Objective

To learn more advanced features of the CUDA APIs for data transfer and kernel launch
- Task parallelism for overlapping data transfer with kernel computation
- CUDA streams

Serialized Data Transfer and GPU computation
- So far, the way we use cudaMemcpy serializes data transfer and GPU computation

Device Overlap
- Most CUDA devices support device overlap
  – Simultaneously execute a kernel while performing a copy 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) …
```
### Overlapped (Pipelined) Timing
- Divide large vectors into segments
- Overlap transfer and compute of adjacent segments

<table>
<thead>
<tr>
<th>Trans</th>
<th>Comp</th>
<th>Trans</th>
<th>Comp</th>
<th>Trans</th>
<th>Comp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1</td>
<td>C.1</td>
<td>A.2</td>
<td>C.2</td>
<td>A.3</td>
<td>C.3</td>
</tr>
<tr>
<td>B.1</td>
<td>A.1 + B.1</td>
<td>B.2</td>
<td>A.2 + B.2</td>
<td>B.3</td>
<td>A.3 + B.3</td>
</tr>
</tbody>
</table>

### Using CUDA Streams and Asynchronous Memcpy
- CUDA supports parallel execution of kernels and cudaMemcpy with **streams**
- Each stream is a **queue of operations** (kernel launches and cudaMemcpy’s)
- Operations (tasks) in different streams
  - can execute in parallel
  - a version of **task parallelism**

### Streams
**Queue operation:**
1. device requests (host code) placed into queue
2. queue processed asynchronously by driver and device
3. each queue processed in order (no overlap), so memory copies end before kernel launch, and so forth

### Multiple Streams Enable Parallelism
- To allow concurrent copying and kernel execution, you need to use multiple streams.
Conceptual View of Streams

A 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
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_A0, h_A+i+SegSize, SegSize*sizeof(float),.., stream1);
cudaMemcpyAsync(d_B0, h_B+i+SegSize, SegSize*sizeof(float),.., stream1);
vecAdd<<<SegSize/256, 256, 0, stream1>>>(d_A0, d_B0, …);
cudaMemcpyAsync(h_C+i+SegSize, d_C1, SegSize*sizeof(float),.., stream1);
}
Not quite the overlap we want

- C.1 blocks A.2 and B.2 in the copy engine queue (head-of-line blocking).

A View Closer to Reality

MemCpy A.1
MemCpy B.1
MemCpy A.2
MemCpy B.2
MemCpy C.1
MemCpy C.2

Copy Engine

Kernel Engine

A Better Multi-Stream Host Code

```c
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(d_C0, h_C+i, SegSize*sizeof(float),..,
                     stream0);
    cudaMemcpyAsync(d_C1, h_C+i+SegSize;
                     SegSize*sizeof(float),..,
                     stream1);
}
```

Better Overlap with Two Streams

- C.1 no longer blocks A.2 and B.2 in the copy engine queue
- However, C.2 still blocks A.1 and A.2 from the next iteration – PCIe used for only one direction
Three streams needed for continuously pipelined) timing
• Divide large vectors into segments
• Overlap transfer and compute of adjacent segments

Hyper Queue
• Provide multiple real stream queues for each engine
• Allow more concurrency by allowing some streams to make progress for an engine while others are blocked

Fermi (and older) Concurrency
Fermi allows 16-way concurrency
- Up to 16 grids can run at once
- But kernels from CUDA streams multiplex into a single queue
- Overlap only at stream edges

Kepler Improved Concurrency
Kepler allows 32-way concurrency
- One work queue per stream
- Concurrency at full-stream level
- No inter-stream dependencies
Smaller Segments Reduce Boundary Effects

**How small should segments be?**

- If we *overlap*
  - transfer of segment N’s inputs,
  - computation of segment N – 1, and
  - transfer of segment N – 2’s results,
- we still have non-overlapping work at the beginning and the end.

So segments should be really small?

---

Execution Time is Ideally Linear in Size

Think about execution time as a function of segment size.

**Ideally, the relationship is linear.**

---

Execution Time Never Reaches Zero

But real execution time has a minimum.

Why?
- Some SMs idle.
- Too few warps to keep SMs busy.
- Too few threads to fill a warp.
- Load balance worse with fewer blocks.
- Kernel launches take time.
Use Moderate Segment Size and Device Query

Data transfers
- have similar non-linearities for small sizes
- due to startup costs on host and DMA.

So how small should segments be?
Moderately sized.
Best size likely to depend on GPU.

ANY QUESTIONS?