CUDA Streams
Serialized Data Transfer and Computation

- So far, the way we use cudaMemcpy serializes data transfer and GPU computation for VecAddKernel()
Device Overlap

- Some CUDA devices support device overlap
  - Simultaneously execute a kernel while copying data between device and host memory

```c
int dev_count;
cudaDeviceProp prop;

cudaGetDeviceCount( &dev_count);
for (int i = 0; i < dev_count; i++) {
    cudaGetDeviceProperties(&prop, i);
    if (prop.deviceOverlap) ...
```
Ideal, Pipelined Timing

- Divide large vectors into segments
- Overlap transfer and compute of adjacent segments

```
Trans A.0  Trans B.0  Comp C.0 = A.0 + B.0
Trans A.1  Trans B.1  Comp C.1 = A.1 + B.1
Trans A.2  Trans B.2
Trans A.3  Trans B.3
```
CUDA Streams

- CUDA supports parallel execution of kernels and cudaMemcpy() with “Streams”
- Each stream is a queue of operations (kernel launches and cudaMemcpy() calls)
- Operations (tasks) in different streams can go in parallel
  - “Task parallelism”
Streams

- Requests made from the host code are put into First-In-First-Out queues
  - Queues are read and processed asynchronously by the driver and device
  - Driver ensures that commands in a queue are processed in sequence. E.g., Memory copies end before kernel launch, etc.
Streams cont.

- To allow concurrent copying and kernel execution, use multiple queues, called “streams”
  - CUDA “events” allow the host thread to query and synchronize with individual queues (i.e. streams).
Conceptual View of Streams

Operations (Kernel launches, `cudaMemcpy()` calls)
OVERLAPPING DATA TRANSFER W/ COMPUTATION
Simple Multi-Stream Host Code

```
cudaStream_t stream0, stream1;
cudaStreamCreate(&stream0);
cudaStreamCreate(&stream1);

float *d_A0, *d_B0, *d_C0; // device memory for stream 0
float *d_A1, *d_B1, *d_C1; // device memory for stream 1

// cudaMalloc() calls for d_A0, d_B0, d_C0, d_A1, d_B1, d_C1 go here
```
Simple Multi-Stream Host Code (Cont.)

for (int i=0; i<n; i+=SegSize*2) {
    cudaMemcpyAsync(d_A0, h_A+i, SegSize*sizeof(float),..., stream0);
    cudaMemcpyAsync(d_B0, h_B+i, SegSize*sizeof(float),..., stream0);
    vecAdd<<<SegSize/256, 256, 0, stream0>>>(d_A0, d_B0, ...);
    cudaMemcpyAsync(h_C+i, d_C0, SegSize*sizeof(float),..., stream0);
    cudaMemcpyAsync(d_A1, h_A+i+SegSize, SegSize*sizeof(float),..., stream1);
    cudaMemcpyAsync(d_B1, h_B+i+SegSize, SegSize*sizeof(float),..., stream1);
    vecAdd<<<SegSize/256, 256, 0, stream1>>>(d_A1, d_B1, ...);
    cudaMemcpyAsync(d_C1, h_C+i+SegSize, SegSize*sizeof(float),..., stream1);
}
A View Closer to Reality in Previous GPUs

Operations (Kernel launches, cudaMemcpy() calls)
Not quite the overlap we want in some GPUs

- C.0 blocks A.1 and B.1 in the copy engine queue
Better Multi-Stream Host Code

for (int i=0; i<n; i+=SegSize*2) {
    cudaMemcpyAsync(d_A0, h_A+i, SegSize*sizeof(float),..., stream0);
    cudaMemcpyAsync(d_B0, h_B+i, SegSize*sizeof(float),..., stream0);
    cudaMemcpyAsync(d_A1, h_A+i+SegSize, SegSize*sizeof(float),..., stream1);
    cudaMemcpyAsync(d_B1, h_B+i+SegSize, SegSize*sizeof(float),..., stream1);

    vecAdd<<<SegSize/256, 256, 0, stream0>>>(d_A0, d_B0, ...);
    vecAdd<<<SegSize/256, 256, 0, stream1>>>(d_A1, d_B1, ...);

    cudaMemcpyAsync(h_C+i, d_C0, SegSize*sizeof(float),..., stream0);
    cudaMemcpyAsync(h_C+i+SegSize, d_C1, SegSize*sizeof(float),..., stream1);
}
C.0 no longer blocks A.1 and B.1

Operations (Kernel launches, cudaMemcpy() calls)
Better, not quite the best overlap

- C.1 blocks next iteration A.0 and B.0 in the copy engine queue
Ideal, Pipelined Timing

- Will need at least three buffers for each original A, B, and C, code is more complicated
Wait until all tasks have completed

- `cudaStreamSynchronize(stream_id)`
  - Used in host code
  - Takes one parameter – stream identifier
  - Wait until all tasks in a stream have completed
  - E.g., `cudaStreamSynchronize(stream0)` in host code ensures that all tasks in the queues of stream0 have completed

- This is different from `cudaDeviceSynchronize()`
  - Also used in host code
  - No parameter
  - `cudaDeviceSynchronize()` waits until all tasks in all streams have completed for the current device