Thread Blocks and Global Comm. and Atomic Operations

Tony Tian(Ph.d)

CSCD439/539 GPU Computing

Blocks must be independent

- Any possible interleaving of blocks should be valid
  - presumed to run to completion without pre-emption
  - can run in any order
  - can run concurrently OR sequentially
- Blocks may coordinate but not synchronize
  - Blocks1 do not know when Block2 completes.
  - shared queue pointer: OK
- Independence requirement gives scalability
  - More data generates more blocks.

Example: Shuffling Data

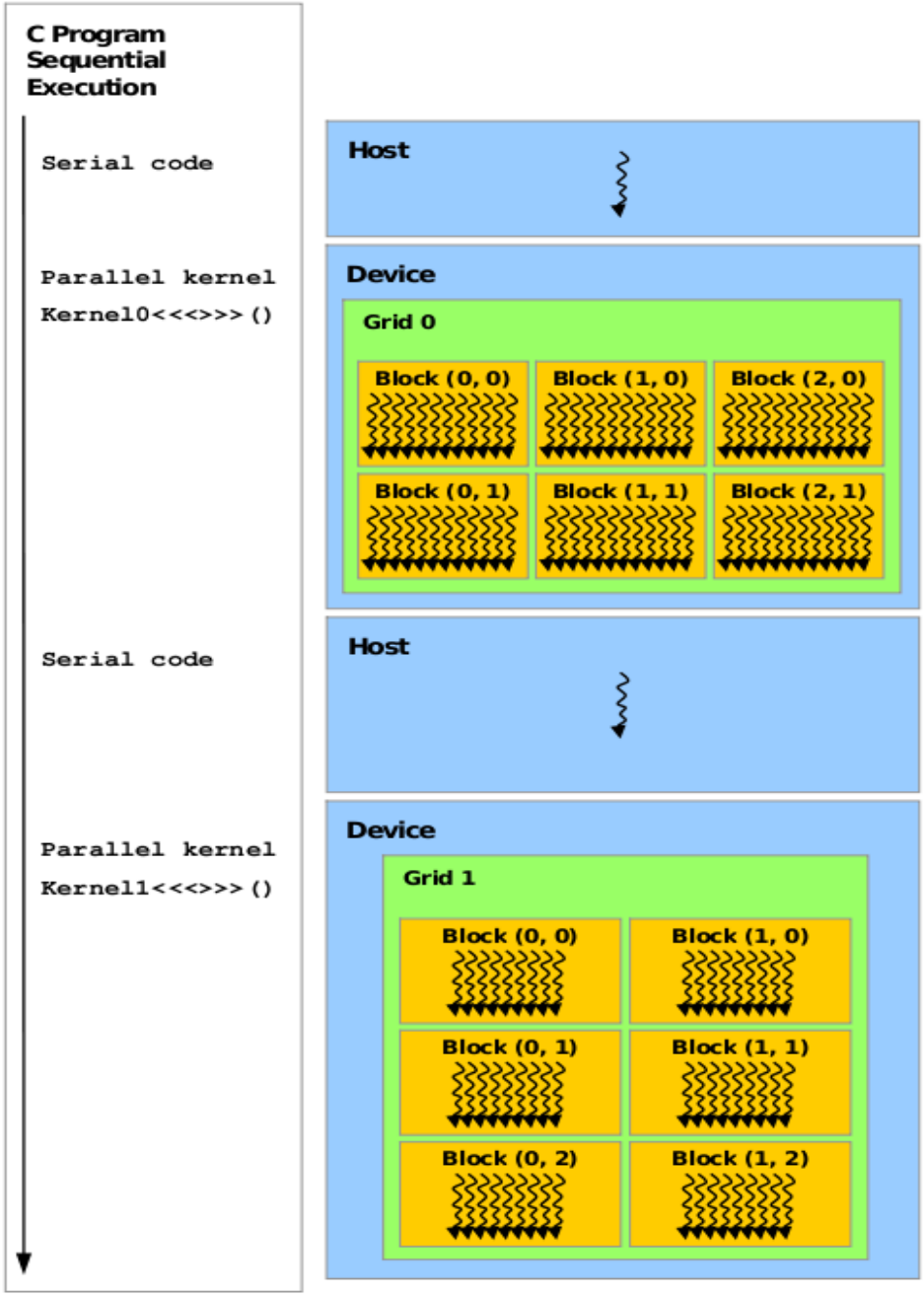
```
// Reorder values based on keys
// Each thread moves one element
__global__ void shuffle(int* prev_array, int*
   new_array, int* indices)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    new_array[i] = prev_array[indices[i]];
}

int main()
{
    // Run grid of N/256 blocks of 256 threads each
    // Assume N is devisable by 256 here.
    shuffle<<< N/256, 256>>>(d_old, d_new, d_ind);
}
```

Host Code

The Problem

- How do you do global communication?
  - E.g. Given a large 2D matrix A, we like to set all elements in A that are below the average to zero.
  - How to compute the average? Then pass the average number to each thread block?
- Solution
  - Finish a grid and start a new one



Serial code executes on the host while parallel code executes on the device.

Global Communication

- Finish a kernel and start a new one
- All writes from all threads complete before a kernel finishes

```
step1<<<grid1,blk1>>>(...);
//step1 brings back the average
// The system ensures that all
// writes from step1 complete.
step2<<<grid2,blk2>>>(...);
//step2 set the elements below
average.
```

Global Communication

- Need to decompose kernels into parts

Race Conditions

- For the average, min or max problem, we have to write to a predefined memory location.
  - Race condition! Updates can be lost.

Race Conditions

```
threadId:0
// vector[0] was equal to 0
vector[0] += 5;
...
a = vector[0];
```

```
threadId:1917

vector[0] += 1;
...
a = vector[0];
```

- What is the value of `a` in thread 0?
- What is the value of `a` in thread 1917?

Race Conditions

- Thread 0 could have finished execution before 1917 started
- Or the other way around
- Or both are executing at the same time

Race Conditions

- Answer: not defined by the programming model, can be arbitrary

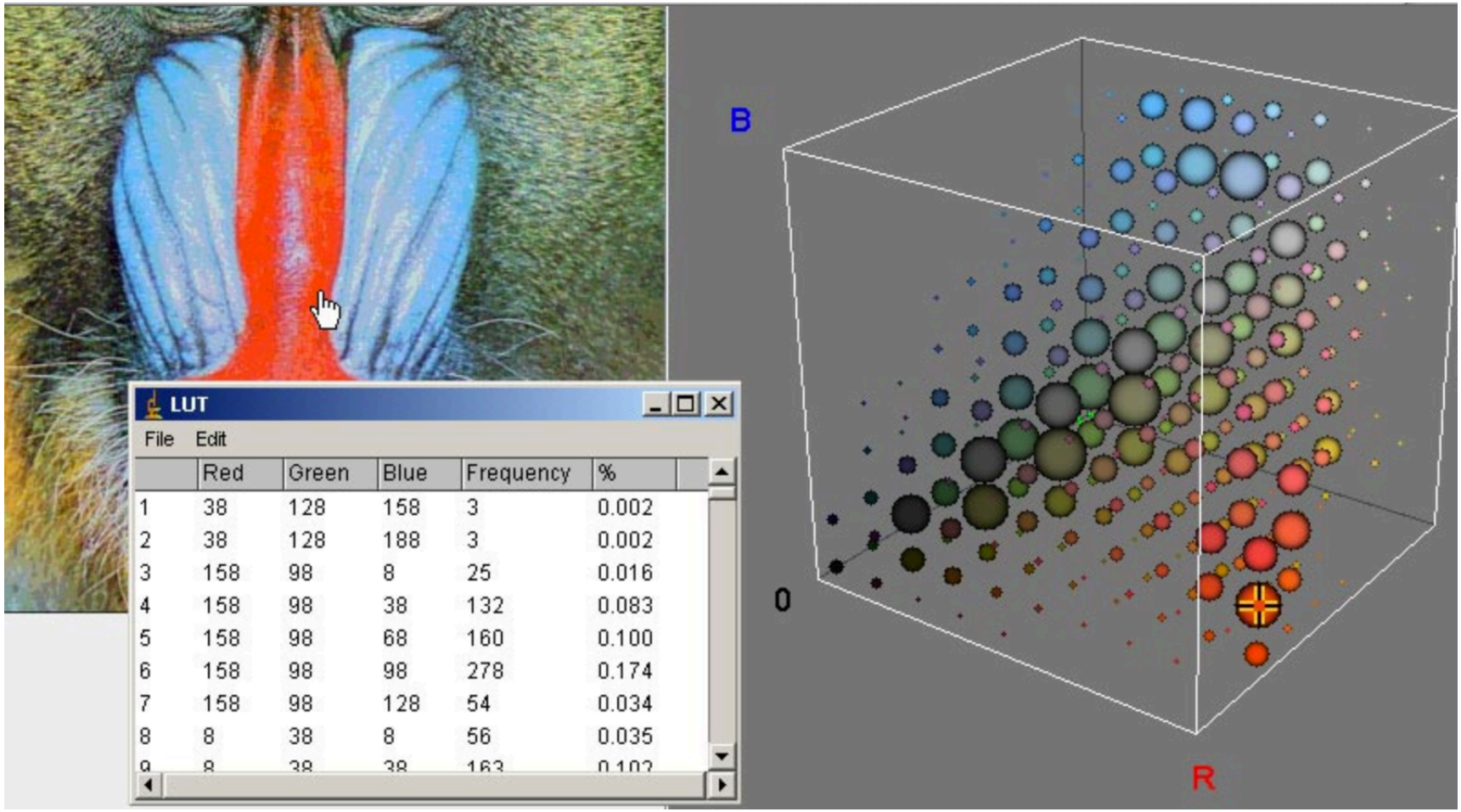
Atomics

- CUDA provides atomic operations to deal with this problem

Atomics

- An atomic operation guarantees that only one single thread has access to a piece of memory at the same time.
- The name atomic comes from the fact that it is uninterruptable.
- No update lost, but ordering is still arbitrary
- Different types of atomic instructions
- `atomic{Add, Sub, Exch, Min, Max, Inc, Dec, CAS, And, Or, Xor}`
- More types in fermi and kepler

Example: Histogram



	Red	Green	Blue	Frequency	%
1	38	128	158	3	0.002
2	38	128	188	3	0.002
3	158	98	8	25	0.016
4	158	98	38	132	0.083
5	158	98	68	160	0.100
6	158	98	98	278	0.174
7	158	98	128	54	0.034
8	8	38	8	56	0.035
9	8	38	38	163	0.102

Example: Histogram

```
// Determine frequency of colors in a picture
// colors have already been converted into ints
// Each thread looks at one pixel and increments
// a counter atomically, here we only consider one color
   channel or the picture is a gray scale image.
__global__ void histogram(int* color,
                          int* buckets)
{
  int i = threadIdx.x
        + blockDim.x * blockIdx.x;
  int c = colors[i];
  atomicAdd(&buckets[c], 1);
}
```

Example: Workqueue

```
// For algorithms where the amount of work per item
// is highly non-uniform, it often makes sense for
// to continuously grab work from a queue
__global__
void workq(int* work_q, int* q_counter,
           int* output, int queue_max)
{
  int i = threadIdx.x
        + blockDim.x * blockIdx.x;
  int q_index =
    atomicInc(q_counter, queue_max);
  int result = do_work(work_q[q_index]);
  output[i] = result;
}
   // each thread gets an unique q_index to process. Why?
```

The API and description found at:
http://docs.nvidia.com/cuda/cuda-c-programming-guide/#atomicinc

Atomics

- Atomics are slower than normal load/store
- You can have the whole machine queuing on a single location in memory
- Atomics unavailable on very early GPUs like G80!

Example: Global Min/Max (Naive)

```
// If you require the maximum across all threads
// in a grid, you could do it with a single global
// maximum value, but it will be VERY slow
__global__
void global_max(int* values, int* gl_max)
{
  int i = threadIdx.x
        + blockDim.x * blockIdx.x;
  int val = values[i];
  atomicMax(gl_max,val);
}
//http://docs.nvidia.com/cuda/cuda-c-programming-
  guide/#atomicmax
```

Example: Global Min/Max (Better)

```
// introduce intermediate maximum results, so that
// most threads do not try to update the global max
__global__
void global_max(int* values, int* max,
                int *reg_maxes,
                int num_regions)
{
  // i and val as before …
  int region = i % num_regions;
  if(atomicMax(&reg_maxes[region],val) < val)
  {
    atomicMax(max,val);
  }
}
```

1, Most of time, threads only update to a regional max, reducing the lock contention.

2, if a val is less than the current regional max, we do NOT need update the global max.

Global Min/Max

- Single value causes serial bottleneck
- Create hierarchy of values for more parallelism
- Performance will still be slow, so use judiciously
- See next few lectures for even better version!

Summary

- Can't use normal load/store for inter-thread communication because of race conditions

- Use atomic instructions for **sparse** and/or unpredictable global communication
  - See next few lectures for shared memory and scan for other communication patterns

- Decompose data (very limited use of single global sum/max/min/etc.) for more parallelism