#define USING\_OPENCV

#include <iostream>

#include <cuda\_runtime.h>

#include <stim/image/image.h>

#include "device\_launch\_parameters.h"

#include <fstream>

#include <ctime>

using std::cerr;

using std::cout;

using std::endl;

using std::ifstream;

void handleError (cudaError\_t error, const char\* text){

if(error != cudaSuccess){

cout << "ERROR " << text << endl;

exit(1);

}

}

class loadPropsFromBinaryFile{

unsigned int r; // rows

unsigned int c; // columns

float\* d; // data

public:

loadPropsFromBinaryFile(){ // default constructor

r = 0;

c = 0;

d = NULL;

}

loadPropsFromBinaryFile(std::string filename){

ifstream infile; // to read the matrix

infile.open(filename, std::ios::in | std::ios::binary); // open as binary input file

if(infile.fail()){ // check for error in opening the file

cerr << "ERROR opening the first file" << endl;

exit(1);

}

infile.read((char\*)&r, sizeof(unsigned int)); // assuming the first value is number of rows

infile.read((char\*)&c, sizeof(unsigned int)); // assuming the second value is the number of columns

d = (float\*) malloc (r \* c \* sizeof(float)); // allocate memory for the matrix

infile.read( (char\*)d , sizeof(float) \* r \* c); // read and copy the data to the memory

infile.close();

}

\_\_host\_\_ \_\_device\_\_ unsigned int rows(){ // return number of rows

return r;

}

\_\_host\_\_ \_\_device\_\_ unsigned int cols(){ // return number of columns

return c;

}

\_\_host\_\_ \_\_device\_\_ float\* data(){ // return the pointer to matrix data

return d;

}

};

\_\_global\_\_ void kernelMatMultiplication( loadPropsFromBinaryFile matA, loadPropsFromBinaryFile matB, float\* matA\_data, float\* matB\_data, float\* matC\_data ){

size\_t idx\_x = blockIdx.x \* blockDim.x + threadIdx.x; // index to the threads across x dimension

size\_t idx\_y = blockIdx.y \* blockDim.y + threadIdx.y; // index to the threads across y dimension

if(idx\_x > matB.cols() || idx\_y > matA.rows()) return;

extern \_\_shared\_\_ float sharedBytes[];

float\* sharedA = sharedBytes;

float\* sharedB = sharedBytes + blockDim.y \* matA.cols() \* sizeof(float);

float\* sharedC = sharedB + blockDim.x \* matB.rows() \* sizeof(float);

//if(idx\_x == 0 && idx\_y == 0)

//printf("%f %f\n", sharedBytes[]);

}

float\* hostMatMultiplication(loadPropsFromBinaryFile a, loadPropsFromBinaryFile b){

float\* c = (float\*) malloc (a.rows() \* b.cols() \* sizeof(float)); // allocate memory for A \* B

float temp = 0;

for(int i = 0 ; i < a.rows() ; i++){ // using matrix A as reference to assign the indices

for(int rowIdxA = 0 ; rowIdxA < a.rows() ; rowIdxA++){

for(int colIdxA = 0 ; colIdxA < a.cols() ; colIdxA++)

temp += a.data()[colIdxA \* a.rows() + rowIdxA] \* b.data()[colIdxA + a.cols() \* i]; // store the dot product in a register

c[i \* a.rows() + rowIdxA] = temp; // store in heap

temp = 0;

}

}

return c;

}

int main(int argc, char \*argv[]){

loadPropsFromBinaryFile A(argv[1]); // read binary file A

loadPropsFromBinaryFile B(argv[2]); // read binary file B

if(A.cols() != B.rows()){ // check if the right marices are passed into the main function

cerr << "ERROR: the dimensions of the two matrices don't agree for multiplication" << endl;

exit(1);

}

// profiling

float\* C = NULL; // pointer to mutliplication result matrix on cpu

C = hostMatMultiplication(A,B); // implementation of matrix multiplication on cpu

// profiling

cudaError\_t error;

cudaDeviceProp prop; // for the device properties

error = cudaGetDeviceProperties(&prop, 0); // getting the device properties

handleError(error, "getting the device properties");

// allocating memory for matrix A on the device

float\* device\_A;

size\_t bytesA = A.rows() \* A.cols() \* sizeof(float);

error = cudaMalloc((void\*\*)&device\_A, bytesA);

handleError(error,"allocating memory on the device for matrix A");

// allocating memory for matrix B on the device

float\* device\_B;

size\_t bytesB = B.rows() \* B.cols() \* sizeof(float);

error = cudaMalloc((void\*\*)&device\_B, bytesB);

handleError(error,"allocating memory on the device for matrix B");

// allocating memory for matrix A \* B = C on the device

float\* device\_C;

size\_t bytesC = A.cols() \* B.rows() \* sizeof(float);

error = cudaMalloc((void\*\*)&device\_C, bytesC);

handleError(error,"allocating memory on the device for matrix C");

// copying matrix A from host to device

error = cudaMemcpy(device\_A, A.data(), bytesA, cudaMemcpyHostToDevice);

handleError(error, "copying matrix A from host to device");

// copying matrix B from host to device

error = cudaMemcpy(device\_B, B.data(), bytesB, cudaMemcpyHostToDevice);

handleError(error, "copying matrix B from host to device");

dim3 threads( sqrt(prop.maxThreadsPerBlock) , sqrt(prop.maxThreadsPerBlock) ); // two dimensional block configuration

dim3 blocks( ceil(B.cols()/threads.x) , ceil(A.rows()/threads.y)); // two dimensional grid configuration

size\_t sharedBytesPerBlock\_A = threads.y \* A.cols() \* sizeof(float);

size\_t sharedBytesPerBlock\_B = threads.x \* B.rows() \* sizeof(float);

size\_t sharedBytesPerBlock\_C = threads.x \* threads.y \* sizeof(float);

size\_t sharedBytesTotal = sharedBytesPerBlock\_A + sharedBytesPerBlock\_B + sharedBytesPerBlock\_C;

if(sharedBytesTotal > prop.sharedMemPerBlock){

cout << "ERROR: insufficient shared memory" << endl;

exit(1);

}

kernelMatMultiplication<<<blocks, threads, sharedBytesTotal>>>(A, B, device\_A, device\_B, device\_C);

//cudaThreadSynchronize();

//cudaFree(device\_A);

return 0;

}