L3: Writing Correct Programs

Administrative

- First assignment out, due Friday at 5PM (extended)
  - Use handin on CADE machines to submit
    - "handin cs6963 lab1 <probfile>"
    - The file <probfile> should be a gzipped tar file of the CUDA program and output
  - Any questions?
- Mailing lists now visible:
  - cs6963s10-discussion@list.eng.utah.edu
    - Please use for all questions suitable for the whole class
    - Feel free to answer your classmates questions!
  - cs6963s10-teach@list.eng.utah.edu
    - Please use for questions to Protonu and me

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
- A Few Words about Debugging

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

- -deviceemu code (to be emulated on host)
  - Support for pthread debugging?
- Can compare GPU output to CPU output, or compare GPU output to device emulation output
  - Race condition may still be present

We'll come back to both of these at the end.

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

Reductions (from last time)

- "Count 6s" example
- This type of computation is called a ***parallel reduction***
  - Operation is applied to large data structure
  - Computed result represents the aggregate solution across the large data structure
  - Large data structure → computed result (perhaps single number) **[dimensionality reduced]**
- Why might parallel reductions be well-suited to GPUs?
- What if we tried to compute the final sum on the GPUs? (next class and assignment)

Reminder: Gathering and Reporting Results

- Global, device functions and excerpts from host, main

```
__device__ int compare(int a, int b) {
  if (a == b) return 1;
  return 0;
}

__global__ v
  *d_out) {

 d_out[thre

 for (i=0; i<S

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

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

Compute individual results for each thread

Serialize final results gathering on host

```
int __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<BLOCKSIZE; i++)
 { sum+=out_array[i]; }
 printf ("Result = %d\n",sum);
}
```


What if we computed sum on GPU?

- Global, device functions and excerpts from host, main

```
__device__ int compare(int a, int b) {
  if (a == b) return 1;
  return 0;
}

__global__ void compute(int *d_in, int
  *sum) {

 *sum = 0;

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

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

   *sum +=
        compare(val, 6);
 }
}
```

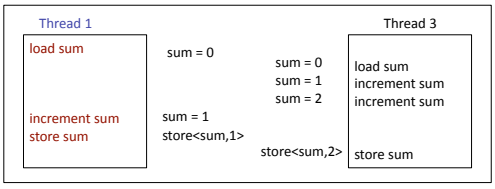
Each thread increments "sum" variable

```
int __host__ void outer_compute
  (int *h_in_array, int *h_sum) {
 ...
 compute<<<1,BLOCKSIZE,msize)>>>
   (d_in_array, d_sum);
 cudaThreadSynchronize();
 cudaMemcpy(h_sum, d_sum,
   sizeof(int),
   cudaMemcpyDeviceToHost);
 ...
int sum;  // an integer
outer_compute(in_array, sum);
printf ("Result = %d\n",sum);
}
```


"Data Race" on sum

threadIdx.x = 0 examines in_array elements 0, 4, 8, 12
threadIdx.x = 1 examines in_array elements 1, 5, 9, 13
threadIdx.x = 2 examines in_array elements 2, 6, 10, 14
threadIdx.x = 3 examines in_array elements 3, 7, 11, 15

} Known as a cyclic data distribution



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 =
= a

Anti-dependence
= a
a =

Output dependence
a =
a =

Input dependence (for locality)
= a
= a

Definition: Data dependence exists from a reference instance i to i' iff
either 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.

Reference: "Optimizing Compilers for Modern Architectures: A Dependence-Based Approach", Allen and Kennedy, 2002, Ch. 2.

Fundamental Theorem of Dependence

- **Theorem 2.2:**
  - Any reordering transformation that preserves every dependence in a program preserves the meaning of that program.
- Now we will discuss abstractions and algorithms to determine whether reordering transformations preserve dependences…

Parallelization as a Reordering Transformation in CUDA

```
__host callkernel() {
  dim3 blocks(bx,by);
  dim3 threads(tx,ty,tz);
  ...
   kernelcode<<<blocks,threads>>>(<a
   rgs>);
}
__global kernelcode(<args>) {
  /* code refers to threadIdx.x,
   threadIdx.y, threadIdx.z, blockIdx.x,
   blockIdx.y */
}
```

```
__host callkernel() {

for (int bIdx_x=0; bIdx_x<bx; bIdx_x++) {
for (int bIdx_y=0; bIdx_y<by; bIdx_y++) {
for (int tIdx_x=0; tIdx_x<tx; tIdx_x++) {
for (int tIdx_y=0; tIdx_y<ty; tIdx_y++) {
for (int tIdx_z=0; tIdx_z<tz; tIdx_z++) {

/* code refers to tIdx_x, tIdx_y, tIdx_z,
   bIdx_x, bIdx_y */
}}}}}
```

EQUIVALENT?

Consider Parallelizable Loops

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

Example

```
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 an important concept for data-parallel programming models?

CUDA Equivalent to "Forall"

```
__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 tIdx_x, tIdx_y, tIdx_z,
   bIdx_x, bIdx_y */
}}}}}
```


Using Data Dependences to Reason about Race Conditions

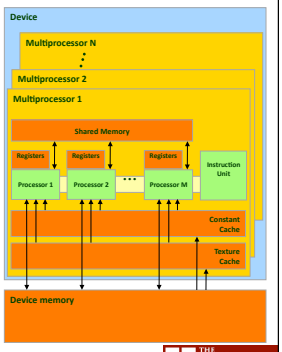
- Compiler research on data dependence analysis provides a systematic way to conservatively identify race conditions on scalar and array variables
  - "Forall" if no dependences cross the iteration boundary of a parallel loop. (no loop-carried dependences)
  - If a race condition is found, either serialize loop(s) carrying dependence, 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 executes. 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

A Look at the Architecture

- What makes it convenient in hardware to efficiently synchronize within blocks?
- And not between blocks?
- Consider device consists of replicated streaming multiprocessors
- And shared instruction unit in SIMD architecture of streaming multiprocessor



Gathering Results on GPU: Barrier Synchronization w/in Block

```
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"

```
__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];
  }
}
```


Gathering Results on GPU: Atomic Update to Sum Variable

```
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"

```
__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);
 }

 atomicAdd(d_sum,
           d_out_array[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 B10 of NVIDIA CUDA Programming Guide

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 __threadfence_block() 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();

- waits until all global and shared memory accesses made by the calling thread prior to __threadfence() are visible to all threads in the device for global memory accesses and all threads in the thread block for shared memory accesses.

Appendix B.5 of NVIDIA CUDA 2.3 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 partialSum = calculatePartialSum(array, N);
if (threadIdx.x == 0) {
  // Thread 0 of each block stores the partial sum
  // to global memory
  result[blockIdx.x] = partia

  // Thread 0 makes sure
  // 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));
}
```

Make sure write to result complete before continuing

```
// Synchronize to make sure that each thread
// reads the correct value of isLastBlockDone
__syncthreads();

if (isLastBlockDone) {
  // The last block sums the partial sums
  // stored in result[0 .. gridDim.x-1]
  t totalSum = calculateTotalSum(result);

  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;
  }
}

}
```


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

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
- Debugging your code
  - Execute single-threaded in device emulation mode on host
  - Compare results to "gold" version implemented on host
  - Other run-time libraries to detect failures
  - More next week on feedback from the compiler

Next Week

- Managing the memory hierarchy
  - Structure of memory system
  - Restrictions on use of different memories
  - Data locality to reduce memory latency
  - Bandwidth optimizations to reduce memory traffic
- Assignment 2