Scan

Scan / Parallel Prefix Sum

Given
- an array $A = [a_0, a_1, \ldots, a_{n-1}]$
- a binary associative operator $@$ with identity $I$

$scan\ (A) = [I, a_0, (a_0 @ a_1), \ldots, (a_0 @ a_1 @ \ldots @ a_{n-2})]$

This is called exclusive scan
Scan / Parallel Prefix Sum

Given
- an array $A = [a_0, a_1, ..., a_{n-1}]$
- a binary associative operator $@$ with identity $I$

$scan(A) = [a_0, (a_0 @ a_1), ..., (a_0 @ a_1 @ ... @ a_{n-1})]$

This is called inclusive scan

Applications of Scan

- Scan is used as a building block for many parallel algorithms
  - Radix sort
  - Quicksort
  - String comparison
  - Lexical analysis
  - Run-length encoding
  - Histograms
  - Etc.

- See:
Sequential Algorithm

```c
void scan( float* output, float* input, int length) {
    output[0] = 0;
    for (int j = 1; j < length; ++j) {
        output[j] = input[j-1] + output[j-1];
    }
}
```

Naïve Parallel Algorithm

```c
def d := 1 to log_2n do
    for all k in parallel do
        if k >= 2^d then x[k] := x[k - 2^(d-1)] + x[k]
```

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>4</td>
<td>1</td>
<td>6</td>
<td>3</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>4</td>
<td>1</td>
<td>6</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>4</td>
<td>8</td>
<td>7</td>
<td>4</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>4</td>
<td>11</td>
<td>11</td>
<td>12</td>
<td>12</td>
<td>11</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>4</td>
<td>11</td>
<td>11</td>
<td>15</td>
<td>16</td>
<td>22</td>
</tr>
</tbody>
</table>
```
Naïve Parallel Algorithm

for $d := 1$ to $\log_2 n$ do
  forall $k$ in parallel do
    if $k \geq 2^d$ then $x[k] := x[k - 2^{d-1}] + x[k]$

0  1  2  3  4  5  6  7
0  1  2  3  4  5  6  7
d = 1, $2^{d-1} = 1$

0  3  1  7  0  4  1  6
0  3  1  7  0  4  1  6
d = 2, $2^{d-1} = 2$

0  3  4  8  7  4  5  7
0  3  4  8  7  4  5  7
d = 3, $2^{d-1} = 4$

0  3  4  8  7  4  5  7
0  3  4  8  7  4  5  7

0  1  2  3  4  5  6  7
0  1  2  3  4  5  6  7

0  3  4  8  7  4  5  7
0  3  4  8  7  4  5  7

0  3  4  8  7  4  5  7
0  3  4  8  7  4  5  7

0  1  2  3  4  5  6  7
0  1  2  3  4  5  6  7
Need Double-Buffering

- First all read
- Then all write

Solution
- Use two arrays:
  - Input & Output
- Alternate at each step

Double Buffering
Naïve Kernel in CUDA

```c
__global__ void
scan_naive(float *g_odata, float *g_idata, int n) {
    extern __shared__ float temp[];
    int thid = threadIdx.x, pout = 0, pin = 1;
    temp[pout*n + thid] = (thid > 0) ? g_idata[thid-1] : 0;
    for (int dd = 1; dd < n; dd *= 2) {
        pout = 1 - pout; pin = 1 - pin;
        int basein = pin * n, baseout = pout * n;
        syncthreads();
        temp[baseout +thid] = temp[basein +thid];
        if (thid >= dd)
            temp[baseout +thid] += temp[basein +thid - dd];
    }
    syncthreads();
    g_odata[thid] = temp[baseout +thid];
}
```

Analysis of naïve kernel

- This scan algorithm executes $\log(n)$ parallel iterations
  - The steps do n-1, n-2, n-4,... n/2 adds each
  - Total adds: $O(n\log(n))$

- This scan algorithm is NOT work efficient
  - Sequential scan algorithm does $n$ adds
Improving Work Efficiency

- A parallel algorithm based on *Balanced Trees*
  - Build balanced binary tree on input data and sweep to and from the root
  - Tree is conceptual, not an actual data structure

- For scan:
  - Traverse from leaves to root building partial sums at internal nodes
    - Root holds sum of all leaves
  - Traverse from root to leaves building the scan from the partial sums

- Algorithm originally described by Blelloch (1990)

Balanced Tree-Based Scan: Up-Sweep

Assume array is already in shared memory
Balanced Tree-Based Scan: Up-Sweep

Iteration 1, $n/2$ threads

Each thread adds value stride elements away to its own value

Iteration 2, $n/4$ threads

Each thread adds value stride elements away to its own value
Balanced Tree-Based Scan: Up-Sweep

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

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

---

Balanced Tree-Based Scan: Up-Sweep

We now have an array of partial sums. Since this is an exclusive scan, set the last element to zero. It will propagate back to the first element.
Balanced Tree-Based Scan: Down-Sweep

Iterate log(n) times. Each thread adds value stride elements away to its own value, and sets the value stride elements away to its own previous value.

Each ♦ corresponds to a single thread.

Balanced Tree-Based Scan: Down-Sweep

Iterate log(n) times. Each thread adds value stride elements away to its own value, and sets the value stride elements away to its own previous value.

Each ♦ corresponds to a single thread.
Balanced Tree-Based Scan: Down-Sweep

Done! We now have a completed scan that we can write out to device memory.

Total steps: \(2 \times \log(n)\).
Total work: \(2 \times (n-1)\) adds = \(O(n)\) \hspace{1cm} Work Efficient!

Up-Sweep

Additional explanatory slides for scan by A. Moshovos
Up-Sweep

Nodes without value are places where intermediate results are written to (possibly many times)

Additional explanatory slides for scan by A. Moshovos

Down-Sweep

Edge directions are reversed.
Down-Sweep

recursive subtree structure

must be added to the entire right sub-tree
Down-Sweep

must be added to the entire right sub-tree

must be pushed down the left sub-tree

GPU Programming, Shiraz University, Winter 88/Spring 89, Reza Azimi
Down-Sweep

Up-Sweep Pseudo-Code

\[
\text{for } d := 0 \text{ to } \log_2 n - 1 \text{ do } \\
\text{ for } k \text{ from } 0 \text{ to } n - 1 \text{ by } 2^{d+1} \text{ in parallel do } \\
x[k + 2^d + 1 - 1] := x[k + 2^d - 1] + x[k + 2^{d+1} - 1]
\]
Down-Sweep Pseudo-Code

\[
x[n-1] := 0 \\
\text{for } d := \log_2 n \text{ down to } 0 \text{ do} \\
\text{for } k \text{ from } 0 \text{ to } n-1 \text{ by } 2^{d+1} \text{ in parallel do} \\
t := x[k+2^d-1] \\
x[k+2^d-1] := x[k+2^d-1] + t \\
x[k+2^{d+1}-1] := x[k+2^{d+1}-1] + t \\
\]

CUDA Implementation

```cpp
__global__ void scan(float *g_odata, float *g_idata, int n) {
    extern __shared__ float temp[];
    int thid = threadIdx.x;
    int offset = 1;
    // load input into shared memory
    temp[2*thid] = g_idata[2*thid];
    temp[2*thid+1] = g_idata[2*thid+1];
    ...
```
CUDA Implementation: Up Sweep

```c
// up sweep: essentially a reduction
for (int d = 1; d < (n >> 1); d <<= 1) {
    int index = 2 * d * thid;
    if (index < (n >> 1)) {
        // note that the result is supposed
        // to be stored in the last element
        // of the array
        temp[index + d] += temp[index];
    }
    __syncthreads();
}
```

CUDA Implementation: Down-Sweep

```c
// clear the last element
if (thid == 0) { temp[n - 1] = 0; }

// traverse down tree & build scan
for (int d = (n >> 1); d > 0; d >>= 1) {
    int index = 2 * d * thid;
    if (index < (n >> 1)) {
        float t = temp[index + d];
        temp[index + d] += temp[index];
        temp[index] = t;
    }
    __syncthreads();
}
```
CUDA Implementation: Copy Back

```c
__syncthreads();

// all threads write results to global memory
g_odata[2*thid] = temp[2*thid];
g_odata[2*thid+1] = temp[2*thid+1];

} // end of the scan kernel
```

Bank Conflicts

- Current scan implementation has many shared memory bank conflicts
  - These really hurt performance on hardware

- Occur when multiple threads access the same shared memory bank with different addresses

- No penalty if all threads access different banks
  - Or if all threads access exact same address

- Access costs $2^M$ cycles if there is a conflict
  - Where $M$ is max number of threads accessing single bank
Loading from Global Memory to Shared

- Original code interleaves loads:

  ```
  temp[2*thid] = g_idata[2*thid];
  temp[2*thid+1] = g_idata[2*thid+1];
  ```

- Threads: (0, 1, 2, ..., 8, 9, 10, ...) →
  - banks: (0, 2, 4, ..., 0, 2, 4, ...)
  - banks: (1, 3, 5, ..., 1, 3, 5, ...)

- Better to load one element from each half of the array

  ```
  temp[thid] = g_idata[thid];
  temp[thid + (n/2)] = g_idata[thid + (n/2)];
  ```

---

Bank Conflicts in Up-Sweep

First iteration: 2 threads access each of 8 banks.

Each ✈️ corresponds to a single thread.

Like-colored arrows represent simultaneous memory accesses.
## Bank Conflicts in Up-Sweep

<table>
<thead>
<tr>
<th>Bank:</th>
<th>t0</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
</tbody>
</table>

2nd iteration: 4 threads access each of 4 banks.

Each green arrow corresponds to a single thread. Like-colored arrows represent simultaneous memory accesses.

---

## Using Padding to Prevent Conflicts

- Just add a word of padding every 16 words:

<table>
<thead>
<tr>
<th>Bank:</th>
<th>t0</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
<th>t5</th>
<th>t6</th>
<th>t7</th>
<th>t8</th>
<th>t9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
</tr>
</tbody>
</table>

GPU Programming, Shiraz University, Winter 88/Spring 89, Reza Azimi
Using Padding to Remove Conflicts

- Add padding

```c
const int LOG_NUM_BANKS = 4;
...
address += (address >> LOG_NUM_BANKS);
```

- This removes most bank conflicts (why?)
  - Not all, in the case of deep trees

Padding in Up-Sweep

```c
// up sweep: essentially a reduction
for (int d = 1; d < (n >> 1); d <<=1) {
    int index = 2 * d * thid;
    if (index < (n >> 1)) {
        // note that the result is supposed
        // to be stored in the last element
        // of the array
        int addr1 = index;
        int addr2 = index + d;
        addr1 += addr1 >> LOG_NUM_BANKS;
        addr2 += addr2 >> LOG_NUM_BANKS;
        temp[addr2] += temp[addr1];
    }
    __syncthreads();
}
...
Large Arrays

- So far:
  - Array can be processed by a block
    - 1024 elements

- Larger arrays?
  - Divide into blocks
  - Scan each with a block of threads
  - Produce partial scans
  - Scan the partial scans
  - Add the corresponding scan result back to all elements of each block

- See Scan Large Array in the NVIDIA_CUDA_SDK
Application: Stream Compaction

Input: we want to preserve the gray elements
Set a “1” in each gray input

Scan

Scatter input to output, using scan result as scatter address

1M elements: ~0.6-1.3ms
16M elements: ~8-20ms

Performance depends on # elements retained