PQC-AMX: Accelerating Saber and FrodoKEM on the Apple M1 and M3 SoCs

Décio Luiz Gazzoni Filho (decio.gazzoni@ic.unicamp.br)
Guilherme Brandão
Gora Adj
Arwa Alblooshi
Isaac A. Canales-Martínez
Jorge Chávez-Saab
Julio López

Agenda

- Introduction
- The AMX matrix multiplication accelerator
- Arithmetic for Saber and FrodoKEM
- Saber on AMX
  - Baseline polynomial multiplication and reduction
  - Polynomial multiplication from linear algebra
  - Higher-level improvements
- FrodoKEM on AMX
- Gaussian sampling
- Conclusions

Introduction

- Modern cryptography is under threat by quantum computers
- **Post-quantum cryptography**: cryptosystems resistant to both classical and quantum adversaries
- Key/ciphertext size and performance a concern
- Efficient and secure implementations are a must
- Turing award lecture by Hennessy and Patterson: "Innovations like **domain-specific hardware** (...) will lead the way."
- AI/deep learning requires TOPS-level processing power for linear algebra (matrix multiplication)
- Many accelerators and instruction set extensions available: TPUs, GPUs, Intel AMX, ARM SME, **Apple AMX**, etc.

AMX

- Apple SoCs: multiple computational resources such as the CPU, Neural Engine, GPU and **AMX**, the latter of which is an undocumented coprocessor mainly for matrix multiplication
- Reverse engineering by D. Johnson, P. Cawley, M. Handley
- First application to crypto: NTRU (Gazzoni Filho et al, 2024)
- Programmer's model: X,Y vector registers (8×64 bytes each) and Z matrix registers (64×64 bytes)

	$\mathtt{X}[0]$	$\mathtt{X}[1]$	$\cdots$	$\mathtt{X}[n]$
$\mathtt{Y}[0]$	$\mathtt{Z}[0][0] \mathrel{+}= \mathtt{Y}[0]\mathtt{X}[0]$	$\mathtt{Z}[0][1] \mathrel{+}= \mathtt{Y}[0]\mathtt{X}[1]$	$\cdots$	$\mathtt{Z}[0][n] \mathrel{+}= \mathtt{Y}[0]\mathtt{X}[n]$
$\mathtt{Y}[1]$	$\mathtt{Z}[1][0] \mathrel{+}= \mathtt{Y}[1]\mathtt{X}[0]$	$\mathtt{Z}[1][1] \mathrel{+}= \mathtt{Y}[1]\mathtt{X}[1]$	$\cdots$	$\mathtt{Z}[1][n] \mathrel{+}= \mathtt{Y}[1]\mathtt{X}[n]$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$\mathtt{Y}[n]$	$\mathtt{Z}[n][0] \mathrel{+}= \mathtt{Y}[n]\mathtt{X}[0]$	$\mathtt{Z}[n][1] \mathrel{+}= \mathtt{Y}[n]\mathtt{X}[1]$	$\cdots$	$\mathtt{Z}[n][n] \mathrel{+}= \mathtt{Y}[n]\mathtt{X}[n]$

AMX instructions

Instruction Type	Mnemonics(Opcode)	Description
Loads and stores	`ldx(0)`, `ldy(1)`, `stx(2)`, `sty(3)`, `ldz(4)`, `stz(5)`, `ldzi(6)`, `stzi(7)`	Data movement between memory and AMX registers
Extract	`extrh(8)`, `extrx(8)`, `extrv(9)`, `extry(9)`	Data movement within the AMX register file
First generation matrix/vector	`fma64(10)`, `fms64(11)`, `fma32(12)`, `fms32(13)`, `mac16(14)`, `fma16(15)`, `fms16(16)`	Outer or pointwise products with accumulation/subtraction
Second generation matrix and vector	`vecint(18)`, `vecfp(19)`, `matint(20)`, `matfp(21)`	Outer or pointwise products with accumulation/subtraction
Miscellaneous	`set(17)`, `clr(17)`, `genlut(22)`	Context switching, lookup tables

- Matrix-mode `mac16`, `matint`: outer products
- Vector-mode `mac16`, `vecint`: pointwise vector operations
- Bytewise register addressing $\Rightarrow$ free bytewise shifts

AMX performance (M1)

- Note: operation counts consider multiply-accumulate (MACs) – multiplies or additions only halves throughput
- Outer product: building block for matrix multiplication
  - 16-bit FP or 8-bit integer: 3.05 TOPS/s
  - 16-bit integer: 1.53 TOPS/s
- Vector (pointwise) operations
  - FP or 8-bit integer: 381 GOPS/s
  - 16-bit integer: 191 GOPS/s
- Superscalar (2/cycle) execution of some vector instructions on M3, doubling throughput
- NEON 16-bit MAC throughput for comparison: 204.8 GOPS/s
- **Matrix operations much faster than vector operations**

Saber

- Based on the module-Learning with Rounding (LWR) problem
- Small matrix-vector products in $R_q = \mathbb{Z}_q[x]/(x^{256} + 1)$
- Arithmetic in R_q: polynomial arithmetic modulo $q = 2^{13} < 2^{16}$, no explicit modular reduction needed
- NTT-unfriendly ring
- Different security levels by increasing module dimension

Parameter set	Sec. level	ℓ	n	q
LightSaber	1	2	256	2^{13}
Saber	3	3	256	2^{13}
FireSaber	5	4	256	2^{13}

FrodoKEM

- Unstructured lattices – multiplication of "large" square $(n \times n)$ by "thin" $(n \times \overline{n})$ matrices
- Coefficient arithmetic modulo $q = 2^b \leq 2^{16}$

Parameter set	Sec. level	n	q	$\overline{m} = \overline{n}$
Frodo-640	1	640	2^{15}	8
Frodo-976	3	976	2^{16}	8
Frodo-1344	5	1344	2^{16}	8

AMX polynomial multiplication

- Basic idea applied to NTRU by Gazzoni Filho et al (2024)
- Outer products vs. polynomial multiplication parallelogram:

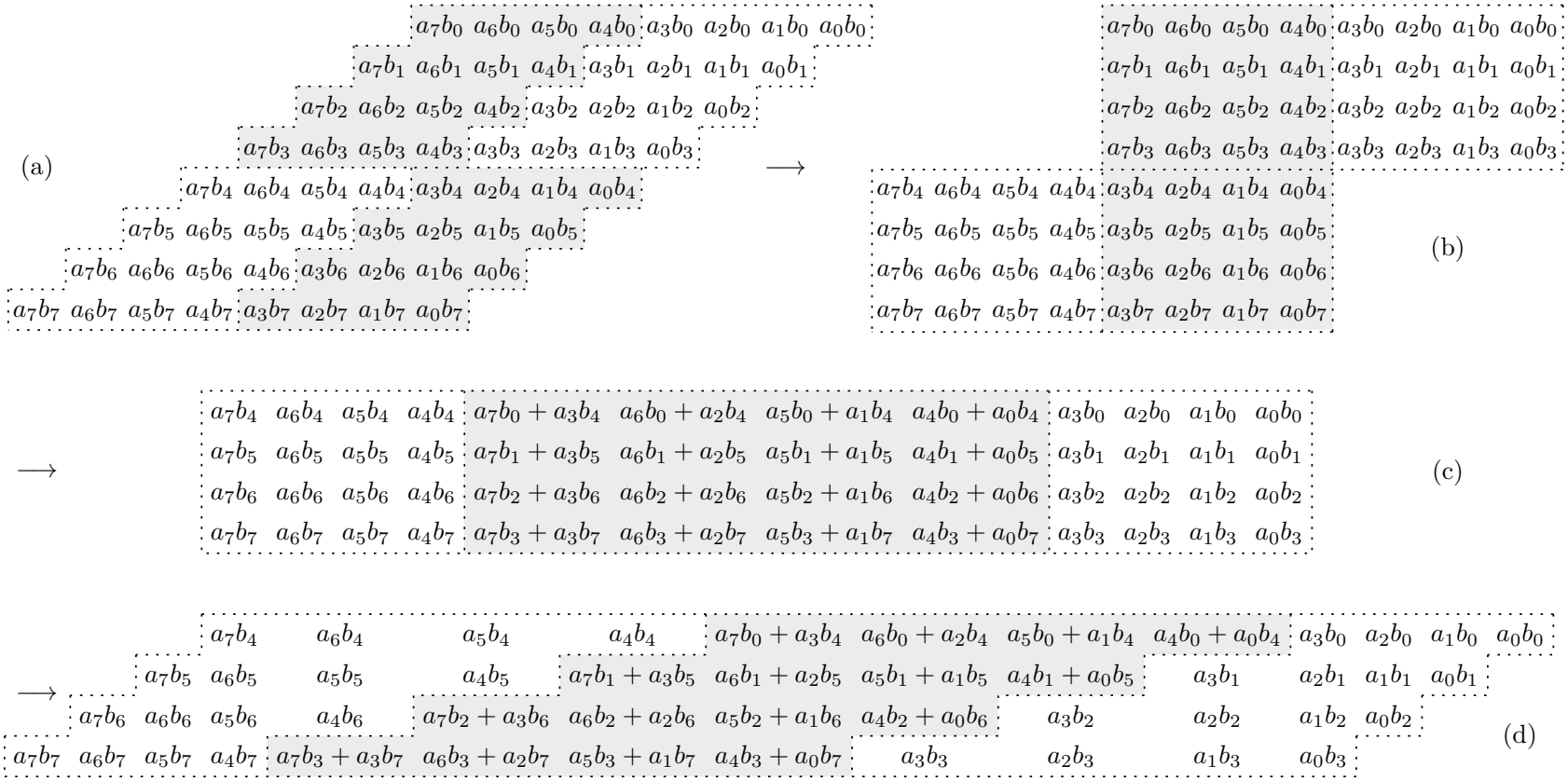
	a_3	a_2	a_1	a_0
b_0	a_3b_0	a_2b_0	a_1b_0	a_0b_0
b_1	a_3b_1	a_2b_1	a_1b_1	a_0b_1
b_2	a_3b_2	a_2b_2	a_1b_2	a_0b_2
b_3	a_3b_3	a_2b_3	a_1b_3	a_0b_3

Left shift i-th row by i positions $\longrightarrow$

		a_3b_0	a_2b_0	a_1b_0	a_0b_0	
	a_3b_1	a_2b_1	a_1b_1	a_0b_1		
a_3b_2	a_2b_2	a_1b_2	a_0b_2			
a_3b_3	a_2b_3	a_1b_3	a_0b_3			

- Sum-reduction of columns to finish polynomial multiplication
- Natural AMX dimension: 64 bytes i.e. 32, 16-bit words
- Combine these basic blocks using a product scanning (columnwise) approach, delaying shifting and sum-reduction to the end of the computation

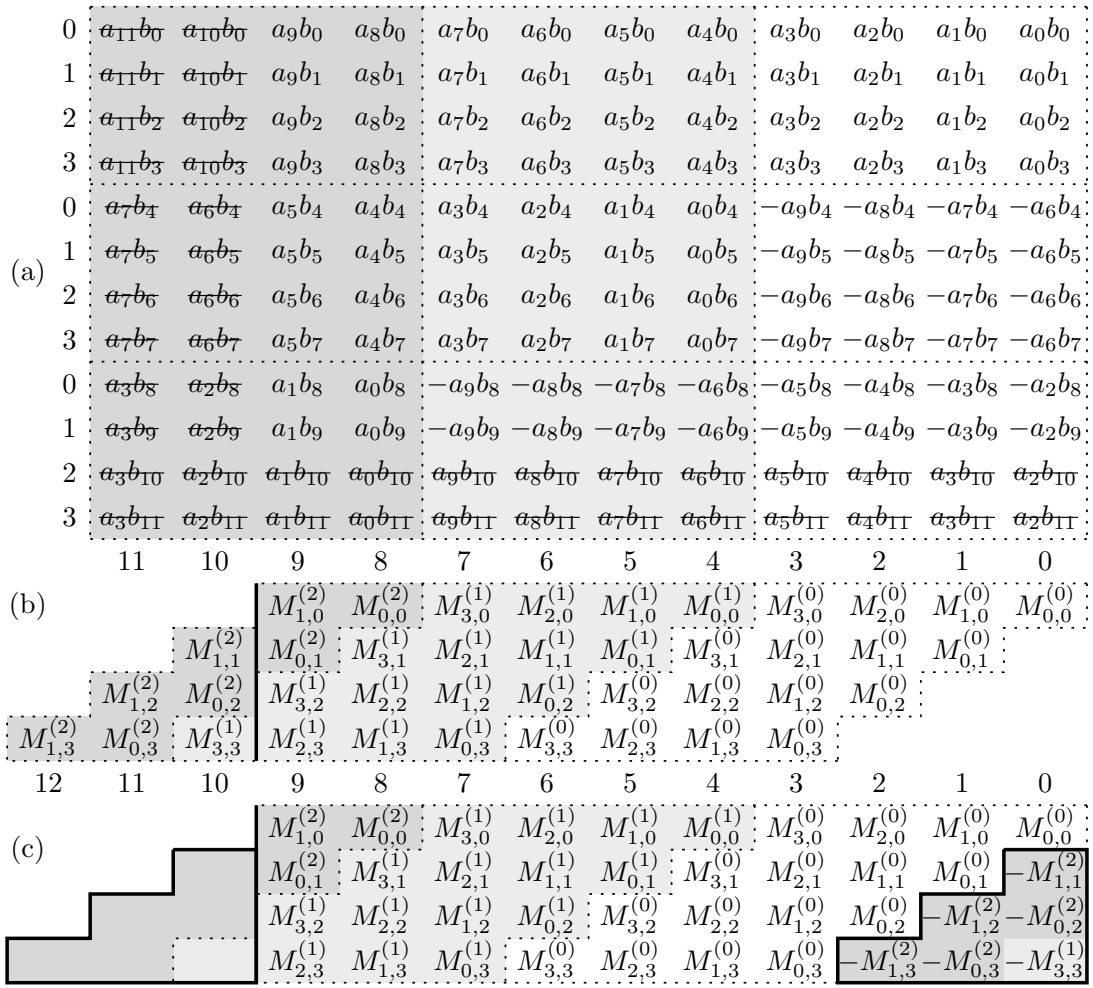
AMX polynomial multiplication



AMX polynomial multiplication, reduction modulo $x^n + 1$

- $O(n)$ shifts and sum-reductions vs. $O(n^2)$ outer products $\Rightarrow$ high ratio of matrix to vector operations
- Must perform polynomial reduction modulo $x^n + 1$
- An integrated procedure decreases (!) the vector operation count, by postponing shifts and sum-reductions after the polynomial reduction
- Next slide: toy example for reduction modulo $x^{10} + 1$

AMX polynomial multiplication, reduction modulo $x^n + 1$



Polynomial multiplication from linear algebra

- The previous techniques were applied to NTRU by Gazzoni Filho et al (2024), which uses a similar (implementation-wise) ring, but modulo $x^n - 1$ rather than $x^n + 1$
- Alone it was competitive, but not a definite improvement, over the state-of-the-art (Becker et al, 2021)
- We propose an alternative technique, casting polynomial multiplication in the language of linear algebra

Polynomial multiplication from linear algebra

- Polynomial multiplication modulo $x^{256} + 1$ can be viewed as a multiplication of a skew-circulant matrix by a vector:

$$\mathbf{Mv} = \begin{bmatrix} a_0 & -a_{255} & -a_{254} & \cdots & -a_2 & -a_1 \\ a_1 & a_0 & -a_{255} & \cdots & -a_3 & -a_2 \\ a_2 & a_1 & a_0 & \cdots & -a_4 & -a_3 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \\ a_{254} & a_{253} & a_{252} & \cdots & a_0 & -a_{255} \\ a_{255} & a_{254} & a_{253} & \cdots & a_1 & a_0 \end{bmatrix} \cdot \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{254} \\ b_{255} \end{bmatrix}$$

- Can decompose $\mathbf{M}$ into 8×8 block Toeplitz matrix, and $\mathbf{v}$ into block 8×1 vector

Polynomial multiplication from linear algebra

- We can write $\mathbf{Mv}$ as

$$\begin{bmatrix} \mathbf{M}_0 & -\mathbf{M}_7 & -\mathbf{M}_6 & -\mathbf{M}_5 & -\mathbf{M}_4 & -\mathbf{M}_3 & -\mathbf{M}_2 & -\mathbf{M}_1 \\ \mathbf{M}_1 & \mathbf{M}_0 & -\mathbf{M}_7 & -\mathbf{M}_6 & -\mathbf{M}_5 & -\mathbf{M}_4 & -\mathbf{M}_3 & -\mathbf{M}_2 \\ \mathbf{M}_2 & \mathbf{M}_1 & \mathbf{M}_0 & -\mathbf{M}_7 & -\mathbf{M}_6 & -\mathbf{M}_5 & -\mathbf{M}_4 & -\mathbf{M}_3 \\ \mathbf{M}_3 & \mathbf{M}_2 & \mathbf{M}_1 & \mathbf{M}_0 & -\mathbf{M}_7 & -\mathbf{M}_6 & -\mathbf{M}_5 & -\mathbf{M}_4 \\ \mathbf{M}_4 & \mathbf{M}_3 & \mathbf{M}_2 & \mathbf{M}_1 & \mathbf{M}_0 & -\mathbf{M}_7 & -\mathbf{M}_6 & -\mathbf{M}_5 \\ \mathbf{M}_5 & \mathbf{M}_4 & \mathbf{M}_3 & \mathbf{M}_2 & \mathbf{M}_1 & \mathbf{M}_0 & -\mathbf{M}_7 & -\mathbf{M}_6 \\ \mathbf{M}_6 & \mathbf{M}_5 & \mathbf{M}_4 & \mathbf{M}_3 & \mathbf{M}_2 & \mathbf{M}_1 & \mathbf{M}_0 & -\mathbf{M}_7 \\ \mathbf{M}_7 & \mathbf{M}_6 & \mathbf{M}_5 & \mathbf{M}_4 & \mathbf{M}_3 & \mathbf{M}_2 & \mathbf{M}_1 & \mathbf{M}_0 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{v}_0 \\ \mathbf{v}_1 \\ \mathbf{v}_2 \\ \mathbf{v}_3 \\ \mathbf{v}_4 \\ \mathbf{v}_5 \\ \mathbf{v}_6 \\ \mathbf{v}_7 \end{bmatrix}$$

$$= \begin{bmatrix} \mathbf{M}_0\mathbf{v}_0 - \mathbf{M}_7\mathbf{v}_1 - \mathbf{M}_6\mathbf{v}_2 - \mathbf{M}_5\mathbf{v}_3 - \mathbf{M}_4\mathbf{v}_4 - \mathbf{M}_3\mathbf{v}_5 - \mathbf{M}_2\mathbf{v}_6 - \mathbf{M}_1\mathbf{v}_7 \\ \mathbf{M}_1\mathbf{v}_0 + \mathbf{M}_0\mathbf{v}_1 - \mathbf{M}_7\mathbf{v}_2 - \mathbf{M}_6\mathbf{v}_3 - \mathbf{M}_5\mathbf{v}_4 - \mathbf{M}_4\mathbf{v}_5 - \mathbf{M}_3\mathbf{v}_6 - \mathbf{M}_2\mathbf{v}_7 \\ \mathbf{M}_2\mathbf{v}_0 + \mathbf{M}_1\mathbf{v}_1 + \mathbf{M}_0\mathbf{v}_2 - \mathbf{M}_7\mathbf{v}_3 - \mathbf{M}_6\mathbf{v}_4 - \mathbf{M}_5\mathbf{v}_5 - \mathbf{M}_4\mathbf{v}_6 - \mathbf{M}_3\mathbf{v}_7 \\ \mathbf{M}_3\mathbf{v}_0 + \mathbf{M}_2\mathbf{v}_1 + \mathbf{M}_1\mathbf{v}_2 + \mathbf{M}_0\mathbf{v}_3 - \mathbf{M}_7\mathbf{v}_4 - \mathbf{M}_6\mathbf{v}_5 - \mathbf{M}_5\mathbf{v}_6 - \mathbf{M}_4\mathbf{v}_7 \\ \mathbf{M}_4\mathbf{v}_0 + \mathbf{M}_3\mathbf{v}_1 + \mathbf{M}_2\mathbf{v}_2 + \mathbf{M}_1\mathbf{v}_3 + \mathbf{M}_0\mathbf{v}_4 - \mathbf{M}_7\mathbf{v}_5 - \mathbf{M}_6\mathbf{v}_6 - \mathbf{M}_5\mathbf{v}_7 \\ \mathbf{M}_5\mathbf{v}_0 + \mathbf{M}_4\mathbf{v}_1 + \mathbf{M}_3\mathbf{v}_2 + \mathbf{M}_2\mathbf{v}_3 + \mathbf{M}_1\mathbf{v}_4 + \mathbf{M}_0\mathbf{v}_5 - \mathbf{M}_7\mathbf{v}_6 - \mathbf{M}_6\mathbf{v}_7 \\ \mathbf{M}_6\mathbf{v}_0 + \mathbf{M}_5\mathbf{v}_1 + \mathbf{M}_4\mathbf{v}_2 + \mathbf{M}_3\mathbf{v}_3 + \mathbf{M}_2\mathbf{v}_4 + \mathbf{M}_1\mathbf{v}_5 + \mathbf{M}_0\mathbf{v}_6 - \mathbf{M}_7\mathbf{v}_7 \\ \mathbf{M}_7\mathbf{v}_0 + \mathbf{M}_6\mathbf{v}_1 + \mathbf{M}_5\mathbf{v}_2 + \mathbf{M}_4\mathbf{v}_3 + \mathbf{M}_3\mathbf{v}_4 + \mathbf{M}_2\mathbf{v}_5 + \mathbf{M}_1\mathbf{v}_6 + \mathbf{M}_0\mathbf{v}_7 \end{bmatrix}$$

Polynomial multiplication from linear algebra

- If we "reshape" the 256×1 result into a 32×8 matrix, it can be computed as:

$$
\begin{aligned}
\mathbf{M} \cdot \mathbf{v} = \mathbf{M}_0 \cdot [\ \mathbf{v}_0 \quad \mathbf{v}_1 \quad \mathbf{v}_2 \quad \mathbf{v}_3 \quad \mathbf{v}_4 \quad \mathbf{v}_5 \quad \mathbf{v}_6 \quad \mathbf{v}_7\ ] \\
+ \mathbf{M}_1 \cdot [\ -\mathbf{v}_7 \quad \mathbf{v}_0 \quad \mathbf{v}_1 \quad \mathbf{v}_2 \quad \mathbf{v}_3 \quad \mathbf{v}_4 \quad \mathbf{v}_5 \quad \mathbf{v}_6\ ] \\
+ \mathbf{M}_2 \cdot [\ -\mathbf{v}_6 \quad -\mathbf{v}_7 \quad \mathbf{v}_0 \quad \mathbf{v}_1 \quad \mathbf{v}_2 \quad \mathbf{v}_3 \quad \mathbf{v}_4 \quad \mathbf{v}_5\ ] \\
+ \mathbf{M}_3 \cdot [\ -\mathbf{v}_5 \quad -\mathbf{v}_6 \quad -\mathbf{v}_7 \quad \mathbf{v}_0 \quad \mathbf{v}_1 \quad \mathbf{v}_2 \quad \mathbf{v}_3 \quad \mathbf{v}_4\ ] \\
+ \mathbf{M}_4 \cdot [\ -\mathbf{v}_4 \quad -\mathbf{v}_5 \quad -\mathbf{v}_6 \quad -\mathbf{v}_7 \quad \mathbf{v}_0 \quad \mathbf{v}_1 \quad \mathbf{v}_2 \quad \mathbf{v}_3\ ] \\
+ \mathbf{M}_5 \cdot [\ -\mathbf{v}_3 \quad -\mathbf{v}_4 \quad -\mathbf{v}_5 \quad -\mathbf{v}_6 \quad -\mathbf{v}_7 \quad \mathbf{v}_0 \quad \mathbf{v}_1 \quad \mathbf{v}_2\ ] \\
+ \mathbf{M}_6 \cdot [\ -\mathbf{v}_2 \quad -\mathbf{v}_3 \quad -\mathbf{v}_4 \quad -\mathbf{v}_5 \quad -\mathbf{v}_6 \quad -\mathbf{v}_7 \quad \mathbf{v}_0 \quad \mathbf{v}_1\ ] \\
+ \mathbf{M}_7 \cdot [\ -\mathbf{v}_1 \quad -\mathbf{v}_2 \quad -\mathbf{v}_3 \quad -\mathbf{v}_4 \quad -\mathbf{v}_5 \quad -\mathbf{v}_6 \quad -\mathbf{v}_7 \quad \mathbf{v}_0\ ]
\end{aligned}
$$

Outer level Saber M-V product improvements

- Saber performs matrix-vector products, with elements in $\mathbb{Z}_q/(x^{256}+1)$: $\mathbf{A}^T\mathbf{s}+\mathbf{h}$ and $\mathbf{A}\mathbf{s}'+\mathbf{h}$ in PKE key generation and encryption, respectively ($\mathbf{A}$ is $\ell \times \ell$ for $\ell \in \{2,3,4\}$)
- We noticed that (example for PKE encryption and $\ell = 2$):

$$\mathbf{A}\mathbf{s}' = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \begin{pmatrix} s'_0 \\ s'_1 \end{pmatrix} = s'_0 \begin{pmatrix} A_{00} \\ A_{10} \end{pmatrix} + s'_1 \begin{pmatrix} A_{01} \\ A_{11} \end{pmatrix},$$

- Allows batching multiplications by s'_0 and s'_1
- We reuse the technique of the previous slides to write $\mathbf{A}\mathbf{s}'$ as a sum of 8 multiplications of 32×32 by $32 \times 8\ell$ matrices, increasing AMX utilization (full utilization for $\ell = 4$)
- One of the reviewers pointed out an analogous trick for the baseline (polymul) implementation, which sped it up nicely

Saber performance results (kilocycles, M3 only)

Best-performing implementation and memory allocation in each case

Sec. level	Type	Keygen	Encaps	Decaps	M-V mul
1	NEON	19.1	26.3	25.7	3.96
	AMX-NTRU	18.5	25.6	24.5	3.28
	AMX-TMVP	17.5	24.3	23.2	2.32
NEON/AMX-TMVP (×)		**1.09**	**1.08**	**1.11**	**1.71**
3	NEON	31.4	40.2	40.3	7.52
	AMX-NTRU	32.3	40.9	40.9	7.43
	AMX-TMVP	28.4	36.5	36.4	3.62
NEON/AMX-TMVP (×)		**1.11**	**1.10**	**1.11**	**2.08**
5	NEON	48.3	59.7	59.8	12.1
	AMX-NTRU	51.3	62.2	62.4	13.3
	AMX-TMVP	42.9	53.2	53.5	4.83
NEON/AMX-TMVP (×)		**1.12**	**1.12**	**1.12**	**2.51**

FrodoKEM on AMX

- FrodoKEM is based on matrix multiplication and should be a nice match to AMX, but we ran into some issues
- Data layout mismatches require transpositions (easy but not free with AMX); can't generate $\mathbf{A}^\mathsf{T}$ directly
- **AS** in particular requires two transposes, **SA** only one
- AMX is underutilized due to $n \times 8$ dimension of $\mathbf{S}$ and $\mathbf{S}'$
- Use batching by a factor of 4 in encapsulation/decapsulation
- Given there is no publicly available FrodoKEM implementation in NEON, we wrote our own to have a fair baseline

Gaussian sampling

- Random sampling hard to perform in constant-time for some distributions (e.g. FrodoKEM's rounded continuous Gaussian)
- Inversion sampling using a table, requires an inefficient full scan to implement in constant-time
- `genlut`: peculiar instruction with generate and lookup modes
- Two input (vector) registers, a source **S** and a table **T**
- For sorted tables, generate mode works like a search
- Lookup similar to Intel `pshufb`, NEON `tbl` instructions
- In either case, works in parallel on all 32 lanes (and is fast)
- Trivial changes to FrodoKEM sampling tables allowed us to repurpose `genlut` for table-based inversion sampling, with excellent performance; experiments suggest it is constant-time

FrodoKEM performance results (kilocycles, M3 only)

Best-performing implementation and memory allocation in each case

Sec. level	Type	Keygen	Encaps	Decaps	E 4×	D 4×	Sampling
1	Opt	558	669	641	1755	1755	4.59
	NEON	468	561	532	1395	1387	4.35
	AMX	414	494	447	907	905	0.84
NEON/AMX (×)		**1.13**	**1.14**	**1.19**	**1.54**	**1.53**	**5.18**
3	Opt	1220	1310	1255	3264	3098	5.99
	NEON	940	1070	1005	2594	2419	5.65
	AMX	839	930	845	1555	1381	1.28
NEON/AMX (×)		**1.12**	**1.15**	**1.19**	**1.67**	**1.75**	**4.41**
5	Opt	1931	2156	2061	5807	5569	5.48
	NEON	1573	1766	1681	4249	4004	5.09
	AMX	1396	1500	1388	2352	2101	1.76
NEON/AMX (×)		**1.13**	**1.18**	**1.21**	**1.81**	**1.91**	**2.88**

FrodoKEM performance results (kilocycles, M3 only)

Best-performing implementation and memory allocation in each case

Sec. level	Type	$\mathbf{AS+E}$		$\mathbf{S'A+E}$		$\mathbf{S'A+E}$ 4×	
		Full	Mat mul	Full	Mat mul	Full	Mat mul
1	Opt	354	192	345	189	924	745
	NEON	263	118	254	108	614	452
	AMX	226	77.7	197	52.4	211	64.1
NEON/AMX (×)		**1.16**	**1.52**	**1.29**	**2.05**	**2.91**	**7.05**
3	Opt	880	418	824	384	2001	1533
	NEON	600	247	586	247	1423	1001
	AMX	530	181	461	125	485	148
NEON/AMX (×)		**1.13**	**1.37**	**1.27**	**1.97**	**2.93**	**6.78**
5	Opt	1479	808	1474	766	4000	3111
	NEON	1116	465	1113	471	2690	1942
	AMX	970	328	855	229	887	252
NEON/AMX (×)		**1.15**	**1.42**	**1.30**	**2.06**	**3.03**	**7.70**

Conclusions and future work

- AMX suitable for polynomial multiplication, but must ensure high matrix/vector operation ratio
- Current cryptosystems are best suited to CPU implementations; designers may wish to revisit parameter choices to favor matrix multiplication accelerators
- Symmetric primitives becoming the bottleneck of PQC
- Future work:
  - Application to NTT-based schemes (e.g. Kyber, Dilithium)
  - Application of `genlut` to other schemes

Thank you!

Questions?

Memory performance issues

- Performance anomalies observed in NTRU (and Saber), especially in the M1
- Observation by M. Handley that concurrent AMX/CPU memory accesses block; not only for the same addresses or even the same cache line, but rather a full memory page
- Arrays of polynomial coefficients in reference/optimized implementation are allocated on the stack, right next to other variables used by the CPU for other routines
- Solution: allocate arrays using `mmap` (one page per array)

Saber PKE

Algorithm II.1
`Saber.PKE.KeyGen()`

Input: None
Output: Key pair (pk, sk)
1: $seed_{\mathbf{A}} \leftarrow \mathcal{U}(\{0,1\}^{256})$
2: $\mathbf{A} \leftarrow \mathtt{gen}(seed_{\mathbf{A}}) \in R_q^{l \times l}$
3: $r \leftarrow \mathcal{U}(\{0,1\}^{256})$
4: $\mathbf{s} \leftarrow \beta_\mu(R_q^{l \times 1}; r)$
5: $\mathbf{b} \leftarrow ((\mathbf{A}^\mathsf{T}\mathbf{s}+\mathbf{h}) \bmod q) \gg (\epsilon_q - \epsilon_p) \in R_p^{l \times 1}$
6: **return** $(pk := (seed_{\mathbf{A}}, \mathbf{b}), sk := \mathbf{s})$

Algorithm II.2
`Saber.PKE.Dec`(sk, c)

Input: Secret key sk, ciphertext c
Output: Message m'
1: $v \leftarrow \mathbf{b'}^\mathsf{T}(\mathbf{s} \bmod p) \in R_p$
2: $m' \leftarrow ((v - 2^{\epsilon_p - \epsilon_T} c_m + h_2) \bmod p) \gg (\epsilon_p - 1) \in R_2$
3: **return** m'

Algorithm II.3
`Saber.PKE.Enc`$(pk, m; r)$

Input: Public key pk, message $m \in R_2$, optional randomness r
Output: Ciphertext c
1: $\mathbf{A} \leftarrow \mathtt{gen}(seed_{\mathbf{A}}) \in R_q^{l \times l}$
2: **if** r is not specified **then**
3: $\quad r \leftarrow \mathcal{U}(\{0,1\}^{256})$
4: $\mathbf{s'} \leftarrow \beta_\mu(R_q^{l \times 1}; r)$
5: $\mathbf{b'} \leftarrow ((\mathbf{A}\mathbf{s'} + \mathbf{h}) \bmod q) \gg (\epsilon_q - \epsilon_p) \in R_p^{l \times 1}$
6: $v' \leftarrow \mathbf{b}^\mathsf{T}(\mathbf{s'} \bmod p) \in R_p$
7: $c_m \leftarrow (v' + h_1 - 2^{\epsilon_p - 1} m \bmod p) \gg (\epsilon_p - \epsilon_T) \in R_T$
8: **return** $c := (c_m, \mathbf{b'})$

FrodoKEM PKE

Algorithm II.4
`FrodoPKE.KeyGen()`

Input: None
Output: Key pair (pk, sk)

1: $seed_{\mathbf{A}} \leftarrow \mathcal{U}(\{0,1\}^{l_{\mathbf{A}}})$
2: $\mathbf{A} \leftarrow Gen(seed_{\mathbf{A}})$
3: $seed_{\mathbf{SE}} \leftarrow \mathcal{U}(\{0,1\}^{l_{\mathbf{SE}}})$
4: $\mathbf{r} \leftarrow \text{SK}(\texttt{0x5F}||seed_{\mathbf{SE}}, 2n\overline{n} \cdot 16)$
5: $\mathbf{S}^{\mathsf{T}} \leftarrow SM(\mathbf{r}[0 : n\overline{n} - 1], \overline{n}, n)$
6: $\mathbf{E} \leftarrow SM(\mathbf{r}[n\overline{n} : 2n\overline{n} - 1], n, \overline{n})$
7: $\mathbf{B} = \mathbf{AS} + \mathbf{E}$
8: **return** $(pk := (seed_{\mathbf{A}}, \mathbf{B}), sk := \mathbf{S}^{\mathsf{T}})$

Algorithm II.5
`FrodoPKE.Dec`(sk, c)

Input: Secret key sk, ciphertext c
Output: Message m'

1: $\mathbf{M} = \mathbf{C}_2 - \mathbf{C}_1\mathbf{S}$
2: **return** $m' := Decode(\mathbf{M})$

Algorithm II.6
`FrodoPKE.Enc`(pk, m, r)

Input: Public key pk, message m
Output: Ciphertext c

1: $\mathbf{A} \leftarrow Gen(seed_{\mathbf{A}})$
2: $seed_{\mathbf{SE}} \leftarrow \mathcal{U}(\{0,1\}^{l_{\mathbf{SE}}})$
3: $\mathbf{r} \leftarrow \text{SK}(\texttt{0x96}||seed_{\mathbf{SE}}, (2\overline{m}n + \overline{m}n) \cdot 16)$
4: $\mathbf{S}' \leftarrow SM(\mathbf{r}[0 : \overline{m}n - 1], \overline{m}, n)$
5: $\mathbf{E}' \leftarrow SM(\mathbf{r}[\overline{m}n : 2\overline{m}n - 1], \overline{m}, n)$
6: $\mathbf{E}'' \leftarrow SM(\mathbf{r}[2\overline{m}n : 2\overline{m}n + \overline{m}n - 1], \overline{m}, \overline{n})$
7: $\mathbf{B}' = \mathbf{S}'\mathbf{A} + \mathbf{E}'; \mathbf{V} = \mathbf{S}'\mathbf{B} + \mathbf{E}''$
8: **return** $c := (\mathbf{C}_1, \mathbf{C}_2) = (\mathbf{B}', \mathbf{V} + Encode(m))$