L8: Writing Correct Programs, cont. and Control Flow

Administrative

- Next assignment available
  - Goals of assignment:
    - simple memory hierarchy management
    - block-thread decomposition tradeoff
  - Due Thursday, Feb. 10, 5PM
  - Use handin program on CADE machines
    - "handin cs6963 lab2 <probfile>"
- Project proposals due Wednesday, March 9
  - Questions/discussion
- Mailing lists
  - cs6963s11-discussion@list.eng.utah.edu
    - Please use for all questions suitable for the whole class
    - Feel free to answer your classmates questions!
  - cs6963s1-teach@list.eng.utah.edu
    - Please use for questions to Sriram and me

Questions/comments from previous lectures

- Is there a shared memory bank conflict for when each thread accesses contiguous 8-bit or 16-bit data?
  - YES for compute capability below 2.0, see G3.3 CUDA 3.2 programming guide
  - NO for compute capability 2.0 and greater, see G4.3 CUDA 3.2 programming guide
- GTX 460 and 560 do have 48 cores per SM (7 or 8)
  - Seem to use a different warpsize?

Outline

- Finish discussion of page-locked memory on host
- Control Flow
  - SIMT execution model in presence of control flow
  - Divergent branches
- Improving Control Flow Performance
  - Organize computation into warps with same control flow path
  - Avoid control flow by modifying computation
  - Tests for aggregate behavior (warp voting)
- Read (a little) about this:
  Kirk and Hwu, Ch. 5
  NVDIA Programming Guide, 5.4.2 and B.12
  http://www.realworldtech.com/page.cfm?ArticleID=RWT090808195242&p=1

Host-Device Transfers (implicit in synchronization discussion)

- Host-Device Data Transfers
  - Device to host memory bandwidth much lower than device to device bandwidth
  - 8 GB/s peak (PCI-e x16 Gen 2) vs. 102 GB/s peak (Tesla C1060)
- Minimize transfers
  - Intermediate data can be allocated, operated on, and deallocated without ever copying them to host memory
- Group transfers
  - One large transfer much better than many small ones

Slide source: Nvidia, 2008

Asynchronous Copy To/From Host (compute capability 1.1 and above)

- Warning: I have not tried this!
- Concept:
  - Memory bandwidth can be a limiting factor on GPUs
  - Sometimes computation cost dominated by copy cost
  - But for some computations, data can be "tiled" and computation of tiles can proceed in parallel (some of our projects)
  - Can we be computing on one tile while copying another?
- Strategy:
  - Use page-locked memory on host, and asynchronous copies
  - Primitive **cudaMemcpyAsync**
  - Effect is GPU performs DMA from Host Memory
  - Synchronize with **cudaThreadSynchronize()**

Page-Locked Host Memory

- How the Async copy works:
  - DMA performed by GPU memory controller
  - CUDA driver takes virtual addresses and translates them to physical addresses
  - Then copies physical addresses onto GPU
  - Now what happens if the host OS decides to swap out the page???
- Special malloc holds page in place on host
  - Prevents host OS from moving the page
  - CudaMallocHost()
- But performance could degrade if this is done on lots of pages!
  - Bypassing virtual memory mechanisms

Example of Asynchronous Data Transfer

```
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
cudaMemcpyAsync(dst1, src1, size, dir, stream1);
kernel<<<grid, block, 0, stream1>>>(…);
cudaMemcpyAsync(dst2, src2, size, dir, stream2);
kernel<<<grid, block, 0, stream2>>>(…);
```

src1 and src2 must have been allocated using cudaMallocHost
stream1 and stream2 identify streams associated with asynchronous call (note 4th "parameter" to kernel invocation)

Code from asyncAPI SDK project

```
// allocate host memory
CUDA_SAFE_CALL( cudaMallocHost((void**)&a, nbytes) );
memset(a, 0, nbytes);

// allocate device memory
CUDA_SAFE_CALL( cudaMalloc((void**)&d_a, nbytes) );
CUDA_SAFE_CALL( cudaMemset(d_a, 255, nbytes) );

… // declare grid and thread dimensions and create start and stop events

// asynchronously issue work to the GPU (all to stream 0)
cudaEventRecord(start, 0);
cudaMemcpyAsync(d_a, a, nbytes, cudaMemcpyHostToDevice, 0);
increment_kernel<<<blocks, threads, 0, 0>>>(d_a, value);
cudaMemcpyAsync(a, d_a, nbytes, cudaMemcpyDeviceToHost, 0);
cudaEventRecord(stop, 0);

// have CPU do some work while waiting for GPU to finish

// release resources
CUDA_SAFE_CALL( cudaFreeHost(a) );
CUDA_SAFE_CALL( cudaFree(d_a) );
```


More Parallelism to Come (Compute Capability 2.0)

Stream concept: create, destroy, tag asynchronous operations with stream

- Special synchronization mechanisms for streams: queries, waits and synchronize functions
- Concurrent Kernel Execution
  - Execute multiple kernels (up to 4) simultaneously
- Concurrent Data Transfers
  - Can concurrently copy from host to GPU and GPU to host using asynchronous Memcpy

Section 3.2.6 of CUDA 3.2 manual

Debugging: Using Device Emulation Mode

- An executable compiled in device emulation mode (`nvcc -deviceemu`) runs completely on the host using the CUDA runtime
  - No need of any device and CUDA driver
  - Each device thread is emulated with a host thread
- When running in device emulation mode, one can:
  - Use host native debug support (breakpoints, inspection, etc.)
  - Access any device-specific data from host code and vice-versa
  - Call any host function from device code (e.g. `printf`) and vice-versa
  - Detect deadlock situations caused by improper usage of `__syncthreads`

Debugging: Device Emulation Mode Pitfalls

- Emulated device threads execute sequentially, so simultaneous accesses of the same memory location by multiple threads could produce different results.
- Dereferencing device pointers on the host or host pointers on the device can produce correct results in device emulation mode, but will generate an error in device execution mode
- Results of floating-point computations will slightly differ because of:
  - Different compiler outputs, instruction sets
  - Use of extended precision for intermediate results
    - There are various options to force strict single precision on the host

Debugging: Run-time functions & macros for error checking

In CUDA run-time services,

cudaGetDeviceProperties(deviceProp &dp, d);
check number, type and whether device present

In libcutil.a of Software Developers' Kit,

cutComparef (float *ref, float *data, unsigned len);
compare output with reference from CPU implementation

In cutil.h of Software Developers' Kit (with #define _DEBUG or -D_DEBUG compile flag),

CUDA_SAFE_CALL(f(<args>)), CUT_SAFE_CALL(f(<args>))
check for error in run-time call and exit if error detected

CUT_SAFE_MALLOC(cudaMalloc(<args>));
similar to above, but for malloc calls

CUT_CHECK_ERROR("error message goes here");
check for error immediately following kernel execution and if detected, exit with error message

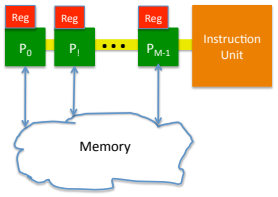
A Very Simple Execution Model

- No branch prediction
  - Just evaluate branch targets and wait for resolution
  - But wait is only a small number of cycles once data is loaded from global memory
- No speculation
  - Only execute useful instructions

SIMD Execution of Control Flow

Control flow example

```
if (threadIdx >= 2) {
   out[threadIdx] += 100;
}
else {
   out[threadIdx] += 10;
}
```

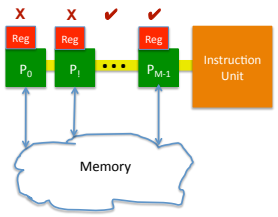


compare threadIdx,2

SIMD Execution of Control Flow

Control flow example

```
if (threadIdx.x >= 2) {
   out[threadIdx.x] += 100;
}
else {
   out[threadIdx.x] += 10;
}
```

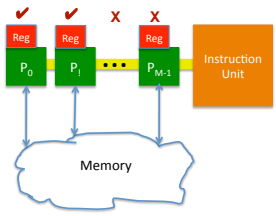



```
/* Condition code cc =
true branch set by
predicate execution */
(CC) LD R5,
        &(out+threadIdx.x)
(CC) ADD R5, R5, 100
(CC) ST R5,
        &(out+threadIdx.x)
```

SIMD Execution of Control Flow

Control flow example

```
if (threadIdx >= 2) {
   out[threadIdx] += 100;
}
else {
   out[threadIdx] += 10;
}
```




```
/* possibly predicated
using CC */
(not CC) LD R5,
            &(out+threadIdx)
(not CC) ADD R5, R5, 10
(not CC) ST R5,
            &(out+threadIdx)
```


Terminology

- Divergent paths
  - Different threads within a warp take different control flow paths within a kernel function
  - N divergent paths in a warp?
    - An N-way divergent warp is serially issued over the N different paths using a hardware stack and per-thread predication logic to only write back results from the threads taking each divergent path.
    - Performance decreases by about a factor of N

How thread blocks are partitioned

- Thread blocks are partitioned into warps
  - Thread IDs within a warp are consecutive and increasing
  - Warp 0 starts with Thread ID 0
- Partitioning is always the same
  - Thus you can use this knowledge in control flow
  - However, the exact size of warps may change from generation to generation
  - (Covered next)
- **However, DO NOT rely on any ordering between warps**
  - If there are any dependences between threads, you must __syncthreads() to get correct results

First Level of Defense: Avoid Control Flow

- Clever example from MPM

$$m_i = \sum_p S_{ip} m_p + 1.0x10^{-100}$$

$$\mathbf{V}_i = \frac{\sum_p S_{ip} m_p \mathbf{V}_p}{m_i}$$

Add small constant to mass so that velocity calculation never divides by zero

- No need to test for divide by 0 error, and slight delta does not impact results

Control Flow Instructions

- A common case: avoid divergence when branch condition is a function of thread ID
  - Example with divergence:
    - `If (threadIdx.x > 2) { }`
    - This creates two different control paths for threads in a block
    - Branch granularity < warp size; threads 0 and 1 follow different path than the rest of the threads in the first warp
  - Example without divergence:
    - `If (threadIdx.x / WARP_SIZE > 2) { }`
    - Also creates two different control paths for threads in a block
    - Branch granularity is a whole multiple of warp size; all threads in any given warp follow the same path

Parallel Reduction Example (related to "count 6")

- Assume an in-place reduction using shared memory
  - The original vector is in device global memory
  - The shared memory is used to hold a partial sum vector
  - Each iteration brings the partial sum vector closer to the final sum
  - The final solution will be in element 0

How to Accumulate Result in Shared Memory

In original implementation (Lecture 1), we collected per-thread results into d_out[threadIdx.x].

In updated implementation (Lecture 7), we collected per-block results into d_out[0] for a single block, thus serializing the accumulation computation on the GPU.

Suppose we want to exploit some parallelism in this accumulation part, which will be particularly important to performance as we scale the number of threads.

A common idiom for reduction computations is to use a tree-structured results-gathering phase, where independent threads collect their results in parallel. Assume SIZE=16 and BLOCKSIZE(elements computed per thread)=4.

Recall: Serialized Gathering of Results on GPU for "Count 6"

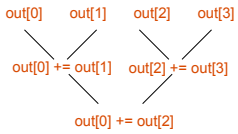
```
__global__ void compute(int *d_in, int *d_out) {

 d_out[threadIdx.x] = 0;

 for (i=0; i<SIZE/BLOCKSIZE; i++) {

   int val = d_in[i*BLOCKSIZE +
     threadIdx.x];

   d_out[threadIdx.x] +=
         compare(val, 6);

 }
}
```

```
__global__ void compute(int *d_in, int *d_out, int *d_sum) {

 d_out[threadIdx.x] = 0;

 for (i=0; i<SIZE/BLOCKSIZE; i++) {

   int val = d_in[i*BLOCKSIZE +
     threadIdx.x];

   d_out[threadIdx.x] +=
         compare(val, 6);

 }
  __syncthreads();
   if (threadIdx.x == 0) {
    for 0..BLOCKSIZE-1

     *d_sum += d_out[i];
   }
}
```

Tree-Structured Computation

Tree-structured results-gathering phase, where independent threads collect their results in parallel.

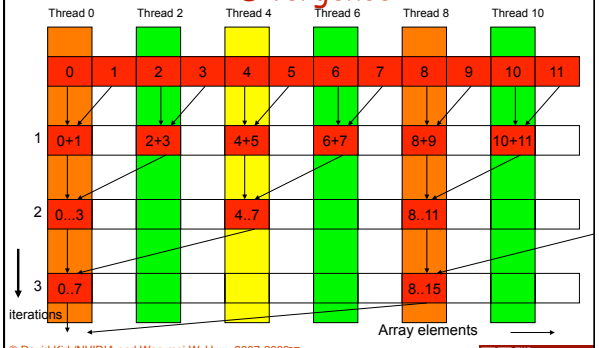
Assume SIZE=16 and BLOCKSIZE(elements computed per thread)=4.



A possible implementation for just the reduction

```
unsigned int t = threadIdx.x;
for (unsigned int stride = 1;
     stride < blockDim.x;  stride *= 2)
{
  __syncthreads();
  if (t % (2*stride) == 0)
     d_out[t] += d_out[t+stride];
}
```


Vector Reduction with Branch Divergence



Some Observations

- In each iteration, two control flow paths will be sequentially traversed for each warp
  - Threads that perform addition and threads that do not
  - Threads that do not perform addition may cost extra cycles depending on the implementation of divergence
- No more than half of threads will be executing at any time
  - All odd index threads are disabled right from the beginning!
  - On average, less than ¼ of the threads will be activated for all warps over time.
  - After the 5th iteration, entire warps in each block will be disabled, poor resource utilization but no divergence.
    - This can go on for a while, up to 4 more iterations (512/32=16= 2^4), where each iteration only has one thread activated until all warps retire

What's Wrong?

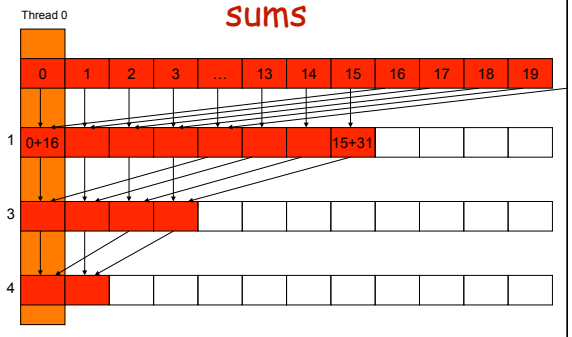
```
unsigned int t = threadIdx.x;
for (unsigned int stride = 1;
     stride < blockDim.x;  stride *= 2)
{
  __syncthreads();
  if (t % (2*stride) == 0)
     d_out[t] += d_out[t+stride];
}
```

BAD: Divergence due to interleaved branch decisions

A better implementation

```
unsigned int t = threadIdx.x;
for (unsigned int stride = blockDim.x >> 1;
     stride >= 1;  stride >> 1)
{
  __syncthreads();
  if (t < stride)
     d_out[t] += d_out[t+stride];
}
```


No Divergence until < 16 sub-sums



A shared memory implementation

- Assume we have already loaded array into

```
__shared__ float partialSum[];

unsigned int t = threadIdx.x;
for (unsigned int stride = blockDim.x >> 1;
     stride >= 1;  stride >> 1)
{
  __syncthreads();
  if (t < stride)
     partialSum[t] += partialSum[t+stride];
}
```

Some Observations About the New Implementation

- Only the last 5 iterations will have divergence
- Entire warps will be shut down as iterations progress
  - For a 512-thread block, 4 iterations to shut down all but one warp in each block
  - Better resource utilization, will likely retire warps and thus blocks faster
- Recall, no bank conflicts either

Predicated Execution Concept

```
<p1> LDR r1,r2,0
```

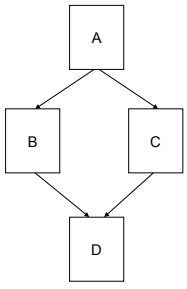
- If p1 is TRUE, instruction executes normally
- If p1 is FALSE, instruction treated as NOP

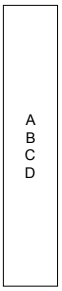
Predication Example

```
:
:
if (x == 10)
   c = c + 1;
:
:
```

```
      :
      :
      LDR r5, X
      p1 <- r5 eq 10
<p1> LDR  r1 <- C
<p1> ADD r1, r1, 1
<p1> STR  r1 -> C
      :
      :
```


Predication can be very helpful for if-else





If-else example

```
    :                                   :
    :                                   :
     p1,p2 <- r5 eq 10                   p1,p2 <- r5 eq 10
<p1> inst 1 from B                  <p1> inst 1 from B
<p1> inst 2 from B                  <p2> inst 1 from C
<p1>  :                                 :
    :                   schedule    <p1> inst 2 from B
<p2> inst 1 from C                  <p2> inst 2 from C
<p2> inst 2 from C                      :
      :                             <p1>  :
    :                                   :
```

The cost is extra instructions will be issued each time the code is executed. However, there is no branch divergence.

Instruction Predication in G80

- Comparison instructions set condition codes (CC)
- Instructions can be predicated to write results only when CC meets criterion (CC != 0, CC >= 0, etc.)
- Compiler tries to predict if a branch condition is likely to produce many divergent warps
  - If guaranteed not to diverge: only predicates if < 4 instructions
  - If not guaranteed: only predicates if < 7 instructions
- May replace branches with instruction predication
- ALL predicated instructions take execution cycles
  - Those with false conditions don't write their output
    - Or invoke memory loads and stores
  - Saves branch instructions, so can be cheaper than serializing divergent paths (for small # instructions)

Warp Vote Functions (Compute Capability > 1.2)

- Can test whether condition on all threads in a warp evaluates to same value

  int __all(int predicate):

  evaluates predicate for all threads of a warp and returns non-zero iff predicate evaluates to non-zero for ***all*** of them.

  int __any(int predicate):

  evaluates predicate for all threads of a warp and returns non-zero iff predicate evaluates to non-zero for ***any*** of them.

Using Warp Vote Functions

- Can tailor code for when none/all take a branch.
- Eliminate overhead of branching and predication.
- Particularly useful for codes where most threads will be the same
  - Example 1: looking for something unusual in image data
  - Example 2: dealing with boundary conditions

Summary of Lecture

- More concurrent execution and its safety
  - Host page-locked memory
  - Concurrent streams
- Debugging your code
- Impact of control flow on performance
  - Due to SIMD execution model for threads
- Execution model/code generated
  - Stall based on CC value (for long instr sequences)
  - Predicated code (for short instr sequences)
- Strategies for avoiding control flow
  - Eliminate divide by zero test (MPM)
  - Warp vote function
- Group together similar control flow paths into warps
  - Example: "tree" reduction

Next Time

- Finish Control Flow
  - Divergent branches
- More project organization