CME213/ME339
Lecture 16

Eric Darve
Erich Elsen
Austin Gibbons

Department of Mechanical Engineering
Institute for Computational and Mathematical Engineering
Stanford University

Spring 2012

Overview

- Scan / Prefix Sum Overview
- Uses - Stream Compaction / Multiple Outputs / Duplicate Deletion / Splitting / Counting Sort!
- CUDA Implementation of scan - warp and block level.
- Stable vs unstable sorting
- Radix Sort with Scan
- Brief Overview of merge sort

Thrust remove_if

- Up until now we have mainly considered `transform` type algorithms
- But there are many problems that cannot be solved using only this primitive
- How do we deal with algorithms that have a *variable* number of outputs?

```
vals: 1 2 3 4 5 6 7 8 9
pred: 1 0 1 0 0 1 0 1 0

out:  1 3 6 8
```

- This is known as stream compaction
- A special case of the more general multiple outputs
- The number of outputs is either 0 or 1

Stream Compaction with Scan

We take `pred` and (exclusive) scan it; add blues to get red:

```
pred: 1 0 1 0 0 1 0 1 0
scan: 0
```

- An exclusive scan starts with the identity element
- At pos n, exclusive scan contains $\sum_{0}^{n-1}$
- An inclusive scan starts with the first value in the array
- At pos n, inclusive scan contains $\sum_{0}^{n}$
- The operator can be any associative operator, not just addition

Stream Compaction with Scan

We take `pred` and (exclusive) scan it; add blues to get red:

```
pred: 1 0 1 0 0 1 0 1 0
scan: 0 1
```

Stream Compaction with Scan

We take `pred` and (exclusive) scan it; add blues to get red:

```
pred: 1 0 1 0 0 1 0 1 0
scan: 0 1 1
```

Stream Compaction with Scan

We take `pred` and (exclusive) scan it; add blues to get red:

```
pred: 1 0 1 0 0 1 0 1 0
scan: 0 1 1 2
```

Stream Compaction with Scan

We take `pred` and (exclusive) scan it; add blues to get red:

```
pred: 1 0 1 0 0 1 0 1 0
scan: 0 1 1 2 2
```

Stream Compaction with Scan

We take `pred` and (exclusive) scan it; add blues to get red:

```
pred: 1 0 1 0 0 1 0 1 0
scan: 0 1 1 2 2 2
```

Stream Compaction with Scan

We take `pred` and (exclusive) scan it; add blues to get red:

```
pred: 1 0 1 0 0 1 0 1 0
scan: 0 1 1 2 2 2 3
```

Stream Compaction with Scan

We take `pred` and (exclusive) scan it; add blues to get red:

```
pred: 1 0 1 0 0 1 0 1 0
scan: 0 1 1 2 2 2 3 3
```

Stream Compaction with Scan

We take `pred` and (exclusive) scan it; add blues to get red:

```
pred: 1 0 1 0 0 1 0 1 0
scan: 0 1 1 2 2 2 3 3 4
```

Stream Compaction with Scan

Now we perform a `scatter_if`

```
1  if(pred[tid])
2      out[scan[tid]] = vals[tid];
```

```
vals: 1 2 3 4 5 6 7 8 9
scan: 0 1 1 2 2 2 3 3 4
pred: 1 0 1 0 0 1 0 1 0

out: 1 3 6 8
```

Other Applications of Scan

- Scan lets us coordinate between processors / threads *without* locks or any kind of direct coordination
- This lets us solve a wide variety of problems

Removing Duplicates

- Mark the first element of each run (known as head flags)
- Can use either `adjacent_difference` or a `transform` with a suitable functor
- Follow with stream compaction

```
vals: 1 1 1 2 2 3 1 1
pred: 1 0 0 1 0 1 1 0
```

Splitting

Separate vector into two parts (say even and odd elements)

```
in:  2 5 6 4 9 1 8

out: 2 6 4 8 5 9 1
```

- For each even element, we need to figure out how many odd elements come before it
- For each odd element, we need to figure out how many even elements come after it
- Two scans!

Splitting

First scan the odd elements, so each even element knows how many odd elements are before it

```
in:   2 5 6 4 9 1 8
scan: 0 0 1 1 1 2 3
```

Next do a *reverse* scan on the even elements, so each odd element knows how many even elements come after it

Scan starts with number of even elements = $numTotal - numOdd$, numOdd was determined by the first scan

```
in:   2 5 6 4 9 1 8
scan: 4 3 3 2 1 1 1
```

Splitting

Move even elements left for each odd element before it

```
1  finalpos[i] = pos[i] - oddscan[i];
```

Move odd elements right for each even element after it

```
1  finalpos[i] = pos[i] + evenscan[i];
```

```
in:          2 5 6 4 9 1 8
odd-scan:    0 0 1 1 1 2 3
even-scan:   4 3 3 2 1 1 1
pos:         0 1 2 3 4 5 6
final pos:   0 4 1 2 5 6 3
```

Splitting

Perform `scatter` with `in` and `finalpos`

```
1  out[finalpos[i]] = in[i];
```

```
in:          2 5 6 4 9 1 8
final pos:   0 4 1 2 5 6 3
out:         2 6 4 8 5 9 1
```

- In binary splitting can be used for sorting
- We can optimize down to only one scan instead of two

Implementing Scan

Serial Approach

```
1  for (int i = 1; i < N; ++i)
2      vals[i] += vals[i - 1];
```

- Time Complexity: O(N)
- Work Complexity: O(N)
- Work Complexity = Time Complexity $\times$ Work Per Step
- For serial algorithms, Work Per Step is always 1
- Therefore serial algorithms always have equal Time and Work Complexity

Implementing Scan

Parallel Approach 1

```
      0  4  2   6   3   2   0   1

0:    0  4  6   8   9   5   2   1  val[i] += val[i-1];
1:    0  4  6  12  15  13  11   6  val[i] += val[i-2];
2:    0  4  6  12  15  17  17  18  val[i] += val[i-4];
```

Properties

- Time complexity: O(log N)
- Work complexity: O(N log N)
- A parallel algorithm with the same work complexity as the serial algorithm is called Work Efficient
- This algorithm is NOT Work Efficient
- Each step requires GLOBAL synchronization

Implementing Scan

Multi-Scan

- Up-sweep - Reduce each group
- Scan the group totals
- Down-sweep - Scan each group starting from scan value

```
                0 2 3 0 1 3 0 2  0  0  2  3  1  0  2  3
UpSweep:          5       6        5           6
Scan:             0       5       11          16
DownSweep: 0 0 2 5 5 6 9 9 11 11 11 13 16 17 17 19
```

- Can use more than two levels
- Global Synchronization only needed during middle scan, with a much smaller number of elements

Scan within Warp

```
1  volatile __shared__ float smem[32];
2
3  #pragma unroll
4  for (int i = 0; i < 5; ++i) {
5      int offset = 1 << i;
6      if (lane >= offset)
7          smem[lane] += smem[lane - offset];
8  }
```

- Note the lack of `syncthreads()`, known as *warp-synchronous* programming
- All threads in a warp are guaranteed to be synchronized
- The `volatile` is necessary for correctness
- Otherwise the compiler will optimize out loads of values that are changed by other threads

Better Warp Scan

If we make the array slightly larger and fill it with identity values, then we can get rid of the `if` statement

```
1  volatile __shared__ float smem_[48];
2  smem_[lane] = 0;
3  volatile float *smem = smem_ + 16 + lane;
4
5  #pragma unroll
6  for (int i = 0; i < 5; ++i) {
7      int offset = 1 << i;
8      smem[0] += smem[-offset];
9  }
```

Best Warp Scan

The previous scan, reads twice and writes once to shared memory each loop. We can eliminate one read.

```
1  volatile __shared__ float smem_[48];
2  smem_[lane] = 0;
3  volatile float *smem = smem_ + 16 + lane;
4
5  float sum = smem[0];
6  #pragma unroll
7  for (int i = 0; i < 5; ++i) {
8      int offset = 1 << i;
9      sum += smem[-offset];
10     smem[0] = sum;
11 }
```

Examining Disassembly

An often useful technique to confirm that these optimizations have indeed made a difference is to examine the actual machine code produced by the compiler.

We can do this as follows:

```
nvcc -o scan.cubin scan.cu --cubin -arch=sm_20
cuobjdump -sass scan.cubin > scan.isa
```

Scan.cu will be posted for you to examine / compile / play with

Block Scan

- We will follow the up-sweep, scan, down-sweep pattern.
- Each warp will compute the sum of all elements (with a scan)
- These totals will be scanned
- The totals will be pushed back down and added to the earlier scan

```
1  template<int numWarps, logNumWarps>
2  __global__ void BlockScan(int *in, int *out) {
3    volatile __shared__ int smem_[numWarps][48];
4    const int tid = threadIdx.x;
5    const int warp = tid / 32;
6    const int lane = tid % 32;
7
8    volatile int* smem = smem_[warp] + lane;
9    smem[0] = 0;
10   smem += 16;
11 }
```

Block Scan

```
1   int sum = in[tid];
2   smem[0] = sum;
3
4   WarpScan(smem);
5   __syncthreads();
6
7   volatile __shared__ int warpTotals_[numWarps * 2];
8   if (tid < numWarps) {
9     int tot = smem_[tid][47]; //pick off warp totals
10    warpTotals_[tid] = 0;
11    volatile int* warpTotals = warpTotals_ + numWarps;
12    warpTotals[tid] = tot;
13
14    WarpScan(warpTotals);
15
16    warpTotals[tid] -= tot; //exclusive scan
17  }
18
19  __syncthreads();
20  out[tid] = sum + warpTotals[warp];
```

Counting Sort

- The idea is count how many times each number occurs (create a histogram)
- Then scan the histogram
- Finally scatter each value to the correct location

Counting Sort

keys:	relative digit counts:	Digit Histogram:	Scan of Digit Histogram:	Scan plus relative digit count:	Sorted:
3	0			6	0
2	0	0: 2	0: 0	3	0
2	1	1: 1	1: 2	4	1
0	0	2: 3	2: 3	0	2
1	0	3: 2	3: 6	2	2
3	1			7	2
2	2			5	3
0	1			1	3

Properties

- Time complexity of O(n + k) where n is the number of keys and k is biggest key
- Better than n log(n) !! Possible because it is not a comparison sort
- Space complexity of O(n + k)
- Out-of-Place (Input and Output must be different arrays)
- Really good if k is small; really bad if k is large
- How can we handle large keys without requiring an enormous amount of storage?

Stable Sorts

- A sort is stable if it preserves the ordering of two equal elements

1 4 2 8 4 1 5

1 1 2 4 4 5 8

- At first this may seem like a useless property
- But consider when there is payload data that goes along with the sort
- Lexicographical sorting or dictionary sorting

Stable Sorts

keys:		sort by last digit		stable sort by first digit
13		21		09
45		13		13
16		94		16
34		34		21
75	->	75	->	34
94		45		39
09		16		45
39		09		75
21		39		94

Radix Sort

- Counting Sort is stable
- By applying it to successively larger digits we can sort arbitrarily large keys
- Known as the Least-Significant-Digit (LSD) Radix Sort
- O(kn) where k is the number of digits and n is the number of elements
  - There also exists a MSD Radix Sort which sorts in the opposite order
- While keeping the space requirements reasonable (number of digits $\times$ base)
- A common choice is to sort 8 bits per pass

Parallel Radix Sort

or Parallel Counting Sort

- If we can parallelize counting sort then we've parallelized radix sort
- Use a strategy similar to parallelizing scan
- Break into blocks, perform local histograms, push the results up
- Scan these histograms at a higher level, push the result down
- Use the result of the scan as offsets for the local scan resulting in global offsets

Merge Sort

```
9 3 6 1 2 8 6 3

sort separately

1 3 6 9         2 3 6 8

merge

1 2 3 3 6 6 8 9
```

- Paralellizing the sorts is obvious
- Sort each half independently
- How do paralellize the merge?

Parallel Merge

- Find the median of list A in list B
- Merging elements less than the median in both lists is independent of merging elements greater than the median
- The red elements can be merged independently of the blue

A	B	median A	A	B
0	0		0	0
1	2		1	2
1	3		1	3
3	4	3	3	4
5	4		5	4
6	4		6	4
8	7		8	7