GPU Programming

Parallel Patterns

Miaoqing Huang
University of Arkansas
Spring 2016
Outline

Introduction

Reduction

All-Prefix-Sums
  Applications
  Avoiding Bank Conflicts

Segmented Scan

Sorting
Getting out of the trenches

- Focus on low-level details of kernel programming so far
  - Mapping of threads to work
  - Launch grid configuration
  - `__shared__` memory management
  - Resource allocation

- Hard to see the forest for the trees

```c
__global__ void foo(...)
{
    extern __shared__ smem[];
    int i = ???
    .......
}

......

dim3 B = ???
dim3 N = ???
dim3 S = ???
foo<<B,N,S>>>();
```
Getting out of the trenches

- Focus on low-level details of kernel programming so far
  - Mapping of threads to work
  - Launch grid configuration
  - `__shared__` memory management
  - Resource allocation

- Hard to see the forest for the trees

```c
__global__ void foo(...) {
    extern __shared__ smem[];
    int i = ???
    .......
}

......
dim3 B = ???
dim3 N = ???
dim3 S = ???
foo<<<B,N,S>>>();
```

- Now what?
Parallel Patterns

- Think at a higher level than individual CUDA kernels
- Specify **what** to compute, not **how** to compute it
- Let programmer worry about algorithm
  - Defer pattern implementation to someone else
- Common Parallel Computing Scenarios
  - Many parallel threads need to generate a single result →
    - Reduce
  - Many parallel threads need to partition data →
    - Split
  - Many parallel threads produce variable output / thread →
    - Compact / Expand
  - Parallel prefix sum, a.k.a,
    - scan
Primordial CUDA Pattern: Blocking

- Partition data to operate in well-sized blocks
  - Small enough to be staged in shared memory
  - Assign each data partition to a thread block
  - No different from cache blocking!
- Provide several performance benefits
  - Have enough blocks to keep processors busy
  - Working in shared memory cuts memory latency dramatically
  - Likely to have coherent access patterns on load/store to shared memory
- All CUDA kernels are built this way
  - Blocking may not matter for a particular problem, but you’re still forced to think about it
  - Not all kernels require __shared__ memory
  - All kernels do require registers
- All of the parallel patterns we’ll discuss have CUDA implementations that exploit blocking in some fashion
1. Partition data into subsets that fit into shared memory
2. Handle each data subset with one thread block
3. Load the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism
4. Perform the computation on the subset from shared memory
5. Copy the result from shared memory back to global memory
Outline

Introduction

Reduction

All-Prefix-Sums
  Applications
  Avoiding Bank Conflicts

Segmented Scan

Sorting
Reduction

- Reduce vector to a single value
  - Via an associative operator ($+, \times, \min/\max, \text{AND}/\text{OR}, ...$)
  - CPU: sequential implementation
    - for(int $i = 0; i < n; ++i) ...$
  - GPU: “tree”-based implementation
Serial Reduction

- Reduction via serial iteration

```c
float sum(float *data, int n)
{
    float result = 0;
    for(int i = 0; i < n; ++i) {
        result += data[i];
    }

    return result;
}
```
Parallel Reduction
Interleaved

Values (in shared memory)

<table>
<thead>
<tr>
<th>Values</th>
<th>Thread IDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 1 8 -1 0 -2 3 5 -2 -3 2 7 0 11 0 2</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
</tbody>
</table>

Step 1
Stride 1

<table>
<thead>
<tr>
<th>Values</th>
<th>Thread IDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>11 1 7 -1 -2 -2 8 5 -5 -3 9 7 11 11 2 2</td>
<td>0 1 2 3</td>
</tr>
</tbody>
</table>

Step 2
Stride 2

<table>
<thead>
<tr>
<th>Values</th>
<th>Thread IDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>18 1 7 -1 6 -2 8 5 4 -3 9 7 13 11 2 2</td>
<td>0 1</td>
</tr>
</tbody>
</table>

Step 3
Stride 4

<table>
<thead>
<tr>
<th>Values</th>
<th>Thread IDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>24 1 7 -1 6 -2 8 5 17 -3 9 7 13 11 2 2</td>
<td>0</td>
</tr>
</tbody>
</table>

Step 4
Stride 8

<table>
<thead>
<tr>
<th>Values</th>
<th>Thread IDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>41 1 7 -1 6 -2 8 5 17 -3 9 7 13 11 2 2</td>
<td>0</td>
</tr>
</tbody>
</table>
### Parallel Reduction

#### Contiguous

<table>
<thead>
<tr>
<th>Step 1</th>
<th>Thread IDs</th>
<th>Values (in shared memory)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stride 8</td>
<td>0 1 2 3 4 5 6 7</td>
<td>10 1 8 -1 0 -2 3 5 -2 -3 2 7 0 11 0 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step 2</th>
<th>Thread IDs</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stride 4</td>
<td>0 1 2 3</td>
<td>8 -2 10 6 0 9 3 7 -2 -3 2 7 0 11 0 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step 3</th>
<th>Thread IDs</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stride 2</td>
<td>0 1</td>
<td>8 7 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step 4</th>
<th>Thread IDs</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stride 1</td>
<td>0</td>
<td>21 20 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2</td>
</tr>
</tbody>
</table>
# CUDA Reduction – Per-block

```c
__global__ void block_sum(float *input, float *results, size_t n)
{
    extern __shared__ float sdata[];
    int i = blockIdx.x * blockDim.x + threadIdx.x; int tx = threadIdx.x;

    // load input into __shared__ memory
    float x = 0;
    if(tx < n)
        sdata[tx] = input[tx];
    __syncthreads();

    // block-wide reduction in __shared__ mem
    for(int offset = blockDim.x / 2; offset > 0; offset >>= 1) {
        if(tx < offset) {
            // add a partial sum upstream to our own
            sdata[tx] += sdata[tx + offset];
        }
        __syncthreads();
    }

    // finally, thread 0 writes the result
    if(threadIdx.x == 0) {
        // note that the result is per-block, not per-thread
        results[blockIdx.x] = sdata[0];
    }
}
```
CUDA Reduction – Per-block
Improved Version

```c
__global__ void block_sum(float *input, float *results, size_t n) {
    extern __shared__ float sdata[];
    int i = blockIdx.x * blockDim.x + threadIdx.x; int tx = threadIdx.x;

    // load input into __shared__ memory
    float x = 0;
    if(i < n)
        x = input[i];
    sdata[tx] = x;
    __syncthreads();

    // block-wide reduction in __shared__ mem
    for(int offset = blockDim.x / 2; offset > 0; offset >>= 1) {
        if(tx < offset) {
            // add a partial sum upstream to our own
            sdata[tx] += sdata[tx + offset];
        }
        __syncthreads();
    }

    // finally, thread 0 writes the result
    if(threadIdx.x == 0) {
        // note that the result is per-block, not per-thread
        results[blockIdx.x] = sdata[0];
    }
}
```
Barrier Divergence

- Is this barrier divergent?

```c
for(int offset = blockDim.x / 2; offset > 0; offset >>= 1) {
    ...
    __syncthreads();
}
```
Barrier Divergence

- Is this barrier divergent?

```cpp
for(int offset = blockDim.x / 2; offset > 0; offset >>= 1) {
    ...
    __syncthreads();
}
```

- How about this one?

```cpp
__global__ void do_i_halt(int *input)
{
    int i = ......
    if(input[i]) {
        ......
        __syncthreads();
    }
}
```
CUDA Reduction
The first option: launch the kernel twice

```c
// global sum via per-block reductions
float sum(float *d_input, size_t n)
{
    size_t block_size = ...;
    size_t num_blocks = n/block_size + (n%block_size==0)?0:1;

    // allocate per-block partial sums plus a final total sum
    float *d_sums = 0;
    cudaMalloc((void**)&d_sums, sizeof(float) * (num_blocks + 1));

    // reduce per-block partial sums
    int smem_sz = block_size*sizeof(float);
    block_sum<<<num_blocks,block_size,smem_sz>>>(d_input, d_sums, n);
    // reduce partial sums to a total sum
    block_sum<<<1,block_size,smem_sz>>>(d_sums, d_sums + num_blocks, num_blocks);

    // copy result to host
    float result = 0;
    cudaMemcpy(&result, d_sums+num_blocks, ...);
    return result;
}
```
Caveat Reduction

- What happens if there are too many partial sums to fit into \texttt{__shared__} memory in the second stage?
Caveat Reduction

- What happens if there are too many partial sums to fit into `__shared__` memory in the second stage?
- Give each thread more work in the kernel specification
  - Sum is associative & commutative
    - Order does not matter to the result
  - We can schedule the sum any way we want
    - E.g., serial accumulation before block-wide reduction
Caveat Reduction

- What happens if there are too many partial sums to fit into \texttt{__shared__} memory in the second stage?
- Give each thread more work in the kernel specification
  - Sum is associative & commutative
    - Order does not matter to the result
  - We can schedule the sum any way we want
    - E.g., serial accumulation before block-wide reduction
- Or, launch the kernel $\geq 2$ iterations
Parallel Reduction Complexity

- $\log_2 N$ parallel steps, each step $S$ does $\frac{N}{2^S}$ independent operations
Parallel Reduction Complexity

- \( \log_2 N \) parallel steps, each step \( S \) does \( \frac{N}{2^S} \) independent operations
  - Step Complexity: \( O(\log_2 N) \)
Parallel Reduction Complexity

- $\log_2 N$ parallel steps, each step $S$ does $\frac{N}{2^S}$ independent operations
  - Step Complexity: $O(\log_2 N)$

- For $N = 2^D$, performs $\sum_{S=1}^{D} 2^{D-S} = N - 1$ operations
Parallel Reduction Complexity

- $\log_2 N$ parallel steps, each step $S$ does $\frac{N}{2^S}$ independent operations
  - Step Complexity: $O(\log_2 N)$

- For $N = 2^D$, performs $\sum_{S=1}^{D} 2^{D-S} = N - 1$ operations
  - Work Complexity: $O(N)$
  - It is work-efficient, i.e., it does not perform more operations than a sequential algorithm
Parallel Reduction Complexity

- $\log_2 N$ parallel steps, each step $S$ does $\frac{N}{2^S}$ independent operations
  - Step Complexity: $O(\log_2 N)$

- For $N = 2^D$, performs $\sum_{S=1}^{D} 2^{D-S} = N - 1$ operations
  - Work Complexity: $O(N)$
  - It is work-efficient, i.e., it does not perform more operations than a sequential algorithm

- With $P$ threads physically in parallel ($P$ processors)
Parallel Reduction Complexity

- $\log_2 N$ parallel steps, each step $S$ does $\frac{N}{2^S}$ independent operations
  - Step Complexity: $O(\log_2 N)$

- For $N = 2^D$, performs $\sum_{S=1}^{D} 2^{D-S} = N - 1$ operations
  - Work Complexity: $O(N)$
  - It is work-efficient, i.e., it does not perform more operations than a sequential algorithm

- With $P$ threads physically in parallel ($P$ processors)
  - Time complexity: $O(\frac{N}{P})$
  - Compared with $O(N)$ for sequential reduction
The Second Option
Use atomic operator atomicAdd

```c
__global__ void block_sum(float *input, float *results, size_t n)
{
    extern __shared__ float sdata[];
    int i = blockIdx.x * blockDim.x + threadIdx.x; int tx = threadIdx.x;

    // load input into __shared__ memory
    float x = 0;
    if(i < n)
    x = input[i];
    sdata[tx] = x;
    __syncthreads();

    // block-wide reduction in __shared__ mem
    for(int offset = blockDim.x / 2; offset > 0; offset >>= 1) {
        if(tx < offset) {
            // add a partial sum upstream to our own
            sdata[tx] += sdata[tx + offset];
        }
        __syncthreads();
    }

    // finally, thread 0 writes the result
    // NOTE: atomicAdd(float*, float) only work on Fermi above
    if(threadIdx.x == 0) {
        atomicAdd(&results[0], sdata[0]);
    }
}
```
How about this kernel?

```c
__global__ void block_sum(float *input, float *results, size_t n, size_t num_blocks) {
    extern __shared__ float sdata[];
    int i = blockIdx.x * blockDim.x + threadIdx.x; int tx = threadIdx.x;
    // load input into __shared__ memory
    float x = 0;
    if(i < n)
        x = input[i];
    sdata[tx] = x;
    __syncthreads();

    // block-wide reduction in __shared__ mem
    for(int offset = blockDim.x / 2; offset > 0; offset >>= 1) {
        if(tx < offset)
            sdata[tx] += sdata[tx + offset];
        __syncthreads();
    }

    if(threadIdx.x == 0)
        results[blockIdx.x] = sdata[0];

    // --> next slide
}
```
How about this kernel? (cont.)

```c
__global__ void block_sum(float *input, float *results, size_t n, size_t num_blocks)
{
    // Let the first block to reduce the partial sums to the final sums.
    if (blockIdx.x == 0) {
        x=0;
        if (tx < num_blocks)
            x=results[tx];
        sdata[tx] = x;
        __syncthreads();

        for(int offset = blockDim.x / 2; offset > 0; offset >>= 1) {
            if(tx < offset)
                sdata[tx] += sdata[tx + offset];
            __syncthreads();
        }

        if(threadIdx.x == 0)
            results[0] = sdata[0];
    }
}
```
Outline

Introduction

Reduction

All-Prefix-Sums
  Applications
  Avoiding Bank Conflicts

Segmented Scan

Sorting
What is all-prefix-sums?

**Definition**
The all-prefix-sums operation takes a binary associative operator $\oplus$, and an array of $n$ elements
\[ [a_0, a_1, \ldots, a_{n-1}] \]
and returns the array
\[ [a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus a_2 \oplus \cdots \oplus a_{n-2} \oplus a_{n-1})]. \]

**Example**
If $\oplus$ is addition, then the all-prefix-sums operation on the array
\[ [3, 1, 7, 0, 4, 1, 6, 3], \]
would return
\[ [3, 4, 11, 11, 15, 16, 22, 25]. \]

**Pseudo code**
```plaintext```
out[0] = in[0];
for (j=1; j<n; j++) {
    out[j] = out[j-1] operator in[j];
}
```plaintext```
Exclusive Scan (Prescan)

**Definition**
The exclusive scan operation takes a binary associative operator $\oplus$ with identify $I$, and an array of $n$ elements $[a_0, a_1, \ldots, a_{n-1}]$ and returns the array $[I, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus a_2 \oplus \cdots \oplus a_{n-2})]$.

**Example**
If $\oplus$ is addition, then the prescan operation on the array $[3, 1, 7, 0, 4, 1, 6, 3]$, would return $[0, 3, 4, 11, 11, 15, 16, 22]$.

**Pseudo code**
```python
out[0] = I;
for (j=1; j<n; j++) {
    out[j] = out[j-1] operator in[j-1];
}
```
Parallel Scan in CUDA

3 1 7 0 4 1 6 3

Assume array is already in shared memory
Parallel Scan in CUDA

Each node corresponds to a single thread.

Iterate log(n) times. Each thread adds value stride elements away to its own value.
Parallel Scan in CUDA

Iterate $\log(n)$ times. Each thread adds value $offset$ elements away to its own value.

Each $\bigoplus$ corresponds to a single thread.

Iteration 1, $n-2$ threads
Parallel Scan in CUDA

Iterate log(n) times. Each thread adds value offset elements away to its own value.

Note that this algorithm operates in-place: no need for double buffering.
Outline

Introduction

Reduction

All-Prefix-Sums
   Applications
      Avoiding Bank Conflicts

Segmented Scan

Sorting
Line-of-Sight

Problem
Given (1) a terrain map in the form of a grid of altitude and an observation point $X$ on the grid, and (2) the distance between interesting points and the observation point, find which points are visible from $X$.

▶ Altitude vector

<table>
<thead>
<tr>
<th></th>
<th>100</th>
<th>150</th>
<th>200</th>
<th>300</th>
<th>400</th>
<th>350</th>
<th>400</th>
<th>500</th>
<th>600</th>
<th>700</th>
<th>650</th>
<th>600</th>
<th>500</th>
</tr>
</thead>
</table>

▶ Distance vector

|   | 0   | 100 | 200 | 300 | 400 | 500 | 600 | 700 | 800 | 900 | 1000 | 1100 | 1200 |
Solving Line-of-Sight

Given

- **Altitude vector**

| 100 | 150 | 200 | 300 | 400 | 350 | 400 | 500 | 600 | 700 | 650 | 600 | 500 |

- **Distance vector**

| 0  | 100 | 200 | 300 | 400 | 500 | 600 | 700 | 800 | 900 | 1000 | 1100 | 1200 |
Solving Line-of-Sight

Given

- **Altitude vector**

<table>
<thead>
<tr>
<th></th>
<th>100</th>
<th>150</th>
<th>200</th>
<th>300</th>
<th>400</th>
<th>350</th>
<th>400</th>
<th>500</th>
<th>600</th>
<th>700</th>
<th>650</th>
<th>600</th>
<th>500</th>
</tr>
</thead>
</table>

- **Distance vector**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>100</th>
<th>200</th>
<th>300</th>
<th>400</th>
<th>500</th>
<th>600</th>
<th>700</th>
<th>800</th>
<th>900</th>
<th>1000</th>
<th>1100</th>
<th>1200</th>
</tr>
</thead>
</table>

**Steps**

- **Vector of difference of altitude**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>50</th>
<th>100</th>
<th>200</th>
<th>300</th>
<th>250</th>
<th>300</th>
<th>400</th>
<th>500</th>
<th>600</th>
<th>550</th>
<th>500</th>
<th>400</th>
</tr>
</thead>
</table>

- **Vector of angle**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0.5</th>
<th>0.5</th>
<th>0.67</th>
<th>0.75</th>
<th>0.5</th>
<th>0.5</th>
<th>0.57</th>
<th>0.625</th>
<th>0.67</th>
<th>0.55</th>
<th>0.45</th>
<th>0.33</th>
</tr>
</thead>
</table>

- **Max-Scan of vector of angle**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0.5</th>
<th>0.5</th>
<th>0.67</th>
<th>0.75</th>
<th>0.75</th>
<th>0.75</th>
<th>0.75</th>
<th>0.75</th>
<th>0.75</th>
<th>0.75</th>
<th>0.75</th>
<th>0.75</th>
</tr>
</thead>
</table>

- **Compare**

|   | 0   | 0.5 | 0.5 | 0.67 | 0.75 | 0.5 | 0.5 | 0.57 | 0.625 | 0.67 | 0.55 | 0.45 | 0.33 |
Split Operation

- Given an array of true and false elements (and payloads)

<table>
<thead>
<tr>
<th>Payload</th>
<th>5</th>
<th>7</th>
<th>3</th>
<th>1</th>
<th>4</th>
<th>2</th>
<th>7</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flag</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
## Split Operation

- Given an array of true and false elements (and payloads)
- Return an array with all false (or true) elements at the beginning

<table>
<thead>
<tr>
<th>Payload</th>
<th>5</th>
<th>7</th>
<th>3</th>
<th>1</th>
<th>4</th>
<th>2</th>
<th>7</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flag</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4</th>
<th>2</th>
<th>2</th>
<th>5</th>
<th>7</th>
<th>3</th>
<th>1</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
A Similar Operation – Compact

- Given an array of true and false elements (and payloads)
- Remove all null elements

<table>
<thead>
<tr>
<th>Payload</th>
<th>5</th>
<th>7</th>
<th>3</th>
<th>1</th>
<th>4</th>
<th>2</th>
<th>7</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flag</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Split – Approach

- Determine the new index for each element and then permute
  - False elements, i.e., elements with flag 0
    - Invert the flags and execute a prescan with integer addition
  - True elements, i.e., elements with flag 1
    - Execute a $+ -$-scan in reverse order (i.e., starting from the tail of the vector) and subtract the results from $n$ (i.e., the length of the vector)

<table>
<thead>
<tr>
<th>Payload</th>
<th>Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>
Split – Approach

- Determine the new index for each element and then permute
  - False elements, i.e., elements with flag 0
    - Invert the flags and execute a prescan with integer addition
  - True elements, i.e., elements with flag 1
    - Execute a $+$-scan in reverse order (i.e., starting from the tail of the vector) and subtract the results from $n$ (i.e., the length of the vector)

<table>
<thead>
<tr>
<th>Payload</th>
<th>5</th>
<th>7</th>
<th>3</th>
<th>1</th>
<th>4</th>
<th>2</th>
<th>7</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flag</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>I-down</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
Split – Approach

- Determine the new index for each element and then permute
  - False elements, i.e., elements with flag 0
    - Invert the flags and execute a prescan with integer addition
  - True elements, i.e., elements with flag 1
    - Execute a \( + \)-scan in reverse order (i.e., starting from the tail of the vector) and subtract the results from \( n \) (i.e., the length of the vector)

<table>
<thead>
<tr>
<th>Payload</th>
<th>Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

Payload Flag
I-down
\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 \\
\end{array}
\]

I-up
\[
\begin{array}{cccccccc}
5 & 4 & 3 & 2 & 1 & 1 & 1 & 0 \\
\end{array}
\]
Split – Approach

- Determine the new index for each element and then permute
  - False elements, i.e., elements with flag 0
    - Invert the flags and execute a prescan with integer addition
  - True elements, i.e., elements with flag 1
    - Execute a $\pm$-scan in reverse order (i.e., starting from the tail of the vector) and subtract the results from $n$ (i.e., the length of the vector)

<table>
<thead>
<tr>
<th>Payload</th>
<th>5</th>
<th>7</th>
<th>3</th>
<th>1</th>
<th>4</th>
<th>2</th>
<th>7</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flag</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>I-down</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>I-up</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Subtract</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
Split – Approach

- Determine the new index for each element and then permute
  - False elements, i.e., elements with flag 0
    - Invert the flags and execute a prescan with integer addition
  - True elements, i.e., elements with flag 1
    - Execute a + -scan in reverse order (i.e., starting from the tail of the vector) and subtract the results from \( n \) (i.e., the length of the vector)

<table>
<thead>
<tr>
<th>Payload</th>
<th>Flag</th>
<th>I-down</th>
<th>I-up</th>
<th>Subtract</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Payload: 54 / 102
Split – Approach

- Determine the new index for each element and then permute
  - False elements, i.e., elements with flag 0
    - Invert the flags and execute a prescan with integer addition
  - True elements, i.e., elements with flag 1
    - Execute a $+$-scan in reverse order (i.e., starting from the tail of the vector) and subtract the results from $n$ (i.e., the length of the vector)

<table>
<thead>
<tr>
<th>Payload</th>
<th>5</th>
<th>7</th>
<th>3</th>
<th>1</th>
<th>4</th>
<th>2</th>
<th>7</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flag</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>I-down</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>I-up</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Subtract</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>Index</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>0</td>
<td>1</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>Permute</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>7</td>
<td>3</td>
<td>1</td>
<td>7</td>
</tr>
</tbody>
</table>
Outline

Introduction

Reduction

All-Prefix-Sums

Applications

Avoiding Bank Conflicts

Segmented Scan

Sorting
Shared Memory Banks – Relocation of Data in Shared Memory

| core 0 | core 1 | core 2 | core 3 | core 4 | core 5 | core 6 | core 7 | core 8 | core 9 | core 10 | core 11 | core 12 | core 13 | core 14 | core 15 | core 16 | core 17 | core 18 | core 19 | core 20 | core 21 | core 22 | core 23 | core 24 | core 25 | core 26 | core 27 | core 28 | core 29 | core 30 | core 31 |
|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|
| 0 63 94 | 1 32 95 | 2 33 64 | 3 34 65 | 4 35 66 | 5 36 67 | 6 37 68 | 7 38 69 | 8 39 70 | 9 40 71 | 10 41 72 | 11 42 73 | 12 43 74 | 13 44 75 | 14 45 76 | 15 46 77 | 16 47 78 | 17 48 79 | 18 49 80 | 19 50 81 | 20 51 82 | 21 52 83 | 22 53 84 | 23 54 85 | 24 55 86 | 25 56 87 | 26 57 88 | 27 58 89 | 28 59 90 | 29 60 91 | 30 61 92 | 31 62 93 |
Shared Memory Banks – Conflict Access Free!
Outline

Introduction

Reduction

All-Prefix-Sums
  Applications
  Avoiding Bank Conflicts

Segmented Scan

Sorting
Segmented Scan

What is segmented scan?

- Scan + Barriers/Flags associated with certain positions in the input arrays
- Operations do not propagate beyond barriers

<table>
<thead>
<tr>
<th>Value</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>1</th>
<th>3</th>
<th>9</th>
<th>10</th>
<th>12</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flag</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Output</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>4</td>
<td>9</td>
<td>15</td>
<td>22</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>10</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>
**Segmented Scan**

**What is segmented scan?**

- Scan + Barriers/Flags associated with certain positions in the input arrays
- Operations do not propagate beyond barriers

<table>
<thead>
<tr>
<th>Value</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>1</th>
<th>3</th>
<th>9</th>
<th>10</th>
<th>12</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flag</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Output</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>4</td>
<td>9</td>
<td>15</td>
<td>22</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>10</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

- How to deal with it?
Segmented Scan

What is segmented scan?

- Scan + Barriers/Flags associated with certain positions in the input arrays
- Operations do not propagate beyond barriers

<table>
<thead>
<tr>
<th>Value</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>1</th>
<th>3</th>
<th>9</th>
<th>10</th>
<th>12</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flag</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Output</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>4</td>
<td>9</td>
<td>15</td>
<td>22</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>10</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

- How to deal with it?
  - Deal with the segments one by one
Segmented Scan

What is segmented scan?

- Scan + Barriers/Flags associated with certain positions in the input arrays
- Operations do not propagate beyond barriers

<table>
<thead>
<tr>
<th>Value</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>1</th>
<th>3</th>
<th>9</th>
<th>10</th>
<th>12</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flag</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

| Output | 1 | 3 | 6 | 4 | 9 | 15 | 22 | 1 | 4 | 9 | 10 | 22 | 23 | 24 | 1 | 3 |   |

- How to deal with it?
  - Deal with the segments one by one
  - Do many scans at once, no matter their sizes
### Segmented Scan – Example (of the Naive Approach)

<table>
<thead>
<tr>
<th>Value</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>1</th>
<th>3</th>
<th>9</th>
<th>10</th>
<th>12</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flag</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
### Segmented Scan – Example (of the Naive Approach)

<table>
<thead>
<tr>
<th>Value</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>1</th>
<th>3</th>
<th>9</th>
<th>10</th>
<th>12</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flag</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- **Value**: Represents the data values to be scanned.
- **Flag**: Indicates the start and end of scanned regions.

The arrows indicate the scan path from left to right, following the flagged values.
### Segmented Scan – Example (of the Naive Approach)

<table>
<thead>
<tr>
<th>Value</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>1</th>
<th>3</th>
<th>9</th>
<th>10</th>
<th>12</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flag</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Value

| Value | 1 | 3 | 5 | 7 | 9 | 11 | 13 | 1 | 4 | 12 | 19 | 22 | 13 | 2 | 2 | 3 |
|-------|---|---|---|---|---|-----|-----|---|---|----|----|----|----|---|---|---|---|
| Flag  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Value

| Value | 1 | 3 | 6 | 10 | 14 | 18 | 22 | 1 | 4 | 13 | 23 | 34 | 32 | 24 | 15 | 5 |
|-------|---|---|---|----|----|----|----|---|---|----|----|----|----|----|----|---|---|
| Flag  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |

Flag
Segmented Scan – Example (of the Naive Approach)

<table>
<thead>
<tr>
<th>Value</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>1</th>
<th>3</th>
<th>9</th>
<th>10</th>
<th>12</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flag</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

| Value | 1 | 3 | 5 | 7 | 9 | 11 | 13 | 1 | 4 | 12 | 19 | 22 | 13 | 2 | 2 | 3 |
|-------|---|---|---|---|---|----|----|---|---|----|----|----|----|---|---|---|---|
| Flag  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| Value | 1 | 3 | 6 | 10 | 14 | 18 | 22 | 1 | 4 | 13 | 23 | 34 | 32 | 24 | 15 | 5 |
|-------|---|---|---|----|----|----|----|---|---|----|----|----|----|----|----|---|---|
| Flag  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |

<table>
<thead>
<tr>
<th>Value</th>
<th>1</th>
<th>3</th>
<th>6</th>
<th>10</th>
<th>15</th>
<th>21</th>
<th>28</th>
<th>1</th>
<th>4</th>
<th>13</th>
<th>23</th>
<th>35</th>
<th>36</th>
<th>37</th>
<th>38</th>
<th>39</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flag</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
### Segmented Scan – Example (of the Naive Approach)

<table>
<thead>
<tr>
<th>Value</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>1</th>
<th>3</th>
<th>9</th>
<th>10</th>
<th>12</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flag</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>1</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>9</th>
<th>11</th>
<th>13</th>
<th>1</th>
<th>4</th>
<th>12</th>
<th>19</th>
<th>22</th>
<th>13</th>
<th>2</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flag</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>1</th>
<th>3</th>
<th>6</th>
<th>10</th>
<th>14</th>
<th>18</th>
<th>22</th>
<th>1</th>
<th>4</th>
<th>13</th>
<th>23</th>
<th>34</th>
<th>32</th>
<th>24</th>
<th>15</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flag</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>1</th>
<th>3</th>
<th>6</th>
<th>10</th>
<th>15</th>
<th>21</th>
<th>28</th>
<th>1</th>
<th>4</th>
<th>13</th>
<th>23</th>
<th>35</th>
<th>36</th>
<th>37</th>
<th>38</th>
<th>39</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flag</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>1</th>
<th>3</th>
<th>6</th>
<th>10</th>
<th>15</th>
<th>21</th>
<th>28</th>
<th>1</th>
<th>4</th>
<th>13</th>
<th>23</th>
<th>35</th>
<th>36</th>
<th>37</th>
<th>38</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flag</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
//x[]: value; f[]: flag
for d = 0 to log(n) - 1 do
    forall k in parallel do
        if k>=2^d then
            if f[k] is NOT set then
                x[out][k] = x[in][k-2^d] + x[in][k]
                f[out][k] = f[in][k-2^d] | f[in][k]
            else
                x[out][k] = x[in][k]
                f[out][k] = f[in][k]
        else
            x[out][k] = x[in][k]
            f[out][k] = f[in][k]
    else
        swap(in, out)
// x[]: value; f[]: flag
for d = 0 to log(n) - 1 do
    forall k in parallel do
        if k>=2^d then
            if f[k] is NOT set then
                x[out][k] = x[in][k-2^d] + x[in][k]
                f[out][k] = f[in][k-2^d] | f[in][k]
            else
                x[out][k] = x[in][k]
                f[out][k] = f[in][k]
        else
            x[out][k] = x[in][k]
            f[out][k] = f[in][k]
    end
end
swap(in, out)

- Work-efficient implementation
  - See “Scan Primitives for GPU Computing” by Sengupta, Harris, Zhang, and Owens
Outline

Introduction

Reduction

All-Prefix-Sums
  Applications
  Avoiding Bank Conflicts

Segmented Scan

Sorting
Sort

- Useful for almost everything
- Optimized versions for the GPU already exist
- Two examples
  - Radix sort
  - Quick sort
Radix Sort

Definition

- Sort integers by processing individual digits, by comparing individual digits sharing the same significant position
- Least Significant Digit (LSD) radix sort
  1. Take the least significant digit (or group of bits, both being examples of radices) of each key
  2. Group the keys based on that digit, but otherwise keep the original order of keys
  3. Repeat the grouping process with each more significant digit
### Radix Sort – Example

<table>
<thead>
<tr>
<th>original list</th>
</tr>
</thead>
<tbody>
<tr>
<td>170</td>
</tr>
</tbody>
</table>
Radix Sort – Example

original list

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>170</td>
<td>45</td>
<td>75</td>
<td>90</td>
<td>2</td>
<td>24</td>
<td>802</td>
</tr>
</tbody>
</table>
Radix Sort – Example

original list

1s place
### Radix Sort – Example

**Original List**

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>170</td>
<td>45</td>
<td>75</td>
<td>90</td>
<td>2</td>
<td>24</td>
<td>802</td>
<td>66</td>
</tr>
</tbody>
</table>

**1s Place**

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>170</td>
<td>90</td>
<td>02</td>
<td>802</td>
<td>24</td>
<td>45</td>
<td>75</td>
<td>66</td>
</tr>
</tbody>
</table>
Radix Sort – Example

Original list

| 170 | 45 | 75 | 90 | 2  | 24 | 802 | 66 |

1s place

| 170 | 90 | 02 | 802 | 24 | 45 | 75 | 66 |

10s place

| 002 | 802 | 024 | 045 | 066 | 170 | 075 | 090 |
Radix Sort – Example

original list

1s place

10s place
Radix Sort – Example

original list

1s place

10s place

100s place
Radix Sort on GPU

- Integers are represented in radix-2 format on computer
- **Split** can be used for radix sort
Radix Sort on GPU

- Integers are represented in radix-2 format on computer
- **Split** can be used for radix sort

\[
A = [ 5 \ 7 \ 3 \ 1 \ 4 \ 2 \ 7 \ 2 ]
\]
Radix Sort on GPU

- Integers are represented in radix-2 format on computer
- **Split** can be used for radix sort

\[
\begin{align*}
A &= \begin{bmatrix} 5 & 7 & 3 & 1 & 4 & 2 & 7 & 2 \end{bmatrix} \\
A<0> &= \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \end{bmatrix}
\end{align*}
\]
Radix Sort on GPU

- Integers are represented in radix-2 format on computer
- **Split** can be used for radix sort

\[
A = \begin{bmatrix} 5 & 7 & 3 & 1 & 4 & 2 & 7 & 2 \end{bmatrix}
\]

\[
A^{<0>} = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \end{bmatrix}
\]

\[
A \leftarrow \text{split} \ (A, \ A^{<0>}) = \begin{bmatrix} 4 & 2 & 2 & 5 & 7 & 3 & 1 & 7 \end{bmatrix}
\]
Radix Sort on GPU

- Integers are represented in radix-2 format on computer
- **Split** can be used for radix sort

\[
\begin{align*}
A &= [5 \ 7 \ 3 \ 1 \ 4 \ 2 \ 7 \ 2] \\
A<0> &= [1 \ 1 \ 1 \ 1 \ 0 \ 0 \ 1 \ 0] \\
A &\leftarrow \text{split (A, A<0>)} = [4 \ 2 \ 2 \ 5 \ 7 \ 3 \ 1 \ 7] \\
A<1> &= [0 \ 1 \ 1 \ 0 \ 1 \ 1 \ 0 \ 1]
\end{align*}
\]
Radix Sort on GPU

- Integers are represented in radix-2 format on computer
- **Split** can be used for radix sort

\[
\begin{align*}
A &= [5 \ 7 \ 3 \ 1 \ 4 \ 2 \ 7 \ 2 ] \\
A<0> &= [1 \ 1 \ 1 \ 1 \ 0 \ 0 \ 1 \ 0 ] \\
A \leftarrow \text{split} \ (A, A<0>) &= [4 \ 2 \ 2 \ 5 \ 7 \ 3 \ 1 \ 7 ] \\
A<1> &= [0 \ 1 \ 1 \ 0 \ 1 \ 1 \ 0 \ 1 ] \\
A \leftarrow \text{split} \ (A, A<1>) &= [4 \ 5 \ 1 \ 2 \ 2 \ 7 \ 3 \ 7 ]
\end{align*}
\]
Radix Sort on GPU

- Integers are represented in radix-2 format on computer
- Split can be used for radix sort

\[
A = [5, 7, 3, 1, 4, 2, 7, 2]
\]
\[
A<0> = [1, 1, 1, 1, 0, 0, 1, 0]
\]
\[
A \leftarrow \text{split} \left( A, A<0> \right) = [4, 2, 2, 5, 7, 3, 1, 7]
\]
\[
A<1> = [0, 1, 1, 0, 1, 1, 0, 1]
\]
\[
A \leftarrow \text{split} \left( A, A<1> \right) = [4, 5, 1, 2, 2, 7, 3, 7]
\]
\[
A<2> = [1, 1, 0, 0, 0, 1, 0, 1]
\]
Radix Sort on GPU

- Integers are represented in radix-2 format on computer
- Split can be used for radix sort

\[
\begin{align*}
A &= [5 \ 7 \ 3 \ 1 \ 4 \ 2 \ 7 \ 2] \\
A^{<0>} &= [1 \ 1 \ 1 \ 1 \ 0 \ 0 \ 1 \ 0] \\
A \leftarrow \text{split} (A, A^{<0>}) &= [4 \ 2 \ 2 \ 5 \ 7 \ 3 \ 1 \ 7] \\
A^{<1>} &= [0 \ 1 \ 1 \ 0 \ 1 \ 1 \ 0 \ 1] \\
A \leftarrow \text{split} (A, A^{<1>}) &= [4 \ 5 \ 1 \ 2 \ 2 \ 7 \ 3 \ 7] \\
A^{<2>} &= [1 \ 1 \ 0 \ 0 \ 0 \ 1 \ 0 \ 1] \\
A \leftarrow \text{split} (A, A^{<2>}) &= [1 \ 2 \ 2 \ 3 \ 4 \ 5 \ 7 \ 7]
\end{align*}
\]
Quick Sort

- Quick sort sorts by employing a *divide and conquer* strategy to divide a list into two sub-lists

Steps

1. Pick an element, called a *pivot*, from the list
2. Reorder the list so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way)
   - After this partitioning, the pivot is in its final position
3. Recursively sort the sub-list of lesser elements and the sub-list of greater elements
Quick Sort

- Quick sort sorts by employing a *divide and conquer* strategy to divide a list into two sub-lists

- Steps
  1. Pick an element, called a *pivot*, from the list
  2. Reorder the list so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way)
    - After this partitioning, the pivot is in its final position
  3. Recursively sort the sub-list of lesser elements and the sub-list of greater elements

- Example is given at the chalkboard
Quick Sort on GPU

- Use segmented scan and split
Quick Sort on GPU

- Use segmented scan and split

Key = [ 6.4  9.2  3.4  1.6  8.7  4.1  9.2  3.4 ]
Quick Sort on GPU

- Use segmented scan and split

Key = [ 6.4  9.2  3.4  1.6  8.7  4.1  9.2  3.4 ]

Seg-Flags = [ 1  0  0  0  0  0  0  0  0 ]
Quick Sort on GPU

- Use segmented scan and split

<table>
<thead>
<tr>
<th>Key</th>
<th>=</th>
<th>[ 6.4 9.2 3.4 1.6 8.7 4.1 9.2 3.4 ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seg-Flags</td>
<td>=</td>
<td>[ 1 0 0 0 0 0 0 0 ]</td>
</tr>
<tr>
<td>Pivots</td>
<td>=</td>
<td>[ 6.4 6.4 6.4 6.4 6.4 6.4 6.4 6.4 ]</td>
</tr>
</tbody>
</table>
Quick Sort on GPU

- Use segmented scan and split

Key = [ 6.4 9.2 3.4 1.6 8.7 4.1 9.2 3.4 ]

Seg-Flags = [ 1 0 0 0 0 0 0 0 0 0 ]

Pivots = [ 6.4 6.4 6.4 6.4 6.4 6.4 6.4 6.4 ]

F = [ = > < < > < > < ]
Quick Sort on GPU

- Use segmented scan and split

Key = [ 6.4 9.2 3.4 1.6 8.7 4.1 9.2 3.4 ]

Seg-Flags = [ 1 0 0 0 0 0 0 0 0 0 ]

Pivots = [ 6.4 6.4 6.4 6.4 6.4 6.4 6.4 6.4 ]

F = [ = > < < > < > < ]

Key ← split (Key, F) = [ 3.4 1.6 4.1 3.4 6.4 9.2 8.7 9.2 ]
Quick Sort on GPU

- Use segmented scan and split

Key = [ 6.4 9.2 3.4 1.6 8.7 4.1 9.2 3.4 ]

Seg-Flags = [ 1 0 0 0 0 0 0 0 0 0 ]

Pivots = [ 6.4 6.4 6.4 6.4 6.4 6.4 6.4 6.4 ]

F = [ = > < < > < > < ]

Key ← split (Key, F) = [ 3.4 1.6 4.1 3.4 6.4 9.2 8.7 9.2 ]

Seg-Flags = [ 1 0 0 0 1 1 0 0 0 ]
Quick Sort on GPU

- Use segmented scan and split

<table>
<thead>
<tr>
<th>Key</th>
<th>=</th>
<th>[ 6.4  9.2  3.4  1.6  8.7  4.1  9.2  3.4 ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seg-Flags</td>
<td>=</td>
<td>[ 1  0  0  0  0  0  0  0  0  0 ]</td>
</tr>
<tr>
<td>Pivots</td>
<td>=</td>
<td>[ 6.4  6.4  6.4  6.4  6.4  6.4  6.4  6.4 ]</td>
</tr>
<tr>
<td>F</td>
<td>=</td>
<td>[ =  &gt;  &lt;  &lt;  &gt;  &lt;  &gt;  &lt; ]</td>
</tr>
<tr>
<td>Key ← split (Key, F)</td>
<td>=</td>
<td>[ 3.4  1.6  4.1  3.4  6.4  9.2  8.7  9.2 ]</td>
</tr>
<tr>
<td>Seg-Flags</td>
<td>=</td>
<td>[ 1  0  0  0  0  1  1  0  0  0 ]</td>
</tr>
<tr>
<td>Pivots</td>
<td>=</td>
<td>[ 3.4  3.4  3.4  3.4  6.4  9.2  9.2  9.2 ]</td>
</tr>
</tbody>
</table>
Quick Sort on GPU

- Use segmented scan and split

Key = [ 6.4 9.2 3.4 1.6 8.7 4.1 9.2 3.4 ]

Seg-Flags = [ 1 0 0 0 0 0 0 0 0 0 ]

Pivots = [ 6.4 6.4 6.4 6.4 6.4 6.4 6.4 6.4 ]

F = [ = > < < > < > < ]

Key ← split (Key, F) = [ 3.4 1.6 4.1 3.4 6.4 9.2 8.7 9.2 ]

Seg-Flags = [ 1 0 0 0 0 1 1 0 0 0 ]

Pivots = [ 3.4 3.4 3.4 3.4 6.4 9.2 9.2 9.2 ]

F = [ = < > = = = < = ]
Quick Sort on GPU

- Use segmented scan and split

<table>
<thead>
<tr>
<th>Key</th>
<th>=</th>
<th>[ 6.4 9.2 3.4 1.6 8.7 4.1 9.2 3.4 ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seg-Flags</td>
<td>=</td>
<td>[ 1 0 0 0 0 0 0 0 0 ]</td>
</tr>
<tr>
<td>Pivots</td>
<td>=</td>
<td>[ 6.4 6.4 6.4 6.4 6.4 6.4 6.4 6.4 ]</td>
</tr>
<tr>
<td>F</td>
<td>=</td>
<td>[ = &gt; &lt; &lt; &gt; &lt; &gt; &lt; ]</td>
</tr>
<tr>
<td>Key ← split (Key, F)</td>
<td>=</td>
<td>[ 3.4 1.6 4.1 3.4 6.4 9.2 8.7 9.2 ]</td>
</tr>
<tr>
<td>Seg-Flags</td>
<td>=</td>
<td>[ 1 0 0 0 1 1 0 0 0 ]</td>
</tr>
<tr>
<td>Pivots</td>
<td>=</td>
<td>[ 3.4 3.4 3.4 3.4 6.4 9.2 9.2 9.2 ]</td>
</tr>
<tr>
<td>F</td>
<td>=</td>
<td>[ = &lt; &gt; = = = &lt; = ]</td>
</tr>
<tr>
<td>Key ← split (Key, F)</td>
<td>=</td>
<td>[ 1.6 3.4 4.1 3.4 6.4 8.7 9.2 9.2 ]</td>
</tr>
</tbody>
</table>
## Quick Sort on GPU

- Use segmented scan and split

<table>
<thead>
<tr>
<th>Key</th>
<th>=</th>
<th>[   6.4  9.2  3.4  1.6  8.7  4.1  9.2  3.4 ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seg-Flags</td>
<td>=</td>
<td>[   1   0   0   0   0   0   0   0   0 ]</td>
</tr>
<tr>
<td>Pivots</td>
<td>=</td>
<td>[   6.4  6.4  6.4  6.4  6.4  6.4  6.4  6.4 ]</td>
</tr>
<tr>
<td>F</td>
<td>=</td>
<td>[   =   &gt;   &lt;   &lt;   &gt;   &lt;   &gt;   &lt; ]</td>
</tr>
<tr>
<td>Key ← split (Key, F)</td>
<td>=</td>
<td>[   3.4  1.6  4.1  3.4  6.4  9.2  8.7  9.2 ]</td>
</tr>
<tr>
<td>Seg-Flags</td>
<td>=</td>
<td>[   1   0   0   0   1   1   0   0   0 ]</td>
</tr>
<tr>
<td>Pivots</td>
<td>=</td>
<td>[   3.4  3.4  3.4  3.4  6.4  9.2  9.2  9.2 ]</td>
</tr>
<tr>
<td>F</td>
<td>=</td>
<td>[   =   &lt;   &gt;   =   =   =   &lt;   = ]</td>
</tr>
<tr>
<td>Key ← split (Key, F)</td>
<td>=</td>
<td>[   1.6  3.4  4.1  3.4  6.4  8.7  9.2  9.2 ]</td>
</tr>
<tr>
<td>Seg-Flags</td>
<td>=</td>
<td>[   1   1   1   0   1   1   1   1   1 ]</td>
</tr>
</tbody>
</table>