CUDA Execution Model – III
Streams and Events

Objectives

• Become familiar with CUDA streams
   Usage to overlap GPU computation and data transfers
   Usage to improve GPU utilization: multi-kernel execution on the GPU (for later devices)

• Use of CUDA events
   Usage in timing of GPU kernel execution
Reading Assignment

- NVIDIA presentation and links in class Resources page
  - This presentation is in part a summary of that NVIDIA presentation
  - See example programs (look on class Resources page)
- CUDA Programming Guide

---

Offloading Activities via Streams

- CUDA Runtime commands by host issued through streams
  - Operationally, a stream is a FIFO structure
  - Commands in the same stream executed sequentially
  - Commands in different streams may be executed concurrently
- Streams are mapped to hardware queues in the device
  - Multiple streams mapped to each queue → serializes commands
- Default stream is stream 0
  - Need not be explicitly specified
• Synchronous API calls
  - Enqueue call into the stream and wait for completion

• Asynchronous API calls
  - Enqueue call into the stream and return
  - Host execution is overlapped with call execution
  - Kernel calls are asynchronous

• How do we know when all calls are completed?

Host waits until the stream is empty

• Performance gain primarily with host/device overlap
Creating Streams

- `cudaStream_t stream;`
  - Declares a stream handle

- `cudaStreamCreate(&stream);`
  - Allocates a stream

- `cudaStreamDestroy(stream);`
  - Deallocates a stream
  - Synchronizes host until work in stream has completed

- The stream ID is an argument to API calls
  - `kernel<<< blocks, threads, stream>>>();`

---

CUDA Events

- Cannot get accurate timing information with CPU timers \(\rightarrow\) need to perform timing on the GPU!

- Create events used to record timestamps

- Place event into the stream

- Timestamps will be recorded on the GPU when the event is processed at the GPU
CUDA Events

- cudaEvent_t start, stop;
- cudaEventCreate(&start,);
- cudaEventCreate(&stop);
- cudaEventRecord(start, stream);
- cudaEventRecord(stop, stream);
- cudaEventSynchronize(stop);
- cudaEventElapsedTime(&time, start, stop);
- cudaEventDestroy(start);

Create Events

Place events in the stream

Ensure stop event occurred

Measure time
& free

Event Synchronization

- Need to ensure that this event is executed before reading the counters
- Use cudaEventSynchronize() on host

Code Example:
**Example Code Segment**

```c
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);
cudaEventRecord(start);
saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);
cudaEventRecord(stop);
cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);
cudaEventSynchronize(stop);

float milliseconds = 0;
cudaEventElapsedTime(&milliseconds, start, stop);
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

See vector add example in repository


---

**Asynchronous Communication**

- Like kernel calls we want communication to be asynchronous
  - Use DMA engines that operate concurrently with the host
  - Would like to launch a `cudaMemcpy()` and let the host continue
- DMA engines operate with physical addresses and we have a host with virtual memory and paged address spaces → operate with pinned memory
Using Pinned Memory

- Pageable vs. pinned memory
  - Avoid page faults
  - Can use physical address → direct memory access (DMA)
  - Allocate pinned memory: `cudaHostAlloc();`
- Overlap DMA with host execution
- Required by `cudaMemcpyAsync(dst, src, size, dir, stream);`

Code Example:

```
……
cudaStreamCreate(&stream1);  -- create streams
……
cudaMallocHost(&a, sizeof(int) * size)  -- pin memory for each array
……
cudaMalloc(&dev_a, sizeof(int) * size)  -- allocate device memory
……
cudaMemcpyAsync(..., stream1)  -- asynchronous copy call
……
vecAdd<<<GRID_SIZE, CTA_SIZE, 0, stream1>>>( );  -- kernel call
……
cudaDeviceSynchronize()  -- wait for kernel to complete
……
cudaMemcpyAsync(..., stream1)  -- copy back data
……
cudaStreamDestroy(stream1);  -- remove stream
……
cudaFree(dev_a); cudaFreeHost(a)  -- free host and device memory
```

(13)
Performance Optimizations

• With asynchronous operation how can we improve performance
  ❖ Multiple streams!

• Pipelining and concurrency across communication and computation.
Pipelined Operation: A Functional View

- Break up a single problem into multiple chunks, or
- Concurrency across multiple kernels

Multiple Streams

- Lets look at the vector add implementation parallelized across streams
Stream Level Concurrency

- Break the vector up into chunks: a0, a1, b0, b1, c0, etc.
- Compute vector sum for each chunk separately
  - e.g., a0[i] + b0[i] = c0[i], across the chunk
- Interleave chunk computations across the two streams

Code Example:

How do we distribute API calls across streams?

Using Multiple Streams

- Hardware data transfer engines and the kernel scheduling queues are distinct entities
- Must maintain correct ordering of API calls between data transfer and kernel execution
  - They are placed in two different hardware queues ➔ be conservative!
- The driver sees a sequence of calls
  - Does the sequence matter?
Maximizing Concurrency

Stream 0
- copy a0
- copy b0
- kernel call k0
- copy c0
- copy a1
- copy b1
- kernel call k1
- copy c1

Stream 1
- copy a0
- copy a1
- copy b0
- copy b1
- kernel call k0
- kernel call k1
- copy c0
- copy c1

Application runtime call sequence on host

Driver sees serial sequence of calls

Dependencies that must be honored

Example from "CUDA by Example," J. Sanders and E. Kandrot, Chapter 10.6-10.7

Lessons

- Many subtle performance issues in using multiple streams
- The number and form of stream support depends on the GPU generation
- Check device properties for more information on GPU capabilities

Code Example:

(21)
Questions?