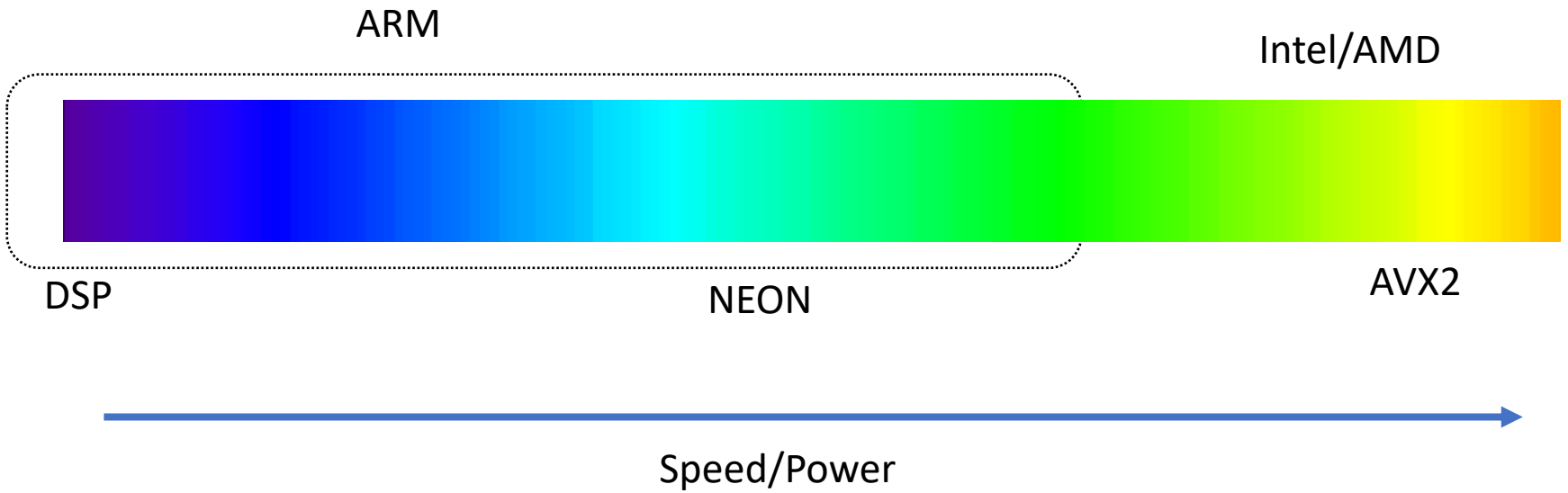
Fast Falcon Signature Generation and Verification Using ARMv8 NEON Instruction

Cryptographic Engineering Research Group @ George Mason University

Duc Tri Nguyen and Kris Gaj

Introductions



- Most optimized software implementations of PQC candidates on:
  - Intel/AMD with Advanced Vector Extensions 2 (AVX2)
  - Cortex-M4 with Digital Signal Processing (DSP) extensions

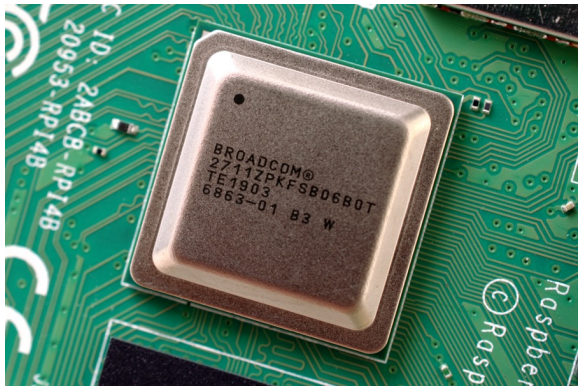
- Lack of NEON implementations on ARMv7 and ARMv8 architectures

Introductions

- NEON is an alternative name for Advanced Single Instruction Multiple Data (ASIMD) extension to the ARM Instruction Set Architecture, mandatory since ARMv7-A.

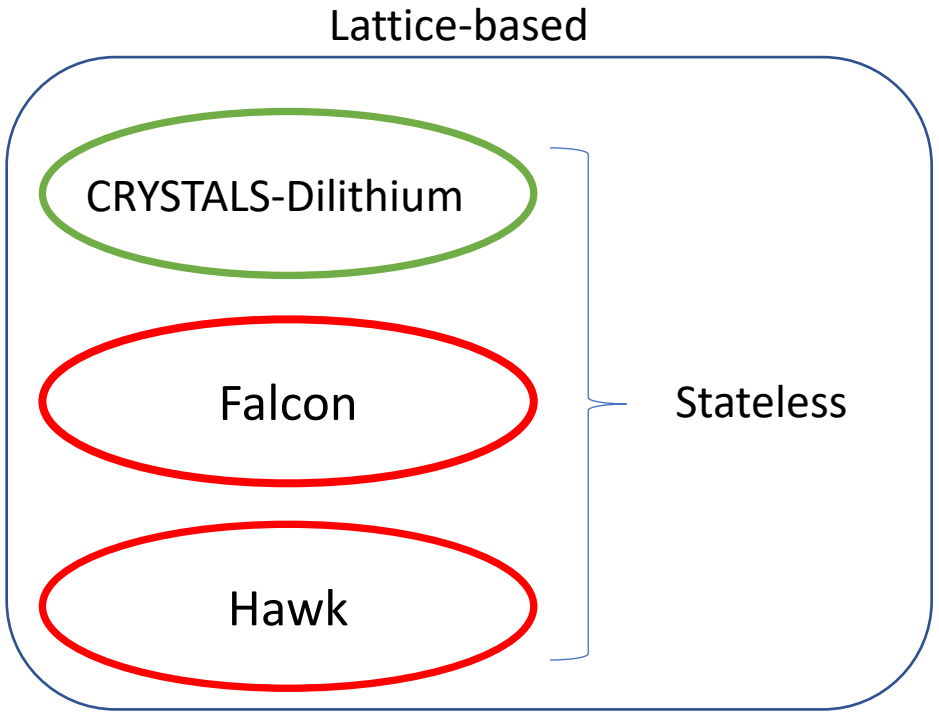
- NEON provides 32 **128-bit** vector registers. Compared with Single Instruction Single Data (SISD), ASIMD can have ideal speed-up in the range 2..16 (for 64..8-bit operands).

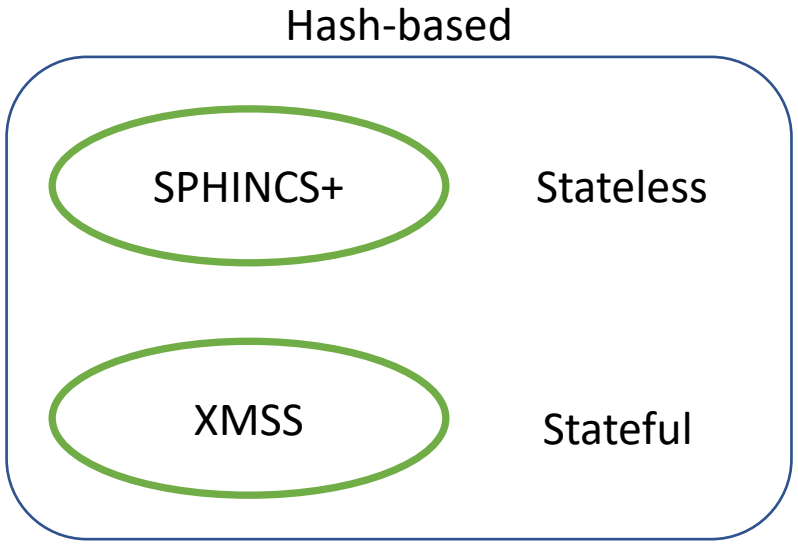
Apple M1:
part of new MacBook Air, MacBook Pro, Mac Mini, iMac, and iPad Pro



Broadcom SoC, BCM2711:
part of the Raspberry Pi 4 single-board computer

PQC Digital Signatures





In this work, we provide fast NEON implementations of two candidates:

- Falcon
- Hawk (sharing code with Falcon)

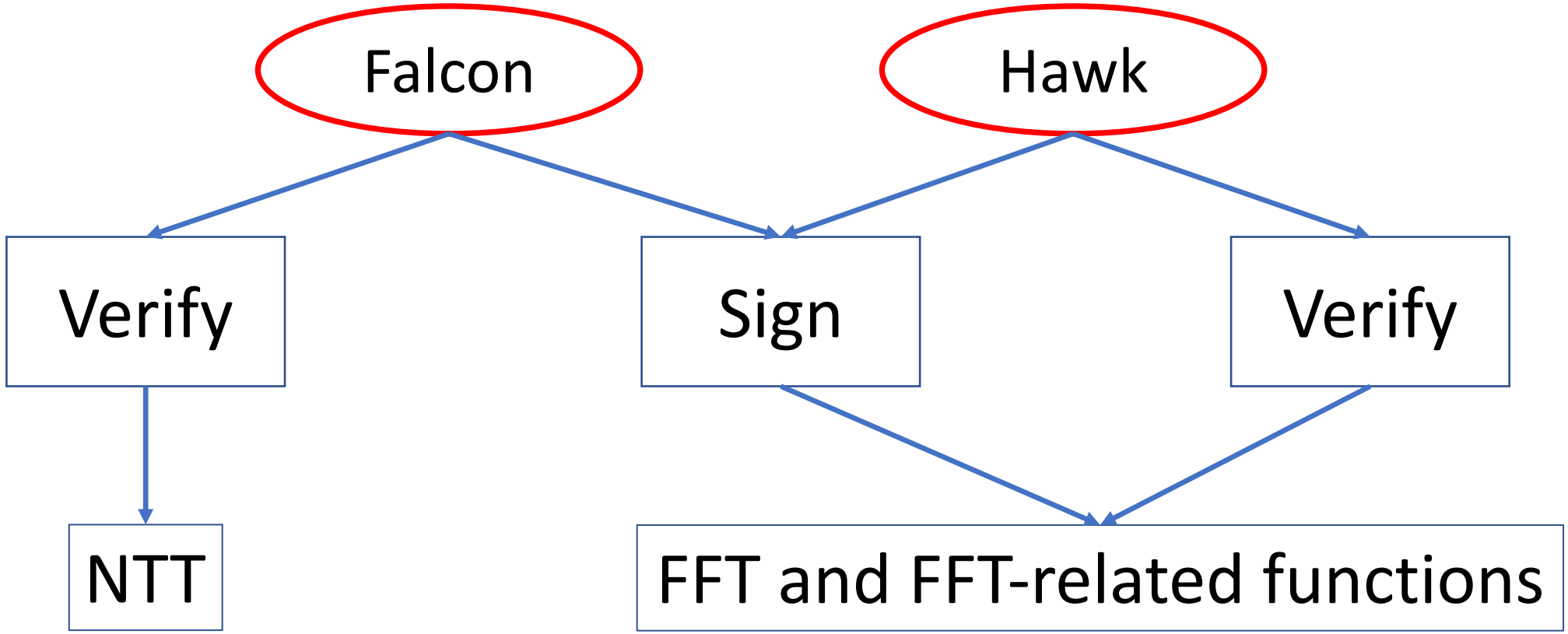
Note: Hawk is a new lattice-based signature published in 2022.

PQC Digital Signatures

Table 1: Parameter sets, key sizes, and signature sizes for FALCON, HAWK, DILITHIUM, XMSS, and SPHINCS$^+$ in comparison with FALCON512, DILITHIUM3, FALCON1024 according to three security levels

	NIST level	n	q	|pk|	|sig|	|pk + sig|	|sig| ratio
FALCON512	I	512	12,289	897	652	1,549	1.00
HAWK512			65,537	1,006	542	1,548	0.83
DILITHIUM2	II	256	8,380,417	1,312	2,420	3,732	3.71
XMSS^{16}-SHA256		-	-	64	2,692	2,756	4.12
SPHINCS$^+$128s	I	-	-	32	7,856	7,888	12.05
SPHINCS$^+$128f		-	-	32	17,088	17,120	26.21
DILITHIUM3	III	256	8,380,417	1,952	3,293	5,245	1.00
SPHINCS$^+$192s		-	-	48	16,224	16,272	4.93
SPHINCS$^+$192f		-	-	48	35,664	35,712	10.83
FALCON1024	V	1024	12,289	1,793	1,261	3,054	1.00
HAWK1024			65,537	2,329	1,195	3,524	0.95
DILITHIUM5		256	8,380,417	2,592	4,595	7,187	3.64
SPHINCS$^+$256s		-	-	64	29,792	29,856	23.62
SPHINCS$^+$256f		-	-	64	49,856	49,920	39.53

Optimized Functions



Number Theoretic Transform (NTT)

Complete NTT

$$C(x) = A(x) \times B(x)$$
$$= \mathcal{NTT}^{-1}(\mathcal{NTT}(A) * \mathcal{NTT}(B))$$

where $A(x), B(x)$ and $C(x) \in \mathbf{Z}_q[x]/(x^n + 1)$ and $q = 1 \bmod 2n$

Forward and Inverse NTT

Algorithm 3: Signed Barrett multiplication by a known constant [BHK+21]

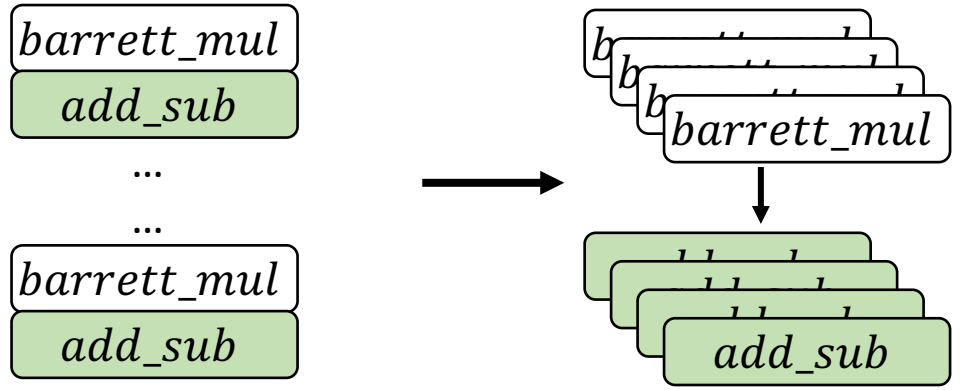
Input: Any $|a| < R = 2^w$, constant $|b| < q$ and $b' = \lfloor (b \cdot R/q)/2 \rceil$

Output: $c = \texttt{barrett\_mul}(a, b, b') = a * b \bmod q$, and $|c| < \frac{3q}{2} < \frac{R}{2}$

1 $t \leftarrow \texttt{sqrdmulh}(a, b')$ ▷ $\texttt{hi}(\texttt{round}((2 \cdot a \cdot b'))$

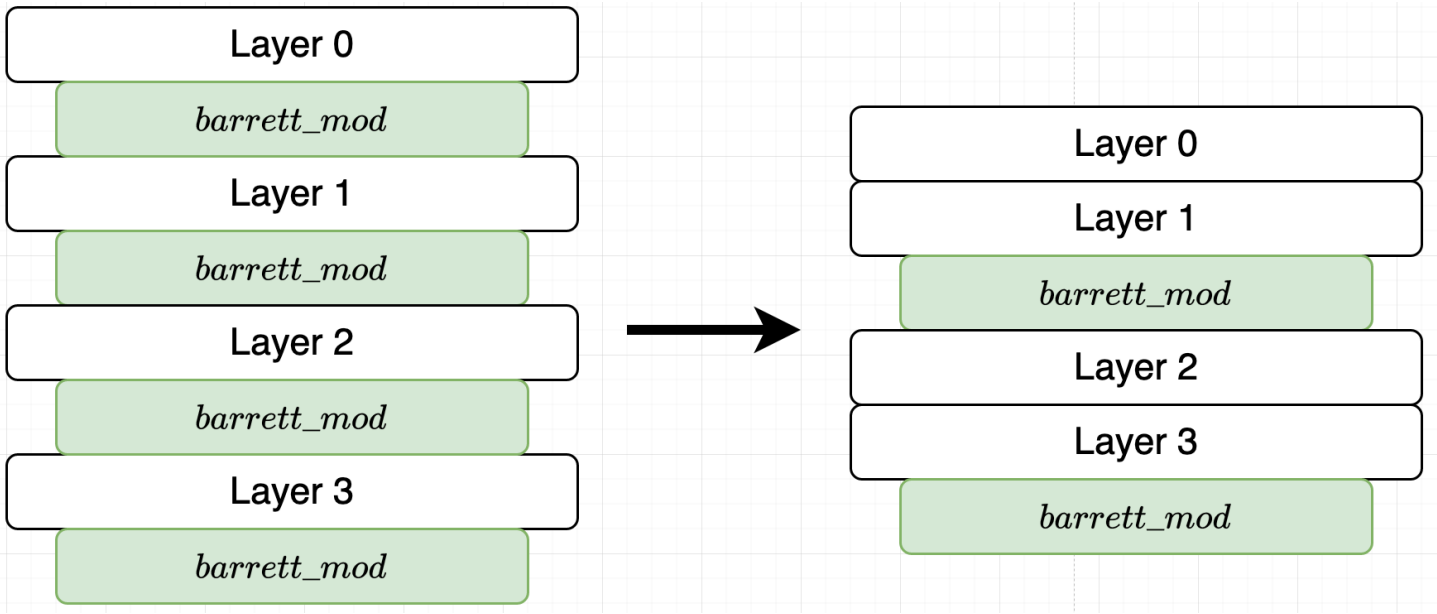
2 $c \leftarrow \texttt{mul}(a, b)$ ▷ $\texttt{lo}(a \cdot b)$

3 $c \leftarrow \texttt{mls}(c, t, q)$ ▷ $\texttt{lo}(c - t \cdot q)$



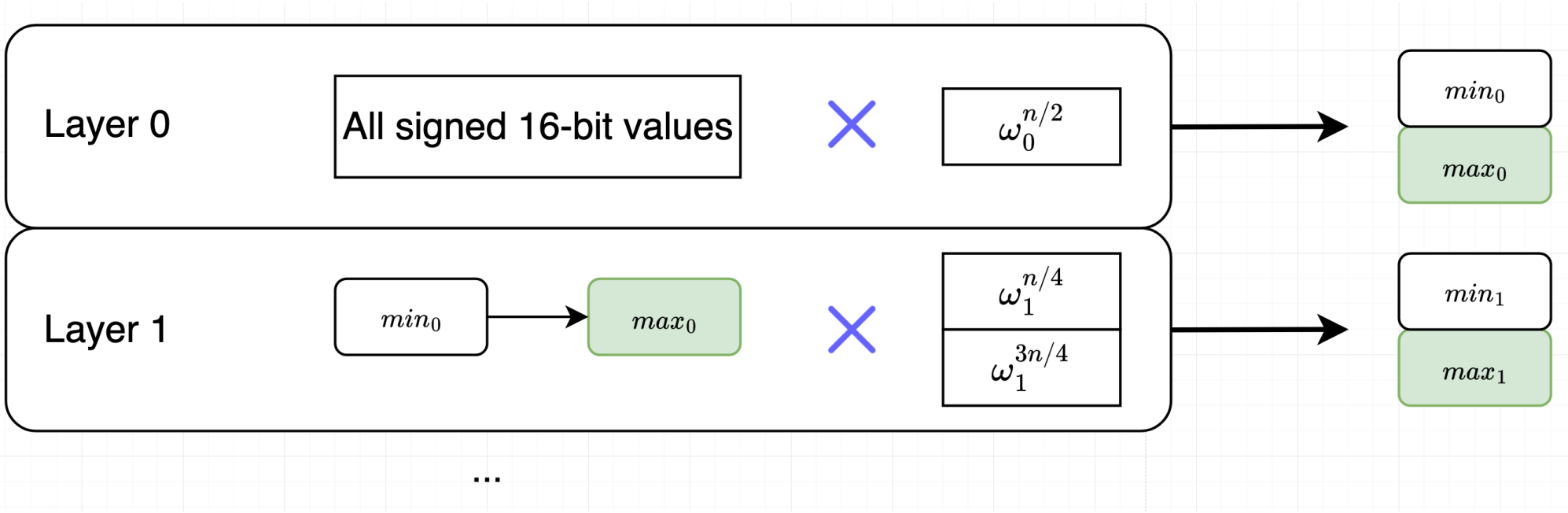
- Extensive usage of Barrett multiplication ($barrett\_mul$) with 3-instructions
- Grouping multiple $barrett\_mul$ together to enhance the pipeline, thus improving NTT performance by 10% on Cortex-A72.
- Theoretical bound $|c| < \frac{3q}{2}$ led to Barrett reduction after each NTT layer

Improved bound of $barrett\_mul$



- The cost of Barrett reduction ($barrett\_mod$) is as high as $barrett\_mul$
- Tighter output bound of $barrett\_mul$ helps reduce the number of Barrett reductions
- Improve performance of NTT significantly

Tighter output bound of $barrett\_mul$



<u>Observation</u>: Output values of $barrett\_mul$ are much *smaller* than the ideal bound $|c| < \frac{3q}{2}$

- Tighter $barrett\_mul$ output bound by exhaustive search for all possible signed 16-bit values. Collect *min, max* value of each NTT level
- Will not work well with bigger search space, e.g., 32-bit, 64-bit space

Pointwise multiplication

$$\mathcal{NTT}(A) * \mathcal{NTT}(B)$$

Algorithm 2: Signed Montgomery multiplication *via doubling* [BHK+21]

Input: $|a| < R,$ and $|bR| < R, \quad R = 2^w \quad$ and $\quad w = [16, 32]$

Output: $c = \texttt{mont\_mul}(a, bR) = a * (bR) * (R^{-1}) = a * b \bmod q$ and $-q \leq c < q$

1 $t \leftarrow \texttt{mul}(b, q^{-1})$ ▷ $\texttt{lo}(b \cdot q^{-1})$

2 $c \leftarrow \texttt{sqdmulh}(a, b)$ ▷ $\texttt{hi}(2 \cdot a \cdot b)$

3 $t \leftarrow \texttt{mul}(a, t)$ ▷ $\texttt{lo}(a \cdot t)$

4 $t \leftarrow \texttt{sqdmulh}(t, q)$ ▷ $\texttt{hi}(2 \cdot t \cdot q)$

5 $c \leftarrow \texttt{shsub}(c, t)$ ▷ $(c - t)/2$

1. one polynomial is converted to the Montgomery domain
2. two polynomials are pointwise multiplied $a_i * b_i$ using Algorithm 2
3. Inverse NTT is applied, including the multiply by the scaling factor n^{-1}

Improved Inverse NTT from Montgomery conversion

$$\mathcal{NTT}(A) * \mathcal{NTT}(B) \longrightarrow mont\_mul(a_i R, b_i)$$

$$\downarrow$$

$$n^{-1} * (\mathcal{NTT}(A) * \mathcal{NTT}(B)) \longrightarrow mont\_mul(a_i R n^{-1}, b_i)$$

If $n = 2^k$, then $R \cdot n^{-1} = 2^w \cdot 2^{-k} = 2^{w-k}$, thus $a_i R n^{-1} = a_i \ll (w - k)$

- At algorithm level: save n multiplications by the scaling factor at *the end* of Inverse NTT
- At instruction level: reduce from 3 multiply instructions down to 2 multiply instructions.
- Can be applied to reference C implementation as well.

Embedding n^{-1} in Montgomery conversion

Algorithm 1a: Montgomery conversion using Barrett multiplication

Input: Any $|a| < R = 2^w$, $(n, i) \leftarrow (512, 7), (1024, 6)$ and $b' = [(b \cdot R/q)/2]$

Output: $c = \texttt{barrett\_mul}(a, b, b') = a * b \bmod q$, c in Montgomery domain

1 $t \leftarrow \texttt{sqrdmulh}(a, b')$	1 $t \leftarrow \texttt{sqrdmulh}(a, b')$
2 $c \leftarrow \texttt{mul}(a, 2^i)$	2 $c \leftarrow shl(a, i)$
3 $c \leftarrow \texttt{mls}(c, t, q)$	3 $c \leftarrow \texttt{mls}(c, t, q)$

- At algorithm level: save n multiplications by the scaling factor at *the end* of Inverse NTT
- At instruction level: reduce from 3 multiply instructions down to 2 multiply instructions.
- Can be applied to reference C implementation as well.

NEON NTT Results

Apple M1

	Forward NTT(*cycles*)			Inverse NTT(*cycles*)		
N	ref	neon	ref/neon	ref	neon	ref/neon
512	6,607	840	7.87	6,449	811^n	7.95
1024	13,783	1,693	8.14	13,335	$1,702^n$	7.83

Cortex-A72

	Forward NTT(*cycles*)			Inverse NTT(*cycles*)		
N	ref	neon	ref/neon	ref	neon	ref/neon
512	22,582	3,561	6.34	22,251	$3,563^n$	6.25
1024	48,097	7,688	6.26	47,196	$7,872^n$	6.00

Compared to reference implementation:

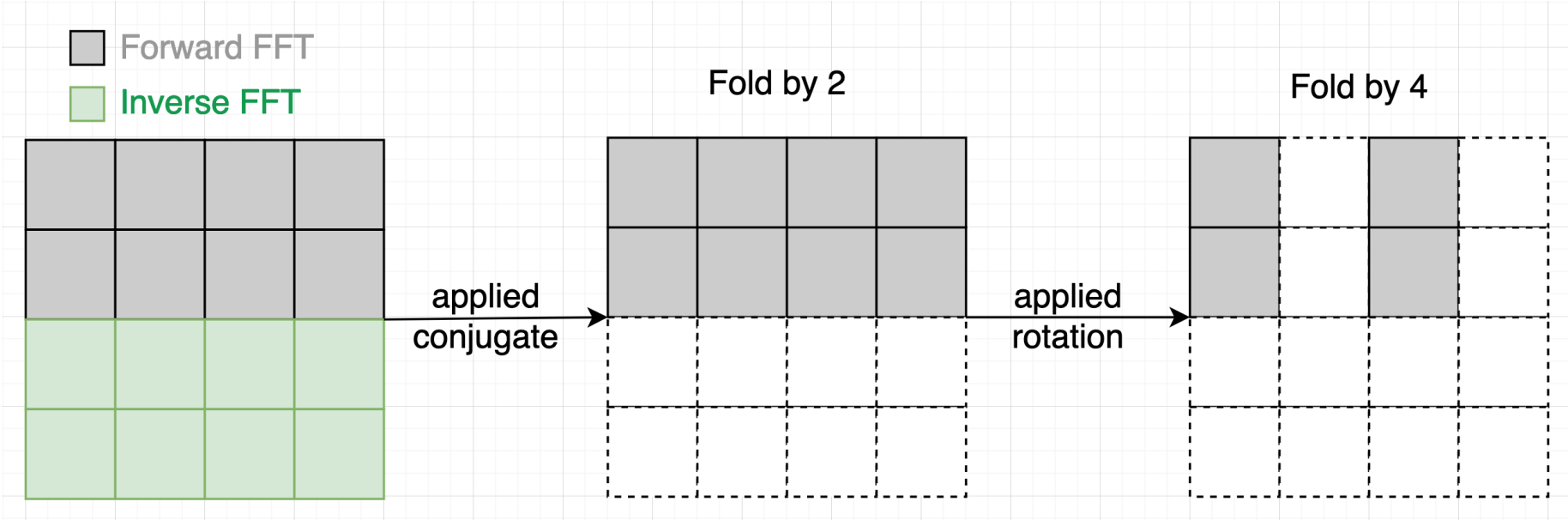
- On Apple M1, we achieve $7.8 \rightarrow 8.0\times$ speed up
- On Cortex-A72, we achieve $6.0 \rightarrow 6.3\times$ speed up

Fast Fourier Transform (FFT)

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi kn/N} \text{ with } k \in [0, N-1]$$

- The Fast Fourier Transform (FFT) is an algorithm that computes a Discrete Fourier Transform (DFT) in $O(nlogn)$, instead of $O(n^2)$
- Fourier Transform converts values from the Time domain to the Frequency domain and vice versa
- Falcon uses FFT mainly for Sampling

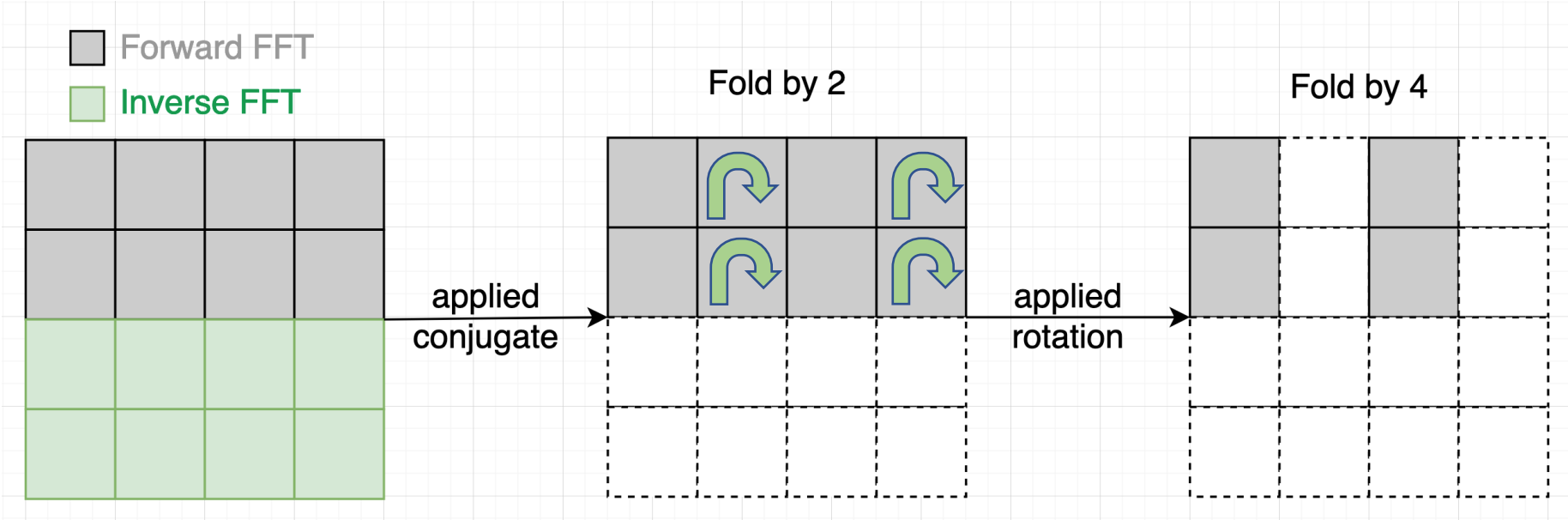
Compressed Twiddle Factor Table



- Observation: Repeated constant pair values in the reference implementation, different in signs and orders.
- We can derive variants of a pair by conjugation and rotation on the fly for free
- Compressed the Twiddle Factor table $4\times$

=> Need to rewrite the algorithm for Forward and Inverse FFT to match the new Table!

Compressed Twiddle Factor Table



- Observation: Repeated constant pair values in the reference implementation, different in signs and orders.
- We can derive variants of a pair by conjugation and rotation on the fly for free
- Compressed the Twiddle Factor table $4\times$

=> Need to rewrite the algorithm for Forward and Inverse FFT to match the new Table!

Choice of FFT implementation

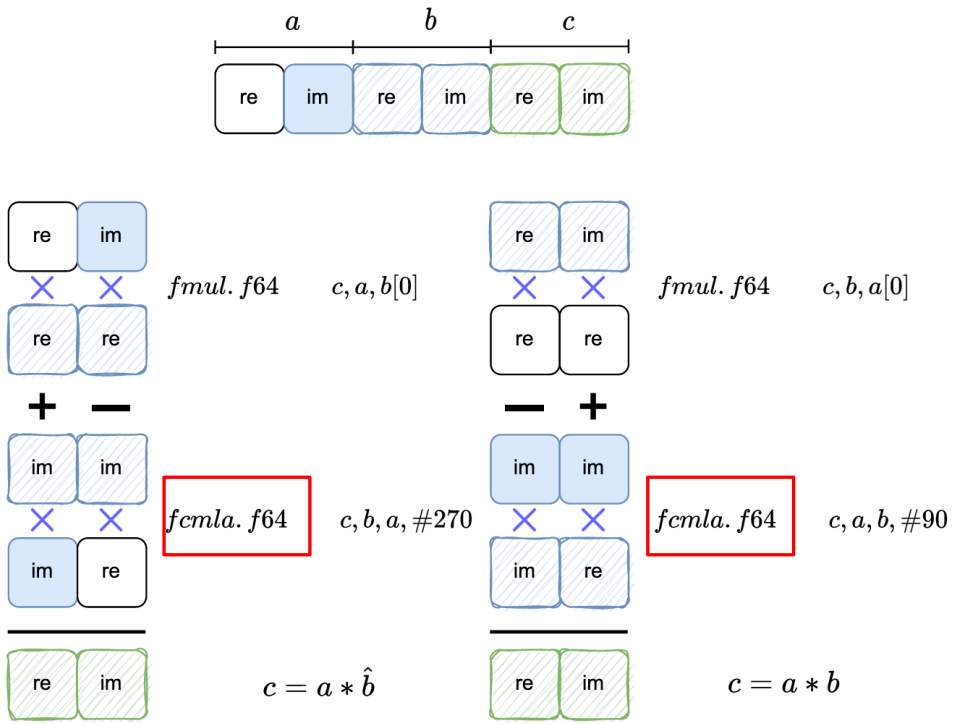


Figure 1: Single pair complex multiplication using `fmul, fcmla`. Real and imagine points are stored adjacently.

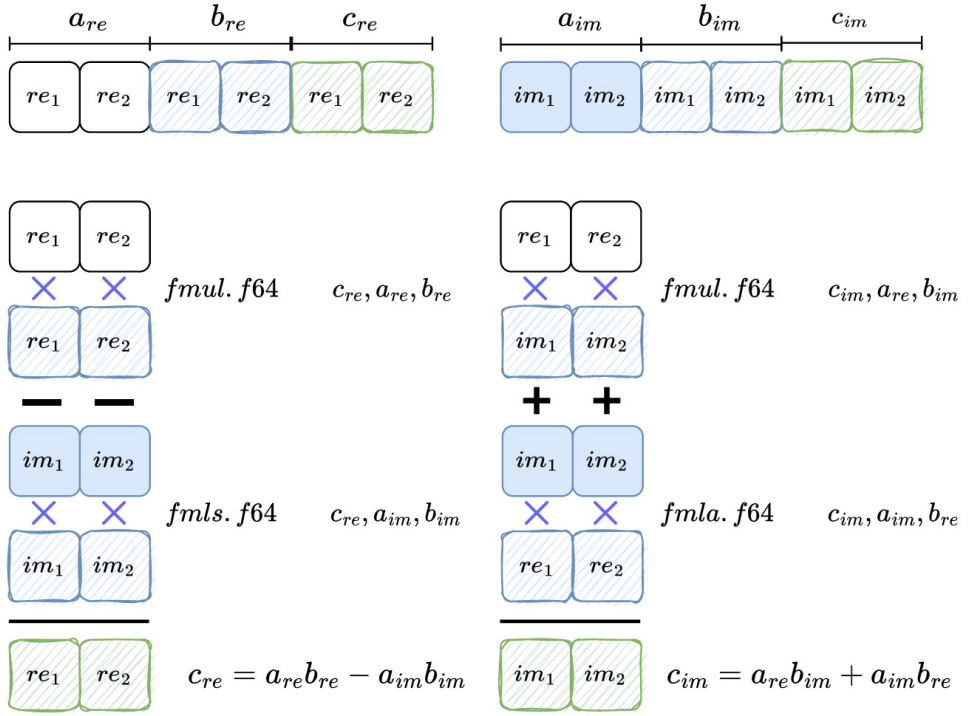


Figure 2: Two pairs complex multiplication using `fmul, fmls, fmla`. Real and imagine points are stored separately.

- Complex point storage: Adjacently (left) and Split (right)
- Complex multiplication with the same cycle count:
  - Figure 1 uses complex instructions (only available since ARMv8.3)
  - Figure 2 uses normal instructions

Choice of FFT implementation

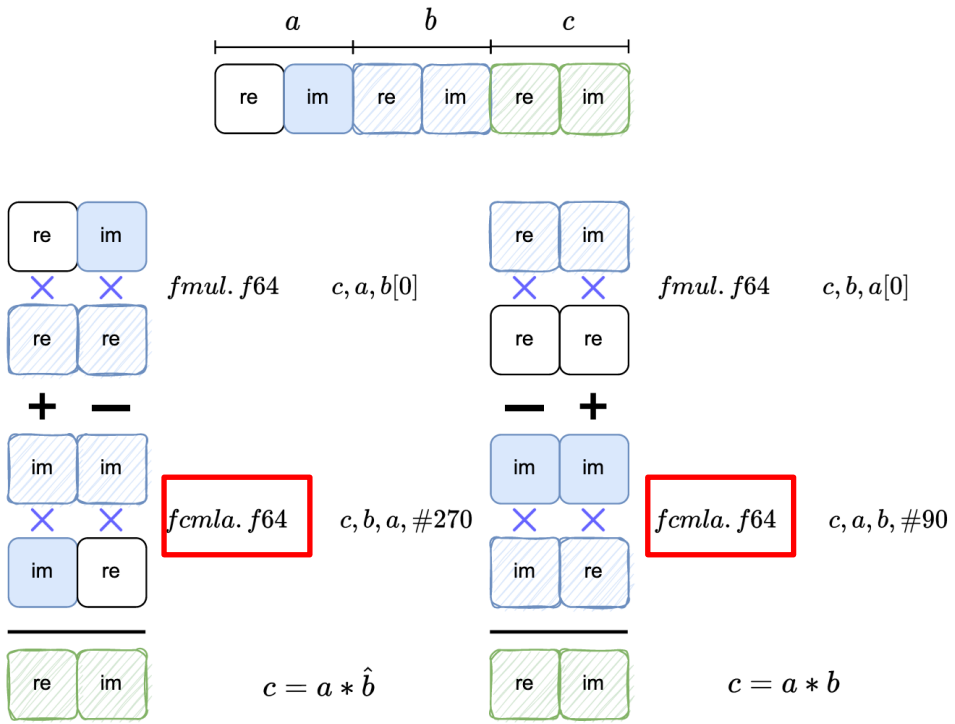


Figure 1: Single pair complex multiplication using `fmul, fcmla`. Real and imagine points are stored adjacently.

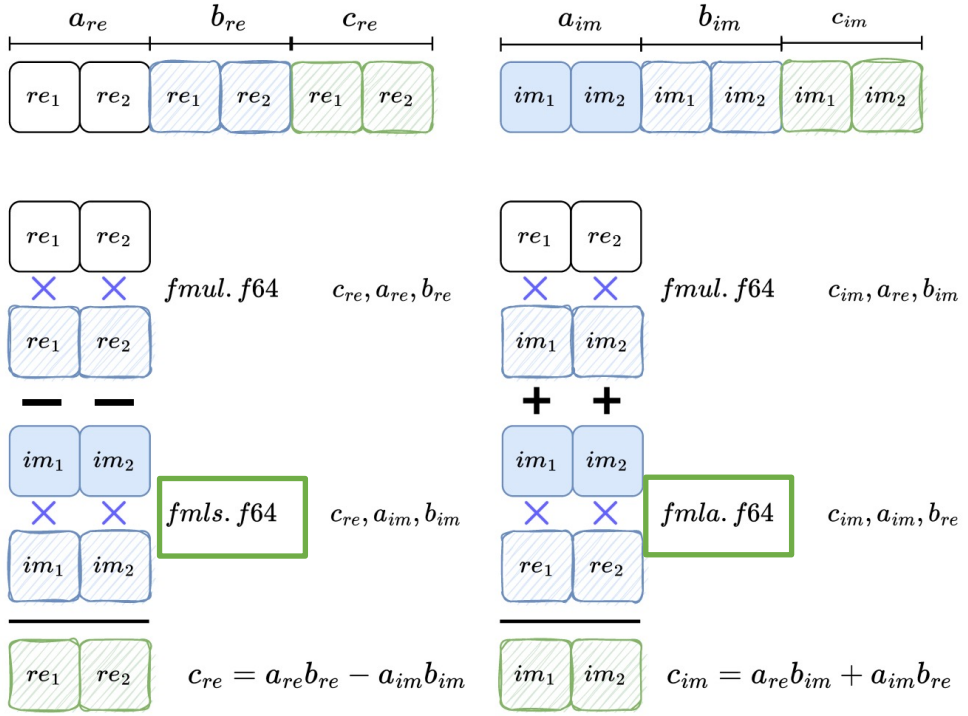
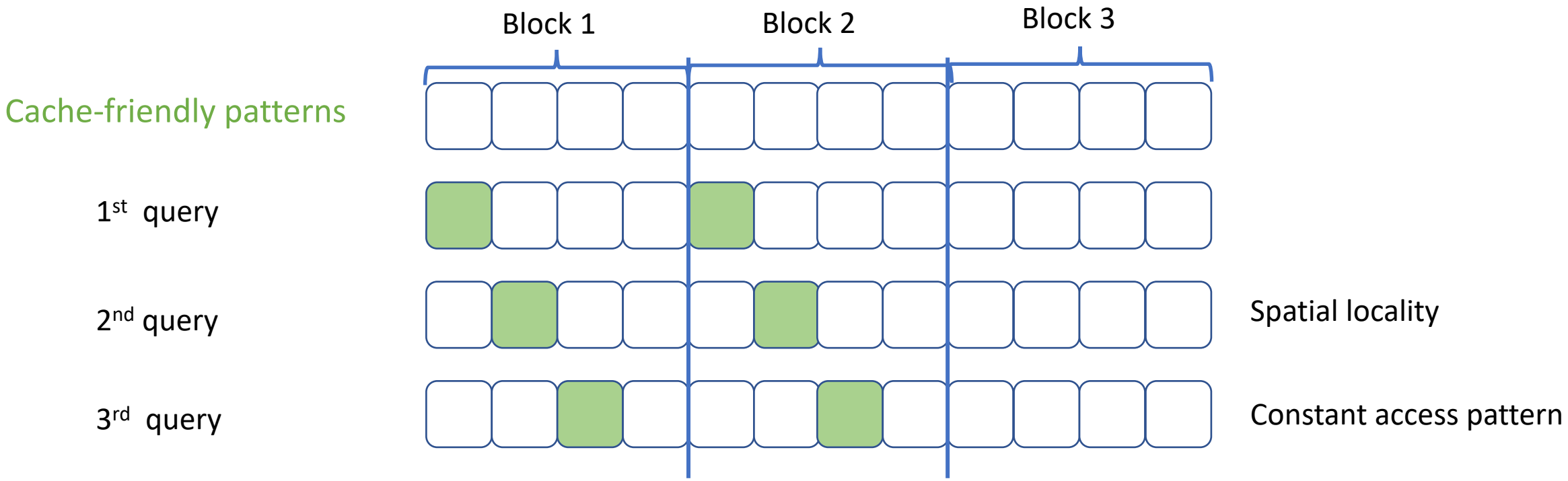


Figure 2: Two pairs complex multiplication using `fmul, fmls, fmla`. Real and imagine points are stored separately.

- Complex point storage: Adjacently (left) and Split (right)
- Complex multiplication with the same cycle count:
  - Figure 1 uses complex instructions (only available since ARMv8.3)
  - Figure 2 uses normal instructions

What is cache-friendly?



- Distanced accesses cause cache misses
- Irregular access patterns hinder data from being prefetched into cache
- Cache-friendly: Constant access pattern, spatial locality

Cache-friendly Forward FFT

Algorithm 8: In-place cache-friendly Forward FFT

Input: Polynomial $f \in \mathbb{Q}[x]/(x^{N/2}+1)$, twiddle factor table `tw`
Output: $f = \texttt{FFT}(f)$

```
1  ω ← tw[0][0]
2  for j = 0 to N/4 − 1 do
3  |  CT_BF(f[j], f[j + N/4], ω)                ▷ exploit ω_re = ω_im
4  |  j ← j + 1
5  level ← 1
6  for len = N/8 to 1 do
7  |  k ← 0                                     ▷ reset k at new level
8  |  for s = 0 to N/2 − 1 do
9  |  |  ω ← tw[level][k]                       ▷ ω is shared between two loops
10 |  |  for j = s to s + len − 1 do            ┐
11 |  |  |  CT_BF(f[j], f[j + len], ω)          │ 1st loop
12 |  |  |  j ← j + 1                           │
13 |  |  s ← s + (len ≪ 1)                      ┘
14 |  |  for j = s to s + len − 1 do            ┐
15 |  |  |  CT_BF_90(f[j], f[j + len], ω)       │ 2nd loop
16 |  |  |  j ← j + 1                           │
17 |  |  s ← s + (len ≪ 1)                      ┘
18 |  |  k ← k + 1                              ▷ increase by one point
19 |  level ← level + 1                         ▷ increase level
20 |  len ← len ≫ 1                             ▷ half distance
```

Constant access pattern

Spatial locality

- The 1st and 2nd loop can be executed in parallel
- May be interesting in terms of hardware development

Cache-friendly Inverse FFT

Algorithm 12: In-place cache-friendly Inverse FFT

Input: Polynomial $f \in \mathbb{Q}[x]/(x^{N/2}+1)$, twiddle factor table `tw`
Output: $f = \texttt{invFFT}(f)$

1 $level \leftarrow \log_2(N) - 2$ ▷ `tw` index starts at 0, and N/2 *re, im* points
2 **for** $len = 1$ **to** $N/8$ **do**
3 $k \leftarrow 0$ ▷ reset k at new *level*
4 **for** $s = 0$ **to** $N/2 - 1$ **do**
5 $\omega \leftarrow \texttt{tw}[level][k]$ ▷ ω is shared between two loops
6 **for** $j = s$ **to** $s + len - 1$ **do**
7 GS_BF$(f[j], f[j+len], \omega)$
8 $j \leftarrow j + 1$
9 $s \leftarrow s + (len \ll 1)$
10 **for** $j = s$ **to** $s + len - 1$ **do**
11 GS_BF_270$(f[j], f[j+len], \omega)$
12 $j \leftarrow j + 1$
13 $s \leftarrow s + (len \ll 1)$
14 $k \leftarrow k + 1$ ▷ increase by *one* point
15 $level \leftarrow level - 1$ ▷ decrease *level*
16 $len \leftarrow len \ll 1$ ▷ double distance
17 $\omega \leftarrow \texttt{tw}[0][0] \cdot \frac{2}{N}$
18 **for** $j = 0$ **to** $N/4 - 1$ **do**
19 GS_BF$(f[j], f[j+N/4], \omega)$ ▷ exploit $\omega_{re} = \omega_{im}$
20 $f[j] \leftarrow f[j] \cdot \frac{2}{N}$
21 $j \leftarrow j + 1$

Constant access pattern

Spatial locality

- The 1st and 2nd loop can be executed in parallel
- May be interesting in terms of hardware development

Minimize load/store for arbitrary FFT levels

Table 2: Summary of butterfly loops for NTT and FFT in Falcon $N = 512, 1024$

	$\log_2(N)$	Forward (CT butterfly)	Inverse (GS butterfly)
NTT	9	$2 + 7$	$7 + 2$
	10	$3 + 7$	$7 + 3$
FFT	5	5	5
	6	$1 + 5$	$5 + 1$
	7	$2 + 5$	$5 + 2$
	8	$1 + 2 + 5$	$5 + 2 + 1$
	9	$(2 \times 2) + 5$	$5 + (2 \times 2)$
	10	$1 + (2 \times 2) + 5$	$5 + (2 \times 2) + 1$

- Strategy: Compute 5, 2 and 1 FFT levels per one load and store.
- When $\#FFT\ levels \leq 5$, we use the unrolled FFT implementation
- When $\#FFT\ levels > 5$, we use this formula: $\#\text{FFT levels} = 5 + 2 \cdot x + 1 \cdot y$ to find maximum x and minimum $y \in \{0, 1\}$

NEON FFT Results

Apple M1	Forward FFT(*cycles*)			Inverse FFT(*cycles*)		
N	ref	neon	ref/neon	ref	neon	ref/neon
512	3,640	1,577	2.31	3,930	1,609	2.44
1024	7,998	3,489	2.29	8,541	3,547	2.41
Cortex-A72						
N						
512	11,807	5,951	1.98	13,136	6,135	2.14
1024	27,366	14,060	1.95	28,151	14,705	1.91
Intel i7-1165G7						
N	REF	AVX2	REF/AVX2	REF	AVX2	REF/AVX2
512	3,486	2,040	1.71	3,790	2,138	1.77
1024	7,341	4,370	1.68	7,961	4,572	1.74

- Vectorized cache-friendly FFT proved to be better than the original implementation: 2.3× and 1.9× faster on Apple M1 and Cortex-A72
- Our NEON FFT implementation has a better speed up ratio than AVX2

Fused Multiply-Add (FMA) instructions

FMA Enable

- FMLS
- FMLA

FMA Disable

- FMUL + FSUB
- FMUL + FADD

- Fused instructions improve the performance, use less numbers of instruction
- When FMA Enable: Multiply and Add with single rounding step
- When FMA Disable: Rounding after Multiply, and after Add

Floating-point Rounding concerns

Table 9: Floating point FMA rounding differences in our experiment of `fpr_expm_p63`. Two input, output pairs show that the bits are flipped at $(9, 11, 12)$ position.

Input	*float* value	*hex* value
$(ccs,\ x)_1$	0.9879566266, 0.7112028419	`0x3fef9d57371f9d57, 0x3fe6c22c7656c22c`
$(ccs,\ x)_2$	0.7176312343, 0.5160871303	`0x3fe6f6d5c736f6d6, 0x3fe083c9285083c9`

Output	*float* value	*hex* value
a_1	`1.457683362948139e-09`	`0x3e190af590033c00`
b_1	`1.457683362948986e-09`	`0x3e190af590034c00`
a_2	`1.3426859485919116e-44`	`0x36d329d822c80400`
b_2	`1.3426859485925488e-44`	`0x36d329d822c80e00`

Differences		*hex* value
$\lvert a_1 - b_1 \rvert$	`8.470329472543003e-22`	`0x1000`
$\lvert a_2 - b_2 \rvert$	`6.372367644529809e-57`	`0x0a00`

- When Fused Multiply-Add (FMA) is enabled, our experiment on `fpr_expm_p63` shows 7% values have bit flipped compared to FMA disabled
- The security impact of `FMLA` versus `FMUL + FADD` rounding is unknown
- For now, we recommend disabling FMA

The cost of Fused Multiply-Add

Table 3: Performance of Signature generation with Fused Multiply-Add instructions *enabled* (`fmla`), and *disabled* (`fmul, fadd`).

CPU	neon Sign	(fmul, fadd)	fmla	fmla/(fmul, fadd)
Cortex-A72	falcon512	1,038.14	1,000.31	0.964
	falcon1024	2,132.08	2,046.53	0.960
Apple M1	falcon512	459.19	445.91	0.971
	falcon1024	914.91	885.63	0.968

CPU	neon Function	(fmul, fadd)	fmla	fmla/(fmul, fadd)
Cortex-A72	poly_LDL_fft512	7,181	6,415	0.89
	poly_LDL_fft1024	14,554	12,791	0.87
Apple M1	poly_LDL_fft512	923	739	0.80
	poly_LDL_fft1024	1,699	1,305	0.76

When FMA is disabled:

- On a single function, about $11\% \rightarrow 24\%$ impact on performance
- At top level Falcon **S**ign, about $3\% \rightarrow 4\%$ slower

Tuned up Hash-based Signatures

Authors	Ways	Settings	Cycles (per hash)	Ratio
[Wes21]	2×	neon + SHA3	485 (243)	1.00
[Ngu20]	2×	neon + SHA3	548 (274)	1.13
[BK22]	3×	neon + SHA3 + scalar	1,052 (350)	1.44
[BK22]	4×	neon + SHA3 + scalar	2,094 (524)	2.15
[BK22]	5×	neon + SHA3 + scalar	3,184 (637)	2.64

	C	neon (per hash)	ref/neon
THASH-SHAKE	882	498 (249)	3.54
THASH-SHA256	1,534	268 (268)	5.72

We search for the best settings to accelerate Keccak-f1600 and SHA256 on Apple M1:

- Keccak 2-ways with SHA3 Crypto Instruction by Westerbaan is the fastest setting for SPHINCS+, and provides 3.54× speed-up as compared with C Implementation
- SHA256 with SHA2 Crypto Instruction (from OpenSSL) is faster by 5.72× , this substantially improved the performance of XMSS

Benchmarking Methodology

Apple M1 System on Chip	Firestorm core, 3.2 GHz[1], MacBook Air
Broadcom BCM2711 System on Chip	**Cortex-A72** core, 1.8 GHz, Raspberry Pi 4
Operating System	MacOS 13.0, Raspberry Pi OS (Sep/2022)
Compiler	clang 14.0 (MacBook Air and Raspberry Pi 4)
Compiler Options	-O3 -mtune=native -fomit-frame-pointer
Cycles count on Cortex-A72	From PQAX[2]
Cycles count on Apple M1	Modified[3] from Dougall Johnson's work[4]
Iterations	1,000,000 on Apple M1 to force CPU to run on high-performance "FireStorm" core; 100,000 otherwise

[1] https://www.anandtech.com/show/16252/mac-mini-apple-m1-tested
[2] https://github.com/mupq/pqax#enable-access-to-performance-counters
[3] https://github.com/GMUCERG/PQC_NEON/blob/main/neon/kyber/m1cycles.c
[4] https://github.com/dougallj

Previous Work by Kim et al.

Jetson AGX Xavier:

Memory	64GB 256-bit LPDDR4x 136.5GB/s
CPU	8-core NVIDIA Carmel Armv8.2 64-bit CPU 8MB L2 + 4MB L3

- No published source code
- No access to device

NVIDIA Jetson AGX Xavier Developer Kit (32GB)

Visit the NVIDIA Store

81 ratings | 13 answered questions

8 Highlights found by Fakespot

$2,426.00

Pay $202.17/month for 12 months (plus S&H, tax) with 0% interest equal monthly payments when you're approved for the Prime Store Card.

In Stock

Brand	NVIDIA
Operating System	Linux
Memory Storage Capacity	16 GB
Screen Size	1 Inches
Ram Memory Installed Size	32 GB

See more

About this item

- Newly updated version with an additional 16GB of memory for a total of 32GB of 256-bit wide LPDDR4X memory.
- NVIDIA Jetson Xavier is an AI computer for Autonomous Machines with the performance of a GPU workstation in under 30W
- The Jetson Xavier Developer Kit with Jetson Xavier module and reference carrier board is the fastest way to start prototyping with robots, drones and other autonomous machines

Youngbeom Kim, Jingyo Song, and Seog Chung Seo.
“Accelerating Falcon on ARMv8”.
IEEE Access, 10:44446–44460, 2022.

Comparison with Previous Work

Table 6: Speed-up ratio comparison (with FMA *enabled*) with previous work from Kim et al. [KSS22] on Raspberry Pi 4, Apple M1, and Jetson AGX Xavier - *kc*-kilocycles

	Jetson AGX Xavier					
	ref(*kc*)		neon(*kc*)		ref/neon	
	S	**V**	**S**	**V**	**S**	**V**
falcon512 [KSS22]	580.7	48.0	498.6	29.0	1.16	1.65
falcon1024 [KSS22]	1,159.6	106.0	990.5	62.5	1.17	1.69
	Apple M1					
falcon512 (Ours)	654.0	43.5	442.0	22.7	1.48	1.92
falcon1024 (Ours)	1,310.8	89.3	882.1	42.9	1.49	2.08
	Raspberry Pi 4					
falcon512 (Ours)	1,490.7	126.3	1,001.9	58.8	1.49	2.15
falcon1024 (Ours)	3,084.8	274.3	2,048.9	130.4	1.51	2.10

Our speed-up ratio, as compared with the reference implementation, is better than in the previous work by Kim et al. under same FMA setting:

- Falcon Sign: $1.48\times \rightarrow 1.51\times$ vs. $1.16\times \rightarrow 1.17\times$
- Falcon Verify: $1.92\times \rightarrow 2.15\times$ vs. $1.65\times \rightarrow 1.69\times$

Ranking: PQ Signatures on Apple M1

Apple-M1 3.2 GHz	NIST level	ref(kc) S	ref(kc) V	neon(kc) S	neon(kc) V	ref/neon S	ref/neon V
FALCON512	I, II	654.0	43.5	459.2	22.7	1.42	1.92
HAWK512		138.6	34.6	117.7	27.1	1.18	1.27
DILITHIUM2^b		741.1	199.6	224.1	69.8	3.31	2.86
XMSS^{16}-SHA256^x		26,044.3	2,879.4	4,804.2	531.0	5.42	5.42
SPHINCS$^+$128s^s		1,950,265.0	1,982.4	549,130.7	658.6	3.55	3.01
SPHINCS$^+$128f^s		93,853.9	5,483.8	26,505.3	1,731.2	3.54	3.16
DILITHIUM3^b	III	1,218.0	329.2	365.2	104.8	3.33	3.14
SPHINCS$^+$192s^s		3,367,415.5	2,753.1	950,869.9	893.2	3.54	3.08
SPHINCS$^+$192f^s		151,245.2	8,191.5	42,815.1	2,515.8	3.53	3.25
FALCON1024	V	1,310.8	89.3	915.0	42.9	1.43	2.08
HAWK1024		279.7	73.7	236.9	58.5	1.18	1.26
DILITHIUM5^b		1,531.1	557.7	426.6	167.5	3.59	3.33
SPHINCS$^+$256s^s		2,938,702.4	3,929.3	840,259.4	1,317.5	3.50	2.98
SPHINCS$^+$256f^s		311,034.3	8,242.5	88,498.9	2,593.8	3.51	3.17

Ranking: PQ Signatures on Cortex-A72

Table 7: **S**ignature generation and **V**erification speed comparison (with FMA *disabled*) over three security levels, signing a 59-byte message. Ranking over the **V**erification speed ratio. `ref` and `neon` results for Cortex-A72. *kc*-kilocycles.

Cortex-A72	NIST	`ref`(kc)		`neon`(kc)		`ref/neon`		
1.8 GHz	Level	**S**	**V**	**S**	**V**	**S**	**V**	**V** ratio
FALCON512		1,553.4	127.8	1,044.6	<u>59.9</u>	1.49	2.09	<u>1.0</u>
HAWK512	I, II	400.3	127.1	315.9	94.8	1.26	1.34	1.6
DILITHIUM2[b]		1,353.8	449.6	649.2	272.8	2.09	1.65	4.5
DILITHIUM3[b]	III	2,308.6	728.9	1,089.4	447.5	2.12	1.63	-
FALCON1024		3,193.0	272.1	2,137.0	<u>125.2</u>	1.49	2.17	<u>1.0</u>
HAWK1024	V	822.1	300.0	655.2	236.9	1.25	1.27	1.9
DILITHIUM5[b]		2,903.6	1,198.7	1,437.0	764.9	2.02	1.57	6.1

[b] the work from Becker et al. [BHK+21]

Conclusions

- We improved NTT and made FFT implementation cache-friendlier and more memory-efficient
- We developed the new state-of-the-art Falcon NEON implementation

- Falcon has smaller bandwidth and faster **V**erification than CRYSTALS-Dilithium
- Lattice-based Signatures are better than Hash-based Signatures in terms of key size and efficiency

- We present the rounding-error effect of Fused Multiply-Add (FMA) instructions. The security impacts of FMA on Falcon and Hawk requires more research

- Hawk outperforms CRYSTALS-Dilithium and has faster **S**ignature generation than Falcon. The performance and bandwidth of Hawk may be attractive to the community.

Thanks for your attention!

Q&A