GPU Programming

Parallel Patterns

Miaoqing Huang
University of Arkansas

Outline

Outline

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

```
__global__ void foo(...)
{
    extern __shared__ smem[];
    int i = ???
    ......
}
......
dim3 dim_grid = ???
dim3 dim_block = ???
foo<<<dim_grid, dim_block>>>();
```

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

```
__global__ void foo(...)
{
    extern __shared__ smem[];
    int i = ???
    ......
}
......
dim3 dim_grid = ???
dim3 dim_block = ???
foo<<<dim_grid, dim_block>>>();
```

- Now what?

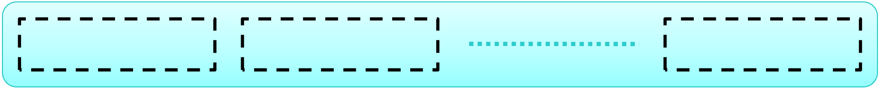
Parallel Patterns

- 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 $\rightarrow$
    - Reduce
  - Many parallel threads need to partition data $\rightarrow$
    - Split
  - Many parallel threads produce variable output / thread $\rightarrow$
    - Compact / Expand
  - Parallel prefix sum, a.k.a,
    - scan

Primordial CUDA Pattern: Blocking

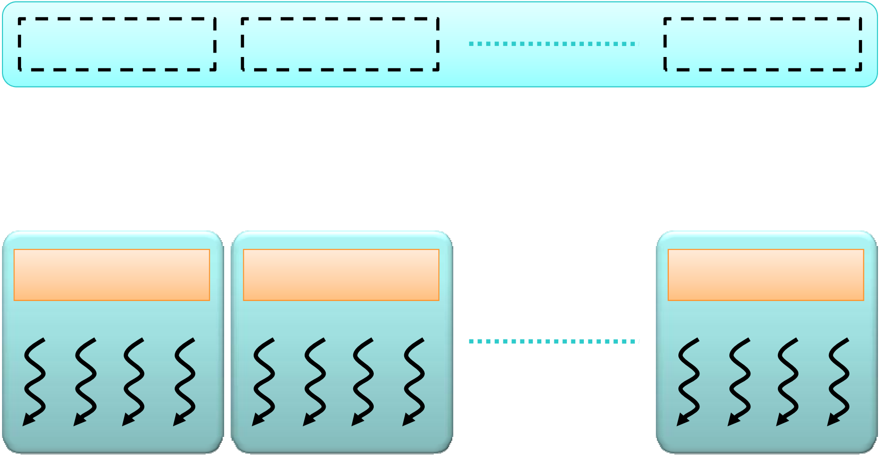
- Partition data to operate in well-sized blocks
  - Small enough to be staged in shared memory
  - Assign each data partition to a thread block
  - No different from cache blocking!
- Provide several performance benefits
  - Have enough blocks to keep processors busy
  - Working in shared memory cuts 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

A Common Programming Strategy – Demo



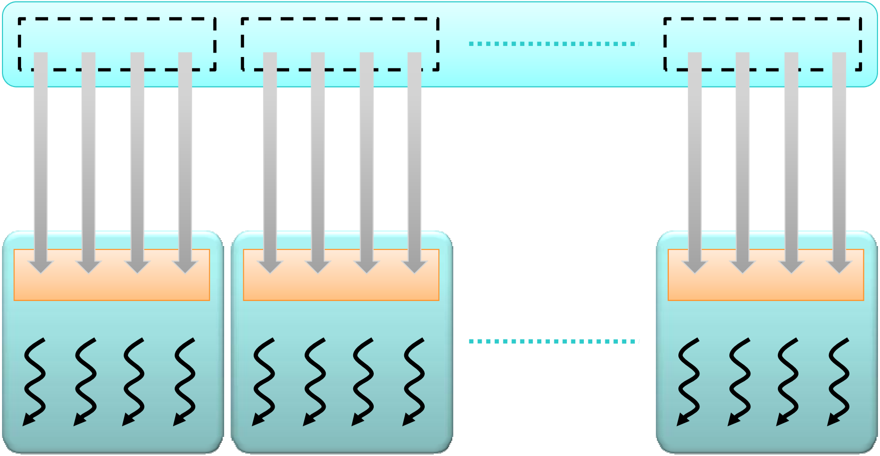
1. Partition data into subsets that fit into shared memory

A Common Programming Strategy – Demo



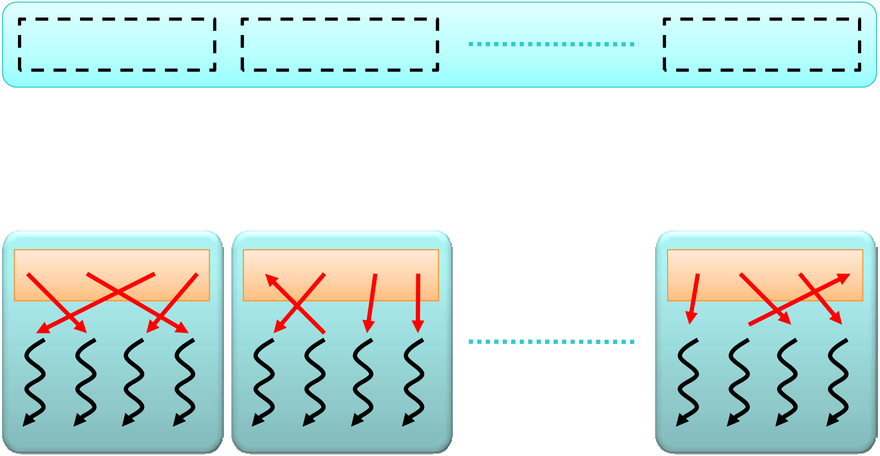
2. Handle each data subset with one thread block

A Common Programming Strategy – Demo



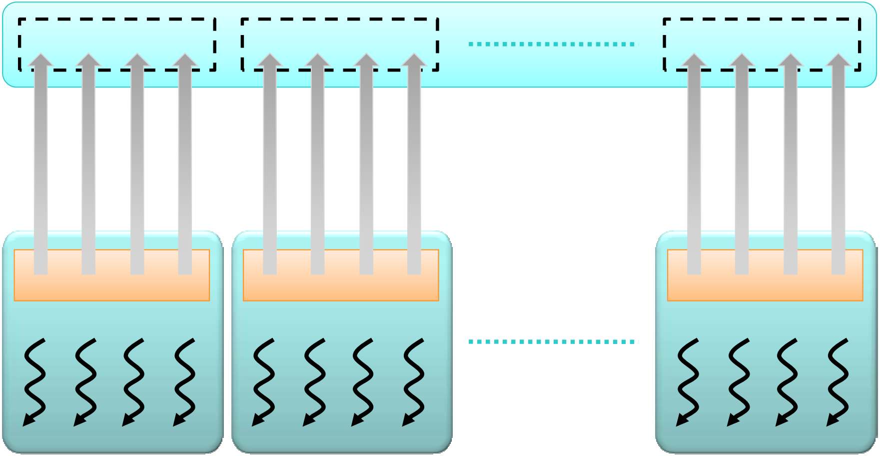
3. Load the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism

A Common Programming Strategy – Demo



4. Perform the computation on the subset from shared memory

A Common Programming Strategy – Demo

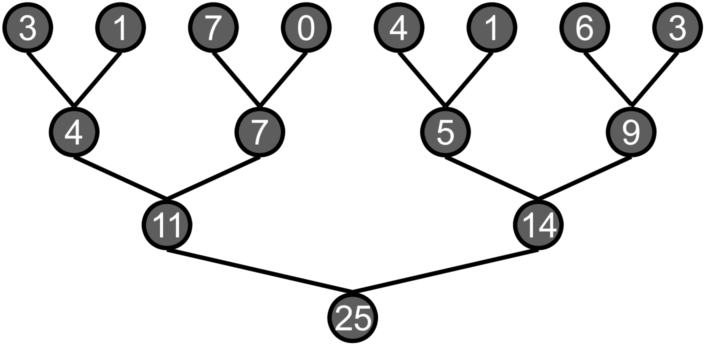


5. Copy the result from shared memory back to global memory

Outline

Reduction

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



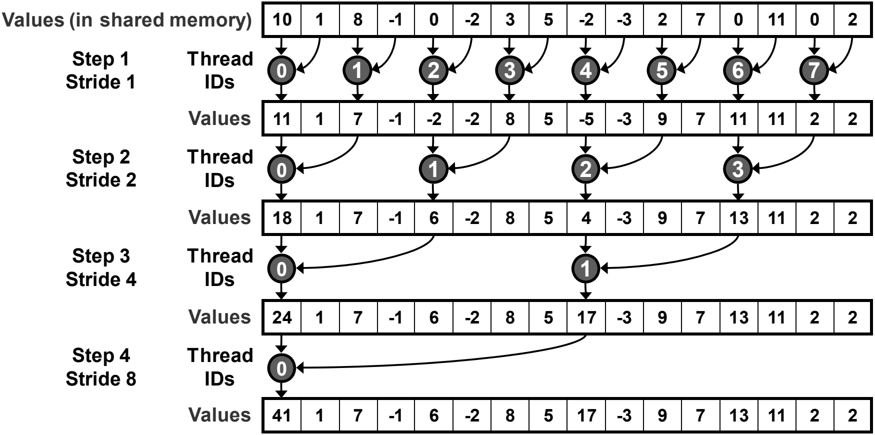
Serial Reduction

- Reduction via serial iteration

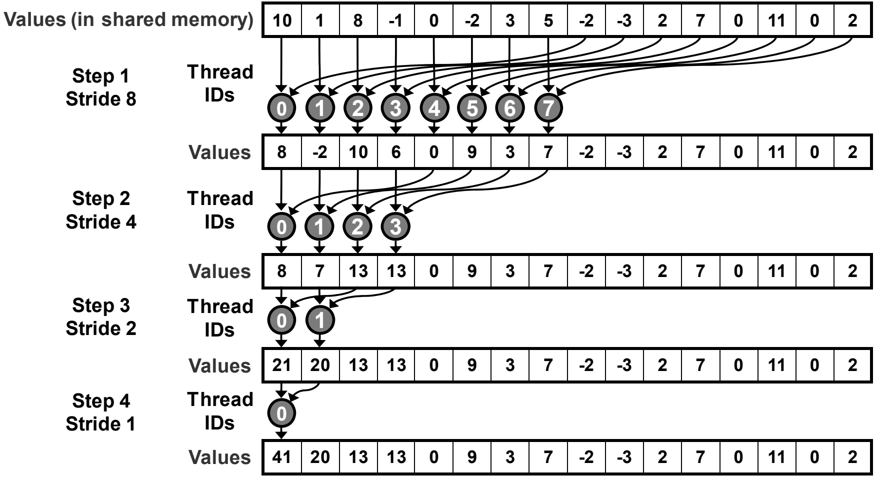
```
float sum(float *data, int n)
{
  float result = 0;
  for(int i = 0; i < n; ++i) {
    result += data[i];
  }

  return result;
}
```

Parallel Reduction

Interleaved



Parallel Reduction
Contiguous



CUDA Reduction – Per-block

```
__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

```
__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?

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

Barrier Divergence

- Is this barrier divergent?

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

- How about this one?

```
__global__ void do_i_halt(int *input)
{
  int i = ......
  if(input[i]) {
    ......
    __syncthreads();
  }
}
```

CUDA Reduction

The first option: launch the kernel twice

```
// 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;
}
```

Discussion

- What happens if there are too many partial sums to fit into `__shared__` memory in the second stage?

Discussion

- 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

Discussion

- 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
- Or, launch the kernel ≥ 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`

```
__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?

```
__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.)*

```
__global__ void block_sum(float *input, float *results, size_t n, size_t num_blocks)
  // <-- from previous slide
  // Let the first block to reduce the partial sums to the final sum.
  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

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

```
out[0] = in[0];
for (j=1; j<n; j++) {
    out[j] = out[j-1] operator in[j];
}
```

Exclusive Scan (Prescan)

Definition

The exclusive scan operation takes a binary associative operator $\oplus$ with identity I, and an array of n elements

$$[a_0, a_1, \dots, a_{n-1}]$$

and returns the array

$$[I, a_0, (a_0 \oplus a_1), \dots, (a_0 \oplus a_1 \oplus a_2 \oplus \cdots \oplus a_{n-2})].$$

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

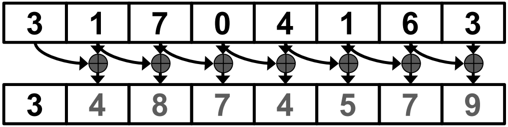
```
out[0] = I;
for (j=1; j<n; j++) {
    out[j] = out[j-1] operator in[j-1];
}
```

Parallel Scan in CUDA

3	**1**	**7**	**0**	**4**	**1**	**6**	**3**

Assume array is already in shared memory

Parallel Scan in CUDA

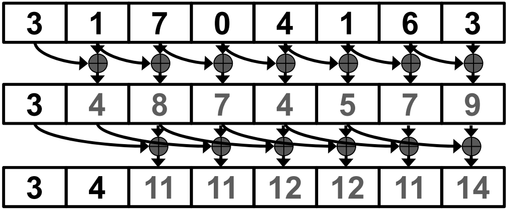


Iteration 0, *n-1* threads

Each ⊕ corresponds to a single thread.

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

Parallel Scan in CUDA

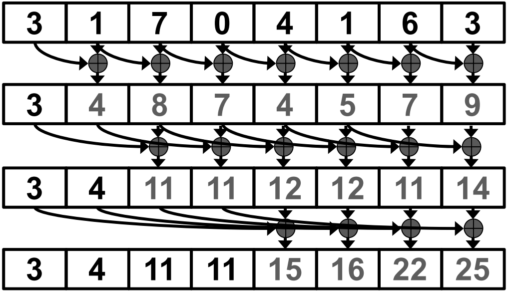


Iteration 1, *n-2* threads

Each ⊕ corresponds to a single thread.

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

Parallel Scan in CUDA



Iteration *i*, *n*-2*^i^* threads

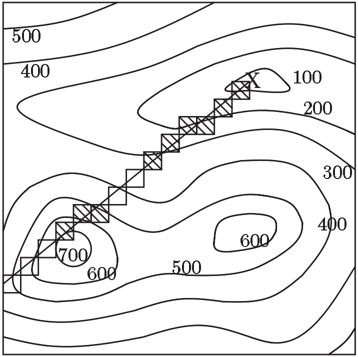
Each ⊕ corresponds to a single thread.

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

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.

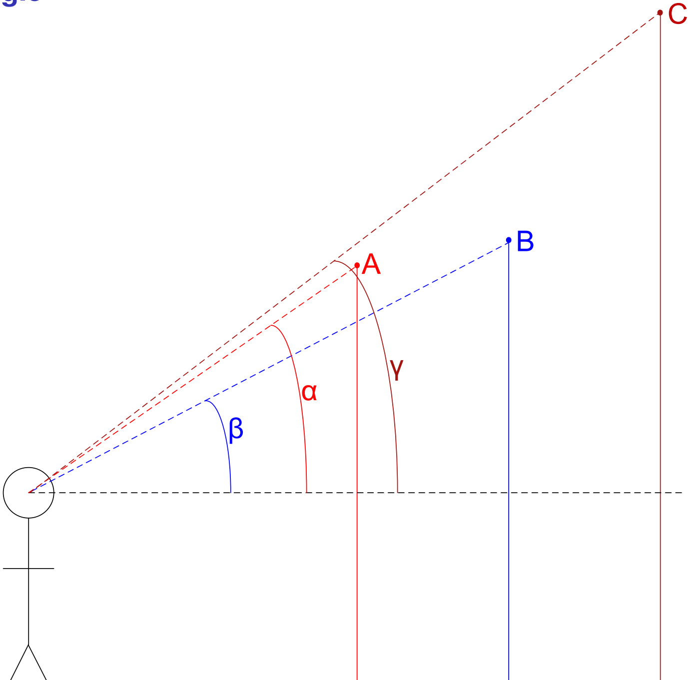
- Altitude vector

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

View Angle



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

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

Steps

- Vector of difference of altitude

0	50	100	200	300	250	300	400	500	600	550	500	400

- Vector of angle

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

- Max-Scan of vector of angle

0	0.5	0.5	0.67	0.75	0.75	0.75	0.75	0.75	0.75	0.75	0.75	0.75

- 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

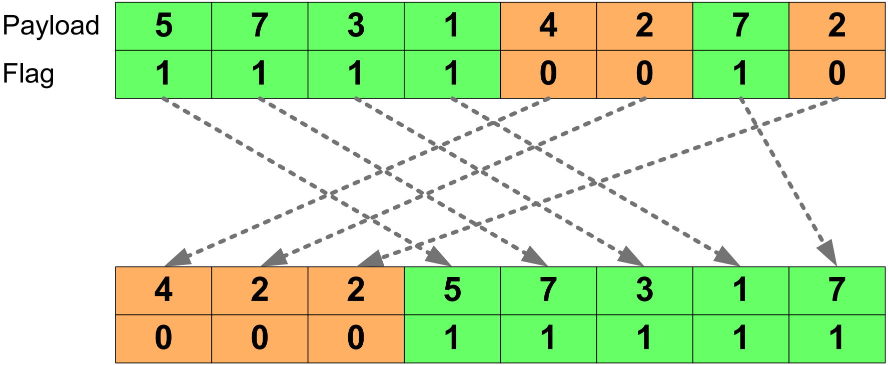
Split Operation

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

Payload	**5**	**7**	**3**	**1**	**4**	**2**	**7**	**2**
Flag	**1**	**1**	**1**	**1**	**0**	**0**	**1**	**0**

Split Operation

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



A Similar Operation – Compact

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

Payload	**5**	**7**	**3**	**1**	**4**	**2**	**7**	**2**
Flag	**1**	**1**	**1**	**1**	**0**	**0**	**1**	**0**

5	**7**	**3**	**1**	**7**
1	**1**	**1**	**1**	**1**

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)

Payload	**5**	**7**	**3**	**1**	**4**	**2**	**7**	**2**
Flag	**1**	**1**	**1**	**1**	**0**	**0**	**1**	**0**

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)

Payload	**5**	**7**	**3**	**1**	**4**	**2**	**7**	**2**
Flag	**1**	**1**	**1**	**1**	**0**	**0**	**1**	**0**
I-down →	**0**	**0**	**0**	**0**	$\boxed{\mathbf{0}}$	$\boxed{\mathbf{1}}$	**2**	$\boxed{\mathbf{2}}$

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)

Payload	5	7	3	1	4	2	7	2
Flag	1	1	1	1	0	0	1	0
I-down →	0	0	0	0	0	1	2	2
I-up ←	5	4	3	2	1	1	1	0

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)

Payload	5	7	3	1	4	2	7	2
Flag	1	1	1	1	0	0	1	0
I-down →	0	0	0	0	0	1	2	2
I-up ←	5	4	3	2	1	1	1	0
Subtract	3	4	5	6	7	7	7	8

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)

Payload	5	7	3	1	4	2	7	2
Flag	1	1	1	1	0	0	1	0
I-down $\rightarrow$	0	0	0	0	0	1	2	2
I-up $\leftarrow$	5	4	3	2	1	1	1	0
Subtract	3	4	5	6	7	7	7	8
Index	3	4	5	6	0	1	7	2

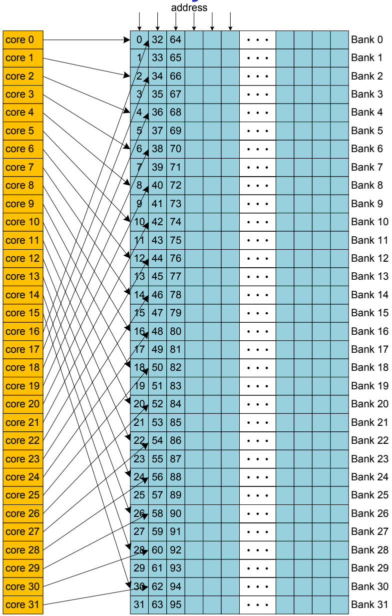
Split – Approach

- 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)

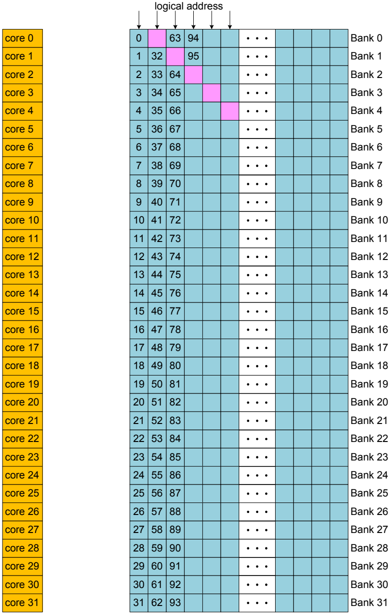
Payload	5	7	3	1	4	2	7	2
Flag	1	1	1	1	0	0	1	0
I-down →	0	0	0	0	0	1	2	2
I-up ←	5	4	3	2	1	1	1	0
Subtract	3	4	5	6	7	7	7	8
Index	3	4	5	6	0	1	7	2
Permute	4	2	2	5	7	3	1	7

Outline

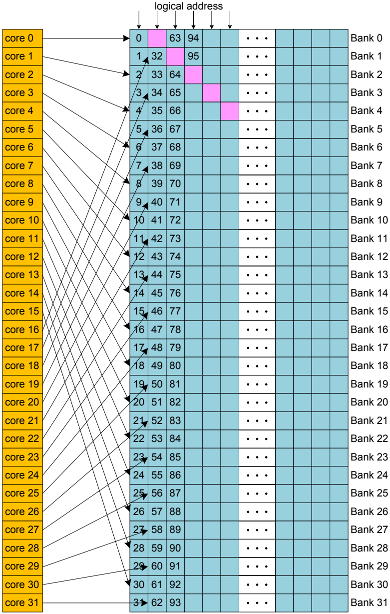
Shared Memory Banks – Two Way Conflict Detail



Shared Memory Banks – Relocation of Data in Shared Memory



Shared Memory Banks – Conflict Access Free!



Outline

Segmented Scan

What is segmented scan?

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

Value	1	2	3	4	5	6	7	1	3	9	10	12	1	1	1	2
Flag	0	0	0	1	0	0	0	1	0	1	1	0	0	0	1	0

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

Segmented Scan

What is segmented scan?

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

Value	1	2	3	4	5	6	7	1	3	9	10	12	1	1	1	2
Flag	0	0	0	1	0	0	0	1	0	1	1	0	0	0	1	0

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

- 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

Value	1	2	3	4	5	6	7	1	3	9	10	12	1	1	1	2
Flag	0	0	0	1	0	0	0	1	0	1	1	0	0	0	1	0

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

Segmented Scan

What is segmented scan?

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

Value	1	2	3	4	5	6	7	1	3	9	10	12	1	1	1	2
Flag	0	0	0	1	0	0	0	1	0	1	1	0	0	0	1	0

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)

Value	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**1**	**3**	**9**	**10**	**12**	**1**	**1**	**1**	**2**
Flag	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

Segmented Scan – Example (of the Naive Approach)

Value	1	2	3	4	5	6	7	1	3	9	10	12	1	1	1	2
Flag	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0

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

Segmented Scan – Example (of the Naive Approach)

Value	1	2	3	4	5	6	7	1	3	9	10	12	1	1	1	2
Flag	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0

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	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

Segmented Scan – Example (of the Naive Approach)

Value	1	2	3	4	5	6	7	1	3	9	10	12	1	1	1	2
Flag	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0

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	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

Value	1	3	6	10	15	21	28	1	4	13	23	35	36	37	38	39
Flag	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0

Segmented Scan – Example (of the Naive Approach)

Value	1	2	3	4	5	6	7	1	3	9	10	12	1	1	1	2
Flag	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
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	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
Value	1	3	6	10	15	21	28	1	4	13	23	35	36	37	38	39
Flag	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0
Value	1	3	6	10	15	21	28	1	4	13	23	35	36	37	38	40
Flag	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1

Segmented Scan – Pseudo Code (of the Naive Approach)

```
//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]
    swap(in,out)
```

Segmented Scan – Pseudo Code (of the Naive Approach)

```
//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]
    swap(in,out)
```

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

Outline

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

original list	170	45	75	90	2	24	802	66

Radix Sort – Example

original list	17<u>0</u>	4<u>5</u>	7<u>5</u>	9<u>0</u>	<u>2</u>	2<u>4</u>	80<u>2</u>	6<u>6</u>

Radix Sort – Example

original list	170	45	75	90	2	24	802	66
1s place	170	90	02	802	24	45	75	66

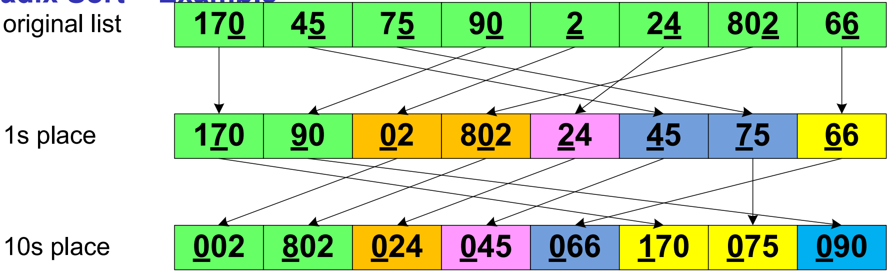
Radix Sort – Example

original list	170	45	75	90	2	24	802	66
1s place	170	90	02	802	24	45	75	66

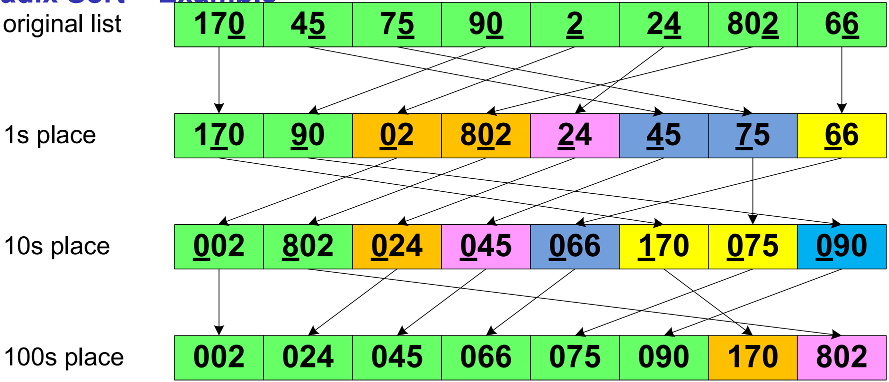
Radix Sort – Example

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



Radix Sort – Example



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

$$A = [\; 5 \quad 7 \quad 3 \quad 1 \quad 4 \quad 2 \quad 7 \quad 2 \;]$$

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	]

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 ← split (A, A<0>)	=	[4 2 2 5 7 3 1 7]

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 $\leftarrow$ split (A, A<0>)	=	[4 2 2 5 7 3 1 7]
A<1>	=	[0 1 1 0 1 1 0 1]

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 ← split (A, A<0>)	=	[	4	2	2	5	7	3	1	7	]
A<1>	=	[	0	1	1	0	1	1	0	1	]
A ← split (A, A<1>)	=	[	4	5	1	2	2	7	3	7	]

Radix Sort on GPU

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

$$
\begin{array}{lcl}
A & = & [\ 5\ \ 7\ \ 3\ \ 1\ \ 4\ \ 2\ \ 7\ \ 2\ ] \\
A{<}0{>} & = & [\ 1\ \ 1\ \ 1\ \ 1\ \ 0\ \ 0\ \ 1\ \ 0\ ] \\
A \leftarrow \text{split}\ (A, A{<}0{>}) & = & [\ 4\ \ 2\ \ 2\ \ 5\ \ 7\ \ 3\ \ 1\ \ 7\ ] \\
A{<}1{>} & = & [\ 0\ \ 1\ \ 1\ \ 0\ \ 1\ \ 1\ \ 0\ \ 1\ ] \\
A \leftarrow \text{split}\ (A, A{<}1{>}) & = & [\ 4\ \ 5\ \ 1\ \ 2\ \ 2\ \ 7\ \ 3\ \ 7\ ] \\
A{<}2{>} & = & [\ 1\ \ 1\ \ 0\ \ 0\ \ 0\ \ 1\ \ 0\ \ 1\ ]
\end{array}
$$

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	]
$A \leftarrow \text{split}\ (A, A<0>)$	=	[	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	]

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]

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

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]
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]
Pivots	=	[6.4	6.4	6.4	6.4	6.4	6.4	6.4	6.4]
F	=	[=	>	<	<	>	<	>	<]
Key $\leftarrow$ 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]
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]

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]
Pivots	=	[6.4 6.4 6.4 6.4 6.4 6.4 6.4 6.4]
F	=	[= > < < > < > <]
Key $\leftarrow$ 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]
Pivots	=	[3.4 3.4 3.4 3.4 6.4 9.2 9.2 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	]
Pivots	=	[	6.4	6.4	6.4	6.4	6.4	6.4	6.4	6.4	]
F	=	[	=	>	<	<	>	<	>	<	]
Key $\leftarrow$ 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	]
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

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	]
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	]
Pivots	=	[	3.4	3.4	3.4	3.4	6.4	9.2	9.2	9.2	]
F	=	[	=	<	>	=	=	=	<	=	]
Key ← split (Key, F)	=	[	1.6	3.4	4.1	3.4	6.4	8.7	9.2	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
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
Pivots	=	3.4	3.4	3.4	3.4	6.4	9.2	9.2	9.2
F	=	=	<	>	=	=	=	<	=
Key ← split (Key, F)	=	1.6	3.4	4.1	3.4	6.4	8.7	9.2	9.2
Seg-Flags	=	1	1	1	0	1	1	1	1