CS 677: Parallel Programming for Many-core Processors
Lecture 5

Instructor: Philippos Mordohai
Webpage: mordohai.github.io
E-mail: Philippos.Mordohai@stevens.edu

Logistics

- Midterm: March 11
- Project proposal presentations: March 26
  - Have to be approved by me by March 12

Project Proposal

- Problem description
  - What is the computation and why is it important?
  - Abstraction of computation: equations, graphic or pseudo-code, no more than 1 page
- Suitability for GPU acceleration
  - Amdahl's Law: describe the inherent parallelism. Argue that it is close to 100% of computation.
  - Synchronization and Communication: Discuss what data structures may need to be protected by synchronization, or communication through host.
  - Copy Overhead: Discuss the data footprint and anticipated cost of copying to/from host memory.
- Intellectual Challenges
  - Generally, what makes this computation worthy of a project?
  - Point to any difficulties you anticipate at present in achieving high speedup

Some Ideas

- k-means
- Perceptron
- Boosting
  - General
  - Face detector (group of 2)
- Mean Shift
- Normal estimation for 3D point clouds

More Ideas

- Look for parallelizable problems in:
  - Image processing
  - Cryptanalysis
  - Graphics
    - GPU Gems
  - Nearest neighbor search

Version	Time Elapsed*	Step Speedup	Cumulative Speedup
C# CPU Version w/ GUI and CPU-only solver	~900 seconds	n/a	n/a
C CPU Version Command-line only CPU solver	236.65 seconds	Reference	Reference
Kernel1 Working solver on GPU	16.07 seconds	14.73x	14.73x
Kernel3 Added reduction kernel	9.18 seconds	1.75x	25.78x
Kernel4 Changed data structure to array instead of AoS	8.47 seconds	1.08x	27.94x
Kernel5 Simple caching w/ shared memory	7.25 seconds	1.17x	32.64x

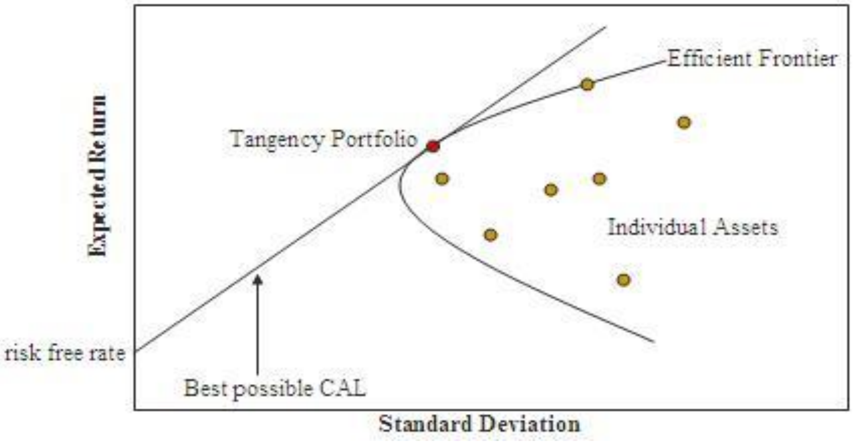
GPU: Shared Memory
512 Zombies
Average FPS: 45.9

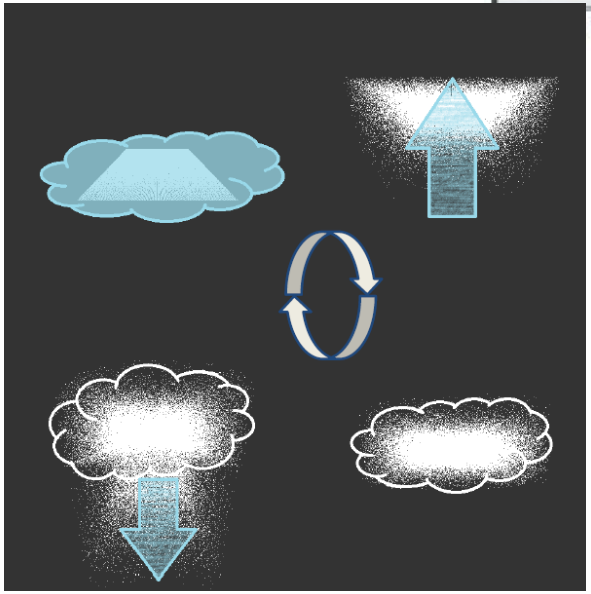
Even More…

- Particle simulations
- Financial analysis
- MCMC
- Games/puzzles
  - Mastermind example

Figure 3: Snowfall

Figure 4: Interactive Snow





k-means

- See also mordohai.github.io/classes/cs559_f16.html
  - Notes 13

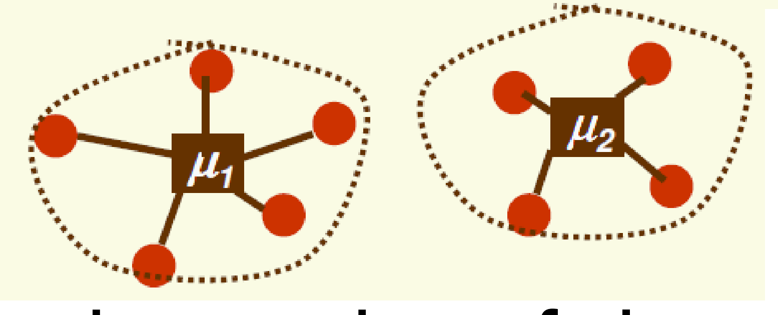
SSE Criterion Function

- Let **n**$_i$ be the number of samples, then the mean is:

$$\mu_i = \frac{1}{n_i} \sum_{x \in D_i} x$$

- The sum-of-squared errors criterion function (to minimize) is:

$$J_{SSE} = \sum_{i=1}^{c} \sum_{x \in D_i} \| x - \mu_i \|^2$$



- Note that the number of clusters, **c**, is fixed

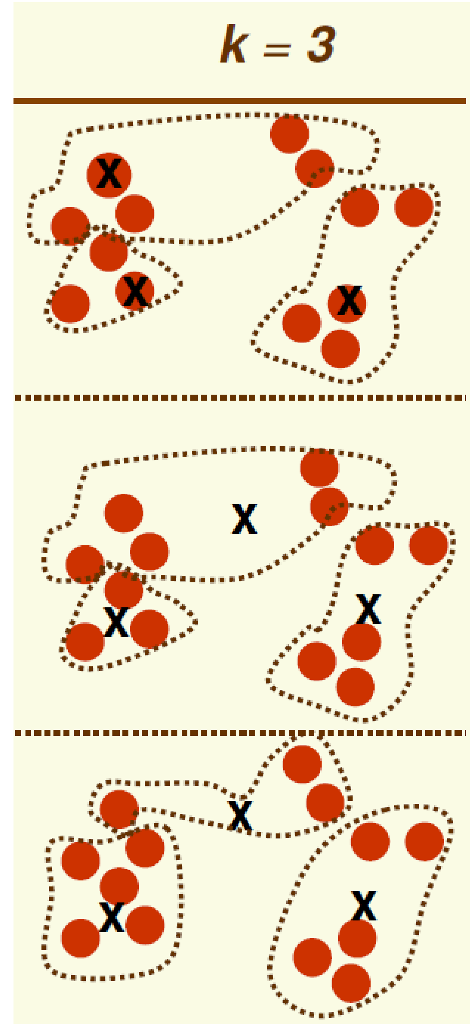
K-means Clustering

1. Initialize
   - Pick *k* cluster centers arbitrarily
   - Assign each example to closest center

2. Compute sample means for each cluster

3. Reassign all samples to the closest mean

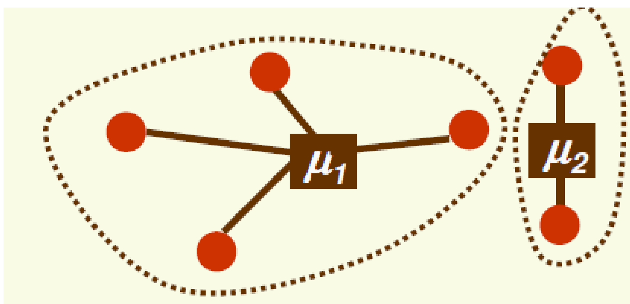
4. If clusters changed at step 3, go to step 2



K-means Clustering

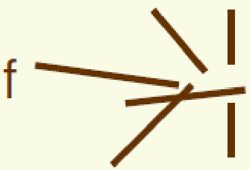
Consider steps 2 and 3 of the algorithm

2. compute sample means for each cluster

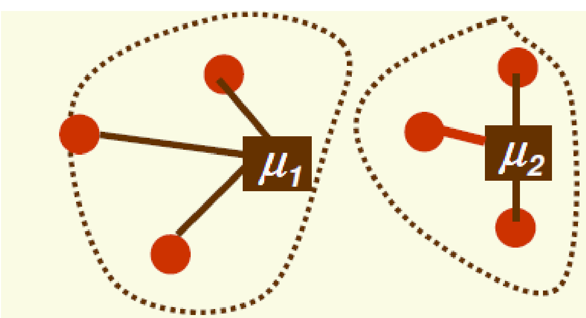


$$J_{SSE} = \sum_{i=1}^{k} \sum_{x \in D_i} || \mathbf{x} - \boldsymbol{\mu}_i ||^2$$

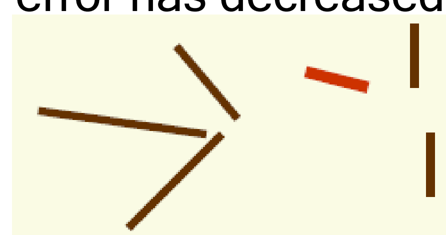
= sum of



3. reassign all samples to the closest mean

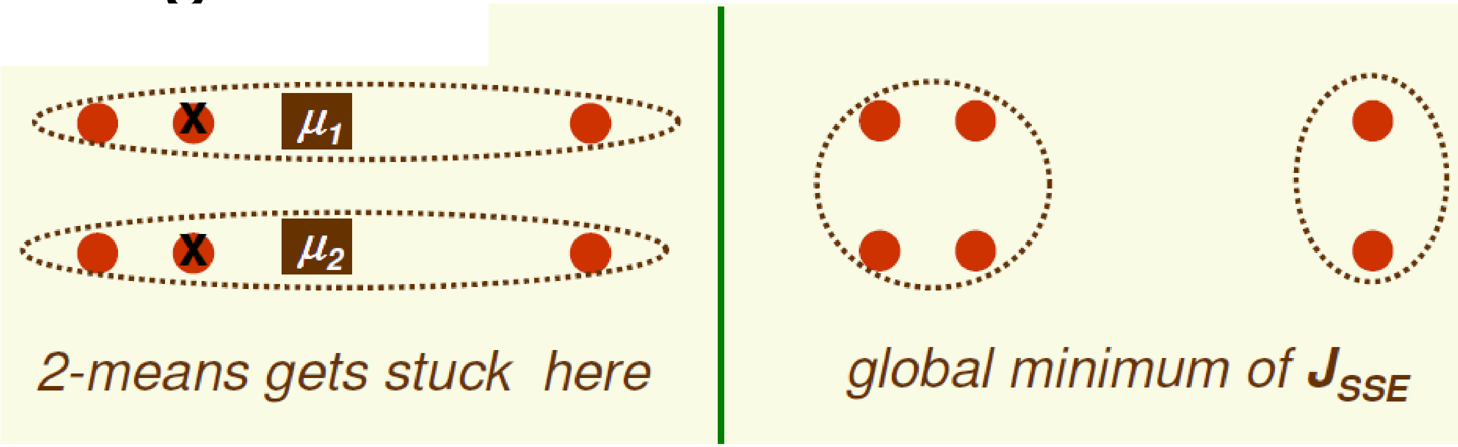


If we represent clusters by their old means, the error has decreased



K-means Clustering

- We can prove that by repeating steps 2 and 3, the objective function is reduced
- Thus k-means converges after a finite number of iterations of steps 2 and 3
- However k-means is not guaranteed to find a global minimum



K-means Clustering

- Finding the optimum of J_{SSE} is NP-hard
- In practice, k-means clustering usually performs well
- To avoid local minima, in practice we randomly re-initialize it several times

Perceptron

- See also mordohai.github.io/classes/cs559_f16.html
  - Notes 9

The Problem

- Assume we have 2 classes
  - Samples: $y_1,\dots, y_n$, some in class 1, some in class 2
- Use samples to determine weights a in the discriminant function $g(y) = a^t y$
- We want to minimize the training error (the number of misclassified samples $y_1,\dots, y_n$)
- If: $g(y_i)>0 \Rightarrow y_i$ ***classified as*** c_1
  $g(y_i)<0 \Rightarrow y_i$ ***classified as*** c_2
- Thus training error is **0** if $\begin{cases} g(y_i) > 0 & \forall y_i \in c_1 \\ g(y_i) < 0 & \forall y_i \in c_2 \end{cases}$

“Normalization”

- Thus training error is 0 if: $\begin{cases} a^t y_i > 0 & \forall y_i \in c_1 \\ a^t y_i < 0 & \forall y_i \in c_2 \end{cases}$
- Equivalently, training error is 0 if: $\begin{cases} a^t y_i > 0 & \forall y_i \in c_1 \\ a^t(-y_i) > 0 & \forall y_i \in c_2 \end{cases}$
- This suggests “normalization” (a.k.a. reflection):
  1. Replace all examples from class 2 by:

$$y_i \to -y_i \qquad \forall y_i \in c_2$$

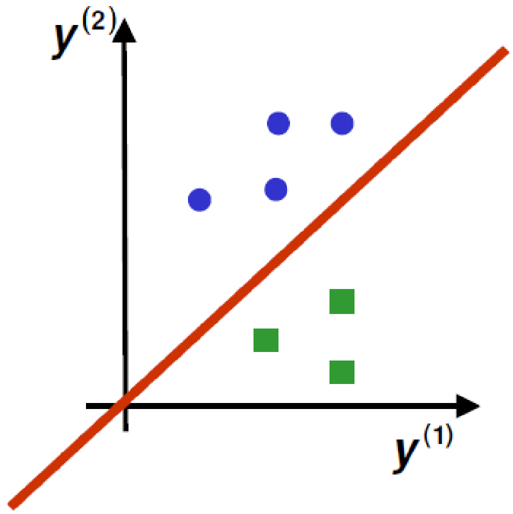
  2. Seek weight vector ***a*** such that

$$a^t y_i > 0 \qquad \forall y_i$$

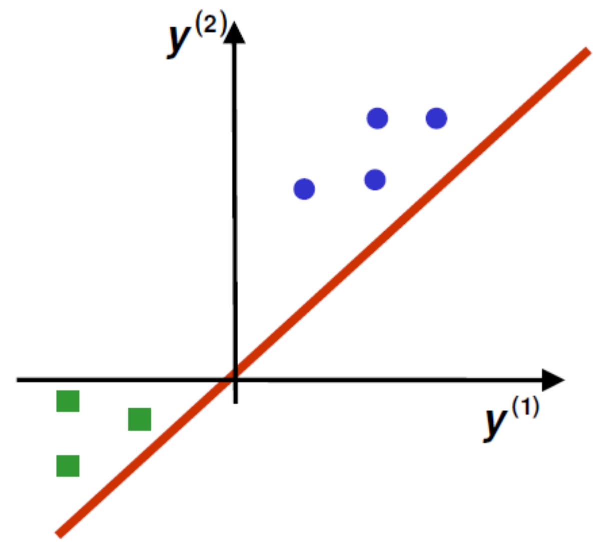
  – If such ***a*** exists, it is called a separating or solution vector
  – Original samples $x_1, \ldots, x_n$ can indeed be separated by a line

Normalization

before normalization



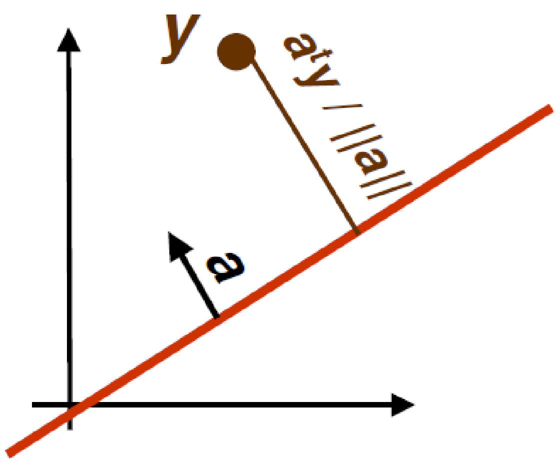
after "normalization"



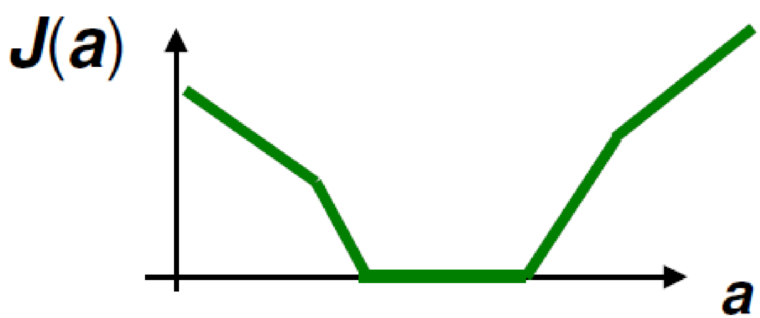
- Seek a hyperplane that separates patterns from different categories
- Seek hyperplane that puts *normalized* patterns on the same(positive) side

Perceptron Criterion Function

$$J_p(a) = \sum_{y \in Y_M} \left(-a^t y\right)$$



- If y is misclassified, $a^t y < 0$
- Thus $J_p(a) > 0$
- $J_p(a)$ is $||a||$ times the sum of distances of misclassified examples to decision boundary
- **Jp(a)** is piecewise linear and thus suitable for gradient descent



Perceptron Batch Rule

$$J_p(a) = \sum_{y \in Y_M} \left(-a^t y\right)$$

- Gradient of $J_p(\mathbf{a})$ is: $\nabla J_p(a) = \sum_{y \in Y_M} (-y)$
  - Y_M are samples misclassified by $\mathbf{a}^{(k)}$
  - It is not possible to solve $\nabla J_p(a) = 0$ analytically because of Y_M
- Update rule for gradient descent: $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \eta^{(k)} \nabla J(x)$
- Thus the *gradient decent batch update rule for* $J_p(\mathbf{a})$ is:

$$a^{(k+1)} = a^{(k)} + \eta^{(k)} \sum_{y \in Y_M} y$$

- It is called batch rule because it is based on all misclassified examples

Boosting

- See also mordohai.github.io/classes/cs559_f16.html
  - Notes 10

Boosting

- Idea: given a set of weak learners, run them multiple times on (reweighted) training data, then let learned classifiers vote
- At each iteration *t:*
  - Weight each training example by how incorrectly it was classified
  - Learn a hypothesis – h_t
  - Choose a strength for this hypothesis – α_t
- Final classifier: weighted combination of weak learners

Learning from Weighted Data

- Sometimes not all data points are equal
  - Some data points are more equal than others
- Consider a weighted dataset
  - D(i) – weight of i th training example $(\mathbf{x}_i, \mathbf{y}_i)$
  - Interpretations:
    - i th training example counts as D(i) examples
    - If I were to "resample" data, I would get more samples of "heavier" data points
- Now, in all calculations the i^{th} training example counts as D(i) "examples"

Definition of Boosting

- Given training set $(\mathbf{x}_1,\mathbf{y}_1),\ldots, (\mathbf{x}_m,\mathbf{y}_m)$
- $y_i \in\{-1,+1\}$ correct label of instance $x_i \in X$
- For $t=1,\ldots,T$
  - construct distribution D_t on $\{1,\ldots,m\}$
  - find weak hypothesis
  - h_t: $X \rightarrow \{-1,+1\}$
    with small error ε_t on D_t

$$\epsilon_t = \Pr_{i \sim D_t}\left[h_t(x_i) \neq y_i\right]$$

- Output final hypothesis H_{final}

AdaBoost

- Constructing D_t
  - $D_1 = 1/m$
  - Given D_t and h_t: $$D_{t+1}(i) = \frac{D_t(i)}{Z_t} \cdot \begin{cases} e^{-\alpha_t} & \text{if } y_i = h_t(x_i) \\ e^{\alpha_t} & \text{if } y_i \neq h_t(x_i) \end{cases}$$ $$= \frac{D_t(i)}{Z_t} \cdot \exp(-\alpha_t\, y_i\, h_t(x_i))$$

    where Z_t is a normalization constant $$Z_t = \sum_{i=1}^{m} D_t(i) \exp(-\alpha_t y_i h_t(x_i))$$

- Final hypothesis:

$$\boxed{H_{\text{final}}(x) = \text{sign}\left(\sum_t \alpha_t h_t(x)\right)}$$

$$\alpha_t = \frac{1}{2} \ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right) > 0$$

Face Detection

- I see this as a two person project
  - One implements boosting as before
  - One implements the face-specific parts

- See also mordohai.github.io/classes/cs559_f16.html
  - Notes 10

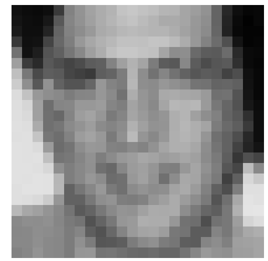
Classifier is Learned from Labeled Data

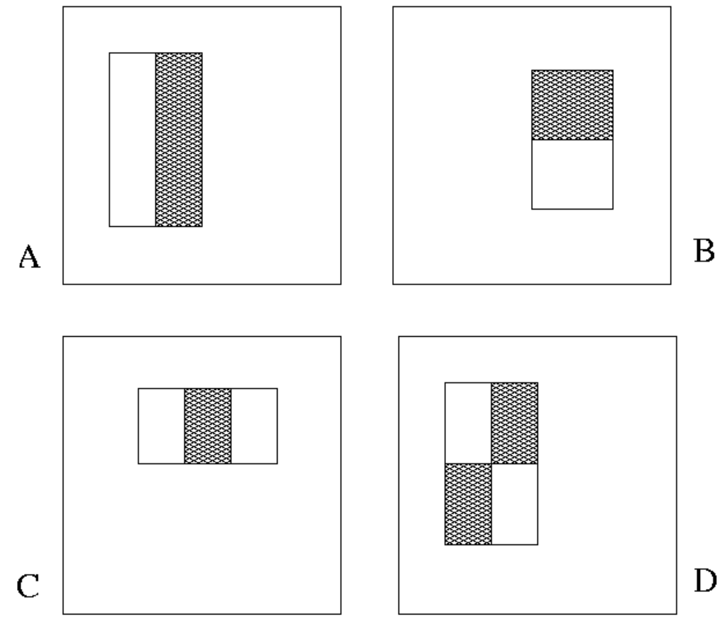
- Training Data
  - 5000 faces
    - All frontal
  - 10^8 non faces
  - Faces are normalized
    - Scale, translation
- Many variations
  - Across individuals
  - Illumination
  - Pose (rotation both in plane and out)

Boosted Face Detection: Image Features

"Rectangle filters"

Similar to Haar wavelets





$$h_t(x_i) = \begin{cases} \alpha_t & \text{if } f_t(x_i) > \theta_t \\ \beta_t & \text{otherwise} \end{cases}$$

$$C(x) = \theta\left(\sum_t h_t(x) + b \right)$$

$$60{,}000 \times 100 = 6{,}000{,}000$$

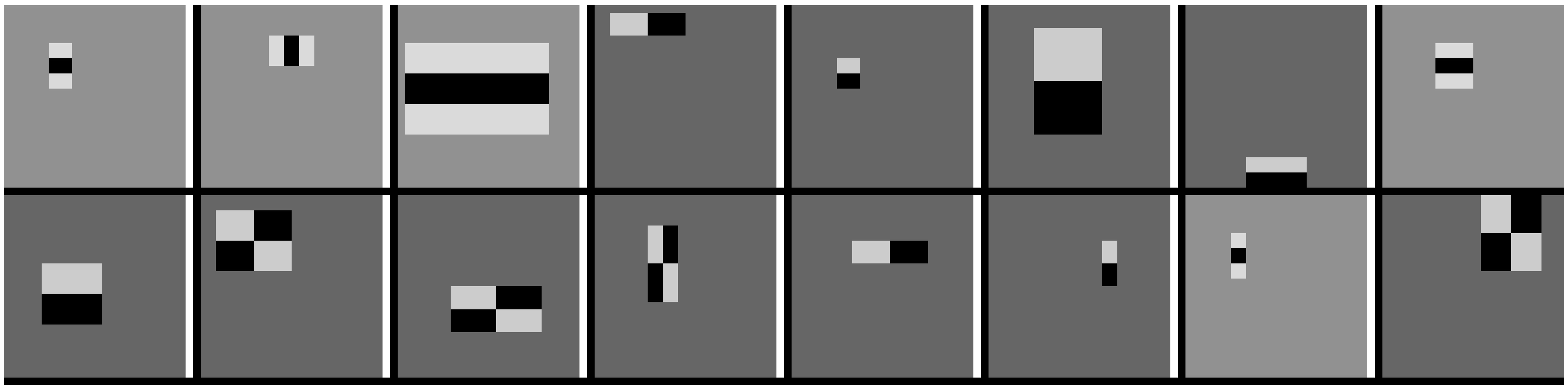
Unique Binary Features

Feature Selection

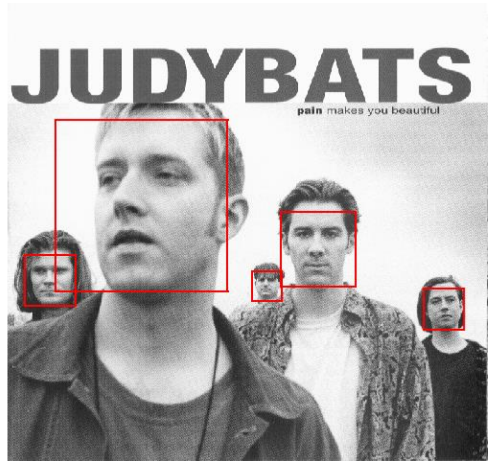
- For each round of boosting:
  - Evaluate each rectangle filter on each example
  - Sort examples by filter values
  - Select best threshold for each filter
  - Select best filter/threshold (= Feature)
  - Reweight examples

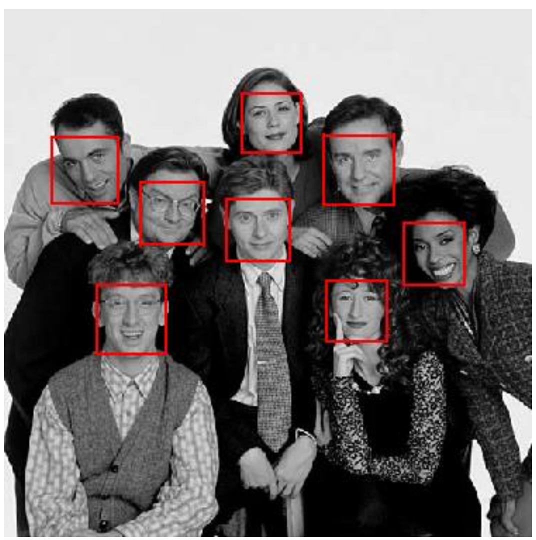
Feature Localization

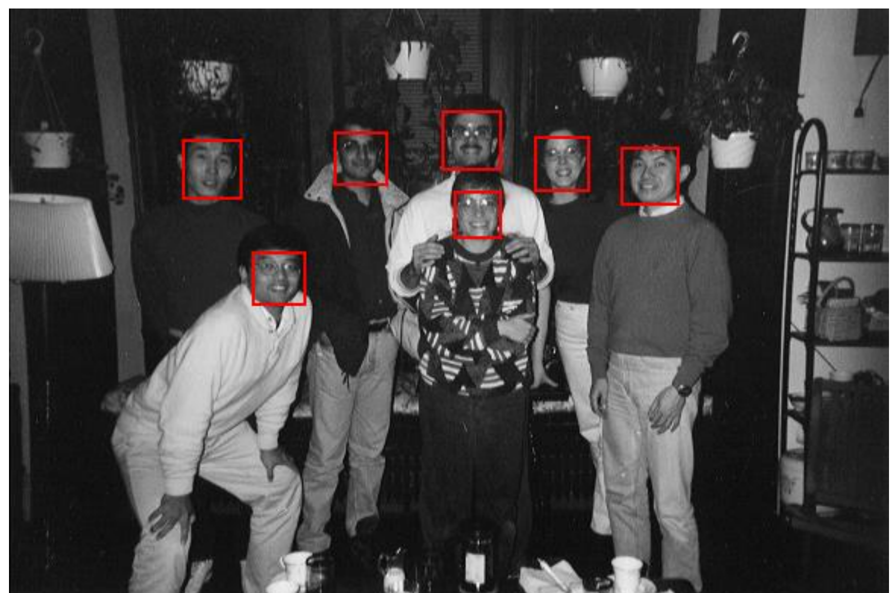
- Learned features reflect the task



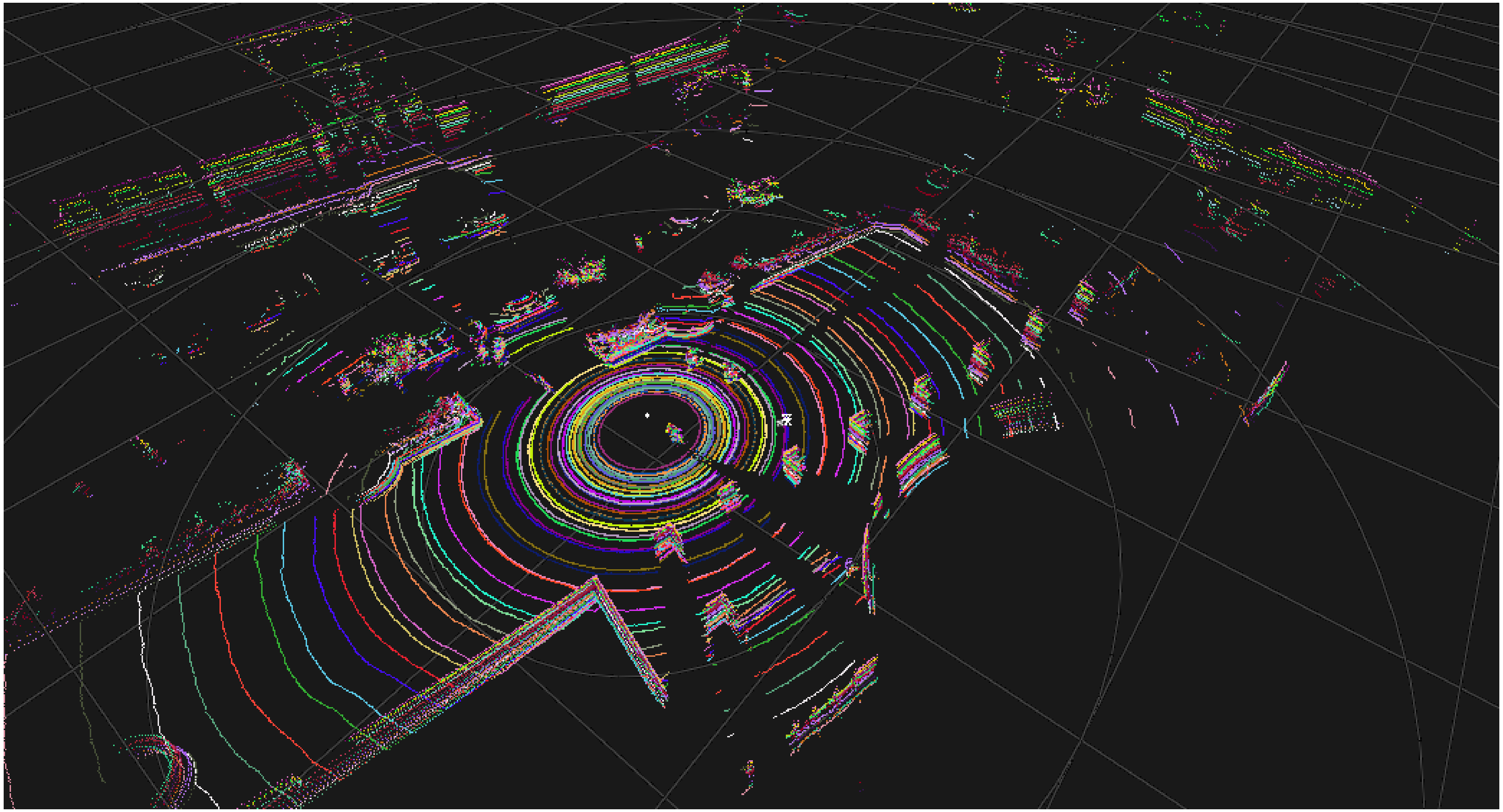
Output of Face Detector on Test Images







Normal Estimation for 3D Point Clouds



Scatter Matrix

- Compute the symmetric positive definite covariance matrix from N neighbors of a 3-D point
  - $\{X_i\} = \{(x_i, y_i, z_i)^T\}$

$$\frac{1}{N}\sum_{i=i}^{N}(X_i - \overline{X})(X_i - \overline{X})^T$$

- Then, the eigenvector that corresponds to the smallest eigenvalue is the normal to the surface at each point
  - If each point belonged to a smooth surface

Classification

- Points can be classified according to eigenvalues into surfaces, foliage, ground plane etc.
  - Images from Lalonde et al. 2006

Markov Chain Monte Carlo

- Randomized algorithms based on sampling from probability distributions to generate sequences of observations
- Applications
  - Approximate integration
  - Optimization of energy/cost functions in very large search spaces
  - Risk assessment in finance

Sample Proposal

3 Intellectual Challenges

The main challenge is going to be how to partition the work. As mentioned above, the overall algorithm is finding the minimum across a set. However, there is also an internal operation that involves a maximum operation. In terms of mapping this to CUDA, there are going to need to be some testing to determine how heavy a thread should be. For example, one configuration would be to make every thread calculate the worst-case scenario for one element in the set. Another configuration would be to calculate that maximum on the block-level, making the threads perform much less work.

The main obstacle for performance is going to be synchronization. Especially in a case where every block produces one out of 32,768 results that need to be minimized, doing atomic operations to a global memory location is bound to have consequences. A lot of parameterization is going to be necessary so that different combinations of strategies can be fully tested.

The Problem Description above focused on Knuth's algorithm for solving Mastermind puzzles. There have been a few papers published since then which propose better solutions, such as the often cited 1993 paper by Koyama and Lai[2] and a more recent 2005 paper by Kooi[3]. In the course of the actual project, I plan to investigate those other algorithms and if they are equally parallel-capable and seem to perform better, I will switch the algorithm.

I believe this project has a great chance to show how CUDA can be used to improve the performance of existing algorithms, increasing their domain of effectiveness.

Convolution

Convolution Applications

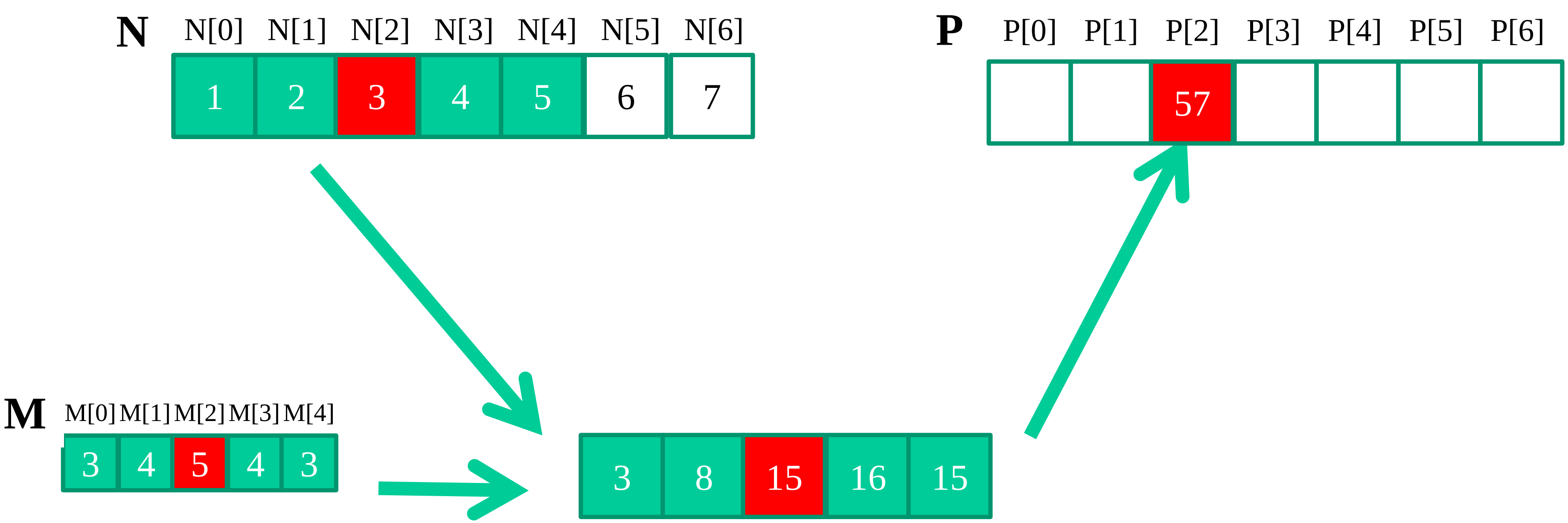
- A popular array operation that is used in various forms in signal processing, digital recording, image processing, video processing, and computer vision
- Convolution is often performed as a filter that transforms signals and pixels into more desirable values
  - Some filters smooth out the signal 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*

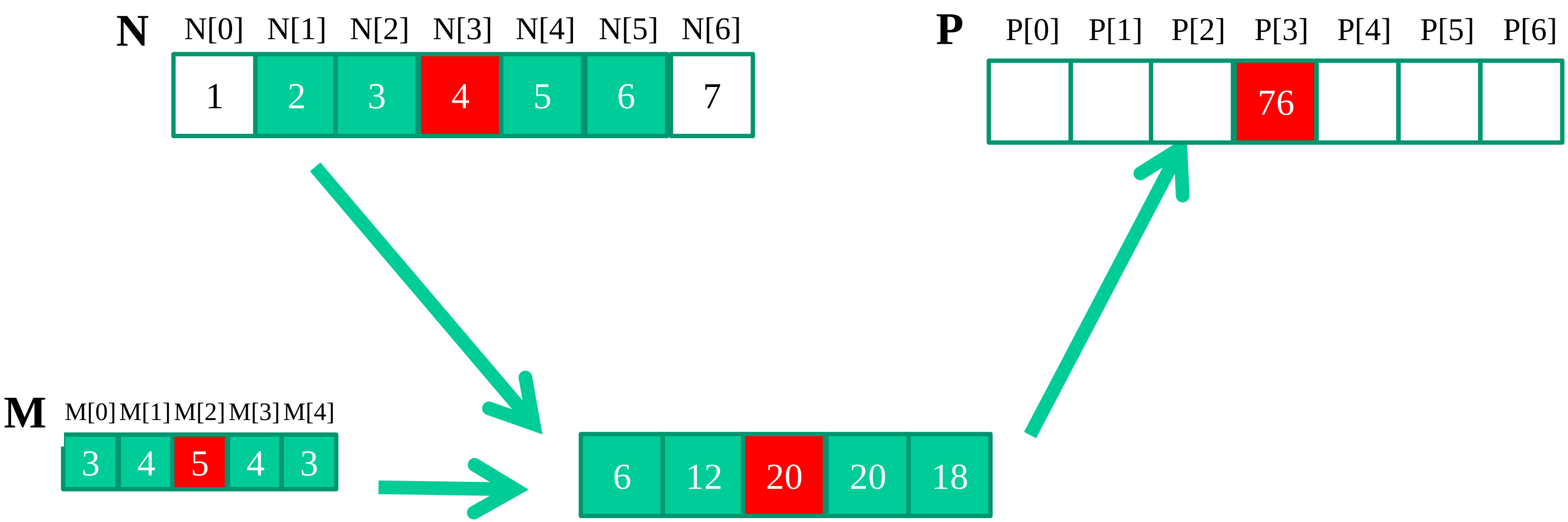
1D Convolution Example

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



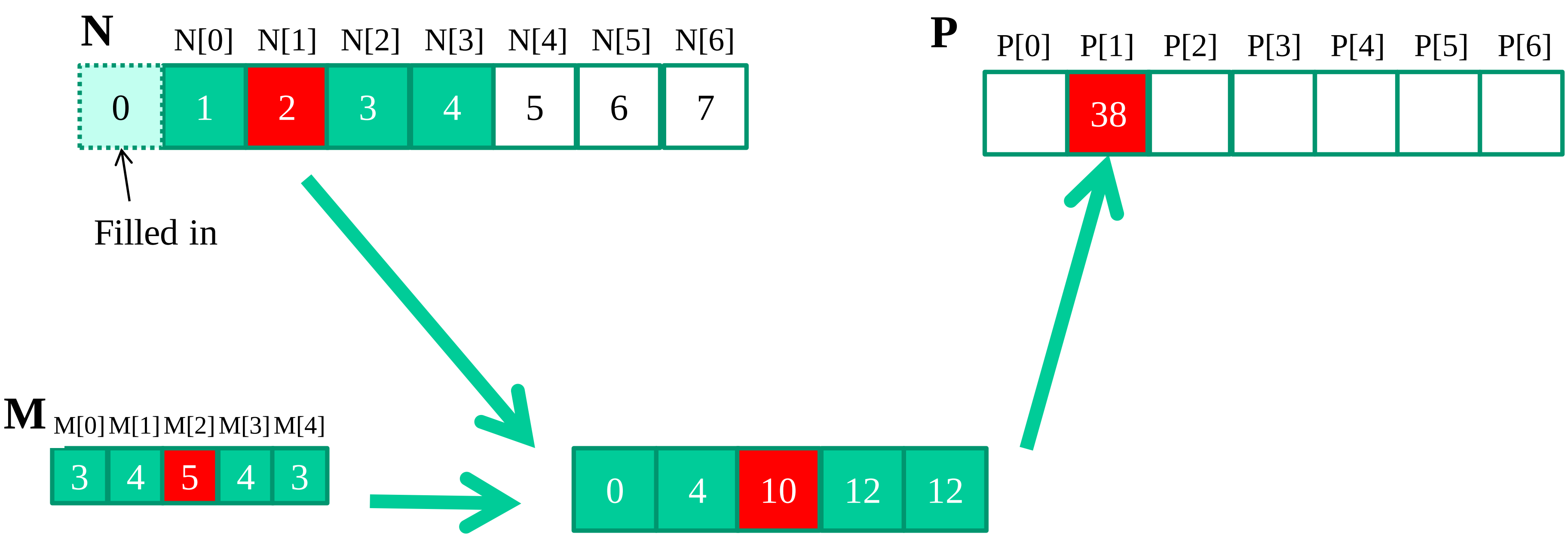
1D Convolution Example

- 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

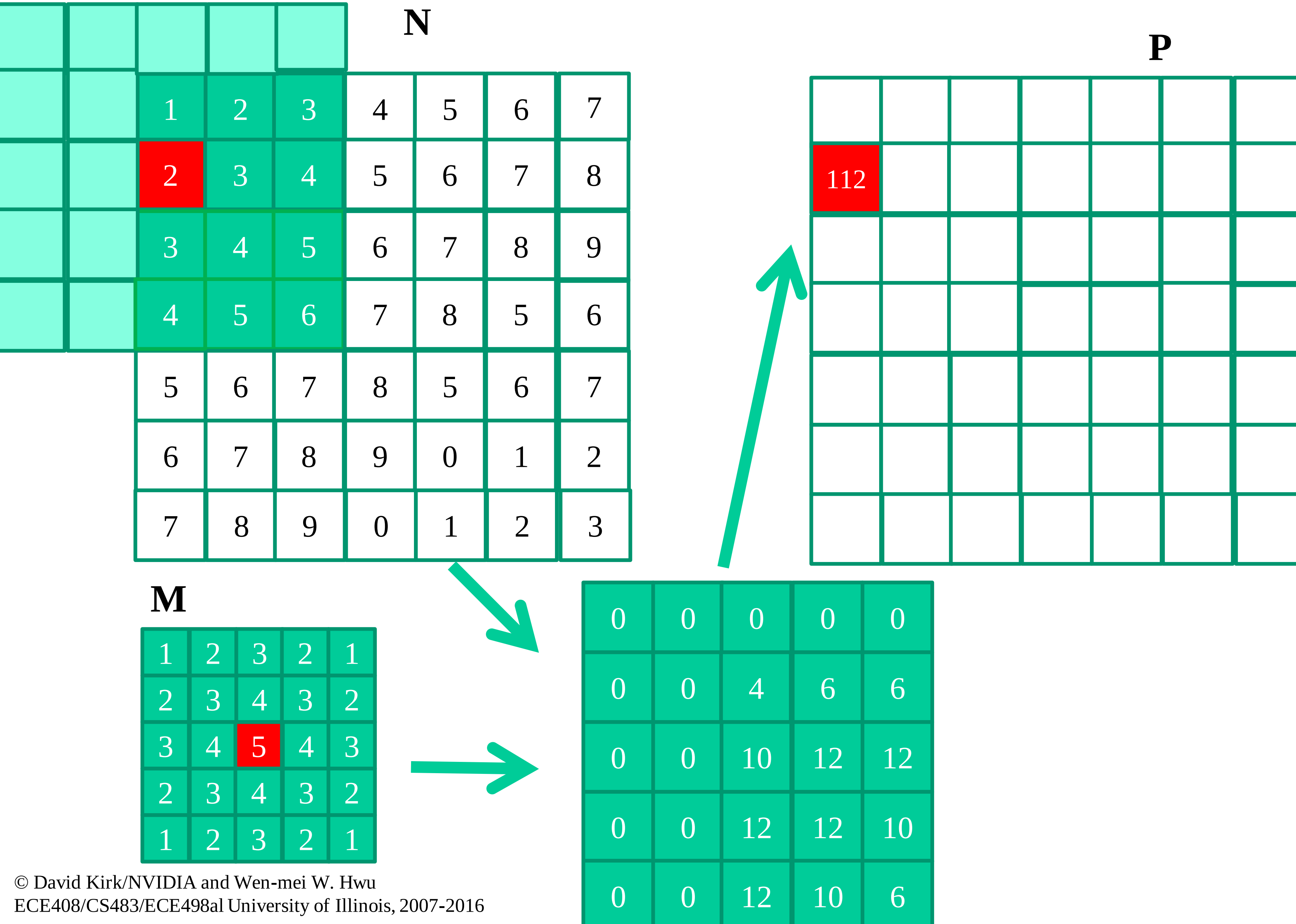
		321				

M

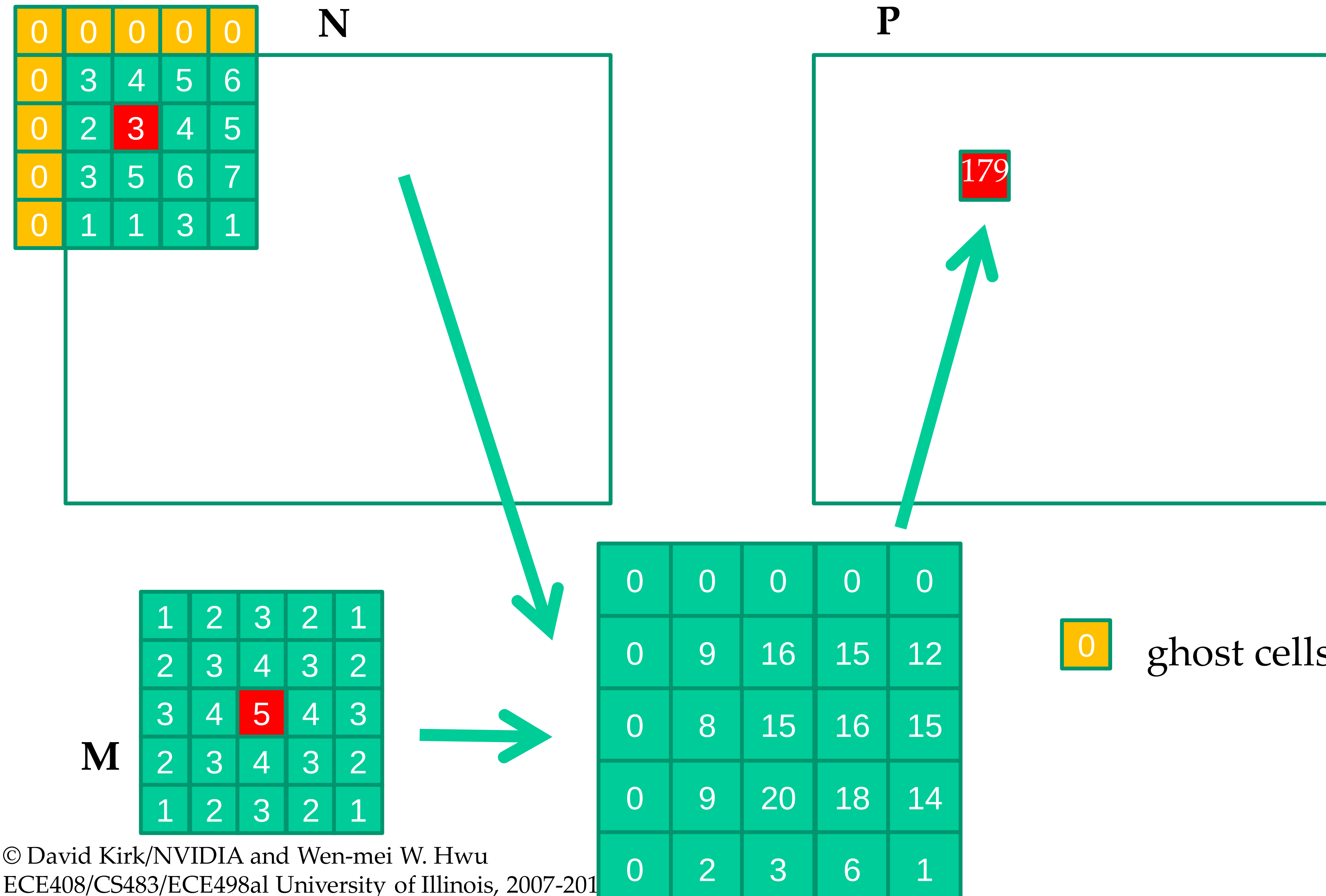
1	2	3	2	1
2	3	4	3	2
3	4	5	4	3
2	3	4	3	2
1	2	3	2	1

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

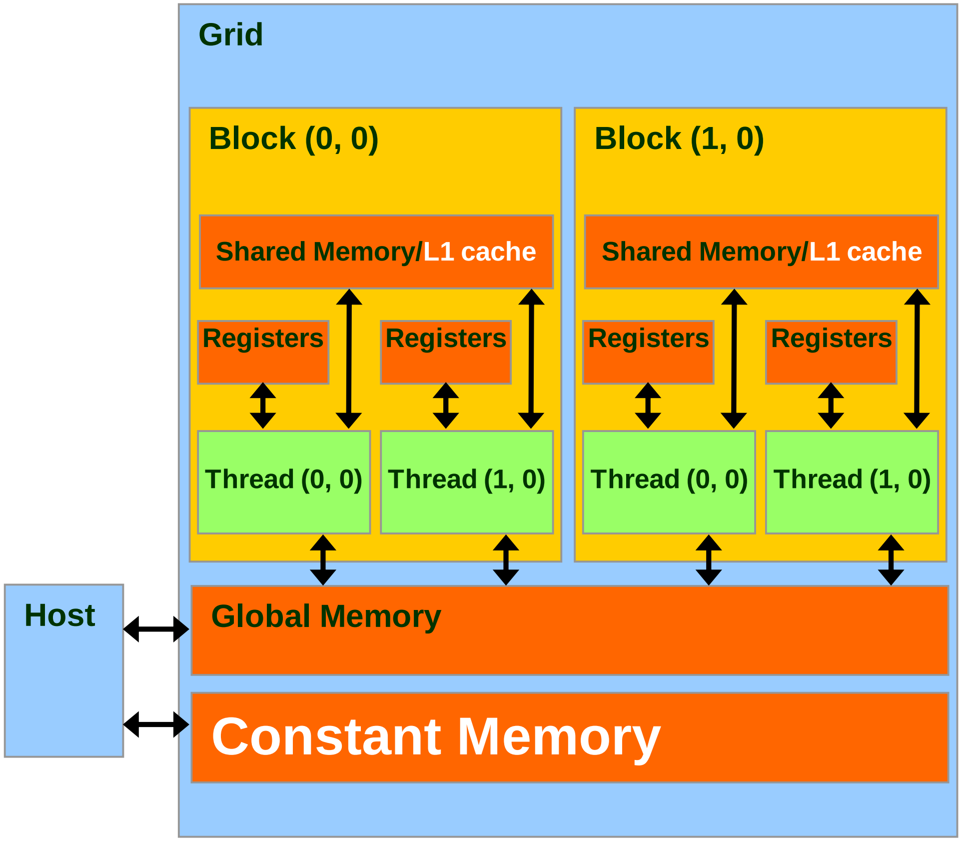
- M is referred to as mask (a.k.a. kernel, filter, etc.)
  – Elements of M are called mask (kernel, filter) coefficients
- Calculation of all output P elements need M
- M is not changed during kernel

- 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

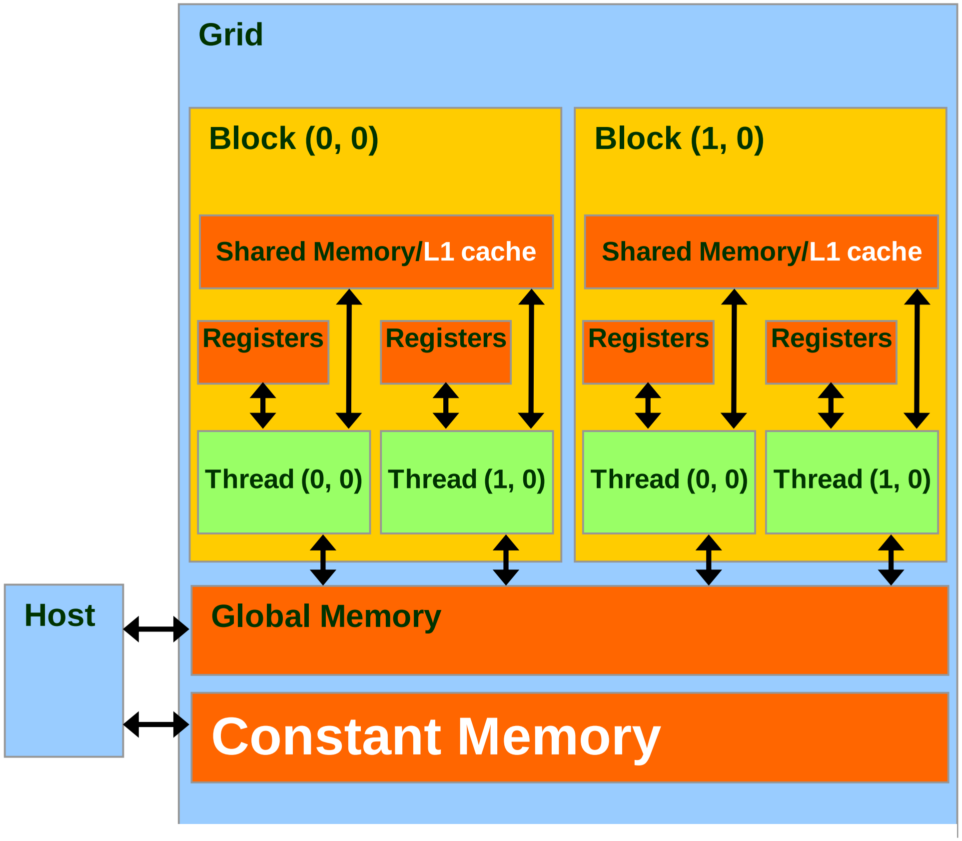
Some definitions:

- 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

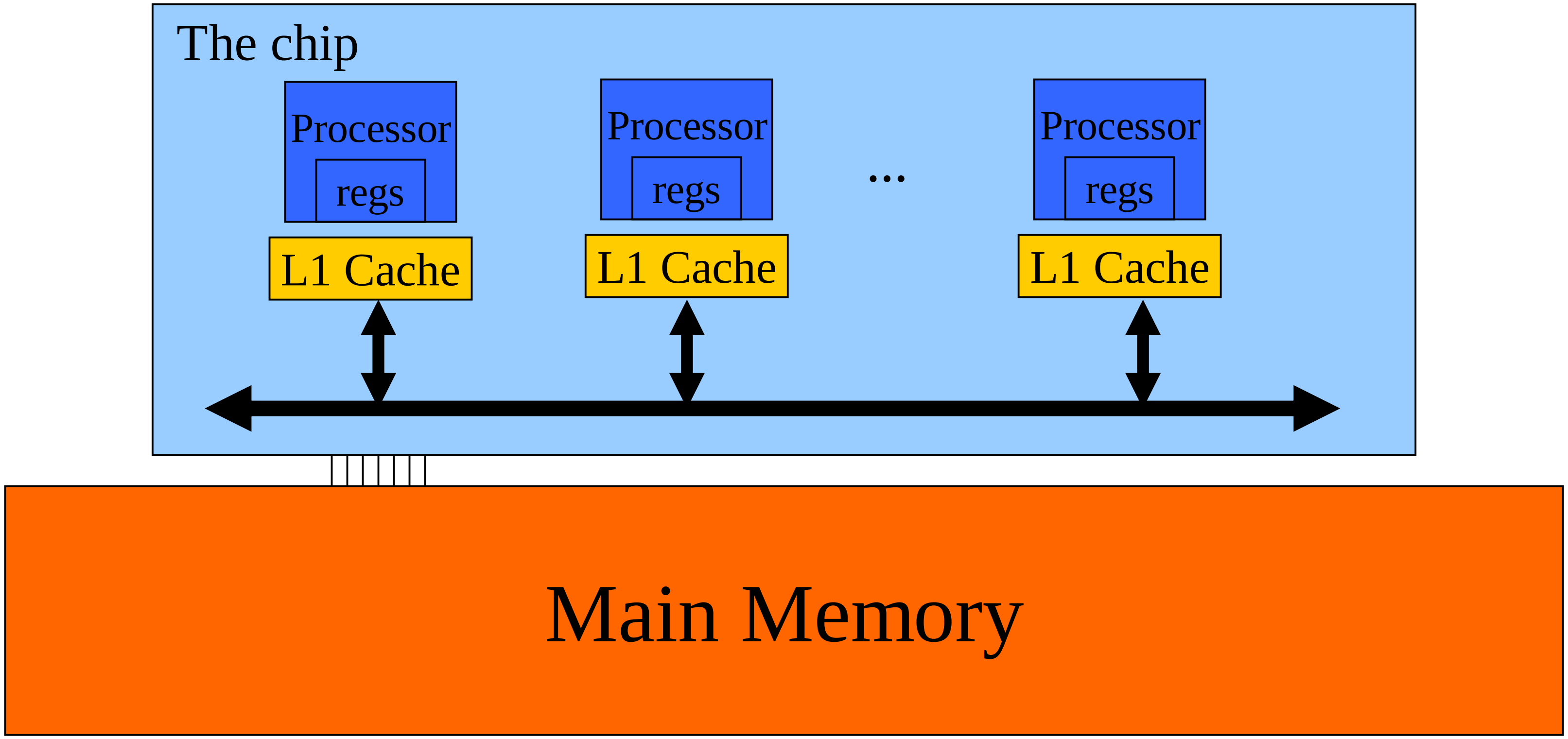
- Each SM has its own L1 cache
  - Low latency, high bandwidth access by all threads
- However, there is no way for threads in one SM to update the L1 cache in other SMs
  - No L1 cache coherence



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 `__constant__ 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

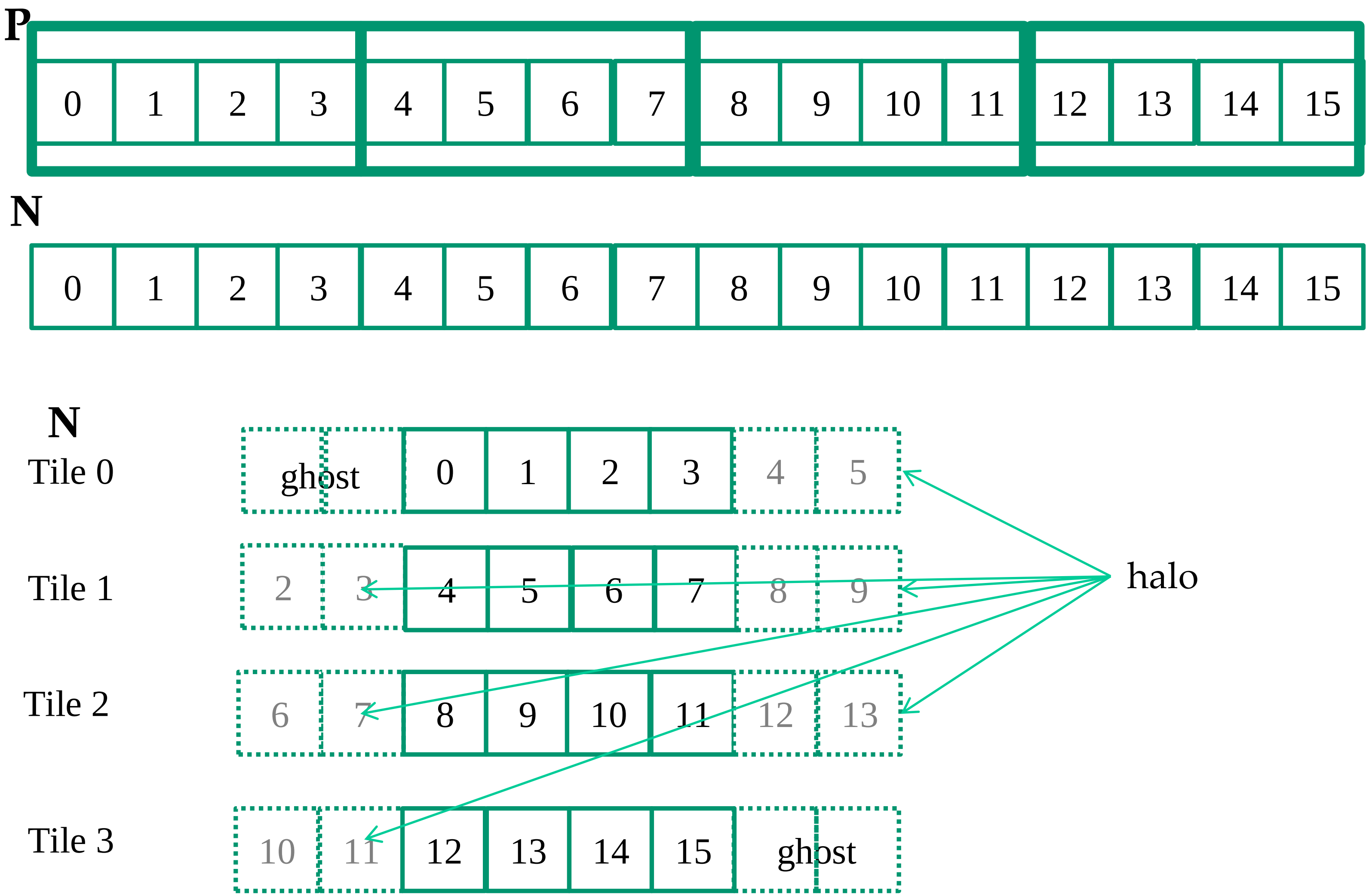
```
// global variable, outside any kernel/function
   __constant__ float Mc[MASK_WIDTH][MASK_WIDTH];
…
   // allocate N, P, initialize N elements, copy N to Nd
   Matrix  M;
   M  = AllocateMatrix(MASK_WIDTH, MASK_WIDTH, 1);
   // initialize M elements
….
   cudaMemcpyToSymbol(Mc, M.elements,
     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

Tiled 1D Convolution

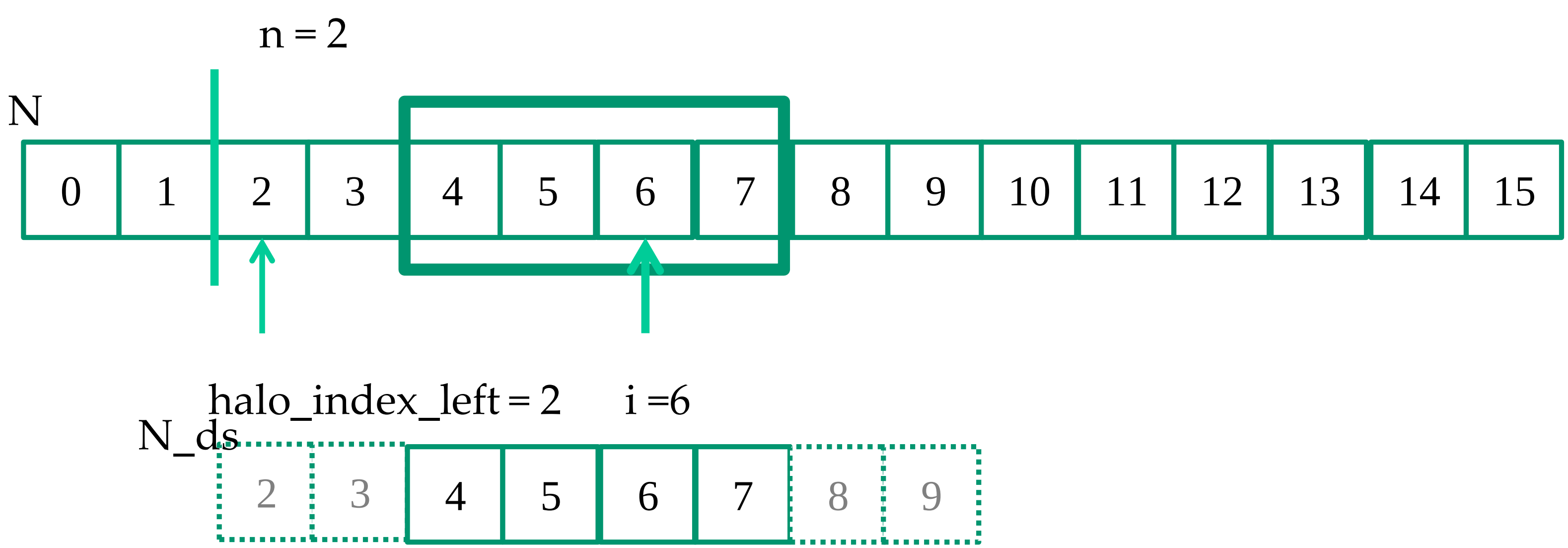
- Elements of the input vector are used in multiple computations
- Opportunity to use **shared memory**

- Shared memory tile must be larger than mask

Tiled 1D Convolution Basic Idea

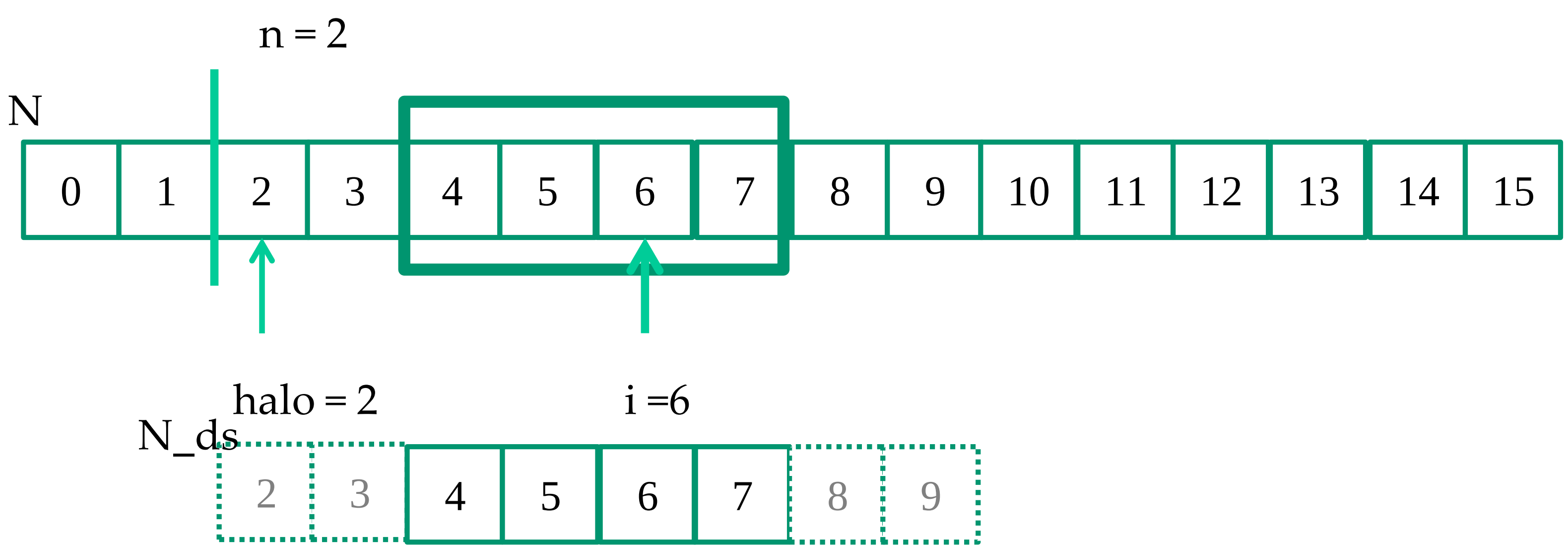


Loading Left Halo



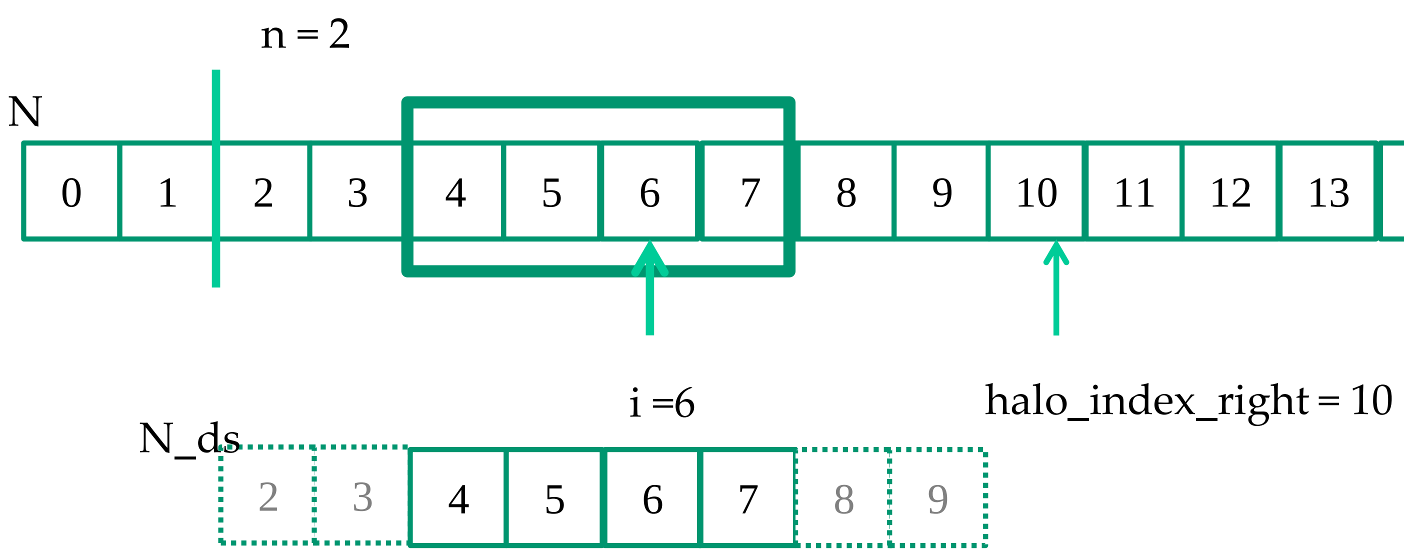

```
int n = Mask_Width/2;
int halo_index_left = (blockIdx.x - 1)*blockDim.x + threadIdx.x;
  if (threadIdx.x >= blockDim.x - n) {
    N_ds[threadIdx.x - (blockDim.x - n)] =
      (halo_index_left < 0) ? 0 : N[halo_index_left];
  }
```

Loading Internal Elements




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

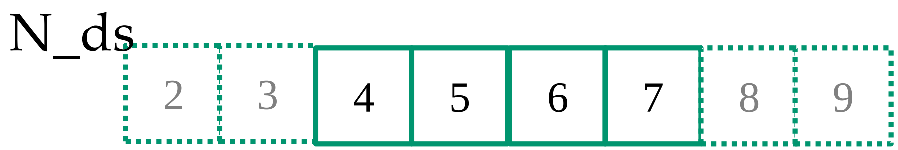
Loading Right Halo




```
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;

}
```

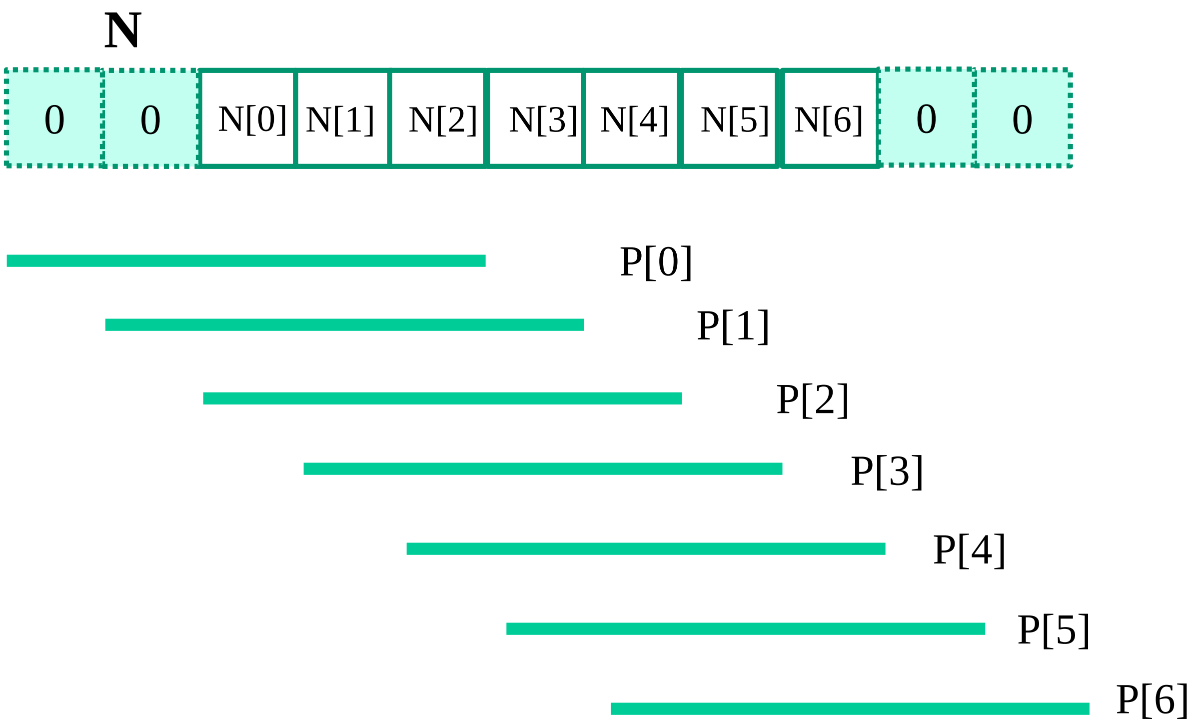
Shared Memory Data Reuse



Mask_Width is 5

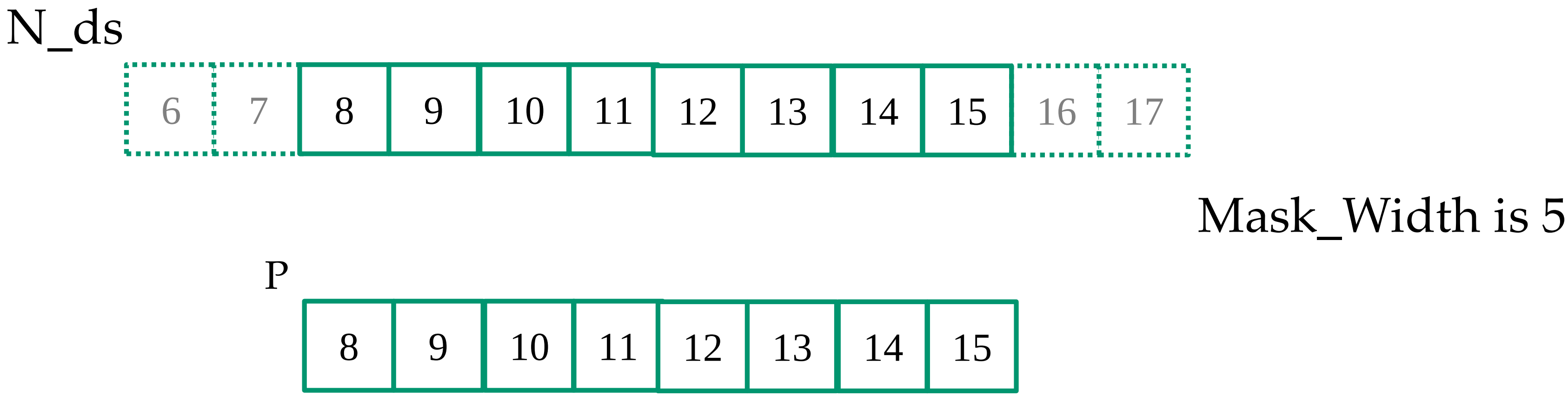
- Element 2 is used by thread 4 (1X)
- Element 3 is used by threads 4, 5 (2X)
- Element 4 is used by threads 4, 5, 6 (3X)
- Element 5 is used by threads 4, 5, 6, 7 (4X)
- Element 6 is used by threads 4, 5, 6, 7 (4X)
- Element 7 is used by threads 5, 6, 7 (3X)
- Element 8 is used by threads 6, 7 (2X)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

Ghost Cells



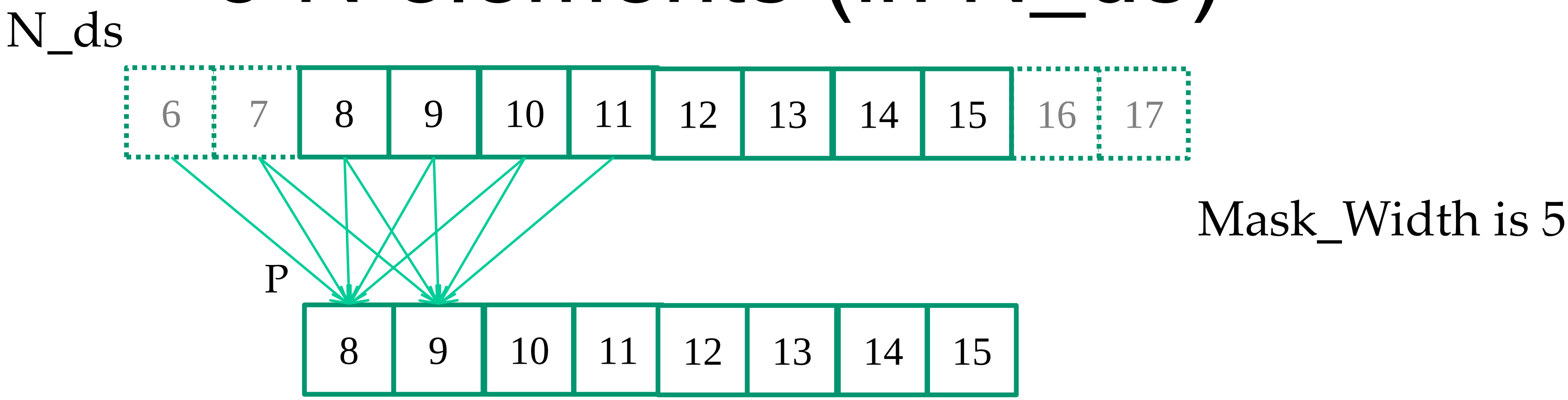
```
__global__ void convolution_1D_tiled_cache_kernel(float *N, float *P,
int Mask_Width, int Width) {

  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

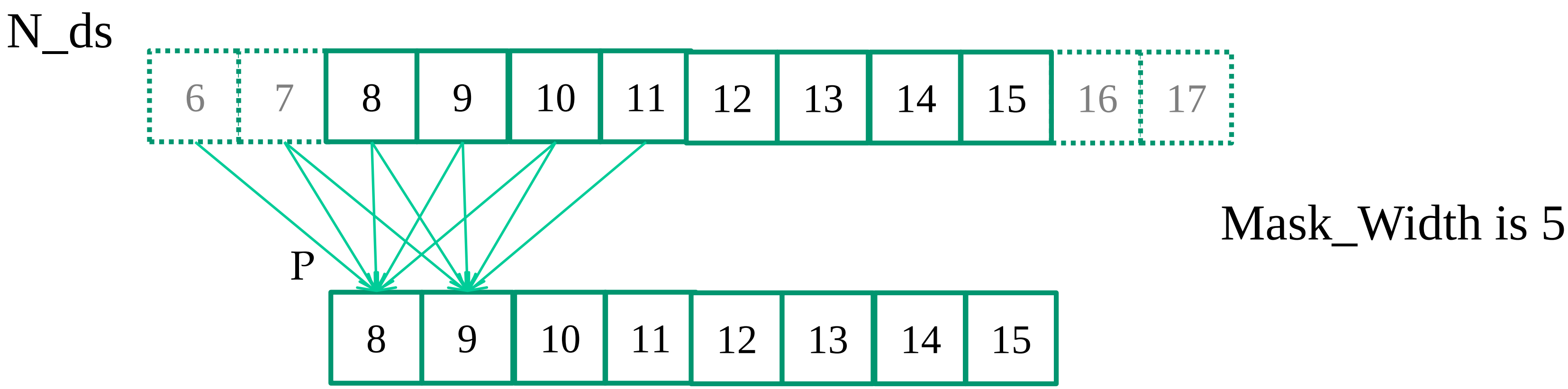
In General, in 1D

- 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

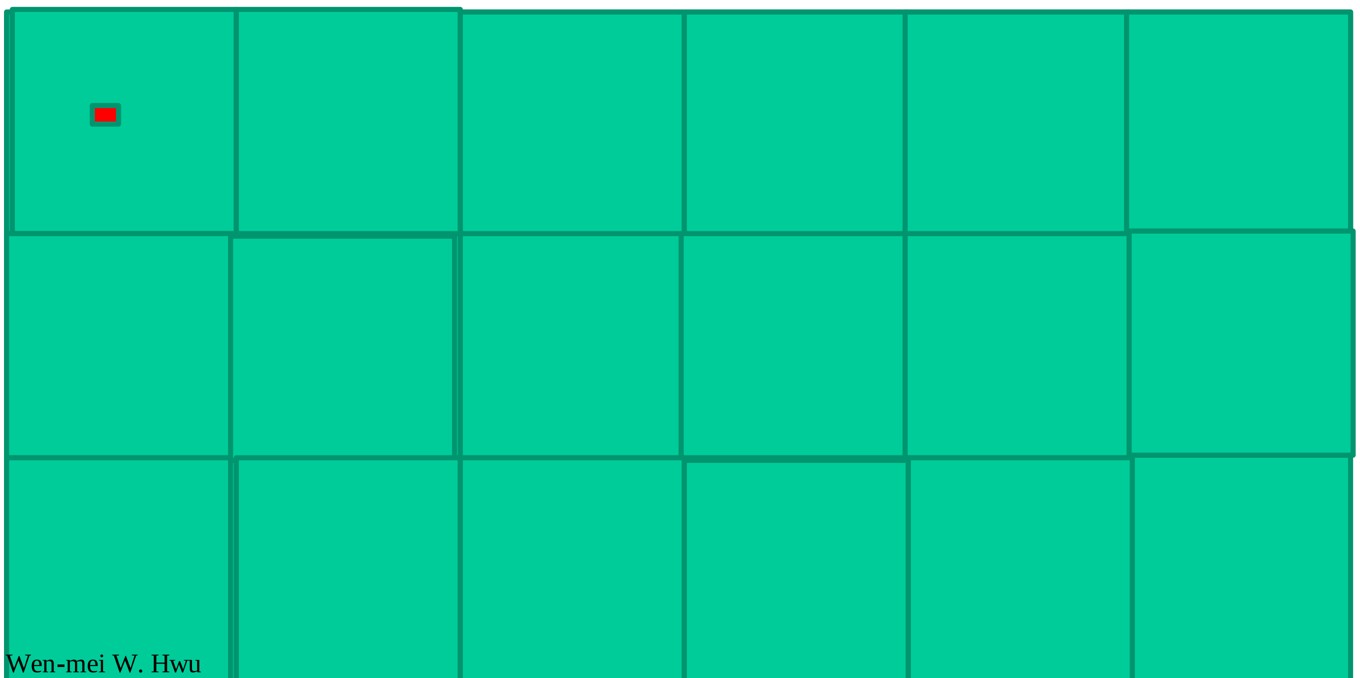
- The reduction is

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

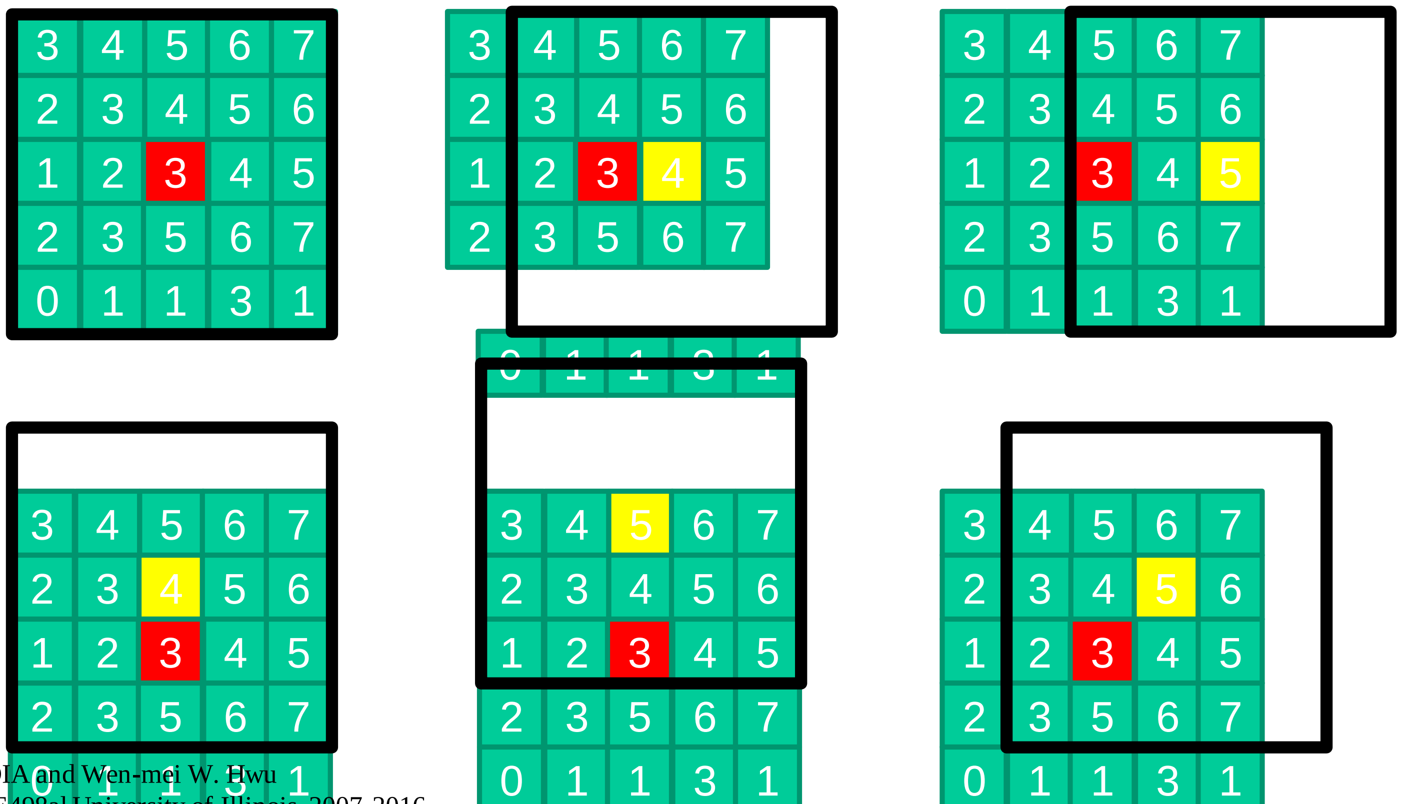
Tiling P

- 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;`

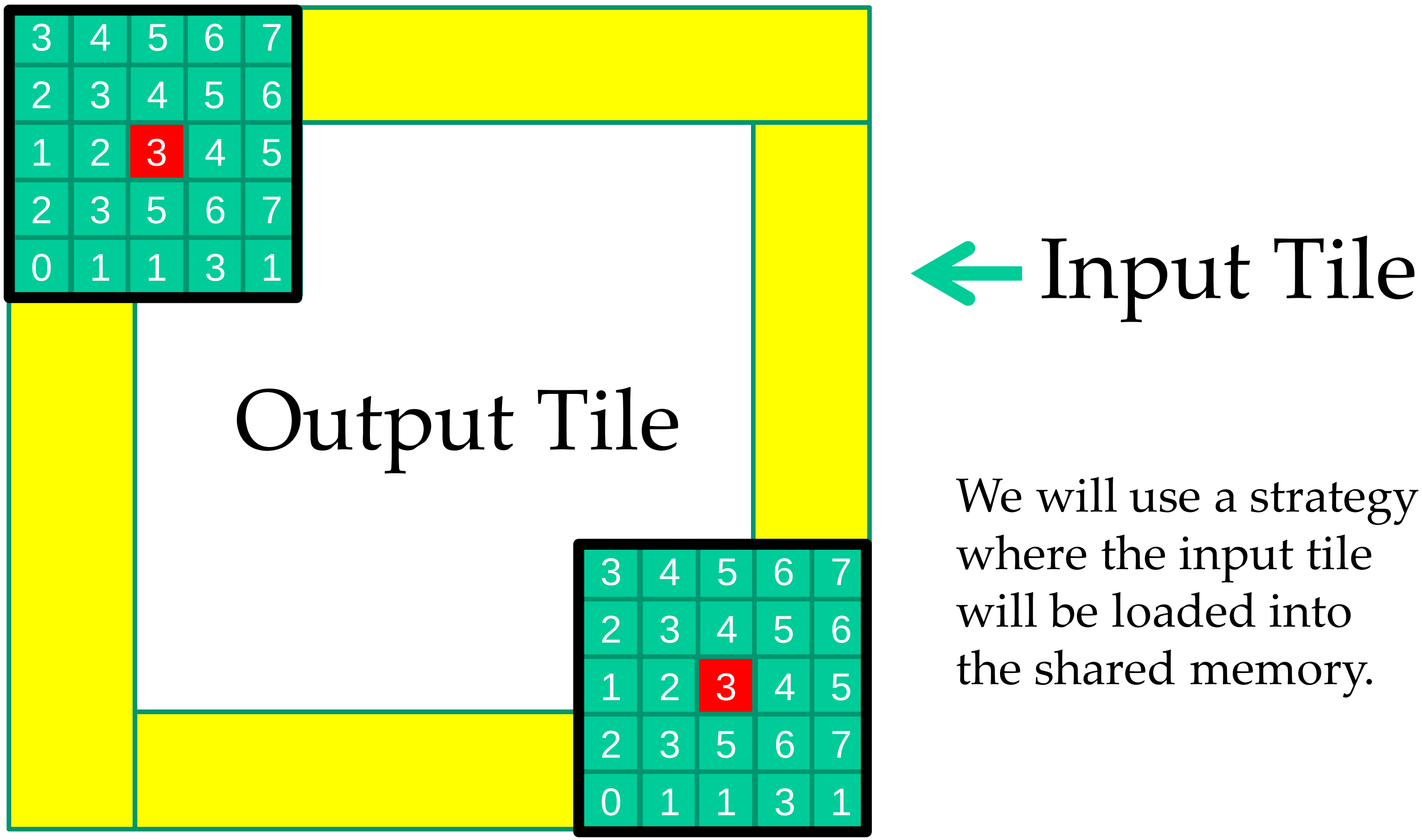


Tiling N

- 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

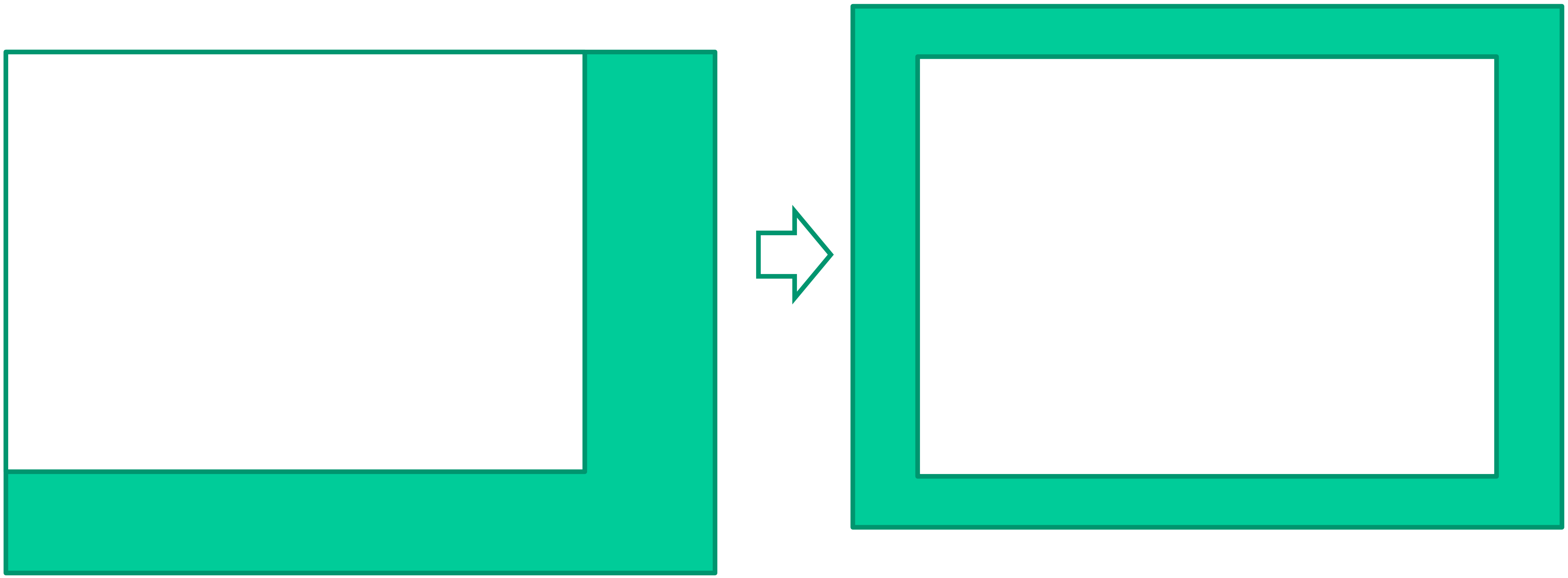


We will use a strategy where the input tile will be loaded into the shared memory.

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

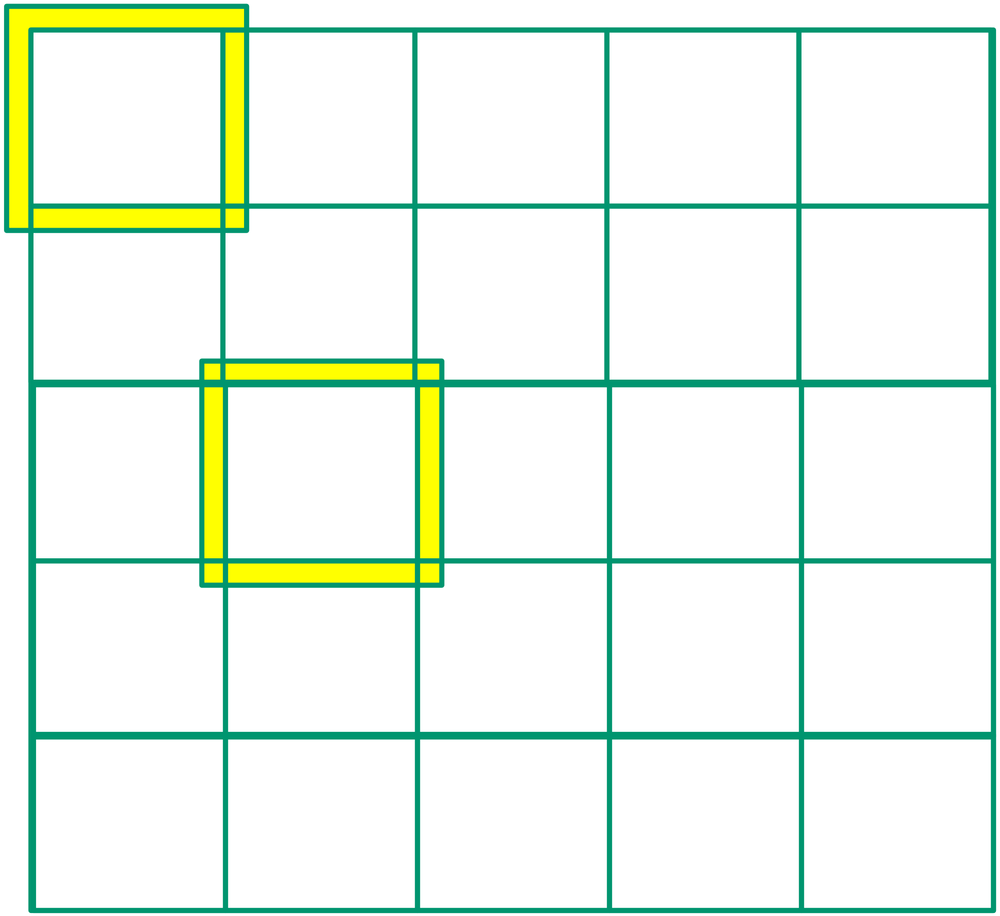
Shifting from output coordinates to input coordinates



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
  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

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

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

Bandwidth Reduction in 2D

- The reduction is

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

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