Lecture 11
Parallel Computation Patterns – Parallel Prefix Sum (Scan)
Objective

• To master parallel Prefix Sum (Scan) algorithms
  – frequently used for parallel work assignment and resource allocation
  – A key primitive to in many parallel algorithms to convert serial computation into parallel computation
  – Based on reduction tree and reverse reduction tree

• Reading – Efficient Parallel Scan Algorithms for GPUs
(Inclusive) Prefix-Sum (Scan) Definition

**Definition:** The all-prefix-sums operation takes a binary associative operator $\oplus$, and an array of $n$ elements

$$[x_0, x_1, \ldots, x_{n-1}],$$

and returns the array

$$[x_0, (x_0 \oplus x_1), \ldots, (x_0 \oplus x_1 \oplus \ldots \oplus x_{n-1})].$$

**Example:** If $\oplus$ is addition, then the all-prefix-sums operation on the array

$$[3 \ 1 \ 7 \ 0 \ 4 \ 1 \ 6 \ 3],$$

would return

$$[3 \ 4 \ 11 \ 11 \ 15 \ 16 \ 22 \ 25].$$
A Inclusive Scan Application Example

• Assume that we have a 100-inch sausage to feed 10
• We know how much each person wants in inches
  – [3 5 2 7 28 4 3 0 8 1]
• How do we cut the sausage quickly?
• How much will be left

• Method 1: cut the sections sequentially: 3 inches first, 5 inches second, 2 inches third, etc.
• Method 2: calculate Prefix scan
  – [3, 8, 10, 17, 45, 49, 52, 52, 60, 61] (39 inches left)
Typical Applications of Scan

- Scan is a simple and useful parallel building block
  - Convert recurrences from sequential:
    ```c
    for(j=1; j<n; j++)
        out[j] = out[j-1] + f(j);
    ```
  - into parallel:
    ```c
    forall(j) { temp[j] = f(j) };
    scan(out, temp);
    ```

- Useful for many parallel algorithms:
  - radix sort
  - quicksort
  - String comparison
  - Lexical analysis
  - Stream compaction
  - Polynomial evaluation
  - Solving recurrences
  - Tree operations
  - Histograms
  - Etc.
Other Applications

• Assigning camp slots
• Assigning farmer market space
• Allocating memory to parallel threads
• Allocating memory buffer for communication channels
• …
An Inclusive Sequential Scan

Given a sequence \([x_0, x_1, x_2, \ldots]\)

Calculate output \([y_0, y_1, y_2, \ldots]\)

Such that
\[
\begin{align*}
y_0 &= x_0 \\
y_1 &= x_0 + x_1 \\
y_2 &= x_0 + x_1 + x_2 \\
&\ldots
\end{align*}
\]

Using a recursive definition
\[
y_i = y_{i-1} + x_i
\]
A Work Efficient C Implementation

\[
y[0] = x[0];
\]
\[
\text{for } (i = 1; i < \text{Max}_i; i++) \ y[i] = y \ [i-1] + x[i];
\]

Computationally efficient:

\[
\text{N additions needed for N elements} - \mathcal{O}(N)!
\]
How do we do this in parallel?

• What is the relationship between Parallel Scan and Reduction?
  – Multiple reduction operations!
  – What if we implement it that way?

• How many operations?
  – Each reduction tree is $O(n)$ operations
  – Reduction trees of size $n$, $n-1$, $n-2$, $n-3$, … 1
  – Very work inefficient! Important concept
A Slightly Better Parallel Inclusive Scan Algorithm

1. Read input from device memory to shared memory

Each thread reads one value from the input array in device memory into shared memory array T0. Thread 0 writes 0 into shared memory array.

| T0 | 3 | 1 | 7 | 0 | 4 | 1 | 6 | 3 |
A Slightly Better Parallel Scan Algorithm

1. (previous slide)

2. Iterate $\log(n)$ times: Threads $stride$ to $n$: Add pairs of elements $stride$ elements apart. Double $stride$ at each iteration. (note must double buffer shared mem arrays)

- Active threads: $stride$ to $n-1$ ($n-stride$ threads)
- Thread $j$ adds elements $j$ and $j-stride$ from T0 and writes result into shared memory buffer T1 (ping-pong)
A Slightly Better Parallel Scan Algorithm

1. Read input from device memory to shared memory.

2. Iterate log(n) times: Threads stride to n: Add pairs of elements stride elements apart. Double stride at each iteration. (note must double buffer shared mem arrays)

Iteration #2
Stride = 2
A Slightly Better Parallel Scan Algorithm

1. Read input from device memory to shared memory. Set first element to zero and shift others right by one.

2. Iterate log(n) times: Threads \textit{stride} to \textit{n}: Add pairs of elements \textit{stride} elements apart. Double \textit{stride} at each iteration. (note must double buffer shared memory arrays)

3. Write output from shared memory to device memory

<table>
<thead>
<tr>
<th>T0</th>
<th>3</th>
<th>1</th>
<th>7</th>
<th>0</th>
<th>4</th>
<th>1</th>
<th>6</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>3</td>
<td>4</td>
<td>8</td>
<td>7</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>9</td>
</tr>
</tbody>
</table>

Iteration #3
Stride = 4

<table>
<thead>
<tr>
<th>T1</th>
<th>3</th>
<th>4</th>
<th>11</th>
<th>11</th>
<th>12</th>
<th>12</th>
<th>11</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td>3</td>
<td>4</td>
<td>11</td>
<td>11</td>
<td>12</td>
<td>12</td>
<td>11</td>
<td>14</td>
</tr>
</tbody>
</table>
Handling Dependencies

- During every iteration, each thread can overwrite the input of another thread
  - Need barrier synchronization to ensure all inputs have been properly generated
  - All threads secure input operand that can be overwritten by another thread
  - Barrier synchronization to ensure that threads have secured their inputs
  - All threads perform addition to write output
Work inefficient scan kernel

__shared__ float XY[SECTION_SIZE];
int i = blockIdx.x * blockDim.x + threadIdx.x;

//load into shared memory
if (i < InputSize) { XY[threadIdx.x] = X[i];}

//perform iterative scan on XY
for (unsigned int stride = 1; stride <= threadIdx.x; stride *=2) {
    __syncthreads();
    float in1 = XY[threadIdx.x – stride];
    __syncthreads();
    XY[threadIdx.x]+=in1;
}
Work Efficiency Considerations

• The first-attempt Scan executes log(n) parallel iterations
  – The steps do (n-1), (n-2), (n-4),..(n- n/2) adds each
  – Total adds: n * log(n) - (n-1) \( \rightarrow \) \( O(n*\log(n)) \) work

• This scan algorithm is not very work efficient
  – Sequential scan algorithm does \( n \) adds
  – A factor of \( \log(n) \) hurts: 20x for \( 10^6 \) elements!

• A parallel algorithm can be slow when execution resources are saturated due to low work efficiency
Improving Efficiency

• A common parallel algorithm pattern:

  *Balanced Trees*

  – Build a balanced binary tree on the input data and sweep it to and from the root
  – Tree is not an actual data structure, but a concept to determine what each thread does at each step

• For scan:
  – Traverse down from leaves to root building partial sums at internal nodes in the tree
    • Root holds sum of all leaves
  – Traverse back up the tree building the scan from the partial sums
Parallel Scan - Reduction Step

\[
\begin{align*}
\sum x_{0..1} &+ \sum x_{2..3} + \sum x_{4..5} + \sum x_{6..7} \\
\sum x_{0..3} &+ \sum x_{4..7} \\
\sum x_{0..7} &
\end{align*}
\]

In place calculation

Final value after reduce
Reduction Step Kernel Code

// scan_array[BLOCK_SIZE*2] is in shared memory

for(int stride=1; stride<= BLOCK_SIZE; stride *=2) {
    int index = (threadIdx.x+1)*stride*2 - 1;
    if(index < 2*BLOCK_SIZE) {
        scan_array[index] += scan_array[index-stride];
        stride = stride*2;
    }
    __syncthreads();
}
Inclusive Post Scan Step

Move (add) a critical value to a central location where it is needed.
Inclusive Post Scan Step

\[ x_0 + \sum_{0..1} x_1 + x_2 + \sum_{0..3} x_3 + x_4 + \sum_{4..5} x_5 + x_6 + \sum_{0..7} x_7 \]

\[ \sum_{0..2} x_2 + \sum_{0..4} x_4 + \sum_{0..6} x_6 \]

\[ + \]

\[ + \]

\[ + \]

\[ + \]
Putting it Together
ANY MORE QUESTIONS?