Agenda

- Questions from previous lectures.

- CUDA Threads, Warps, and Scheduling.

- Synchronization.

- Atomic Functions.
THREAD EXECUTION AND THREAD DIVERGENCE
CUDA Thread Execution

- Explanation of how a streaming multiprocessor executes threads in a thread block

- Single-Instruction Multiple-Thread (SIMT) execution model and performance model

- Control Flow Divergence
Scheduling Thread Blocks

- Hardware dispatches thread blocks to available processor (streaming multiprocessor)
A GPU has lots of processors (streaming multiprocessors)
- The GPUs found in contemporary HPC clusters usually have 14-16

Each processor (streaming multiprocessors) can execute multiple blocks concurrently
- Programmers need to ensure that kernel launches creates enough thread blocks to keep machine busy

Hardware dispatches a block when resources become available, typically when a previous block completes
- No specific order in which blocks are dispatched and executed
- Design algorithms to be insensitive to block execution order
Thread Blocks are Executed as Warps

- Each thread block is mapped to one or more warps
  - When the thread block size is not a multiple of the warp size, unused threads within the last warp are disabled automatically

- The hardware schedules each warp independently
  - Warps within a thread block can execute independently
Warps and SIMT

- A warp is a group of threads within a block that are launched together and (usually) execute together

Conceptual Programming Model

Control  Control  Control  Control  Control  Control

ALU  ALU  ALU  ALU  ALU  ALU

Conceptual SIMT Execution Model

Control

ALU  ALU  ALU  ALU  ALU  ALU  ALU
Warps and SIMT

- **SIMT = Single Instruction Multiple Threads**
  - Within CUDA context, refers to issuing a single instruction to the (multiple) threads in a warp.

- The warp size is currently 32 threads
- The warp size could change in future GPUs

- While we are on the topic of warp size
  - Some code one will encounter relies on the warp size being 32 threads, and so you may notice the constant 32 in code
  - In general, it is poor form to exploit the fact that a warp consists of 32 threads that (usually) execute in lock-step
  - Code may not be portable to future architectures
Thread and Warp Scheduling

- The processors (streaming multiprocessors) can switch between warps with no apparent overhead.
- Warps with instruction whose inputs are ready are eligible to execute, and will be considered when scheduling.
- When a warp is selected for execution, all (active) threads execute the same instruction.
Filling Warps

- Prefer thread block sizes that result in mostly full warps
  - **Bad**: `kernel<<<N, 1>>> ( ... )`
  - **Okay**: `kernel<<<N / 32, 32>>> ( ... )`
  - **Better**: `kernel<<<N / 128, 128>>> ( ... )`

- Prefer to have enough threads per block to provide hardware with many warps to switch between
  - This is how the GPU hides memory access latency

- Resource like `__shared__` may constrain threads per block
  - Algorithm and decomposition will establish some preferred amount of shared data and `__shared__` allocation
Filling Warps

- When number of threads is not a multiple of preferred block size, insert bounds test into kernel

  ```c
  __global__ void kn(int n, int* x)
  {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i < n)
    {
      // very important code
    }
  }
  ```

- Otherwise, threads may access memory outside of arrays

- Do not launch a second grid to process residual elements

  ```c
  kernel<<<n/128, 128>>>(...);
  kernel<<<1, n % 128>>>(...); // !!! very bad !!!
  ```
Control Flow Divergence

- Consider the following code

```c
__global__ void odd_even(int n, int* x)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (((i & 0x01) == 0))
    {
        x[i] = x[i] + 1;
    }
    else
    {
        x[i] = x[i] + 2;
    }
}
```

- Half the threads in the warp must execute the if clause, the other half the else clause
Control Flow Divergence

- The system automatically handles control flow divergence, conditions in which threads within a warp execute different paths through a kernel.

- Often, this requires that the hardware execute multiple paths through a kernel for a warp
  - For example, both the if clause and the corresponding else clause
Control Flow Divergence

__global__ void kv(int* x, int* y) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int t;
    bool b = f(x[i]);
    if (b) {
        // g(x)
        t = g(x[i]);
    } else {
        // h(x)
        t = h(x[i]);
    }
    y[i] = t;
}
Control Flow Divergence

- Nested branches are handled similarly
  - Deeper nesting results in more threads being temporarily disabled

- In general, one does not need to consider divergence when reasoning about the correctness of a program
  - Certain code constructs, such as those involving schemes in which threads within a warp spin-wait on a lock, can cause deadlock.
  - However, most programmers are unlikely to be tempted to code such constructs.

- In general, one does need to consider divergence when reasoning about the performance of a program
Performance of Divergent Code

- Performance decreases with degree of divergence in warps

```c
__global__ void dv(int* x)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    switch (i % 32)
    {
        case 0 : x[i] = a(x[i]);
        break;
        case 1 : x[i] = b(x[i]);
        break;
        ...
        case 31: x[i] = v(x[i]);
        break;
    }
}
```
Compiler and hardware can detect when all threads in a warp branch in the same direction
- For example, all take the if clause, or all take the else clause
- The hardware is optimized to handle these cases without loss of performance

The compiler can also compile short conditional clauses to use predicates (bits that conditional convert instructions into null ops)
- Avoids some branch divergence overheads, and is more efficient
- Often acceptable performance with short conditional clauses
Data Address Divergence

- Concept is similar to control divergence and often conflated

- Hardware is optimized for accessing contiguous blocks of global memory when performing loads and stores
  - Global memory blocks are aligned to multiples of 32, 64, 128 bytes
  - If requests from a warp span multiple data blocks, multiple data blocks will be fetched from memory
  - Entire block is fetched even if only a single byte is accessed, which can waste bandwidth

- Hardware handles divergence within `__shared__` memory more efficiently
  - Designed to support parallel accesses from all threads in warp
  - Still need to worry about addresses that map to the same bank
Data Address Divergence

- Hardware may need to issue multiple loads and stores when a warp accesses addresses that are far apart
  - Conceptually similar to executing the load or store multiple times

- Global memory accesses are most efficient when all load and store addresses generated within a warp are within the same memory block
  - For example, when addresses of loads and stores have stride 1 within a warp
  - Common when array index is a linear function of `threadIdx.x`

- Consider both address and control divergence when designing algorithms and optimizing code
Primordial CUDA Pattern: Blocking

- Partition data to operate in well-sized blocks
  - Small enough to be staged in shared memory
  - Assign each data partition to a thread block
  - No different from cache blocking!

- Provides several significant performance benefits
  - Have enough blocks to keep processors busy
  - Working in shared memory reduces memory latency dramatically
  - More likely to have address access patterns that coalesce well on load/store to shared memory
Partition data into subsets that fit into __shared__ memory
Primordial CUDA Pattern: Blocking

- Process each data subset with one thread block
Primordial CUDA Pattern: Blocking

- Load the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism
Primordial CUDA Pattern: Blocking

- Perform the computation on the subset from shared memory
Primordial CUDA Pattern: Blocking

- Copy the result from __shared__ memory back to global memory
Primordial CUDA Pattern: Blocking

- Almost all CUDA kernels are built this way
  - Blocking may not impact the performance of a particular problem, but one is still forced to think about it
  - Not all kernels require __shared__ memory
  - All kernels do require registers

- Most high-performance CUDA kernels one encounters exploit blocking in some fashion
Questions about Threads and Divergence?
SYNCHRONIZATION
Synchronization

- Communication
- Race conditions
- Synchronizing accesses to shared data
Global Communication

- Device threads communicate through shared memory locations

- Threads in different blocks and different grids
  - Locations in global memory (global variables)

- Threads in same blocks
  - Locations in global memory
  - Locations in shared memory (`__shared__` variables)
Race Conditions

- Race conditions arise when 2+ threads attempt to access the same memory location concurrently and at least one access is a write.

```c++
// race.cu
__global__ void race(int* x)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  *x = i;
}

// main.cpp
int x;
race<<<1,128>>>(d_x);
cudaMemcpy(&x, d_x, sizeof(int), cudaMemcpyDeviceToHost);
```
Race Conditions

- Programs with race conditions may produce unexpected, seemingly arbitrary results
  - Updates may be missed, and updates may be lost

```c
// race.cu
__global__ void race(int* x)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    *x = *x + 1;
}

// main.cpp
int x;
race<<<1,128>>>>(d_x);
cudaMemcpy(&x, d_x, sizeof(int), cudaMemcpyDeviceToHost);
```
Synchronization

- Accesses to shared locations need to be correctly synchronized (coordinated) to avoid race conditions.

- In many common shared memory multithreaded programming models, one uses coordination objects such as locks to synchronize accesses to shared data.

- CUDA provides several scalable synchronization mechanisms, such as efficient barriers and atomic memory operations.

- In general, always most efficient to design algorithms to avoid synchronization whenever possible.
Synchronization

- Assume thread T1 reads a value defined by thread T0

```c
// update.cu
__global__ void update_race(int* x, int* y)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i == 0) *x = 1;
    if (i == 1) *y = *x;
}
```

```c
// main.cpp
update_race<<<1,2>>>(d_x, d_y);
cudaMemcpy(&y, d_y, sizeof(int), cudaMemcpyDeviceToHost);
```

- Program needs to ensure that thread T1 reads location after thread T0 has written location.
Synchronization within Block

- Threads in same block: can use `__syncthreads()` to specify synchronization point that orders accesses

```c
// update.cu
__global__ void update(int* x, int* y)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i == 0) *x = 1;
    __syncthreads();
    if (i == 1) *y = *x;
}
```

```c
// main.cpp
update<<<1,2>>>(d_x, d_y);
cudaMemcpy(&y, d_y, sizeof(int), cudaMemcpyDeviceToHost);
```

- Important: all threads within the block must reach the `__syncthreads()` statement
Synchronization between Grids

- Threads in different grids: system ensures writes from kernel happen before reads from subsequent grid launches.

```c
// update.cu
void update_x(int* x, int* y)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i == 0) *x = 1;
}

void update_y(int* x, int* y)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i == 1) *y = *x;
}

// main.cpp
update_x<<<1,2>>>(d_x, d_y);
update_y<<<1,2>>>(d_x, d_y);
cudaMemcpy(&y, d_y, sizeof(int), cudaMemcpyDeviceToHost);
```
Synchronization within Grid

- Often not reasonable to split kernels to synchronize reads and writes from different threads to common locations
  - Values of \texttt{__shared__} variables are lost unless explicitly saved
  - Kernel launch overhead is non-trivial, and introducing extra launches can degrade performance

- CUDA provides \textbf{atomic functions} (commonly called atomic memory operations) to enforce atomic accesses to shared variables that may be accessed by multiple threads

- Programmers can synthesize various coordination objects and synchronization schemes using atomic functions.
ATOMICS
Introduction to Atomics

- Atom memory operations (atomic functions) are used to solve all kinds of synchronization and coordination problems in parallel computer systems.

- General concept is to provide a mechanism for a thread to update a memory location such that the update appears to happen atomically (without interruption) with respect to other threads.

- This ensures that all atomic updates issued concurrently are performed (often in some unspecified order) and that all threads can observe all updates.
Atomic Functions

- Atomic functions perform read-modify-write operations on data residing in global and shared memory

```c
__global__ void update(unsigned int* x)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int j = atomicAdd(x, 1);  // j = *x; *x = j + i;
}
```

- Atomic functions guarantee that only one thread may access a memory location while the operation completes

```c
// main.cpp
int x = 0;
cudaMemcpy(d_x, x, cudaMemcpyHostToDevice);
update<<<1,128>>>
; cudaMemcpy(&x, d_x, cudaMemcpyHostToDevice);
```
Atomic Functions

- Synopsis of atomic function `atomicOP(a,b)` is typically

```c
  t1 = *a;       // read
  t2 = t1 OP b;  // modify
  *a = t2;       // write
  return t;
```

- The hardware ensures that all statements are executed atomically without interruption by any other atomic functions.
- The atomic function returns the initial value, not the final value, stored at the memory location.
Atomic Functions

- The name atomic is used because the update is performed atomically: it cannot be interrupted by other atomic updates.
- The order in which concurrent atomic updates are performed is not defined, and may appear arbitrary.
- However, none of the atomic updates will be lost.
- Many different kinds of atomic operations
  - Add (add), Sub (subtract), Inc (increment), Dec (decrement)
  - And (bit-wise and), Or (bit-wise or), Xor (bit-wise exclusive or)
  - Exch (Exchange)
  - Min (Minimum), Max (Maximum)
  - Compare-and-Swap
// Compute histogram of colors in an image
//
//  color – pointer to picture color data
//  bucket – pointer to histogram buckets, one per color
//
__global__ void histogram(int n, int* color, int* bucket)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  if (i < n)
  {
    int c = colors[i];
    atomicAdd(&bucket[c], 1);
  }
}
Work Queue Example

// For algorithms where the amount of work per item
// is highly non-uniform, it often makes sense for
// to continuously grab work from a queue
__device__ int do_work(int x)
{
    return f(x-1) + f(x) + f(x+1);
}

__global__ void process_work_q(int* work_q, int* q_counter, int* output, int queue_max)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int q_index = atomicInc(q_counter, queue_max);
    int result = do_work(work_q[q_index]);
    output[i] = result;
}
Performance Notes

- Atomics are slower than normal accesses (loads, stores)
- Performance can degrade when many threads attempt to perform atomic operations on a small number of locations
- Possible to have all threads on the machine stalled, waiting to perform atomic operations on a single memory location.
Example: Global Min/Max (Naive)

- Compute maximum across all threads in a grid
- One can use a single global maximum value, but it will be VERY slow.

```c
__global__ void global_max(int* values, int* global_max)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int val = values[i];
    atomicMax(global_max, val);
}
```
Example: Global Min/Max (Better)

- Introduce local maximums and update global only when new local maximum found.

```c
__global__ void global_max(int* values, int* global_max,
                           int *local_max, int num_locals)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  int val = values[i];
  int li = i % num_locals;
  int old_max = atomicMax(&local_max[li], val);
  if (old_max < val)
  {
    atomicMax(global_max, val);
  }
}
```

- Reduces frequency at which threads attempt to update the global maximum, reducing competition access to location.
Lessons from global Min/Max

- Many updates to a single value causes serial bottleneck
- One can create a hierarchy of values to introduce more parallelism and locality into algorithm
- However, performance can still be slow, so use judiciously
Important note about Atomics

- Atomic updates are not guaranteed to appear atomic to concurrent accesses using loads and stores

```c
__global__ void broken(int n, int* x)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (n == 0)
    {
        *x = *x + 1;
    }
    else
    {
        int j = atomicAdd(x, 1); // j = *x; *x = j + i;
    }
}

// main.cpp
broken<<<1,128>>>(128, d_x); // d_x = d_x + {1, 127, 128}
```
Summary of Atomics

- Cannot use normal load/store for reliable inter-thread communication because of race conditions

- Use atomic functions for infrequent, sparse, and/or unpredictable global communication

- Decompose data (very limited use of single global sum/max/min/etc.) for more parallelism

- Attempt to use shared memory and structure algorithms to avoid synchronization whenever possible
Questions on Atomics?