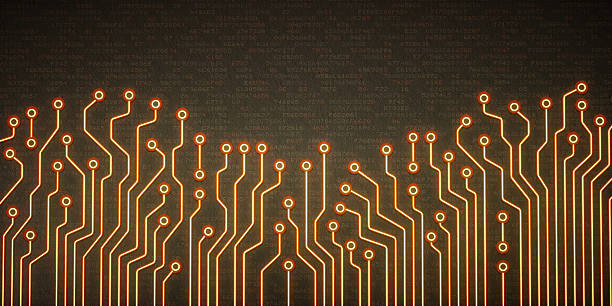


Final Project Report

Parallelizing Gaussian Elimination Algorithm



Kamal Dbouk 202203415

Nour Ghalayini 202201928

Yara Malaeb 202102116

# Overview of the Gaussian Elimination

## What is the Gaussian Elimination?

The Gaussian (Gaus) Elimination method is a widely used method in linear algebra to solve systems of linear equations and finds the solution. The system is represented as an augmented matrix where row reductions occur to transform the matrix into the row-echelon form, which is the upper-triangular form. Following this, the unknowns are solved by back-substitution from bottom to top. Starting with the last equation solving the last unknown, all the way up to the first equaition. The figure below shows a simple example of how this would work. (Math 1220, n.d.).

A math symbols on a white background

Description automatically generated A black background with a black square

Description automatically generated with medium confidence A number with black numbers

Description automatically generated with medium confidence

We get:

* X2=3
* X1 + X2 = 4, X1 = 4 - 3 = 1

# Serial Code

The following excerpt contains the serial code for the Gauss Algorithm. It takes for input the order of the matrix [N x (N+1)].

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

void freeMatrix(float \*\*A, int n) {

for(int i = 0; i < n; i++) {

free(A[i]);

}

free(A);

}

int main() {

int i, j, k, n;

float \*\*A, \*x, c, sum;

clock\_t start, end;

double cpu\_time\_used;

printf("\nEnter the order of the matrix: ");

scanf("%d", &n);

// Dynamically allocate memory for the augmented matrix A[n][n+1] and vector x[n]

A = (float\*\*)malloc(n \* sizeof(float\*));

for(i = 0; i < n; i++) {

A[i] = (float\*)malloc((n + 1) \* sizeof(float));

}

x = (float\*)malloc(n \* sizeof(float));

// Check if memory allocation was successful

if(A == NULL || x == NULL) {

printf("Memory allocation failed.\n");

return -1;

}

srand(time(NULL)); // Seed the random number generator

// Fill the augmented matrix with random floating-point numbers

for(i = 0; i < n; i++) {

for(j = 0; j <= n; j++) {

A[i][j] = ((float)rand() / RAND\_MAX) \* 99.9f + 0.1f; // Range: [0.1, 100.0]

}

}

start = clock(); // Start timing

// Gaussian elimination to form upper triangular matrix

for(j = 0; j < n; j++) {

for(i = j + 1; i < n; i++) {

if(A[j][j] == 0.0f) {

printf("Mathematical Error!");

freeMatrix(A, n);

free(x);

return -1;

}

c = A[i][j] / A[j][j];

for(k = j; k <= n; k++) {

A[i][k] -= c \* A[j][k];

}

}

}

// Backward substitution

for(i = n - 1; i >= 0; i--) {

sum = 0.0;

for(j = i + 1; j < n; j++) {

sum += A[i][j] \* x[j];

}

x[i] = (A[i][n] - sum) / A[i][i];

}

end = clock(); // End timing

cpu\_time\_used = ((double) (end - start)) / CLOCKS\_PER\_SEC;

printf("\nThe solution is:\n");

for(i = 0; i < n; i++) {

printf("x[%d] = %f\n", i, x[i]);

}

printf("Execution time: %f seconds\n", cpu\_time\_used);

// Free dynamically allocated memory

freeMatrix(A, n);

free(x);

return 0;

}

## Execution Time Calculation (Kali Linux)

|  |  |
| --- | --- |
| **Matrix Size** | **Execution Time (seconds)** |
| 10 | 0.000014 |
| 100 | 0.005507 |
| 1000 | 3.824536 |
| 1500 | 12.975288 |
| 2000 | 30.315610 |
| 3000 | 103.024908 |

# MPI

## How was MPI Parallelized?

Only the Gauss Elimination process is parallelized in this method. This is because the backwards elimination part of the code contains heavy data dependency as each element in the solution vector is dependenent on the preceeding element. The Gauss elimination part is split across porocesses where each process operates on a subset of rows. Thus, the outer loop of J (see code) is parallelized. Each column is processed sequentially to eliminate the variables below the diagonal. These rows are distrubuted among different processes so that each one handles a subset of rows. To better envision this, imagine 4 processors. Rows are divided as follows:

* + - P1: Handles Rows 0, 4, 8, …
    - P2: Handles Rows 1, 5, 9, …
    - P3: Handles Rows 2, 6, 10, …
    - P4: Handles Rows 3, 7, 11, …

Each processor works on its assigned row. After elimination is complete, each processor can communicate as needed. The process that owns the current pivot column, of rank (j%size), will broadcast the pivot row to all other processses. This means that if you need a row from a different subset of rows at a different processor, this is done at the line “MPI\_Bcast(A[j], n + 1, MPI\_FLOAT, j % size, MPI\_COMM\_WORLD);”. This will ensure that the elimination process is coherent across all processes. Each process will perform the elimination calculation for its subset of rows using the received pivot row. After this we return to the serial code.

1. Code

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <mpi.h>

// Function to free dynamically allocated memory for matrix A and vector x

void freeMatrix(float \*\*A, int n) {

    for (int i = 0; i < n; i++) {

        free(A[i]);

    }

    free(A);

}

int main() {

    int i, j, k, n;

    float \*\*A, \*x, c, sum;

    clock\_t start, end;

    double cpu\_time\_used;

    int rank, size;

    MPI\_Init(NULL, NULL);

    MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

    MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

    if (rank == 0) {

    printf("\nEnter the order of the matrix: ");

    fflush(stdout);

    scanf("%d", &n);

    }

    MPI\_Bcast(&n, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

    // Dynamically allocate memory for the augmented matrix A[n][n+1] and vector x[n]

    A = (float \*\*)malloc(n \* sizeof(float \*));

    for (i = 0; i < n; i++) {

        A[i] = (float \*)malloc((n + 1) \* sizeof(float));

    }

    x = (float \*)malloc(n \* sizeof(float));

    // Check if memory allocation was successful

    if (A == NULL || x == NULL) {

        printf("Memory allocation failed.\n");

        MPI\_Finalize();

        return -1;

    }

    srand(time(NULL)); // Seed the random number generator

    // Fill the augmented matrix with random floating-point numbers

    for (i = 0; i < n; i++) {

        for (j = 0; j <= n; j++) {

            A[i][j] = ((float)rand() / RAND\_MAX) \* 99.9f + 0.1f; // Range: [0.1, 100.0]

        }

    }

    start = clock();

    // Parallelized Gaussian elimination to form upper triangular matrix

    for (j = 0; j < n; j++) {

        if (rank == j % size) {

            for (i = j + 1; i < n; i++) {

                c = A[i][j] / A[j][j];

                for (k = j; k <= n; k++) {

                    A[i][k] -= c \* A[j][k];

                }

            }

        }

        MPI\_Bcast(A[j], n + 1, MPI\_FLOAT, j % size, MPI\_COMM\_WORLD); // Broadcast the pivot row

    }

    // Backward substitution

    for (i = n - 1; i >= 0; i--) {

        sum = 0.0;

        for (j = i + 1; j < n; j++) {

            sum += A[i][j] \* x[j];

        }

        x[i] = (A[i][n] - sum) / A[i][i];

    }

    /\*

    This step is not parallelized, as each element of the solution vector depends on previous computations.

    \*/

    if (rank == 0) {

        end = clock();

        cpu\_time\_used = ((double)(end - start)) / CLOCKS\_PER\_SEC;

        printf("\nThe solution is:\n");

        for (i = 0; i < n; i++) {

            printf("x[%d] = %f\n", i, x[i]);

        }

        printf("Execution time: %f seconds\n", cpu\_time\_used);

    }

    freeMatrix(A, n);

    free(x);

    MPI\_Finalize();

    return 0;

}

## Execution Time Calculation (Four Processors on Kali Linux)

|  |  |  |  |
| --- | --- | --- | --- |
| **Matrix Size** | **Serial** | **Parallelized (seconds) w/ num\_proc = 2** | **Parallelized (seconds) w/ num\_proc = 4** |
| 10 | 0.000005 | 0.000038 | 0.000019 |
| 100 | 0.001916 | 0.003091 | 0.00414 |
| 1000 | 0.262630 | 0.349180 | 0.178376 |
| 1500 | 2.257842 | 1.838382 | 1.299453 |
| 2000 | 8.275761 | 4.712199 | 3.826674 |
| 3000 | 25.67638 | 14.059772 | 9.888792 |

1. Other Calculations

S(p) = Sequential time / Parallel time  
Efficiency = S(p) / p \* 100%

|  |  |  |
| --- | --- | --- |
| **Matrix Size** | **Speed Up**  **w/ num\_proc = 2** | **Speed Up**  **w/ num\_proc = 4** |
| 10 | 0.1315 | 0.2631 |
| 100 | 0.6198 | 0.4628 |
| 1000 | 0.7521 | 1.4723 |
| 1500 | 1.2281 | 1.7375 |
| 2000 | 1.7562 | 2.1626 |
| 3000 | 1.8262 | 2.5965 |

|  |  |  |
| --- | --- | --- |
| **Matrix Size** | **Efficiency**  **w/ num\_proc = 2** | **Efficiency**  **w/ num\_proc = 4** |
| 10 | 6.5789% | 6.5789% |
| 100 | 30.9932% | 11.5711% |
| 1000 | 37.6066% | 36.8084% |
| 1500 | 61.4084% | 43.4383% |
| 2000 | 87.8121% | 54.0662% |
| 3000 | 91.3115% | 64.9128% |

1. Results Discussion

There exists a reasonable increase in speedup as the matrix size increases. Accordingly, the efficiency increases as well. However, we notice that at small inputs, such as matrix of order 10, the speedup is very small, indicating that the overhead such is process creation, communication and synchronization becomes significant.

# OpenMP

## How was OpenMP Parallelized?

**for** (**int** pivot = **0**; pivot < MATRIX\_SIZE; pivot++) {

**if** (pivot >= start\_row && pivot < end\_row) {

**for** (**int** j = pivot + **1**; j < MATRIX\_SIZE; j++) {

matrix[pivot][j] /= matrix[pivot][pivot];

}

solution[pivot] = vector[pivot] / matrix[pivot][pivot];

matrix[pivot][pivot] = **1.0**;

}

#pragma omp barrier

* For loop over ‘pivot’, iterates through each diagonal element of the matrix.
* Each pivot row is normalized so that the diagonal element becomes 1.
* #pragma omp barrier ensures that all threads reach this point before proceeding

#pragma omp **for** schedule(dynamic, rows\_per\_thread) nowait

**for** (**int** i = pivot + **1**; i < MATRIX\_SIZE; i++) {

**for** (**int** j = pivot + **1**; j < MATRIX\_SIZE; j++) {

matrix[i][j] -= matrix[i][pivot] \* matrix[pivot][j];

}

vector[i] -= matrix[i][pivot] \* solution[pivot];

matrix[i][pivot] = **0.0**;

}

* Outer for loop over ‘i’: iterates through rows below pivot row.
* Inner for loop over ‘j’: performs elimination to make elements below the pivot element zero.
* #pragma omp for schedule(dynamic, rows\_per\_thread) nowait: we use nowait since there is no need for threads to wait for synchronization at this point.
* We used dynamic scheduling after experimenting with static scheduling and as it turns out each thread will not have a fixed number of iterations so it’s better to have them dynamically distributed.

## Code

%%writefile openmp\_gaussian.c

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <omp.h>

#define MATRIX\_SIZE 10

#define MAX\_THREADS 1

**double** matrix[MATRIX\_SIZE][MATRIX\_SIZE], vector[MATRIX\_SIZE], solution[MATRIX\_SIZE];

**double** start\_time, end\_time;

**void** **initialize\_matrix\_and\_vector**();

**void** **perform\_gaussian\_elimination**();

**void** **print\_matrix\_and\_vector**();

**int** **main**() {

initialize\_matrix\_and\_vector();

start\_time = omp\_get\_wtime();

omp\_set\_num\_threads(MAX\_THREADS);

#pragma omp parallel

{

**int** thread\_id = omp\_get\_thread\_num();

**int** rows\_per\_thread = MATRIX\_SIZE / MAX\_THREADS;

**int** start\_row = thread\_id \* rows\_per\_thread;

**int** end\_row = start\_row + rows\_per\_thread;

**for** (**int** pivot = **0**; pivot < MATRIX\_SIZE; pivot++) {

**if** (pivot >= start\_row && pivot < end\_row) {

**for** (**int** j = pivot + **1**; j < MATRIX\_SIZE; j++) {

matrix[pivot][j] /= matrix[pivot][pivot];

}

solution[pivot] = vector[pivot] / matrix[pivot][pivot];

matrix[pivot][pivot] = **1.0**;

}

#pragma omp barrier

#pragma omp **for** schedule(dynamic, rows\_per\_thread) nowait

**for** (**int** i = pivot + **1**; i < MATRIX\_SIZE; i++) {

**for** (**int** j = pivot + **1**; j < MATRIX\_SIZE; j++) {

matrix[i][j] -= matrix[i][pivot] \* matrix[pivot][j];

}

vector[i] -= matrix[i][pivot] \* solution[pivot];

matrix[i][pivot] = **0.0**;

}

}

}

end\_time = omp\_get\_wtime();

print\_matrix\_and\_vector();

printf("Execution time: %f seconds**\n**", end\_time - start\_time);

**return** **0**;

}

**void** **initialize\_matrix\_and\_vector**() {

**for** (**int** i = **0**; i < MATRIX\_SIZE; i++) {

**for** (**int** j = **0**; j < MATRIX\_SIZE; j++) {

matrix[i][j] = (i == j) ? (**double**)**10.0** \* MATRIX\_SIZE : **1.0**;

}

vector[i] = **2.0**;

solution[i] = **0.0**;

}

}

**void** **print\_matrix\_and\_vector**() {

**for** (**int** i = **0**; i < MATRIX\_SIZE; i++) {

**for** (**int** j = **0**; j < MATRIX\_SIZE; j++) {

printf("%.3f ", matrix[i][j]);

}

printf(" | %.3f**\n**", vector[i]);

}

}

## Execution Time Calculation

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Matrix Size** | **Serial** | **OpenMP**  **w/ 2 threads** | **OpenMP**  **w/ 4 threads** | **OpenMP**  **w/ 6 threads** |
| 10 | 0.000014 | 0.461716 | 0.00232 | 0.01314 |
| 100 | 0.005507 | 0.02288 | 0.001677 | 0.011074 |
| 1000 | 3.824536 | 0.748051 | 0.491177 | 0.437510 |
| 1500 | 12.975288 | 2.449932 | 1.544266 | 1.344961 |
| 2000 | 30.315610 | 5.873106 | 3.588295 | 3.416958 |
| 3000 | 103.024908 | 19.235640 | 13.302757 | 11.164797 |

## Other Calculations for OpenMP

|  |  |  |  |
| --- | --- | --- | --- |
| Matrix Size | Speedup 2 threads | Speedup 4 threads | Speedup 6 threads |
| 10 | 0.00003 | 0.006 | 0.00107 |
| 100 | 0.241 | 3.286 | 0.497 |
| 1000 | 5.112 | 7.784 | 8.742 |
| 1500 | 5.296 | 8.400 | 9.644 |
| 2000 | 5.162 | 8.448 | 8.874 |
| 3000 | 5.355 | 7.746 | 9.228 |

|  |  |  |  |
| --- | --- | --- | --- |
| **Matrix Size** | **Efficiency 2 threads** | **Efficiency 4 threads** | **Efficiency 6 threads** |
| 10 | 0.0015% | 0.15% | 0.0178% |
| 100 | 12.05% | 82.15% | 8.28% |
| 1000 | 255.6% | 194.6% | 145.7% |
| 1500 | 264.8% | 210.0% | 160.7% |
| 2000 | 258.1% | 211.2% | 147.9% |
| 3000 | 267.8% | 193.7% | 153.8% |

## Results Discussions

For smaller size matrices, the overhead parallelization results in minimal, almost null, efficiency. This highlights how parallelization overhead can outweigh computational benefits when task size is too small. However, as matrix sizes increase, we start witnessing the real benefits of parallelization. Speedup and efficiency both increase with larger matrices (1000 x 1000) and beyond, showcasing the potential of parallelizing with OpenMP. When considering execution time, utilizing six threads leads to the shortest computation time, demonstrating enhanced performance with higher thread counts. To sum up to our observation, utilizing two threads often results in better resource management especially for large matrices, offering the best balance between speedup and resource efficiency.

# Cuda

## How was Cuda Parallelized?

We allocate memory for the ‘d\_a’ on the GPU. The data is then copied from the host to the device. We utilize the tiling approach, experimenting with tile sizes 16 and 32:

* We found that with tile size set at 32, time execution is slower than at 16. This can be attributed to thread synchronization overhead. Larger tiles require more threads to be launched, which is causing this overhead.
* More threads are required for each block, leading to overhead.

The kernel is called for each row. The kernel takes input the matrix, its order, and the row its responsible for. We calculate i and j, the thread indices of the grid and ensure that we are within the correct row using i and j. If we are, we calculate the element of the pivot row at a specific index. We ensure that the pivot value is not zero or aproximately zero. If it is not, we calculate the zero factor which is used to avoid numerical instability which was our main problem while working on this implementation. The zero factor that the elements below the pivot should be scaled to to make them zero. Then, the scaled row is subtracted from the current row to make the element below the pivot zero. Synchronization is ensured after each kernel. The result matrix is then copied back to the host and back substitution occurs.

## Code

%%writefile gaus.cu

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define TILE\_SIZE 16

\_\_global\_\_ void makeBelowPivotZero(double \*a, int n, int row) {

int i = blockIdx.x \* blockDim.x + threadIdx.x;

int j = blockIdx.y \* blockDim.y + threadIdx.y;

if (i > row && i < n && j < n + 1) {

double pivotValue = a[row \* (n + 1) + row];

if (fabs(pivotValue) > 1e-8) {

double zeroFactor = a[i \* (n + 1) + row] / pivotValue;

a[i \* (n + 1) + j] -= zeroFactor \* a[row \* (n + 1) + j];

}

}

}

void backSubstitution(double \*a, double \*x, int n) {

for (int i = n - 1; i >= 0; i--) {

x[i] = a[i \* (n + 1) + n];

for (int j = i + 1; j < n; j++) {

x[i] -= a[i \* (n + 1) + j] \* x[j];

}

x[i] /= a[i \* (n + 1) + i];

}

}

int main() {

int n;

printf("Enter the order of the matrix: ");

scanf("%d", &n);

double \*a = (double \*)malloc(n \* (n + 1) \* sizeof(double));

double \*x = (double \*)malloc(n \* sizeof(double));

// Initialize the matrix with random numbers

srand(time(NULL));

for (int i = 0; i < n; i++) {

for (int j = 0; j <= n; j++) {

a[i \* (n + 1) + j] = (double)(rand() % 100 + 1); // Random numbers 1-100

}

}

double \*d\_a;

cudaMalloc((void\*\*)&d\_a, (n + 1) \* n \* sizeof(double));

cudaMemcpy(d\_a, a, (n + 1) \* n \* sizeof(double), cudaMemcpyHostToDevice);

dim3 dimBlock(TILE\_SIZE, TILE\_SIZE);

dim3 dimGrid((n + TILE\_SIZE - 1) / TILE\_SIZE, (n + TILE\_SIZE - 1) / TILE\_SIZE);

clock\_t start = clock();

for (int i = 0; i < n; i++) {

makeBelowPivotZero<<<dimGrid, dimBlock>>>(d\_a, n, i);

cudaDeviceSynchronize();

}

clock\_t stop = clock();

cudaMemcpy(a, d\_a, (n + 1) \* n \* sizeof(double), cudaMemcpyDeviceToHost);

cudaFree(d\_a);

backSubstitution(a, x, n);

double time = ((double)(stop - start)) / CLOCKS\_PER\_SEC;

printf("Execution Time: %.6f seconds\n", time);

free(a);

free(x);

return 0;

}

## Execution Time Calculation

|  |  |  |
| --- | --- | --- |
| **Matrix Size** | **Serial** | **Parallelized (seconds)** |
| 10 | 0.000003 | 0.000313 |
| 100 | 0.001239 | 0.001522 |
| 1000 | 1.266512 | 0.136837 |
| 1500 | 5.141530 | 0.323688 |
| 2000 | 11.142518 | 0.622999 |
| 3000 | 37.195411 | 1.809242 |

## Other Calculations

S(p) = Sequential time / Parallel time  
Efficiency = S(p) / p \* 100%

Number of SMs = 40

|  |  |
| --- | --- |
| **Matrix Size** | **Speed Up** |
| 10 | 0.00958 |
| 100 | 0.81406 |
| 1000 | 9.25562 |
| 1500 | 15.8842 |
| 2000 | 17.8852 |
| 3000 | 20.5585 |

|  |  |
| --- | --- |
| **Matrix Size** | **Efficiency** |
| 10 | 0.023% |
| 100 | 2.035% |
| 1000 | 23.139% |
| 1500 | 39.711% |
| 2000 | 44.713% |
| 3000 | 51.396% |

Question to Answer:

Q: Why didn’t you use shared memory in Cuda C?

A: Gaussian elimination involves memory access and computations on the global memory. The memory in this case is constantly using and updating the data of the rows. Using a shared memory would have prompted us to synchronize at each step, eliminating the “parallel” aspect of the algorithm. The nature of this problem doesn’t need significant optimizations with shared memory.

# OpenACC

## Parallelization Process:

* The #pragma acc data directive was used to create a data environment where array A, holding the matrix data, is allocated to the GPU.
* #pragma acc parallel loop gang directive is applied to the outer loop across different gangs (groups of worker threads), where each gang handles a subset of rows from the matrix, allowing concurrent execution across multiple rows.
* Private(i, j, k) clause ensures that each gang has its own private copies of loop indices and variables, preventing data conflicts between gangs. Each gang operates on different pivot elements independently, reducing the matrix row to row echelon form.

## Code:

%%writefile openacc.c

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <openacc.h>

**void** freeMatrix(**float** \*A, **int** n) {

free(A);

}

**int** main() {

**int** i, j, k, n;

**float** \*A, \*x;

**clock\_t** start, end;

**double** cpu\_time\_used;

printf("**\n**Enter the order of the matrix: ");

scanf("%d", &n);

A = (**float**\*)malloc(n \* (n + **1**) \* **sizeof**(**float**));

x = (**float**\*)malloc(n \* **sizeof**(**float**));

**if**(A == NULL || x == NULL) {

printf("Memory allocation failed.**\n**");

**return** -**1**;

}

srand(time(NULL));

start = clock();

// Initialize matrix

#pragma acc data create(A[**0**:n\*(n+**1**)])

{

#pragma acc parallel loop gang copyin(n)

**for** (i = **0**; i < n; i++) {

**for** (j = **0**; j < n + **1**; j++) {

A[i \* (n + **1**) + j] = ((**float**)rand() / RAND\_MAX) \* **99.9** + **0.1**;

}

}

// Gaussian elimination

#pragma acc parallel loop gang private(i, j, k)

**for** (j = **0**; j < n; j++) {

**for** (i = j + **1**; i < n; i++) {

**float** c = A[i\*(n+**1**) + j] / A[j\*(n+**1**) + j];

**for** (k = j; k <= n; k++) {

A[i\*(n+**1**) + k] -= c \* A[j\*(n+**1**) + k];

}

}

}

}

// Sequential backward substitution

**for** (i = n - **1**; i >= **0**; i--) {

**float** sum = **0.0**;

**for** (j = i + **1**; j < n; j++) {

sum += A[i\*(n+**1**) + j] \* x[j];

}

x[i] = (A[i\*(n+**1**) + n] - sum) / A[i\*(n+**1**) + i];

}

end = clock();

cpu\_time\_used = ((**double**) (end - start)) / CLOCKS\_PER\_SEC;

printf("**\n**The solution is:**\n**");

**for** (i = **0**; i < n; i++) {

printf("x[%d] = %f**\n**", i, x[i]);

}

printf("Execution time: %f seconds**\n**", cpu\_time\_used);

freeMatrix(A, n);

free(x);

**return** **0**;

}

|  |  |  |
| --- | --- | --- |
| **Matrix Size** | **Serial** | **OpenACC (seconds)** |
| 10 | 0.000014 | 0.000329 |
| 100 | 0.005507 | 0.002106 |
| 1000 | 3.824536 | 2.099747 |
| 1500 | 12.975288 | 4.323688 |
| 2000 | 30.315610 | 10.622999 |
| 3000 | 103.024908 | 19.809242 |

## Other Calculations

|  |  |  |
| --- | --- | --- |
| Matrix Size | Speedup | Efficiency (over 2 cores) |
| 10 | 0.043 | 2.15% |
| 500 | 2.614 | 130.7% |
| 1000 | 1.421 | 91.05% |
| 1500 | 3.001 | 150.05% |
| 2000 | 2.854 | 142.7% |
| 3000 | 5.199 | 259.95% |

Limitations: Efficiency for OpenACC was calculated as divided by 2, which is the number of default processors in google colab. Determining the exact number of cores which involved dividing the work among gangs and workers is challenging due to hardware limitations.

## References

*Math 1220 linear algebra*. The Gaussian elimination algorithm. (n.d.). https://linearalgebra.math.umanitoba.ca/math1220/section-12.html

## Work distribution:

MPI – Kamal Dbouk

Cuda – Nour Ghalayini – Yara Malaeb – Kamal Dbouk

OpenMP & OpenACC Yara Malaeb