L7: Writing Correct Programs

Outline

- How to tell if your parallelization is correct?
- Definitions:
  - Race conditions and data dependences
  - Example
  - Reasoning about race conditions
  - A Look at the Architecture:
    - how to protect memory accesses from race conditions?
    - Synchronization within a block: __syncthreads();
    - Synchronization across blocks (through global memory)
      - atomicOperations (example)
      - memoryFences
  - Debugging

What can we do to determine if parallelization is correct in CUDA?

- _deviceemu code (to be emulated on host, executed serially)
  - Versions prior to CUDA 3.x
- Can compare GPU output to CPU output, or compare GPU output to device emulation output
  - Race condition may still be present
- Debugging environments (new!)
  - Cuda gdb (Linux)
  - Parallel Nsight (Windows and Vista)
We'll come back to both of these at the end.

- Or can (try to) prevent introduction of race conditions (bulk of lecture)

Administrative

- Next assignment available
  - Goals of assignment:
    - simple memory hierarchy management
    - block-thread decomposition tradeoff
  - Due Tuesday, Feb. 8, 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
    - cs6963s11-teach@list.eng.utah.edu
  - Please use for all questions suitable for the whole class
  - Feel free to answer your classmates questions!
- Questions/discussion
  - Mailing lists
    - cs6963s11-discussion@list.eng.utah.edu
    - cs6963s11-teach@list.eng.utah.edu
  - Please use for questions to Sriram and me

CS6963
Reminder: Count 6s from L1

- Global, device functions and excerpts from host, main

```c
__host__
void
outer_compute(int *h_in_array, int *h_out_array)
{
...
compute<<<1,BLOCKSIZE,msize>>>(d_in_array, d_out_array);
cudaMemcpy(h_out_array, d_out_array, BLOCKSIZE*sizeof(int), cudaMemcpyDeviceToHost);
}
main(int argc, char **argv)
{
...for (int i=0; i<NUMBER; i++)
{sum+=out_array[i];}
printf("Result = \d",sum);
}
```

What if we computed sum on GPU?

- Global, device functions and excerpts from host, main

```c
__host__
void
outer_compute(int *h_in_array, int *h_sum)
{
...compute<<<1,BLOCKSIZE,msize>>>(d_in_array, d_sum);
cudaMemcpy(h_sum, d_sum, sizeof(int), cudaMemcpyDeviceToHost);
}
main(int argc, char **argv)
{
...int *sum;
...outer_compute(in_array, sum);
printf("Result = \d",sum);
}
```

Each thread increments "sum" variable

Threads Access the Same Memory!

- Global memory and shared memory within an SM can be freely accessed by multiple threads
- Requires appropriate sequencing of memory accesses across threads to same location if at least one access is a write

More Formally: Race Condition or Data Dependence

- **A race condition** exists when the result of an execution depends on the timing of two or more events.
- **A data dependence** is an ordering on a pair of memory operations that must be preserved to maintain correctness.
Data Dependence

• Definition:
  Two memory accesses are involved in a data dependence if they may refer to the same memory location and one of the references is a write.

  A data dependence can either be between two distinct program statements or two different dynamic executions of the same program statement.

• Two important uses of data dependence information (among others):
  Parallelization: no data dependence between two computations ⇒ parallel execution safe
  Locality optimization: absence of data dependences & presence of reuse ⇒ reorder memory accesses for better data locality (next week)

Data Dependence of Scalar Variables

True (flow) dependence
\[ a \rightarrow a \]

Anti-dependence
\[ a \rightarrow a \]

Output dependence
\[ a \rightarrow a \]

Input dependence (for locality)
\[ a \rightarrow a \]

Definition: Data dependence exists from a reference instance \( i \) to \( i' \) iff
  - \( i \) or \( i' \) is a write operation
  - \( i \) and \( i' \) refer to the same variable
  - \( i \) executes before \( i' \)

Some Definitions (from Allen & Kennedy)

• Definition 2.5:
  - Two computations are equivalent if, on the same inputs,
    • they produce identical outputs
    • the outputs are executed in the same order

• Definition 2.6:
  - A reordering transformation
    • changes the order of statement execution
    • without adding or deleting any statement executions.

• Definition 2.7:
  - A reordering transformation preserves a dependence if
    • it preserves the relative execution order of the dependences’ source and sink.

Fundamental Theorem of Dependence

• Theorem 2.2:
  - Any reordering transformation that preserves every dependence in a program preserves the meaning of that program.
Parallelization as a Reordering Transformation in CUDA

```c
__host callkernel() {
    dim3 blocks(bx, by);
    dim3 threads(tx, ty, tz);
    
    kernelcode<<<blocks, threads>>>(args);
}

__global kernelcode(args) {
    /* code refers to threadIdx.x, threadIdx.y, threadIdx.z, blockIdx.x, blockIdx.y */
}
```

CUDA Equivalent to “Forall”

```c
__host callkernel() {
    forall (int bIdx_x=0; bIdx_x<bx; bIdx_x++) {
        forall (int bIdx_y=0; bIdx_y<by; bIdx_y++) {
            forall (int tIdx_x=0; tIdx_x<tx; tIdx_x++) {
                forall (int tIdx_y=0; tIdx_y<ty; tIdx_y++) {
                    forall (int tIdx_z=0; tIdx_z<tz; tIdx_z++) {
                        /* code refers to threadIdx.x, threadIdx.y, threadIdx.z, blockIdx.x, blockIdx.y */
                    }
                }
            }
        }
    }
```

Consider Parallelizable Loops

Foreach (or CUDA kernels or Doall) loops:
Loops whose iterations can execute in parallel (a particular reordering transformation)

Example

```c
forall (i=1; i<=n; i++)
    A[i] = B[i] + C[i];
```

Meaning?

Each iteration can execute independently of others
Free to schedule iterations in any order

Why are parallelizable loops important for data-parallel programming models?

Using Data Dependences to Reason about Race Conditions

- Compiler research on data dependence analysis provides a systematic way to conservatively identify race conditions on scalar and array variables
  - "Forall" if no dependences cross the iteration boundary of a parallel loop. (no loop-carried dependences)
  - If a race condition is found,
    - EITHER serialize loop(s) carrying dependence by making it internal to thread program, or part of the host code
    - OR add "synchronization"
Back to our Example: What if Threads Need to Access Same Memory Location

- Dependence on sum across iterations/threads
  - But reordering OK since operations on sum are associative
- Load/increment/store must be done **atomically** to preserve sequential meaning
- Add Synchronization
  - Protect memory locations
  - Control-based (what are threads doing?)
- Definitions:
  - **Atomicity**: a set of operations is atomic if either they all execute or none execute. Thus, there is no way to see the results of a partial execution.
  - **Mutual exclusion**: at most one thread can execute the code at any time
  - **Barrier**: forces threads to stop and wait until all threads have arrived at some point in code, and typically at the same point

Gathering Results on GPU: Barrier Synchronization w/in Block

```c
void __syncthreads();
```

- **Functionality**: Synchronizes all threads in a block
  - Each thread waits at the point of this call until all other threads have reached it
  - Once all threads have reached this point, execution resumes normally
- **Why is this needed?**
  - A thread can freely read the shared memory of its thread block or the global memory of either its block or grid.
  - Allows the program to guarantee partial ordering of these accesses to prevent incorrect orderings.
- **Watch out!**
  - Potential for deadlock when it appears in conditionals

Gathering Results on GPU for “Count 6”

```c
__global__
void compute(int *d_in, int *d_out)
{
  d_out[threadIdx.x] = 0;
  for (i = 0; i < SIZE/BLOCKSIZE; i++)
    d_out[threadIdx.x] += compare(val, 6);
}
```

Gathering Results on GPU: Atomic Update to Sum Variable

```c
int atomicAdd(int* address, int val);
```

- Increments the integer at address by val.
  - Atomic means that once initiated, the operation executes to completion without interruption by other threads
Gathering Results on GPU for "Count 6"

```c
__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);
    }
    atomicAdd(d_sum, d_out[threadIdx.x]);
}
```

Available Atomic Functions

All but CAS take two operands (unsigned int *address, int (or other type) val):

- Arithmetic:
  - atomicAdd() – add val to address
  - atomicSub() – subtract val from address
  - atomicExch() – exchange val at address, return old value
  - atomicMin()
  - atomicMax()
  - atomicInc()
  - atomicDec()
  - atomicCAS()

- Bitwise Functions:
  - atomicAnd()
  - atomicOr()
  - atomicXor()

See Appendix B.11 of NVIDIA CUDA 3.2 Programming Guide

Atomic Operation News

- Only available for devices with compute capability 1.1 or higher
- Operating on shared memory and for either 32-bit or 64-bit global data for compute capability 1.2 or higher
- 64-bit in shared memory for compute capability 2.0 or higher
- atomicAdd for floating point (32-bit) available for compute capability 2.0 or higher (otherwise, just signed and unsigned integer).

Synchronization Within/Across Blocks: Memory Fence Instructions

- `void __threadfence_block();`:
  - waits until all global and shared memory accesses made by the calling thread prior to call are visible to all threads in the thread block. In general, when a thread issues a series of writes to memory in a particular order, other threads may see the effects of these memory writes in a different order.

- `void __threadfence();`:
  - Similar to above, but visible to all threads in the device for global memory accesses and all threads in the thread block for shared memory accesses.

- `void __threadfence_system();`:
  - Similar to above, but also visible to host for "page-locked" host memory accesses.

See Appendix B.5 of NVIDIA CUDA 3.2 Programming Manual
Memory Fence Example

```
__device__
unsigned
int
count
=
0;
__shared__
bool
isLastBlockDone;
__global__
void
sum(const
float*
array,
unsigned
int
N,
float*
result)
{
//
Each
block
sums
a
subset
of
the
input
array
float
paralSum
=
calculateParalSum(array,
N);
if
(threadIdx.x
==
0)
{
//
Thread
0
of
each
block
stores
the
paral
sum
//
to
global
memory
result[blockIdx.x]
=
paralSum;
//
Thread
0
makes
sure
its
result
is
visible
to
//
all
other
threads
__threadfence();
//
Thread
0
of
each
block
signals
that
it
is
done
unsigned
int
value
=
atomicInc(&count,
gridDim.x);
//
Thread
0
of
each
block
determines
if
its
block
is
//
the
last
block
to
be
done
isLastBlockDone
=
(value
==
(gridDim.x
‐
1));
}
//
Synchronize
to
make
sure
that
each
thread
//
reads
the
correct
value
of
isLastBlockDone
__syncthreads();
if
(
isLastBlockDone
)
{
//
The
last
block
sums
the
paral
sums
//
stored
in
result[0
..
gridDim.x‐1]
float
totalSum
=
calculateTotalSum(result
);
if
(
threadIdx.x
==
0)
{
//
Thread
0
of
last
block
stores
total
sum
//
to
global
memory
and
resets
count
so
that
//
next
kernel
call
works
properly
result[0]
=
totalSum;
count
=
0;
}
}
```

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

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 cudaMemcpyHost

stream1 and stream2 identify streams associated with asynchronous call (note 4th "parameter" to kernel invocation)

Code from asyncAPI SDK project

```c
// 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
// synchronously 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(cudaFreeHost(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 asynchronousMemcpy

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
```
CUDA_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

Summary of Lecture

- Data dependence can be used to determine the safety of reordering transformations such as parallelization
  - preserving dependences = preserving "meaning"
- In the presence of dependences, synchronization is needed to guarantee safe access to memory
- Synchronization mechanisms on GPUs:
  - __syncthreads() barrier within a block
  - Atomic functions on locations in memory across blocks
  - Memory fences within and across blocks, and host page-locked memory
- More concurrent execution
  - Host page-locked memory
  - Concurrent streams
- Debugging your code

Next Time

- Control Flow
  - Divergent branches
- More project organization