CME 213

Lecture 21: Atomics & Segmented Scan

David Tarjan, NVIDIA Research

Overview

- Atomics
- Segmented Scan

ATOMICS

The Problem

- How do you do global communication?
- Finish a grid and start a new one
- Scan
- Atomics

Race Conditions

- Coordinate by writing to a predefined memory location
  - Race condition! Updates can be lost

Race Conditions

```
threadId:0                              threadId:1917
      // vector[0] was equal to 0
vector[0] += 5;                         vector[0] += 1;
...                                     ...
a = vector[0];                          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 a single thread has access to a piece of memory while an operation completes
- The name atomic comes from the fact that it is uninterruptable
- No dropped data, 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

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
__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;
}
```

Atomics

- Atomics are slower than normal load/store
- You can have the whole machine queuing on a single location in memory
- Atomics unavailable on 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);
}
```

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* gl_max,
                int *reg_max,
                int num_regions)
{
  // i and val as before …
  int region = i % num_regions;
  if(atomicMax(&reg_max[region],val) < val)
  {
    atomicMax(gl_max,val);
  }
}
```

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 lecture 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
  - Scan is good for dense communication pattern and where ordering is needed

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

SEGMENTED SCAN

Segmented Scan

- What it is:
  - Scan + Barriers/Flags associated with certain positions in the input arrays
  - Operations don't propagate beyond barriers
- Do many scans at once, no matter their size

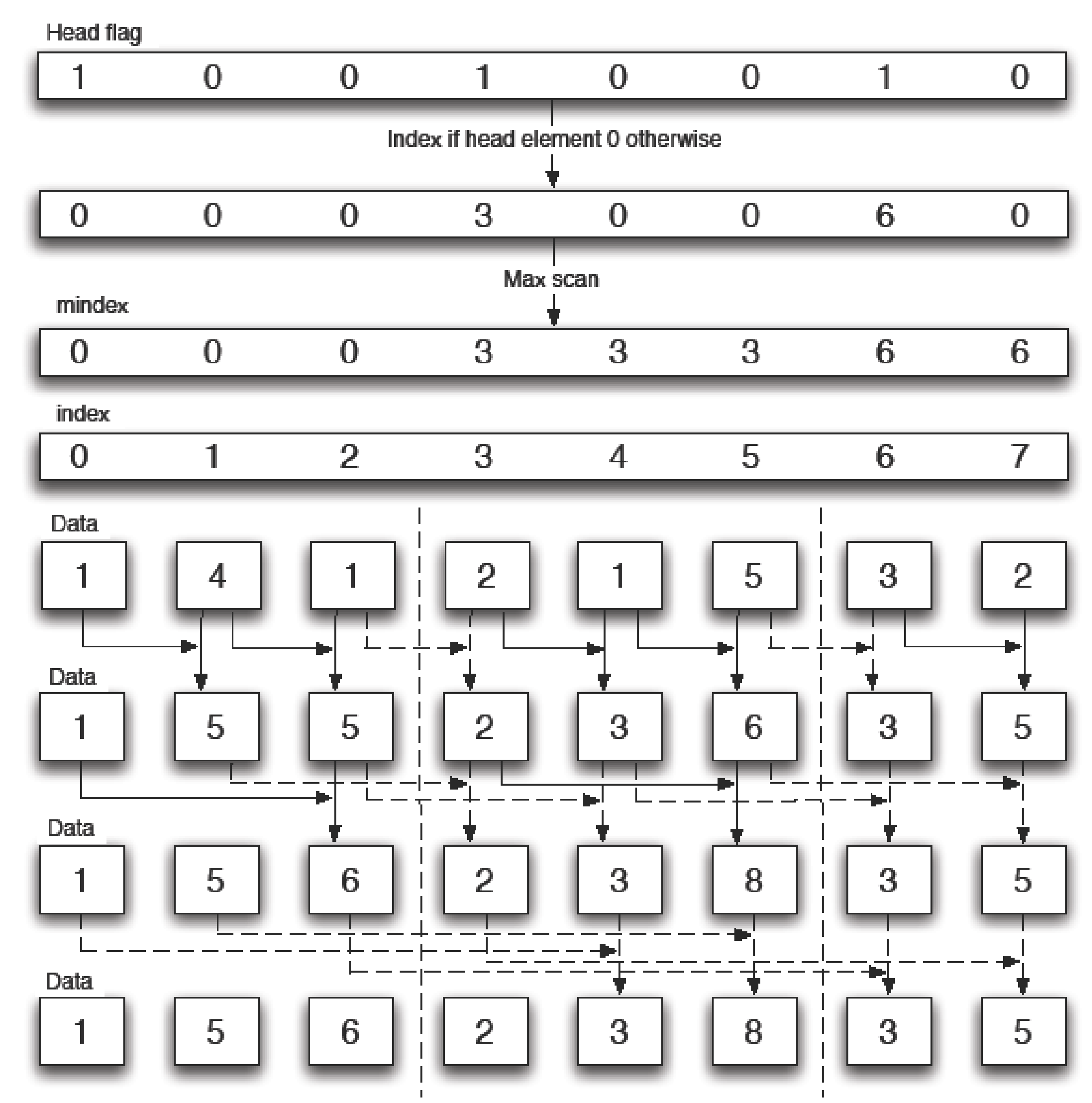

Image taken from "Efficient parallel scan algorithms for GPUs" by S. Sengupta, M. Harris, and M. Garland

Segmented Scan

```
__global__ void segscan(int * data,
   int * flags)
{
    __shared__ int s_data[BL_SIZE];
    __shared__ int s_flags[BL_SIZE];
    int idx = threadIdx.x + blockDim.x
   * blockIdx.x;
   // copy block of data into shared
   // memory
   s_data[idx] = …; s_flags[idx] = …;
   __syncthreads();
```

Segmented Scan

```
   …
   // choose whether to propagate
   s_data[idx] = s_flags[idx] ?
      s_data[idx] :
      s_data[idx - 1] + s_data[idx];

   // create merged flag
   s_flags[idx] =
      s_flags[idx - 1] | s_flags[idx];
   // repeat for different strides
}
```

Segmented Scan

- Doing lots of reductions of unpredictable size at the same time is the most common use

- Think of doing sums/max/count/any over arbitrary sub-domains of your data

Segmented Scan

- Common Usage Scenarios:
  - Determine which region/tree/group/object class an element belongs to and assign that as its new ID
  - Sort based on that ID
  - Operate on all of the regions/trees/groups/objects in parallel, no matter what their size or number

Segmented Scan

- Also useful for implementing divide-and-conquer type algorithms
  - Quicksort and similar algorithms

Questions?

Backup Slides

Example Segmented Scan

```
int data[10]   = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1};
int flags[10]  = {0, 0, 0, 1, 0, 1, 1, 0, 0, 0};

int step1[10] = {1, 2, 1, 1, 1, 1, 1, 2, 1, 2};
int flg1[10]   = {0, 0, 0, 1, 0, 1, 1, 1, 0, 0};

int step2[10] = {1, 2, 1, 1, 1, 1, 1, 2, 1, 2};
int flg2[10]   = {0, 0, 0, 1, 0, 1, 1, 1, 0, 0};
…
```

Example Segmented Scan

```
int step2[10] = {1, 2, 1, 1, 1, 1, 1, 2, 1, 2};
int flg2[10]    = {0, 0, 0, 1, 0, 1, 1, 1, 0, 0};
...

int result[10] = {1, 2, 3, 1, 2, 1, 1, 2, 3, 4};
```