“Life isn't about finding yourself. Life is about creating yourself.”

-- George Bernard Shaw, playwright, Nobel Prize winner
[1856-1950]
Before We Get Started

- Issues covered last time:
  - Further CUDA optimization issues
  - Case study: parallel prefix scan in CUDA

- Today’s topics
  - Case study: parallel prefix scan in CUDA – wrap up, second approach
  - CUDA Share Memory issues
  - Case study: parallel reduction in CUDA

- Assignment:
  - HW06 – due Oct 28 at 11:59 PM
The kernel was very simple, approach easy to understand

$O(N \log_2 N)$ algorithm – significant overhead when $N$ gets large

As is, solution only works when entire array processed by one block
  - One block in CUDA has at the most 1024 threads
  - Assignment: 16 million entries array, more kernel calls in order
Parallel Scan Algorithm: Solution #2
Harris-Sengupta-Owen (2007)

- A common parallel algorithm pattern:
  - **Balanced Trees**
    - Build a balanced binary tree on the input data and sweep it to the root and then back into the leaves
    - Tree is not an actual data structure, but a concept to determine what each thread does at each step

- For scan:
  - Traverse from leaves to root building partial sums at internal nodes in the tree
    - Root holds sum of all leaves → nice, this is a reduction algorithm
  - Traverse the tree back building the scan from the partial sums
    - Called down-sweep phase
Picture and Pseudocode
~ Reduction Step~

\[ j \cdot 2^{k+1} - 1 = \begin{cases} 1 & 3 & 5 & 7 \\ 3 & 7 & -1 & -1 \\ 7 & -1 & -1 & -1 \end{cases} \]

\[ j \cdot 2^{k+1} - 2^k - 1 = \begin{cases} 0 & 2 & 4 & 6 \\ 1 & 5 & -1 & -1 \\ 3 & -1 & -1 & -1 \end{cases} \]

\[ \text{for } k = 0 \text{ to } M-1 \]
\[ \quad \text{offset } = 2^k \]
\[ \quad \text{for } j = 1 \text{ to } 2^M \cdot 2^{k-1} \text{ in parallel do} \]
\[ \quad \quad x[j \cdot 2^{k+1} - 1] = x[j \cdot 2^{k+1} - 1] + x[j \cdot 2^{k+1} - 2^k - 1] \]
\[ \quad \text{end for} \]
\[ \text{end for} \]

NOTE: “-1” entries indicate no-ops
Operation Count, Reduce Phase

By inspection:

\[
\sum_{k=0}^{M-1} 2^{M-k-1} = 2^M - 1 = n - 1
\]

Looks promising...
The Down-Sweep Phase

<table>
<thead>
<tr>
<th>d=0</th>
<th>x₀</th>
<th>Σ(x₀..x₁)</th>
<th>x₂</th>
<th>Σ(x₀..x₃)</th>
<th>x₄</th>
<th>Σ(x₄..x₅)</th>
<th>x₆</th>
<th>Σ(x₀..x₇)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x₀</td>
<td>Σ(x₀..x₁)</td>
<td>x₂</td>
<td>Σ(x₀..x₃)</td>
<td>x₄</td>
<td>Σ(x₄..x₅)</td>
<td>x₆</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>d=1</th>
<th>x₀</th>
<th>Σ(x₀..x₁)</th>
<th>x₂</th>
<th>0</th>
<th>x₄</th>
<th>Σ(x₄..x₅)</th>
<th>x₆</th>
<th>Σ(x₀..x₃)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x₀</td>
<td>Σ(x₀..x₁)</td>
<td>x₂</td>
<td>Σ(x₀..x₃)</td>
<td>x₄</td>
<td>Σ(x₄..x₅)</td>
<td>x₆</td>
<td>Σ(x₀..x₃)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>d=2</th>
<th>x₀</th>
<th>0</th>
<th>x₂</th>
<th>Σ(x₀..x₁)</th>
<th>x₄</th>
<th>Σ(x₄..x₅)</th>
<th>x₆</th>
<th>Σ(x₀..x₅)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x₀</td>
<td>0</td>
<td>x₂</td>
<td>Σ(x₀..x₁)</td>
<td>x₄</td>
<td>Σ(x₄..x₅)</td>
<td>x₆</td>
<td>Σ(x₀..x₅)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>d=3</th>
<th>0</th>
<th>x₀</th>
<th>Σ(x₀..x₁)</th>
<th>Σ(x₀..x₂)</th>
<th>Σ(x₀..x₃)</th>
<th>Σ(x₀..x₄)</th>
<th>Σ(x₀..x₅)</th>
<th>Σ(x₀..x₆)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>x₀</td>
<td>Σ(x₀..x₁)</td>
<td>Σ(x₀..x₂)</td>
<td>Σ(x₀..x₃)</td>
<td>Σ(x₀..x₄)</td>
<td>Σ(x₀..x₅)</td>
<td>Σ(x₀..x₆)</td>
</tr>
</tbody>
</table>

```
for k=M-1 to 0
    offset = 2^k
    for j=1 to 2^{M-k-1} in parallel do
        dummy = x[j \cdot 2^{k+1} - 2^k - 1]
        x[j \cdot 2^{k+1} - 2^k - 1] = x[j \cdot 2^{k+1} - 1]
        x[j \cdot 2^{k+1} - 1] = x[j \cdot 2^{k+1} - 1] + dummy
    endfor
endfor
```

NOTE: This is just a mirror image of the reduction stage. Easy to come up with the indexing scheme…
Down-Sweep Phase, Remarks

- Number of operations for the down-sweep phase:
  - Additions: n-1
  - Swaps: n-1 (each swap shadows an addition)

- Total number of operations associated with this algorithm
  - Additions: 2n-2
  - Swaps (overwrite operations): n-1
  - Looks very comparable with the work load in the sequential solution

- Convoluted algorithm, indexing tricky to figure out
  - Kernel shown on next slide
__global__ void prescan(float *g_odata, float *g_idata, int n)
{
    extern volatile __shared__ float temp[]; // allocated on invocation

    int thid = threadIdx.x;
    int offset = 1;

    temp[2*thid] = g_idata[2*thid]; // load input into shared memory
    temp[2*thid+1] = g_idata[2*thid+1];

    for (int d = n>>1; d > 0; d >>= 1) // build sum in place up the tree
    {
        __syncthreads();
        if (thid < d)
        {
            int ai = offset*(2*thid+1)-1;
            int bi = offset*(2*thid+2)-1;
            temp[bi] += temp[ai];
        }
        offset <<= 1; // multiply by 2 implemented as bitwise operation
    }
    if (thid == 0) { temp[n - 1] = 0; } // clear the last element

    for (int d = 1; d < n; d *= 2) // traverse down tree & build scan
    {
        offset >>= 1;
        __syncthreads();
        if (thid < d)
        {
            int ai = offset*(2*thid+1)-1;
            int bi = offset*(2*thid+2)-1;
            float t = temp[ai];
            temp[ai] = temp[bi];
            temp[bi] += t;
        }
        __syncthreads();
    }
    __syncthreads();
    g_odata[2*thid] = temp[2*thid]; // write results to device memory
    g_odata[2*thid+1] = temp[2*thid+1];
}
Upon first invocation of the kernel (kernel #1), each will bring into shared memory 2048 elements:

- 1024 “lead” elements (see vertical arrows ↑ on slide 6), and…
- 1024 mating elements (the blue, oblique, arrows on slide 6)
- Two consecutive “lead” elements are separated by a stride of \( k=2^1 \)
- A “lead” element and its “mating” element are separated by a stride of \( k/2=1 \)

Suppose you take 6 reduction steps in this first kernel and bail out after writing into the global memory the preliminary data that you computed and stored in shared memory

The next kernel invocation should pick up the unfinished business where the previous kernel left…

- Call this a “flawless reentry requirement”
Upon the second kernel call, each block will bring into shared memory 2048 elements:

- 1024 “lead” elements, and…
- 1024 “mating” elements
- Two consecutive “lead” elements will now be separated by a stride of $k=2^6$
- A “lead” element and its “mating” element are separated by a stride of $k/2=2^5$
  - Thus, when bringing in data from global memory, you are not going to bring over a contiguous chunk of memory of size 2048, rather you’ll have to jump $2^5$ locations between successive “lead and mating element” pairs

However, once you bring data in shared memory, you process as before

Before you exit kernel #2 you have to write back data from shared memory into global memory

- Again, you have to choreograph this shared to global memory store since there is a $2^5$ stride that comes into play

If you exit kernel #2 after say 4 more reduction steps, the next time you re-enter the kernel (#3) you will have $k=2^{10}$
Going Beyond 2048 Entries

[3/3]

- You will continue the reduction stage until the stride is $2^{M-1}$
  - At this point you are ready to start the down-sweep phase
  - Down-sweep phase carried out in a similar fashion: we will have to invoke the kernel several times
  - Always work in shared memory and copy back data to global memory before bailing out

- The challenges here are:
  - Understanding the indexing into the global memory to bring data to ShMem
  - How to loop across the data in shared memory

- Numerous shared memory bank conflicts since strides are powers of 2
  - Shared memory bank conflicts: discussed next
  - Advanced topic: get rid of the bank conflict through padding
Concluding Remarks, Parallel Scan

- Intuitively, the scan operation is not the type of procedure ideally suited for parallel computing
  - Even if it doesn’t fit like a glove, leads to good speedup:

<table>
<thead>
<tr>
<th># elements</th>
<th>CPU Scan (ms)</th>
<th>GPU Scan (ms)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>0.002231</td>
<td>0.079492</td>
<td>0.03</td>
</tr>
<tr>
<td>32768</td>
<td>0.072663</td>
<td>0.106159</td>
<td>0.68</td>
</tr>
<tr>
<td>65536</td>
<td>0.146326</td>
<td>0.137006</td>
<td>1.07</td>
</tr>
<tr>
<td>131072</td>
<td>0.726429</td>
<td>0.200257</td>
<td>3.63</td>
</tr>
<tr>
<td>262144</td>
<td>1.454742</td>
<td>0.326900</td>
<td>4.45</td>
</tr>
<tr>
<td>524288</td>
<td>2.911067</td>
<td>0.624104</td>
<td>4.66</td>
</tr>
<tr>
<td>1048576</td>
<td>5.900097</td>
<td>1.118091</td>
<td>5.28</td>
</tr>
<tr>
<td>2097152</td>
<td>11.848376</td>
<td>2.099666</td>
<td>5.64</td>
</tr>
<tr>
<td>4194304</td>
<td>23.835931</td>
<td>4.062923</td>
<td>5.87</td>
</tr>
<tr>
<td>8388688</td>
<td>47.390906</td>
<td>7.987311</td>
<td>5.93</td>
</tr>
<tr>
<td>16777216</td>
<td>94.794598</td>
<td>15.854781</td>
<td>5.98</td>
</tr>
</tbody>
</table>

Source: 2007 paper of Harris, Sengupta, Owens
Concluding Remarks, Parallel Scan

- Hillis-Steele (HS) solution simple, but suboptimal

- Harris-Sengupta-Owen (HSO) solution convoluted, yet $O(N)$ scaling
  - Algorithm is complex (particularly if implementation avoids bank conflicts)

- Problem not solved yet: we only looked at the case when our array has up to 2048 elements
  - How do we handle the $16,777,216 = 2^{24}$ elements case?
  - Likewise, how would you implement the case when the number of elements is not a power of 2?
Shared Memory Issues
**Shared Memory: Syntax & Semantics**

- You can statically declare shared memory like in the code snippet below:

```c
__global__ void coalescedMultiply(float *a, float* b, float *c, int N)
{
    __shared__ float aTile[TILE_DIM][TILE_DIM];
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0f;
    aTile[threadIdx.y][threadIdx.x] = a[row*TILE_DIM+threadIdx.x];
    for (int i = 0; i < TILE_DIM; i++) {
        sum += aTile[threadIdx.y][i] * b[i*N+col];
    }
    c[row*N+col] = sum;
}
```

- The variable `aTile` visible to all threads in each block, and only to those threads
  - The thread that executes the kernel above sees the `aTile` declaration and understands that all its sibling-threads in the block are going to see it too. They share this variable collectively

- The same thread, when it sees the variable `row` it understands that it has sole ownership of this variable (variable stored in a register)
3 Ways to Set Aside Shared Memory

- First way: Statically, declared inside a kernel
  - See previous slide…

- Second way: Through the execution configuration
  - \textbf{Ns} below indicates size (in bytes) to be allocated in shared memory

```c
__global__ void MyFunc(float*) // __device__ or __global__ function
{
    extern __shared__ float shMemArray[];
    // Size of shMemArray determined through the execution configuration
    // You can use shMemArray as you wish here…
}
```

// invoke like this
MyFunc<<< Dg, Db, \textbf{Ns} >>>(parameter);

- Third way: Dynamically, via CUDA Driver API
  - Advanced feature, uses API function cuFuncSetSharedSize(), not discussed here
Shared Memory Architecture

- Common sense observation: in a parallel machine many threads access memory at the same time
  - To service more than one thread, memory is divided into independent banks
  - This layout essential to achieve high bandwidth

- Each SM has ShMem organized in 32 Memory banks

- Recall that shared memory and L1 cache draw on the same physical memory inside an SM
  - Fermi: this physical memory can be partitioned as
    - 48 KB of ShMem and 16 KB of L1 cache
    - The other way around
  - Note: shared memory can store less data than the registers (48 KB vs. 128 KB)
Shared Memory Architecture

The 32 banks of the Shared Memory are organized like benches in a movie theater:
- You have multiple rows of benches
- Each row has 32 benches which are separated and grouped in long columns
- In each bench you can “seat” a family of four bytes (32 bits total)
- Note that a bank represents a column of benches in the movie theater, which is perpendicular to the screen

Each bank has a bandwidth of 32 bits per two clock cycles
Shared Memory: Transaction Rules & Bank Conflicts

- When reading in four-byte words, 32 threads in a warp attempt to access shared memory simultaneously.

- Bank conflict: the scenario where two different threads access *different* words in the same bank.

- Note that there is no conflict if different threads access any bytes within the same word.

- Bank conflicts enforce the hardware to serialize a ShMem access, which adversely impacts bandwidth.
Shared Memory Bank Conflicts

- **If there are no bank conflicts:**
  - Shared memory access is fast, but not as fast as register access
  - On the bright side, latency is roughly 100x lower than global memory latency

- **Share memory access, the fast case:**
  - If all threads of a warp access different banks, there is no bank conflict
  - If all threads of a warp access an identical address for a fetch operation, there is no bank conflict (broadcast)

- **Share memory access, the slow case:**
  - Worst case: 32 threads access 32 different words in the same bank
  - Must serialize all the accesses
  - In general, cost = max # of simultaneous accesses to a single bank
How Addresses Map to Banks on Fermi

- Successive 32-bit word addresses are assigned to successive banks

- Bank you work with = (address of offset) % 32
  - This is because Fermi has 32 banks
  - Example: 1D shared mem array, `myShMemVar`, of 1024 floats
    - `myShMemVar[4]`: accesses bank #4 (physically, the fifth one – first row)
    - `myShMemVar[31]`: accesses bank #31 (physically, the last one – first row)
    - `myShMemVar[50]`: access bank #18 (physically, the 19th one – second row)
    - `myShMemVar[128]`: access bank #0 (physically, the first one – fifth row)
    - `myShMemVar[178]`: access bank #18 (physically, the 19th one – sixth row)
  - NOTE: If, for instance, the third thread in a warp accesses `myShMemVar[50]` and the eighth thread in the warp access `myShMemVar[178]`, then you have a two-way bank conflict and the two transactions get serialized

- IMPORTANT: There is no such thing as “bank conflicts” between threads belonging to different warps
Bank Addressing Examples
Transactions Involving 4 Byte Words

- No Bank Conflicts
  - Linear addressing stride == 1

- No Bank Conflicts
  - Random 1:1 Permutation
A case in which only the even banks end up being accessed, and the warps access different words in each bank.

A case in which the 32 threads in a warp access only banks 0, 8, 16, and 24 (and different words in these four banks).
Other Examples

- Two “no conflict” scenarios:
  - Broadcast: all threads in a warp access the same word in a bank
  - Multicast: several threads in a warp access the same word in the same bank
Linear Addressing

- Given:
  ```
  __shared__ float sharedM[256];
  float foo = sharedM[baseIndex + s * threadIdx.x];
  ```

- This is bank-conflict-free if $s$ shares no common factors with the number of banks
  - Conclusion: you are fine if $s$ is odd
Data types and bank conflicts

- No conflicts below if `shrd` is a 32-bit data type:
  
  ```
  foo = shrd[baseIndex + threadIdx.x]
  ```

- Also if accessing one byte/thread, no conflict since *different* bytes of the same word are accessed
  
  - No conflicts:
    ```
    extern __shared__ char shrd[];
    foo = shrd[baseIndex + threadIdx.x];
    ```

  - No conflicts:
    ```
    extern __shared__ short shrd[];
    foo = shrd[baseIndex + threadIdx.x];
    ```
Exercise: Is ShMem access below good or bad?

- Each thread loads two floats into shared memory:

```c
int tid = threadIdx.x;
shared[2*tid   ] = global[2*tid   ];
shared[2*tid+1] = global[2*tid+1];
```

- This makes sense for traditional CPU threads, locality in cache line usage and reduced sharing traffic
  - Doesn’t make sense in shared memory usage where there is no cache line effects but banking effects
  - 2-way-interleaved loads result in 2-way bank conflicts

- Adding insult to injury: you don’t have coalesced global memory loads – basically you are halving the device mem bandwidth
A Better Array Access Pattern

- Here's a better way of doing it
  - Each thread loads one element in every consecutive group of blockDim elements.

```c
shared[tid] = global[tid];
shared[tid + blockDim.x] = global[tid + blockDim.x];
```
Shared Memory: Important Fact

- It used to be that any access to Shared Memory was a direct access (in compute capability 1.x)

- Fermi (2.x) has a load/store architecture that can bring data into registers
  - No guarantee for coherence between the shared memory block and the value stored in the register

- Problem is typically addressed by making that shared memory volatile:
  - In 1.x, this was always ok:
    ```c
    __shared__ int myShVars[256];
    ```
  - In 2.x, you might have to do this (prevents compiler from optimizing instructions related to `myShVars`):
    ```c
    volatile __shared__ int myShVars[256];
    ```

More information about shared memory: Programming Guide, Sections 3.2.3, 5.3.2.3, and Appendix F4.3

CUDA Case Study: Parallel Reduction
Parallel Reduction in CUDA

- Exercise draws on material made available by Mark Harris of NVIDIA

- Parallel Reduction: Common and important data parallel primitive
  - Example: Used to compute the norm of a large vector

- Easy to implement in CUDA
  - Challenging to get it to run fast though

- Serves as a good optimization example
  - Walk step by step through several different versions
  - Demonstrates several important optimization strategies
Parallel Reduction

- Basic Idea: tree-based approach used within each thread block

![Tree Diagram]

- Need to be able to use multiple thread blocks
  - Why? To process very large arrays
  - Why? To keep all multiprocessors on the GPU busy
  - How? Each thread block reduces a portion of the array to one single value

- Q: How do we communicate partial results between thread blocks?
Problem: Global Synchronization

- If we could synchronize across all thread blocks, could easily reduce very large arrays, right?
  - Global sync after each block produces its result
  - Once all blocks reach sync, continue recursively

- But CUDA has no global synchronization
  - Only synchronization of threads that belong to the same block

- Solution: decompose into multiple kernels
  - Kernel launch serves as a global synchronization point
  - Kernel launch has negligible SW overhead
Multiple Kernel Calls

[An Example, and how it all works out…]

- Imagine you launch a 1D grid in which each 1D block has 256 threads
- Assume that the number or elements in the array is $N=100,000$
  - Note that $100,000 = 390 \times 256 + 160$, therefore $[N + 255/256] = 391$ blocks needed

- For the first stage, you launch 391 blocks of 256 threads
  - At the end of this stage you still have to operate on 391 elements

- For the second stage, you launch two blocks of 256 threads
  - At the end of this stage you only have to operate on two elements

- For the third and last stage, you launch one block of 32 threads
  - Almost all threads idle…

- NOTE: after the first stage, each subsequent stage operates on a number of entries equal to the number of blocks in the previous stage
Vector Reduction: 30,000 Feet Perspective

- At the block level: Bring data in shared memory, then start adding in parallel
- Fewer and fewer threads of a block participate
- The process is memory bound, low arithmetic intensity…
What is Our Optimization Goal?

- We should strive to reach GPU peak performance
  - Choose the right metric:
    - GFLOP/s: for compute-bound kernels
    - Bandwidth: for memory-bound kernels

- Reductions have very low arithmetic intensity
  - 1 flop per element loaded (bandwidth-optimal)
  - Therefore we should strive for peak bandwidth

- This example uses results generated using a G80 GPU
  - Compute capability (CC) 1.0
  - 384-bit memory interface, 900 MHz DDR
  - \( 384 \times 900 \times 2 / 8 = 86.4 \text{ GB/s} \)
  - Example carries over to other CCs, this algorithm will be memory bound
Parallel Reduction: Interleaved Addressing

Note: in stage $s$, only threads divisible to $2^s$ get to work. Stride: $2^{(s-1)}$
Reduction #1: Interleaved Addressing

```c
__global__ void reduce1(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

    // each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();

    // do reduction in shared mem
    for(unsigned int s=1; s < blockDim.x; s *= 2) {
        if (tid % (2*s) == 0) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }

    // write result for this block to global memory
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Why do we need to sync threads?

Problem: highly divergent warps are very inefficient, and % operator is very slow.
Performance for 4M element reduction

<table>
<thead>
<tr>
<th>Kernel 1: interleaved addressing with divergent branching</th>
<th>Time ($2^{22}$ ints)</th>
<th>Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8.054 ms</td>
<td>2.083 GB/s</td>
</tr>
</tbody>
</table>

Note: Block Size = 128 threads for all tests
Parallel Reduction: Interleaved Addressing

Values (shared memory)

<table>
<thead>
<tr>
<th>Step 1</th>
<th>Thread IDs</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stride $2^0$</td>
<td>[0]</td>
<td>10 1 8 -1 0 -2 3 5 -2 -3 2 7 0 11 0 2</td>
</tr>
<tr>
<td></td>
<td>[1]</td>
<td>1 7 -1 -2 -2 8 5 -5 -3 9 7 11 11 2 2</td>
</tr>
<tr>
<td></td>
<td>[2]</td>
<td>18 1 7 -1 6 -2 8 5 4 -3 9 7 13 11 2 2</td>
</tr>
<tr>
<td></td>
<td>[3]</td>
<td>24 1 7 -1 6 -2 8 5 17 -3 9 7 13 11 2 2</td>
</tr>
<tr>
<td></td>
<td>[4]</td>
<td>41 1 7 -1 6 -2 8 5 17 -3 9 7 13 11 2 2</td>
</tr>
</tbody>
</table>

New Problem: Shared Memory Bank Conflicts
Reduction #2: Interleaved Addressing

Just replace divergent branch in inner loop...

```c
for (unsigned int s=1; s < blockDim.x; s *= 2) {
    if (tid % (2*s) == 0) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

...with strided index and non-divergent branch:

```c
for (unsigned int s=1; s < blockDim.x; s *= 2) {
    int index = 2 * s * tid;
    if (index < blockDim.x) {
        sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
```
# Performance for 4M element reduction

<table>
<thead>
<tr>
<th>Kernel 1:</th>
<th>Time (2^{22} ints)</th>
<th>Bandwidth</th>
<th>Step Speedup</th>
<th>Cumulative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>interleaved addressing with divergent branching</td>
<td>8.054 ms</td>
<td>2.083 GB/s</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Kernel 2:</th>
<th>Time (2^{22} ints)</th>
<th>Bandwidth</th>
<th>Step Speedup</th>
<th>Cumulative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>interleaved addressing with bank conflicts</td>
<td>3.456 ms</td>
<td>4.854 GB/s</td>
<td>2.33x</td>
<td>2.33x</td>
</tr>
</tbody>
</table>
Parallel Reduction: Sequential Addressing

Sequential addressing is Shared Mem conflict free
Reduction #3: Sequential Addressing

Just replace strided indexing in inner loop...

```
for (unsigned int s=1; s < blockDim.x; s *= 2) {
    int index = 2 * s * tid;
    if (index < blockDim.x) {
        sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
```

...with reversed loop and threadID-based indexing:

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
    if (tid < s) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

- First you go 1/2 the block size to match a thread with its second operand
- Then you go 1/4 of the block size
- Then you go 1/8 of the block size, et.
Performance for 4M element reduction

<table>
<thead>
<tr>
<th>Kernel 1: interleaved addressing with divergent branching</th>
<th>Time (2^{22} ints)</th>
<th>Bandwidth</th>
<th>Step Speedup</th>
<th>Cumulative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8.054 ms</td>
<td>2.083 GB/s</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Kernel 2: interleaved addressing with bank conflicts    | 3.456 ms            | 4.854 GB/s| 2.33x        | 2.33x              |

| Kernel 3: sequential addressing                         | 1.722 ms            | 9.741 GB/s| 2.01x        | 4.68x              |
Idle Threads...

Current solution:

```c
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
    if (tid < s) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

Note that half of the threads are idle on first loop iteration! This is wasteful...
Reduction #4: First Add During Load

Replace single load:

```c
// each thread loads one element from global to shared mem
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
sdata[tid] = g_idata[i];
__syncthreads();
```

...With two loads and first add of the reduction:

```c
// perform first level of reduction upon reading from
// global memory and writing to shared memory
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```

One side effect: the number of blocks you need now is half of what it used to be...
## Performance for 4M element reduction

<table>
<thead>
<tr>
<th>Kernel 1:</th>
<th>Time (2^{22} ints)</th>
<th>Bandwidth</th>
<th>Step Speedup</th>
<th>Cumulative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>interleaved addressing with divergent branching</td>
<td>8.054 ms</td>
<td>2.083 GB/s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel 2:</td>
<td>3.456 ms</td>
<td>4.854 GB/s</td>
<td>2.33x</td>
<td>2.33x</td>
</tr>
<tr>
<td>interleaved addressing with bank conflicts</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel 3:</td>
<td>1.722 ms</td>
<td>9.741 GB/s</td>
<td>2.01x</td>
<td>4.68x</td>
</tr>
<tr>
<td>sequential addressing</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel 4:</td>
<td>0.965 ms</td>
<td>17.377 GB/s</td>
<td>1.78x</td>
<td>8.34x</td>
</tr>
<tr>
<td>first add during global load</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Instruction Bottleneck

- At 17 GB/s, we’re far from bandwidth bound

- Therefore a likely bottleneck is instruction overhead
  - Ancillary instructions that are not loads, stores, or core arithmetic
  - In other words: address arithmetic and loop overhead

- Strategy: unroll loops
Unrolling the Last Warp

- As reduction proceeds, the number of “active” threads decreases
  - When \( s \leq 32 \), we have only one warp left

- Instructions executed in lockstep fashion within a warp

- That means when \( s \leq 32 \):
  - We don’t need to \texttt{__syncthreads()}\texttt{()}
  - We don’t need “if (tid < s)” because it doesn’t save any work

- The key idea: unroll the last 6 iterations of the inner loop
Reduction #5: Unroll the Last Warp

```c
__device__ void warpReduce(volatile int* sdata, int tid) {
    sdata[tid] += sdata[tid + 32];
    sdata[tid] += sdata[tid + 16];
    sdata[tid] += sdata[tid + 8];
    sdata[tid] += sdata[tid + 4];
    sdata[tid] += sdata[tid + 2];
    sdata[tid] += sdata[tid + 1];
}
```

Note: This saves useless work in all warps, not just the last one!
Without unrolling, all warps execute every iteration of the for loop and if statement
## Performance for 4M element reduction

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Addressing Method</th>
<th>Time (2^{22} ints)</th>
<th>Bandwidth</th>
<th>Step Speedup</th>
<th>Cumulative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel 1:</td>
<td>interleaved addressing</td>
<td>8.054 ms</td>
<td>2.083 GB/s</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>with divergent branching</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel 2:</td>
<td>interleaved addressing</td>
<td>3.456 ms</td>
<td>4.854 GB/s</td>
<td>2.33x</td>
<td>2.33x</td>
</tr>
<tr>
<td></td>
<td>with bank conflicts</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel 3:</td>
<td>sequential addressing</td>
<td>1.722 ms</td>
<td>9.741 GB/s</td>
<td>2.01x</td>
<td>4.68x</td>
</tr>
<tr>
<td>Kernel 4:</td>
<td>first add during global load</td>
<td>0.965 ms</td>
<td>17.377 GB/s</td>
<td>1.78x</td>
<td>8.34x</td>
</tr>
<tr>
<td>Kernel 5:</td>
<td>unroll last warp</td>
<td>0.536 ms</td>
<td>31.289 GB/s</td>
<td>1.8x</td>
<td>15.01x</td>
</tr>
</tbody>
</table>

NVIDIA [M. Harris]→
Complete Unrolling

- If we knew the number of iterations (or equivalently, of threads in a block) at compile time, we could completely unroll the reduction
  - Luckily, the block size on G80 is limited by the GPU to 512 threads
    - 1024 on newer Fermi GPUs
  - Also, we are sticking to power-of-2 block sizes

- So we can easily unroll for a fixed block size
  - But we need to be generic – how can we unroll for block sizes that we don’t know at compile time?

- Use of templates can solve this issue…
  - CUDA supports C++ template parameters on device and host functions
Unrolling with Templates

- Specify block size as a function template parameter
- The kernel is parameterized:

```cpp
template <unsigned int blockSize>
__global__ void reduce6(int *g_idata, int *g_odata)
```
Reduction #6: Completely Unrolled

This is the key part of the kernel

```c
if (blockSize >= 512) {
    if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads();
    if (blockSize >= 256) {
        if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads();
        if (blockSize >= 128) {
            if (tid < 64) { sdata[tid] += sdata[tid + 64]; } __syncthreads();
        }
    }
    if (tid < 32) warpReduce<blockSize>(sdata, tid); // last warp only
}
```

This is a helper function (device only)

```c
template <unsigned int blockSize>
__device__ void warpReduce(volatile int* sdata, int tid) {
    if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
    if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
    if (blockSize >= 16) sdata[tid] += sdata[tid + 8];
    if (blockSize >= 8) sdata[tid] += sdata[tid + 4];
    if (blockSize >= 4) sdata[tid] += sdata[tid + 2];
    if (blockSize >= 2) sdata[tid] += sdata[tid + 1];
}
```

- All code in RED will be evaluated at compile time. Results in a very efficient inner loop.
- For Fermi, you’d have one more if statement that covers the case when blockSize >= 1024
- You can call the warpReduce function only when you got to one wrap. Reason: you don’t have to synchronize at that point.
Invoking Template Kernels

```
switch (threads) {
    case 512:
        reduce6<512><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 256:
        reduce6<256><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 128:
        reduce6<128><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 64:
        reduce6< 64><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 32:
        reduce6< 32><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 16:
        reduce6< 16><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case  8:
        reduce6<  8><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case  4:
        reduce6<  4><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case  2:
        reduce6<  2><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case  1:
        reduce6<  1><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
}
```
## Performance for 4M element reduction

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Description</th>
<th>Time (2^{22} ints)</th>
<th>Bandwidth</th>
<th>Step Speedup</th>
<th>Cumulative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel 1:</td>
<td>interleaved addressing with divergent branching</td>
<td>8.054 ms</td>
<td>2.083 GB/s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel 2:</td>
<td>interleaved addressing with bank conflicts</td>
<td>3.456 ms</td>
<td>4.854 GB/s</td>
<td>2.33x</td>
<td>2.33x</td>
</tr>
<tr>
<td>Kernel 3:</td>
<td>sequential addressing</td>
<td>1.722 ms</td>
<td>9.741 GB/s</td>
<td>2.01x</td>
<td>4.68x</td>
</tr>
<tr>
<td>Kernel 4:</td>
<td>first add during global load</td>
<td>0.965 ms</td>
<td>17.377 GB/s</td>
<td>1.78x</td>
<td>8.34x</td>
</tr>
<tr>
<td>Kernel 5:</td>
<td>unroll last warp</td>
<td>0.536 ms</td>
<td>31.289 GB/s</td>
<td>1.8x</td>
<td>15.01x</td>
</tr>
<tr>
<td>Kernel 6:</td>
<td>completely unrolled</td>
<td>0.381 ms</td>
<td>43.996 GB/s</td>
<td>1.41x</td>
<td>21.16x</td>
</tr>
</tbody>
</table>
Parallel Reduction Complexity

- Assume that the number of elements in array is of the form $N=2^D$

- $\log(N)$ parallel stages, each stage $S$ requires $N/2^S$ independent ops
  - Stage Complexity is $O(\log N)$

- For $N=2^D$, approach requires a total of $\sum_{S \in [1..D]} 2^{D-S} = N-1$ operations
  - Work Complexity is $O(N)$ – It is work-efficient
  - That is, it does not perform more operations than a sequential algorithm

- Time complexity, for $P$ threads physically in parallel ($P$ processors): $O(N/P + \log N)$
  - Compare to $O(N)$ for sequential reduction
  - In a thread block, $N=P$, so $O(\log N)$
What About Cost?

- Cost of a parallel algorithm is processors × time complexity
  - Allocate threads instead of processors: $O(N)$ threads
  - Time complexity is $O(\log N)$, so cost is $O(N \log N)$: not cost efficient!

- Brent’s theorem suggests $O(N/\log N)$ threads
  - Each thread does $O(\log N)$ sequential work
  - Then all $O(N/\log N)$ threads cooperate for $O(\log N)$ stages
  - Cost = $O((N/\log N) \times \log N) = O(N)$ \rightarrow cost efficient

- Sometimes called algorithm cascading
  - Can lead to significant speedups in practice
Algorithm Cascading

- Combine sequential and parallel reduction
  - Each thread loads and sums multiple elements into shared memory
  - Tree-based reduction in shared memory

- Brent’s theorem says each thread should sum $O(\log n)$ elements
  - i.e. 1024 or 2048 elements per block vs. 256

- Probably beneficial to push it even further
  - Possibly better latency hiding with more work per thread
  - More threads per block reduces levels in tree of recursive kernel invocations
  - High kernel launch overhead in last levels with few blocks

- On G80, best performance with 64-256 blocks of 128 threads
  - 1024-4096 elements per thread
Kernel 7, Comments

- For the first six kernels a large number of blocks was used to “tile” the array

- Kernel 7: reduce the number of blocks and have a thread do more work than just fetch something to shared memory

- Example [cooked up, not related to actual CUDA warp size, typical CUDA block dim, etc.]:
  - Say you have 1024 elements stored in an array; you need to reduce that array
  - You start with 32 blocks, each with 4 threads
  - Then, 128 threads total. It means that a thread, say in block 11, would have to add two numbers, then two numbers, then two numbers, then two more numbers.
  - At this point, everything is in the union of the shared memory associated with the 32 blocks. At this point proceed like before with kernel 6.
Reduction #7: Multiple Adds / Thread

Replace load and add of two elements:

```c
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```

With a while loop to add as many as necessary:

```c
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockSize*2) + threadIdx.x;
unsigned int gridSize = blockSize*2*gridDim.x;
sdata[tid] = 0;

while (i < n) {
    sdata[tid] += g_idata[i] + g_idata[i+blockSize];
    i += gridSize;
}
__syncthreads();
```

Note: gridSize loop stride to maintain coalescing!
Performance for 4M element reduction

Kernel 7 on 32M elements: 73 GB/s!

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Addressing Method</th>
<th>Time (2^{22} ints)</th>
<th>Bandwidth</th>
<th>Step Speedup</th>
<th>Cumulative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>interleaved addressing with divergent branching</td>
<td>8.054 ms</td>
<td>2.083 GB/s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>interleaved addressing with bank conflicts</td>
<td>3.456 ms</td>
<td>4.854 GB/s</td>
<td>2.33x</td>
<td>2.33x</td>
</tr>
<tr>
<td>3</td>
<td>sequential addressing</td>
<td>1.722 ms</td>
<td>9.741 GB/s</td>
<td>2.01x</td>
<td>4.68x</td>
</tr>
<tr>
<td>4</td>
<td>first add during global load</td>
<td>0.965 ms</td>
<td>17.377 GB/s</td>
<td>1.78x</td>
<td>8.34x</td>
</tr>
<tr>
<td>5</td>
<td>unroll last warp</td>
<td>0.536 ms</td>
<td>31.289 GB/s</td>
<td>1.8x</td>
<td>15.01x</td>
</tr>
<tr>
<td>6</td>
<td>completely unrolled</td>
<td>0.381 ms</td>
<td>43.996 GB/s</td>
<td>1.41x</td>
<td>21.16x</td>
</tr>
<tr>
<td>7</td>
<td>multiple elements per thread</td>
<td>0.268 ms</td>
<td>62.671 GB/s</td>
<td>1.42x</td>
<td>30.04x</td>
</tr>
</tbody>
</table>
template <unsigned int blockSize>
__device__ void warpReduce(volatile int *sdata, unsigned int tid) {
    if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
    if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
    if (blockSize >= 16) sdata[tid] += sdata[tid + 8];
    if (blockSize >= 8) sdata[tid] += sdata[tid + 4];
    if (blockSize >= 4) sdata[tid] += sdata[tid + 2];
    if (blockSize >= 2) sdata[tid] += sdata[tid + 1];
}

template <unsigned int blockSize>
__global__ void reduce7(int *g_idata, int *g_odata, unsigned int n) {
    extern __shared__ int sdata[];
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*(blockSize*2) + tid;
    unsigned int gridSize = blockSize*2*gridDim.x;
    sdata[tid] = 0;

    while (i < n) { sdata[tid] += g_idata[i] + g_idata[i+blockSize]; i += gridSize; } __syncthreads();

    if (blockSize >= 512) { if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads(); }
    if (blockSize >= 256) { if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads(); }
    if (blockSize >= 128) { if (tid < 64) { sdata[tid] += sdata[tid + 64]; } __syncthreads(); }

    if (tid < 32) warpReduce(sdata, tid);
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
Performance Comparison

1: Interleaved Addressing: Divergent Branches
2: Interleaved Addressing: Bank Conflicts
3: Sequential Addressing
4: First add during global load
5: Unroll last warp
6: Completely unroll
7: Multiple elements per thread (max 64 blocks)
Sources of Efficiency Improvement

- Algorithmic optimizations
  - Changes to addressing, algorithm cascading
  - 11.84x speedup, combined!

- Code optimizations
  - Loop unrolling
  - 2.54x speedup, combined

NVIDIA [M. Harris]→
Lessons Learned, Vector Reduction

- Understand CUDA performance characteristics
  - Memory coalescing
  - Warp divergence
  - Bank conflicts
  - Latency hiding

- Use peak performance metrics to guide optimization

- Know how to identify type of bottleneck
  - E.g. memory, core computation, or instruction overhead

- Optimize your algorithm, *then* unroll loops

- Use template parameters to generate optimal code

- Understand parallel algorithm complexity theory
CUDA Streams
CUDA Streams: Why Bother?

- In the CPU-GPU interplay, a CUDA enabled GPU can count on two engines
  - An execution engine
  - A copy engine, which actually has 2 subengines that can work simultaneously
    - A H2D copy subengine
    - A D2H copy subengine

- Goal of this segment: learn how to use both engines at the same time

- Remark:
  - In this segment of the lecture the important things happen on the host side, not on the device side
Asynchronous Concurrent Execution

- In order to facilitate concurrent execution on host and device, some function calls are asynchronous
  - Control is returned to the host thread before the device has completed the requested task

- Examples of asynchronous calls
  - Kernel launches
  - Device $ device memory copies
  - Host $ device memory copies of a memory block of 64 KB or less
  - Memory copies performed by functions that are suffixed with Async

- NOTE: When an application is run via a CUDA debugger or profiler (cuda-gdb, nvvp, Parallel Nsight), all launches are synchronous
Host-Device Data Transfer Issues

- In general, host $ device data transfers using `cudaMemcpy()` are blocking
  - Control is returned to the host thread only after the data transfer is complete

- There is a non-blocking variant, `cudaMemcpyAsync()`

  ```c
  cudaMemcpyAsync(a_d, a_h, size, cudaMemcpyHostToDevice, 0);
  kernel<<<grid,block>>>(a_d);
  cpuFunction();
  ```

  - The host does not wait on the device to finish the mem copy and the kernel call for it to start execution of `cpuFunction()` call
  - The launch of “kernel” only happens after the mem copy call finishes

- NOTE 1: the asynchronous transfer version requires pinned host memory (allocated with `cudaHostAlloc()`), and it contains an additional argument (a stream ID)

- NOTE 2: up until this point we are still not using the two GPU engines
  - We only make the CPU stay busy (which is nonetheless quite good)
Overlapping Host $\&$ Device Data Transfer with Device Execution

- **When is this overlapping useful?**
  - Imagine a kernel executes on the device and only works with the lower half of the device global memory
  - Then, you can copy data from host to device into the upper half of the device global memory
  - These two operations can take place simultaneously

- **Note that there is an issue with this idea:**
  - The device execution stack is FIFO, one function call on the device is not serviced until all the previous device function calls completed
  - This would prevent overlapping execution with data transfer

- **This issue was addressed by the use of CUDA “streams”**
CUDA Streams: Overview

- A programmer can manage concurrency through *streams*

- A stream is a sequence of CUDA commands that execute in issue-order
  - Look at a stream as a queue of GPU operations
  - The execution order in a stream is identical to the order in which the GPU operations are added to the stream
  - NOTE: an operation in a stream does not commence prior to the previous operation being fully completed
    - There is a distinction between queuing an operation in a stream and the moment when it actually starts to be executed on the GPU
One host thread can define multiple CUDA streams

What are the typical operations in a stream?
- Invoking a data transfer
- Invoking a kernel execution
- Handling events

With respect to each other, different CUDA streams execute their commands as they see fit
- Inter-stream relative behavior is not guaranteed and should therefore not be relied upon for correctness (e.g. inter-kernel communication for kernels allocated to different streams is undefined)
- Another way to look at it: streams can be synchronized at barrier points, but correlation of sequence execution within different streams is not supported
CUDA Streams: Creation

- A stream is defined by creating a stream object
  - It is subsequently used by specifying it as the stream parameter to a sequence of kernel launches and host $ device memory copies

- The following code sample creates two streams and allocates an array “hostPtr” of float in page-locked memory
  - hostPtr will be used in asynchronous host $ device memory transfers

```c
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);
float* hostPtr;
cudaMallocHost(&hostPtr, 2 * size);
```

- NOTE: As soon you invoke a CUDA function you create a default stream (stream 0)
  - If you don’t explicitly state a stream in the execution configuration of a kernel it is assumed it’s launched as part of stream 0

Notice the length of the array
CUDA Streams: Making Use of Them

- In the code below, each of the two streams is defined as a sequence of
  - One memory copy from host to device,
  - One kernel launch, and
  - One memory copy from device to host

```c
for (int i = 0; i < 2; ++i) {
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size, size, cudaMemcpyHostToDevice, stream[i]);
    MyKernel<<<100, 512, 0, stream[i]>>>(outputDevPtr + i * size, inputDevPtr + i * size, size);
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size, size, cudaMemcpyDeviceToHost, stream[i]);
}
```

- There are some wrinkles to it, we’ll revisit shortly…
CUDA Streams: Clean Up Phase

- Streams are released by calling `cudaStreamDestroy()`

```c
for (int i = 0; i < 2; ++i)
    cudaStreamDestroy(stream[i]);
```

- `cudaStreamDestroy()` waits for all preceding commands in the given stream to complete before destroying the stream and returning control to the host thread.
CUDA Streams: Caveats

- Two commands from different streams cannot run concurrently if either one of the following operations is issued in-between them by the host thread:
  - A page-locked host memory allocation,
  - A device memory allocation,
  - A device memory set,
  - A device $ device memory copy,
  - Any CUDA command to stream 0 (including kernel launches and host $ device memory copies that do not specify any stream parameter)
  - A switch between the L1/shared memory configurations
CUDA Streams: Synchronization Aspects

\texttt{cudaDeviceSynchronize()} halts execution on the host until all preceding commands in all CUDA streams have completed.

\texttt{cudaStreamSynchronize()} takes a stream as a parameter and halts execution on the host until all preceding commands in the given CUDA stream have completed. It can be used to synchronize the host with a specific stream, allowing other streams to continue executing on the device.

\texttt{cudaStreamWaitEvent()} takes a CUDA stream and an event as parameters and makes all the commands added to the given stream after the call to \texttt{cudaStreamWaitEvent()} delay their execution until the given event has completed. Note: this halts the execution of tasks in a stream!

\texttt{cudaStreamQuery()} provides applications with a way to know if all preceding commands in a stream have completed.

- NOTE: To avoid unnecessary slowdowns, all these synchronization functions are usually best used for timing purposes or to isolate a launch or memory copy that is failing.
Example:
Use of cudaStreamWaitEvent

- Assume `stream1` and `stream2` have been defined/initialized already
- The point of this example:
  - Use the two copy sub-engines at the same time
  - Wait onto the launching of the `myKernel` until the copy in `stream 1` is finished

```c
cudaEvent_t event;
cudaEventCreate (&event);  // create event

cudaMemcpyAsync ( d_in, in, size, H2D, stream1 );  // 1) H2D copy of new input
cudaEventRecord ( event, stream1 );  // record event

cudaMemcpyAsync ( out, d_out, size, D2H, stream2 );  // 2) D2H copy of previous result

cudaStreamWaitEvent ( stream2, event );  // wait for event in stream1
myKernel<<< 1000, 512, 0, stream2 >>> ( d_in, d_out );  // 3) GPU must wait for 1 and 2
someCPUfunction ( blah )  // this gets executed right away
```