



YAŞAR UNIVERSITY

FACULTY OF ENGINEERING

DEPARTMENT OF COMPUTER ENGINEERING

COMP4920 Senior Design Project 2, Spring 2020

Supervisor: Hüseyin Hışıl

FIWE: Factoring Integers with ECM

Final Report

(Bachelor of Science Thesis)

24.05.2020

By

Asena Durukan, Student ID: 16070001016

Aslı Altıparmak, Student ID: 15070001003

Elif Özbay, Student ID: Ç15070002011

Hasan Ozan Soğukpınar, Student ID: 17070001047

Nuri Furkan Pala, Student ID: 15070006030

Revision History

Revision	Date	Explanation
1.0	28.12.2019	Initial Final Report
2.0	24.05.2020	Projective and affine coordinate comparison is removed Adjusted to the new format Design section is extended Test design is extended and test results are added Appendices are updated

ACKNOWLEDGEMENTS

We would like to thank our academic advisor Dr. Hüseyin Hışıl for his great support and encouragement in this project. He guided us and helped us see it's not as complicated as it seems. Besides our advisor, we would like to thank Dr. Mutlu Beyazıt who conducted this lecture without any ambiguity.

KEYWORDS

Elliptic Curve Method (ECM), Factorization, safegcd, Projective Coordinates, Montgomery Curves, Paralellism, Single Instruction Multiple Data (SIMD).

ABSTRACT

In this paper, implementation and the parallelization of the ECM algorithm are aimed. ECM is an algorithm that factorizes an integer by using the properties of the elliptic curve arithmetic. It multiplies a point on the curve with a constant value to find a factor of n where the curve is defined modulo n . The ECM algorithm is implemented using projective space and Montgomery curves for the sake of performance. The parallelization is done on SIMD architecture of GPU. During the implementation phase, several programming languages and libraries are used such as C, Python, CUDA, Magma, SageMath and GMP library. In addition, NSIGHT is used as a supportive platform integrated with Eclipse for analyzing and debugging the parallelization process. To implement arithmetic operations with multi-precision integers, a multi-precision library is coded. Moreover, one of the steps of the ECM algorithm contains the calculation of the Greatest Common Divisor (GCD) of two integers. For this purpose, the binary GCD and the safegcd algorithms are coded and put inside the multi-precision library. The binary GCD is used inside the ECM algorithm and the integration of safegcd to the software is left as a future work. A library that can manage addition, doubling and multiplication of the points of an elliptic curve is also constructed. Finally, the main library that implements the ECM algorithm is coded. Concerning these needs, Requirement Specification Document (RSD), Design Specification Document (DSD) and the Product Manual are written and attached to this report. At the end of this project, a software that can factorize multiple multi-precision integers simultaneously, using ECM, is outputted.

ÖZET

Bu raporda ECM algoritmasının paralel bir şekilde kodlanması amaçlanmıştır. ECM, eliptik eğri aritmetiğinin özelliklerini kullanarak tam sayıları çarpanlarına ayıran bir algoritmadır. $(\text{mod } n)$ 'de tanımlanmış bir eğri üzerindeki bir noktayı sabit bir değerle çarparak n 'nin bir çarpanını bulmaya çalışır. İmplementasyon, yüksek performans amacıyla projektif uzayda Montgomery eğrileri kullanarak yapılmıştır. Parallelleştirme işlemi GPU'nun SIMD mimarisi üzerinde yapılmıştır. İmplementasyon safhasında, C, Python, CUDA, Magma, SageMath ve GMP gibi programlama dilleri ve kütüphaneler kullanılmıştır. Ek olarak, paralelizasyon sürecinde hata ayıklama ve analiz amacıyla, Eclipse ile entegre olmuş destekleyici bir platform olan NSIGHT'tan yararlanılmıştır. Çok basamaklı sayılarla aritmetik işlemler yapabilmek için çok basamaklı sayı kütüphanesi oluşturulmuştur. Dahası, ECM'nin adımlarından biri, iki sayının En Büyük Ortak Bölen (EBOB)'unun hesaplanmasını gerektirir. Bu amaç için binary GCD ve safegcd algoritmaları kodlanıp çok basamaklı sayı kütüphanesine konulmuştur. Binary GCD algoritması, ECM algoritması içerisinde kullanılmış, safegcd'nin yazılıma entegrasyonu ise gelecek çalışması olarak bırakılmıştır. Ayrıca, eliptik eğrilerdeki noktalarla, toplama, ikiye katlama ve çarpma işlemlerini yapabilen bir kütüphane kodlanmıştır. Bu ihtiyaçlar doğrultusunda, Gereksinim Spesifikasyon Dokümanı, Tasarım Dokümanı ve Kullanım Kılavuzu hazırlanmış ve bu rapora iliştilmiştir. Bu projenin sonunda, birden fazla çok basamaklı sayıyı ECM kullanarak, eş zamanlı, hızlıca çarpanlarına ayırabilen bir yazılım geliştirilmiştir.

TABLE OF CONTENTS

Revision History	ii
ACKNOWLEDGEMENTS	iii
KEYWORDS	iv
ABSTRACT	v
ÖZET	vi
TABLE OF CONTENTS	vii
LIST OF FIGURES	ix
LIST OF TABLES	x
LIST OF ALGORITHMS	xi
LIST OF CODES	xii
LIST OF ABBREVIATIONS	xii
1 INTRODUCTION	2
1.1 Description of the Problem	2
1.2 Project Goals	2
1.3 Project Outputs	3
1.4 Project Activities and Schedule	3
2 DESIGN	5
2.1 High Level Design	5
2.1.1 High Level Architecture	5
2.1.2 High Level Structure	6
2.1.3 High Level Environment	6
2.2 Detailed Design	6

2.2.1	Multi-precision Module	6
2.2.2	Montgomery Module	7
2.2.3	ECM Module	8
2.2.4	CUDA Module	8
2.3	Realistic Restrictions and Conditions in the Design	8
3	IMPLEMENTATION, TESTS and TEST DISCUSSIONS	10
3.1	Implementation of the Product	10
3.2	Tests and Results of Tests	11
4	CONCLUSIONS	12
4.1	Summary	12
4.2	Cost Analysis	13
4.3	Benefits of the Project	13
4.4	Future Work	14
	References	15
	APPENDICES	16
	APPENDIX A: REQUIREMENTS SPECIFICATION DOCUMENT	20
	APPENDIX B: DESIGN SPECIFICATION DOCUMENT	30
	APPENDIX C: PRODUCT MANUAL	50

LIST OF FIGURES

1.1	Gantt Chart of the Project	4
1	Flowchart of the ECM Algoritm	17
2	Flowchart of the Safegcd Algorithm	18

LIST OF TABLES

4.1 Cost Analysis 13

LIST OF ALGORITHMS


1	ECM Algorithm	19
2	Safegcd Algorithm	19

LIST OF ABBREVIATIONS

ECM	Elliptic Curve Method	iv, v, vi, 2, 3, 5, 6, 9,
GCD	Greatest Common Divisor	v, vi, 2, 6
RSD	Requirement Specification Document	v, 3, 12
DSD	Design Specification Document	v, 3, 12
SIMD	Single Instruction Multiple Data	iv, v, vi, 2, 5, 6, 9, 12
EBOB	En Büyük Ortak Bölen	vi


PLAGIARISM STATEMENT

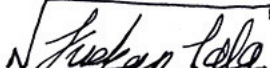
This report was written by the group members and in our own words, except for quotations from published and unpublished sources which are clearly indicated and acknowledged as such. We are conscious that the incorporation of material from other works or a paraphrase of such material without acknowledgement will be treated as plagiarism according to the University Regulations. The source of any picture, graph, map or other illustration is also indicated, as is the source, published or unpublished, of any material not resulting from our own experimentation, observation or specimen collecting.

Asena Durukan


Aslı ALTIPARMAK


Elif Üzbaş


Hasan Ozan Soğukpınar


Nuri Furkan PALA


Chapter 1

INTRODUCTION

In this part, description of the problem, project goals, project outputs, project activities and schedule are explained in detail.

1.1 Description of the Problem

Factorization of composite integers has been a challenge since old ages and it currently has no algorithm that works in polynomial time. So, this problem has a significant importance in the fields of cryptology due to its difficulty. Since this problem is used in cryptosystems, an algorithm that factorizes integers as fast as possible is a concern of cryptanalysis. The problem in detail and famous algorithms that try to solve it are explained in Appendix A.

1.2 Project Goals

The main goal of this project is the implementation of the ECM algorithm which is one of the solutions to the factorization problem [2]. This algorithm requires the GCD operation in some of its steps. Binary GCD is used inside ECM, and the integration of safegcd to the software is

left as a future work. Last but not least, the whole algorithm is aimed to be implemented on SIMD processors in a parallelized manner.

1.3 Project Outputs

The outputs for the first half of the project are the RSD (Appendix A) and the DSD (Appendix B). These outputs also act as the inputs of the second half of the project. Under the guidance of RSD and DSD files, a software that can factorize big composite integers with ECM is implemented. This software, this report and a Product Manual (Appendix C) are the outputs of the second half of the project.

1.4 Project Activities and Schedule

The schedule of the project is presented in Figure 1.1. All tasks are accomplished on time.

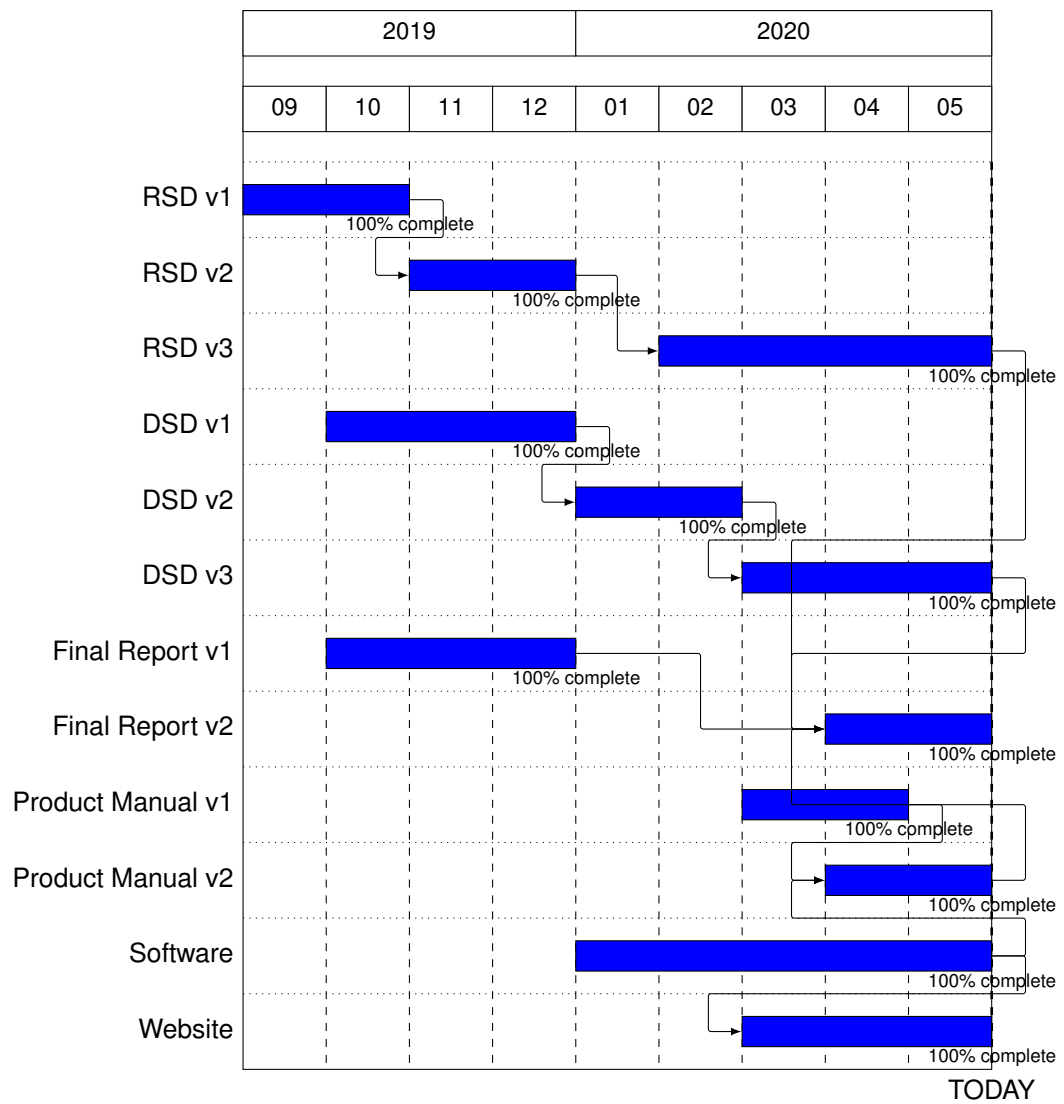


Figure 1.1: Gantt Chart of the Project

Chapter 2

DESIGN

Design part includes three main titles which are High Level Design, Detailed Design and Realistic Restrictions and Conditions in the Design. In Section 2.1, the design is split into four main parts and explained briefly. The design is detailly examined and clarified in Section 2.2. The limitations and the restrictions of the project that are determined and observed are mentioned in Section 2.3.

2.1 High Level Design

The High Level Design explains the architecture, structure, and environment of the system. The details of the design are mentioned in Section 2.2, Appendix B, and in the flowcharts Figure 1 and Figure 2.

2.1.1 High Level Architecture

One of the aims of this project is to factorize multiple multi-precision numbers simultaneously. Parallelization on the SIMD architecture is ideally suited for this purpose. Because, the SIMD

architecture allows applying a single instruction to multiple data at the same time. Therefore, the ECM algorithm is parallelized on the SIMD architecture. The details of the high level architecture are described on Section 3.1 of Appendix B.

2.1.2 High Level Structure

The project can be divided into four modules; Multi-precision module, Montgomery module, ECM module and CUDA module which are listed in Section 3.2 of Appendix B.

- **Multi-precision Module:** To represent multi-precision numbers, do arithmetic operations with them, calculate GCD and modular inverse
- **Montgomery Module:** To represent Montgomery curves, points and do operations on the points
- **ECM Module:** To implement ECM
- **CUDA Module:** To parallelize the ECM algorithm

2.1.3 High Level Environment

Due to the specifications of SIMD architecture, a GPU is used as an environment. Besides, CPU is necessary for the functions which cannot be parallelized. Other than the hardware environment, several programming languages, libraries and editors are used throughout the project such as; CUDA, C, Python, GMP, and so on. Details are described in Section 3.3 of Appendix B.

2.2 Detailed Design

This section lists the content of the modules. Detailed description of the content is mentioned in Section 4 of Appendix B.

2.2.1 Multi-precision Module

- **W:** Word size of the architecture

- **fiweCopy:** Copies a multi-precision integer to another
- **fiweRand:** Generates random multi-precision integers up to a given size
- **fiweModRand:** Generates random multi-precision integers up to given size in mod $\text{mod } n$
- **fiwePrint:** Prints the multi-precision number in base 10
- **fiweAdd:** Adds a and b
- **fiweModAdd:** Calculates $a + b \pmod{n}$
- **fiweSub:** Subtracts b from a
- **fiweModSub:** Calculates $a - b \pmod{n}$
- **fiweMul:** Multiplies a with b
- **fiweModMul:** Calculates $a * b \pmod{n}$
- **fiweBarretReduction:** Calculates $m \pmod{n}$
- **fiweBinaryGCD:** Calculates $\gcd(a, b)$ using the binary GCD algorithm
- **fiweSafeGCD:** Calculates $\gcd(a, b)$ using the safegcd algorithm
- **fiweInvert:** Calculates $a^{-1} \pmod{n}$

2.2.2 Montgomery Module

- **fiweMontgCurve:** Structure to hold a Montgomery curve
- **fiweProPoint:** Structure to hold a projective point
- **fiweProCurvePoint:** Initializes a randomly generated Montgomery curve and a projective point on it
- **fiweProAdd:** Adds two projective points on a curve
- **fiweProDbI:** Doubles a projective point on the curve
- **fiweProLadder:** Multiplies a projective point on the curve with the given constant

2.2.3 ECM Module

- **fiweECM:** Factorizes the given composite n using ECM

2.2.4 CUDA Module

All the functions mentioned above are also implemented in CUDA. However CUDA programming language requires additional definitions and functions.

- **Device Array:** An array to be used by GPU
- **SIZE:** Total number of multi-precision numbers to be factorized
- **Block Number:** Number of thread blocks
- **Thread Number:** Number of threads
- **Kernel Functions:** Functions that are executed by the GPU
- **Thread Index:** Index of each thread
- **cudaMalloc:** Allocates memory for GPU arrays
- **cudaMemcpy:** Copies a GPU array to a CPU array, or vice versa
- **cudaMallocManaged:** Allocates memory for the destination address that can be used by both GPU and CPU

2.3 Realistic Restrictions and Conditions in the Design

There are some assumptions needed to be defined.

Mathematical Assumptions

Some mathematical specifications of the ECM algorithm are listed below where; $P = (X, Y, Z)$ is a point on an elliptic curve, ∞ is the point at infinity defined for the curve, A and B are the coefficients of the curve, and $B * Y^2 * Z = X^3 + A * X^2 * Z + X * Z^2$.

- $P + \infty = P$

- If $P = (x, y)$, then $\neg P = (x, -y)$
- $P + \neg P = \infty$
- $4A^2 + 27B^3 \neq 0$

Hardware Assumptions

- The user must have a computer with SIMD architecture
 - NVIDIA Video cards / CUDA

Software Assumptions

- Several software are needed for the compilation
 - NVCC is needed in order to compile CUDA codes
 - GCC 7.5.0 is needed for NVCC to work and to compile C codes
 - GMP should be installed
 - A Linux distribution should be used as the operating system for GMP to run

Restrictions

- ECM does not guarantee to find a factor. Therefore the algorithm terminates after a number of trials.
- ECM has a higher chance of success when n is B-smooth.
- Even though parallel programming on GPU is faster than CPU multithreading, it has some memory limitations.
- While synchronization methods are effective for protecting shared state, they can also be used incorrectly, failing to accomplish the proper synchronization, over-synchronizing, or causing the program to hang as a result of deadlock [1]. Synchronization on GPUs might expose these issues.

Chapter 3

IMPLEMENTATION, TESTS and TEST DISCUSSIONS

There are two main parts in this section. In the first one, implementation of the project is briefly described. The second one explains the tests and demonstrates the results of them.

3.1 Implementation of the Product

The implementation of the project had started with coding ECM in C language which was an abstraction layer for CUDA. Later on, the completed C functions were transferred to CUDA language. Due to this transfer, parallelization process was started. Upon completion of the CUDA implementation, it became possible to factorize multiple multi-precision numbers simultaneously. Moreover, concurrent with the ECM and CUDA implementations, `safegcd` is implemented in C, ready to be run on CPU. The details of the implementation are described in Appendix C.

3.2 Tests and Results of Tests

Throughout the project, each function written is tested individually. All the test procedures are coded and put into a module. The test procedures are applied to all functions in C, and to the crucial functions in CUDA. Briefly, test procedures are composed of; generating random multi-precision numbers, passing the numbers to the functions and confirming the result using a high level calculator such as GMP and Magma. The test functions are listed below. Step-by-step descriptions of the test functions can be found in Appendix C. All functions are tested 10000 times and resulted with 100% success rate.

- Montgomery Curve and Projective Point Generation Function GMP Test
- Projective Addition Function GMP Test
- Projective Addition Function Magma Test
- Projective Doubling Function Magma Test
- Projective Ladder Function GMP Test
- Projective Ladder Function Magma Test
- safegcd Function Python3 Test
- ECM Function GMP Test

Chapter 4

CONCLUSIONS

Within the scope of this project, the ECM algorithm is implemented and parallelized on the SIMD architecture. Furthermore, safegcd is implemented to be run on CPU. The overall summary, cost analysis, benefits of the project and future work parts are explained in detail in the below sections.

4.1 Summary

The project started with the literature review for the subject. With the guidance of the researches, RSD document is started to be constructed. Simultaneously, the implementation of the ECM and safegcd algorithms are started. The implementation is done using Python and C programming languages to get used to the algorithms and to generate an abstraction layer for the parallelized version of ECM. The requirements, methods and the processes of the system are explained in the RSD while DSD is started to be constructed by mentioning the architecture, structure and the environment for the project. It followed by the CUDA language implementation. The ECM code is parallelized on the SIMD architecture of the GPU using CUDA. Test procedures for both C and CUDA are generated and described in Appendix C. In the end, a software that can be used as a library or a command-line application is produced

month	work hour per member				total	
	meeting	report	design	implementation		
2019-09	8	8	6	0	22	
2019-10	10	12	10	0	32	
2019-11	10	14	10	0	34	
2019-12	12	14	6	0	32	
2020-01	9	3	6	18	36	
2020-02	8	16	5	25	54	
2020-03	9	15	3	25	52	
2020-04	10	21	2	29	62	
2020-05	8	26	3	27	64	
total	84	129	51	124	388	total hour per member
					1940	total hour for group
					242	total man-days for group

Table 4.1: Cost Analysis

as an output.

4.2 Cost Analysis

For this project, 242 man-days are spent in total. It can also be interpreted as 48 man-days for each group member in a period of 9 months. The details are represented in Table 4.1.

4.3 Benefits of the Project

One of the benefits of this project is to provide a resource that can guide people who study in the fields of cryptology, cryptanalysis and some other related subjects. Anyone can understand and use this software by reading the reports and examining the source codes. Moreover, this project has provided the group members to understand the integer factorization problem and a solution to this problem, no matter how professional it is, and to have a basic knowledge of cryptology and cryptanalysis. It is going to help users understand this concept and implement it in their own work.

4.4 Future Work

- The ECM algorithm has a second phase which aims to find the factor that first phase could not found. Second phase can be implemented as a future work.
- The integration of the safegcd algorithm to the ECM algorithm is possible.
- The ECM algorithm is implemented using projective coordinates, however affine coordinates can also be used. A comparison can be made between the performances of ECM using projective and affine coordinates.
- For the parallelization part, the software can be implemented on brand new GPUs and CPUs or with different architectures.

References

- [1] J. DeNero. Synchronization pitfalls.
- [2] H. W. Lenstra. Factoring integers with elliptic curves. volume 126, pages 649–673. *Annals of Mathematics*, 1987.

APPENDICES

```
1  __global__ void fiwEcm(ui d, ui n, ui_t nl, MONTG_CURVE c, PRO_POINT p, PRO_POINT p1, PRO_POINT RO,  
2    PRO_POINT RO_, PRO_POINT R1, PRO_POINT R1_, ui mu, ui_t mul){  
3  
4    int t_index = threadIdx.x + (blockIdx.x * blockDim.x);  
5  
6    if(t_index < SIZE){  
7  
8        ui A24 = new ui_t[nl];  
9        ui k = new ui_t[1];  
10       ui temp_k = new ui_t[nl];  
11  
12       int i, j, flag = 0;  
13       ui_t is_zero = 0, is_n = 0, is_one = 0;  
14  
15       for(i = 0; i < CRV_THRESHOLD; i++) {  
16           do {  
17               fiweProCurvePoint(d, c, p1, n, nl, mu, mul, &flag);  
18           } while(flag == -1);  
19           if(flag == 0) {  
20               //returnArr[t_index] = 1;  
21               return;  
22           } else {  
23               fiweGetA24(A24, c[t_index].A, n, nl, mu, mul, &flag);  
24  
25               if(flag == 1) {  
26                   fiweGenerateBSmooth(k, 1);  
27  
28                   j = 0;  
29                   while(j < 5) {  
30                       fiweProLadder(p, p1, RO, RO_, R1, R1_, A24, k, 1, n, nl, mu, mul);  
31  
32                       fiweIsEqualUi(&is_zero, p[t_index].Z, nl, 0L);  
33  
34                       if(is_zero == 1) {  
35                           temp_k[0] = k[0];  
36                           for(i = 1; i < nl; i++)  
37                               temp_k[i] = 0;  
38                           fiweBinaryGCD(d, temp_k, nl, n, nl);  
39  
40                           fiweIsEqualUiD(&is_one, d, nl, 1L);  
41  
42                           fiweIsEqualD(&is_n, d, n, nl);  
43  
44                           if(!is_one && !is_n) {  
45                               //returnArr[t_index] = 2;  
46                               return;  
47                           }  
48                           fiweGenerateBSmooth(k, 1);  
49  
50                           j++;  
51                       }  
52                   } else {  
53                       //returnArr[t_index] = 3;  
54                       return;  
55                   }  
56               }  
57           }  
58       }  
59   }
```

```

56 |     }
57 | }
58 |
59 | //returnArr[t_index] = 0;
60 | return;
61 | }
62 | }

```

Code 1: ECM CUDA Implementation

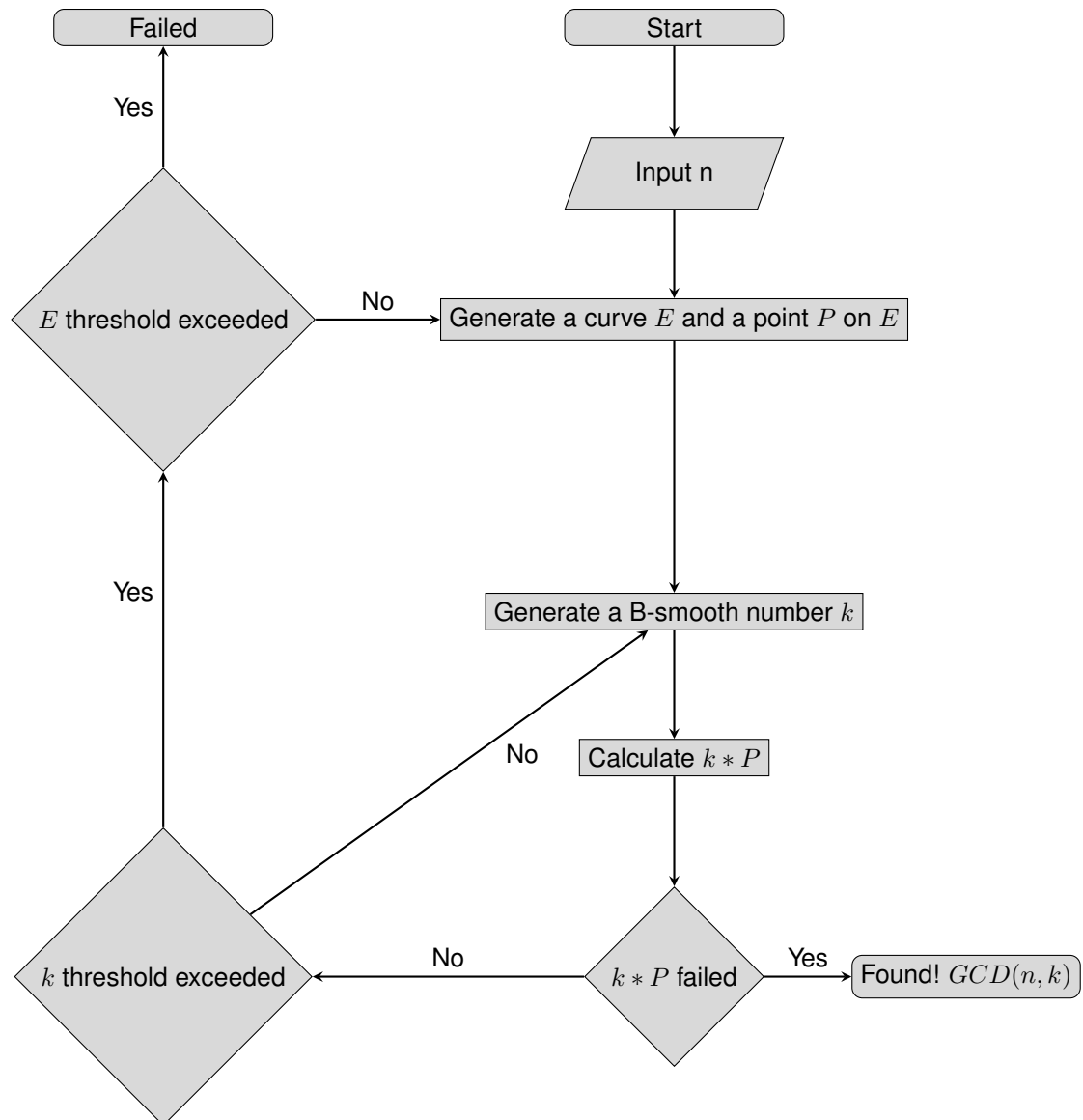


Figure 1: Flowchart of the ECM Algorithm

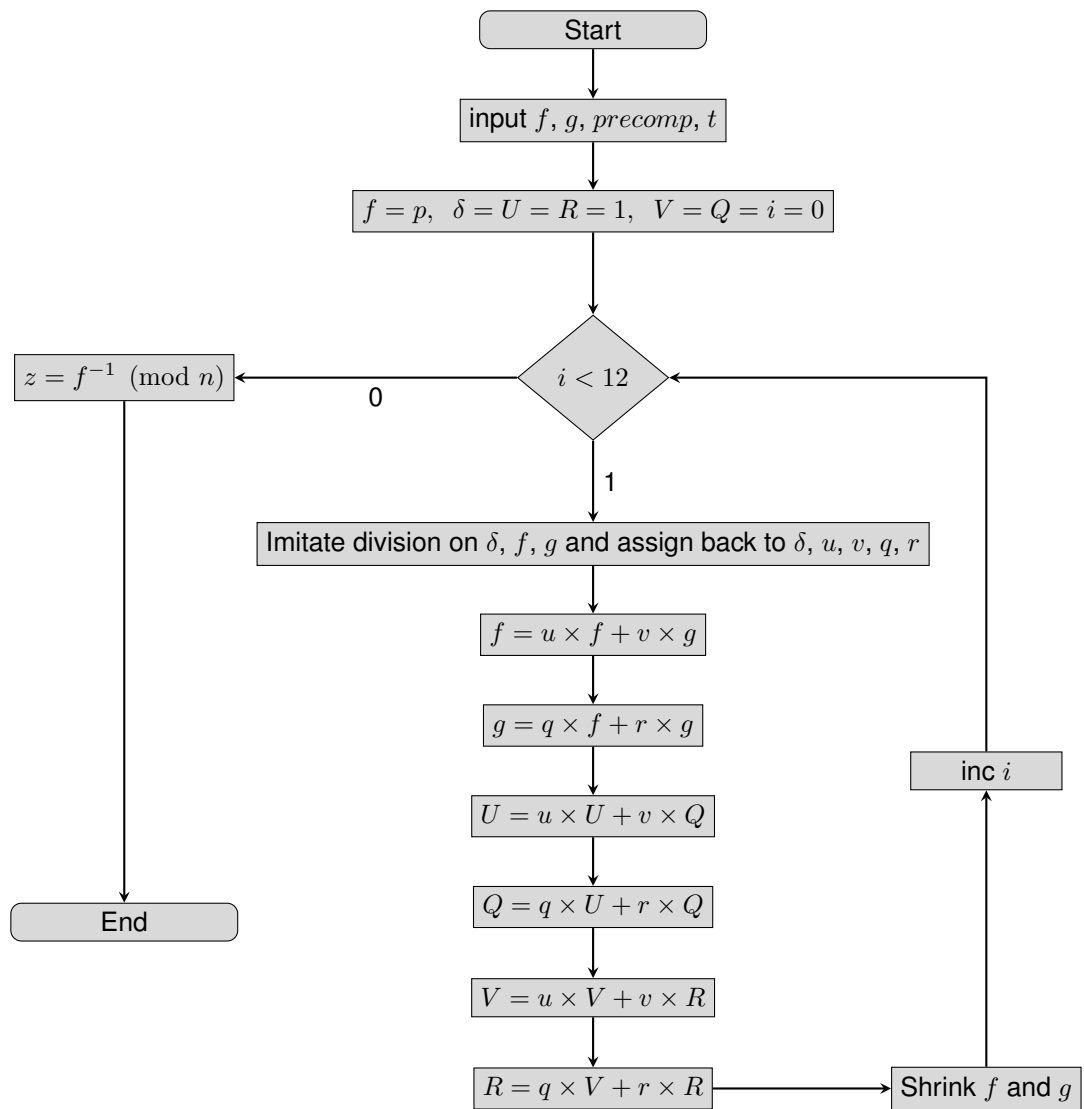


Figure 2: Flowchart of the Safegcd Algorithm

Algorithm 1: ECM Algorithm

```

input   :  $n$ : composite number to be factorized
output  :  $d$ : factor of  $n$ 
1 Generate a random projective curve  $E$  and a point  $P$  on it
2 Choose an integer  $k$  that is composed of small prime factors
3 Compute  $k * P$  on  $E$ 
4 if  $k * P$  failed due to the lack of inverse of an element then
5   | return  $d = \gcd(k, n)$ 
6 else
7   | if Threshold for  $k$  has not been exceeded then
8   |   | Go back to Step 2.
9   | else
10  |   | if Threshold for  $E$  has not been exceeded then
11  |   |   | Go back to Step 1.
12  |   |   | else
13  |   |   |   | Failed. Stop.
14  |   |   |   | end
15  |   | end
16 end

```

Algorithm 2: Safegcd Algorithm

```

input   :  $f, g, precomp, t$ 
output  :  $z$ 
1 Let  $mul(a, b, c, d)$  be  $a \times b + c \times d$ 
2 Initialize  $f$  as  $p$ 
3 Initialize  $\delta, U, R$  as 1
4 Initialize  $V, Q, i$  as 0
5 while  $i < 12$  do
6   | Imitate division on  $\delta, f, g$  and assign back to  $\delta, u, v, q, r$ 
7   | Assign  $mul(u, f, v, g)$  to  $f$ 
8   | Assign  $mul(q, f, r, g)$  to  $g$ 
9   | Assign  $mul(u, U, v, Q)$  to  $U$ 
10  | Assign  $mul(q, U, r, Q)$  to  $Q$ 
11  | Assign  $mul(u, V, v, R)$  to  $V$ 
12  | Assign  $mul(q, V, r, R)$  to  $R$ 
13  | Shrink  $f$  and  $g$ 
14  | Increment  $i$ 
15 end
16 Assign  $f^{-1} \pmod{n}$  to  $z$ 

```

APPENDIX A: REQUIREMENTS SPECIFICATION DOCUMENT

COMP4920 Senior Design Project 2, Spring 2020
Advisor: Dr. Hüseyin Hışıl

FIWE: Factoring Integers with ECM Requirements Specification Document

24.05.2020
Revision 3.0

By:
Asena DURUKAN
Student ID Number: 16070001016
Aslı ALTIPARMAK
Student ID Number: 15070001003
Elif ÖZBAY
Student ID Number: Ç15070002011
Hasan Ozan SOĞUKPINAR
Student ID Number: 17070001047
Nuri Furkan PALA
Student ID Number: 15070006030

Revision History

Revision	Date	Explanation
1.0	26.11.2019	Initial Requirements
2.0	28.12.2019	Requirements Model
3.0	24.05.2020	Projective and affine coordinate comparison is removed Group law example is converted to projective coordinates ECM pseudocode is modified Usability requirement is added

Contents

Revision History	1
Contents	2
1 Introduction	3
1.1 Background	3
1.1.1 Factorization of Integers	3
1.1.2 Factorization Methods	3
1.1.3 Elliptic Curves	4
1.1.4 Elliptic Curve Method	5
1.1.5 Parallelization of the Algorithm	5
1.1.6 Safegcd	6
2 Requirements List	6
2.1 Functional Requirements	6
2.1.1 Mathematical Requirements	6
2.1.2 Parallelization Requirement	7
2.1.3 Usability Requirement	7
2.2 Non-functional Requirement	7
3 Actors and Use Cases	7
4 Glossary	8
References	8

1 Introduction

Lenstra's Elliptic Curve Method (ECM) is one of the algorithms that are proposed as a solution to the factorization of integers problem which is commonly used in asymmetric cryptosystems. The method simply does arithmetic operations with the points on an elliptic curve and finds a factor of the integer that the curve is defined over. The aim of this project is to implement the ECM algorithm on GPU's Single Instruction Multiple Data (SIMD) architecture by parallelizing the algorithm.

1.1 Background

1.1.1 Factorization of Integers

Factorization of composite integers is a difficult problem which has no known algorithm that works in polynomial time. The difficulty of prime factorization makes it a valuable problem in cryptology. Many cryptosystems are constructed upon a one-way function which means fast to solve in one-way and slow in the other way, such as factorization. It is quite simple to multiply 5 with 17. However, it takes a significantly longer time to find the prime factors of 85. This computation time increases exponentially with respect to the number being factorized when the composite number is chosen larger in size. Since this problem is used as a basis of common cryptosystems, many studies have been conducted to figure out an algorithm that factorizes integers as fast as it can be. The most famous algorithms are briefly explained in Section 1.1.2.

1.1.2 Factorization Methods

Factorization methods are usually categorized into two classes which are special and general algorithms. Special algorithms are applicable to a specific group of numbers while general algorithms are appropriate for all composite numbers [3].

Common Special Algorithms

Pollard's $p - 1$ method is an algorithm that relies on *Fermat's Little Theorem* which states that $a^{p-1} \equiv 1 \pmod{p}$ where p is prime and $\gcd(a, p) = 1$. The performance of the algorithm depends strictly on the selection of a . Moreover, the method works efficiently if $p - 1$ is $B - \text{smooth}$ for an arbitrarily chosen B (composed of small primes until B), where p is a prime factor of the number to be factorized. Pollard's ρ method is another special algorithm that is based upon the birthday paradox which says that two persons have the same birthday if they are chosen from a group of at least 23 people, with the probability of 50%.

Elliptic Curve Method is a derivation of the $p - 1$ method which replaces the multiplicative group with the group of points on an elliptic curve. This modification overcomes the disadvantage of the $p - 1$ method that requires the smoothness of $p - 1$ [5].

Common General Algorithms

Congruent Squares is based on the equation $x^2 - y^2 = (x + y)(x - y)$. If n is a multiple of $x^2 - y^2$, then the factors of n must divide $x + y$ and $x - y$ [3].

Quadratic Sieve tries to find x and y values such that $x^2 \equiv y^2 \pmod{n}$ and in case of success, applies *Euclidean Algorithm* for the probability of a nontrivial factor [4].

Number Field Sieve is a method similar to *Quadratic Sieve* but combined with algebraic number fields. The difference is the field that is used [3].

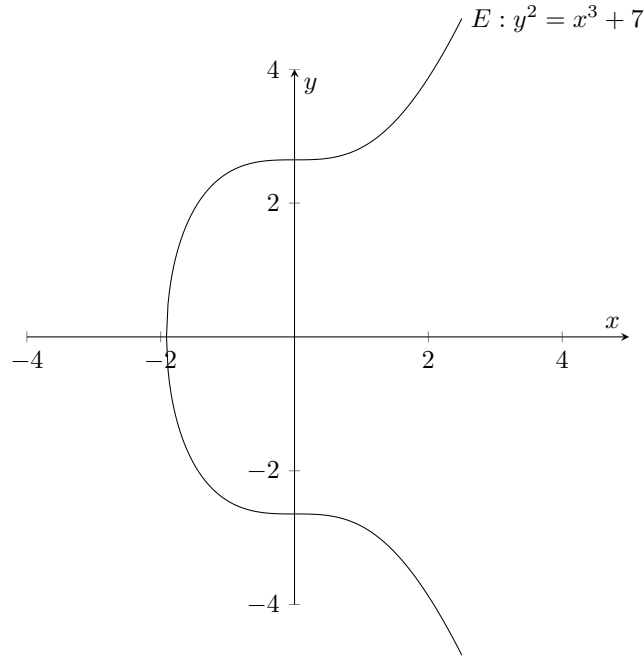


Figure 1: Elliptic Curve

1.1.3 Elliptic Curves

An elliptic curve is a set of points that lies in the equation

$$y^2 = x^3 + Ax^2 + B \quad (1)$$

called *Short Weierstrass Equation* where A and B are constants, together with a hypothetical point ∞ . The x and y values that satisfy the equation represent the points on the curve defined. The demonstration of an elliptic curve is shown in Figure 1. A curve can be defined over a group, ring, or field. Defining an elliptic curve over a finite structure makes the calculations easier.

The points can be represented as x, y pairs which are called the affine representation. With a simple map, the points can be represented in projective coordinates by adding the third parameter Z , such as X, Y, Z . A point can be represented in both ways, and the mapping from one to the other is always possible.

The Group Law

Since an elliptic curve is a group, the addition operation must be defined. Let $P_1 = (X_1, Y_1, Z_1)$, $P_2 = (X_2, Y_2, Z_2)$ and $P_3 = P_1 + P_2$. Also, assume a point $P_d = (X_d, Y_d, Z_d)$ is the difference of P_1 and P_2 .

$$\begin{aligned} X_3 &= Z_d * ((X_1 - Z_1) * (X_2 + Z_2) + (X_1 + Z_1) * (X_2 - Z_2))^2 \\ Z_3 &= X_d * ((X_1 - Z_1) * (X_2 + Z_2) - (X_1 + Z_1) * (X_2 - Z_2))^2 \end{aligned} \quad (2)$$

Addition of two points P_1 and P_2 in projective coordinates is declared algebraically in Equation 2 [2]. It can also be described geometrically. Geometrically, the addition operation has two steps. In the first step, the line that goes through the points P_1 and P_2 is calculated. That line certainly intersects the curve on a third point. In the second step, the symmetric of that calculated point is taken, according to the x-axis. That point P_3 is the result of the addition.

Doubling is similar to the addition operation. To find the line that goes through the point, its derivative is taken. The second step is same with the addition.

1.1.4 Elliptic Curve Method

The method aims to find the factors of an integer n greater than 1. The algorithm is explained below.

Algorithm 1: Elliptic Curve Method Algorithm

```

input :  $n$ : composite number to be factorized
output:  $d$ : factor of  $n$ 
1 Generate a random projective curve  $E$  and a point  $P$  on it
2 Choose an integer  $k$  that is composed of small prime factors
3 Compute  $k * P$  on  $E$ 
4 if  $k * P$  failed due to the lack of inverse of an element then
5   | return  $d = \gcd(k, n)$ ;
6 else
7   | if Threshold for  $k$  has not been exceeded then
8     | Go back to Step 2.
9   | else
10    | if Threshold for  $E$  has not been exceeded then
11      | Go back to Step 1.
12    | else
13      | Failed. Stop.
14    | end
15  | end
16 end

```

1.1.5 Parallelization of the Algorithm

Parallel computing is the simultaneous use of multiple computation resources to solve a computational problem. In other words, it is the process of large computations which can be broken down into multiple processors using multiple threads that can process independently and whose results are combined upon the completion [6]. A key problem of parallelism is to reduce data dependencies in order to be able to perform computations on independent computation units with minimal communication between them. However, on the SIMD architecture, data dependency is not a problem to be considered since each data is processed by an identical set of instructions.

The aim of this project is to parallelize the mathematical operations of the ECM algorithm on the CUDA platform.

1.1.6 Safegcd

A Greatest Common Divisor (GCD) algorithm is needed in the ECM algorithm. GCD is a concept in number theory defining the greatest of the integers that divides each operand. Generally, it is represented as $\gcd(a, b)$ where a and b are positive integers.

Modular inversion operation is also a part of the ECM algorithm. Basically, the formula of modular multiplicative inversion is $a \times a^{-1} = 1 \pmod{n}$ where a^{-1} is the modular inverse of a in mod n .

An algorithm called *safegcd* is announced recently which is a variation of the binary GCD algorithm [1]. This algorithm is fast and it can be easily converted to a constant time algorithm, which is essential in crypto since non-constant time algorithms cause security vulnerabilities. Additionally, the method has the potential to be parallelized and speeded up which is this project seeks for. The safegcd algorithm is implemented within the scope of this project. It is chosen due to its speed, recency and the potential to be parallelized. One of the parts is for integer inputs, and the other is for polynomial inputs [1]. Implementing the algorithm for integer inputs, which is given in Algorithm 2 is enough for the ECM algorithm.

Algorithm 2: Safegcd Algorithm

input : $f, g, precomp, t$
output: z

- 1 Let $mul(a, b, c, d)$ be $a \times b + c \times d$
- 2 Initialize f as p
- 3 Initialize δ, U, R as 1
- 4 Initialize V, Q, i as 0
- 5 **while** $i < 12$ **do**
- 6 Imitate division on δ, f, g and assign back to δ, u, v, q, r
- 7 Assign $mul(u, f, v, g)$ to f
- 8 Assign $mul(q, f, r, g)$ to g
- 9 Assign $mul(u, U, v, Q)$ to U
- 10 Assign $mul(q, U, r, Q)$ to Q
- 11 Assign $mul(u, V, v, R)$ to V
- 12 Assign $mul(q, V, r, R)$ to R
- 13 Shrink f and g
- 14 Increment i
- 15 **end**
- 16 Assign $f^{-1} \pmod{n}$ to z

2 Requirements List

2.1 Functional Requirements

2.1.1 Mathematical Requirements

Arithmetic

- Storing multi-precision integers
- Arithmetic operations on multi-precision integers

Elliptic Curves

- Defining and storing elliptic curves
- Arithmetic operations on elliptic curves

Greatest Common Divisor

- Calculating the GCD of two integers

Integer Modular Inversion

- Calculating the modular inverse of an integer

2.1.2 Parallelization Requirement

- To be able to factorize multiple composite numbers simultaneously

2.1.3 Usability Requirement

- A user interface which user can factorize a composite number simply

2.2 Non-functional Requirement

- The GCD must have a small time complexity

3 Actors and Use Cases

The only purpose of the software is factoring integers. The user is going to enter the large composite number as the input and get the factors of it as the output. The use case diagram of the software is shown in Figure 2.

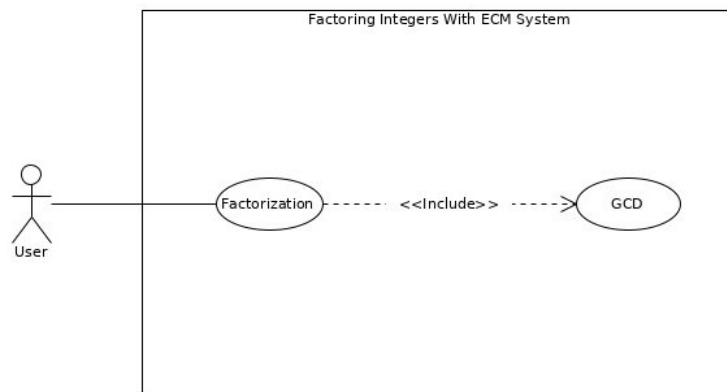


Figure 2: Use Case of the Software

4 Glossary

In this paper, some notations are used to eliminate repetitions. The notations are listed below. Unless otherwise stated, they are going to refer to the following descriptions in the list.

- B : An integer that composed of small primes
- CUDA: Computer Unified Device Architecture
- E : Elliptic curve
- ECM: Elliptic Curve Method
- GCD: Greatest Common Divisor
- GPU: Graphics Processing Unit
- n : The composite number to be factorized
- $P = (x, y)$: A point on the curve
- SIMD: Single Instruction Multiple Data
- ∞ : Point at infinity on E

References

- [1] D. J. Bernstein and B. Yang. Fast constant-time gcd computation and modular inversion. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(3):340–398, 2019.
- [2] H. Hışıl. *Elliptic Curves, Group Law and Efficient Computation*. PhD thesis, Queensland University of Technology, 4 2010.
- [3] P. L. Jensen. Integer factorization. Master’s thesis, University of Copenhagen, 2005.
- [4] E. Landquist. The quadratic sieve factoring algorithm. 01 2001.
- [5] H. W. Lenstra. Factoring integers with elliptic curves. volume 126, pages 649–673. *Annals of Mathematics*, 1987.
- [6] D. Thakur. What is parallel computing? - definition.

APPENDIX B: DESIGN SPECIFICATION DOCUMENT

COMP4920 Senior Design Project II, Spring 2020
Advisor: Dr. Hüseyin Hışıl

FIWE: Factoring Integers with ECM Design Specification Document

Revision 3.0
24.05.2020

By:

Asena Durukan, Student ID: 16070001016
Aslı Altıparmak, Student ID: 15070001003
Elif Özbay, Student ID: Ç15070002011
Hasan Ozan Soğukpınar, Student ID: 17070001047
Nuri Furkan Pala, Student ID: 15070006030

Revision History

Revision	Date	Explanation
1.0	28.12.2019	Initial High Level Design
2.0	16.03.2020	Software System Detailed Design is added Software Subsystem Design is revised Testing Design is revised Adjusted to the new format Unused paragraphs from v1.0 are grayed out
3.0	24.05.2020	Projective and affine coordinate comparison is removed Added GMP to the environment Renamed Arithmetic Module as Multi-precision Module Added new functions to modules Safegcd functions are modified

Table of Contents

Revision History	2
Table of Contents	3
1 Introduction	5
2 FIWE System Design	5
3 FIWE Software Subsystem Design	5
3.1 FIWE Software Subsystem Architecture	6
3.1.1 SIMD	6
3.1.2 PASCAL	6
3.2 FIWE Software Subsystem Structure	6
3.2.1 Multi-precision Arithmetic	6
3.2.2 Montgomery Curve Operations	7
3.2.3 Greatest Common Divisor	7
3.3 FIWE Software Subsystem Environment	7
3.3.1 INTEL CPU	7
3.3.2 NVIDIA GPU	8
3.3.3 CUDA (10.0 - 2015)	8
3.3.4 C (C11 - 2011)	8
3.3.5 Python (3.8.1 - 2019)	8
3.3.6 Sage (9.0 - 2020)	8
3.3.7 Magma (2.25-4 - 2020)	8
3.3.8 GMP (6.2.0 - 2020)	8
3.3.9 Inline x86-64 Assembly (AT&T)	9
3.3.10 Nsight (9.2 - 2018)	9
4 FIWE Software System Detailed Design	9
4.1 FIWE Main Module	9
4.2 FIWE Subsystem Multi-precision Module	9
4.2.1 Multi-precision Module Macros	9
4.2.2 Multi-precision Module Functions	10
4.2.3 Safeged Functions	12
4.3 FIWE Subsystem Montgomery Module	13
4.3.1 Montgomery Module Structures	13
4.3.2 Montgomery Module Functions	14
4.4 FIWE Subsystem ECM Module	15
4.4.1 ECM Module Functions	16
4.5 FIWE Subsystem CUDA Implementation	17
4.5.1 CUDA Module Macros	17
4.5.2 CUDA Module Functions	18
5 Testing Design	18

1 Introduction

Elliptic Curve Method (ECM) is an algorithm, developed by Arjen K. Lenstra in 1987, to speed up integer factorization using elliptic curves which is a variation of Pollard's $p-1$ method. However, ECM eliminates a pitfall of the $p-1$ method. The ECM algorithm can be realized using many different elliptic curves such as Weierstrass, Edwards, Montgomery, and so on. In this project, Montgomery curves are chosen due to their speed and suitability. The curves and the points on them can be represented in two ways; affine coordinates and projective coordinates. Since arithmetic operations with projective coordinates are much faster than affine coordinates, the implementation of the ECM algorithm is done using projective coordinates.

The ECM algorithm needs GCD and modular inversion operations, and the `safegcd` algorithm is suitable for this purpose. `Safegcd` is a variant of the binary GCD algorithm. Besides, it is constant time, fast and recent. Also, it is appropriate for parallelization. The code is optimized and parallelized on the Single Instruction Multiple Data (SIMD) architecture of GPU. Hence, multiple composite numbers can be factorized simultaneously. The details of ECM, `safegcd`, and parallelization are described in RSD [7].

The overall design, by means of software and hardware, is going to be mentioned in Section 2. In Section 3, the software design components are going to be explained in detail. In Section 4, Section 2 is going to be expanded comprehensively. The methodology used to verify and validate the software is going to be described in Section 5.

2 FIWE System Design

The system consists of the implementation of the ECM algorithm on the SIMD architecture of GPU. This implementation requires different mathematical functionalities, and these functionalities are separated into various libraries. The libraries are named as; Multi-precision Library, Montgomery Library, and ECM Library. These libraries include several structures and functions.

The system consists of 5 main parts including the libraries mentioned:

- Construction of a Multi-precision Library
- Construction of a library for Montgomery curves
- Construction of ECM Library
- Implementation of `safegcd`
- Parallelization of the software on SIMD architecture

The need for the operations on multi-precision integers is met in the Multi-precision Library. Therefore, the integers which are bigger than the word size of the computer can be factorized. Since ECM requires operations on the points on the curve, Montgomery Library is constructed for this purpose. The ECM Library implements the ECM algorithm using the other libraries. Besides, GCD calculation is a common operation in the ECM algorithm. Finally, to factorize multiple multi-precision numbers simultaneously, the whole implementation is transferred to CUDA to be able to parallelize it on NVIDIA GPU. Section 4 contains additional details about these libraries.

3 FIWE Software Subsystem Design

The Software Subsystem Design section describes software specifications such as software architecture, software structure and, software environment.

3.1 FIWE Software Subsystem Architecture

The overall software system is based on SIMD architecture. However, the subsystems of this project are monolithic in particular.

3.1.1 SIMD

The parallelization of the ECM algorithm is done on SIMD architecture. This allows the factorization of multiple multi-precision integers simultaneously.

SIMD is a class of parallel computers in Flynn's taxonomy. It describes computers with multiple processing elements that perform the same operation on multiple data points simultaneously [10]. Kindervater and Lenstra explained the SIMD architecture as "One type of instruction is performed at a time, possibly on different data" [9].

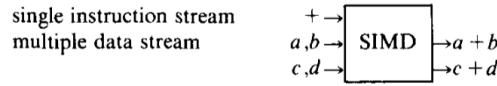


Figure 1: SIMD Architecture [9]

3.1.2 PASCAL

Pascal is an architecture designed by NVIDIA, as the successor of the Maxwell architecture which is NVIDIA's next generation architecture for computing CUDA. Pascal's Streaming Multiprocessor Architecture improves CUDA Core utilization and power efficiency, resulting in significant overall GPU performance improvements, and allowing higher core clock speeds compared to the previous GPUs [5]. Pascal Architecture has 6 Streaming Multiprocessors with 128 cores in each and 64KB register files of 32-bit registers as shown in Figure 2.

3.2 FIWE Software Subsystem Structure

The ECM algorithm needs different functions to handle unsupported mathematical operations. For this reason, various libraries are constructed with different functionalities. These libraries are described in the following sections.

3.2.1 Multi-precision Arithmetic

Factoring integers can be done with trial and error method. However, when the integer that is going to be factorized gets bigger, it takes a long time. The purpose of the ECM algorithm is to factorize multi-precision integers in a short time interval. These integers cannot be represented to the processor using common data types. Therefore, it is a requirement to represent integers bigger than the word size, and do arithmetic operations with them.

The integers are represented in base 2^W where W is the word size, by using arrays with W bit integers in them. Such as

$$[a_0, a_1, a_2, a_3] = a_0 * 2^{W^0} + a_1 * 2^{W^1} + a_2 * 2^{W^2} + a_3 * 2^{W^3} \quad (1)$$

Since the numbers are stored in such a way, the built-in arithmetic operations cannot be implemented on them. In the Multi-precision Arithmetic Library, addition, subtraction, multiplication, reduction, and other auxiliary operations are coded as functions.



Figure 2: Pascal Architecture [5]

3.2.2 Montgomery Curve Operations

The curves are represented in projective spaces, to be used in ECM. For this purpose, a structure is used to store necessary information about the curve in it. Points are also stored similarly.

Functions for point addition and doubling is implemented in the library. In addition to these, an algorithm called ladder is coded. This function multiplies a point with a constant k . To do this, it uses both addition and doubling.

3.2.3 Greatest Common Divisor

The Greatest Common Divisor (GCD) algorithm calculates the greatest common divisor of given integers. The GCD operation is needed in the ECM algorithm [7]. Accordingly, two GCD algorithms are implemented; binary GCD and safegcd. The safegcd algorithm is announced in March 2019 by Daniel J. Bernstein and Bo-Yin Yang in the article named "Fast constant-time GCD computation and modular inversion". It calculates GCD, Bézout's identity, and modular inversion. Also, this is a constant time algorithm. Nevertheless, this algorithm is fast enough for applying ECM [3]. The binary GCD algorithm was included inside the ECM algorithm, and the integration of safegcd to ECM was left as a future work.

3.3 FIWE Software Subsystem Environment

3.3.1 INTEL CPU

Within the scope of this project, to parallelize the ECM algorithm, a CPU is needed besides the GPU. In this project, the Intel Core i7-7700HQ model is used which was announced in January 2017 at CES (7th generation Core). The processor base frequency is 2.80 GHz (4 cores, 8 threads) [12]. Moreover, it is supported by 8GB RAM and it has 6 MB Cache Memory.

3.3.2 NVIDIA GPU

In order to execute the ECM algorithm with high performance, NVIDIA GTX 1050 Ti Graphics Card is used. This graphic card has 4GB GDDR5 memory size, 1290 MHz core speed, and 768 CUDA Cores [6]. These cores can execute the same operation on each core simultaneously due to the SIMD architecture. In more details, NVIDIA GTX 1050 Ti supports CUDA, 3D Vision, PhysX, NVIDIA G-SYNC, Ansel technologies. Therefore, the CUDA programming language is used in this project with NVIDIA GTX 1050 Ti Graphics Card.

3.3.3 CUDA (10.0 - 2015)

In November 2006, NVIDIA introduced CUDA, a general purpose parallel computing platform and programming model that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU [1]. In more details, the developers can program efficiently by using a language similar to C, C++, and Fortran and integrate some basic keywords to run the code on GPU. To implement ECM on GPU, CUDA language is used.

3.3.4 C (C11 - 2011)

C is a general purpose programming language providing economy of expression, modern flow control and data structures, and a rich set of operators. C was originally designed for and implemented on the UNIX operating system on the DEC PDP-11, by Dennis Ritchie [4] and became the most widely used computer programming language. Moreover, CUDA comes with a software environment that allows developers to use C as a high level programming language [1]. Because of that, in this project, C programming language is used.

3.3.5 Python (3.8.1 - 2019)

To initialize the implementation of the software, a high level programming language is considered as a need. For this purpose, Python is chosen, due to its simplicity. In addition, Sage is a Python library, and it allows easy switches between them.

3.3.6 Sage (9.0 - 2020)

Sage is a Python library which has its customized interpreter. At first, it is written for the elliptic curve and modular form operations. Thereafter, it became an open source framework, for number theory, algebra, and geometry [11]. Since it is free and easy to learn, it is chosen as an auxiliary mathematical tool.

3.3.7 Magma (2.25-4 - 2020)

Magma is a computer algebra system for computations in algebra and geometry. Magma deals with several fields of mathematics such as group theory, algebraic number theory, arithmetic geometry, and so on [8]. Because of the complexity of the elliptic curve equations and multi-precision arithmetic operations, the magma calculator is used. It allows easy computation for testing purposes.

3.3.8 GMP (6.2.0 - 2020)

GMP is a multi-precision arithmetic library that does arithmetic operations with the numbers greater than the word size of the computer. GMP is written in C programming language which makes it easier to embed inside the software. It is used for testing purposes throughout the project.

3.3.9 Inline x86-64 Assembly (AT&T)

Inline assembly is a special concept which allows adding assembly code in high level code. Special C macros are used for multi-precision arithmetic in the arithmetic module. Hereby, multi-precision numbers became available to use much more fast in safegcd and ECM.

3.3.10 Nsight (9.2 - 2018)

NVIDIA Nsight Eclipse Edition is a unified integrated development environment for both CPU and GPU to develop CUDA applications on Linux and Mac OS X for the x86, POWER, and ARM platforms [2]. Nsight is a platform that provides an all-in-one integrated environment to edit, build, debug, and profile CUDA-C applications. Nsight is used for coding and debugging in CUDA language.

4 FIWE Software System Detailed Design

Section 4 is divided into subsections which are Main Module and Subsystem Modules. Main Module describes the main of the software where ECM is called and the result is printed. In the Subsystem Modules, the necessary libraries such as Multi-precision, Montgomery, and ECM are explained with their functions and structures. Additionally, CUDA declarations are mentioned in Section 4.5.

4.1 FIWE Main Module

- User enters the multi-precision number as an input.
- ECM gets called for the factorization of the number.
- The return value of ECM gets printed as an output.

4.2 FIWE Subsystem Multi-precision Module

In this module, the necessary arithmetic operations and the safegcd operation on multi-precision integers are implemented. The integers are stored in base 2^{32} since the Pascal architecture has 32-bit registers. However, the word size is defined generic so that it can be adapted to any other architecture.

4.2.1 Multi-precision Module Macros

- W : Word size of the architecture
- `fiweCopy(unsigned int *z, unsigned int startZ, unsigned int endZ, unsigned int *a, unsigned int startA)`
 - Copies multi-precision integer a to z
 - z : The destination of the copy operation
 - $startZ$: Starting index of the destination range
 - $endZ$: Ending index of the destination range
 - a : The source of the copy operation
 - $startA$: Starting index of the source range

4.2.2 Multi-precision Module Functions

- `fiweRand(unsigned int *z, unsigned int l)`
 - Generates random multi-precision integers up to a given size.
 - `z`: The pointer to hold the random integer generated
 - `l`: Number of digits of the random integer in base 2^W
- `fiweModRand(unsigned int *z, unsigned int l, unsigned int *n, unsigned int nl, unsigned int *mu, unsigned int mul)`
 - Generates random multi-precision integers up to given size $\bmod n$.
 - `z`: The pointer to hold the random integer generated
 - `l`: Number of digits of the random integer in base 2^W
 - `n`: Modular base for `z`
 - `nl`: Number of digits of `n` in base 2^W
 - `mu`: Precalculated value of $\lfloor (2^W)^{2 \times nl} / n \rfloor$
 - `mul`: Number of digits of `mu` in base 2^W
- `fiwePrint(unsigned int *a, unsigned int al, char *s)`
 - Prints the multi-precision number in base 10.
 - The format is compatible with Magma assignments.
 - Such as $s := a_0 + a_1 * (2^W)^1 + a_2 * (2^W)^2$ where W .
 - `a`: The number to be printed
 - `al`: Number of digits of `a` in base 2^W
 - `s`: The variable name of the integer
- `fiweAdd(unsigned int *z, unsigned int *a, unsigned int al, unsigned int *b, unsigned int bl)`
 - Adds `a` and `b`.
 - `z`: The pointer to hold the result of the addition
 - `a`: First operand
 - `al`: Number of digits of `a` in base 2^W
 - `b`: Second operand
 - `bl`: Number of digits of `b` in base 2^W
- `fiweModAdd(unsigned int *z, unsigned int *a, unsigned int al, unsigned int *b, unsigned int bl, unsigned int *n, unsigned int nl, unsigned int *mu, unsigned int mul)`
 - Calculates $a + b \pmod n$.
 - `z`: The pointer to hold the result of the addition
 - `a`: First operand
 - `al`: Number of digits of `a` in base 2^W
 - `b`: Second operand
 - `bl`: Number of digits of `b` in base 2^W
 - `n`: Modular base for `z`
 - `nl`: Number of digits of `n` in base 2^W

- *mu*: Precalculated value of $\lfloor (2^W)^{2 \times nl} / n \rfloor$
- *mul*: Number of digits of *mu* in base 2^W
- `fiweSub(unsigned int *z, unsigned int *a, unsigned int al, unsigned int *b, unsigned int bl)`
 - Subtracts *b* from *a*.
 - *z*: The pointer to hold the result of the subtraction
 - *a*: First operand
 - *al*: Number of digits of *a* in base 2^W
 - *b*: Second operand
 - *bl*: Number of digits of *b* in base 2^W
- `fiweModSub(unsigned int *z, unsigned int *a, unsigned int al, unsigned int *b, unsigned int bl, unsigned int *n, unsigned int nl)`
 - Calculates $a - b \pmod{n}$.
 - *z*: The pointer to hold the result of the subtraction
 - *a*: First operand
 - *al*: Number of digits of *a* in base 2^W
 - *b*: Second operand
 - *bl*: Number of digits of *b* in base 2^W
 - *n*: Modular base for *z*
 - *nl*: Number of digits of *n* in base 2^W
- `fiweMul(unsigned int *z, unsigned int *a, unsigned int al, unsigned int *b, unsigned int bl)`
 - Multiplies *a* with *b*.
 - *z*: The pointer to hold the result of the multiplication
 - *a*: First operand
 - *al*: Number of digits of *a* in base 2^W
 - *b*: Second operand
 - *bl*: Number of digits of *b* in base 2^W
- `fiweModMul(unsigned int *z, unsigned int *a, unsigned int al, unsigned int *b, unsigned int bl, unsigned int *n, unsigned int nl, unsigned int *mu, unsigned int mul)`
 - Calculates $a * b \pmod{n}$.
 - *z*: The pointer to hold the result of the multiplication
 - *a*: First operand
 - *al*: Number of digits of *a* in base 2^W
 - *b*: Second operand
 - *bl*: Number of digits of *b* in base 2^W
 - *n*: Modular base for *z*
 - *nl*: Number of digits of *n* in base 2^W
 - *mu*: Precalculated value of $\lfloor (2^W)^{2 \times nl} / n \rfloor$
 - *mul*: Number of digits of *mu* in base 2^W

- `fiweBarretReduction(unsigned int *z, unsigned int *m, unsigned int ml, unsigned int *n, unsigned int nl, unsigned int *mu, unsigned int mul)`
 - Calculates $m \bmod n$.
 - z : The pointer to hold the result of the reduction
 - m : First operand
 - ml : Number of digits of m in base 2^W
 - n : Second operand
 - nl : Number of digits of n in base 2^W
 - mu : Precalculated value of $\lfloor (2^W)^{2 \times nl} / n \rfloor$ where W is the word size
 - mul : Number of digits of mu in base 2^W
- `fiweBinaryGCD(unsigned int *d, unsigned int *a, unsigned int al, unsigned int *n, unsigned int nl)`
 - Calculates $\gcd(a, n)$.
 - d : The pointer to hold the result of the reduction
 - a : First operand
 - al : Number of digits of a in base 2^W
 - n : Second operand
 - nl : Number of digits of n in base 2^W
- `fiweInvert(unsigned int *z, unsigned int *a, unsigned int al, unsigned int *n, unsigned int nl, unsigned int *mu, unsigned int mul)`
 - Calculates $a^{-1} \pmod{n}$.
 - z : The pointer to hold the result of the inversion
 - a : Number to be inverted
 - al : Number of digits of a in base 2^W
 - n : Modular base
 - nl : Number of digits of n in base 2^W
 - mu : Precalculated value of $\lfloor (2^W)^{2 \times nl} / n \rfloor$ where W is the word size
 - mul : Number of digits of mu in base 2^W

4.2.3 Safegcd Functions

- `fiweSafegcd(long *z, long *f, long *g, long *U, long *V, long *Q, long *R, long *precomp, long len)`
 - Calculates GCD of two integers and the modular inverse of one
 - Activity diagram of `xgcd` is shown in Figure 3
 - z : The pointer to hold the modular inverse of f
 - f : First operand and the pointer to hold the result of $\gcd(f, g)$
 - g : Second operand
 - U : A constant to be used in the algorithm
 - V : A constant to be used in the algorithm
 - Q : A constant to be used in the algorithm
 - R : A constant to be used in the algorithm

- *precomp*: Precomputed value of $(\frac{f+1}{2})^{m-1}$
- *len*: Number of digits of the operands in base 2^W

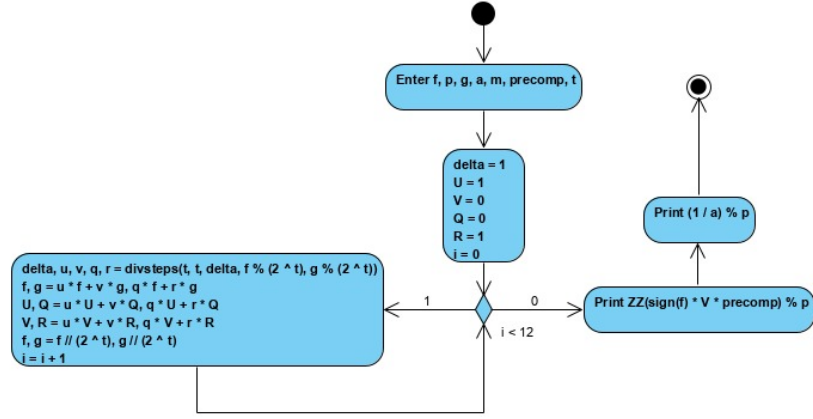


Figure 3: Activity Diagram of fiweSafegcd

- `fiweSafegcdDivsteps(long n, long *δ, long f, long g, long *uu, long *vv, long *qq, long *rr)`
 - Imitates division on δ , f and g
 - Activity diagram of `divsteps` is shown in Figure 4
 - n : Bitwise length of f and g
 - δ : A constant to be used in the algorithm
 - f : Low part of the f in the `safegcd`
 - g : Low part of the g in the `safegcd`
 - uu : Local variables coming from `safegcd`
 - vv : Local variables coming from `safegcd`
 - qq : Local variables coming from `safegcd`
 - rr : Local variables coming from `safegcd`

4.3 FIWE Subsystem Montgomery Module

The point operations on Montgomery curves are implemented in this library. There are 2 structures to represent the points and the curves.

4.3.1 Montgomery Module Structures

- `fiweMontgCurve`
 - unsigned int A : Coefficient A in the curve equation
 - unsigned int B : Coefficient B in the curve equation

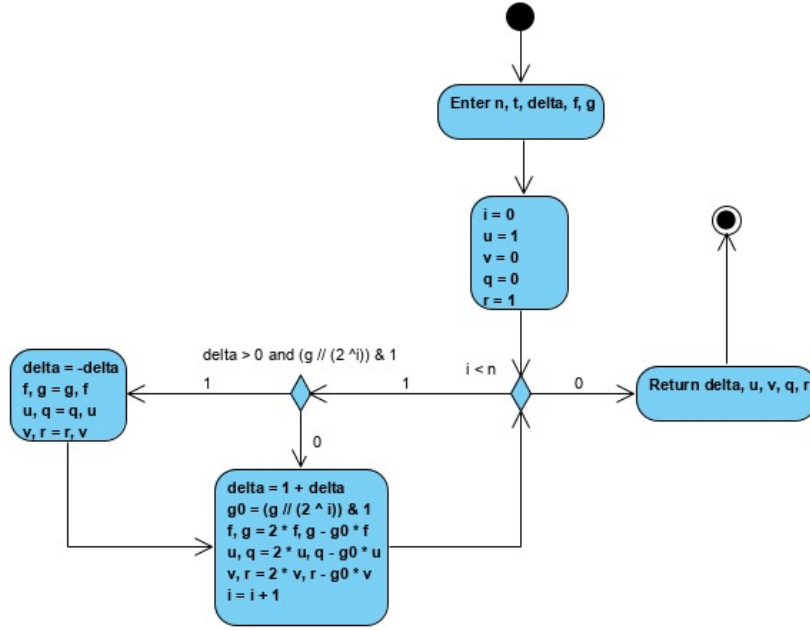


Figure 4: Activity Diagram of fiweSafegcdDivsteps

- unsigned int n : The modular base of the curve
- fiweProPoint
 - unsigned int X : The X coordinate of the point
 - unsigned int Y : The Y coordinate of the point
 - unsigned int Z : The Z coordinate of the point

4.3.2 Montgomery Module Functions

- fiweProCurvePoint(unsigned int $*d$, fiweMontgCurve $*c$, fiweProPoint $*p$, unsigned int $*n$, unsigned int nl , unsigned int $*mu$, unsigned int mul , int $*flag$)
 - Initializes a randomly generated Montgomery curve and a projective point on it.
 - d : Factor of n , that may be found while generating the curve
 - c : Pointer to hold the curve initialized
 - p : Pointer to hold the projective point initialized
 - n : The modular base of the curve
 - nl : Number of digits of n in base 2^W
 - mu : Precalculated value of $\lfloor (2^W)^{2 \times nl} / n \rfloor$
 - mul : Number of digits of mu in base 2^W

- *flag*: 0 when factor found, 1 when curve and point generated, -1 when function failed due to singular curve generation
- `fiweProAdd(fiweProPoint *p, fiweProPoint *p1, fiweProPoint *p2, fiweProPoint *pd, unsigned int *n, unsigned int nl, unsigned int *mu, unsigned int mul)`
 - Adds two projective points on a curve.
 - *p*: Pointer to hold the calculated point
 - *p1*: First operand
 - *p2*: Second operand
 - *pd*: Differences of the first and second operands
 - *n*: The modular base of the curve
 - *nl*: Number of digits of *n* in base 2^W
 - *mu*: Precalculated value of $\lfloor (2^W)^{2 \times nl} / n \rfloor$
 - *mul*: Number of digits of *mu* in base 2^W
- `fiweProDbl(fiweProPoint *p, fiweProPoint *p1, unsigned int *A24, unsigned int *n, unsigned int nl, unsigned int *mu, unsigned int mul)`
 - Doubles a projective point on the curve.
 - *p*: Pointer to hold the calculated point
 - *p1*: The operand
 - *A24*: Precalculated value of $\frac{A+2}{4}$ where *A* is the coefficient of the curve
 - *n*: The modular base of the curve
 - *nl*: Number of digits of *n* in base 2^W
 - *mu*: Precalculated value of $\lfloor (2^W)^{2 \times nl} / n \rfloor$
 - *mul*: Number of digits of *mu* in base 2^W
- `fiweProLadder(fiweProPoint *p, fiweProPoint *p1, unsigned int *A24, unsigned int k, unsigned int *n, unsigned int nl, unsigned int *mu, unsigned int mul)`
 - Multiplies a projective point on the curve with *k*.
 - Activity diagram of Ladder is shown in Figure 5.
 - *p*: Pointer to hold the calculated point
 - *p1*: The operand
 - *A24*: Precalculated value of $\frac{A+2}{4}$ where *A* is the coefficient of the curve
 - *k*: The constant to be multiplied with *p1*
 - *n*: The modular base of the curve
 - *nl*: Number of digits of *n* in base 2^W
 - *mu*: Precalculated value of $\lfloor (2^W)^{2 \times nl} / n \rfloor$
 - *mul*: Number of digits of *mu* in base 2^W

4.4 FIWE Subsystem ECM Module

The ECM algorithm is realized in this library.

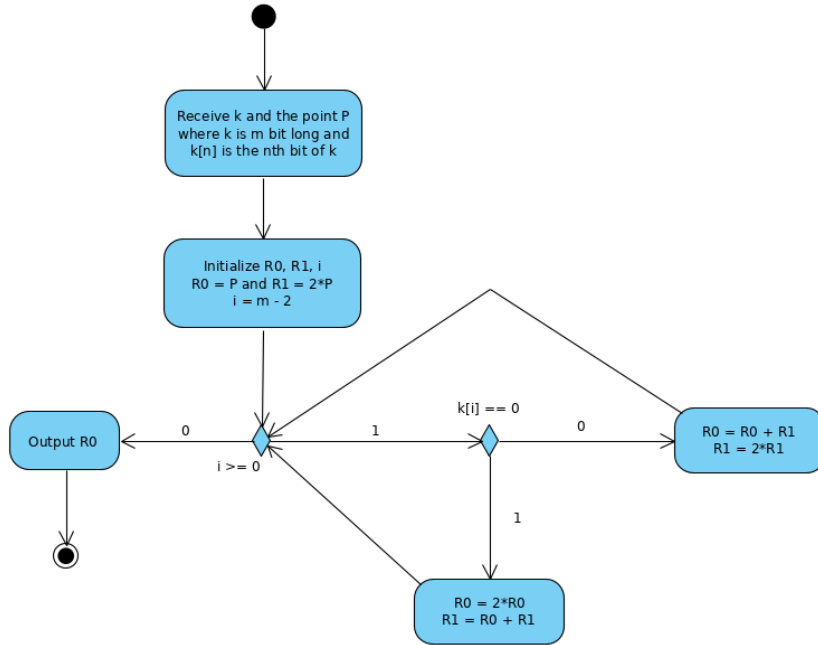


Figure 5: Activity Diagram of fiweProLadder

4.4.1 ECM Module Functions

- `fiwECM(unsigned int *d, unsigned int *n, unsigned int nl)`
 - Factorizes the given composite n using ECM.
 - Activity diagram of ECM is shown in Figure 6.
 - d : Pointer to hold the factor found
 - n : The multi-precision number to be factorized
 - nl : Number of digits of n in base 2^W

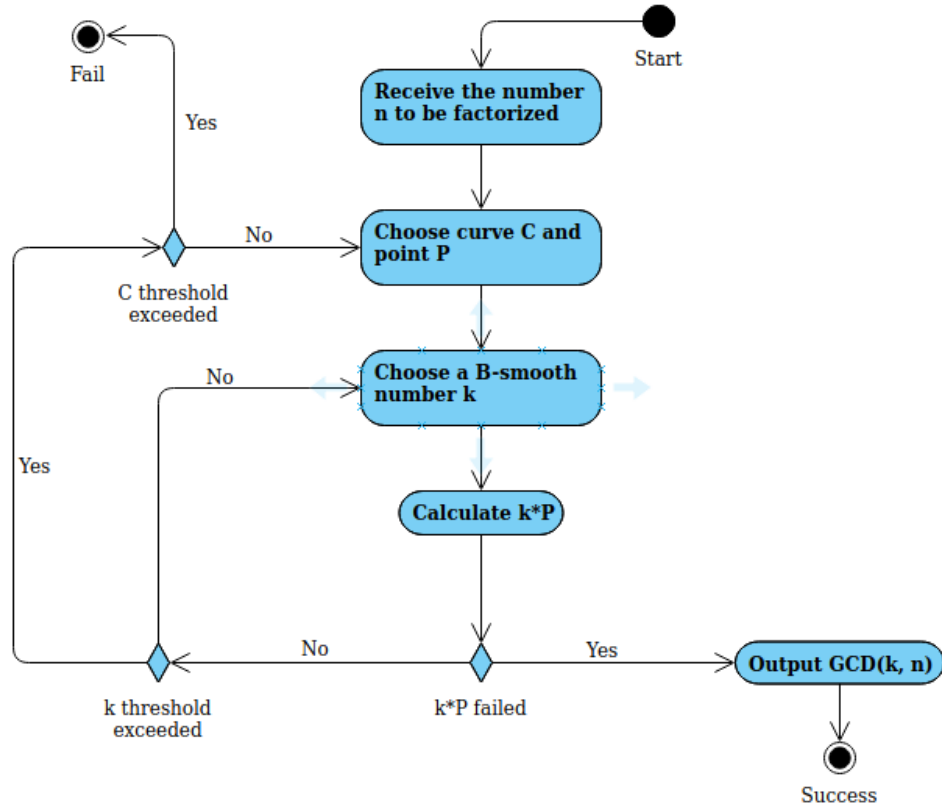


Figure 6: Activity Diagram of the Software

4.5 FIWE Subsystem CUDA Implementation

CUDA implementation requires additional calculations, functions, and parameters which are explained below.

4.5.1 CUDA Module Macros

Device (GPU) Array

Since GPU cannot access CPU arrays directly, a different array for GPU named device array must be defined.

SIZE

Defines how many multi-precision numbers are going to be processed simultaneously.

Block Number and Thread Number

As known, parallelism is done with blocks and threads. Therefore, these variables define how many blocks and threads are going to be used.

Kernel Functions: <<< Block Number, Thread Number >>>

Kernel functions are the functions which are executed on GPU by many threads in parallel.

Thread Index

While working with multiple threads, it is crucial to avoid collisions. In other words, every thread must deal with its own elements only. This can be done by generating a global index for each thread with the predefined variables of CUDA as follows:

$$ThreadIndex = blockIdx.x \times blockDim.x + threadIdx.x$$

4.5.2 CUDA Module Functions**cudaMalloc(void** devPtr, size_t size)**

Memory allocation for device arrays.

cudaMemcpy (void *dst, const void *src, size_t count, cudaMemcpyKind kind)

As the name implies, this function copies either from host array to device array or vice versa. For the first case, when allocating memory with cudaMalloc() is successfully completed, cudaMemcpy() function is used to initialize the device array by copying the elements of the host array. On the other hand, when processing is done with a GPU array, the results must be copied back to the CPU array to be used by CPU. To do this, cudaMemcpy() must be called once more.

cudaMallocManaged (void *dst, size_t size)

Allocates memory for the destination address that can be used by both GPU and CPU.

5 Testing Design

All algorithms and functions are tested to be sure they work correctly. There are two main parts of this project which are ECM and safegcd. In order to test the correctness of the ECM algorithm, a high level calculator was needed. For this purpose, different platforms are reviewed, Magma and GMP found suitable for the operations. Both can be used for arithmetic operations on elliptic curves and multi-precision numbers.

To confirm the correctness of the functions coded, Magma codes are generated which are equivalents of the functions in C. The results of the functions are compared in between Magma and C. Each arithmetic operation is implemented with GMP respectively, to verify the result, and also to find the bug. Moreover, GMP provided the functionality of verifying the operations step by step and debugging the code.

Furthermore, tests for safegcd were also required. Firstly, the results of GCD were needed to be tested with already confirmed mathematical methods. Hence, a function is declared that implements iterative Extended Euclidean Algorithm in C to validate safegcd.

Also, since Bézout's Identity is one of the outputs of the safegcd algorithm, those identities were needed to be tested. This output is tested using Bézout's Identity formula. Additionally, there are subprocedures of safegcd which operate on multi-precision numbers. Those are tested by using Magma Calculator. Finally, modular inversion of safegcd is tested by multiplying the number with its inverse and checking whether it is 1 (mod n) or not.

References

- [1] Cuda c programming guide design guide, 2018.

- [2] Nsight eclipse edition getting started guide, 2019.
- [3] D. J. Bernstein and B.-Y. Yang. Fast constant-time gcd computation and modular inversion. 2019:340–398, May 2019.
- [4] D. M. R. Brian W. Kernighan. *The C Programming Language*. Prentice-Hall, 2011.
- [5] N. Corporation. Nvidia tesla p100, 2016.
- [6] J. Cugnot. Pny technologies europe, 2016.
- [7] A. Durukan, A. Altıparmak, E. Özbay, H. O. Soğukpınar, and N. F. Pala. Requirement specification document, 2019.
- [8] F. Hess. Introduction to magma. unpublished, 2016.
- [9] G. A. Kindervater and J. K. Lenstra. An introduction to parallelism in combinatorial optimization. *Discrete Applied Mathematics*, 14(2):135–156, 1986.
- [10] S. Scargall. *Programming Persistent Memory*. Apress Open, 2020.
- [11] W. Stein and D. Joyner. SAGE: System for Algebra and Geometry Experimentation. 39(2):61–64, June 2005.
- [12] R. Szczepanski, T. Tarczewski, and L. M. Grzesiak. Parallel computing applied to auto-tuning of state feedback speed controller for pmsm drive, 2019.

APPENDIX C: PRODUCT MANUAL

COMP4920 Senior Design Project II, Spring 2020
Advisor: Dr. Hüseyin Hışıl

FIWE: Factoring Integers with ECM Product Manual

Revision 2.0
24.05.2020

By:
Asena Durukan, Student ID: 16070001016
Aslı Altıparmak, Student ID: 15070001003
Elif Özbay, Student ID: Ç15070002011
Hasan Ozan Soğukpınar, Student ID: 17070001047
Nuri Furkan Pala, Student ID: 15070006030

Revision History

Revision	Date	Explanation
1.0	13.04.2020	Initial Product Manual
2.0	24.05.2020	New functions are added GUI and website implementations are discarded Existing C functions are removed from CUDA functions in Section 2

Table of Contents

Revision History	2
Table of Contents	3
1 Introduction	4
2 FIWE Software Subsystem Implementation	4
2.1 Source Code and Executable Organization	4
2.1.1 C	4
2.1.2 CUDA	6
2.2 Software Development Tools	6
2.2.1 Compilers	6
2.2.2 Development Environments	6
2.2.3 Testing	7
2.2.4 Version Control	7
2.2.5 Web Services	7
2.3 Hardware and Software System Platform	7
2.3.1 NVIDIA GPU	7
2.3.2 INTEL CPU	8
2.3.3 Ubuntu 18.04	8
3 FIWE Software Subsystem Testing	8
4 FIWE Installation, Configuration and Operation	9
4.1 Installation	10
4.2 Configuration	10
4.3 Operation	11
4.3.1 Source Code	11
4.3.2 Bash Script	12
References	12

1 Introduction

This document aims to briefly explain the FIWE Software in terms of installation, setup, usage, and testing.

In Section 2, how the source code structured and organized, which specialized software tools are used in the development process and what kind of platforms the development process operated on are mentioned. Section 3, describes the testing procedures for software subsystems. Finally, how one can install the software for use, arrange it according to her/his needs, and operate it on a device, are mentioned in Section 4.

For additional information; the requirement specifications are defined in the RSD [1], the design specifications, the testing methods and their results are explained in the DSD [2].

2 FIWE Software Subsystem Implementation

2.1 Source Code and Executable Organization

In the final version of this project, all source codes are written in both C and CUDA language. Hence, source codes are divided into different files/parts.

2.1.1 C

main.c

- `fiweBash(int argc, char *argv[]);`

mplib.c

mplib.h

- `#define W;`
- `#define fiweCopy(z, startZ, endZ, a, startA);`
- `void fiweRand(unsigned int *z, unsigned int l);`
- `void fiweModRand(unsigned int *z, unsigned int l, unsigned int *n, unsigned int nl, unsigned int *mu, unsigned int mul);`
- `void fiwePrint(unsigned int *a, unsigned int al, char *s);`
- `void fiweAdd(unsigned int *z, unsigned int *a, unsigned int al, unsigned int *b, unsigned int bl);`
- `void fiweModAdd(unsigned int *z, unsigned int *a, unsigned int al, unsigned int *b, unsigned int bl, unsigned int *n, unsigned int nl, unsigned int *mu, unsigned int mul);`
- `void fiweSub(unsigned int *z, unsigned int *a, unsigned int al, unsigned int *b, unsigned int bl);`
- `void fiweModSub(unsigned int *z, unsigned int *a, unsigned int al, unsigned int *b, unsigned int bl, unsigned int *n, unsigned int nl);`
- `void fiweMul(unsigned int *z, unsigned int *a, unsigned int al, unsigned int *b, unsigned int bl);`
- `void fiweModMul(unsigned int *z, unsigned int *a, unsigned int al, unsigned int *b, unsigned int bl, unsigned int *n, unsigned int nl, unsigned int *mu, unsigned int mul);`
- `void fiweBarretReduction(unsigned int *z, unsigned int *m, unsigned int ml, unsigned int *n, unsigned int nl, unsigned int *mu, unsigned int mul);`
- `void fiweBinaryGCD(unsigned int *d, unsigned int *a, unsigned int al, unsigned int *n, unsigned int nl);`

- void fiweInvert(unsigned int *z, unsigned int *a, unsigned int al, unsigned int *n, unsigned int nl, unsigned int *mu, unsigned int mul);
- void fiweIsEqual(unsigned int *z, unsigned int *a, unsigned int *b, unsigned int l);
- void fiweIsEqualUi(unsigned int *z, unsigned int *a, unsigned int al, unsigned int *b);
- void fiweGetMu(unsigned int *z, unsigned int *n, unsigned int nl);
- void fiweGetA24(unsigned int *A24, unsigned int *A, unsigned int *n, unsigned int nl, unsigned int *mu, unsigned int mul, int *flag);

safegcd.c

safegcd.h

- #define fiweSafegcdUFVGC(u, f, v, g, c, h, l);
- void fiweSafegcdPrint(FILE *file, char *str, long *a, long len);
- void fiweSafegcdMul(long *zn, long u, long *fn, long v, long *gn, long len);
- void fiweSafegcdDivSteps(long n, long t, long *δ, long f, long g, long *uu, long *vv, long *qq, long *rr);
- void fiweSafegcd(long *z, long *f, long *g, long *U, long *V, long *Q, long *R, long *precomp, long len)

montgomery.c

montgomery.h

- struct fiweMontgCurve;
- struct fiweProPoint;
- void fiweProCurvePoint(unsigned int *d, fiweMontgCurve *c, fiweProPoint *p, unsigned int *n, unsigned int nl, unsigned int *mu, unsigned int mul, int *flag);
- void fiweProAdd(fiweProPoint *p, fiweProPoint *p1, fiweProPoint *p2, fiweProPoint *pd, unsigned int *n, unsigned int nl, unsigned int *mu, unsigned int mul);
- void fiweProDbl(fiweProPoint *p, fiweProPoint *p1, unsigned int *A24, unsigned int *n, unsigned int nl, unsigned int *mu, unsigned int mul);
- void fiweProLadder(fiweProPoint *p, fiweProPoint *p1, unsigned int *A24, unsigned int k, unsigned int *n, unsigned int nl, unsigned int *mu, unsigned int mul);

ecm.c

ecm.h

- void fiweECM(unsigned int *d, unsigned int *n, unsigned int nl);

test.c

test.h

- void fiweProCurvePointGMPTTest(int THRESHOLD);
- void fiweProAddGMPTTest(int THRESHOLD);
- void fiweProAddMagmaTest(int THRESHOLD);
- void fiweProDblMagmaTest(int THRESHOLD);
- void fiweProLadderGMPTTest(int THRESHOLD);
- void fiweProLadderMagmaTest(int THRESHOLD);
- void fiweECMGMPTest(int THRESHOLD);
- void fiweSafegcdMulPyTest(int THRESHOLD);

2.1.2 CUDA

All functions listed in C section are also implemented in CUDA programming language. The additional functions and macros for CUDA are listed below.

main.cu

montgomery.cu

montgomery.h

mplib.cu

mplib.h

- `#define __sub_cc(r,a,b) ASM ("sub.cc.u32")`
- `#define __subc_cc(r,a,b) ASM ("subc.cc.u32")`
- `#define __addcy(carry) ASM ("addc.u32")`
- `#define __addcy2(carry) ASM ("addc.cc.u32")`
- `#define __subcy(carry) ASM ("subc.u32")`
- `#define __mul_lo(r,a,b) ASM("mul.lo.u32")`
- `#define __mul_hi(r,a,b) ASM("mul.hi.u32")`
- `__device__ float fiweGPUGenerate (curandState* globalState);`

test.cu

test.h

ecm.cu

ecm.h

2.2 Software Development Tools

In this section, various software tools, compilers, and development environments which are used in this project are described.

2.2.1 Compilers

GCC (7.5.0 - 2019)

At first, GNU Compiler Collections (GCC) is developed by Richard Stallman for his GNU project. Nowadays, it is used to compile major languages such as C, C++ and many others. In this project, the functions are written in C language before they are implemented on CUDA. Therefore, GCC is used as a compiler of C programming language.

NVCC - (CUDA 10.0 2015)

NVCC stands for NVIDIA CUDA Compiler which is developed by NVIDIA in order to compile CUDA codes. Since CUDA runs both on the GPU and CPU, the code must be splitted with using a compiler. NVCC seperates the GPU and CPU codes, and then it compiles device codes and sends host codes to a C compiler such as GCC.

2.2.2 Development Environments

Nsight (9.2 - 2018)

NVIDIA Nsight is a CUDA development environment for numerous platforms of GPU computing. In this project, Nsight, integrated with Eclipse IDE on Linux operating system, is used. In addition, Nsight provides a debugger on GPU which makes parallel coding much easier.

Visual Studio Code (1.44.0 - 2015)

Visual Studio Code is a code editor for many languages such as C, C++ and Python. It is also a cross-platform editor which means it is available for Windows, macOS and Linux. As mentioned, before the CUDA implementation of the project, all functions are written in C programming language. This step is done on Visual Studio Code.

2.2.3 Testing

GMP (6.2.0 - 2020)

GMP is a multi-precision arithmetic library written in C programming language. It allows fast computation with the numbers that are bigger than the word size of personal computers. It is designed to be fast by optimizing the code according to the hardware architectures, implementing well-designed algorithms, using assembly for efficiency and so on [3]. Since GMP is fast and easy to implement on C codes, it is used for testing purposes throughout the project.

Magma Calculator (2.25-4 - 2020)

Magma is a programming language designed for the examination of algebraic, geometric and combinatorial structures. The syntax of Magma is similar to other well-known programming languages [4]. Due to the complexity of the elliptic curve and multi-precision arithmetic operations, Magma calculator is used in this project. It allows easy computation for testing purposes [2].

Python (3.8.1 - 2019)

Python is a high level programming language with having an exceptionally fast edit-test-debug cycle. Since it has easy to build data structures and dynamic semantics, it is one of the best platforms for testing purposes. Thus, it is used for the testing of GCD functions in this project.

2.2.4 Version Control

Git (2.17.1 - 2018)

Git is an open-source version control system that allows project members to trace the development process of a system. Repositories are created in this project for both the software development and documentation purposes. Those repositories are hosted on GitHub which is explained below.

2.2.5 Web Services

GitHub

GitHub is the most common hosting service, which released in February 2008, for developers to share their source codes. It also provides interfaces for computer and mobile users. GitHub is used by all project members for sharing their source codes to synchronize the project.

Dropbox

Dropbox is another hosting service which founded in 2007. It provides many features such as cloud storage, file synchronization and so on. In this project, it is used for file sharing across the project members.

2.3 Hardware and Software System Platform

2.3.1 NVIDIA GPU

Within the scope of this project, a GPU was required to execute the ECM algorithm in parallel. NVIDIA GTX 1050 Ti is used to met this need. It has 768 CUDA Cores which are necessary to be used in order to run the algorithm on SIMD architecture. Furthermore, NVIDIA GTX 1050 Ti supports many technologies such as CUDA programming language. Thus, it provides the usage of CUDA language to parallelize the ECM algorithm on the graphic card.

2.3.2 INTEL CPU

Considering that CUDA language works both on CPU and GPU, a CPU was also needed. For instance, a lot of functions which are C programming language related, can only be used on CPU. In this project, Intel Core i7-7700HQ model is used.

2.3.3 Ubuntu 18.04

Ubuntu is an open source complete Linux operating system based on Debian which is appropriate for both desktop and server use. When compared with other operating systems, Ubuntu is significantly more flexible to run the generated codes on GPU. For this reason, Ubuntu is used as an operating system.

3 FIWE Software Subsystem Testing

This section describes the testing procedures for the software system. It includes the step-by-step explanations of the test cases, their outcomes and the total time spent for each test case.

For all functions in the montgomery and ECM libraries, test cases are generated by the help of GMP and Magma. For test cases using GMP, after the test case is run and completed, it printouts the number of successes and fails. For test cases using Magma, test case printouts the calculations to a file in Magma syntax, then from Magma, this file is read and number of successes & fails are written into another file. All tests are applied to C functions. The crucial ones are tested in CUDA also.

The generated test cases are explained below.

- Montgomery Curve and Projective Point Generation Function GMP Test
 1. Generates a random curve and a projective point using the function
 2. Checks if the coefficients of the curve and the point satisfies the curve equation using GMP
- Projective Addition Function GMP Test
 1. Generates two random points
 2. Computes the addition by implementing the algebraic calculations using GMP
 3. Compares the result of the function with the GMP result
- Projective Addition Function Magma Test
 1. Generates two random points
 2. Computes the addition by implementing the algebraic calculations using Magma
 3. Compares the result of the function with the Magma result
- Projective Doubling Function Magma Test
 1. Generates a random point
 2. Computes the double by implementing the algebraic calculations using Magma
 3. Compares the result of the function with the Magma result
- Projective Ladder Function GMP Test
 1. Generates a curve c and a projective point $p1$ on it
 2. Generates random multipliers k and l
 3. Multiplies the point with first k and then l such that $p3 = l * (k * p1)$
 4. Multiplies the point with first l and then k such that $p5 = k * (l * p1)$

5. Calculates $p3 \rightarrow X * p5 \rightarrow Z$ using GMP
 6. Calculates $p5 \rightarrow X * p3 \rightarrow Z$ using GMP
 7. Compares the results of step 5 and 6 using GMP
- Projective Ladder Function Magma Test
 1. Generates a curve c and a projective point $p1$ on it
 2. Generates random multipliers k and l
 3. Multiplies the point with first k and then l such that $p3 = l * (k * p1)$
 4. Multiplies the point with first l and then k such that $p5 = k * (l * p1)$
 5. Calculates $p3 \rightarrow X * p5 \rightarrow Z$ using Magma
 6. Calculates $p5 \rightarrow X * p3 \rightarrow Z$ using Magma
 7. Compares the results of step 5 and 6 using Magma
 - fiweGCDMul Function Python3 Test
 1. Generates three 64 bits, two multi-precision numbers
 2. Computes $u \times f + v \times g + c$
 3. Compares the result of the function with the Python3 result
 - fiweECM Function GMP Test
 1. Generates a random composite number
 2. Calculates a factor of the number using the function
 3. Checks whether the factor found actually divides the composite number

Table 1: Test Case Statistics

Function Name	Is Tested?	Size of Inputs	# of Trial	# of Success	# of Fail	Time Spent in C (sec)
proCurvePointGMP	Yes	1-100	10000	10000	0	15
proAddGMP	Yes	1-100	10000	10000	0	12
proAddMagma	Yes	1-100	10000	10000	0	45
proDblMagma	Yes	1-100	10000	10000	0	33
proLadderGMP	Yes	1-100	10000	10000	0	27
proLadderMagma	Yes	1-100	10000	10000	0	143
safegcdPyMul	Yes	1-100	10000	10000	0	37
ecmGmpTest	Yes	1-10	1000	1000	0	34

As seen in Table 1, all the functions are tested and validated which verifies that the software works as planned.

4 FIWE Installation, Configuration and Operation

There are installation and operation processes for the FIWE project as all qualified software projects. Users can use FIWE in two different ways which are listed below.

- Source code as a library
- Bash script on Unix-like terminal

Installation is needed to use the Bash script.

4.1 Installation

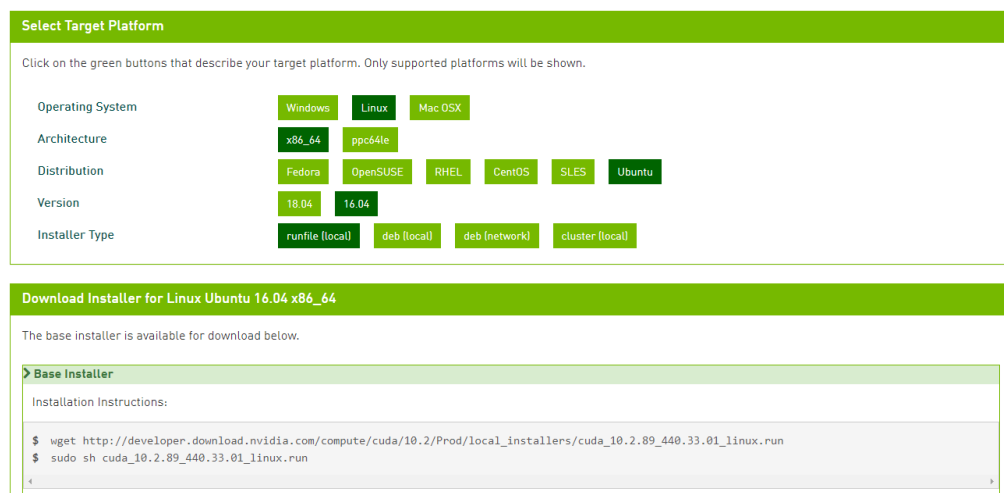
There is an executable file to setup the Bash script. This executable file is programmed for Unix-like operating systems. When the executable is run, fiwecm program gets callable from any directory.

System Requirements

The requirements for FIWE to work are listed below.

- CUDA-capable GPU
- Linux Environment
- GCC compiler
- NVIDIA CUDA Toolkit
 - The user can easily install the Toolkit from the NVIDIA website (<https://developer.nvidia.com/cuda-10.0-download-archive>) as shown in Figure 1. The CUDA version used in this project is 10.0.

Figure 1: Cuda Download



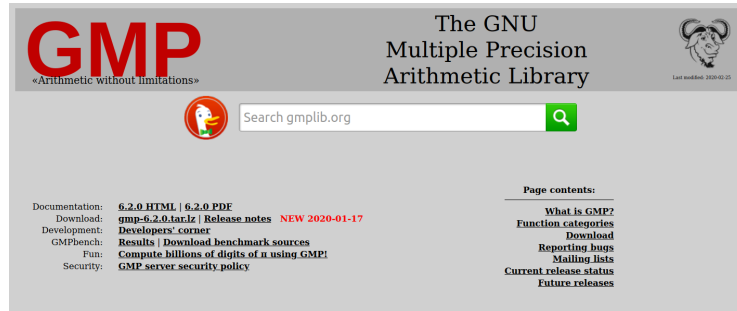
- GMP library
 - The user should install and include the GMP library to run the test cases. Since one of the operations which requires division was out of scope for this project, it was handled by GMP. For this reason, the user must download the GMP library from the official website <https://gmplib.org/> shown in Figure 2.

4.2 Configuration

A CUDA code can be compiled using NVCC. Therefore, the user must switch from GCC to NVCC with using the following terminal command.

```
$ sudo prime-select nvidia
```


Figure 2: GMP Download



4.3 Operation

Since there are two different ways to use this software, operations for each is different. Simply, user enters the number to be factorized, and the factors get printed out. In case of any unexpected behaviour which the software cannot handle, error messages are displayed.

4.3.1 Source Code

The source code of FIWE is written in the C programming language. So, the ECM function and other auxiliary functions can be called by including the library as below.

```
1 #include <fiwe.h>
```

After including this library, the ECM function gets callable. There are three parameters of this function which are; the pointer for the factor found, the pointer of the number to be factorized, the size of the number in base 2^W where W is the word size of the architecture. The multi-precision numbers are represented as *unsigned int* arrays, so the parameter types are; *unsigned int **, *unsigned int ** and *unsigned int* respectively. The first parameter for the factor acts as an output for the function. An example of the usage of the ECM function is given below.

```
1 unsigned int number[3];
2 unsigned int factor[3];
3 number = {<digit1>, <digit2>, <digit3>};
4 ecm(factor, number, 3);
```

To factorize multiple multi-precision numbers simultaneously, the numbers should be given as a flat array. A flat array is constructed by adding multiple numbers sequentially into an array. An example of factorizing three 3-digit numbers is given below.

```
1 unsigned int numbers[9];
2 unsigned int factors[9];
3 numbers = {<num1digit1>, <num1digit2>, <num1digit3>, <num2digit1>, <num2digit2>, <num2digit3>, <
4 num3digit1>, <num3digit2>, <num3digit3>};
5 ecm(factors, numbers, 3);
```

To compile the code on CPU, user must go to the ECM folder and use the compile command below.

```
$ gcc main.c -o main.o ecm.c montgomery.c mplib.c test.c -lgmp
```

To compile the code on GPU, user must go to the CUDA folder and use the compile command below.

```
$ nvcc main.cu -o main.o ecm.cu montgomery.cu mplib.cu test.cu -lgmp
```

4.3.2 Bash Script

This software can be run from the Bash terminal in any directory. The usage of the script with all options are given in Table 2.

Table 2: Commands & Explanations

Command	Explanation
<i>fiwecm - -help</i>	Shows the manual
<i>fiwecm <number></i>	Printouts a factor of the number
<i>fiwecm -f <in.txt></i>	Printouts the factors of the numbers inside "in.txt"
<i>fiwecm <number> -o <out.txt></i>	Writes a factor of the number to "out.txt"
<i>fiwecm -f <in.txt> -o <out.txt></i>	Writes the factors of the numbers inside "in.txt" to "out.txt"

References

- [1] A. Durukan, A. Altıparmak, E. Özbay, H. O. Soğukpınar, and N. F. Pala. Requirement specification document, 2019.
- [2] A. Durukan, A. Altıparmak, E. Özbay, H. O. Soğukpınar, and N. F. Pala. Desing specification document, 2020.
- [3] T. Granlund and G. D. Team. *GNU MP 6.0 Multiple Precision Arithmetic Library*. Samurai Media Limited, London, GBR, 2015.
- [4] C. P. John J. Cannon. *An Introduction to Algebraic Programming with Magma*. 2001.