Lecture 6

Programming with Message Passing
Message Passing Interface (MPI)
Announcements
Today’s lecture

- Finish CUDA
- Programming with message passing
Constraints

- **SM**
  - Up to 8 resident blocks
  - Not more than 1024 threads
  - Up to 32 warps
- All threads in a warp execute the same instruction
  - All branches followed
  - Instructions disabled
  - Divergence, serialization
- Grid: 1 or 2-dimensional (64K-1)
- Blocks – 1, 2, or 3-dimensional
  - ≤ 512 threads
  - Max dimensions: 512, 512, 64
  - Registers subdivided over threads
  - Synchronization among all threads in the block
- Synchronization only within block
Fermi

- Running incrementArray on cseclass
  - Device 0 has **15** cores
  - Device is a GeForce GTX 570, capability: **2.0**
- Improvements
  - Cache
  - Higher peak double precision performance (on some models)
  - Increases the number of cores (448 to 512)
  - Dual thread schedulers
  - Concurrent kernel execution (some models)
  - Reduced kernel launch overhead (25 µs)
  - Improved predication
Fermi Block Diagram
Shared memory banks

• Any memory load or store of $n$ addresses that spans $n$ distinct memory banks can be serviced simultaneously, yielding an effective bandwidth that is $n$ times as high as the bandwidth of a single bank.

• If multiple addresses of a memory request map to the same memory bank, the accesses are serialized. The hardware splits a memory request that has bank conflicts into as many separate conflict-free requests as necessary. Exception: if all access the same bank: broadcast.

• Exception: all threads in a half warp address the same shared memory location, resulting in a broadcast.

• Devices of compute capability 2.x have the additional ability to multicast shared memory accesses.

• See *CUDA C Best Practices Guide Version 3.2*
Shared memory design

- Successive 32-bit words assigned to successive banks
- For devices of compute capability 1.x
  - Number of banks = 16
  - Bandwidth is 32 bits per bank per clock cycle
  - Shared memory request for a warp is split in two
  - No conflict occurs if only one memory location per bank is accessed by a half warp of threads
- For devices of compute capability 2.x [Fermi]
  - Number of banks = 32
  - Bandwidth is 32 bits per bank per 2 clock cycles
  - Shared memory request for a warp is not split
  - Increased susceptibility to conflicts
  - But no conflicts if access to bytes in same 32 bit word
  - Unlike 1.x, no bank conflicts here:

```c
__shared__ char shared[32];
char data = shared[BaseIndex + tid];
```
Shared Memory/Cache

- L1 cache for each vector unit
- L2 cache shared by all vector units
- Cache accesses to local or global memory, including temporary register spills
- Cache inclusion ($L1 \subseteq L2$?) partially configurable on per-access basis with mem. ref. instruction modifiers
- On-chip memory: partially shared memory, partially L1
  - 16KB + 48 KB  shared memory + L1
  - 48KB + 16 KB  shared memory + L1
- 128 byte cache line size
Global Memory

- 128 byte cache line size
- If accessed word > 4 bytes, warp’s memory request split into separate, independently issued 128-byte memory requests
- Non-atomic, concurrent writes within a warp: writer not defined
Shared memory on Fermi

• Successive 32-bit words assigned to successive banks
• The bandwidth of shared memory is 32 bits per bank per clock 2 cycle
• For devices of compute capability 1.x
  • Number of banks = 16
  • Shared memory request for a warp is split in two
  • No conflict occurs if only one memory location per bank is accessed by a half warp of threads
• For devices of compute capability 2.x
  • Number of banks = 32
  • Shared memory request for a warp is not split
  • Increased susceptibility to conflicts
Vector units

- Each vector unit
  - 32 CUDA cores for integer and floating-point arithmetic
  - 4 special function units for Single Precision transcendental functions
  - FMA without truncation (32 or 64 bits)
- For devices of compute capability 2.1
  - 48 CUDA cores for arithmetic operations
  - 8 special function units for single-precision
- *CUDA C Programming Guide Version 3.2, §4.3*
Instruction issue

• Each vector unit
  • 32 CUDA cores for integer and floating-point arithmetic
  • 4 special function units for Single Precision transcendentals
• 2 Warp schedulers: each scheduler issues: 1 (2) instructions for capability 2.0 (2.1)
• One scheduler in charge of odd warp IDs, the other even warp IDs
• Only 1 scheduler can issue a double-precision floating-point instruction at a time
• Warp scheduler can issue an instruction to $\frac{1}{2}$ the CUDA cores
• Scheduler must issue the instruction over 2 clock cycles for an integer or floating-point arithmetic instruction
Concurrency – Host & Device

- Nonbocking operations
  - Kernel launch
  - Device← {Host,Device}
  - Async memory copies
- Multiple kernel invocations: certain capability 2.x devices
- CUDA Streams (§3.2.6.5), with limitations
  ```c
  cudaStream_t stream[2];
  for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);
  for (int i = 0; i < 2; ++i)
    Kernel<<<100, 512, 0, stream[i]>>> ( ... );
  ```
Predication on Fermi

- All instructions support predication in 2.x
- Condition code or *predicate* per thread:
  - set to true or false
- Execute only if predicate is true
- Compiler replaces a branch instruction with predicated instructions only if the number of instructions controlled by branch condition is not too large
- If the compiler predicts too many divergent warps….
  - threshold = 7, else 4
Reduction
Warp scheduling

- Blocks are divided into *warps* of 32 (SIMD) threads which are ….
- Subdivided into schedulable units: 16 threads (→ 32 on Fermi)
- Warps are scheduled with zero overhead in hardware
- Scheduler finds an eligible warp: all operands are ready
  - Scoreboarding
  - Priorities
- Branches serialize execution within a warp
Thread Divergence

• Blocks are divided into warps of 32 threads
• All the threads in a warp execute the same instruction
• Different control paths are serialized
• Divergence when predicate is a function of the threadId
  
  if (threadId < 2) { }

• No divergence if all follow the same path
  
  if (threadId / WARP_SIZE < 2) { }

• Consider reduction, e.g. summation $\sum_i x_i$
A naïve reduction

Thread 0  Thread 2  Thread 4  Thread 6  Thread 8  Thread 10

0  1  2  3  4  5  6  7  8  9  10  11

0+1  2+3  4+5  6+7  8+9  10+11

0...3  4..7  8..11  8..15

0...7  4...7  8...11  8...15

DavidKirk/NVIDIA & Wen-mei Hwu/UIUC

© 2011 Scott B. Baden / CSE 262 / Spring 2011
The naïve code

```c
__global__ void reduce(int *input, unsigned int N, int *total){
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
    __shared__ int x[BSIZE];
    x[tid] = (i<N) ? input[i] : 0;
    __syncthreads();

    for (unsigned int stride = 1; stride < blockDim.x; stride *= 2) {
        __syncthreads();
        if (tid % (2*stride) == 0)
            x[tid] += x[tid + stride];
    }
    if (tid == 0) atomicAdd(total,x[tid]);
}
```
Reducing divergence and avoiding bank conflicts

Thread 0
The Code

```cpp
unsigned int tid = threadIdx.x;
__shared__ int x[BSIZE];

...

for (unsigned int s = blockDim.x / 2; s > 1; s /= 2) {
    __syncthreads();
    if (tid < s)
        x[tid] += x[tid + s];
}
```
Message Passing
Programming with Message Passing

- **The** primary model for implementing parallel applications
- Programs execute as a set of P processes
  - We specify P when we run the program
  - Assume each process is assigned a different physical processor
- Each process
  - is initialized with the same code, but has private state
    - SPMD = “Same Program Multiple Data”
  - executes instructions at its own rate
  - has an associated *rank*, a unique integer in the range 0:P-1
  - may or may not be assigned a different physical processor
- The sequence of instructions each process executes depends on its rank and the messages it sends and receives
- Program execution is often called “bulk synchronous” or “loosely synchronous”
Message Passing

• Messages are like email; to send one, we specify
  ‣ A destination
  ‣ A message body (can be empty)

• To receive a message we need similar information, including a receptacle to hold the incoming data

• Requires a sender and an explicit recipient that must be aware of one another

• Message passing performs two events
  ‣ Memory to memory block copy
  ‣ Synchronization signal on receiving end: “Data arrived”
A minimal interface

• Query functions
  \[ \text{nproc}(\ ) = \# \text{ processors} \]
  \[ \text{myRank}(\ ) = \text{this process's rank} \]

• *Point-to-point* communication
  ‣ Simplest form of communication
  ‣ Send a message to another process
    \[ \text{Send}(\text{Object}, \text{Destination process ID}) \]
  ‣ Receive a message from another process
    \[ \text{Receive}(\text{Object}) \]
    \[ \text{Receive}(\text{Source process, Object}) \]
Send andRecv

- When `Send( )` returns, the message is “in transit”
  - A return doesn’t tell us if the message has been received
  - Somewhere in the system
  - Safe to overwrite the buffer
- `Receive( )` blocks until the message has been received
  - Safe to use the data in the buffer
An unsafe program

- A `Send()` *may or may not* complete…
- … before a `Recv()` has been posted
- “May or may not” depends on the implementation
- Some programs may deadlock on certain message passing implementations

<table>
<thead>
<tr>
<th></th>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Send(x,1)</code></td>
<td></td>
<td><code>Send(y,0)</code></td>
</tr>
<tr>
<td><code>Recv(y,1)</code></td>
<td></td>
<td><code>Recv(x,0)</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Send(x,1)</code></td>
<td><code>Recv(x,0)</code></td>
<td></td>
</tr>
<tr>
<td><code>Recv(y,1)</code></td>
<td><code>Send(y,0)</code></td>
<td></td>
</tr>
</tbody>
</table>
Buffering

- Where does the data go when you send it?
- It might be buffered
- Preferable to avoid the extra copy
Causality

- If a process sends multiple messages to the same destination, then the messages will be received in the order sent.
- If different processes send messages to the same destination, the order of receipt isn’t defined across processes.
Causality

• If different processes send messages to the same destination
  ‣ The order of receipt is defined from a single source
  ‣ The order of receipt is not defined across multiple sources
Asynchronous, non-blocking communication

• Immediate return, does not wait for completion
  ‣ Required to express certain algorithms
  ‣ Optimize performance: message flow problems

• Split-phased
  ‣ Phase 1: initiate communication with the immediate ‘I’ variant of the point-to-point call
    \texttt{IRcvc( ), ISend( )}
  ‣ Phase 2: synchronize
    \texttt{Wait( )}
  ‣ Perform unrelated computations between the two phases

• Building a blocking call
  \texttt{Recv( ) = IRecv( ) + Wait( )}
Restrictions on non-blocking communication

- The message buffer may not be accessed between an \texttt{IRecv()} (or \texttt{ISend()}) and its accompanying \texttt{Wait()}

\texttt{ISend(data,destination)} \\
\texttt{Wait()} on \texttt{ISend()}

Use the data

- Each pending \texttt{IRecv()} must have a distinct buffer
### Overlap behavior

<table>
<thead>
<tr>
<th>No Overlap</th>
<th>Overlap</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>IRrecv(x, req)</code></td>
<td><code>Recv(x)</code></td>
</tr>
<tr>
<td><code>Send(..)</code></td>
<td><code>Send(…)</code></td>
</tr>
<tr>
<td><code>Compute(y)</code></td>
<td><code>Compute(x)</code></td>
</tr>
<tr>
<td><code>Wait(req)</code></td>
<td><code>Compute(y)</code></td>
</tr>
<tr>
<td><code>Compute(x)</code></td>
<td></td>
</tr>
</tbody>
</table>

A message buffer may not be accessed between an `IRrecv( )` (or `ISend( )`) and its accompanying `wait( )`