Compsci 590.3: Introduction to Parallel Computing

Alvin R. Lebeck

Some slides based on those from NVIDIA
Admin

Logistics
- Homework #3
- Projects
  - Be creative
  - Email me team members (names and NetIDs)
- Project Proposals
  - Document: see web site
    - Due 10am Wed 10/7
  - Lightning talk
    - 10-11 Groups
    - One person per group
    - 5 minutes / group
    - 2-3 slides, no more!
    - Send to me PPT by Tuesday @ 8pm

Outline
- GPU architecture
  - Shared memory (scratchpad)
  - Thread synchronization
- CUDA streams
- Fork Join
CONCEPTS

Heterogeneous Computing
Blocks
Threads
Indexing
Shared memory
__syncthreads()
Asynchronous operation
Handling errors
Managing devices

COOPERATING THREADS
1D Stencil

- Consider applying a 1D stencil to a 1D array of elements
  - Each output element is the sum of input elements within a radius

- If radius is 3, then each output element is the sum of 7 input elements:
Implementing Within a Block

• Each thread processes one output element
  - `blockDim.x` elements per block

• Input elements are read several times
  - With radius 3, each input element is read seven times
Sharing Data Between Threads

- Terminology: within a block, threads share data via shared memory
- Extremely fast on-chip memory, user-managed
  - Not the same as general purpose multicore shared memory!
- Declare using __shared__, allocated per block
- Data is not visible to threads in other blocks
Implementing With Shared Memory

- Cache data in shared memory
  - Read \((\text{blockDim.x} + 2 \times \text{radius})\) input elements from global memory to shared memory
  - Compute \(\text{blockDim.x}\) output elements
  - Write \(\text{blockDim.x}\) output elements to global memory

- Each block needs a **halo** of \(\text{radius}\) elements at each boundary
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[blockDim.x + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[lindex] = in[gindex];
  if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + blockDim.x] = in[gindex + blockDim.x];
  }

  // Apply the stencil
  int result = 0;
  for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];

  // Store the result
  out[gindex] = result;
}

- Which threads copy the halo into shared memory?
- Is this a correct implementation?
Data Race!

- The stencil example will not work...
- Suppose thread 15 reads the halo before thread 0 has fetched it...
  - How can that happen?

```c
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];  // Skipped, threadIdx > RADIUS
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}
int result = 0;
result += temp[lindex + 1];
```

- Store at temp[18]
- Load from temp[19]
__syncthreads()  

- void __syncthreads();

- Synchronizes all threads within a block
  - Used to prevent RAW / WAR / WAW hazards
  - It’s a barrier synchronization within a block…

- All threads in the block must reach the barrier
  - In conditional code, the condition must be uniform across the block
  - All threads in the block must call __synchthreads()
__global__ void stencil_1d(int *in, int *out) {
__shared__ int temp[blockDim.x + 2 * RADIUS];

int gindex = threadIdx.x + blockIdx.x * blockDim.x;
int lindex = threadIdx.x + RADIUS;

// Read input elements into shared memory
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + blockDim.x] = in[gindex + blockDim.x];
}

// Synchronize (ensure all the data is available)
__syncthreads();

// Apply the stencil
int result = 0;
for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];

// Store the result
out[gindex] = result;
}
But What About Synchronization Across Blocks?

• \_syncthreads() is within a block

• If problem needs synch across blocks

• Best approach is to divide the kernel into two and execute second only after first completes.

• How do we do that given what we know so far?
CUDA Streams

Motivation
• Dynamic work
• Problems that are small but can still use the GPU
• Want to use both CPU and GPU
• Want one kernel to start after the previous finishes

Solution
• Stream is a queue of commands issued to the GPU that execute in sequential order
• Operations from different streams are not ordered
CUDA Streams

Solution

• Stream is a queue of commands issued to the GPU that execute in sequential order

• Operations from different streams are not ordered

• Default stream (0), barrier synchronization for streams
  ▪ Command on default stream waits for currently executing commands to finish before starting
  ▪ Subsequent commands in later streams wait for default command to finish

• Stream is additional argument to kernel launch
  kernel<<<blocks, threads, bytes>>>( ); // default stream
  kernel<<<blocks, threads, bytes, 0 >>>(); // stream 0
CUDA Stream Behavior

• Synchronous copy and host execution, synchronous device execution
  1. cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);
  2. increment<<<1,N>>>(d_a)
  3. cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
  4. cudaDeviceSynchronize();  // wait for things to finish

• Operations are sent to the same (default) stream and thus execute in order.
CUDA Stream Behavior

- Can still do other work on CPU while kernel is executing

1. cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);
2. increment<<<1,N>>>(d_a)
3. myCpuFunction(b)
4. cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
5. cudaDeviceSynchronize(); // wait for things to finish

- Host executes 3 while 2 is executing on the GPU
- What if data is so large it doesn’t fit on the device? What can we do?
Non-default Streams

• Declare, create and destroy a stream
  1. \texttt{cudaStream_t stream1;}
  2. \texttt{cudaError_t result;}
  3. \texttt{result = cudaMemcpyAsync(d_a, a, numBytes, cudaMemcpyHostToDevice, stream1);}
  4. \texttt{result = cudaMemcpyAsync(d_a, a, numBytes, cudaMemcpyHostToDevice, stream1);}

• Specifying which stream to use
  1. \texttt{cudaMemcpyAsync(d_a, a, numBytes, cudaMemcpyHostToDevice, stream1);}
  2. \texttt{increment<<<1,N,0,stream1>>>(d_a)
Stream Synchronization

• When you need to synchronize host code with the device
• Option 1
  ▪ `cudaDeviceSynchronize()`
  ▪ Blocks host until all previous issued streams finish
  ▪ Too heavy weight (stops everything)
• Option 2
  ▪ `cudaStreamSynchronize(stream)`
  ▪ Blocks host until all previous issued commands to `stream` finish
  ▪ `cudaStreamQuery(stream)`
  ▪ Checks all previous issued commands to `stream` finish doesn’t block host
• Option 3
  ▪ `cudaEventSynchronize(event)`
  ▪ `cudaEventQuery(event)`
  ▪ Waits/checks for event to finish
Overlapping Operations

• Can do other work on CPU while kernel is executing
• Loop over chunks of a large structure

1. for (int i=0; i < nstreams; i++) {
2.   int offset = i * streamSize;
3.   cudaMemcpyAsync(&d_a[offset], &a[offset], streamBytes, cudaMemcpyHostToDevice, stream[i]);
4.   increment<<<streamSize/blockSize, blockSize, 0, stream[i]>>>(d_a, offset)
5.   cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
6. }

• Overlap 3, 4 & 5 from different streams (iterations)!
• Performance depends on architecture (need two memory copy engines)
Overlapping Operations

- Overlap 3, 4 & 5 from different streams (iterations)!
- Performance depends on architecture (need two memory copy engines)
- Assume 4 streams
Summary

• Vector execution
• Multithreaded execution
• SIMT execution
• CUDA Shared Memory
• CUDA Streams