GPU Programming

(in CUDA)

Summer 2019

Designed by Julian Gutierrez, Presented by Nicolas Agostini

Session 7

NUCAR

Outline

- Applications
  - Parallel Reduction
  - Prefix Sum (Scan)
  - Histogram
  - Convolution

Parallel Reduction

- A popular class of computation
- Goal: To master the concept of control divergence through reduction trees (how to be work efficient).

Parallel Reduction

- Use: Summarize a set of input values into one value using a "reduction operation"
  - Max
  - Min
  - Sum
  - Product

- You need to 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
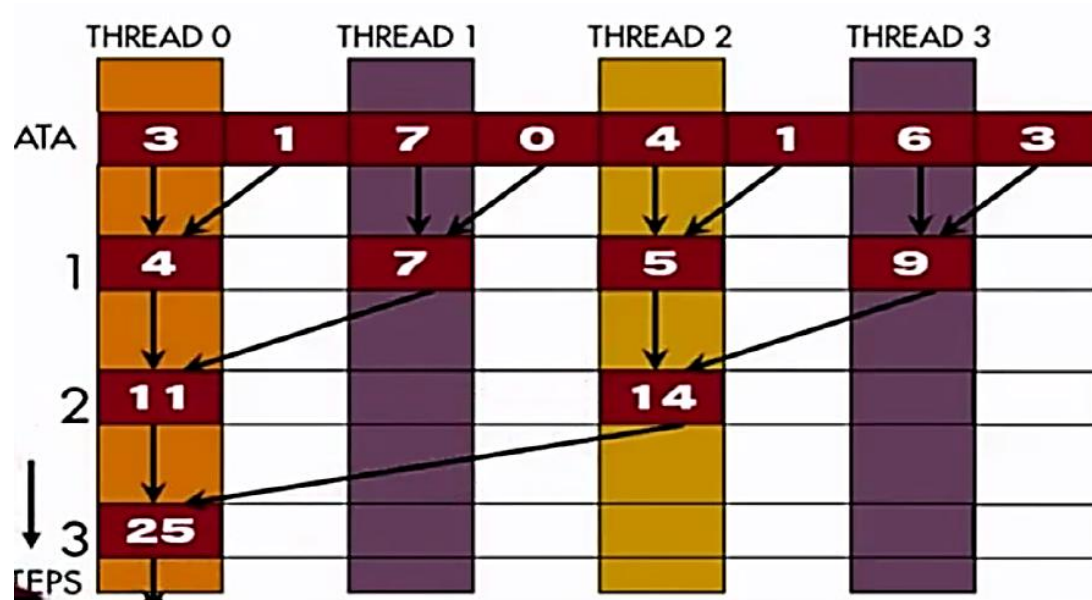 - 1 for product reduction

Parallel Reduction

- Example
- Calculate Max Value from vector:
- A parallel reduction tree algorithm performs N-1 operations in log(N) steps

- An example

Parallel Reduction

- For N input values, the reduction tree performs:
    - $(1/2)N + (1/4)N + (1/8)N + \dots (1/N) =$ 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

Parallel Reduction

- 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 process a chunk
  - Use a reduction tree to summarize the results from each chunk into the final answer
- Google and Hadoop MapReduce frameworks are examples of this pattern
- We will focus on the reduction tree step for now.

- Example: Handling 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

Parallel Reduction

- An example
  - Each thread block takes 2*BlockDim input elements
  - Each threads loads 2 elements into shared memory

```
__ 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];
```

Parallel Reduction

- Why do we need syncthreads()?

```
for (unsigned int stride = 1;
    stride <= blockDim.x;  stride *= 2)
{
    _ _syncthreads();
  if (t % stride == 0)
    partialSum[2*t]+= partialSum[2*t+stride];
}
```

- For the case where the vector is larger than the size of the block:

- Thread 0 in each thread block write the sum of the thread block in partialSum[0] into a vector indexed by the blockIdx.x

- This means that we would have to iterate the algorithm and launch another kernel until we only have one block left
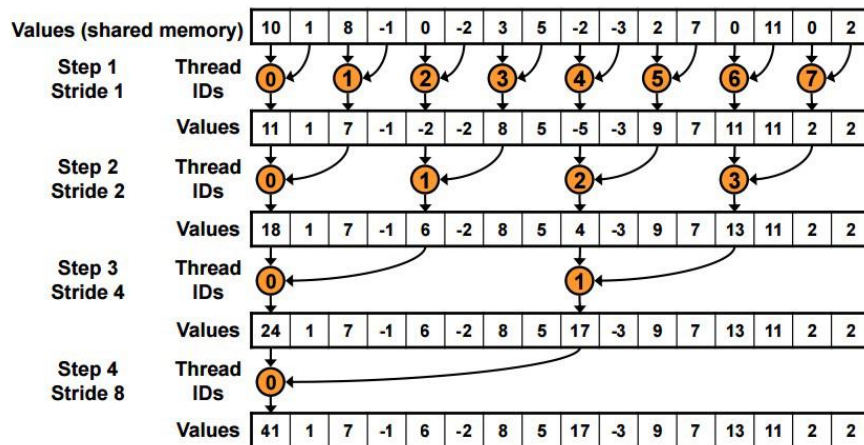
Parallel Reduction

Parallel Reduction: Interleaved Addressing
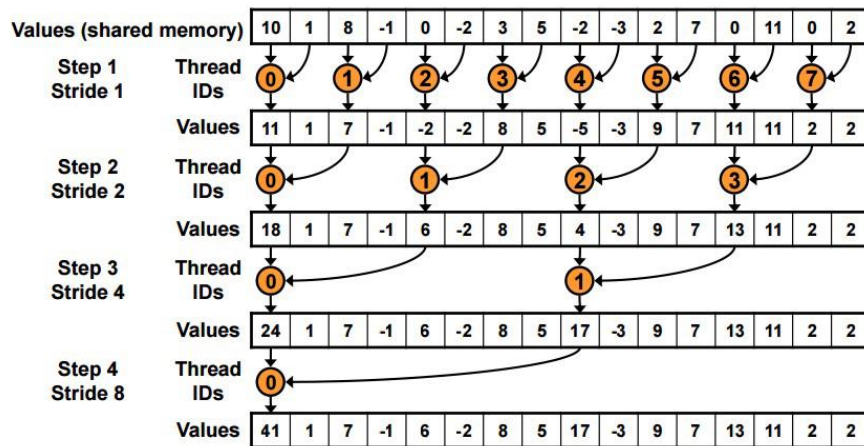
Parallel Reduction

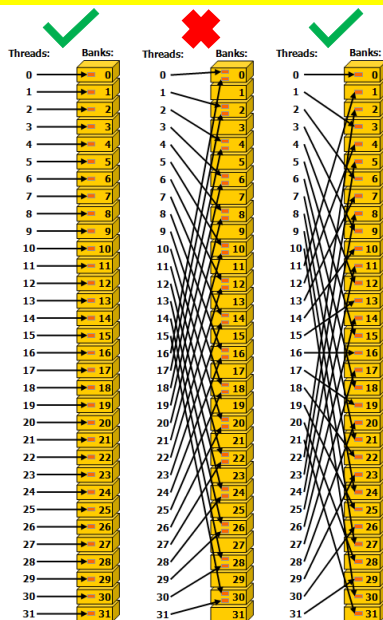Parallel Reduction: Interleaved Addressing

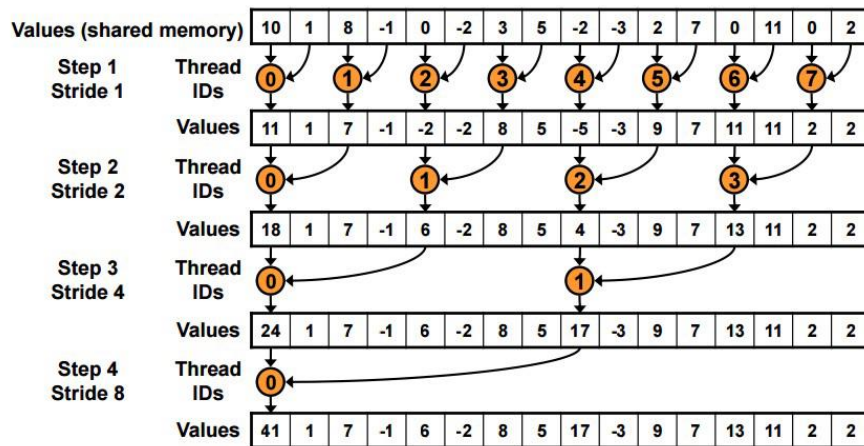What happens if the warp has more than 16 threads executing?

Parallel Reduction

Remember! Shared memory may have bank conflicts

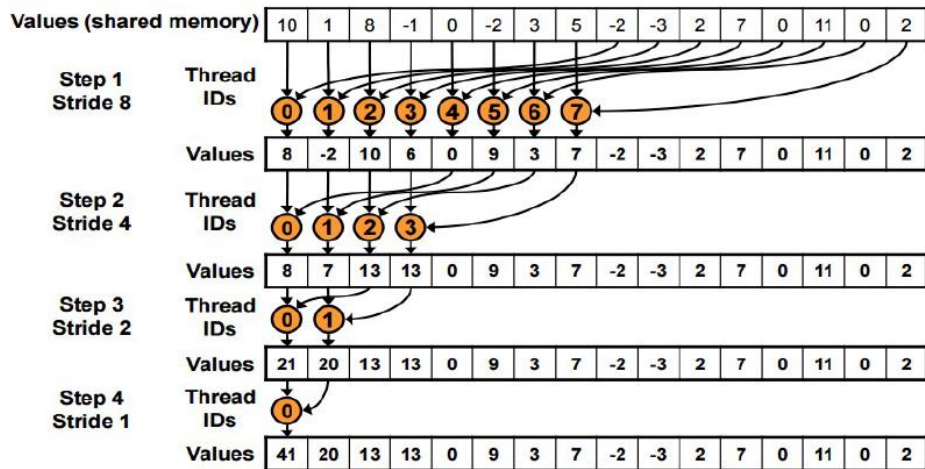**Parallel Reduction: Interleaved Addressing**

Load left side:
Thread 0
  load values[0]
Thread 1
  load values[2]
...
Thread 16
  load values[32]
Thread 17
  Load values[34]
...

Load right side:
Thread 0
  load values[1]
Thread 1
  load values[3]
...
Thread 16
  load values[33]
Thread 17
  Load values[35]
...

New Problem: Shared Memory Bank Conflicts

12

Parallel Reduction

- A better reduction?

Parallel Reduction: Sequential Addressing

Sequential addressing is conflict free

14

Parallel Reduction

- There's a problem
  - Idle threads!
  - Half of the threads are idle on first loop iteration!
  - This is wasteful....

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
    if (tid < s) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

Parallel Reduction

- Halve the number of blocks, and replace single load:

```
// each thread loads one element from global to shared mem
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
sdata[tid] = g_idata[i];
__syncthreads();
```

- With two loads and first add of the reduction:

```
// perform first level of reduction,
// reading from global memory, writing to shared memory
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```

Parallel Reduction

- Unrolling the last warp
  - As reduction proceeds, # "active" threads decreases
    - When s <= 32, we have only one warp left
  - Instructions are SIMD synchronous within a warp
  - That means when s <= 32:
    - We don't need to __syncthreads()
    - We don't need "if (tid < s)" because it doesn't save any work
  - Lets unroll the last 6 iterations of the inner loop

Parallel Reduction

- Note: This saves useless work in all warps, not just the last one!
- Without unrolling, all warps execute every iteration of the for loop and if statement

```
__device__ void warpReduce(volatile int* sdata, int tid) {
    sdata[tid] += sdata[tid + 32];
    sdata[tid] += sdata[tid + 16];
    sdata[tid] += sdata[tid +  8];
    sdata[tid] += sdata[tid +  4];
    sdata[tid] += sdata[tid +  2];
    sdata[tid] += sdata[tid +  1];
}
```

IMPORTANT:
For this to be correct,
we must use the
"volatile" keyword!

```
// later...
for (unsigned int s=blockDim.x/2; s>32; s>>=1) {
    if (tid < s)
        sdata[tid] += sdata[tid + s];
    __syncthreads();
}

if (tid < 32) warpReduce(sdata, tid);
```

Parallel Reduction

```cpp
template <unsigned int blockSize>
__device__ void warpReduce(volatile int *sdata, unsigned int tid) {
  if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
  if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
  if (blockSize >= 16) sdata[tid] += sdata[tid +  8];
  if (blockSize >=  8) sdata[tid] += sdata[tid +  4];
  if (blockSize >=  4) sdata[tid] += sdata[tid +  2];
  if (blockSize >=  2) sdata[tid] += sdata[tid +  1];
}
template <unsigned int blockSize>
__global__ void reduce6(int *g_idata, int *g_odata, unsigned int n) {
  extern __shared__ int sdata[];
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x*(blockSize*2) + tid;
  unsigned int gridSize = blockSize*2*gridDim.x;
  sdata[tid] = 0;

  while (i < n) { sdata[tid] += g_idata[i] + g_idata[i+blockSize];  i += gridSize;  }
  __syncthreads();

  if (blockSize >= 512) { if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads(); }
  if (blockSize >= 256) { if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads(); }
  if (blockSize >= 128) { if (tid <  64) { sdata[tid] += sdata[tid +  64]; } __syncthreads(); }

  if (tid < 32) warpReduce(sdata, tid);
  if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Final Optimized Kernel

nVIDIA.

35

Parallel Reduction

- Please note: this warp synchronized technique only works in older models. After Volta, this technique is no longer valid due to the change in GPU architecture.
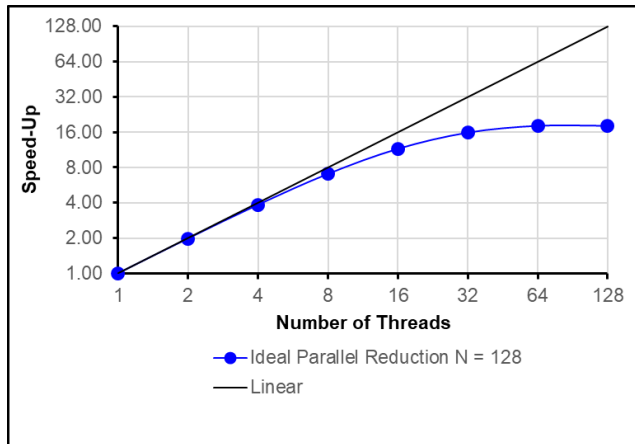
- More on this:

  - https://devblogs.nvidia.com/cooperative-groups/

  - https://devblogs.nvidia.com/inside-volta/

Thoughts on Reduction for CPU

NUCAR

$n = \log_2 Row\ Size$

$tp = \log_2 Num\ Threads$

$Ideal\ Exec\ Time = \max(1, 2^{n-tp}) - 1 + \min(n, tp)$

Ideal Parallel Reduction N = 128
Linear

**Row Matrix Size: 48000
Sparsity: >99 %
Average Row Density: ~128**

- Why reduction is not ideal
  - Cache sharing between threads in different cores (inefficient memory accesses), looses locality of having each thread access all consecutive memory
  - Reduction requires synchronization between threads (after each step) = reduced efficiency
  - Reduction ideal speedup is lower than linear in certain scenarios
  - Row parallelism does the same amount of work at the end, though threads can focus on independent work

- Definition
  - Prefix sum, cumulative sum, inclusive scan, or simply scan of a sequence of numbers x_0, x_1, x_2, ... Gives a second sequence of numbers y_0, y_1, y_2, ..., the sums of prefixes (running totals) of the input sequence.
  - Any binary operation (not just the addition operation).
- Example
  - If (+) is addition, then scan on the set: [3 1 7 0 4 1 6 3]
    - Returns the set: [0 3 4 11 11 15 16 22]

Note: Exclusive scan: last input element is not included in the result

- A Naïve inclusive parallel Scan

  - Assign one thread to calculate each y element

  - Have every thread add up all x elements needed for

the y element

$$y0 = x0$$
$$y1 = x0 + x1$$
$$y2 = x0 + x1 + x2$$

- Note: Parallel programming is easy as long as you do not care about performance

- Hillis Steele Scan: A slightly better version compared to naïve version.

- Iterate log(n) times: Threads stride to n: Add pairs of elements stride elements apart. Double stride at each iteration.

- This scan algorithm is not that work efficient
  - Sequential scan algorithm does n−1 adds
  - How many does this one do?
  - What happens if the # of elements is 10^6?
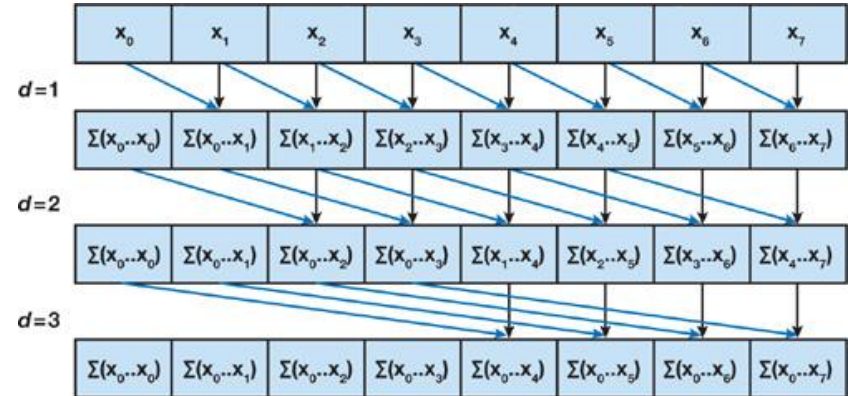
Prefix Sum (Scan)

- Even better one: Blelloch Scan.
- Reduces in log(n), downsweeps in log(n).
- Total steps: 2 log(n)
- This scan algorithm is work efficient
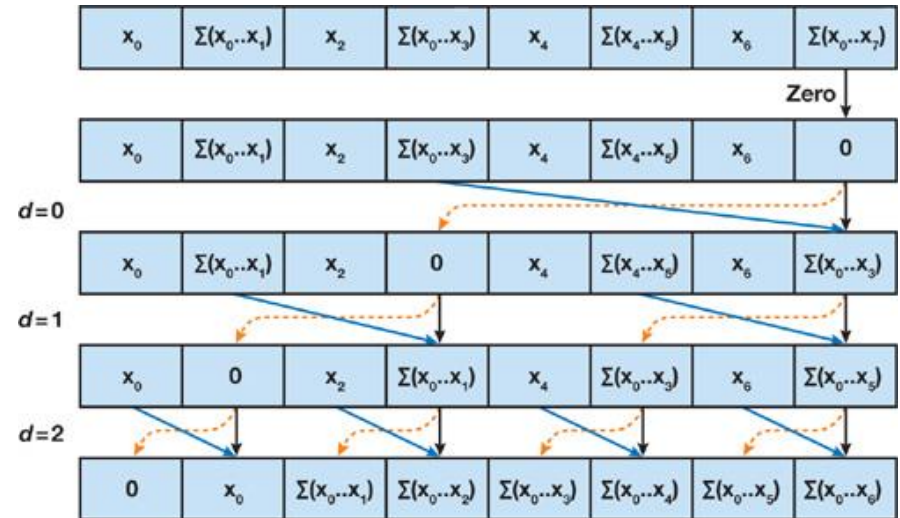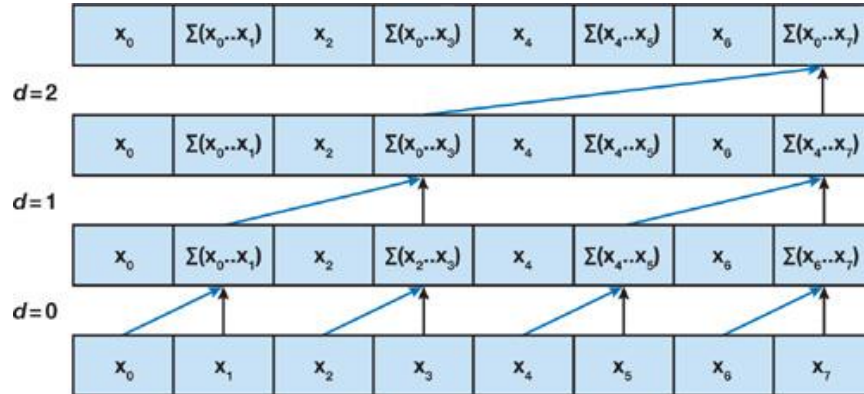
Prefix Sum (Scan)

- Useful in implementation of several parallel algorithms:
  - Radix sort
  - Quicksort
  - String comparison
  - Lexical analysis
  - Stream compaction
  - Polynomial evaluation
  - Solving recurrences
  - Tree operations
  - histograms

- Examples
  - Assigning camp slots
  - Assigning farmer market space
  - Allocating memory to parallel threads
  - Allocating memory buffer for communication channels

More info: https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch39.html
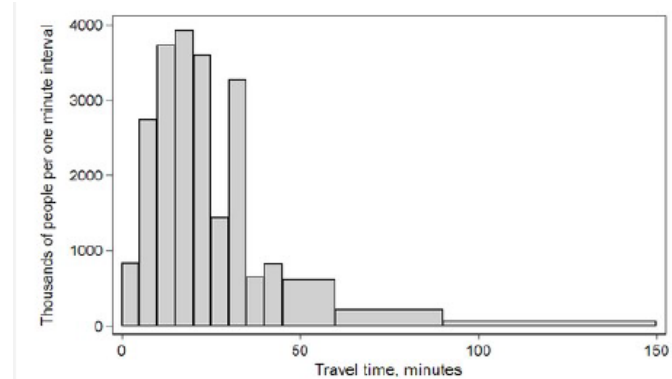
Histogram

- A histogram is a graphical representation of the distribution of numerical data.

  - Bar Graph

- Serial Algorithm

```
For (i=0; I < BIN_COUNT; i++)
    result[i]=0;
For (i=0; I < measurements.size(); i++)
    result[computeBin(measurements[i])]++;
```
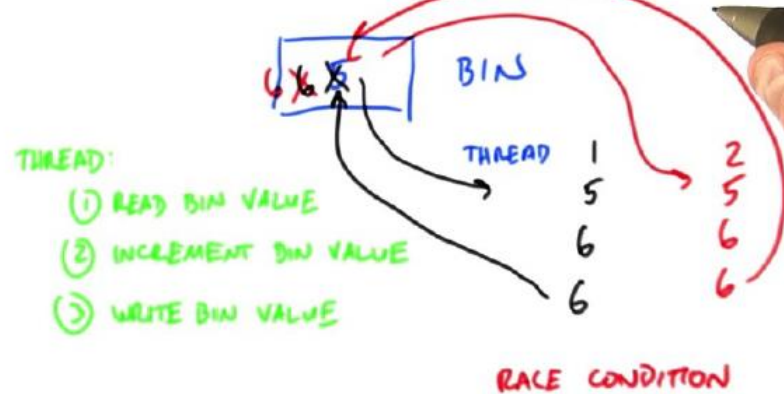
- Naïve implementation:

```
__global__ void naive_histo(int *d_bins, const int *d_in, const int BIN_COUNT)
{
    int myId = threadIdx.x + blockDim.x * blockIdx.x;
    int myItem = d_in[myId];
    int myBin = myItem % BIN_COUNT;
    d_bins[myBin]++;
}
```

- Naïve implementation:

```
__global__ void naive_histo(int *d_bins, const int *d_in, const int BIN_COUNT)
{
    int myId = threadIdx.x + blockDim.x * blockIdx.x;
    int myItem = d_in[myId];
    int myBin = myItem % BIN_COUNT;
    d_bins[myBin]++;
}
```


WHY THE OBVIOUS METHOD DOESN'T WORK

BIN

THREAD:
① READ BIN VALUE
② INCREMENT BIN VALUE
③ WRITE BIN VALUE

THREAD 1   2
        5   5
        6   6
        6   6

RACE CONDITION

- Naïve implementation using atomic operations (Method 1):

  - This will avoid RAW hazards.

```cuda
__global__ void simple_histo(int *d_bins, const int *d
{
    int myId = threadIdx.x + blockDim.x * blockIdx.x;
    int myItem = d_in[myId];
    int myBin = myItem % BIN_COUNT;
    atomicAdd(&(d_bins[myBin]), 1);
}
```

Histogram

- Naïve implementation using atomic operations (Method 1):
  - This will avoid RAW hazards.

```
__global__ void simple_histo(int *d_bins, const int *d
{
    int myId = threadIdx.x + blockDim.x * blockIdx.x;
    int myItem = d_in[myId];
    int myBin = myItem % BIN_COUNT;
    atomicAdd(&(d_bins[myBin]), 1);
}
```

QUIZ
- Histogram with 1M elements
- You can choose # of bins

10   100   1000
▢   ▢   ▢

- Redefining the method. Local histogram + reduction (Method 2)

PER-THREAD PRIVATIZED (LOCAL) HISTOGRAMS, THEN REDUCE

128 ITEMS · 8 THREADS · 3 BINS

(EACH THREAD GETS 16 ITEMS)

THREAD 0   1   2   7

BIN 0 □   □   □   □

1 □   □   □   ...   □

2 □   □   □   □

- Redefining the method. Local histogram + reduction (Method 2)

PER-THREAD PRIVATIZED (LOCAL) HISTOGRAMS, THEN REDUCE
128 ITEMS · 8 THREADS · 3 BINS
(EACH THREAD GETS 16 ITEMS)

THREAD 0   1   2   7

BIN 0
1
2

QUIZ: DO WE NEED ATOMICS TO MANAGE ACCESS TO THESE LOCAL PER-THREAD HISTOGRAMS?   YES   NO

Histogram

- Redefining the method. Sort then reduce by key (Method 3)
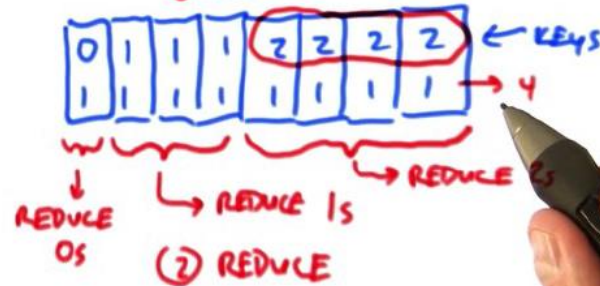
Histogram

- Final thoughts on histogram:

  - Using Atomic operations

  - Using per-thread histograms, and then reduce

  - Sort, then reduce by key

- Question

  - 256 elements, 8 bins. How many atomic adds are needed?

    - Naive atomic technique. (256 threads)

    - Processing 16 elements per thread with local histogram, and then atomics.

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
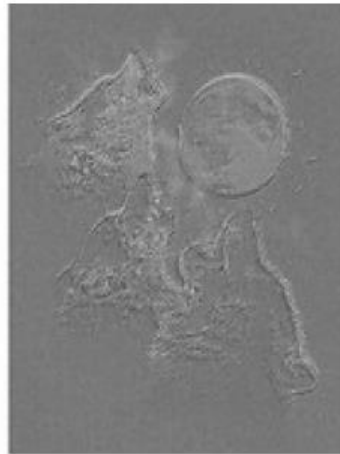
Convolution

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

- The weights used in the weighted sum calculation are defined by an input mask array, commonly referred to as the convolution kernel

  - We will refer to these mask arrays as convolution masks to avoid confusion.

  - The same convolution mask is typically used for all elements of the array.

- ## 1D Convolution example

  - ### Commonly used for audio processing

   - Mask size is usually an odd number of elements for symmetry
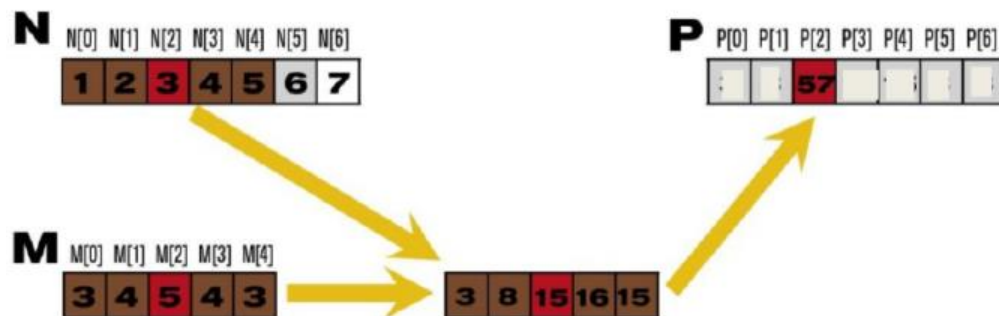
- Example
  - A causal discrete-time FIR filter of order N, each value of the output sequence is a weighted sum of the most recent input values:

$$y[n] = b_0 x[n] + b_1 x[n-1] + \cdots + b_N x[n-N]$$
$$= \sum_{i=0}^{N} b_i \cdot x[n-i],$$

  - Where:
    - X[n] is the input signal
    - Y[n] is the output signal,
    - N is the filter order, an Nth-order filter has (N+1) terms on the right-hand side
    - Bi is the value of the impulse response at the ith instance for 0<= I <= N of an Nth-order filter. If the filter is a direct form FIR filter then bi is also a coefficient of the filter.
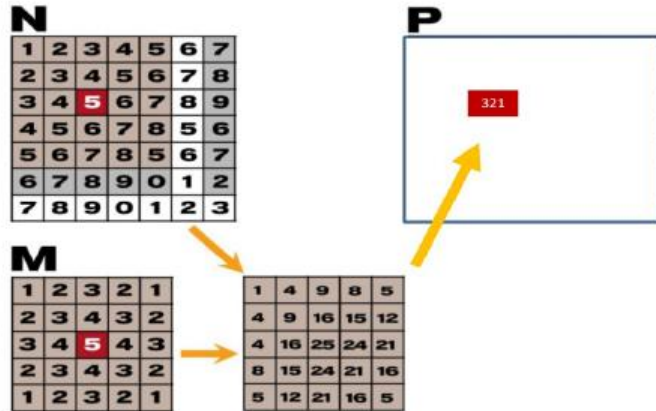
- Calculation of output elements near the boundaries (beginning and end) of the input array need to deal with "ghost" elements

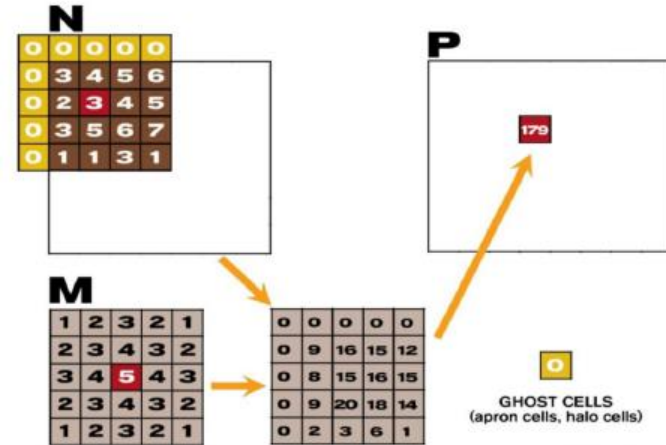 - Different policies (0, replicates of boundary values, etc.)
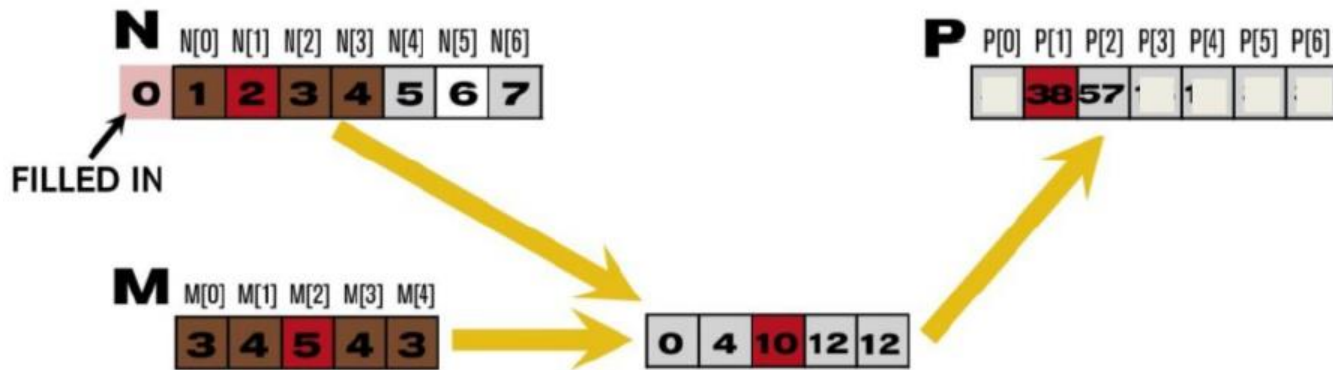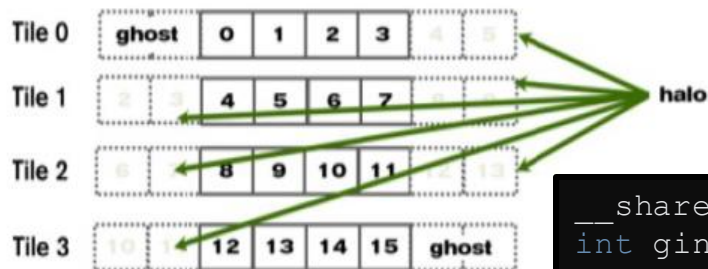
• 2D convolution

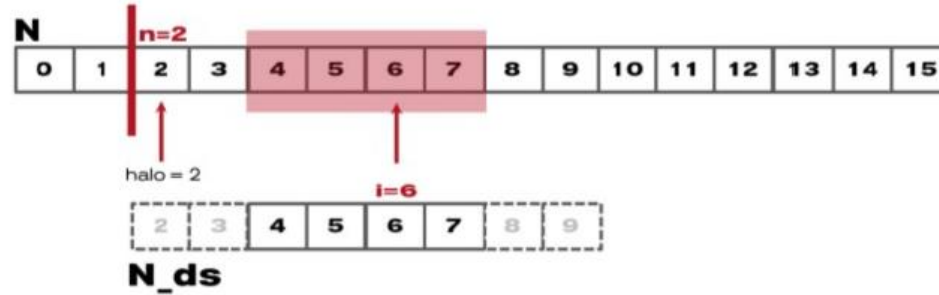2D convolution – ghost cells

- Tiled Convolution for 1D


```cuda
__shared__ double N_ds[BLOCK_SIZE + 2 * RADIUS];
int gindex = threadIdx.x + blockIdx.x * blockDim.x;
int lindex = threadIdx.x + RADIUS;
```
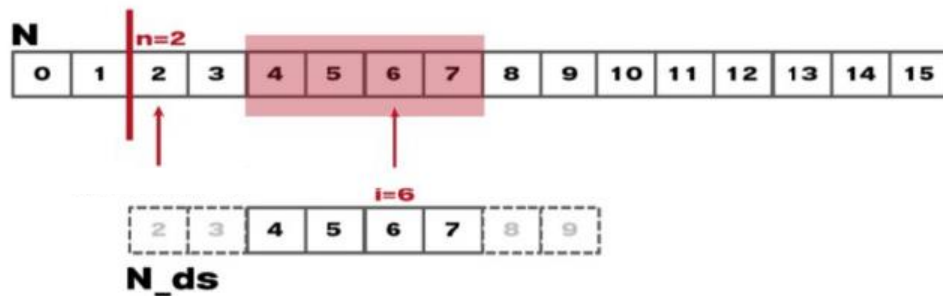
Convolution

- Loading the internals


```
// Read input elements into shared memory
N_ds[lindex] = in[gindex]
```

- Loading the left halo


```
if (threadIdx.x < RADIUS) {
    N_ds[lindex - RADIUS] = (gindex - RADIUS >= 0) ? in[gindex - RADIUS]: 0.0;
}
```
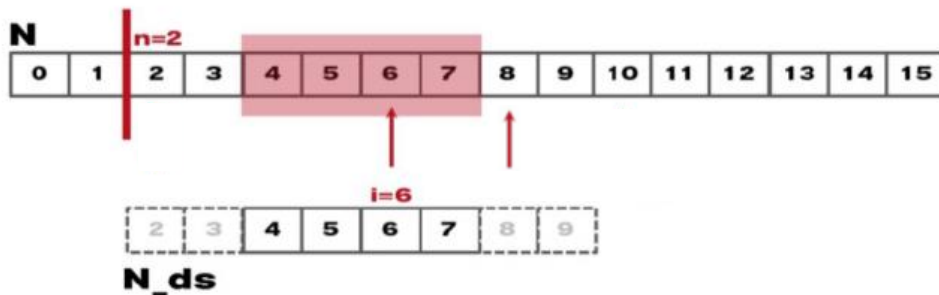
Convolution

- Loading the right halo


```
if (threadIdx.x < RADIUS) {
    N_ds[lindex + BLOCK_SIZE] = (gindex+BLOCK_SIZE < vector_size) ? in[gindex + BLOCK_SIZE]: 0.0;
}
```

```
__global__ void stencil_shared(double *in, double *out, int vector_size) {

    __shared__ double temp[BLOCK_SIZE + 2 * RADIUS];

    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;

    if (gindex < vector_size){

        // Read input elements into shared memory
        temp[lindex] = in[gindex];
        // At both end of a block, the sliding window moves beyond the block boundary.
        // E.g, for thread id = 512, we will read in[505] and in[1030] into temp.
        if (threadIdx.x < RADIUS) {
            temp[lindex - RADIUS] = (gindex - RADIUS >= 0) ? in[gindex - RADIUS]: 0.0;
            temp[lindex + BLOCK_SIZE] = (gindex + BLOCK_SIZE < vector_size) ? in[gindex + BLOCK_SIZE]: 0.0;
        }

    } else {
        temp[lindex] = 0.0;
    }

    __syncthreads();

    if (gindex < vector_size){
        // Apply the stencil
        double result = 0.0;
        //#pragma unroll
        for (int offset = -RADIUS ; offset <= RADIUS ; ++offset)
            result += temp[lindex + offset];

        // Store the result
        out[gindex] = result;
    }
}
```

Convolution

- Example: Shared memory data reuse, with radius of 2.

- A stencil/convolution operation has a lot of potential for data reuse.

- Image processing algorithms can commonly use this resource as well.

- The Sobel filter is an example of this algorithm we will cover in the image processing class.

N_ds   Mask_Width is 5

| 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |