CS/EE 217 GPU Architecture and Parallel Programming

Lecture 12
Parallel Computation Patterns – Parallel Prefix Sum (Scan) Part-2
Recall: a Slightly Better Parallel Inclusive Scan Algorithm

| T0 | 3 | 1 | 7 | 0 | 4 | 1 | 6 | 3 |

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.
### A Slightly Better Parallel Scan Algorithm

<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>
</table>

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)

**Stride 1**

<table>
<thead>
<tr>
<th>T1</th>
<th>3</th>
<th>4</th>
<th>8</th>
<th>7</th>
<th>4</th>
<th>5</th>
<th>7</th>
<th>9</th>
</tr>
</thead>
</table>

**Iteration #1**

- Active threads: *stride* to \( n-1 \) (\( n\text{-}stride \) threads)
- Thread \( j \) adds elements \( j \) and \( j\text{-}stride \) 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 \(\text{stride}\) to \(n\): Add pairs of elements \(\text{stride}\) elements apart. Double \(\text{stride}\) at each iteration. (note must double buffer shared mem arrays)

---

<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>
<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>

Iteration #2
Stride = 2
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 \(\text{stride}\) to \(n\): Add pairs of elements \(\text{stride}\) elements apart. Double \(\text{stride}\) at each iteration. (note must double buffer shared memory arrays)

3. Write output from shared memory to device memory
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

\[ x_0 \quad x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6 \quad x_7 \]

\[ \sum x_{0..1} \quad \sum x_{2..3} \quad \sum x_{4..5} \quad \sum x_{6..7} \]

In place calculation

Final value after reduce

Time

\[ \sum x_{0..7} \]
Inclusive Post Scan Step

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

\[ x_0 + \sum_{x_0..x_1} x_2 + \sum_{x_0..x_3} x_4 + \sum_{x_0..x_5} x_6 + \sum_{x_0..x_7} \]

\[ \sum_{x_0..x_2} + \sum_{x_0..x_4} + \sum_{x_0..x_6} + \sum_{x_0..x_5} \]
Putting it Together
Reduction Step Kernel Code

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

for(int stride=1; stride <= BLOCK_SIZE; stride=stride*2)
{
    int index = (threadIdx.x+1)*stride*2 - 1;
    if(index < 2*BLOCK_SIZE)
        XY[index] += XY[index-stride];
    stride = stride*2;

    __syncthreads();
}

threadIdx.x+1 = 1, 2, 3, 4....
stride = 1, index =
Putting it together
Post Scan Step

for(int stride=BLOCK_SIZE/2; stride > 0; stride /= 2) {
    __syncthreads();
    int index = (threadIdx.x+1)*stride*2 - 1;
    if(index + stride < 2*BLOCK_SIZE) {
        XY[index+stride] += XY[index];
    }
    stride = stride / 2;
}
__syncthreads();
if(I < InputSize) Y[i] = XY[threadIdx.x];
Work Analysis

• Work efficient kernel executes log(n) iterations in reduction step
  – Identical to reduction; O(n) operations.

• log(n)-1 iterations in post reduction reverse step
  – 2-1, 4-1, 8-1, … n/2 -1 operations in each
  – Total? (n-1) – log(n) or O(n) work

• Both perform no more than 2*(n-1) adds

• Is this ok? What needs to happen for the parallel implementation to be better than sequential?
Some Tradeoffs

• Work efficient kernel is normally superior
  – Better energy efficiency (why?)
  – Less execution resource requirements

• However, the work inefficient kernel could be better under some special circumstances
  – What needs to happen for that?
  – Small lists where there are sufficient execution resources
(Exclusive) 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 $[0, x_0, (x_0 \oplus x_1), \ldots, (x_0 \oplus x_1 \oplus \ldots \oplus x_{n-2})]$.

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

• To find the beginning address of allocated buffers

• Inclusive and Exclusive scans can be easily derived from each other; it is a matter of convenience

\[
\begin{align*}
\text{Exclusive} & \quad [0 \ 3 \ 4 \ 11 \ 11 \ 15 \ 16 \ 22] \\
\text{Inclusive} & \quad [3 \ 4 \ 11 \ 11 \ 15 \ 16 \ 22 \ 25]
\end{align*}
\]
Simple exclusive scan kernel

• Adapt work inefficient scan kernel

• Block 0:
  – Thread 0 loads 0 in XY[0]
  – Other threads load X[threadIdx.x-1] into XY[threadIdx.x]

• All other blocks:
  – Load X[blockIdx.x*blockDim.x+threadIdx.x-1] into XY[threadIdx.x]

• Similar adaptation for work efficient kernel, except each thread loads two values – only one zero should be loaded
Alternative (read Harris Article)

• Uses add-move operation pairs
• Similar in complexity to the work efficient algorithm
• We’ll quickly overview
An Exclusive Post Scan Step
(Add-move Operation)
Exclusive Post Scan Step

\[
\begin{align*}
\Sigma x_{0..1} & \quad x_2 & \quad \Sigma x_{0..3} & \quad x_4 & \quad \Sigma x_{4..5} & \quad x_6 & \quad 0 \\
\Sigma x_{0..3} & \quad \Sigma x_{0..1} & \quad \Sigma x_{0..2} & \quad \Sigma x_{0..3} & \quad \Sigma x_{0..4} & \quad \Sigma x_{0..5} & \quad \Sigma x_{0..6}
\end{align*}
\]
Exclusive Scan Example – Reduction Step

Assume array is already in shared memory
Reduction Step (cont.)

Stride 1

Iteration 1, \( n/2 \) threads

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

Each \( \oplus \) corresponds to a single thread.
Reduction Step (cont.)

Stride 1

T 3 1 7 0 4 1 6 3

Stride 2

T 3 4 7 7 4 5 6 9

Iteration 2, \( n/4 \) threads

T 3 4 7 11 4 5 6 14

Each \( \bigcirc \) corresponds to a single thread.

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

<table>
<thead>
<tr>
<th>T</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>
</table>

**Stride 1**

<table>
<thead>
<tr>
<th>T</th>
<th>3</th>
<th>4</th>
<th>7</th>
<th>7</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>9</th>
</tr>
</thead>
</table>

**Stride 2**

<table>
<thead>
<tr>
<th>T</th>
<th>3</th>
<th>4</th>
<th>7</th>
<th>11</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>14</th>
</tr>
</thead>
</table>

**Stride 4**

<table>
<thead>
<tr>
<th>T</th>
<th>3</th>
<th>4</th>
<th>7</th>
<th>11</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>25</th>
</tr>
</thead>
</table>

Iteration log(n), 1 thread

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.

Each ✿ corresponds to a single thread.
Zero the Last Element

| T | 3 | 4 | 7 | 11 | 4 | 5 | 6 | 0 |

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.
Post Scan Step from Partial Sums

| T | 3 | 4 | 7 | 11 | 4 | 5 | 6 | 0 |
Iterate log(n) times. Each thread adds value \textit{stride} elements away to its own value, and sets the value \textit{stride} elements away to its own \textit{previous} value.
Iterate log(n) times. Each thread adds value \textit{stride} elements away to its own value, and sets the value \textit{stride} elements away to its own \textit{previous} value.

Each \(\bigcirc\) corresponds to a single thread.
Post Scan Step From Partial Sums (cont.)

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 = $O(n)$  
Work Efficient!
Work Analysis

• The parallel Inclusive Scan executes $2^* \log(n)$ parallel iterations
  – $\log(n)$ in reduction and $\log(n)$ in post scan
  – The iterations do $n/2$, $n/4$,..1, 1, …., $n/4$. $n/2$ adds
  – Total adds: $2^* (n-1) \Rightarrow O(n)$ work

• The total number of adds is no more than twice of that done in the efficient sequential algorithm
  – The benefit of parallelism can easily overcome the 2X work when there is sufficient hardware
Working on Arbitrary Length Input

• Build on the scan kernel that handles up to $2 \times \text{blockDim.x}$ elements
• Have each section of $2 \times \text{blockDim}$ elements assigned to a block
• Have each block write the sum of its section into a Sum array indexed by blockIdx.x
• Run parallel scan on the Sum array
  – May need to break down Sum into multiple sections if it is too big for a block
• Add the scanned Sum array values to the elements of corresponding sections
Overall Flow of Complete Scan

Initial Array of Arbitrary Values

Scan Block 0 → Scan Block 1 → Scan Block 2 → Scan Block 3

Store Block Sum to Auxiliary Array

Scan Block Sums

Add Scanned Block Sum i to All Values of Scanned Block i + 1

Final Array of Scanned Values
ANY MORE QUESTIONS?  
READ CHAPTER 9