# EN605.617 Module 6: Stretch problem

The file didn’t state a #include statement for stdio.h which contains the printf() method.

The kernels perform multiplication and then adds result to the result destination. Which seems wrong unless the result matrix is zero. Mathematically it can be expressed as:

```
result[i] += multiplier[i]*multiplicand[i]
```

The shared memory portion seem to be more generic since multiplcand values are loaded into shared memory. The constant memory version seems to just declare a local array with pre-defined values from 0 to 15 and multiply the input by that.

I like how there are methods to transfer from global to shared memory. This is useful to see exactly how the transfer is being performed.

The code seems to be a bit disorganized, I had to jump around much.

A version of the code I modified is presented in the following pages. It compiles and runs on my local machine and it presents the same results I saw for the assignment program I made. Shared memory is overall faster. This is with both kernels using pinned memory.

```
#include <stdio.h>

// constant values that do not change at runtime
const unsigned int NUM_ROWS = 640*32;
const unsigned int NUM_COLS = 16;
const unsigned int NUM_SHARED_COEFF = 32;
const unsigned int NUM_REGISTER_COEFF = 16;
unsigned int num_threads = 256;

 // from global_memory.cu file provided in Module 4 Vocareum lab
// create a timer object to test the duration of an operation
__host__ cudaEvent_t get_time(void)
{
      cudaEvent_t time;
      cudaEventCreate(&time);
      cudaEventRecord(time);
      return time;
}

// file previously delcared matrix with default values.  Each element contains the
sum of the row and
// product index
void fill_matrix(unsigned int num_rows, unsigned int num_cols, unsigned int *
matrix_to_fill) {

      for (int col_index = 0; col_index < num_cols; col_index++)
      {
            for (int row_index = 0; row_index < num_rows; row_index++)
            {
                  matrix_to_fill[row_index*(num_cols)+col_index] = row_index +
col_index;
            }
      }
}

// copy data from one type of device memory to another
__device__ void copy_btw_device_memory(unsigned int *mem_to_copy, unsigned int
*mem_to_fill, unsigned int num_elements)
{
      for (int elem_index = 0; elem_index < num_elements; elem_index++)
      {
            mem_to_fill[elem_index] = mem_to_copy[elem_index];
      }
}

// multiply matrices using shared memory (e.g. mx, where m is constant)
__device__ void multiply_values_shared(unsigned int * var_mat, unsigned int *
result_mat, unsigned int *shared_coeff_mat)
{
	unsigned int index = blockIdx.x * blockDim.x + threadIdx.x;
	atomicAdd(&result_mat[index / NUM_COLS], var_mat[index] *
shared_coeff_mat[index % NUM_COLS]);
}

// store coefficients in shared memory based on thread location.  Threads in the
top half otf the
// coeff_mat use one set of coefficients, while threads in the bottom half use an
alternate set
__device__ void calc_shared_mem(unsigned int * coeff_mat, unsigned int *
shared_mat)
{
	int index = blockIdx.x * blockDim.x + threadIdx.x;
	unsigned int padding = (index / ((NUM_COLS*NUM_ROWS) / 2))*NUM_COLS;
	for (int i = 0; i < NUM_COLS; i++) {
		shared_mat[i] = coeff_mat[i + padding];
	}
}

// load data into shared memory and perform linear model multiplication (y=mx)
using shared memory
__global__ void linear_multiply_shared_mem_device(unsigned int * var_mat, unsigned
int * coeff_mat, unsigned int * result_mat)
{
	// declare shared memory, and load values into shared memory
	__shared__ unsigned int shared_coeff_mat[NUM_SHARED_COEFF];
	calc_shared_mem(coeff_mat, shared_coeff_mat);
	multiply_values_shared(var_mat, result_mat, shared_coeff_mat);
}

// perform linear model multipliation (y=mx) using constant memory
__global__ void linear_multiply_register_mem_device(unsigned int * var_mat,
unsigned int * result_mat)
{
	unsigned int index = blockIdx.x * blockDim.x + threadIdx.x;
  unsigned int register_mem[] = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
	atomicAdd(&result_mat[index / NUM_COLS], var_mat[index] * register_mem[index
% NUM_COLS]);
}

// print results of the GPU linear model multipliation
void print_select_results(unsigned int * data_to_print)
{
	printf("head of results \n");
	for (int i = 0; i < 10; i++) {
		printf("%i ", data_to_print[i]);
	}
	printf("\n transition between coefficients \n");
	for (int i = NUM_ROWS / 2 - 5; i < NUM_ROWS / 2 + 5; i++) {
```

```
            printf("%i ", data_to_print[i]);
      }
      printf(" tail of results \n");
      for (int i = NUM_ROWS - 10; i < NUM_ROWS; i++) {
            printf("%i ", data_to_print[i]);
      }
      printf("\n");
}

// perform QA/QC test for copying memory from GPU register memory to GPU global
memory
__global__ void test_register_copy(unsigned int * output_data, unsigned int
num_elements)
{
      unsigned int register_mem[] = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
      copy_btw_device_memory(register_mem, output_data, num_elements);
}

// perform QA/QC test for copying memory from global memory to GPU shared memory
bakc to GPU global memory
__global__ void test_shared_copy(unsigned int * input_data, unsigned int *
output_data, unsigned int num_elements)
{
      __shared__ unsigned int shared_int_data[16];
      copy_btw_device_memory(input_data, shared_int_data, num_elements);
      copy_btw_device_memory(shared_int_data, output_data, num_elements);
}

// test ability to copy to/from register memory
void test_register_mem_cpy() {
      unsigned int *h_results, *h_input, *d_results;
      cudaMalloc((void **)&d_results, sizeof(unsigned int)*16);
      cudaError_t status = cudaMallocHost((void**)&h_results, sizeof(unsigned int)
*16);
      if (status != cudaSuccess)
            printf("Error allocating pinned hot memory\n"); // from Mark Harris,
NVidia blogs
      status = cudaMallocHost((void**)&h_input, sizeof(unsigned int) * 16);
      if (status != cudaSuccess)
            printf("Error allocating pinned hot memory\n"); // from Mark Harris,
NVidia blogs

      fill_matrix(16, 1, h_input);


      test_register_copy<<< 1, 16 >>>(d_results, 16);

      cudaMemcpy(h_results, d_results, sizeof(unsigned int)*16,
cudaMemcpyDeviceToHost);
```

```
	printf("register memory copy test \n");
	for(int i = 0; i < 16; i++) {
		printf("expected: %i, actual: %i \n" , h_input[i], h_results[i]);
	}

	cudaFree(d_results);
	cudaFreeHost(h_input);
	cudaFreeHost(h_results);
}

// test ability to copy to/from shared memory
void test_shared_mem_cpy() {
	unsigned int *h_results, *h_input, *d_results, *d_input;
	cudaMalloc((void **)&d_results, sizeof(unsigned int)*16);
	cudaMalloc((void **)&d_input, sizeof(unsigned int)*16);
	cudaError_t status = cudaMallocHost((void**)&h_results, sizeof(unsigned int)
*16);
	if (status != cudaSuccess)
		printf("Error allocating pinned hot memory\n"); // from Mark Harris,
NVidia blogs
	status = cudaMallocHost((void**)&h_input, sizeof(unsigned int) * 16);
	if (status != cudaSuccess)
		printf("Error allocating pinned hot memory\n"); // from Mark Harris,
NVidia blogs

	fill_matrix(16, 1, h_input);

	cudaMemcpy(d_input, h_input, sizeof(unsigned int)*16,
cudaMemcpyHostToDevice);
	test_shared_copy<<< 1, 16 >>>(d_input,d_results,16);

	cudaMemcpy(h_results, d_results, sizeof(unsigned int)*16,
cudaMemcpyDeviceToHost);

	printf("shared memory copy test \n");
	for(int i = 0; i < 16; i++) {
		printf("expected: %i, actual: %i \n" , h_input[i], h_results[i]);
	}

	cudaFree(d_results);
	cudaFree(d_input);
	cudaFreeHost(h_input);
	cudaFreeHost(h_results);
}


// host and device operations to perform linear multiplication (e.g. mx) on the
device using register memory
```

```
float linear_multiply_register_mem_host(bool debug, int num_thrd)
{

      num_threads = num_thrd;

      // delcare and allocate device memory
      unsigned int *d_var;
      unsigned int *d_results;
      cudaMalloc((void **)&d_var, sizeof(unsigned int)* (NUM_COLS*NUM_ROWS));
      cudaMalloc((void **)&d_results, sizeof(unsigned int)*NUM_ROWS);

      // declare and allocate pinned host memory
  unsigned int *h_var, *h_results, *h_coeff;
  cudaError_t status = cudaMallocHost((void**)&h_var, sizeof(unsigned int)
*(NUM_ROWS*NUM_COLS));
      if (status != cudaSuccess)
            printf("Error allocating pinned hot memory\n"); // from Mark Harris,
NVidia blogs
  status = cudaMallocHost((void**)&h_results, sizeof(unsigned int) *NUM_ROWS);
      if (status != cudaSuccess)
            printf("Error allocating pinned hot memory\n"); // from Mark Harris,
NVidia blogs
  status = cudaMallocHost((void**)&h_coeff, sizeof(unsigned int)
*NUM_REGISTER_COEFF);
      if (status != cudaSuccess)
            printf("Error allocating pinned hot memory\n"); // from Mark Harris,
NVidia blogs

      // fill declared matrices with default values (row + column index)
      fill_matrix(NUM_ROWS, NUM_COLS, h_var);
      fill_matrix(NUM_REGISTER_COEFF, 1, h_coeff);

    // measure how long it takes to copy data to device, run a kernel, and copy the
data back to host
      cudaEvent_t start_time = get_time();

      // copy data to device
      cudaMemcpy(d_var, h_var, sizeof(unsigned int)* (NUM_COLS*NUM_ROWS),
cudaMemcpyHostToDevice);
      cudaMemcpy(d_results, h_results, sizeof(unsigned int)*(NUM_ROWS),
cudaMemcpyHostToDevice);

      // perform linear multiplication on GPU and copy data from GPU to host
      linear_multiply_register_mem_device <<< (NUM_ROWS*NUM_COLS + num_threads - 1)
/ num_threads, num_threads >>> (d_var, d_results);
      cudaMemcpy(h_results, d_results, sizeof(unsigned int)*NUM_ROWS,
cudaMemcpyDeviceToHost);

      // finish timing performance and record result
      cudaEvent_t end_time = get_time();
      cudaEventSynchronize(end_time);
      float delta = 0;
      cudaEventElapsedTime(&delta, start_time, end_time);

      // print select rows of results for debugging, quality assurance purposes
      if (debug) {
            print_select_results(h_results);
      }

      // free device memory
      cudaFree(d_var);
      cudaFree(d_results);

      // free pinned memory
      cudaFreeHost(h_var);
      cudaFreeHost(h_results);
      cudaFreeHost(h_coeff);

      return delta;
}

// host and device operations to perform linear multiplication (e.g. mx) on the
device using shared memory
float linear_multiply_shared_mem_host(bool debug, int num_thrd)
{

      num_threads = num_thrd;

      // delcare and allocate device memory
      unsigned int * d_coeff, *d_var, *d_results;
      cudaMalloc((void **)&d_coeff, sizeof(unsigned int) * NUM_SHARED_COEFF);
      cudaMalloc((void **)&d_var, sizeof(unsigned int)* (NUM_COLS*NUM_ROWS));
      cudaMalloc((void **)&d_results, sizeof(unsigned int)*NUM_ROWS);

      // declare and allocate pinned host memory
      unsigned int *h_var, *h_results, *h_coeff;
    cudaError_t status = cudaMallocHost((void**)&h_var, sizeof(unsigned int)
*(NUM_ROWS*NUM_COLS));
      if (status ≠ cudaSuccess) printf("Error allocating pinned hot memory\n"); //
from Mark Harris, NVidia blogs
    status = cudaMallocHost((void**)&h_results, sizeof(unsigned int) *NUM_ROWS);
      if (status ≠ cudaSuccess) printf("Error allocating pinned hot memory\n"); //
from Mark Harris, NVidia blogs
    status = cudaMallocHost((void**)&h_coeff, sizeof(unsigned int)
*NUM_REGISTER_COEFF);
      if (status ≠ cudaSuccess) printf("Error allocating pinned hot memory\n"); //
from Mark Harris, NVidia blogs
```

```
	// fill declared matrices with default values (row + column index)
	fill_matrix(NUM_ROWS, NUM_COLS, h_var);
	fill_matrix(NUM_SHARED_COEFF, 1, h_coeff);

	// measure how long it takes to copy data to device, run a kernel, and copy
the data back to host
	cudaEvent_t start_time = get_time();

	// copy data to device, including constant memory
	cudaMemcpy(d_coeff, h_coeff, sizeof(unsigned int) * NUM_SHARED_COEFF,
cudaMemcpyHostToDevice);
	cudaMemcpy(d_var, h_var, sizeof(unsigned int)* (NUM_COLS*NUM_ROWS),
cudaMemcpyHostToDevice);
	cudaMemcpy(d_results, h_results, sizeof(unsigned int)*(NUM_ROWS),
cudaMemcpyHostToDevice);

	// perform linear multiplication on GPU and copy data from GPU to host
	linear_multiply_shared_mem_device <<< (NUM_ROWS*NUM_COLS + num_threads - 1) /
num_threads, num_threads >>> (d_var, d_coeff, d_results);
	cudaMemcpy(h_results, d_results, sizeof(unsigned int)*NUM_ROWS,
cudaMemcpyDeviceToHost);

	// finish timing performance and record result
	cudaEvent_t end_time = get_time();
	cudaEventSynchronize(end_time);
	float delta = 0;
	cudaEventElapsedTime(&delta, start_time, end_time);

	// print select rows of results for debugging, quality assurance purposes
	if (debug) { print_select_results(h_results);}

	// free device memory
	cudaFree(d_coeff);
	cudaFree(d_var);
	cudaFree(d_results);

	// free pinned memory
	cudaFreeHost(h_var);
	cudaFreeHost(h_results);
	cudaFreeHost(h_coeff);

	return delta;
}

// process input args to determine if debug statements should be printed
bool set_debug_flag(int argc, char** argv) {
	bool debug = false;
	if (argc>1)
	{
```

```
            if (argv[1][0] == 't' or argv[1][0] == 'T')
            {
                  debug = true;
                  printf("run debug \n");
            }
      }
      return(debug);
}

int main(int argc, char** argv)
{
      float duration;

      // process input and determine if debug statements should be printed
      bool should_debug = set_debug_flag(argc, argv);

      test_register_mem_cpy();
      test_shared_mem_cpy();

      // perform linear multipliation (y =mx) using shared memory
      duration = linear_multiply_shared_mem_host(should_debug, 256);
      printf("time from copy to completion for linear multiply algorithm with
shared memory: %f \n", duration);

      // perform linear multipliation (y=mx) using register memory
      duration = linear_multiply_register_mem_host(should_debug, 256);
      printf("time from copy to completion for linear multiply algorithm with
register memory: %f \n", duration);

      printf("finished program");
      return 1;
}
```

Results:

```
register memory copy test
expected: 0, actual: 0
expected: 1, actual: 1
expected: 2, actual: 2
expected: 3, actual: 3
expected: 4, actual: 4
expected: 5, actual: 5
expected: 6, actual: 6
expected: 7, actual: 7
expected: 8, actual: 8
expected: 9, actual: 9
expected: 10, actual: 10
expected: 11, actual: 11
expected: 12, actual: 12
expected: 13, actual: 13
expected: 14, actual: 14
expected: 15, actual: 15
shared memory copy test
expected: 0, actual: 0
expected: 1, actual: 1
expected: 2, actual: 2
expected: 3, actual: 3
expected: 4, actual: 4
expected: 5, actual: 5
expected: 6, actual: 6
expected: 7, actual: 7
expected: 8, actual: 8
expected: 9, actual: 9
expected: 10, actual: 10
expected: 11, actual: 11
expected: 12, actual: 12
expected: 13, actual: 13
expected: 14, actual: 14
expected: 15, actual: 15
time from copy to completion for linear multiply algorithm with shared memory:
1.332064
time from copy to completion for linear multiply algorithm with register memory:
1.599648
finished program
```