Appendix

Advanced Atomics

Chapter 9 covered some of the ways in which we can use atomic operations to enable hundreds of threads to safely make concurrent modifications to shared data. In this appendix, we'll look at an advanced method for using atomics to implement locking data structures. On its surface, this topic does not seem much more complicated than anything else we've examined. And in reality, this is accurate. You've learned a lot of complex topics through this book, and locking data structures are no more challenging than these. So, why is this material hiding in the appendix? We don't want to reveal any spoilers, so if you're intrigued, read on, and we'll discuss this through the course of the appendix.

A.1 Dot Product Revisited

In Chapter 5, we looked at the implementation of a vector dot product using CUDA C. This algorithm was one of a large family of algorithms known as *reductions*. If you recall, the algorithm computed the dot product of two input vectors by doing the following:

1. Each thread in each block multiplies two corresponding elements of the input vectors and stores the products in shared memory.
2. Although a block has more than one product, a thread adds two of the products and stores the result back to shared memory. Each step results in half as many values as it started with (this is where the term *reduction* comes from)
3. When every block has a final sum, each one writes its value to global memory and exits.
4. If the kernel ran with `N` parallel blocks, the CPU sums these remaining `N` values to generate the final dot product.

This high-level look at the dot product algorithm is intended to be review, so if it's been a while or you've had a couple glasses of Chardonnay, it may be worth the time to review Chapter 5. If you feel comfortable enough with the dot product code to continue, draw your attention to step 4 in the algorithm. Although it doesn't involve copying much data to the host or performing many calculations on the CPU, moving the computation back to the CPU to finish is indeed as awkward as it sounds.

But it's more than an issue of an awkward step to the algorithm or the inelegance of the solution. Consider a scenario where a dot product computation is just one step in a long sequence of operations. If you want to perform *every* operation on the GPU because your CPU is busy with other tasks or computations, you're out of luck. As it stands, you'll be forced to stop computing on the GPU, copy intermediate results back to the host, finish the computation with the CPU, and finally upload that result back to the GPU and resume computing with your next kernel.

Since this is an appendix on atomics and we have gone to such lengths to explain what a pain our original dot product algorithm is, you should see where we're heading. We intend to fix our dot product using atomics so the entire computation can stay on the GPU, leaving your CPU free to perform other tasks. Ideally,

instead of exiting the kernel in step 3 and returning to the CPU in step 4, we want each block to add its final result to a total in global memory. If each value were added atomically, we would not have to worry about potential collisions or indeterminate results. Since we have already used an `atomicAdd()` operation in the histogram operation, this seems like an obvious choice.

Unfortunately, prior to compute capability 2.0, `atomicAdd()` operated only on integers. Although this might be fine if you plan to compute dot products of vectors with integer components, it is significantly more common to use floating-point components. However, the majority of NVIDIA hardware does not support atomic arithmetic on floating-point numbers! But there's a reasonable explanation for this, so don't throw your GPU in the garbage just yet.

Atomic operations on a value in memory guarantee only that each thread's read-modify-write sequence will complete without other threads reading or writing the target value while in process. There is no stipulation about the order in which the threads will perform their operations, so in the case of three threads performing addition, sometimes the hardware will perform `(A+B)+C` and sometimes it will compute `A+(B+C)`. This is acceptable for integers because integer math is associative, so `(A+B)+C = A+(B+C)`. Floating-point arithmetic is *not* associative because of the rounding of intermediate results, so `(A+B)+C` often does not equal `A+(B+C)`. As a result, atomic arithmetic on floating-point values is of dubious utility because it gives rise to nondeterministic results in a highly multithreaded environment such as on the GPU. There are many applications where it is simply unacceptable to get two different results from two runs of an application, so the support of floating-point atomic arithmetic was not a priority for earlier hardware.

However, if we are willing to tolerate some nondeterminism in the results, we can still accomplish the reduction entirely on the GPU. But we'll first need to develop a way to work around the lack of atomic floating-point arithmetic. The solution will still use atomic operations, but not for the arithmetic itself.

A.1.1 ATOMIC LOCKS

The `atomicAdd()` function we used to build GPU histograms performed a read-modify-write operation without interruption from other threads. At a low level, you can imagine the hardware locking the target memory location while this operation is underway, and while locked, no other threads can read or write the value at the location. If we had a way of emulating this lock in our CUDA C kernels, we could perform arbitrary operations on an associated memory location

or data structure. The locking mechanism itself will operate exactly like a typical CPU *mutex*. If you are unfamiliar with mutual exclusion (*mutex*), don't fret. It's not any more complicated than the things you've already learned.

The basic idea is that we allocate a small piece memory to be used as a *mutex*. The mutex will act like something of a traffic signal that governs access to some resource. The resource could be a data structure, a buffer, or simply a memory location we want to modify atomically. When a thread reads a 0 from the mutex, it interprets this value as a "green light" indicating that no other thread is using the memory. Therefore, the thread is free to lock the memory and make whatever changes it desires, free of interference from other threads. To lock the memory location in question, the thread writes a 1 to the mutex. This 1 will act as a "red light" for potentially competing threads. The competing threads must then wait until the owner has written a 0 to the mutex before they can attempt to modify the locked memory.

A simple code sequence to accomplish this locking process might look like this:

```
void lock( void ) {
    if( *mutex == 0 ) {
        *mutex = 1; //store a 1 to lock
    }
}
```

Unfortunately, there's a problem with this code. Fortunately, it's a familiar problem: What happens if another thread writes a 1 to the mutex after our thread has read the value to be zero? That is, both threads check the value at `mutex` and see that it's zero. They then both write a 1 to this location to signify to other threads that the structure is locked and unavailable for modification. After doing so, both threads think they own the associated memory or data structure and begin making unsafe modifications. Catastrophe ensues!

The operation we want to complete is fairly simple: We need to compare the value at `mutex` to 0 and store a 1 at that location if and only if the `mutex` was 0. To accomplish this correctly, this entire operation needs to be performed atomically so we know that no other thread can interfere while our thread examines and updates the value at `mutex`. In CUDA C, this operation can be performed with the function `atomicCAS()`, an atomic compare-and-swap. The function `atomicCAS()` takes a pointer to memory, a value with which to compare the value at that location, and a value to store in that location if the comparison is successful. Using this operation, we can implement a GPU lock function as follows:

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

The call to `atomicCAS()` returns the value that it found at the address `mutex`. As a result, the `while()` loop will continue to run until `atomicCAS()` sees a 0 at `mutex`. When it sees a 0, the comparison is successful, and the thread writes a 1 to `mutex`. Essentially, the thread will spin in the `while()` loop until it has successfully locked the data structure. We'll use this locking mechanism to implement our GPU hash table. But first, we dress the code up in a structure so it will be cleaner to use in the dot product application:

```
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, 1 );
    }
};
```

Notice that we restore the value of `mutex` with `atomicExch( mutex, 1 )`. The function `atomicExch()` reads the value that is located at `mutex`, exchanges

it with the second argument (a 1 in this case), and returns the original value it read. Why would we use an atomic function for this rather than the more obvious method to reset the value at `mutex`?

```
*mutex = 1;
```

If you're expecting some subtle, hidden reason why this method fails, we hate to disappoint you, but this would work as well. So, why not use this more obvious method? Atomic transactions and generic global memory operations follow different paths through the GPU. Using both atomics and standard global memory operations could therefore lead to an `unlock()` seeming out of sync with a subsequent attempt to `lock()` the mutex. The behavior would still be functionally correct, but to ensure consistently intuitive behavior from the application's perspective, it's best to use the same pathway for all accesses to the mutex. Because we're required to use an atomic to lock the resource, we have chosen to also use an atomic to unlock the resource.

A.1.2 DOT PRODUCT REDUX: ATOMIC LOCKS

The only piece of our earlier dot product example that we endeavor to change is the final CPU-based portion of the reduction. In the previous section, we described how we implement a mutex on the GPU. The `Lock` structure that implements this mutex is located in `lock.h` and included at the beginning of our improved dot product example:

```
#include "../common/book.h"
#include "lock.h"

#define imin(a,b) (a<b?a:b)

const int N = 33 * 1024 * 1024;
const int threadsPerBlock = 256;
const int blocksPerGrid =
            imin( 32, (N+threadsPerBlock-1) / threadsPerBlock );
```

With two exceptions, the beginning of our dot product kernel is identical to the kernel we used in Chapter 5. Both exceptions involve the kernel's signature:

```
__global__ void dot( Lock lock, float *a, float *b, float *c )
```

In our updated dot product, we pass a `Lock` to the kernel in addition to input vectors and the output buffer. The `Lock` will govern access to the output buffer during the final accumulation step. The other change is not *noticeable* from the signature but involves the signature. Previously, the `float *c` argument was a buffer for `N` floats where each of the `N` blocks could store its partial result. This buffer was copied back to the CPU to compute the final sum. Now, the argument `c` no longer points to a temporary buffer but to a single floating-point value that will store the dot product of the vectors in `a` and `b`. But even with these changes, the kernel starts out exactly as it did in Chapter 5:

```
__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;
    }
```

At this point in execution, the 256 threads in each block have summed their 256 pairwise products and computed a single value that's sitting in `cache[0]`. Each thread block now needs to add its final value to the value at `c`. To do this safely, we'll use the lock to govern access to this memory location, so each thread needs to acquire the lock before updating the value `*c`. After adding the block's partial sum to the value at `c`, it unlocks the mutex so other threads can accumulate their values. After adding its value to the final result, the block has nothing remaining to compute and can return from the kernel.

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

The `main()` routine is very similar to our original implementation, though it does have a couple differences. First, we no longer need to allocate a buffer for partial results as we did in Chapter 5. We now allocate space for only a single floating-point result:

```
int main( void ) {
    float   *a, *b, c = 0;
    float   *dev_a, *dev_b, *dev_c;

    // allocate memory on the CPU side
    a = (float*)malloc( N*sizeof(float) );
    b = (float*)malloc( N*sizeof(float) );

    // allocate the memory on the GPU
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a,
                              N*sizeof(float) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_b,
                              N*sizeof(float) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c,
                              sizeof(float) ) );
```

As we did in Chapter 5, we initialize our input arrays and copy them to the GPU. But you'll notice an additional copy in this example: We're also copying a zero to `dev_c`, the location that we intend to use to accumulate our final dot product. Since each block wants to read this value, add its partial sum, and store the result back, we need the initial value to be zero in order to get the correct result.

```
    // fill in the host memory with data
    for (int i=0; i<N; i++) {
        a[i] = i;
        b[i] = i*2;
    }

    // copy the arrays 'a' and 'b' to the GPU
    HANDLE_ERROR( cudaMemcpy( dev_a, a, N*sizeof(float),
                              cudaMemcpyHostToDevice ) );
    HANDLE_ERROR( cudaMemcpy( dev_b, b, N*sizeof(float),
                              cudaMemcpyHostToDevice ) );
    HANDLE_ERROR( cudaMemcpy( dev_c, &c, sizeof(float),
                              cudaMemcpyHostToDevice ) );
```

All that remains is declaring our `Lock`, invoking the kernel, and copying the result back to the CPU.

```
    Lock    lock;
    dot<<<blocksPerGrid,threadsPerBlock>>>( lock, dev_a,
                                            dev_b, dev_c );

    // copy c back from the GPU to the CPU
    HANDLE_ERROR( cudaMemcpy( &c, dev_c,
                              sizeof(float),
                              cudaMemcpyDeviceToHost ) );
```

In Chapter 5, this is when we would do a final `for()` loop to add the partial sums. Since this is done on the GPU using atomic locks, we can skip right to the answer-checking and cleanup code:

```
    #define sum_squares(x)  (x*(x+1)*(2*x+1)/6)
    printf( "Does GPU value %.6g = %.6g?\n", c,
             2 * sum_squares( (float)(N - 1) ) );

    // free memory on the GPU side
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );

    // free memory on the CPU side
    free( a );
    free( b );
}
```

Because there is no way to precisely predict the order in which each block will add its partial sum to the final total, it is very likely (almost certain) that the final result will be summed in a different order than the CPU will sum it. Because of the nonassociativity of floating-point addition, it's therefore quite probable that the final result will be slightly different between the GPU and CPU. There is not much that can be done about this without adding a nontrivial chunk of code to ensure that the blocks acquire the lock in a deterministic order that matches the summation order on the CPU. If you feel extraordinarily motivated, give this a try. Otherwise, we'll move on to see how these atomic locks can be used to implement a multithreaded data structure.

A.2 Implementing a Hash Table

The hash table is one of the most important and commonly used data structures in computer science, playing an important role in a wide variety of applications. For readers not already familiar with hash tables, we'll provide a quick primer here. The study of data structures warrants more in-depth study than we intend to provide, but in the interest of making forward progress, we will keep this brief. If you already feel comfortable with the concepts behind hash tables, you should skip to the hash table implementation in Section A.2.2: A CPU Hash Table.

HASH TABLE OVERVIEW

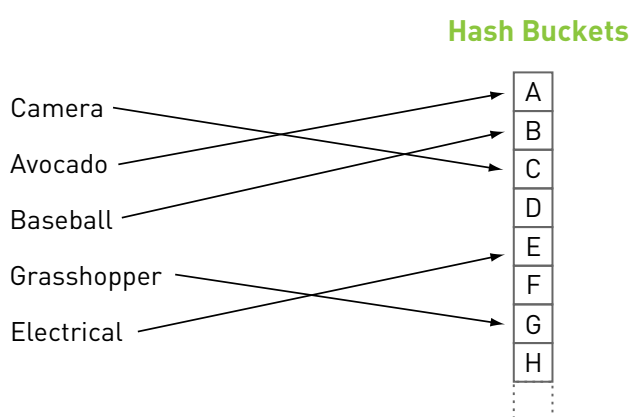

Figure A.1 Hashing of words into buckets

Given what we know about the distribution of words in the English language, this hash function leaves much to be desired because it will not map words uniformly across the 26 buckets. Some of the buckets will contain very few key/value pairs, and some of the buckets will contain a large number of pairs. Accordingly, it will take much longer to find the value associated with a word that begins with a common letter such as S than it would take to find the value associated with a word that begins with the letter X. Since we are looking for hash functions that will give us constant-time retrieval of any value, this consequence is fairly undesirable. An immense amount of research has gone into the study of hash functions, but even a brief survey of these techniques is beyond the scope of this book.

The last component of our hash table data structure involves the buckets. If we had a perfect hash function, every key would map to a different bucket. In this case, we can simply store the key/value pairs in an array where each entry in the array is what we've been calling a *bucket*. However, even with an excellent hash function, in most situations we will have to deal with *collisions*. A collision occurs when more than one key maps to a bucket, such as when we add both the words *avocado* and *aardvark* to our dictionary hash table. The simplest way to store all of the values that map to a given bucket is simply to maintain a list of values in the bucket. When we encounter a collision, such as adding *aardvark* to a dictionary that already contains *avocado*, we put the value associated with *aardvark* at the end of the list we're maintaining in the "A" bucket, as shown in Figure A.2.

After adding the word *avocado* in Figure A.2, the first bucket has a single key/value pair in its list. Later in this imaginary application we add the word *aardvark*, a word that collides with *avocado* because they both start with the letter *A*. You will notice in Figure A.3 that it simply gets placed at the end of the list in the first bucket:

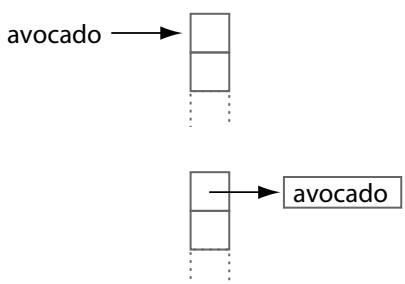

Figure A.2 Inserting the word *avocado* into the hash table

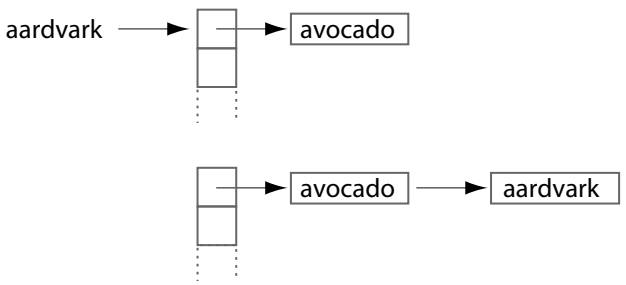

Figure A.3 Resolving the conflict when adding the word *aardvark*

Armed with some background on the notions of a *hash function* and *collision resolution*, we're ready to take a look at implementing our own hash table.

A.2.2 A CPU HASH TABLE

As described in the previous section, our hash table will consist of essentially two parts: a hash function and a data structure of buckets. Our buckets will be implemented exactly as before: We will allocate an array of length `N`, and each entry in the array holds a list of key/value pairs. Before concerning ourselves with a hash function, we will take a look at the data structures involved:

```
#include "../common/book.h"

struct Entry {
    unsigned int    key;
    void*           value;
    Entry           *next;
};

struct Table {
    size_t  count;
    Entry   **entries;
    Entry   *pool;
    Entry   *firstFree;
};
```

As described in the introductory section, the structure `Entry` holds both a key and a value. In our application, we will use unsigned integer keys to store our key/value pairs. The value associated with this key can be any data, so we have declared `value` as a `void*` to indicate this. Our application will primarily be concerned with creating the hash table data structure, so we won't actually store anything in the `value` field. We have included it in the structure for completeness, in case you want to use this code in your own applications. The last piece of data in our hash table `Entry` is a pointer to the next `Entry`. After collisions, we'll have multiple entries in the same bucket, and we have decided to store these entries as a list. So, each entry will point to the next entry in the bucket, thereby forming a list of entries that have hashed to the same location in the table. The last entry will have a `NULL next` pointer.

At its heart, the `Table` structure itself is an array of "buckets." This bucket array is just an array of length `count`, where each bucket in `entries` is just a pointer to an `Entry`. To avoid incurring the complication and performance hit of allocating memory every time we want to add an `Entry` to the table, the table will maintain a large array of available entries in `pool`. The field `firstFree` points to the next available `Entry` for use, so when we need to add an entry to the table, we can simply use the `Entry` pointed to by `firstFree` and increment that pointer. Note that this will also simplify our cleanup code because we can free all of these entries with a single call to `free()`. If we had allocated every entry as we went, we would have to walk through the table and free every entry one by one.

After understanding the data structures involved, let's take a look at some of the other support code:

```
void initialize_table( Table &table, int entries,
                       int elements ) {
    table.count = entries;
    table.entries = (Entry**)calloc( entries, sizeof(Entry*) );
    table.pool = (Entry*)malloc( elements * sizeof( Entry ) );
    table.firstFree = table.pool;
}
```

Table initialization consists primarily of allocating memory and clearing memory for the bucket array `entries`. We also allocate storage for a pool of entries and initialize the `firstFree` pointer to be the first entry in the pool array.

At the end of the application, we'll want to free the memory we've allocated, so our cleanup routine frees the bucket array and the pool of free entries:

```
void free_table( Table &table ) {
    free( table.entries );
    free( table.pool );
}
```

In our introduction, we spoke quite a bit about the hash function. Specifically, we discussed how a good hash function can make the difference between an excellent hash table implementation and poor one. In this example, we're using unsigned integers as our keys, and we need to map these to the indices of our bucket array. The simplest way to do this would be to select the bucket with an index equal to the key. That is, we could store the entry `e` in `table.entries[e.key]`. However, we have no way of guaranteeing that every key will be less than the length of the array of buckets. Fortunately, this problem can be solved relatively painlessly:

```
size_t hash( unsigned int key, size_t count ) {
    return key % count;
}
```

If the hash function is so important, how can we get away with such a simple one? Ideally, we want the keys to map uniformly across all the buckets in our table, and all we're doing here is taking the key modulo the array length. In reality, hash functions may not normally be this simple, but because this is just an example program, we will be randomly generating our keys. If we assume that the random number generator generates values roughly uniformly, this hash function should map these keys uniformly across all of the buckets of the hash table. In your own hash table implementation, you may require a more complicated hash function.

Having seen the hash table structures and the hash function, we're ready to look at the process of adding a key/value pair to the table. The process involves three basic steps:

1. Compute the hash function on the input key to determine the new entry's bucket.
2. Take a preallocated `Entry` from the pool and initialize its `key` and `value` fields.
3. Insert the entry at the front of the proper bucket's list.

We translate these steps to code in a fairly straightforward way.

```
void add_to_table( Table &table, unsigned int key, void* value )
{
    //Step 1
    size_t hashValue = hash( key, table.count );

    //Step 2
    Entry *location = table.firstFree++;
    location->key = key;
    location->value = value;

    //Step 3
    location->next = table.entries[hashValue];
    table.entries[hashValue] = location;
}
```

If you have never seen linked lists (or it's been a while), step 3 may be tricky to understand at first. The existing list has its first node stored at `table.entries[hashValue]`. With this in mind, we can insert a new node at the head of the list in two steps: First, we set our new entry's `next` pointer to point to the first node in the existing list. Then, we store the new entry in the bucket array so *it* becomes the first node of the new list.

Since it's a good idea to have some idea whether the code you've written works, we've implemented a routine to perform a sanity check on a hash table. The check involves first walking through the table and examining every node. We compute the hash function on the node's key and confirm that the node is stored in the correct bucket. After checking every node, we verify that the number of nodes *actually* in the table is indeed equal to the number of elements we *intended* to add to the table. If these numbers don't agree, then either we've added a node accidentally to multiple buckets or we haven't inserted it correctly.

```
#define SIZE    (100*1024*1024)
#define ELEMENTS    (SIZE / sizeof(unsigned int))

void verify_table( const Table &table ) {
    int count = 0;
    for (size_t i=0; i<table.count; i++) {
        Entry   *current = table.entries[i];
        while (current != NULL) {
            ++count;
            if (hash( current->value, table.count ) != i)
                printf( "%d hashed to %ld, but was located "
                        "at %ld\n", current->value,
                        hash( current->value, table.count ), i );
            current = current->next;
        }
    }
    if (count != ELEMENTS)
        printf( "%d elements found in hash table.  Should be %ld\n",
                count, ELEMENTS );
    else
        printf( "All %d elements found in hash table.\n", count);
}
```

With all the infrastructure code out of the way, we can look at `main()`. As with many of this book's examples, a lot of the heavy lifting has been done in helper functions, so we hope that `main()` will be relatively easy to follow:

```
#define HASH_ENTRIES     1024

int main( void ) {
    unsigned int *buffer =
                    (unsigned int*)big_random_block( SIZE );

    clock_t start, stop;
    start = clock();

    Table table;
    initialize_table( table, HASH_ENTRIES, ELEMENTS );

    for (int i=0; i<ELEMENTS; i++) {
        add_to_table( table, buffer[i], (void*)NULL );
    }

    stop = clock();
    float   elapsedTime = (float)(stop - start) /
                          (float)CLOCKS_PER_SEC * 1000.0f;
    printf( "Time to hash:  %3.1f ms\n", elapsedTime );

    verify_table( table );

    free_table( table );
    free( buffer );
    return 0;
}
```

As you can see, we start by allocating a big chunk of random numbers. These randomly generated unsigned integers will be the keys we insert into our hash table. After generating the numbers, we read the system time in order to measure the performance of our implementation. We initialize the hash table and then insert each random key into the table using a `for()` loop. After adding all the keys, we read the system time again to compute the elapsed time to initialize and add the keys. Finally, we verify the hash table with our sanity check routine and free the buffers we've allocated.

You probably noticed that we are using `NULL` as the value for every key/value pair. In a typical application, you would likely store some useful data with the key, but because we are primarily concerned with the hash table implementation itself, we're storing a meaningless value with each key.

A.2.3 MULTITHREADED HASH TABLE

There are some assumptions built into our CPU hash table that will no longer be valid when we move to the GPU. First, we have assumed that only one node can be added to the table at a time in order to make the addition of a node simpler. If more than one thread were trying to add a node to the table at once, we could end up with problems similar to the multithreaded addition problems in the example from Chapter 9.

For example, let's revisit our "avocado and aardvark" example and imagine that threads A and B are trying to add these entries to the table. Thread A computes a hash function on *avocado*, and thread B computes the function on *aardvark*. They both decide their keys belong in the same bucket. To add the new entry to the list, thread A and B start by setting their new entry's `next` pointer to the first node of the existing list as in Figure A.4.

Then, both threads try to replace the entry in the bucket array with their new entry. However, the thread that finishes second is the only thread that has its update preserved because it overwrites the work of the previous thread. So consider the scenario where thread A replaces the entry *altitude* with its entry for *avocado*. Immediately after finishing, thread B replaces what it believe to be the entry for *altitude* with its entry for *aardvark*. Unfortunately, it's replacing *avocado* instead of *altitude*, resulting in the situation illustrated in Figure A.5.

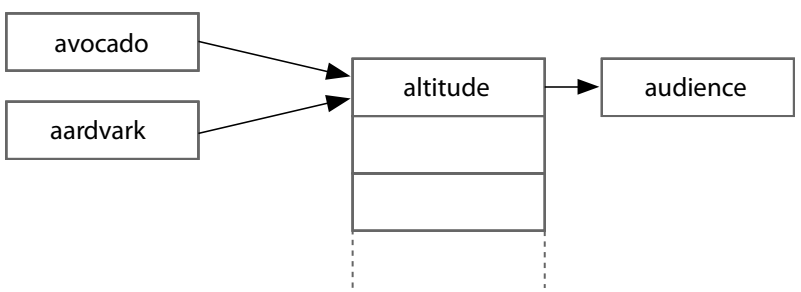

Figure A.4 Multiple threads attempting to add a node to the same bucket

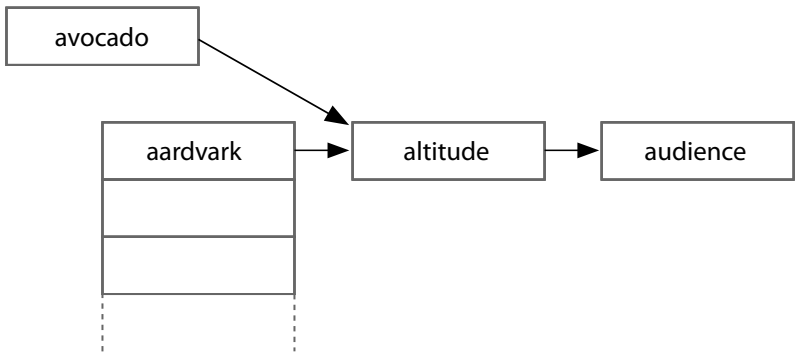

Figure A.5 The hash table after an unsuccessful concurrent modification by two threads

Thread A's entry is tragically "floating" outside of the hash table. Fortunately, our sanity check routine would catch this and alert us to the presence of a problem because it would count fewer nodes than we expected. But we still need to answer this question: How do we build a hash table on the GPU?! The key observation here involves the fact that only one thread can safely make modifications to a bucket at a time. This is similar to our dot product example where only one thread at a time could safely add its value to the final result. If each bucket had an atomic lock associated with it, we could ensure that only a single thread was making changes to a given bucket at a time.

A.2.4 A GPU HASH TABLE

Armed with a method to ensure safe multithreaded access to the hash table, we can proceed with a GPU implementation of the hash table application we wrote in Section A.2.2: A CPU Hash Table. We'll need to include `lock.h`, the implementation of our GPU `Lock` structure from Section A.1.1 Atomic Locks, and we'll need to declare the hash function as a `__device__` function. Aside from these changes, the fundamental data structures and hash function are identical to the CPU implementation.

```
#include "../common/book.h"
#include "lock.h"

struct Entry {
    unsigned int    key;
    void*           value;
    Entry           *next;
};

struct Table {
    size_t  count;
    Entry   **entries;
    Entry   *pool;
};

__device__ __host__ size_t hash( unsigned int value,
                                 size_t count ) {
    return value % count;
}
```

Initializing and freeing the hash table consists of the same steps as we performed on the CPU, but as with previous examples, we use CUDA runtime functions to accomplish this. We use `cudaMalloc()` to allocate a bucket array and a pool of entries, and we use `cudaMemset()` to set the bucket array entries to zero. To free the memory upon application completion, we use `cudaFree()`.

```
void initialize_table( Table &table, int entries,
                       int elements ) {
    table.count = entries;
    HANDLE_ERROR( cudaMalloc( (void**)&table.entries,
                              entries * sizeof(Entry*)) );
    HANDLE_ERROR( cudaMemset( table.entries, 0,
                              entries * sizeof(Entry*) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&table.pool,
                               elements * sizeof(Entry)) );
}
```

```
void free_table( Table &table ) {
    cudaFree( table.pool );
    cudaFree( table.entries );
}
```

We used a routine to check our hash table for correctness in the CPU implementation. We need a similar routine for the GPU version, so we have two options. We could write a GPU-based version of `verify_table()`, or we could use the same code we used in the CPU version and add a function that copies a hash table from the GPU to the CPU. Although either option gets us what we need, the second option seems superior for two reasons: First, it involves reusing our CPU version of `verify_table()`. As with code reuse in general, this saves time and ensures that future changes to the code would need to be made in only one place for both versions of the hash table. Second, implementing a copy function will uncover an interesting problem, the solution to which may be very useful to you in the future.

As promised, `verify_table()` is identical to the CPU implementation and is reprinted here for your convenience:

```
#define SIZE    (100*1024*1024)
#define ELEMENTS    (SIZE / sizeof(unsigned int))
#define HASH_ENTRIES     1024

void verify_table( const Table &dev_table ) {
    Table   table;
    copy_table_to_host( dev_table, table );

    int count = 0;
    for (size_t i=0; i<table.count; i++) {
        Entry   *current = table.entries[i];
        while (current != NULL) {
            ++count;
            if (hash( current->value, table.count ) != i)
                printf( "%d hashed to %ld, but was located "
                        "at %ld\n", current->value,
                        hash(current->value, table.count), i );
            current = current->next;
        }
    }
```

```
    if (count != ELEMENTS)
        printf( "%d elements found in hash table.  Should be %ld\n",
                count, ELEMENTS );
    else
        printf( "All %d elements found in hash table.\n", count );

    free( table.pool );
    free( table.entries );
}
```

Since we chose to reuse our CPU implementation of `verify_table()`, we need a function to copy the table from GPU memory to host memory. There are three steps to this function, two relatively obvious steps and a third, trickier step. The first two steps involve allocating host memory for the hash table data and performing a copy of the GPU data structures into this memory with `cudaMemcpy()`. We have done this many times previously, so this should come as no surprise.

```
void copy_table_to_host( const Table &table, Table &hostTable) {
    hostTable.count = table.count;
    hostTable.entries = (Entry**)calloc( table.count,
                                         sizeof(Entry*) );
    hostTable.pool = (Entry*)malloc( ELEMENTS *
                                     sizeof( Entry ) );

    HANDLE_ERROR( cudaMemcpy( hostTable.entries, table.entries,
                              table.count * sizeof(Entry*),
                              cudaMemcpyDeviceToHost ) );
    HANDLE_ERROR( cudaMemcpy( hostTable.pool, table.pool,
                              ELEMENTS * sizeof( Entry ),
                              cudaMemcpyDeviceToHost ) );
```

The tricky portion of this routine involves the fact that some of the data we have copied are pointers. We cannot simply copy these pointers to the host because they are addresses on the GPU; they will no longer be valid pointers on the host. However, the relative offsets of the pointers *will* still be valid. Every GPU pointer

to an `Entry` points somewhere within the `table.pool[]` array, but for the hash table to be usable on the host, we need them to point to the same `Entry` in the `hostTable.pool[]` array.

Given a GPU pointer X, we therefore need to add the pointer's offset from `table.pool` to `hostTable.pool` to get a valid host pointer. That is, the new pointer should be computed as follows:

```
(X - table.pool) + hostTable.pool
```

We perform this update for every `Entry` pointer we've copied from the GPU: the `Entry` pointers in `hostTable.entries` and the `next` pointer of every `Entry` in the table's pool of entries:

```
    for (int i=0; i<table.count; i++) {
        if (hostTable.entries[i] != NULL)
            hostTable.entries[i] =
                (Entry*)((size_t)hostTable.entries[i] -
                (size_t)table.pool + (size_t)hostTable.pool);
    }
    for (int i=0; i<ELEMENTS; i++) {
        if (hostTable.pool[i].next != NULL)
            hostTable.pool[i].next =
                (Entry*)((size_t)hostTable.pool[i].next -
                (size_t)table.pool + (size_t)hostTable.pool);
    }
}
```

Having seen the data structures, hash function, initialization, cleanup, and verification code, the most important piece remaining is the one that actually involves CUDA C atomics. As arguments, the `add_to_table()` kernel will take an array of keys and values to be added to the hash table. Its next argument is the hash table itself, and the final argument is an array of locks that will be used to lock each of the table's buckets. Since our input is two arrays that our threads will need to index, we also need our all-too-common index linearization:

```
__global__ void add_to_table( unsigned int *keys, void **values,
                              Table table, Lock *lock ) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x * gridDim.x;
```

Our threads walk through the input arrays exactly like they did in the dot product example. For each key in the `keys[]` array, the thread will compute the hash function in order to determine which bucket the key/value pair belongs in. After determining the target bucket, the thread locks the bucket, adds its key/value pair, and unlocks the bucket.

```
    while (tid < ELEMENTS) {
        unsigned int key = keys[tid];
        size_t hashValue = hash( key, table.count );
        for (int i=0; i<32; i++) {
            if ((tid % 32) == i) {
                Entry *location = &(table.pool[tid]);
                location->key = key;
                location->value = values[tid];
                lock[hashValue].lock();
                location->next = table.entries[hashValue];
                table.entries[hashValue] = location;
                lock[hashValue].unlock();
            }
        }
        tid += stride;
    }
}
```

There is something remarkably peculiar about this bit of code, however. The `for()` loop and subsequent `if()` statement seem decidedly unnecessary. In Chapter 6, we introduced the concept of a *warp*. If you've forgotten, a warp is a collection of 32 threads that execute together in lockstep. Although the nuances of how this gets implemented in the GPU are beyond the scope of this book, only one thread in the warp can acquire the lock at a time, and we will suffer many a headache if we let all 32 threads in the warp contend for the lock simultaneously. In this situation, we've found that it's best to do some of the work in software and simply walk through each thread in the warp, giving each a chance to acquire the data structure's lock, do its work, and subsequently release the lock.

The flow of `main()` should appear identical to the CPU implementation. We start by allocating a large chunk of random data for our hash table keys. Then we create start and stop CUDA events and record the start event for our performance

measurements. We proceed to allocate GPU memory for our array of random keys, copy the array up to the device, and initialize our hash table:

```
int main( void ) {
    unsigned int *buffer =
                    (unsigned int*)big_random_block( SIZE );

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

    unsigned int *dev_keys;
    void         **dev_values;
    HANDLE_ERROR( cudaMalloc( (void**)&dev_keys, SIZE ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_values, SIZE ) );
    HANDLE_ERROR( cudaMemcpy( dev_keys, buffer, SIZE,
                              cudaMemcpyHostToDevice ) );

    // copy the values to dev_values here
    // filled in by user of this code example

    Table table;
    initialize_table( table, HASH_ENTRIES, ELEMENTS );
```

The last step of preparation to build our hash table involves preparing locks for the hash table's buckets. We allocate one lock for each bucket in the hash table. Conceivably we could save a lot of memory by using only one lock for the whole table. But doing so would utterly destroy performance because every thread would have to compete for the table lock whenever a group of threads tries to simultaneously add entries to the table. So we declare an array of locks, one for every bucket in the array. We then allocate a GPU array for the locks and copy them up to the device:

```
    Lock    lock[HASH_ENTRIES];
    Lock    *dev_lock;
    HANDLE_ERROR( cudaMalloc( (void**)&dev_lock,
                              HASH_ENTRIES * sizeof( Lock ) ) );
    HANDLE_ERROR( cudaMemcpy( dev_lock, lock,
                              HASH_ENTRIES * sizeof( Lock ),
                              cudaMemcpyHostToDevice ) );
```

The rest of `main()` is similar to the CPU version: We add all of our keys to the hash table, stop the performance timer, verify the correctness of the hash table, and clean up after ourselves:

```
    add_to_table<<<60,256>>>( dev_keys, dev_values,
                              table, dev_lock );

    HANDLE_ERROR( cudaEventRecord( stop, 0 ) );
    HANDLE_ERROR( cudaEventSynchronize( stop ) );
    float   elapsedTime;
    HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
                                        start, stop ) );
    printf( "Time to hash:  %3.1f ms\n", elapsedTime );

    verify_table( table );

    HANDLE_ERROR( cudaEventDestroy( start ) );
    HANDLE_ERROR( cudaEventDestroy( stop ) );
    free_table( table );
    cudaFree( dev_lock );
    cudaFree( dev_keys );
    cudaFree( dev_values );
    free( buffer );
    return 0;
}
```

A.2.5 HASH TABLE PERFORMANCE

Using an Intel Core 2 Duo, the CPU hash table example in Section A.2.2: A CPU Hash Table takes 360ms to build a hash table from 100MB of data. The code was built with the option `-O3` to ensure maximally optimized CPU code. The multithreaded GPU hash table in Section A.2.4: A GPU Hash Table takes 375ms to complete the same task. Differing by less than 5 percent, these are roughly comparable execution times, which raises an excellent question: Why would such a massively parallel machine such as a GPU get beaten by a single-threaded CPU version of the same application? Frankly, this is because GPUs were not designed to excel at multithreaded access to complex data structures such as a hash table. For this reason, there are very few performance motivations to build a data structure such as a hash table on the GPU. So if *all* your application needs to do is build a hash table or similar data structure, you would likely be better off doing this on your CPU.

On the other hand, you will sometimes find yourself in a situation where a long computation pipeline involves one or two stages that the GPU does not enjoy a performance advantage over comparable CPU implementations. In these situations, you have three (somewhat obvious) options:

- Perform every step of the pipeline on the GPU
- Perform every step of the pipeline on the CPU
- Perform some pipeline steps on the GPU and some on the CPU

The last option sounds like the best of both worlds; however, it implies that you will need to synchronize your CPU and GPU at any point in your application where you want to move computation from the GPU to CPU or back. This synchronization and subsequent data transfer between host and GPU can often kill any performance advantage you might have derived from employing a hybrid approach in the first place.

In such a situation, it may be worth your time to perform every phase of computation on the GPU, even if the GPU is not ideally suited for some steps of the algorithm. In this vein, the GPU hash table can potentially prevent a CPU/GPU synchronization point, minimize data transfer between the host and GPU and free the CPU to perform other computations. In such a scenario, it's possible that the overall performance of a GPU implementation would exceed a CPU/GPU hybrid approach, despite the GPU being no faster than the CPU on certain steps (or potentially even getting trounced by the CPU in some cases).

A.3 Appendix Review

We saw how to use atomic compare-and-swap operations to implement a GPU mutex. Using a lock built with this mutex, we saw how to improve our original dot product application to run entirely on the GPU. We carried this idea further by implementing a multithreaded hash table that used an array of locks to prevent unsafe simultaneous modifications by multiple threads. In fact, the mutex we developed could be used for any manner of parallel data structures, and we hope that you'll find it useful in your own experimentation and application development. Of course, the performance of applications that use the GPU to implement mutex-based data structures needs careful study. Our GPU hash table gets beaten by a single-threaded CPU version of the same code, so it will make sense to use the GPU for this type of application only in certain situations. There is no blanket rule that can be used to determine whether a GPU-only, CPU-only, or hybrid approach will work best, but knowing how to use atomics will allow you to make that decision on a case-by-case basis.