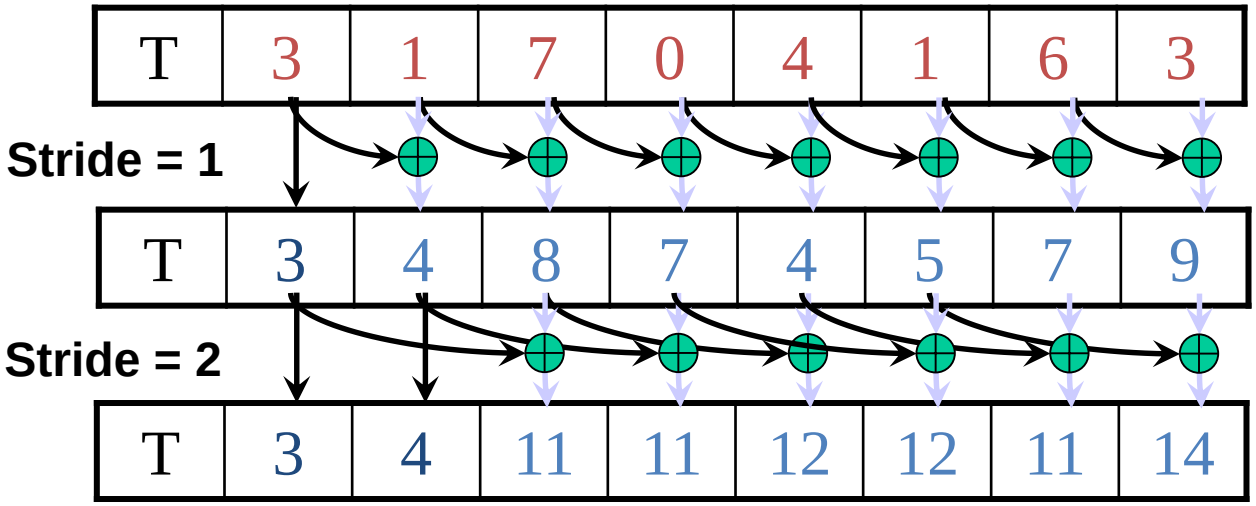
A Kogge-Stone Parallel Scan Algorithm



Iteration #1
Stride = 1

- Active threads: *stride* to *n*-1 (*n-stride* threads)
- Thread *j* adds elements *j* and *j-stride* from T and writes result into shared memory buffer T
- Each iteration requires two syncthreads
  - syncthreads(); // make sure that input is in place
  - float temp = T[j] + T[j - stride];
  - syncthreads(); // make sure that previous output has been consumed
  - T[j] = temp;

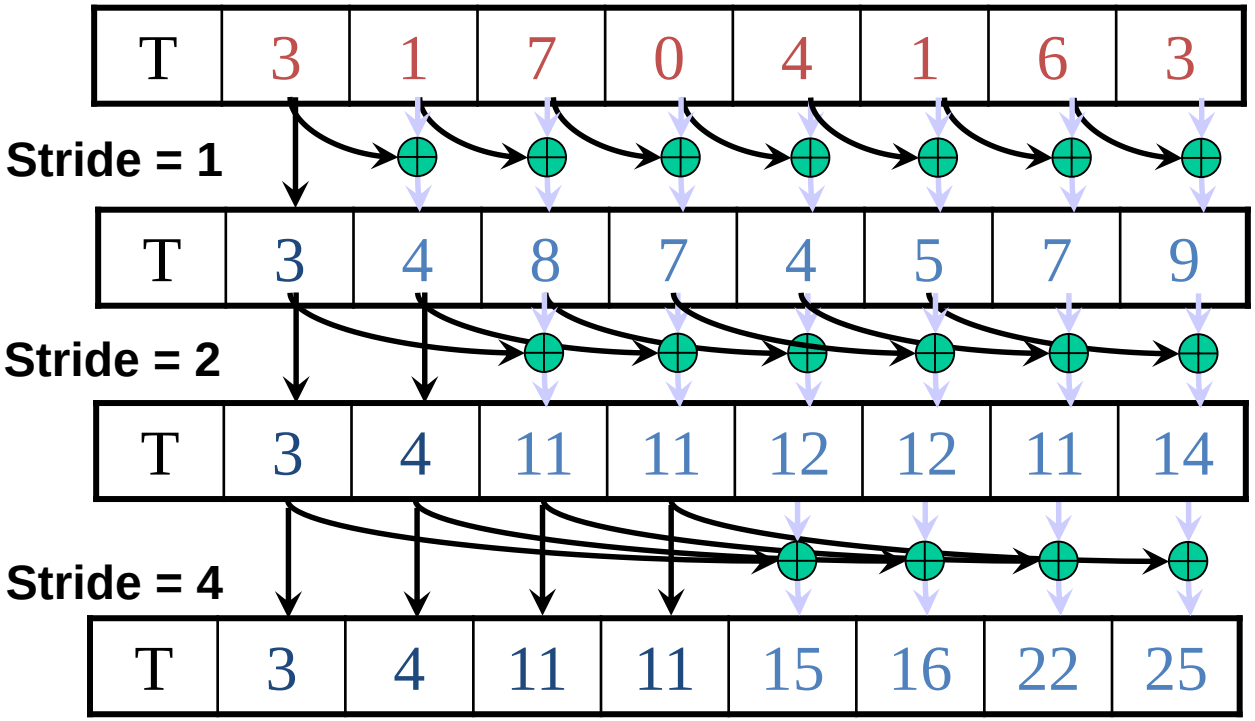
A Kogge-Stone Parallel Scan Algorithm



1. …
2. Iterate log(n) times, stride from 1 to ceil(n/2.0). Threads *stride* to *n-1 active:* add pairs of elements that are s*tride* elements apart.

Iteration #2
Stride = 2

A Kogge-Stone Parallel Scan Algorithm



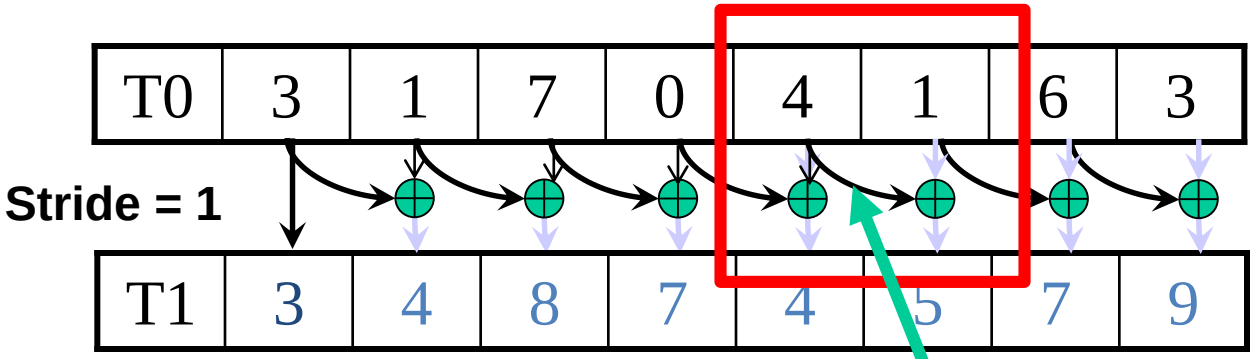
1. Load input from global memory to shared memory.
2. Iterate log(n) times, stride from 1 to ceil(n/2.0). Threads *stride* to *n-1 active:* add pairs of elements that are s*tride* elements apart.
3. Write output from shared memory to device memory

Iteration #3
Stride = 4

Double Buffering

- Use two copies of data T0 and T1
- Start by using T0 as input and T1 as output
- Switch input/output roles after each iteration
  - Iteration 0: T0 as input and T1 as output
  - Iteration 1: T1 as input and T0 and output
  - Iteration 2: T0 as input and T1 as output
- This is typically implemented with two pointers, source and destination that swap their contents from one iteration to the next
- This eliminates the need for the second syncthreads

A Double-Buffered Kogge-Stone Parallel Scan Algorithm



Thread 5

1. (previous slide)
2. Iterate log(n) times, stride from 1 to ceil(n/2.0). Threads *stride* to *n-1 active:* add pairs of elements that are s*tride* elements apart.

- Active threads: *stride* to *n*-1 (*n-stride* threads)
- Thread *j* adds elements *j* and *j-stride* from T and writes result into shared memory buffer T
- Each iteration requires only one syncthreads
  - syncthreads(); // make sure that input is in place
  - float destination[j] = source[j] + source[j - stride];
  - temp = destination; destination = source; source = temp;

Iteration #1
Stride = 1

Work Efficiency Analysis

- A Kogge-Stone scan kernel executes log(n) parallel iterations
  - The steps do (n-1), (n-2), (n-4),..(n- n/2) add operations each
  - Total # of add operations: n * log(n) - (n-1) → O(n*log(n)) work

- This scan algorithm is not very work efficient
  - Sequential scan algorithm does *n* adds
  - A factor of log(n) hurts: 20x for 1,000,000 elements!
  - Typically used within each block, where n ≤ 1,024

- A parallel algorithm can be slow when execution resources are saturated due to low work efficiency

To be continued…