Parallel Patterns: Parallel Histogram Computation with Atomic Operations

Prof. Seokin Hong

Objective

- To understand atomic operations
  - Read-modify-write in parallel computation
  - Use of atomic operations in CUDA

- Parallel Histogram Computation as an example application of atomic operations

Atomic Operations

Race Condition

Example: Mem[x]+=1

thread1: Old $\leftarrow$ Mem[x]
New $\leftarrow$ Old + 1
Mem[x] $\leftarrow$ New

thread2: Old $\leftarrow$ Mem[x]
New $\leftarrow$ Old + 1
Mem[x] $\leftarrow$ New

- If Mem[x] was initially 0,
  - What would the value of Mem[x] be after threads 1 and 2 have completed?
  - What does each thread get in their Old variable?

- The answer may vary due to data races.

Race Condition(cont'd)

Timing Scenario #1

Time	Thread 1	Thread 2
1	(0) Old ← Mem[x]	
2	(1) New ← Old + 1	
3	(1) Mem[x] ← New	
4		(1) Old ← Mem[x]
5		(2) New ← Old + 1
6		(2) Mem[x] ← New

- Thread 1 Old = 0
- Thread 2 Old = 1
- Mem[x] = 2 after the sequence

Race Condition(cont'd)

Timing Scenario #2

Time	Thread 1	Thread 2
1		(0) Old ← Mem[x]
2		(1) New ← Old + 1
3		(1) Mem[x] ← New
4	(1) Old ← Mem[x]	
5	(2) New ← Old + 1	
6	(2) Mem[x] ← New	

- Thread 1 Old = 1
- Thread 2 Old = 0
- Mem[x] = 2 after the sequence

Race Condition(cont'd)

Timing Scenario #3

Time	Thread 1	Thread 2
1	(0) Old ← Mem[x]	
2	(1) New ← Old + 1	
3		(0) Old ← Mem[x]
4	(1) Mem[x] ← New	
5		(1) New ← Old + 1
6		(1) Mem[x] ← New

- Thread 1 Old = 0
- Thread 2 Old = 0
- Mem[x] = 1 after the sequence

Race Condition(cont'd)

Timing Scenario #4

Time	Thread 1	Thread 2
1		(0) Old ← Mem[x]
2		(1) New ← Old + 1
3	(0) Old ← Mem[x]	
4		(1) Mem[x] ← New
5	(1) New ← Old + 1	
6	(1) Mem[x] ← New	

- Thread 1 Old = 0
- Thread 2 Old = 0
- Mem[x] = 1 after the sequence

Race Condition(cont'd)

- In the parallel execution,
  - Execution Sequence is not defined by the programming model,
  - It can be arbitrary

- Then, what is the solution?

- **Atomic operations !**
  - CUDA provides **atomic operations** to deal with this problem

Atomic Operations

- To Ensure Good Outcomes

thread1: Old $\leftarrow$ Mem[x]
New $\leftarrow$ Old + 1
Mem[x] $\leftarrow$ New

thread2: Old $\leftarrow$ Mem[x]
New $\leftarrow$ Old + 1
Mem[x] $\leftarrow$ New

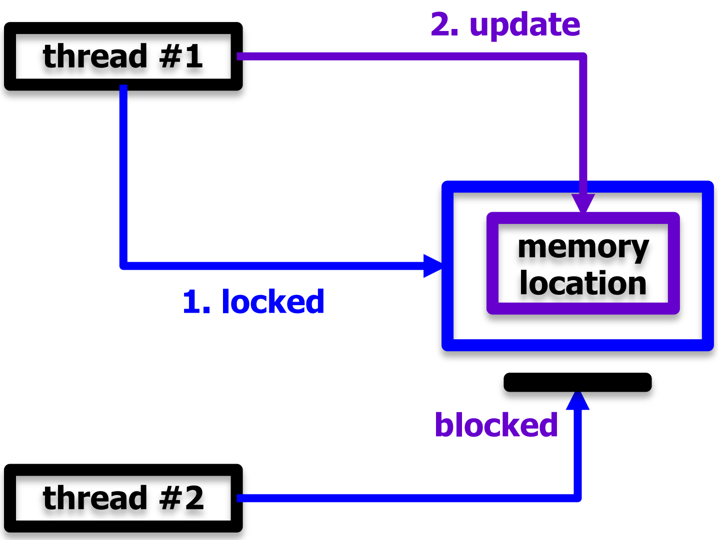
Or

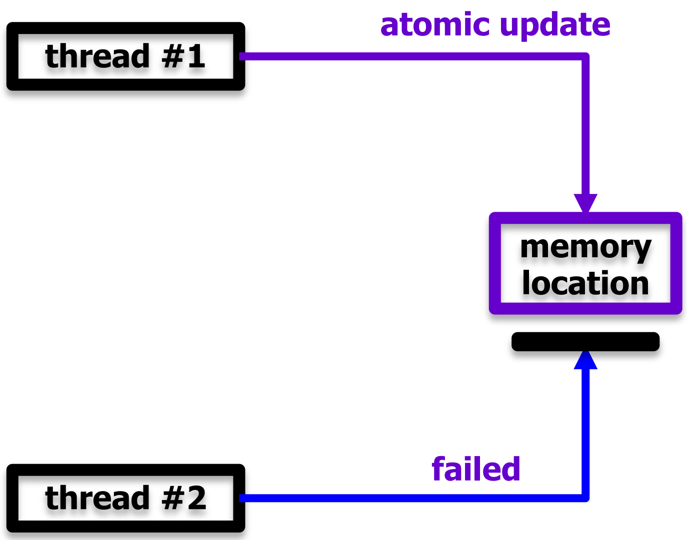
thread2: Old $\leftarrow$ Mem[x]
New $\leftarrow$ Old + 1
Mem[x] $\leftarrow$ New

thread1: Old $\leftarrow$ Mem[x]
New $\leftarrow$ Old + 1
Mem[x] $\leftarrow$ New

Atomic Operations (cont'd)

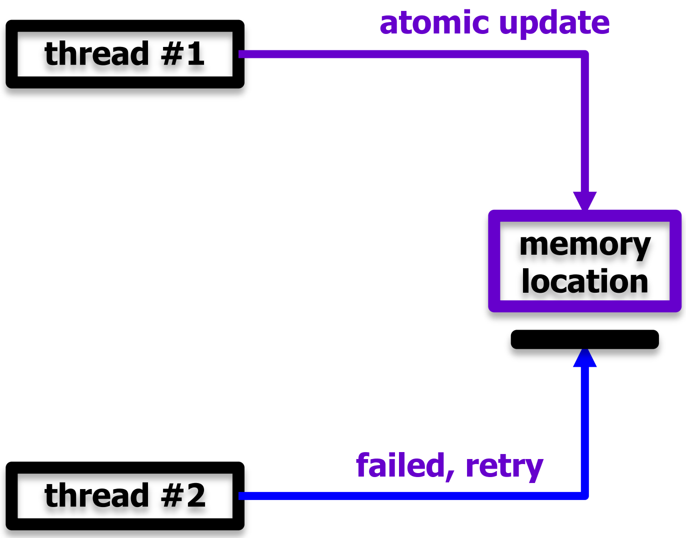
- memory locking
- atomic operation



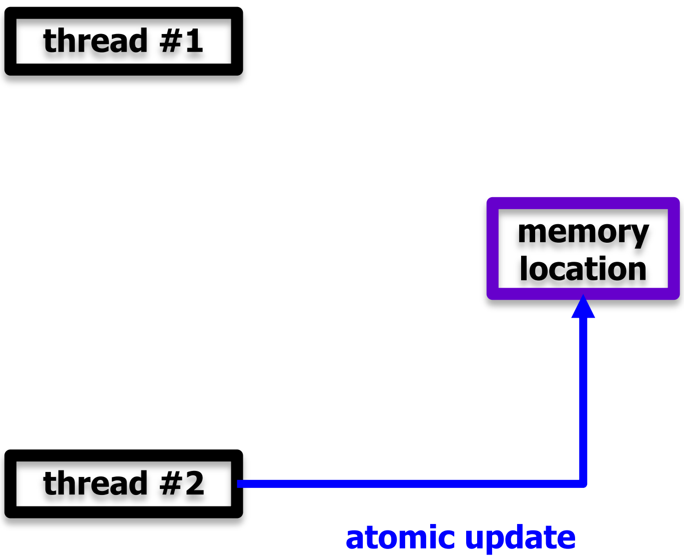


Atomic Operations

- atomic operation #1



- atomic operation #2



Atomic Operations

- **The operation is atomic.**
  - It can not be divided into sub-steps. It is **a single step.**

- The name **atomic** comes from the fact that it is uninterruptable
  - An atomic operation guarantees that only a single thread has access to a piece of memory until an operation completes.

- Different types of atomic instructions
  - Add, Sub, Exch, Min, Max, Inc, Dec, **CAS,** And, Or, Xor,…

CUDA atomic CAS operation

- CAS : compare and swap
  - the most fundamental operation among all atomic operations

- **int atomicCAS(int* address, int expected, int newVal);**
  - step 1. oldVal = read (*address)
  - step 2. (*address) = (oldVal == expected) ? newVal : oldVal
  - step 3. return oldVal
  - do it in an atomic manner !

- How to use it?
  - oldVal = *address;
  - readback = atomicCAS(address, oldVal, newVal);
  - if (oldVal == readback) success!
  - else fail… (another thread did another CAS)

atomicCAS example

```
__device__ inline void MyAtomicAdd(float *address, float value) {
   int oldval, newval, readback;
   oldval = __float_as_int(*address);
   newval = __float_as_int(__int_as_float(oldval) + value);
   while ((readback=atomicCAS((int *)address, oldval, newval)) != oldval) {
      oldval = readback;
      newval = __float_as_int(__int_as_float(oldval) + value);
   }
}
```

CUDA Atomic Functions

- int atomicAdd(int* address, int val);
- int atomicSub(int* address, int val);
- int atomicExch(int* address, int val);
- int atomicMin(int* address, int val);
- int atomicMax(int* address, int val);
- unsigned int atomicInc(unsigned int* address, unsigned int val);
- unsigned int atomicDec(unsigned int* address, unsigned int val);
- int atomicCAS(int* address, int compare, int val);
- int atomicAnd(int* address, int val);
- int atomicOr(int* address, int val);
- int atomicXor(int* address, int val);

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

Example: Count

- make many threads
- set count = 0
- for each thread, count = count + 1
- at the end, show the count value

Count : non-atomic version

```
#include <stdio.h>
#include <stdlib.h>

#define GRIDSIZE       (32 * 1024)
#define BLOCKSIZE       1024
#define TOTALSIZE       (GRIDSIZE * BLOCKSIZE)

__global__ void kernel(unsigned long long int* pCount) {
    (*pCount) = (*pCount) + 1; // we met race condition !
}

int main(void) {
    unsigned long long int aCount[1];
    // prepare timer
    cudaEvent_t start;
    cudaEvent_t stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
```

Count : non-atomic version

```
// CUDA: allocate device memory
unsigned long long int* pCountDev = NULL;
cudaMalloc((void**)&pCountDev, sizeof(unsigned long long int));
cudaMemset(pCountDev, 0, sizeof(unsigned long long int));


// start timer
cudaEventRecord(start, 0);


// CUDA: launch the kernel
dim3 dimGrid(GRIDSIZE, 1, 1);
dim3 dimBlock(BLOCKSIZE, 1, 1);
kernel<<<dimGrid, dimBlock>>>(pCountDev);


// CUDA: copy from device to host
cudaMemcpy(aCount, pCountDev, sizeof(unsigned long long int),
        cudaMemcpyDeviceToHost);
printf("total number of threads = %llu\n", TOTALSIZE);
printf("count = %llu\n", aCount[0]);
```

Count : non-atomic version

```
    // end timer
    float time;
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&time, start, stop);
    printf("elapsed time = %f msec\n", time);
    cudaEventDestroy(start);
    cudaEventDestroy(stop);

    // CUDA: free the memory
    cudaFree(pCountDev);
}
```

Count : non-atomic version

- execution result:

total number of threads = 33554432 ← 32M threads

count = 6323 ← failure !

elapsed time = 4.196864 msec

Count : Atomic Version

- with a new atomic kernel:

```
__global__ void kernel(unsigned long long int* pCount) {
    atomicAdd(pCount, 1ULL);
}
```

- no change in the main( ).

Compilation and Execution Result

- atomic operations are only supported new CUDA architectures…
- So, NVCC with "–arch sm_61" option:

  nvcc -w -arch sm_61 -o count-atomic count-atomic.cu

- execution result

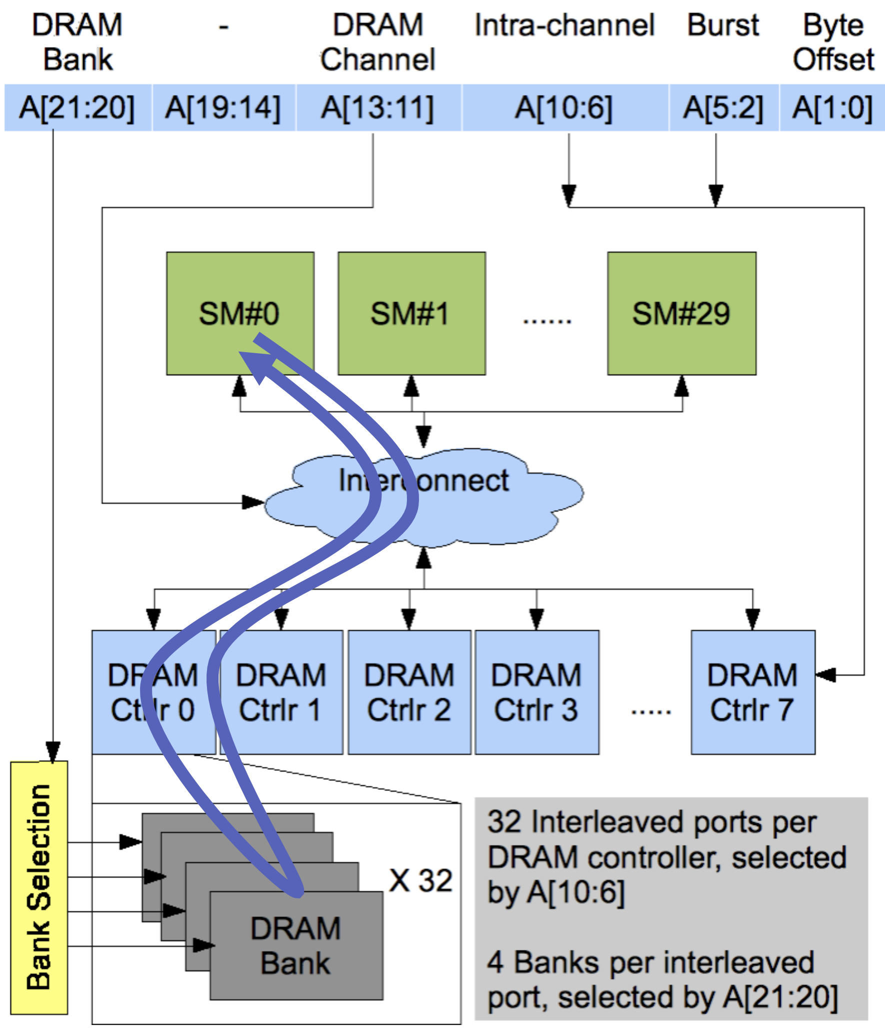
total number of threads = 33554432

count = 33554432

elapsed time = 452.034241 msec

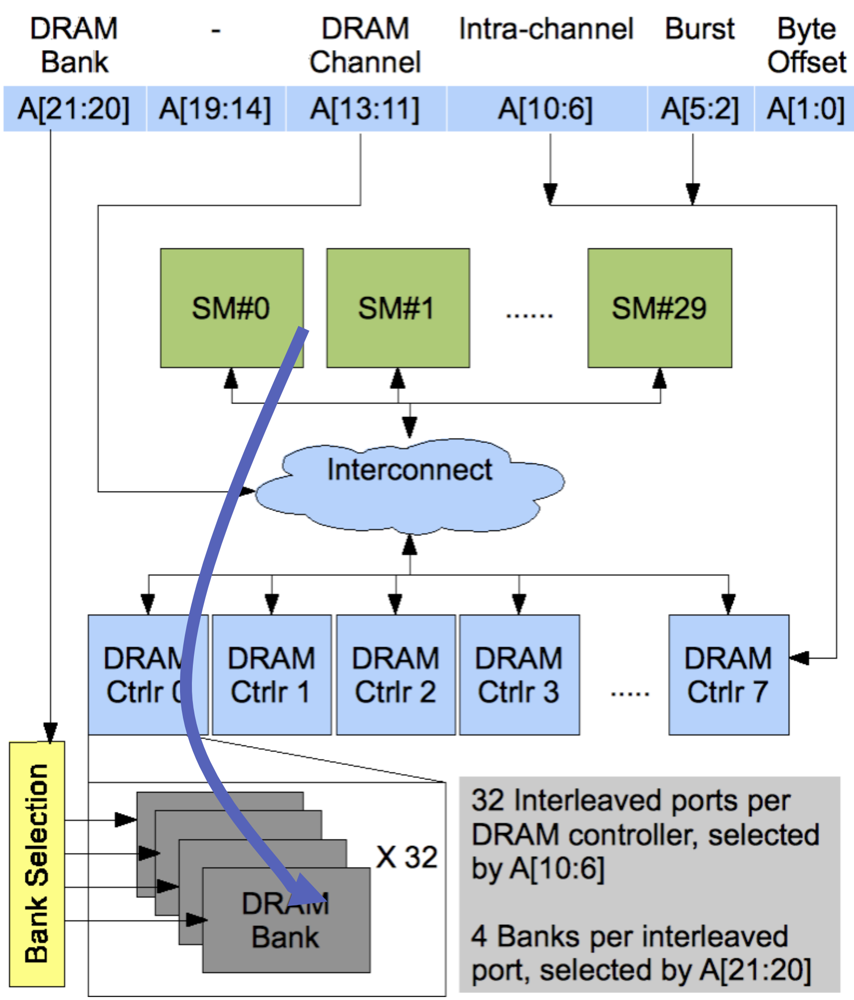
- success !
- but, too slow… → Why?

Atomic Operations on DRAM (Global memory)



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

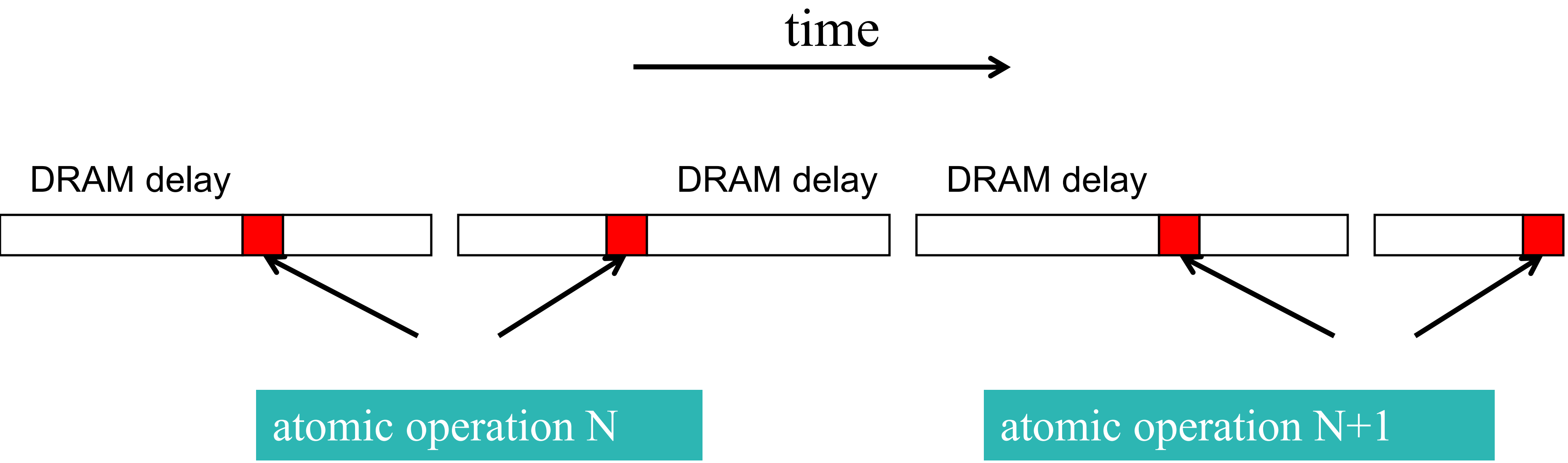
Atomic Operations on DRAM (Global memory)



- 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**

Atomic Operations on DRAM (Global memory)

- Each Load-Modify-Store has two full memory access delays
  - All atomic operations on the same variable (RAM location) are **serialized**



Latency determines throughput of atomic operations

- **Throughput** of an atomic operation is the **rate** at which the application can execute an atomic operation on a particular location.

- **The rate is limited by the total latency** of the read-modify-write sequence, **typically more than 1000 cycles for global memory** (DRAM) locations.

- This means that if many threads attempt to do atomic operation on the same location (contention), **the memory bandwidth is reduced to < 1/1000!**

Hardware Improvements

- Atomic operations on **L2 cache**
  - medium latency, but still serialized

- Atomic operations on **Shared Memory**
  - Very short latency, but still serialized
  - Private to each thread block
  - Need algorithm work by programmers

Count : shared memory version

- new kernel with shared memory

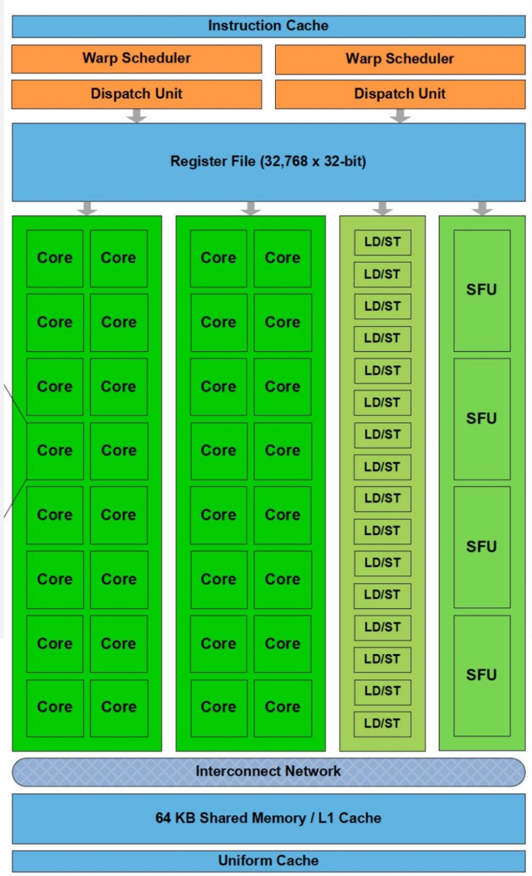
```
__global__ void kernel(int* pCount) {
        __shared__ int nCountShared;

        if (threadIdx.x == 0) {
                nCountShared = 0;
        }
        __syncthreads();

        atomicAdd(&nCountShared, 1);

        __syncthreads();

        if (threadIdx.x == 0) {
                atomicAdd(pCount, nCountShared);
        }
}
```



Execution Result

- execution result

total number of threads = 33554432

count = 33554432

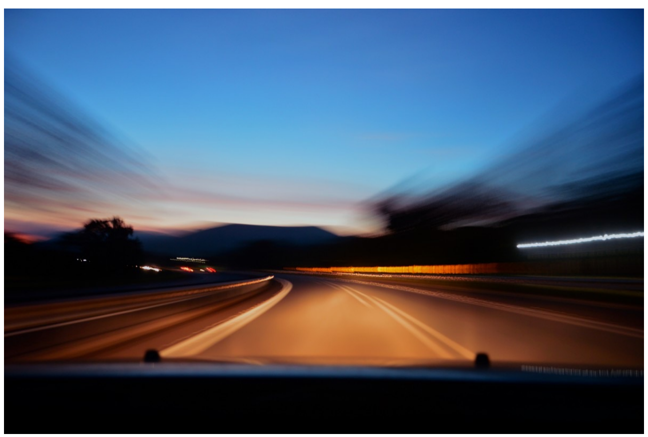
elapsed time = 319.288818 msec

- We achieved some speed ups!

Parallel Histogram Computation

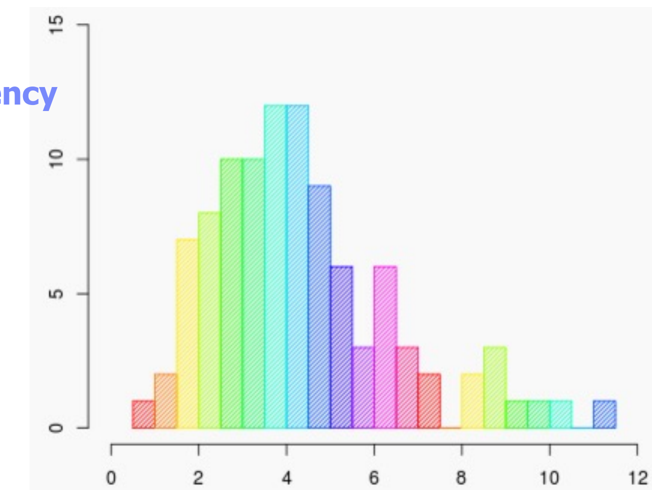
Another Example: Histogram

- histogram analysis
  - a basic tool for image processing



target image

frequency



value

Another Example: Histogram

- in this example,
  - make 1,024 x 1,024 image (integer values)
  - each pixel contains: 0 ~ 14 values randomly
  - make its histogram
    - histogram[0] = number of pixels with value 0
    - histogram[1] = number of pixels with value 1
    - histogram[2] = number of pixels with value 2
    - …

Histogram : host version

```
#include <stdio.h>
#include <stdlib.h>

#define GRIDSIZE        1024
#define BLOCKSIZE       1024
#define TOTALSIZE       (GRIDSIZE * BLOCKSIZE)
#define     NUMHIST             16

void genData(unsigned int* ptr, unsigned int size) {
    while (size--) {
            *ptr++ = (unsigned int)(rand() % (NUMHIST - 1)); // 0 ~ 14
    }
}
void kernel(unsigned int* hist, unsigned int* img, unsigned int size) {
    while (size--) {
            unsigned int pixelVal = *img++;
            hist[pixelVal] = hist[pixelVal] + 1;
    }
}
```

Histogram : host version

```
int main(void) {
    unsigned int* pImage = NULL;
    unsigned int* pHistogram = NULL;
    int i;

    // malloc memories on the host-side
    pImage = (unsigned int*)malloc(TOTALSIZE * sizeof(unsigned int));
    pHistogram = (unsigned int*)malloc(NUMHIST * sizeof(unsigned int));
    for (i = 0; i < NUMHIST; ++i) {
            pHistogram[i] = 0;
    }
    // generate source data
    genData(pImage, TOTALSIZE);
```

Histogram : host version

```
    // perform the action
    kernel(pHistogram, pImage, TOTALSIZE);
    …
    // print the histogram
    long total = 0L;
    for (i = 0; i < NUMHIST; ++i) {
            printf("%2d: %10d\n", i, pHistogram[i]);
            total += pHistogram[i];
    }
    printf("total: %10ld (should be %ld)\n", total, TOTALSIZE);
    …
}
```

Execution Result:

elapsed time = 4339.358362 usec

```
 0:    69692
 1:    69634
 2:    70121
 3:    70277
 4:    70215
 5:    69479
 6:    69988
 7:    70344
 8:    69984
 9:    69976
10:    70099
11:    69686
12:    69810
13:    69909
14:    69362
15:        0
total:  1048576 (should be 1048576)
```

Histogram : GPU version

```
#include <stdio.h>
#include <stdlib.h>

#define GRIDSIZE      1024
#define BLOCKSIZE     1024
#define TOTALSIZE     (GRIDSIZE * BLOCKSIZE)
#define     NUMHIST              16

void genData(unsigned int* ptr, unsigned int size) {
    while (size--) {
            *ptr++ = (unsigned int)(rand() % (NUMHIST - 1));
    }
}
__global__ void kernel(unsigned int* hist, unsigned int* img, unsigned int size) {
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int pixelVal = img[i];
    atomicAdd(&(hist[pixelVal]), 1);
}
```

Histogram : GPU version

```
int main(void) {
    unsigned int* pImage = NULL;
    unsigned int* pHistogram = NULL;
    int i;
    // prepare timer
    cudaEvent_t start;
    cudaEvent_t stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    // malloc memories on the host-side
    pImage = (unsigned int*)malloc(TOTALSIZE * sizeof(unsigned int));
    pHistogram = (unsigned int*)malloc(NUMHIST * sizeof(unsigned int));

    // generate source data
    genData(pImage, TOTALSIZE);
```

Histogram : GPU version

```
// CUDA: allocate device memory
unsigned int* pImageDev;
unsigned int* pHistogramDev;
cudaMalloc((void**)&pImageDev, TOTALSIZE * sizeof(unsigned int));
cudaMalloc((void**)&pHistogramDev, NUMHIST * sizeof(unsigned int));
cudaMemset(pHistogramDev, 0, NUMHIST * sizeof(unsigned int));

// CUDA: copy from host to device
cudaMemcpy(pImageDev, pImage, TOTALSIZE * sizeof(unsigned int), cudaMemcpyHostToDevice);

// start the timer
cudaEventRecord(start, 0);
// perform the action
dim3 dimGrid(GRIDSIZE, 1, 1);
dim3 dimBlock(BLOCKSIZE, 1, 1);
kernel<<<dimGrid, dimBlock>>>(pHistogramDev, pImageDev, TOTALSIZE);
```

Histogram : GPU version

```
// end the timer
float time;
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&time, start, stop);
printf("elapsed time = %f msec\n", time);
cudaEventDestroy(start);
cudaEventDestroy(stop);
// CUDA: copy from device to host
cudaMemcpy(pHistogram, pHistogramDev, NUMHIST * sizeof(unsigned int),
cudaMemcpyDeviceToHost);
```

Histogram : GPU version

```
    // print the histogram
    long total = 0L;
    for (i = 0; i < NUMHIST; ++i) {
            printf("%2d: %10d\n", i, pHistogram[i]);
            total += pHistogram[i];
    }
    printf("total: %10ld (should be %ld)\n", total, TOTALSIZE);
    // CUDA: free the memory
    cudaFree(pImageDev);
    cudaFree(pHistogramDev);
    // free the memory
    free(pImage);
    free(pHistogram);
}
```

Execution Result

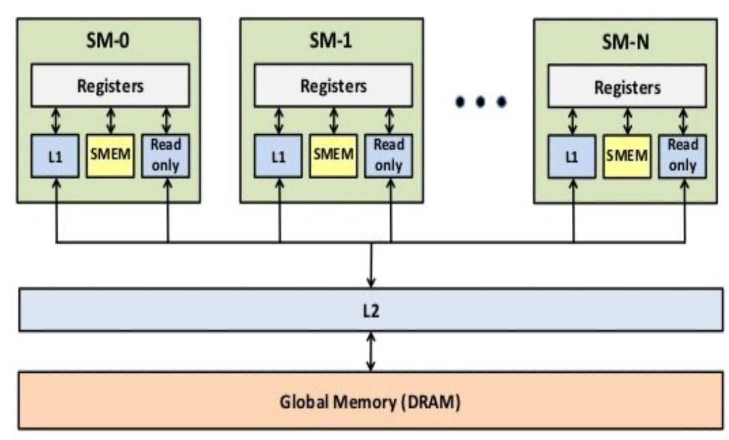
elapsed time = 1.612192 msec

```
 0:    69692
 1:    69634
 2:    70121
 3:    70277
 4:    70215
 5:    69479
 6:    69988
 7:    70344
 8:    69984
 9:    69976
10:    70099
11:    69686
12:    69810
13:    69909
14:    69362
15:        0
total:  1048576 (should be 1048576)
```

Histogram : shared memory

```
__global__ void kernel(unsigned int* hist, unsigned int* img, unsigned int size) {
    __shared__ int histShared[NUMHIST];
    if (threadIdx.x < NUMHIST) {
            histShared[threadIdx.x] = 0;
    }
    __syncthreads();
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int pixelVal = img[i];
    atomicAdd(&(histShared[pixelVal]), 1);
    __syncthreads();

    if (threadIdx.x < NUMHIST) {
            atomicAdd(&(hist[threadIdx.x]), histShared[threadIdx.x]);
    }
}
```



Next?

- **Parallel Patterns: Convolution**