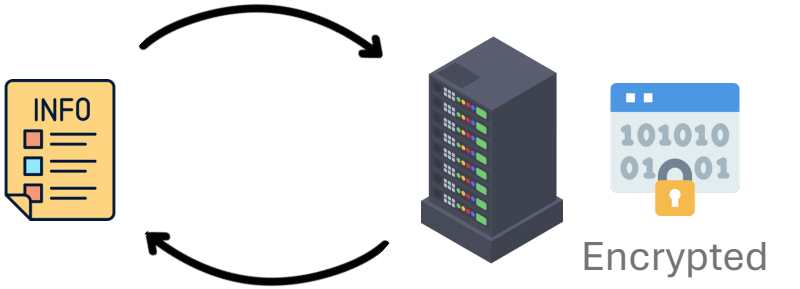
Carnegie Mellon University

Towards Closing the Performance Gap for Cryptographic Kernels Between CPUs and Specialized Hardware

Naifeng Zhang, Sophia Fu, Franz Franchetti

MICRO 2025

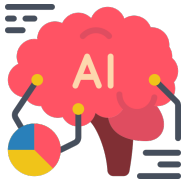
Great Data Security Comes at a High Cost



Encrypted Computation

Fully homomorphic encryption (FHE)

Healthcare



Artificial Intelligence

Finance

Cost: Prohibitive computational overhead

Cryptographic Operations with LARGE Integer Arithmetic

2,000-bit integer

29945058656012390289752201899548961002216938026898764670213921626494125002097316037309083423551230
90801402354741281525320130875756031797076046167772169925045685236286727961252942656722365196302071
16603324681021157993534366932467361582713800585917357036433047500540160547946659133404014756269446
50403492615533104586108008512036436642324307076750180870819991866737106993558380421464216659826172
30815293083164172838554002986172170145077240914636310277685115293248536940258787378968137743709119
53496219555899915324743599705479002269036442663447672197590995176588031386455684062123400247497680
58860138217379

Residue Number System (RNS)

32-bit 305,419,896

64-bit 17,920,781,457,621,970,241

128-bit 15,107,846,090,143,992,465,023,504,163,010,990,279

Polynomial Operations: Large Integer Arithmetic in Action

Polynomial addition over a finite field $\mathbb{Z}_q$: $c_i = a_i + b_i \; mod \; q$

$$\begin{array}{rl} & a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n \\ \odot - + & b_0 + b_1 x + b_2 x^2 + \cdots + b_n x^n \\ \hline & c_0 + c_1 x + c_2 x^2 + \cdots + c_n x^n \end{array}$$

If q has 128 bits

$$105847670885136696452273223349042939 61 + 62446778956155594592553396455561501 49 \; mod \; 151078460901439924650235041630109902 79$$

Cryptographic Kernel I: BLAS(-Like) Operations

Polynomial addition (over $\mathbb{Z}_q$)

$c_i = a_i + b_i \; mod \; q$

$a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$

↳ $[a_0, a_1, a_2, \cdots, a_n]$

Polynomial subtraction

***Point-wise* polynomial multiplication**

Vector addition

$c_i = a_i + b_i \; mod \; q$

$[c_0, c_1, c_2, \cdots, c_n] =$
$[a_0, a_1, a_2, \cdots, a_n] + [b_0, b_1, b_2, \cdots, b_n]$

Vector subtraction

***Point-wise* vector multiplication**

Basic Linear Algebra Subprograms (BLAS)

Cryptographic Kernel II: Number Theoretic Transform

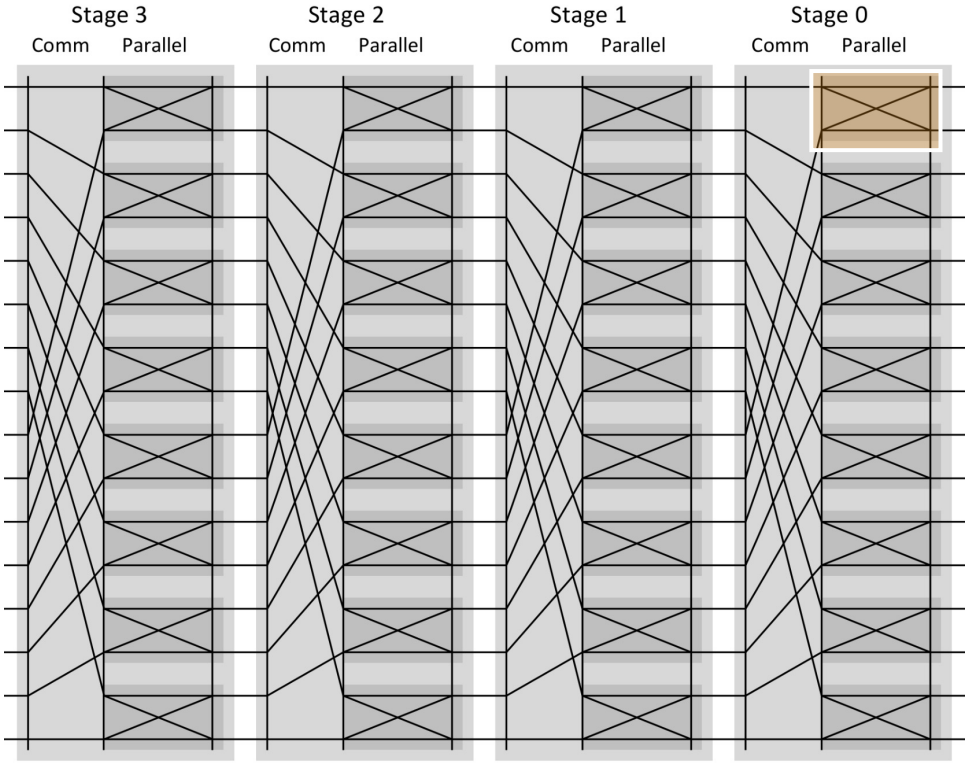
Polynomial multiplication

- Schoolbook multiplication takes $O(n^2)$

$$\begin{array}{rl} & a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n \\ \times & b_0 + b_1 x + b_2 x^2 + \cdots + b_n x^n \\ \hline & c_0 + c_1 x + c_2 x^2 + \cdots + c_n x^n \end{array}$$

Number Theoretic Transform (NTT): $O(n \log n)$

NTT, the Butterfly, and MORE Large Integer Arithmetic



Pease NTT algorithm

Butterfly

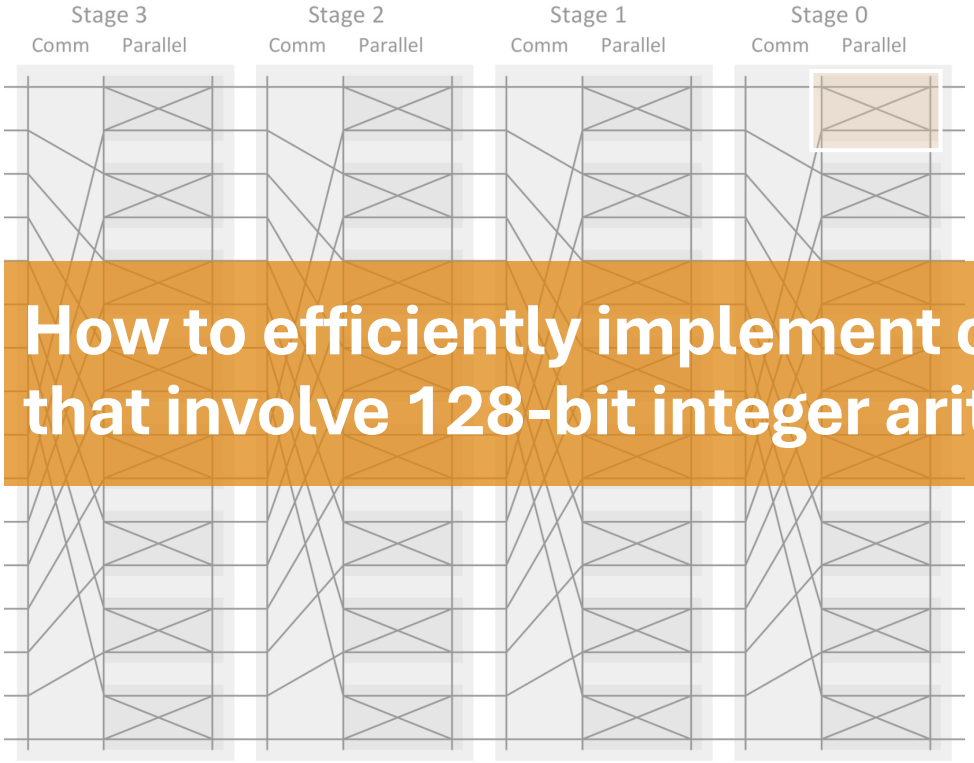
- 1 modular addition
- 1 modular subtraction
- 1 modular multiplication

...on large integers

$$105847670885136696452273223349042939 61 + 624467789561555945925533964555615014 9 \bmod 151078460901439924650235041630109902 79$$

>90% runtime for FHE-based workloads

NTT, the Butterfly, and MORE Large Integer Arithmetic



Pease NTT algorithm

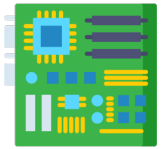
Butterfly

- 1 modular addition
- 1 modular subtraction
- 1 modular multiplication

How to efficiently implement cryptographic kernels that involve 128-bit integer arithmetic?

>90% runtime for FHE-based workloads

State-of-the-Art Solutions



Specialized hardware support on **application-specific integrated circuits (ASICs)**

Code generation-based approach on **GPUs**

- Multi-word modular arithmetic (MoMA)

Arbitrary precision **libraries** on **CPUs**

- GNU multiple precision (GMP) library
- OpenFHE MATHBACKEND

1,157x slowdown!

[ISPASS'23] D. Soni et al. "RPU: The Ring Processing Unit"

Part I: Modular Arithmetic

Math (over $\mathbb{Z}_q$)

$$c = a + b \mod q$$

$$c = a - b \mod q$$

$$c = ab \mod q$$

Algorithm

$$c = \begin{cases} a + b - q, & \text{if } (a + b) \geq q, \\ a + b, & \text{otherwise.} \end{cases}$$

$$c = \begin{cases} a - b + q, & \text{if } a < b, \\ a - b, & \text{otherwise.} \end{cases}$$

$$c = ab - \lfloor ab\mu/2^k \rfloor q, \quad \mu = \lfloor 2^k/q \rfloor$$

Barrett reduction

Part II: Double-Word Arithmetic

Double-word representation: $[x_0, x_1]_z = x_0 z + x_1 = x$

$z = 2^{64}$

Examples

$[8,9]_{10} = 8 \cdot 10 + 9 = 89$

$[11529215046068469 75,18446744073709550897]_{2^{64}}$
$= 212676479325586539664609129644855124 97$

Part II: Double-Word Arithmetic

Double-word representation: $[x_0, x_1]_z = x_0 z + x_1 = x$ $\quad z = 2^{64}$

Examples

$[8,9]_{10} = 8 \cdot 10 + 9 = 89$

$[11529215046068469 75,18446744073709550897]_{2^{64}}$
$= 212676479325586539664609129644855124 97$

Part I: Modular addition algorithm

$$c = \begin{cases} a + b - q, & \text{if } (a + b) > q, \\ a + b, & \text{otherwise.} \end{cases}$$

$$a = [a_0, a_1]_z = a_0 z + a_1$$
$$b = [b_0, b_1]_z = b_0 z + b_1$$

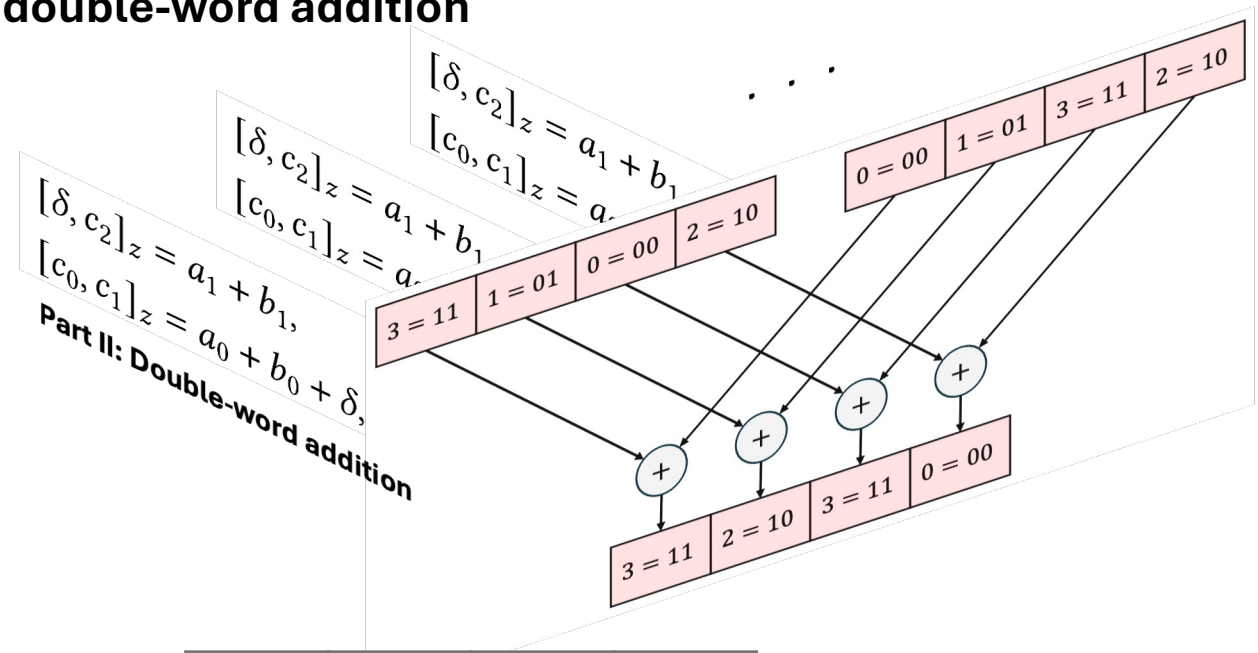
Part II: Double-word addition

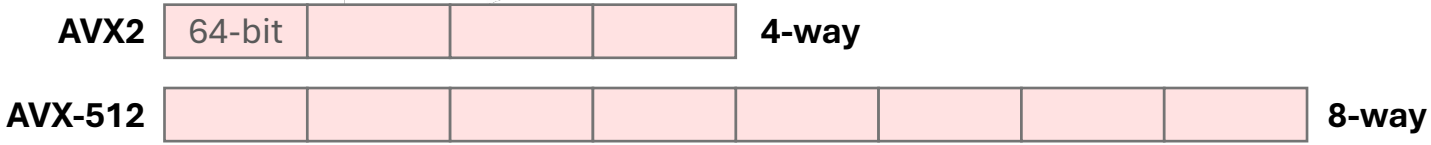
$$[\delta, c_2]_z = a_1 + b_1,$$
$$[c_0, c_1]_z = a_0 + b_0 + \delta,$$

where $c = [c_0, c_1, c_2]_z$ and $\delta \in \{0, 1\}$.

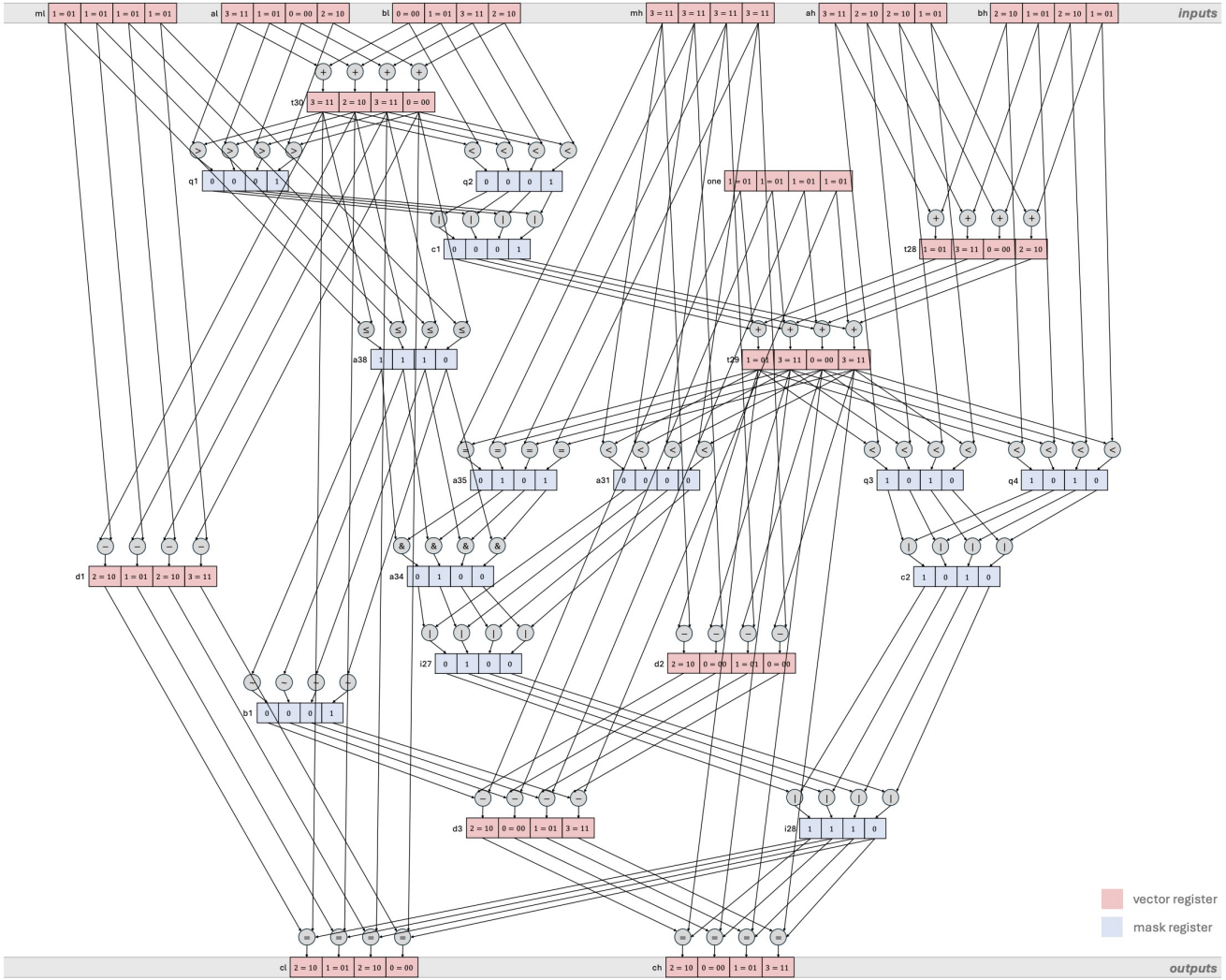
Part III: Single Instruction, Multiple Data

SIMD double-word addition

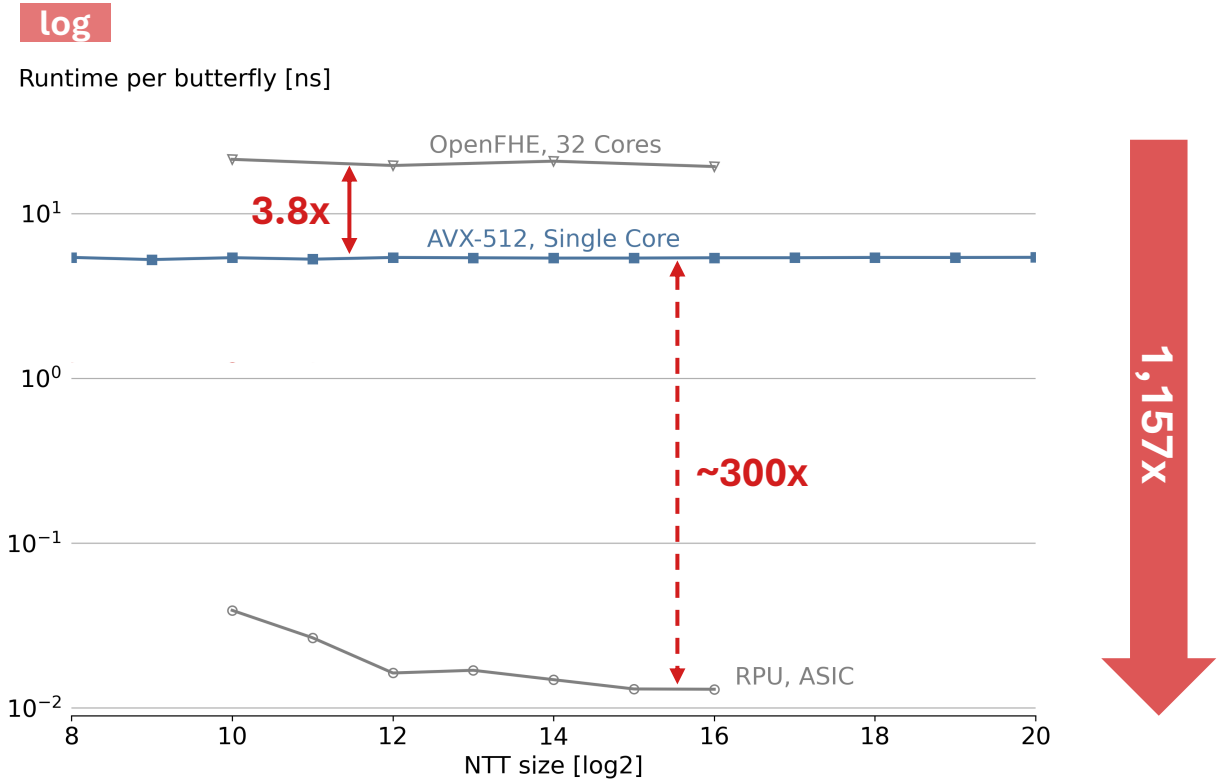




SIMD Double-word Modular Addition



Progress So Far: vs. Prior Work



Towards Better Performance

ADC

Addition with carry

$$[c_0, c_1]_z = a_0 + b_0 + \delta$$

SIMD ADC

Scalar

```
1  // In: uint64_t a, b; _Bool ci
2  // Out: uint64_t c; _Bool co
3  uint64_t t0, t1; _Bool q0, q1;
4  t0 = a + b;
5  t1 = t0 + ci;
6  q0 = (t1 < a);
7  q1 = (t1 < b);
8  co = q0 || q1;
```

AVX-512

```
1  // In: __m512i a, b; __mmask8 ci
2  // Out: __m512i c; __mmask8 co
3  __m512i t0, t1, one; __mmask8 q0, q1;
4  t0 = _mm512_add_epi64(a, b);
5  one = _mm512_set1_epi64(1);
6  t1 = _mm512_mask_add_epi64 (t0, ci, t0, one);
7  q0 = _mm512_cmp_epu64_mask(t1, a, _MM_CMPINT_LT);
8  q1 = _mm512_cmp_epu64_mask(t1, b, _MM_CMPINT_LT);
9  co = q0 | q1;
```

MQX

```
1  // In: __m512i a, b; __mmask8 ci
2  // Out: __m512i c; __mmask8 co
3  c = _mm512_adc_epi64(a, b, ci, &co);
```

Design Philosophy of MQX

- Minimizing required hardware engineering effort
- Only 3 new instructions
- Grounded in existing x86 and Intel-explored SIMD instructions

Larrabee New Instructions (LRBni)

Knights Corner (KNC) intrinsics

```
void addmod128(__m512i* ch, __m512i* cl, __m512i ah, __m512i al,
               __m512i bh, __m512i bl, __m512i mh, __m512i ml) {
    __m512i t30, t28, t29, d1, d2, d3;
    __mmask8 q1, q2, c1, q3, q4, c2, a31, a35, ...;
    t30 = _mm512_add_epi64 (al, bl);
    q1 = _mm512_cmp_epu64_mask(t30, al, _MM_CMPINT_LT);
    q2 = _mm512_cmp_epu64_mask(t30, bl, _MM_CMPINT_LT);
    c1 = q1 | q2;
    t28 = _mm512_add_epi64 (ah, bh);
    t29 = _mm512_mask_add_epi64 (t28, c1, t28, one);
    q3 = _mm512_cmp_epu64_mask(t29, ah, _MM_CMPINT_LT);
    q4 = _mm512_cmp_epu64_mask(t29, bh, _MM_CMPINT_LT);
    c2 = q3 | q4;
    a31 = _mm512_cmp_epu64_mask(mh, t29, _MM_CMPINT_LT);
    a35 = _mm512_cmp_epu64_mask(mh, t29, _MM_CMPINT_EQ);
    a38 = _mm512_cmp_epu64_mask(ml, t30, _MM_CMPINT_LE);
    a34 = a35 & a38; i27 = a31 | a34; i28 = c2 | i27;
    d1 = _mm512_sub_epi64(t30, ml); b1 = ~a38;
    d2 = _mm512_sub_epi64(t29, mh);
    d3 = _mm512_mask_sub_epi64(d2, b1, d2, one);
    *ch = _mm512_mask_blend_epi64(i28, t29, d3);
    *cl = _mm512_mask_blend_epi64(i28, t30, d1);
}
```

AVX-512 Multi-word Extension (MQX)

Instruction	Emulation	Description
`void _mm512_mul_epi64(__m512i* ch, __m512i* cl, __m512i a, __m512i b)`	`for (i = 0; i < 8; i++) { *ch[i] = ((i128) a[i] * (i128) b[i]) >> 64; *cl[i] = a[i] * b[i]; }`	Multiply two words, and output the high and low parts of the result as two words.
`__m512i _mm512_adc_epi64(__m512i a, __m512i b, __mmask8 ci, __mmask8* co)`	`for (i = 0; i < 8; i++) { *co[i] = ((i128) a[i] + (i128) b[i] + (i128) ci[i]) >> 64; c[i] = a[i] + b[i] + ci[i]; return c; }`	Add two words and a carry bit, and output a word and a carry bit.
`__m512i _mm512_sbb_epi64(__m512i a, __m512i b, __mmask8 bi, __mmask8* bo)`	`for (i = 0; i < 8; i++) { *bo[i] = ((i128) a[i] - (i128) b[i] - (i128) bi[i]) >> 127; c[i] = a[i] - b[i] - bi[i]; return c; }`	Subtract two words and a borrow bit, and output a word and a borrow bit.

AVX-512 Multi-word Extension (MQX)

Instruction	Emulation	Description
`void _mm512_mul_epi64(__m512i* ch, __m512i* cl, __m512i a, __m512i b)`	`for (i = 0; i < 8; i++) { *ch[i] = ((i128) a[i] * (i128) b[i] >> 64; *cl[i] = a[i] * b[i]; }`	Multiply two words, and output the high and low parts of the result as two words.
`__m512i _mm512_adc_epi64(__m512i a, __m512i b, __mmask8 ci, __mmask8* co)`	`for (i = 0; i < 8; i++) { *co[i] = ((i128) a[i] + (i128) b[i] >> 64; c[i] = a[i] + b[i] + ci[i]; return c; }`	Add two words and a carry bit, and output a word and a carry bit.
`__m512i _mm512_sbb_epi64(__m512i a, __m512i b, __mmask8 bi, __mmask8* bo)`	`for (i = 0; i < 8; i++) { *bo[i] = ((i128) a[i] - (i128) b[i] >> 64; c[i] = a[i] - b[i] - bi[i]; return c; }`	Subtract two words and a borrow bit, and output a word and a borrow bit.

SIMD widening multiplication

SIMD addition with carry

SIMD subtraction with borrow

More in Sec. 4.1

CMU

SIMD Addition with Carry

```
__m512i _mm512_adc_epi64(__m512i a, __m512i b, __mmask8 ci, __mmask8* co)
```

Per-lane 64-bit addition with carry-in and outputting both the addition result and carry-out

Similar instructions

ISA	Instruction
x86	ADC
LRBni	`vadcpi`
KNC	`__m512i _mm512_adc_epi32 (__m512i v2, __mmask16 k2, __m512i v3, __mmask16* k2_res)`

How to model MQX instructions' performance?

Limitations of Open-Source Simulation Tools

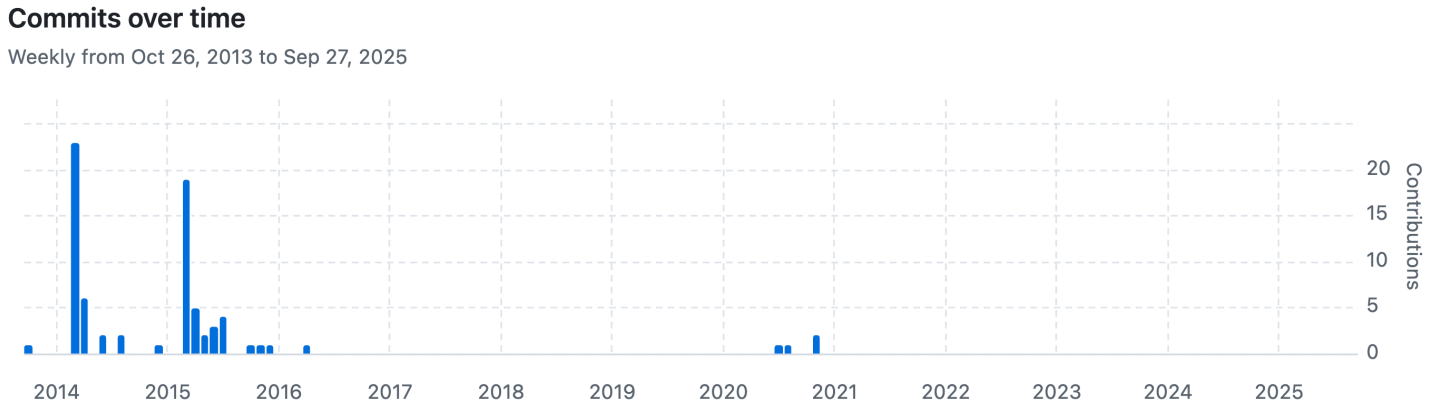
gem5

No official support, limited community support for AVX intrinsics

zSim

Designed for Haswell-era processors and not actively updated

Hard to capture proprietary designs



Commits to the zSim GitHub repository over time
(Source: https://github.com/s5z/zsim)

More in Sec. 4.2 & 5.2

CMU

PISA: Performance Projection with Proxy ISA

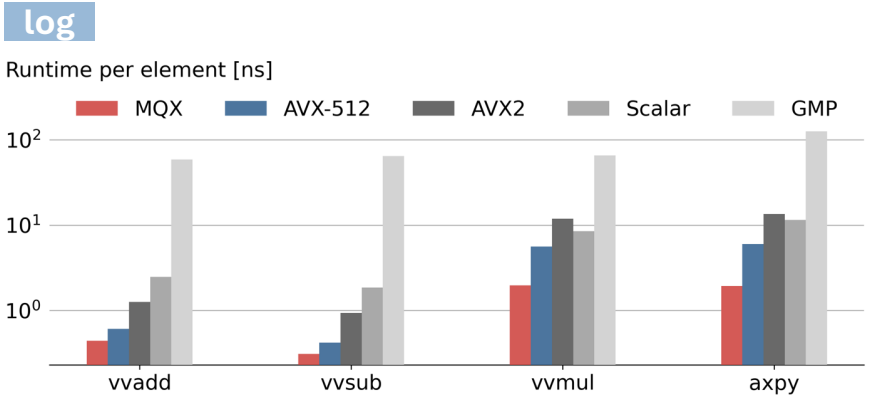
Mapping each MQX instruction to the most structurally similar AVX-512 instruction

MQX instruction	AVX-512 proxy instruction
_mm512_mul_epi64	_mm512_mullo_epi64
_mm512_adc_epi64	_mm512_mask_add_epi64
_mm512_sbb_epi64	_mm512_mask_sub_epi64

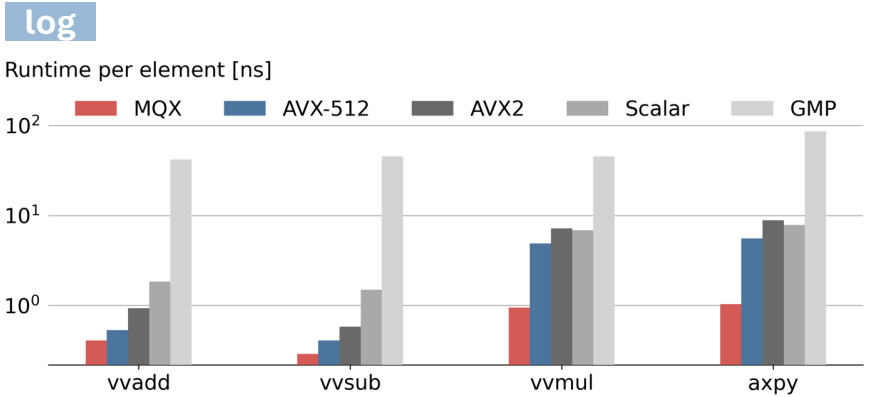
Allowing real measurements on actual hardware
Relative error < 8%

[2023] Intel. “Intel® 64 and IA-32 Architectures Optimization Reference Manual”

BLAS Operations Results



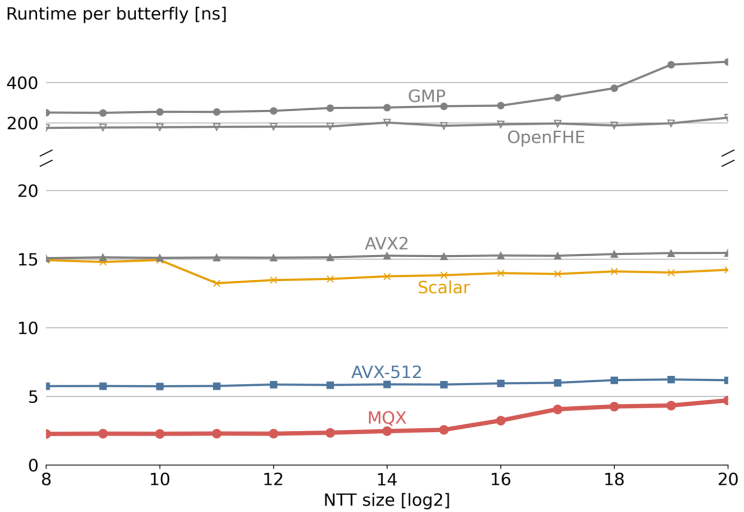
Intel Xeon 8352Y



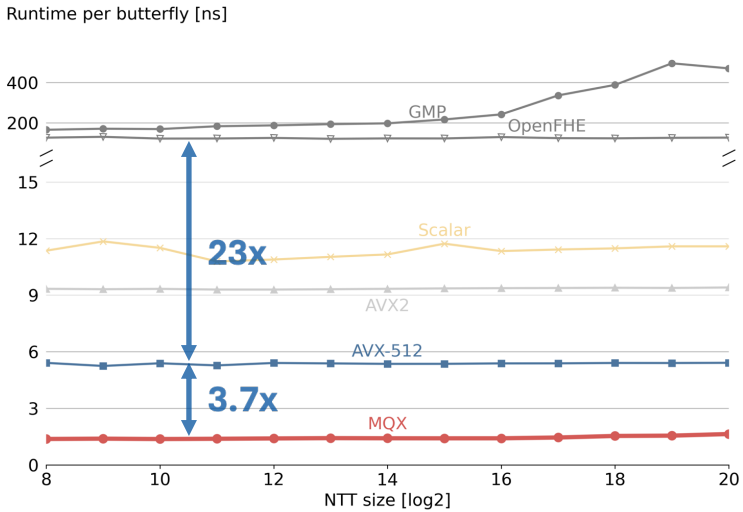
AMD EPYC 9654

Performance of BLAS operations on a single CPU core

NTT Results: Single Core



Intel Xeon 8352Y

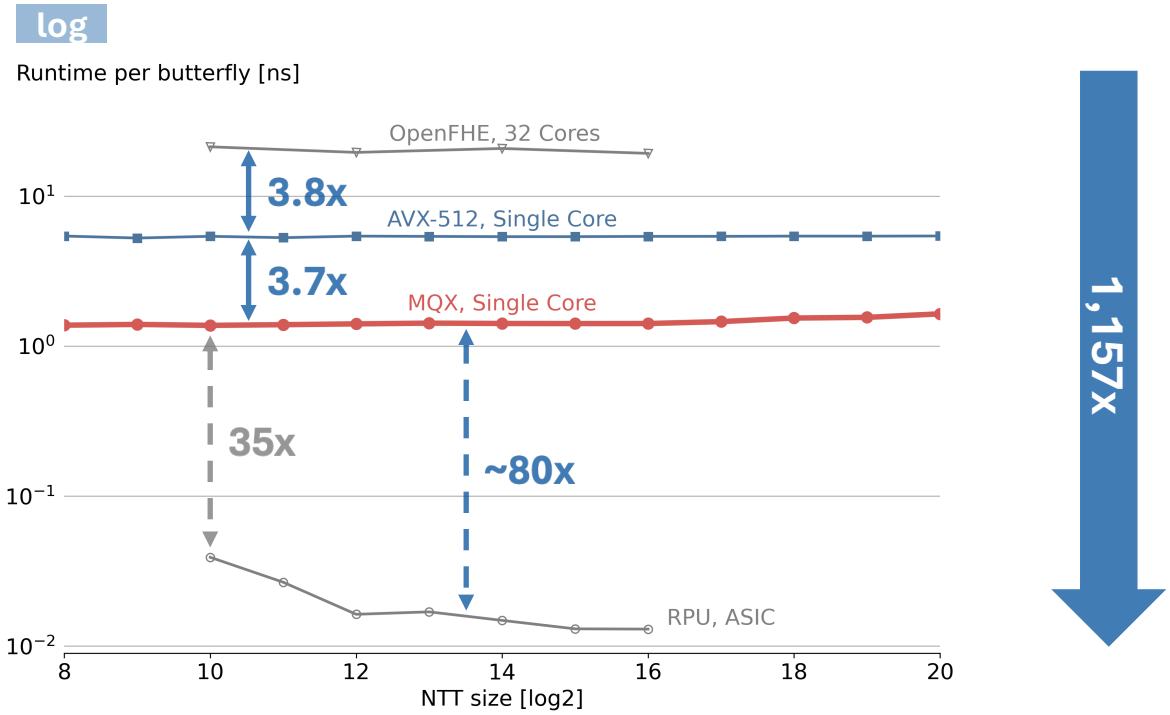


AMD EPYC 9654

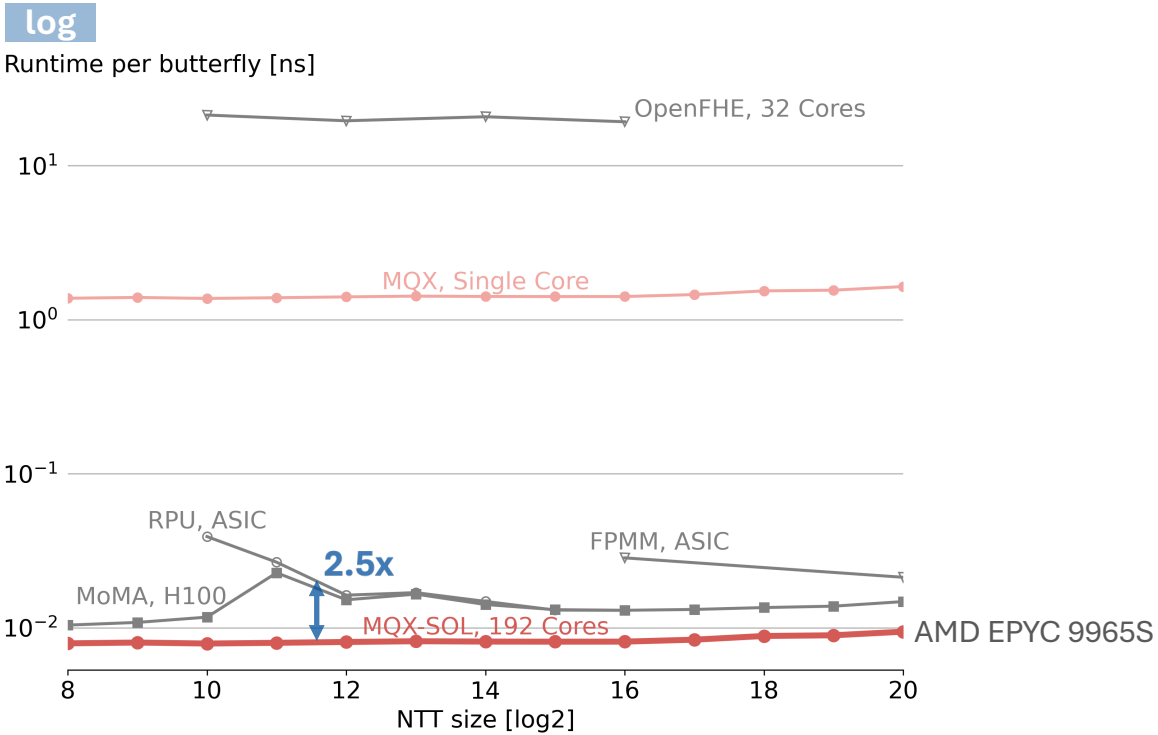
Performance of NTT on a single CPU core

NTT Results: vs. Prior Work

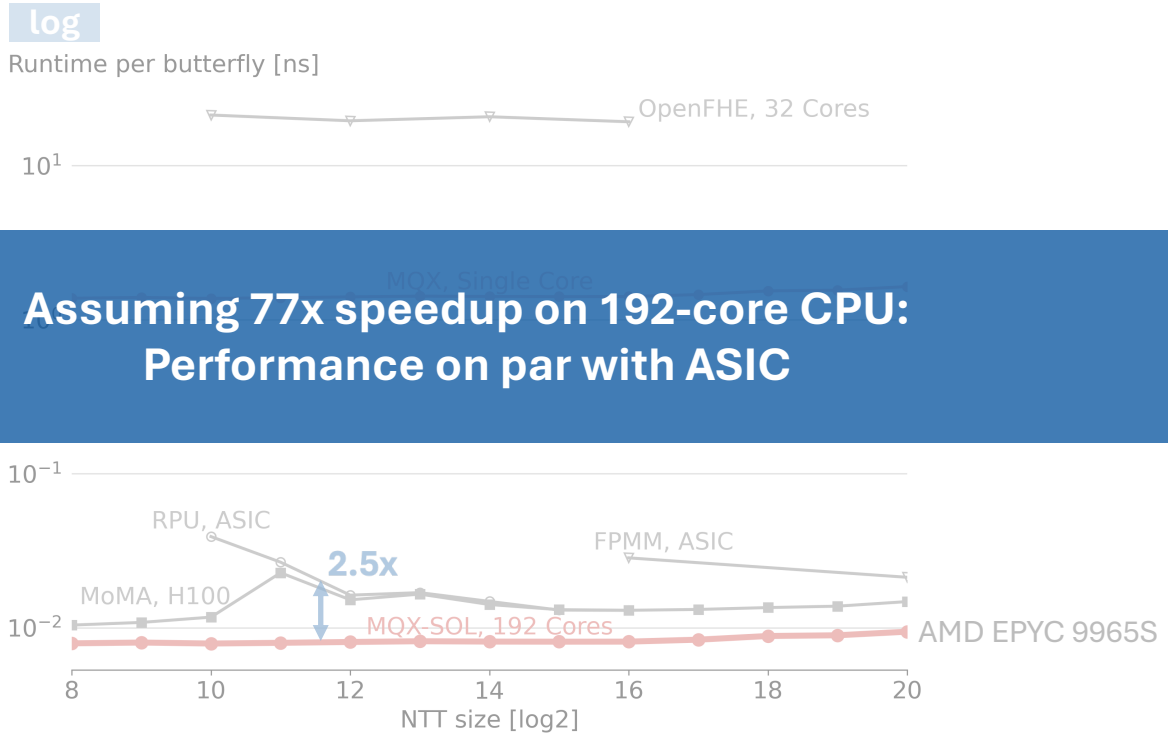
As low as 35x slowdown on a ***single*** CPU core



Speed-of-Light (SOL) Analysis $t_{\text{sol}} = t_{\text{m}} \cdot \frac{c_1}{c_2} \cdot \frac{f_{\text{m}}}{f_{\text{max}}}$



Speed-of-Light (SOL) Analysis $t_{\text{sol}} = t_{\text{m}} \cdot \frac{c_1}{c_2} \cdot \frac{f_{\text{m}}}{f_{\text{max}}}$



Assuming 77x speedup on 192-core CPU:
Performance on par with ASIC

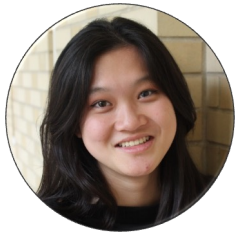
감사합니다!

Code available at github.com/naifeng/benchntt

Reach us at naifengz@cmu.edu

Naifeng Zhang

Seeking **industry/*academia*** opportunities
June 2026 ~ May 2027



Sophia Fu

Seeking ***industry*** opportunities
May 2026

Franz Franchetti