# ECSE 420: Parallel Computing Lab 2

November $4^{th}$, 2019

David DAlleva - 260689096
Jingxu Hu - 260606017

Lab Group: 39
Professor: Zeljko Zilic

# 1 Convolution

Convolution was a simple algorithm to design and run sequentially, but, parallelizing the same algorithm was tricky to say the least. There were a lot more factors at play such as the mapping between the larger pixel positions and the new smaller pixel positions of the output image, when to shift window to next row of pixels, and when to stop the kernel from going out of the image bounds.

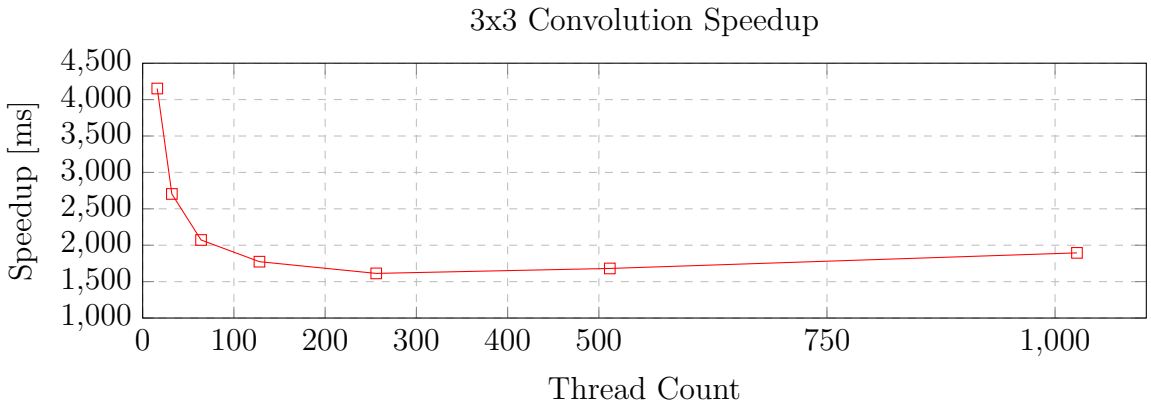
In sequential form, the algorithm only needed the information of a reference pixel. This reference pixel could be any of the 9 pixels in the 3x3 kernel, or any of the 49 pixels in the 7x7, or any of the 25 pixels in the 5x5. Once the starting offset was determined, all the other pixel locations in the kernel window were just offsets from the reference pixel. To move to the reference pixel's neighbors, add 4 each time to reference position. This could be done for any of the RGBA channels. To move the the next row of pixels in the kernel, add 4*imgWidth + the appropriate offsets. Thus on each iteration, the kernel window slides over by 1 pixel. We loop through the original image array by increments of 1 pixel at a time to find the reference's new position. For the actual convolution, on each iteration, we extract and flatten all RGBA channels in the kernel's window. Then we multiply them by the flattened kernel's weights. Finally, summing all the RGBA channels for all pixels in the window and storing them in the output image array. This process was repeated until the convolution window's reference starting position had hit a certain pixel position in the image array where any further offsetting would bring the kernel off of the image plane. Please note, this ending pixel location would change based on how one chose the starting reference pixel's position in the kernel's window.

In parallel form, the algorithm remained the same for the most part. The new issue was figuring out how to map the convoluted RGBA pixel channels into the smaller output array while running however many threads in parallel. Since many threads are performing the nxn convolution at different offsets in the input image array, they require a way to map their outputs into the correct locations of the smaller image array. In order to ensure correct mapping at all offsets, we decided to loop through the size of the smaller image array when sliding the kernel's window to different offsets based on the number of threads being ran at the time. Since now the reference pixel's starting positions correspond to the smaller image indices, we need to remap the indices back into the indices of the larger input image array. After the remapping was done, the convolution procedure was identical to the sequential form with all the RGBA convoluted channels all inserted at the correct positions.

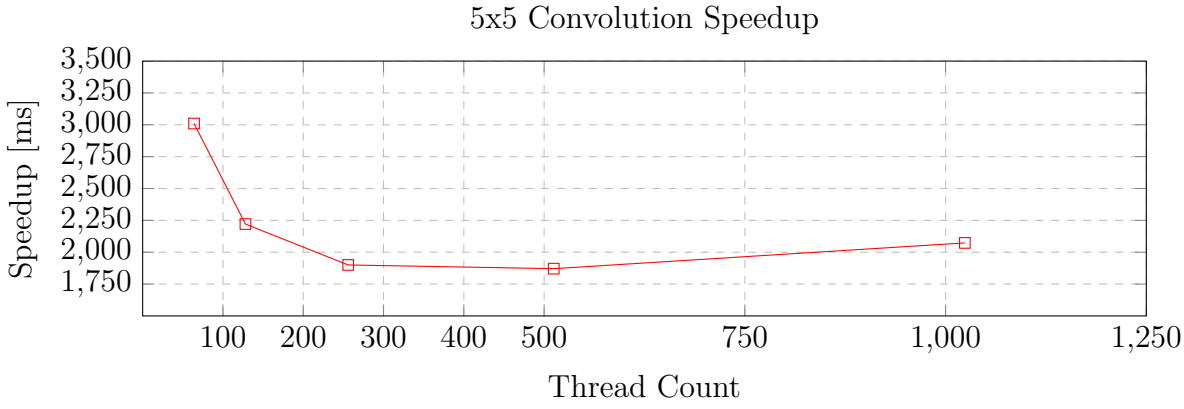
Throughout the duration of the lab, we've noticed some interesting phenomena about how the convolution algorithm actually executed. Firstly, convolution between image kernel's weights and the pixels of an image is expensive because of the amount of of-setting that needs to happen. For a very large image and for a small kernel size, the whole convolution algorithm requires the kernel's window to slide row by row until the ending condition as was discussed in the previous paragraphs. The amount of sliding is proportional to (imgWidth*imgHeight) which is the amount of pixels in the image. This value for very large images are on the order

of millions. Thus, when this operation was parallelized, each thread had to handle a certain range of offsets when itself was ofsetted to a starting pixel location in the image. We noticed that for each kernel size, there was a minimum thread count number that the program could execute with and successfully finish the convolution. The reason being, the threads within the GPU have a timer on how long they will run before all timing out. If one chooses a number of threads to run such that the amount of time to service all convolution locations in the image per thread is less than the time allotted by GPU for that thread, then the convolution would be successful. If it is the opposite, then one would see a blank output.

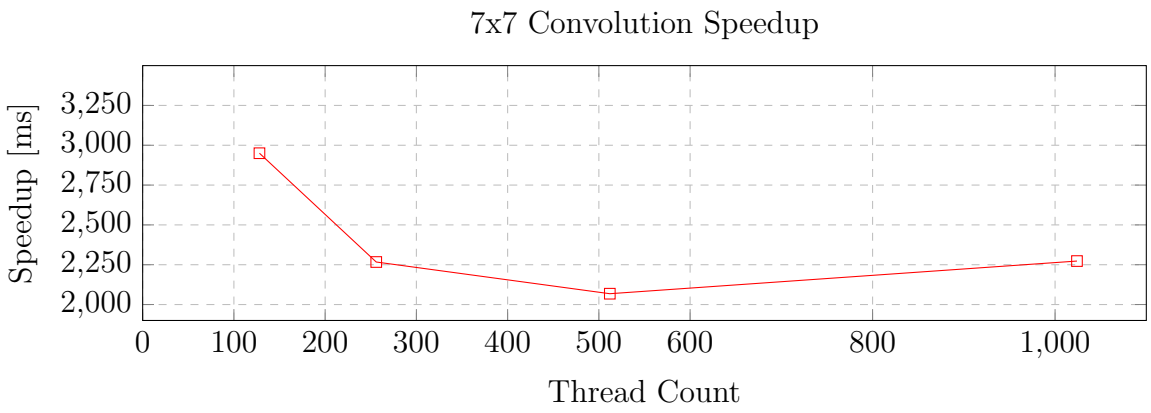
As for the speedup, for each kernel size we see there is a certain number of threads where the time it takes to finish the convolution begins to increase again. This is because for CUDA, when you run with more threads than what is necessary, there's more overhead in managing the threads. Thus, the run time increases again.




For the 3x3 convolution, the minimum thread count used to run the algorithm was 16. Any thread numbers lower would not produce the correct output image. It would show grey or white instead.




For the 5x5 convolution, the minimum thread count used to run the algorithm was 64. Any thread number lower would not produce the correct output image. It would show grey or white instead.




For the 7x7 convolution, the minimum thread count used to run the algorithm was 128. Any thread numbers lower would not produce the correct output image. It would show grey or white instead.

Figure 1: Original

Figure 2: Convolution 3x3

Figure 3: Convolution 5x5

Figure 4: Convolution 7x7

## 2 Matrix Inversion and Linear System Solution

For this section of the lab, the first step was to invert the input matrix A. We were able to

do this part sequentially, using reference code inspired from [1]. Our inversion only works in a realistic duration for matrices of sizes up to 10 rows by 10 columns. Larger matrices (32, 512, 1024 would need to be already inverted before they get inputted) due to running time constraints. The inversion part uses a sequential algorithm based off the Gauss-Jordan elimination technique which uses the GaussJordan helper function. With the Gauss-Jordan elimination method, the goal is to put the identity matrix on the right side and to use row reduction techniques for it to end up on the left side. When came time to multiply the inverse of A by the b matrix, the process needs to be done in parallel using Cuda threads. The threads were generated by the following line of code: int r = (blockIdx.x * blockDim.x) + threadIdx.x. Each thread will take care of multiplying one row of the matrix A by one row of the matrix b. Thus, it is important to note that the number of threads defined by the user must be greater or equal than then number of of rows in the inverted matrix A.

It is also interesting to see how the speedup varied with the number of threads for this problem. Since we are always using a number of threads that is equal or greater than the number of rows, we will have the exact number of threads needed to run the algorithm in parallel or some threads left over. Therefore, when the size of the input matrix stays constant (for our case we collected speedup values for a 32 by 32 matrix A), we are using the minimal amount of threads needed (32). The added threads are therefore not necessary in making the algorithm run quicker, which results in a rather constant function plot of the speedup. The time actually starts increasing when we keep adding threads because of the overhead associated with the instantiated of many threads.

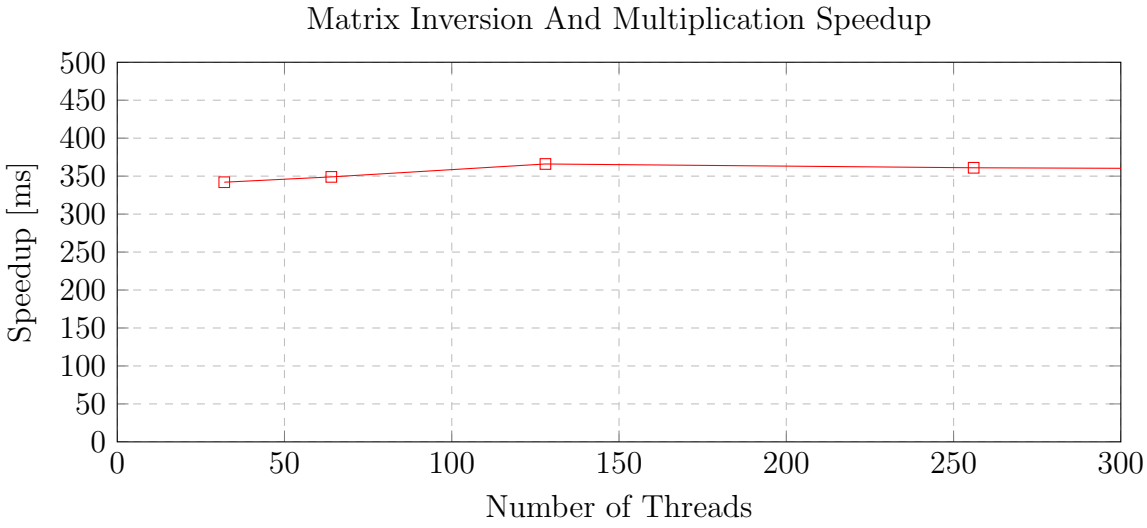



Figure 7: Speedup for Matrix Inversion  Multiplication

# 3 Appendix

The code within this section is a bit off the page. We tried to make it more in line but it didn't work :(. Most of the code is readable but full version could be found within our zip file that we had to submit with our report.

```
// Imports
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include "lodepng.h"
#include "wm.h"
#include "A_10.h"
#include "b_10.h"
#include "b_32.h"
#include "b_512.h"
#include "b_1024.h"
#include "X_512.h"
#include "X_1024.h"
#include "X_32.h"
#include "A_32.h"
#include "A_512.h"
#include "A_1024.h"
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <cuda_runtime.h>
#include <iostream>
using namespace std;
#define max_Threads 1024 // Max num of threads
#define N 32  // Order of the matrix have to change this every time you run a different

// Final version of convolution kernel
__global__ void convolutionKernelFinalVersion(unsigned char* input, unsigned inputSize,
unsigned char* output, unsigned outputSize, unsigned outWidth,
int kernelSize, int numThreads, float* w) {
    // Finding the offset for where the program is starting to do the convolution
int rgbaOffset = (blockIdx.x * blockDim.x + threadIdx.x);
printf("%d\n", rgbaOffset);
    // Looping to do the convolution
for (int i = rgbaOffset; i < outputSize; i += numThreads) {
unsigned colSmall = i % (4 * outWidth);
unsigned rowSmall = i / (4 * outWidth);
unsigned bRow = rowSmall * 4 * inputWidth;
```

```
unsigned largeOffset = bRow + colSmall;
        // Doing the direct convolution in the 3x3, 5x5, 7x7 window
float sumRGBA = 0.0;
for (int j = 0; j < kernelSize; ++j) {
for (int k = 0; k < kernelSize; ++k) {
sumRGBA += input[largeOffset + 4 * k + 4 * inputWidth * j] * w[j * kernelSize + k];
}
}
        // Clamping the RBGA values
if (sumRGBA > 255.0 || (i + 1) % 4 == 0) {
sumRGBA = 255.0;
}
if (sumRGBA < 0.0) {
sumRGBA = 0.0;
}
        // Storing the R,G,B,A channels directly into the smaller image array indices
output[i] = (unsigned char)((int)sumRGBA);
}
}

// Parallelized matrix operations
__global__ void matrixInversionLSSKernel(double* input, double* b, double* x, int width)
{
// Need to run with # of threads that is at least equal to or greater than the dimension
// Row offset to where the threads are doing matrix multiplications with each row
int r = (blockIdx.x * blockDim.x) + threadIdx.x;
//printf("\n%d\n", r);

// The row number has to be less than the dimension of the matrix bc we dont want it to
if (r < width) {
double sum = 0;
for (int k = 0; k < width; k++) {
sum += input[r * width + k] * b[k];
}
// Put sum for each row in the designated row number but since the output is just an Nx1
x[r] = sum;
}
}

// Helper method to load correct kernel
float* loadKernel(int kernelSize) {
// Mallocing for the image weights
float* fw = (float*)malloc(kernelSize * kernelSize * sizeof(float));
```

```
// Finding out which kernel size to use
if (kernelSize == 3) {
printf("Using kernel 3x3\n");
float w3_ar[9] = { 1,2,-1, 2,0.25,-2,1,-2,-1 };
for (int j = 0; j < kernelSize * kernelSize; j++) {
fw[j] = w3_ar[j];
//printf("%f\n", fw[j]);
}
return fw;
}
else if (kernelSize == 5) {
printf("Using kernel 5x5\n");
for (int i = 0; i < kernelSize; i++) {
for (int j = 0; j < kernelSize; j++) {
fw[i * kernelSize + j] = w5[i][j];
}
}
return fw;
}
else if (kernelSize == 7) {
printf("Using kernel 7x7\n");
for (int i = 0; i < kernelSize; i++) {
for (int j = 0; j < kernelSize; j++) {
fw[i * kernelSize + j] = w7[i][j];
}
}
return fw;
}
else {
printf("The kernel size is invalid/n");
return NULL;
}
return NULL;
}

void computeConvolution(char* input_filename, char* output_filename, int numThreads, int
{
// For timing
clock_t start, end;
int duration;

// Start timer
start = clock();
```

```
// Blocks variable
int blocks;
// Checks for num of threads and allocate appropriate number of blocks
if ((numThreads % max_Threads) == 0) {
blocks = (numThreads / 1024);
}
else if (numThreads <= max_Threads) {
blocks = 1;
}
else if (numThreads <= 0) {
printf("\nError: number of threads can not be negative!\n");
return;
}
else {
blocks = (numThreads / max_Threads) + 1;
}
// Checking our logic
printf("\nNumber of blocks: %d and threads: %d needed!\n", blocks, numThreads);
// Defininging image input variables
unsigned err;
unsigned imgWidth;
unsigned imgHeight;
// Host & Device variables
unsigned char* inputImg;
unsigned char* output;
// Testing to load an image and see if any errors occured;
err = lodepng_decode32_file(&inputImg, &imgWidth, &imgHeight, input_filename);
if (err) {
printf("error occurred %u: %s\n", err, lodepng_error_text(err));
}
// Output image width and height
unsigned outputWidth = imgWidth - (kernelSize - 1);
unsigned outputHeight = imgHeight - (kernelSize - 1);
// Input image and output image size
unsigned int inputSize = 4 * imgWidth * imgHeight;
unsigned int outputSize = 4 * outputHeight * outputWidth;
    // Mallocing local memory for the output image array
    output = (unsigned char*)malloc(outputSize*sizeof(unsigned char));
    // Loading the correct kernel to use
    float *fw = loadKernel(kernelSize);
// Allocating device memory
    unsigned char* cudaInput;
```

```
    unsigned char* cudaOutput;
float* fWeightsCuda;
cudaMalloc((void**)& cudaInput, inputSize * sizeof(unsigned char));
cudaMalloc((void**)& cudaOutput, outputSize * sizeof(unsigned char));
cudaMalloc((void**)& fWeightsCuda, kernelSize * kernelSize * sizeof(float));
// Coping over data from cpu memory to GPU memory
cudaMemcpy(cudaInput, inputImg, inputSize * sizeof(unsigned char), cudaMemcpyHostToDevic
cudaMemcpy(fWeightsCuda, fw, kernelSize * kernelSize * sizeof(float), cudaMemcpyHostToDe
// Calling the GPU to parallelize our convolutions
convolutionKernelFinalVersion <<< blocks, (numThreads/blocks) >>> (cudaInput, inputSize,
// Copying from device memory back to cpu memory for the output image array
cudaMemcpy(output, cudaOutput, outputSize*sizeof(unsigned char), cudaMemcpyDeviceToHost)
    // Display and save image to local folder
lodepng_encode32_file(output_filename, output, outputWidth, outputHeight);
free(inputImg);
free(output);
    free(fw);
cudaFree(cudaInput);
cudaFree(cudaOutput);
    cudaFree(fWeightsCuda);
    printf("\nFinished the convolution!\n\n");

// end time
end = clock();
duration = (end - start);
printf("\nTime took to run: %d ms\n", duration);
}


// Function to get cofactor of A[p][q] in temp[][]. n is current
// dimension of A[][]
void getCofactor(double A[N][N], double temp[N][N], int p, int q, int n)
{
int i = 0, j = 0;

// Looping for each element of the matrix
for (int row = 0; row < n; row++)
{
for (int col = 0; col < n; col++)
{
//  Copying into temporary matrix only those element
//  which are not in given row and column
if (row != p && col != q)
```

```
{
temp[i][j++] = A[row][col];

// Row is filled, so increase row index and
// reset col index
if (j == n - 1)
{
j = 0;
i++;
}
}
}
}

}

/* Recursive function for finding determinant of matrix.
   n is current dimension of A[][]. */
int determinant(double A[N][N], int n)
{
int D = 0; // Initialize result

//  Base case : if matrix contains single element
if (n == 1)
return A[0][0];

double temp[N][N]; // To store cofactors

int sign = 1;  // To store sign multiplier

 // Iterate for each element of first row
for (int f = 0; f < n; f++)
{
// Getting Cofactor of A[0][f]
getCofactor(A, temp, 0, f, n);
D += sign * A[0][f] * determinant(temp, n - 1);

// terms are to be added with alternate sign
sign = -sign;
}

return D;
}
```

```
// Function to get adjoint of A[N][N] in adj[N][N].
void adjoint(double A[N][N], double adj[N][N])
{
if (N == 1)
{
adj[0][0] = 1;
return;
}

// temp is used to store cofactors of A[][]
int sign = 1;
double temp[N][N];

for (int i = 0; i < N; i++)
{
for (int j = 0; j < N; j++)
{
// Get cofactor of A[i][j]
getCofactor(A, temp, i, j, N);

// sign of adj[j][i] positive if sum of row
// and column indexes is even.
sign = ((i + j) % 2 == 0) ? 1 : -1;

// Interchanging rows and columns to get the
// transpose of the cofactor matrix
adj[j][i] = (sign) * (determinant(temp, N - 1));
}
}
}

// Function to calculate and store inverse, returns false if
// matrix is singular
bool inverse(double A[N][N], double inverse[N][N])
{
// Find determinant of A[][]
int det = determinant(A, N);
if (det == 0)
{
cout << "Singular matrix, can't find its inverse";
return false;
}
```

```
// Find adjoint
double adj[N][N];
adjoint(A, adj);
printf("\nFinished adjoint\n");

// Find Inverse using formula "inverse(A) = adj(A)/det(A)"
for (int i = 0; i < N; i++)
for (int j = 0; j < N; j++)
inverse[i][j] = adj[i][j] / double(det);

printf("\nFinished inverse\n");
return true;
}


void GaussJordan(double matrix[N][N], int MatrixOrder, double inverse[N][N]) {

double r = 0.000;
//change the size of the matrix here
double bigMatrix[32][32 * 2];
// Create gauss jordan matrix: matrix | identity matrix
for (int i = 0; i < MatrixOrder; i++) {
for (int j = 0; j < MatrixOrder; j++) {
bigMatrix[i][j] = matrix[i][j];
}
}
for (int i = 0; i < MatrixOrder; i++) {
for (int j = 0; j < MatrixOrder; j++) {
if (i == j) {
bigMatrix[i][j + MatrixOrder] = 1;
}
else {
bigMatrix[i][j + MatrixOrder] = 0;
}
}
}
// Apply Gauss Jordan Elimination
for (int i = 0; i < MatrixOrder; i++) {
if (bigMatrix[i][i] == 0) {
exit(0);
}
for (int j = 0; j < MatrixOrder; j++) {
```

```
if (i != j) {
r = bigMatrix[j][i] / bigMatrix[i][i];
for (int k = 0; k < 2 * MatrixOrder; k++) {
bigMatrix[j][k] = bigMatrix[j][k] - (bigMatrix[i][k] * r);
}
}
}
}

// Row operation to make principal diagonal to 1
for (int i = 0; i < MatrixOrder; i++) {
double temp = bigMatrix[i][i];
for (int j = 0; j < MatrixOrder * 2; j++) {
bigMatrix[i][j] = bigMatrix[i][j] / temp;
}
}

// Taking the correct values from the augmented matrix
for (int i = 0; i < MatrixOrder; i++) {
for (int j = MatrixOrder; j < MatrixOrder * 2; j++) {
inverse[i][j] = bigMatrix[i][j];
}
}
}



// Generic function to display the matrix.  We use it to display
// both adjoin and inverse. adjoin is integer matrix and inverse
// is a float.
template<class T>
void display(T A[N][N])
{
for (int i = 0; i < N; i++)
{
for (int j = 0; j < N; j++)
cout << A[i][j] << " ";
cout << endl;
}
}

void matrixMultiplication(double inverted[N][N], double b[N][1], int numThreads, int ord
```

```
// For timing
clock_t start, end;
int duration;

// Start timer
start = clock();

// Blocks variable
int blocks;
// Checks for num of threads and allocate appropriate number of blocks
if ((numThreads % max_Threads) == 0) {
blocks = (numThreads / 1024);
}
else if (numThreads <= max_Threads) {
blocks = 1;
}
else if (numThreads <= 0) {
printf("\nError: number of threads can not be negative!\n");
return;
}
else {
blocks = (numThreads / max_Threads) + 1;
}
// Checking our logic
printf("\nNumber of blocks: %d and threads: %d needed!\n", blocks, numThreads);

// Allocate local memory for the flattened matrices of the original ones
double* flattenedINV = (double*)malloc(N * N * sizeof(double));
double* flattenedB = (double*)malloc(N * 1 * sizeof(double));
double* output = (double*)malloc(N * 1 * sizeof(double));

// Flattening of the matrices
for (int j = 0; j < N; j++) {
for (int i = 0; i < N; i++) {
flattenedINV[j * N + i] = inverted[j][i];
//printf(" %f ", flattenedINV[j * N + i]);
}
}

for (int j = 0; j < N; j++) {
flattenedB[j] = b[j][0];
}
```

```
// Allocate device memory
double* cuda_FlattenedINV;
double* cuda_FlattenedB;
double* x;
cudaMalloc((void**)& cuda_FlattenedB, N*1*sizeof(double));
cudaMalloc((void**)& cuda_FlattenedINV, N*N*sizeof(double));
cudaMalloc((void**)& x, N * 1 * sizeof(double));

// copy memeory from
cudaMemcpy(cuda_FlattenedB, flattenedB, N*1*sizeof(double), cudaMemcpyHostToDevice);
cudaMemcpy(cuda_FlattenedINV, flattenedINV, N*N*sizeof(double), cudaMemcpyHostToDevice);

// run in GPU
matrixInversionLSSKernel <<< blocks, (numThreads / blocks) >>> (cuda_FlattenedINV, cuda_

// copy back to cpu
cudaMemcpy(output, x, N*1*sizeof(double), cudaMemcpyDeviceToHost);

// display output
printf("\nOutput\n");
for (int i = 0; i < N; i++) {
printf("\n%lf", output[i]);
}
printf("\n");
// end time
end = clock();
duration = (end - start);
printf("\nTime took to run: %d ms\n", duration);
}


int main(void) {
 //   // Run with minimum 32 threads or else the algorithm times out before the whole co
//// For 3x3 run with minimum 16
//// For 5x5 run with minimum 32
//// For 7x7 run with minimum 64
//int kernelSize = 7;
//int numThreads = 1024;
 //   char *input = "test.png";
 //   char *output = "convolved7x7.png";
//computeConvolution(input, output, numThreads, kernelSize);

// Gauss Jordan Elimination
```

```
// Output matrix
// If the matrix is of order N then run with minimum N threads or else the matrix multip
// It will compute most of it but then have little bits left over
// Before running our code you need to make sure N is changed to the appropriate order t
/*double inv[N][N];
cout << "\nThe Inverse is :\n";
GaussJordan(A_10, N, inv);
display(inv);*/

/*if (inverse(A_10, inv))
display(inv);*/

/*(invertibleMatrix, bVector, numThreads to run with, order of matrix)*/
matrixMultiplication(A_32, X_32, 2048, N);
}
```

# References

[1] *GeeksforGeeks. (2019). Adjoint and Inverse of a Matrix - GeeksforGeeks. [online] Available at: https://www.geeksforgeeks.org/adjoint-inverse-matrix/ [Accessed 4 Nov. 2019].*

[2] *CUDA - Matrix Multiplication. Tutorialspoint. [Online]. Available at: https://www.tutorialspoint.com/cuda/cuda*$_m$*atrix*$_m$*ultiplication.html*[*Accessed*4*Nov*.2019].