DREXEL UNIVERSITY
Electrical and
Computer Engineering
College of Engineering

Drexel University
Electrical and Computer Engineering Dept.
Parallel Computer Architecture ECEC-622

TITLE: CUDA Assignment 2: Vector Dot Product

GROUP MEMBERS: Gregory Matthews and Mark Klobukov

INSTRUCTOR: Dr. Kandasamy

DATE SUBMITTED: 3/8/2017

DATE DUE: 3/19/2017

Assignment description:

Given two n-element vectors **a** and **b**, their dot product $\mathbf{a} \cdot \mathbf{b}$ is given by

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i,$$

From the above mathematical equation for vector dot product multiplication, the assignment calls to parallelize the function using the CUDA API to allow for the GPU to process the computation given pointers to the vectors and result variable on the Host. The program will be provided with n elements ranging from [-0.5, 0.5], which will be the size of the vectors being passed. The program should efficiently use shared memory to access variables within the kernel function to allow for more efficient cache loading, and reducing the delay that would ensue with global memory calls.

Parallelization Approach:

For efficient cache loading on the kernel side function, an array of floats will be initialized that will hold the partial sum of each thread that computes a thread-block's worth of summation for the vector dot product of vectors A and B. This will allow for parallel reduction to iterate through the array of partial sums for each thread block, all the way down to a single value. Once each thread block has its own single partial sum stored in the float array of partial sums, an atomic operation will then be used to sum the last partial sums into the result pointer variable that lives on the CPU. This will eliminate any possibilities of multiple threads trying to access the result variable, which could cause cache coherence problems. The atomic operation can be done using a mutex lock to keep the operation critical.

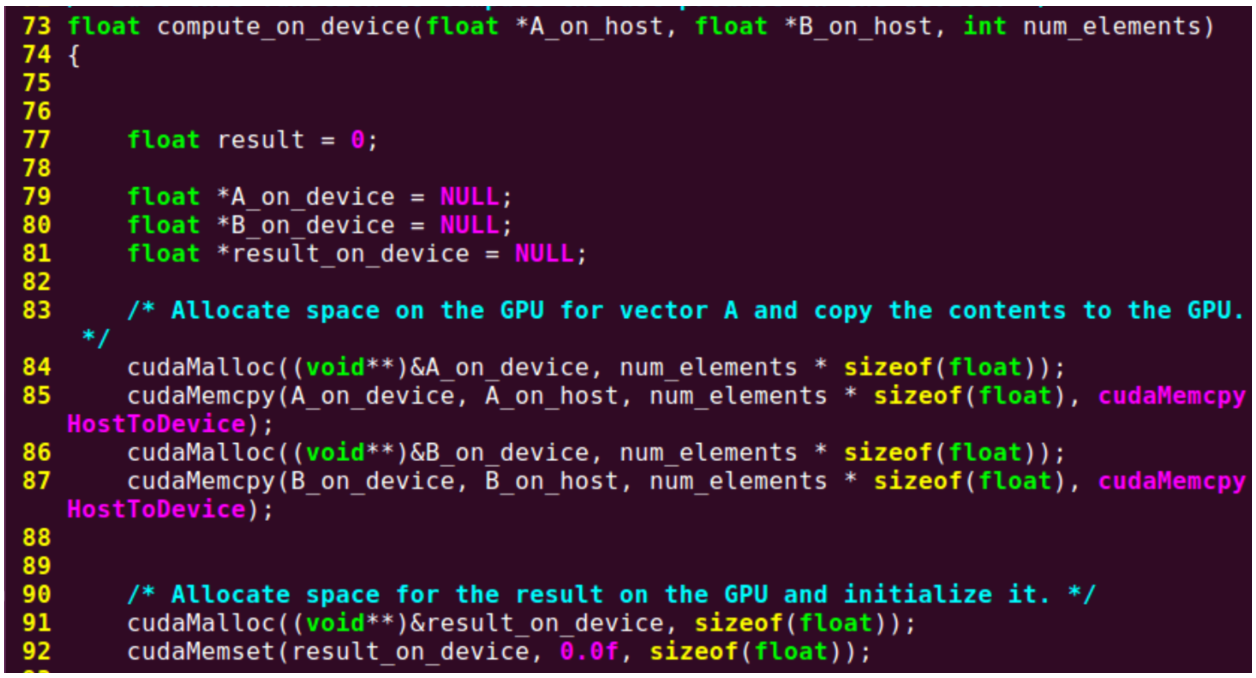
CUDA code explanation:

1) Host function **compute_on_device()**

The screenshot below shows the first part of the host-side function that allocates memory on the GPU and copies the operand vectors into the GPU's global memory.

On line 84, GPU memory is allocated for vector A. Then the vector contents are copied from the host into the GPU memory on the next line. Lines 86-87 repeat the procedure for vector B.

On lines 91-92, space is allocated for the dot product result on the GPU. The **cudaMemset()** function initializes the result variable to zero.



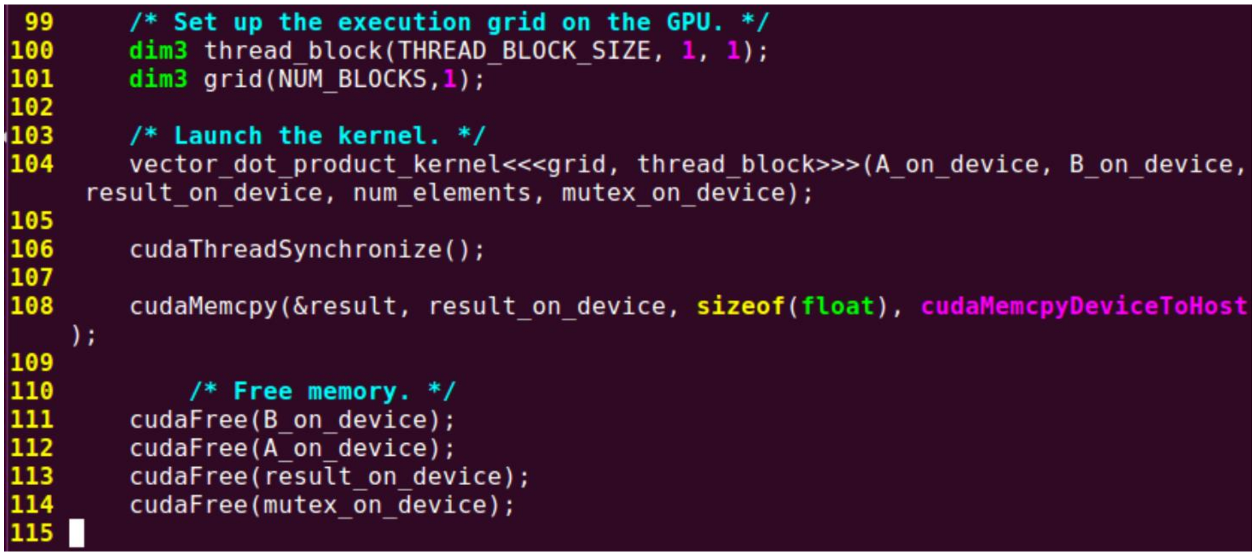
```
73 float compute_on_device(float *A_on_host, float *B_on_host, int num_elements)
74 {
75
76
77     float result = 0;
78
79     float *A_on_device = NULL;
80     float *B_on_device = NULL;
81     float *result_on_device = NULL;
82
83     /* Allocate space on the GPU for vector A and copy the contents to the GPU. */
84     cudaMalloc((void**)&A_on_device, num_elements * sizeof(float));
85     cudaMemcpy(A_on_device, A_on_host, num_elements * sizeof(float), cudaMemcpyHostToDevice);
86     cudaMalloc((void**)&B_on_device, num_elements * sizeof(float));
87     cudaMemcpy(B_on_device, B_on_host, num_elements * sizeof(float), cudaMemcpyHostToDevice);
88
89
90     /* Allocate space for the result on the GPU and initialize it. */
91     cudaMalloc((void**)&result_on_device, sizeof(float));
92     cudaMemset(result_on_device, 0.0f, sizeof(float));
```

The statements on lines 100-101 define grid and block dimensions. Since vector is a one-dimensional data structure, only the x dimension is used when defining the grid. The number of blocks and the number of threads per block are determined by the constants defined in the kernel function.

On line 104, the kernel is launched using the defined grid parameters. The kernel is provided the operand vectors, a pointer to the variable for storing the result, the vector length, the number of elements in each vector, and a mutex, which will be explained in the next section of this report.

After the threads finish executing the kernel, they are synchronized, and the dot product result is copied from the GPU memory into the host memory on line 108.

Finally, lines 111-114 free the GPU memory that was allocated back on the lines 84-92.



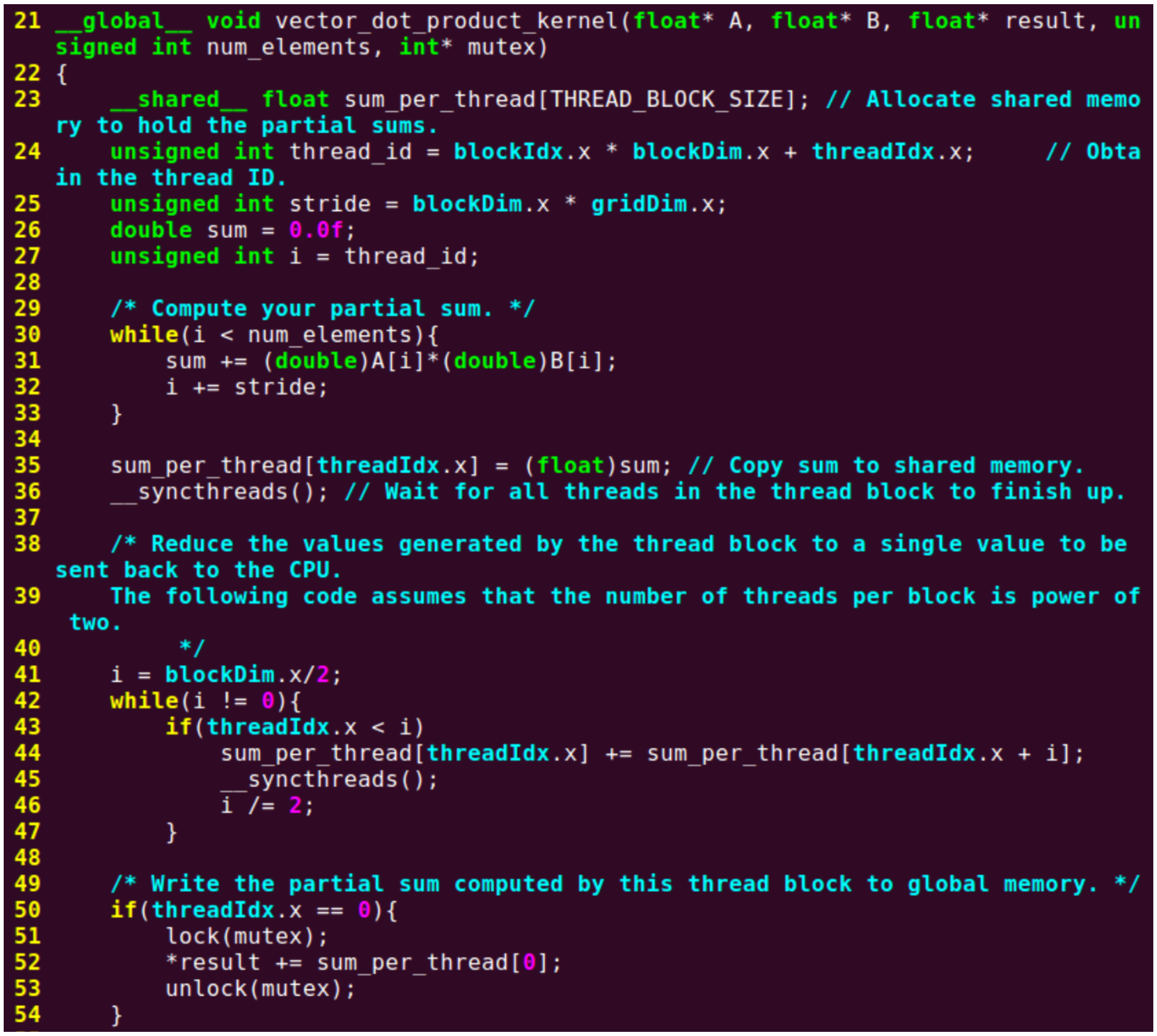
```
 99     /* Set up the execution grid on the GPU. */
100     dim3 thread_block(THREAD_BLOCK_SIZE, 1, 1);
101     dim3 grid(NUM_BLOCKS,1);
102
103     /* Launch the kernel. */
104     vector_dot_product_kernel<<<grid, thread_block>>>(A_on_device, B_on_device,
     result_on_device, num_elements, mutex_on_device);
105
106     cudaThreadSynchronize();
107
108     cudaMemcpy(&result, result_on_device, sizeof(float), cudaMemcpyDeviceToHost
    );
109
110         /* Free memory. */
111     cudaFree(B_on_device);
112     cudaFree(A_on_device);
113     cudaFree(result_on_device);
114     cudaFree(mutex_on_device);
115
```

2) Device function **vector_dot_product_kernel()**

The device side function takes vectors A and B, the result, the number of elements, and the mutex variable as parameters. Before the computation, the shared memory array **sum_per_thread** is created that will store the partial sums of each thread's computation. This will allow for parallel reduction later on in the code. The **thread_Id** is created by taking the block index and multiplying it by the block dimension size, then adding the offset which is the thread Id. The **stride**, which will be the iterator for the thread blocks in choosing the next location for each thread in the thread block, is set to the block dimension * grid dimension. The first while loop that is shown on line 30 below, uses vector dot product multiplication to calculate the total partial sum for each specified thread. The stride will move the position of each thread to the next corresponding thread block. Each thread will block at the function on line 36; **_syncthreads()** to synchronize the threads before moving on.

The next step is to use parallel reduction to take the partial sums inside each **sum_per_thread** variable and sum them into the 0th index. This is done on line 42, where the while loop will check until the ith index has converged to 0 until it is done the reduction step. Each thread will index its position in the **sum_per_thread()** variable and sum up the current index by the ith index, which is simply the grid dimension that gets divided by 2 each iteration. This is only the case for threads with a threadId less than i, because the number of partial sums needed each iteration gets cut in half, so some threads will not be doing work.

The final step is to sum up the final partial sum that is stored in the 0th index of the **sum_per_thread()** function and sum them up into the global variable called **result**. This is done using an atomic operation, by means of a mutex lock, with the lock and unlock functions shown in lines 11-19. This will allow each thread to attempt to accept the mutex lock, and add a partial sum to the result variable, then unlock the mutex. For all other threads, they will block until the mutex is available again. This will guarantee that the operation is critical, and will run serially, because summation into the result variable cannot be done in parallel.



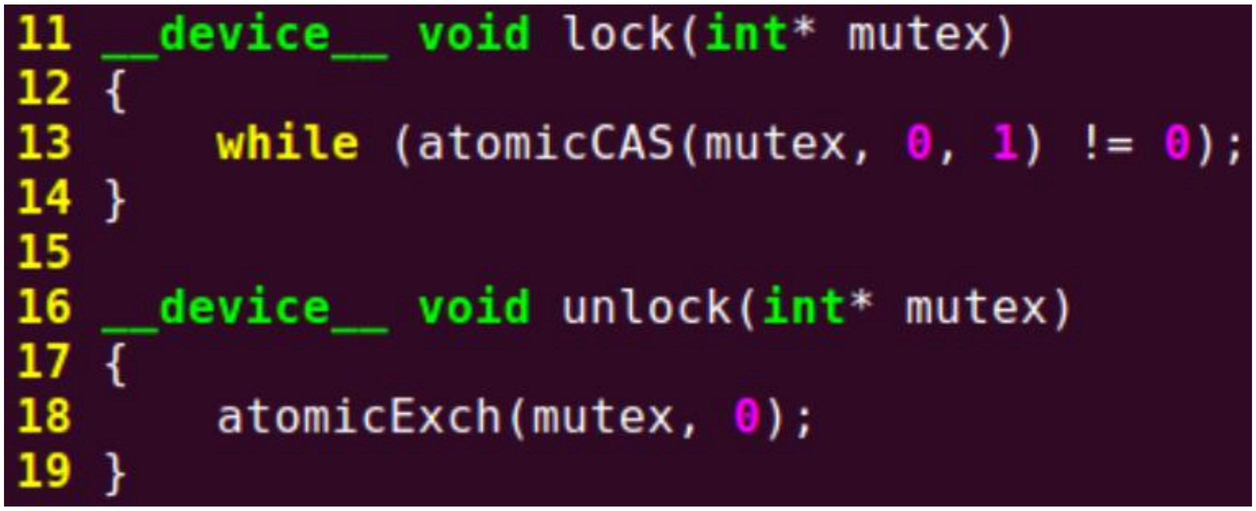
```
__global__ void vector_dot_product_kernel(float* A, float* B, float* result, unsigned int num_elements, int* mutex)
{
    __shared__ float sum_per_thread[THREAD_BLOCK_SIZE]; // Allocate shared memory to hold the partial sums.
    unsigned int thread_id = blockIdx.x * blockDim.x + threadIdx.x;     // Obtain the thread ID.
    unsigned int stride = blockDim.x * gridDim.x;
    double sum = 0.0f;
    unsigned int i = thread_id;

    /* Compute your partial sum. */
    while(i < num_elements){
        sum += (double)A[i]*(double)B[i];
        i += stride;
    }

    sum_per_thread[threadIdx.x] = (float)sum; // Copy sum to shared memory.
    __syncthreads(); // Wait for all threads in the thread block to finish up.

    /* Reduce the values generated by the thread block to a single value to be sent back to the CPU.
    The following code assumes that the number of threads per block is power of two.
         */
    i = blockDim.x/2;
    while(i != 0){
        if(threadIdx.x < i)
            sum_per_thread[threadIdx.x] += sum_per_thread[threadIdx.x + i];
            __syncthreads();
            i /= 2;
        }

    /* Write the partial sum computed by this thread block to global memory. */
    if(threadIdx.x == 0){
        lock(mutex);
        *result += sum_per_thread[0];
        unlock(mutex);
    }
```



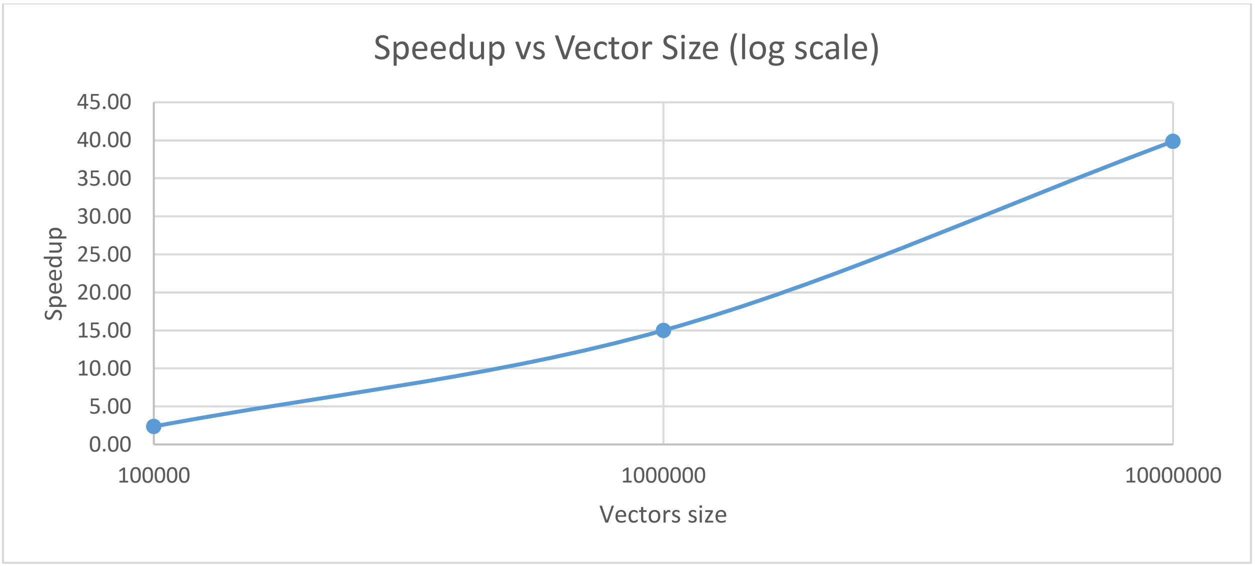
```
__device__ void lock(int* mutex)
{
    while (atomicCAS(mutex, 0, 1) != 0);
}

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

Results:

The program was tested with 12 combinations of vector sizes and thread block sizes. The results are summarized in the table below.

Vector size	Thread Block size	CPU time (sec)	CUDA time (sec)	Speedup
10^5	128	0.00025	0.00011	2.27
	256	0.00026	0.00011	2.36
	512	0.00026	0.00012	2.17
	1024	0.00024	0.00011	2.18
10^6	128	0.00196	0.00014	14.00
	256	0.00195	0.00013	15.00
	512	0.00194	0.00013	14.92
	1024	0.000186	0.00012	1.55
10^7	128	0.01735	0.00044	39.43
	256	0.01714	0.00043	39.86
	512	0.01691	0.00039	43.36
	1024	0.01693	0.00039	43.41



Sensitivity of the GPU run time on thread block size is minimal. For each vector size, four block sizes were tested: 128, 256, 512, and 1024 threads. For the 10,000-element vector, changing the number of threads per block from 256 to 512 changes the runtime by 0.00001 sec. Division of the change in threads by the change in time results in 39.0625 nanoseconds per additional thread. Similar computation for other vector sizes produces the following results.

Vector Size	**Change in run time per extra thread in a block (ns)**
10^5	39.0625
10^6	-78.125
10^7	-156.25

We hypothesize that the reason behind little sensitivity of the GPU run time on thread block size is due to the constraint on the total number of threads that can run concurrently, which is equal to the number of cores on the GPU. In the case of GTX-1080, the number of cores is 2560. As long as the product of the block dimension and the number of blocks is above 2560, increase in the thread block size or the number of blocks will not change the performance because all of the cores are already being utilized.