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

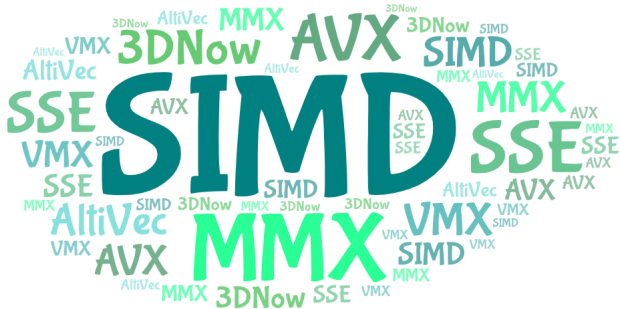
SAC 2020

UNIVERSITÉ DU LUXEMBOURG

securityandtrust.lu

FutureTPM

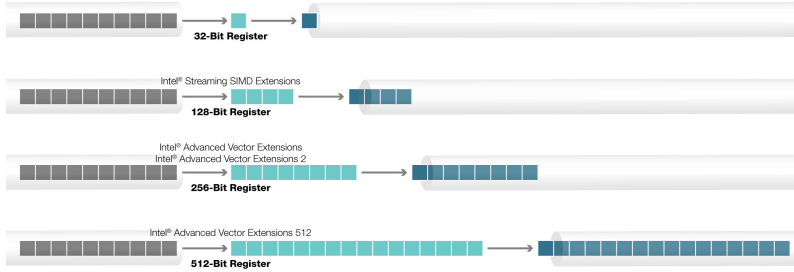
SIMD



Single **I**nstruction **M**ultiple **D**ata

Intel x86/x64 Vector Extensions

ISA	Year	SIMD registers		
		#	FP	Integer
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



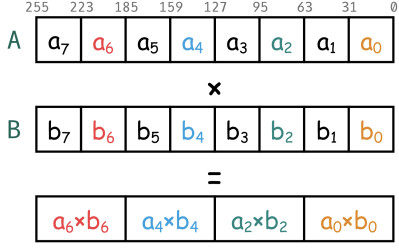
Intel® **A**dvanced **V**ector e**X**tensions (AVX) series[1] (bottom two rows)

[1]figure from `https://www.prowesscorp.com/what-is-intel-avx-512-and-why-does-it-matter/`

AVX2

Properties

- SIMD fashions : 8-bit × 32 16-bit × 16 32-bit × 8 64-bit × 4
- Multiplier : 32-bit



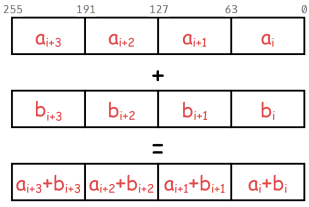

```
__m256i _mm256_mul_epu32 (__m256i A, __m256i B)
```

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)

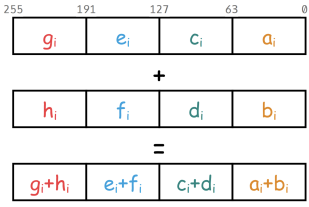
ECC with SIMD Acceleration

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



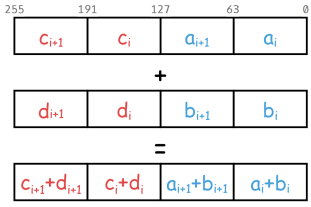
1) Accelerate the single addition

A + B



2) Parallel additions

G + H | E + F | C + D | A + B

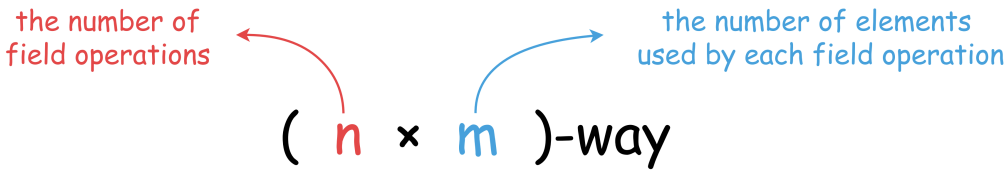


3) Combination of 1) and 2)

C + D | A + B

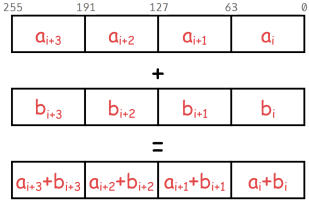
(Each x_i is one limb of the large integer X)

$(n \times m)$-Way Parallelism



$(n \times m)$-Way Parallelism

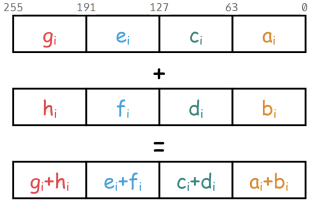
(1×4)-way



1) Accelerate the single addition

$A + B$

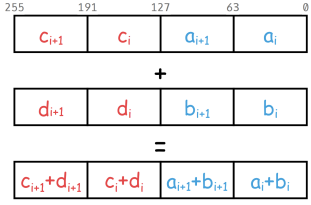
(4×1)-way



2) Parallel additions

$G + H$ | $E + F$ | $C + D$ | $A + B$

(2×2)-way

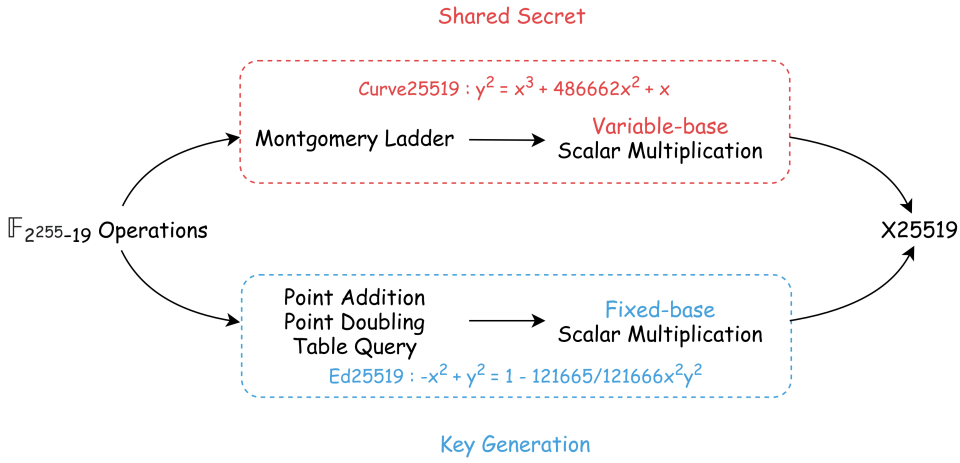


3) Combination of 1) and 2)

$C + D$ | $A + B$

(Each x_i is one limb of the large integer X)

X25519


Shared Secret
Curve25519 : $y^2 = x^3 + 486662x^2 + x$
Montgomery Ladder
Variable-base
Scalar Multiplication
$\mathbb{F}_{2^{255}-19}$ Operations
X25519
Point Addition
Point Doubling
Table Query
Fixed-base
Scalar Multiplication
Ed25519 : $-x^2 + y^2 = 1 - 121665/121666x^2y^2$
Key Generation

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
[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%
[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%
[FHLD19]	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	27.6%
[FHLD19]	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	

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

  Computation $\ll$ Transmission !
- Why **high-throughput** implementations?

Throughput v.s. Latency

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

} 64-bit element of 256-bit AVX2 vector

“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])

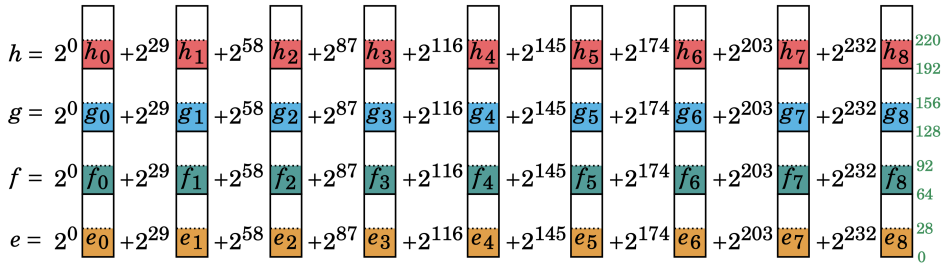
$$f = f_0 + 2^{26} f_1 + 2^{51} f_2 + 2^{77} f_3 + 2^{102} f_4 + 2^{128} f_5 + 2^{153} f_6 + 2^{179} f_7 + 2^{204} f_8 + 2^{230} f_9$$

Radix-2^{29} (this work)

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

- (2×2)-way — both use five limbs
- (4×1)-way — Radix-$2^{25.5}$ uses ten limbs
  Radix-2^{29} uses nine limbs

Field Element Vector Set



$$\boldsymbol{A} = [e,f,g,h] = [\sum_{i=0}^{8} 2^{29i} e_i, \sum_{i=0}^{8} 2^{29i} f_i, \sum_{i=0}^{8} 2^{29i} g_i, \sum_{i=0}^{8} 2^{29i} h_i]$$

$$= \sum_{i=0}^{8} 2^{29i} [e_i, f_i, g_i, h_i] = \sum_{i=0}^{8} 2^{29i} \boldsymbol{a_i} \quad \text{with} \quad \boldsymbol{a_i} = [e_i, f_i, g_i, h_i].$$

Arithmetic in $\mathbb{F}_{2^{255}-19}$

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

Arithmetic in $\mathbb{F}_{2^{255}-19}$

- Modulus $p = 2^6 \cdot (2^{255} - 19)$ ← 29-bit×9 = 261-bit
- Addition → ordinary integer addition $r = a + b$

Arithmetic in $\mathbb{F}_{2^{255}-19}$

- Modulus $p = 2^6 \cdot (2^{255} - 19)$ ← $29\text{-bit} \times 9 = 261\text{-bit}$
- Addition → ordinary integer addition $r = a + b$
- Subtraction
  - ordinary subtraction $r = 2p + a - b$
  - modular subtraction $r = 2p + a - b \bmod p$

Arithmetic in $\mathbb{F}_{2^{255}-19}$

- Modulus $p = 2^6 \cdot (2^{255} - 19)$ ← 29-bit×9 = 261-bit
- Addition → ordinary integer addition $r = a + b$
- Subtraction
  - ordinary subtraction $r = 2p + a - b$
  - modular subtraction $r = 2p + a - b \bmod p$
- **Multiplication** $r = a * b \bmod p$

Arithmetic in $\mathbb{F}_{2^{255}-19}$

- Modulus $p = 2^6 \cdot (2^{255} - 19)$ ← 29-bit×9 = 261-bit
- Addition → ordinary integer addition $r = a + b$
- Subtraction
  - ordinary subtraction $r = 2p + a - b$
  - modular subtraction $r = 2p + a - b \bmod p$
- **Multiplication** $r = a * b \bmod p$
- Squaring → special multiplication $r = a^2 = a * a \bmod p$

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 

Field Multiplication

Design Principles

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

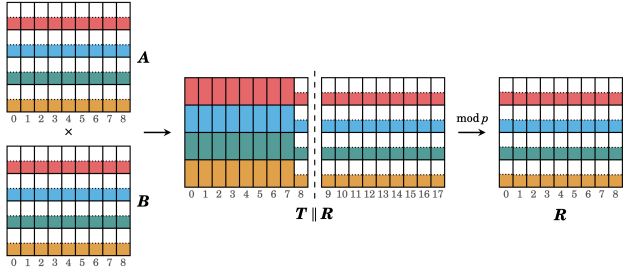
a dozen of candidates → benchmark → the lowest latency 

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_epu32(X,Y)   /* VPMULUDQ */
#define AND(X,Y) _mm256_and_si256(X,Y)   /* VPAND */
#define SRL(X,Y) _mm256_srli_epi64(X,Y) /* VPSRLQ */
#define BCAST(X) _mm256_set1_epi64x(X)   /* VPBROADCASTQ */
#define MASK29   0x1fffffff              /* mask of 29 LSBs */

void fp_mul(__m256i *r, const __m256i *a, const __m256i *b)
{
  int i, j, k; __m256i t[9], 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[i-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)));
}
```



Point Arithmetic

Take advantage of two types of field subtraction !

```
1  void point_add(ExtPoint *R, ExtPoint *P, ProPoint *Q)
2  {
3    __m256i t[9];
4
5    fp_mul(t, P->e, P->h);              /* T = E_P × H_P   */
6    fp_sub(R->e, P->y, P->x);           /* E_R = Y_P − X_P */
7    fp_add(R->h, P->y, P->x);           /* H_R = Y_P + X_P */
8    fp_mul(R->x, R->e, Q->y);           /* X_R = E_R × Y_2 */
9    fp_mul(R->y, R->h, Q->x);           /* Y_R = H_R × X_2 */
10   fp_sub(R->e, R->y, R->x);           /* E_R = Y_R − X_R */
11   fp_add(R->h, R->y, R->x);           /* H_Y = Y_R + X_R */
12   fp_mul(R->x, t, Q->z);              /* X_R = T × Z_2   */
13   fp_sbc(t, P->z, R->x);              /* T = Z_P − X_R   */
14   fp_add(R->x, P->z, R->x);           /* X_R = Z_P + X_R */
15   fp_mul(R->z, t, R->x);              /* Z_R = T × X_R   */
16   fp_mul(R->y, R->x, R->h);           /* Y_R = X_R × H_R */
17   fp_mul(R->x, R->e, t);              /* X_R = E_R × T   */
18 }
```

Point addition on Ed25519

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 ✗
  - Intel® Hyper-Threading ✗

Performance Evaluation

CPU cycles of (4×1)-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 curve	Point Addition	1096	833	965	705
	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** 60.4%	329,455	**30,336** 45.7%

Comparison on Skylake – 2.0 GHz

Work	Impl.	CPU	Compiler	Key Generation Latency [cycles]	Key Generation Throughput [ops/sec]	Shared Secret Latency [cycles]	Shared Secret Throughput [ops/sec]
[FHLD19]	(2×2)-way	i7-6700K	Clang 5.0.2	34,500	57,971	99,400	20,150
	(2×2)-way	i5-6360U	Clang 10.0.0	35,629	55,955	95,129	20,939
[HEY20]	(4×1)-way	i9-7900X	GCC 5.4	n/a	n/a	98,484	20,308
	(4×1)-way	i5-6360U	GCC 8.4.0	n/a	n/a	116,595	16,656
[NS20]	(4×1)-way	i7-6500U	GCC 7.3.0	84,047	23,796	95,437	20,956
	(4×1)-way	i5-6360U	GCC 8.4.0	82,054	24,406	93,657	21,168
This work	(4×1)-way	i5-6360U	Clang 10.0.0	80,249	**99,363** 71.4%	246,636	**32,318** 52.7%

Conclusion

- 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!