MORE PROGRAMMING IN CUDA C: PERFORMANCE MEASUREMENT AND ATOMICS

Will Landau, Prof. Jarad Niemi

EVENTS: MEASURING PERFORMANCE ON THE GPU

Event: a time stamp for the GPU.

Use events to measure the amount of time the GPU spends on a task.

```
int main(){
  float   elapsedTime;
  cudaEvent_t start, stop;

  cudaEventCreate(&start);
  cudaEventCreate(&stop);
  cudaEventRecord( start, 0 );

  // SOME GPU KERNEL YOU WANT TIMED HERE

  cudaEventRecord( stop, 0 );
  cudaEventSynchronize( stop );
  cudaEventElapsedTime( &elapsedTime, start, stop );

  cudaEventDestroy( start );
  cudaEventDestroy( stop );

  printf("GPU Time elapsed: %f\n", elapsedTime);
}
```

The variable, `elapsedTime`, is the GPU time spent on the task. You can now print it any way you like.

Note: only GPU elapsed time is measured, not CPU time.

GPU time and CPU time must be measured separately.

MEASURING CPU TIME

```
#include <stdio.h>
#include <time.h>

int main(){

  clock_t start = clock();

  // SOME CPU CODE YOU WANT TIMED HERE

  float elapsedTime = ((double)clock() - start) /
                                CLOCKS_PER_SEC;

  printf("CPU Time elapsed: %f\n", elapsedTime);
}
```

ATOMICS

Consider the following GPU operation on integer `x`:

```
x++;
```

which tells the GPU to do 3 things...

```
x++;
```

1. Read the value stored in x.
2. Add 1 to the value read in step 1.
3. Write the result back to x.

Say we need threads A and B to increment x. We want:

STEP	EXAMPLE
1. Thread A reads the value in `x`.	`A` reads 7 from `x`.
2. Thread A adds 1 to the value it read.	`A` computes 8.
3. Thread A writes the result back to `x`.	`x <- 8`.
4. Thread B reads the value in `x`.	`B` reads 8 from `x`.
5. Thread B adds 1 to the value it read.	`B` computes 9.
6. Thread B writes the result back to `x`.	`x <- 9`.

but we might get:

STEP	EXAMPLE
Thread A reads the value in `x`.	`A` reads 7 from `x`.
Thread B reads the value in `x`.	`B` reads 7 from `x`.
Thread A adds 1 to the value it read.	`A` computes 8.
Thread B adds 1 to the value it read.	`B` computes 8.
Thread A writes the result back to `x`.	`x <- 8`.
Thread B writes the result back to `x`.	`x <- 8`.

Race Condition: A computational hazard that arises when the results of a program depend on the timing of uncontrollable events, such as threads.

Atomic Operation: A command that is executed one thread at a time, thus avoiding a race condition.

To avoid the race condition in our example, we atomically add 1 to x:

```
atomicAdd( &x, 1 );
```

instead of using `x++;`

LIST OF CUDA C BUILT-IN ATOMICS

atomicAdd()
atomicSub()
atomicMin()
atomicMax()
atomicInc()
atomicDec()
atomicExch()
atomicCAS()

```
int atomicAdd(int* address, int val);
unsigned int atomicAdd(unsigned int* address,
                       unsigned int val);
unsigned long long int atomicAdd(unsigned long long int* address,
                                 unsigned long long int val);
float atomicAdd(float* address, float val);
```

reads the 32-bit or 64-bit word **old** located at the address **address** in global or shared memory, computes **(old + val)**, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns **old**.

```
int atomicSub(int* address, int val);
```

```
unsigned int atomicSub(unsigned int* address,
                       unsigned int val);
```

reads the 32-bit word **old** located at the address **address** in global or shared memory, computes **(old - val)**, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns **old**.

```
int atomicExch(int* address, int val);
unsigned int atomicExch(unsigned int* address,
                        unsigned int val);
unsigned long long int atomicExch(unsigned long long int* address,
                                  unsigned long long int val);
float atomicExch(float* address, float val);
```

reads the 32-bit or 64-bit word **old** located at the address **address** in global or shared memory and stores **val** back to memory at the same address. These two operations are performed in one atomic transaction. The function returns **old**.

```
int atomicMin(int* address, int val);
unsigned int atomicMin(unsigned int* address,
                       unsigned int val);
```

reads the 32-bit word **old** located at the address **address** in global or shared memory, computes the minimum of **old** and **val**, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns **old**.

```
int atomicMax(int* address, int val);
unsigned int atomicMax(unsigned int* address,
                       unsigned int val);
```

reads the 32-bit word **old** located at the address **address** in global or shared memory, computes the maximum of **old** and **val**, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns **old**.

```
unsigned int atomicInc(unsigned int* address,
                       unsigned int val);
```

reads the 32-bit word **old** located at the address **address** in global or shared memory, computes **((old >= val) ? 0 : (old+1))**, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns **old**.

```
unsigned int atomicDec(unsigned int* address,
                       unsigned int val);
```

reads the 32-bit word **old** located at the address **address** in global or shared memory, computes **(((old == 0) | (old > val)) ? val : (old-1))**, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns **old**.

```
int atomicCAS(int* address, int compare, int val);
unsigned int atomicCAS(unsigned int* address,
                       unsigned int compare,
                       unsigned int val);
unsigned long long int atomicCAS(unsigned long long int* address,
                                 unsigned long long int compare,
                                 unsigned long long int val);
```

reads the 32-bit or 64-bit word **old** located at the address **address** in global or shared memory, computes **(old == compare ? val : old)**, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns **old** (Compare And Swap).

```
int atomicAnd(int* address, int val);
unsigned int atomicAnd(unsigned int* address,
                       unsigned int val);
```

reads the 32-bit word **old** located at the address **address** in global or shared memory, computes **(old & val)**, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns **old**.

```
int atomicOr(int* address, int val);
unsigned int atomicOr(unsigned int* address,
                      unsigned int val);
```

reads the 32-bit word **old** located at the address **address** in global or shared memory, computes **(old | val)**, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns **old**.

```
int atomicXor(int* address, int val);
unsigned int atomicXor(unsigned int* address,
                       unsigned int val);
```

reads the 32-bit word **old** located at the address **address** in global or shared memory, computes **(old ^ val)**, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns **old**.

LOCKS

Lock: a mechanism in parallel computing that forces a block of code to be executed atomically.

mutex: short for "mutual exclusion", the idea behind locks: while a thread is running code inside a lock, it blocks all other threads from running the code.

lock.h:

```
struct Lock {
    int *mutex;
    Lock( void ) {
        int state = 0;
        HANDLE_ERROR( cudaMalloc( (void**)& mutex,
                                  sizeof(int) ) );
        HANDLE_ERROR( cudaMemcpy( mutex, &state, sizeof(int),
                                  cudaMemcpyHostToDevice ) );
    }

    ~Lock( void ) {
        cudaFree( mutex );
    }

    __device__ void lock( void ) {
        while( atomicCAS( mutex, 0, 1 ) != 0 );
    }

    __device__ void unlock( void ) {
        atomicExch( mutex, 0 );
    }
};
```

NEW DOT PRODUCT CODE

```
__global__ void dot( Lock lock, float *a,
                     float *b, float *c ) {
    __shared__ float cache[threadsPerBlock];
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int cacheIndex = threadIdx.x;

    float   temp = 0;
    while (tid < N) {
        temp += a[tid] * b[tid];
        tid += blockDim.x * gridDim.x;
    }

    // set the cache values
    cache[cacheIndex] = temp;

    // synchronize threads in this block
    __syncthreads();
```

```
    // for reductions, threadsPerBlock must be a power of 2
    // because of the following code
    int i = blockDim.x/2;
    while (i != 0) {
        if (cacheIndex < i)
            cache[cacheIndex] += cache[cacheIndex + i];
        __syncthreads();
        i /= 2;
    }

    if (cacheIndex == 0) {
        lock.lock();
        *c += cache[0];
        lock.unlock();
    }
}
```

REFERENCES

NVIDIA CUDA C Programming Guide. Version 3.2. 2010.