Edwards curves and FFT-based multiplication

Pavel Atnashev, George Woltman
patnashev@gmail.com, woltman@alum.mit.edu

August 15, 2021

Abstract This paper introduces fast algorithms for performing group operations on Edwards curves using FFT-based multiplication. Previously known algorithms can use such multiplication too, but better results can be achieved if particular properties of FFT-based arithmetic are accounted for. The introduced algorithms perform operations in extended Edwards coordinates and in Montgomery single coordinate.

1 Introduction

Edwards curves [1] and twisted Edwards curves [2] are families of elliptic curves with the fastest known group operations. They've become the elliptic curves of choice in many applications, particularly in cryptography. FFT-based multiplication is a general term for a range of algorithms performing multiplication of two numbers and, in some cases, fast modulo operation [3]. Despite the theoretical advantage of FFT-based algorithms, they outperform classical multiplication methods at number sizes way too big to be of interest for cryptography or any other real-world application. Still, they find their use in number theory and experimental mathematics.

One can find many formulas related to Edwards curves in the literature. Most of the papers consider multiplication and squaring as atomic operations and express the cost of an algorithm as the number of multiplications and squarings. But FFT-based multiplication is a complex multi-stage process, some stages of which are independent of others and can be reused. Some operations can be stretched through the whole algorithm, making it hard to estimate how many multiplications are happening there. Different metrics and different algorithms are needed to get the best performance of FFT-based calculations on modern computers.

In this paper we introduce optimized algorithms to perform group operations on Edwards curves modulo numbers from 1 000 to 100 000 000 digits long. For smaller numbers computational complexity is minimized. For larger numbers an additional problem of data transfers is investigated, because even a trivial operation like copy becomes expensive. Yet, thanks to FFT-based multiplication, one can run elliptic curve calculations for numbers this big in reasonable time.

2 Edwards curves

A twisted Edwards curve is defined by:

$$ax^2 + y^2 = 1 + dx^2y^2$$

where a and d are the curve parameters. Edwards curves are a special case of twisted Edwards curves with $a = 1$.

There are several coordinate systems to represent and perform operations on curves without the need to do expensive division. Extended Edwards coordinates are defined as:

$$(X, Y, Z, T) : aX^2 + Y^2 = Z^2 + dT^2, x = \frac{X}{Z}, y = \frac{Y}{Z}, T = \frac{XY}{Z} = xyZ$$

Projective Edwards coordinates (X, Y, Z) can be obtained by just abandoning T, which is a feature widely used in our algorithms. Obtaining affine coordinates (x, y) requires division by Z and is called normalization.

Hisil, Wong, Carter, and Dawson in [4] introduced doubling and addition formulas in extended coordinates which do not require the parameter d. Doubling can be performed as $2(X_1, Y_1, Z_1, T_1) = (X_3, Y_3, Z_3, T_3)$ where

$$\begin{aligned} X_3 &= 2X_1Y_1(2Z_1^2 - Y_1^2 - aX_1^2), \\ Y_3 &= (Y_1^2 + aX_1^2)(Y_1^2 - aX_1^2), \\ Z_3 &= (Y_1^2 + aX_1^2)(2Z_1^2 - Y_1^2 - aX_1^2), \\ T_3 &= 2X_1Y_1(Y_1^2 - aX_1^2). \end{aligned} \tag{1}$$

Several things can be noted here. First, each formula is a multiple of two values, with only 4 distinct values in all formulas. Second, if $a = 1$ the formulas become completely parameter-free. Third, T_1 is not used at all. The same formulas can be used with both extended and projective input, and can produce either extended or projective output. One can even perform a series of projective doublings with the final extended doubling, which is exactly what is needed in signed window scalar multiplication.

Addition can be performed as $(X_1, Y_1, Z_1, T_1) + (X_2, Y_2, Z_2, T_2) = (X_3, Y_3, Z_3, T_3)$ where

$$\begin{aligned} X_3 &= (X_1Y_2 - Y_1X_2)(T_1Z_2 + Z_1T_2), \\ Y_3 &= (Y_1Y_2 + aX_1X_2)(T_1Z_2 - Z_1T_2), \\ Z_3 &= (Y_1Y_2 + aX_1X_2)(X_1Y_2 - Y_1X_2), \\ T_3 &= (T_1Z_2 + Z_1T_2)(T_1Z_2 - Z_1T_2). \end{aligned} \tag{2}$$

Here again each formula is a multiple of two values, with only 4 distinct values in all formulas. But both T_1 and T_2 are used. On the other hand, there are clear benefits if at least one of the inputs is normalized.

Montgomery in [5] introduced doubling and differential addition formulas for the single coordinate x_M, which are significantly faster than group operations for both coordinates. While Edwards curves perform better in scalar multiplication, there are still applications where differential addition of a single coordinate is enough, like stage 2 of ECM factorization. Since Montgomery curves are birationally equivalent to twisted Edwards curves, one can easily rewrite Montgomery formulas with the following transformations:

$$\begin{aligned} Y_{Ed} &= X_M - Z_M, \\ Z_{Ed} &= X_M + Z_M, \\ a/(a-d) &= (A+2)/4. \end{aligned}$$

Here (Y_{Ed}, Z_{Ed}) is a projective y-coordinate on a twisted Edwards curve with parameters a and d, (X_M, Z_M) is a projective x-coordinate on a Montgomery curve with parameter A.

Then y-coordinate doubling can be performed as $2(Y_1, Z_1) = (Y_3, Z_3)$ where

$$\begin{aligned} A_3 &= Z_1^2(Y_1^2 - \frac{d}{a}Y_1^2), \\ B_3 &= (Z_1^2 - Y_1^2)(Z_1^2 - \frac{d}{a}Y_1^2), \\ Y_3 &= A_3 - B_3, \\ Z_3 &= A_3 + B_3. \end{aligned} \tag{3}$$

y-coordinate differential addition can be performed as $(Y_1, Z_1) - (Y_2, Z_2) = (Y_0, Z_0)$, $(Y_1, Z_1) + (Y_2, Z_2) = (Y_3, Z_3)$ where

$$\begin{aligned} A_3 &= (Z_0 - Y_0)(Y_1 Z_2 + Z_1 Y_2)^2, \\ B_3 &= (Z_0 + Y_0)(Y_1 Z_2 - Z_1 Y_2)^2, \\ Y_3 &= A_3 - B_3, \\ Z_3 &= A_3 + B_3. \end{aligned} \tag{4}$$

3 FFT-based multiplication

There are several flavours of FFT-based multiplication algorithms. But they all have the same basic idea: input numbers are transformed into vectors of n independent residues, per-element multiplication performed, the resulting vector is inversely transformed into a number, carry is propagated from least significant to most significant word of the output. The complexity of transforms is $\mathrm{O}(n \log n \log \log n)$, while all other steps are $\mathrm{O}(n)$. That makes transforms the most computationally expensive part of multiplication.

A squaring requires these steps: forward transform, per-element squaring, inverse transform, carry propagation. A multiplication of two different numbers requires more steps: forward transform of the first number, forward transform of the second number, per-element multiplication, inverse transform, carry propagation. A squaring requires 2 transforms while a multiplication requires 3 transforms, which is a significant difference in performance. On the other hand, once a forward transform is performed, the vector can be used in multiple operations. Multiplication of previously transformed numbers requires only a single inverse transform. This is why the count of multiplications and squarings is an inaccurate metric for FFT-based algorithms. The count of transforms is much more relevant, minimizing transforms requires a different approach to algorithm design.

Another area of concern is the size of data structures. For relatively small numbers, merely thousands digits long, all necessary data can fit into fast caches of modern processor cores. Performance is limited only by the core's computing power. For numbers tens of millions digits long, vectors become so big that the only place where they can reside is slow main memory. A core can spend most of its time waiting for data to arrive from main memory.

In practice this means that FFT-based arithmetic libraries need to provide interfaces that allow algorithms to exploit opportunities to reduce memory accesses. Consider a simple vector operation $(a+b) \cdot c$. A traditional approach is to compute a temporary vector $a+b$, then multiply it by c, which requires 4 reads, 2 writes total. A better approach is to perform $(a + b) \cdot c$ per-element, which requires 3 reads, 1 write. A programming interface can become bloated with such functions, but it's a small price to pay for improved performance.

Unfortunately, algorithms that save writes for huge numbers may not be optimal for smaller numbers. If an algorithm needs to compute $(a + b) \cdot c$ and $(a + b) \cdot d$, it may be beneficial to compute $a + b$ twice for 6 reads, 2 writes total, instead of using a temporary vector resulting in 6 reads, 3 writes. For smaller numbers where writes are fast computing $a + b$ twice becomes an unnecessary processor load. In some cases two versions of the same algorithm need to be written, one that saves writes and one that saves processor instructions.

As demonstrated by previous examples, transformed vectors can be used not only for multiplication and squaring, but also addition and subtraction, the same field operations that can be performed with the original number. The only limitation here is that words of the output must be capable of holding enough data before carry is propagated. Since vector elements are independent of each other, the concept of carry is meaningless for transformed data. Only after applying inverse transform one can discover that a word of

the output is overflowing. The amount of data that fits into a word is determined by the underlying number system and by the size of transform n. Multiplication routines choose transform size to be big enough to hold the result of a multiplication. But usually there is spare "word space" to hold more data, because n is not fine-grained. Some libraries require n to be a power of 2, others allow more flexibility, but still n increments in large steps. Making use of that additional word space when available can greatly speed up a calculation.

4 Algorithms

For the purpose of this paper we are introducing an ideal programming interface which offers functions common to all FFT-based algorithms. Each function is considered a black box.

Let's define the following functions:

transform Performs a forward transform.

inv_transform Performs an inverse transform.

carry Propagates carry of a number and performs modulo operation (if necessary).

carry_with_mul It is trivial to multiply a result by a small constant during carry propagation for a negligible extra cost.

safe Boolean function that returns *true* if a given operation is allowed without intermediate carry propagation. In our experience with our own FFT-based arithmetic library, we have found that if the a^2 operation is safe, then $a \cdot b$ and $(a + b) \cdot c$ are also safe. For the rest of this paper it is assumed that the library in use guarantees a^2, $a \cdot b$, $(a + b) \cdot c$ operations are all safe.

Transformed values are denoted with a line over the variable: $\overline{X}$. It is allowed that the transform is performed in-place, overwriting untransformed value. The algorithms never try to access original variable once it was transformed. There's also an assumption that writing into a variable participating in operation is faster than into an unrelated variable.

Operations on vectors are per-element, with the minimum possible amount of reads and a single write. $\pm$ operation performs addition and subtraction, producing two outputs that can be written over the original two values.

Algorithm costs can be expressed in four metrics: transforms, carries, reads and writes.

The following functions can be used with any algorithm:

Algorithm 1: Naive implementation of arithmetic operations

```
1  function square(X)
2  begin
3      return carry(inv_transform(transform(X)^2));
4  end
5  function mul(X, Y)
6  begin
7      X̄ ← transform(X);
8      return carry(inv_transform(transform(Y) · X̄));
9  end
10 function add(X, Y)
11 begin
12     return carry(X + Y);
13 end
14 function add_transformed_with_carry(X̄, Ȳ)
15 begin
16     X ← inv_transform(X̄);
17     Y ← inv_transform(Ȳ);
18     return transform(carry(X + Y));
19 end
```

Note that multiplication is implemented with one transformed value written to memory and the other used on the fly. It is obviously a benefit not to store temporary data, but we still do it for one of the values. The reason for this is the way caches work. Two sufficiently large transforms start to compete with each other for cache space, degrading overall performance. It is often better to perform them sequentially.

From the above one can see that squaring costs 2 transforms, 1 carry, 1 read, 1 write. Multiplication costs 3 transforms, 1 carry, 3 reads, 2 writes. Addition costs 1 carry, 2 reads, 1 write. Addition of transformed values with carry is an expensive operation that should be avoided. It is given here as an example of bad algorithm design.

4.1 Doubling in extended Edwards coordinates

The best published algorithm [4, section 3.3] performs doubling in projective Edwards coordinates using 4 squarings, 3 multiplications, 6 additions/subtractions. A naive implementation costs 17 transforms, 13 carries, 25 reads and 16 writes.

Algorithm 2 improves on the naive implementation of doubling in projective Edwards coordinates using 14 transforms, 9 carries, 22 reads, and 17 writes (a savings of 4 transforms, 4 carries, 3 reads, a loss of 1 write). Algorithm 2 achieves these savings by using a library that allows saving transformed values, carry operations with an optional small multiplier, and a single $\pm$ operator to save two reads.

Algorithm 2 implements (1) using 14 transforms in projective coordinates and 15 transforms in extended coordinates for Edwards curves, 16 transforms in projective coordinates and 17 transforms in extended coordinates for twisted Edwards curves. The algorithm keeps intermediate values in the output variables

and requires only one temporary variable (T_3 in projective case and *tmp* in extended case).

Algorithm 2: Doubling, less transforms

Input: $X_1, Y_1, Z_1, flag\_extended, \overline{a}$=transform($a$).

Output: X_3, Y_3, Z_3, T_3 (optional)

1 $\overline{X}_1 \leftarrow$ transform(X_1);
2 $\overline{Y}_1 \leftarrow$ transform(Y_1);
3 $T_3 \leftarrow$ carry_with_mul(inv_transform($\overline{X}_1 \cdot \overline{Y}_1$), 2);
4 $\overline{Z}_1 \leftarrow$ transform(Z_1);
5 $Z_3 \leftarrow$ carry_with_mul(inv_transform($\overline{Z}_1^2$), 2);
6 $X_3 \leftarrow$ carry(inv_transform($\overline{X}_1^2$));
7 **if** $\overline{a}$ **then**
8 $\quad \overline{X}_3 \leftarrow$ transform(X_3);
9 $\quad X_3 \leftarrow$ carry(inv_transform($\overline{X}_3 \cdot \overline{a}$));
10 **end**
11 $Y_3 \leftarrow$ carry(inv_transform($\overline{Y}_1^2$));
12 Y_3, $X_3 \leftarrow\leftarrow$ carry($Y_3 \pm X_3$);
13 $Z_3 \leftarrow Z_3 - Y_3$;
14 $\overline{X}_3 \leftarrow$ transform(X_3);
15 $\overline{Y}_3 \leftarrow$ transform(Y_3);
16 $\overline{Z}_3 \leftarrow$ transform(Z_3);
17 $\overline{T}_3 \leftarrow$ transform(T_3);
18 **if** *flag_extended* **then** $tmp \leftarrow$ carry(inv_transform($\overline{X}_3 \cdot \overline{T}_3$));
19 $T_3 \leftarrow$ carry(inv_transform($\overline{Z}_3 \cdot \overline{T}_3$));
20 $Z_3 \leftarrow$ carry(inv_transform($\overline{Y}_3 \cdot \overline{Z}_3$));
21 $Y_3 \leftarrow$ carry(inv_transform($\overline{X}_3 \cdot \overline{Y}_3$));
22 swap(X_3, T_3);
23 **if** *flag_extended* **then** swap(T_3, tmp);

Note that if the input is normalized, line 5 can be replaced with a simple $Z_3 = 2$. If a has a numerically small value, lines 6-10 can be replaced with a single $X_3 \leftarrow$ carry_with_mul(inv_transform($\overline{X}_1^2$), a).

The dual carry operation on line 12 of Algorithm 2 can often be safely eliminated. Algorithm 3 uses 7, 8, or 9 carry operations (a savings of up to 2 carries over Algorithm 2). Algorithm 3 requires a library that tells us when adds-without-carries can be safely used in future multiplication operations.

Lines 4 & 5 in Algorithm 2 can be combined saving one write and subsequent read provided the library has a more robust interface that allows this combination.

Algorithm 3 uses 21 reads and 16 writes (saving 1 read and 1 write).

Algorithm 3: Doubling, less carries

Input: $X_1, Y_1, Z_1, flag\_extended, \overline{a}$=transform($a$).

Output: X_3, Y_3, Z_3, T_3 (optional)

1 $\overline{X}_1 \leftarrow$ transform(X_1);
2 $\overline{Y}_1 \leftarrow$ transform(Y_1);
3 $T_3 \leftarrow$ carry_with_mul(inv_transform($\overline{X}_1 \cdot \overline{Y}_1$), 2);
4 $Z_3 \leftarrow$ carry_with_mul(inv_transform(transform(Z_1)^2), 2);
5 $X_3 \leftarrow$ carry(inv_transform($\overline{X}_1^2$));
6 **if** $\overline{a}$ **then** $X_3 \leftarrow$ carry(inv_transform(transform(X_3) $\cdot\ \overline{a}$));
7 $Y_3 \leftarrow$ carry(inv_transform($\overline{Y}_1^2$));
8 **if** *safe*$((a+b+c)\cdot(d+e))$ **then** `// Ensures line 24 is safe`
9 $Y_3, X_3 \leftarrow\leftarrow Y_3 \pm X_3$;
10 $Z_3 \leftarrow Z_3 - Y_3$;
11 **else if** *safe*$((a+b)\cdot(c+d))$ **then** `// Ensures line 25 is safe`
12 $Y_3, X_3 \leftarrow\leftarrow Y_3 \pm X_3$;
13 $Z_3 \leftarrow$ carry($Z_3 - Y_3$);
14 **else** `// Always safe`
15 $Y_3, X_3 \leftarrow\leftarrow$ carry($Y_3 \pm X_3$);
16 $Z_3 \leftarrow Z_3 - Y_3$;
17 **end**
18 $\overline{X}_3 \leftarrow$ transform(X_3);
19 $\overline{Y}_3 \leftarrow$ transform(Y_3);
20 $\overline{Z}_3 \leftarrow$ transform(Z_3);
21 $\overline{T}_3 \leftarrow$ transform(T_3);
22 **if** *flag_extended* **then** $tmp \leftarrow$ carry(inv_transform($\overline{X}_3 \cdot \overline{T}_3$));
23 $T_3 \leftarrow$ carry(inv_transform($\overline{Z}_3 \cdot \overline{T}_3$));
24 $Z_3 \leftarrow$ carry(inv_transform($\overline{Y}_3 \cdot \overline{Z}_3$));
25 $Y_3 \leftarrow$ carry(inv_transform($\overline{X}_3 \cdot \overline{Y}_3$));
26 swap(X_3, T_3);
27 **if** *flag_extended* **then** swap(T_3, tmp);

One can go even further. It is possible to store transformed values for reuse and actually use them at the same time. Such an operation has two outputs, which are denoted by two $\leftarrow$ symbols.

Algorithm 4 uses 17 reads and 14 writes, a savings of 4 reads and 2 writes over Algorithm 3.

Algorithm 4: Doubling, less reads

Input: $X_1, Y_1, Z_1, flag\_extended, \overline{a}$=transform($a$).

Output: X_3, Y_3, Z_3, T_3 (optional)

1 $\overline{X}_1 \leftarrow \text{transform}(X_1)$;
2 $T_3 \leftarrow \text{carry\_with\_mul}(\text{inv\_transform}((\overline{Y}_1 \leftarrow \text{transform}(Y_1)) \cdot \overline{X}_1), 2)$;
3 $Z_3 \leftarrow \text{carry\_with\_mul}(\text{inv\_transform}(\text{transform}(Z_1)^2), 2)$;
4 $X_3 \leftarrow \text{carry}(\text{inv\_transform}(\overline{X}_1^2))$;
5 **if** $\overline{a}$ **then** $X_3 \leftarrow \text{carry}(\text{inv\_transform}(\text{transform}(X_3) \cdot \overline{a}))$;
6 $Y_3 \leftarrow \text{carry}(\text{inv\_transform}(\overline{Y}_1^2))$;
7 **if** *safe*$((a+b+c)\cdot(d+e))$ **then** `// Ensures lines 21,27 are safe`
8 $Y_3, X_3 \leftarrow\leftarrow Y_3 \pm X_3$;
9 $Z_3 \leftarrow Z_3 - Y_3$;
10 **else if** *safe*$((a+b)\cdot(c+d))$ **then** `// Ensures lines 22,28 are safe`
11 $Y_3, X_3 \leftarrow\leftarrow Y_3 \pm X_3$;
12 $Z_3 \leftarrow \text{carry}(Z_3 - Y_3)$;
13 **else** `// Always safe`
14 $Y_3, X_3 \leftarrow\leftarrow \text{carry}(Y_3 \pm X_3)$;
15 $Z_3 \leftarrow Z_3 - Y_3$;
16 **end**
17 **if** *flag_extended* **then**
18 $\overline{T}_3 \leftarrow \text{transform}(T_3)$;
19 $tmp \leftarrow \text{carry}(\text{inv\_transform}((\overline{X}_3 \leftarrow \text{transform}(X_3)) \cdot \overline{T}_3))$;
20 $T_3 \leftarrow \text{carry}(\text{inv\_transform}((\overline{Z}_3 \leftarrow \text{transform}(Z_3)) \cdot \overline{T}_3))$;
21 $Z_3 \leftarrow \text{carry}(\text{inv\_transform}((\overline{Y}_3 \leftarrow \text{transform}(Y_3)) \cdot \overline{Z}_3))$;
22 $Y_3 \leftarrow \text{carry}(\text{inv\_transform}(\overline{X}_3 \cdot \overline{Y}_3))$;
23 $\text{swap}(X_3, tmp)$;
24 **else**
25 $\overline{Z}_3 \leftarrow \text{transform}(Z_3)$;
26 $T_3 \leftarrow \text{carry}(\text{inv\_transform}(\text{transform}(T_3) \cdot \overline{Z}_3))$;
27 $Z_3 \leftarrow \text{carry}(\text{inv\_transform}((\overline{Y}_3 \leftarrow \text{transform}(Y_3)) \cdot \overline{Z}_3))$;
28 $Y_3 \leftarrow \text{carry}(\text{inv\_transform}(\text{transform}(X_3) \cdot \overline{Y}_3))$;
29 **end**
30 $\text{swap}(X_3, T_3)$;

Finally, a few more writes can be saved if the library allows computing $ab + cd$ in a single operation. Note that it is not always safe to compute $a^2 + b^2$ without doing a carry operation, because due to its nature squaring tends to amplify fringe cases. It limits applicability of this last algorithm, although it's still useful in many cases.

Algorithm 5 uses 17 reads and 12 writes, a savings of 2 writes over Algorithm 4. Algorithm 5 assumes $a = 1$

and performs several subtractions two times, trading computation efficiency for memory access efficiency.

Algorithm 5: Doubling, less writes

Input: $X_1, Y_1, Z_1, flag\_extended$.

Output: X_3, Y_3, Z_3, T_3 (optional)

1 **if** *not safe*$(a^2 + b^2)$ **then** goto Algorithm 4; `// Ensures line 5 is safe`
2 $\overline{X}_1 \leftarrow \text{transform}(X_1)$;
3 $T_3 \leftarrow \text{carry\_with\_mul}(\text{inv\_transform}((\overline{Y}_1 \leftarrow \text{transform}(Y_1)) \cdot \overline{X}_1), 2)$;
4 $Z_3 \leftarrow \text{carry\_with\_mul}(\text{inv\_transform}(\text{transform}(Z_1)^2), 2)$;
5 $Y_3 \leftarrow \text{carry}(\text{inv\_transform}(\overline{Y}_1^2 + \overline{X}_1^2))$;
6 $X_3 \leftarrow \text{carry\_with\_mul}(\text{inv\_transform}(\overline{X}_1^2), 2)$;
7 $\overline{Z}_3 \leftarrow \text{transform}(Z_3)$;
8 $\overline{Y}_3 \leftarrow \text{transform}(Y_3)$;
9 $\overline{X}_3 \leftarrow \text{transform}(X_3)$;
10 **if** *flag_extended* **then**
11 $\quad tmp \leftarrow \text{carry}(\text{inv\_transform}((\overline{T}_3 \leftarrow \text{transform}(T_3)) \cdot (\overline{Y}_3 - \overline{X}_3)))$;
12 $\quad T_3 \leftarrow \text{carry}(\text{inv\_transform}(\overline{T}_3 \cdot (\overline{Z}_3 - \overline{Y}_3)))$;
13 $\quad Z_3 \leftarrow \text{carry}(\text{inv\_transform}(\overline{Y}_3 \cdot (\overline{Z}_3 - \overline{Y}_3)))$;
14 $\quad Y_3 \leftarrow \text{carry}(\text{inv\_transform}(\overline{Y}_3 \cdot (\overline{Y}_3 - \overline{X}_3)))$;
15 $\quad \text{swap}(X_3, tmp)$;
16 **else**
17 $\quad T_3 \leftarrow \text{carry}(\text{inv\_transform}(\text{transform}(T_3) \cdot (\overline{Z}_3 - \overline{Y}_3)))$;
18 $\quad Z_3 \leftarrow \text{carry}(\text{inv\_transform}(\overline{Y}_3 \cdot (\overline{Z}_3 - \overline{Y}_3)))$;
19 $\quad Y_3 \leftarrow \text{carry}(\text{inv\_transform}(\overline{Y}_3 \cdot (\overline{Y}_3 - \overline{X}_3)))$;
20 **end**
21 $\text{swap}(X_3, T_3)$;

All four algorithms may be useful depending on multiplication library, hardware architecture, cache sizes and number sizes. We pay so much attention to doubling algorithms because doubling takes most of the runtime of windowed scalar multiplication with sufficiently big window.

4.2 Addition in extended Edwards coordinates

The typical use case for addition is in left-to-right scalar multiplication. After a doubling or a series of doublings, a curve point from precomputed dictionary corresponding to the current bit string is added to the current curve point [6]. It is assumed that all values stored in the dictionary are already transformed, and those transforms are not a part of the addition algorithm.

To compute (2) one needs 4 intermediate values: $T_1Z_2 \pm Z_1T_2$, $Y_1Y_2 + aX_1X_2$ and $X_1Y_2 - Y_1X_2$. The first two are trivial to compute, especially if $Z_2 = 1$. The last two require a lot of multiplication. The best published algorithm [4, section 3.2] uses a smart trick to save one multiplication for the cost of three additions. It can be adapted to FFT-based multiplication, costing 16 transforms in projective coordinates, with an inevitable complexity of tracking carry operations. But if the library allows computing $ab + cd$ in a single operation, it all becomes unnecessary. Straightforward computation of the formulas becomes the safest and the fastest method.

Algorithm 6 implements (2) using 15 transforms in projective coordinates and 16 transforms in extended coordinates for Edwards curves, 17 transforms in projective coordinates and 18 transforms in extended coordinates for twisted Edwards curves. Besides addition it also implements subtraction by multiplying X_2

and T_2 by -1. Normalization of the dictionary saves two transforms in all cases.

Algorithm 6: Addition/Subtraction

Input: $X_1, Y_1, Z_1, T_1, \overline{X}_2, \overline{Y}_2, \overline{Z}_2, \overline{T}_2$, *flag_subtraction*, *flag_extended*, $\overline{a}$=transform(a).
Output: X_3, Y_3, Z_3, T_3 (optional)

1 **if** *flag_subtraction* **then**
2 $\quad Z_3 \leftarrow$ carry_with_mul(inv_transform(transform(Z_1) $\cdot$ $\overline{T}_2$), -1);
3 **else**
4 $\quad Z_3 \leftarrow$ carry(inv_transform(transform(Z_1) $\cdot$ $\overline{T}_2$));
5 **end**
6 $T_3 \leftarrow$ carry(inv_transform(transform(T_1) $\cdot$ $\overline{Z}_2$));
7 **if** *not flag_extended or safe*$((a+b)\cdot(c+d))$ **then** `// Ensures line 30 is safe`
8 $\quad T_3, Z_3 \leftarrow\leftarrow T_3 \pm Z_3$;
9 **else**
10 $\quad T_3, Z_3 \leftarrow\leftarrow$ carry($T_3 \pm Z_3$);
11 **end**
12 $\overline{X}_1 \leftarrow$ transform(X_1);
13 $\overline{Y}_1 \leftarrow$ transform(Y_1);
14 $\overline{tmp} \leftarrow \overline{X}_1$;
15 **if** $\overline{a}$ **then**
16 $\quad tmp \leftarrow$ carry(inv_transform($\overline{a} \cdot \overline{tmp}$));
17 $\quad \overline{tmp} \leftarrow$ transform(tmp);
18 **end**
19 **if** *flag_subtraction* **then**
20 $\quad X_3 \leftarrow$ carry(inv_transform($\overline{X}_1 \cdot \overline{Y}_2 + \overline{Y}_1 \cdot \overline{X}_2$));
21 $\quad Y_3 \leftarrow$ carry(inv_transform($\overline{Y}_1 \cdot \overline{Y}_2 - \overline{tmp} \cdot \overline{X}_2$));
22 **else**
23 $\quad X_3 \leftarrow$ carry(inv_transform($\overline{X}_1 \cdot \overline{Y}_2 - \overline{Y}_1 \cdot \overline{X}_2$));
24 $\quad Y_3 \leftarrow$ carry(inv_transform($\overline{Y}_1 \cdot \overline{Y}_2 + \overline{tmp} \cdot \overline{X}_2$));
25 **end**
26 $\overline{X}_3 \leftarrow$ transform(X_3); // $X_1Y_2 - Y_1X_2$
27 $\overline{Y}_3 \leftarrow$ transform(Y_3); // $Y_1Y_2 + aX_1X_2$
28 $\overline{Z}_3 \leftarrow$ transform(Z_3); // $T_1Z_2 - Z_1T_2$
29 $\overline{T}_3 \leftarrow$ transform(T_3); // $T_1Z_2 + Z_1T_2$
30 **if** *flag_extended* **then** $tmp \leftarrow$ carry(inv_transform($\overline{Z}_3 \cdot \overline{T}_3$));
31 $Z_3 \leftarrow$ carry(inv_transform($\overline{Y}_3 \cdot \overline{Z}_3$));
32 $T_3 \leftarrow$ carry(inv_transform($\overline{X}_3 \cdot \overline{T}_3$));
33 $Y_3 \leftarrow$ carry(inv_transform($\overline{X}_3 \cdot \overline{Y}_3$));
34 swap(Y_3, Z_3);
35 swap(X_3, T_3);
36 **if** *flag_extended* **then** swap(T_3, tmp);

Note that because safe($ab + cd$) $\equiv$ safe($(a + b) \cdot c$) there should always be enough word space to perfrom 24-25, 27-28 operations safely. Line 6 can be replaced with $T_3 \leftarrow T_1$ if $Z_2 = 1$.

4.3 Doubling of projective Edwards y-coordinate

The following algorithm implements (3) using 10 transforms if there's enough word space and 11 transforms otherwise. Also, A_3 and B_3 may be reused in differential addition as the difference, in that case it is benefitial to transform them before computing Y_3 and Z_3 to save two transforms later.

Algorithm 7: Doubling of y-coordinate

Input: $Y_1, Z_1, flag\_transform, \overline{d}$=transform($\frac{d}{a}$).
Output: Y_3, Z_3, A_3, B_3

1 $Y_3 \leftarrow$ carry(inv_transform(transform$(Y_1)^2$));
2 $Z_3 \leftarrow$ carry(inv_transform(transform$(Z_1)^2$));
3 **if** *safe*$((a+b)\cdot(c+d))$ **then** `// Ensures line 15 is safe`
4 $\quad\overline{Y}_3 \leftarrow$ transform(Y_3);
5 $\quad\overline{Z}_3 \leftarrow$ transform(Z_3);
6 $\quad\overline{B}_3 \leftarrow \overline{Z}_3 - \overline{Y}_3$;
7 **else**
8 $\quad B_3 \leftarrow$ carry$(Z_3 - Y_3)$;
9 $\quad\overline{Y}_3 \leftarrow$ transform(Y_3);
10 $\quad\overline{Z}_3 \leftarrow$ transform(Z_3);
11 $\quad\overline{B}_3 \leftarrow$ transform(B_3);
12 **end**
13 $A_3 \leftarrow$ carry(inv_transform$(\overline{d}\cdot\overline{Y}_3)$);
14 $\overline{A}_3 \leftarrow$ transform(A_3);
15 $B_3 \leftarrow$ carry(inv_transform$((\overline{Z}_3 - \overline{A}_3)\cdot\overline{B}_3)$);
16 $A_3 \leftarrow$ carry(inv_transform$((\overline{Y}_3 - \overline{A}_3)\cdot\overline{Z}_3)$);
17 **if** *flag_transform and safe*$((a+b)^2)$ **then** `// Ensures lines 1-2 are safe`
18 $\quad\overline{A}_3 \leftarrow$ transform(A_3);
19 $\quad\overline{B}_3 \leftarrow$ transform(B_3);
20 $\quad\overline{Z}_3, \overline{Y}_3 \leftarrow\leftarrow \overline{A}_3 \pm \overline{B}_3$;
21 **else**
22 $\quad Z_3, Y_3 \leftarrow\leftarrow$ carry$(A_3 \pm B_3)$;
23 **end**

Note that safe$((a+b)^2)$ requires a lot of word space and may not be *true* often. Lines 3-12 are an example why word space matters.

4.4 Differential addition of projective Edwards y-coordinates

The following algorithm implements (4) using 12 transforms. Here too A_3 and B_3 can be reused later.

Algorithm 8: Differential addition of y-coordinates

Input: $Y_1, Z_1, \overline{Y}_2, \overline{Z}_2, \overline{A}_0 = \overline{Z}_0 + \overline{Y}_0, \overline{B}_0 = \overline{Z}_0 - \overline{Y}_0$, flag_transform.
Output: Y_3, Z_3, A_3, B_3

$Y_3 \leftarrow \text{carry}(\text{inv\_transform}(\text{transform}(Y_1) \cdot \overline{Z}_2))$;
$Z_3 \leftarrow \text{carry}(\text{inv\_transform}(\text{transform}(Z_1) \cdot \overline{Y}_2))$;
if *safe*$((a+b)^2)$ **then** `// Ensures lines 12-13 are safe`
    $Y_3, Z_3 \leftarrow\leftarrow Y_3 \pm Z_3$;
else
    $Y_3, Z_3 \leftarrow\leftarrow \text{carry}(Y_3 \pm Z_3)$;
end
$Y_3 \leftarrow \text{carry}(\text{inv\_transform}(\text{transform}(Y_3)^2))$;
$Z_3 \leftarrow \text{carry}(\text{inv\_transform}(\text{transform}(Z_3)^2))$;
$A_3 \leftarrow \text{carry}(\text{inv\_transform}(\text{transform}(Y_3) \cdot \overline{B}_0))$;
$B_3 \leftarrow \text{carry}(\text{inv\_transform}(\text{transform}(Z_3) \cdot \overline{A}_0))$;
if *flag_transform and safe*$((a+b)\cdot(c+d))$ **then** `// Ensures lines 1-2 are safe`
    $\overline{A}_3 \leftarrow \text{transform}(A_3)$;
    $\overline{B}_3 \leftarrow \text{transform}(B_3)$;
    $\overline{Z}_3, \overline{Y}_3 \leftarrow\leftarrow \overline{A_3} \pm \overline{B_3}$;
else
    $Z_3, Y_3 \leftarrow\leftarrow \text{carry}(A_3 \pm B_3)$;
end

Note that if $Z_2 = 1$, it saves two transforms but complicates tracking of carry.
An alternative ending is possible here.

Algorithm 9: Alternative ending of differential addition of y-coordinates

$\overline{Y}_3 \leftarrow \text{transform}(Y_3)$;
$\overline{Z}_3 \leftarrow \text{transform}(Z_3)$;
$Y_3 \leftarrow \text{carry}(\text{inv\_transform}(\overline{B}_0 \cdot \overline{Y}_3 - \overline{A}_0 \cdot \overline{Z}_3))$;
$Z_3 \leftarrow \text{carry}(\text{inv\_transform}(\overline{B}_0 \cdot \overline{Y}_3 + \overline{A}_0 \cdot \overline{Z}_3))$;
if *flag_transform and safe*$((a+b)\cdot(c+d))$ **then** `// Ensures lines 1-2 are safe`
    $\overline{Y}_3 \leftarrow \text{transform}(Y_3)$;
    $\overline{Z}_3 \leftarrow \text{transform}(Z_3)$;
    $\overline{A}_3, \overline{B}_3 \leftarrow\leftarrow \overline{Z_3} \pm \overline{Y_3}$;
else
    $A_3, B_3 \leftarrow\leftarrow \text{carry}(Z_3 \pm Y_3)$;
end

5 Implementation

5.1 FFT library interface

There is one more important optimization an FFT library should provide that saves one read and write in many situations.

The description of squaring in Algorithm 1 was a simplification of the work an FFT library performs when an input number is too large to fit in the CPU caches. FFT libraries use some variation of D.H Bailey's multi-pass approach [7]. A squaring operation consists of three steps designed to do as much work as possible while data is in the CPU caches.

1. Read a block of data into the CPU cache, perform first half of the forward transform, write results, repeat until all data processed.
2. Read a block of data into the CPU cache, perform second half of the forward transform, point-wise squaring, perform first half of the inverse transform, write results, repeat until all data processed.
3. Read a block of data into the CPU cache, perform second half of the inverse transsform, carry propagation, write results, repeat until all data processed.

Thus, data is read and written three times.

Multiplication is similar, replacing point-wise squaring with point-wise multiplication by a second already-transformed input. Multiplication of large numbers requires 4 reads and 3 writes.

If the result of a squaring or multiplication will be transformed it is advantageous to change step 3 to also do the first half of a forward transform before writing results. When the result is later used in a subsequent squaring or multiplication step 1 can be skipped entirely, saving one read and write. In practice, we've seen this optimization yield performance improvements of over 20%.

Below is a recap of the desirable functions an FFT library should provide.

Addition/subtraction routines:

$X \leftarrow A + B$; `// two versions with and without carry`
$X \leftarrow A - B$; `// two versions with and without carry`
$X, Y \leftarrow\leftarrow A \pm B$; `// two versions with and without carry`
$\overline{X} \leftarrow \overline{A} + \overline{B}$; `// no carry`
$\overline{X} \leftarrow \overline{A} - \overline{B}$; `// no carry`
$\overline{X}, \overline{Y} \leftarrow\leftarrow \overline{A} \pm \overline{B}$; `// no carry`

The following FFT multiplication routines are desirable where

- Carry operation supports an optional small multiplier
- One of the inputs can be in an untransformed state, all other inputs must be in a transformed state
- If one of the inputs is in an untransformed state, it is transformed and can optionally be written back to memory for later use
- Optionally start the forward FFT on the result

$\overline{X} \leftarrow \text{transform}(A)$;
$A \leftarrow \text{carry}(\text{inv\_transform}(\overline{X} \cdot \overline{Y}))$;
$A \leftarrow \text{carry}(\text{inv\_transform}((\overline{X} + \overline{Y}) \cdot \overline{Z}))$;
$A \leftarrow \text{carry}(\text{inv\_transform}((\overline{X} - \overline{Y}) \cdot \overline{Z}))$;
$A \leftarrow \text{carry}(\text{inv\_transform}(\overline{X} \cdot \overline{Y} + \overline{W} \cdot \overline{Z}))$;
$A \leftarrow \text{carry}(\text{inv\_transform}(\overline{X} \cdot \overline{Y} - \overline{W} \cdot \overline{Z}))$;
$bool \leftarrow \text{safe}(\textit{various multiplication operations})$;

It may seem onerous to support so many multiplication operations. In practice, all that is required is replacing the point-wise squaring in step 2 above, while reading auxiliary sources in as cache-friendly way as possible.

It is left as an exercise to the reader to modify the algorithms presented to take advantage of a start-next-forward-transform option. Note that the option is not available if the result is later used in an addition/subtraction operation with carry.

5.2 Performance data

All presented algorithms were implemented in Prefactor program (`https://github.com/patnashev/prefactor`) using GWNum FFT library (`https://www.mersenne.org/download/`), which was significantly extended to provide a better interface for such algorithms.

The following table shows performance data for sequential doubling in projective Edwards coordinates modulo numbers of different size. The data was obtained on a CPU with 8 MB L3 cache supporting AVX instruction set. Smaller values are better.

Table 1. Doubling performance.

Number	FFT size	safe $(a+b+c)\cdot(d+e)$	Algorithm 2	Algorithm 3	Algorithm 4	Algorithm 5
$F_{12} = 2^{4096} + 1$	192	$true$	0.728	0.701	0.657	0.680
$921 \cdot 2^{2937988} + 1$	200K	$true$	1 343	1 253	1 214	1 217
$2^{43112609} - 1$	2304K	$true$	21 724	19 959	18 935	18 812
$2^{43512653} - 1$	2304K	$false$	21 671	22 209	21 156	18 789

For smaller numbers like F_{12} additional computational complexity of Algorithm 5 doesn't pay off. But for larger numbers Algorithm 5 performs either as well as Algorithm 4 or significantly better in certain circumstances. Note that the two Mersenne numbers are similar in size and have statistically identical performance in Algorithms 2 and 5, but not Algorithms 3 and 4. That is because some operations become unsafe for the larger number. Poor performance of Algorithm 3 is explained by start-next-forward-transform option available for Algorithm 2 which outweighs less carries in Algorithm 3.

References

[1] Harold M. Edwards. A normal form for elliptic curves. *Bulletin of the American Mathematical Society, 44*, pages 393–422, 2007. https://doi.org/10.1090/S0273-0979-07-01153-6.

[2] Daniel J. Bernstein, Peter Birkner, Marc Joye, Tanja Lange, and Christiane Peters. Twisted Edwards Curves. Cryptology ePrint Archive, Report 2008/013, 2008. https://eprint.iacr.org/2008/013.

[3] Richard Crandall and Barry Fagin. Discrete weighted transforms and large-integer arithmetic. *Mathematics of Computation, 62*, pages 305–324, 1994. https://doi.org/10.1090/S0025-5718-1994-1185244-1.

[4] Huseyin Hisil, Kenneth Koon-Ho Wong, Gary Carter, and Ed Dawson. Twisted Edwards Curves Revisited. Cryptology ePrint Archive, Report 2008/522, 2008. https://eprint.iacr.org/2008/522.

[5] Peter L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation, 48*, pages 243–264, 1987. https://doi.org/10.1090/S0025-5718-1987-0866113-7.

[6] Christophe Doche. Exponentiation. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*, pages 145–168, 2006. http://hyperelliptic.org/HEHCC/chapters/chap09.pdf.

[7] D. H. Bailey. FFTs in External of Hierarchical Memory. *Supercomputing '89: Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 234–242, 1989. https://doi.org/10.1145/76263.76288.