DREXEL UNIVERSITY
Electrical and Computer Engineering
College of Engineering

Drexel University

Electrical and Computer Engineering Dept.

Parallel Computer Architecture ECEC-622

TITLE: CUDA Assignment 1: Matrix-Vector Multiplication

GROUP MEMBERS: Gregory Matthews and Mark Klobukov

INSTRUCTOR: Dr. Kandasamy

DATE SUBMITTED: 3/5/2017

DATE DUE: 3/5/2017

**Assignment description:**

The first CUDA Assignment involved multiplication of a square NxN matrix by a Nx1 matrix (or, equivalently, an N-element vector). The algorithm for this multiplication is shown in the pseudocode below:

```
1: procedure VEC_MAT_MULT($A$, $x$, $y$)
2: int $i$, $j$;
3: for $i := 0$ to $n - 1$ do
4:     $y[i] := 0$;
5:     for $j := 0$ to $n - 1$ do
6:         $y[i] := y[i] + A[i, j] \times x[j]$;
7:     end for
8: end for
```

**Parallelization approach:**

None of the elements in either multiplicand are being modified, and computation of the ith element of the output vector is independent of (i-1)th or (i+1)th element of the vector. Therefore, one thread can be assigned to compute of each result vector element.

In part one of the assignment, global memory was used to load the matrix A and input vector x, one element at a time inside the loop corresponding to the one on line 6 of the pseudocode above. While this implementation produced a speedup, it was not very efficient because of redundant loads of the same vector multiplicand elements. The better approach is to create a shared memory variable for matrices that are constant among threads to reduce on cache usage.

Second part of the assignment fixed this issue by reducing the number of loads from global memory due to the use of shared memory. Each thread in a given block was instructed to load one element of the multiplicand vector, and the loaded elements were shared by all threads in the block. This approach reduced the number of loads from global memory by the factor of MATRIX_SIZE/BLOCK_WIDTH.

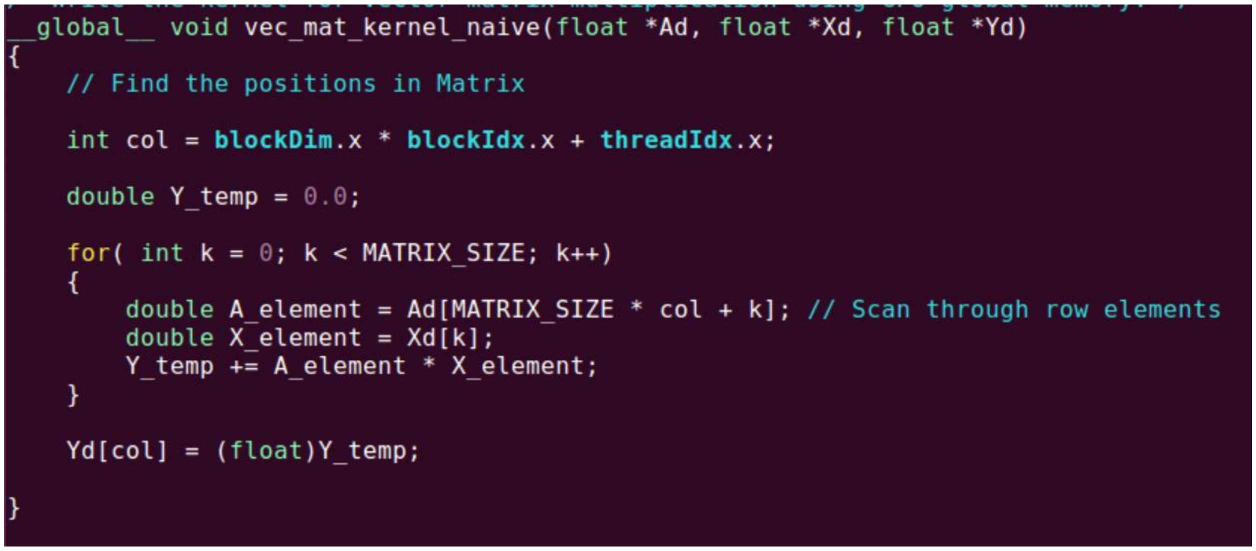
All compilation instructions were left unchanged as compared to the source code from BBLearn. To compile and run, execute the following statements

**make**

**./vec_mat_mult**

**Code explanation: global memory implementation**

For the global memory implementation of vector-matrix multiplication, the method doesn’t take into account redundant loads and simply uses global memory access for each thread’s computation. The code that is run on the device (GPU) is shown below, which includes the function vec_mat_kernel_naive() which takes as arguments, the preallocated variable addresses for matrix A, vector X, and matrix A. The code begins by taking the blockDim.x variable, which references the number of threads per block, multiplies it by blockIdx.x which is the current block index number, and adds the offset of that current block by adding threadIdx.x, which gives the column index for the output vector. this variable is also used to index the row of the matrix A, which should be taken into account that it’s a row and not the column index.

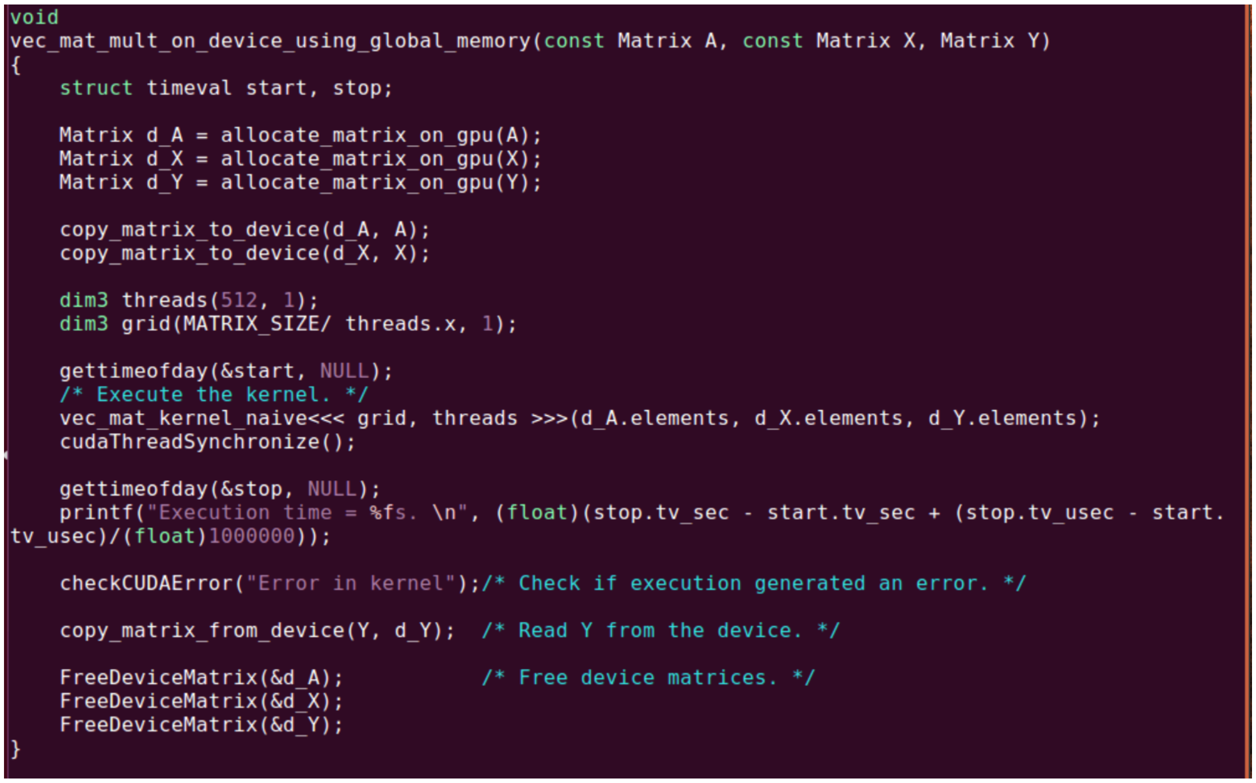


```
__global__ void vec_mat_kernel_naive(float *Ad, float *Xd, float *Yd)
{
    // Find the positions in Matrix

    int col = blockDim.x * blockIdx.x + threadIdx.x;

    double Y_temp = 0.0;

    for( int k = 0; k < MATRIX_SIZE; k++)
    {
        double A_element = Ad[MATRIX_SIZE * col + k]; // Scan through row elements
        double X_element = Xd[k];
        Y_temp += A_element * X_element;
    }

    Yd[col] = (float)Y_temp;

}
```

We initialize a temporary variable for Y called Y_temp, which will allow for all Y_temp to accumulate the partial sums of each thread to have local computations in memory, and them transfer the completed partial sum into the pointer to the global memory matrix for Y.

For computing Y, we incremented over the kth column index for each thread in a thread block, with the current column index dependant on the block index and thread index to locate. The X index is incremented by the kth index, which will incremented over the row indices. The indexed Matrix A and Vector X are then multiplied and then partially summed into local variable Y_temp by each thread, until putting the total sum into Yd at the current column index. The global memory implementation was a simple implementation that doesn’t take into account the redundant loads, and constant cache access by each thread. The global memory function simply splits up the work among thread blocks and makes sure each thread block computes a particular section of Matrix A, and Vector X, and accesses separate locations for Matrix Y when computing the partial sum.

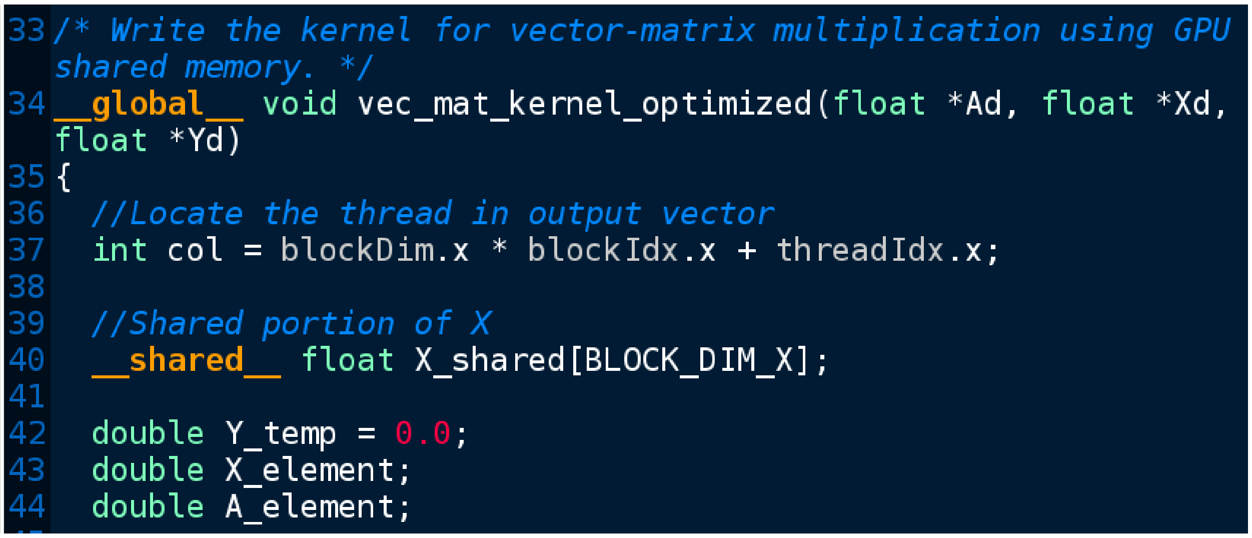
Below is the host side (CPU) code that will begin communication with the kernel to do computation over on the device side (GPU). Given Matrix A, X, and Y as parameters, the host function will allocate memory locations for the given variables to be sent to the GPU. Once allocated, matrix A and X are copied into the corresponding A_device, and X_device variables located on the device side. We initialize our thread block to be one dimensional and house 512 threads in the x direction. the grid is setup to divide the Matrix_Size among the thread blocks so each block has an equal partitioning of the matrix. We time our kernel function and send the allocated memory locations for A, X, and Y into the GPU to be used for the computation, and for the Y address to be equated during the GPU's operation. We check for any errors in the kernel call by calling using the function checkCudaError, and then finally copy back the address location from the GPU to the host side's local variable Y, and free all local matrix pointers.



```
void
vec_mat_mult_on_device_using_global_memory(const Matrix A, const Matrix X, Matrix Y)
{
    struct timeval start, stop;

    Matrix d_A = allocate_matrix_on_gpu(A);
    Matrix d_X = allocate_matrix_on_gpu(X);
    Matrix d_Y = allocate_matrix_on_gpu(Y);

    copy_matrix_to_device(d_A, A);
    copy_matrix_to_device(d_X, X);

    dim3 threads(512, 1);
    dim3 grid(MATRIX_SIZE/ threads.x, 1);

    gettimeofday(&start, NULL);
    /* Execute the kernel. */
    vec_mat_kernel_naive<<< grid, threads >>>(d_A.elements, d_X.elements, d_Y.elements);
    cudaThreadSynchronize();

    gettimeofday(&stop, NULL);
    printf("Execution time = %fs. \n", (float)(stop.tv_sec - start.tv_sec + (stop.tv_usec - start.
tv_usec)/(float)1000000));

    checkCUDAError("Error in kernel");/* Check if execution generated an error. */

    copy_matrix_from_device(Y, d_Y);  /* Read Y from the device. */

    FreeDeviceMatrix(&d_A);           /* Free device matrices. */
    FreeDeviceMatrix(&d_X);
    FreeDeviceMatrix(&d_Y);
}
```

**Code explanation: shared memory implementation**

The screenshot below shows the first half of the kernel function that uses shared memory. Each thread's location in the grid is stored into **col** variable. A shared array called **X_shared** is declared on line 40. Its length corresponds to the number of threads in a given block. This gives the programmer flexibility as to how many elements to store inside each SM and not overflow the shared memory capacity. The three variables on lines 42-44 will be used in the second part of the function for dot-product computations.

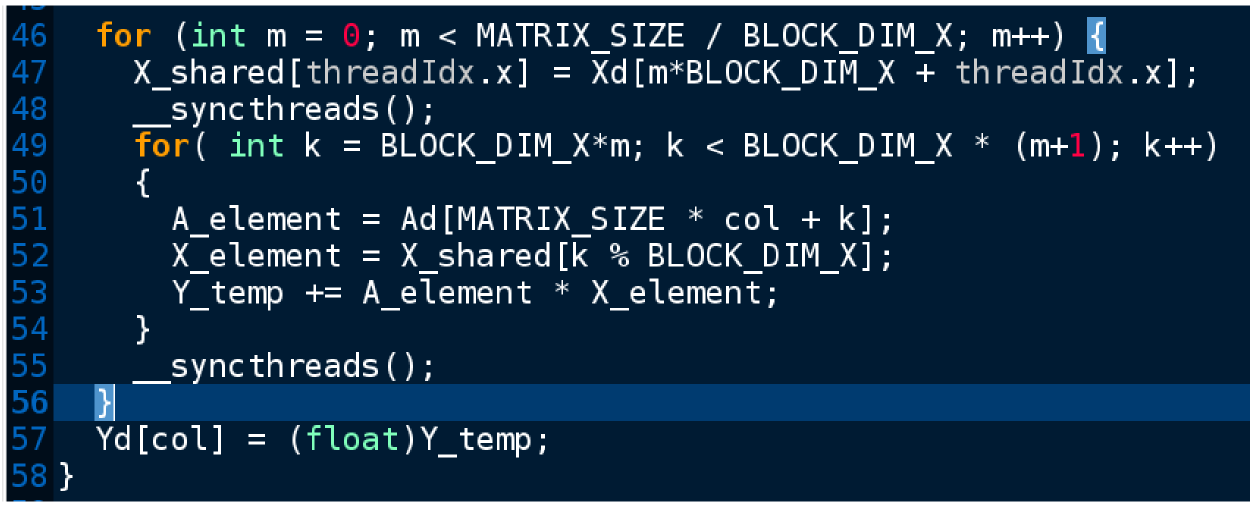


```
33 /* Write the kernel for vector-matrix multiplication using GPU
   shared memory. */
34 __global__ void vec_mat_kernel_optimized(float *Ad, float *Xd,
   float *Yd)
35 {
36   //Locate the thread in output vector
37   int col = blockDim.x * blockIdx.x + threadIdx.x;
38 
39   //Shared portion of X
40   __shared__ float X_shared[BLOCK_DIM_X];
41 
42   double Y_temp = 0.0;
43   double X_element;
44   double A_element;
```

The second part of the function is shown in the next screenshot. Each block's threads load one element from vector X. On line 48, the threads are synchronized because it is necessary to make sure that the X_shared array is fully populated before computations are performed.

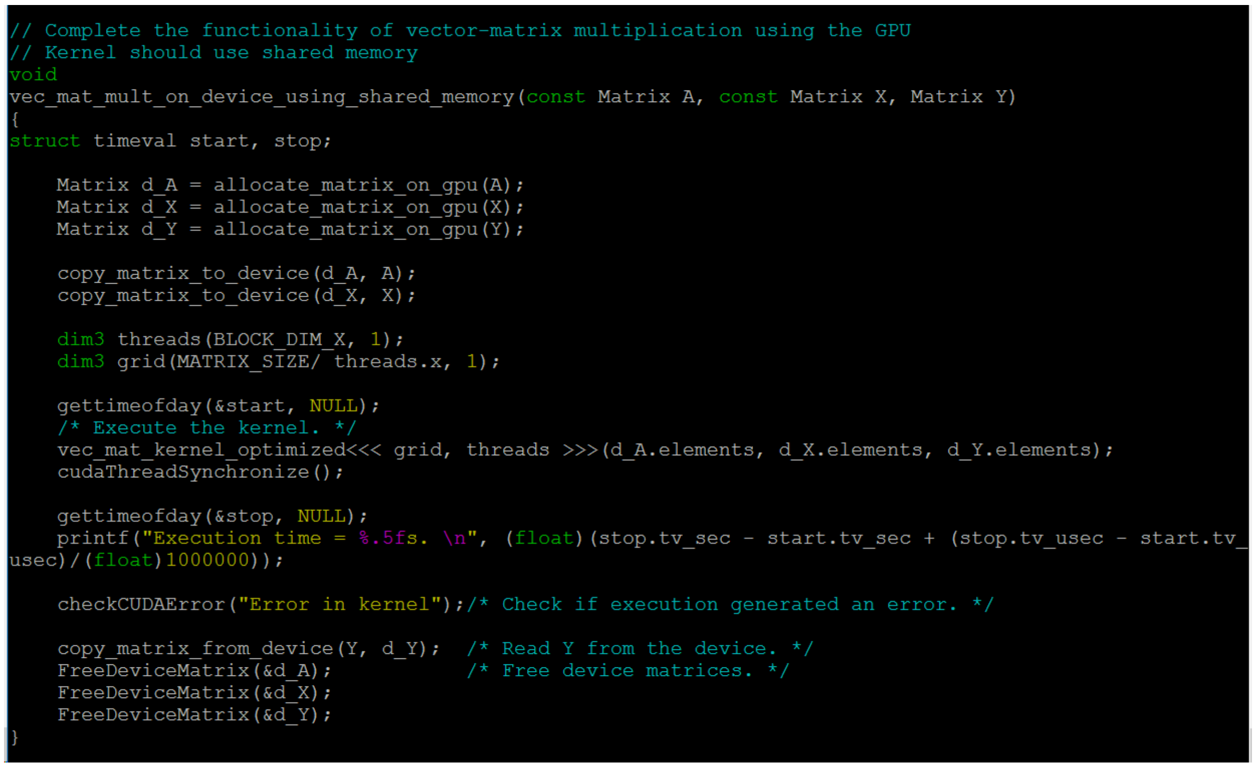
The loop on line 49 iterates through a given block of elements (determined by the outer loop with index m). Each thread is responsible for computation of one element of the output vector, which is why **col** variable is used to identify the row of the matrix. Line 52 accesses the appropriate element in the shared array. The modulo operator is necessary to ensure the shared array is indexed within bounds. The inner for-loop accumulates the partial dot product (computed over a fraction of the row) into the variable **Y_temp.** At the end of the iterations of the outer for loop, all these partial dot-products will be summed for each row and stored into the output vector **Yd**.

It is worth mentioning that none of the elements of the matrix **Ad** are loaded into the shared memory. This is because each element of the matrix A is used exactly once for computation.



```
  for (int m = 0; m < MATRIX_SIZE / BLOCK_DIM_X; m++) {
    X_shared[threadIdx.x] = Xd[m*BLOCK_DIM_X + threadIdx.x];
    __syncthreads();
    for( int k = BLOCK_DIM_X*m; k < BLOCK_DIM_X * (m+1); k++)
    {
      A_element = Ad[MATRIX_SIZE * col + k];
      X_element = X_shared[k % BLOCK_DIM_X];
      Y_temp += A_element * X_element;
    }
    __syncthreads();
  }
  Yd[col] = (float)Y_temp;
}
```

The kernel described above was called in the following function. Space is allocated on the GPU for the matrix and the vector to be multiplied as well as for the result vector. Then the kernel is called, and the grid dimensions are determined by the global variables BLOCK_DIM_X and MATRIX_SIZE. Once the kernel is done, all memory is freed, and the function exits.

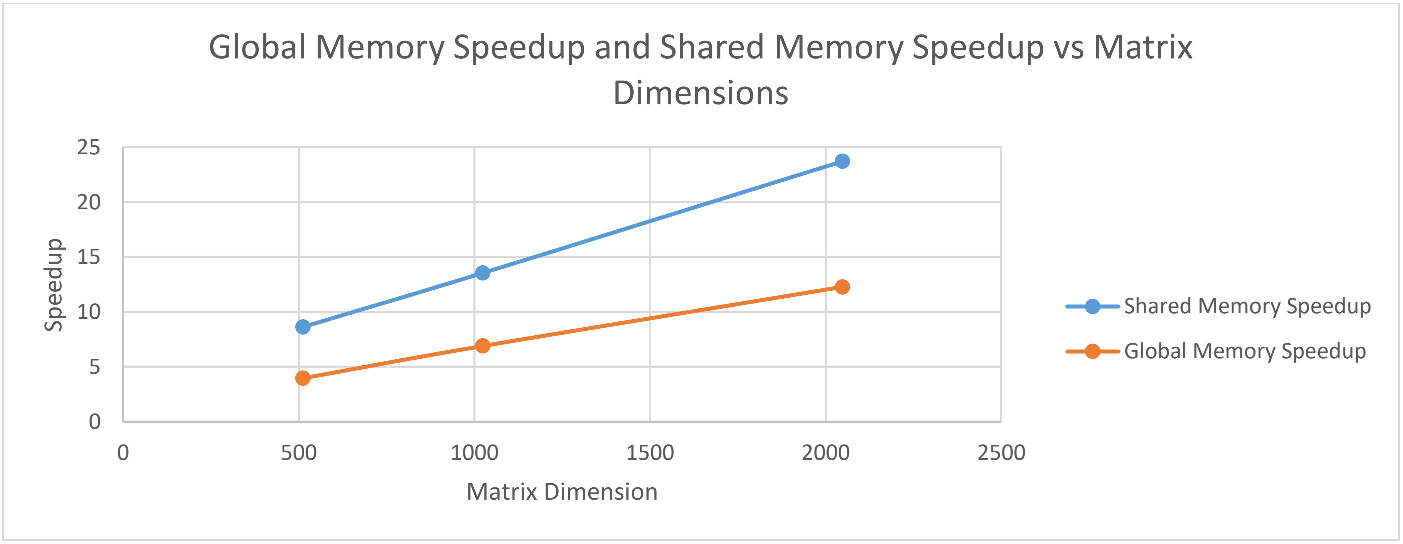


```
// Complete the functionality of vector-matrix multiplication using the GPU
// Kernel should use shared memory
void
vec_mat_mult_on_device_using_shared_memory(const Matrix A, const Matrix X, Matrix Y)
{
struct timeval start, stop;

    Matrix d_A = allocate_matrix_on_gpu(A);
    Matrix d_X = allocate_matrix_on_gpu(X);
    Matrix d_Y = allocate_matrix_on_gpu(Y);

    copy_matrix_to_device(d_A, A);
    copy_matrix_to_device(d_X, X);

    dim3 threads(BLOCK_DIM_X, 1);
    dim3 grid(MATRIX_SIZE/ threads.x, 1);

    gettimeofday(&start, NULL);
    /* Execute the kernel. */
    vec_mat_kernel_optimized<<< grid, threads >>>(d_A.elements, d_X.elements, d_Y.elements);
    cudaThreadSynchronize();

    gettimeofday(&stop, NULL);
    printf("Execution time = %.5fs. \n", (float)(stop.tv_sec - start.tv_sec + (stop.tv_usec - start.tv_
usec)/(float)1000000));

    checkCUDAError("Error in kernel");/* Check if execution generated an error. */

    copy_matrix_from_device(Y, d_Y);  /* Read Y from the device. */
    FreeDeviceMatrix(&d_A);           /* Free device matrices. */
    FreeDeviceMatrix(&d_X);
    FreeDeviceMatrix(&d_Y);
}
```

**Result:**

Matrix-vector multiplication was performed on a 512x512, 1024x1024, and 2048x2048 matrix. The following table summarizes the timing results.

| **Matrix Size** | **Serial Time (s)** | **Global Memory Time (s)** | **Shared Memory Time (s)** | **Global Memory Speedup** | **Shared Memory Speedup** |
|---|---|---|---|---|---|
| 512x512 | 0.000863 | 0.000218 | 0.0001 | 3.9587 | 8.6300 |
| 1024x1024 | 0.00271 | 0.000393 | 0.0002 | 6.8957 | 13.5500 |
| 2048x2048 | 0.009013 | 0.000734 | 0.00038 | 12.2793 | 23.7184 |



As the graph above shows, the shared memory implementation outperformed the global memory implementation for each matrix size. The shared memory approach has less redundant loads and lower memory access times due to shared memory resource for vector X. Moreover, the speedup increased for each implementation with increasing matrix size, which confirms Gustafson's law.

**Question:** the GTX 1080 GPU can achieve a peak processing rate of about 8800 GFLOPs. The memory bandwidth on the device is 320 GB/s. How many floating-point operations must be performed per load operation to achieve the peak processing rate? What is the performance of your kernels (both naïve as well as the one that uses shared memory), in terms of GFLOPs?

Since the memory bandwidth is 320 GB/s and a float is stored in 4 bytes, $80\times10^9$ floats can be loaded on the device per second. Dividing 8800 GFLOPs by the number of floats that can be loaded from memory per second, we get the total number of total number of floating-point operations per load operation, which is 110 FP operations.

Matrix-vector multiplication involves $2n^2$ floating-point operations for an nxn matrix. The performance of our kernels can be calculated using the following equation:

$$GFLOPs = \frac{Total\ Floating\ Point\ Operations}{Execution\ Time \times 10^9}$$

The following table shows the performance of our kernels in terms of GFLOPs for each matrix size using both global memory and shared memory implementations.

| **Matrix Width** | **# operations** | **Time Global (s)** | **Time shared (s)** | **GFLOPS global** | **GFLOPS shared** |
|---|---|---|---|---|---|
| 512 | 524288 | 0.000218 | 0.0001 | 2.4050 | 5.2429 |
| 1024 | 2097152 | 0.000393 | 0.0002 | 5.3363 | 10.4858 |
| 2048 | 8388608 | 0.000734 | 0.00038 | 11.4286 | 22.0753 |