# CME213/ME339 Lecture 9

Erich Elsen

Department of Mechanical Engineering Institute for Computational and Mathematical Engineering Stanford University

Spring 2013



#### **Lecture Overview**

- Reduction implementations
  - Warp
  - Block
  - Multi-Block
- Reductions and floating point
- Atomic Operations
- Matrix-Vector Product
- Matrix-Matrix Product



Elsen CME213/ME339 2 / 13

## **Non-type Template Parameters**

- Template parameters can also be integers, not just types
- Useful for specifying block size, shared memory size
- Size of statically declared shared memory must be known at compile time

```
template<int N>
int add(const int &x) {
   return x + N;
}

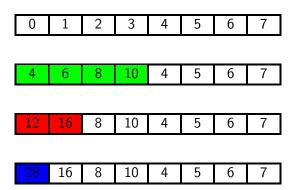
int x;
add<3>(x);
add<5>(x);
```



Elsen CME213/ME339 3 / 13

## Warp Reduce

```
for (int shift = 4; shift > 0; shift >>= 1) {
   if (lane < shift) {
      smem[lane] += smem[lane + shift];
   }
   __syncthreads();
}</pre>
```





Elsen CME213/ME339 4 / 13

#### **Block Reduce**

- Similar idea to Warp Reduce
- Need to handle arbitrary block sizes, not just power of 2
- Bump loop limit up to next power of 2 and add conditional to avoid going past edge
- Alternatively, can make smem array larger and fill with identity value



Elsen CME213/ME339 5 / 13

## Reducing Arbitrary Sizes with One Block

- Before block reduce, loop through the array
- Each thread does a local accumulation of:
   val[threadIdx.x] + val[threadIdx.x + blockSize] + ...
- After local accumulation the block does a tree reduction
- Calculating performance how much memory do we read?
- Every item is read from memory once
- N \* sizeof(int) bytes
- We can ignore the write of one value



Elsen CME213/ME339 6 / 13

## Reducing with Multiple Blocks

• Each block can reduce the values at:

```
[bId * blockDim.x, (bId + 1) * blockDim.x) +
[bId * blockDim.x + gridDim.x, (bId + 1) * blockDim.x + gridDrim.x)
```

- But then what to do about combining the values from each block?
- Multi-pass
  - Write each block's value to memory
  - Launch one more kernel of only one block to reduce these values
- Use Atomic Operations to combine the value from each block



Elsen CME213/ME339 7 / 13

# **Atomic Operations**

- Used to serialize updates to the same memory location and prevent race conditions
- Thread 1 in block 0 and Thread 4 in block 1 both want to add to output [5]
- If they both operate as below, the final value of output[5] will have either the accumulation of thread 1 or thread 4, but not both

```
int gval = output[5];
myVal += output[5];
output[5] = myVal; //race condition!
```



Elsen CME213/ME339 8 / 13

# **Atomic Operations**

1

- The hardware serializes the access from different threads
- Ensures correctness
- But isn't fast and isn't really parallel code
- Used carefully and sparingly, can simplify algorithms and sometimes even boost performance

```
int oldVal = atomicAdd(output + 5, myVal);
```



Elsen CME213/ME339 9 / 13

## **Atomic Operations**

- Atomic Operations can operate on locations in both shared and global memory
- Operations in shared memory are about an order of magnitude faster than global memory
- Operations supported on latest hardware:
  - Add supported for ints, unsigned ints, longs and floats (but not doubles)
  - Sub, Min, Max supported only for integer types
  - Bitwise And, Or, Xor support only for integer types
- Atomic Compare and Swap can be used to implement any atomic operation
- T oldVal = atomicOp(Mem Address, val);



Elsen CME213/ME339 10 / 13

#### **Atomic CAS**

- If oldVal is the same as what is in memory then newVal replaces oldVal
- Whatever was at the memory address is returned
- If you understand this, you understand atomics

```
int memoryVal = memory[10];
int newVal = myVal + memoryVal;
int oldVal = memoryVal
while(memoryVal = atomicCAS(memory + 10, oldVal, newVal)
!= oldVal) {
    newVal = myVal + memoryVal;
    oldVal = memoryVal;
}
```



Elsen CME213/ME339 11 / 13

#### **Back to Reductions**

- We use atomic operations to reduce the values of different blocks
- One thread from each block performs the atomic add
- How does performance change with a different number of blocks?
- Each SM can have maximum 8 resident blocks
- ullet There are 16 SMs on this GPU = 128 blocks to completely fill GPU
- Expect max performance at multiples of 128 blocks



Elsen CME213/ME339 12 / 13

## More about SM Occupancy

- In addition to 8 block limit
- 1536 thread limit
- 8 \* 192 = 1536, called 100% Occupancy
- 6 \* 256 = 1536
- 8 \* 128 = 1024 j 1536 = 66% Occupancy
- Blocks of 128 have lower performance because there are less threads available to hide latency



Elsen CME213/ME339 13 / 13