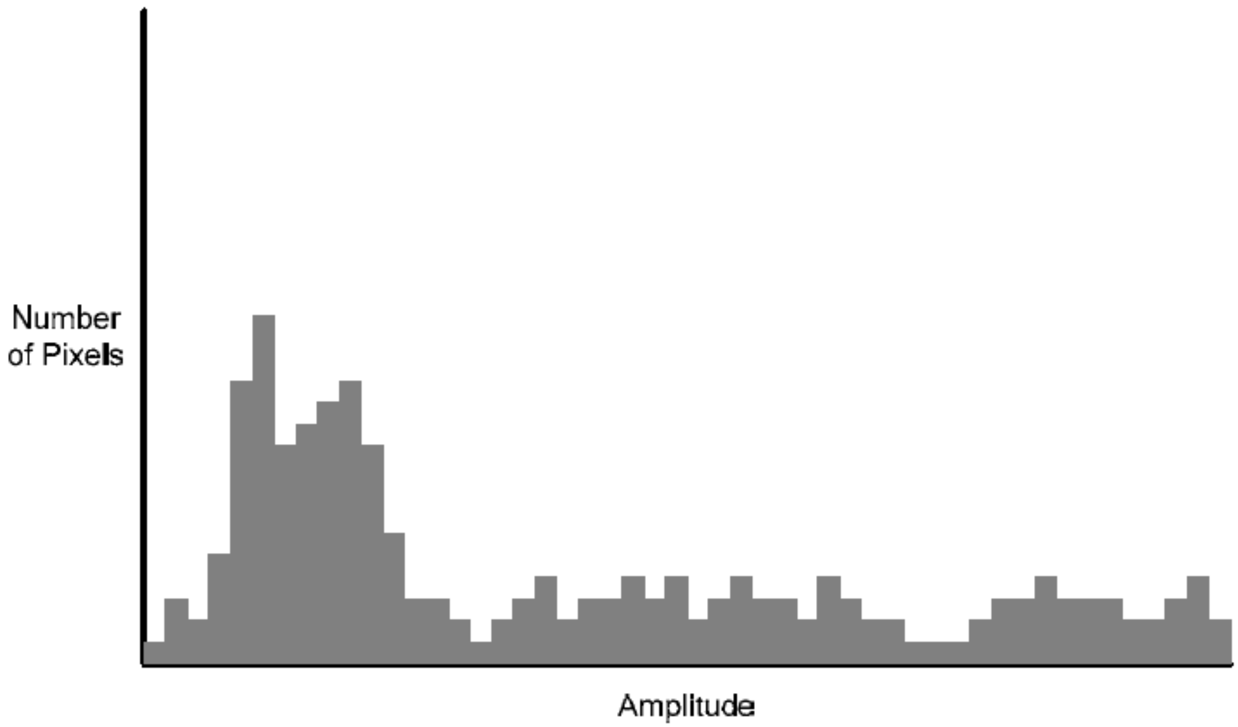
# Advanced Computer Architecture

**Histogram - GPU**

## Histogram

- Given an image (or data array) 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]]++;
```

## Parallel Strategy

- Distribute work across multiple blocks
  - Divide input data to blocks
- Each block process a portion of the data
  - Two Options:
    - Multiple threads: one thread per data value
    - Multiple elements per thread
      - Two variations: arrangement of threads: stride & contiguous
  - Both options produce a partial histogram
    - Could produce multiple histograms
- Merge all partial histograms
  - Produces the final histogram

## Number of Elements: One Thread per Element

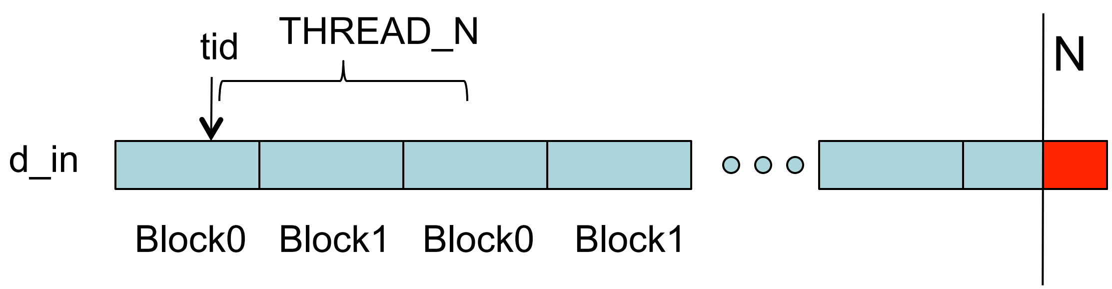
- How to deal with a generic number of elements?
  - Maximum number of blocks = 65536
  - Max number of threads per block = 512
- Limit N to 33Million
  - However, not an effective way to combine results
  - Each thread needs to update global histogram array

## Multiple Elements Per Thread: Stride

- General kernel format: adding elements (reduce)

```
#define N   1024
__global__ void kernel( int *d_in, int *d_out, int N) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int sum = 0;
    int THREAD_N = blockDim.x * gridDim.x;
    for (int opt=tid; opt < N; opt = opt + THREAD_N) {
        sum = sum + d_in[opt];
    }
    d_out[tid] = sum;
}
kernel <<< 2, 32>>> (d_in, d_out, N);
                      // Launching 2 blocks of 32 threads
                      // d_in has 1024 elements
                      // d_out has 64 elements to reduce partial sums
                      // 64 total threads, each thread processes 16 items
```

N prevents threads from going beyond data size

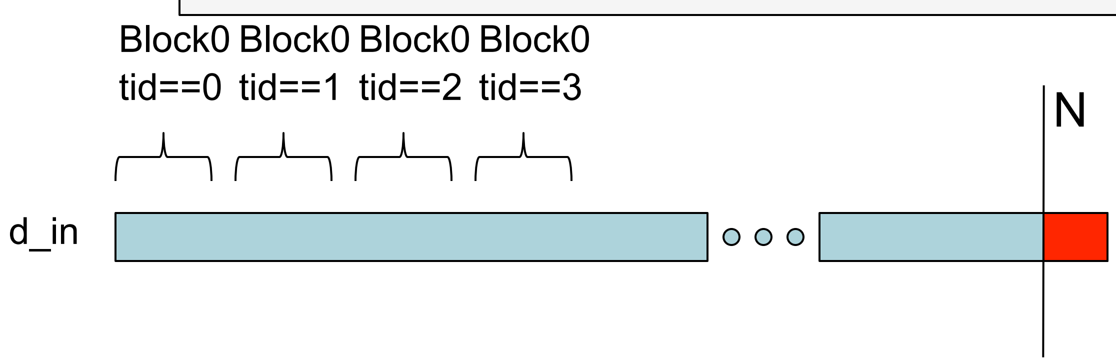



THREAD_N = the number of blocks times the block size, represents the offset to the next item to be processed

## Multiple Elements Per Thread: Contiguous

- General kernel format: adding elements (reduce)

```
#define N   1024
__global__ void kernel( int *d_in, int *d_out, int N) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int sum = 0;
    int THREAD_N = blockDim.x * gridDim.x;
    for (int opt=tid*THREAD_N; opt < N/(THREAD_N); opt = opt + 1) {
        sum = sum + d_in[opt];
    }
    d_out[tid] = sum;
}
kernel <<< 2, 32>>> (d_in, d_out, N);
                      // Launching 2 blocks of 32 threads
                      // d_in has 1024 elements
                      // d_out has 64 elements to reduce partial sums
                      // 64 total threads, each thread processes 16 items
```




Each thread handles a contiguous region of the input data array

## Solving Histogram (Without Atomics): Sub-Histograms

- How many sub-histograms can we fit in shared memory?
  - Input value range: 0-63
  - Each histogram needs 64 entries

- If 32 threads per block, each block would require
  - 32*64 entries of shared memory

## Algorithm Overview

- Kernel1 “Generate Partial Histograms”
  - Initialize partial histogram of each thread
  - Each thread set their own:
    - s_Hist[0…63] = 0
  - Update histogram
  - Each thread:
    - Write out s_Hist to partial histogram in global
  - Summary each thread generates 1 partial histogram
- Kernel2 “Merge”: 1 block per each histogram value will perform a reduction to global histogram
  - Each thread: Update global histogram (each block controls one histogram entry, but can be carried out with multiple threads)
    - Repeat across all partial histograms
      - Read partial histogram[index]
      - Update global histogram

# Generate Kernel

```
#define HISTOGRAM64_THREADBLOCK_SIZE 32
#define HISTOGRAM64_BIN_COUNT 64
__global__ void histogram64Kernel(uint *d_PartialHistograms, data_t *d_Data, uint
dataCount){
    __shared__ int s_Hist[HISTOGRAM64_THREADBLOCK_SIZE * HISTOGRAM64_BIN_COUNT];
    int tid = threadIdx.x + blockIdx.x * blockDim.x;

    for(int i = 0; i < HISTOGRAM64_BIN_COUNT; i++)
       s_Hist[threadIdx.x* HISTOGRAM64_THREADBLOCK_SIZE +i] = 0;

    int THREAD_N = blockDim.x * gridDim.x;
    for (int pos=tid; pos < dataCount; pos = pos + THREAD_N) {
        int data = d_Data[pos];
        s_Hist[data+theadIdx.x*HISTOGRAM64_BIN_COUNT] += 1;
    }
    __syncthreads();

   for(int i = 0; i < HISTOGRAM64_BIN_COUNT; i++)
      d_PartialHistograms[tid*HISTOGRAM64_BIN_COUNT + i] =
                      s_Hist[threadIdx.x*HISTOGRAM64_THREADBLOCK_SIZE +i];
}
```

histogram64Kernel<<<histogramCount, HISTOGRAM64_THREADBLOCK_SIZE>>>

// histogramCount calculated by dataCount / HISTOGRAM64_THREADBLOCK_SIZE
// d_PartialHistograms- allocate histogramCount * HISTOGRAM64_BIN_COUNT

# Merge Kernel

```
#define MERGE_THREADBLOCK_SIZE 32
#define HISTOGRAM64_BIN_COUNT 64
__global__ void mergeHistogram64Kernel(
    uint *d_Histogram,
    uint *d_PartialHistograms,
    uint histogramCount)
{
    __shared__ uint data[MERGE_THREADBLOCK_SIZE];

    uint sum = 0;
    for(uint i = threadIdx.x; i < histogramCount; i += MERGE_THREADBLOCK_SIZE)
        sum += d_PartialHistograms[blockIdx.x + i * HISTOGRAM64_BIN_COUNT];
    data[threadIdx.x] = sum;

    for(uint stride = MERGE_THREADBLOCK_SIZE / 2; stride > 0; stride >>= 1){
        __syncthreads();
        if(threadIdx.x < stride)
            data[threadIdx.x] += data[threadIdx.x + stride];
    }

    if(threadIdx.x == 0)
        d_Histogram[blockIdx.x] = data[0];
}
```

histogramCount – number of partial histograms

Reduce threads of each block to 1

Example: blockIdx.x == 0, sums all partial histograms positions 0
d_PartialHistograms[0], d_PartialHistograms[64], d_PartialHistograms[128]
all correspond to histogram position for value “0”

# Merge Kernel - launch

```
#define MERGE_THREADBLOCK_SIZE 32
#define HISTOGRAM64_BIN_COUNT 64
__global__ void mergeHistogram64Kernel(
    uint *d_Histogram,
    uint *d_PartialHistograms,
    uint histogramCount)
{
    __shared__ uint data[MERGE_THREADBLOCK_SIZE];

    uint sum = 0;
    for(uint i = threadIdx.x; i < histogramCount; i += MERGE_THREADBLOCK_SIZE)
        sum += d_PartialHistograms[blockIdx.x + i * HISTOGRAM64_BIN_COUNT];
    data[threadIdx.x] = sum;

    for(uint stride = MERGE_THREADBLOCK_SIZE / 2; stride > 0; stride >>= 1){
        __syncthreads();
        if(threadIdx.x < stride)
            data[threadIdx.x] += data[threadIdx.x + stride];
    }

    if(threadIdx.x == 0)
        d_Histogram[blockIdx.x] = data[0];
}
```

```
mergeHistogram64Kernel<<<HISTOGRAM64_BIN_COUNT, MERGE_THREADBLOCK_SIZE>>>(
    d_Histogram,     d_PartialHistograms,   histogramCount
  );
```

## 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

## atomicAdd,

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

- Replace second kernel with atomicAdd in the first kernel (add to single global position for each value)