OPTIMIZING ELLIGATOR 1 ON CURVE1174

Christopher Vogelsanger, Freya Murphy, Miro Haller

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

We implement and optimize Elligator 1 on Curve1174 using standard optimization techniques as well as sophisticated algorithmic specializations taking advantage of the curve prime and special structure in the mapping. Elligator is a bijective mapping of a subset of points on an elliptic curve to uniformly random bit-strings and back that was proposed by Bernstein et al.[1].

We find that this problem is heavily compute-bound. We optimize Elligator 1 for Intel Haswell i7-4980HQ 2.8 GHz and achieve a maximal speedup of 49.6x for the mappings and up to 83.8x for arithmetic operations on Curve1174. We compare the runtime on Intel's Haswell, Intel's Kaby Lake, AMD's Zen 2, and Apple's M1 platform.

1. INTRODUCTION

Bernstein et al. describe in [1] the Elligator 1 algorithm for Curve1174 and a more complex Elligator 2 algorithm for other elliptic curves. We chose to implement Elligator 1 (henceforth referred to as 'Elligator') as it is simpler and more focused on the mapping operations.

We implemented Elligator in Sage, Python and C. Our final optimized version is written in C with our own implementation of big integer arithmetic, although we also provide an implementation using the GMP (GNU Multi-Precision Arithmetic) library [2]. We set up a testing and benchmark framework to automate the verification process and to track progress.

Motivation. Due to their nature, points on elliptic curves are easily identifiable. Since elliptic curves are commonly used in cryptography, a simple way for malicious actors to stop encrypted data flows is to detect and drop encrypted messages. Therefore, it is important to hide the fact that curve points are sent. This is non-trivial and requires specialized algorithms such as Elligator. Since these mappings have to be done each time a point is sent and received, these operations are time-critical to enable fast communication.

Contribution. We provide the first open-source implementation of Elligator 1 in Sage, Python, and C. Note that we optimize those implementations for performance and not security. Implementation risks such as side-channels were *not* taken into account. Furthermore, we create a big integer arithmetic library for Curve1174 in C and describe generally applicable optimizations for modular arithmetic with a modulus close to a power of two. We compare the runtime of our implementation on different architectures, including Apple's M1 processor.

Related work. The Elligator algorithm was introduced by Bernstein et al. in 2013 [1]. While the paper covers in detail how to implement the algorithm, no open-source implementation is provided. Implementations of Elligator 2 can be found online [3, 4, 5, 6], but we are not aware of any for Elligator 1. The original paper focuses on the functionality of Elligator with little emphasis on performance. Algorithms for big integer arithmetic are covered in [7]. Those descriptions make simplifying assumptions and aim for understandable pseudo-code instead of best performance. GMP [2] is a highly optimized arbitrary-precision arithmetic library for C. GMP is general-purpose and has, to the best of our knowledge, not been applied to the specific Elligator 1 problem before nor optimized for Curve1174 operations.

2. BACKGROUND

In this section, we first briefly introduce Curve1174 and formally specify the Elligator 1 mappings. Next, we briefly discuss our initial BigInt library. Furthermore, we provide a cost analysis and roofline plot to understand Elligator's bottlenecks.

2.1. Notation

General notation:

- $\mathbf{a}||\mathbf{b}$: concatenation of the binary representations of a and b.
- $|\mathbf{a}|$: bit length of a.
- $[\mathbf{a}, \mathbf{b}]$: set of integers $\{a, a+1, \ldots, b\}$ for $a \leq b$. We use $[a, b)$ for $[a, b-1]$.
- `0bx`: binary representation for $x \in \{0,1\}^*$.
- **bit strings**: bit strings $\mathtt{0b}\mathtt{a}_\mathtt{x}\mathtt{a}_{\mathtt{x}-1}\ldots\mathtt{a}_0$ are interpreted as integer $X = \sum_{i=0}^{x} a_i \cdot 2^i$.
- **BigInt**: We denote big integers with greek characters. Each BigInt β has a corresponding set of chunks

$\{\beta_0, \beta_1, \ldots, \beta_k\}$ such that $\beta = \sum_{i=0}^{k} \beta_i \cdot r^i$, where r is the radix. For us, $r = 2^{32}$ (i.e., $|\beta_i| = 32$).

- **op$_{64}$, op$_{256}$**: We use op$_{64}$ to denote an arithmetic operation op on 64-bit integers, and op$_{256}$ for BigInt operations.

We use the following abbreviations for our optimization versions of the Elligator C implementation:

- V1: Straightforward C implementation.
- V2: Standard optimizations (pre-computation, strength-reduction) and algorithmic optimizations (modulus, square) described in Section 3.1.
- V3: Vectorization using AVX described in Section 3.2.

2.2. Elliptic Curves and Elligator

Elliptic curves, in our case, are algebraic curves defined over a prime field. For Curve1174, this is the set of points (x, y) which satisfy the equation $x^2 + y^2 = 1 - 1174x^2y^2$ over the field $\mathbf{F}_q$ for prime $q = 2^{251} - 9$ (i.e., $x, y \in \mathbf{F}_q$).

The Elligator 1 algorithm works exclusively for points on Curve1174. Let $\mathcal{P} \subset \mathbf{F}_q^2$ be the subset of points used in Elligator and $\mathcal{S} \subset \mathbf{F}_q$ the set of bit strings interpreted as integers in $\mathbf{F}_q$ (see [1]). Elligator 1 describes the mapping pnt2str: $\mathcal{P} \to \mathcal{S}$ from a curve point onto a uniformly random bit string and its inverse mapping str2pnt: $\mathcal{S} \to \mathcal{P}$. Note that we have $|\mathcal{P}| = |\mathcal{S}|$. The mathematical notation of those mappings are shown in Lst. 1 and Lst. 2. The function $\chi : \mathbf{F}_q \to \mathbf{F}_q$ is defined as $\chi(x) = x^{(q-1)/2}$ and shows the quadratic character of x (whether it is a square, non-square, or zero). $s \in \mathbf{F}_q$ is a constant field element. We define $c = 2/s^2$ and $r = c + 1/c$. Note that all operations are performed over $\mathbf{F}_q$ and we have a number of expensive power operations.

```
1  function str2pnt(t): // where t ∈ S
2      if t = ±1 return (0,1)
3      u = (1 − t)/(1 + t)
4      v = u^5 + (r^2 − 2)u^3 + u
5      X = χ(v)u
6      Y = (χ(v)v)^((q+1)/4) χ(v)χ(u^2 + 1/c^2)
7      x = (c − 1)sX(1 + X)/Y
8      y = (rX − (1 + X)^2) / (rX + (1 + X)^2)
9      return (x,y) // where p = (x, y) ∈ P
```

Lst. 1. The Elligator str2pnt mapping.

```
1  function pnt2str(p): // where p = (x, y) ∈ P
2      if p = (0,1) return 1
3      η = (y − 1)/(2(y + 1))
4      X̄ = −(1 + ηr) + ((1 + ηr)^2 − 1)^((q+1)/4)
5      z = χ((c − 1)sX̄(1 + X̄)x(X̄^2 + 1/c^2))
6      ū = zX̄
7      t̄ = (1 − ū)/(1 + ū)
8      return t̄ // where t̄ ∈ S
```

Lst. 2. The Elligator pnt2str mapping.

2.3. Basic BigInt Implementation

The Elligator 1 operations from Section 2.2 are all performed over a finite field for Curve1174 with a 251-bit prime. Normal integer operations are limited to 64-bit operands. Therefore, we implemented our own BigInt library, such that we can adapt and optimize it specifically for Elligator. We represent a single BigInt as a collection of 64-bit chunks. For each chunk, we use the lower 32 bits to store the integer's bits, and reserve the upper 32 bits for intermediate internal results (e.g., when multiplying two chunks). Initially, we allocated those chunks on the heap and later moved them to the stack (see Section 3.1). Metadata for the BigInt tracks the number of chunks and the sign of the stored large integer. For the basic operations (add$_{256}$, sub$_{256}$, mul$_{256}$, and div$_{256}$) we generalized the algorithms from "The Art of Computer Programming" [7] for signed variable-chunk integers. Furthermore, we implemented more advanced operations, including power pow$_{256}$ (using square-and-multiply), modular inverses inv$_{256}$ (using the Extended Euclidean algorithm), and Elligator's χ function chi$_{256}$.

Implementing the Elligator mapping functions is very simple: we just call the BigInt functions from our library which correspond to the mathematical operations described in Listings 1 and 2.

2.4. Cost Analysis

Since we have a fixed input size we optimize for runtime instead of performance. The algorithm uses BigInt operations that internally use 64-bit basic integer operations. We looked into profiling our code with Apple's *Instruments* [8]. However, it only allowed us to make coarse-grained measurements on the execution time in micro-seconds of single function calls. We were missing the option to make multiple measurements, count the number of operations, or observe the execution unit usage of single functions. For those reasons, we decided to manually tag all basic and BigInt operations in our code to get fine-grained statistics. We are aware that the compiler might be able to optimize certain operations into cheaper ones which would lead to a discrepancy as our tagging is done in the source code. However, we are confident that the overall picture we get from the tagging is consistent. In this tagging we distinguished between add$_{64}$, mul$_{64}$, div$_{64}$, shift$_{64}$, and bitop$_{64}$ operations.

Using these tags we get the instruction mix shown in Table 1, where we mapped each of the tagged functions onto the used execution unit in the Haswell processor. Table 2 shows the port distribution of integer execution units on the Intel Haswell architecture. Since there is no bottleneck due to instruction imbalance, we decided to simply use the sum of all operations for our cost function. Table 3 shows the cost function for both Elligator functions and all our code versions. We see that the number of instructions decreases

version	function	ALU	Shift	Mul	Div
V1	pnt2str	79.3	2.2	6.9	11.6
	str2pnt	79.4	2.1	6.8	11.6
V2	pnt2str	76.4	11.8	11.8	0.0
	str2pnt	76.4	11.8	11.8	0.0
V3	pnt2str	74.3	13.0	12.7	0.0
	str2pnt	74.2	12.9	12.9	0.0

Table 1. Percentage of instructions using a certain integer execution unit during Elligator's `str2pnt` and `pnt2str` functions.

Port 0	Port 1	Port 5	Port 6
ALU, Shift	ALU	ALU	ALU, Shift
Div	Mul		

Table 2. Ports with integer execution units on the Intel Haswell i7-4980HQ 2.8 GHz

for every new version.

Note that both mappings have a time complexity of $\mathcal{O}(1)$ since we do not vary the input size. Bernstein observes in [1] that the χ function (or in general, all power operations) are the main bottleneck. Computing b^e for $b, e \in \mathbf{F}_q$ using the well-known square-and-multiply algorithm takes $\mathcal{O}(\log(q))$ multiplications and $\mathcal{O}(\log(q))$ squares.

2.5. Roofline plot

As part of our cost analysis, we created a roofline plot for our main optimization target (Intel Haswell i7-4980HQ 2.8 GHz). The Haswell has four ports with execution units for integers [9], as shown in Table 2.

This gives a peak performance without vectorization of 4 integer operations (iops) per cycle since there are 4 ALUs on different ports. We can fit four 64-bit integers into one 256-bit AVX vector, which gives a peak performance with vectorization of 16 iops per cycle. We measured the actual memory bandwidth using the benchmarking tool *Novabench* [10] as 25 GB/s (or 8.9 B/cycle). We decided against taking the RAM manufacturer's data because our device is

version	str2pnt	pnt2str
V1	2.90	4.34
V2	1.23	1.84
V3	0.61	0.93

Table 3. Cost function for our different optimization versions in Miops (10^6 iops).

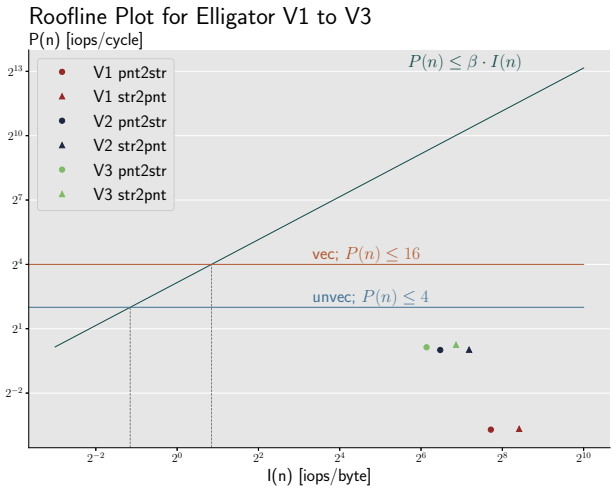

Fig. 1. Roofline plot for Elligator `pnt2str` and `str2pnt` functions for version 1 to 3.

rather old and unlikely to come close to the optimal bandwidth.

The roofline plot in Fig. 1 shows that Elligator is heavily compute-bound. This justifies our decision to focus on algorithmic optimizations rather than memory optimizations. Each successive version moves to the left on the intensity axis since it was refined to require fewer operations (compare to Table 3) and therefore less work while the amount of data moved stays the same. The greatest increase in performance was from `V1` to `V2`, with a small increase provided by the use of AVX instructions in `V3`.

The `str2pnt` algorithm, represented in Fig. 1 by triangles, has a slightly higher intensity than its `pnt2str` counterpart because it has more of the expensive power operations while having slightly less input data (one BigInt instead of two).

3. OPTIMIZATIONS

We first describe in Section 3.1 the non-vector optimizations we applied in `V2` to improve on `V1`. This includes standard optimizations (strength reduction, pre-computation, and changing compilation) as well as more sophisticated algorithmic improvements (square function, division-avoiding modulus, Fermat inverse). Next, Section 3.2 describes different approaches that we applied in `V3` to vectorize addition and multiplication.

On a high level, many of our optimizations utilize special properties of Curve1174 to transform generic BigInt operations into specialized functions tailored to their use in Elligator.

flags	V2	V3*	V3**
-O0	3.04	3.9	3.9
-O1	1.13	1.21	1.2
-O2	1.05	1.02	1.0
-O3	1.08	1.02	**0.98**
-Ofast	1.08	1.01	0.99

Table 4. Average of the runtime in multiples of 10^6 cycles (rounded to three digits) for `str2pnt` and `pnt2str` when compiled with different optimization flags. For `V2` we additionally used the `-fno-tree-vectorize` flag. For `V3` we added `-march=native -m64 -mavx2`. `V3*` has the `-fno-tree-vectorize` flag and `V3**` does not. The target laptop is specified in Section 4.1.

3.1. Non-Vector Optimizations

Strength reductions. We replace $\texttt{mod}_{64}$ with modulus 2^x by a bitwise $\texttt{and}_{256}$ with $2^x - 1$. This is used often since our representation uses only 32 of 64 bits for every chunk. We performed this optimization manually as we suspected the compiler might not recognize all instances.

Following the same reasoning, we replace divisions by a power of two with right shifts. This operation appears when we extract the carry of chunk additions from the upper 32 bits.

To avoid memory copies, we enforce that most BigInt functions do not accept aliasing inputs.

We measured that over 99.9% of the BigInts have eight chunks. This is expected, since there are $2^{i \cdot 32}$ i-chunk BigInts. Therefore, we create specially optimized versions of functions that are called when the chunk size is eight. In those functions, we can unroll loops and avoid special cases and size-dependent branching. The resulting simpler code makes it easier for the compiler to do optimizations.

For the Elligator 1 mapping functions we reduced the number of BigInts used to store intermediate results. Furthermore, we used strength reductions for multiplications by the $\chi(.)$ function. The $\chi(.)$ function is guaranteed to return 1, 0 or -1. For the 1 and -1 cases, we only have to XOR the signs of the BigInts[1] instead of doing a full $\texttt{mul}_{256}$. To further speed this up, we changed the $\chi(.)$ function to return the sign directly as an 8-bit integer value instead of a BigInt. We were able to avoid $\chi(.) = 0$ case by either catching special inputs earlier or verifying in the proof of Elligator in [1] that no input ever leads to this output.

Pre-computation. As a simple way to reduce operations we pre-computed all parts of the Elligator mappings that only depend on the fixed Curve1174. Those values are (see Listings 1 and 2): $c = 2/s^2$, $(c-1)s$, $1/c^2$, $r = c+1/c$, and $r^2 - 2$.

[1] A positive sign is stored as a 0 bit, the negative sign as a 1 bit.

Compilation. We use Makefiles [11] to easily gather and compile our project in a unified fashion across multiple operating systems and architectures. Makefiles are designed for large projects and accelerate the build process by compiling code to intermediate object files and then link them together to a final executable. We rewrote those Makefiles completely to compile all files at once and thereby enable more compiler optimizations and encourage code in-lining.

Memory optimizations. We noticed we had a large number of of memory operations that were taking a significant amount of time (see Table 7). These included calls to `alloc`, `calloc`, `memcpy`, and `destroy`.

One of our early optimizations was therefore refactoring to reduce memory operations. To do this we stored BigInts on the stack instead of the heap.

Create specific functions. Some BigInt operations do not need two large operands. We created functions to multiply a BigInt with a 32-bit integer (which corresponds to a single chunk). We often use this in the optimized $\texttt{mod}_{256}$ (described below) to multiply by 288. Moreover, we added a specialized $\texttt{pow}_{256}$ for integer (instead of BigInt) exponents.

Square function. Another function that has an often used special case is $\texttt{mul}_{256}$: when squaring a BigInt (e.g., in $\texttt{pow}_{256}$), both operands are the same. We created a separate $\texttt{squared}_{256}$ function to which we only have to pass one argument. This not only cuts down on the number of data accesses but additionally saves around half of the $\texttt{mul}_{64}$ chunk multiplications. For a normal $\texttt{mul}_{256}$ between BigInts α and β we would have to compute the k-th chunk γ_k of the resulting γ the following way:

$$\gamma_k = \sum_{i=0}^{k} \alpha_i \cdot \beta_{k-i}$$

However, we can simplify this as follows for $\texttt{squared}_{256}$:

$$\gamma_k = \alpha_{\frac{k}{2}}^2 |_{k \equiv_2 0} + 2 \sum_{i=0}^{\lceil \frac{k}{2} \rceil - 1} \alpha_i \cdot \alpha_{k-i}$$

Instead of computing chunk multiplications twice, once as $\alpha_i \cdot \alpha_j$ and once as $\alpha_j \cdot \alpha_i$, we just double their product using a left shift, which is faster. The only exception is the squaring of chunks $\alpha_i \cdot \alpha_i$ which do not get doubled. Those only appear chunks γ_k where k is even. This gets handled separately in the sum above in the first term, where $|_{k \equiv_2 0}$ stands for the condition that this term only appears when k is even.

Modulo q. Since we do finite field arithmetic, the $\texttt{mod}_{256}$ operation is omnipresent and optimizing it has large benefits. General-purpose mod operations require $\texttt{divrem}_{256}$, a division with rest, which is by far the most complex BigInt operation in our library. Furthermore, $\texttt{divrem}_{256}$ has many

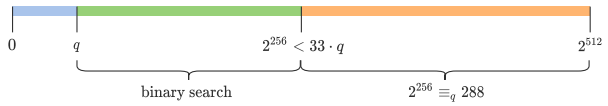

Fig. 2. Overview of the two tricks used to avoid $\texttt{divrem}_{256}$ in $\texttt{mod}_{256}$ and the range of numbers to which they are applicable.

data dependencies and is difficult to optimize. Our strategy is therefore to avoid $\texttt{divrem}_{256}$ in the first place. Fig. 2 gives an overview of the two tricks we use to achieve this.

First, for numbers[2] $2^{256} \leq X < 2^{512}$, we adapt Bernstein's fast modular arithmetic for Curve25519 [12] to our Curve1174 and use:

$$q = 2^{251} - 9 \Rightarrow 2^{251} - 9 \equiv_q 0 \Rightarrow 2^{256} \equiv_q 288$$

since our curve prime q is close to 2^{256}. Therefore, we can write $X = X_1 \cdot 2^{256} + X_0$, i.e., split it into two 256-bit chunks. Next, we reduce X to the smaller number $X \equiv_q X_1 \cdot 288 + X_0$.

Second, for numbers $q \leq Y < 2^{256}$ we note that $2^{256} < 33q$. Therefore, we need to find $i \cdot q$ such that $0 \leq Y - i \cdot q < q$ for $i \in \{1, 2, \ldots, 32\}$. For this purpose, we perform a binary search comparing Y to precomputed multiples of q. Thus, we only perform 5-6 cheap comparisons[3] and one $\texttt{sub}_{256}$ instead of an expensive $\texttt{divrem}_{256}$.

Both of those tricks may need to be applied repeatedly until the number is reduced to $[0, q)$. Overall, we therefore replaced all $\texttt{divrem}_{256}$ operations with cheaper $\texttt{mul}_{256}$, comparisons, and $\texttt{sub}_{256}$ operations.

Fermat inverse. Instead of computing the modular inverse with the Extended Euclidean Algorithm `egcd`, we use that the curve modulus q is prime. For a prime q, we have by Fermat's theorem $\forall a \in [1, q).a^{q-1} \equiv_q 1$. Thus, it follows that $a^{-1} \equiv_q a^{q-2}$.

3.2. Vector optimizations

We tried AVX optimizations for $\texttt{add}_{256}$ and $\texttt{mul}_{256}$ for the common case of eight chunks. As discussed, Elligator mostly operates on random BigInts which thus have eight chunks with high probability.

Vectorize $\texttt{add}_{256}$. For add, the necessity to manually move carries between chunks means that even with AVX instructions we have to do linearly dependant adds relative to the BigInt's number of chunks. As a result, using AVX instructions does not reduce the number of carry operations necessary. Additionally, the need to move the carry horizontally within an AVX register, over lanes and over registers is costly. The resulting AVX code looks similar to the scalar code as pipelined loading with offset was cheaper than shuffling and permuting the data afterward.

Vectorize $\texttt{mul}_{256}$. For $\texttt{mul}_{256}$ we ran into similar problems. The basic multiplication of different chunks can be done effectively by loading them with different offsets and then multiplying four chunks at a time with AVX instructions. The problem arises from adding up the different chunk multiplications. We once again run into the "data movement over lanes and registers" problem but this time we sum over many more than two chunks. Another hurdle comes from the fact that the result of the chunk multiplication will use all 64 bits. We cannot simply add more than two chunks up but have to potentially handle a carry after each chunk addition. This led to us extracting the results from the chunk multiplications and then doing the adding up in a scalar manner using compiler-supported 128-bit variables. This ended up being faster and far less complicated than a pure AVX version.

Prepare for 4-way multiplication. Since vectorizing single instructions proved to be difficult, we decided to take a different approach and perform four independent multiplications in parallel.

```
function pow(b, e, q):
    r = 1
    while e > 0:
        if e & 1:
            r = (r * b) mod q
        b = b^2 mod q
        e = e/2
    return r
```

Lst. 3. Normal square-and-multiply.

In order to make use of a 4-way $\texttt{mul}_{256}$, called $\texttt{mul4}_{256}$, we needed to restructure our code to have independent multiplications. We heavily customized the $\texttt{pow}_{256}$ operations for this purpose. First, note that we have some power operations with large, constant exponents: Section 2.2 shows two operations with exponent $(q+1)/4$, hidden in the χ function is a power with $(q-1)/2$, and the Fermat inverse (see Section 3.1) uses the exponent $q-2$. Those exponents, written in binary, are of the form: $e = \texttt{0b1}^{\texttt{n}}||\hat{\texttt{e}}$ with $n = 248, 247, 247$ and $\hat{e} = \texttt{0b011}, \texttt{0b0}, \texttt{0b0101}$. Therefore, we can rewrite the normal square-and-multiply algorithm shown in Lst. 3 to an optimized version shown in Lst. 4. The latter removes the branch on the condition `e&1` since we know it is always taken for the prefix of n ones. In addition to saving comparison operations, this enables the vectorization shown in Lst. 5. We unroll the loop four times and gather the result products in four *independent* variables r_i for $i \in [0, 3]$. In the end we calculate the final result as $r = \prod_{i=0}^{3} r_i$. To enable this optimization, we need to add

[2] Our internal numbers are guaranteed to be have less than 512 bits since we multiply at most two 251-bit numbers before doing a $\texttt{mod}_{256}$.

[3] The extra comparison is necessary due to the corner case where $Y = i \cdot q$ for some $i \in \{1, 2, \ldots, 32\}$ when the binary search is implemented with less-than-or-equal comparisons.

more cleanup code on lines 4 to 7 to handle corner cases ($\hat{e}$ as well as when n is not divisible by 4). Note that squaring the base b cannot be optimized due to its linear dependency. In our actual implementation, we had to unroll the loop eight times and use two sets of variables $r_{i,j}, b_{i,j}$ for $i \in [0,3], j \in [0,1]$ alternately. This enabled us to avoid aliasing and thus `memcpy`'s in the multiplication.

```
1  function pow(b, e, q):
2      r = b^ê
3      b = b^(|ê|+1)
4      while e > 0:
5          r = (r * b) mod q
6          b = b^2 mod q
7          e = e/2
8      return r
```

Lst. 4. Square-and-multiply for constant exponent $e = \texttt{0b1}^{\texttt{n}}||\hat{\texttt{e}}$ with $n \gg |\hat{e}|$.

```
1  function pow(b, e, q):
2      α = ⌊n/4⌋
3      β = n mod 4
4      r_1 = b^ê
5      r_2 = b^(1^β · 2^|ê|)
6      r_3 = r_4 = 1
7      b_3 = b^(β+|ê|+1)
8      for i ∈ [0, α]:
9          b_0 = (b_3)^2 mod q
10         b_1 = (b_0)^2 mod q
11         b_2 = (b_1)^2 mod q
12         b_3 = (b_2)^2 mod q
13         // Computes r_j = (r_j · b_j) mod q, j ∈ [0,3]
14         mul4_256(r_0, b_0, r_1, b_1, r_2, b_2, r_3, b_3)
15     return r_0 · r_1 · r_2 · r_3
```

Lst. 5. Vectorized square-and-multiply for constant exponent $e = 1^n||\hat{e}$ with $n \gg |\hat{e}|$.

Vectorize 4-way multiplication. The $\texttt{mul4}_{256}$ function does four independent $\texttt{mul}_{256}$ in parallel using AVX vector instructions. To make this possible we first need to interleave the four chunk arrays of the different $\texttt{mul}_{256}$ into one big array for every operand where the same chunk of the four $\texttt{mul}_{256}$ are adjacent. Next, we write a vectorized version of the BigInt multiplication algorithm. This works the same as the normal $\texttt{mul}_{256}$ algorithm but thanks to the change in representation we always load, operate, and store four independent 64-bit integer values. Note that this removes the need for horizontal data movement completely. Finally, the interleaved result BigInts have to be untangled to get the four resulting BigInts.

4. EXPERIMENTAL RESULTS

In this section, we first provide some background on our experiments and then analyze the speedup of `V3` compared to `V1`. Furthermore, we discuss the change in the function call numbers. Finally, we compare our C implementation for Elligator with our GMP implementation as well as on different devices.

function	V2	V3
`str2pnt`	30.0	47.2
`pnt2str`	31.3	49.6
$\texttt{add}_{256}$	5.9	6.2
$\texttt{chi}_{256}$	54.3	69.6
$\texttt{inv}_{256}$	7.1	11.1
$\texttt{mod}_{256}$	42.7	62.6
$\texttt{mul}_{256}$	4.8	10.2
$\texttt{pow}_{256}$	16.0	23.7
$\texttt{pow}_{256}$, exp: $(q+1)/4$	53.6	83.8
$\texttt{squared}_{256}$	45.5	61.7
$\texttt{sub}_{256}$	8.7	9.3

Table 5. Speedup of the functions in `V2` and `V3` relative to `V1`.

4.1. Experimental Setup

Target platform. Our target platform is a MacBook Pro Mid 2015 with an Intel Haswell i7-4980HQ 2.8 GHz and Apple clang version 12.0.0. We compile our code with the compiler flags `-O3 -mavx2 -march=native -m64` unless specified otherwise.

Benchmarks. As all our operations are on finite fields, we have fixed-sized inputs. Since we observe most operands to be random, we use random curve points and bit strings as inputs in our benchmarks. We measured all BigInt functions as well as both Elligator mappings to get a fine-grained overview and detect bottlenecks. Since we have a fixed input size, we measure runtime instead of performance. We additionally keep track of BigInt function calls and the use of basic integer operations.

4.2. Results

Compilation. Table 4 shows the results of using different optimization flags for `V2` and `V3`. We observe that moving from `-O0` to a better optimization suite makes a large difference. Afterward, the benefits diminish. The best compiler flag combination, which we picked for the final state of our project, is: `-O3 -mavx2 -march=native -m64`. This, together with compiling all files at once, gives us a 3x performance improvement compared to naïve compilation.

Modulo q. Table 5 shows that $\texttt{mod}_{256}$ achieves 62.6x speedup compared to the initial implementation. This is because we avoid the expensive division operations (see Table 7 where the stat count of `div` is reduced to zero in `V2` and `V3`).

Fermat inverse. Initially, the Fermat inverse was not faster than using `egcd`. However, due to our in-depth op-

fn	chunks	type	with flag	without flag
add	≤ 8	general	95	95
	8	no AVX	57	58
	8	AVX	73	73
	8	optimal	47	47
mul	≤ 8	general	401	402
	8	no AVX	188	188
	8	AVX	283	287

Table 6. Runtime in cycles of $\texttt{add}_{256}$ and $\texttt{mul}_{256}$ implemented for up to 8-chunk-BigInts without vectorization (*general*) or assuming 8-chunk inputs and without vectorization (*no AVX*) or with vectorization (*AVX*). Additionally, for $\texttt{add}_{256}$, there is a lower bound using AVX and ignoring carries (*optimal*). For all functions, we give the runtime with compiler flags `-Ofast -march=native -m64 -mavx2` and with `-fno-tree-vectorize` (*with flag*) or without it (*without flag*).

function	V1	V2	V3
add/sub	22074	4114	4281
chi	1	1	1
comparisons	36655	7827	8163
div	1460	0	0
inv	3	2	2
memory ops	103844	3	0
mod	1021	4162	4337
mul	11328	3162	112
mul4	0	0	244
mul single chunk	0	0	2249
other	63655	5	1
pow general	5	0	0
pow const. exp.	0	4	4
square	0	999	999

Table 7. Statistics of the change in function calls changed from `V1` to `V3` for Elligator `pnt2str`.

timization of power functions described in Section 3.2, we achieve 11.1x speedup (see Table 5).

Squared function. Table 5 shows that $\texttt{squared}_{256}$ is around 6x faster than $\texttt{mul}_{256}$ for squaring a BigInt.

4-way multiplication. The 4-way multiplication function gave us approx. 2.1x speedup (see Table 5: from `V2` to `V3` the speedup increases by this factor). However, due to the significant overhead of changing the chunk representation at the start and end of $\texttt{mul}_{256}$, we do not see the ideal 4x speedup. Some more speedup might be achievable by changing the BigInt representation in the power function instead of every multiplication. However, this would require complex rewriting of all other internal functions and force them to use a less optimal representation.

Vectorization. Using AVX instructions in the $\texttt{add}_{256}$ and $\texttt{mul}_{256}$ function lead to a worse runtime as shown in Table 6. In the case of $\texttt{add}_{256}$ using AVX on 8 chunk BigInts is 28% worse than using the specialized 8 chunk $\texttt{add}_{256}$ function without AVX. In the case of $\texttt{mul}_{256}$ using AVX on 8 chunk BigInts is 52% worse than using the specialized 8 chunk $\texttt{mul}_{256}$ function without AVX. Out of curiosity, we implemented an AVX add which ignores carries to obtain a lower bound on the achievable runtime. Table 6 shows that this optimal add only has a speedup of approximately 1.2x over the no AVX version. Therefore, we conclude that more sophisticated optimization for the carries could not achieve significant speedup.

Function calls. We reduced the overall number of function calls, especially to functions with a high runtime (see Table 7). The number of memory operations in `V1` is an order of magnitude larger than the number of $\texttt{mul}_{256}$ operations. Refactoring to store BigInts on the stack resulted in the removal of almost[4] all memory operations. This optimization alone led to a significant decrease in runtime.

Another key decision was to avoid making calls to $\texttt{div}_{256}$ rather than optimizing this expensive function. We achieved this through various algorithmic improvements discussed in Section 3.1.

We greatly reduced the calls to $\texttt{mul}_{256}$ by turning these into calls to specialized functions for 4-way multiplication, single chunk multiplication and squaring.

4.3. Comparison to GMP

We also implemented Elligator using the GMP library [2] in place of our own BigInt function calls. The GMP implementation corresponds to our `V1` implementation as future versions required specialized BigInt functions not available in GMP. GMP is a multi-precision arithmetic library with support for various number formats including integers and floats. It has been in development since 1991 and is thus highly optimized.

We benchmarked individual GMP functions against our own BigInt library (see Fig. 3) as well as benchmarking the GMP Elligator implementation against our own. The results showed that GMP, even without curve-specific optimizations, was still up to 6.6x faster than our implementation. We hypothesize this is due to a variety of reasons:

- GMP has been in development for a long time and has a large user-base. Hence it is highly optimized and refined. In contrast, we only had a few weeks to work on our implementation.

[4] Some rare copies are necessary to preserve values that will be overwritten inside a computation.

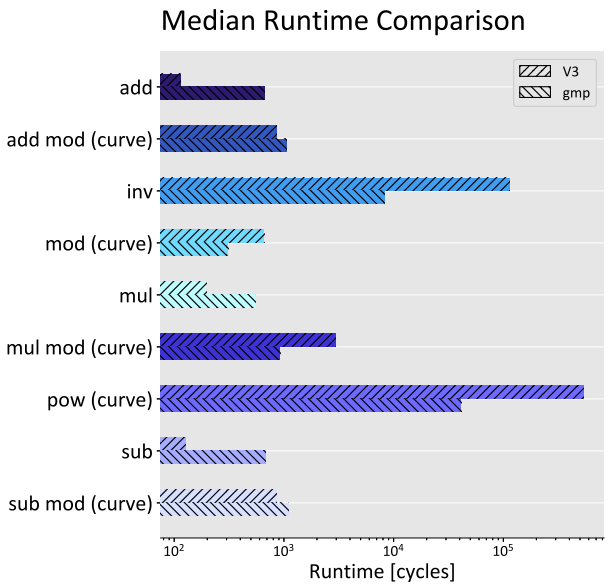

Fig. 3. Median runtime comparison for the most relevant BigInt functions to their equivalent in GMP. The operations tagged with *curve* use the modulus of Curve1174.

- GMP uses manually crafted assembly for some functions to achieve the best possible speedup.
- GMP implements multiple division algorithms highly optimized for different operand sizes.

Although our final Elligator implementation was not as fast as GMP, the fact that we implemented a GMP version of Elligator is still useful as we are not aware of any other Elligator 1 reference implementations. Using our Curve1174-specific optimizations we managed to outperform some of the generic GMP operations such as `add`, `sub`, and `mul`. GMP's additional speedup is mainly due to their extremely optimized division. This operation is so fast, that they can simply implement `mod` and `inv` directly with `div` and still outperform our specialized implementations. The `mod` operation is used extensively in `pow`, making GMP's `pow` faster than ours. Those operations where GMP has an advantage over ours happen to be particularly frequent in the Elligator mappings, resulting in an impressive performance of the Elligator implementation with GMP.

4.4. Processor Comparison

Finally, we repeated benchmarks on three other processors:

- Apple M1 with Apple clang version 12.0.0
- AMD Ryzen 9 3900X (Zen 2) @4.1GHz with Windows 10 WSL Ubuntu and gcc compiler
- Intel Kaby Lake i5-7200U @3.1GHz with Windows 10 WSL Ubuntu and gcc compiler

Processor	V1	V2	V3
Haswell	38.3	1.22	0.77
M1	8.16	0.30	-
Zen 2	15.5	0.98	0.82
Kaby Lake	18.1	0.97	0.83

Table 8. Runtime (in multiples of 10^6 cycles) comparison for Elligator `pnt2str` executed on different platforms. The M1 has no support for AVX instructions and therefore cannot run the vectorized `V3`.

Table 8 shows that the Intel Haswell, which was our main optimization target, saw the largest speedup but the slowest initial runtime.

Interestingly, the Apple M1 (which is a much newer processor) had by far the lowest runtime. This is even without AVX instructions, which are not supported by the M1's ARM architecture. ARM architectures instead use Neon intrinsics which only work with 128-bit vectors rather than the 256 bits provided on Intel architectures. This would fundamentally change the use of vector instructions throughout the project, such as optimizing for two-way multiplication rather than four-way multiplication.

5. CONCLUSIONS

We provide the first open-source implementation of Elligator 1 in Sage, Python and C, with C implementations both for our own BigInt arithmetic library optimized for Curve1174 and the generic GMP arithmetic library. We achieve a maximal speedup of 49.6x for our final Elligator implementation compared with our initial version. We also provide insights into the optimization potential on various different platforms, including the Apple M1. Although our BigInt arithmetic library is optimized for Curve1174 it is still outperformed in the context of Elligator by GMP.

Future research. Further work could be done to investigate how GMP achieves its additional speedup over our BigInt library. We are confident that their implementation tricks, alongside our Curve1174-specific optimizations would beat both implementations in terms of runtime. It would also be worth investigating how the Elligator algorithm could be optimized for specific architectures such as the Apple M1, which already shows high potential even without the use of vector instructions.

Next steps to further customize operations for the Elligator mapping could look into different representations of Curve1174. Bernstein uses the Montgomery formulas in [13] to implement high-speed Diffie-Hellman operations on Curve25519. Fermat inversion only needs 11 multiplications in that setting.

6. REFERENCES

[1] Daniel J Bernstein, Mike Hamburg, Anna Krasnova, and Tanja Lange, "Elligator: Elliptic-curve points indistinguishable from uniform random strings," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 967–980.

[2] Torbjrn Granlund and the GMP development team, *GNU MP: The GNU Multiple Precision Arithmetic Library*, 6.2.1 edition, 11 2020, `https://gmplib.org/`.

[3] Adam Langley, "Implementing elligator for curve25519," online, 12 2013, `https://www.imperialviolet.org/2013/12/25/elligator.html`.

[4] A. Faz-Hernandez, S. Scott, N. Sullivan, R.S. Wahby, and C.A. Wood, "Hashing to elliptic curves," online, 3 2020, `https://tools.ietf.org/id/draft-irtf-cfrg-hash-to-curve-06.html#name-elligator-2-method`.

[5] Biryuzovye Kleshni, "Elligator 2," online, 2021, `https://github.com/Kleshni/Elligator-2`.

[6] David McClain, "Edwards-curves," online, 2015, `https://github.com/dbmcclain/Edwards-Curves`.

[7] Donald E Knuth, *Art of computer programming, volume 2: Seminumerical algorithms*, Addison-Wesley Professional, 2014.

[8] Apple Inc., "Xcode - features," online, 2021, `https://developer.apple.com/xcode/features/`.

[9] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 3b edition, 09 2019.

[10] Novawave Inc., "Novebench," online, 2021, `https://novabench.com`.

[11] Inc. Free Software Foundation, "The gnu make manual for gnu make version 4.3," online, 2021, `https://www.gnu.org/software/make/manual/make.html`.

[12] Daniel J Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang, "High-speed high-security signatures," *Journal of cryptographic engineering*, vol. 2, no. 2, pp. 77–89, 2012.

[13] Daniel J Bernstein, "Curve25519: new diffie-hellman speed records," in *International Workshop on Public Key Cryptography*. Springer, 2006, pp. 207–228.