Compsci 590.3: Introduction to Parallel Computing

Alvin R. Lebeck

Some slides from NVIDIA and couple slides from sources that include Profs. Roth, Martin, Asanovic, Hill, Lebeck, Sorin Falsafi, Hoe, Lipasti, Shen, Smith, Sohi, Vijaykumar, Wood, and Lee
Admin

Logistics
• Office hours start @ 2:15 today
• Homework #3
• Projects
  ▪ Be creative
  ▪ Email me team members (names and NetIDs)

Outline
• Vector/SIMD architecture
• GPU architecture
  ▪ Shared memory (scratchpad)
  ▪ Thread synchronization
• CUDA streams
Scalar Execution

- **Instructions**: a fixed set of built-in operations
- **Fetch-Execute Cycle**
  
  ```c
  while (!done)
    fetch instruction
    execute instruction
  ```

- This is done by the hardware for speed
- **One data item per instruction, how can hardware do better?**
Vector/SIMD Execution

• Fetch and execute one vector/SIMD instruction but operate on multiple data items!

• Fetch-Execute Cycle

while (!done)
    fetch instruction
    execute instruction

© Alvin R. Lebeck  some slides from NVIDIA  Compsci 590.3 Parallel Computing
Multiple Data Lanes

- Vector elements interleaved across lanes
  - Example: V[0, 4, 8, …] on Lane 1, V[1, 5, 9,…] on Lane 2, etc.

- Compute for multiple elements per cycle
  - Example: Lane 1 computes on V[0] and V[4] in one cycle

- Modular, scalable design
- No inter-lane communication needed for most vector instructions

© Alvin R. Lebeck  some slides from NVIDIA
Compsci 550 / ECE 552 (Lebeck)
Conditional Execution

• Suppose you want to vectorize this code:
  - for (i=0 ; i<N ; i++) {
    - if(A[i] != B[i]) {A[i] -= B[i]; } }

• Solution: vector conditional execution
  - Add vector flag registers, single-bit mask per vector element
  - Use vector-compare to set the vector flag register
  - Use vector flag register to control vector-subtract
  - Vector op executed only if corresponding flag element is set

  - vld V1, Ra
  - vld V2, Rb
  - vcmp.neq.vv M0, V1, V2 # vector compare for mask
  - vsub.vv V3, V2, V1, M0 # conditional vadd
  - vst V3, Ra
Vector/SIMD Execution

- Fetch and execute one vector/SIMD instruction but operate on multiple data items!

- Fetch-Execute Cycle
  
  ```
  while (!done)
      fetch instruction
      execute instruction
  ```

- Gaps where we are doing anything, can we fill those gaps? How?
Multithreaded Execution

• Fetch and execute one instruction but operate on multiple data items!

• Fetch-Execute Cycle

\[
\text{while (!done)} \\
\text{fetch thread i instruction} \\
\text{execute instruction} \\
i = i + 1 \mod \#HW\ threads
\]

• Example: 2 threads 4-wide vector
Processor Vector/SIMD support

- CPU scalar units and vector units w/ width of 4-32 (double to byte).
  - Phi vector widths up to 64
- GPU uses wide SIMD: 8/16/24/32/64… PEs (same as functional unit)
- Which parts of the processor are used by all threads?
Hardware Support

- Supporting interleaved threading + SIMD execution

© Alvin R. Lebeck  some slides from NVIDIA
Comp sci 590.3 Parallel Computing
Single Instruction Multiple Thread (SIMT)

- Hide vector width using scalar threads.
Example of SIMT Execution

- Assume 32 threads are grouped into one warp.
SMX (Streaming Multiprocessor)

- 320 total units
  - 192 cores (FU)
    - Floating point multiply+add (FMA)
  - 64 Double precision
  - 32 SPUs
  - 32 Ld/st for memory
- 64K registers
  - 255/thread
- 2048 threads
- 4 warp schedulers
  - 2 inst/warp
  - 256 ops/cycle/SMX
- L1 can also be scratchpad (shared among threads)

Source: NVIDIA Kepler Architecture White Paper
© Alvin R. Lebeck  some slides from NVIDIA
CompSci 590.3 Parallel Computing
Divergence

• Threads in a warp want to do different things….

• Control Divergence
  ▪ Want to issue same instruction for all threads in warp, but…
  ▪ Conditional Branches
  ▪ Some threads execute if block others else block

• Memory Address Divergence
  ▪ Variability in delay of memory accesses
  ▪ Need all load responses before warp can execute dependent instruction
  ▪ E.g., accesses to different cache blocks (no spatial locality)

   int a[N]; // contains random values
   if (a[idx] < pivot)
     // if code block
   else
     // else code block
   a[(32*idx) % N]

Addresses 0, 32, 64, 96, 128 …
Stencil Pattern

• A stencil pattern is a map where each output depends on a “neighborhood” of inputs
• These inputs are a set of fixed offsets relative to the output position
• A stencil output is a function of a “neighborhood” of elements in an input collection
  ▪ Applies the stencil to select the inputs
• Data access patterns of stencils are regular
  ▪ Stencil is the “shape” of “neighborhood”
  ▪ Stencil remains the same
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

![Diagram showing cache data in shared memory with halo elements at boundaries]
__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?
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];
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}
int result = 0;
result += temp[lindex + 1];
```

Store at temp[18]
Skipped, threadIdx > RADIUS
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…

- All threads must reach the barrier
  - In conditional code, the condition must be uniform across the block
  - All threads 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;
}
CUDA Streams

Motivation
• Dynamic work
• Problems that are small but can still use the GPU
• Want to use both CPU and GPU

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. `cudaStream_t stream1;`
2. `cudaError_t result;`
3. `result = cudaMemcpyAsync(&stream1)`
4. `result = cudaMemcpyAsync(stream)`

• Specifying which stream to use
1. `result = cudaMemcpyAsync(d_a, a, numBytes, cudaMemcpyHostToDevice, stream1);`
2. `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

© Alvin R. Lebeck  some slides from NVIDIA

Compseci 590.3 Parallel Computing
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)