CUDA/GPU – In Depth: misc.

Outline

1. Atomic operations and synchronization
2. Consistency
3. Asynchronous Data Transfers

Example: Shuffling Data

```
// Reorder values based on keys
// Each thread moves one element
__global__ void
shuffle(int* prev_array, int* new_array, int* indices)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    new_array[i] = prev_array[indices[i]];
}
```

Host Code

```
int main()
{
    // Run grid of N/256 blocks of 256 threads each
    shuffle<<< N/256, 256>>>(d_old, d_new, d_ind);
}
```

Blocks must be independent

- Any possible interleaving of blocks should be valid
  - presumed to run to completion without preemption
  - can run in any order
  - can run concurrently OR sequentially
- Blocks may coordinate but not synchronize
  - shared queue pointer: **OK**
  - shared lock: **BAD** … can easily deadlock
- Independence requirement gives *scalability*

ATOMICS

THE PROBLEM

How do you do global communication?

Finish a grid and start a new one

Global Communication

Finish a kernel and start a new one

All writes from all threads complete before a kernel finishes

```
step1<<<grid1,blk1>>>(...);
// The system ensures that all
// writes from step1 complete.
step2<<<grid2,blk2>>>(...);
```

Would need to decompose kernels into “before” and “after” parts

Or, write to a predefined memory location

Race condition! Updates can be lost

Race Conditions

```
threadId:0                          threadId:1917
    // vector[0] was equal to 0
vector[0] += 5;                     vector[0] += 1;
...                                 ...
a = vector[0];                      a = vector[0];
```

- What is the value of **a** in thread 0?
- What is the value of **a** in thread 1917?

- Thread 0 could have finished execution before 1917 started
- Or the other way around
- Or both are executing at the same time

Answer: not defined by the programming model, can be arbitrary

Atomics

- CUDA provides atomic operations to deal with this problem
- An atomic operation guarantees that only a single thread has access to a ***piece*** of memory while an operation completes
- The name atomic comes from the fact that it is uninterruptable
- No dropped data, but ordering is still arbitrary
- Different types of atomic instructions
  - `atomic{Add, Sub, Exch, Min, Max, Inc, Dec, CAS, And, Or, Xor}`
- More types in Fermi

- *Atomics are slower than normal load/store*
- *You can have the whole machine queuing on a single location in memory*

Example: Histogram

```
// Determine frequency of colors in a picture colors have
// already been converted into ints.  Each thread looks
// at one pixel and increments a counter atomically.
__global__
void histogram(int* color, int* buckets)
{
  int i = threadIdx.x
        + blockDim.x * blockIdx.x;
  int c = colors[i];
  atomicAdd(&buckets[c], 1);
}
```

Example: Workqueue

```
// For algorithms where the amount of work per item
// is highly non-uniform, it often makes sense
// to continuously grab work from a queue
__global__
void workq(int* work_q, int* q_counter,
           int* output, int queue_max)
{
  int i = threadIdx.x
        + blockDim.x * blockIdx.x;
  int q_index =
    atomicInc(q_counter, queue_max);
  int result = do_work(work_q[q_index]);
  output[i] = result;
}
```

Example: Global Min/Max (Naive)

```
// If you require the maximum across all threads
// in a grid, you could do it with a single global
// maximum value, but it will be VERY slow
__global__
void global_max(int* values, int* gl_max)
{
  int i = threadIdx.x
          + blockDim.x * blockIdx.x;
  int val = values[i];
  atomicMax(gl_max,val);
}
```

Example: Global Min/Max (Better)

```
// introduce intermediate maximum results, so that
// most threads do not try to update the global max
__global__
void global_max(int* values, int* max,
                int *regional_maxes,
                int num_regions)
{
  // i and val as before …
  int region = i % num_regions;
  if(atomicMax(&reg_max[region],val) < val)
  {
    atomicMax(max,val);
  }
}
```

Global Min/Max

- Single value causes serial bottleneck
- Create hierarchy of values for more parallelism
- Performance will still be slow, so use judiciously

Previous solution (based on reduction) much better!

ATOMICS – Summary (so far)

- Can’t use normal load/store for inter-thread communication because of race conditions
- Use atomic instructions for sparse and/or unpredictable global communication
  - Use shared memory and scan for other communication patterns
- Decompose data for more parallelism
  - But still must have very limited use of single global sum/max/min/etc.

Communication Through Memory

Question:

```
__global__ void race(void)
{
  __shared__ int my_shared_variable;
  my_shared_variable = threadIdx.x;

  // what is the value of
  // my_shared_variable?
}
```

Communication Through Memory

- This is a race condition
- The result is undefined
- The order in which threads access the variable is undefined without explicit coordination
- Use barriers (`e.g., __syncthreads`) or atomic operations (e.g., `atomicAdd`) to enforce well-defined semantics

Communication Through Memory

```
// Use __syncthreads to ensure data is
// ready for access
__global__ void share_data(int *input)
{
  __shared__ int data[BLOCK_SIZE];

  data[threadIdx.x] = input[threadIdx.x];
  __syncthreads();

  // the state of the entire data array is now
  // well-defined for all threads in this block
}
```

Communication Through Memory

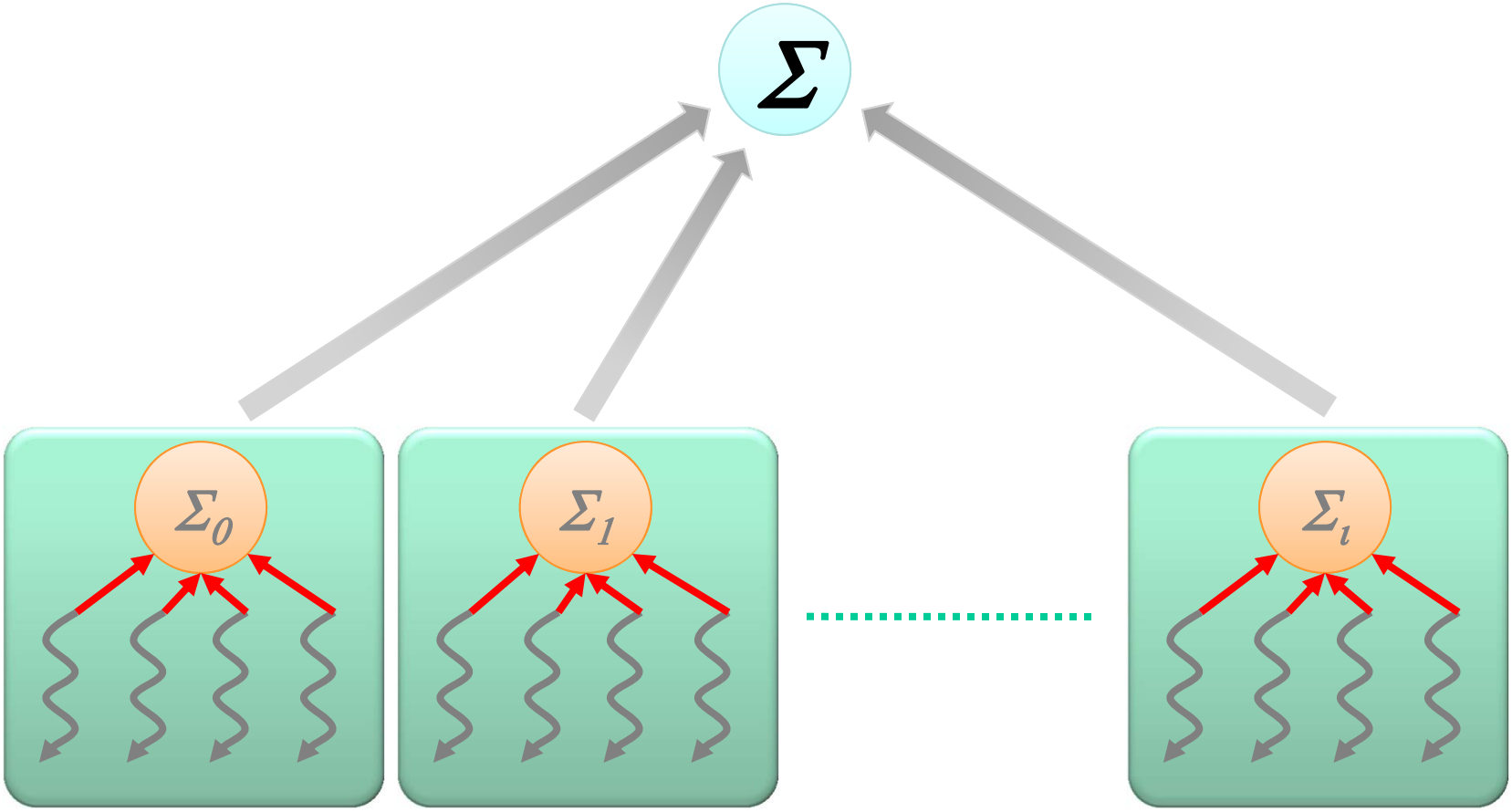
```
// Use atomic operations to ensure exclusive
// access to a variable

// assume *result is initialized to 0
__global__ void sum(int *input, int *result)
{
  atomicAdd(result, input[threadIdx.x]);

  // after this kernel exits, the value of
  // *result will be the sum of the input
}
```

Resource Contention

- Atomic operations aren't cheap!
- They imply (force) serialized access to a variable

```
__global__ void sum(int *input, int *result)
{
  atomicAdd(result, input[threadIdx.x]);
}
...
// how many threads will contend
// for exclusive access to result?
sum<<<B,N/B>>>(input,result);
```

Hierarchical Atomics



- Divide & Conquer
  - Per-thread `atomicAdd` to a `__shared__` partial sum
  - Per-block `atomicAdd` to the total sum

Hierarchical Atomics

```
__global__ void sum(int *input, int *result)
{
  __shared__ int partial_sum;  // shared!

  // thread 0 is responsible for
  // initializing partial_sum
  if(threadIdx.x == 0) partial_sum = 0;
  __syncthreads();

  ...
}
```

Hierarchical Atomics

```
__global__ void sum(int *input, int *result)
{
  ...
  // each thread updates the partial sum
  atomicAdd(&partial_sum,     // shared!
            input[threadIdx.x]);
  __syncthreads();

  // thread 0 updates the total sum
  if(threadIdx.x == 0)
    atomicAdd(result, partial_sum);
}
```

Advice

- Use barriers such as `__syncthreads` to wait until `__shared__` data is ready
- Prefer barriers to atomics when data access patterns are regular or predictable
- Prefer atomics to barriers when data access patterns are sparse or unpredictable
- Atomics with `__shared__` variables are much faster than atomics to `__global__` variables
- Don’t synchronize or serialize unnecessarily

Ex: Count 6s

- Global, device functions and excerpts from host, main

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

__global__ void compu
  d_out[threadIdx.x] =
  for (i=0; i<SIZE/BLOC
     int val = d_in[i*BL
                threadIdx.
     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);
   }
}
```

```
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);
}

main(int argc, char **
…
  int *sum;  // an integer
  outer_compute(in_array, sum);
  printf (”Result = %d\n",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***

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 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

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]);

}
```

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.

Appendix B.5 of NVIDIA CUDA 3.2 Programming Manual

Memory Consistency

Writes to a location become visible to all in the same order
But when does a write become visible

- How to establish orders between a write and a read by different procs?
  - Typically use event synchronization, by using more than one location

P_1	P_2
`/* Assume initial values of A and  flag are 0 */`	
`A = 1;`	`while (flag == 0); /*spin idly*/`
`flag = 1;`	`print A;`

- Intuition not guaranteed by coherence
- Sometimes expect memory to respect order between accesses to *different* locations issued by a given process
  - to preserve orders among accesses to same location by different processes
- Coherence doesn't help: pertains only to single location

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 th
   // to global memory
   result[blockIdx.x] = parti

   // Thread 0 makes sure its
   // 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]
   float 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;
   }
 }
}
```

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

- 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
  - 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) );
```