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

Décio Luiz Gazzoni Filho*†, Guilherme Brandão‡, Gora Adj§,
Arwa Alblooshi§, Isaac A. Canales-Martínez§, Jorge Chávez-Saab§ and Julio López*
* Instituto de Computação, Universidade Estadual de Campinas (UNICAMP), Campinas, Brazil
Email: {decio.gazzoni,jlopez}@ic.unicamp.br
† Dept. of Electrical Engineering, State University of Londrina, Londrina, Brazil. Email: dgazzoni@uel.br
‡ Independent Researcher, Londrina, Brazil. Email: brandaogbs@gmail.com
§ Cryptography Research Centre, Technology Innovation Institute, Abu Dhabi, UAE
Email: {gora.adj,arwa.alblooshi,isaac.canales,jorge.saab}@tii.ae

***Abstract*—As CPU performance is unable to keep up with the dramatic growth of the past few decades, CPU architects are looking into domain-specific architectures to accelerate certain tasks. A recent trend is the introduction of matrix-multiplication accelerators to CPUs by manufacturers such as IBM, Intel and ARM, some of which have not launched commercially yet. Apple's systems-on-chip (SoCs) for its mobile phones, tablets and personal computers include a proprietary, undocumented CPU-coupled matrix multiplication coprocessor called AMX. In this paper, we leverage AMX to accelerate the post-quantum lattice-based cryptosystems Saber and FrodoKEM, and benchmark their performance on Apple M1 and M3 SoCs. We propose a variant of the Toeplitz Matrix-Vector Product algorithm for polynomial multiplication, which sets new speed records for Saber using AMX (up to 13% for the main KEM operations, and 151% for matrix-vector multiplication of polynomials). For FrodoKEM, we set new speed records with our AMX implementation (up to 21% for the main KEM operations, and 124% for matrix multiplication, with even greater improvements for $4\times$-batching). Such speedups are relative to our optimized NEON implementation, also presented here, which improves upon the state-of-the-art implementation for ARMv8 CPUs.**

***Index Terms*—Post-quantum cryptography, AMX, ARM, NEON, FrodoKEM, Saber**

I. Introduction

Quantum computers pose a threat to cryptographic schemes whose security rely on the presumed hardness of computational problems such as finding discrete logarithms or factoring integers. Post-quantum cryptography (PQC) refers to cryptographic systems that remain secure against attacks employing quantum and classical computers. In 2017, the National Institute of Standards and Technology (NIST) called for a PQC standardization process; three out of the four selected candidates for standardization are lattice-based. Saber [1] and FrodoKEM [2] are lattice-based Key Encapsulation Mechanisms that reached round 3 in the standardization process; the latter is recommended by the German BSI [3] and is under consideration for standardization by ISO [4].

The performance bottlenecks of lattice-based cryptography usually lie in polynomial/matrix multiplication and symmetric-cryptography operations, prompting extensive research efforts to enhance their efficiency. Various manufacturers have developed high-performance AI accelerators, such as NVIDIA's tensor cores [5], Intel's Advanced Matrix Extensions (AMX) [6] and ARMv9-A's Scalable Matrix Extensions (SME) [7], to cater to the high demands of AI applications. Apple's AMX (unrelated to Intel's) is an undocumented coprocessor found in its SoCs, starting with the 2019's A13 [8, Section 7.6], which we apply to Saber and FrodoKEM in this work.

A. Related works

Many studies target GPU cores, achieving high throughput via huge batching levels, but compromising latency. Gazzoni Filho et al. [9] presented the first cryptographic implementation on a CPU-linked matrix multiplication accelerator, setting new NTRU speed records using Apple's AMX coprocessor on M1/M3 SoCs, beating state-of-the-art NEON implementations with low latency, no batching and running in constant time.

Becker et al. [10] set the current speed record for Saber [1] on ARMv8-A using $\mathcal{O}(n \log n)$ Number Theoretical Transform (NTT) methods combined with a novel "Barrett multiplication" algorithm for modular multiplication, achieving a speedup of 56% over the previous state-of-the-art on the Apple M1. We also remark the work in [11], which introduced innovative Toeplitz Matrix-Vector Product (TMVP) formulas, with the "four-way" formula standing out as the best non-NTT-like multiplication algorithm for Saber's ring on Cortex-M4.

The state-of-the-art implementation for FrodoKEM [2] is the work of Bos et al. [12], which improves matrix multiplication through a row-wise blocking and packing approach, and also proved that Strassen's algorithm improves throughput for use cases with high batching levels. An ARMv8 implementation using NEON was presented shortly after in [13], claiming a speedup of $10.22\times$ at the protocol level. However, they do not acknowledge the improvements of [12] which, while not ARM-specific, appear to be superior on M1 and M3.[1]

B. Our contributions

We present an optimized AMX implementation of Saber, adapting the techniques from [9] as well as a novel method that

[1] The implementation of [13] is not publicly available and the authors could not be reached for clarification. Our benchmarks of Section V show speedups for [12] that exceed those reported by [13] for their implementation.

increases AMX utilization in Saber's matrix-vector products based on the TMVP approach [11]. This sets new speed records on Apple M1 and M3, with speedups of up to 13% at the protocol level and 151% for the polynomial operations.

For FrodoKEM, we first present a NEON implementation of our own to use as a baseline, which already sets new speed records on the M1/M3. We then present our AMX implementation, which improves further on our NEON record. Both implementations explore possible matrix multiplication strategies and use a novel technique for generating FrodoKEM-AES's $\mathbf{A}$ matrix. We make an innovative use of AMX's unique `genlut` instruction to perform Gaussian sampling, improving it by up to 418% versus a NEON implementation. This might be of particular interest for other applications. Compared to the state of the art, we improve on the M1 and M3 by up to 21% at the protocol level and 124% for matrix multiplication. Then, we develop $4\times$-batched NEON and AMX implementations, showing that AMX is significantly faster than NEON, by up to 91% at the protocol level and 708% for matrix multiplication.

We make all our code available at https://github.com/... [2]

II. Preliminaries

A public-key encryption scheme (PKE) is a tuple of algorithms $(\texttt{KeyGen}, \texttt{Enc}, \texttt{Dec})$. `KeyGen` generates a public key pk and a secret key sk. `Enc` outputs a ciphertext c given pk and a message m. `Dec` outputs a message m' from sk and c. A key encapsulation mechanism (KEM) is a tuple of algorithms $(\texttt{KeyGen}, \texttt{Encaps}, \texttt{Decaps})$. `KeyGen` generates a public key pk and a secret key sk. `Encaps` outputs a shared key ss and a ciphertext c given pk. `Decaps` outputs a shared key ss' from sk and c. We present next KEMs obtained from PKEs via a variant of the Fujisaki-Okamoto transform; we only show PKE algorithms, which are the target of our optimizations.

Bold lower case denotes vectors and bold upper case denotes matrices. We write $\mathbf{v}[i : j : k]$ for a matrix/vector slice of coefficients $i, i+j, i+2j, \ldots, i+k$; $j = 1$ if omitted. Sampling from a uniform distribution over a set S is denoted $x \leftarrow \mathcal{U}(S)$.

A. Saber

Saber [1] is a lattice-based KEM relying on the hardness of Module Learning With Rounding. Its NIST submission specifies the parameter sets below for security levels 1, 3, and 5.

Parameter set	Sec. level	l	n	q	p	T	μ
LightSaber	1	2	256	2^{13}	2^{10}	2^3	10
Saber	3	3	256	2^{13}	2^{10}	2^4	8
FireSaber	5	4	256	2^{13}	2^{10}	2^6	6

Saber works over the ring $R_q := \mathbb{Z}_q[X]/(X^n + 1)$ and employs the binomial distribution centered at μ, denoted β_μ, hash functions $\mathcal{F}, \mathcal{G}, \mathcal{H}$, and a function `gen` to generate a pseudorandom matrix from a seed. We have that $q = 2^{\epsilon_q}, p = 2^{\epsilon_p}, T = 2^{\epsilon_T}$. Let $\mathbf{s} \leftarrow \beta_\mu(R_q^l; r)$ denote sampling each coordinate of a vector $\mathbf{s} \in R_q^l$ pseudorandomly from the distribution $\beta_\mu(R_q)$ with seed r. Algorithms II.1, II.3 and II.2 are a verbatim reproduction of Saber's PKE specification.

[2] A GitHub repository will be made available following the paper's publication.

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

Input: None
Output: Key pair (pk, sk)
$seed_{\mathbf{A}} \leftarrow \mathcal{U}(\{0,1\}^{256})$
$\mathbf{A} \leftarrow \texttt{gen}(seed_{\mathbf{A}}) \in R_q^{l \times l}$
$r \leftarrow \mathcal{U}(\{0,1\}^{256})$
$\mathbf{s} \leftarrow \beta_\mu(R_q^{l\times 1}; r)$
$\mathbf{b} \leftarrow ((\mathbf{A}^\mathsf{T}\mathbf{s} + \mathbf{h}) \bmod q) \gg (\epsilon_q - \epsilon_p) \in R_p^{l\times 1}$
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'
$v \leftarrow \mathbf{b}'^\mathsf{T}(\mathbf{s} \bmod p) \in R_p$
$m' \leftarrow ((v - 2^{\epsilon_p - \epsilon_T} c_m + h_2) \bmod p) \gg (\epsilon_p - 1) \in R_2$
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
$\mathbf{A} \leftarrow \texttt{gen}(seed_{\mathbf{A}}) \in R_q^{l \times l}$
if r is not specified **then**
  $r \leftarrow \mathcal{U}(\{0,1\}^{256})$
$\mathbf{s}' \leftarrow \beta_\mu(R_q^{l\times 1}; r)$
$\mathbf{b}' \leftarrow ((\mathbf{A}\mathbf{s}' + \mathbf{h}) \bmod q) \gg (\epsilon_q - \epsilon_p) \in R_p^{l\times 1}$
$v' \leftarrow \mathbf{b}^\mathsf{T}(\mathbf{s}' \bmod p) \in R_p$
$c_m \leftarrow (v' + h_1 - 2^{\epsilon_p - 1} m \bmod p) \gg (\epsilon_p - \epsilon_T) \in R_T$
return $c := (c_m, \mathbf{b}')$

B. FrodoKEM

FrodoKEM [2] is a lattice-based KEM that relies on the hardness of Learning With Errors. The submission to NIST specifies the parameters as in the table below.

Parameter set	Sec. level	n	q	$\overline{m} = \overline{n}$	$l_{\mathbf{A}}$	$l_{\mathbf{SE}}$
Frodo-640	1	640	2^{15}	8	128	128
Frodo-976	3	976	2^{16}	8	128	192
Frodo-1344	5	1344	2^{16}	8	128	256

FrodoKEM uses a function $Gen(s)$ to generate a matrix $\mathbf{A} \in \mathbb{Z}_q^{n\times n}$ pseudorandomly from a seed s of length $l_{\mathbf{A}}$ (using AES or SHAKE), and a function $SM(r, s, t)$ for inversion sampling of a matrix in $\mathbb{Z}_q^{s\times t}$ using a pseudorandom array of 16-bit integers $\mathbf{r}$ and a precomputed table T_χ for an error distribution χ. Let SK denote SHAKE. The PKE specification is given by Algorithms II.4, II.6 and II.5.

Algorithm II.4 `FrodoPKE.KeyGen()`

Input: None
Output: Key pair (pk, sk)
$seed_{\mathbf{A}} \leftarrow \mathcal{U}(\{0,1\}^{l_{\mathbf{A}}})$
$\mathbf{A} \leftarrow Gen(seed_{\mathbf{A}})$
$seed_{\mathbf{SE}} \leftarrow \mathcal{U}(\{0,1\}^{l_{\mathbf{SE}}})$
$\mathbf{r} \leftarrow \mathrm{SK}(\texttt{0x5F} || seed_{\mathbf{SE}}, 2n\overline{n} \cdot 16)$
$\mathbf{S}^\mathsf{T} \leftarrow SM(\mathbf{r}[0 : n\overline{n} - 1], \overline{n}, n)$
$\mathbf{E} \leftarrow SM(\mathbf{r}[n\overline{n} : 2n\overline{n} - 1], n, \overline{n})$
$\mathbf{B} = \mathbf{AS} + \mathbf{E}$
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'
$\mathbf{M} = \mathbf{C}_2 - \mathbf{C}_1\mathbf{S}$
return $m' := Decode(\mathbf{M})$

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

Input: Public key pk, message m
Output: Ciphertext c
$\mathbf{A} \leftarrow Gen(seed_{\mathbf{A}})$
$seed_{\mathbf{SE}} \leftarrow \mathcal{U}(\{0,1\}^{l_{\mathbf{SE}}})$
$\mathbf{r} \leftarrow \mathrm{SK}(\texttt{0x96} || seed_{\mathbf{SE}}, (2\overline{m}n + \overline{m}\overline{n}) \cdot 16)$
$\mathbf{S}' \leftarrow SM(\mathbf{r}[0 : \overline{m}n - 1], \overline{m}, n)$
$\mathbf{E}' \leftarrow SM(\mathbf{r}[\overline{m}n : 2\overline{m}n - 1], \overline{m}, n)$
$\mathbf{E}'' \leftarrow SM(\mathbf{r}[2\overline{m}n : 2\overline{m}n + \overline{m}\overline{n} - 1], \overline{m}, \overline{n})$
$\mathbf{B}' = \mathbf{S}'\mathbf{A} + \mathbf{E}'; \mathbf{V} = \mathbf{S}'\mathbf{B} + \mathbf{E}''$
return $c := (\mathbf{C}_1, \mathbf{C}_2) = (\mathbf{B}', \mathbf{V} + Encode(m))$

C. The AMX coprocessor

AMX is a matrix multiplication coprocessor found in Apple SoCs. It lacks official documentation, so we turn to the reverse engineering efforts of [14]–[16]. We briefly review some concepts and refer to them for more details, as well as the description in [9], on which our algorithmic notation is based.

AMX's register file is comprised of 80 64-byte registers: 16 input registers, split as 8 `X` and 8 `Y` registers, and 64 output `Z` registers viewed as rows of a matrix, as depicted in Figure 1. Some instructions can address `X` and `Y` registers bytewise as 512-byte circular buffers. AMX instructions are encoded within a reserved opcode space of A64; once no longer speculative, the CPU forwards them to the AMX coprocessor.

	X[0]	$\cdots$	X[n]
Y[0]	Z[0][0] += Y[0]X[0]	$\cdots$	Z[0][n] += Y[0]X[n]
Y[1]	Z[1][0] += Y[1]X[0]	$\cdots$	Z[1][n] += Y[1]X[n]
$\vdots$	$\vdots$	$\ddots$	$\vdots$
Y[n]	Z[n][0] += Y[n]X[0]	$\cdots$	Z[n][n] += Y[n]X[n]

Fig. 1. AMX register file organization.

Data transfer between the CPU is AMX is solely done through memory, using load (`ldx`, `ldy`, `ldz`) and store (`stx`, `sty` and `stz`) instructions. `extrh` and `extrv` move rows and columns, respectively, of `Z` to `X` or `Y` registers.

The vector-mode `mac16` and `vecint` instructions realize vector operations such as addition $+$ and the Hadamard (pointwise) product $\circ$. Outer product of a column by a row vector (the BLAS Level-2 rank-1 update operation `xGER`) is realized by the matrix-mode `mac16` and `matint` instructions. For 16-bit integers, vectors (or matrix rows/columns) are up to 32 elements long; each instruction's enable modes can mask part of the computation if smaller sizes are needed.

We illustrate the notation with AMX's primary application, matrix multiplication (in our case, 32×32 matrices with 16-bit integer data), in Algorithm II.7. It is also a basic block, with suitable modifications, for our Saber and FrodoKEM AMX implementations. If $\mathbf{A}^\intercal$ rather than $\mathbf{A}$ is input to the algorithm, we eschew the transposition by removing lines 1 and 2, and replacing line 4 with a load of the i-th row of $\mathbf{A}$.

Algorithm II.7 MATMULADD($\mathbf{A},\mathbf{B}$): Compute $\mathtt{Z}[0:2:62] \leftarrow \mathtt{Z}[0:2:62] + \mathbf{AB}$ using AMX.

Input: $\mathbf{A},\mathbf{B} \in \mathbb{Z}_{2^{16}}^{32\times 32}$ in row-major memory layout.
Output: $\mathtt{Z}[0:2:62] + \mathbf{AB} \in \mathbb{Z}_{2^{16}}^{32\times 32}$ in even `Z` registers.

1: **for** $i = 0$ to 31 **do** $\triangleright$ Load $\mathbf{A}$ to odd `Z` rows
2: $\quad \mathtt{Z}[2i+1] \leftarrow \mathtt{ldz}(\mathbf{A}[i][0:31])$
3: **for** $i = 0$ to 31 **do**
4: $\quad \mathtt{Y}_0 \leftarrow \mathtt{extrv}(\mathtt{Z}[1:2:63][i])$ $\triangleright$ $\mathbf{A}$ transpose step
5: $\quad \mathtt{X}_0 \leftarrow \mathtt{ldx}(\mathbf{B}[i][0:31])$
6: $\quad \mathtt{Z}[0:2:62] \leftarrow \mathtt{mac16}(\mathtt{Z}[0:2:62] + \mathtt{Y}_0^\intercal \mathtt{X}_0)$

We also review the `genlut` instruction, which is instrumental to our table-based sampling technique of Section IV-D. It has two distinct modes, *generate* and *lookup*. The latter is similar to NEON's `TBL` instruction: given an input register with a densely packed array of lane indices (in a format fully described in [16]) and another register containing a table, it performs a table lookup operation; in 16-bit mode, registers are 32 elements wide. The *generate* mode is especially interesting, and unlike any CPU instruction we are familiar with. It takes a table T and source register V as input, and generates a packed array of lane indices, in the format used by *lookup* mode, by searching for the minimum index i such that $T[i] > V[l]$, for each lane l of the source. If T is sorted in ascending order, `genlut` returns i such that $T[i] \le V[l] < T[i+1]$.

III. SABER ON AMX

We now discuss AMX-accelerated multiplication in $\mathbb{Z}_{2^{16}}[X]/(X^n+1)$, which is Saber's main algorithmic task.

A. Baseline implementation

An AMX-based algorithm was previously proposed in [9] for multiplication in $\mathbb{Z}_{2^{16}}[X]/(X^n-1)$, which is very similar to the ring used in Saber, implementation-wise. Indeed, the reduction modulus X^n+1 is identical to the reduction modulus X^n-1, except for flipping signs of terms with powers greater than $n-1$ before reducing them. We achieve this via `vecint` and `matint` instructions, which generalize vector- and matrix-mode `mac16` (respectively) with accumulation by either adding or subtracting. We refer to [9] for a full explanation of the techniques, and only mention the key changes needed to adapt its `PolyModMul` algorithm to Saber:

- Replacing the `mac16` instruction in line 9 of the `AccumulateOuterProductsReduction` subroutine by `matint` using accumulation by subtraction.
- Modifying the `vecint` instructions in lines 8 and 13 of the `MergeFirstAndLastBlocks` subroutine to perform accumulation by subtraction.

B. TMVP-based implementation

We present a second method for polynomial multiplication, which has the option of performing the batched multiplication of a single polynomial $b(x)$ by multiple polynomials $a^{(l)}(x)$. The method is based on the *Toeplitz matrix-vector product* approach introduced by Paksoy and Cenk [11], which computes the coefficients for a single product $c(x) := a(x)b(x)$ as

$$\begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{n-1} \end{pmatrix} = \begin{pmatrix} b_0 & -b_{n-1} & -b_{n-2} & \cdots & -b_1 \\ b_1 & b_0 & -b_{n-1} & \cdots & -b_2 \\ b_2 & b_1 & b_0 & \cdots & -b_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ b_{n-1} & b_{n-2} & b_{n-3} & \cdots & b_0 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{pmatrix},$$

which we denote $\mathbf{c} = \mathbf{Ma}$. We represent the batched products by promoting $\mathbf{c}$ and $\mathbf{a}$ to matrices, with one column per $a^{(l)}(x)$, a trick first used in the CUDA implementation from [17]. For the remainder, we fix $n = 256$ and assume for illustration purposes that only two multiplications are batched (this will be the case in LightSaber). By splitting M into 32×32 blocks and each of the $\mathbf{c}^{(l)}, \mathbf{a}^{(l)}$ into 32×1 blocks, we get

$$\begin{pmatrix} C_0^{(0)} & C_0^{(1)} \\ C_1^{(0)} & C_1^{(1)} \\ \vdots & \vdots \\ C_7^{(0)} & C_7^{(1)} \end{pmatrix} = \begin{pmatrix} B_0 & -B_7 & \cdots & -B_2 & -B_1 \\ B_1 & B_0 & \cdots & -B_2 & -B_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ B_7 & B_6 & \cdots & B_1 & B_0 \end{pmatrix} \begin{pmatrix} A_0^{(0)} & A_0^{(1)} \\ A_1^{(0)} & A_1^{(1)} \\ \vdots & \vdots \\ A_7^{(0)} & A_7^{(1)} \end{pmatrix},$$

and by exploiting the Toeplitz shape retained by the B_i, this can be rearranged to

$$\begin{aligned}
&\begin{pmatrix} C_0^{(0)} & C_0^{(1)} & C_1^{(0)} & C_1^{(1)} & \cdots & C_7^{(0)} & C_7^{(1)} \end{pmatrix} = \\
&B_0 \begin{pmatrix} A_0^{(0)} & A_0^{(1)} & A_1^{(0)} & A_1^{(1)} & \ldots & A_7^{(0)} & A_7^{(1)} \end{pmatrix} \\
&+B_1 \begin{pmatrix} -A_7^{(0)} & -A_7^{(1)} & A_0^{(0)} & A_0^{(1)} & \ldots & A_6^{(0)} & A_6^{(1)} \end{pmatrix} \\
&\vdots \\
&+B_7 \begin{pmatrix} -A_1^{(0)} & -A_1^{(1)} & -A_2^{(0)} & -A_2^{(1)} & \ldots & A_0^{(0)} & A_0^{(1)} \end{pmatrix}
\end{aligned} \tag{1}$$

which we denote $\sum_{i=0}^{7} B_i \mathcal{A}_i$, defining the 32×16 matrices in parenthesis as $\mathcal{A}_i$. Note that each $\mathcal{A}_i$ can be obtained from different 16-element-wide slices of the 32×30 matrix

$$\mathcal{A} := \begin{pmatrix} -A_1^{(0)} & -A_1^{(1)} & \cdots & -A_7^{(0)} & -A_7^{(1)} & A_0^{(0)} & A_0^{(1)} & \cdots & A_7^{(0)} & A_7^{(1)} \end{pmatrix}. \tag{2}$$

Algorithm III.1 stores the transpose of this matrix (since it is more efficient to load coefficient slices into rows) to the 30 largest odd-numbered `Z` registers. Likewise, the B_i matrices for $i = 0$ and $i > 0$ are given respectively by

$$B_0 = \begin{pmatrix} b_0 & -b_{255} & \cdots & -b_{225} \\ b_1 & b_0 & \cdots & -b_{226} \\ \vdots & \vdots & \ddots & \vdots \\ b_{31} & b_{30} & \cdots & b_0 \end{pmatrix}, \quad B_i = \begin{pmatrix} b_{32i} & b_{32i-1} & \cdots & b_{32i-31} \\ b_{32i+1} & b_{32i} & \cdots & b_{32i-30} \\ \vdots & \vdots & \ddots & \vdots \\ b_{32i+31} & b_{32i+30} & \cdots & b_{32i} \end{pmatrix},$$

so every column of every matrix can be obtained from a 32-element-tall slice of the 287-element column vector

$$\begin{pmatrix} -b_{225} & -b_{226} & \cdots & -b_{255} & b_0 & b_1 & \cdots & b_{255} \end{pmatrix}^{\mathsf{T}}.$$

Note that all but the negative terms (used only for B_0) fit in the `X` registers, so we load only $(b_0 \cdots b_{255})$ to those registers and replace $b_{224}, \ldots, b_{255}$ by their negated version as needed.

Our algorithm works with the transpose of (1), so output coefficients can be stored to memory in the natural row-major layout; thus, we compute $\sum_{i=0}^{7} \mathcal{A}_i^{\mathsf{T}} B_i^{\mathsf{T}} = \sum_{i=0}^{7} \sum_{j=0}^{31} \mathcal{A}_i^{\mathsf{T}}[:, j] B_i^{\mathsf{T}}[j, :]$. Here, $\mathcal{A}_i^{\mathsf{T}}[:, j] B_i^{\mathsf{T}}[j, :]$ corresponds to the outer product of $X[32i-j : 32i-j+31]$ and $\mathtt{Z}[33-4i : 2 : 63-4i][j]$ (with the latter obtainable via an `extrv` instruction). The resulting algorithm is presented as Algorithm III.2.

Remark 1: It is straightforward to generalize the method in this section to the case of batching l polynomial multiplications of $\mathbf{b}(x)$ by $\mathbf{a}^{(0)}(x)$, $\mathbf{a}^{(1)}(x)$, $\ldots$, $\mathbf{a}^{(l-1)}(x)$; the B_i remain the same whereas the $\mathcal{A}_i$ become matrices of dimension $32 \times 8l$. The outer products grow to size $32 \times 8l$ and can still be computed with a single `mac16` instruction as long as $l \leq 4$ (which covers all Saber parameter sets). Meanwhile, the matrix $\mathcal{A}^{\mathsf{T}}$ becomes of size $15l \times 32$, so it is no longer possible to store it in one half of the `Z` registers for $l > 2$. Instead, one can modify Algorithm III.1 and Algorithm III.2 to spill and reload rows of $\mathcal{A}^{\mathsf{T}}$ on demand using an external array, introducing some overhead due to AMX loads and stores.

C. Application to Saber

Encryption and decryption in Saber need to compute inner products of polynomial vectors, for which there is no clear way to benefit from batching, so it is computed using the baseline polynomial multiplication method from Section III-A.

Algorithm III.1 PREPAREMATRIXA($\mathbf{a^{(0)}}, \mathbf{a^{(1)}}$): Loads $\mathcal{A}^{\mathsf{T}}$ from equation (2) to odd Z registers.

Input: $\mathbf{a^{(0)}}$ and $\mathbf{a^{(1)}}$ (arrays of 256 coefficients each)
Output: Loads $-a^{(l)}_{32i:32i+31}$ to `Z[2(2i+l)+1]` for $0 < i \leq 7$, and $a^{(l)}_{32i:32i}$ to `Z[2(2i+l+16)+1]` for $0 \leq i \leq 7$

```
Y₀ ← ldy([−1, ..., −1])
for l = 0 to 1 do
    for i = 0 to 7 do
        Z[2(2i + l + 16) + 1] ← ldz(a^(l)[32i : 32i+31])
        X₀ ← ldx(a^(l)[32i : 32i + 31])
        Z[2(2i + l) + 1] ← mac16(X₀ ∘ Y₀)
```

Algorithm III.2 POLYMODMULTMVP($\mathbf{a^{(0)}}, \mathbf{a^{(1)}}, \mathbf{b}$): Multiplication in $\mathbb{Z}_{2^{16}}[X]/(X^{256}+1)$ of a polynomial $\mathbf{b}$ by two polynomials $\mathbf{a}^{(l)}$ using AMX.

Input: $\mathbf{b}$, $\mathbf{a^{(0)}}$, $\mathbf{a^{(1)}}$ (arrays of 256 coefficients)
Output: Accumulates to $\mathtt{Z}[0:2:30]$ the coefficients for $c^{(l)}(x) = a^{(l)}(x)b(x)$, mapping $c^{(l)}_{32j:32j+31}$ to $\mathtt{Z}[4j + 2l]$.

```
X₀, ..., X₇ ← ldx(b[0 : 31]), ..., ldx(b[224 : 255])
tmp ← stz(mac16(X₇ ∘ [−1, ..., −1]))
PrepareMatrixA(a^(0), a^(1))          ▷ load A^T to odd Z
for j = 0 to 31 do
    Y₀ ← extrv(Z[1 : 2 : 63][j])      ▷ extract A^T[:][j]
    for i = 0 to 7 do
        if i == 0: X₇ ← tmp            ▷ negate [b₂₂₄, ..., b₂₅₅]
        Z[0 : 2 : 30] ← mac16(Z[0 : 2 : 30] + Y[16 − 2i :
            31 − 2i]^T X[32i − j : 32i − j + 31])
        if i == 0: X₇ ← ldx(b[224 : 255])   ▷ restore
```

On the other hand, encryption and key generation multiply an $l \times l$ polynomial matrix by an $l \times 1$ polynomial vector, which is well suited for the batched multiplication from Section III-B. For instance, for LightSaber ($l = 2$), this can be expressed as

$$\mathbf{As} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \begin{pmatrix} s_0 \\ s_1 \end{pmatrix} = s_0 \begin{pmatrix} A_{00} \\ A_{10} \end{pmatrix} + s_1 \begin{pmatrix} A_{01} \\ A_{11} \end{pmatrix},$$

so the multiplications are performed in two batches of two. Additions are free by accumulation in `Z` registers by Alg. III.2.

Two other tasks (in encryption and key generation) can be accelerated by AMX. The first is adding a constant polynomial vector $\mathbf{h}$ to the result, which can be done by loading $\mathbf{h}$ to the `Z` registers before accumulating the matrix-vector product. As all coefficients of $\mathbf{h}$ are identical, they can be distributed to all `Z` registers with one `mac16` instruction. The second task is right-shifting the output by a fixed constant $\epsilon_q - \epsilon_p$ element-wise, using a specific ALU mode of the `vecint` instruction. Algorithm III.3 covers the entire computation of $(\mathbf{As} + \mathbf{h}) \gg (\epsilon_q - \epsilon_p)$ for LightSaber, with straightforward generalizations for Saber ($l = 3$) and FireSaber ($l = 4$) per Remark 1.

IV. FRODOKEM ON NEON AND AMX

In this section, we discuss implementation techniques to speed up FrodoKEM using NEON, and then our AMX implementation which achieves further significant speedups.

Algorithm III.3 SABERMATVECMUL($\mathbf{c}, \mathbf{A}, \mathbf{b}, \mathbf{h}, \epsilon$): Computes $(\mathbf{A}\mathbf{b} + \mathbf{h}) \gg \epsilon$ in $\mathbb{Z}_{2^{16}}[X]/(X^{256}+1)$ with coefficient-wise shifting, for $\mathbf{A}$ a 2×2 polynomial matrix and $\mathbf{b}, \mathbf{h}$ 2×1 polynomial vectors with all coefficients in $\mathbf{h}$ equal.

Input: h (repeating coefficient of $\mathbf{h}$), $\mathbf{b}$ (256-coefficient array), $\mathbf{A}$ ($2 \times 2 \times 256$ array of coefficients) and ϵ (integer).
Output: $\mathbf{c}$ (array of 256 coefficients for the result)

$\mathtt{Z}[0:2:62] \leftarrow \mathtt{mac16}(\mathtt{ldx}([h, h, \ldots, h]))$ ▷ copies of h
for $l = 0$ to 1: $\mathtt{PolyModMulTMVP}(\mathbf{b}, \mathbf{A}[0][l], \mathbf{A}[1][l])$
for $j = 0$ to 31: $\mathbf{c}[j] \leftarrow \mathtt{stz}(\mathtt{vecint}(\mathtt{Z}[2j] \gg \epsilon))$

A. NEON optimizations

The main improvements in our NEON implementation come from (i) refining the generation of the matrix $\mathbf{A}$ in the AES variant and (ii) a careful loop order for matrix multiplication.

Recall that $\mathbf{A}$ is obtained via the *Gen* function. When using AES, the reference NEON implementation initializes the "plaintext" matrix in a first pass, then encrypts it with AES in a second pass. Our implementation generates $\mathbf{A}$ in a single-pass. We generate the "plaintext" in NEON registers and immediately encrypt it with ARMv8 AES instructions.

Let $\mathbf{U}$ and $\mathbf{V}$ be matrices of size $m \times n$ and $n \times p$, respectively. Then, $t_{i,j} = \sum_{k=1}^{n} u_{i,k} v_{k,j}$ is the entry in row i and column j of $\mathbf{T} = \mathbf{U}\mathbf{V}$. Thus, computing $\mathbf{T}$ requires three nested `for` loops, iterating through the values of i, j and k. Although the order of the loops is arbitrary, the data access patterns are different. For each of FrodoKEM's matrix multiplication routines, we evaluated all loop orders to find those with a high ratio of arithmetic to load/store operations, so as to ensure NEON ALUs rather than LSUs are the bottleneck. We checked via performance counters that our choices of loop order, shown in the next table, yield at least 90% ALU usage.

Implementation	Operation			
	$\mathbf{AS}+\mathbf{E}$	$\mathbf{S'A}+\mathbf{E}$	$\mathbf{B'S}$	$\mathbf{S'B}+\mathbf{E}$
Reference	ijk	jik	ijk	ijk
Optimized	ijk	kij	ijk	ijk
NEON	ijk	kji	ikj	kij

Also, for a potential doubling in throughput on the M1 and M3, we employ multiply-accumulate instructions instead of separate multiplication and addition instructions as in [13].

B. Matrix multiplication on AMX

We focus on the computations $\mathbf{AS}+\mathbf{E}$ and $\mathbf{S'A}+\mathbf{E'}$, where $\mathbf{A}$ has size $n \times n$, $\mathbf{S}, \mathbf{E}$ have size $n \times \bar{n}$ and $\mathbf{S'}, \mathbf{E'}$ have size $\bar{n} \times n$. For both multiplications, $\mathbf{AS}$ and $\mathbf{S'A}$, the main idea is to load $\mathbf{A}$ and $\mathbf{S}$ (or $\mathbf{S'}$) into the `X` and `Y` input registers, and compute multiply-accumulates to the `Z` output registers. By initializing `Z` with $\mathbf{E}$ or $\mathbf{E'}$, we obtain addition "for free".

In AMX, the natural size of matrices for multiplication is up to 32×32. So, $\mathbf{AS}$ and $\mathbf{S'A}$ are computed via block matrix multiplication. We decompose $\mathbf{A}$ into 32×32 blocks, $\mathbf{S}$ into 32×8 blocks and $\mathbf{S'}$ into 8×32 blocks. However, one of the dimensions being < 32 is sub-optimal since the remaining 24 rows/columns are masked out, but `mac16` does not appear to execute any faster. This wasted computational power is reclaimed through batching in Section IV-C. For Frodo-976, $32 \nmid n$, so part of the computation for blocks at the edges is masked out, thus behaving as if they were padded with zeros.

Recall that AMX multiplies matrices via outer products of vectors. To compute $\mathbf{AS}$, we read blocks of $\mathbf{A}$ by columns and $\mathbf{S}$ by rows. $\mathbf{A}$ and $\mathbf{S}^T$ are generated in row-major order by Algorithm II.4 (lines 2 and 5); thus, $\mathbf{S}$ is in column-major order, and we must transpose both $\mathbf{A}$ and $\mathbf{S}^T$. For $\mathbf{S'A}$, both matrices are generated in row-major order by Algorithm II.6 (lines 1 and 4). Thus, we transpose $\mathbf{S'}$ only.

We report on transposition strategies that performed best among all that we tried. For $\mathbf{AS}$, we first transpose the full $\mathbf{S}^T$ directly in C (which the compiler autovectorizes to NEON), while $\mathbf{A}$ is transposed online with AMX during multiplication (as in Algorithm II.7), after generating 32 rows with our one-pass strategy of Section IV-A; this is shown in Algorithm IV.1. For $\mathbf{S'A}$, AMX transposition of $\mathbf{A}$ also performs best.

Algorithm IV.1 FRODO-AS-PLUS-E-32ROWS($\mathbf{C}$,$\bar{\mathbf{A}}$,$\mathbf{S}^{\mathsf{T}}$,$\mathbf{E}$,r): Computes rows $r, \ldots, r+31$ of $\mathbf{C} \leftarrow \mathbf{AS}+\mathbf{E}$; $\bar{\mathbf{A}}$ is the submatrix of $\mathbf{A}$ containing rows $r, \ldots, r+31$.

Input: $\bar{\mathbf{A}} \in \mathbb{Z}_{2^{16}}^{32 \times n}$; $\mathbf{S}^{\mathsf{T}}$,$\mathbf{E} \in \mathbb{Z}_{2^{16}}^{n \times \bar{n}}$; $r \in \{0, 32, \ldots, n-32\}$
Output: $\mathbf{C} \in \mathbb{Z}_{2^{16}}^{n \times \bar{n}}$ with rows $r, \ldots, r+31$ updated.

Load $\mathbf{E}[r : r+31]$ to $\mathtt{Z}[0:2:62]$
for $j_0 = 0, 32, 64, \ldots, n-32$ **do**
  Load $\mathbf{S}^{\mathsf{T}}[j_0][0:7] \mid\mid \ldots \mid\mid \mathbf{S}^{\mathsf{T}}[j_0+31][0:7]$ to `X`
  Load $\bar{\mathbf{A}}[0:31][j_0 : j_0+31]$ to $\mathtt{Z}[1:2:63]$
  for $j = 0, \ldots, 31$ **do**
    $\mathtt{Y}_0 \leftarrow \mathtt{extrv}(\mathtt{Z}[1:2:63][i])$
    $\mathtt{Z}[0:2:62][0:7] \leftarrow \mathtt{mac16}(\mathtt{Z}[0:2:62][0:7] + \mathtt{Y}_0^T \mathtt{X}[8j : 8j+7])$
Store $\mathtt{Z}[0:2:62]$ to $\mathbf{C}[r : r+31]$

C. Use of batching

AMX's throughput is underutilized with single KEM operations as above. We overcome this by introducing an alternate API that batches KEM operations with the same (sk, pk) pair, i.e., batching multiplications with the same $\mathbf{A}$. Thus, it applies to encapsulation and decapsulation (which compute $\mathbf{S'A}$) but not to key generation (which computes $\mathbf{AS}$).

Batch $\mathbf{S'A}+\mathbf{E}$ computation reuses the strategy of Section IV-B, except we do 4 computations at once. Recall that $\mathbf{A}$ has size $n \times n$ while $\mathbf{S'}$ and $\mathbf{E}$ have size $8 \times n$. We vertically stack four $\mathbf{S'}$ or $\mathbf{E}$ matrices to get $32 \times n$ matrices, thus fully using AMX's processing power. The ALUs are nearly saturated in our NEON implementation (see Section IV-A), so batching is implemented straightforwardly (a loop over the 4 copies).

The operations $\mathbf{S'B}+\mathbf{E''}$ and $\mathbf{B'S}$, without batching, yield small 8×8 matrices, which would severely underutilize AMX's processing power. With batching, however, one dimension grows to 32, as in $\mathbf{AS}+\mathbf{E}$ and $\mathbf{S'A}$, and may become worthwhile. $\mathbf{S'B}+\mathbf{E''}$ requires a single transposition, which we perform online in AMX during multiplication as in $\mathbf{S'A}+\mathbf{E}$;

for $\mathbf{B}'\mathbf{S}$, which requires two transpositions (as $\mathbf{S}$ is stored transposed in the secret key), we failed to achieve a speedup.

D. Gaussian sampling using the AMX `genlut` instruction

We now present a novel technique for table-based inversion sampling, applied to FrodoKEM Gaussian sampling. At its core is the search operation done by AMX's `genlut` instruction in generate mode (see Section II-C) to perform parallel search on 32 16-bit source values. For distributions with non-negative support and tables of up to 31 elements, its use is straightforward, as well as for full support, if the table fits an `X` or `Y` register and inputs use two's complement representation. The former condition is met by all parameter sets, but for the latter, an incompatible representation (sign-magnitude with the sign given by the least-significant bit) is prescribed. Thus, we must condition the inputs, at some performance cost.

In lieu of actual two's complement representation, we place the sign at the most significant bit by right-rotating each input (lines 4 and 5 of Algorithm IV.2), and adapt the $\mathbf{T}_\chi$ tables to work with this format. The algorithm specified in [2] uses a table for the non-negative support only, and applies the sign bit to the output. We avoid separate application of the sign bit in AMX by using two shifted copies of the table. These fit in the table register since the largest $\mathbf{T}_\chi$ (for Frodo-640) has $j+1=13$ elements. Concretely, if $\mathbf{T}_\chi=[t_0,t_1,\ldots,t_j]$ is the original table, we map it to the `genlut`-specific table

$$\mathbf{T}'_\chi=[0,t_0+1,\ldots,t_j+1,t_0+2^{15}+1,\ldots,t_j+2^{15}+1].$$

Finally, `genlut` in generate mode outputs a densely packed representation (20 bytes representing the results of 32 parallel searches). The remainder of the FrodoKEM code expects the usual 16 bits per element representation. We use `genlut` in lookup mode to map results to the range $[-j,j]$, in accordance with our choice of $\mathbf{T}'_\chi$. The 32-element mapping is given by

$$\iota=[0,1,\ldots,j,0,-1,\ldots,-j,-j,\ldots,-j].$$

We display this procedure as Algorithm IV.2.

Algorithm IV.2 SAMPLEMATRIX$(\mathbf{s},\mathbf{T}'_\chi,\iota)$: $s_i\leftarrow\mathbf{T}'_\chi[s_i]$ for $0\le i<\overline{n}\cdot n$.

Input: $\mathbf{s}\in\mathbb{Z}_{2^{16}}^{\overline{n}\times n}$ (uniform samples); $\mathbf{T}'_\chi,\iota\in\mathbb{Z}_{2^{16}}^{32}$ (as above).
Output: $\mathbf{s}\in\mathbb{Z}_{2^{16}}^{\overline{n}\times n}$ (Gaussian samples)

1: $\mathtt{Y}_0,\mathtt{Y}_1,\mathtt{Y}_2\leftarrow\mathtt{ldy}(\mathbf{T}'_\chi),\mathtt{ldy}(\iota),\mathtt{ldy}([2^{15},\ldots,2^{15}])$
2: **for** $i=0,32,64,\ldots,\overline{n}\cdot n-32$ **do** ▷ Process 32 elements
3: $\quad\mathtt{X}_0\leftarrow\mathtt{ldx}(\mathbf{s}[i:i+31])$
4: $\quad\mathtt{Z}[0]\leftarrow\mathtt{vecint}(\mathtt{X}_0\circ\mathtt{Y}_2)$
5: $\quad\mathtt{Z}[0]\leftarrow\mathtt{vecint}(\mathtt{Z}[0]+\mathtt{X}_0>>1)$
6: $\quad\mathtt{X}[0]\leftarrow\mathtt{extrh}(\mathtt{Z}[0])$
7: $\quad\mathtt{X}_0\leftarrow\mathtt{genlut}(\text{mode}=\text{gen},\text{src}=\mathtt{X}_0,\text{tbl}=\mathtt{Y}_0)$
8: $\quad\mathtt{X}_0\leftarrow\mathtt{genlut}(\text{mode}=\text{lookup},\text{src}=\mathtt{X}_0,\text{tbl}=\mathtt{Y}_1)$
9: $\quad\mathbf{s}[i:i+31]\leftarrow\mathtt{stx}(\mathtt{X}_0)$

V. Experimental results

In this section, we describe our experimental setup, report and analyze performance results, and report on experiments on the constant-time execution of AMX's `genlut` instruction.

A. Experimental setup

We benchmark on M1- and M3-series Apple computers, running macOS 14 and version 15 of Apple's clang compiler.

As in [9], we explore distinct array allocation strategies. In the usual stack allocation, neighboring variables of a function are very likely allocated in the same memory page, risking concurrent CPU and AMX accesses, which cause performance degradation. The POSIX `mmap()` function returns new memory pages for each allocation, sidestepping this issue.

As symmetric operations make up the bulk of execution time in Saber and FrodoKEM, we use fast implementations of SHAKE, AES and NIST's `randombytes()` function, all using instructions of ARMv8's Cryptographic Extensions. We use the 2×-batched SHAKE implementation of Becker et al. [10], and modify their unbatched SHAKE code to use ARMv8's SHA-3 instructions. We implement AES-ECB (for FrodoKEM) and AES-CTR-DRBG (for `randombytes()`), ensuring outputs match with existing implementations.

We use the macOS cycle counting code of [18], and report a median of 1024 executions for most measurements; however, for small runtimes (up to a few thousands of cycles), our experience is that medians, while less noisy, underreport cycle counts by a few hundred cycles, especially for AMX codes. Thus, we opted to report averages instead for SABER matrix-vector multiplication, FrodoKEM Gaussian sampling and the constant-time experiments of Section V-C.

B. Performance results

We report Saber and FrodoKEM performance data for KEM operations and specific subroutines accelerated by AMX. For memory allocation strategies (`mmap()` or stack), we pick the fastest strategy for each individual measurement, and indicate this in the tables by font style: regular font for `mmap()`, and italics for stack. We compute speedups as ratios between the previous state-of-the-art implementation and our AMX one; for FrodoKEM, we compare our NEON implementation to the previous state-of-the-art ones, and our AMX implementation to the fastest CPU implementation (usually, our NEON one).

Saber results are shown in Table I. "MVMR" refers to matrix-vector multiplication of polynomials with rounding. NIST levels 1, 3 and 5 map to LightSaber, Saber and FireSaber parameter sets, respectively. FrodoKEM results without batching are shown in Table II, and with $4\times$ batching in Table III. We benchmark KEM operations as well as matrix operations $\mathbf{AS}+\mathbf{E}$ and $\mathbf{S}'\mathbf{A}+\mathbf{E}$. The "full" subheading includes the cost of $\mathbf{A}$ matrix generation, while "mat. mul." does not.

In lieu of a full discussion, we highlight the main takeaways:

- Most of the cost in both schemes is for symmetric operations, which use the same implementation everywhere. Amdahl's law bounds gains due to polynomial/matrix multiplication; results should be viewed in that context.
- Saber's KEM operations are sped up by 8 to 13%, and matrix-vector multiplication of polynomials by up to 70%, 108% and 150% for LightSaber, Saber and FireSaber, respectively, despite replacing $\mathcal{O}(n\log n)$ NTT methods in NEON by a schoolbook $\mathcal{O}(n^2)$ algorithm

Sec lvl	Work	Operation							
		Key gen.		Encaps.		Decaps.		MVMR	
		M1	M3	M1	M3	M1	M3	M1	M3
1	[10]	19.2	*19.1*	*26.3*	*26.3*	*25.8*	25.7	4.02	*3.96*
	III-A	19.7	18.5	26.5	25.6	25.8	24.5	3.97	3.28
	III-B	17.4	17.5	24.4	24.3	23.7	23.2	2.37	2.32
Speedup($\times$)		**1.11**	**1.09**	**1.08**	**1.08**	**1.09**	**1.11**	**1.69**	**1.71**
3	[10]	*31.5*	*31.4*	*40.3*	*40.2*	*40.5*	40.3	7.60	7.52
	III-A	33.9	32.3	42.7	40.9	43.3	40.9	8.98	7.43
	III-B	28.4	28.4	36.9	36.5	37.4	36.4	3.66	3.62
Speedup($\times$)		**1.11**	**1.11**	**1.09**	**1.10**	**1.08**	**1.11**	**2.08**	**2.08**
5	[10]	48.5	*48.3*	*59.6*	*59.7*	*60.0*	*59.8*	12.2	12.1
	III-A	54.1	51.3	65.8	62.2	66.4	62.4	16.0	13.3
	III-B	42.9	42.9	54.1	53.2	54.6	53.5	4.92	4.83
Speedup($\times$)		**1.13**	**1.12**	**1.10**	**1.12**	**1.10**	**1.12**	**2.49**	**2.51**

TABLE I
CYCLE COUNTS FOR SABER OPERATIONS, IN THOUSANDS OF CYCLES, COMPARING THE IMPLEMENTATION OF [10] TO THE ALGORITHMS OF SECTIONS III-A AND III-B.

in AMX. Gains rise with the parameter l due to better utilization of AMX, per Remark 1. As decapsulation performs reencryption, it also benefits from these gains.

- FrodoKEM's optimized implementation [12] is up to $\approx 15\times$ faster than the reference one, reinforcing our belief that it is on par or faster than that of Kwon et al. [13]. Our NEON implementation achieved further speedups of up to 30%, 22% and 25% for key generation, encapsulation and decapsulation, respectively. AMX improves upon NEON by up to 13%, 19% and 21% for these operations.
- Matrix operations are further sped up due to the reduced cost of symmetric operations (even more so when $\mathbf{A}$ matrix generation is removed). AMX achieves up to 124% gains over our already improved NEON implementation.
- Our AMX-specific Gaussian sampling technique of Section IV-D achieves gains of up to 418% over NEON.
- Batching amortizes the cost of $\mathbf{A}$ matrix generation across 4 operations and ensures full AMX utilization. Our NEON implementation improves upon the optimized one by up to 50%, and AMX improves that further by 91%. For matrix multiplication only, AMX gains up to 708%.

C. Constant-time behavior of AMX's `genlut` *instruction*

Cryptographic implementations must execute in time independent of secret data (or *constant time*) to avoid timing side-channel attacks. Gazzoni Filho et al. [9] verify this for many AMX instructions, but critically not for `genlut`, on which our sampling technique of Section IV-D is based. We sought to do so by benchmarking AMX and NEON sampling routines for different inputs: 0, $2^{16}-2$, $2^{16}-1$ or fully random inputs. The first three map to specific fixed points in the sampling table: respectively, the first, midpoint and last elements.

We report results for Frodo-640 on the M3; results for other parameter sets and the M1 are similar, and are included in the full results dataset in our GitHub repository. Cycle counts for sampling the full 8×640 matrix, across all four inputs, vary from 4585 to 4593, 4346 to 4350 and 839 to 841 cycles for optimized, NEON and AMX implementations, respectively. Thus, modulo small variations across benchmark runs, all implementations appear to run in constant time.

VI. CONCLUSION AND FUTURE WORK

We have implemented the post-quantum cryptosystems Saber and FrodoKEM using the undocumented AMX tightly-coupled matrix multiplication coprocessor, obtaining considerable speedups over CPU-only implementations.

We highlight the difficulties of fully exploiting AMX's available processing power. Some strides were made over the work of Gazzoni Filho et al. [9], by recasting Saber polynomial multiplication in matrix-multiplication language; still, only the FireSaber parameter set makes full use of AMX. For FrodoKEM, a batched implementation is needed to achieve this goal. Future cryptosystem designs may wish to revisit parameter choices to favor matrix multiplication accelerators.

We note that the performance of many PQC schemes is dictated by the cost of symmetric operations, rather than arithmetic ones such as polynomial/matrix multiplication. To ensure improvements to the latter are duly reflected in protocol performance, more research is needed (from design, implementation and hardware standpoints) into reducing the share of symmetric operations in the execution time of PQC schemes.

An important class of lattice-based cryptosystems are based on NTTs, such as Kyber and Dilithium; we echo the suggestion of [9] to investigate AMX implementations of such schemes.

Table-based sampling is perceived as difficult to implement efficiently in constant time. Our novel technique of Section IV-D, using AMX's `genlut` instruction, brings renewed hope for such methods. CPU architects would do well to extend instruction set architectures with a similar instruction.

REFERENCES

[1] J.-P. D'Anvers, A. Karmakar, S. Sinha Roy, and F. Vercauteren, "Saber: Module-LWR based key exchange, CPA-secure encryption and CCA-secure KEM," in *AFRICACRYPT 2018*, A. Joux, A. Nitaj, and T. Rachidi, Eds. Cham: Springer, 2018, pp. 282–305.

[2] E. Alkim, J. W. Bos, L. Ducas, P. Longa, I. Mironov, M. Naehrig, V. Nikolaenko, C. Peikert, A. Raghunathan, and D. Stebila, "FrodoKEM learning with errors key encapsulation," Submission to the NIST PQC Standardization Project, 2021, https://frodokem.org/files/FrodoKEM-specification-20210604.pdf.

[3] BSI, "Migration to post quantum cryptography: Recommendations for action by the BSI," 2021. [Online]. Available: https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Crypto/Migration_to_Post_Quantum_Cryptography.pdf?__blob=publicationFile&v=2

[4] Intl. Organization for Standardization, "FrodoKEM: Learning with errors key encapsulation preliminary draft standard," 2023. [Online]. Available: https://frodokem.org/files/FrodoKEM-ISO-20230314.pdf

[5] S. Markidis, S. Chien, E. Laure, I. Peng, and J. S. Vetter, "NVIDIA tensor core programmability, performance & precision," in *2018 IEEE Intl. Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. Los Alamitos, CA, USA: IEEE CompSoc, May 2018, pp. 522–531.

[6] Intel Corporation, "Intel® architecture instruction set extensions and future features: Programming reference (revision 047)," December 2022, https://cdrdv2-public.intel.com/671368/architecture-instruction-set-extensions-programming-reference.pdf.

[7] F. Wilkinson and S. McIntosh-Smith, "An initial evaluation of Arm's Scalable Matrix Extension," in *2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2022, pp. 135–140.

[8] A. Rodriguez, *Deep Learning Systems: Algorithms, Compilers, and Processors for Large-Scale Production*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, Oct. 2020.

Sec lvl	Work	Operation																	
		Key gen.		Encaps.		Decaps.		$\mathbf{AS+E}$				$\mathbf{S'A+E}$				Gaussian sampling	
								full		mat. mul.		full		mat. mul.			
		M1	M3	M1	M3	M1	M3	M1	M3	M1	M3	M1	M3	M1	M3	M1	M3
1	[2]	816	779	7266	6639	7272	6472	582	557	305	296	6919	6232	6631	5992	4.62	4.59
	[12]	624	558	730	669	717	641	398	354	202	192	362	345	186	189	4.62	4.59
	Ours (NEON)	504	468	639	561	623	532	274	263	109	118	280	254	111	108	4.34	4.35
	Ours (AMX)	468	414	580	494	541	447	267	226	79.2	77.7	261	197	52.6	52.4	1.32	0.84
Speedup NEON(×)		**1.24**	**1.19**	**1.14**	**1.19**	**1.15**	**1.21**	**1.45**	**1.34**	**1.85**	**1.63**	**1.30**	**1.36**	**1.68**	**1.75**	**1.07**	**1.05**
Speedup AMX(×)		**1.08**	**1.13**	**1.10**	**1.14**	**1.15**	**1.19**	**1.03**	**1.16**	**1.38**	**1.52**	**1.07**	**1.29**	**2.11**	**2.05**	**3.29**	**5.18**
3	[2]	1776	1686	8886	8789	8833	8731	1418	1349	764	748	8360	8312	7690	7700	6.03	5.99
	[12]	1242	1220	1392	1310	1306	1255	902	880	412	418	854	824	384	384	6.03	5.99
	Ours (NEON)	982	940	1144	1070	1087	1005	642	600	247	247	652	586	256	247	5.64	5.65
	Ours (AMX)	887	839	992	930	928	845	578	530	184	181	540	461	125	125	2.01	1.28
Speedup NEON(×)		**1.27**	**1.30**	**1.22**	**1.22**	**1.20**	**1.25**	**1.41**	**1.47**	**1.67**	**1.69**	**1.31**	**1.41**	**1.50**	**1.56**	**1.07**	**1.06**
Speedup AMX(×)		**1.11**	**1.12**	**1.15**	**1.15**	**1.17**	**1.19**	**1.11**	**1.13**	**1.34**	**1.37**	**1.21**	**1.27**	**2.05**	**1.97**	**2.80**	**4.41**
5	[2]	2983	2866	32389	30154	32323	29760	2534	2424	1310	1284	31703	29752	30464	29186	5.50	5.48
	[12]	2120	1931	2265	2156	2145	2061	1665	1479	824	808	1537	1474	775	766	5.50	5.48
	Ours (NEON)	1662	1573	1915	1766	1839	1681	1204	1116	465	465	1256	1113	512	471	5.07	5.09
	Ours (AMX)	1502	1396	1612	1500	1527	1388	1108	970	334	328	1033	855	229	229	2.77	1.76
Speedup NEON(×)		**1.28**	**1.23**	**1.18**	**1.22**	**1.17**	**1.23**	**1.38**	**1.33**	**1.77**	**1.74**	**1.22**	**1.33**	**1.51**	**1.63**	**1.08**	**1.08**
Speedup AMX(×)		**1.11**	**1.13**	**1.19**	**1.18**	**1.20**	**1.21**	**1.09**	**1.15**	**1.39**	**1.42**	**1.22**	**1.30**	**2.24**	**2.06**	**1.83**	**2.88**

TABLE II

CYCLE COUNTS FOR FRODOKEM-AES OPERATIONS, IN THOUSANDS OF CYCLES, WITHOUT BATCHING.

Sec lvl	Work	Operation							
		Encaps. 4×		Decaps. 4×		$\mathbf{S'A+E}$ 4×			
						full		mat. mul.	
		M1	M3	M1	M3	M1	M3	M1	M3
1	[12]	1896	1755	1869	1755	934	924	758	7545
	Ours (NEON)	1528	1395	1496	1387	639	614	470	452
	Ours (AMX)	1028	907	993	905	234	211	65.5	64.1
Speedup NEON(×)		**1.24**	**1.26**	**1.25**	**1.27**	**1.46**	**1.50**	**1.61**	**1.65**
Speedup AMX(×)		**1.49**	**1.54**	**1.51**	**1.53**	**2.73**	**2.91**	**7.18**	**7.05**
3	[12]	3354	3264	3218	3098	2093	2001	1542	1533
	Ours (NEON)	2685	2594	2534	2419	1482	1423	1086	1001
	Ours (AMX)	1624	1555	1467	1381	543	485	148	148
Speedup NEON(×)		**1.25**	**1.26**	**1.27**	**1.28**	**1.41**	**1.41**	**1.42**	**1.53**
Speedup AMX(×)		**1.65**	**1.67**	**1.73**	**1.75**	**2.73**	**2.93**	**7.34**	**6.78**
5	[12]	6422	5807	6262	5569	4488	4000	3214	3111
	Ours (NEON)	4395	4249	4185	4004	2796	2690	2054	1942
	Ours (AMX)	2479	2352	2261	2101	997	887	254	252
Speedup NEON(×)		**1.46**	**1.37**	**1.50**	**1.39**	**1.61**	**1.49**	**1.57**	**1.60**
Speedup AMX(×)		**1.77**	**1.81**	**1.85**	**1.91**	**2.80**	**3.03**	**8.08**	**7.70**

TABLE III

CYCLE COUNTS FOR FRODOKEM-AES OPERATIONS, IN THOUSANDS OF CYCLES, WITH 4× BATCHING.

[9] D. L. Gazzoni Filho, G. Brandão, and J. López, “Fast polynomial multiplication using matrix multiplication accelerators with applications to NTRU on Apple M1/M3 SoCs,” 2024. [Online]. Available: https://ia.cr/2024/002

[10] H. Becker, V. Hwang, M. J. Kannwischer, B.-Y. Yang, and S.-Y. Yang, “Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1,” 2021. [Online]. Available: https://ia.cr/2021/986

[11] İ. K. Paksoy and M. Cenk, “TMVP-based multiplication for polynomial quotient rings and application to Saber on ARM Cortex-M4,” 2020. [Online]. Available: https://ia.cr/2020/1302

[12] J. W. Bos, M. Ofner, J. Renes, T. Schneider, and C. van Vredendaal, “The Matrix Reloaded: Multiplication strategies in FrodoKEM,” in *Cryptology and Network Security: 20th International Conference, CANS 2021, Vienna, Austria, December 13-15, 2021, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2021, p. 72–91.

[13] H. Kwon, K. Jang, H. Kim, H. Kim, M. Sim, S. Eum, W.-K. Lee, and H. Seo, “ARMed Frodo,” in *Information Security Applications*, H. Kim, Ed. Cham: Springer, 2021, pp. 206–217.

[14] D. Johnson, “IDA (disassembler) and Hex-Rays (decompiler) plugin for Apple AMX,” 2022, https://gist.github.com/dougallj/7a75a3be1ec69ca550e7c36dc75e0d6f.

[15] M. Handley, “AArch64-Explore: Exploration of Apple CPUs – volume 3: SoC,” 2023, https://github.com/name99-org/AArch64-Explore.

[16] P. Cawley, “Apple AMX instr. set,” 2023, https://github.com/corsix/amx/.

[17] L. Wan, F. Zheng, and J. Lin, “TESLAC: Accelerating lattice-based cryptography with AI accelerator,” in *Security and Privacy in Communication Networks*, J. Garcia-Alfaro, S. Li, R. Poovendran, H. Debar, and M. Yung, Eds. Cham: Springer, 2021, pp. 249–269.

[18] D. T. Nguyen and K. Gaj, “Fast NEON-based multiplication for lattice-based NIST post-quantum cryptography finalists,” in *Post-Quantum Cryptography*, J. H. Cheon and J.-P. Tillich, Eds. Cham: Springer, 2021, pp. 234–254.