CS4402/CS9535b: Quiz 3. **UWO, March 28, 2013.**

Student ID number:	
Student Last Name:	

Guidelines. The quiz consists of two exercises. All answers should be written in the *answer boxes*. No justifications for the answers are needed, unless explicitly required. You are expected to do this quiz on your own without assistance from anyone else in the class. If possible, please avoid pencils and use pens with dark ink. Thank you.

CUDA Cheat Sheet 1: Matrix multiplication

```
#include <iostream>
#include <string>
#include <cassert>
#include <ctime>

using namespace std;

struct cuda_exception {
    explicit cuda_exception(const char *err) : error_info(err) {}
    explicit cuda_exception(const string &err) : error_info(err) {}
    string what() const throw() { return error_info; }

    private:
    string error_info;
};

void checkCudaError(const char *msg) {
    cudaError_t err = cudaGetLastError();
    if (cudaSuccess != err) {
        string error_info(msg);
        error_info += " : ";
        error_info += cudaGetErrorString(err);
        throw cuda_exception(error_info);
    }
}
```

```
template<typename T>
void random_matrix(T *M, size_t height, size_t width, int p = 2) {
    for(size_t i = 0; i < height; ++i) {
        for (size_t j = 0; j < width; ++j) {
            M[i * width + j] = rand() % p;
        }
    }
}

template<typename T>
void print_matrix(const T *M, size_t height, size_t width) {
    if (height >= 32 || width >= 32) {
        cout << "a matrix of height " << height << ", of width " << width <<
        return;
    }

    for(size_t i = 0; i < height; ++i) {
        for (size_t j = 0; j < width; ++j) {
            cout << M[i * width + j] << "   ";
        }
        cout << endl;
    }
    cout << endl;
}

#define BLOCK_SIZE 16

/**
 * CUDA kernel for matrix multiplication, blockwise multiplcation
 *
 * @C, the output matrix C = A * B
 * @A, the first input matrix
 * @B, the second input matrix
 * @wa, is the width of A
 * @wb, is the width of B
 *
 * returns void.
 *
 */
```

```
    template <typename T>
__global__ void matrix_mul_ker(T* C, const T *A, const T *B,
        size_t wa, size_t wb)
{
    // Block index; WARNING: should be at most 2^16 - 1
    int bx = blockIdx.x;
    int by = blockIdx.y;

    // Thread index
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Index of the first submatrix of A processed by the block
    // This is the y-coordinate of the NW corner of the working tile
    int aBegin = wa * BLOCK_SIZE * by;

    // Index of the last submatrix of A processed by the block
    // This is the y-coordinate of the SE corner of the working tile
    int aEnd = aBegin + wa - 1;

    // Step size used to iterate through the submatrices of A
    //
    int aStep = BLOCK_SIZE;

    // Index of the first submatrix of B processed by the block
    // This is the x-coordinate of the NW corner of the working tile
    int bBegin = BLOCK_SIZE * bx;

    // Step size used to iterate through the submatrices of B
    int bStep = BLOCK_SIZE * wb;

    // The element of the block submatrix that is computed by the thread
    // WARNING: This is a local variable for the working thread
    int Csub = 0;

    // Loop over all the submatrices of A and B required to
    // compute the block submatrix
    // This loop iterates  through the tiles in A ndn B that
    // contribute to the working tile of C
    for(int a = aBegin, b = bBegin;  a <= aEnd; a += aStep, b += bStep) {
        // shared memory for the submatrix of A
        __shared__ int As[BLOCK_SIZE][BLOCK_SIZE];

        // shared memory for the submatrix of B
        __shared__ int Bs[BLOCK_SIZE][BLOCK_SIZE];

        // Load the matrices from global memory to shared memory
        // each thread loads one element of each matrix
        As[ty][tx] = A[a + wa * ty + tx];
        Bs[ty][tx] = B[b + wb * ty + tx];

        // synchronize to make sure the matrices are loaded
        __syncthreads();

        // Multiply the two matrices together
        // each thread computes one element of the block submatrix
        for(int k = 0; k < BLOCK_SIZE; ++k) {
            Csub += As[ty][k] * Bs[k][tx];
        }
        // synchronize to make sure that the preceding computation is
        // done before loading two new submatrices of A dnd B in the next ite
        __syncthreads();
    }
    // Write the block submatrix to global memory;
    // each thread writes one element
    int c = wb * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wb * ty + tx] = Csub;
}

template <typename T>
void matrix_mul_dev(T* C, const T* A, const T* B, int ha, int wa, int wb) {

    assert(wa % BLOCK_SIZE == 0);
    assert(wb % BLOCK_SIZE == 0);

    // load A and B to the device
    size_t mem_size = ha * wa * sizeof(T);

    T* Ad;
    cudaMalloc((void **)&Ad, mem_size);
    checkCudaError("allocate GPU memory for the first matrix");
    cudaMemcpy(Ad, A, mem_size, cudaMemcpyHostToDevice);

    T* Bd;
    mem_size = wa * wb * sizeof(T);
    cudaMalloc((void **)&Bd, mem_size);
    checkCudaError("allocate GPU memory for the second matrix");
    cudaMemcpy(Bd, B, mem_size, cudaMemcpyHostToDevice);

    // allocate C on the device
    T* Cd;
    mem_size = ha * wb * sizeof(int);
    cudaMalloc((void**)&Cd, mem_size);
    checkCudaError("allocate GPU memory for the output matrix");

    // compute the execution configure
    // assume that the matrix dimensions are multiples of BLOCK_SIZE
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    size_t dgx = wb / dimBlock.x;
    size_t dgy = ha / dimBlock.y;
    dim3 dimGrid(dgx, dgy);

    // launch the device computation
    matrix_mul_ker<<<dimGrid, dimBlock>>>(Cd, Ad, Bd, wa, wb);
    cudaThreadSynchronize();
    checkCudaError("call the matrix multiplication kernel");

    // read C from the device
    cudaMemcpy(C, Cd, mem_size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(Ad);
    cudaFree(Bd);
    cudaFree(Cd);
}

/**
 * Returns the time spent in seconds
 */
template <typename T>
```

```
double matrix_mul_gpu(T* C, const T* A, const T* B, int ha, int wa, int wb) {
    clock_t t1 = clock();

    // do the multiplication
    matrix_mul_dev(C, A, B, ha, wa, wb);

    clock_t t2 = clock();
    return (t2 - t1) / double(CLOCKS_PER_SEC);
}

/**
 * ha = 2^eha,
 * wa = 2^ewa,
 * wb = 2^ewb;
 *
 * If no parameter entered, then default values are used.
 * If one parameter rentered, it is eha = ewa = ewb = argv[1].
 * If three parameters rentered, there are eha, ewa, and ewb, respectively.
 *
 */
int matrix_mul_test(int argc, char **argv) {
    int *A, *B, *C;
    size_t eha = 4;
    size_t ewa = 4;
    size_t ewb = 4;

    if (argc == 2) {
        eha = ewa = ewb = atoi(argv[1]);
    } else if (argc >= 3) {
        eha = atoi(argv[1]);
        ewa = atoi(argv[2]);
        ewb = atoi(argv[3]);
    }

    size_t ha = (1L << eha);
    size_t wa = (1L << ewa);
    size_t wb = (1L << ewb);

    try {
        A = new int[ha * wa];
        B = new int[wa * wb];
        C = new int[ha * wb];
        random_matrix(A, ha, wa);
        random_matrix(B, wa, wb);

        // timing gpu based method
        cout << matrix_mul_gpu(C, A, B, ha, wa, wb) << "(seconds)" << endl;

    } catch (cuda_exception& err) {
        cout << err.what() << endl;
        delete [] A;
        delete [] B;
        delete [] C;
        return EXIT_FAILURE;
    } catch (...) {
        delete [] A;
        delete [] B;
        delete [] C;
        cout << "unknown exeception" << endl;
        return EXIT_FAILURE;
    }

    print_matrix(A, ha, wa);
    print_matrix(B, wa, wb);
    print_matrix(C, ha, wb);

    delete [] A;
    delete [] B;
    delete [] C;
    return 0;
}

int main(int argc, char** argv) {
    matrix_mul_test(argc, argv);
    return 0;
}
```

Exercise 1. The following C function adds two `float` vectors `iA` and `iB` into oC.

```
 void vector_add (float *iA, float *iB, float* oC, int width) {
    int i;
    for (i=0; i<width ; i++) {
       oC[i] = iA[i] + iB[i];
}
```

Write a CUDA kernel with the same specification, for a 1-D grid, with 1-D thread blocks, assuming that each thread is in charge of computing one element in `oC`.

Exercise 2. Write a CUDA kernel (and the launching code) implementing the reversal of an input integer array `A` od size `n`. This reversing process will be out-of-place. You are asked to proceed in two steps.

(1) First write a "naive" kernel which does not use shared memory.

(2) Then, write a kernel using shared memory.