L17: Generalizing CUDA: Concurrent Dynamic Execution, and Unified Address Space

Administrative

- Remaining Lectures
  - Wednesday, April 17: Parallel Architectures and Getting to Exascale
- Final projects
  - Poster session, April 24 (dry run April 22)
  - Final report, May 1

Final Project Presentation

- Dry run on April 22, Presentation on April 24
  - Easels, tape and poster board provided
  - Tape a set of Powerpoint slides to a standard 2'x3' poster, or bring your own poster.
- Final Report on Projects due May 1
  - Submit code
  - And written document, roughly 10 pages, based on earlier submission.
  - In addition to original proposal, include
    - Project Plan and How Decomposed (from DR)
    - Description of CUDA implementation
    - Performance Measurement
    - Related Work (from DR)

Sources for Today's Lecture

References:

CUDA Dynamic Parallelism Programming Guide:

http://docs.nvidia.com/cuda/pdf/CUDA_Dynamic_Parallelism_Programming_Guide.pdf

Examples: http://developer.download.nvidia.com/GTC/PDF/GTC2012/PresentationPDF/S0338-GTC2012-CUDA-Programming-Model.pdf

CUDA 5 Programming Guide

concurrentKernels example in /usr/local/cuda/samples/6_Advanced

cdpAdvancedQuicksort example in /usr/local/cuda/samples/6_Advanced

Progress in Generalizing CUDA

- Early versions of CUDA had a lot of restrictions on parallelism that limited applicability and performance
  - First, synchronous kernel calls (host stalled)
  - Then, asynchronous, but only a single kernel executing on the GPU at a time (what we've been doing all semester)
  - More recently (CUDA 3 and compute capability 2), up to 4 kernels could be concurrently launched in the host using streams
  - Now in CUDA 5, kernels can launch other kernels (even recursive ones!)
- Early versions of CUDA dealt with memory in restrictive ways
  - Dynamic global and shared memory allocation only on host, otherwise static
  - Explicit copying between host and GPU required
  - Unified virtual address space added to CUDA 4, "pointers returned by cudaHostAlloc() can be used directly from within kernels running on these devices"

Recall some forms of asynchronous overlap and concurrent execution (Lecture 7)

- Asynchronous copy and kernel execution
  - Using page-locked memory on host
- Concurrent Execution
  - Using streams (up to 4)

Asynchronous Copy To/From Host (compute capability 1.1 and above)

- Concept:
  - Memory bandwidth can be a limiting factor on GPUs
  - Sometimes computation cost dominated by copy cost
  - But for some computations, data can be "tiled" and computation of tiles can proceed in parallel (some of your projects may want to do this, particularly for large data sets)
  - Can we be computing on one tile while copying another?
- Strategy:
  - Use page-locked memory on host, and asynchronous copies
  - Primitive **cudaMemcpyAsync**
  - Effect is GPU performs DMA from Host Memory
  - Synchronize with **cudaThreadSynchronize()**

Page-Locked Host Memory

- How the Async copy works:
  - DMA performed by GPU memory controller
  - CUDA driver takes virtual addresses and translates them to physical addresses
  - Then copies physical addresses onto GPU
  - Now what happens if the host OS decides to swap out the page???
- Special malloc holds page in place on host
  - Prevents host OS from moving the page
  - CudaMallocHost()
- But performance could degrade if this is done on lots of pages!
  - Bypassing virtual memory mechanisms

Example of Asynchronous Data Transfer

```
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
cudaMemcpyAsync(dst1, src1, size, dir, stream1);
kernel<<<grid, block, 0, stream1>>>(…);
cudaMemcpyAsync(dst2, src2, size, dir, stream2);
kernel<<<grid, block, 0, stream2>>>(…);
```

src1 and src2 must have been allocated using cudaMallocHost
stream1 and stream2 identify streams associated with asynchronous call (note 4th "parameter" to kernel invocation, by default there is one stream)

Code from asyncAPI SDK project

```
// allocate host memory
CUDA_SAFE_CALL( cudaMallocHost((void**)&a, nbytes) );
memset(a, 0, nbytes);
// allocate device memory
CUDA_SAFE_CALL( cudaMalloc((void**)&d_a, nbytes) );
CUDA_SAFE_CALL( cudaMemset(d_a, 255, nbytes) );
… // declare grid and thread dimensions and create start and stop events
// asynchronously issue work to the GPU (all to stream 0)
cudaEventRecord(start, 0);
cudaMemcpyAsync(d_a, a, nbytes, cudaMemcpyHostToDevice, 0);
increment_kernel<<<blocks, threads, 0, 0>>>(d_a, value);
cudaMemcpyAsync(a, d_a, nbytes, cudaMemcpyDeviceToHost, 0);
cudaEventRecord(stop, 0);
// have CPU do some work while waiting for GPU to finish
// release resources
CUDA_SAFE_CALL( cudaFreeHost(a) );
CUDA_SAFE_CALL( cudaFree(d_a) );
```

Concurrent Execution (Compute Capability 2.0)

- Stream concept: create, destroy, tag asynchronous operations with stream
  - Special synchronization mechanisms for streams: queries, waits and synchronize functions
- Concurrent Kernel Execution
  - Execute multiple kernels (up to 4) simultaneously
  - Example: concurrentKernels in samples/6_Advanced

CUDA Dynamic Parallelism

- Key Idea: Ability to dynamically create CUDA threads on the GPU from kernel functions
- Things to think about
  - What is the syntax?
  - How does this affect the execution model?
  - How does a thread know when launched threads have completed?
  - What does a __syncthreads() mean?
  - What happens to memory state, consistency?
- Let's look at some examples
  - Hello World
  - From GTC presentation
  - From Quicksort