ECSE 420: Parallel Computing – Fall 2019

Lab 2 Report: CUDA Convolution and Matrix Inversion

Group 26

Anna Bieber - 260678856

Walid Chabchoub – 260677008

Date of Submission: Monday November 4th, 2019

Instructor: Prof. Zeljko Zilic

I. Introduction

The goal of this lab is to implement code for simple signal processing and parallelize it using CUDA. The first part of this lab is a continuation of the previous one, where a convolution operation is performed on an image. The second part consists in performing matrix inversion as well as matrix and vector multiplication. The implementation, testing and results are detailed in the next sections.

II. Convolution

a. Implementation

In a similar fashion to max pooling, convolution travels throughout the image pixels using a window and performing weighted sum of the pixel in the center and its neighbors inside that window. This process can be summed up by the equation below:

$$output[i][j] = \sum_{ii=0}^{2} \sum_{jj=0}^{2} input[i+ii-1][j+jj-1]w[ii][jj]$$

For $1 \leq i \leq m-1$ and $1 \leq j \leq n-1$ where m is the number of rows in the input image, and n is the number of columns in the input image. We have set up the kernel to have dimensions very similar to how we distributed them in Lab 1. In fact, we are navigating through the image array using x + width * y but using the block ids and thread ids as well as the grid & block dimensions. That is, we are adding the result of x + width * y to the pointer address of the char array representing the image.

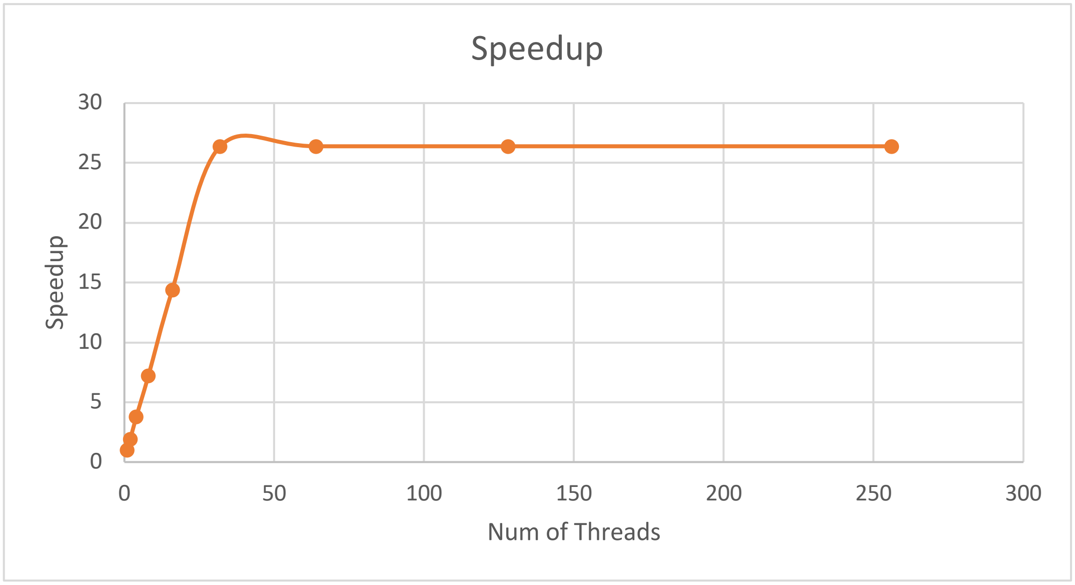
However, in order to implement convolution, we decided to approach it from a channel perspective similarly to what was done for pooling in Lab1, that is, focusing on treating each channel on its own. Nevertheless, each thread is in charge of one 2x2 pool in the entire image, so each thread is computing the maximum of one convolution window (using threadIdx.x and blockIdx.x as well as blockIdx.y). Thus, for the entire image, we would iterate through the 4 channels (R, G, B and Alpha) and determine the block offset to access the line below and above the current pixel at the center of the window.

Finally, once we access these pixels, we multiply their value by their corresponding weight in the weight matrix (using their index).

b. Testing

The testing procedure that we implemented for convolution ressembles the one we performed for pooling and is fairly simple and boiled down to two parts which we deemed sufficient for our application. First we would ensure that our program was outputting an image of the correct dimensions (that is, its height and width are reduced by dimensions of weight matrix – 1, so for 3x3, we would reduce the height and width by). Next, we would use the test script provided to us in Lab 1 to assess the MSE between the reference output image that was given to us for a specific weight matrix (3x3, 5x5 or 7x7) and our obtained output image.

c. Speed Up



III. Matrix Inversion and Linear System Solution

a. Implementation

We begin with an input matrix *A* and a vector *b*, and find a vector *x* such that *Ax* = *b*. To be able to find vector *x*, two steps are required: perform matrix inversion on *A* and multiply with *b*. Before thinking about parallelization, we wanted to be sure that the core logic was working, so we implemented the code without threads and blocks. Then, once the inverse matrix was obtained, we could proceed to implement the multiplication. This code was implemented before we realized that we would not be able to parallelize the inverse matrix process, thus the code is very intricate and cannot be split up into two sections. Thus, we decided to create a method only for the multiplication that we will able to parallelize. The two methods implemented are detailed below.

The core logic of the first method which performs a gaussian elimination and matrix multiplication is done using 4 for loops. The first for loop iterates over N (NxN being the size of the matrix), we then want to find the biggest row and swap it, this is done over all N. The swapping function consists of swapping the two rows of the Matrix A, which is done with temp values. The rows are also swapped in b to ensure a continuity. Once this is done, we perform some simple arithmetic functions to A and b to find the vector x.

The simple matrix multiplication consisted of two for loops iterating over the rows and columns, multiplied A and x, added all the values of the column and outputted it to a vector b. Then, we attempted to a remove the for loops and replace them with threads. We believe our implementations is correct, but the code would always output zero for the b vector. The outermost for loop was replaced with a index based of the thread index as well as block offset, however this did not work.

b. Testing

The testing was split up into three sections. Initially, we wanted to test that the matrix inversion and multiplication was giving us the correct output vector. We inputted multiple sample matrixes and checked them against online calculators to ensure that we had the correct results. Then we used the A_10 and b_10 provided and got the following output:

$$\{0.994357, 2.00452, 2.99916, 4.00494, 5.00008, 6.99566, 7.00046, 7.99974, 9.00029, 0.99980\}$$

Next, we implemented the simple matrix multiplication function and used the provided A and X matrices to get a vector b, we checked the obtained results against the provided b vector. So far, we encountered no issues.

Then we tried implementing the matrix multiplication with blocks and threads and ran into multiple issues. Multiple implementations were attempted, but we got the same results every time: vector b consisting of only zeros. We believe it might have something to do with indexing, however we were unable to fix the problems leaving us without a fully functioning parallelized implementations. The code compiles without errors.

c. Speed up

As we were unable to properly parallelize the matrix multiplication, we have no data or graphs to provide. However, we are expecting the speedup to have a similar shape to the one in the previous section and the previous labs. It should speedup significantly between 1 and 32 threads then it will speedup slightly between 32 and 64, and finally it will stagnate after 64 threads.

IV. Conclusion

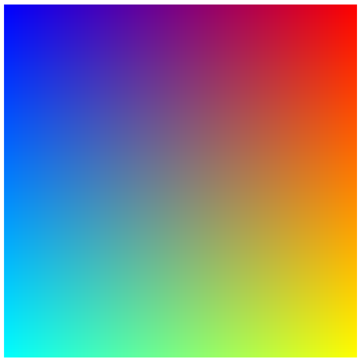
The goal of this experiment was to implement code for simple signal processing and linear system solution, then parallelize it using CUDA. By implementing the convolution of an image, the concepts of parallelizing more complicated operations on pixels is understood. Through the implementation of matrix inversion and linear system solutions, we were able to understand that not all parts of an implementation can be parallelized (depending on the algorithm).

This lab has allowed for a broader understanding of the CUDA infrastructure as well as different use cases.

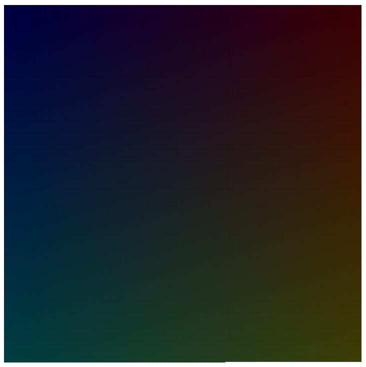
APPENDIX

A. Original test images & output images for convolution

Original

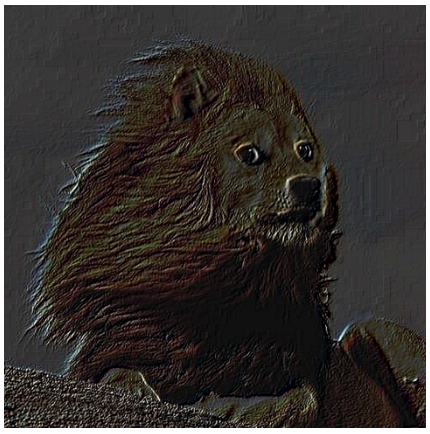


Convoluted

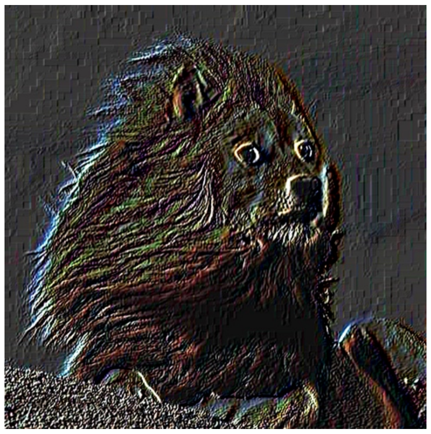


Original

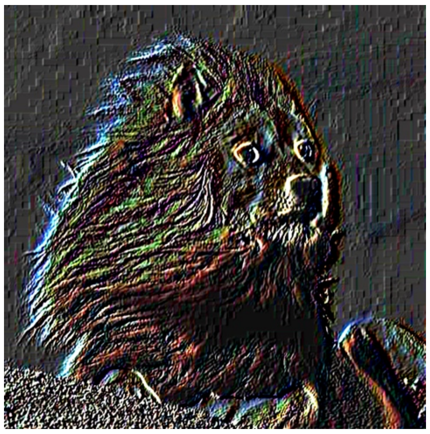
3x3



5x5



7x7



B. Convolution Code

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include "lodepng.h"
#include "wm.h"

#include <math.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>


void convoluteWithCuda(unsigned char* image_in, unsigned char* image_out, unsigned num
Pixels_in, unsigned numPixels_out, unsigned sizeChars_in, unsigned sizeChars_out, unsi
gned width, unsigned height, unsigned dimensions_w, unsigned int threads_per_block);

__global__ void convolute(unsigned char* dev_image_in, unsigned char* dev_image_out, u
nsigned width_in, unsigned height_in, float* dev_weights, unsigned dimensions_w, unsig
ned offset)
{
    // Numbers of blocks in a grid
    unsigned gridWidth = gridDim.x;
    // Fetching index of current block in 1-
D (in comparison to the entire grid - equivalent of x + y * width)
    int currentBlockIndex = blockIdx.x + blockIdx.y * gridWidth;

    // Numbers of threads per block
    unsigned blockWidth = blockDim.x;
    // Fetching index of current thread in 1-
D inside the current block (in comparison to the entire grid)
    int currentThreadIndex = threadIdx.x + currentBlockIndex * blockWidth;

    // In case total numbers of pixel is higher than total numbers of threads availabl
e, then will use offset
    // This offset will "pickup" the rest of the work where it was last left.
    int currentPixelStartIndex = currentThreadIndex + offset;

    // Computing the offsets between blocks representing convolution windows
    int blocksOffset = (((currentPixelStartIndex) / (width_in - (dimensions_w - 1))) *
 (width_in)+((currentPixelStartIndex) % (width_in - (dimensions_w-1))));

    // computing the increment factor to navigate between these convolution windows in
 both the y and x axis
    int i = (blocksOffset) / (width_in);
    int j = (blocksOffset) % (width_in);

    // Pointer with which we will navigate through out the 2x2 pool window
    unsigned char* currentPixelPointer;

    // Variable that will hold the current value of a channel
    unsigned channelValue;
    signed convolutedValue;

    // We will be traversing channel by channel throughout the entire image
    // to fetch the maximum of each channel even if it is not in the same image inside
 a pool
    for (int channelIndex = 0; channelIndex < 4; channelIndex++) {
        // For each RGB channel
        if (channelIndex < 3) {
            convolutedValue = 0;
            // Applying formula from handout
            for (int ii = 0; ii < dimensions_w; ii++) {
                for (int jj = 0; jj < dimensions_w; jj++) {
                    // Fetching pointer location for current pixel
                    currentPixelPointer = (dev_image_in + 4 * ((i + ii) * (width_in)+(
j + jj)));
                    // Fetching the current RGB channel value at this current pixel
                    channelValue = currentPixelPointer[channelIndex];
                    // Applying convolution using the weight matrix value correspondin
g to its index
                    convolutedValue += channelValue * dev_weights[ii * 3 + jj];
                }
            }
            // Making sure we don't go above or below the limits of a pixel value
            if (convolutedValue < 0) {
                convolutedValue = 0;
            }
            if (convolutedValue > 255){
                convolutedValue = 255;
            }
        }
        // Alpha channel leaving it at 255 for no convolution
        else {
            convolutedValue = 255;
        }
        // After performing the convolution of the entire image for a specific RGB cha
nnel, paste the convolution
        // results to our output image pointer
        currentPixelPointer = (dev_image_out + 4 * ((i) * (width_in - (dimensions_w -
1)) + (j)));
        currentPixelPointer[channelIndex] = convolutedValue;
    }
}

int main(int argc, char* argv[])
```

```c
{
    if (argc < 3) {
        printf("Not enough arguments.");
        return 1;
    }

    // Extracting CLI arguments
    char* input_filename = argv[1];
    char* output_filename = argv[2];
    int dimensions_w = atoi(argv[3]);
    int threads_per_block = atoi(argv[4]);


    if (threads_per_block > 1024) {
        printf("Error, number of threads per block cannot exceed 1024. Please try another value.");
        return 1;
    }

    unsigned error;
    unsigned char* inputImage, * outputImage;
    unsigned width, height, width_out, height_out;

    // Decoding and loading the PNG image to the image pointer and setting the width and height values as well
    error = lodepng_decode32_file(&inputImage, &width, &height, input_filename);
    if (error) printf("error %u: %s\n", error, lodepng_error_text(error));

    width_out = width - (dimensions_w-1);
    height_out = height - (dimensions_w - 1);
    unsigned numPixels_in = width * height;
    unsigned numPixels_out = width_out * height_out;


    // Total size of output image is going to be the output number of pixels times the size of a char times 4
    // given that each pixels is 32-bits so 4 chars (4 channels of 8 bits each)
    outputImage = (unsigned char*)malloc(4 * numPixels_out * sizeof(char));

    // Given that we have 4 channels, we have 4 chars per pixel (8 bytes per channel so 32 bits in total per pixel)
    unsigned sizeChars_in = numPixels_in * 4;
    unsigned sizeChars_out = numPixels_out * 4;

    // Method handling device memory setup and launching the kernel
    convoluteWithCuda(inputImage, outputImage, numPixels_in, numPixels_out, sizeChars_in, sizeChars_out, width, height, dimensions_w, threads_per_block);
    // Encoding and saving the new max pooled image in a PNG
    lodepng_encode32_file(output_filename, outputImage, width_out, height_out);

    // Freeing memory
    free(inputImage);
    free(outputImage);
    return 0;
}

void convoluteWithCuda(unsigned char* image_in, unsigned char* image_out, unsigned num
Pixels_in, unsigned numPixels_out, unsigned sizeChars_in, unsigned sizeChars_out, unsi
gned width, unsigned height, unsigned dimensions_w, unsigned int threads_per_block)
{
    // Counter for timing
    double time_elapsed = 0.0;

    // Pointer for device copy of the image
    unsigned char* dev_image_in, * dev_image_out;
    float* dev_weights;

    // Allocating device memory to it
    cudaMalloc((void**)(&dev_image_in), sizeChars_in * sizeof(char));
    cudaMalloc((void**)(&dev_image_out), sizeChars_out * sizeof(char));

    // performing power operation to get the total number of weights in the matrix usi
ng its dimensions
    cudaMalloc((void**)(&dev_weights), pow(dimensions_w, 2) * sizeof(float));

    if (dimensions_w == 3) {
        cudaMemcpy(dev_weights, w3, pow(dimensions_w, 2) * sizeof(float), cudaMemcpyHo
stToDevice);
    }
    else if (dimensions_w == 5) {
        cudaMemcpy(dev_weights, w5, pow(dimensions_w, 2) * sizeof(float), cudaMemcpyHo
stToDevice);
    }
    else {
        cudaMemcpy(dev_weights, w7, pow(dimensions_w, 2) * sizeof(float), cudaMemcpyHo
stToDevice);
    }

    // Perfoming the copying
    cudaMemcpy(dev_image_in, image_in, sizeChars_in, cudaMemcpyHostToDevice);

    // Variables used to create the dim3 struct
    int numXBlocks, numYBlocks;

    // Computing the numbers of blocks (x-axis & y-axis)
    numXBlocks = numPixels_out / threads_per_block;
    numYBlocks = 1;

    // Creating dim3 structs to contain the dimensionality of our GPU that we will be
requiring
    // (Z axis set to 1 for threads and blocks and Y axis set to 1 as well for threads
 because of hardware possible limitationss)
    dim3 totalThreadsPerBlock(threads_per_block, 1, 1);
    dim3 totalBlocks(numXBlocks, numYBlocks, 1);

    printf("Using %d blocks (%d in x, %d in y), each with %d threads .\n", numXBlocks
* numYBlocks, numXBlocks, numYBlocks, threads_per_block);

    // Starting timer
    clock_t star_time = clock();

    // Calling the kernel with an offset value of 0 given that we are at the start (no
 remainder involved yet)
    convolute << <totalBlocks, totalThreadsPerBlock >> > (dev_image_in, dev_image_out,
 width, height, dev_weights, dimensions_w, 0);

    // Given that the division of numPixels_out / threads_per_block can have a remaind
er, then we will be using an "offset"
    // 2^10 - 1 = 1023 is the maximum value of a remainder for decimal numbers divisio
ns so we can use a single block
    int remainder = numPixels_in - (threads_per_block * numXBlocks * numYBlocks);
    printf("Remaining pixels %d.\n", remainder);

    // Offset can never be negative
    unsigned offset = numPixels_out - remainder;
    // Launching another kernel to deal with the remaining pixels and giving the offse
t to locate the starting pixel
    // of these remaining ones
    convolute << <1, remainder >> > (dev_image_in, dev_image_out, width, height, dev_w
eights, dimensions_w, offset);

    cudaDeviceSynchronize();

    // After all pooling is done, copy the processed image from the device memory to h
ost memory (overwriting)
    cudaMemcpy(image_out, dev_image_out, sizeChars_out, cudaMemcpyDeviceToHost);

    // Clear the device memory allocation
    cudaFree(dev_image_in);
    cudaFree(dev_image_out);
    cudaFree(dev_weights);

    // Ending timer
    clock_t end_time = clock();
    time_elapsed += (double)(end_time - star_time) / CLOCKS_PER_SEC;
    printf("Time elapsed for CUDA operation is %f seconds", time_elapsed);

}
```

C. Matrix inversion and multiplication code

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "b_32.h"
#include "a_32.h"
#include "b_10.h"
#include "a_10.h"
#include "x_32.h"

#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include "lodepng.h"

#define mat_elem(a, y, x, n) (a + ((y) * (n) + (x)))
#define N 32

// Struct that defines a matrix with its dimensions and its elements
typedef struct {
    int width;
    int height;
    float* elements;
} Matrix;

//input full arrays a and b, diagonale value, the max_row and size n
void swap_row(float* a, float* b, int r1, int r2)
{
    //create variables for later
    float tmp, * p1, * p2;

    //if the diagonal is the max then we're not swapping cause they will be the same
    if (r1 == r2) return;
    //swap the elements of each row
    for (int i = 0; i < N; i++) {
        p1 = mat_elem(a, r1, i, N);
        p2 = mat_elem(a, r2, i, N);
        tmp = *p1, * p1 = *p2, * p2 = tmp;
    }
    tmp = b[r1], b[r1] = b[r2], b[r2] = tmp;
}
```

```
//input two full arrays a and b, one empty array x and size of x and b
void gauss_eliminate(float a[][N], float b[], float* x)
{
    //create variables for later
    int i, j, col, row, max_row, dia;
    float max, tmp;

    //iterate over the matrix diagonaly, n is the s
    //iterate by row/diagonal
    for (dia = 0; dia < N; dia++) {

        //max_row takes on the current row/diagonal
        //max are the values of the diagonal at every step i.e. (0,0), (1,1), ... , (n
,n)
        max_row = dia, max = a[dia][dia];

        //iterate over the rows
        for (row = dia + 1; row < N; row++)
            //if the aboslute value of A at the row and column is bigger than the diag
onal
            //want to check if the values on the row is bigger than the value where th
e 1 for the identity matrix
            if ((tmp = fabs(a[row][dia])) > max)
                //swap max_row with row and max with temp
                max_row = row, max = tmp;

        //swap rows - see method above
        swap_row(*a, b, dia, max_row);

        //iterate over the rows again
        for (row = dia + 1; row < N; row++) {
            //divide the row and dia by dia, dia (position of the 1 of the identity ma
trix) and store it in a temp variable
            tmp = a[row][dia] / a[dia][dia];

            //iterate over the columns in the matrix
            for (col = dia + 1; col < N; col++)
        //calculate the new value at every column and row
                a[row][col] -= tmp * a[dia][col];
            a[row][dia] = 0;
          //calculate the new value for b for each row
            b[row] -= tmp * b[dia];
        }
    }

    double temp;
    //find x, iterate over the rows in A, b and x
    for (int row = N - 1; row >= 0; row--) {
        //store the value of b[i] in a temp variable
        temp = b[row];
        //iterate over the column in A and multiply by the value of b
        for (int j = N - 1; j > row; j--)
            temp -= x[j] * a[row][j];
        //divide by A
        x[row] = temp / a[row][row];
    }
}

// Multiply kernel
__global__ void multiply(Matrix dev_A, Matrix dev_x, Matrix dev_b)
{

    float b_value = 0;
    // fetching current num of row and num of col according to the current block index
es and thread indexes
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    // iterating through an entire row of A and x
    for (int currentCol = 0; currentCol < dev_A.width; currentCol++)
        // Performing the multiplication and iterating through the array and vector us
ing the x + y * width formula
        b_value += dev_A.elements[row * dev_A.width + currentCol] * dev_x.elements[cur
rentCol * dev_x.width + col];
    // putting the sum of the row of A and x into an index of B
    dev_b.elements[row * dev_b.width + col] = b_value;
}

int main(void)
{
    float a[N][N] = { {1, 4}, {2, 4} };
    float b[N];

    int number_threads = 32;
    //float x[N];
    int i;

    Matrix A;
    A.height = N;
    A.width = N;
    A.elements = A_32;

    float b_elem[N] = {
             1, 2, 3, 4, 5, 6, 7, 8, 9, 1,
             2, 3, 4, 5, 6, 7, 8, 9, 1, 2,
             3, 4, 5, 6, 7, 8, 9, 1, 2, 3,
             4, 5
    };
    Matrix B;
    B.height = N;
    B.width = 1;
    B.elements = b_elem;

    Matrix x;
    x.height = N;
    x.width = 1;
    x.elements = X_32;

    float b_10[10] =
    {
     2046, 4845, 4189, 3989, 7990, 6908, 7677, 7187, 8653, 6864
    };

    //If running the matrix inversion and multiplication together run the following
    //gauss_eliminate(A_10, b_10, x);
    //gauss_eliminate(a, b, x);

    //if running only the multiplication part run the following instruction.

    Matrix dev_A32;
    dev_A32.height = A.height; dev_A32.width = A.width;
    size_t size = A.width * A.height * sizeof(float);
    // allocating memory for matrices and copying the host copy to the device
    cudaMalloc(&dev_A32.elements, size);
    cudaMemcpy(dev_A32.elements, A.elements, size,cudaMemcpyHostToDevice);

    Matrix dev_x;
    dev_x.height = x.height; dev_x.width = x.width;
    size = x.width * x.height * sizeof(float);
    cudaMalloc(&dev_x.elements, size);
    cudaMemcpy(dev_x.elements, x.elements, size, cudaMemcpyHostToDevice);

    Matrix dev_b;
    dev_b.width = B.width; dev_b.height = B.height;
    size = B.width * B.height * sizeof(float);
    cudaMalloc(&dev_b.elements, size);

    dim3 dimBlock(16, 16);
    dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
    multiply << <dimGrid, dimBlock >> > (dev_A32, dev_x, dev_b);

    cudaMemcpy(B.elements, dev_b.elements, size,cudaMemcpyDeviceToHost);

    for (i = 0; i < N; i++)
```

```c
        printf("%f\n", B.elements[i]);

    return 0;
}
```