CS 193G

Lecture 7: Parallel Patterns II

Overview

- Segmented Scan
- Sort
- Mapreduce
- Kernel Fusion

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

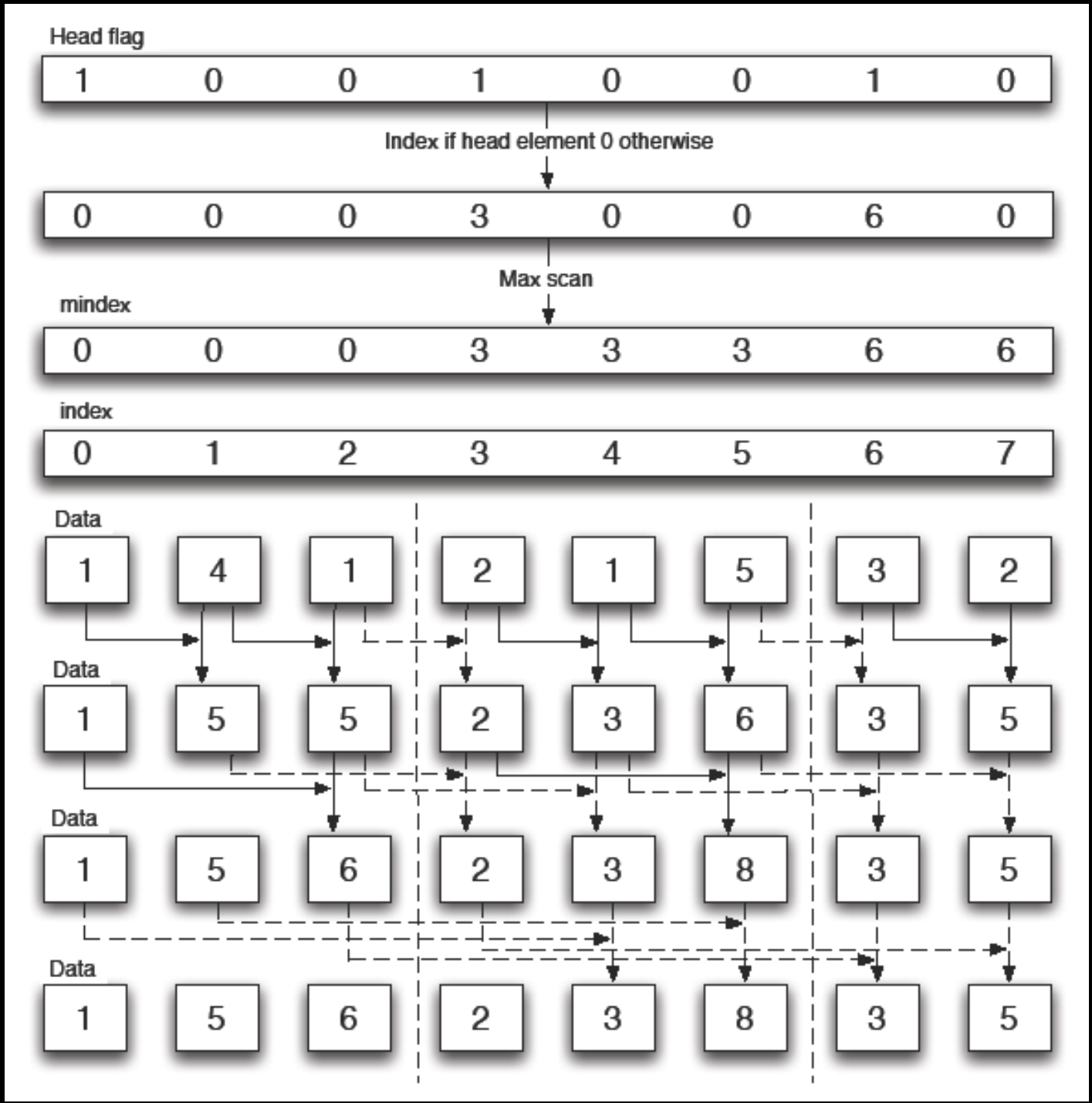

Image taken from "Efficient parallel scan algorithms for GPUs" by S. Sengupta, M. Harris, and M. Garland

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

SORT

Sort

- Useful for almost everything
- Optimized versions for the GPU already exist
- Sorted lists can be processed by segmented scan
- Sort data to restore memory and execution coherence

Sort

- binning and sorting can often be used interchangeably
- Sort is standard, but can be suboptimal
- Binning is usually custom, has to be optimized, can be faster

Sort

- Radixsort is faster than comparison-based sorts
- If you can generate a fixed-size key for the attribute you want to sort on, you get better performance

MAPREDUCE

Mapreduce

- Old concept from functional progamming
- Repopularized by Google as parallel computing pattern
- Combination of sort and reduction (scan)

Mapreduce

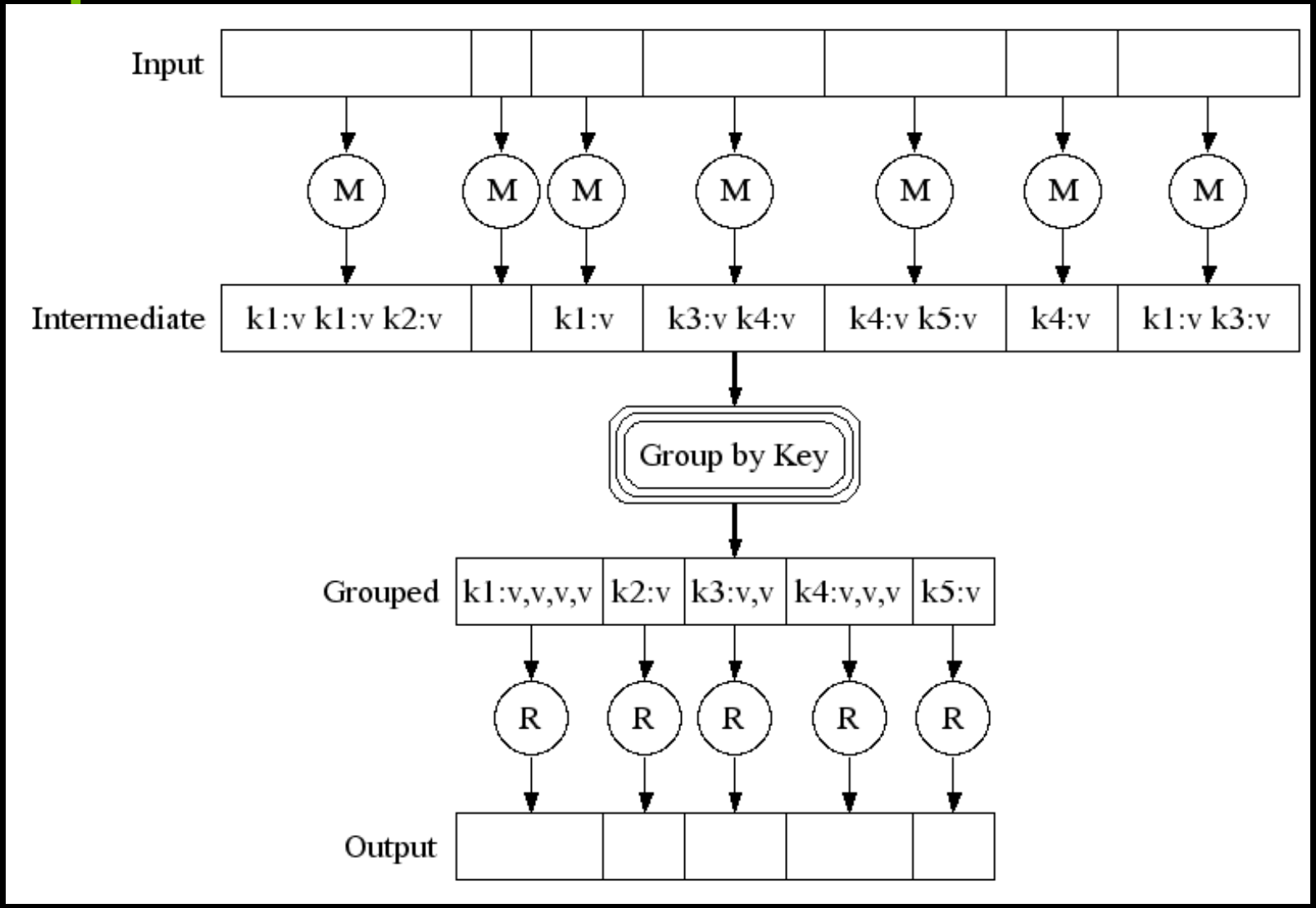

Image taken from Jeff Dean's presentation at http://labs.google.com/papers/mapreduce-osdi04-slides/index-auto-0007.html

Mapreduce: Map Phase

- Map a function over a domain
- Function is provided by the user
- Function can be anything which produces a (key, value) pair
  - Value can just be a pointer to arbitrary datastructure

Mapreduce: Sort Phase

- All the (key,value) pairs are sorted based on their keys
- Happens implicitly
- Creates runs of (k,v) pairs with same key
- User usually has no control over sort function

Mapreduce: Reduce Phase

- Reduce function is provided by the user
  - Can be simple plus, max,…
- Library makes sure that values from one key don’t propagate to another (segscan)
- Final result is a list of keys and final values (or arbitrary datastructures)

KERNEL FUSION

Kernel Fusion

- Combine kernels with simple producer->consumer dataflow
- Combine generic data movement kernel with specific operator function
- Save memory bandwidth by not writing out intermediate results to global memory

Separate Kernels

```
__global__ void is_even(int * in, int
   * out)
{
    int i = …
   out[i] = ((in[i] % 2) == 0) ? 1: 0;
}

__global__ void scan(…)
{
   …
}
```

Fused Kernel

```
__global__ void fused_even_scan(int *
   in, int * out, …)
{
   int i = …
   int flag = ((in[i] % 2) == 0) ? 1:
   0;
   // your scan code here, using the
   flag directly
}
```

Kernel Fusion

- Best when the pattern looks like

```
output[i] = g(f(input[i]));
```

- Any simple one-to-one mapping will work

Fused Kernel

```
template <class F>
__global__ void opt_stencil(float *
   in, float * out, F f)
{ // your 2D stencil code here
   for(i,j)
   {
      partial = f(partial,in[…],i,j);
   }
   float result = partial;
}
```

Fused Kernel

```
class boxfilter
{ private:
   table[3][3];
   boxfilter(float input[3][3])
   public:
   float operator()(float a, float b,
   int i, int j)
   {
      return a + b*table[i][j];
   }
}
```

Fused Kernel

```
class maxfilter
{ public:
   float operator()(float a, float b,
   int i, int j)
   {
      return max(a,b);
   }
}
```

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};
```