GPU Course Review

CMPUT 398

Introduction to GPU Programming

Pierre Boulanger and Nilanjan Ray
• Threads are organized in blocks

• A grid is a collection of thread blocks of the same thread dimensionality which all execute the same kernel

• A block is executed by a multiprocessing unit

• The threads of a block can be identified (indexed) using 1Dimension(x), 2Dimensions (x,y) or 3Dim indexes (x,y,z)

• Blocks may be also indexed 1D, 2D or 3D.

Threads and Blocks
Threads in a Block can Communicate Using Shared Memory
Now a Simple Case: Processing a 512x512 Image

- Suppose we want one thread to process one pixel \((i,j)\).

- We can use blocks of 64 threads each. Then we need \(\frac{512 \times 512}{64} = 4096\) blocks (so to have 512x512 threads = 4096\(\times 64\))

- It's common to organize (to make indexing the image easier) the threads in 2D blocks having \(\text{blockDim} = 8 \times 8\) (the 64 threads per block). I prefer to call it \(\text{threadsPerBlock}\).

  - \(\text{dim3 threadsPerBlock}(8, 8)\);

- 64 threads and 2D \(\text{gridDim} = 64 \times 64\) blocks (the 4096 blocks needed). I prefer to call it \(\text{numBlocks}\).

  - \(\text{dim3 numBlocks}(\text{imageWidth/threadsPerBlock}.x, \text{imageHeight/threadsPerBlock}.y)\);
Now a Simple Case: Processing a 512x512 Image

- The kernel is launched like this:
  ```
  myKernel <<<numBlocks,threadsPerBlock>>>( /* params for the kernel function */);
  ```

- Finally: there will be something like "a queue of 4096 blocks", where a block is waiting to be assigned to one of the multiprocessors of the GPU to get its 64 threads executed.

- In the kernel the pixel \((i,j)\) to be processed by a thread is calculated this way:
  ```
  uint i = (blockIdx.x * blockDim.x) + threadIdx.x;
  uint j = (blockIdx.y * blockDim.y) + threadIdx.y;
  ```
SMs are SIMD Processors

- Control unit for instruction fetch, decode, and control is shared among multiple processing units
  - Control overhead is minimized (Module 1)
Block Execution Scalability

Multithreaded CUDA Program

Block 0  Block 1  Block 2  Block 3
Block 4  Block 5  Block 6  Block 7

Time

GPU with 2 SMs
SM 0  SM 1
Block 0  Block 1
Block 2  Block 3
Block 4  Block 5
Block 6  Block 7

GPU with 4 SMs
SM 0  SM 1  SM 2  SM 3
Block 0  Block 1  Block 2  Block 3
Block 4  Block 5  Block 6  Block 7
SIMD Execution Among Threads in a Warp

- All threads in a warp must execute the same instruction at any point in time.
- This works efficiently if all threads follow the same control flow path:
  - All if-then-else statements make the same decision.
  - All loops iterate the same number of times.
Control Divergence

- Control divergence occurs when threads in a warp take different control flow paths by making different control decisions
  - Some take the then-path and others take the else-path of an if-statement
  - Some threads take different number of loop iterations than others

- The execution of threads taking different paths are serialized in current GPUs
  - The control paths taken by the threads in a warp are traversed one at a time until there is no more.
  - During the execution of each path, all threads taking that path will be executed in parallel
  - The number of different paths can be large when considering nested control flow statements
Heterogeneous Programming
Hardware View of CUDA Memories
Layout Latency

Significant wire delay just getting from the CPU to the memory controller.

More wire delay getting to the memory chips.

Width/Speed varies depending on memory type.

(plus the return trip...)

x16 DRAM
x16 DRAM
x16 DRAM
x16 DRAM
x16 DRAM
x16 DRAM
x16 DRAM
x16 DRAM
Each address space is partitioned into burst sections
- Whenever a location is accessed, all other locations in the same section are also delivered to the processor
- Basic example: a 16-byte address space, 4-byte burst sections
  - In practice, we have at least 4GB address space, burst section sizes of 128-bytes or more
Memory Coalescing

- When all threads of a warp execute a load instruction, if all accessed locations fall into the same burst section, only one DRAM request will be made and the access is fully coalesced.
Memory Cache

- Kepler has cache for global memory. Caches automatically coalesce most of kernel access patterns.
Shared Memory Bank Conflicts

- Shared memory consists of 32 banks of width 4 bytes, Element \( i \) is in bank \( i \mod 32 \).

- A bank conflict occurs when 2 threads in a warp access different elements in the same bank.

- Bank conflicts cause serial memory accesses rather than parallel, are bad for performance.
Example of Memory Bank Conflicts

- Left: Conflict free with stride 1
- Center: 2-way bank conflict due to stride 2
- Right: Conflict free with stride 3
CPU-GPU Data Transfer using DMA

- DMA (Direct Memory Access) hardware is used for `cudaMemcpy()` for better efficiency
  - Frees CPU for other tasks
  - Hardware unit specialized to transfer a number of bytes requested by OS
  - Between physical memory address space regions (some can be mapped I/O memory locations)
  - Uses system interconnect, typically PCIe in today’s systems

![Diagram showing CPU Main Memory (DRAM), DMA, and Global Memory connected via PCIe]
Pinned Memory and DMA Data Transfer

- Pinned memory are virtual memory pages that are specially marked so that they cannot be paged out
- Allocated with a special system API function call
- a.k.a. Page Locked Memory, Locked Pages, etc.
- CPU memory that serve as the source or destination of a DMA transfer must be allocated as pinned memory
CUDA Data Transfer Uses Pinned Memory

- CudaMemcpy() assumes that any source or destination in the host memory is allocated as pinned memory

- If a source or destination of a cudaMemcpy() in the host memory is not allocated in pinned memory, it needs to be first copied to a pinned memory – extra overhead

- cudaMemcpy() is faster if the host memory source or destination is allocated in pinned memory since no extra copy is needed
Allocate/Free Pinned Memory

- `cudaHostAlloc()`, **three parameters**
  - Address of pointer to the allocated memory
  - Size of the allocated memory in bytes
  - Option – use `cudaHostAllocDefault` for now

- `cudaFreeHost()`, **one parameter**
  - Pointer to the memory to be freed
Concurrency

The ability to perform multiple CUDA operations simultaneously (beyond multi-threaded parallelism)

- CUDA Kernel <<<>>>
- cudaMemcpyAsync (HostToDevice)
- cudaMemcpyAsync (DeviceToHost)
- Operations on the CPU

Fermi architecture can simultaneously support (compute capability 2.0+)

- Up to 16 CUDA kernels on GPU
- 2 cudaMemcpyAsyncs (must be in different directions)
- Computation on the CPU
Concurrency Example

Serial
- cudaMemcpyAsync(H2D)
- Kernel
- cudaMemcpyAsync(D2H)

Concurrent – overlap kernel and D2H copy
- cudaMemcpyAsync(H2D)
- K1
- DH1
- K2
- DH2
- K3
- DH3
- K4
- DH4

1.33x performance improvement

NVIDIA®
Amount of Concurrency

Serial (1x)

cudaMemcpyAsync(H2D) → Kernel → cudaMemcpyAsync(D2H)

2-way concurrency (up to 2x)

cudaMemcpyAsync(H2D) → K1 → DH1 → K2 → DH2 → K3 → DH3 → K4 → DH4

3-way concurrency (up to 3x)

HD1 → K1 → DH1 → HD2 → K2 → DH2 → HD3 → K3 → DH3 → HD4 → K4 → DH4

4-way concurrency (3x+)

HD1 → K1,1 → K1,2 → K1,3 → DH1 → HD2 → K2,1 → K2,2 → K2,3 → DH2 → HD3 → K3,1 → K3,2 → K3,3 → DH3 → HD4 → K4,1 → K4,2 → K4,3 → DH4 → HD5 → K5,1 → K5,2 → K5,3 → DH5 → HD6 → K6,1 → K6,2 → K6,3 → DH6 → K7 on CPU

4+ way concurrency
Requirements for Concurrency

- CUDA operations must be in different, non-0, streams
- `cudaMemcpyAsync` with host from 'pinned' memory
  - Page-locked memory
  - Allocated using `cudaMallocHost()` or `cudaHostAlloc()`
- Sufficient resources must be available
  - `cudaMemcpyAsync`s in different directions
  - Device resources (SMEM, registers, blocks, etc.)
Streams

• Stream
  • A sequence of operations that execute in issue-order on the GPU

• Programming model used to effect concurrency
  • CUDA operations in different streams may run concurrently
  • CUDA operations from different streams may be interleaved
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.
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).
Default Stream (aka Stream '0')

• Stream used when no stream is specified

• Completely synchronous w.r.t. host and device
  • As if cudaDeviceSynchronize() inserted before and after every CUDA operation

• Exceptions – asynchronous w.r.t. host
  • Kernel launches in the default stream
  • cudaMemcpy*Async
  • cudaMemcpy*Async
  • cudaMemcpy within the same device
  • H2D cudaMemcpy of 64kB or less
Simple Example: Synchronous

cudaMalloc ( &dev1, size ) ;
double* host1 = (double*) malloc ( &host1, size ) ;
...
cudaMemcpy ( dev1, host1, size, H2D ) ;
kernel2 <<< grid, block, 0 >>> ( ..., dev2, ... ) ;
kernl3 <<< grid, block, 0 >>> ( ..., dev3, ... ) ;
cudaMemcpy ( host4, dev4, size, D2H ) ;
...

All CUDA operations in the default stream are synchronous
Simple Example: Asynchronous, No Streams

cudaMalloc ( &dev1, size ) ;
double* host1 = (double*) malloc ( &host1, size ) ;
...
cudaMemcpy ( dev1, host1, size, H2D ) ;
kernel2 <<< grid, block >>> ( ..., dev2, ... ) ;
some_CPU_method () ;
kernel3 <<< grid, block >>> ( ..., dev3, ... ) ;
cudaMemcpy ( host4, dev4, size, D2H ) ;
...

GPU kernels are asynchronous with host by default
Explicit Synchronization Example

- Resolve using an event

```
{

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

kernel <<< , , , stream2 >>> ( d_in, d_out );     // 3) must wait for 1 and 2

asynchronousCPUmethod ( … )         // Async GPU method

}
```
Implicit Synchronization

- These operations implicitly synchronize all other CUDA operations
- Page-locked memory allocation
  - cudaMallocHost
  - cudaHostAlloc
- Device memory allocation
  - cudaMalloc
- Non-Async version of memory operations
  - cudaMemcpy* (no Async suffix)
  - cudaMemcpy* (no Async suffix)
- Change to L1/shared memory configuration
  - cudaDeviceSetCacheConfig
Stream Scheduling

- Fermi hardware has 3 queues
  - 1 Compute Engine queue
  - 2 Copy Engine queues – one for H2D and one for D2H

- CUDA operations are dispatched to HW in the sequence they were issued
  - Placed in the relevant queue
  - Stream dependencies between engine queues are maintained, but lost within an engine queue

- A CUDA operation is dispatched from the engine queue if:
  - Preceding calls in the same stream have completed,
  - Preceding calls in the same queue have been dispatched, and
  - Resources are available

- CUDA kernels may be executed concurrently if they are in different streams
  - Threadblocks for a given kernel are scheduled if all threadblocks for preceding kernels have been scheduled and there still are SM resources available

- Note a blocked operation blocks all other operations in the queue, even in other streams
MultiGPU Programming

[Diagram showing a system with multiple GPUs connected through PCIe switches to a CPU, and another diagram illustrating the timeline for memory copy operations between GPUs and the CPU.]
Assume a tiled matrix multiplication that handles boundary conditions as explained in Lecture 4.5. Assume that we use 32X32 tiles to process square matrices of 1,000X1,000. Within EACH thread block, what is the maximal number of warps that will have control divergence due to handling boundary conditions for loading A tiles throughout the kernel execution?

a. 32  
b. 24  
c. 16  
d. 8
Answer: (A)
Explanation: Control divergence happens due to the handling of the right edge. For thread blocks processing tiles that are totally within the valid range in the y-dimension, all 32 warps in a block will experience divergence at the right boundary. For the thread blocks that process the bottom A tiles on the right edge, only 8 warps will experience control divergence because all threads in the lower 24 warps will fail the boundary test.
Quiz 4 Question 1

We are to process a 600X800 (800 pixels in the x or horizontal direction, 600 pixels in the y or vertical direction) picture with the PictureKernel(). That is m’s value is 600 and n’s value is 800.

```c
__global__ void PictureKernel(float* d_Pin, float* d_Pout, int n, int m) {
    // Calculate the row # of the d_Pin and d_Pout element to process
    int Row = blockIdx.y * blockDim.y + threadIdx.y;
    // Calculate the column # of the d_Pin and d_Pout element to process
    int Col = blockIdx.x * blockDim.x + threadIdx.x;
    // each thread computes one element of d_Pout if in range
    if ((Row < m) && (Col < n)) {
        d_Pout[Row*n+Col] = 2 * d_Pin[Row*n+Col];
    }
}
```

Assume that we decided to use a grid of 16X16 blocks. That is, each block is organized as a 2D 16X16 array of threads. How many warps will be generated during the execution of the kernel?

(A) 37*16
(B) 38*50
(C) 38*8*50
(D) 38*50*2
Answer: (C)
Explanation: There are \( \text{ceil}(800/16.0) = 50 \) blocks in the x direction and \( \text{ceil}(600/16.0) = 38 \) blocks in the y direction. Each block contributes \( (16*16)/32 = 8 \) warps. So there are \( 38*50*8 \) warps.
Quiz 4 Question 2

In Question 1, how many warps will have control divergence?

a. (A) 37 + 50*8
b. 38*16
c. 50
d. 0
Answer: (D)
Explanation: The size of the picture in the x dimension is a multiple of 16 so there is no block in the x direction that has any threads in the invalid range. The size of the picture in the y dimension is 37.5 times of 16. This means that the threads in the last block are divided into halves: 128 in the valid range and 128 in the invalid range. Since 128 is a multiple of 32, all warps will fall into either one or the other range. There is no control divergence.
Quiz 5 Question 1

Assume that we want to use each thread to calculate two (adjacent) output elements of a vector addition. Assume that variable i should be initialized with the index for the first element to be processed by a thread. Which of the following should be used for such initialization to allow correct, coalesced memory accesses to these first elements in the following statement?

if(i<n) C[i] = A[i] + B[i];

(A) int i=(blockIdx.x*blockDim.x)*2 + threadIdx;
(B) int i=(blockIdx.x*blockDim.x + threadIdx.x)*2;
(C) int i=(threadIdx.x*blockDim.x)*2 + blockIdx.x;
(D) int i=(threadIdx.x*blockDim.x + blockIdx.x)*2;
Quiz 5 Question 1

Answer: (A)
Explanation: Each thread block will process a section of each vector that contains twice as many elements as the number of threads in the thread block. We want to have all threads to process the first half of the section with each adjacent thread processing adjacent elements. The expression in (A) allows such a pattern.
Quiz 5 Question 2

Continuing from Question 1, what would be the correct statement for each thread to process the second element?

(A) If (i<n) C[i+1] = A[i+1] + B[i+1];
(B) If (i+1<n) C[i+1] = A[i+1] + B[i+1];
(C) If (i+threadIdx.x < n) C[i+threadIdx.x] = A[i+threadIdx.x] + B[i+threadIdx.x];
(D) if(i+blockDim.x < n) C[i+blockDim.x] = A[i+blockDim.x] + B[i+blockDim.x];
Quiz 5 Question 2

Answer: (D)
Explanation: all threads should be shifting to the next half of the section. Thus everyone should be moving to the element that is of BlockDim.x elements away from the first element it processed. The expression is really (blockIdx.x*blockDim.x)*2 + threadDim.x + threadIdx.x.