Lesson 5.3

Parallel Computation Patterns

Atomic Operations in CUDA
Objective

• To learn about the atomic operations in CUDA
  • Atomic operation in general
  • Types of atomic operations in CUDA
  • Intrinsic functions
  • A basic histogram kernel
Race Condition Without Atomic Operations

Mem[x] initialized to 0

thread1:  Old ← Mem[x]
New ← Old + 1
Mem[x] ← New

thread2:  Old ← Mem[x]
New ← Old + 1
Mem[x] ← New

• Both threads receive 0
• Mem[x] becomes 1
Atomic Operations in General

- Performed by a single instruction on a memory location *address*
  - Read the old value, calculate a new value, and write the new value to the location
- The hardware ensures that no other threads can access the location until the atomic operation is complete
  - Any other threads that access the location will typically be held in a queue until its turn
  - All threads perform the atomic operation *serially* if they modify the same location
• Function calls that are translated into single instructions (a.k.a. *intrinsic functions* or *intrinsics*)
  • Atomic add, sub, inc, dec, min, max, exch (exchange), CAS (compare and swap)
  • Read CUDA C programming Guide 4.0 or later for details

• Atomic Add

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

reads the 32-bit word `old` pointed to by `address` in global or shared memory, computes `(old + val)`, and stores the result back to memory at the same address. The function returns `old`. 
More Atomic Adds in CUDA

• Unsigned 32-bit integer atomic add
  
  ```c
  unsigned int atomicAdd(unsigned int* address, unsigned int val);
  ```

• Unsigned 64-bit integer atomic add
  
  ```c
  unsigned long long int atomicAdd(unsigned long long int* address, unsigned long long int val);
  ```

• Single-precision floating-point atomic add (capability > 2.0)
  
  ```c
  float atomicAdd(float* address, float val);
  ```
Uncoalesced memory accesses in Sectoned Histogram Algorithm

- Reads from the input array are not coalesced
  - Adjacent threads process non-adjacent input letters
A Better Thread to Data Mapping

- Reads from the input array are coalesced
- Assign inputs to each thread in a strided pattern
- Adjacent threads process adjacent input letters
Iteration 2

- All threads move to the next section of input.
A Basic Histogram Kernel

- The kernel receives a pointer to the input buffer of byte values
- Each thread process the input in a strided pattern

```c
__global__ void histo_kernel(unsigned char *buffer,
                             long size, unsigned int *histo)
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;

    // stride is total number of threads
    int stride = blockDim.x * gridDim.x;

    // All threads handle blockDim.x * gridDim.x
    // consecutive elements
    while (i < size) {
        atomicAdd( &(histo[buffer[i]]), 1);
        i += stride;
    }
}
```
No Recommended Reading