Advanced Computer Architecture

CUDA Atomics

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

Atomics

- Atomics are slower than normal load/store
- Most of these are associative operations on signed/unsigned integers:
  - quite fast for data in shared memory
  - slower for data in device (DRAM) memory
- You can have the whole machine queuing on a single location in memory

Atomics (cont.)

- CUDA provides atomic operations to deal with this problem
  - Requires hardware with compute capability 1.1 and above
  - Different types of atomic instructions
    - Addition/subtraction: `atomicAdd, atomicSub`
    - Minimum/maximum: `atomicMin, atomicMax`
    - Conditional increment/decrement: `atomicInc, atomicDec`
    - Exchange/compare-and-swap: `atomicExch, atomicCAS`
  - More types in fermi: `atomicAnd, atomicOr, atomicXor`

Example: Global Min/Max (Naïve)

```
// 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)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  int val = values[i];
  int region = i % num_regions;
  if (atomicMax(&reg_max[region],val) < val)
  {
    atomicMax(max,val);
  }
}
```

Global Max/Min

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

Example: Histogram

```
// Determine frequency of colors in a picture
// colors have already been converted into integers
// 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);
}
```

- `atomicAdd` returns the previous value at a certain address
- Useful for grabbing variable amounts of data from the list

Atomics: Summary

- Can't use normal load/store for inter-thread communication because of race conditions
- Use atomic instructions for sparse and/or unpredictable global communication
- Decompose data (very limited use of single global sum/max/min/etc.) for more parallelism