Histograms

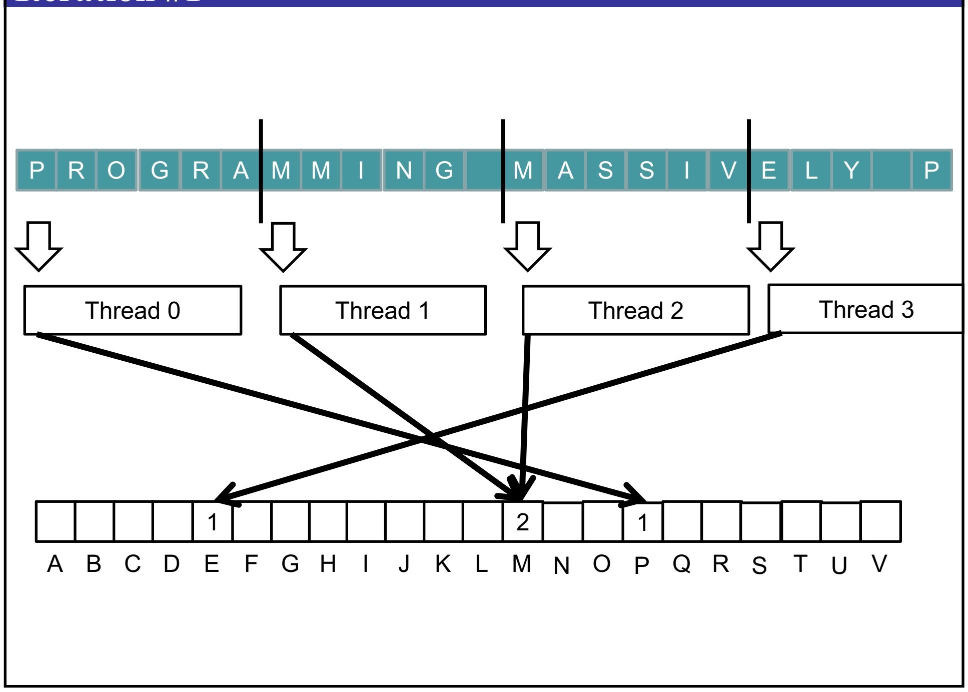
Nikos Hardavellas

Some slides/material from:
UToronto course by Andreas Moshovos
UIUC course by Wen-Mei Hwu and David Kirk
UCSB course by Andrea Di Blas
Universitat Jena by Waqar Saleem
NVIDIA by Simon Green and many others
Patterson & Hennessy, Computer Organization: The HW/SW Interface
Cedric Nugteren, Gert-Jan van den Braak, Henk Corporaal, Bart Mesman

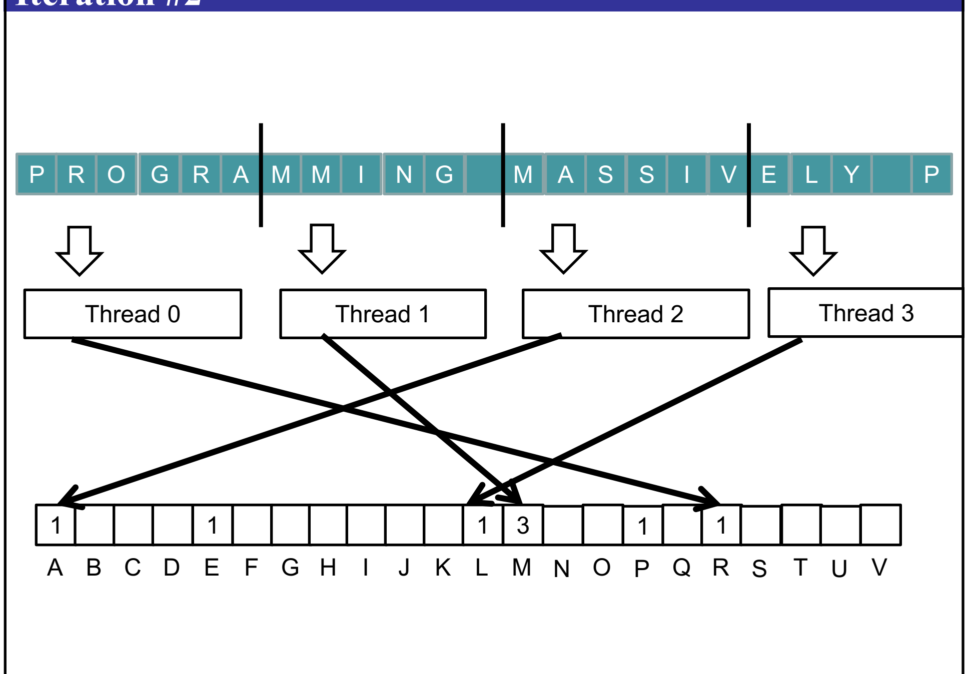
A Histogram Example

- In the sentence
  "Programming Massively Parallel Processors"
  build a histogram of frequencies of each letter
- A(4), C(1), E(1), G(1), …
- How do you do this in parallel?

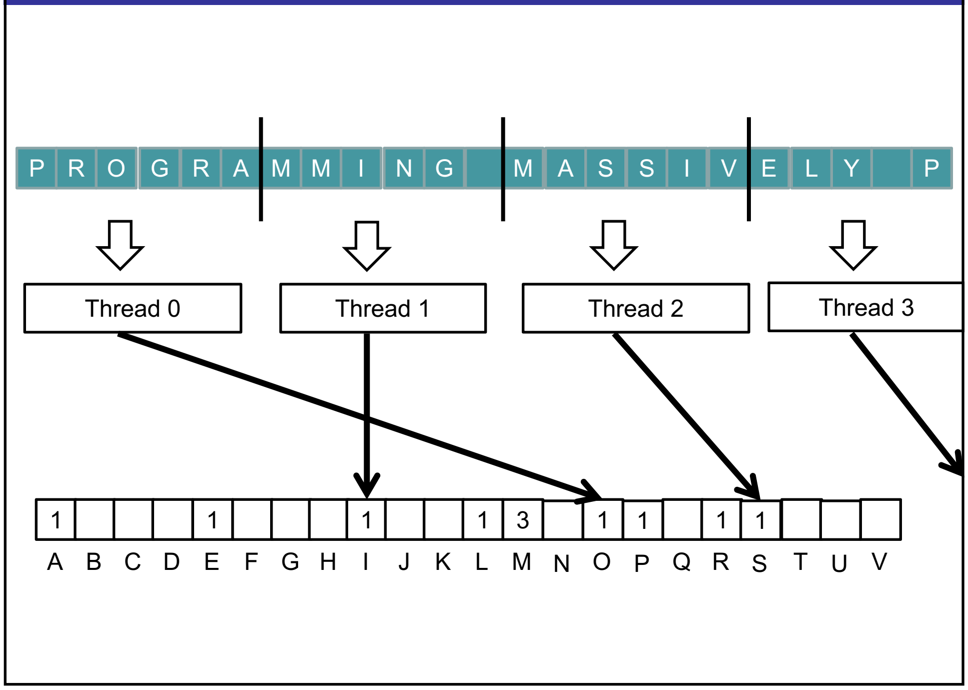
Iteration #1



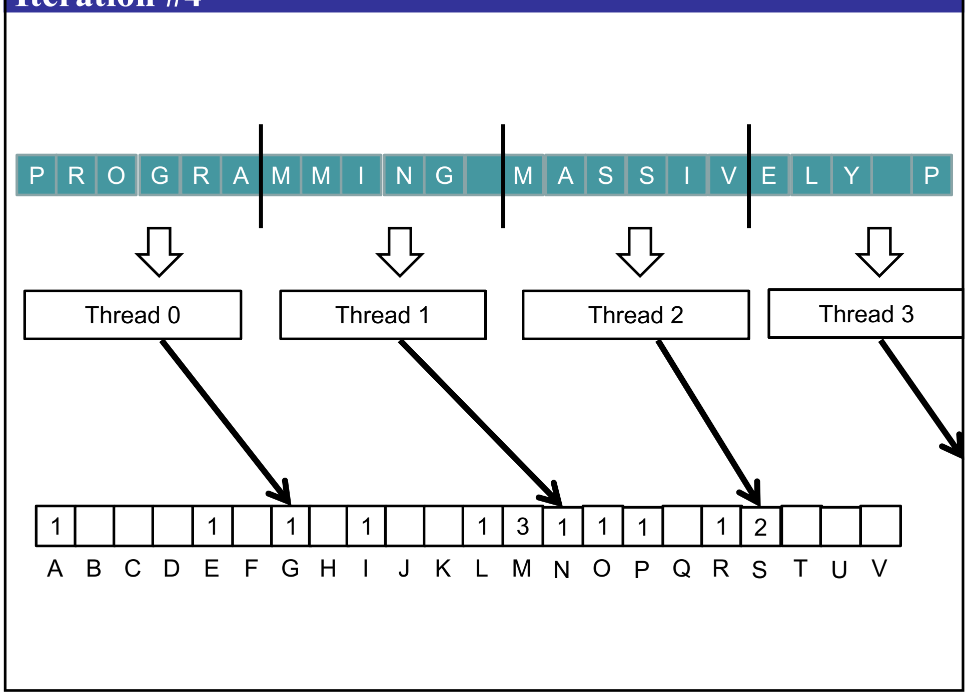
Iteration #2



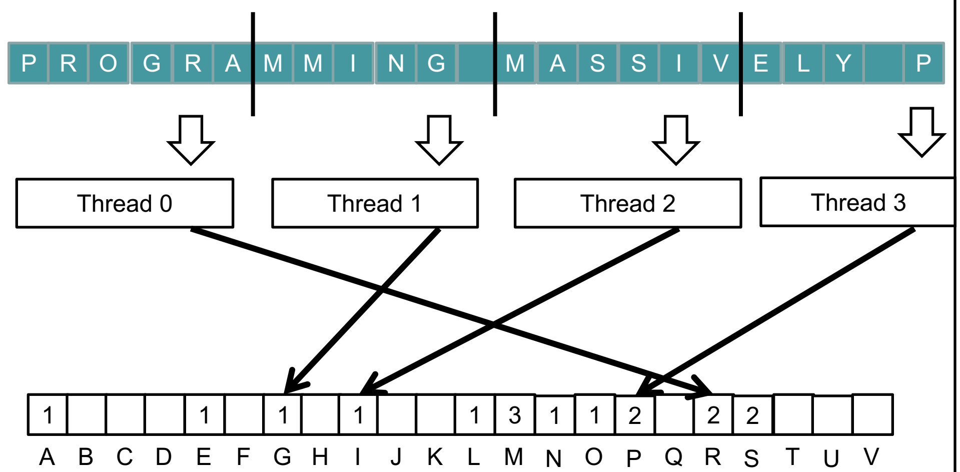
Iteration #3



Iteration #4



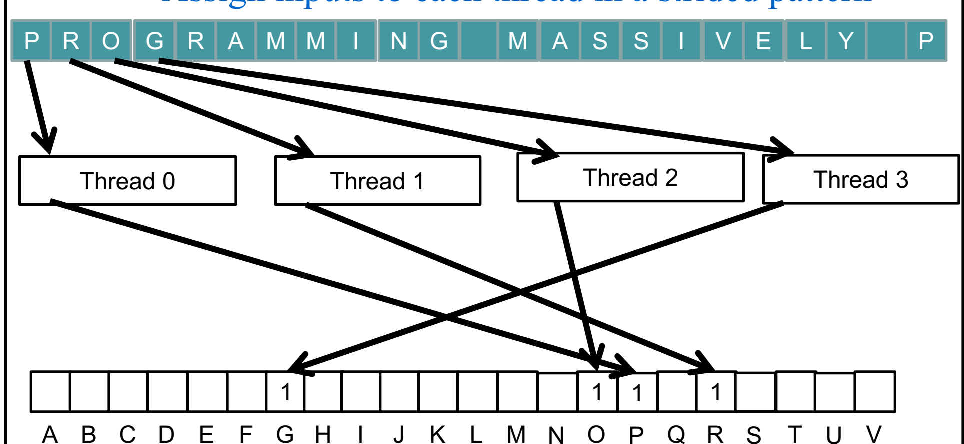
Iteration #5



What is wrong with this algorithm?

What is wrong with the algorithm?

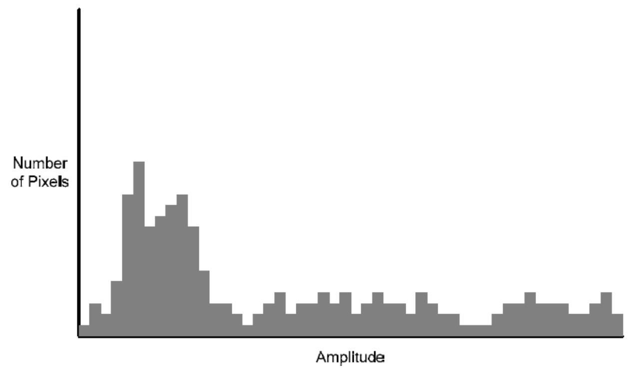
- Reads from the input array are not coalesced
  - Assign inputs to each thread in a strided pattern



Races & conflicts in results array!!!

Histogram

- E.g., Given an image calculate this:



- Distribution of values

Sequential Algorithm

```
for (int i = 0; i < BIN_COUNT; i++)
    result[i] = 0;

for (int i = 0; i < dataN; i++)
    result[data[i]]++;
```

The challenge is that the write access pattern is data dependent

No ordering of accesses to memory

Data Race

- Thread 1 Thread 2

  X++ X++

- X++ is really
  - **tmp** = Read x
  - **tmp++**
  - Write **tmp** into x

Data Race

- Start with X = 10
- Thread 1 Thread 2

Thread 1	Thread 2
tmp = X (10)	tmp = X (10)
tmp++ (11)	tmp++ (11)
X = tmp (**11**)	X = tmp (**11**)

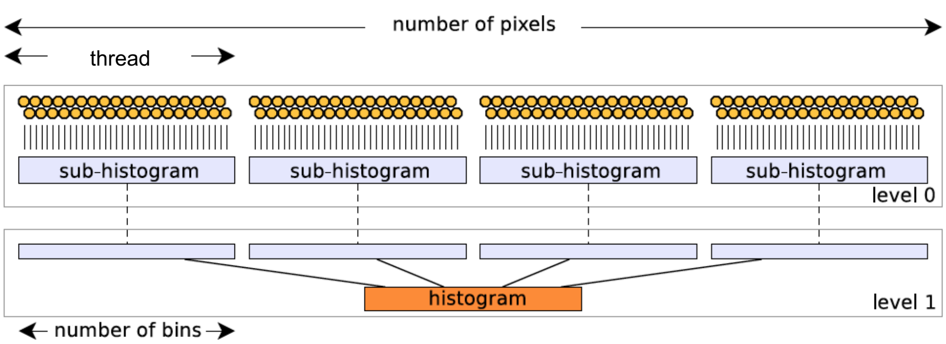
- Thread 1 Thread 2

Thread 1	Thread 2
tmp = X (10)	ZZzzzz
tmp++ (11)	ZZzzzz
X = tmp (11)	ZZzzzz
	tmp = X (11)
	tmp++ (12)
	X = tmp (**12**)

Parallel Strategy

- Distribute work across multiple blocks
  - Divide input data to blocks
- Each block process its own portion (**privatization**)
  - Multiple threads
    - #pixels / #threads pixels per thread
  - Produces a partial histogram per thread
    - Could produce multiple histograms
    - One per thread → no ordering problems (data races) here
- Merge all partial histograms
  - Produces the final histogram

Parallel Strategy #1



- Input array in global memory
- Histogram array in shared memory
- One histogram per thread
- One column per possible pixel value

Privatization

- Privatization is one of the most powerful and frequently used techniques for parallelizing applications
- Each thread works on its private data; merge later
- The operation needs to be
  - Commutative: $x * y = y * x$
  - Associative: $(x * y) * z = x * (y * z)$
  - Histogram add operation is associative and commutative

What do we have control over?

```
result[ data[i] ] ++;
```

- **Data[ ]:**
  - *We control its access*
    - Can be accessed sequentially
  - Each element accessed only once
  - No races on data[ ] – it is read only
- **Result[ ]:**
  - Access is data-dependent
  - Each element may be accessed multiple times
  - Different pixels, read by different threads, same amplitude
- *We control memory placement*
  - Data[ ] in global memory (or constant?)
  - Result[ ] in shared memory
    - Needs to be small enough to fit in shared memory

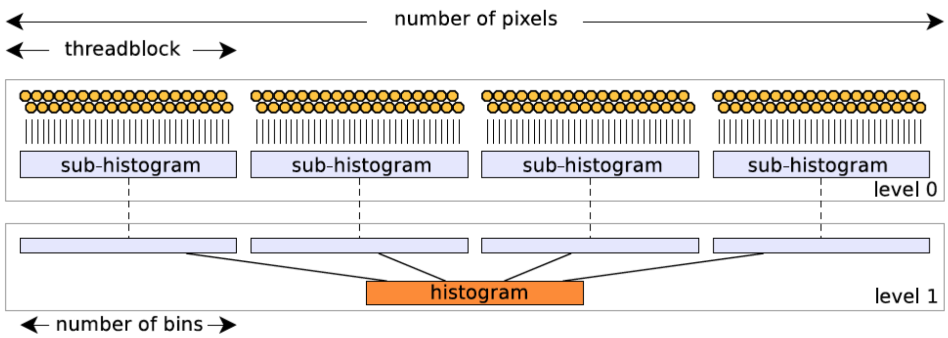
Sub-Histograms

- How big is each histogram (KB)?
  - Input value range: 0-255, 1 byte
  - Each histogram needs 256 entries
  - How many bytes per entry?
    - That's data dependent
  - Let's assume 32-bits or 4 bytes: 256*4 = 1KB / histogram
- How many sub-histograms can we fit in shared mem?
  - Max shared mem in GTX680: 48KB
  - 48KB shared mem → only 1 warp (32KB in histograms)
  - Use only 1 warp per SM
  - Use only 32 out of 192 cores per SM
  - Unused hardware, low occupancy

Sub-Histograms

- Let's try one histogram per block
  - Many threads per block
  - Ordering problem persists but within a block
  - However, threads within a block can ***synchronize!!!***
  - ***synchronization → no more race conditions***

Parallel Strategy #2



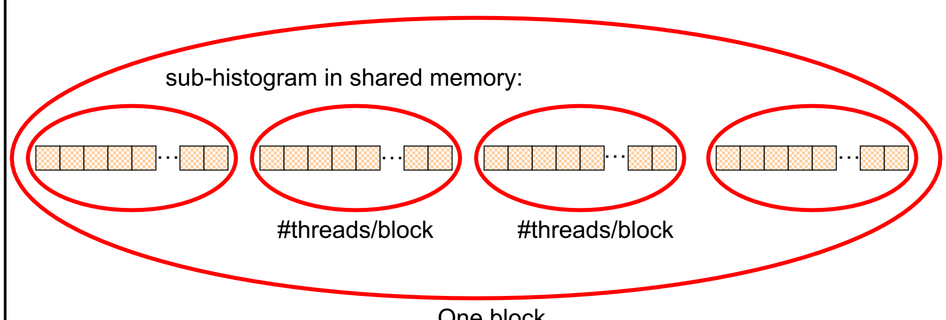
- Input array in global memory
- Histogram array in shared memory
- One histogram per threadblock; all threads in block update it
- One column per possible pixel value
- Each block updates global histogram at the end

Dataset Sketch for Global Histogram Update

Histogram in global memory:

But, it could be #bins > #threads/block

→ Break histogram into #threads/block pieces, work them iteratively



All threadblocks may be doing the update simultaneously!

Algorithm Overview

- Step 1:
  - Initialize partial histogram
  - Each thread:
    - s_Hist[index] = 0
    - index += threads per block
      - Until all bins are taken care of
- Step 2:
  - Generate partial histogram
  - Each thread:
    - read data[index]
    - **update s_Hist[]** ← **conflicts possible (within block)**
    - index += Total number of threads
      - Until all input/block is taken care of
- Step 3:
  - Update global histogram
  - Each thread
    - read s_hist[index]
    - **update global histogram** ← **conflicts possible (across blocks)**
    - index += threads per block
      - Until all bins are taken care of

Simultaneous Updates?

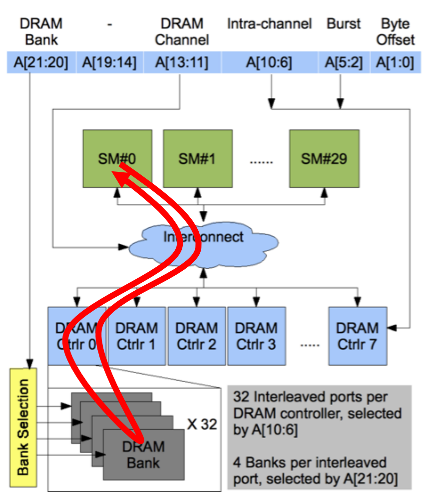
- **Threads in a block:**
  - **update s_Hist[]**
- **All threads:**
  - **update global histogram**

- Without special support this becomes:
  - register X = value of A
  - X ++
  - A = register X
- This is a **read-modify-write** sequence

The problem with simultaneous updates

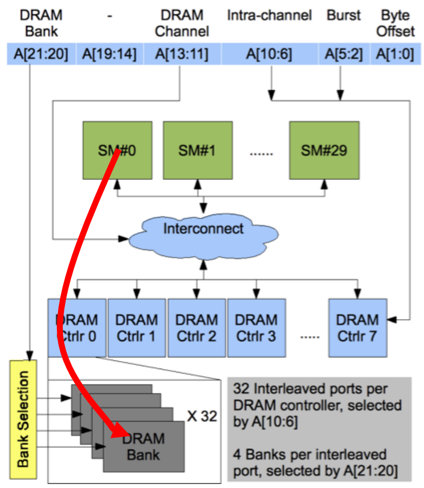
- What if we do each step individually
  - r10 = mem[100]   10   r100 = mem[100] 10
  - r10++   11   r100++   11
  - mem[100] = r10   11   mem[100] = r100 11
- But we really wanted 12
- What if we had 32 threads running in parallel?
- Starting with 10 we would want: 10+32
  - We may still get 11
- Need to think about this:
  - Special support: **Atomic operations**

Atomic Operations on DRAM



- An atomic operation starts with a read, with a latency of a few hundred cycles

Atomic Operations on DRAM



- An atomic operation starts with a read, with a latency of a few hundred cycles
- The atomic operation ends with a write, with a latency of a few hundred cycles
- During this whole time, no one else can access the location
- Similar for shared memory
- Latencies have improved!

Atomic Operations

- Read-Modify-Write operations that are guaranteed to happen "atomically"
  - Produces the same result as if the sequence executed in isolation in time
  - Think of it as "serializing the execution" of all atomics
  - This is not what necessarily happens
    - This is how you should think about them

Atomic Operations

- Supported both in Shared and Global memory
- Example:
  - atomicAdd (pointer, value)
  - does: *pointer += value

- Atomic Operations
  - Add, Sub, Inc, Dec
  - Exch, Min, Max, CAS (compare-and-swap)
  - Bitwise: And, Or, Xor
- Work with (unsigned) integers
- Exch works with floats as well

atomicExch, atomicMin, atomicMax, atomicCAS

- atomicExch (pointer, value)
  - tmp = * pointer
  - *pointer = value
  - return tmp

- atomicMin (pointer, value) (max is similar)
  - tmp = *pointer
  - if (*pointer > value) *pointer = value
  - return tmp

- atomicCAS (pointer, value1, value2)
  - tmp = *pointer
  - if (*pointer == value1) *pointer = value2
  - return tmp

atomicInc, atomicDec

- atomicInc (pointer, value)
  - tmp = *pointer
  - if (*pointer < value) (*pointer)++
  - else *pointer = 0
  - return tmp
- atomicDec (pointer, value)
  - tmp = *pointer
  - if (*pointer == 0 || *pointer > value) *pointer = value
  - else (*pointer)--
  - return tmp
- Allow for wrap-around work queues

atomicAnd, atomicOr, atomicXOR

- atomicAnd (pointer, value)
  - tmp = *pointer
  - *pointer = *pointer & value
  - return tmp
- Others similar
- Now, you have two choices for the histogram:
  1. Use atomicInc(pointer, max_value)
  2. Use atomicAdd(pointer, 1)

Should you use Add or Inc ???

- atomicAdd (pointer, value)
  - tmp = *pointer
  - *pointer = *pointer + value
  - return tmp

Good!!!

- atomicInc (pointer, value)
  - tmp = *pointer
  - if (*pointer < value)
  - (*pointer)++
  - else
  - *pointer = 0
  - return tmp

- Extra work ?
- Thread Divergence ?

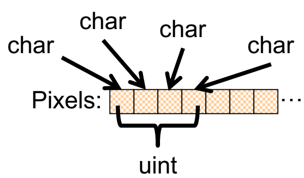
CUDA Implementation - Declarations

```
__global__ void histogram256Kernel
(uint *d_Result, uint *d_Data, int dataN){

  //Current global thread index
  const int
      globalTid = blockIdx.x * blockDim.x + threadIdx.x;

  //Total number of threads in the compute grid
  const int
      numThreads = blockDim.x * gridDim.x;

  __shared__ uint s_Hist[BIN_COUNT];
```



Why `uint *d_Data`?
Typically work with integers and floats
These are 4-byte types
Memory optimized for 4 bytes/access

Clear partial histogram buffer

```
//Clear shared memory buffer for current block
 before processing
   for ( int pos = threadIdx.x;
         pos < BIN_COUNT;
         pos += blockDim.x)
              s_Hist[pos] = 0;

    __syncthreads ();
    // All threads finished clearing out the
    // histogram
```

Generate partial histogram

```
for (int pos = globalTid;
     pos < dataN;
     pos += numThreads){

  uint data4 = d_Data[pos]; // coalesced
  // shared memory is word interleaved
  // read four pixels per thread with a load word

  atomicAdd (s_Hist + ((data4 >>  0) & 0xFFU),1);
  atomicAdd (s_Hist + ((data4 >>  8) & 0xFFU),1);
  atomicAdd (s_Hist + ((data4 >> 16) & 0xFFU),1);
  atomicAdd (s_Hist + ((data4 >> 24) & 0xFFU),1);
  // we are not using atomicInc which has a more
  // complex structure that atomicAdd
}

__syncthreads();
```

Merge partial histogram with global histogram

```
for (int pos = threadIdx.x;
     pos < BIN_COUNT;
     pos += blockDim.x){

  atomicAdd(d_Result + pos, s_Hist[pos]);
  // these operate on global memory
}
```

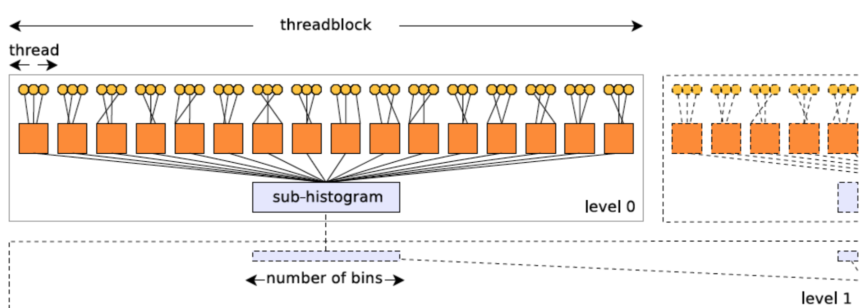
Code overview

```
__global__ void histogram256Kernel (uint *d_Result, uint *d_Data,
   int dataN){
   const int   globalTid = blockIdx.x * blockDim.x + threadIdx.x;
   const int   numThreads = blockDim.x * gridDim.x;
   __shared__ uint s_Hist[BIN_COUNT];
   for (int pos = threadIdx.x; pos < BIN_COUNT; pos += blockDim.x)
                 s_Hist[pos] = 0;
   __syncthreads ();
   for (int pos = globalTid; pos < dataN; pos += numThreads){
       uint data4 = d_Data[pos]; // coalesced
       atomicAdd (s_Hist + (data4 >>  0) & 0xFFU, 1);
       atomicAdd (s_Hist + (data4 >>  8) & 0xFFU, 1);
       atomicAdd (s_Hist + (data4 >> 16) & 0xFFU, 1);
       atomicAdd (s_Hist + (data4 >> 24) & 0xFFU, 1);
   }
   __syncthreads();
   for (int pos = threadIdx.x; pos < BIN_COUNT; pos += blockDim.x)
       atomicAdd(d_Result + pos, s_Hist[pos]);
}
```

Discussion

- s_Hist updates
  - Conflicts in shared memory
  - Data Dependent
  - 32-way conflicts possible and likely
- Is there an alternative?
  - One histogram per thread?
    - Not enough shared memory (if shared memory < 32 * Hist_Size)
    - Low occupancy (only 1 warp/block possible)
  - Load a tile of data in shared memory (or constant?)
    - Each thread produces a portion of the s_Hist that maps onto the same bank
    - All threads read entire data tile

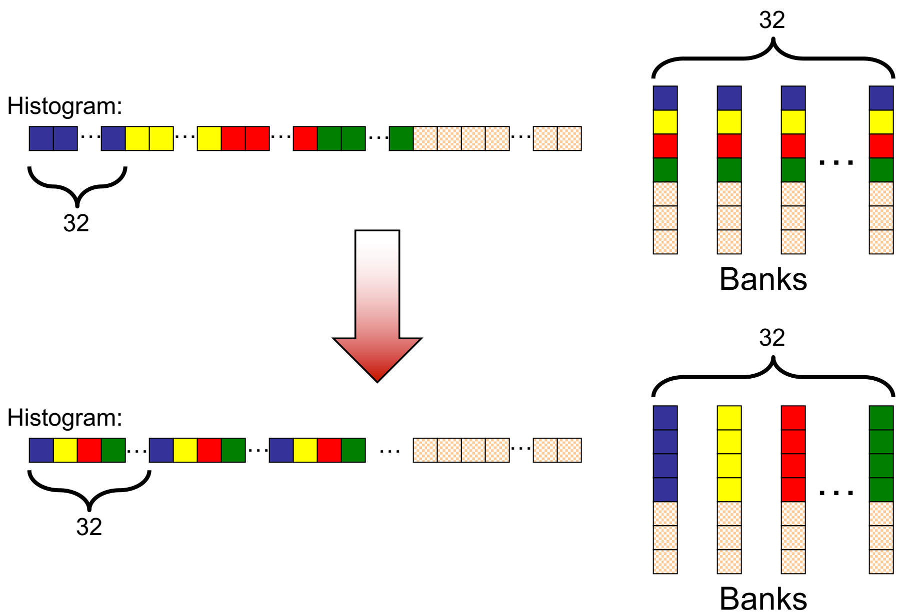
Parallel Strategy #3



- Input array tile in shared memory
  - Every thread reads entire tile (in consecutive #threads * 4 bytes chunks)
  - Wrap-around: in cycle i, where 0 ≤ i ≤ 31, read tile[(i+threadID) % 32]
- Histogram array in shared memory
  - One column per possible pixel value
- One histogram per threadblock
  - Each thread responsible for a part of the sub-histogram
    - Only the bins that map to the same bank. Divergence?

Map it to the same bank

Same color: sub-histogram bins for same thread



This leads us to the Histogram Lab

- How many blocks?
- How many threads per block?
- How much shared memory used per thread?
  - One histogram per thread
    - ✓ Conflict/race free for partial histograms
    - ✗ But too few threads and blocks → Low occupancy
  - One histogram per threadblock
    - ✓ Higher occupancy
    - ✗ But conflicts in updates → atomics, bank conflicts
  - One histogram per threadblock, interleave among warp threads
    - ✓ Privatize portion of histogram per thread → no conflicts
    - ✗ But must read the input 32 times, (some) thread divergence
- Atomics or privatization? Conflicts or divergence? Low occupancy or conflicts? Read the input once or 32 times?
  - What is the right balance for histogram computation?
  - What other optimizations can you think of?