Introduction to CUDA Programming

Reductions and Prefix Sums
REDUCTIONS
Partition and Summarize

A commonly used strategy for processing large input data sets

- There is no required order of processing elements in a data set (associative and commutative)
- Partition the data set into smaller chunks
- Have each thread to process a chunk
- Use a reduction tree to summarize the results from each chunk into the final answer

We will focus on the reduction tree step for now.

Google and Hadoop MapReduce frameworks are examples of this pattern
Reduction enables other techniques

Reduction is also needed to clean up after some commonly used parallelizing transformations

Privatization
- Multiple threads write into an output location
- Replicate the output location so that each thread has a private output location
- Use a reduction tree to combine the values of private locations into the original output location
What is a reduction computation

Summarize a set of input values into one value using a “reduction operation”

- Max
- Min
- Sum
- Product
- Often with user defined reduction operation function as long as the operation
  - Is associative and commutative
  - Has a well-defined identity value (e.g., 0 for sum)
A sequential reduction algorithm performs $N$ operations

- Initialize the result as an identity value for the reduction operation
  - Smallest possible value for max reduction
  - Largest possible value for min reduction
  - 0 for sum reduction
  - 1 for product reduction

- Scan through the input and perform the reduction operation between the result value and the current input value
A parallel reduction performs $N-1$ Operations in $\log(N)$ steps.
A Quick Analysis

For N input values, the reduction tree performs

- \((1/2)N + (1/4)N + (1/8)N + \ldots (1/N) = (1 - (1/N))N = N - 1\) operations
- In Log (N) steps – 1,000,000 input values take 20 steps
  - Assuming that we have enough execution resources
- Average Parallelism \((N - 1)/\text{Log}(N))\)
  - For \(N = 1,000,000\), average parallelism is 50,000
  - However, peak resource requirement is 500,000!

This is a work-efficient parallel algorithm

- The amount of work done is comparable to sequential
- Many parallel algorithms are not work efficient
Parallel implementation:

- Recursively halve # of threads, add two values per thread in each step
- Takes log(n) steps for n elements, requires n/2 threads

Assume an in-place reduction using shared memory

- The original vector is in device global memory
- The shared memory is used to hold a partial sum vector
- Each step brings the partial sum vector closer to the sum
- The final sum will be in element 0
- Reduces global memory traffic due to partial sum values
### Vector Reduction with Branch Divergence

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
<th>Thread 4</th>
<th>Thread 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>0+1</td>
<td>2+3</td>
<td>4+5</td>
<td>6+7</td>
<td>8+9</td>
<td>10+11</td>
</tr>
<tr>
<td>0...3</td>
<td>4..7</td>
<td>8..11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0..7</td>
<td></td>
<td></td>
<td>8..15</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Introduction to CUDA Programming**
A Sum Example

Thread 0  Thread 1  Thread 2  Thread 3

Data: 3 1 7 0 4 1 6 3

1 4 7 5 9

2 11 14

3 25

steps

Active Partial Sum elements
Simple Thread Index to Data Mapping

- Each thread is responsible of an even-index location of the partial sum vector
  - One input is the location of responsibility

- After each step, half of the threads are no longer needed

- In each step, one of the inputs comes from an increasing distance away
A Simple Thread Block Design

- Each thread block takes 2 * blockDim input elements
- Each thread loads 2 elements into shared memory

```c
__global__ void Reduction(float *input)
{
    __shared__ float partialSum[2 * BLOCK_SIZE];

    unsigned int t = threadIdx.x;
    unsigned int start = 2 * blockIdx.x * blockDim.x;
    partialSum[t] = input[start + t];
    partialSum[blockDim + t] = input[start + blockDim.x + t];
}
```
unsigned int stride;
for (stride = 1; stride < blockDim.x; stride *= 2) {
    __syncthreads();
    if (t % stride == 0)
        partialSum[2 * t] += partialSum[2 * t + stride];
}

Why do we need \texttt{syncthreads()}?

\begin{itemize}
\item \texttt{syncthreads()} are needed to ensure that all elements of each version of partial sums have been generated before we proceed to the next step.
\item Why do we not need another \texttt{syncthread()} at the end of the reduction loop?
\end{itemize}
Thread 0 in each thread block write the sum of the thread block in partialSum[0] into a vector indexed by the blockIdx.x.

There can be a large number of such sums if the original vector is very large.
- The host code may iterate and launch another kernel.

If there are only a small number of sums, the host can simply transfer the data back and add them together.
Some Observations

In each iteration, two control flow paths will be sequentially traversed for each warp
- Threads that perform addition and threads that do not
- Threads that do not perform addition still consume execution resources

No more than half of threads will be executing after the first step
- All odd index threads are disabled after first step
- After the 5th step, entire warps in each block will fail the if test, poor resource utilization but no divergence.
  - This can go on for a while, up to 5 more steps \((1024/32=16=2^5)\), where each active warp only has one productive thread until all warps in a block retire
Thread Index Usage Matters

- In some algorithms, one can shift the index usage to improve the divergence behavior
  - Commutative and associative operators

Example - given an array of values, "reduce" them to a single value in parallel
  - Sum reduction: sum of all values in the array
  - Max reduction: maximum of all values in the array
  - ...

- Always compact the partial sums into the first locations in the partialSum[] array
- Keep the active threads consecutive
An Example of 16 threads

Thread 0  Thread 1  Thread 2  Thread 14  Thread 15

0   1   2   3   ...   13  14  15  16  17  18  19
0+16 15+31
unsigned int stride;
for (stride = blockDim.x / 2; stride >= 1; stride >>= 1) {
    __syncthreads();
    if (t < stride)
        partialSum[t] += partialSum[t + stride];
}

- For a 1024 thread block
  - No divergence in the first 5 steps
  - 1024, 512, 256, 128, 64, 32 consecutive threads are active in each step
  - The final 5 steps will still have divergence
PREFIX SUM
(Inclusive) Prefix-Sum (Scan) Definition

**Definition:** *The all-prefix-sums operation takes a binary associative operator $\oplus$, and an array of $n$ elements $[x_0, x_1, …, x_{n-1}]$, and returns the array $[x_0, (x_0 \oplus x_1), …, (x_0 \oplus x_1 \oplus … \oplus x_{n-1})]$.***

**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 people.

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)
Other Applications

- Assigning camp slots
- Assigning farmer market space
- Allocating memory to parallel threads
- Allocating memory buffer for communication channels
- ...

Introduction to CUDA Programming
A Inclusive Sequential Prefix-Sum

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

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

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

Using a recursive definition
\[
y_i = y_{i-1} + x_i
\]
y[0] = x[0];
for (i = 1; i < Max_i; i++)
    y[i] = y [i-1] + x[i];

Computationally efficient:
- N additions needed for N elements - O(N)!
A Naïve Inclusive Parallel Scan

- Assign one thread to calculate each y element
- Have every thread to add up all x elements needed for the y element

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

“Parallel programming is easy as long as you do not care about performance.”
Let's Look at the Reduction Tree Again
Reduction Scan Step

\[ \sum x_0..x_1 \]

\[ \sum x_2..x_3 \]

\[ \sum x_4..x_5 \]

\[ \sum x_6..x_7 \]

Time

In place calculation

Final value after reduce

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

\[
\sum_{0}^{1} x_0 + \sum_{0}^{3} x_2 + \sum_{0}^{5} x_4 + \sum_{0}^{7} x_6
\]
Putting All Together
/* scan_array[BLOCK_SIZE] is in shared memory */

int stride = 1;

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

    stride = stride * 2;
    __syncthreads();
}

threadIdx.x + 1 = 1, 2, 3, 4....

stride = 1, index =
Putting All Together
int stride = BLOCK_SIZE >> 1;

while(stride > 0) {
    int index = (threadIdx.x + 1) * stride * 2 - 1;
    if(index < BLOCK_SIZE) {
        scan_array[index+stride] += scan_array[index];
    }
    stride = stride >> 1;
    __syncthreads();
}
(Exclusive) Prefix-Sum (Scan) Definition

**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*

$$[0, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-2})].$$

**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

| Exclusive | [0 3 4 11 11 15 16 22] |
| Inclusive | [3 4 11 11 15 16 22 25] |
Exclusive Post Scan Step

\[
\begin{align*}
\chi_0 & \quad \sum_{x_0}^{x_1} \\
\sum_{x_0}^{x_3} & \quad \chi_2 \\
\sum_{x_4}^{x_5} & \quad \chi_4 \\
0 & \quad \chi_6 \\
\sum_{x_0}^{x_3} &
\end{align*}
\]
Exclusive Post Scan Step

\[ x_0 \]
\[ \sum x_{0..1} \]
\[ x_2 \]
\[ \sum x_{0..3} \]
\[ x_4 \]
\[ \sum x_{4..5} \]
\[ x_6 \]
\[ 0 \]
Inclusive Post Scan Step

\[ x_0 \quad \sum_{x_0}^{x_1} \quad x_2 \quad \sum_{x_0}^{x_3} \quad x_4 \quad \sum_{x_4}^{x_5} \quad x_6 \quad \sum_{x_0}^{x_7} \]

\[ \sum_{x_0}^{x_2} \quad + \quad \sum_{x_0}^{x_4} \quad + \quad \sum_{x_0}^{x_6} \]
Exclusive Scan Example – Reduction Step

| T | 3 | 1 | 7 | 0 | 4 | 1 | 6 | 3 |

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

Stride 1

|   T   |   3   |   1   |   7   |   0   |   4   |   1   |   6   |   3   |

|   T   |   3   |   4   |   7   |   7   |   4   |   5   |   6   |   9   |

Iteration 1, \( n/2 \) threads

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

Iterate \( \log(n) \) times. Each thread adds value stride elements away to its own value.
Iterate log(n) times. Each thread adds value \textit{stride} elements away to its own value.
Iterate \( \log(n) \) times. Each thread adds value \( \text{stride} \) elements away to its own value.

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

<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>0</th>
</tr>
</thead>
</table>
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 $\oplus$ corresponds to a single thread.
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 \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 $2 \times \log(n)$ 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: $2 \times (n-1) \Rightarrow O(n)$ 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
1. Read input from device memory to shared memory. Set first element to zero and shift others right by one.

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 Plausible Parallel Scan Algorithm

1. (previous slide)

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)

Iteration #1

Stride = 1

\begin{array}{cccccccc}
\text{In} & 3 & 1 & 7 & 0 & 4 & 1 & 6 & 3 \\
\hline
\text{T0} & 0 & 3 & 1 & 7 & 0 & 4 & 1 & 6 \\
\hline
\text{T1} & 0 & 3 & 4 & 8 & 7 & 4 & 5 & 7 \\
\end{array}

• Active threads: \( \text{stride} \) 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 Plausible 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 \( \text{stride} \) to \( n \): Add pairs of elements \( \text{stride} \) elements apart. Double \( \text{stride} \) at each iteration. (note must double buffer shared mem arrays)

**Iteration #2**

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

**Iteration #3**

Stride = 4
A Plausible Parallel Scan Algorithm

<table>
<thead>
<tr>
<th>In</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>T0</td>
<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>
<tr>
<td>Stride 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T1</td>
<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>
<tr>
<td>Stride 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T0</td>
<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>
<tr>
<td>T1</td>
<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>
<tr>
<td>Out</td>
<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>

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 *stride* to *n*: Add pairs of elements *stride* elements apart. Double *stride* at each iteration. (note must double buffer shared mem arrays)

3. Write output to device memory.
The plausible parallel Scan executes log(n) parallel iterations
- The steps do (n-1), (n-2), (n-4),..(n- n/2) adds
- Total adds: n * log(n) + (n-1) \( \Rightarrow O(n\times\log(n)) \) 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
Working on Arbitrary Length Input

- Build on the scan kernel that handles up to $2 \times \text{blockDim}$ elements.
- Have each section of $2 \times \text{blockDim}$ elements assigned to each 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