Optimization and Implementation of various algorithms to speed up scalar multiplication over Elliptic Curves

Prathamesh Dhake 190070048
Atharva Raut 190070050
Jaideep Chawla 190110030

May 3, 2023

1 Abstract

The project aims to obtain speedup on scalar multiplication algorithms on Elliptic Curves and benchmark the existing algorithms to get a relstive idea of the performance metrics. We test our implementations on state-of-the-art NIST P Curves which are a family of Weisstrass Elliptic Curves such as P-192, P-224, P-256 among others. We have successfully concluded the benchmarking of certain basic algorithms using Python's native int class and automating the tests to obtain an idea of the performances. We are currently implementing our custom data type which will be compatible with various paralelization techniques.

2 Introduction

Elliptic Curves form an important basis for existing encryption algorithms. It generates security between key pairs for public key encryption by using the mathematics of elliptic curves. All elliptic curve cryptography is based on the difficulty of finding a secret integer n given the scalar multiple $Q = [n]P = P + \cdots + P$ of a base point P on an elliptic curve, say E/k where k is the prime field.

An example would be the curve **Curve25519** represented as $y^2 = x^3 + 486662x^2 + x$ where k is the prime field GF($2^{255} - 19$)

Scalar multiplication is an essential building block for public-key elliptic curve cryptography and has a significant influence on the execution time of ECC algorithms. Therefore, optimized scalar multiplication methods are vital for good ECC performance.

3 Outline

The goal of this project is to optimize an elliptic curve-based scalar multiplication algorithm using an appropriate parallelization strategy. Therefore, we first conducted a thorough literature review to gain an understanding of the existing algorithms. However, our analysis revealed that none of the existing implementations could be optimized using numpy and compyle. They either utilized Python's native int class or utilized class based structures which were incompatible with numpy and compyle python libraries for speedup. This led us to conclude that developing our own custom implementation was necessary to achieve the desired speedup.

So, we started by designing a custom data type to store and manipulate the necessary values efficiently. The data type was to be one which would be compatible with numba and compyle, thus, consist of numpy arrays. Once we settled upon the implementation of the custom data type, we proceeded to implement some basic algorithms required for elliptic curve calculations as stated in [2]. We took a modular approach, implementing each algorithm as a separate function, and testing each one thoroughly to ensure correctness. We also timed each function on on a curve once to get a general idea of the time taken by various algorithms and to get an estimate of their relative performance. To automate these tests over the 5 Weistrass NIST P Curves in question and the various algorithms, we used the automan [1] package to automate these runs and produce various time graphs.'

After implementing the custom data type, we proceeded to implement basic arithmetic functions for the same including division, multiplicative inverse which proved to very challenging. Currently, we are working upon perfecting the implementation of python's *divmod* function to be compatible with our custom data type, which will help us implement multiplicative inverse which will prove to be very beneficial for us to use our custom data type for the implementation of scalar multiplication algorithms, and thereby measuring speedup.

4 Methodologies

4.1 LongNum

We have implemented a custom data type to store our numbers which ranged from a size of 192 bits (P-192 curve) to 521 bit (P-521 curve). The custom data type is in the form of a numpy array of size 32 wherein each element is a 32-bit integer of the data type *numpy.uint32*.

The implementation and the required functions are present in the *arithmetic* folder hosted on GitHub repository of this project. An example of the same can be seen as,

$bin(999904040455042) = '0b11100011010110100001001101001001000000101110000010'$

Dividing in chunks of 32 bits we get,
$'0b10011010010010000001011100000010'$, and $'0b111000110101101000'$

Which get therefore stored as, 1294207874, and 232808.

We went for this custom 32-bit data type since it is built using numpy arrays and can therefore be sped up using numba and compyle. Furthermore, operations like multiplication can benefit from parallelization.

We proceed to write custom functions for addition, subtraction, multiplication, and division for this data type. As seen in the notebooks in the repository, we see that our custom data type operations such as *addition* and *Subtraction* are about **5 times slower** and *Multiplication* is about **10 times slower**.

The divmod algorithm was implemented, but was relatively very slow as compared to the native python implementation. To resolve this, we decided to use scalene to profile the code and hypothesize on potential methods to improve the performance of this function.

4.2 Multiplicative Inverse

For computing multiplicative inverse of int in python, we benchmarked algorithms, one being a basic implementation, one based on Extended Euclidean Algorithm and Fermat's Algorithm, (Fermat's Little Theorem, the relative time complexity for a inverse in field p was found to be in the order of,

Algorithm	Complexity
Trivial	O(p)
Extended Euclidean Algorithm	O(log p)
Fermat's Little Theorem	O(log p)

After implementation and benchmarking we find that **Extended Euclidean Algorithm** is the fastest out of the three and we proceed to use the same for inverse calculation in the simple algorithm implementations.

4.3 Automation for benchmarking

We proceed to implement three basic algorithms for scalar multiplication, *Double and Add*, *Montgomery's Ladder* and *Joye's Double and Add* algorithm, and write a recursive implementation for Double and Add algorithm, we implement the algorithms as illustrated in the reference[2]. We test these algorithms initially on a basic elliptic curve namely

$$y^2 \ = \ x^3 \ + \ 2x + \ 2 \tag{1}$$

Using the generator point (10, 6) over the Galois Field GF(17). For testing on higher order curves, we chose standard NIST P Curves which are of the form $y^2 \ = \ x^3 \ + \ ax + \ b$, in the field of $GF(2^p - 1)$. We used the curves, *P-192*, *P-224*, *P-256* and *P-384*, the curve parameters were readily available in a JSON format here[3]

We used json module to read the respective json ile to extract the curve parameters and the generator points to perform point arithmetic on them to benchmark the algorithms across a basic curve described above and the varous NIST P Curves. We used *automan* package to automate the tests over approximately 80 test cases where in the program would calculate the time taken to calculate the multiples of the point P, on a curve *curve* using the algorithm *algo* from P upto kP for a given scalar k , *scalar*.

The final graphs thus obtained were,

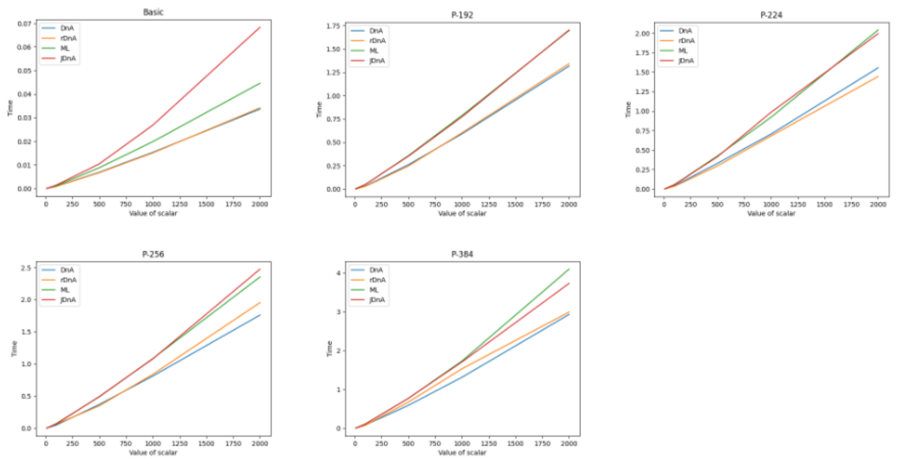

Figure 1: Time taken by various algorithms over various curves

For comparing the time taken across various curve and algorithms, we have the following graph:

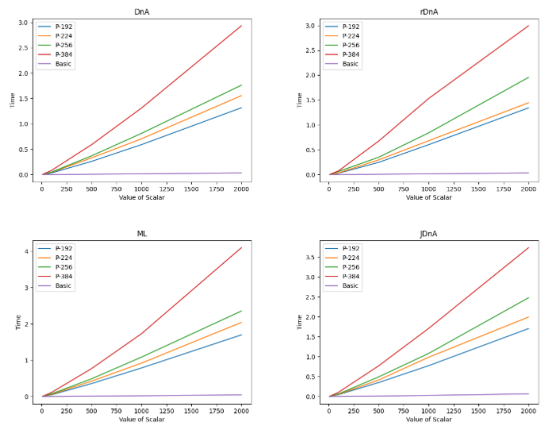

Figure 2: Time taken by various curves over various algorithms

Thus we see that the time taken by all the algorithms is larger as the number of bits in the elliptic curve field increases.

5 Future work and scope for improvement

Our original plan was to develop a complete library for performing big number computations that could be used with numba or compyle. We started our work in two parallel streams - benchmarking algorithms and implementing the data structure to support these operations without using python objects. However, in the interest of time, we could not integrate the two streams. This would be the immediate next steps towards continuing this project.

Even after optimizations, we were still a little far from the native python performance. This could be considered as a potential future direction, especially for things like partial product calculation which can be easily parallelized.

After the first stage of custom data type based algorithm implementations have been implemented, we plan to move onto more complicated and detailed algorithms such as the one which utilize Jacobian coordinates suh as Co-XY and Co-Z systems or Sakai-Sakurai Method for Direct Doubling algorithms. .

References

[1] Prabhu Ramachandran, "automan: A Python-Based Automation Framework for Numerical Computing," in Computing in Science I& Engineering, vol. 20, no. 5, pp. 81-97, 2018. doi:10.1109/MCSE.2018.05329818

[2] Gerwin Gsenger. (Year 2014). *Speeding Up Elliptic Curve Cryptography by Optimizing Scalar Multiplication in Software Implementations*. Unpublished master's thesis, Graz University of Technology

[3] P-192. Accessed May 3, 2023. https://neuromancer.sk/std/nist/P-192.