FAST IMPLEMENTATIONS OF CURVE25519 ON INTEL SKYLAKE

Philippe Goetschmann *Florian Moser* *Fabio Streun* *Lukas Tobler*

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

Curve25519 is an important primitive for elliptic curve Diffie-Hellman key exchange. We implement it using Intel's AVX2 instructions set in a 4-way vectorized fashion, targeting Intel's Skylake architecture. We explore using integer arithmetic and floating point arithmetic, applying 4-way FMA units to elliptic cryptography as a novel idea. Our designs manage to beat the performance of the currently fastest implementations in the SUPERCOP benchmarking suite.

1. INTRODUCTION

We wanted to implement the fastest possible version of an Elliptic Curve Diffie-Hellman (ECDH) protocol. While optimizing performance, we wanted to avoid introducing timing attacks. We chose the Skylake-architecture to be able to use AVX2.

We decided to focus on Curve25519 because it is used in many important protocols, like OpenSSH [1] and WireGuard [2] and because it was specifically chosen by its designer for its performance [3].

The main component that dictates performance of the ECDH scheme with Curve25519 is the scalar multiplication: It requires 255 bit numbers, which cannot directly be represented on modern CPU architectures. We analyzed different 255 bit representations and multiplication algorithms.

Substantial speedup was achieved by using vectorization. We present two 4-way vectorized implementations using integer and floating point arithmetic.

Related work. The elliptic curve known as Curve25519 was introduced by D. J. Bernstein in his seminal paper [3]. Many subsequent papers have focused on fast implementation of this curve for ARM NEON [4], 2-way vectorization on Intel Sandy Bridge [5], using Intel's scalar 64-bit multiplier [6] and AVX2 4-way vectorization [7].

2. BACKGROUND: SCALAR MULTIPLICATION

In this sections we describe the scalar multiplication on the Curve25519. In [3] an in-depth discussion can be found how the algorithm was constructed. We mainly used RFC 7748 [8], which describes the algorithm in a compact form.

The scalar multiplication (Algorithm 1) expects a scalar n and a point coordinate P as inputs and produces a point coordinate as output. It uses the Montgomery ladder approach such that the algorithm always executes in a fixed amount of time to avoid timing attacks. The function $\textsc{Bit}_i$ returns the i-th bit of its argument. The function $\textsc{Cswap}$ is a conditional swap function, which should be implemented in constant time.

2.1. Field representation

Note that all variables used in the algorithm besides n, i and bit are elements of the underlying field $GF(2^{255}-19)$ and require 255 bits to be represented. A common approach is to represent an element $f \in GF(2^{255}-19)$ using a radix-2^r representation:

$$f = \sum_{i=0}^{\lceil 255/r \rceil - 1} f_i 2^{\lceil ir \rceil}$$

This representation consists of $\lceil 255/r \rceil$ limbs. Different values of r allow different data types to be used for the limbs, but r must be smaller than the target register size.

We use the radix-$2^{25.5}$ representation with 10 limbs (same as in the original paper [3]) as a running example:

$$f = f_0 2^0 + f_1 2^{26} + f_2 2^{51} + f_3 2^{77} + f_4 2^{102} + f_5 2^{128} + f_6 2^{153} + f_7 2^{179} + f_8 2^{204} + f_9 2^{230}$$

2.2. Field operations

All operations with field elements, i.e. additions, subtractions, multiplications, squaring and inversion, are performed on the underlying finite field $GF(2^{255}-19)$.

Addition/Subtraction. For two field elements f, g, represented in some radix-2^r, we perform addition / subtraction piecewise on each limb:

$$h_i = f_i \pm g_i, \forall i \in \{0, ..., \lceil 255/r \rceil - 1\}$$

Algorithm 1 Scalar multiplication on Curve25519

```
Input: $n \in \{2^{254} + 8 \cdot [0, 2^{252} - 1]\}$,
       $P \in GF(2^{255} - 19)$
 1: $x_1 \leftarrow P$
 2: $x_2 \leftarrow 1$
 3: $z_2 \leftarrow 0$
 4: $x_3 \leftarrow P$
 5: $z_3 \leftarrow 1$
 6: $swap \leftarrow 0$
 7: for $i \leftarrow 254$ to $0$ do
 8:     $bit \leftarrow \textsc{Bit}_i(n)$
 9:     $swap \leftarrow swap \oplus bit$
10:     $(x_2, x_3) \leftarrow \textsc{Cswap}(swap, x_2, x_3)$
11:     $(z_2, z_3) \leftarrow \textsc{Cswap}(swap, z_2, z_3)$
12:     $swap \leftarrow bit$
13:     $A \leftarrow x_2 + z_2$
14:     $AA \leftarrow A^2$
15:     $B \leftarrow x_2 - z_2$
16:     $BB \leftarrow B^2$
17:     $C \leftarrow x_3 - z_3$
18:     $D \leftarrow x_3 + z_3$
19:     $E \leftarrow AA - BB$
20:     $DA \leftarrow A \cdot D$
21:     $CB \leftarrow B \cdot C$
22:     $x_2 \leftarrow AA \cdot BB$
23:     $x_3 \leftarrow (DA + CB)^2$
24:     $z_2 \leftarrow (E \cdot 121665 + AA) \cdot E$
25:     $z_3 \leftarrow x_1 \cdot (DA - CB)^2$
26: end for
27: $(x_2, x_3) \leftarrow \textsc{Cswap}(swap, x_2, x_3)$
28: $(z_2, z_3) \leftarrow \textsc{Cswap}(swap, z_2, z_3)$
29: return $z_2^{-1} \cdot x_2$
```

When using an unsigned data type for the limbs, a multiple of the fields prime (i.e. $2^{255} - 19$) should be added to f before subtracting g to avoid underflows.

Multiplication/Squaring. A common approach to multiply two field elements f, g represented in some radix-2^r is the schoolbook method. It multiplies each limb of f with each limb of g while at the same time performing the modular reduction.

The following formula shows the schoolbook multiplication when using the radix-$2^{25.5}$ representation:

$$\begin{array}{llllll}
h_0 = f_0g_0 + 2 \cdot 19 f_1 g_9 & + \ldots + 19 f_8 g_2 + 2 \cdot 19 f_9 g_1 \\
h_1 = f_0 f_1 + \quad f_1 g_0 & + \ldots + 19 f_8 g_3 + 19 \quad f_9 g_2 \\
\vdots & \ddots & \vdots \\
h_8 = f_0 g_8 + \quad 2 f_1 g_7 & + \ldots + \quad f_8 g_0 + 2 \cdot 19 f_9 g_9 \\
h_9 = f_0 g_9 + \quad f_1 g_8 & + \ldots + \quad f_8 g_1 + \quad f_9 g_0
\end{array}$$

For the complete formula see [5]. Each new limb is the sum of 10 products from the original limbs. The factor 2 is to correct for the alternating limb sizes and the factor 19 is the carry-over factor for the modulo operation on the finite field.

Squaring is implemented analogously but due to its symmetry some limb multiplications can be reused.

Coefficient reduction. After performing addition/subtraction and multiplication/squaring some limbs of the field may have overflown. To correct this the coefficient reduction transfers the overflowing bits from one limb to the following limb. For radix-$2^{25.5}$ with unsigned integers it looks like this:

$$\begin{array}{ll}
h_1 \mathrel{+}= h_0 >> 26 & h_0 \mathrel{\&}= 2^{26} - 1 \\
h_2 \mathrel{+}= h_1 >> 25 & h_1 \mathrel{\&}= 2^{25} - 1 \\
\vdots & \vdots \\
h_0 \mathrel{+}= (h_9 >> 25) \cdot 19 & h_9 \mathrel{\&}= 2^{25} - 1 \\
h_1 \mathrel{+}= h_0 >> 26 & h_0 \mathrel{\&}= 2^{26} - 1
\end{array}$$

The factor 19 is again the carry-over factor for the modulo operation.

By choosing a small enough r and big enough data type for the limbs it is possible to perform the coefficient reduction only after multiplications/squarings and not after additions/subtractions. This is because multiplications/squarings only use results of additions/subtractions as input.

Inversion. As a last step in the scalar multiplication the field element z_2 has to be inverted. By Fermats little theorem, $z_2^{-1} = z_2^{p-2}$, where $p = 2^{255} - 19$. This can be calculated in a sequence of multiplications and squarings.

2.3. Cost analysis

For the operation count we only consider operations on field elements.

The loop body is executed 255 times. One execution contains 2 CSWAP, 4 field additions, 4 field subtractions, 5 field multiplications, 4 field squarings and 1 field multiplication with a constant value (121665).

After the loop, two CSWAP, one field inversion and one field multiplication are performed. A field inversion consists of 11 field multiplications and 254 field squarings.

After each field multiplication and field squaring a coefficient reduction has to be performed. This sums up to a total of 512 CSWAP, 1020 field additions, 1020 field subtractions, 1286 field multiplications, 1274 field squarings and 255 field multiplications with a constant.

The processor instruction count depends on the chosen field representation and the data type used for limbs. With the radix-$2^{25.5}$ representation and unsigned 32-bit integers as data type for the limbs, the following integer operations are required:

- Field addition: 10 add

- Field substraction: 10 add, 10 sub
- Field multiplication: 90 add (64-bit), 114 mul (64-bit)
- Field squaring: 45 add (64-bit), 68 mul (64-bit)
- Coefficient reduction: 11 add (64-bit), 1 mul, 11 shift (64-bit), 11 and

This results in a total of 61576 add/sub, 51968 and, 173160 add (64 bit), 30976 shift (64 bit) and 238716 mul (64 bit), which sums up to 556396 integer operations.

3. ANALYSIS

In this section we analyze different radix-2^r representations and multiplication algorithms.

3.1. Field representation

A smaller radix implies more operations, because there are more limbs. However, less bits per limb are used, which might enable other forms of improvements, like vectorization using AVX2 (there is no vectorized 64-bit multiplication).

To avoid coefficient reduction after each addition / subtraction we need more bits than those strictly required by the radix. The largest feasible radix on our architecture is 2^{51}, which uses 64-bit integer limbs. For radixes smaller or equal to $2^{25.5}$, 32-bit integer limbs can be used.

Figure 1 shows the operation count and cycles needed for a naive implementation over different radixes. We observe that the trivial implementation of the largest radix is the fastest and it also uses the fewest instructions, even when counting 128-bit additions and shifts as two operations. Another sweet spot can be seen at radix $2^{25.5}$, precisely when we can switch our limb representation to 32-bit integers. As a third contribution, this graph allows us get a feeling of how much smaller radixes tend to increase the runtime. This allows us to estimate if an additional optimization for a smaller radix might offset the additional instructions needed to implement that radix.

3.2. Field operations

We attempted improving the multiplication method. Squaring and multiplication together are accountable for more than 80% of the operations for radix $2^{25.5}$, hence an even slightly different operation mix or operation count could have a large impact on overall runtime. We believe it is unlikely that we can improve upon the other operations; they are trivial (like add, which simply adds component wise) or seldom used (like encoding/decoding of the input).

We expect the observations for the multiplication method to apply to the squaring, as transforming a fast multiplication to an even faster squaring is likely possible. For our analysis, we compared naive implementations against each

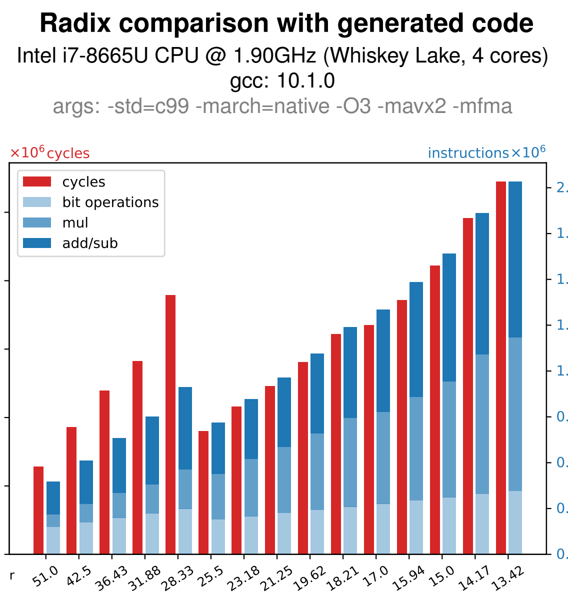

Fig. 1. Comparison of instructions and performance for different radixes on Skylake. We count 128-bit adds / shifts as two operations.

other, because we believed the results would also apply to heavily optimized versions.

Schoolbook multiplication. The Schoolbook method multiplies each limb with each other, and performs modular reduction at the same time. All existing implementations we investigated used this method.

Karatsuba. The Karatsuba method transforms a big multiplication into two of half the size, at the cost of some more additions [9]. It requires an integer radix for the splitting to work.

We used a code generator to compare the Karatsuba against the Schoolbook method for multiplication. For example, we observed that radix 2^{17}, which needed 210 adds and 239 mults with the Schoolbook method, needed only 218 adds, 15 subs and 191 mults using the Karatsuba method. For integer radixes we tried 2^{51}, 2^{17} and 2^{15}. We indeed measured a slightly lower execution time of the naive implementations using the Karatsuba method versus those that used the schoolbook method.

To improve upon Karatsuba, we also considered using the Toom-Cook generalization [10]. It further reduces the asymptotic complexity of Karatsuba but requires divisions [11].

However, with hand-written code for radix 2^{51} we could not measure a performance benefit compared to the Schoolbook method. Further, the smaller radixes (2^{17}, 2^{15}) showed not enough improvement to evaluate them further.

4. FLOATING POINT IMPLEMENTATION

In this section we show how we designed the field operations for floating point arithmetic and how we optimized the Montgomery ladder for SIMD.

Idea. Bernstein implemented his Curve25519 originally for a 32-bit x86 CPU [3]. He made use of the x87 (floating point) instruction set with the extended precision format, because this was faster than 32-bit integer arithmetic at the time.

Contemporary implementations however, rely on integer arithmetic, either using 64-bit multipliers or vectorized 32-bit multipliers. The x87 instruction set has been deprecated.

We propose to re-evaluate the use of floating point arithmetic for prime field computations, as recent CPUs, like Intel's Skylake, have powerful FMA (fused multiply-add) units. On Skylake, FMA units can calculate operations of the form $d = a * b + c$ in a 4-way vectorized fashion, with a throughput of 2 instructions per cycle, which totals to 16 floating point operations per cycle. We believe prime field operations, especially multiplication and squaring, can benefit from this because they have a similar structure to computing dot products, for which FMAs are supposed to be fast.

4.1. Field representation

Bernstein proposed to use 22-bit limbs for architectures that only had double precision arithmetic [3]. We decided to use a radix $2^{21.25}$ representation, because 21.25 divides 255 allowing us to reuse some code. However, we don't believe there is good reason *not* to use radix 2^{22}. A field element F_p with $p = 2^{255} - 19$ is represented as follows in radix $2^{21.25}$:

$$\begin{aligned} F_p = \sum_{i=0}^{11} a_i 2^{\lceil 21.25i \rceil} = \\ a_0 2^0 + a_1 2^{22} + a_2 2^{43} + a_3 2^{64} \\ + a_4 2^{85} + a_5 2^{107} + a_6 2^{128} + a_7 2^{149} \\ + a_8 2^{170} + a_9 2^{192} + a_{10} 2^{213} + a_{11} 2^{234} \end{aligned}$$

A term $a_i 2^{\lceil 21.25i \rceil}$ can be exactly represented with a double precision number, as long as $|a_i| < 2^{53}$, the mantissa length of a 64-bit double precision number. For a more rigorous treatment of the arithmetic used in this section, see [12].

Tuple format. Because AVX2 has the limitation that permutations across 128-bit lanes incur more latency than within lanes, we tried to avoid them. Previous work ([7], [5]) overcame this by packing two field elements into a tuple and performing the same operation (multiplication, squaring) on two field elements at the same time. We chose to use this approach.

Such a tuple of field elements $\langle A, B \rangle$ is stored in six vectors like this:

$$\langle A, B \rangle_i = [a_{2i}, a_{2i+1}, b_{2i}, b_{2i+1}], \quad i \in \{0, \ldots, 5\}$$

The Montgomery ladder algorithm defined in RFC [8] had to be modified such that it works with these tuples (see section 4.3).

4.2. Field operations

Addition. Addition (and subtraction) is performed by element-wise addition of the terms representing the field. One field addition therefore consists of *12 additions*.

Multiplication. Multiplication is done in the typical "Schoolbook" fashion. Because we have more limbs than in the common radix $2^{25.5}$, there is an increase in operations. However, on Skylake, additions, multiplications and FMAs have the same latency and throughput. One can argue that when considering FMA operations as single operations, the instruction count is actually significantly lower compared to the 10-limb integer representation. The field multiplication looks like this:

$$\begin{aligned} r_0 &= a_0 b_0 + c_{19} a_1 b_{11} + \ldots + c_{19} a_{10} b_2 + c_{19} a_{11} b_1 \\ r_1 &= a_0 b_1 + a_1 b_0 + \ldots + c_{19} a_{10} b_3 + c_{19} a_{11} b_2 \\ &\vdots \qquad\qquad\qquad \ddots \qquad\qquad \vdots \\ r_{10} &= a_0 b_{10} + a_1 b_9 + \ldots + a_{10} b_0 + c_{19} a_{11} b_{11} \\ r_{11} &= a_0 b_{11} + a_1 b_{10} + \ldots + a_{10} b_1 + a_{11} b_0 \end{aligned}$$

Here $r_0, \ldots, r_{11}$ is the resulting field element, and c_{19} is $19 \cdot 2^{-255}$. The multiplication by c_{19} is done to reduce terms modulo $2^{255} - 19$.

In principle, 210 multiplications and 132 additions would have to be performed. However, because terms of the form $c_{19} b_i$ occur more than once, every such multiplication can be pre-calculated and reused. This means, field multiplications require just *$12 + 144 = 156$ multiplications and 132 additions*.

We now defined this operation in such a way that it can be used with the aforementioned tuple format to produce two field multiplications at the same time. This is shown in algorithm 2.

Squaring. Squaring works analogously to multiplication, as it is simply a multiplication where some terms occur twice. These terms are simply multiplied by 2, instead of calculating them again. In that case, squaring takes *96 multiplications and 66 additions*.

Coefficient reduction. Because we are working with limbs with finite precision, we have to perform carry operations that ensure they stay in acceptable bounds. In the integer case, this is quite simple: A carry from $a_i \rightarrow a_j$ can be done with $SHIFT$ and AND operations ([5], [7]). With double precision numbers, this does not work.

Instead, we rely on a concept called "loss of precision

Algorithm 2 SIMD vectorized multiplication of two fields at once

```
Input: Interleaved tuples $\langle A, C\rangle, \langle B, D\rangle$
Output: Interleaved tuple $\langle AB,\ CD\rangle = \langle A \cdot B,\ C \cdot D\rangle$
 1: $\mathbf{Z_k} \leftarrow [0,0,0,0], \quad k \in \{0,\dots,5\}$
 2: $\mathbf{bd_k} \leftarrow \langle B, D\rangle_k, \quad k \in \{0,\dots,5\}$
 3: $\mathbf{bd_k^{c_{19}}} \leftarrow c_{19} \cdot \mathbf{bd_k}, \quad k \in \{0,\dots,5\}$
 4: for $i \in \{0,2,4,6,8,10\}$ do    ▷ Columns with even $a_i$
 5:     $\mathbf{ac_i} \leftarrow [a_i, a_i, c_i, c_i]$    ▷ shuffle
 6:     for $j \leftarrow 0;\ j < \frac{i}{2};\ j \leftarrow j+1$ do    ▷ $\times c_{19}$ case
 7:         $s \leftarrow (j - \frac{i}{2}) \mod 6$
 8:         $\mathbf{Z_j} \leftarrow$ FMA($\mathbf{ac_i},\ \mathbf{bd_s^{c_{19}}},\ \mathbf{Z_j}$)
 9:     end for
10:     for $k \leftarrow \frac{i}{2};\ k \leq 5;\ k \leftarrow k+1$ do    ▷ $\times 1$ case
11:         $s \leftarrow k - \frac{i}{2}$
12:         $\mathbf{Z_k} \leftarrow$ FMA($\mathbf{ac_i},\ \mathbf{bd_s},\ \mathbf{Z_k}$)
13:     end for
14: end for
15: $\mathbf{bd'_k} \leftarrow [b_{2k+1}, b_{2k+2}, d_{2k+1}, d_{2k+2}],\ k \in \{0,\dots,5\}$
16: $\mathbf{bd'^{c_{19}}_k} \leftarrow c_{19} \cdot \mathbf{bd'_k}, \quad k \in \{0,\dots,4\}$
17: $\mathbf{bd'^{c_{19}}_5} \leftarrow [c_{19}b_{11},\ b_0,\ c_{19}d_{11},\ d_0]$
18: for $i \in \{1,3,5,7,9,11\}$ do    ▷ Columns with odd $a_i$
19:     $\mathbf{ac_i} \leftarrow [a_i, a_i, c_i, c_i]$    ▷ shuffle
20:     for $j \leftarrow 0;\ j < \frac{i+1}{2};\ j \leftarrow j+1$ do    ▷ $\times c_{19}$ case
21:         $s \leftarrow (j - \frac{i}{2}) \mod 6$
22:         $\mathbf{Z_j} \leftarrow$ FMA($\mathbf{ac_i},\ \mathbf{bd'^{c_{19}}_s},\ \mathbf{Z_j}$)
23:     end for
24:     for $k \leftarrow \frac{i+1}{2};\ k \leq 5;\ k \leftarrow k+1$ do    ▷ $\times 1$ case
25:         $s \leftarrow k - \frac{i}{2}$
26:         $\mathbf{Z_k} \leftarrow$ FMA($\mathbf{ac_i},\ \mathbf{bd'_s},\ \mathbf{Z_k}$)
27:     end for
28: end for
29: $\langle AB,\ CD\rangle_i \leftarrow \mathbf{Z_i}, \quad i \in 0,\dots,5$
```

from cancellation" ([12, Chapter 2]). Floating point precision loss is cleverly used to extract the high bits of the mantissa. We define the carry $a_i \to a_j$ as follows:

$$\begin{aligned} k &= \lceil 21.25j \rceil \\ c_i &= 3 \cdot 2^{k+53-2} \\ carry_i &= a_i + c_i - c_i \\ a_i &= a_i - carry_i \\ a_j &= a_j + carry_i \end{aligned}$$

We then have: $|a_i| \leq 2^{k-1}$. For a proof of this, see [12, Chapter 2]. This requires four floating point additions and subtractions, because c_i can be simply defined as a constant for all i.

The question remains: what carry chain should be used? The simplest would be $a_0 \to a_1 \to \cdots \to a_{11} \to a_0 \to a_1$, as in previous implementations ([3]). This however has a very high latency cost, especially in double precision arithmetic. One carry operation would take 12 cycles in latency alone (on Skylake). We found it to be best to optimize the carry chain for maximum pipeline usage, at the cost of more operations:

$$\begin{aligned} a_0 &\to a_1 &&\to a_2 &&\to a_3 \\ a_2 &\to a_3 &&\to a_4 &&\to a_5 \\ a_4 &\to a_5 &&\to a_6 &&\to a_7 \\ a_6 &\to a_7 &&\to a_8 &&\to a_9 \\ a_8 &\to a_9 &&\to a_{10} &&\to a_{11} \\ a_{10} &\to a_{11} &&\to a_0 &&\to a_1 \end{aligned}$$

This results in *72 additions and subtractions* for one field carry operation.

Additionally, as shown in algorithm 3, carries are not performed immediately after multiplications, but they are delayed so that four field carries can be executed at the same time.

Inversion. Since the field inversion consists of sequentially executed field multiplications and field squarings, combining multiple multiplications and squarings is not possible. Instead of vectorization, switching to the radix-2^{51} representation for the field inversion gave us the best results, because this representation requires the least operations of all representations.

Operational lower bound. To calculate a lower bound on the operations we use the number of required field operations from section 2 and the operation count for each field operation presented in the previous paragraphs.

The loop and the two CSWAP operations after the loop are performed in radix $2^{21.25}$ and require at least 157080 adds/subs, 24576 bit operations, 64515 muls and 235620 FMAs. This results in a total of 717411 floating point operations (counting FMAs as two operations).

The field inversion and the following field multiplication are performed in radix-2^{51} and require 1596 bit operations, 6916 adds (128 bit), 1596 shifts (128 bit) and 4932 muls (128-bit), or 23552 integer operations (counting 128-bit adds and shifts as two operations).

4.3. Montgomery ladder

We use the ladder structure described in algorithm 3. The following functions were introduced:

- MUL2: Multiply two field elements at the same time in packed format. $\langle A, C\rangle, \langle B, D\rangle \to \langle A \cdot B, C \cdot D\rangle$
- SQUARE2: Square two field elements at the same time in packed format. $\langle A, B\rangle \to \langle A^2, B^2\rangle$
- MULX1: Multiply by x_1. Because x_1 is constant during the ladder, appropriate vectors are prepared (PREPAREX1) for vectorized multiplication at the start of the ladder. Note that this functions reads and stores

its input to the upper part of a tuple, because this is convenient for our ladder design.

- CARRY4: Carry four field elements at the same time, using the carry chain described in the previous section.
- CSWAP2: Swap two tuples in constant time. If swap is 1, the arguments are swapped, if it is 0, they stay the same.

Note that there are many more configurations for vectorization. We found that our approach (Algorithm 3) has good performance, but we cannot claim it is optimal. Some field elements are depicted as $*$, which indicates that these are ignored. There is still some inefficiency inherent to this design.

Cost. This ladder consists of the following operations:

- *12 multiplications, 96 additions*
- 2 MUL2 (2×156 *multiplications,* 2×132 *additions*)
- 2 SQUARE2 (2×96 *multiplications,* 2×66 *addition*)
- 1 MULX1 (*144 multiplications, 132 additions*)
- 3 CARRY4 (4×72 *additions*)

This sums to *1164 multiplications, 1884 additions* for one ladder step. Multiplied by 255, this totally amounts to *296820 multiplications, 480420 additions*, or *777240* floating point operations.

Note that we don't count the final step of the algorithm, the inversion and multiplication (algorithm 1, line 29), as we do it in radix 51 on integers. Effectively, we consider it to be a constant operation.

Algorithm 3 Montgomery ladder optimized for SIMD

```
Input: $n \in \{2^{254} + 8 \cdot [0, 2^{252} - 1]\}$,
       $P \in GF(2^{255} - 19)$
 1: $\langle x_2, z_2 \rangle \leftarrow$ PACK2(1, 0)
 2: $\langle x_3, z_3 \rangle \leftarrow$ PACK2($P$, 1)
 3: $x_1^{prep} \leftarrow$ PREPAREX1($P$)
 4: $swap \leftarrow 0$
 5: for $i \leftarrow 254$ to 0 do
 6:     $bit \leftarrow$ BIT$_i(n)$
 7:     $swap \leftarrow swap \oplus bit$
 8:     $\langle x_2, z_2 \rangle, \langle x_3, z_3 \rangle$
            $\leftarrow$ CSWAP2($swap, \langle x_2, z_2 \rangle, \langle x_3, z_3 \rangle$)
 9:     $swap \leftarrow bit$
10:     $\langle A, B \rangle \leftarrow \langle x_2 + z_2, x_2 - z_2 \rangle$
11:     $\langle D, C \rangle \leftarrow \langle x_3 + z_3, x_3 - z_3 \rangle$
12:     $\langle AA, BB \rangle \leftarrow$ SQUARE2($\langle A, B \rangle$)
13:     $\langle DA, CB \rangle \leftarrow$ MUL2($\langle A, B \rangle, \langle D, C \rangle$)
14:     $\langle AA, BB \rangle, \langle DA, CB \rangle \leftarrow$
            CARRY4($\langle AA, BB \rangle, \langle DA, CB \rangle$)
15:     $\langle t_0, t_1 \rangle \leftarrow \langle DA + CB, DA - CB \rangle$
16:     $\langle t_0^2, t_1^2 \rangle \leftarrow$ SQUARE2($\langle t_0, t_1 \rangle$)
17:     $\langle BB, AA \rangle \leftarrow$ PERMUTE($\langle AA, BB \rangle$)
18:     $\langle *, E \rangle \leftarrow \langle *, AA - BB \rangle$
19:     $\langle *, t_4 \rangle \leftarrow \langle *, E \cdot 121665 + AA \rangle$
20:     $\langle t_0^2, t_1^2 \rangle, \langle *, t_4 \rangle \leftarrow$ CARRY4($\langle t_0^2, t_1^2 \rangle, \langle *, t_4 \rangle$)
21:     $\langle AA, t4 \rangle \leftarrow$ BLEND($\langle AA, BB \rangle, \langle *, t_4 \rangle$)
22:     $\langle BB, E \rangle \leftarrow$ BLEND($\langle BB, AA \rangle, \langle *, E \rangle$)
23:     $\langle x_2, z_2 \rangle \leftarrow$ MUL2($\langle AA, t4 \rangle, \langle BB, E \rangle$)
24:     $\langle *, z_3 \rangle \leftarrow$ MULX1($x_1^{prep}, \langle t_0^2, t_1^2 \rangle$)
25:     $\langle x_2, z_2 \rangle, \langle *, z_3 \rangle \leftarrow$ CARRY4($\langle x_2, z_2 \rangle, \langle *, z_3 \rangle$)
26:     $\langle x_3, z_3 \rangle \leftarrow$ BLEND($\langle t_0^2, t_1^2 \rangle, \langle *, z_3 \rangle$)
27: end for
28: $\langle x_2, z_2 \rangle, \langle x_3, z_3 \rangle$
        $\leftarrow$ CSWAP2($swap, \langle x_2, z_2 \rangle, \langle x_3, z_3 \rangle$)
29: $x_2, z_2 \leftarrow$ UNPACK($\langle x_2, z_2 \rangle$)
30: return $z_2^{-1} \cdot x_2$                ▷ use radix 51 for inversion
```

5. INTEGER IMPLEMENTATION

In this section we describe our implementation using the radix $2^{25.5}$ representation and unsigned 32-bit integers for the limbs.

Our base implementation is a scalar version inspired by *ref10* from SUPERCOP and already contains many optimizations like loop unrolling, precomputation, and so forth.

To vectorize our implementation we used the same approach as described in section 4 and interleaved multiple multiplications/squarings. Note that this is mostly an implementation of [7], but with a different Montgomery ladder.

Two field elements A, B are interleaved as follows:

$$\langle A, B \rangle_i = [a_{2i}, a_{2i+1}, b_{2i}, b_{2i+1}], \quad i \in \{0, \dots, 4\}$$

Multiplication/Squaring. For the integer implementation we used the packed field multiplication and field squaring from [7] and unrolled it.

Coefficient reduction. Like in the floating point implementation, coefficient reductions is done for four field elements in parallel. If signed integers were used for the limbs, the coefficient reduction would require arithmetic shift operations. However, these are not available on AVX2, hence, we used unsigned integers.

Inversion. Like in the floating point implementation, the radix 2^{51} representation is used for the field inversion.

Operational lower bound. The loop and two cswap operations are performed in radix-$2^{25.5}$ and require 58650 adds/subs, 49042 bit operations, 160650 adds/subs (64 bit), 28050 shifts (64 bit), 219810 muls (64 bit), or 516202 integer operations. The field inversion and multiplication performed in radix-51 requires (like for the floating point implementation) adds another 23552 integer operations, which then results in a total of 539754 integer operations.

Montgomery ladder. We use a ladder structure very similar to the one in the floating point implementation (Algorithm 3).

6. EXPERIMENTAL RESULTS

Experimental setup. For our benchmarking, we used the Kaby Lake (i7-7500U) and Haswell (i7-4770K) platform. We evaluated the three most common compilers: gcc (10.1.0), clang (10.0.0) and icc (19.1.0.166). For all of them, the following flags were used: *-std=c99 -march=native -O3 -mavx2 -mfma*.

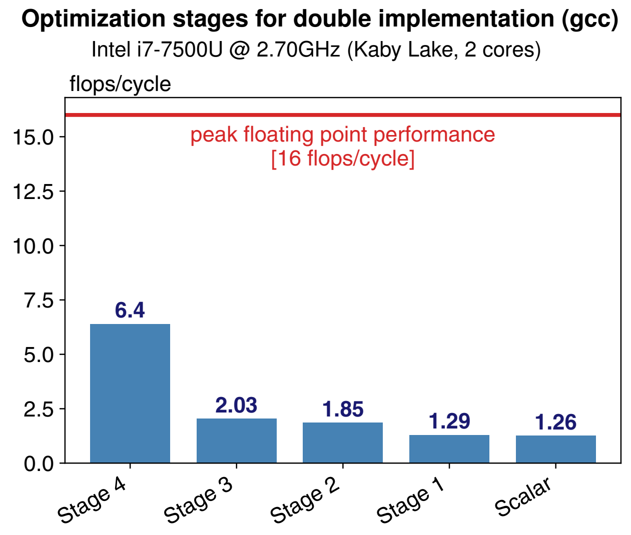

Fig. 2. Floating point performance of our double implementation at different optimization stages.

Optimization stages floating point. To evaluate the effectiveness of different optimizations, we partitioned our implementation into multiple stages, where in every stage more optimizations are added. The expected outcome of this would be to see an incremental improvement in performance. This is confirmed by our measurements (see Figure 2) for the following stages:

- **Scalar:** A basic scalar implementation, without any explicit vectorization.
- **Stage 1:** Field additions, subtractions are vectorized. Additonally in the double implementation, the multiplications with the constant 121655 are vectorized.
- **Stage 2:** Field multiplications and squarings are vectorized using the discussed tuple format.
- **Stage 3:** Some carry operations are performed vectorized.
- **Stage 4:** The SIMD Montgomery ladder is applied, multiplications with the constant x_1 is vectorized and all carry operations are vectorized.

Note that for the figure 2 where the performance of the floating point implementation is displayed, we only consider the floating point computation part. This is because for the last step of the algorithm, the inversion, we change to a radix 2^{51} representation using integer arithmetic. This step is effectively constant for all the stages, so to arrive at the numbers shown, the runtime of the final inversion is deducted.

Also note that we use the basic Montgomery ladder as defined in the RFC [8] up to stage 3. This means, to use our vectorized implementations for multiplication, squaring etc. expensive packing and unpacking has to be performed. These additional operations offset the performance gain. In Stage 4 we finally introduce the Montgomery ladder algorithm that is specifically tuned to work with our data representation (Algorithm 3), eliminating almost all packing and unpacking operations. This left us with a final improvement of stage 4 over the scalar version of 5.1×.

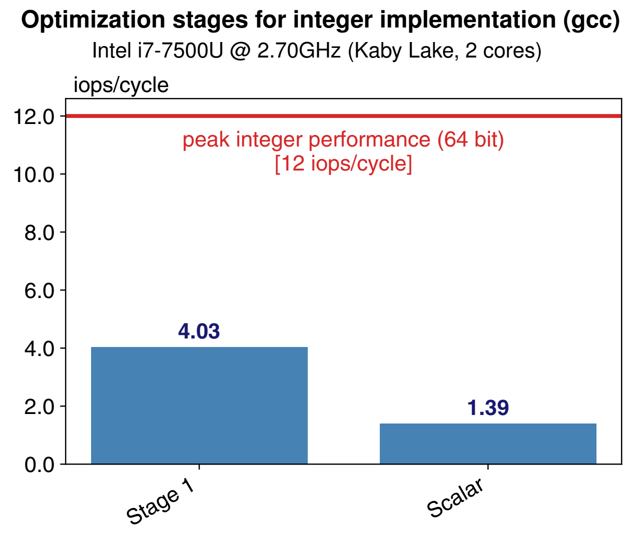

Fig. 3. Integer performance of our integer implementation at different optimization stages.

Optimization stages integer. Figure 3 shows the improvement from the scalar to the fully vectorized version. We omit the intermediate stages here, as they do not increase incrementally. Because we were unable to implement the Montgomery ladder as elegantly as in the floating point version, there is quite a bit more overhead from packing and unpacking tuples. Nevertheless, we got a speedup of about 2.9×.

Benchmarking. Our set goal was to perform better than the fastest implementation in the reference benchmarking suite for cryptographic algorithms, *SUPERCOP*. The two best competitors are *sandy2x* [5] and *amd64-51* [6], which actually have very comparable performance.

Figure 4 and 5 show our measured performance for both

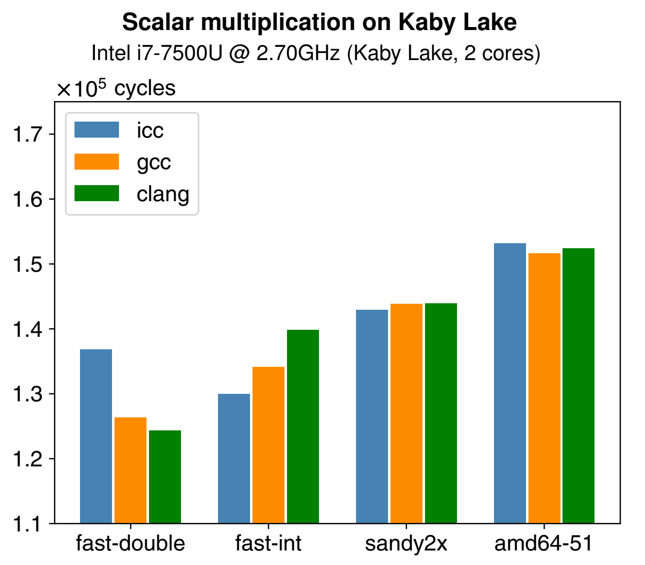

Fig. 4. Runtime comparison for scalar multiplication on Kaby Lake, using different compilers.

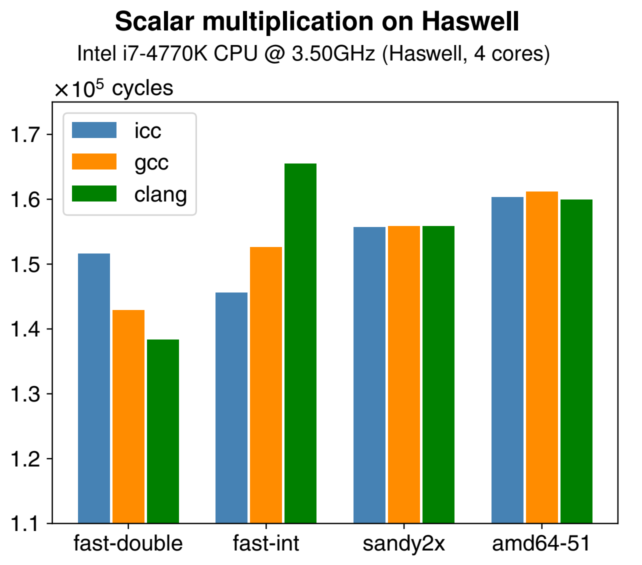

Fig. 5. Runtime comparison for scalar multiplication on Haswell, using different compilers.

our double and integer implementations on the Kaby Lake and Haswell architectures. Even though we focused on optimizing for Skylake, we also achieved good results on Haswell.

The percentage speed improvements are shown in the following table. Note that for these calculations, we only consider the best performing compiler for a specific implementation.

	Kaby Lake		**Haswell**	
	sandy2x	amd64-51	sandy2x	amd64-51
fast-double	13%	18%	11%	13%
fast-int	9%	14%	6%	9%

Notes. It is certainly curious why the compilers generate such different code, because almost the entire code uses Intel intrinsics. *icc* seems to be very good for the integer implementation, but clearly the worst for the floating point version. The references show no difference, because they are almost entirely implemented in assembly. More investigation would be required to find the exact reasons why certain compilers generate better code in our scenario.

7. CONCLUSIONS

We were able to improve upon the fastest algorithms in the SUPERCOP benchmarking suite. A 4-way vectorized version based on 32-bit integer arithmetic was implemented, but it was surpassed in performance by our 4-way vectorized floating point implementation. This shows that the industry standard implementations can still be improved by tuning for specific platforms (like Skylake).

This work shows how floating point FMA units can be applied to cryptographic primitives, and how the Montgomery ladder can be optimized for SIMD architectures.

The next step in this research could be a more rigorous approach to finding an optimal Montgomery ladder, systematically comparing different options.

Another interesting area is the new AVX512 instruction set: It introduces FMA units for integers (The IFMA instruction set). Ideas from our work could be applied there.

8. CONTRIBUTIONS OF TEAM MEMBERS

Fabio implemented radix-2^{51} version using signed integers, investigated vectorization problems with 64 bits limbs and other possiblities to use intrinsics (like addcarry, mulx). Looked into using unsigned integers for arbitrary versions and determined which constant (multiple of field prime) can be added before the field subtraction. Implemented radix-$2^{25.5}$ and radix-2^{51} using unsigned integers for the limbs. Worked with Lukas on the Carry4 for the integer implementation.

Florian implemented a fully generic implementation of the algorithm to be able to generate it for any chosen radix. Looked into which multiplication algorithms could be used and added them to the code generation. Tested combinations of radixes, data types and multiplication algorithms, comparing their operation count and runtime. Implemented the squaring of the integer AVX2 algorithm, discovering a bug in the paper which described the algorithm.

Lukas designed and implemented floating point version, especially the FMA based multiplication/squaring, pipeline-optimized carry operation and SIMD-optimized Montgomery ladder. Separated optimizations in floating point version into stages. Made benchmarking plots with different compilers and optimization stage plots. Made some improvements in the vectorized integer version by applying the SIMD Montgomery ladder to it.

Philippe worked on several different approaches to vectorize the integer implementation. Implemented the unrolled AVX2 version of the multiplication in the integer version. Contributed to the rewrite from the signed AVX2 integer version to the unsigned AVX2 integer version. Implemented the basic AVX2 version of the Montogmery ladder in the integer version. Translated the carry algorithm from the SUPERCOP integer implementation to AVX2.

9. REFERENCES

[1] Aris Adamantiadis, "Openssh introduces curve25519-sha256@libssh.org key exchange !," Accessed: 2020-06-02.

[2] Jason A. Donenfeld, "WireGuard: Next generation kernel network tunnel," in *Proceedings 2017 Network and Distributed System Security Symposium*. 2017, Internet Society.

[3] Daniel J. Bernstein, "Curve25519: New diffie-hellman speed records," in *Public Key Cryptography - PKC 2006*, pp. 207–228. Springer Berlin Heidelberg, 2006.

[4] Daniel J. Bernstein and Peter Schwabe, "NEON crypto," in *Cryptographic Hardware and Embedded Systems – CHES 2012*, pp. 320–339. Springer Berlin Heidelberg, 2012.

[5] Tung Chou, "Sandy2x: New curve25519 speed records," in *Lecture Notes in Computer Science*, pp. 145–160. Springer International Publishing, 2016.

[6] 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, Aug. 2012.

[7] Armando Faz-Hernández and Julio López, "Fast implementation of curve25519 using AVX2," in *Progress in Cryptology – LATINCRYPT 2015*, pp. 329–345. Springer International Publishing, 2015.

[8] S. Turner A. Langley, M. Hamburg, "Rfc 7748," Accessed: 2020-06-02.

[9] Michael Scott, "Missing a trick: Karatsuba variations," *Cryptography and Communications*, vol. 10, no. 1, pp. 5–15, 2018.

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

[11] "Toom-cook 3-way multiplication," ftp://ftp.gnu.org/old-gnu/Manuals/gmp/html_node/Toom-Cook-3-Way-Multiplication.html.

[12] D. J. Bernstein, "Floating-point arithmetic and message authentication," 2001.