High-Throughput Elliptic Curve Cryptography using AVX2 Vector Instructions

Hao Cheng  Johann Großschädl  Jiaqi Tian  Peter B. Rønne  Peter Y. A. Ryan

University of Luxembourg

SAC 2020
SIMD

Single Instruction Multiple Data
## Intel x86/x64 Vector Extensions

| ISA   | Year | SIMD registers |  |
|-------|------|----------------|---|
| MMX   | 1997 | 8 64-bit 64-bit |   |
| SSE   | 1999 | 8 128-bit 128-bit | |
| AVX   | 2011 | 16 256-bit 128-bit |  |
| AVX2  | 2013 | 16 256-bit 256-bit | |
| AVX512| 2016 | 32 512-bit 512-bit | |

Intel® Advanced Vector eXtensions (AVX) series\(^1\) (bottom two rows)

---

\(^1\)figure from [https://www.prowesscorp.com/what-is-intel-avx-512-and-why-does-it-matter/](https://www.prowesscorp.com/what-is-intel-avx-512-and-why-does-it-matter/)
AVX2

Properties

- SIMD fashions: 8-bit \times 32 \quad 16-bit \times 16 \quad 32-bit \times 8 \quad 64-bit \times 4
- Multiplier: 32-bit

\[
\begin{array}{cccccccc}
255 & 223 & 185 & 159 & 127 & 95 & 63 & 31 & 0 \\
\hline
A & a_7 & a_6 & a_5 & a_4 & a_3 & a_2 & a_1 & a_0 \\
\times & & & & & & & & \\
B & b_7 & b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 \\
\hline
\end{array}
\]

\[a_6 \times b_6 \quad a_4 \times b_4 \quad a_2 \times b_2 \quad a_0 \times b_0\]

\mbox{__m256i \_mm256\_mul\_epu32 (__m256i A, __m256i B)}
ECC with SIMD Acceleration

1) Field arithmetic ← limbs
2) Curve arithmetic ← field operations
3) Combination of ← 1) and 2)
4) Mixed use of 1), 2) and 3)
ECC with SIMD Acceleration

1) Field arithmetic ← limbs
2) Curve arithmetic ← field operations
3) Combination of ← 1) and 2)
4) Mixed use of 1), 2) and 3)

1) Accelerate the single addition
   \[ A + B \]

2) Parallel additions
   \[ G + H | E + F | C + D | A + B \]

3) Combination of 1) and 2)
   \[ C + D | A + B \]

( Each \( x_i \) is one limb of the large integer \( X \) )
(n \times m)-Way Parallelism

the number of field operations

the number of elements used by each field operation

\[(n \times m)\]-way
\((n \times m)\)-Way Parallelism

**(1 \times 4)\)-way**

\[
\begin{array}{cccc}
255 & 191 & 127 & 63 & 0 \\
a_{i+3} & a_{i+2} & a_{i+1} & a_i \\
+ & b_{i+3} & b_{i+2} & b_{i+1} & b_i \\
= & a_{i+3} + b_{i+3} & a_{i+2} + b_{i+2} & a_{i+1} + b_{i+1} & a_i + b_i \\
\end{array}
\]

1) Accelerate the single addition

\[A + B\]

**(4 \times 1)\)-way**

\[
\begin{array}{cccc}
255 & 191 & 127 & 63 & 0 \\
g_i & e_i & c_i & a_i \\
+ & h_i & f_i & d_i & b_i \\
= & g_i + h_i & e_i + f_i & c_i + d_i & a_i + b_i \\
\end{array}
\]

2) Parallel additions

\[G + H \mid E + F \mid C + D \mid A + B\]

***(2 \times 2)\)-way***

\[
\begin{array}{cccc}
255 & 191 & 127 & 63 & 0 \\
c_{i+1} & c_i & a_{i+1} & a_i \\
+ & d_{i+1} & d_i & b_{i+1} & b_i \\
= & c_{i+1} + d_{i+1} & c_i + d_i & a_{i+1} + b_{i+1} & a_i + b_i \\
\end{array}
\]

3) Combination of 1) and 2)

\[C + D \mid A + B\]

(Each \(x_i\) is one limb of the large integer \(X\))
Curve25519: $y^2 = x^3 + 486662x^2 + x$

Montgomery Ladder → Variable-base Scalar Multiplication

$F_{2^{255-19}}$ Operations

Point Addition
Point Doubling
Table Query

Ed25519: $-x^2 + y^2 = 1 - 121665/121666x^2y^2$

Fixed-base Scalar Multiplication

Key Generation

Shared Secret
### Low-Latency X25519 using AVX

| Work       | Authors                  | ISA       | Impl.          | Var-base scalar mul. |
|------------|--------------------------|-----------|----------------|-----------------------|
| [Chou15]   | Chou                     | AVX       | (2 × 1)-way    | 137.2 k cycles        |
| [FHLD19]   | Faz-H., López, Dahab      | AVX2      | (2 × 2)-way    | 99.4 k cycles         |
| [HEY20]    | Hisil, Egrice, Yassi     | AVX512    | (4 × 2)-way    | 74.4 k cycles         |
| [NS20]     | Nath, Sarkar              | AVX2 assembly | (4 × 1)-way   | 95.4 k cycles         |
## Latency-Optimized Work

### Low-Latency X25519 using AVX

| Work   | Authors           | ISA      | Impl.     | Var-base scalar mul. |
|--------|-------------------|----------|-----------|----------------------|
| [Chou15] | Chou              | AVX      | (2 × 1)-way | 137.2 k cycles (27.6%) |
| [FHL19]  | Faz-H., López, Dahab | AVX2     | (2 × 2)-way | 99.4 k cycles       |
| [HEY20]  | Hisil, Egrice, Yassi | AVX512   | (4 × 2)-way | 74.4 k cycles       |
| [NS20]   | Nath, Sarkar      | AVX2 assembly | (4 × 1)-way | 95.4 k cycles       |
## Latency-Optimized Work

| Work       | Authors                | ISA      | Impl.       | Var-base scalar mul. |
|------------|------------------------|----------|-------------|----------------------|
| [Chou15]   | Chou                   | AVX      | (2 × 1)-way | 137.2 k cycles       | 27.6%                 |
| [FHL19]    | Faz-H., López, Dahab   | AVX2     | (2 × 2)-way | 99.4 k cycles        | 25.2%                 |
| [HEY20]    | Hisil, Egrice, Yassi  | AVX512   | (4 × 2)-way | 74.4 k cycles        |                       |
| [NS20]     | Nath, Sarkar           | AVX2 assembly | (4 × 1)-way | 95.4 k cycles        |                       |
## Latency-Optimized Work

### Low-Latency X25519 using AVX

| Work   | Authors                  | ISA    | Impl.       | Var-base scalar mul. |
|--------|--------------------------|--------|-------------|----------------------|
| [Chou15]| Chou                     | AVX    | (2 × 1)-way | 137.2 k cycles       |
| [FHL19]| Faz-H., López, Dahab     | AVX2   | (2 × 2)-way | 99.4 k cycles        |
| [HEY20]| Hisil, Egrice, Yassi    | AVX512 | (4 × 2)-way | 74.4 k cycles        |
| [NS20] | Nath, Sarkar             | AVX2 assembly | (4 × 1)-way | 95.4 k cycles        |

*Do not scale very well!*
Throughput v.s. Latency

- How to exploit the massive parallelism of future SIMD extensions?
Throughput v.s. Latency

- How to exploit the massive parallelism of future SIMD extensions?
- Why low-latency implementations?
Throughput v.s. Latency

- How to exploit the massive parallelism of future SIMD extensions?
- Why low-latency implementations?
  - reduces the overall handshake-latency for a TLS client side
Throughput v.s. Latency

- How to exploit the massive parallelism of future SIMD extensions?
- Why low-latency implementations?
  - reduces the overall handshake-latency for a TLS client side

Computation  $\ll$  Transmission!
Throughput v.s. Latency

- **How** to exploit the massive parallelism of future SIMD extensions?

- **Why** low-latency implementations?
  - reduces the overall handshake-latency for a TLS client side

  \[ \text{Computation} \quad \ll \quad \text{Transmission} \]

- **Why** high-throughput implementations?
Why throughput-optimized?

TLS servers of big organizations ← several 10,000 TLS handshakes per second

- Latency ✗
- Throughput ✓
Throughput v.s. Latency

Why throughput-optimized?

TLS servers of big organizations ← several 10,000 TLS handshakes per second

- Latency  
- Throughput  

High throughput instead of low latency?

What throughput can it achieve?
This Work

- Takes first step to answer these questions
- Introduces a throughput-optimized AVX2 implementation of X25519
  - variable-base scalar multiplication on Curve25519
  - fixed-base scalar multiplication on Ed25519
Methodology – (4 × 1)-way scalar multiplication

Perform **FOUR** scalar multiplications simultaneously!
“Coarse-Grained” Parallelism

- Scalar multiplication
- Point arithmetic
- Field arithmetic

\[
\begin{align*}
\text{64-bit element of 256-bit AVX2 vector}
\end{align*}
\]
“Coarse-Grained” Parallelism

- Scalar multiplication
- Point arithmetic
- Field arithmetic

64-bit element of 256-bit AVX2 vector

Advantages

1) Easy to implement
2) Fully exploit parallelism
3) Support various SIMD extensions (straightforward extension to AVX512)
Multi-Precision Representation

Radix-$2^{25.5}$ (e.g. [Chou15], [FHLD19])

\[ f = f_0 + 2^{26} f_1 + 2^{51} f_2 + 2^{77} f_3 + 2^{102} f_4 + 2^{128} f_5 + 2^{153} f_6 + 2^{179} f_7 + 2^{204} f_8 + 2^{230} f_9 \]
## Multi-Precision Representation

| Radix-$2^{25.5}$ (e.g. [Chou15], [FHLD19]) | 
|------------------------------------------------|
| $f = f_0 + 2^{26} f_1 + 2^{51} f_2 + 2^{77} f_3 + 2^{102} f_4 + 2^{128} f_5 + 2^{153} f_6 + 2^{179} f_7 + 2^{204} f_8 + 2^{230} f_9$ |

| Radix-$2^{29}$ (this work) | 
|------------------------------------------------|
| $f = f_0 + 2^{29} f_1 + 2^{58} f_2 + 2^{87} f_3 + 2^{116} f_4 + 2^{145} f_5 + 2^{174} f_6 + 2^{203} f_7 + 2^{232} f_8$ |
Multi-Precision Representation

Radix-$2^{25.5}$ (e.g. [Chou15], [FHLD19])

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

Radix-$2^{29}$ (this work)

$$f = f_0 + 2^{29} f_1 + 2^{58} f_2 + 2^{87} f_3 + 2^{116} f_4 + 2^{145} f_5 + 2^{174} f_6 + 2^{203} f_7 + 2^{232} f_8$$

- (2 × 2)-way both use five limbs
### Multi-Precision Representation

| Radix-$2^{25.5}$ (e.g. [Chou15], [FHLD19]) |
|-------------------------------------------|
| $f = f_0 + 2^{26} f_1 + 2^{51} f_2 + 2^{77} f_3 + 2^{102} f_4 + 2^{128} f_5 + 2^{153} f_6 + 2^{179} f_7 + 2^{204} f_8 + 2^{230} f_9$ |

| Radix-$2^{29}$ (this work) |
|-------------------------------------------|
| $f = f_0 + 2^{29} f_1 + 2^{58} f_2 + 2^{87} f_3 + 2^{116} f_4 + 2^{145} f_5 + 2^{174} f_6 + 2^{203} f_7 + 2^{232} f_8$ |

- (2 x 2)-way both use five limbs
- (4 x 1)-way Radix-$2^{25.5}$ uses ten limbs Radix-$2^{29}$ uses nine limbs
$A = [e, f, g, h] = \left[ \sum_{i=0}^{8} 2^{29i} e_i, \sum_{i=0}^{8} 2^{29i} f_i, \sum_{i=0}^{8} 2^{29i} g_i, \sum_{i=0}^{8} 2^{29i} h_i \right]$

$= \sum_{i=0}^{8} 2^{29i} [e_i, f_i, g_i, h_i] = \sum_{i=0}^{8} 2^{29i} a_i \quad \text{with} \quad a_i = [e_i, f_i, g_i, h_i].$
Arithmetic in $\mathbb{F}_{2^{255} - 19}$

- Modulus $p = 2^6 \cdot (2^{255} - 19) \leftrightarrow 29$-bit $\times 9 = 261$-bit
Arithmetic in $\mathbb{F}_{2^{255}-19}$

- Modulus $p = 2^6 \cdot (2^{255} - 19) \leftarrow 29$-bit $\times 9 = 261$-bit

- Addition $\rightarrow$ ordinary integer addition $r = a + b$
Arithmetic in $\mathbb{F}_{2^{255}}$  

- **Modulus**  
  \[ p = 2^6 \cdot (2^{255} - 19) \quad \leftarrow \quad 29\text{-bit} \times 9 = 261\text{-bit} \]

- **Addition**  
  \[ r = a + b \]

- **Subtraction**  
  - ordinary subtraction  
    \[ r = 2p + a - b \]
  - modular subtraction  
    \[ r = 2p + a - b \mod p \]
Arithmetic in $\mathbb{F}_{2^{255} - 19}$

- **Modulus** $p = 2^6 \times (2^{255} - 19) \leftarrow 29$-bit $\times 9 = 261$-bit

- **Addition** $\rightarrow$ ordinary integer addition $r = a + b$

- **Subtraction**
  - ordinary subtraction $r = 2p + a - b$
  - modular subtraction $r = 2p + a - b \mod p$

- **Multiplication** $r = a \times b \mod p$
Arithmetic in $\mathbb{F}_{2^{255}-19}$

- **Modulus** \( p = 2^6 \cdot (2^{255} - 19) \) \(\leftarrow\) \(29\)-bit$\times$9 = 261-bit

- **Addition** \(\rightarrow\) ordinary integer addition \( r = a + b \)

- **Subtraction**
  - ordinary subtraction \( r = 2p + a - b \)
  - modular subtraction \( r = 2p + a - b \mod p \)

- **Multiplication** \( r = a \times b \mod p \)

- **Squaring** \(\rightarrow\) special multiplication \( r = a^2 = a \times a \mod p \)
Field Multiplication

Design Principles

- Make full use of execution ports
- Reduce the sequential dependencies
Field Multiplication

Design Principles

- Make full use of execution ports
- Reduce the sequential dependencies

a dozen of candidates → benchmark → the lowest latency 1
# Field Multiplication

## Design Principles
- Make full use of execution ports
- Reduce the sequential dependencies

| a dozen of candidates | benchmark | the lowest latency |
|-----------------------|-----------|-------------------|

## Distinctions of candidates
1. Reduction & multiplication → **separated** or **interleaved**?
2. Different carry propagation plans
3. Intermediate values → local variables?
Field Multiplication

```c
#include <immintrin.h>
#define ADD(X,Y) _mm256_add_epi64(X,Y) /* VPADDQ */
#define MUL(X,Y) _mm256_mul_epi32(X,Y) /* VPADDLQ */
#define AND(X,Y) _mm256_and_si256(X,Y) /* VPAND */
#define SRL(X,Y) _mm256_srl_epi64(X,Y) /* VPSRLQ */
#define BCAST(X) _mm256_set1_epi64x(X) /* VPBROADCASTQ */
#define MASK29 0xffffffff /* mask of 29 LSBs */

void fp_mul(__m256i *r, const __m256i *a, const __m256i *b) {
    int i, j, k; __m256i t[8], accu;

    /* 1st loop of the product-scanning multiplication */
    for (i = 0; i < 9; i++) {
        t[i] = BCAST(0);
        for (j = 0, k = i; k >= 0; j++, k--)
            t[i] = ADD(t[i], MUL(a[j], b[k]));
    }
    accu = SRL(t[8], 29);
    t[8] = AND(t[8], BCAST(MASK29));

    /* 2nd loop of the product-scanning multiplication */
    for (i = 9; i < 17; i++) {
        for (j = i-8, k = 8; j < 9; j++, k--)
            accu = ADD(accu, MUL(a[j], b[k]));
        r[1-9] = AND(accu, BCAST(MASK29));
        accu = SRL(accu, 29);
    }
    r[8] = accu;

    /* Modulo reduction and conversion to 29-bit limbs */
    accu = BCAST(0);
    for (i = 0; i < 9; i++) {
        accu = ADD(accu, MUL(r[i], BCAST(64*19)));
        accu = ADD(accu, t[i]);
        r[i] = AND(accu, BCAST(MASK29));
        accu = SRL(accu, 29);
    }

    /* limbs in r[0] can finally be 30 bits long */
    r[0] = ADD(r[0], MUL(accu, BCAST(64*19)));
}
```
Take advantage of two types of field subtraction!

```c
void point_add(ExtPoint *R, ExtPoint *P, ProPoint *Q)
{
    __m256i t[9];

    fp_mul(t, P->e, P->h);        /* T = E_p * H_p */
    fp_sub(R->e, P->y, P->x);     /* E_H = Y_p - X_p */
    fp_add(R->h, P->y, P->x);     /* H_R = Y_p + X_p */
    fp_mul(R->x, R->e, Q->y);     /* X_R = E_R * Y_Q */
    fp_mul(R->y, R->h, Q->x);     /* Y_R = H_R * X_Q */
    fp_sub(R->e, R->y, R->x);     /* E_R = Y_R - X_R */
    fp_add(R->h, R->y, R->x);     /* H_R = Y_R + X_R */
    fp_mul(R->x, t, Q->z);        /* X_R = T * Z_Q */
    fp_sbc(t, P->z, R->x);        /* T = Z_p - X_R */
    fp_add(R->x, P->z, R->x);     /* X_R = Z_p + X_R */
    fp_mul(R->z, t, R->x);        /* Z_R = T * X_R */
    fp_mul(R->y, R->x, R->h);     /* Y_R = X_R * H_R */
    fp_mul(R->x, R->e, t);        /* X_R = E_R * T */
    fp_mul(R->x, R->e, t);
}
```
Measurement Environment

Platform

- a **Haswell** Intel® Core™ i7-4710HQ CPU clocked at 2.5 GHz
- a **Skylake** Intel® Core™ i5-6360U CPU clocked at 2.0 GHz

- Compiler  Clang 10.0.0

- Disabled Features
  - Intel® Turbo Boost  \( \times \)
  - Intel® Hyper-Threadin\( \times \)
# Performance Evaluation

**CPU cycles of \((4 \times 1)\)-way field and point arithmetic**

| Domain               | Operation          | [FHLD19] | This Work |
|----------------------|--------------------|----------|-----------|
|                      |                    | Haswell  | Skylake   | Haswell  | Skylake   |
| \(\mathbb{F}_{2^{255}-19}\) | Addition           | 12       | 12        | 11       | 11        |
|                      | Ord. Subtraction   | n/a      | n/a       | 14       | 12        |
|                      | Mod. Subtraction   | n/a      | n/a       | 32       | 31        |
|                      | Multiplication     | 159      | 105       | 122      | 88        |
|                      | Squaring           | 114      | 85        | 87       | 65        |
| twisted Edwards      | Point Addition     | 1096     | 833       | 965      | 705       |
| curve                | Point Doubling     | n/a      | n/a       | 830      | 624       |
|                      | Table Query        | 208      | 201       | 218      | 205       |
| Montgomery curve     | Ladder Step        | n/a      | n/a       | 1118     | 818       |
Performance Evaluation

| Platform | CPU Frequency | Key Generation | Shared Secret | Table Size |
|----------|---------------|----------------|---------------|------------|
|          |               | Latency | Throughput | Latency | Throughput | |
| Haswell  | 2.5 GHz       | 104,579 cycles | 95,568 ops/sec | 329,455 cycles | 30,336 ops/sec | 24 kB |
| Skylake  | 2.0 GHz       | 80,249 cycles  | 99,363 ops/sec | 246,636 cycles | 32,318 ops/sec | 24 kB |

30% stronger on Skylake than on Haswell
## Comparison on Haswell – 2.5 GHz

| Work            | Impl.            | CPU         | Compiler   | Key Generation |                |                | Shared Secret |                |
|-----------------|------------------|-------------|------------|----------------|----------------|----------------|---------------|----------------|
|                 |                  |             |            | Latency  [cycles] | Throughput  [ops/sec] | Latency  [cycles] | Throughput  [ops/sec] |
| [FHLD19]        | (2 × 2)-way      | i7-4770     | Clang 5.0.2 | 43,700          | 57,208         | 121,000        | 20,661        |
|                 | (2 × 2)-way      | i7-4710HQ   | Clang 10.0.0 | 41,938          | 59,575         | 121,499        | 20,563        |
| [NS20]          | (4 × 1)-way      | i7-6500U    | GCC 7.3.0  | 100,127         | 24,968         | 120,108        | 20,815        |
|                 | (4 × 1)-way      | i7-4710HQ   | GCC 8.4.0  | 100,669         | 24,820         | 120,847        | 20,676        |
| **This work**   | (4 × 1)-way      | i7-4710HQ   | Clang 10.0.0 | 104,579         | **95,568**     | 329,455        | **30,336**    |
|                 |                  |             |            |                | 60.4%          |                | 45.7%         |
## Comparison on Skylake – 2.0 GHz

| Work      | Impl.     | CPU       | Compiler  | Key Generation | Shared Secret |
|-----------|-----------|-----------|-----------|----------------|---------------|
|           |           |           |           | Latency [cycles] | Throughput [ops/sec] | Latency [cycles] | Throughput [ops/sec] |
| [FHL19]   | (2 x 2)-way | i7-6700K  | Clang 5.0.2 | 34,500          | 57,971         | 99,400          | 20,150          |
|           | (2 x 2)-way | i5-6360U  | Clang 10.0.0 | 35,629         | 55,955         | 95,129          | 20,939          |
| [HEY20]   | (4 x 1)-way | i9-7900X  | GCC 5.4    | n/a            | n/a            | 98,484          | 20,308          |
|           | (4 x 1)-way | i5-6360U  | GCC 8.4.0  | n/a            | n/a            | 116,595         | 16,656          |
| [NS20]    | (4 x 1)-way | i7-6500U  | GCC 7.3.0  | 84,047         | 23,796         | 95,437          | 20,956          |
|           | (4 x 1)-way | i5-6360U  | GCC 8.4.0  | 82,054         | 24,406         | 93,657          | 21,168          |
| This work | (4 x 1)-way | i5-6360U  | Clang 10.0.0 | 80,249        | 246,636        | 99,363          | 32,318          |

This work: 71.4% 52.7%
AVX2 offers great potential to optimize ECC

The first to use AVX2 to maximize throughput

1.5x ~ 1.7x throughput compared to the state of the art

Straightforward extension to AVX512
Future Work

- Support AVX512
- Isogeny-based cryptography
Source code at
https://gitlab.uni.lu/APSIA/AVXECC

Thank you for your attention!