

Project Report: Applying F4 Polynomial Division Using GPU Computing

Project Team Members: Khoi Nguyen, Jeniffer Limondo, Jamen Tenney

Abstract

This project investigates the use of GPU (Graphics Processing Unit) computing to efficiently perform the F4 algorithm for polynomial division. The focus is on computing a row-echelon form using Gaussian elimination. The main objective of this research was to demonstrate how the GPU can handle the Gaussian elimination process, a central component of the F4 algorithm, when transforming matrices into their row-echelon form. We used the CUDA language for GPU computations, which has exceptional parallel processing abilities crucial for managing the large matrices encountered in F4 polynomial division. In contrast, CPU (Central Processing Unit) calculations were conducted using the Singular program (<https://www.singular.uni-kl.de/>), providing a comparative perspective. Our results indicate that GPUs are significantly more efficient for processing large matrices due to their superior parallel processing capabilities, while CPUs tend to be more effective for smaller matrices. This study highlights the potential of GPU computing in advanced algebraic operations and emphasizes the importance of selecting the right processing unit for specific computational tasks.

1. Introduction - Problem Statement

The main aim of this project is to compare the performance of a Graphics Processing Unit (GPU) and a Central Processing Unit (CPU) in executing Gaussian elimination on large matrices. Gaussian elimination is an important numerical method used in linear algebra to solve linear equations and determine matrix inverses. We want to determine whether the parallel processing capabilities of a GPU provide a significant advantage over the sequential processing abilities of a CPU when dealing with the intensive computational load of Gaussian elimination on large matrices. This investigation aims to determine whether using GPU acceleration for such tasks can benefit efficiency and guide the selection of appropriate hardware for large matrix computations.

2. Preliminary

Linear Algebra Concepts: In this project, we are using basic linear algebra techniques, precisely Gaussian elimination, which is a method that helps solve systems of linear equations. We transform matrices into row-echelon form, making finding solutions to these systems easier.

GPU Architecture: Knowing the architecture of Graphics Processing Units (GPUs) is essential. GPUs are built for high throughput and parallel processing, which makes them perfect for tasks that require large-scale computations. In contrast, Central Processing Units (CPUs) are designed for sequential processing.

CUDA Programming: CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model developed by NVIDIA. It allows software developers to utilize a CUDA-enabled GPU for general-purpose processing. This approach is GPGPU (General-Purpose computing on Graphics Processing Units).

The F4 algorithm is essential in computational algebra, particularly in polynomial division. It simplifies the process of Gaussian elimination when applied to polynomials, thereby improving computational efficiency.

Various metrics, such as computation time and efficiency, are considered to assess the performance of the GPU and CPU in this project. These metrics offer a measurable basis for comparing parallel processing (GPU) versus sequential processing (CPU) in terms of their effectiveness in managing matrix operations on a large scale.

These initial details lay the groundwork for understanding the project's methodology, the computational tools, and the performance evaluation criteria.

3. Contributions and Approach

3.1 Learning Process

In this project, we worked on exploring the effectiveness of GPU computing for the F4 algorithm used in polynomial division. We utilized CUDA, a parallel processing platform, to implement GPU computations, emphasizing its exceptional capabilities for managing large matrices in F4 polynomial division. We focused on matrix construction and research, contributing to a deeper theoretical understanding, ensuring the project was grounded in solid concepts. We also researched F4 polynomial division and matrix

construction. Our learning process involved mastering concepts such as Gaussian elimination, GPU architecture, and CUDA programming. We implemented various techniques, including matrix construction in both Singular and CUDA, and optimized code for efficient execution. The project's outcomes, presented in detailed GPU and CPU performance comparisons, highlighted the superior efficiency of GPUs for large matrices, reinforcing the significance of hardware selection for specific computational tasks. This project not only achieved its objectives but also provided valuable insights into parallel processing, numerical methods, and the interplay between GPU and CPU performance.

3.2 Implementation

3.2.1 Software Aspect

Our project involves four key programs designed to carry out specific tasks in our Gaussian elimination study. These programs are as follows:

- *gaussian_elimination.cu*: This CUDA-based program executes Gaussian elimination on GPUs and is optimized to fully utilize the parallel processing power of the GPU, making it ideal for large matrix operations.
- *generate_matrix_file.cpp*: This C++ program generates random matrices that can be used as test data. It ensures we have a wide range of matrices to test our Gaussian elimination processes on the GPU and CPU.
- *convert_data_to_Singular_data.cpp*: This converter program, also written in C++, transforms the data structure from our GPU-based Gaussian elimination into a format compatible with the Singular program, enabling CPU processing.
- *Singular_gaussian.c*: The CPU counterpart runs Gaussian elimination using the Singular program. It is essential to compare CPU performance against our GPU results.

Together, these components comprise a streamlined system that efficiently compares GPU and CPU performance in Gaussian elimination tasks.

**Note: We tested our code in the CADE lab to ensure consistency across all tests.*

3.2.1.1 GPU Programming Code

To begin with, instead of creating a matrix manually, it is better to generate it through a program as it saves time. For this purpose, we have created a C++ file named *generate_matrix_file.cpp*. This program generates a random matrix based on the given

parameters for the number of rows and columns. However, the numbers generated by this program fall within the range of $[-25, 25]$ only.

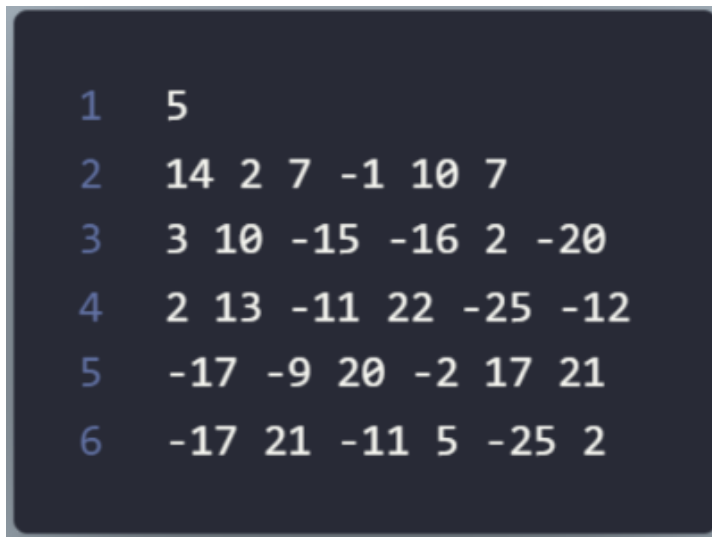
Before running the program to generate the matrix, we must define the size of the matrix by changing the desired size on line 10.

```
string filename = "data/data'n'.txt"; //change 'n' to the matrix size
```

For example, if you want to generate the matrix with the size 5 x 6, then you will change it to:

```
string filename = "data/data5.txt";
```

A text file named *data5.txt* will be generated in the data folder with the following structure:



```
1 5
2 14 2 7 -1 10 7
3 3 10 -15 -16 2 -20
4 2 13 -11 22 -25 -12
5 -17 -9 20 -2 17 21
6 -17 21 -11 5 -25 2
```

The "top of the line" refers to a matrix with a size of 5 x 6. Each subsequent line in the file is a matrix equation.

We can use the data file generated to perform Gaussian-Elimination using the file *gaussian_elimination.cu*. Before running the file, we need to utilize the following code snippet, written in CUDA, to perform forward elimination and back sub elimination:

Forward Elimination Kernel

This kernel is designed to accelerate the forward elimination stage of Gaussian elimination by utilizing a GPU. The matrices are divided into different sections, which are then processed by individual threads in parallel. This technique is frequently used to solve linear equations or invert matrix systems. It is essential to keep in mind that this method assumes that the matrix is square and augmented to solve a system of linear equations.

```
1 // CUDA kernel for forward elimination
2 __global__ void forwardElimKernel(double *mat, int numvar, int k) {
3     // Get the row index
4     int idx = blockDim.x * blockIdx.x + threadIdx.x + k + 1;
5
6     // Perform forward elimination
7     if (idx < numvar) {
8
9         // Check if the pivot element is 0
10        double factor = mat[idx * (numvar + 1) + k] / mat[k * (numvar + 1) + k];
11
12        // Update the matrix
13        for (int j = k; j <= numvar; j++) {
14            // Update the matrix element
15            mat[idx * (numvar + 1) + j] -= factor * mat[k * (numvar + 1) + j];
16        }
17    }
18 }
```

Back substitution Kernel

The following code snippet is a CUDA kernel function that performs back substitution. This is the second phase of solving a system of linear equations after forward elimination, which is usually performed after Gaussian elimination.

```
1 // CUDA kernel for back substitution
2 __global__ void backSubKernel(double *mat, double *x, int numvar) {
3     int idx = blockDim.x * blockIdx.x + threadIdx.x;
4
5     // Perform back substitution
6     if (idx < numvar) {
7         x[idx] = mat[idx * (numvar + 1) + numvar];
8
9         // Synchronize before starting the back substitution
10        __syncthreads();
11
12        // Back substitution
13        for (int i = numvar - 1; i >= 0; i--) {
14
15            // Synchronize before updating the value
16            __syncthreads();
17
18            // Update the value
19            if (idx <= i) {
20
21                // Check if the pivot element is 0
22                if (idx == i) {
23                    x[idx] /= mat[i * (numvar + 1) + i];
24                }
25
26                // Synchronize before updating the value
27                __syncthreads();
28
29                if (idx < i) {
30                    x[idx] -= mat[idx * (numvar + 1) + i] * x[i];
31                }
32            }
33        }
34    }
35 }
```

It's interesting to observe the differences between programming in CUDA compared to other high-level languages like C or C++.

Prior to running the gaussian_elimination.cu program, make sure to set the path to the data text file being generated to line 16.

```
1  #include <cmath>
2  #include <cuda_runtime.h>
3  #include <iomanip>
4  #include <fstream>
5  #include <chrono>
6  #include <string>
7  #include <vector>
8  #include <unistd.h>
9
10 using namespace std;
11
12 // define shouldPrint to print the matrix at each step
13 bool shouldPrint = false;
14
15 // define data_file to read the matrix from a file
16 const char *data_file = "data/data30.txt";
17
18 // Define a small threshold value
19 const double EPSILON = 1e-10;
```

We have incorporated a boolean function called *"shouldPrint"* in our project to manage the output of step-by-step matrix processing. Enabling *"shouldPrint"* to *'true'* allows the program to print every step of the matrix operations, providing detailed tracking and analysis. However, since printing these steps can increase the total runtime, one can turn off this feature by setting *"shouldPrint"* to *'false'*. This flexibility helps to balance the in-depth process visibility and optimal performance measurement.

```
1  #include <cmath>
2  #include <cuda_runtime.h>
3  #include <iomanip>
4  #include <fstream>
5  #include <chrono>
6  #include <string>
7  #include <vector>
8  #include <unistd.h>
9
10 using namespace std;
11
12 // define shouldPrint to print the matrix at each step
13 bool shouldPrint = false;
14
15 // define data_file to read the matrix from a file
16 const char *data_file = "data/data30.txt";
17
18 // Define a small threshold value
19 const double EPSILON = 1e-10;
```

To run the file, you must ensure that the CUDA Toolkit is installed. You can find installation instructions for Windows and Linux

1. <https://docs.nvidia.com/cuda/cuda-installation-guide-microsoft-windows/index.html>
2. <https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html>

Once installed, you can execute the program using the command:


```
nvcc -o gaussian_elimination gaussian_elimination.cu
./gaussian_elimination
```

Note that if you want to have output as a fraction, you can uncomment the lines 248-252 and comment the line 244-245

```
244     for (int i = 0; i < numvar; i++)
245     {
246         std::cout << "X" << i + 1 << " = " << x[i] << std::endl;
247
248         // Uncomment the following code to print the solution as fractions
249         // for (int i = 0; i < numvar; i++) {
250         //     int numerator, denominator;
251         //     decimalToFrac(x[i], numerator, denominator);
252         //     cout << "X" << i << " = " << numerator << "/" << denominator << endl;
253         // }
```

Here is the outcome of the program with shouldPrint = true:

```
[u1140015@lab1-17 gaussian]$ nvcc -o gaussian_elimination gaussian_elimination.cu
[u1140015@lab1-17 gaussian]$ ./gaussian_elimination
Input Matrix:
    1/3    1/2    0    0
    2     0     1     0
    0     4     0    -1

Matrix after step 0:
    1/3    1/2    0    0
    0    -3     1     0
    0     4     0    -1

Matrix after step 1:
    1/3    1/2    0    0
    0    -3     1     0
    0     0    4/3    -1

Matrix after step 2:
    1/3    1/2    0    0
    0    -3     1     0
    0     0    4/3    -1

Solution for the system:
X1 = 0.375
X2 = -0.25
X3 = -0.75

Time taken for doing Gaussian-Elimination: 58 milliseconds
```

And here is the output with the fraction result:

```
[u1140015@lab1-17 gaussian]$ nvcc -o gaussian_elimination gaussian_elimination.cu
[u1140015@lab1-17 gaussian]$ ./gaussian_elimination
Input Matrix:
    1/3    1/2    0    0
    2     0    1    0
    0     4    0   -1

Matrix after step 0:
    1/3    1/2    0    0
    0    -3    1    0
    0     4    0   -1

Matrix after step 1:
    1/3    1/2    0    0
    0    -3    1    0
    0     0   4/3   -1

Matrix after step 2:
    1/3    1/2    0    0
    0    -3    1    0
    0     0   4/3   -1

Solution for the system:
X0 = 3/8
X1 = -1/4
X2 = -3/4

Time taken for doing Gaussian-Elimination: 60 milliseconds
```

3.2.1.2 Singular Programming Code

To run the Gaussian elimination by using Singular, you need to make sure to have Singular available on your machine. You can download Singular and also instructions on how to install it via this link:

<https://www.singular.uni-kl.de/index.php/singular-download.html>

Once you have installed Singular, it is important to set the matrix size and copy the matrix numbers before running the program. For this project, we will be using the same data we used in *gaussian_elimination.cu*. However, because of the limitations of Singular programming, it is difficult for us to implement a Singular program that can load and import data directly. Therefore, we have built a C++ file named "*convert_data_to_Singular_data.cpp*" that will take our data as input and convert it into a matrix format that can be easily copied and pasted into Singular.

If you already have a data text file already, you only need to change the name of the data text file you want to convert as the following picture:

```

1  #include <iostream>
2  #include <fstream>
3  #include <sstream>
4  #include <string>
5
6  using namespace std;
7
8  string data_file_name = "data/data3.txt";
9
10 string constructOutputFileName(const string& inputFilePath) {

```

After that, you can run the file by this command:

```
g++ -o convert_data_to_Singular_data convert_data_to_Singular_data.cpp
```

It will generate a file *Singular_data'n'.txt* in the data folder.

In this example, It will convert *gaussian_elimination.cu* data style to this data-style so you can copy and paste it into a matrix on Singular.



For example, suppose you have a matrix size 3 x 4. In that case, you need to change 'n = 3' on line 6 and also copy text from generated *Singular_data3.txt* to the line below the 'matrix A[n][n+1] =' as highlighted:

```
1 LIB "matrix.lib";
2 system("--ticks-per-sec",1000); // milliseconds
3 ring S=0,x,lp;
4
5 // size of the matrix
6 int n = 30;
7
8 matrix A[n][n+1] =
```

After that, make sure you are in the same folder of the Singular file and run this command to run the program:

Singular Singular_gaussian.c.

Here is the outcome of the program:

```

[u1140015@lab1-17 gaussian]$ Singular Singular_gaussian.c
SINGULAR /
A Computer Algebra System for Polynomial Computations / version 3-1-1
0<
by: G.-M. Greuel, G. Pfister, H. Schoenemann \ Feb 2010
FB Mathematik der Universitaet, D-67653 Kaiserslautern \
// ** loaded /usr/local/stow/singular/singular-3-1-1/Singular/3-1-1/LIB/matrix.lib (12898,2010-06-23)
// ** loaded /usr/local/stow/singular/singular-3-1-1/Singular/3-1-1/LIB/nctools.lib (12790,2010-05-14)
// ** loaded /usr/local/stow/singular/singular-3-1-1/Singular/3-1-1/LIB/poly.lib (12443,2010-01-19)
// ** loaded /usr/local/stow/singular/singular-3-1-1/Singular/3-1-1/LIB/general.lib (12904,2010-06-24)
// ** loaded /usr/local/stow/singular/singular-3-1-1/Singular/3-1-1/LIB/random.lib (12827,2010-05-28)
// ** loaded /usr/local/stow/singular/singular-3-1-1/Singular/3-1-1/LIB/ring.lib (12231,2009-11-02)
// ** loaded /usr/local/stow/singular/singular-3-1-1/Singular/3-1-1/LIB/primdec.lib (12962,2010-07-09)
// ** loaded /usr/local/stow/singular/singular-3-1-1/Singular/3-1-1/LIB/absfact.lib (12231,2009-11-02)
// ** loaded /usr/local/stow/singular/singular-3-1-1/Singular/3-1-1/LIB/triang.lib (12231,2009-11-02)
// ** loaded /usr/local/stow/singular/singular-3-1-1/Singular/3-1-1/LIB/elim.lib (12231,2009-11-02)
// ** loaded /usr/local/stow/singular/singular-3-1-1/Singular/3-1-1/LIB/inout.lib (12541,2010-02-09)
print(A);
1/3,1/2,0,0,
2, 0, 1,0,
0, 4, 0,-1

1/3, 1/2, 0, 0,
2, 0, 1, 0,
0, 4, 0, -1
0, 0, 0, 0,
0, -3, 1, 0,
0, 4, 0, -1
0, 0, 0, 0,
0, 0, 0, 0,
0, 0, 4/3, -1
0, 0, 0, 0,
0, 0, 0, 0,
0, 0, 0, 0
Solution for the system is:
X[1] = 3/8
X[2] = -1/4
X[3] = -3/4

Total time for rowred function in Singular to compile: 0 microseconds

```

3.2.1.3 Verification of the Result

We utilized a robust verification strategy to ensure the accuracy and reliability of the results obtained from our implementations in our CUDA and Singular code. This involved cross-referencing our solutions with those generated by a well-established online tool. We used the Gaussian elimination calculator available at OnlineMSchool (<https://onlinemschool.com/math/assistance/equation/gaus/>) to achieve this. This online resource is renowned for its precision in solving linear equations through the Gaussian elimination method.

Validating our computational algorithms was a crucial step in this process. By comparing the outcomes from our CUDA and Singular programs against the solutions

provided by the online resource, we could confirm the correctness of our implementations. This verification process helped bolster the credibility of our results and also provided an external benchmark for evaluating the effectiveness of our computational approaches.

3.2.2 Result

In this section, we present the results of our experiments. The table and graphs below compare the GPU and CPU run times.

TABLE 1: The difference between GPU and CPU run times

Matrix	Compute Time (in milliseconds [ms])	
	GPU	CPU
3 x 4	88	0
16 x 17	90	240
27 x 28	95	1580
30 x 31	92	2320
60 x 61	90	48870
90 x 91	90	363940
120 x 121	90	1492430
150 x 151	96	4509850

Table 1 compares the computation times of GPU and CPU for various matrix sizes in milliseconds (ms). The comparison revealed that GPU compute times are significantly faster than CPU compute times for these calculations. This implies that GPUs have better performance for such computations.

For the smallest matrix (3 x 4), the CPU time was 0 ms, suggesting that the computation time was too short to measure. As the size of the matrices increased, the GPU times increased slowly, while the CPU times grew exponentially. For example, the GPU compute time for a 150 x 151 matrix was around 96 ms, whereas the CPU time was approximately 4509850 ms.

To obtain the result for the 150 x 151 matrix, it took us 4509850 ms, equivalent to 1.25 hours. However, using CUDA, we were able to obtain the result in only 96 ms. We then tested CUDA with larger matrices to see how it performs beyond 150 x 151. With a matrix size of 500 x 501, it only took 162 ms. We could not imagine how long it would have taken to use Singular. We continued testing with larger sizes, as shown in the graph below.

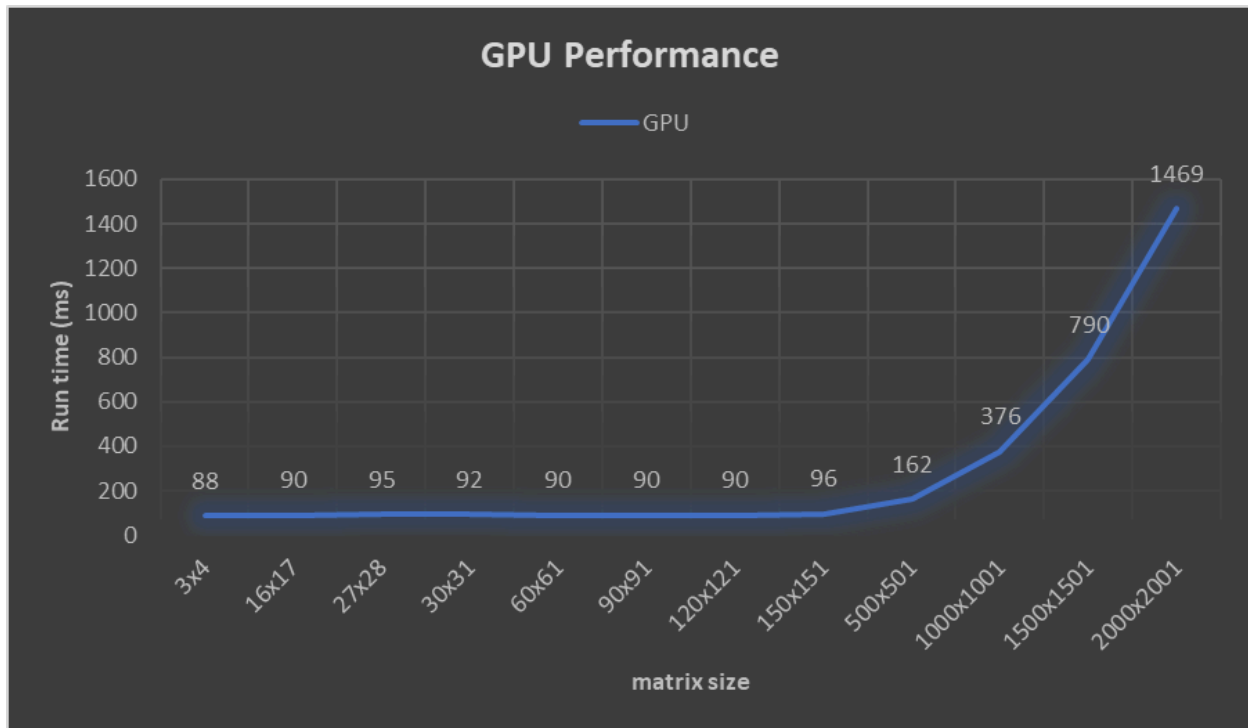


Figure 1. GPU Performance

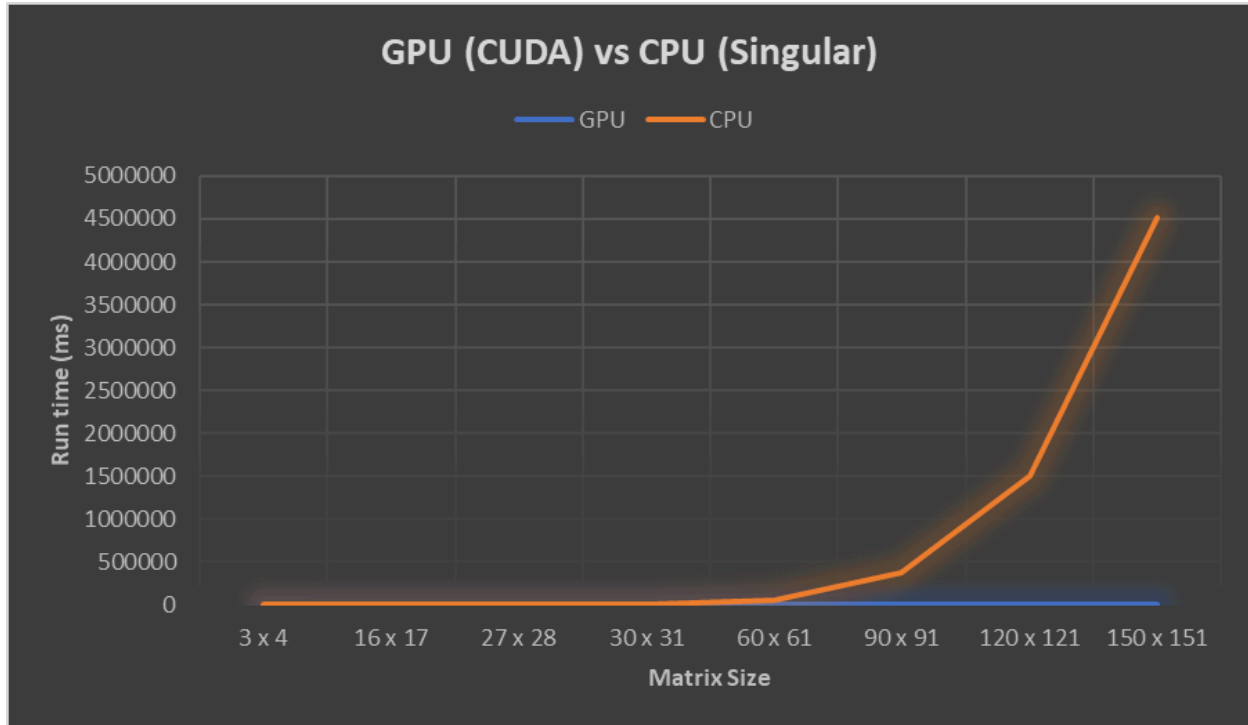


Figure 2. GPU vs. CPU

In Figure 2, we can see two lines representing the performance of the GPU and CPU when it comes to matrix operations. The blue line shows the GPU's performance, which is relatively stable regardless of the matrix size. GPUs are designed for parallel computations, so this flat line is expected. On the other hand, the orange line represents the CPU's performance. It starts similar to the GPU's performance for smaller matrix sizes, but as the matrix size increases, it rises more sharply. This indicates that the CPU takes significantly longer to perform calculations on larger matrices because it has limited capability for parallel processing.

4. Conceptual Learnings

Our project focused on implementing the F4 algorithm for polynomial division using GPU computing. This provided us with a valuable learning experience that combined theoretical knowledge with practical application. Our project was divided into three main areas:

1. **Coding in Singular and CUDA:** We developed proficiency in both Singular and CUDA programming languages. This helped us improve our coding versatility and adaptability.

2. Understanding the F4 Algorithm: This was central to our study. We explored the F4 algorithm's role in polynomial division and gained insights into its efficiency in simplifying Gaussian elimination for polynomials.

3. Gaussian Elimination and Matrix Operations: We delved into Gaussian elimination, which is a key method in linear algebra. We learned to apply it effectively in large-scale GPU computations.

We also explored GPU architecture and its advantages in parallel processing. We applied optimization techniques in CUDA and assessed performance metrics. This comprehensive exploration enhanced our understanding of high-performance computing and numerical methods. It equipped us with valuable skills for future endeavors in computational algebra.

5. Division of Labour

In our project, we divided the tasks among team members Khoi Nguyen, Jeniffer Limondo, and Jamen Tenney, who each brought their skills and expertise to different aspects of the work.

Team Member - Khoi Nguyen focused mainly on developing the computational core code, which involved writing the CUDA code for GPU-based Gaussian elimination and the accompanying C/C++ code for matrix generation and data conversion. Khoi's contribution was pivotal in building the technical backbone of the project.

Team Member - Jeniffer Limondo was responsible for matrix construction and research. They diligently gathered information online about Gaussian elimination, facilitating a deeper understanding of the algorithm. Additionally, Jeniffer communicates proactively, engaging with the professor and other teammates to clarify concepts and resolve queries. Their role was instrumental in ensuring the project was grounded in solid theoretical understanding and practical know-how.

Team Member - Jamen Tenney played a key role in researching F4 polynomial division, an essential theoretical component of our project. He also took part in manually constructing matrices and contributed significantly to writing a large portion of the project report. Jamen's efforts were crucial in integrating theoretical insights with practical applications in our report.

Each team member's unique contributions were integral to the project's success, combining technical programming skills, research and theoretical understanding, and effective communication and documentation.

6. Conclusion

In summary, our project demonstrated that GPUs, using CUDA for Gaussian elimination in F4 polynomial division, outperform CPUs in handling large matrices. This highlights GPUs' superior parallel processing capabilities and underscores the importance of hardware selection for complex computations. We achieved our goal of comparing GPU and CPU performance, enhancing our understanding of parallel processing and numerical methods.

Future directions include optimizing GPU implementations and applying our findings to practical fields like scientific computing, machine learning, and cryptography. This work lays a foundation for further advancements in parallel processing and computational efficiency.

Appendix A: CUDA

```
//-----
// Copyright 2023 University of Utah, All rights reserved
//-----
// File Name: gaussian_elimination.cu
// Author: Khoi Nguyen, Jeniffer Limondo & Jamen Tenney
// Create Date: 12/18/2023
// Purpose:
// Modified (or history): See Below
//-----
////////////////////////////////////
//
// Description:
// Course Name: ECE 5745/6755
//
// Revision:
// Revision 23182023 - kn/jml/jt - file created
// Additional Comments:
//
////////////////////////////////////

#include <iostream>
#include <cmath>
#include <cuda_runtime.h>
#include <iomanip>
#include <fstream>
#include <chrono>
#include <string>
#include <vector>
#include <unistd.h>

using namespace std;

// define shouldPrint to print the matrix at each step
bool shouldPrint = false;

/* This data3.txt matrix came from the paper, "EFFICIENT GROBNER BASIS REDUCTIONS FOR FORMAL
VERIFICATION OF GALOIS FIELD ARITHMETIC CIRCUITS in Section VII. IMPROVING POLYNOMIAL DIVISION
USING F4-STYLE REDUCTION" that was provided to us*/
// define data_file to read the matrix from a file
const char *data_file = "data/data3.txt";

// Define a small threshold value
const double EPSILON = 1e-10;

// CUDA kernel for forward elimination
__global__ void forwardElimKernel(double *mat, int numvar, int k) {
    int idx = blockDim.x * blockIdx.x + threadIdx.x + k + 1;
    if (idx < numvar) {
        double factor = mat[idx * (numvar + 1) + k] / mat[k * (numvar + 1) + k];
        for (int j = k; j <= numvar; j++) {
            mat[idx * (numvar + 1) + j] -= factor * mat[k * (numvar + 1) + j];
        }
    }
}

// CUDA kernel for back substitution
__global__ void backSubKernel(double *mat, double *x, int numvar) {
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
```

```

    if (idx < numvar) {
        x[idx] = mat[idx * (numvar + 1) + numvar];
        __syncthreads(); // Synchronize before starting the back substitution

        for (int i = numvar - 1; i >= 0; i--) {
            __syncthreads();
            if (idx <= i) {
                if (idx == i) {
                    x[idx] /= mat[i * (numvar + 1) + i];
                }
                __syncthreads(); // Synchronize after updating the value

                if (idx < i) {
                    x[idx] -= mat[idx * (numvar + 1) + i] * x[i];
                }
            }
        }
    }
}

```

```

// Function to convert a string fraction to decimal
double fractionToDecimal(const string& frac) {
    istringstream iss(frac);
    double num, denom = 1;
    char slash;
    iss >> num >> slash >> denom;
    if (denom != 0) return num / denom;
    return num; // Handle non-fraction case
}

```

```

// Function to print the matrix with fractions
void printFractionMatrix(const vector<string>& mat, int numvar) {
    for (int i = 0; i < numvar; i++) {
        for (int j = 0; j <= numvar; j++) {
            cout << setw(10) << mat[i * (numvar + 1) + j] << " ";
        }
        cout << endl;
    }
    cout << endl;
}

```

```

double parseFraction(const string& frac) {
    size_t slashPos = frac.find('/');
    if (slashPos != string::npos) {
        double numerator = stod(frac.substr(0, slashPos));
        double denominator = stod(frac.substr(slashPos + 1));
        if (denominator != 0) return numerator / denominator;
    }
    return stod(frac); // Handle non-fraction case
}

```

```

// Function to find the greatest common divisor (GCD)
int gcd(int a, int b) {
    if (b == 0) {
        return a;
    }
    return gcd(b, a % b);
}

```

```

// Function to convert a decimal to a fraction

```

```

void decimalToFrac(double value, int &numerator, int &denominator) {
    // Check if the value is close to an integer
    double diff = fabs(value - round(value));
    if (diff < EPSILON) {
        numerator = static_cast<int>(round(value));
        denominator = 1;
        return; // Early return as the number is effectively an integer
    }

    const double precision = 1E-6; // Precision for the conversion
    double integral = floor(value);
    double frac = value - integral;
    const int max_denominator = 10000; // Limits the denominator size

    // Initialize denominator as 1
    int lower_n = 0;
    int lower_d = 1;
    int upper_n = 1;
    int upper_d = 1;

    while (lower_d <= max_denominator && upper_d <= max_denominator) {
        int middle_n = lower_n + upper_n;
        int middle_d = lower_d + upper_d;

        if (fabs(frac - (double)middle_n / middle_d) < precision) {
            if (middle_d > max_denominator) {
                if (lower_d > upper_d) {
                    lower_n = upper_n;
                    lower_d = upper_d;
                }
                break;
            }

            lower_n = upper_n = middle_n;
            lower_d = upper_d = middle_d;
        } else if (frac > (double)middle_n / middle_d) {
            lower_n = middle_n;
            lower_d = middle_d;
        } else {
            upper_n = middle_n;
            upper_d = middle_d;
        }
    }

    // Adjust fraction to combine with the integral part
    numerator = (int)integral * lower_d + lower_n;
    denominator = lower_d;

    // Reduce the fraction
    int commonDivisor = gcd(abs(numerator), denominator);
    numerator /= commonDivisor;
    denominator /= commonDivisor;
}

// Function to print the matrix
void printMatrix(double *mat, int numvar) {
    for (int i = 0; i < numvar; i++) {
        for (int j = 0; j <= numvar; j++) {
            // Convert each element to fraction
            int numerator, denominator;
            decimalToFrac(mat[i * (numvar + 1) + j], numerator, denominator);

```

```

        // Check if the value is smaller than EPSILON in absolute terms
        if (fabs(mat[i * (numvar + 1) + j]) < EPSILON) {
            cout << setw(10) << 0 << " ";
        } else {
            // Display as a fraction
            if (denominator == 1) { // Print as an integer if the denominator is 1
                cout << setw(10) << numerator << " ";
            } else { // Otherwise, print as a fraction
                cout << setw(10) << numerator << "/" << denominator << " ";
            }
        }
    }
    cout << endl;
}
cout << endl;
}

// Forward elimination function with an option to print matrix at each step
void forwardElim(double *mat, int numvar, bool printSteps) {
    double *d_mat;
    size_t size = numvar * (numvar + 1) * sizeof(double);
    cudaMalloc(&d_mat, size);
    cudaMemcpy(d_mat, mat, size, cudaMemcpyHostToDevice);

    dim3 block(256);
    dim3 grid((numvar + block.x - 1) / block.x);

    for (int k = 0; k < numvar; k++) {
        forwardElimKernel<<<grid, block>>>(d_mat, numvar, k);
        cudaDeviceSynchronize();

        if (printSteps) {
            // Copy back the matrix to the host to print
            cudaMemcpy(mat, d_mat, size, cudaMemcpyDeviceToHost);
            cout << "Matrix after step " << k << ":" << endl;
            printMatrix(mat, numvar);
        }
    }

    if (!printSteps) {
        cudaMemcpy(mat, d_mat, size, cudaMemcpyDeviceToHost);
    }
    cudaFree(d_mat);
}

//
void backSub(double *mat, int numvar) {
    double *dev_mat, *dev_x;
    cudaMalloc((void **)&dev_mat, numvar * (numvar + 1) * sizeof(double));
    cudaMalloc((void **)&dev_x, numvar * sizeof(double));

    cudaMemcpy(dev_mat, mat, numvar * (numvar + 1) * sizeof(double), cudaMemcpyHostToDevice);

    int blockSize = 256; // or another suitable block size
    int numBlocks = (numvar + blockSize - 1) / blockSize;
    backSubKernel<<<numBlocks, blockSize>>>(dev_mat, dev_x, numvar);

    double *x = new double[numvar];
    cudaMemcpy(x, dev_x, numvar * sizeof(double), cudaMemcpyDeviceToHost);
}

```

```

std::cout << "\nSolution for the system:\n";
for (int i = 0; i < numvar; i++)
    std::cout << "X" << i + 1 << " = " << x[i] << std::endl;

// Uncomment the following code to print the solution as fractions
// for (int i = 0; i < numvar; i++) {
//     int numerator, denominator;
//     decimalToFrac(x[i], numerator, denominator);
//     cout << "X" << i << " = " << numerator << "/" << denominator << endl;
// }

delete[] x;
cudaFree(dev_mat);
cudaFree(dev_x);
}

// Main function
int main() {
    ifstream file(data_file);
    if (!file.is_open()) {
        cerr << "Error opening file" << endl;
        return -1;
    }

    int numvar;
    file >> numvar;

    vector<string> matStr(numvar * (numvar + 1));
    string frac; // Temporary string to store fraction input

    // Read the matrix data as fractions or decimal numbers
    for (int i = 0; i < numvar; i++) {
        for (int j = 0; j <= numvar; j++) {
            file >> frac;
            matStr[i * (numvar + 1) + j] = frac;
        }
    }

    file.close();

    // Print the input matrix as fractions or decimal numbers
    cout << "Input Matrix:\n";
    printFractionMatrix(matStr, numvar);

    // Convert the string representations to decimal values for calculations
    double *mat = new double[numvar * (numvar + 1)];
    for (int i = 0; i < numvar * (numvar + 1); ++i) {
        mat[i] = fractionToDecimal(matStr[i]);
    }

    // Start the timer
    auto start = chrono::high_resolution_clock::now();

    // Perform Gaussian elimination
    forwardElim(mat, numvar, shouldPrint);

    // Perform back substitution and print the results
    backSub(mat, numvar);

    // Stop the timer
    auto end = chrono::high_resolution_clock::now();

```

```
// Calculate the duration
auto duration = chrono::duration_cast<chrono::milliseconds>(end - start);

cout << "\nTime taken for doing Gaussian-Elimination: " << duration.count() << "
milliseconds" << endl;

// Free the dynamically allocated memory
delete[] mat;

return 0;
}
```


Appendix B: CUDA

```
//-----
// Copyright 2023 University of Utah, All rights reserved
//-----
// File Name: generate_matrix_file.cpp
// Author:  Jeniffer Limondo, Khoi Nguyen & Jamen Tenney
// Create Date: 12/18/2023
// Purpose:
// Modified (or history):  See Below
//-----
////////////////////////////////////
//
// Description:
// Course Name: ECE 5745/6755
//
// Revision:
// Revision 23182023 - jml/kn/jt - file created
// Additional Comments:
//
////////////////////////////////////

#include <iostream>
#include <fstream>
#include <cstdlib> // For rand() and srand()
#include <ctime>   // For time()
#include <string>
#include <regex>

using namespace std;

string filename = "data/data3.txt"; // change data3.txt to any matrix size file

int extractNumberFromFilename(const string& filename) {
    regex pattern(R"(data(\d+)\.txt)");
    smatch matches;

    if (regex_search(filename, matches, pattern) && matches.size() > 1) {
        return stoi(matches.str(1));
    }
    return -1; // Return -1 if no number is found
}

int main() {

    int n = extractNumberFromFilename(filename);

    if (n == -1) {
        cerr << "Error: Number not found in filename." << endl;
        return 1;
    }

    ofstream file(filename);

    if (!file) {
        cerr << "Error opening file for writing." << endl;
        return 1;
    }

    srand(time(NULL)); // Initialize random seed
```

```
int m = n + 1;

file << n << endl;

for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        int random_number = rand() % 51 - 25; // Generate a random number between -25 and
25        file << random_number;
        if (j < m - 1) {
            file << " "; // Space between numbers, not after the last number
        }
    }
    file << endl;
}

file.close();

return 0;
}
```

Appendix C: Singular

```
//-----
// Copyright 2023 University of Utah, All rights reserved
//-----
// File Name: convert_data_to_Singular_data.cpp
// Author: Jeniffer Limondo, Khoi Nguyen & Jamen Tenney
// Create Date: 12/18/2023
// Purpose:
// Modified (or history): See Below
//-----
////////////////////////////////////
//
// Description:
// Course Name: ECE 5745/6755
//
// Revision:
// Revision 23182023 - jml/kn/jt - file created
// Additional Comments:
//
////////////////////////////////////

#include <iostream>
#include <fstream>
#include <sstream>
#include <string>

using namespace std;

string data_file_name = "data/data3.txt"; // change data3.txt to any matrix size file

string constructOutputFileName(const string& inputFilePath) {
    // Extract the file name from the input file path
    size_t pos = inputFilePath.find_last_of("/\\");
    string fileName = (pos != string::npos) ? inputFilePath.substr(pos + 1) : inputFilePath;

    // Prefix the file name with "Singular_" and return the new path
    return inputFilePath.substr(0, pos + 1) + "Singular_" + fileName;
}

int main() {
    string inputFilePath = data_file_name; // Path to the input file
    string outputFilePath = constructOutputFileName(inputFilePath);

    ifstream inputFile(inputFilePath);
    ofstream outputFile(outputFilePath);

    if (!inputFile.is_open() || !outputFile.is_open()) {
        cerr << "Unable to open file(s)";
        return 1;
    }

    string line;
    bool isFirstLine = true;

    // Skip the first line
    getline(inputFile, line);

    while (getline(inputFile, line)) {
        if (!isFirstLine) {
            outputFile << ", " << endl;
        }
    }
}
```

```
    }

    istream iss(line);
    int number;
    bool isFirstNumber = true;

    while (iss >> number) {
        if (!isFirstNumber) {
            outputFile << ", ";
        }
        outputFile << number;
        isFirstNumber = false;
    }

    isFirstLine = false;
}

outputFile << ";";

inputFile.close();
outputFile.close();

return 0;
}
```

Appendix D: Singular

```
//-----
// Copyright 2023 University of Utah, All rights reserved
//-----
// File Name: Singular_gaussian.c
// Author:  Jeniffer Limondo, Khoi Nguyen & Jamen Tenney
// Create Date: 12/18/2023
// Purpose:
// Modified (or history):  See Below
//-----
////////////////////////////////////
//
// Description:
// Course Name: ECE 5745/6755
//
// Revision:
// Revision 23182023 - jml/kn/jt - file created
// Additional Comments:
//
////////////////////////////////////

LIB "matrix.lib";
//system("--ticks-per-sec",1000000000); // nanoseconds
//system("--ticks-per-sec",1000000); // microseconds
system("--ticks-per-sec",1000); // milliseconds

ring S=0,x,lp;

int n = 3;

/* This matrix came from the paper, "EFFICIENT GROBNER BASIS REDUCTIONS FOR FORMAL
VERIFICATION OF GALOIS FIELD ARITHMETIC CIRCUITS in Section VII. IMPROVING POLYNOMIAL DIVISION
USING F4-STYLE REDUCTION" that was provided to us */
matrix A[n][n+1] =
1/3, 1/2, 0, 0,
2, 0, 1, 0,
0, 4, 0, -1;

printf("print(A);");
print(A);
print("");

timer = 1; // Start the timer
int startTime = timer; // Record the start time

list L=rowred(A,1);

int endTime = timer; // Record the end time

int elapsedTime = endTime - startTime; // Calculate the elapsed time

print("Solution for the system is: ");

for (int i = 1; i <= n; i++) {
    printf("X[%s] = %s", i, L[1][n + 1][i]);
}

print("");
```

```
//printf("Total time for rowred function in Singular to compile: %s nanoseconds",
elapsedTime);
//print("");

//printf("Total time for rowred function in Singular to compile: %s microseconds",
elapsedTime);
//print("");

printf("Total time for rowred function in Singular to compile: %s milliseconds", elapsedTime);
print("");

quit;
```

References

[1] J. Lv, P. Kalla and F. Enescu, "Efficient Gröbner Basis Reductions for Formal Verification of Galois Field Arithmetic Circuits," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 9, pp. 1409-1420, Sept. 2013, doi: 10.1109/TCAD.2013.2259540.

[2] Yu, Miao. "An F4-Style Involute Basis Algorithm." Master of Science, University of Southern Mississippi, 2010.

[3] "NVIDIA CUDA Toolkit 12.1 Downloads." *NVIDIA Developer*, developer.nvidia.com/cuda-downloads.

[4] *NVIDIA Developer Documentation*. docs.nvidia.com/cuda-libraries/index.html.

[5] "An Even Easier Introduction to CUDA | NVIDIA Technical Blog." *NVIDIA Technical Blog*, 21 Aug. 2022, developer.nvidia.com/blog/even-easier-introduction-cuda.

[6] *Gaussian Elimination*. people.cs.rutgers.edu/~venugopa/parallel_summer2012/ge.html.

[7] *Singular Manual: Singular Manual*. www.singular.uni-kl.de/Manual/latest/index.htm.

[8] Røne, Bjarke Hammersholt. "The F4 Algorithm: Speeding Up Gröbner Basis Computations Using Linear Algebra." <https://static1.squarespace.com/static/64f4f4bbed9a5e630f983d0f/t/64f55b151e67627eb1d90227/1693801238043/f4.pdf>.