Logistics

• Midterm: March 11
• Project proposal presentations: March 4
  – Have to be approved by me by March 2
Overview

• Parallel Patterns: Convolution
  – Constant memory
  – Cache
• Parallel Patterns: Reduction Trees
• Parallel Patterns: Parallel Prefix Sum (Scan)
Convolution, Constant Memory and Constant Caching
Convolution

• Array operation where each output is a weighted sum of a collection of neighboring input elements

• Weights are defined in a *mask array* a.k.a. *convolution kernel*
1D Convolution
1D Convolution
1D Convolution - Boundary Condition
Simple Kernel

```c
__global__ void convolution_1d_basic(float *N, float *M,
float *P, int mask_width, int width) {

int i = blockIdx.x*blockDim.x+threadIdx.x;

float Pvalue = 0;
int N_start = i-(mask_width/2);
for( int j=0; j< mask_width; j++){
    if(N_start +j >=0 && N_start+j < width){
        Pvalue += N[N_start+j]*M[j];
    }
}
P[i] = Pvalue;
}
```
### 2D Convolution - Inside Cells

#### Matrix Representation

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

#### Filter Representation

<table>
<thead>
<tr>
<th>M</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

#### Output Representation

<table>
<thead>
<tr>
<th></th>
<th>3</th>
<th>8</th>
<th>15</th>
<th>12</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>9</td>
<td>16</td>
<td>15</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>8</td>
<td>15</td>
<td>16</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>9</td>
<td>20</td>
<td>18</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>6</td>
<td>1</td>
</tr>
</tbody>
</table>

© David Kirk/NVIDIA and Wen-mei W. Hwu
University of Illinois, 2007-2011
Access Pattern for M

• M is referred to as mask (a.k.a. kernel, filter, etc.)
  – Elements of M are called mask (kernel, filter) coefficients
• Calculation of all output P elements need M
• M is not changed during kernel

• Bonus - M elements are accessed in the same order when calculating all P elements

• M is a good candidate for Constant Memory
How to Use Constant Memory

• Host code allocates, initializes variables the same way as any other variables that need to be copied to the device

• Use `cudaMemcpyToSymbol(dest, src, size)` to copy the variable into the device memory
  – Declare `__const__ float M[MASK_WIDTH]` first

• This copy function tells the device that the variable will not be modified by the kernel and can be safely cached
Kernel using Constant Memory

__const__ float Mc[MASK_WIDTH]

__global__ void convolution_1d_basic(float *N,
    float *P, int mask_width, int width){

    int i = blockIdx.x*blockDim.x+threadIdx.x;

    float Pvalue = 0;
    int N_start = i-(mask_width/2);
    for( int j=0; j < mask_width; j++ ){
        if(N_start +j >=0 && N_start+j < width){
            Pvalue += N[N_start+j]*Mc[j];
        }
    }

    P[i] = Pvalue;
}

... 
cudaMemcpyToSymbol(Mc, M, mask_width *sizeof(float));
Using Shared Memory

• Elements of the input vector are used in multiple computations

• Opportunity to use shared memory

• Shared memory tile must be larger than mask!
Using Shared Memory

N_ds in shared memory contains 8 elements

P

Mask_Width is 5

• For Mask_Width = 5, we load 8+5-1 = 12 elements (12 memory loads)
Each Output uses 5 Input Elements

- P[8] uses N[6], N[7], N[8], N[9], N[10]
- P[10] uses N[8], N[9], N[10], N[11], N[12]
- ...
- P[14] uses N[12], N[13], N[14], N[15], N[16]
- P[15] uses N[13], N[14], N[15], N[16], N[17]

Mask_Width is 5
Benefits from Tiling

• \((8+5-1)=12\) elements loaded
• \(8\times5\) global memory accesses replaced by shared memory accesses
• This gives a bandwidth reduction of \(40/12=3.3\)
Benefits from Tiling

• Tile_Width + Mask_Width -1 elements loaded
• Tile_Width * Mask_Width global memory accesses replaced by shared memory access
• This gives a reduction of bandwidth by

\[
\frac{(\text{Tile}_\text{Width} \times \text{Mask}_\text{Width})}{(\text{Tile}_\text{Width} + \text{Mask}_\text{Width} - 1)}
\]
Another Way to Look at Reuse

- $N_{6}$ is used by $P_{8}$ (1X)
- $N_{7}$ is used by $P_{8}$, $P_{9}$ (2X)
- $N_{8}$ is used by $P_{8}$, $P_{9}$, $P_{10}$ (3X)
- $N_{9}$ is used by $P_{8}$, $P_{9}$, $P_{10}$, $P_{11}$ (4X)
- $N_{10}$ is used by $P_{8}$, $P_{9}$, $P_{10}$, $P_{11}$, $P_{12}$ (5X)
- ... (5X)
- $N_{14}$ is used by $P_{12}$, $P_{13}$, $P_{14}$, $P_{15}$ (4X)
- $N_{15}$ is used by $P_{13}$, $P_{14}$, $P_{15}$ (3X)

Mask_Width is 5
Another Way to Look at Reuse

- The total number of global memory accesses (to the \((8+5-1)=12\) N elements) replaced by shared memory accesses is

\[
1 + 2 + 3 + 4 + 5 \times (8-5+1) + 4 + 3 + 2 + 1 \\
= 10 + 20 + 10 \\
= 40
\]

So the reduction is

\[
40/12 = 3.3
\]
Ghost Elements

• For a boundary tile, we load $\text{Tile\_Width} + (\text{Mask\_Width}-1)/2$ elements
  – 10 in our example of Tile\_Width = 8 and Mask\_Width = 5

• Computing boundary elements does not access global memory for ghost cells
  – Total accesses is $3 + 4 + 6 \times 5 = 37$ accesses

The reduction is $37/10 = 3.7$
In General for 1D

- The total number of global memory accesses to the \((\text{Tile\_Width} + \text{Mask\_Width} - 1)\) \(N\) elements replaced by shared memory accesses is

\[
1 + 2 + \ldots + \text{Mask\_Width}-1 + \text{Mask\_Width} \times (\text{Tile\_Width} - \text{Mask\_Width} + 1) + \text{Mask\_Width}-1 + \ldots + 2 + 1
\]

\[
= (\text{Mask\_Width}-1) \times \text{Mask\_Width} + \text{Mask\_Width} \times (\text{Tile\_Width} - \text{Mask\_Width} + 1)
\]

\[
= \text{Mask\_Width} \times (\text{Tile\_Width})
\]
Bandwidth Reduction in 1D

• The reduction is

Mask_Width * (Tile_Width)/(Tile_Width+Mask_Size-1)

<table>
<thead>
<tr>
<th>Tile_Width</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reduction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mask_Width = 5</td>
<td>4.0</td>
<td>4.4</td>
<td>4.7</td>
<td>4.9</td>
<td>4.9</td>
</tr>
<tr>
<td>Reduction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mask_Width = 9</td>
<td>6.0</td>
<td>7.2</td>
<td>8.0</td>
<td>8.5</td>
<td>8.7</td>
</tr>
</tbody>
</table>
2D Output Tiling and Indexing

- Use a thread block to calculate a tile of P
  - Each output tile is of TILE_SIZE for both x and y

\[
\text{col}_o = \text{blockIdx.x} \times \text{TILE_WIDTH} + \text{tx}; \\
\text{row}_o = \text{blockIdx.y} \times \text{TILE_WIDTH} + \text{ty};
\]
Halo Elements

Mask_Width = 5

Input Tile

Output Tile
8x8 Output Tile

• 12X12=144 N elements need to be loaded into shared memory
• The calculation of each P element needs to access 25 N elements
• 8X8X25 = 1600 global memory accesses are converted into shared memory accesses
• A reduction of 1600/144 = 11X
In General for 2D

- 
  \[(\text{Tile\_Width} + \text{Mask\_Width} - 1)^2 \text{ N elements need to be loaded into shared memory}

- The calculation of each P element needs to access \(\text{Mask\_Width}^2\) N elements

- \(\text{Tile\_Width}^2 \times \text{Mask\_Width}^2\) global memory accesses are converted into shared memory accesses

- The reduction is

  \[
  \frac{\text{Tile\_Width}^2 \times \text{Mask\_Width}^2}{(\text{Tile\_Width} + \text{Mask\_Width} - 1)^2}
  \]
Bandwidth Reduction in 2D

- The reduction is

\[ \frac{\text{Tile Width}^2 \times \text{Mask Width}^2}{(\text{Tile Width} + \text{Mask Width} - 1)^2} \]

<table>
<thead>
<tr>
<th>Tile_Width</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reduction</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mask_Width = 5</td>
<td>11.1</td>
<td>16</td>
<td>19.7</td>
<td>22.1</td>
</tr>
<tr>
<td>Reduction</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mask_Width = 9</td>
<td>20.3</td>
<td>36</td>
<td>51.8</td>
<td>64</td>
</tr>
</tbody>
</table>
Programmer View of CUDA Memories (Review)

- Each thread can:
  - Read/write per-thread registers (~1 cycle)
  - Read/write per-block shared memory (~5 cycles)
  - Read/write per-grid global memory (~500 cycles)
  - Read/only per-grid constant memory (~5 cycles with caching)
Memory Hierarchies

• If every time we needed a piece of data, we had to go to main memory to get it, computers would take a lot longer to do anything

• On today’s processors, main memory accesses take hundreds of cycles

• One solution: Caches
Cache

- In order to keep cache fast, it needs to be small, so we cannot fit the entire data set in it.
Cache

- Cache is unit of volatile memory storage

- A cache is an “array” of cache lines

- Cache line can usually hold data from several consecutive memory addresses

- When data is requested from memory, an entire cache line is loaded into the cache, in an attempt to reduce main memory requests
Caches

Some definitions:

– Spatial locality: is when the data elements stored in consecutive memory locations are accessed consecutively

– Temporal locality: is when the same data element is accessed multiple times in short period of time

• Both spatial locality and temporal locality improve the performance of caches
Scratchpad vs. Cache

- Scratchpad (shared memory in CUDA) is another type of temporary storage used to relieve main memory contention.
- In terms of distance from the processor, scratchpad is similar to L1 cache.
- Unlike cache, scratchpad does not necessarily hold a copy of data that is in main memory.
- It requires explicit data transfer instructions, whereas cache does not.
Cache Coherence Protocol

- A mechanism for caches to propagate updates by their local processor to other caches (processors)
CPU and GPU have different caching philosophy

• CPU L1 caches are usually coherent
  – L1 is also replicated for each core
  – Even data that will be changed can be cached in L1
  – Updates to local cache copy invalidate (or less commonly update) copies in other caches
  – Expensive in terms of hardware and disruption of services (cleaning bathrooms at airports..)

• GPU L1 caches are usually incoherent
  – Avoid caching data that will be modified
GPU Cache Coherence

• Current CUDA implementation:
  – Provides coherence by disabling L1 cache after writes
  – There is room for improvement

• Custom implementations
  – Temporal coherence: invalidates cache using synchronized counters without message passing
  – Stall writes to cache blocks until they have been invalidated in other caches
More on Constant Caching

• Each SM has its own L1 cache
  – Low latency, high bandwidth access by all threads

• However, there is no way for threads in one SM to update the L1 cache in other SMs
  – No L1 cache coherence

This is not a problem if a variable is NOT modified by a kernel.
Reduction Trees
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 - O(N)

• 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 tree algorithm performs N-1 Operations in log(N) steps
A tournament is a reduction tree with “max” operation
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 $(N-1)/\log(N))$
    - 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

© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al, University of Illinois, 2007-2011
A Sum Reduction Example

• 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
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
  – …
A Better Strategy

• Always compact the partial sums into the first locations in the partialSum[] array

• Keep the active threads consecutive
An Example of 16 threads

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 14</th>
<th>Thread 15</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>0+16</td>
<td>15+31</td>
</tr>
</tbody>
</table>

© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al, University of Illinois, 2007-2011
A Better Reduction Kernel

for (unsigned int stride = blockDim.x/2;
     stride >= 1; stride >>= 1)
{
    __syncthreads();
    if (t < stride)
        partialSum[t] += partialSum[t+stride];
}
A Quick Analysis

• 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
Parallel Algorithm Overhead

```c
__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];
for (unsigned int stride = blockDim.x/2;
     stride >= 1;  stride >>= 1)
{
    __syncthreads();
    if (t < stride)
        partialSum[t] += partialSum[t+stride];
}
```
Parallel Algorithm Overhead

```c
__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];
for (unsigned int stride = blockDim.x/2;  
     stride >= 1;  stride >>= 1)
{
    __syncthreads();
    if (t < stride)
        partialSum[t] += partialSum[t+stride];
}
```

---

© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al, University of Illinois, 2007-2011
Parallel Execution Overhead

- Although the number of “operations” is $N$, each operation involves much more complex address calculation and intermediate result manipulation.

- If the parallel code is executed on a single-thread hardware, it would be significantly slower than the code based on the original sequential algorithm.
Parallel Prefix Sum (Scan)
Objective

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

• Additional reading - Mark Harris, Parallel Prefix Sum with CUDA
(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 $[3 \ 4 \ 11 \ 11 \ 15 \ 16 \ 22 \ 25]$. 

© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al, University of Illinois, 2007-2011
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)
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

\[y_0 = x_0\]
\[y_1 = x_0 + x_1\]
\[y_2 = x_0 + x_1 + x_2\]

\[\vdots\]

*Using a recursive definition*

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

```c
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

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

- After the i\textsuperscript{th} iteration \( y_i \) contains its final value
Simple Inclusive Parallel Scan

```c
__global__ void work_inefficient_scan_kernel(float *X, float *Y, int InputSize)
{
    __shared__ float XY[SECTION_SIZE];

    int i = blockIdx.x*blockdim.x + threadIdx.x;
    if( i<Inputsize ){
        XY[threadIdx.x] = X[i];
    }

    for(int stride =1; stride <= threadIdx.x; stride *=2)
    {
        __syncthreads();     XY[threadIdx.x] += XY[threadIdx.x-stride];
    }
}
```
Simple Inclusive Parallel Scan
Work Efficiency Considerations

- Total amount of work: \( \sum (N - \text{stride}) \) for stride=1, 2, 4, … , N/2
  - Total logN terms
- Total amount of work: NlogN - (N-1)

- Sequential code: N-1

- For 1024 elements, GPU code performs 9 times more operations

“Parallel programming is easy as long as you do not care about performance.”
Let’s Look at the Reduction Tree Again
Work-Efficient Parallel Scans

- Reuse intermediate results
- Distribute them to different threads

- Reduction tree can generate sum of N numbers in logN steps
- Also generates number of useful sub-sums

- Two step algorithm
  - Reduction scan
  - Partial sum distribution using reverse tree
Reduction Scan Step

\[
\sum x_{0..x_1} + \sum x_{2..x_3} + \sum x_{4..x_5} + \sum x_{6..x_7}
\]

In place calculation

Final value after reduce

Time

© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al, University of Illinois, 2007-2011
Reduction Scan Step

- First step: modify elements at odd indexes
- Second step: modify elements at 4n-1
- Third step: modify elements at 8n-1
- ...
- Total ops: \( N/2 + N/4 + ... = N-1 \)
Reduction Scan Step: Simple Kernel

for(int stride =1; stride <= blockDim.x; stride *=2) 
{
    __syncthreads();
    if((threadIdx.x+1)%(2*stride) ==0){
        XY[threadIdx.x] += XY[threadIdx.x-stride];
    }
}
Reduction Scan Step: Less Divergent Kernel

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

Uses consecutive threads for computation
Inclusive Post Scan Step

Move (add) a critical value to a central location where it is needed.

© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al, University of Illinois, 2007-2011
Inclusive Post Scan Step

• After reduction, \( XY[2^n-1] \) contain final values

• Largest gap between middle and last elements of input
  – Assume \( N \) is power of 2

• Need one addition to produce final value at the midpoint of this gap

• In the next step, largest gap between final values is half the previous gap, etc.
Putting it together

© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al, University of Illinois, 2007-2011
Post Scan Step: Kernel

for(int stride=SECTION_SIZE/4; stride > 0; stride /=2){
    __syncthreads();
    int index = (threadIdx.x+1)*2*stride-1;
    if(index+stride < SECTION_SIZE){
        XY[index+stride] += XY[index];
    }
}
__syncthreads();

Y[i] = XY[threadIdx.x.x];

At each iteration, push the value from a position in XY that is a multiple of stride -1 to a position that is stride away
Efficiency Analysis

• Total operations for post scan step:
  \( \frac{N}{2} + \frac{N}{4} + \ldots + 4 + 2 - 1 < N-2 \)

• Grand total: \( 2N-3 \)

• Compared to:
  – \( N-1 \) for sequential implementation
  – \( N\log N \) for naïve parallel implementation
(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

\[
\begin{array}{cccccccccc}
3 & 1 & 7 & 0 & 4 & 1 & 6 & 3 \\
\end{array}
\]

Exclusive
\[
\begin{array}{cccccccccccc}
0 & 3 & 4 & 11 & 11 & 15 & 16 & 22 \\
\end{array}
\]

Inclusive
\[
\begin{array}{cccccccccccc}
3 & 4 & 11 & 11 & 15 & 16 & 22 & 25 \\
\end{array}
\]
Applications of Scan

• Scan is a simple and useful parallel building block for many parallel algorithms:
  • Radix sort
  • Quicksort
  • String comparison
  • Lexical analysis
  • Stream compaction
  • Run-length encoding
  • Polynomial evaluation
  • Solving recurrences
  • Tree operations
  • Histograms
  • Allocation
  • Etc.

• Scan is unnecessary in sequential computing!
Other Applications

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

© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al, University of Illinois, 2007-2011
Hierarchical Scan