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

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].

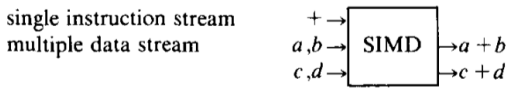

Figure 1: SIMD Architecture [9]

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 detail, 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 detail, 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 faster 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 $\mod n$.
  - 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 $m \mod n$.
  - 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

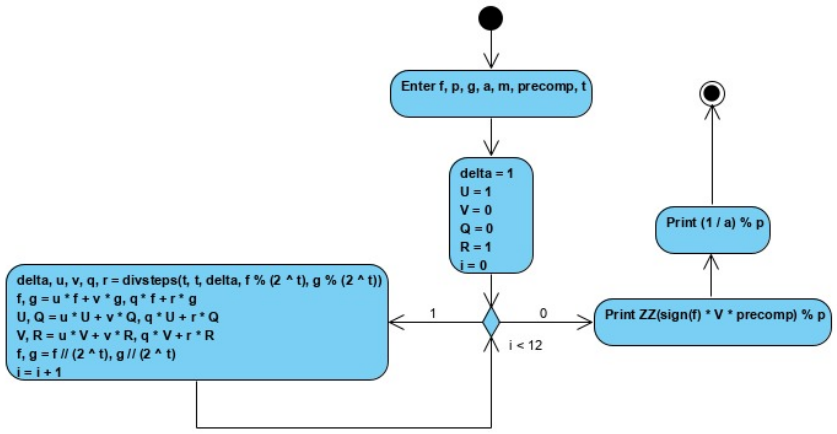

Figure 3: Activity Diagram of fiweSafegcd

- 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

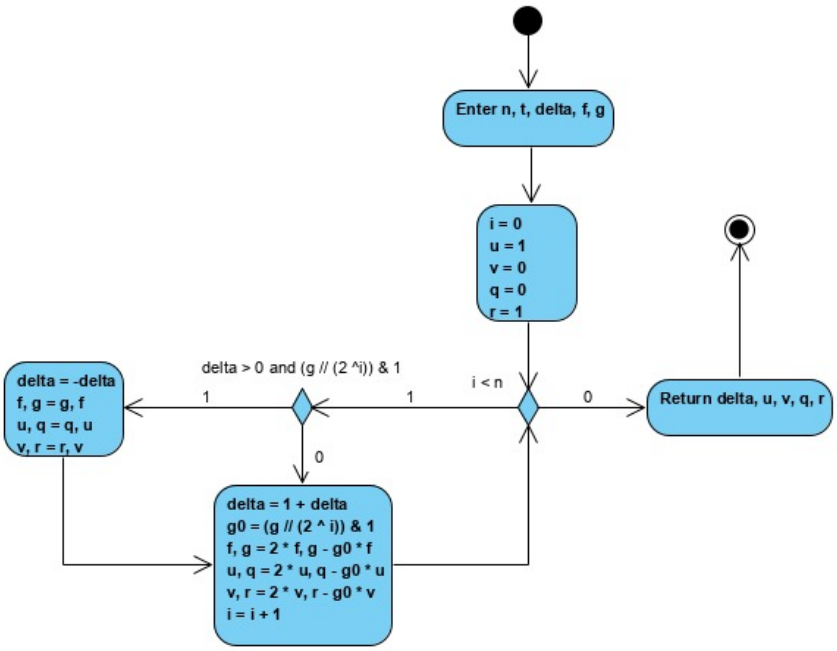

Figure 4: Activity Diagram of fiweSafegcdDivsteps

  - 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.

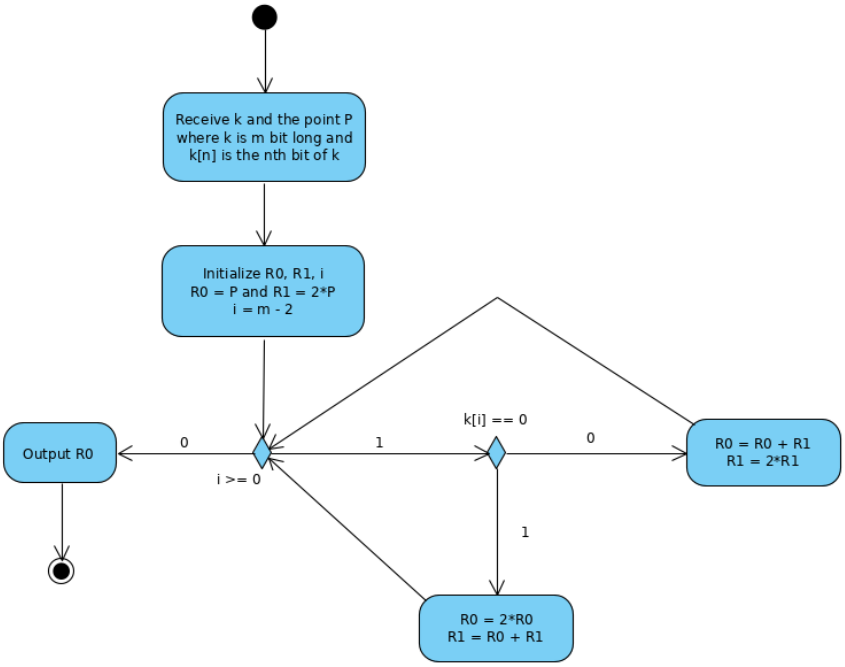

Figure 5: Activity Diagram of fiweProLadder

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

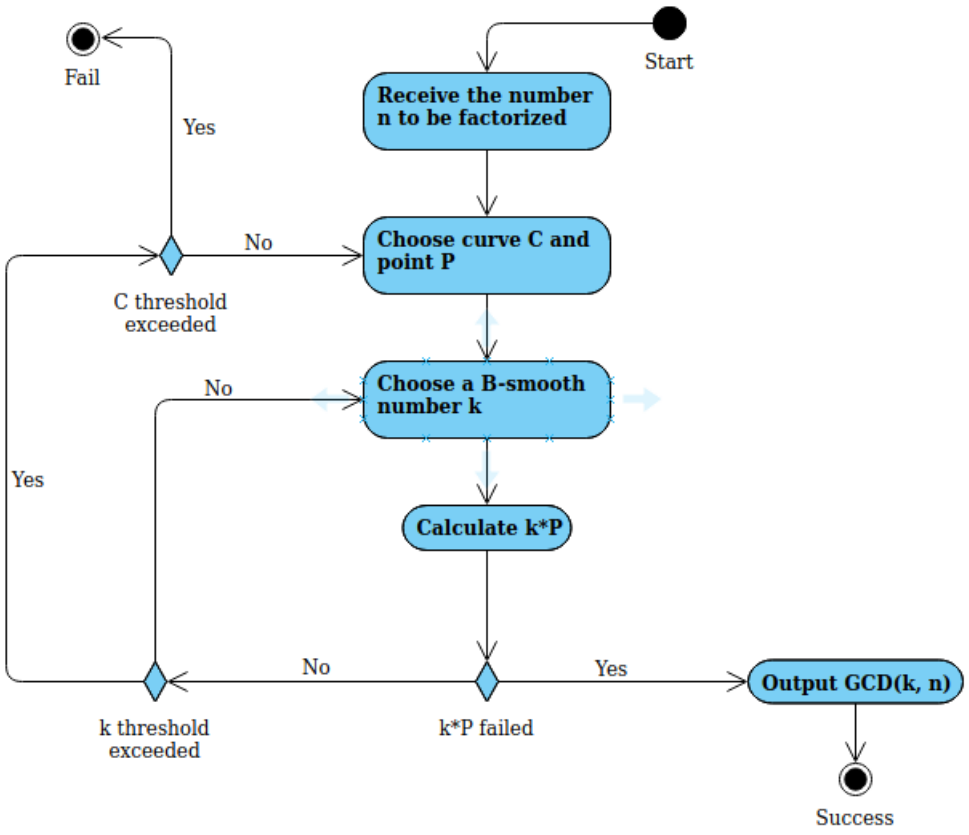

Figure 6: Activity Diagram of the Software

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 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 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.