ECSE 420 Parallel Computing

Lab 2 – CUDA Convolution and Matrix Inversion

Daniel Chernis 260707258
David Gilbert 260746680

Introduction

This lab consisted of two parts. First, we applied convolution to an image using 3X3, 5X5, and 7X7 weight matrices. Second, A matrix equation AX = B was solved by finding the inverse of A in serial and then applying matrix multiplication with CUDA on A^-1 * B.

Image Convolution

To compute the convolution, the output image size is calculated the following way:

- Output Width = Width of the original image - weight matrix size + 1
- Output Height = Height of the original image - weight matrix size + 1

Then we loop over the RGBA of that output image and based on the offset, map the row and column to the position of RGBA in the original image in the following way:

- For the row in the original image, divide the current offset by the Output Width.
- For the column, let the offset take the modulo of the Output Width.

Next, calculate the reference pixel in the initial image that maps to the resulting position. That reference point is considered the top left pixel of the block of 3X3, 5X5, 7X7 block.

```
__global__ void convolutionParallel(unsigned char* image, unsigned char* new_image, unsigned height,
unsigned width, int thread_count, int convolution_size)
{
    // process image
    int offset = (blockIdx.x * blockDim.x + threadIdx.x);
    int width_out = (width - convolution_size + 1);
    int height_out = (height - convolution_size + 1);

    //Loop over pixels of smaller image
    for (int i = offset; i < width_out * height_out * 4; i += thread_count)
    {
        int row = i / (4*width_out);
        int col = i % (4*width_out);
        int reference_pixel_offset = 4 * row * width + col;
        float sum = 0.0;
```

Then depending on the weight matrix size, use the appropriate hardcoded weight matrix w in to sum up all the Rs, Gs, or Bs for that block of pixels.

This uses the appropriate offset to calculate the RGB value multiplied by the weight from the weight matrix.

```
for (int j = 0; j < convolution_size; j++)
    for (int k = 0; k < convolution_size; k++)
        sum += image[reference_pixel_offset + 4 * k + 4 * j * width] * w[j * convolution_size + k];
```

Finally, clamp the output of the convolution between 0 and 255 and save the resulting sum to the appropriate position in the output image.

```
if (sum <= 0)           sum = 0;
if (sum >= 255)         sum = 255;
if ((i + 1) % 4 == 0)   sum = 255; // Set a = 255

new_image[i] = (int) sum;
```

Results

Initial Image:

3X3 Convolution

5X5 Convolution

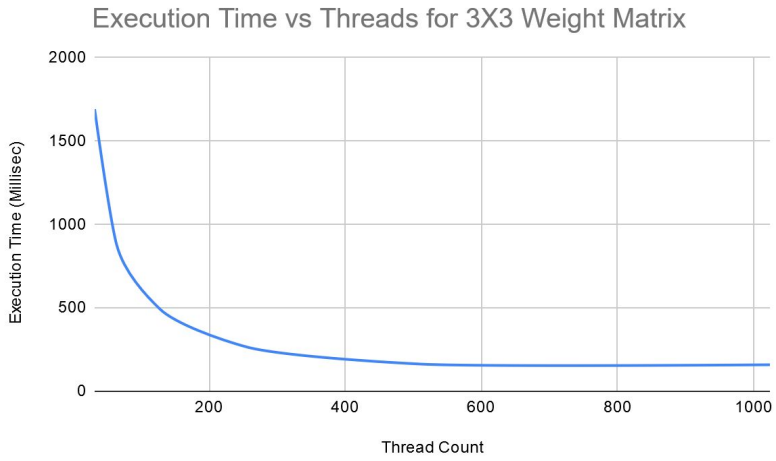
7X7 *Convolution*

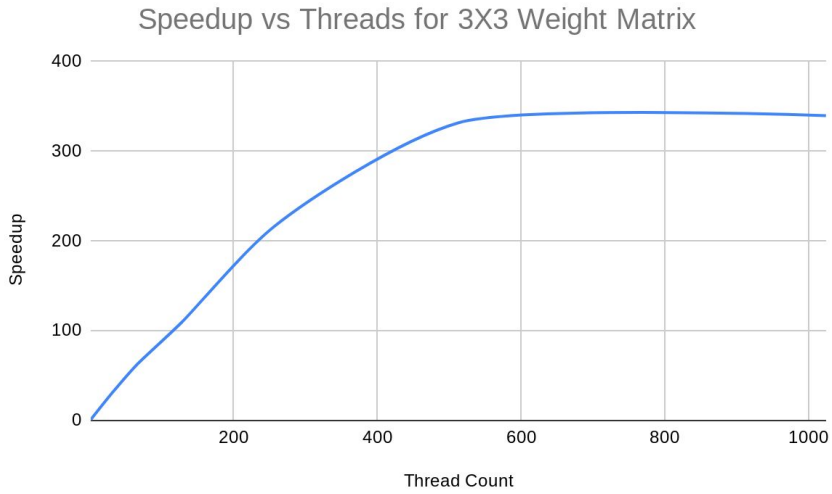
Table:

Execution time vs Thread Count for each Weight Matrix

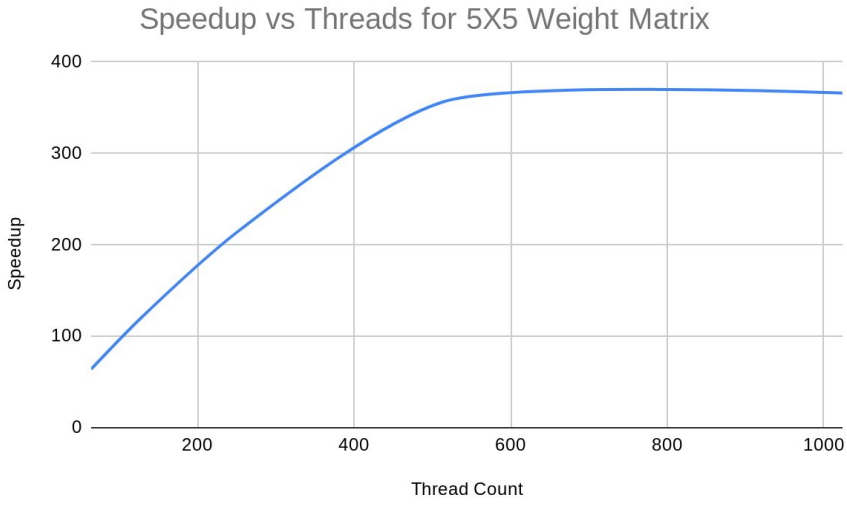
# Thread	Time 3x3 (millisec)	5x5 (millisec)	7x7 (millisec)
1 (original)	174	407	708
1 (adjusted)	54065.79686	121846.75	227448.4687
32	1689.556152	-	-
64	881.740784	1903.855469	-
128	492.338867	1013.135498	1776.941162
256	265.089691	559.841492	926.310974
512	163.032898	342.227356	590.171143
1024	159.131226	332.78894	545.391846

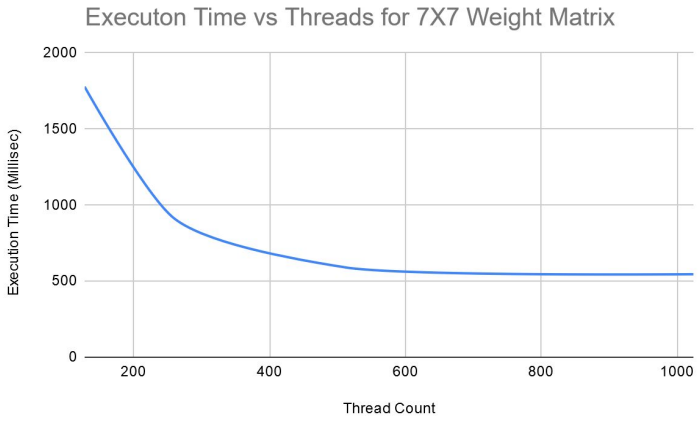
Graphs*

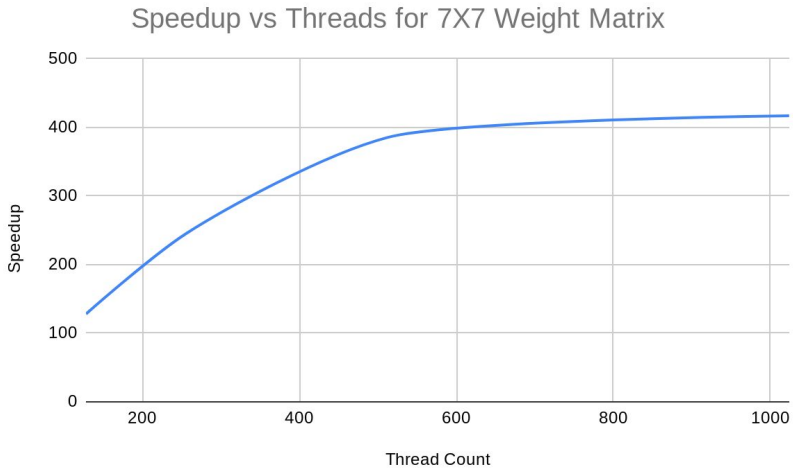





Execution Time vs Threads for 5X5 Weight Matrix
2000
1500
1000
500
0
Execution Time (Millisec)
200
400
600
800
1000
Thread Count


Speedup vs Threads for 5X5 Weight Matrix
400
300
200
100
0
Speedup
200
400
600
800
1000
Thread Count





The Execution time graph display a steady exponential decrease in execution time that starts to converge around 1024 threads.
The Speed Up graphs displays a logarithmic growth in processing speed as the number of threads increase. *Speed up was calculated as follows:

Speed Up = Time for 1-Thread / Time for K-Threads

The execution time of the parallelizable part of the code was measured with different thread number, the speed up was then computed and plotted.

Since the operation gets heavier as the weight matrix increases in size you see an elevation in the plot. Lab machines timeout on smaller # threads so we were only able to get the following:

- 3x3 only works starting 32 threads
- 5x5 only works starting 64 threads
- 7x7 only works starting 128 threads

*Runtime for 1 thread was extrapolated to calculate the speedup. The real time in millisec for 3x3, 5x5, and 7x7 respectively was 174, 407, and 708 millisec which is a lot faster than the time it took for parallel execution (almost 2 seconds).

Matrix Inversion and System Solution

In order to find a vector x such that Ax =b given A and b we needed to compute the inverse of A and then multiply it by b following : x = $A^{-1}b$.

Matrix Inverse

For the inversion we used code from : http://www.sourcecodesworld.com/source/show.asp?ScriptID=1086 using the Gauss Jordan method to inverse the input matrix sequentially . For matrices bigger then 10x10 i.e 32x32 the inversion fails due to precision error even while using double data type (64 bits signed float) and for matrices of size 10x10 the precision of the answer for a given cell is +/-0.00001.

Matrix Multiplication

To do matrix multiplication in parallel we start by establishing the initial position of each thread by dividing the number of cells to process by the number of threads available.
Each thread will sum over the product of a row and a cell following matrix multiplication rules and repeat for as many cells as the thread number requires each thread to do.

```
__global__ void matrixmul(double *a, double *b, double *c, int n)
{
    //starting offset for current thread
    int offset = threadIdx.x *  n / THREAD_COUNT;
    //stop processing cell once youve processed n / number of threads
    int limit = offset + n / THREAD_COUNT;


    if (threadIdx.x < n) {
        //each threads takes care of a limited number of cells ( n / number of threads)
        for (offset; offset < limit; offset++) {
            double sum = 0;
            //iterate over for each cell, all over corresponding sum of product
            for (int i = 0; i < n; i++) {
                sum += a[offset * n + i] * b[i];
            }
            c[offset] = sum;
        }
    }

}
```

Results

For the 10x10 matrix we inverted the matrix A then multiplied the result by the vector b (b_10.h) and then subtracted the result of the multiplication by the initial matrix A to confirm the

correctness of the program, since the difference between the two matrices was the zero matrix 10x1 up to a certain error the correctness was confirmed.

```
int error = 0;
// verify correctness

    for (int j = 0; j < MATRIX_SIZE; j++) {
        error += X_10[j] - d_3[j];
    }
```

Cumulative error was less than 10*10*error_margin_cell (which we established as 0.0001).

In order to find the speed up of parallelizing the multiplication process we used X_1024 the 1024x1 and multiplied it by A_1024 then compared the result with b_1024 for correctness. We then changed the number of threads the process had available and plotted it here :

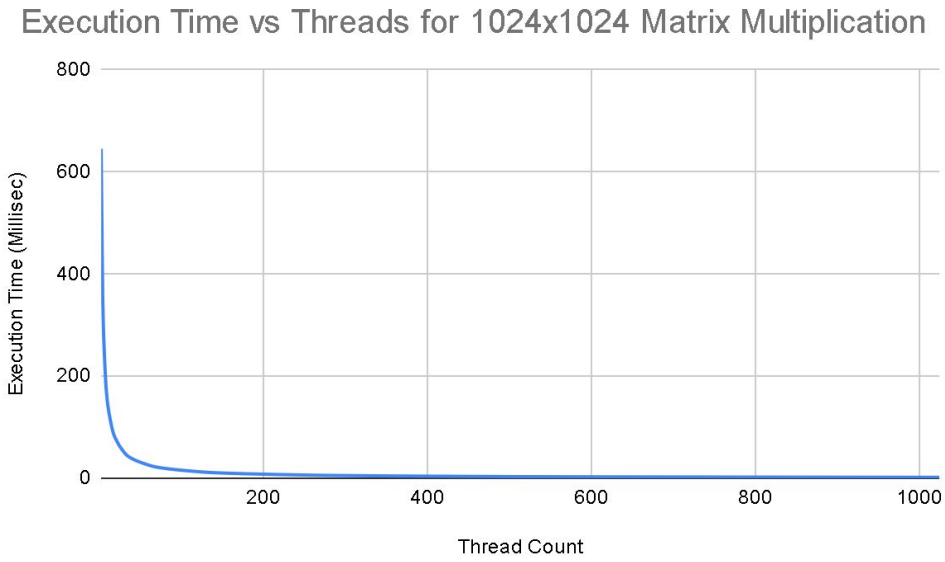


Table of number of threads vs execution time for 1024 x 1024 matrix multiplication

# Threads	Matrix Multiplication execution time (millisec)
2	644.609924
4	343.640076
8	177.038437
16	92.960960
32	46.974625
64	23.745089
128	11.918784
256	6.056544

512	3.096608
1024	1.735456

The speedup was only done over 1024x1024 matrix because the speed up over smaller matrices is very small due to the process itself already being very short (~1 millisec).The correctness for size 32 and up was verified in a similar way as for the 10x10 but instead we were given x_size, so we did the difference between A_size*x_size and b_size and got a zero matrix which confirms correctness.

Appendix

Convolution:

```
__global__ void convolutionParallel(unsigned char* image, unsigned char*
new_image, unsigned height, unsigned width, int thread_count, int
convolution_size)
{
    // process image
    int offset = (blockIdx.x * blockDim.x + threadIdx.x);
    int width_out = (width - convolution_size + 1);
    int height_out = (height - convolution_size + 1);

    //Loop over pixels of smaller image
    for (int i = offset; i < width_out * height_out * 4; i +=
thread_count)
    {
        int row = i / (4*width_out);
        int col = i % (4*width_out);
        int reference_pixel_offset = 4 * row * width + col;
        float sum = 0.0;

        if (convolution_size == 3)
        {
            float w[9] =
            {
              1,    2,      -1,
              2,    0.25,   -2,
```

```
              1,    -2,     -1
            };

            for (int j = 0; j < convolution_size; j++)
                for (int k = 0; k < convolution_size; k++)
                    sum += image[reference_pixel_offset + 4 * k + 4 * j *
width] * w[j * convolution_size + k];
        }

        if (convolution_size == 5)
        {
            float w[25] =
            {
                0.5,    0.75,   1,      -0.75,  -0.5,
                0.75,   1,      2,      -1,     -0.75,
                1,      2,      0.25,   -2,     -1,
                0.75,   1,      -2,     -1,     -0.75,
                0.5,    0.75,   -1,     -0.75,  -0.5
            };

            for (int j = 0; j < convolution_size; j++)
                for (int k = 0; k < convolution_size; k++)
                    sum += image[reference_pixel_offset + 4 * k + 4 * j *
width] * w[j * convolution_size + k];

        }
        if (convolution_size == 7)
        {
            float w[49] =
            {
                0.25,   0.3,    0.5,    0.75,   -0.5,   -0.3,   -0.25,
                0.3,    0.5,    0.75,   1,      -0.75,  -0.5,   -0.3,
                0.5,    0.75,   1,      2,      -1,     -0.75,  -0.5,
                0.75,   1,      2,      0.25,   -2,     -1,     -0.75,
                0.5,    0.75,   1,      -2,     -1,     -0.75,  -0.5,
                0.3,    0.5,    0.75,   -1,     -0.75,  -0.5,   -0.3,
                0.25,   0.3,    0.5,    -0.75,  -0.5,   -0.3,   -0.25
```

```
            };

            for (int j = 0; j < convolution_size; j++)
                for (int k = 0; k < convolution_size; k++)
                    sum += image[reference_pixel_offset + 4 * k + 4 * j *
width] * w[j * convolution_size + k];
        }

        if (sum <= 0)           sum = 0;
        if (sum >= 255)         sum = 255;
        if ((i + 1) % 4 == 0)   sum = 255; // Set a = 255

        new_image[i] = (int) sum;

    }
}
```

Main for Convolution:

```
unsigned error;
    unsigned char* image;
    unsigned char* new_image;
    unsigned int width, height;
    unsigned char* d_image;
    unsigned char* d_new_image;

    int matrix_size_offset = CONVOLUTION_SIZE - 1;

    error = lodepng_decode32_file(&image, &width, &height, "test.png");
    if (error) printf("error %u: %s\n", error, lodepng_error_text(error));

    d_image = (unsigned char*)malloc(height * width * 4 * sizeof(unsigned
char));
    cudaMallocManaged((void**)& d_image, width * height * 4 *
sizeof(unsigned char));
```

```
    d_new_image = (unsigned char*)malloc((width - matrix_size_offset) *
(height - matrix_size_offset) * 4 * sizeof(unsigned char));
    cudaMallocManaged((void**)& d_new_image, (width - matrix_size_offset)
* (height - matrix_size_offset) * 4 * sizeof(unsigned char));


    for (int i = 0; i < height * width * 4; i++) { d_image[i] = image[i];
}

    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    convolutionParallel << < 1, THREAD_COUNT >> > (d_image, d_new_image,
height, width, THREAD_COUNT, CONVOLUTION_SIZE);

    cudaDeviceSynchronize();
    cudaEventRecord(stop);
    cudaEventSynchronize(stop);

    float milliseconds = 0;
    cudaEventElapsedTime(&milliseconds, start, stop);
    printf("%f", milliseconds);
    lodepng_encode32_file("convolutionOut.png", d_new_image, width -
matrix_size_offset, height - matrix_size_offset);

    cudaFree(d_image);
    cudaFree(d_new_image);
```

Matrix Inversion Based of : http://www.sourcecodesworld.com/source/show.asp?ScriptID=1086

```c
int main()
{
    //MATRIX INVERSION --------------------------------
    double** A, ** I, temp;
    int i, j, k;

    I = (double**)malloc(MATRIX_SIZE * sizeof(double*));
    for (i = 0; i < MATRIX_SIZE; i++) {
    I[i] = (double*)malloc(MATRIX_SIZE * sizeof(double));
}
    A = (double**)malloc(MATRIX_SIZE * sizeof(double*));
    for (i = 0; i < MATRIX_SIZE; i++) {
    A[i] = (double*)malloc(MATRIX_SIZE * sizeof(double));
}

    for (i = 0; i < MATRIX_SIZE; i++) {
        for (j = 0; j < MATRIX_SIZE; j++) {
            A[i][j] = A_32[i][j];
        }
    }

    for (i = 0; i < MATRIX_SIZE; i++) {
        for (j = 0; j < MATRIX_SIZE; j++) {
            if (i == j)
                I[i][j] = 1;
            else
                I[i][j] = 0;
        }
    }
```

```
/*---------------LoGiC starts here------------------*/
for (k = 0; k < MATRIX_SIZE; k++)
{
    temp = A[k][k];
    for (j = 0; j < MATRIX_SIZE; j++)
    {
        A[k][j] /= temp;
        I[k][j] /= temp;
    }                                                   //R1=R1-R0*A[1][0] similarly for I
    for (i = 0; i < MATRIX_SIZE; i++)                               //R2=R2-R0*A[2][0]      ,,
    {
        temp = A[i][k];                                 //R1=R1/A[1][1]
        for (j = 0; j < MATRIX_SIZE; j++)                           //R0=R0-R1*A[0][1]
        {                                               //R2=R2-R1*A[2][1]
            if (i == k)
                break;                                  //R2=R2/A[2][2]
            A[i][j] -= A[k][j] * temp;                      //R0=R0-R2*A[0][2]
            I[i][j] -= I[k][j] * temp;                      //R1=R1-R2*A[1][2]
        }
    }
}

const int size_of_b = 1;
printf("test matrix mult \n");
double* inverseMat = (double*)malloc(MATRIX_SIZE * MATRIX_SIZE * sizeof(double));
double* bMat = (double*)malloc(MATRIX_SIZE * sizeof(double));

/*---------------LoGiC ends here--------------------*/
printf("The inverse of the matrix is:               ");
for (i = 0; i < MATRIX_SIZE; i++)
{
    for (j = 0; j < MATRIX_SIZE; j++) {
        inverseMat[i + j * MATRIX_SIZE] = I[i][j];
        printf("%f ", I[i][j]);
    }
    printf("\n");

}
```

Matrix Multiplication:

```
//MATRIX MULTIPLICATION ----------------------------



double* d_1 = (double*)malloc(MATRIX_SIZE * MATRIX_SIZE * sizeof(double));
cudaMallocManaged((void**)& d_1, MATRIX_SIZE * MATRIX_SIZE * sizeof(double));

double* d_2 = (double*)malloc(MATRIX_SIZE * MATRIX_SIZE * sizeof(double));
cudaMallocManaged((void**)& d_2, MATRIX_SIZE * MATRIX_SIZE * sizeof(double));

double* d_3 = (double*)malloc(MATRIX_SIZE * MATRIX_SIZE * sizeof(double));
cudaMallocManaged((void**)& d_3, MATRIX_SIZE *MATRIX_SIZE * sizeof(double));


for (int i = 0; i < MATRIX_SIZE*MATRIX_SIZE ; i++) {

    d_1[i] = A_1024[i];

}
for (int i = 0; i < MATRIX_SIZE ; i++) {

    d_2[i] = X_1024[i];
}

cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start);

//BLOCK_COUNT, THREAD_COUNT/ BLOCK_COUNT
matrixmul << <1, THREAD_COUNT >> > (d_1, d_2, d_3,MATRIX_SIZE);
cudaEventRecord(stop);
cudaEventSynchronize(stop);
cudaThreadSynchronize();
```

```
__global__ void matrixmul(double *a, double *b, double *c, int n)
{
    //starting offset for current thread
    int offset = threadIdx.x *  n / THREAD_COUNT;
    //stop processing cell once youve processed n / number of threads
    int limit = offset + n / THREAD_COUNT;


    if (threadIdx.x < n) {
        //each threads takes care of a limited number of cells ( n / number of threads)
        for (offset; offset < limit; offset++) {
            double sum = 0;
            //iterate over for each cell, all over corresponding sum of product
            for (int i = 0; i < n; i++) {
                sum += a[offset * n + i] * b[i];
            }
            c[offset] = sum;
        }
    }

}
```