Case Study: CUDA Reduction
Concurrency through CUDA Streams

October 26, 2015
Quote of the Day

“Be yourself; everyone else is already taken.”
-- Oscar Wilde, playwright
[1854 - 1900]
Before We Get Started

Issues covered last time:
- Case study: parallel prefix scan in CUDA – wrap up, second approach
- CUDA Share Memory issues
- Case study: parallel reduction in CUDA

Today’s topics
- Case study: parallel reduction in CUDA [wrap up]
- CUDA streams

Assignment:
- HW06 – due Wd, Oct 28 at 11:59 PM
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:

```
// 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:

```
// 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</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>8.054 ms</td>
<td>2.083 GB/s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>interleaved addressing with divergent branching</td>
<td></td>
<td></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 \_syncthreads() \n  - We don’t need “if \( \text{tid} < s \)” because it doesn’t save any work

- The key idea: unroll the last 6 iterations of the inner loop
  - Why 6? Since \( 2^6 = 64 \), which is how many entries the last warp ought to reduce
Reduction #5: Unroll the Last Warp

// and use later like this...
for (unsigned int s=blockDim.x/2; s>32; s>>=1) {
    if (tid < s)
        sdata[tid] += sdata[tid + s];
    __syncthreads();
}

if (tid < 32) warpReduce(sdata, tid);

__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

IMPORTANT: For this to be correct, we must use the "volatile" keyword!

This used to be: S>0
# Performance for 4M element reduction

<table>
<thead>
<tr>
<th>Kernel</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>8.054 ms</td>
<td>2.083 GB/s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>interleaved addressing with divergent branching</td>
<td></td>
<td></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>
<tr>
<td>Kernel 5:</td>
<td>0.536 ms</td>
<td>31.289 GB/s</td>
<td>1.8x</td>
<td>15.01x</td>
</tr>
<tr>
<td>unroll last warp</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 global functions
Unrolling with Templates

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

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

```cpp
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 the key part of the kernel

```cpp
if (tid < 32) warpReduce<blockSize>(sdata, tid);
```

```
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];
}
```

This is a helper function (device only)

- 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 warp. Reason: you don’t have to synchronize at that point.
Invoking Template Kernels

```c
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;
}
```

This is code on the host, calling the appropriate kernel.
## 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>

<table>
<thead>
<tr>
<th>Kernel 3:</th>
<th>Time (2^{22} ints)</th>
<th>Bandwidth</th>
<th>Step Speedup</th>
<th>Cumulative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential addressing</td>
<td>1.722 ms</td>
<td>9.741 GB/s</td>
<td>2.01x</td>
<td>4.68x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Kernel 4:</th>
<th>Time (2^{22} ints)</th>
<th>Bandwidth</th>
<th>Step Speedup</th>
<th>Cumulative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<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>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Kernel 5:</th>
<th>Time (2^{22} ints)</th>
<th>Bandwidth</th>
<th>Step Speedup</th>
<th>Cumulative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<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>

<table>
<thead>
<tr>
<th>Kernel 6:</th>
<th>Time (2^{22} ints)</th>
<th>Bandwidth</th>
<th>Step Speedup</th>
<th>Cumulative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<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 $\times$ 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 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>

<table>
<thead>
<tr>
<th>Kernel 2: interleaved addressing with bank conflicts</th>
<th>3.456 ms</th>
<th>4.854 GB/s</th>
<th>2.33x</th>
<th>2.33x</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Kernel 3: sequential addressing</th>
<th>1.722 ms</th>
<th>9.741 GB/s</th>
<th>2.01x</th>
<th>4.68x</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Kernel 4: first add during global load</th>
<th>0.965 ms</th>
<th>17.377 GB/s</th>
<th>1.78x</th>
<th>8.34x</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Kernel 5: unroll last warp</th>
<th>0.536 ms</th>
<th>31.289 GB/s</th>
<th>1.8x</th>
<th>15.01x</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Kernel 6: completely unrolled</th>
<th>0.381 ms</th>
<th>43.996 GB/s</th>
<th>1.41x</th>
<th>21.16x</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Kernel 7: multiple elements per thread</th>
<th>0.268 ms</th>
<th>62.671 GB/s</th>
<th>1.42x</th>
<th>30.04x</th>
</tr>
</thead>
</table>
Final Kernel...

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
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 and *then* unroll loops

- Use template parameters to generate optimal code

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

- A CUDA enabled GPU has 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

- This “streams” segment of ME759 has two goals:
  - Learn how to overlap the execution on the CPU and the execution on the GPU
    - Not strictly a “streams” issue, but this is the right time to talk about it
  - Learn how to simultaneously use both GPU engines

- Remark, in relation to this “streams” segment of the course:
  - The important things happen on the host side, not on the device side
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), all launches are synchronous.
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()`

```
cudaMemcpyAsync(a_d, a_h, size, cudaMemcpyHostToDevice, 0);
myKernel<<<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 “myKernel” only happens after the mem copy call finishes

**NOTE:** the asynchronous transfer version requires pinned host memory (allocated with `cudaHostAlloc()`), and it contains an additional argument (a stream ID)
- The `cudaHostAlloc()` replaces `malloc()` typical call on the host side

What does this buy us?
- We have the CPU busy while copying data to/from device
Overlapping Host ↔ Device Data Transfer with Device Execution

- When is this overlapping useful?
  - Imagine a kernel executes on the device and only works with a portion (say 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 (or whatever portion of the global memory is not used)
  - 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*
  - “*concurrency*” refers to “the copy and the execution engines of the GPU working at the same time” or “multiple different kernels being executed at the same time on the GPU”

- 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 (FIFO)
  - 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
CUDA Streams: Overview

- 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)
    cudaMemcpy(&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” (zero)
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 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,
  - A 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 Stream: More Caveats

- All GPU calls (memcpy, kernel execution, etc.) are placed into default stream unless otherwise specified.

- Stream 0 is special
  - Synchronous with all streams
    - Meaning: Things done in stream 0 cannot overlap other streams
      - Exception: see next bullet

- Streams with non-blocking flag are exception
  - `cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking)`
CUDA Streams: Synchronization Aspects

cudaDeviceSynchronize() halts execution on the host until all preceding commands in all CUDA streams have completed

- Halts execution of the host

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

- Halts execution of the host

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

- Halts execution within a stream

cudaStreamQuery() provides applications with a way to know if all preceding commands in a stream have completed

- NOTE: To avoid unnecessary slowdowns, use these synchronization functions sparingly
Example: Use of cudaStreamWaitEvent

- Assume `stream1` and `stream2` have been defined/initialized already
- The point of this example:
  - Use the two copy subengines at the same time
  - Wait on the Stream 2 launching of the `myKernel` until the copy operation in Stream 1 is completed

```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, blahblah )  // this gets executed right away
```

[S. Rennich]→